UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Giovani Gracioli

# REAL-TIME OPERATING SYSTEM SUPPORT FOR MULTICORE APPLICATIONS

Thesis presented in partial fulfillment of the requirements for the degree of Doctor in Automation and Systems Engineering.
Advisor: Prof. Dr. Antônio Augusto Medeiros Fröhlich

Florianópolis

2014

Giovani Gracioli

# REAL-TIME OPERATING SYSTEM SUPPORT FOR MULTICORE APPLICATIONS

This thesis was accepted in its present form by the Programa de Pós-Graduação em Engenharia de Automação e Sistemas as a partial fulfillment of the requirements for the degree of Doctor in Automation and Systems Engineering.
Florianópolis, 04/07/2014.

Rômulo Silva de Oliveira, Dr.
Graduate Program Coordinator

**Doctoral Committee:**

Antônio Augusto Medeiros Fröhlich, Dr.
Advisor

Rivalino Matias Júnior, Dr., UFU

Rodolfo Pellizzoni, Dr., UW

Mario Antonio Ribeiro Dantas, Dr., UFSC

Rômulo Silva de Oliveira, Dr., UFSC

Carlos Barros Montez, Dr., UFSC

*To my beloved wife and best friend, Juliana, and my mother, Rosilene.*

# ACKNOWLEDGEMENTS

I am grateful to several people that helped me to get in the graduate school and that supported me during these almost five years. First of all, I would to thank my mother, Rosilene Marisa Bertochi, for all the encouragement words and support, and my wife, Juliana Sielski Favretto, for being my best friend and giving all the emotional support, mainly in the last year of this journey.

I would like to thank my advisor, Prof. Antônio Augusto Medeiros Fröhlich, for the valuable scientific discussions and assistance. The academic and administrative support and the scientific environment at LISHA were fundamental to achieve the research results of this thesis. Many thanks also to all colleagues at LISHA.

I would also like to thank Prof. Sebastian Fischmeister and Prof. Rodolfo Pellizzoni for the valuable academic environment and discussions during my stay at the University of Waterloo and for the fruitful research collaborations that we established over the past years. Also, I would like to thank all the co-authors of my published papers.

I am also grateful to Prof. Rivalino Matias Junior, Prof. Rodolfo Pellizzoni, Prof. Mario Antonio Ribeiro Dantas, Prof. Rômulo Silva de Oliveira, and Prof. Carlos Barros Montez for serving on my thesis committee and for their kind comments.

Last but not least, many thanks to CAPES for the financial support during four years in the graduate program, which was essential for living in Florianópolis.

# RESUMO

Plataformas multiprocessadas atuais possuem diversos níveis da memória cache entre o processador e a memória principal para esconder a latência da hierarquia de memória. O principal objetivo da hierarquia de memória é melhorar o tempo médio de execução, ao custo da previsibilidade. O uso não controlado da hierarquia da cache pelas tarefas de tempo real impacta a estimativa dos seus piores tempos de execução, especialmente quando as tarefas de tempo real acessam os níveis da cache compartilhados. Tal acesso causa uma disputa pelas linhas da cache compartilhadas e aumenta o tempo de execução das aplicações. Além disso, essa disputa na cache compartilhada pode causar a perda de prazos, o que é intolerável em sistemas de tempo real críticos.

O particionamento da memória cache compartilhada é uma técnica bastante utilizada em sistemas de tempo real multiprocessados para isolar as tarefas e melhorar a previsibilidade do sistema. Atualmente, os estudos que avaliam o particionamento da memória cache em multiprocessadores carecem de dois pontos fundamentais. Primeiro, o mecanismo de particionamento da cache é tipicamente implementado em um ambiente simulado ou em um sistema operacional de próposito geral. Consequentemente, o impacto das atividades realizados pelo núcleo do sistema operacional, tais como o tratamento de interrupções e troca de contexto, no particionamento das tarefas tende a ser negligenciado. Segundo, a avaliação é restrita a um escalonador global ou particionado, e assim não comparando o desempenho do particionamento da cache em diferentes estratégias de escalonamento.

Ademais, trabalhos recentes confirmaram que aspectos da implementação do SO, tal como a estrutura de dados usada no escalonamento e os mecanismos de tratamento de interrupções, impactam a escalonabilidade das tarefas de tempo real tanto quanto os aspectos teóricos. Entretanto, tais estudos também usaram sistemas operacionais de propósito geral com extensões de tempo real, que afetam os sobrecustos de tempo de execução observados e a escalonabilidade das tarefas de tempo real. Adicionalmente, os algoritmos de escalonamento tempo real para multiprocessadores atuais não consideram cenários onde tarefas de tempo real acessam as mesmas linhas da cache, o que dificulta a estimativa do pior tempo de execução.

Esta pesquisa aborda os problemas supracitados com as estratégias de particionamento da cache e com os algoritmos de escalonamento tempo real multiprocessados da seguinte forma. Primeiro, uma infraestrutura de tempo real para multiprocessadores é projetada e implementada em um sistema operacional embarcado. A infraestrutura consiste em diversos algoritmos de escalonamento tempo real, tais como o EDF global e particionado, e um mecanismo de particionamento da cache usando a técnica de coloração de páginas. Segundo, é apresentada uma comparação em termos da taxa de escalonabilidade considerando o sobrecusto de tempo de execução da infraestrutura criada e de um sistema operacional de propósito geral com extensões de tempo real. Em alguns casos, o EDF global considerando o sobrecusto do sistema operacional embarcado possui uma melhor taxa de escalonabilidade do que o EDF particionado com o sobrecusto do sistema operacional de propósito geral, mostrando claramente como diferentes sistemas operacionais influenciam os escalonadores de tempo real críticos em multiprocessadores. Terceiro, é realizada uma avaliação do impacto do particionamento da memória cache em diversos escalonadores de tempo real multiprocessados. Os resultados desta avaliação indicam que um sistema operacional "leve" não compromete as garantias de tempo real e que o particionamento da cache tem diferentes comportamentos dependendo do escalonador e do tamanho do conjunto de trabalho das tarefas. Quarto, é proposto um algoritmo de particionamento de tarefas que atribui as tarefas que compartilham partições ao mesmo processador. Os resultados mostram que essa técnica de particionamento de tarefas reduz a disputa pelas linhas da cache compartilhadas e provê garantias de tempo real para sistemas críticos. Finalmente, é proposto um escalonador de tempo real de duas fases para multiprocessadores. O escalonador usa informações coletadas durante o tempo de execução das tarefas através dos contadores de desempenho em hardware. Com base nos valores dos contadores, o escalonador detecta quando tarefas de melhor esforço interferem com tarefas de tempo real na cache. Assim é possível impedir que tarefas de melhor esforço acessem as mesmas linhas da cache que tarefas de tempo real. O resultado desta estratégia de escalonamento é o atendimento dos prazos críticos e não críticos das tarefas de tempo real.

**Palavras-chave**: Sistemas operacionais de tempo real, escalonamento tempo real em multiprocessadores, particionamento de tarefas, particionamento da memória cache compartilhada.

# ABSTRACT

Modern multicore platforms feature multiple levels of cache memory placed between the processor and main memory to hide the latency of ordinary memory systems. The primary goal of this cache hierarchy is to improve average execution time (at the cost of predictability). The uncontrolled use of the cache hierarchy by real-time tasks may impact the estimation of their worst-case execution times (WCET), specially when real-time tasks access a shared cache level, causing a contention for shared cache lines and increasing the application execution time. This contention in the shared cache may lead to deadline losses, which is intolerable particularly for hard real-time (HRT) systems.

Shared cache partitioning is a well-known technique used in multicore real-time systems to isolate task workloads and to improve system predictability. Presently, the state-of-the-art studies that evaluate shared cache partitioning on multicore processors lack two key issues. First, the cache partitioning mechanism is typically implemented either in a simulated environment or in a general-purpose OS (GPOS), and so the impact of kernel activities, such as interrupt handlers and context switching, on the task partitions tend to be overlooked. Second, the evaluation is typically restricted to either a global or partitioned scheduler, thereby by falling to compare the performance of cache partitioning when tasks are scheduled by different schedulers.

Furthermore, recent works have confirmed that OS implementation aspects, such as the choice of scheduling data structures and interrupt handling mechanisms, impact real-time schedulability as much as scheduling theoretic aspects. However, these studies also used real-time patches applied into GPOSes, which affects the run-time overhead observed in these works and consequently the schedulability of real-time tasks. Additionally, current multicore scheduling algorithms do not consider scenarios where real-time tasks access the same cache lines due to true or false sharing, which also impacts the WCET.

This thesis addresses these aforementioned problems with cache partitioning techniques and multicore real-time scheduling algorithms as following. First, a real-time multicore support is designed and implemented on top of an embedded operating system designed from scratch. This support consists of several multicore real-time scheduling algorithms, such as global and partitioned EDF, and a cache partitioning mechanism based on page coloring. Second, it is presented a comparison in terms of schedulability ratio considering the run-time overhead of the implemented RTOS and a GPOS patched with real-time extensions. In some cases, Global-EDF considering the overhead of the RTOS is superior to Partitioned-EDF considering the overhead of the patched GPOS, which clearly shows how different OSs impact hard real-time schedulers. Third, an evaluation of the cache partitioning impact on partitioned, clustered, and global real-time schedulers is performed. The results indicate that a lightweight RTOS does not impact real-time tasks, and shared cache partitioning has different behavior depending on the scheduler and the task's working set size. Fourth, a task partitioning algorithm that assigns tasks to cores respecting their usage of cache partitions is proposed. The results show that by simply assigning tasks that shared cache partitions to the same processor, it is possible to reduce the contention for shared cache lines and to provide HRT guarantees. Finally, a two-phase multicore scheduler that provides HRT and soft real-time (SRT) guarantees is proposed. It is shown that by using information from hardware performance counters at run-time, the RTOS can detect when best-effort tasks interfere with real-time tasks in the shared cache. Then, the RTOS can prevent best-effort tasks from interfering with real-time tasks. The results also show that the assignment of exclusive partitions to HRT tasks together with the two-phase multicore scheduler provides HRT and SRT guarantees, even when best-effort tasks share partitions with real-time tasks.

**Keywords**: Real-time operating systems, multicore real-time scheduling, task partitioning, shared cache memory partitioning.

# LIST OF FIGURES

1 (a) An architecture with shared Level-2 cache. (b) An architecture with a cluster sharing Level-2 and all cores sharing the Level-3. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

2 Example of a direct-mapped cache. A main memory block can be placed in only one of the four cache lines. . . 48

3 Example of a fully-associative cache. A main memory block are placed in any of the four cache lines. . . . . . . . . . 49

4 The format of an address in a fully-associative cache organization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

5 An example of a 2-way set-associative cache. A main memory block can be placed in one of the two cache locations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

6 The form of an address in a n-way set-associative and direct-mapped cache organizations. . . . . . . . . . . . . . . . . . . . 50

7 Translation of a linear address into physical address in the Intel 32-bits paging mode (Intel Corporation, 2011). . . . 53

8 Example of the data consistency problem in a multicore processor (HENNESSY; PATTERSON, 2006). . . . . . . . . . . . . 54

9 MESI protocol state diagram (SONG, 2013). . . . . . . . . . . . 59

10 MOESI protocol state diagram (SONG, 2013). . . . . . . . . . . 60

11 Comparison of MESIF (a) and MESI (b) protocols. MESIF protocol reduces the bus traffic (QIAN; YAN, 2008). . . 62

12 Illustration of Local APIC and I/O APIC in the processor underlying the experiments in Chapters 5, 6, 7, and 8. The Local APICs handle per-processor interrupts, such as timer and PMU interrupts, and IPIs. External devices generate interrupts through the I/O APIC. . . . . . . . . . . . . 66

13 Illustration of APIC Timers and TSCs in the processor underlying the evaluations in Chapters 5, 6, 7, and 8. . . . 68

14 States of a task/process in an OS (TANENBAUM, 2007). . 71

15 (a) single-threaded and (b) multi-threaded processes (SILBERSCHATZ *et al.*, 2008). . . . . . . . . . . . . . . . . . . . . . . 72

16 Writing a character to an UART device using PIO. . . . . . 75

17 The handling of an interrupt by the OS (TANENBAUM, 2007). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

18 Illustration of HRT and SRT utility functions (BURNS, 1991). (a) HRT utility function. (b) SRT utility function. There is no value in a late HRT job, whereas for a late SRT job, the value decreases with increasing tardiness. 83
19 Overview of the three schedulability test classes. . . . . . . . 84
20 An example of RM scheduling with two tasks. . . . . . . . . . . 88
21 An example of EDF scheduling with two tasks. . . . . . . . . . 91
22 Example of the first-fit bin-packing heuristic. . . . . . . . . . . 95
23 Example of the best-fit bin-packing heuristic. . . . . . . . . . . 95
24 Example of the worst-fit bin-packing heuristic. . . . . . . . . . 96
25 Example of the worst-fit decreasing bin-packing heuristic. 97
26 An example of the sources of overhead in an RTOS (BRANDENBURG; ANDERSON, 2009). . . . . . . . . . . . . . . . . . . . . . . . . 103
27 Classification of cache partitioning approaches: (a) overview of the index-based partitioning. (b) overview of the way-based partitioning. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
28 Examples of cache locking variations: (a) overview of the way locking. (b) overview of the cache line locking. . . . . . 111
29 Physical address view from the cache (on top) and from the OS (bottom) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
30 Mapping physical pages to cache locations. . . . . . . . . . . . . 115
31 Digram of the LITMUS$^{RT}$ main components. Adaptation from (BRANDENBURG, 2011). . . . . . . . . . . . . . . . . . . . . 149
32 During the system compilation, hardware mediators are diluted into components. (a) hardware mediators before compilation and (b) hardware mediators after compilation.167
33 A part of the IA32 CPU hardware mediator. The *cas* method is declared as inline. . . . . . . . . . . . . . . . . . . . . . . . . 168
34 ADESD methodology overview. . . . . . . . . . . . . . . . . . . . . . 170
35 UML class diagram for the proposed PMU hardware mediator API. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
36 Performance monitoring OS component. . . . . . . . . . . . . . . . 174
37 Perf_Mon using the hardware mediator. The hardware mediator code is “dissolved” into the component at compile time. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175
38 An example of how to use the proposed API. . . . . . . . . . . 176
39 API usage example: number of snoops in the L1 data cache for the three benchmark applications. . . . . . . . . . . . . 177

40 Number of snoops in the L1 data cache for the three benchmark applications running in Linux. . . . . . . . . . . . . 178
41 Modifying the OS scheduling. . . . . . . . . . . . . . . . . . . . . . . . . 179
42 UML class diagram of the real-time scheduling components on EPOS: Thread, Criterion, Scheduler, and Scheduling list classes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182
43 UML class diagram of the real-time scheduling components on EPOS: criterion sub classes. . . . . . . . . . . . . . . . 184
44 An example about the interaction among the `Criterion`, `Trait`, and `Scheduler` classes. . . . . . . . . . . . . . . . . . . . . . 185
45 Periodic thread code example. . . . . . . . . . . . . . . . . . . . . . . 186
46 UML sequence diagram of thread sleep method. . . . . . . . . 187
47 UML sequence diagram of thread wake up method. . . . . . 188
48 UML sequence diagram of thread reschedule method. . . . 190
49 CPU hardware mediator UML class digram. . . . . . . . . . . . 191
50 UML communication diagram summarizing the sources of run-time overhead in EPOS. (a) Operations initiated by the periodic thread sleep operation. (b) Operations initiated by the hardware timer. . . . . . . . . . . . . . . . . . . . . . 193
51 IA32 MMU hardware mediator. . . . . . . . . . . . . . . . . . . . . . 195
52 UML class diagram for address space and segment components. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196
53 IA32 MMU trait class responsible for enabling page coloring. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196
54 UML sequence diagram of the colored application heap initialization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 198
55 Overload of the EPOS `new` operator. . . . . . . . . . . . . . . . . . 199
56 Tracing overhead in LITMUS$^{RT}$ and EPOS. (a) Total application execution time and (b) individual thread execution time. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 206
57 Average (a) and worst-case (b) context switch overhead. . 208
58 Average (a) and worst-case (b) IPI latency. . . . . . . . . . . . 210
59 Average (a) and worst-case (b) scheduling overhead. . . . . 212
60 Average (a) and worst-case (b) tick counting overhead. . . 213
61 Average (a) and worst-case (b) thread release overhead. . 215
62 CPMD application code. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 217
63 Cache-related preemption and migration delay varying the WSS in microseconds. Note that the $y$-axis uses a logarithm scale. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 218

64 Comparison between G-EDF and P-EDF with short periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 220
65 Comparison between G-EDF and P-EDF with moderate periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 221
66 Comparison between G-EDF and P-EDF with long periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 223
67 Weighted schedulability for short periods. . . . . . . . . . . . . . 227
68 Weighted schedulability for light uniform distribution and long periods. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 228
69 Comparison between G-EDF and P-EDF using uniform utilizations: (a), (b), and (c) light uniform. (d), (e), and (f) medium uniform, and (g), (h), and (i) heavy uniform. 230
70 Comparison between G-EDF and P-EDF using bimodal utilizations: (a), (b), and (c) light bimodal, (d), (e), and (f) medium bimodal, and (g), (h), and (i) heavy bimodal. 231
71 Part of the task function source code. . . . . . . . . . . . . . . . . . 239
72 (a) Obtained worst-case scaling factors (hard real-time). (b) Obtained average scaling factors (soft real-time). . . . . 242
73 Percentage of tasks that missed their deadline when varying the data size and using the P-EDF and G-EDF schedulers in S1 and S3. . . . . . . . . . . . . . . . . . . . . . . . . . . . 244
74 Total application execution time when varying the data size and P-EDF and G-EDF with and without page coloring. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 245
75 Overview of the proposed color-aware task partitioning mechanism. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 253
76 Example of the CAP WFD algorithm with eight tasks and three cores. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 259
77 The partition of the same task set as in Figure 76 with the original WFD heuristic. . . . . . . . . . . . . . . . . . . . . . . . . . 260
78 Part of the task function source code. . . . . . . . . . . . . . . . . . 262

79 Percentage of missed deadlines of the WFD heuristic. (a) WSS of 32KB and write ratio of 33%; (b) WSS of 32KB and write ratio of 20%. . . . . . . . . . . . . . . . . . . . . . . . . 264
80 Percentage of missed deadlines of the WFD heuristic. (a) WSS of 64KB and write ratio of 33%; (b) WSS of 64KB and write ratio of 20%. . . . . . . . . . . . . . . . . . . . . . . . . 265
81 Percentage of missed deadlines of the WFD heuristic. (a) WSS of 128KB and write ratio of 33%; (b) WSS of 128KB and write ratio of 20%. . . . . . . . . . . . . . . . . . . . . . . 267
82 Percentage of missed deadlines of the WFD heuristic. (a) WSS of 256KB and write ratio of 33%; (b) WSS of 256KB and write ratio of 20%. . . . . . . . . . . . . . . . . . . . . . . 268
83 The partition of the task set in Table 15 with the CAP-GS algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 283
84 The partition of the same task set as in Figure 83 with the original BFD heuristic. . . . . . . . . . . . . . . . . . . . . . . . . . 283
85 Part of the source code of the HPC test application. . . . . 286
86 Measured number of HITM events. (a) WSS of 128 KB (b) WSS of 512 KB. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 288
87 Measured number of HIT remote events. (a) WSS of 128 KB (b) WSS of 512 KB. . . . . . . . . . . . . . . . . . . . . . . . . 289
88 Measured number of local HIT and remote MISS events. (a) WSS of 128 KB (b) WSS of 512 KB. . . . . . . . . . . . . . . 290
89 Measured number of L2 cache lines in the I state. (a) WSS of 128 KB (b) WSS of 512 KB. . . . . . . . . . . . . . . . . . . 291
90 Measured number of L2 cache lines in the S state. (a) WSS of 128 KB (b) WSS of 512 KB. . . . . . . . . . . . . . . . . . . 293
91 UML class diagram of the Color-Aware P-RM scheduling policy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
92 Traits of the Thread class. . . . . . . . . . . . . . . . . . . . . . . . . 295
93 New Traits class of Scheduler<Thread>. . . . . . . . . . . . . . . 296
94 Part of the source code of the Thread init method. . . . . . 297
95 Example of the CA_PRM criterion init method. . . . . . . . . 297
96 Taking a scheduling decision at every PMU sampling period. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 298
97 Collecting the HPCs values at every sampling period. . . . 298
98 Overview of the two-phase multicore real-time scheduler. 299
99 Part of the best-effort tasks function source code. . . . . . . . 302

100 Percentage of missed deadlines of a WSS of 32 KB. (a) Write ratio of 33%; (b) Write ratio of 20%. . . . . . . . . . . . . . 305
101 Percentage of missed deadlines of a WSS of 64 KB. (a) Write ratio of 33%; (b) Write ratio of 20%. . . . . . . . . . . . . . 306
102 Percentage of missed deadlines of a WSS of 128 KB. (a) Write ratio of 33%; (b) Write ratio of 20%. . . . . . . . . . . . . . 307
103 Percentage of missed deadlines of a WSS of 256 KB. (a) Write ratio of 33%; (b) Write ratio of 20%. . . . . . . . . . . . . . 309
104 Average deadline tardiness. (a) Write ratio of 33%; (b) Write ratio of 20%. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 311
105 Total execution time of the test application. (a) Write ratio of 33%; (b) Write ratio of 20%. . . . . . . . . . . . . . . . . . . . 313

# LIST OF TABLES

1 The permitted states of a given cache line for any pair of caches in the MESI protocol (WIKIPEDIA, 2014b). .... 58

2 The permitted states of a given cache line for any pair of caches in the MOESI protocol (WIKIPEDIA, 2014d). .. 60

3 The permitted states of a given cache line for any pair of caches in the MESIF protocol (WIKIPEDIA, 2014c). ... 61

4 Summary of notation and constraints in the periodic and sporadic task models (BRANDENBURG, 2011). .......... 80

5 Summary of sources of overhead and our notation (BRANDENBURG, 2011). ........................ 104

6 Comparative table of the reviewed state-of-the-art mechanisms of index-based cache partitioning with hardware-specific implementations. .................. 134

7 Comparative table of the reviewed state-of-the-art mechanisms of index-based cache partitioning implemented in software. ........................................ 136

8 Comparative table of the reviewed state-of-the-art way-based cache partitioning mechanisms. ................ 137

9 Comparative table of cache locking state-of-the-art mechanisms. ........................................ 139

10 Comparative table of OS memory allocators. .......... 140

11 Comparative table of the features of the analyzed multicore (real-time) scheduling algorithms. .............. 164

12 Intel i7-2600 processor features and LITMUS$^{RT}$ version. 203

13 Summary of the schedulability ratio comparison between P-EDF, C-EDF and G-EDF, considering also the run-time overhead in EPOS and LITMUS$^{\mathrm{RT}}$. ............. 225

14 Parameters of a task set used to exemplify the CAP WFD algorithm. ...................................... 258

15 Parameters of a task set used to exemplify the CAP-GS algorithm. ........................................ 282

16 Selected hardware events for the Sandy Bridge Intel microarchitecture (i7-2600 processor). .................. 285

17 WCET, AVG execution time, and standard deviation (STD) of the timer handler and `Thread::collect_hpcs` methods. All values are in µs. ....................... 314

# LIST OF ABBREVIATIONS

| | |
|---|---|
| ADESD | Application-Driven Embedded System Design. |
| AIRS | Advanced Interactive Real-time Scheduler. |
| AOP | Aspect-Oriented Programming. |
| API | Application Programming Interface. |
| APIC | Advanced Programmable Interrupt Controller. |
| AQuoSA | Adaptive Quality of Service Architecture. |
| ART Linux | Advanced Real-Time Linux. |
| BE | Best-Effort. |
| BFD | Best-Fit Decreasing. |
| C-EDF | Clustered-EDF. |
| C-RM | Clustered-RM. |
| CA P-RM | Color-Aware Partitioned-RM. |
| CAP-GS | Color-Aware task Partitioning with Group Splitting. |
| CBD | Component-Based Design. |
| CBS | Constant Bandwidth Serve. |
| ccNUMA | cache-coherent Non-Uniform Memory Access. |
| CF | Compact-FIT. |
| CIF | Cache-Index Friendly. |
| CPI | Cycles per Retired Instruction. |
| CPMD | Cache-related Preemption and Migration Delay. |
| CPU | Central Processing Unit. |
| CSI | Cycle-Stealing Interrupts. |
| DI | Device Interrupts. |
| DM | Deadline Monotonic. |
| DMA | Direct Memory Access. |
| EDF | Earliest-Deadline First. |
| EDF-WMR | EDF-Window-constrained Migration and Reservation. |
| EPOS | Embedded Parallel Operating |

| | |
|---|---|
| | System. |
| F-CBS | Flexible Constant Bandwidth Serve. |
| FBD | Family-Based Design. |
| FFD | First-Fit Decreasing. |
| FIFO | First In First Out. |
| FP | Fixed-Priority. |
| G-DM | Global-DM. |
| G-EDF | Global-EDF. |
| G-FP | Global Fixed-Priority. |
| G-JLDP | Global Job-Level Dynamic-Priority. |
| G-JLFP | Global Job-Level Fixed-Priority. |
| G-RM | Global-RM. |
| GPOS | General-Purpose Operating System. |
| HAL | Hardware Abstraction Layer. |
| HPC | Hardware Performance Counter. |
| I/O | Input and Output. |
| IC | Interrupt Controller. |
| IMA | Integrated Modular Avionics. |
| IPI | Inter-Processor Interrupts. |
| IRQ | Interrupt Request. |
| ISR | Interrupt Service Routine. |
| JLDP | Job-Level Dynamic-Priority. |
| JLFP | Job-Level Fixed-Priority. |
| KURT Linux | Kansas University Real-Time Linux. |
| Linux/RK | Linux/Resource Kernel. |
| LITMUS$^{RT}$ | LInux Testbed for MUltiprocessor Scheduling in Real-Time systems. |
| LLC | Last Level Cache. |
| LLF | Least-Laxity First. |
| LRU | Least Recently Used. |
| LWFG | Largest Working set size First, Grouping. |
| MMU | Memory Management Unit. |
| MRU | Most Recently Used. |

| | |
|---|---|
| NMI | Non-Maskable Interrupts. |
| OOD | Object-Oriented Design. |
| OS | Operating System. |
| P-EDF | Partitioned-EDF. |
| P-FP | Partitioned-FP. |
| P-LLF | Partitioned-LLF. |
| PAPI | Performance API. |
| PF | Proportional Fairness. |
| PIO | Programmable I/O. |
| PLRU | Pseudo-LRU. |
| PLRUm | MRU-based Pseudo-LRU. |
| PLRUt | Tree-based Pseudo-LRU. |
| PMU | Performance Monitoring Unit. |
| POSIX | Portable Operating System Interface. |
| QoS | Quality of Service. |
| RED Linux | Real-Time and Embedded Linux. |
| RM | Rate Monotonic. |
| RTA | Response Time Analysis. |
| RTAI | Real-Time Application Interface. |
| SMI | System Management Interrupt. |
| SMM | System Management Mode. |
| SMP | Symmetric Multi-Processor. |
| SMT | Symmetric Multi-Threading. |
| TI | Timer Interrupts. |
| TLB | Translation Lookaside Buffer. |
| TLSF | Two-Level Segregated Fit memory allocator. |
| TSC | TimeStamp Counter. |
| UART | Universal Asynchronous Receiver/Transmitter. |
| UMA | Uniform Memory Access. |
| WCET | Worst-Case Execution Time. |
| WFD | Worst-Fit Decreasing. |
| WSS | Working Set Size. |

# LIST OF SYMBOLS

| | |
|---|---|
| $\tau$ | A task set |
| $T_i$ | The $i^{th}$ task |
| $J_{i,j}$ | The $j^{th}$ job of a task $T_i$ |
| $e_i$ | Execution time of a task $T_i$ |
| $p_i$ | Period of $T_i$ or the minimum interval between jobs |
| $d_i$ | Relative deadline of a task $T_i$ |
| $u_i$ | Utilization of a task $T_i$ |
| $\delta_i$ | Density of $T_i$ |
| $r_{i,j}$ | $J'_{i,j}$ release time |
| $d_{i,j}$ | $J'_{i,j}$ absolute deadline |
| $f_{i,j}$ | $J'_{i,j}$ completion time |
| $R_{i,j}$ | $J'_{i,j}$ response time |
| $R_i$ | Maximum response time of $T_i$ |
| $m$ | Number of processors in the target platform |
| $\Delta^{cxs}$ | Time interval to switch the context between two threads |
| $\Delta^{scd}$ | Time interval to choose a thread to run |
| $\Delta^{rel}$ | Time interval to release all threads that have reached their release time |
| $\Delta^{tck}$ | Time interval to count a tick in a periodic timer interrupt |
| $\Delta^{ipi}$ | Delay until IPI is received (IPI latency) |
| $\Delta^{cpd}$ | Delay caused by the loss of cache affinity (CPMD) |

# TABLE OF CONTENTS

**1 INTRODUCTION** .......... 31
1.1 PROBLEM OVERVIEW .......... 32
1.2 CONTRIBUTIONS .......... 36
**1.2.1 Design and Implementation of a Multicore RTOS** 37
**1.2.2 Run-Time Overhead Analysis in an RTOS** .......... 38
**1.2.3 Cache Partitioning Analysis in an RTOS** .......... 39
**1.2.4 Shared Cache-Aware Task Partitioning** .......... 40
**1.2.5 Two Phases Multicore Real-Time Scheduler** .......... 41
1.3 DOCUMENT ORGANIZATION .......... 43
**2 BACKGROUND** .......... 45
2.1 PROCESSOR AND MEMORY ARCHITECTURES .......... 45
**2.1.1 Current Multicore Memory Architectures** .......... 45
**2.1.2 Cache Organization and Cache Line Replacement Algorithms** .......... 47
**2.1.3 Address Translation** .......... 52
**2.1.4 Cache Coherence Protocols** .......... 54
**2.1.5 Hardware Performance Counters** .......... 63
**2.1.6 Interrupts** .......... 64
**2.1.7 Timers and Clocks** .......... 66
2.2 OPERATING SYSTEMS .......... 68
**2.2.1 Process Management** .......... 69
**2.2.2 Memory Management** .......... 73
**2.2.3 Input and Output Management** .......... 74
2.3 REAL-TIME TASK MODELS AND CONSTRAINTS .......... 77
**2.3.1 Hard and Soft Timing Constraints** .......... 81
**2.3.2 Schedulability, Feasibility, and Sustainability** .......... 82
2.4 REAL-TIME SCHEDULING .......... 85
**2.4.1 Uniprocessor Real-time Scheduling** .......... 87
2.4.1.1 Fixed-Priority Scheduling .......... 87
2.4.1.2 Job-Level Fixed-Priority Scheduling .......... 91
2.4.1.3 Job-Level Dynamic-Priority Scheduling .......... 92
**2.4.2 Multiprocessor Real-time Scheduling** .......... 93
2.4.2.1 Partitioned Scheduling .......... 93
2.4.2.2 Global Scheduling .......... 97
2.4.2.3 Clustered Scheduling .......... 100
2.5 RUN-TIME OVERHEAD ANALYSIS .......... 101
**3 RELATED WORK** .......... 105

3.1 MEMORY MANAGEMENT . . . 105
**3.1.1 Index-based Cache Partitioning Methods** . . . 112
**3.1.2 Way-based Cache Partitioning Methods** . . . 120
**3.1.3 Cache Locking Methods** . . . 123
**3.1.4 OS Memory Allocators** . . . 127
**3.1.5 Cache-related Preemption and Migration Delay** . . 131
**3.1.6 Summary** . . . 133
3.2 REAL-TIME OPERATING SYSTEMS . . . 138
**3.2.1 Embedded Real-Time Operating Systems** . . . 142
**3.2.2 POSIX-Like Real-Time Operating Systems** . . . 143
**3.2.3 General-Purpose Operating Systems with Real-Time Extensions** . . . 145
**3.2.4 LITMUS$^{\mathbf{RT}}$** . . . 148
**3.2.5 Summary** . . . 150
3.3 RUN-TIME PERFORMANCE MONITORING . . . 151
**3.3.1 Performance Monitoring APIs and Tools** . . . 151
**3.3.2 Run-Time OS Decisions** . . . 152
**3.3.3 Summary** . . . 155
3.4 MULTICORE REAL-TIME SCHEDULING . . . 155
**3.4.1 Scheduling Algorithms** . . . 155
**3.4.2 Run-time Overhead and Implementation Tradeoffs** 161
**3.4.3 Summary** . . . 162
**4 REAL-TIME SUPPORT ON EPOS** . . . 165
4.1 HARDWARE MEDIATORS . . . 166
**4.1.1 Performance Monitoring Unit** . . . 168
**4.1.2 PMU Hardware Mediator Family** . . . 171
**4.1.3 Performance Monitor** . . . 173
**4.1.4 API Usage** . . . 174
**4.1.5 Applicability for Scheduling** . . . 178
**4.1.6 Guidelines for future PMU designs** . . . 179
**4.1.7 Summary of the PMU support** . . . 181
4.2 SCHEDULING . . . 181
**4.2.1 Periodic Thread Operations** . . . 186
**4.2.2 Context Switching** . . . 189
**4.2.3 Alarm and Timer Interrupt Handler** . . . 191
**4.2.4 Summary of Real-Time Extensions and Overhead Sources** . . . 192
4.3 MEMORY MANAGEMENT . . . 194
**4.3.1 Original Memory Management in EPOS** . . . 194

**4.3.2 Page Coloring Support** . . . 196
**4.3.3 User-Centric Page Coloring** . . . 197
**4.3.4 OS-Centric Page Coloring** . . . 199
**4.3.5 Summary of Memory Management Extensions** . . . 199
**5 RUN-TIME OVERHEAD EVALUATION** . . . 201
5.1 EXPERIMENTS DESCRIPTION . . . 202
5.2 TRACING OVERHEAD . . . 204
5.3 CONTEXT SWITCH OVERHEAD EVALUATION . . . 205
5.4 IPI LATENCY EVALUATION . . . 207
5.5 SCHEDULING OVERHEAD EVALUATION . . . 209
5.6 TICK COUNTING OVERHEAD EVALUATION . . . 211
5.7 THREAD RELEASE OVERHEAD EVALUATION . . . 214
5.8 PREEMPTION/MIGRATION DELAY EVALUATION . . . 214
5.9 SCHEDULABILITY TESTS ANALYSIS . . . 218
**5.9.1 Run-Time Overhead** . . . 219
**5.9.2 Weighted Schedulability** . . . 224
**5.9.3 Schedulability Evaluation for 100 Processors** . . . 228
5.10 DISCUSSION . . . 231
**6 CACHE PARTITIONING EVALUATION** . . . 237
6.1 EXPERIMENT DESCRIPTION . . . 237
6.2 INDIVIDUAL TASK EXECUTION TIME . . . 240
6.3 DEADLINE MISSES . . . 241
6.4 TOTAL EXECUTION TIME . . . 243
6.5 DISCUSSION . . . 245
**7 STATIC COLOR-AWARE TASK PARTITIONING** 249
7.1 ASSUMPTIONS AND NOTATIONS . . . 250
7.2 COLOR-AWARE TASK PARTITIONING . . . 252
**7.2.1 Example: partitioning a task set with CAP WFD** 258
7.3 EVALUATION . . . 260
**7.3.1 Experiment Description** . . . 260
**7.3.2 Percentage of Missed Deadlines** . . . 261
7.4 DISCUSSION . . . 266
**8 DYNAMIC COLOR-AWARE SCHEDULING ALGORITHM** . . . 273
8.1 ASSUMPTIONS AND NOTATIONS . . . 275
8.2 ALGORITHM DESCRIPTION . . . 278
**8.2.1 Color-Aware Task Partitioning with Group Splitting** . . . 278
**8.2.2 Example: partitioning a task set with CAP-GS** . . . 281

**8.2.3 Dynamic Color-Aware Scheduling** . . . . . . . . . . . . . . . . 284
8.2.3.1 Analysis of Hardware Events . . . . . . . . . . . . . . . . . . . . . . 284
8.2.3.2 Design and Implementation . . . . . . . . . . . . . . . . . . . . . . 292
8.3 EVALUATION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 300
**8.3.1 Experiment Description** . . . . . . . . . . . . . . . . . . . . . . . . . 300
**8.3.2 Percentage of Missed Deadlines** . . . . . . . . . . . . . . . . . . 303
**8.3.3 Deadline Tardiness** . . . . . . . . . . . . . . . . . . . . . . . . . . . . 308
**8.3.4 Total Execution Time** . . . . . . . . . . . . . . . . . . . . . . . . . . 310
**8.3.5 Run-Time Overhead of the HPC Analysis** . . . . . . . . 312
8.4 DISCUSSION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 313
**9 CONCLUSION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 319
9.1 SUMMARY OF CONTRIBUTIONS . . . . . . . . . . . . . . . . . . . 319
**9.1.1 Real-Time Support on EPOS** . . . . . . . . . . . . . . . . . . . . 320
**9.1.2 Run-Time Overhead Evaluation** . . . . . . . . . . . . . . . . . . 321
**9.1.3 Cache Partitioning Evaluation** . . . . . . . . . . . . . . . . . . 322
**9.1.4 Static Color-Aware Task Partitioning** . . . . . . . . . . . . 323
**9.1.5 Dynamic Color-Aware Scheduling Algorithm** . . . . . 325
9.2 CLOSING REMARKS . . . . . . . . . . . . . . . . . . . . . . . . . . . 327
9.3 FUTURE DIRECTIONS . . . . . . . . . . . . . . . . . . . . . . . . . . 329
**REFERENCES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 331

# 1 INTRODUCTION

Real-time embedded systems are nowadays present in virtually any environment. Areas such as automotive, avionics, telecommunication, and consumer electronics are examples where real-time embedded systems can be vastly found. In a *real-time system*, the correctness of the system depends not only on its logical behavior, but also on the time in which the computation is performed (LIU; LAYLAND, 1973). The main distinction is between *soft real-time* (SRT) and *hard real-time* (HRT) systems. In both, applications are typically realized as a collection of *real-time tasks* associated with timing constraints and scheduled according to a chosen *scheduling algorithm.* However, in a HRT, the loss of a *deadline* may cause uncountable or catastrophic damage, such as human lives or a considerable amount of money. In a SRT, instead, missing a deadline results in just a degradation of the Quality of Service (QoS) provided by the system. Hence, in order to provide a correct behavior and a good QoS, a fundamental requirement for real-time embedded applications is that they must be designed to always meet their deadlines.

At the same time, the continuous evolution of processor technology, together with its decreasing cost, has enabled multicore architectures (*e.g., Symmetric Multiprocessing - SMP*) to be also used in the real-time embedded system domain (CHO *et al.*, 2006). Besides, real-time embedded applications are demanding more processing power due to the evolution and integration of features that can only be satisfied, in a cost-effective way, by the use of a multicore platform. In an automotive environment, for instance, new safety functionalities like “automatic emergency breaking” and “night view assist” must read and fuse data from sensors, process video streams, and rise warnings when an obstacle is detected on the road under real-time constraints (MOHAN *et al.*, 2011). This is a typical scenario where an increasing demand for advanced features results in a proportional demand for additional computational power. Moreover, an increase in system functionality determines additional costs in terms of power consumption, heat dissipation, and size (*e.g.,* wiring) (CULLMANN *et al.*, 2010). Thereby, multicore processors represent a cost-effective solution to decrease the above-mentioned costs, since the additional computational demand can be allocated from a single processing unit, instead of several processing units spread over the vehicle.

Another advantage of a multicore processor in the context of real-time embedded systems is the development flexibility. Several embedded real-time applications are implemented in hardware logic to obtain maximum performance and fulfill all application's requirements (processing, real-time deadlines, energy consumption, and so on). For instance, digital signal processing algorithms and baseband processing in wireless communication must process big amount of data under real-time conditions. Nevertheless, as they are usually implemented in a dedicated hardware, these applications present restrictions in terms of developing support (*e.g.,* bug fixes, updating, and maintainability). In a multicore processor, instead, the same applications can be implemented in software, with similar performance and more flexibility. In this context, an application is implemented on top of a Real-Time Operating System (RTOS), composed of several real-time cooperating tasks (tasks that share data).

The goal of this thesis is to provide support, at the Operating System (OS) level, for real-time applications to safely execute on top of multicore platforms. However, the execution of real-time tasks on multicore platforms has several challenges, which are discussed below.

## 1.1 PROBLEM OVERVIEW

In general, the capability of a real-time system to meet its deadline is verified through a *schedulability analysis*, i.e., an algorithm that takes as input a scheduling algorithm and a collection of tasks' parameters, and returns true if the task set is guaranteed to meet all deadlines under any possible sequence of operations compatible with the target system specification (LIU, 2000; DAVIS; BURNS, 2011). A basic assumption, which is common to all such schedulability analysis techniques, is that an upper bound on the *Worst-Case Execution Time* (WCET) of each task is known. However, deriving safe yet tight bounds on task WCET is becoming increasingly difficult. This is especially true for multicore architectures, because they have been designed to achieve maximum performance and therefore feature several latency-hiding mechanisms on the paths to hardware resources. The memory hierarchy, busses, and I/O peripherals in such architectures are usually bridged to the processors using caches, FIFOs, read-ahead engines, transaction builders, and many other complex techniques that impact the estimation of the WCET. Also, these hardware resources are shared among the processors or cores. Therefore, operations performed by one processing unit can result in unregulated contention at the level of any shared resource and thus unpredictably delay the execution of a task running on a different processor.

One of the main factors for unpredictability is the *CPU cache memory hierarchy* (WEHMEYER; MARWEDEL, 2005; SUHENDRA; MITRA, 2008; MURALIDHARA *et al.*, 2010; ZHURAVLEV *et al.*, 2010; ZHURAVLEV *et al.*, 2012). CPU caches on modern multiprocessors are typically organized in two or three levels and placed between processor and main memory to bridge the gap on the high processor speed and low memory performance. Usually, a smaller Level-1 (L1) cache is private to each core, while a larger Level-2 (L2) and/or a Level-3 (L3) cache is shared among all the cores or a cluster of cores. The last level of cache before the main memory is usually referred as Last Level Cache (LLC). A cache may experience a *cache hit* when an instruction or data to be fetched already resides in the first level of cache or a *cache miss* when the instruction or data is not in the first level of cache. In case of a cache miss, the requested instruction or data must be brought from the higher levels of cache or directly from the main memory, incurring in larger execution times.

CPU caches are hardware components that are mostly transparent to the programmer and that rely on temporal and spatial locality of memory accesses to reduce the average execution time of applications. Thereby, they employ a set of heuristics to keep data that are more likely to be accessed in a near future and displace old, non-referenced entries. At a high level, the heuristic behavior of a cache means that a memory access in the same location throughout the execution of a task, may or may not result in a *cache hit*, depending on the history of the system. In fact, the execution pattern of the task itself, of different tasks on the same core or even of a set of tasks on a different core can impact the *cache hit ratio* of a given memory access pattern. This means that a complex function of the system history can directly impact the time required by a task to retrieve data from the memory hierarchy, explaining why cache memories are one of the main sources of unpredictability.

The contention for cache space is a complex problem, because cache size is typically limited and real-time tasks compete over the assignment of cache lines throughout their executions. When real-time tasks executing on different cores in parallel access the same cache lines due to true or false sharing[1], the cache coherence protocol implemented in hardware invalidates the private cache's data, causing an implicit delay in the application's execution time. Cache line invalidations performed by the cache coherence protocol take hundreds of cycles (about the same time as accessing the off-chip RAM) and may originate two kinds of scaling problem (BOYD-WICKIZER *et al.*, 2010): access serialization to the same cache line carried out by the cache coherence protocol, which prevents parallel speedup, and saturation in the inter-core interconnection, also preventing parallel processing gains.

The contention for shared cache memory space causes a decrease in the application's throughput and may lead to deadline losses (KIM *et al.*, 2013; KENNA *et al.*, 2013; GRACIOLI; FRÖHLICH, 2013). One can argue that there would be processing speedup by just turning the cache off and using the main memory directly. Nevertheless, it is a misconception that WCETs with caches are equal to ones without caches (LIEDTKE *et al.*, 1997). Moreover, it is common to find a considerable inter-thread interaction in multithreaded applications. For example, some applications from NAS parallel and SPEC OMG benchmark suites have up to 20% of inter-thread interaction, and up to 60% of this interaction is affected by cache contention (MURALIDHARA *et al.*, 2010). Reducing the effects of cache contention can significantly improve the application's overall performance and prevent deadline misses.

These problems in the cache memory hierarchy lead us to a first research question: *Can the RTOS offer a mechanism to reduce the contention for shared cache space and provide HRT/SRT guarantees*?

Several recent works have been proposed to deal with shared caches and to provide real-time guarantees for multicore applications. The most common and successful approach is shared cache partitioning (WOLFE, 1994; LIEDTKE *et al.*, 1997; SUHENDRA; MITRA, 2008; KIM *et al.*, 2013; KENNA *et al.*, 2013; GRACIOLI; FRÖHLICH, 2013). By assigning separate shared cache partitions to individual tasks, it is possible to isolate task workloads that interfere with one another, leading to increased system predictability. An alternative approach is to use cache partitioning together with a cache locking mechanism, which prevents cache lines from being evicted during program execution (SUHENDRA; MITRA, 2008; SARKAR *et al.*, 2012; MANCUSO *et al.*, 2013). The

[1] False sharing occurs when different tasks running on different processors modify data that reside on the same cache lines.

main drawback of cache locking, however, is that it requires specific hardware support that is not present in many commercial processors.

Other recent works proposed to deal with shared cache through real-time scheduling (CALANDRINO; ANDERSON, 2008; GUAN *et al.*, 2009). Tasks that may thrash the shared cache are grouped together offline and at run-time, a scheduling policy reduces the concurrency within groups to increase the predictability for SRT applications (CALANDRINO; ANDERSON, 2008). Another proposed approach is to divide the shared cache into partitions and schedule a task only if it gets an idle core and enough cache partitions (GUAN *et al.*, 2009).

The cache partitioning and cache-aware scheduling approaches proposed so far do not consider scenarios where real-time tasks access the same cache lines due to true or false sharing. Usually, the task model uses independent tasks, underestimating the contention for shared cache lines. When cooperating tasks interact with each other either by sharing data or sharing cache partitions, the cache coherence protocol invalidates shared cache lines whenever they are written, which increases the application execution time.

In general, these recent cache partitioning studies implement and evaluate the proposed approaches either in a simulated environment or in a general-purpose operating system (GPOS) patched with real-time extensions (SARKAR *et al.*, 2012; MANCUSO *et al.*, 2013; KENNA *et al.*, 2013). Despite being a good development platform, real-time Linux-based studies suffer from the inherent non real-time behavior of Linux, which interferes with the cache system (when handling an interrupt, for instance) and may limit the gains obtained by cache partitioning. Also, due to the complexity of the Linux kernel, it is complicated to apply cache partitioning to internal OS data structures and to evaluate the impact of kernel activities, such as interrupt handling and context switching, which may have a considerable impact on real-time tasks.

Moreover, the cache partitioning mechanism may also impact the performance of the scheduling algorithm. In partitioned-based scheduling, tasks are assigned to individual processors and remain on each processor until completing execution. Thus, when a preempted task resumes its execution, part of its data may still be loaded in the same processor's private cache. In global-based scheduling, tasks are instead allowed to migrate among processors during the program execution. Consequently, there is cross-core communication initiated by the cache coherence protocol, which increases the WCET. The limitations of the current cache partitioning approaches lead us to further questions: *Does cache partitioning impact the performance of different scheduling algorithms*? *Does RTOS internal activities cause deadline misses?*

Additionally, considering the RTOS point of view, implementation issues, such as scheduling data structures and interrupt handling, impact schedulability as much as scheduling theoretic issues (BRANDENBURG; ANDERSON, 2009; BASTONI *et al.*, 2010b). All the recent works identified in the literature that have measured the influence of run-time overhead in SRT and HRT multicore schedulers also used a Linux-based infrastructure to support their studies (BRANDENBURG; ANDERSON, 2009; BASTONI *et al.*, 2010b; LELLI *et al.*, 2012). As stated before, Linux-based studies suffer from the inherent non real-time behavior of Linux, which affects the run-time overhead observed in these works. This leads us to further questions: *Are there significant differences between Linux-based real-time patches and an RTOS designed from scratch in terms of run-time overhead*? *Are these differences significant for HRT applications*? *What is the influence of this run-time overhead when incorporated into global and partitioned schedulability analyses*?

These open questions motivated the research behind this thesis in the sense of investigating how an RTOS could use real-time scheduling together with cache partitioning and other OS techniques to decrease the contention for shared cache lines while still providing SRT/HRT guarantees. Below, we present the contributions of this work.

## 1.2 CONTRIBUTIONS

In this thesis, we propose a multicore RTOS infrastructure to support the execution of real-time and best-effort (BE) tasks that interfere with each other, either by explicitly sharing data or accessing the same cache lines (*i.e.,* false sharing). We propose and evaluate two different scheduling approaches that provide SRT and HRT guarantees when real-time and/or BE tasks access the same cache lines. The first scheduling approach relies on task and cache partitioning techniques while the second approach combines task and cache partitioning with scheduling decisions performed at run-time. In the following, we briefly summarize the contributions presented in the subsequent chapters.

### 1.2.1 Design and Implementation of a Multicore RTOS

Central to our work is that current research works of cache partitioning and run-time overhead analysis evaluate their contributions using real-time patches applied to GPOSes. The main reason for that is that there is no RTOS designed from scratch that supports global and clustered multicore real-time schedulers. To bridge this existing gap in the real-time system community, we design and implement a multicore real-time infrastructure on top of the Embedded Parallel Operating System (EPOS) (FRÖHLICH, 2001; EPOS, 2014).

The multicore infrastructure extends the EPOS uniprocessor real-time scheduler to support global, partitioned, and clustered multicore variants of the Earliest Deadline First (EDF), Rate Monotonic (RM), Deadline Monotonic (DM), and Least Laxity First (LLF) scheduling policies. In the proposed multicore scheduling design, it is straightforward to add any new partitioned, global, or clustered scheduling policies to the system.

Moreover, we present the design and implementation of an original cache partitioning mechanism for a component-based RTOS. The mechanism is able to assign partitions to the OS internal data structures and does not rely on any specific hardware support. Additionally, two different approaches are supported that define from which partition data should be allocated: the user-centric and the OS-centric approaches.

Also, we propose an Application Programming Interface (API) for monitoring hardware events specifically designed for multicore (real-time) embedded systems. The API is designed following the Application-Driven Embedded System Design (ADESD) concepts (FRÖHLICH, 2001), and it is able to provide with OSes a simple and lightweight interface for handling the communication between applications and Performance Monitoring Units (PMU). Through an API usage example, in which a hardware event is used by the OS to scheduling decisions, we were able to identify the main drawbacks of the current PMUs. As a consequence, we propose a set of guidelines, such as the monitoring of address space intervals and OS trap generation, that can help hardware designers to improve the PMU capabilities in the future, from the RTOS perspective. The API is used by a proposed multicore real-time scheduler to detect when real-time and best-effort tasks access the same cache lines (see Section 1.2.4).

Additionally, we show how a well-designed object-oriented component-based RTOS allows code reuse of system components (*e.g.,* scheduler, thread, semaphore, etc) and easy global, clustered, and partitioned real-time scheduling extensions. To the best of our knowledge, EPOS is the first open-source RTOS that supports all multicore scheduling variants (global, partitioned, and clustered) of EDF, RM, DM, and LLF. We believe that EPOS can be used to conduct research for multicore HRT/SRT related areas due to higher predictability and smaller overhead compared to real-time patches for GPOSes.

### 1.2.2 Run-Time Overhead Analysis in an RTOS

The design and implementation of a multicore real-time infrastructure in EPOS has opened several research paths. We measure the run-time overhead in EPOS and compare it with the run-time overhead of a GPOS patched with real-time extensions. Then, we compare the schedulability ratio of several task sets using the partitioned, global, and clustered real-time schedulers when they consider the RTOS run-time overhead. In summary, we make the following contributions in the context of run-time overhead analysis of EPOS:

- We show that the RTOS run-time overhead, when incorporated into Global-EDF (G-EDF), Clustered-EDF (C-EDF), and Partitioned-EDF (P-EDF) schedulability tests, can provide HRT guarantees close to the theoretical schedulability tests. Moreover, in some cases, G-EDF considering the overhead in EPOS is superior to P-EDF considering the overhead of a GPOS patched with real-time extensions, which proves that the run-time overhead plays an important role on the G-EDF, C-EDF, and P-EDF schedulability analyses, because partitioned schedulers usually have smaller run-time overhead than global or clustered schedulers (BASTONI *et al.*, 2010b; GRACIOLI *et al.*, 2013). Also, the schedulability ratio achieved by EPOS is better than the GPOS with real-time extensions. For instance, the schedulability ratio for C-EDF considering the overhead of EPOS is up to 26% better than in the patched GPOS.
- We provide a comparison in terms of task set schedulability ra-

tio between P-EDF, C-EDF and G-EDF, considering also the OS overhead, for HRT tasks. P-EDF has obtained the same or better performance than G-EDF and C-EDF for all analyzed scenarios. In our experiments, P-EDF, C-EDF, and G-EDF had the same behavior for task sets composed only of heavy tasks, mainly because of G-EDF's schedulability test bounds. We observed a slight improvement in G-EDF for this heavy tasks scenario compared to related work (CALANDRINO *et al.*, 2006), due to the use of up to date G-EDF schedulability tests (BERTOGNA; CIRINEI, 2007; BARUAH *et al.*, 2009).

- We measure the cache-related preemption and migration delay (CPMD) on a modern eight-core processor, with shared Level-3 cache, using hardware performance counters (HPCs). We use the obtained values to compare P-EDF, C-EDF, and G-EDF through the weighted schedulability metric (BASTONI *et al.*, 2010b).

### 1.2.3 Cache Partitioning Analysis in an RTOS

The original cache partitioning mechanism implemented in EPOS made possible the following contributions:

- We evaluate the performance of cache partitioning using the P-EDF, C-EDF, and G-EDF schedulers when they have total utilization close to the theoretical HRT bounds. Our evaluation is carried out on a modern eight-core processor, with shared Level-3 cache. Our results indicate that cache partitioning has different behavior depending on the scheduler and task's working set size (WSS). We also show an experimental upper-bound in terms of HRT guarantees provided by cache partitioning in each scheduler.

- By allocating a different cache partition to the internal EPOS data structures, we evaluate the cache interference caused by the RTOS. We show that a lightweight RTOS, such as EPOS, does not impact HRT tasks with separated partitions.

### 1.2.4 Shared Cache-Aware Task Partitioning

The cache partitioning analysis in EPOS has shown interesting facts. For example, we observed an increase of up to 15 times on the observed WCET when tasks share cache partitions. The consequent variations on the execution time of real-time tasks made them miss up to 97% of their deadlines. Furthermore, a partitioned real-time scheduler yields tasks sharing cache partitions a more deterministic behavior. Partitioned scheduling for multicore real-time systems is also preferred by many researchers because each partition (or processor) is scheduled and analyzed using well-known uniprocessor scheduling algorithms and schedulability tests (BASTONI *et al.*, 2010b). Moreover, partitioned approaches have better HRT bounds and smaller run-time overhead than global or clustered approaches, which is proved in the run-time overhead analysis.

Thus, to provide a more deterministic behavior when tasks share cache partitions, we propose a task partitioning algorithm that assigns tasks to cores according to the usage of cache memory partitions. Specifically, tasks that share one or more partitions are grouped together and the whole group is assigned to the same core using a bin packing heuristic, avoiding inter-core interference caused by the access to the same cache lines. We compare our partitioning strategy with an original bin packing heuristic (worst-fit decreasing) in a real processor and using EPOS. Our results indicate that our task partitioning mechanism is able to provide HRT guarantees that are not achieved by traditional task partitioning algorithms.

In summary, the main contributions of the shared cache-aware task partitioning are:

- We propose a Color-Aware task Partitioning (CAP) algorithm that assigns tasks to cores respecting the usage of shared cache memory partitions. Shared cache partitioning is performed by the page coloring mechanism implemented in EPOS (GRACIOLI; FRÖHLICH, 2013). We assume that each task uses a set of colors that serves as input to CAP. Then, tasks that use the same colors are grouped together and the entire group is assigned to the same core using a bin packing heuristic. A restriction is that the group utilization must not exceed the processor capacity (given by the scheduling algorithm and deadline category – see Sections 2.3

and 2.4 for details of deadline category and task partitioning, respectively).

- We compare the CAP approach with an original bin packing heuristic (worst-fit decreasing) in terms of deadline misses of several generated task sets using the P-EDF scheduler. Our results indicate that it is possible to avoid inter-core interference and deadline misses by simply assigning tasks that access shared cache lines to the same core.

- We evaluate the partitioned task sets by running them on a modern eight-core processor, with shared L3 cache using EPOS. The experimental evaluation on a real machine and RTOS demonstrates the effectiveness of proposed task partitioning mechanism.

### 1.2.5 Two Phases Multicore Real-Time Scheduler

The CAP algorithm uses a task model composed of only HRT tasks. However, this scenario may not be true for every real-time application. In fact, emerging real-time workloads, such as multimedia and entertainment, and some business computing applications, have highly heterogenous timing requirements and consist of HRT, SRT, and BE tasks (BRANDENBURG; ANDERSON, 2007). In such applications, there are few HRT tasks, which represent a small system utilization, and their deadlines must be guaranteed (RAJKUMAR, 2006). Also, deadline tardiness for SRT tasks and response times for BE jobs must be minimized (BRANDENBURG; ANDERSON, 2007).

Furthermore, CAP limits the utilization of a task group (*i.e.,* tasks that share colors or memory partitions) to the processor capacity (100% when the scheduling policy is EDF, for instance). However, the number of memory partitions in a multicore platform is limited. Consequently, real-time and BE tasks may eventually share memory partitions and the utilization of these tasks may be greater than the processor capacity.

To allow the execution of a heterogenous real-time application and deal with the drawbacks in CAP, we propose a two phases multicore real-time scheduler. The first phase is an extension of the CAP algorithm. Tasks that share memory partitions are also grouped together, but the group utilization is allowed to be greater than the capacity given by the scheduling policy for each core. When a group utilization is greater than the processor capacity, we perform a group splitting, removing the task with smaller utilization from the group and trying to partition the group again. The second phase is performed at run-time and consists of collecting tasks' information through the use of HPCs to detect when BE tasks are interfering with real-time tasks.

In summary, the main contributions of the proposed two phases multicore real-time scheduler are:

- We propose a two phases scheduling algorithm able to deal with a heterogeneous real-time application. The system model considers a real-time application composed of HRT, SRT, and BE tasks. HRT tasks do not share memory partitions with other HRT tasks nor with SRT or BE tasks. SRT tasks can share memory partitions with other SRT tasks or BE tasks. The objective is to have a scheduling mechanism that provides timing guarantees for HRT tasks, while minimizing deadline tardiness for SRT tasks and allowing the execution of BE tasks whenever possible.

- The first phase of the scheduling mechanism is an extension of the CAP algorithm, named Color-Aware task Partitioning with Group Splitting (CAP-GS). The objective of the CAP-GS is to allow a group of tasks to have a utilization greater than the capacity of a processor considering a specific scheduling policy. When this happens, the task with smaller utilization is removed from the group and the group partitioning is performed again. This step is repeated until the group is partitioned or when partitioning fails. Thus, we keep tasks that share memory partitions in the same processor whenever possible, minimizing the contention for shared cache lines, while maintaing HRT deadlines, as will be shown in Chapter 8.

- The second phase is performed at run-time. HPCs are used to collect information of cache coherence activities initiated by the cores. This information is then used by the scheduler to dynamically prevent the execution of BE tasks that share colors with SRT tasks. The objective is to reduce the contention for shared cache lines at run-time and to improve deadline tardiness of SRT tasks.

- We design and implement the dynamic phase of our scheduler in

EPOS. We show how a component-based RTOS allows a straightforward integration of HPCs with scheduling algorithms.

- We compare the dynamic scheduler with CAP-GS without dynamic optimizations and best fit decreasing bin packing heuristic in terms of deadline misses and tardiness of several generated task sets using the Partitioned-RM (P-RM) scheduler. Our results indicate that it is possible to improve deadline tardiness and to provide HRT guarantees by combining a color-aware task partitioning with dynamic scheduling optimizations.
- We evaluate the dynamic scheduler in a modern eight-core processor, with shared L3 cache using EPOS. The experimental evaluation on a real machine and RTOS demonstrates the effectiveness of our dynamic scheduler.

## 1.3 DOCUMENT ORGANIZATION

The remainder of this thesis is organized as follows. **Chapter 2** describes the background concepts needed to follow the rest of the document. The chapter presents the required hardware background, relevant operating system concepts, and related real-time issues, such as real-time scheduling algorithms and run-time overhead accounting techniques.

**Chapter 3** discusses the state-of-the-art in closely related areas. The state-of-the-art is divided in four main topics: memory management, real-time operating systems, run-time performance monitoring, and multicore real-time scheduling. For each topic, we present the main relevant works that have been proposed recently. The chapter ends with a summary of the state-of-the-art of multicore scheduling algorithms.

**Chapter 4** presents the design and implementation of the real-time support on EPOS. The chapter starts discussing hardware mediators, a method to construct device drivers efficiently. Then, it shows the design of a performance monitoring API based on the hardware mediator concept. The design of real-time scheduling on EPOS is subsequently approached, presenting the main changes carried out by this work to the original EPOS scheduling support. Afterwards, considerations about the implemented memory partitioning mechanism are stated, emphasizing its design.

**Chapter 5** shows the run-time overhead evaluation. We measure the run-time overhead of EPOS and LITMUS$^{RT}$ and integrate both overhead into the schedulability analysis of G-EDF, P-EDF, and C-EDF. Thus, we compare the impact of the run-time overhead in EPOS and in LITMUS$^{RT}$ on the schedulability of three multicore real-time schedulers.

**Chapter 6** presents the cache partitioning evaluation. From the cache partitioning mechanism implemented and described in Chapter 4, we evaluate the cache partitioning impact on three multicore real-time schedulers: G-EDF, P-EDF, and C-EDF. We also evaluate the impact of the RTOS on real-time tasks by assigning individual memory partitions to the OS. As consequence, we show that a lightweight RTOS does not have a considerable impact on real-time tasks.

**Chapter 7** describes a Color-Aware task Partitioning (CAP) algorithm that assigns tasks to cores considering the usage pattern of memory partitions. The chapter also presents an evaluation comparing the CAP approach with a traditional task partitioning algorithm.

**Chapter 8** describes a two-phase dynamic multicore real-time scheduler. The first phase is an extension of the CAP algorithm and it is performed offline. The second phase consists of collecting run-time information through the use of HPCs and to prevent BE tasks to interfere with real-time tasks in the shared cache.

**Chapter 9** concludes the thesis. It reasons about the main contributions of the thesis and identifies its limitations. Finally, the perspectives for future works are also considered.

# 2 BACKGROUND

This chapter briefly summarizes the concepts that form the basis for this thesis. It starts presenting the required hardware background, following with a discussion of relevant operating system and real-time system topics, including implementation techniques and scheduling algorithms.

## 2.1 PROCESSOR AND MEMORY ARCHITECTURES

We begin with a discussion of multicore processors and memory hierarchy fundamentals. Computer architecture is a wide topic and a comprehensive review of current multicore processors and memory hierarchy technologies is beyond the scope of this thesis. Instead, we focus on describing the topics that affect the design and implementation of an RTOS, such as memory hierarchy in multicore processors (Section 2.1.1), cache organization and cache line replacement policies (Section 2.1.2), address translation (Section 2.1.3), cache coherence protocols (Section 2.1.4), hardware performance counters (Section 2.1.5), interrupt handling (Section 2.1.6), and timer management (Section 2.1.7).

### 2.1.1 Current Multicore Memory Architectures

Multicore processors can be classified in three categories (DAVIS; BURNS, 2011):

- **Heterogenous**

  The processors or cores are different, for instance, with different frequencies and instruction sets. Thus, the rate of execution of a task depends on both the processor and the task.

- **Homogeneous**

  All processors or cores are identical or symmetric (Symmetric Multi-Processor – SMP). Hence, the rate of execution of a task is the same for all cores.

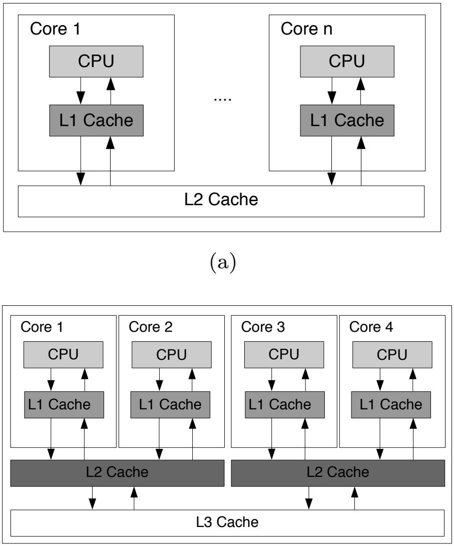
- **Uniform**

> The rate of execution of a task depends only on the processor speed. For example, a processor with speed two executes a task twice faster than a processor with speed one.

In this work, we consider homogeneous or SMP processors. In SMP systems, each processor fetches its own instructions and operates on its own data, characterizing a Multiple Instruction Multiple Data (MIMD) system. They feature a unique shared Uniform Memory Access (UMA) or a cache-coherent Non-Uniform Memory Access (cc-NUMA), easing data sharing. Two memory organizations widely used in today's SMP processors are presented in Figure 1. In Figure 1(a), "n" cores have a private level 1 cache and share a larger level 2 cache and the main memory. An example of processor that uses this architecture is the Intel Core 2 Quad Q9550. Figure 1(b) shows another memory hierarchy with two physical dies, each one composed of two cores. Each core has a private L1 cache, each die shares a L2 cache, and a L3 cache and main memory are shared by all cores. Intel Xeon L7455 with 24 cores and 6 dies is an example of this architecture. Note that there are variations from these two architectures (*e.g., Symmetric Multithreading - SMT*, L2 shared cache for all dies, etc), but the same principle of sharing the higher levels of memory still applies.

When a core accesses data and finds it in the cache, it has a *cache hit*. When a core does not find the requested data in cache, a *cache miss* occurs. Due to *temporal locality* which tell us that we might need the requested data again in the near future, whenever a cache miss occurs, the requested data must be brought back from the main memory. However, since caches are organized in lines, the processor brings back a complete cache line[1], instead of bringing back only a word. Because of *spatial locality*, there is a high chance of another word in the line be requested soon. In a cache miss, the program is stopped until the requested data is brought from the main memory, incurring in an execution time delay (HENNESSY; PATTERSON, 2006). When a cache line is brought back from the main memory, it replaces one that is in the cache. A cache line replacement algorithm chooses which line must be replaced. The next subsection presents an overview of the main cache line replacement algorithms and cache organizations, which have direct impact on the performance of multicore platforms.

[1] The size of cache lines in current processors can vary from 64 to 128 bytes.



(a)

(b)

**Figure 1: (a) An architecture with shared Level-2 cache. (b) An architecture with a cluster sharing Level-2 and all cores sharing the Level-3.**

### 2.1.2 Cache Organization and Cache Line Replacement Algorithms

Ideally, all data should be available on the caches, thus improving the overall program execution time. As caches have a limited size, the problem is to keep in the cache only the most important data for a certain period of time. A cache line replacement algorithm is responsible for choosing which cache line is replaced when a cache miss occurs. The cache line replacement policy depends on the cache organization (*i.e.,* cache mapping). Before describing the main cache line replacement algorithms, we start presenting three used cache organizations (HENNESSY; PATTERSON, 2006):

- **Direct-mapped cache**

  In a direct-mapped cache, an address in the main memory can only be placed in one cache line. In this cache organization there is no cache line replacement policy, since there is no choice of which cache line should be evicted in a cache miss (there is only one possibility). Figure 2 shows an example of a direct-mapped cache. A main memory block can be placed in only one of the four cache lines. The mapping from a memory address to a cache line is performed by the following equation: cache line = (memory block address) modulo (number of cache lines).

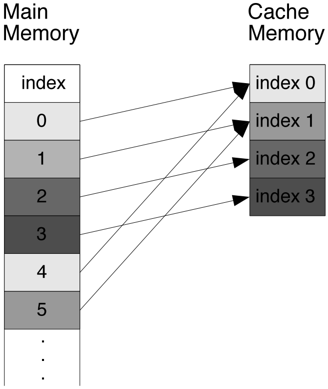


**Figure 2: Example of a direct-mapped cache. A main memory block can be placed in only one of the four cache lines.**

Direct mapping is simple but inefficient. If the cache size is much smaller than the main memory, many memory addresses will map to the same cache lines, thereby leading to conflicts.

- **Fully-associative cache**

  Figure 3 shows an example of a fully-associative cache. Each main memory block is placed anywhere in the cache. Thus, this technique implies in parallel search for cache lines, thereby requiring complex hardware control logic and presenting larger access times.

  Figure 4 shows the format of a memory address for a fully-associative cache. The tag field is a unique cache line identifier and the block offset field is the address within a cache line. A cache line search is performed by comparing the tag field of the line with the tag fields in all cache locations. The tag field has also other information about the cache line, such as validity bit

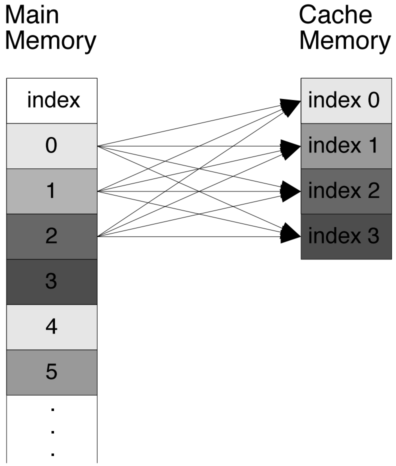


**Figure 3: Example of a fully-associative cache. A main memory block are placed in any of the four cache lines.**

(the line is valid or not), dirty bit (the line has been modified), and coherence protocol bits (bits to keep the coherence among caches, see Section 2.1.4).

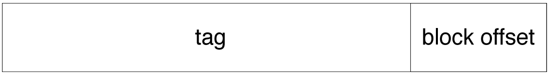


**Figure 4: The format of an address in a fully-associative cache organization.**

- **N-way set-associative cache**

  A set-associative cache combines characteristics from both, direct-mapped and fully-associative schemes. It is considered a compromise between the complex hardware needed by fully-associative caches and the simplistic mapping of direct-mapped caches. In a set-associative cache, there is a fixed number of locations where a cache line can be placed. An n-way set-associative cache has “n” locations for a cache line. Also, the cache consists of a number of sets, each of which consists of “n” cache lines. A main memory block maps to a unique set in the cache and the cache line can be placed in any element of that set. Figure 5 shows an example of a 2-way set-associative cache.

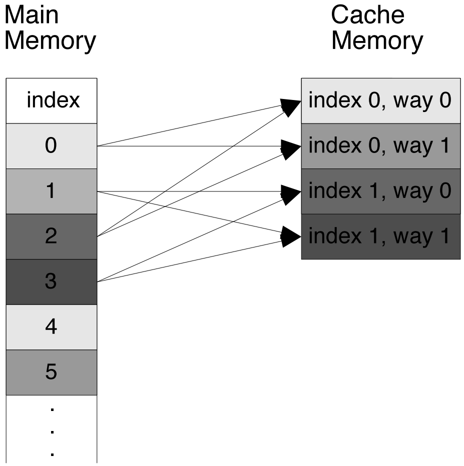


**Figure 5: An example of a 2-way set-associative cache. A main memory block can be placed in one of the two cache locations.**

The set selection is similar to the direct-mapped address translation: set = (memory address) modulo (number of sets). Since the cache line can be placed in any element of the set, there is the need to search all elements within a set. Figure 6 shows the format of a memory address in a n-way set associative cache. The block offset field is the address of the requested data within the cache line, the index field selects the set containing the address, and the tag field represents the cache line address.

| | | |
|---|---|---|
| tag | index | block offset |

**Figure 6: The form of an address in a n-way set-associative and direct-mapped cache organizations.**

Current memory organizations use n-way set associative caches. For example, the Intel i7-2600 processor has an 8-way set-associative Level-1 and Level-2 caches and a 16-way set-associative Level-3 cache. When the set of cache lines is full, the cache line replacement policy must decide which line will be replaced for the new data that is being brought from main memory. This algorithm is implemented in hardware, since its execution time is critical for the overall system performance. Furthermore, the goal is to evict the line as optimally as possible. In this context, cache line replacement algorithms are compared to the theoretical optimal algorithm. The optimal algorithm replaces the lines that will not be used for the longest period of time (TANENBAUM, 2007). Below, we discuss some of the main cache line replacement policies:

- **Random replacement**

  This policy evicts a cache line randomly. A random replacement policy is easy to implement, but its performance is bad, because the choice of an eviction is not based on the cache lines usage. For data caches, random performs on average about 22% worse than the Least Recently Used (LRU) policy (see below) (AL-ZOUBI *et al.*, 2004).

- **Least Recently Used (LRU)**

  The Least Recently Used (LRU) policy uses the principle of locality and is an approximation of the optimal algorithm. The objective is to evict the least recently used line in the hope it will not be used soon. Consequently, the policy must keep track of all accesses to cache lines and replace the block that has been the least recently accessed. For large caches with a great number of ways, the use of LRU is expensive and its implementation consumes time and power (AL-ZOUBI *et al.*, 2004).

- **First In First Out (FIFO)**

  The First In First Out (FIFO) heuristic evicts the cache lines in a sequential order, replacing the oldest cache line. This policy also takes advantage of the principle of locality, but is simpler than the LRU. However, for data caches this policy performs on average about 20% worse than LRU (AL-ZOUBI *et al.*, 2004).

- **Pseudo-LRU (PLRU)**

  The LRU policy has good performance, but requires a complex hardware to keep track of block accesses. The Pseudo-LRU (PLRU) policy is an approximation of LRU policy which uses less computational resources. Due to this approximation, the least recently accessed cache line is not always the evicted cache line (AL-ZOUBI *et al.*, 2004). There are several different ways to implement

PLRU. Two examples are the Most Recently Used (MRU)-based Pseudo-LRU (PLRUm) and the Tree-based Pseudo-LRU (PLRUt). See (AL-ZOUBI *et al.*, 2004) for an explanation of these two policies. PLRUm ranges from 1% worse than to 3% better than LRU and PLRUt is 1-5% worse than LRU (AL-ZOUBI *et al.*, 2004).

Current processors use one of the described policies. For example, the TriCore 1798, several PowerPC variants (MPC603E, MPC755, MPC7448) (GRUND; REINEKE, 2010), and Intel Pentium II-IV (REINEKE *et al.*, 2007) use PLRU. Intel XScale, ARM9, and ARM11 use FIFO (REINEKE *et al.*, 2007). Intel Pentium I and MIPS 24K/34K use LRU (REINEKE *et al.*, 2007).

### 2.1.3 Address Translation

Memory management is essential to control the memory usage by programs or applications. Virtual memory is a memory management technique to translate virtual memory addresses used by programs to physical addresses in the memory system. When a program access a memory address, it is actually accessing a logical address that is mapped into its corresponding physical memory location. Virtual memory was originally proposed by Fotheringham on the Atlas computer systems at the University of Manchester (FOTHERINGHAM, 1961).

The key idea behind virtual memory is to divide the address space of a program in blocks, called *pages.* Each page is a series of contiguous memory addresses, which are mapped into physical memory locations. The Memory Management Unit (MMU) translates logical addresses in a page to physical memory addresses dynamically as programs access their own pages. This mechanism is called **paging** and it is available on the most systems that support virtual memory (TANENBAUM, 2007).

**Paging in Intel processors.** An example of paging in a current processor is shown in Figure 7. This Figure depicts the translation of a linear address into the corresponding physical address in the Intel 32-bit paging mode. A linear address is the output of Intel's segmentation mechanism (Intel Corporation, 2011). The MMU implements a two-level paging system. These two levels are distinguished by naming the top level page table as "page directory". Page tables use regular memory frames and, in this mode, are 4 KB large. Each page can therefore hold 1024 32-bit entries.

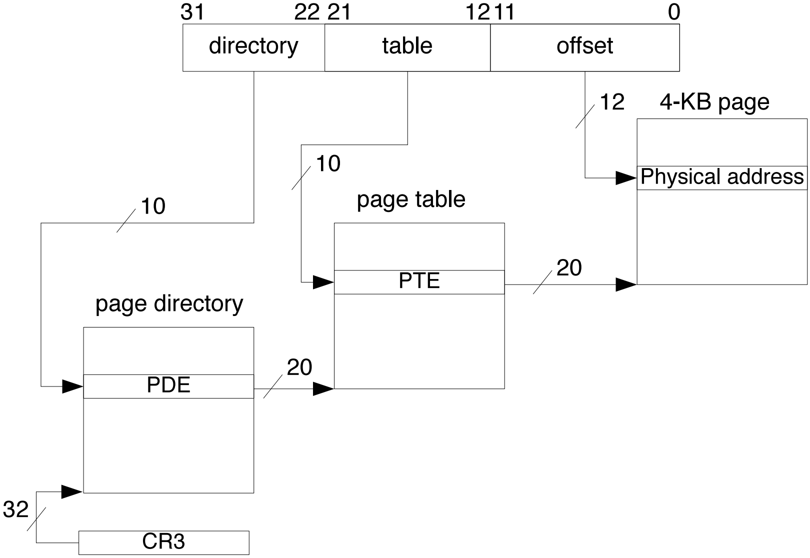


**Figure 7: Translation of a linear address into physical address in the Intel 32-bits paging mode (Intel Corporation, 2011).**

A 32-bit linear address is split in three parts: directory with ten bits, table with ten bits, and offset with twelve bits. Each process has its own page directory. The page directory location in memory is given by a special register (CR3). The directory field is used as an index in the page directory table to locate a pointer to the correct page table. The table field is then used as an index in the page table, retrieving the physical page address. Finally, the offset field is used to access the physical address within a page. The size of a page on Intel processors can be configurable as 4 KB or 4 MB. Depending on the page size, the linear address bits has different meanings. Figure 7 has shown paging with pages of 4 KB.

### 2.1.4 Cache Coherence Protocols

To demonstrate the problem of data consistency in multicore processors, consider the example in Figure 8 (HENNESSY; PATTERSON, 2006). In this example, three cores access a piece of data $u$ in the main memory. Cores 1 and 3 read the value of $u$ (events one and two in the Figure) and keep the current value in their private caches. After, core 3 writes the value 7 into $u$ (event three). However, cores 1 and 2 read the value of $u$ after the writing (events four and five), getting the old and wrong value of 5. Hence, the value of $u$ is inconsistent for cores 1 and 2. For this reason, a cache coherence mechanism able to keep data consistent after reads and writes from/to the same shared data is required.

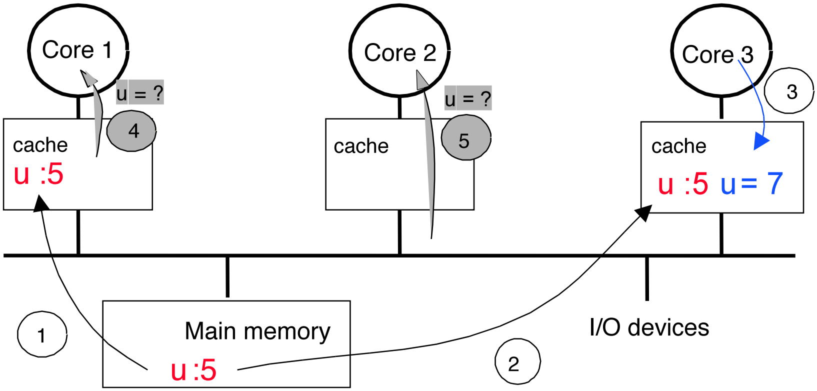


**Figure 8: Example of the data consistency problem in a multicore processor (HENNESSY; PATTERSON, 2006).**

In this context, a shared cache memory is placed between the private caches and the main memory. Each core has its own data and uses its private data cache for speeding up the processing. However, when cores share data, each copy of the data is placed in the core's private cache and a cache coherence protocol is responsible for keeping them consistent, usually by using a bus snooping mechanism or a directory-based mechanism. A memory system is coherent if it fulfills two properties (HENNESSY; PATTERSON, 2006):

**Write Propagation**

A write by one core is eventually visible to other cores.

**Write Serialization**

Every core sees two writes to the same memory location in the same order.

**Cache coherence mechanisms**. In a directory-based mechanism, shared data is placed in a common directory that keeps the consistency among all caches. The directory maintains a global view of the coherence state of each cache line. It tracks which caches hold each cache line and in what states. Whenever a core accesses a memory location, its cache controller sends a coherence request to the directory (*i.e.,* a unicast message), and the directory looks up the state of the cache line to determine what actions to take next (SORIN *et al.*, 2011). Directory-based protocols are used in NUMA processors. To exemplify the actions of a directory-based protocol, consider the following scenario. The cache controller of a core 1 sends a coherence request to the directory. Then, the directory might indicate that the requested cache line is owned by a core 2's cache and consequently the request should be forwarded to core 2 to obtain a copy of the cache line. When the core 2's cache controller receives the request, it unicasts a response to the requesting cache controller. Multiple coherence controllers may send coherence requests to the directory at the same time. However, the transaction order is determined by the order in which the requests are serialized at the directory (SORIN *et al.*, 2011). Directory-based protocols provide greater scalability, because it requires less bandwidth, at the cost of a level indirection, than bus snooping protocols (SORIN *et al.*, 2011). This is the reason why directory-based protocols are preferable for NUMA architectures, which usually feature a large number of processors. Our focus in this thesis, however, is on UMA or ccNUMA architectures, that implement a cache coherence protocol based on bus snooping. Also, bus snooping is the most common cache coherence mechanism used in current multicore processors.

In a bus snooping mechanism, each processor or cache controller monitors the bus in order to identify changes in cached data. Bus snooping tends to be faster than directory-based when there is enough bandwidth in the bus. However, the main drawback of the bus snooping mechanism is its scalability (HENNESSY; PATTERSON, 2006; SORIN *et al.*, 2011). In the next paragraphs, we discuss the main characteristics and protocols based on bus snooping.

When a core writes into a memory location that other cores have cached, the cache coherence protocol invalidates all copies, which may cause an implicit delay in the application's execution time. At the same way, when a core reads a memory position that was just written by another core, the cache coherence protocol does not return the data until it finds the cache that has the data, annotates that cache line to indicate that there is shared data, and recovers the data to the reading core. These operations are performed automatically by the hardware and take hundreds of cycles (about the same time as accessing the off-chip RAM), increasing the application's execution time (BOYD-WICKIZER *et al.*, 2010). Two kinds of scaling problem occur due to shared memory contention (BOYD-WICKIZER *et al.*, 2010): access serialization to the same cache line done by the cache coherence protocol, which prevents parallel speedup, and saturation of the inter-core interconnection, also preventing parallel processing gains.

The contention for shared memory causes a decrease in the application's throughput and deadline losses, in case of real-time systems (see Section 2.3). One can argue that there would be processing speedup by just turning off the cache and using the main memory directly. Nevertheless, it is a misconception that worst-case execution times with caches are equal to ones without caches, because most program's WCET are much better with caches than without, even if neither data nor instructions are accessed twice (LIEDTKE *et al.*, 1997). Caches increase memory bandwidth significantly even in no-hit situations, because an entire cache line is brought from cache, instead of only one word, which favors the spatial locality principle. Moreover, inter-thread interaction in multithreaded application is common. For example, some applications from NAS parallel and SPEC OMG benchmark suites have up to 20% of inter-thread interaction, and up to 60% of this interaction is affected by cache line contention (MURALIDHARA *et al.*, 2010). Reducing the effects of cache line contention can significantly improve the application's overall performance and prevent deadline misses.

A cache coherence protocol has distinct implementations depending on the cache write policy (*i.e.,* write-back or write-through). In a write-back cache, writes are not immediately forwarded to the main memory, instead the written locations are marked as dirty. Data in these locations are written back to the memory when they are evicted from the cache or by bus activity triggers. Thus, a read miss for a write-back cache line requires two memory accesses: one to write replaced data from cache to the store location and one to retrieve the needed data. In a write-through cache, in contrast, all writes result in updating the local cache and in a global bus write that updates the copy in main memory and invalidates/updates all other caches that have the same cache line. The advantage of a write-through is that it is simple to implement. The disadvantage, however, is that it consumes more bus bandwidth than a write-back cache. Thus, write-back schemes are the most used in current multicore processors. We discuss the main bus snooping protocols for write-back caches below.

The MESI protocol is the most common bus snooping-based cache coherence protocol supporting write-back caches (PAPAMARCOS; PATEL, 1984). In the MESI protocol, every cache line has one of the four states:

- **Modified (M)**: the cache line is present only in the current cache and its data is dirty. The value must be written to the memory before a reading. The write-back changes the state to exclusive.

- **Exclusive (E)**: the cache line is present only in the current cache and it is clean. The state can be changed to the Shared state when a read request from another core arrives or to the Modified state when a write operation is performed.

- **Shared (S)**: indicates that the cache line is shared by other caches and its state is clean.

- **Invalid (I)**: a cache line in this state does not hold a valid copy of data. The valid data can be in the main memory or in another processor's cache.

Table 1 shows the possible combination of cache line states in any pair of caches that implement the MESI protocol. For example, if a cache line in a processor is marked as shared (S), the same cache line in another processor is either in I or S states.

In a cache coherence mechanism based on bus snooping, there are two different agents that may change a cache line: the processor/core and the bus. Two operations are performed by processors (SONG, 2013):

- *p-load*: to read a value from memory system.

- *p-store*: to write a value to memory system.

**Table 1: The permitted states of a given cache line for any pair of caches in the MESI protocol (WIKIPEDIA, 2014b).**

| | M | E | S | I |
|---|---|---|---|---|
| M | X | X | X | V |
| E | X | X | X | V |
| S | X | X | V | V |
| I | V | V | V | V |

There are three possible operations for the shared bus (SONG, 2013):

- *bus-read*: to read a value from memory. For each *bus-read*, other caches should respond if it has the reading memory location (*i.e.,* cache line). Two forms of read responses are allowed: (i) *bus-read/s* indicating that the cache has the requested cache line and (ii) *bus-read/ns* indicating that the cache does not have the requested cache line.

- *bus-read-x/bus-write*: bus reading exclusive, which has two different meanings: (i) to inform other caches that some memory location will be written in a cache and (ii) to write a value to the cache.

- *flush*: to write a cache line to main memory.

A processor operation may or may not generate a bus operation, depending on the current cache line state. Figure 9 shows the MESI protocol state diagram correlating the four states with the processor and bus operations, which are responsible for triggering a state transition. Arrows in black denote processor operations, while arrows in grey denote bus operations. A processor operation is followed to an arrow to indicate when a processor operation generates a bus operation. Some observations of the MESI protocol can be made from the state diagram: (i) every state with *p-load* remains the same state except the I state; (ii) A *p-store* operation leads to M state; (iii) snooping a *bus-read-x* operation leads to I state; (iv) snooping a *bus-read* operation leads to S state; and (v) leaving M state triggers a flush operation (SONG, 2013).

The MOESI protocol adds a fifth state to the MESI protocol (AMD, 2010). The **Owned (O)** state represents data that is both modified and shared. This avoids the need to write modified data back

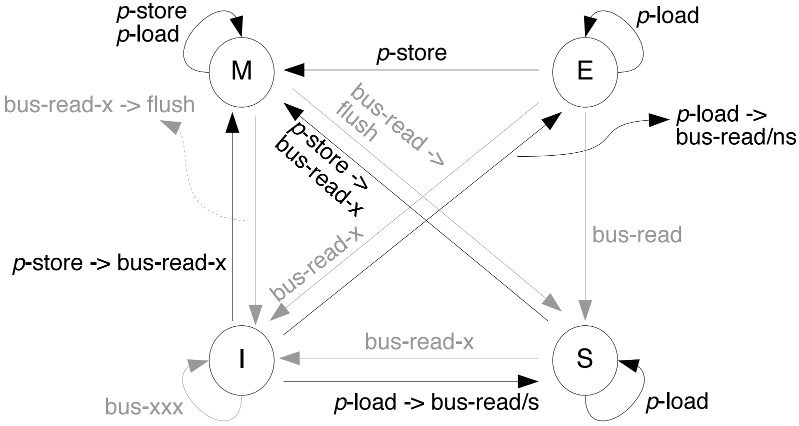


**Figure 9: MESI protocol state diagram (SONG, 2013).**

to the main memory before sharing it. Only one processor can hold a cache line in the O state – the same cache line in other processors must be in the S state. The processor which holds a cache line in the O state is the only one to respond to a snoop request. When a snoop request arrives, the cache line switches to the O state, and the duplicate copy is made into the S state. As a result, any cache line in the O state must be written back to memory before it can be evicted, and the S state no longer implies that the cache line is clean, unlike the MESI protocol. The MOESI protocol requires direct cache-to-cache transfers, so a cache with the data in the M state can supply that data to another reader without transferring it to memory (WIKIPEDIA, 2014d). Table 2 shows the possible combination of cache line states in any pair of caches that use the MOESI protocol. If a processor has a cache line in S state, the same cache line in another processor can be in I, S, or O states. Only one processor can have a cache line in O state, the other copies are either in S or I states.

Figure 10 shows the state transitions of the MOESI protocol. Some observations of the MOESI protocol behavior can be made from the state diagram: (i) snooping a *bus-read* operation in S state remains the same stats; (ii) every snooping of a *bus-write* operation leads to I state; (iii) snooping a *bus-read* operation in M state leads to O state; (iv) a *p-store* operation in O state leads to a coherence communication and a transition to M state; and (v) the *flush* operation is only triggered

**Table 2: The permitted states of a given cache line for any pair of caches in the MOESI protocol (WIKIPEDIA, 2014d).**

| | M | O | E | S | I |
|---|---|---|---|---|---|
| M | X | X | X | X | V |
| O | X | X | X | V | V |
| E | X | X | X | X | V |
| S | X | V | X | V | V |
| I | V | V | V | V | V |

when snooping a *bus-write* in M state.

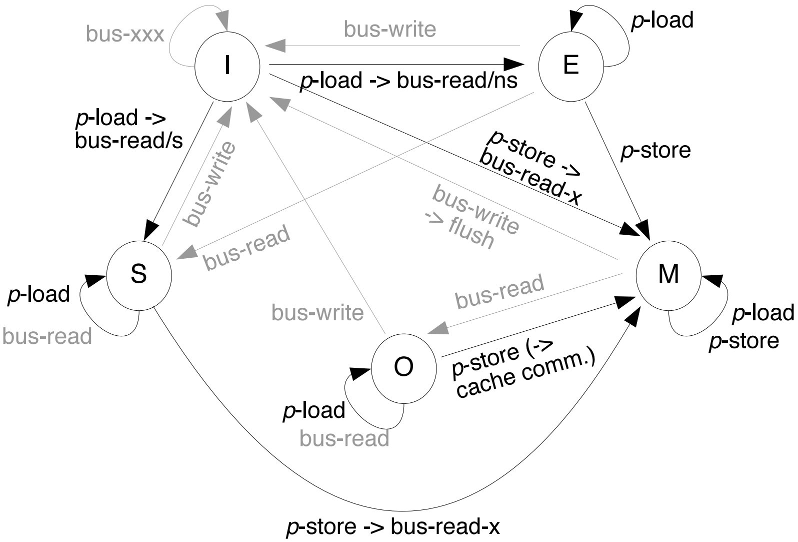


**Figure 10: MOESI protocol state diagram (SONG, 2013).**

The MESIF protocol is a cache coherence protocol developed by Intel for ccNUMA architectures (HUM; GOODMAN, 2005; Intel Corporation, 2009). The M, E, S, and I states are the same as in the MESI protocol. The new **Forward (F)** state is a specialization of S state and is used to indicate that a cache controller should act as the appropriate responder for any request for the given cache line. In a system of caches employing the MESI protocol, a cache line request is served by all caches that hold the cache line in S state. Thus, the requestor may receive multiple redundant responses, wasting bus bandwidth (WIKI-

PEDIA, 2014c). In the MESIF protocol, instead, a cache line request is served only by the cache holding the cache line in the F state, so coherence traffic is substantially reduced when multiple copies of the data exist. As in the MOESI, MESIF requires cache-to-cache communication capability. Because of that, cache coherence responses in MOESI and MESIF protocols are faster than in the MESI protocol (Intel Corporation, 2009).

Figure 11 demonstrates the reduced traffic messages of the MESIF protocol when compared to the MESI protocol. In Figure 11(a), core 3 sends a cache line request. Core 0 has the cache line in I state, core 1 has the cache line in S state, and core 2 has the cache line in F state. Thus, the response for the cache line request is only served by core 3, because it has the cache line in F state. Figure 11(b) exemplify the same case for the MESI protocol. Cores 1 and 2 have the cache line in S state and respond to the cache line request from core 3, which results in redundant responses. In both cases, core 0 does not respond, because it has an invalid copy of the cache line.

In the MESIF protocol, a cache controller may invalidate a line in the S or F states, thus it is possible that no cache has a copy in the F state. In this case, a cache line request is served, less efficiently, by the main memory. To minimize the chances of having no caches in F state, whenever a cache controller requests a cache line, the newly created copy is placed in the F state and the cache line previously in the F state is put in the S state or in the I state. Thus, there is only one copy in the F state and the remaining copies are in the S state. Table 3 shows the possible combination of cache line states for any pair of caches that use the MESIF protocol. If a cache line is in the F state, other caches are in either S or I states.

**Table 3: The permitted states of a given cache line for any pair of caches in the MESIF protocol (WIKIPEDIA, 2014c).**

| | M | E | S | I | F |
|---|---|---|---|---|---|
| M | X | X | X | V | X |
| E | X | X | X | V | X |
| S | X | X | V | V | V |
| I | V | V | V | V | V |
| F | X | X | V | V | X |

MESIF and MOESI protocols are usually used in ccNUMA ar-

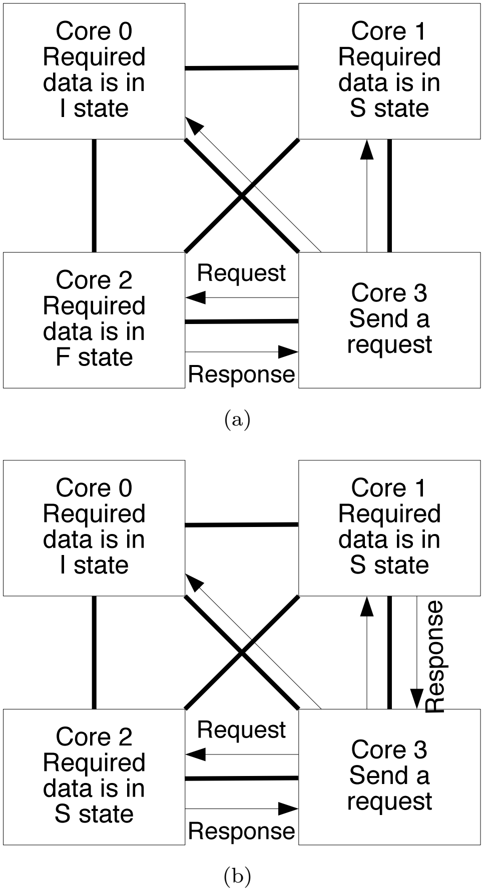


**Figure 11: Comparison of MESIF (a) and MESI (b) protocols. MESIF protocol reduces the bus traffic (QIAN; YAN, 2008).**

chitectures, such as Intel Sandy Bridge and AMD Opteron microarchitectures respectively, while MESI is widely used in UMA architectures, such as Intel dual-core and ARM Cortex MPCore (ARM, 2010).

**True and false sharing.** If two or more cores operate on independent data mapped into the same cache line, the cache coherence protocol invalidates the whole line at every data write, forcing memory stalls and wasting memory bandwidth. This phenomenon is called **false sharing**. False sharing is a well-known performance issue in SMP systems. **True sharing** occurs when two or more cores operates on the same data, which obviously is mapped into the same cache line. The concepts of true and false sharing are used in the rest of this thesis.

### 2.1.5 Hardware Performance Counters

Hardware Performance Counters (HPC) are special registers available in most of the modern microprocessors through a hardware Performance Monitoring Unit (PMU). HPCs offer support for counting or sampling several micro-architectural events in real time, such as cache misses and retired instructions (SPRUNT, 2002).

However, the limited number of hardware registers available on current processors may be a problem, mainly due to errors caused by HPC register multiplexing (AZIMI *et al.*, 2005). Intel processors feature over 200 monitorable events, and multicore ARM processors are reaching the count of 100. One can think that the more, the best, but due to the increase of circuit power and complexity, event signals need to be routed to the PMUs, making it impossible to monitor all events simultaneously. The number of HPCs in Intel and ARM processors are model-dependent but are, at most, 8 and 6, respectively. Besides the reduced quantity, the use of HPCs is further restricted by hardware design when sets of events are mappable only to one HPC or some HPC is fixed to a unique event.

Multiplexing techniques overcome the limitation in the number of HPCs at the cost of accuracy (MAY, 2001; SPRUNT, 2002). For instance, two recent published works correlate power measurements to HPC values empirically. Bircher and John propose abstract submodels for CPU and peripherals (BIRCHER; JOHN, 2012), while Bertran et al. provide detailed submodels for CPU components such as the prefetcher and the branch prediction unit (BERTRAN *et al.*, 2013). Nevertheless, what is significant about these models is the fact that they use, respectively, 7 and 11 distinct events. The cited papers show that the models approximate the power of their systems with acceptable errors. However, the experimentation scenario used in their evaluation methodology is, in fact, very controlled and, perhaps, unrealistic. In more realistic scenarios, several events would be monitored to implement not only power models, but also schedulers (AZIMI *et al.*, 2009; SINGH *et al.*, 2009) and memory partitioning (TAM *et al.*, 2007) mechanisms, increasing the multiplexing degree.

Another technique used to overcome the limitation in the number of HPCs and their complex interface (*i.e.*, low-level and difficult to program) is the use of specific libraries (DONGARRA *et al.*, 2003; GRACIOLI; FRÖHLICH, 2011). These libraries abstract the access to HPCs, creating a simpler interface to configure and read HPCs, without adding expressive overhead to the application. Section 3.3 summarizes the main HPC APIs and libraries and the related works that use HPCs to perform dynamic optimizations.

### 2.1.6 Interrupts

An interrupt is a signal to the processor which is emitted by hardware or software (also called exception or trap) to indicate the occurrence of an event. The processor responds to an interrupt suspending the current execution flow and dispatching an Interrupt Service Routine (ISR). The delay caused by the interrupt increases the execution time of the current task and must be accounted for in order to determine the schedulability of a real-time system, as will be discussed in Section 2.5.

A hardware interrupt is a signal sent by a device, such as disks, network interface cards, and keyboards. For example, a disk controller informs the OS when a requested data is ready to be transferred to main memory by triggering a disk interrupt. The initiation of a hardware interrupt is called Interrupt Request (IRQ). The number of hardware interrupts in a processor depends on the number of IRQ lines. Usually, processors implement a circuit, named Interrupt Controller (IC), which connects the several device interrupts to one or a few CPU lines. Hardware interrupts are asynchronous and may occur between the execution of any instruction.

A software interrupt may be triggered by an exception in the processor or by a special processor instruction, usually called *trap*. For instance, if the CPU finds a division by zero, which is impossible, a *division-by-zero* exception is generated and the operation is canceled. Then, the appropriate ISR is called to handle the exception. Interrupt instructions are used to invoke low-level OS services, such as device drivers services and system calls. Each interrupt, being it a hardware or a software interrupt, has its own ISR.

Interrupts can be maskable or non-maskable. Maskable interrupts may be ignored by the OS by setting a bit in a special control register (*e.g.,* interrupt mask register). Non-maskable interrupts (NMI) cannot be ignored. An example of an NMI device is a watchdog timer that is used to detect and recover from system failures. The watchdog timer restarts the computer when its counter reaches zero.

According to Brandenburg (BRANDENBURG, 2011), interrupts are categorized into four classes: device interrupts (DI), timer interrupts (TI), cycle-stealing interrupts (CSI), and inter-processor interrupts (IPI). DIs are triggered by hardware devices, as discussed above. TIs are controlled by the OS to schedule actions that need timed reaction. For instance, TIs are used to put a task to sleep for a certain period of time and to trigger the OS scheduler periodically. CIs are neither controlled by hardware or software and are used to "steal" processing time for execution of some hardware component. CIs are usually non-maskable and the OS is unaware when they occur. They are implemented as a combination of hardware and software ("firmware"). As a concrete example, consider the system management mode (SMM) in the Intel architecture (Intel Corporation, 2011). When a system management interrupt (SMI) occurs, the processor executes a low-level ISR to handle the interrupt. SMIs are used to provide legacy hardware emulation, such as the emulation of a floppy disk drive. Finally, IPIs are specific to multiprocessor systems. IPIs are used to send messages among processors. These messages are usually related to OS synchronization states and consequently are initiated by the OS. For example, an IPI can inform a remote processor to execute a reschedule operation.

**Interrupts in Intel processors.** Figure 12 shows how interrupts are handled by Intel processors. The evaluation of the OS mechanisms proposed in this thesis uses an Intel processor (see Chapters 5, 6, 7, and 8). Each processor has a Local Advanced Programmable Interrupt Controller (APIC) to manage interrupts. Local APICs are responsible for sending and receiving IPIs to/from other processors, to deliver DIs sent by the I/O APIC, and to handle local processor interrupt sources, such as timer, performance counters, and thermal sensors. I/O APIC handles external device interrupts and the distribution of such interrupts for one of the processors. I/O APIC and Local APICs are connected by a system bus. Thus, each interrupt can be routed by a specific processor through the use of a logical destination ID, which is unique for each core.

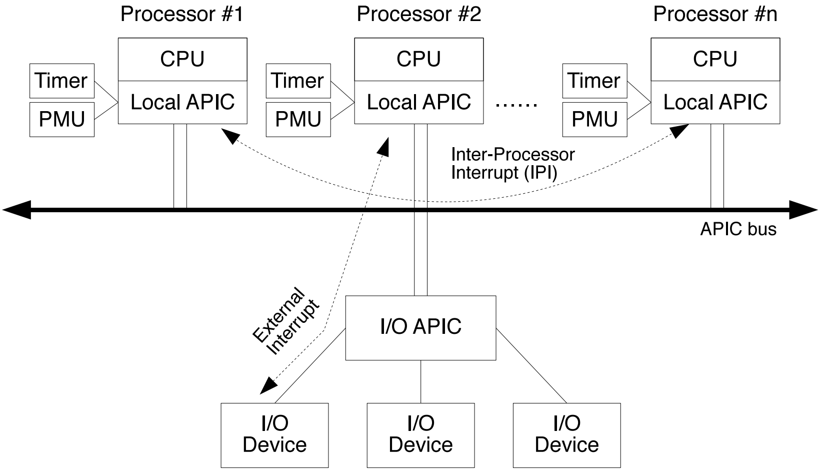


**Figure 12: Illustration of Local APIC and I/O APIC in the processor underlying the experiments in Chapters 5, 6, 7, and 8. The Local APICs handle per-processor interrupts, such as timer and PMU interrupts, and IPIs. External devices generate interrupts through the I/O APIC.**

The Local APIC implements a priority scheme for interrupts. There are in total 15 interrupt priorities. If a higher priority interrupt is received while a lower priority interrupt is executing, the higher priority interrupt is immediately delivered, preempting the lower priority interrupt. The OS can suspend local interrupt delivery and re-enable local interrupts by calling the *cli* and *sti* instructions, respectively.

### 2.1.7 Timers and Clocks

Time management is crucial for the correct functioning of an RTOS. In this section, we describe two commonly used time devices: timer and clock.

Clock is a device that keeps track of the current time. It basically measures the progress of physical time. Clocks count the occurrence of a periodic event, such as the oscillation of a clock crystal. The clock "ticks" at a given frequency and counts the number of these ticks, since the processor startup (BRANDENBURG, 2011). Clocks are used to determine the execution time of an operation (a task execution for instance) by measuring the difference in terms of clock ticks before and after the completion of the operation.

Timer is a configurable device that generates interrupts at every time interval. Computer platforms have at least one hardware timer. A timer can either increment or decrement a counter in a fixed and configurable frequency. When the counter reaches a pre-configured value (stored in a comparison register) or zero, the timer fires an interrupt to indicate the elapsed time. Thus, a timer can also work as a clock, counting the number of ticks. Valid hardware periods are given by Equation 1:

$$\frac{D}{C} \leq P \leq \frac{D \times (2^r - 1)}{C} \tag{1}$$

where $D$ is the clock prescaler (a frequency relative to the processor clock), $C$ is the system's clock frequency, $P$ is the timer's period, and $r$ is the timer's resolution. Thus, when the desired event interval is larger than the timer's hardware resolution, the OS must count ticks in software (FRÖHLICH *et al.*, 2011). Whenever the OS programs the timer, it should convert the desired time interval to timer ticks. Let $T$ be number of ticks, $I$ the desired interval, and $F$ the timer frequency, then $T = \frac{I}{F}$ denotes the conversion from time interval to ticks.

There are two possible timer operation modes: single-shot and periodic. The periodic approach generates an interrupt at the hardware timer period rate, while single-shot timers only generate one interrupt and remain inactive until reprogrammed. The single-shot approach can also fall back to tick counting when the desired time interval is greater than the maximum hardware period (see Equation 1). In general, single-shot tends to cause less interference on the system (FRÖHLICH *et al.*, 2011).

The accuracy of a timer or clock refers to how close the reported time is to the actual physical time (BRANDENBURG, 2011). The accuracy of a timer or clock is affected by the clock drift. Clock drift refers to several related phenomenons where a clock does not run at the same speed when compared to another clock. Clock drift may be a huge problem when two different platforms with two different hardware timers have to keep a synchronous communication. In this thesis, we assume that the clock drift is negligible across short time intervals.

**Timers and clocks in Intel processors.** Figure 13 shows a simplified version of the timers organization in the platform used in our experiments. Each core's local APIC has a timer that is driven by the same bus clock signal. The APIC timer has a counter that decrements on every $2^x$ bus clock cycles, where $x \in \{0, ..., 7\}$ (Intel Corporation, 2011; BRANDENBURG, 2011). The OS configures $x$ by writing into a prescaler register (clock divider register). Since the APIC timer is integrated into the processor chip, there is a very low programming overhead and propagation latency. The APIC timer supports both periodic and single-shot operation modes.

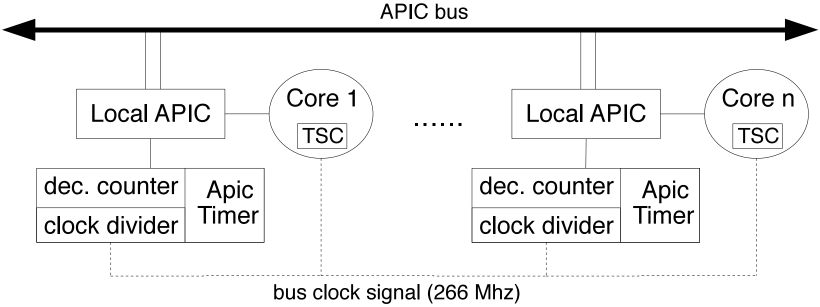


**Figure 13: Illustration of APIC Timers and TSCs in the processor underlying the evaluations in Chapters 5, 6, 7, and 8.**

Each processor also has a TimeStamp Counter (TSC), which is a 64-bit register (Intel Corporation, 2011). The TSC is incremented at each processor clock. Thus, TSC is the highest resolution clock in the platform. For example, considering a 2 GHz processor, the TSC's resolution is only 0.5 ns. Since the TSC is inside the processor, the overhead when accessing it is negligible. However, TSC can be affected by processor frequency scaling changes and TSC values may not be comparable across processors (BRANDENBURG, 2011). In our experiments, we used the TSC to count the execution time of tasks in a scenario in which frequency scaling was disabled and the RTOS uses the local APIC timer to deal with the notion of time.

## 2.2 OPERATING SYSTEMS

This section briefly discusses the main concepts involved in the design and implementation of Operating System (OS). OS fundamentals is an extensive topic. It is not our objective, however, to describe all topics and concepts in this section. For a complete review of all concepts and fundamentals, please refer to specific text books (TANENBAUM, 2007; SILBERSCHATZ *et al.*, 2008).

The objective of an OS is to manage hardware resources and to provide a simple, easy, and efficient interface to access theses resources. In other words, the OS extends and abstracts the usage of hardware devices. Examples of hardware resources are the processor, memory, and input and output (I/O) devices. The next subsections describe basic concepts regarding the management of the processor, memory, and I/O resources by the OS.

### 2.2.1 Process Management

A basic concept for every OS is the **process** (also referred as **task**). A task is basically an abstraction of a running program (TANENBAUM, 2007; SILBERSCHATZ *et al.*, 2008). For instance, a task can be responsible for browsing a website, while another task can print a set of files. A task contains the program code and data. In a multiprogrammed system, the CPU is switched from task to task. Each task executes for a certain period of time, usually a couple of microseconds, giving to users the notion of parallelism. This is known as **multitasking**. In a multitasking OS, tasks or processes share common processing resources, such as CPU and main memory.

In a uniprocessor system, only one task is said to be running at any point of time. This means that the processor is actively executing the instructions of that task. The OS process **scheduler** chooses which task should execute and for how long it can execute. The algorithm used by the scheduler is called **scheduling algorithm**. The main objectives of a scheduling algorithm are to keep the CPU busy at all times and to delivery acceptable response times for all programs, specially those that interact with users. The scheduler meets these objectives by using different scheduling strategies, depending on the application domain. For example, the round-robin scheduler gives each task a slot or quantum (its allowance of CPU time), and interrupts a task if it is not completed by the quantum interval. The task is resumed in the next time a quantum is assigned to it. Thus, round-robin shares the CPU with all tasks fairly. A priority scheduler, in contrast, always chooses the task with the highest priority to execute. Thus, a task that needs quick response time can have the highest priority in the system (TANENBAUM, 2007; SILBERSCHATZ *et al.*, 2008). Round-robin and priority scheduling strategies can be combined. In this case, tasks that have the same priority are scheduled in a round-robin fashion, sharing the CPU fairly. Scheduling is the basis to provide real-time guarantees. We review real-time scheduling strategies in Section 2.4.

Each task has its own context, which is composed by a set of processor registers, such as the program counter, general-purpose registers, floating-point registers, and I/O registers. When the CPU switches from a running task to another, the context of the old task must be saved and the context of the new task must be loaded. This activity is called **context switch** and it is the reason why the illusion of parallelism is achieved. The context of a task/process is kept in a process table in memory.

The **address space** of a task is the set of addresses that a task can use to access the memory. Each task has its own address space. Address spaces are managed by the OS and are used to create a protection layer among tasks (one task cannot access another's address space). The code and data of each task is placed on its address space. The memory available for data is divided in data section (memory space for global and static variables), stack (a space in memory reserved for local variables and function return values), and heap (space in memory reserved for dynamic memory allocation). The address space can be formed by a collection of logical pages that are mapped into physical memory addresses (see Section 2.1.3).

Although each task has its own program counter and set of registers (its internal context), tasks eventually need to communicate with each other. For example, a task can generate an output that serves as input to another task. In this case, the second task that is waiting for the input cannot execute until the first task generates the output. Consequently, the second task waits until its input is available. A task is waiting, because it cannot proceed. Also, a task can be ready to execute but the scheduler has decided to give the CPU to another task. Figure 14 illustrates the states of a task and the transitions among the states. A task can be in any of the three states:

- Running: a task is executing on the CPU.
- Ready: a task is ready to be executed but the CPU is busy executing another task or even OS routines.

- Waiting: the task is waiting for an event, probably waiting for an I/O operation to be finished.

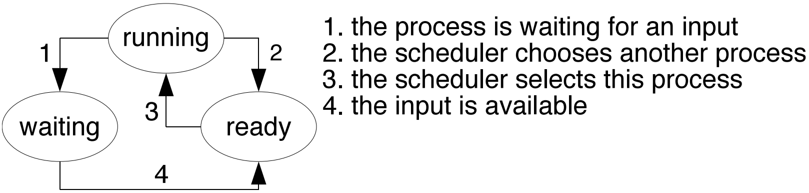
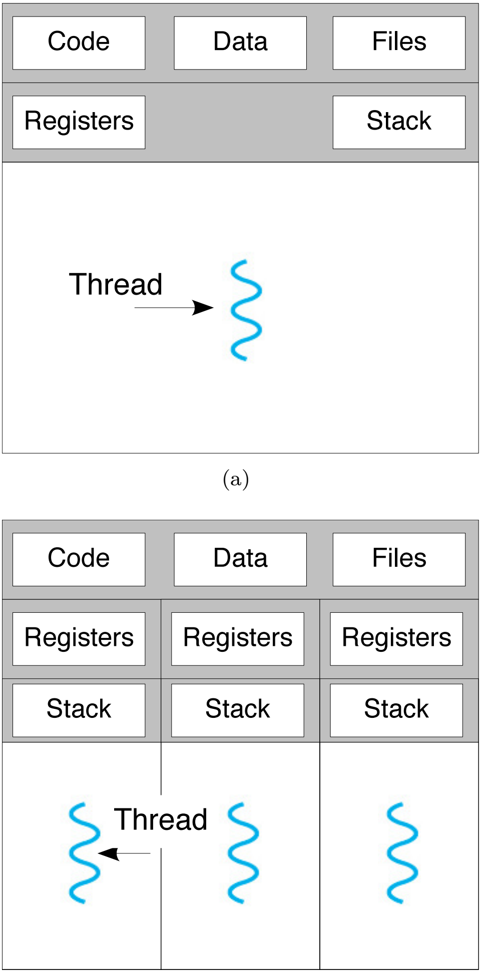

running
waiting
ready
1
2
3
4
1. the process is waiting for an input
2. the scheduler chooses another process
3. the scheduler selects this process
4. the input is available

**Figure 14: States of a task/process in an OS (TANENBAUM, 2007).**

There are four transitions among the three states. The transition one occurs when the task cannot proceed. The transition two occurs when the scheduler decides to give the CPU to another task. This action is called a **preemption**. The transition three is a scheduling decision performed by the scheduling algorithm, because there are more tasks ready than available processors. The fourth transition occurs when the input of a waiting task becomes available. The OS then changes the state of this waiting task to ready and the scheduler can choose it to execute.

In a traditional OS, each task has a single control flow and an address space (TANENBAUM, 2007; SILBERSCHATZ *et al.*, 2008). However, there are situations in which it is desirable to have more than one control flow within the same address space, executing "in parallel" as they were separated processes. A **thread** (also referred as **lightweight process**) is the definition of a control flow inside a task. It is a basic unit of CPU utilization, consisting of a program counter, a stack, a set of registers, current state, and a unique thread ID. Traditional tasks have a single thread of control, which means that there is one program counter and one sequence of instructions that can be executed. Multithreaded tasks, instead, have multiple threads within a single task, each having its own program counter, stack, and set of registers, but sharing common code, data (including the heap), and certain structures such as open files, as illustrated by Figure 15. Consequently, the context switch between two threads is faster than the context switch between two tasks, because the thread context is smaller than the process context.

In an OS that supports **multithreading**, each thread is treated as an individual execution entity, scheduled according to the scheduling



(a)

(b)

**Figure 15: (a) single-threaded and (b) multi-threaded processes (SILBERSCHATZ *et al.*, 2008).**

algorithm and its state (executing, ready, or waiting). Threads are very useful in modern programming models, because they provide developers with a useful abstraction of concurrent execution. The advantage of multithreading is clearer in a multiprocessor system, where several threads can truly execute in parallel, which makes the program to execute faster. Another advantage of multithreading, also applicable for uniprocessor systems, is the increased responsiveness to input requests. In a single-threaded program, for instance, if the main execution thread is blocked due to some reason, the entire application appears to be frozen. By assigning different operations to different threads, it is possible to avoid the aforementioned problem and to remain responsive to user input while executing long-running operations or waiting for an external event.

When more than one thread or task access a shared resource (*e.g.,* a variable or a hardware device), the access for a such resource must be protected to avoid **race condition**. The thread code that accesses a shared resource is called **critical region**. The OS must provide means to guarantee the mutual exclusion of threads/processes when accessing a critical region. Mutual exclusion is satisfied by the use of semaphores, mutexes, condition variables, or monitors (TANENBAUM, 2007; SILBERSCHATZ *et al.*, 2008).

The memory required by a task or a thread is defined as **working set**, and is formed by code and data (DENNING, 1968). The **Working Set Size** (WSS) of a task or thread is the size of its code and data (*i.e.,* the set of all memory pages accessed by it).

### 2.2.2 Memory Management

The part of an OS that manages the memory hierarchy of a computer is called **memory manager**. It is the job of the OS to abstract the memory hierarchy into a useful model and then manage the abstraction (TANENBAUM, 2007). The memory manager function is to efficiently manage memory: keep track of which parts of memory are in use, allocate memory to processes/threads when they request it, and deallocate it when they are done (TANENBAUM, 2007). Below, we provide a summary of the tasks of a traditional OS memory manager.

- **Management of free memory:** when dynamic memory allocation is allowed, the OS must manage which memory blocks are free and which are in use. The management of the available memory can be performed either by a bit map technique or by a linked list (TANENBAUM, 2007). Usually, these techniques are implemented in a memory allocator.

- **Address translation and memory protection:** we reviewed address translation in Section 2.1.3. We have seen that virtual memory is available on computers that feature an MMU and can have different page modes. In order to enable the translation of logical addresses to physical addresses, the OS must configure the MMU to work according to the desired paging mode. Also, the OS provides memory protection by controlling the page tables allocated to each process and their access rights. This isolates the address spaces of each process. The main objective of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug in a process from affecting another process or even the OS. When a process tries to access another's address space, the OS receives an exception (*i.e.,* bound error or segmentation fault in UNIX-like systems) from the MMU and then it can take an action, such as killing the process that caused the exception.

- **Memory allocation/deallocation:** the OS must provide means to allocate memory when threads/processes need and to deallocate when they do not need anymore or when they finish executing. Usually, memory allocation/deallocation is requested via system calls[2] through specific programming language functions. For example, the C language provides dynamic memory allocation/deallocation via a group of functions in the C standard library (malloc, realloc, calloc, and free) while dynamic memory allocation/deallocation in C++ is performed by the new and delete operators. Both malloc function and new operator may use system calls to request memory. For instance, two system calls responsible for managing memory in UNIX-like OSs are the the `sbrk` and `mmap`.

### 2.2.3 Input and Output Management

Besides managing processes and memory hierarchy, the OS also controls I/O devices. It aims at providing a portable and abstract interface for the communication between running programs and I/O

[2] A system call is an interface between processes/threads and the OS. The system call interface provides OS services to programs.

devices. The part of the OS that deals with I/O devices is usually called **Hardware Abstraction Layer** (HAL). HAL can be seen as a software layer between the physical hardware and programs and it is composed by a set of device drivers. Each device driver controls a particular type of hardware device. Processes/threads use system calls to communicate with hardware through the appropriate device driver.

Considering the software perspective, there are three different ways to perform I/O operations (TANENBAUM, 2007; SILBERSCHATZ *et al.*, 2008):

- **Programmed I/O (PIO):** PIO is the simplest form of performing I/O operations. PIO refers to data transfers initiated by the CPU, which executes a piece of software that controls the access to registers or memory on a device. The CPU issues a command and waits for I/O operations to be complete. The CPU, while waiting, must repeatedly check the status of the I/O device (also known as **polling** or **busy waiting**). Thus, there is a degradation of the system performance, because the CPU is just waiting for an I/O operation, instead of executing program's code.

  Figure 16 shows an example of a writing operation to an Universal Asynchronous Receiver/Transmitter (UART) device using PIO. Usually, an UART device has a status control register which informs whether the device is transmitting or not. This status control register is referred as `tx_reg_done` in Figure 16. The status control register is repeatedly check until it returns a non-zero value (line 2). When this happens, it means that the UART device is ready to transmit another character. Then, a character (*c*) is written to an output register (line 3).

```
1  void put(char c) {
2      while(tx_reg_done == 0) ;
3      //write the character c to the output register
4  }
```

**Figure 16: Writing a character to an UART device using PIO.**

- **Interrupt-driven I/O:** it is possible to use interrupts to overcome the busy waiting problem in the PIO. In an interrupt-driven I/O scheme, the CPU sends commands to the I/O device and then proceeds its normal execution until interrupted by the I/O device when it finishes its work. For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the processor. The actions to retrieve the data depend on whether the device uses independent I/O ports or memory-mapped I/O registers. For output, hardware devices issue interrupts to inform when they are ready to accept new data or to acknowledge a successful data transfer.

Interrupt-driven I/O is more efficient than PIO, because the CPU does not need to wait until the I/O operation is done. However, interrupt-driven I/O is still inefficient, mainly when transferring large amount of data, because the CPU must transfer data word by word between the I/O device and main memory. Thus, frequent interrupts must be handled, which waste CPU time. A solution is to use a different I/O scheme: Direct Memory Access (DMA).

- **Direct Memory Access (DMA):** the main idea of DMA is to allow hardware devices to access main memory independently of the CPU. Thus, the CPU is not responsible for handling interrupts or performing PIO as in interrupt-driven I/O and PIO mechanisms, respectively. This results in increased system performance. CPU is only involved at the beginning and end of the transfer and interrupted only after entire data has been transferred to/from the hardware device.

  DMA needs a special hardware called DMA controller that manages data transfers and arbitrates access to the system bus. DMA controllers are configured by the OS with source and destination buffer pointers (where to read/write data) and a count of the numbers of words to transfer. Then, the CPU sends commands to initiate transfer of data. The DMA controller increments a counter on every word transferred from the main memory to the device (or vice-versa) until the entire data is transferred.

  DMA allows a higher degree of concurrency when compared to the previous two I/O methods, because it leaves the CPU free to execute other processes. However, DMA may lead to cache coherence problems (see Section 2.1.4) and concurrent accesses to the shared bus (CPU and DMA controllers accessing main memory at the same time). These problems complicate the hardware design. For a complete review on the I/O management, please refer to specific text books (TANENBAUM, 2007; SILBERSCHATZ *et al.*, 2008). In the following we review the activities performed by the OS when handling an interrupt.

**Interrupt handling.** When a hardware device finishes its work, it sends an interrupt to the processor through a bus (see Section 2.1.6). This interrupt is handled by the OS. Figure 17 shows the activities performed by the OS when handling an interrupt. The processor stops the current instruction and jumps to a pre-defined OS function (step one in Figure 17). This function (dispatch) recovers the interrupt ID, informing which hardware device has generated the interrupt. By using the appropriate interrupt ID, the dispatcher accesses an interrupt vector table and calls the handler function for that interrupt (step three). Right next the conclusion of the interrupt handler execution, the processor returns to the next instruction (step four). Note that the interrupt handling is specific to the processor architecture and OS and may be different from the example depicted in Figure 17.

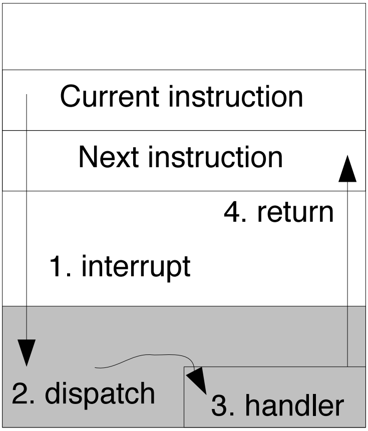


**Figure 17: The handling of an interrupt by the OS (TANENBAUM, 2007).**

## 2.3 REAL-TIME TASK MODELS AND CONSTRAINTS

A real-time system is a system that is subject to timing constraints. The behavior of the system depends not only on its logical correctness, but also on the time in which it is performed. In a real-time system, the unit of work that is scheduled and executed by the system is called a *job* and the set of related jobs which provides some system function is called a *real-time task* (LIU, 2000). Real-time tasks are recurrent, which allows the validation of the timing constraints during the system design phase through the use of a specific real-time task model.

Three task models are the most used in the real-time system literature: periodic (LIU; LAYLAND, 1973), sporadic (MOK, 1983), and aperiodic. Under the periodic task model, a task set $\tau$ is composed of $n$ tasks, $\{T_1, T_2, \ldots, T_n\}$. The $n$ tasks are scheduled on $m$ identical processors or cores $\{P_1, P_2, \ldots, P_m\}$. A task $T_i$ releases a job at every $p_i$ time units. $r_{i,j}$ denotes the release time of the $j^{th}$ job of $T_i$, named $J_{i,j}$. The release of a task can be performed by external events, such as device interrupts, or by expiring timers.

Mok introduced the sporadic task model as a generalization of the periodic task model (MOK, 1983). The period of a task ($p_i$) in the sporadic task model represents a lower bound on job separation, instead of an exact time interval between jobs as in the periodic task model. If one considers the minimum time interval between jobs as the task's period, the sporadic task model can be analyzed as the periodic task model.

In the aperiodic task model, tasks do not have a period or a minimum period interval, they can be released at any time. An aperiodic task has either soft deadlines (see Section 2.3.1 for an overview of soft and hard constraints) or no deadlines (LIU, 2000). In this work we consider the periodic task model, because it is well-studied, simple, and flexible to be implemented in any RTOS (LIU, 2000). In the following paragraphs, we present a complete definition of the periodic task model.

**Tasks.** Each task $T_i$ has three parameters ($e_i$, $p_i$, $d_i$). $e_i$ represents the worst-case execution time (WCET) of $T_i$, where $e_i > 0$; $p_i$ defines the period of $T_i$; and $d_i$ defines the relative deadline of $d_i$. We can interpret the parameters as follows: each task $T_i$ releases a job at $p_i$ time units, each job executes for at most $e_i$ time units and should finish no more than $d_i$ time units after its release. The parameter $e_i$ naturally depends on the hardware platform speed. In contrast, both $p_i$ and $d_i$ are machine-independent, that is, these parameters do not depend on the hardware platform speed (BRANDENBURG, 2011).

**Jobs.** A job $J_{i,j}$ is ready to be executed at its release time $r_{i,j}$, where $r_{i,j} > 0$. The rate of job releases respects the task period: $r_{i,j+1} \geq r_{i,j} + p_i$. Each job $J_{i,j}$ executes for at most $e_i$ time units and completes at time $f_{i,j}$ ($f_{i,j} \geq r_{i,j}$). $R_{i,j}$ defines the $J_{i,j}$'s response time: $R_{i,j} = f_{i,j} - r_{i,j}$. Considering all jobs, $T_i$'s maximum response time is $R_i = max_j\{r_{i,j}\}$.

**Deadlines.** The *relative deadline* $d_i$ of $T_i$ defines the range of acceptable response times. The *absolute deadline* $d_{i,j} = r_{i,j} + d_i$ indicates the time a job $J_{i,j}$ should complete its execution. A job is *tardy* if it completes later than the absolute deadline: $f_{i,j} > d_{i,j}$. If a job is tardy, it means that it has missed a deadline.

Tasks can be categorized by their relative deadlines:

- **Implicit deadlines:** a set of tasks $\tau$ has implicit deadlines if $d_i = p_i$ for every task $T_i \in \tau$.

- **Constrained deadlines:** a set of tasks $\tau$ has constrained deadlines if $d_i \leq p_i$ for each $T_i \in \tau$.

- **Arbitrary deadlines:** if a task set has neither implicit or constrained deadlines, it is said to have arbitrary deadlines.

The type of deadline category has large impact on the schedulability analysis, as will be demonstrated in Section 2.3.2. From the RTOS implementation point of view, implicit, constrained, and arbitrary deadlines are equivalent (BRANDENBURG, 2011).

**Utilization.** The relation $\frac{e_i}{p_i}$ defines the *utilization* of a task $T_i$, called $u_i$. $u_i$ is equal to the fraction of time a periodic task with period $p_i$ and WCET $e_i$ keeps a processor busy (LIU, 2000). Constrained-deadline tasks have a different rate constraint. Since $d_i < p_i$, if $J_{i,j}$ is to meet its deadline, the $J_{i,j}$'s rate of execution is defined by its *density* $\delta_i$, which is the task $T_i$'s utilization normalized by its relative deadline. If $d_i \geq p_i$, the $T_i$'s rate of execution is less constrained by its relative deadline than its period. Thus, density is formally define as $\delta_i = \frac{e_i}{\min(d_i, p_i)}$, which ensures that $\delta_i$ is always greater or equal than $u_i$.

**System utilization and density.** The concepts of task utilization and density are useful in the schedulability analysis and are applied to the whole task set. The total utilization and density of a task set $\tau$ are defined as:

$$u_{sum}(\tau) \triangleq \sum_{T_i \in \tau} u_i \qquad \text{and} \qquad \delta_{sum}(\tau) \triangleq \sum_{T_i \in \tau} \delta_i.$$

We also let

$$u_{max}(\tau) \triangleq \max_{T_i \in \tau}\{u_i\} \qquad \text{and} \qquad u_{min}(\tau) \triangleq \min_{T_i \in \tau}\{u_i\}$$

denote the maximum and the minimum utilizations of tasks in $\tau$, respectively. We also define $\delta_{max}(\tau)$ and $\delta_{min}(\tau)$ as the maximum and minimum density in $\tau$ and $e_{max}(\tau)$ and $e_{min}(\tau)$ as the maximum and minimum WCET in $\tau$, respectively.

Table 4 summarizes the notation and constraints for the periodic and sporadic task models defined in this section and used in the following chapters of this thesis.

**Table 4: Summary of notation and constraints in the periodic and sporadic task models (BRANDENBURG, 2011).**

| Notation | Description | Def/Constraint |
|---|---|---|
| $\tau$ | A task set. | $\tau = \{T_1, \dots, T_n\}$ |
| $T_i$ | The $i^{th}$ task. | $1 \leq i \leq n$ |
| $J_{i,j}$ | The $j^{th}$ job of a task $T_i$ | $j > 1$ |
| $e_i$ | Execution time of a task $T_i$ | $e_i > 0$ |
| $p_i$ | The period of $T_i$. | $p_i \geq e_i$ |
| | Minimum interval between jobs. | |
| $d_i$ | Relative deadline of a task $T_i$ | $d_i \geq e_i$ |
| $u_i$ | Utilization of a task $T_i$ | $u_i = e_i/p_i$ |
| $\delta_i$ | Density of $T_i$ | $\delta_i = e_i/min(d_i, p_i)$ |
| $r_{i,j}$ | $J'_{i,j}$ release time | $r_{i,j} \geq r_{i,j-1} + p_i$ |
| $d_{i,j}$ | $J'_{i,j}$ absolute deadline | $d_{i,j} = r_{i,j} + d_i$ |
| $f_{i,j}$ | $J'_{i,j}$ completion time | $f_{i,j} \geq r_{i,j}$ |
| $R_{i,j}$ | $J'_{i,j}$ response time | $R_{i,j} = f_{i,j} - r_{i,j}$ |
| $R_i$ | Maximum response time of $T_i$ | $R_i = max_j\{R_{i,j}\}$ |
| $m$ | Number of processors | |

### 2.3.1 Hard and Soft Timing Constraints

As stated before, a real-time system should respect timing constraints to have a correct behavior. More specifically, each job should complete by its absolute deadline. We consider two types of temporal correctness: soft real-time (SRT) deadlines and hard real-time (HRT) deadlines.

A HRT system is considered correct if and only if no job misses its absolute deadlines. This is equivalent to affirm that the $T_i$'s maximum response time is less than or equal to its deadline: $R_i \leq d_i$. The response time of a job depends on the RTOS scheduling algorithm, the use of shared resources, and the release sequence of higher priority jobs. Thus, we define a HRT system with respect to a given scheduling algorithm as follows (BRANDENBURG, 2011).

**Definition 2.3.1** *A task set $\tau$ is HRT schedulable under a scheduling algorithm A if and only if, for each $T_i \in \tau$, $T_i$'s maximum response time respects $R_i \leq d_i$ (when scheduled by A in a given hardware platform).*

The loss of a deadline in HRT systems is intolerable and may cause uncountable or catastrophic damage (*i.e.,* loss of human lives and money). System designers use schedulability tests (see Section 2.4) to prove that a given task set is HRT schedulable under a specific real-time scheduling algorithm.

However, for different types of real-time systems, the loss of "some" deadlines is tolerable. The consequence of a deadline miss is the degradation of the QoS provided by the real-time system. In this context, a task can miss a maximum number of deadlines. In other words, the tardiness of a job is bounded (SRINIVASAN *et al.*, 2003; LEONTYEV; ANDERSON, 2008; CALANDRINO; ANDERSON, 2008).

**Definition 2.3.2** *A task set $\tau$ is SRT schedulable under a scheduling algorithm A if and only if, there is a constant B such that $R_i \leq d_i + B$ for each $T_i \in \tau$ (when scheduled by A in a given hardware platform).*

Note that Definition 2.3.2 generalizes the HRT definition. If a system is HRT schedulable, the constant $B$ is zero. Moreover, the term SRT tends to be applied to any system that is not classified as HRT (BRANDENBURG, 2011) and has a vast literature, as the utility function definition below.

**Utility function.** As tasks in a SRT system have bounded-tardiness, we need to identify the magnitude of the tardiness of a job into the quality of the service. Jensen *et al.* propose the use of time utility functions to map the response time of a job to a utility value (JENSEN *et al.*, 1985). The utility value means the contribution of an event has to the system's objectives (BURNS, 1991). Figure 18 shows two examples of utility functions. Figure 18(a) presents the utility value for a HRT system. The computational event is zero before the *start time* and returns to zero after the *deadline*. The mapping of time to utility value is application-dependent and is constant in the Figure (BURNS, 1991). Figure 18(b) shows the utility function for a SRT system. The utility function decreases as the task's response time increases.

### 2.3.2 Schedulability, Feasibility, and Sustainability

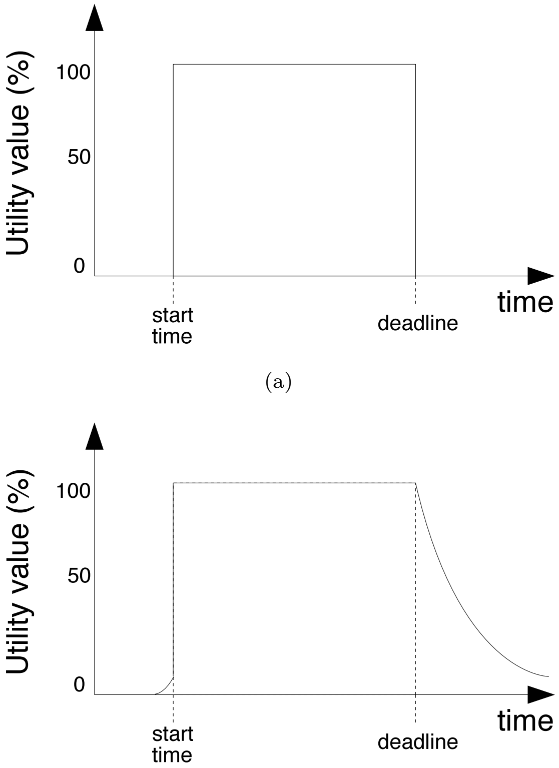
Before presenting real-time scheduling policies, we next define the main concepts related to temporal correctness, such as feasibility, schedulability tests, sustainability, and common assumptions to subsequent scheduling policies.

The criterion used to evaluate the performance of a HRT or SRT scheduling algorithm is its ability to find feasible schedules of a given task set whenever such schedules exists (LIU, 2000). Feasibility and optimality are two definition used in terms of schedulability of a task set.

**Definition 2.3.3** *A task set $\tau$ is SRT or HRT* feasible *if there exists a scheduling algorithm A such that $\tau$ is SRT or HRT schedulable under A.*

**Definition 2.3.4** *A scheduling algorithm A is SRT or HRT* optimal *if using A always produces a feasible schedule if the given task set $\tau$ has feasible schedules.*

Feasibility and optimality are considered according to some class of schedulers or platforms, such as "an optimal uniprocessor scheduler" or "feasible on a two-processor platform" (BRANDENBURG, 2011). A simple and straightforward feasibility test is the total utilization of a task set. The total utilization of a task set cannot exceed the total



(a)

(b)

**Figure 18: Illustration of HRT and SRT utility functions (BURNS, 1991). (a) HRT utility function. (b) SRT utility function. There is no value in a late HRT job, whereas for a late SRT job, the value decreases with increasing tardiness.**

number of processors ($u_{sum} \leq m$, where $m$ is the number of processors). If a task set demands more processor than the target platform provides, obviously it cannot be schedulable.

**Schedulability and sustainability.** The objective of a *schedulability test* for a scheduling algorithm $A$ is to validate that a given task set is HRT or SRT schedulable under $A$ during the design phase. The schedulability test must be *safe*, which means that it may not wrongly claim task sets schedulable, but it may be *pessimistic*, which means that it may state that a task set is not schedulable when the task set is in fact schedulable (BRANDENBURG, 2011).

A schedulability test can be classified into one of three different classes, as demonstrated by Figure 19. A *sufficient* test checks sufficient conditions for schedule. A sufficient test may state that a given task set is not schedule even though it is. A *necessary* test checks the necessary conditions for schedule. A necessary test is usually used to show no schedule exists. There may be cases in which no schedule exists and we cannot prove it (MARWEDEL, 2006). An *exact* schedulability test is both necessary and sufficient, which means that an exact test never returns a wrong answer. Exact tests are difficult to obtain and are NP-hard in many situations (MARWEDEL, 2006).

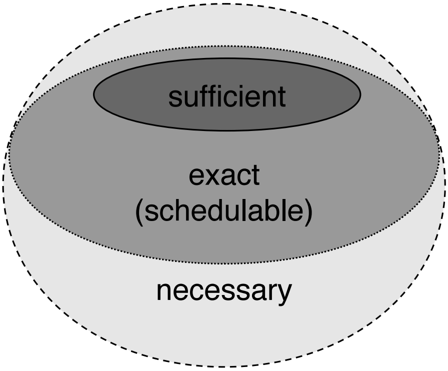


**Figure 19: Overview of the three schedulability test classes.**

Another concept related to schedulability is *sustainability* (BARUAH; BURNS, 2006):

**Definition 2.3.5** *A schedulability test for a scheduling algorithm $A$ is* sustainable *if any system deemed schedulable by the schedulability test of $A$ remains schedulable when the parameters of one or more job(s) change in any, some, or all of the following ways: (i) reduced execution time $e_i$; (ii) larger period $p_i$; and (iii) larger relative deadline $d_i$.*

The definition of sustainability requires that the schedulability of a task set under a scheduling algorithm $A$ be preserved in situations in which it should be "easier" to ensure schedulability, that is, the utilization or density of any task is temporarily reduced (BARUAH; BURNS, 2006). If we consider the run-time perspective, this definition is very important, because tasks usually do not always execute by its WCET $e_i$, for instance. A schedulability test is *self-sustainable* if every task in a task set $\tau$ that passed in the schedulability test still passes in the same schedulability test after a reduction in density or utilization of any task in $\tau$ (BAKER; BARUAH, 2009).

**Common assumptions.** The next section presents some scheduling algorithms and their schedulability tests. These scheduling algorithms use a number of common assumptions. To avoid repeating all these assumptions for each schedulability tests, we define the common assumptions below (BRANDENBURG, 2011).

- **Task are independent:** tasks do not share any kind of resources besides the processor(s).
- **Jobs do not self-suspend:** jobs are always ready to execute when allocated to a processor by the scheduler.
- **Jobs are preemptive:** at any time, the scheduler may replace the executing job by a higher priority job.
- **Jobs respects their periods:** jobs release are separated by their period $p_i$.
- **Run-time overhead is negligible:** the RTOS run-time overhead, such as the time to switch the context among jobs and scheduling decisions (see Section 2.5), is negligible.

These assumptions somehow simplify the scheduling problem. The run-time overhead, for instance, always exists in practice. As will be detailed in Section 2.5, there are techniques used to incorporate the run-time overhead into the schedulability analysis. We first describe the scheduling algorithms and their respective schedulability tests using these ideal assumptions in the next section.

## 2.4 REAL-TIME SCHEDULING

We review in this section the main uniprocessor and multiprocessor real-time scheduling algorithms. This review is important to provide a concise background for the presentation of our real-time implementation in EPOS in Chapter 4 and the proposed real-time schedulers in Chapters 7 and 8. We focus our discussion on presenting the algorithms, examples, and their schedulability tests, and omit proofs which can be found in the cited works.

A real-time scheduler can be either *static* or *dynamic*. Static schedulers, also known as off-line, table-driven, or cyclic executive schedulers, make use of a pre-computed schedule of all jobs (LIU, 2000). This schedule is computed off-line, during the system design phase, and is based on the knowledge of the release and execution times of all jobs for all times. Static schedulers are easy to implement and validate, as it provides a deterministic behavior, because the pre-computed schedule table. This makes static schedulers an ideal choice for safety-critical systems (BRANDENBURG, 2011). A disadvantage of off-line scheduler is inflexibility, mainly when release and execution times of tasks vary (LIU, 2000). Dynamic or on-line schedulers, in contrast, make each scheduling decision at run-time, without any knowledge about the jobs that will be released in the future. The parameters of each job become available when jobs are released. Then, the scheduler uses these parameters to perform a scheduling decision. On-line scheduling is the only option in a system whose the future workload is unpredictable (LIU, 2000). With an on-line scheduler, it is possible to deal with dynamic variations in resources availability. The main drawback, however, is the reduced possibility to achieve optimal scheduling decisions (LIU, 2000).

Scheduling algorithms can also be classified into two categories: *time-driven* and *priority-driven*. In time-driven (also called clock-driven or quantum-driven) schedulers, scheduling decisions are performed at specific time instants. These instants are chosen a priori before the system starts execution (LIU, 2000). The cyclic executive real-time scheduler is an example of a time-driven scheduler.

Priority-driven scheduling algorithms never leave the processor idle intentionally (LIU, 2000). Scheduling decisions are made when events, such as job releases and completions, occur. Thus, priority-driven scheduling can also be called as event-driven (LIU, 2000). Whenever a job is released, it is placed in one or more scheduling queues ordered by the priorities of the jobs. At a scheduling decision, the jobs with highest priorities are chosen to be scheduled on the available processors. If a higher priority job preempts a lower priority job, we say that the scheduler has performed a *reschedule* operation. In this thesis, we consider priority-driven dynamic schedulers.

A scheduling policy is responsible for assigning priorities to jobs. As defined by Brandenburg (BRANDENBURG, 2011), we let $Y(J_i,t)$ denote the priority of $J_i$ at time $t$. Priorities are unique and there exists a total order such that $Y(J_x,t) < Y(J_y,t)$ if and only if $J_x$ has a higher priority than $J_y$ at time $t$. Based on the prioritization function $Y$, the real-time literature defines three classes of priority-based schedulers (BRANDENBURG, 2011):

- **Fixed-priority (FP)**: in a *fixed-priority* (FP) scheduler, all jobs of a task have a constant and common priority. Priorities are assigned to tasks and all jobs are scheduled according to their tasks' priorities. Formally, $Y(J_{i,x},t_x) = Y(J_{i,y},t_y)$ for any two jobs $J_{i,x}$ and $J_{i,y}$ and any two times $t_x \in [r_{i,x}, f_{i,x}]$ and $t_y \in [r_{i,y}, f_{i,y}]$ (BRANDENBURG, 2011).
- **Job-Level Fixed-Priority (JLFP)**: in a *job-level fixed-priority* (JLFP) scheduler, fixed priorities are assigned to jobs, instead of tasks. Thus, jobs of a same task can have different priorities. Formally, $Y(J_{i,x},t_1) = Y(J_{i,y},t_2)$ for any two times $t_1$, $t_2 \in [r_{i,x}, f_{i,x}]$.
- **Job-Level Dynamic-Priority (JLDP)**: the most general class of priority-based schedulers is the *job-level dynamic-priority* (JLDP). In JLDP schedulers, a job's priority may change at any time.

The next subsection presents the main uniprocessor real-time scheduling algorithms for each priority class, showing examples and schedulability tests whenever possible to ease readability.

### 2.4.1 Uniprocessor Real-time Scheduling

The first work on uniprocessor real-time scheduling was proposed by Liu and Layland (LIU; LAYLAND, 1973). The authors have introduced optimal FP and JLFP scheduling policies, as well as their schedulability tests, for periodic task model.

#### 2.4.1.1 Fixed-Priority Scheduling

In FP scheduling, priorities have a decreasing order: the highest priority task has the lowest priority number. Moreover, as tasks priorities are fixed, the priority function can then simply defined as

$Y(J_i, t) = i$ (BRANDENBURG, 2011). Liu and Layland proposed the rate monotonic (RM) priority assignment policy (LIU; LAYLAND, 1973). RM assigns priorities to tasks according to their periods: the shorter the period, the higher the priority (LIU, 2000). The rate of job releases is inverse of its period. Thus, the higher a job's rate, the higher its priority.

Figure 20 shows an example of the RM scheduling with three tasks. $T_1$ has $e_1$ of 2 time units and $p_1$ of 10 time units. $T_2$ has $e_2$ of 5 time units and $p_2$ of 15 time units. $T_3$ has $e_3$ of 10 time units and $p_3$ of 25 time units. Tasks have implicit-deadlines ($d_i = p_i$). The total utilization is 0.93 and the system has one processor. Using the RM policy, $T_1$ is the higher priority task, because it has the shortest period. All tasks are released at the time 0. $T_1$ start executing until it finishes its execution time at the time 2. The same happens with $T_2$ from time 2 to 7 and with $T_3$ from time 7 to 10. At the time 10, a new $T_1$'s job arrives, which preempts $T_3$. $T_3$ is resumed at the time 12, after $T_1$ execution, and executes for more 3 time units. At time 15, a new $T_2$'s job arrives and preempts $T_3$. The same happens at the time 20. At the time 22, $T_3$ resumes and executes for more 3 time units. At this point, there is no enough time left for $T_3$ and it misses a deadline (see the arrow in Figure).

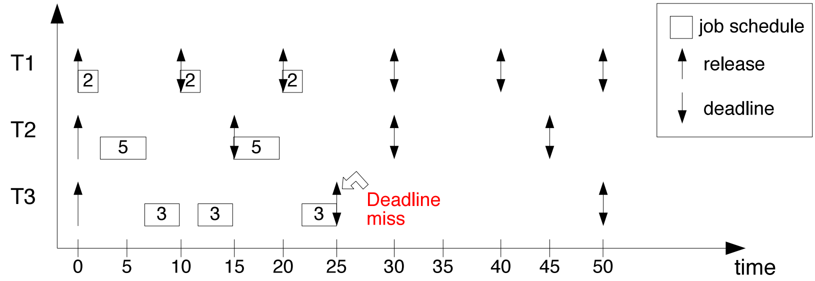


**Figure 20: An example of RM scheduling with two tasks.**

Liu and Layland proved that the RM scheduling is optimal with respect to FP schedulers, which means that if a task set is schedulable under some FP scheduler, this same task set will be also schedulable under RM scheduling (LIU; LAYLAND, 1973). The authors also proposed the following schedulability test.

**Theorem 2.4.1** *(LIU; LAYLAND, 1973). A task set $\tau$ composed of $n$*

*tasks with implicit deadlines is schedulable under RM scheduling policy if $u_{sum} \leq n(2^{1/n} - 1)$.*

Note that when $n$ tends to a large number ($n \to \infty$), the upper bound $n(2^{1/n} - 1)$ converges to ln 2 $\approx 0.69$. Thus, up to 30% of the processor capacity remains unused to ensure HRT schedulability under a FP scheduling. However, the RM schedulability bound improves when all pairs of periods in a task set are in harmonic relation. When this happens, the maximum utilization of the RM is 100% (BUTTAZZO, 2005).

Another well-known fixed-priority scheduling algorithm is the deadline monotonic (DM) algorithm (LEUNG; WHITEHEAD, 1982; LIU, 2000). DM assigns priorities to tasks according to their relative deadlines: the shorter the relative deadline, the higher the priority. The RM priority assignment policy is not optimal for non-implicit deadlines. Instead, Leung and Whitehead proved that DM priority assignment is optimal for constrained-deadline task sets (LEUNG; WHITEHEAD, 1982). DM can be seen as a generalization of the RM scheduling. A simple schedulability test for DM is obtained by replacing the $u_{sum}$ by $\delta_{sum}$ in Theorem 2.4.1 (BRANDENBURG, 2011).

The schedulability test presented by Theorem 2.4.1 is a sufficient test. An exact schedulability test for constrained-deadline task sets under FP scheduling was proposed by Joseph and Pandya (JOSEPH; PANDYA, 1986). The exact schedulability test calculates the maximum response time $R_i^{max}$ for each task $T_i$ explicitly. Giving the maximum response time, HRT schedulability is guaranteed by checking $R_i \leq R_i^{max} \leq d_i$. This method is also called Response Time Analysis (RTA) and is defined below.

**Theorem 2.4.2** *(JOSEPH; PANDYA, 1986). Given a task set $\tau$, with $n$ tasks ordered by decreasing priority. The task set is schedulable by a fixed-priority scheduling algorithm if and only if the maximum response time $R_i^{max}$ of each task $T_i$ is less than or equal to its relative deadline ($R_i^{max} \leq d_i$). Let $hp(i)$ be the set of tasks with higher priority than the task $T_i$. The $R_i^{max}$ is calculated iteratively using the following formula:*

$$\begin{cases} R_i^{max(0)} = e_i \\ R_i^{max(k)} = e_i + \displaystyle\sum_{j \in hp(i)} \left\lceil \frac{R_i^{max(k-1)}}{d_j} \right\rceil \times e_j \end{cases}$$

The maximum response time $R_i^{max}$ of a task $T_i$ is the smallest value of $R_i^{max(k)}$ such that $R_i^{max(k)} = R_i^{max(k-1)}$. For the RM scheduling policy, the RTA test is performed by replacing the term $d_j$ by $p_j$, since the deadline model is implicit.

**RTA example**. As an example of the RTA test, consider the task set with three tasks used in the early RM example (Figure 20). The higher priority task is $T_1$. Hence, $T_1$ maximum response time is equal to its WCET: $R_1^{max} = 2$. To calculate the maximum response time for the task $T_2$, we need the following iterations:

$$R_2^{max(0)} = e_2 = 5$$

$$R_2^{max(1)} = e_2 + \left\lceil \frac{R_2^{max(0)}}{p_1} \right\rceil \times e_1 = 5 + \lceil \tfrac{5}{10} \rceil \times 2 = 7$$

$$R_2^{max(2)} = 5 + \lceil \tfrac{7}{10} \rceil \times 2 = 7$$

The convergence is met in the third iteration, when $R_2^{max(2)} = R_2^{max(1)}$. For the task $T_3$, we need to perform the following iterations:

$$R_3^{max(0)} = e_3 = 10$$

$$R_3^{max(1)} = e_3 + \left\lceil \frac{R_3^{max(0)}}{p_1} \right\rceil \times e_1 + \left\lceil \frac{R_3^{max(0)}}{p_2} \right\rceil \times e_2 =$$

$$10 + \lceil \tfrac{10}{10} \rceil \times 2 + \lceil \tfrac{10}{15} \rceil \times 5 = 17$$

$$R_3^{max(2)} = 10 + \lceil \tfrac{17}{10} \rceil \times 2 + \lceil \tfrac{17}{15} \rceil \times 5 = 24$$

$$R_3^{max(3)} = 10 + \lceil \tfrac{24}{10} \rceil \times 2 + \lceil \tfrac{24}{15} \rceil \times 5 = 26$$

$$R_3^{max(4)} = 10 + \lceil \tfrac{26}{10} \rceil \times 2 + \lceil \tfrac{26}{15} \rceil \times 5 = 26$$

As the maximum response time of $T_3$ (26) is greater than its deadline (25), $T_3$ and consequently the task set is not schedulable. Baruah and Burns showed that RTA is sustainable and well-suited to determining schedulability of practical systems (BARUAH; BURNS, 2006). Two similar tests that may converge more quickly were later proposed by Lehoczky *et al.* (LEHOCZKY *et al.*, 1989) and Audsley *et al.* (AUDSLEY *et al.*, 1991).

#### 2.4.1.2 Job-Level Fixed-Priority Scheduling

The most important JLFP scheduling algorithm is the Earliest-Deadline First (EDF) (LIU; LAYLAND, 1973). In EDF, jobs are ordered by their absolute deadlines: the earliest the absolute deadline, the higher the priority. Formally, a job $J_{i,j}$'s priority can be denoted as $Y(J_{i,j}, t) = (d_{i,j}, i)$, where $(d_{i,j,i}) < (d_{x,y}, x) \Leftrightarrow d_{i,j} < d_{x,y} \vee (d_{i,j} = d_{x,y} \wedge i < x)$ (BRANDENBURG, 2011). Note that this definition considers several jobs with the same absolute deadline. In practice, multiple jobs can have their absolute deadlines at the same point in time. Thus, the definition prioritizes the job with the lowest index to tie-break the priority.

Figure 21 shows an example of the EDF scheduling with the same three tasks used in the previous examples. The total utilization of the task set is 0.93. In comparison with the RM scheduling in Figure 20, at the time 15, the job of the task $T_3$ is not preempted by the new releasing job of $T_2$, because $T_3$ has an earliest absolute deadline. Thus, $T_3$ is able to complete its execution before its deadline at the time 25, and no job misses any deadline.

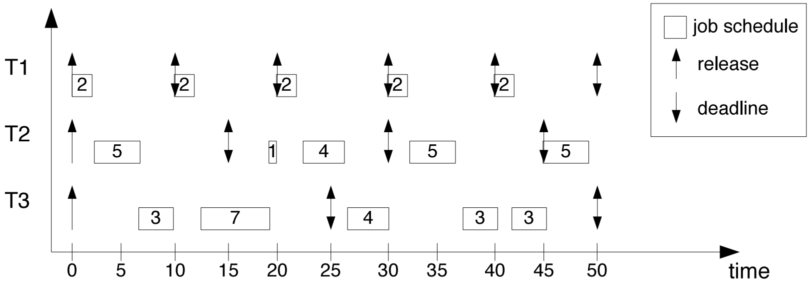


**Figure 21: An example of EDF scheduling with two tasks.**

Liu and Layland proved that EDF is an optimal scheduler for uniprocessor systems. Any feasible arbitrary-deadline task set is HRT schedulable under EDF (LIU; LAYLAND, 1973).

**Theorem 2.4.3** *(LIU; LAYLAND, 1973). A task set with periodic and implicit-deadline tasks is HRT schedulable under EDF on a uniprocessor system if and only if $u_{sum} \leq 1$.*

For task sets with arbitrary-deadlines, EDF HRT schedulability test becomes more complex. A simple, but inexact HRT test for arbitrary-deadlines is replace $u_{sum}$ by $\delta_{sum}$ in Theorem 2.4.3: $\delta_{sum} \leq 1$. This test is very pessimistic for constrained-deadline task sets (BRANDENBURG, 2011). For constrained-deadline task sets, other tests were proposed (BARUAH *et al.*, 1990; ALBERS; SLOMKA, 2005). However, these tests are pseudo-polynomial and expensive in terms of computation (BRANDENBURG, 2011). Other inexact tests which yield slightly more pessimistic results, but are more efficient in terms of computation, were also proposed (DEVI, 2003; MASRUR *et al.*, 2008).

#### 2.4.1.3 Job-Level Dynamic-Priority Scheduling

The most important JLDP scheduling algorithm is the Least-Laxity First (LLF). LLF assigns jobs priorities according to the *slack time* of each job. The smallest the slack time, the higher the priority. The slack time ($s_i$) of a job $J_{i,j}$ is the difference between the relative deadline ($d_i$), the current time ($t$), and the remaining computation time ($e_i$') (LIU, 2000):

$$s_i = (d_i - t) - e_i'$$

The LLF algorithm has two version: strict and non-strict. In the strict version, the slacks of jobs are continuously monitored by the scheduler and priorities are reassigned to jobs whenever their slacks relations change. Hence, the strict version has a heavy overhead due to the recalculation of slacks at any time by the scheduler. Because of that, the strict version is unattractive and not used in practice. In the non-strict version, the computation of the slacks is performed only at release and completion times of jobs. Thus, the non-strict version acts as a JLFP scheduling, changing the job's priority dynamically but not reassigning the job's priority later. The HRT schedulability test for implicit-deadline and periodic task sets in the non-strict LLF version is the same as in EDF: $u_{sum} \leq 1$.

## 2.4.2 Multiprocessor Real-time Scheduling

This section summarizes the main real-time scheduling algorithms for shared-memory multiprocessors. Basically, there are two alternatives to perform scheduling in a multiprocessor: (i) tasks are assigned to available processors using a single, shared ready queue; and (ii) the available processors are subdivided into smaller set of processors. The former approach is called *global scheduling* and the latter approach is called *partitioned* or *clustered scheduling.* We begin the discussion of multiprocessor real-time scheduling presenting the partitioned scheduling in Section 2.4.2.1. Then, we discuss global scheduling in Section 2.4.2.2 and clustered scheduling in Section 2.4.2.3.

### 2.4.2.1 Partitioned Scheduling

Dhall and Liu were the first authors to study partitioned scheduling in the context of a real-time system (DHALL; LIU, 1978). Partitioned scheduling is the most preferable scheduling approach for multiprocessor real-time systems due to that each partition (or processor) is scheduled and analyzed using well-known uniprocessor scheduling algorithms and schedulability tests (BRANDENBURG, 2011). Hence, all the FP, JLFP, and JFDP scheduling algorithms reviewed in the previously section can be used in a partitioned multiprocessor approach.

In practice, each partition or processor could use a different scheduler. For instance, in a multiprocessor systems with four processor, two of them could use the EDF policy while the other two could use the RM policy. However, in this thesis, we only consider a unique scheduling policy for all processors. We consider the *Partitioned-FP (P-FP)* (*e.g.,* P-RM or P-DM), *Partitioned-EDF* (P-EDF), and *Partitioned-LLF* (P-LLF). Moreover, we are not aware of any research work that uses different scheduling policies on different processors.

Given a multiprocessor platform with $m$ processor or cores, the problem is to reduce the multiprocessor scheduling into a set of $m$ simpler uniprocessor schedulers. This problem is known as *task partitioning*, that is, each task must be statically assigned to a specific partition or processor and task migrations are not allowed. The partitioning problem is equivalent to the *bin-packing problem.*

**Bin-packing problem.** The bin packing problem is a NP-hard problem in the strong sense (GAREY; JOHNSON, 1990), and so is task partitioning. This means that partitioning/bin packing heuristics can produce a solution that is not optimal. The bin packing problem is formally defined as follows. Given a set of $n$ objects, $o_1, o_2, \ldots, o_n$, with different sizes, $s_1, s_2, \ldots, s_n$, and a finite number of bins $B$, each of capacity $V$. The goal is to assign each object to a bin in a way that minimizes the number of bins, without exceeding the capacity of any bin. A solution is optimal if it has a minimal $B$.

Considering the partitioned scheduling, the objects are tasks and their respective sizes are their utilizations. Processors correspond to bins and their capacity depends on the scheduling policy and the type of deadline constraint. For instance, if a task set has implicit-deadlines and is scheduled by the EDF policy, each processor has a capacity of 1.0 (100% of utilization as shown by Theorem 2.4.3). Consequently, a task set is deemed feasible on $m$ processors under partitioned scheduling if and only if there is a "packing" of all tasks into $m$ "bins" (BRANDENBURG, 2011).

The literature of bin packing heuristics is extensive and beyond the scope of this thesis. Instead of reviewing all existing heuristics, we present below the most relevant heuristics used by the real-time community.

- **First-fit:** this heuristic considers each bin in index order. It places an object in the first bin that has capacity to fit the object. If no bin exists, the object is placed in a new empty bin that is appended to the packing. The first-fit heuristic achieves an approximation factor of 2 – the required numbers of bins is not more than twice the optimal number of bins (JOHNSON, 1973). The time complexity of the first-fit heuristic is $O(n \; log \; n)$ (JOHNSON, 1973).

  Figure 22 shows an example of the first-fit heuristic with four bins, each of size 1.0, and seven tasks (their utilizations are also shown in the right side of the Figure). The heuristic assigns $T_1$ and $T_5$ to processor or bin 0, $T_2$, $T_6$, and $T_7$ to processor 1, $T_3$ to processor 2, and $T_4$ to processor 3.

- **Best-fit:** the best-fit heuristic considers all bins and places an object into the bin that will have minimal remaining capacity after placing the object. If no bin can accommodate an object,

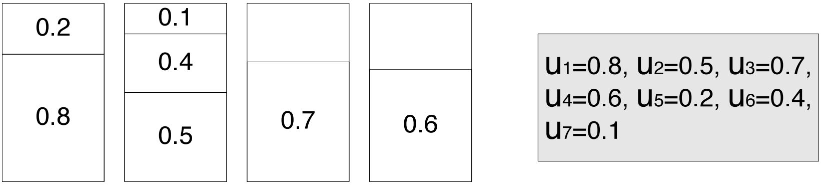


**Figure 22: Example of the first-fit bin-packing heuristic.**

a new bin is created for the object. The time complexity of the best-fit heuristic is also $O(n\ log\ n)$ (BASU, 2005).

Figure 23 shows an example of the best-fit heuristic, also with four bins, each of size 1.0, and the same seven tasks as in the early first-fit example. Tasks $T_1$ and $T_5$ are assigned to processor 0. Task $T_2$ is assigned to processor 1. Tasks $T_3$ and $T_7$ are assigned to processor 2 and tasks $T_4$ and $T_6$ are assigned to processor 3.

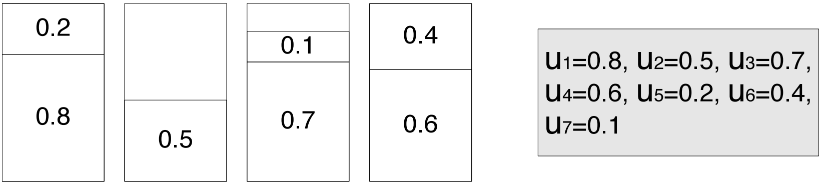


**Figure 23: Example of the best-fit bin-packing heuristic.**

- **Worst-fit:** the worst-fit heuristic is the opposite of the best-fit heuristic. The worst-fit places an object into the bin that will leave the most space left after placing the object. If the object does not fit in any bin, a new bin is created. In other words, the worst-fit places an object into the most empty bin. This heuristic is interesting, because it tends to spread the empty space over the used bins. If one wants to pack the bins with approximately the same size, this heuristic is the most suitable.

  Moreover, in a practical point of view, spreading the tasks into the available processors makes each processor more resilient to transient overloads and makes each processor to have some remaining “idle” capacity, which is required to deal with the RTOS run-time overhead (see Section 2.5) (BRANDENBURG, 2011).

  Figure 24 shows an example of the worst-fit heuristic using the same bins and tasks as in the early best-fit and first-fit examples.

Task $T_1$ is assigned to processor 0. Tasks $T_2$, $T_5$, and $T_7$ are assigned to processor 1. $T_3$ is assigned to processor 2, and $T_4$ and $T_6$ are assigned to processor 3.

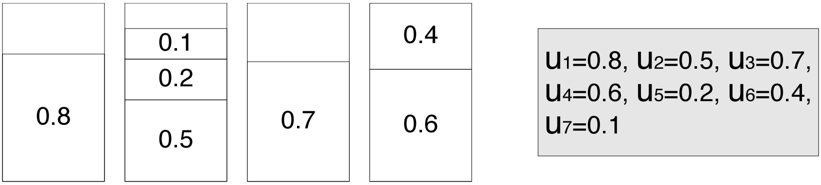


**Figure 24: Example of the worst-fit bin-packing heuristic.**

Generally, it is more difficult to pack large objects, as smaller objects are more likely to fit into the remaining capacity of partially used bins (GAREY; JOHNSON, 1990; BASU, 2005; BRANDENBURG, 2011). Thus, sorting the objects in a decreasing order by sizes improves the performance of the first-fit, best-fit, and worst-fit heuristics. The sorted version of the three heuristics are called First-Fit Decreasing (FFD), Best-Fit Decreasing (BFD), and Worst-Fit Decreasing (WFD), respectively. For a large number of bins, FFD and BFD require at most 1.22 times the number of bins used by an optimal solution, and for a number of bin smaller than four, it requires no more than 1.5 times the number of bins used by an optimal solution (JOHNSON, 1973). The WFD heuristic uses at most 1.25 times the number of bins used by an optimal solution (JOHNSON, 1973). In the experiments carried out in this thesis, we consider the three heuristics: FFD, BFD, and WFD.

As an example, Figure 25 shows the same task set, used in the previously examples, partitioned by the WFD heuristic. The tasks are first ordered by decreasing order of utilizations. Then, each task is partitioned using the worst-fit heuristic. As a result, task $T_1$ is assigned to processor 0, tasks $T_3$ and $T_7$ are assigned to processor 1, tasks $T_4$ and $T_5$ are assigned to processor 2, and tasks $T_2$ and $T_6$ are assigned to processor 3. Note that all processors have similar total utilizations ($u_{sum} \approx 0.8$).

As we stated before, in task partitioning, the number of bins is equivalent to the number of processors. Hence, the number of bins is fixed. If a partitioning heuristic attempts to create a new bin, we assume that the heuristic fails to partition the task set. In other words, the task set is not feasible for that number of processors and scheduling policy.

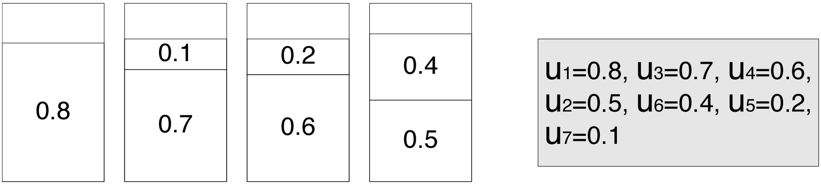


**Figure 25: Example of the worst-fit decreasing bin-packing heuristic.**

**Limitations.** While finding an optimal task assignment is intractable in the general case, the discussed bin packing heuristics find near-optimal solutions and are used in several real-time works due to its simplicity and relative good performance (BRANDENBURG *et al.*, 2008; BASTONI *et al.*, 2010b; GRACIOLI *et al.*, 2013; GRACIOLI; FRÖHLICH, 2013; BARUAH, 2013). Theoretically, a task set that has total utilization smaller than the number of processors could be partitioned. However, there are cases (task sets with heavy tasks[3]) that a task set cannot be partitioned, even though the task set utilization does not exceed the number of processor. For example, consider the scheduling of five tasks with the same utilization of 0.51 on four processors. There is no partitioning algorithm able to allocate these five tasks into four processors.

#### 2.4.2.2 Global Scheduling

In a global priority-driven scheduling, the $m$ (or less than $m$) highest priority jobs are scheduled at any time on the $m$ processors. When a job $J_i$ is released at the time $t$, it preempts the $m^{th}$ highest priority job $J_m$ if $Y(J_i,t) < Y(J_m,t)$ (BRANDENBURG, 2011). Consequently, a job can be preempted due to the release of a higher priority job and can be later resumed on another processor, resulting in a job migration. Migrations and preemptions cause delay with relation to the cache affinity, since a preempting job may evict the preempted task's cached data. In a global scheduling, there is only a single, shared ready queue (or another data structure, as heap or tree for instance), from which the scheduler chooses ready jobs to be scheduled based on available processors, scheduling policy, and jobs' priorities. The global

[3] A task that has an utilization higher than 0.5 is considered a heavy task.

scheduling does not require to solve a task partitioning problem, which is the source of capacity loss under partitioned scheduling (BRANDENBURG, 2011). Thus, there exist optimal SRT and HRT global schedulers for implicit-deadline task sets, but no optimal global scheduler exists for constrained- and arbitrary-deadline task sets (FISHER *et al.*, 2010). For example, the Global-EDF (G-EDF) algorithm is optimal for SRT systems and the *PF* (BARUAH *et al.*, 1996) is optimal with regard to HRT constraints. In this section we discuss *global fixed-priority* (G-FP), *global job-level fixed-priority* (G-JLFP), and *global job-level dynamic-priority* (G-JLDP) scheduling.

**Global FP scheduling.** The G-FP scheduling uses a FP scheduling policy, such as RM and DM, to schedule tasks on the available $m$ processors. Examples of G-FP schedulers are the *Global-RM* (G-RM) and *Global-DM* (G-DM). Dhall and Liu demonstrated that the RM policy is not optimal for G-FP scheduling (DHALL; LIU, 1978). This demonstration was called the *Dhall effect*. The Dhall effect shows that a task set $\tau$ with $n$ tasks, $n > m$, is not HRT schedulable even though $u_{sum(\tau)} \rightarrow 1$. Devi showed that the RM priority assignment policy does not necessarily ensure tardiness bounded under G-FP scheduling (DEVI, 2006). Davis and Burns stated that no provably optimal priority assignment or exact feasibility test is known for the G-FP scheduling of sporadic tasks (DAVIS; BURNS, 2011). However, there are response-time tests that give an upper bound on response times of tasks (DAVIS; BURNS, 2011).

We do not use G-FP in this thesis. Global scheduling is more adequate for SRT systems. In this context, G-EDF received more attention in the recent years, because it can ensure tardiness bound (DEVI; ANDERSON, 2005; DEVI, 2006). Thus, we focus on the G-EDF scheduling instead of G-FP scheduling.

**Global JLFP scheduling.** In the context of HRT systems, G-JLFP scheduling is not better than G-FP scheduling, because the Dhall effect also applies to the G-EDF scheduler (DHALL; LIU, 1978). Moreover, there is no optimal G-JLFP scheduler and no exact HRT feasibility test for G-JLFP. As in the uniprocessor JLFP scheduling, G-EDF is also the most studied G-JLFP scheduler. There are several HRT schedulability tests for the G-EDF scheduling which provide HRT bounds:

- **Density tests:** density tests use the total density ($\delta_{sum}$) to derive the HRT bounds (GOOSSENS *et al.*, 2003; BERTOGNA *et al.*, 2005; BAKER; BARUAH, 2007):

$$\delta_{sum}(\tau) \leq m - (m-1) \times \delta_{max}(\tau).$$

- **Baker's test:** the Baker's test (BAKER, 2003) is consider the seminal work on HRT schedulability analysis, serving as basis for several published schedulability tests (DAVIS; BURNS, 2011).

- **Multiprocessor RTA:** using the Baker's test approach, several works proposed multiprocessor RTAs (BERTOGNA *et al.*, 2005; BERTOGNA; CIRINEI, 2007). These multiprocessor RTAs are less pessimistic than the original Baker's test.

- **Baruah's test:** Baruah proposed a schedulability test based on the Baker's test for constrained- and implicit-deadline task sets (BARUAH, 2007; BARUAH *et al.*, 2009). Baruah's test extends the concept of problem window proposed by Baker and improves the multiprocessor RTA tests.

- **Load-based tests:** these tests consider the maximum-load of a task set to derive a safe HRT bound (BAKER; BARUAH, 2009). Load-based tests provide a more-accurate notion of the processor demand for constrained-deadline task sets and reduces the density test for implicit-deadline task sets.

For a comprehensive review on G-EDF schedulability tests see Davis and Burns and Bertogna and Baruah's recent surveys (DAVIS; BURNS, 2011; BERTOGNA; BARUAH, 2011) and Baker's analysis on EDF schedulability on multiprocessors (BAKER, 2005a).

**Global JLDP scheduling.** Changing a job's priority dynamically is a requirement to achieve optimality in global scheduling (BARUAH *et al.*, 1996). In this context, Baruah proposed *proportionate fairness* or simply *pfair* scheduling and designed the first optimal HRT global multiprocessor scheduler for implicit-deadline task sets, named (PF) (BARUAH *et al.*, 1996). PF is optimal, because it ensures that all deadlines are met, since the total task set utilization does not exceed the number of processors. PF is different from the traditional scheduling algorithm, because it explicitly requires that tasks make progress in a constant rate. In the classic periodic task model, each task has a progress rate given by relation of its WCET and period. Due to this progress rate notion, in PF each task executes in a uniform rate. Each task is divided in a set of sub-tasks, with the same execution time in each sub-task. The main idea is to break the task in smaller and uniform scheduling units, making the assigning of tasks to multiple processors easier than in the traditional periodic task model.

Other G-JLDP scheduling algorithm, although not optimal, is the G-LLF. As in the uniprocessor LLF, there are the strict and non-strict versions of the LLF. In this thesis, whenever we use G-LLF we refer to the strict version of the LLF.

#### 2.4.2.3 Clustered Scheduling

A clustered priority-driven scheduler splits the $m$ processors into $\lceil \frac{m}{c} \rceil$ disjoint sets or clusters of $c$ processors each (CALANDRINO *et al.*, 2007; BAKER; BARUAH, 2007). For example, a cluster can be composed of a group of processors that share a specific level of cache. For convenience, we assume that $m$ is an integer multiple of $c$. As in the partitioned scheduling, tasks are assigned to available clusters using a partitioning heuristic and globally scheduled within each cluster (allowing migrations among the cores of the same cluster). Partitioned and global schedulers are special cases of clustered schedulers: when clusters have size one, they are equivalent to a partitioned scheduler, while when they have size $m$, they are equivalent to a global scheduler. The clustered version of EDF and RM are the *Clustered-EDF* (C-EDF) and *Clustered-RM* (C-RM), respectively. In this thesis, we consider the C-EDF, although the RTOS infrastructure proposed in Chapter 4 supports any clustered scheduler variation.

There is no specific schedulability tests for clustered schedulers. The schedulability test for clustered schedulers involves the partitioned and global tests. A task set under clustered scheduling is deemed schedulable if and only if it could be partitioned by a task partitioning algorithm and each partition passed in a global schedulability test.

## 2.5 RUN-TIME OVERHEAD ANALYSIS

In this section we summarize the main sources of run-time overhead and provide an overview of main RTOS implementation strategies.

There are two basic ways to implement a scheduler in an OS: using event-driven scheduling or quantum-based scheduling (LIU, 2000). In the former, the OS performs every scheduling decision after an event, such as a job release or job completion. In the latter, the OS performs every scheduling decision at a timer interrupt. The hardware timer period (a *tick*) defines the interval of two successive timer interrupts. However, the quantum-driven scheduling can have precision problems. For instance, in a system that generates a timer interrupt every 10 ms, a 15 ms task period interval may have to wait for 20 ms to be released. On the other hand, a periodic timer can be very precise if every interrupt coincides with a job releasing (FRÖHLICH *et al.*, 2011).

In addition, the data structure responsible for ordering the tasks in the scheduler also plays an important role on the performance. The use of a list or heap as ready queue affects the time to insert and remove tasks, and consequently impacts the real-time scheduling performance.

The design of timer interrupts in the system is also relevant. In a multicore processor, a single core can handle timer interrupts or timer interrupts can be distributed across all cores (each core handles its own timer interrupts). Moreover, each timer interrupt can be periodic, based on tick counting, or single-shot, incurring in different interferences on the system.

Additionally, the processor architecture is also a source of overhead for real-time applications. For example, the way that the DRAM controller handles concurrent accesses, the implementation of bus arbiters, and different cache replacement algorithms significantly impact the execution time of an application. Although important, processor architecture overhead is difficult to be controlled by the OS and will not be discussed in this thesis.

In summary, the main sources of overhead in an RTOS are:

- **Context switching:** the process of storing and restoring CPU context of the running task and the next schedulable task. Its time is largely dependent on the CPU (e.g., registers, stack, etc) and OS.

- **Tick counting:** it is the delay spent to handle a timer interrupt and count a tick, in case of periodic timer, or to reprogram the timer, in case of single-shot timer.

- **Task releasing:** it is the delay required to release a task after a timer interrupt.

- **Scheduling:** it is the delay taken to choose a new task to be ran. Scheduling overhead includes operations such as inserting and removing tasks from the run queue and changing the state of a task, from ready to waiting for example.

- **Inter-process interrupt latency:** an inter-process interrupt (IPI) is necessary when a job is released on one core and must be executed on another core, because it has the lowest priority running task. An IPI is issued to call a reschedule operation on another core. In our platform (Intel i7 processor, see Table 12) for example, the WCET of an IPI is 0.3 µs (see Section 5.4).

- **Preemption and migration delay:** the Cache-related Preemption and Migration Delay (CPMD) is the delay caused by a preemption that incurs a loss of cache affinity after resuming the preempted job (BASTONI *et al.*, 2010a). Predicting CPMD on processors with complex shared caches hierarchy is a difficult problem. Related works commonly use empirical approximations and/or static analyses (WILHELM *et al.*, 2008; YAN; ZHANG, 2008; HARDY; PUAUT, 2009; BASTONI *et al.*, 2010a). This delay is highly dependent on the WSS of a task.

Each job incurs a scheduling and context switch overhead twice, a releasing overhead once, and a preemption/migration delay at most once (LIU, 2000; BRANDENBURG; ANDERSON, 2009). Figure 26 shows an example of the main sources of overhead in an RTOS. The OS releases task 1 (T1) during the execution of task 2 (T2) in the processor 1 (P1). Task 3 (T3) has a lower priority than T2, which implies in an IPI to schedule T1 in P2. P1 resumes T3 after the execution of T2. Note that there is no releasing overhead, because is the same job of T3. The figure does not show the tick counting overhead, which occurs at every timer interrupt handler.

The overhead introduced by interrupts (*e.g.,* tick and IPI) must also be considered in schedulability analyses. There are at least four

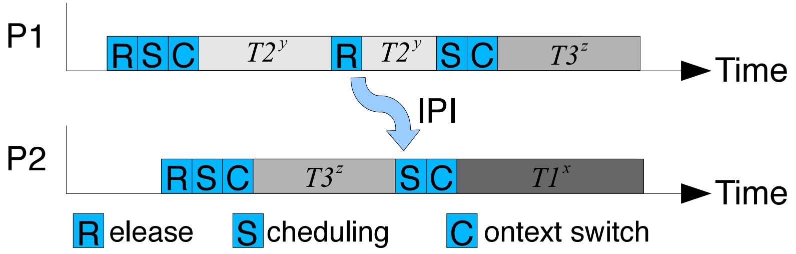


**Figure 26: An example of the sources of overhead in an RTOS (BRANDENBURG; ANDERSON, 2009).**

techniques to incorporate this overhead into schedulability analyses: quantum-centric accounting, task-centric accounting, preemption-centric accounting, and processor-centric-accounting (BRANDENBURG *et al.*, 2011; BRANDENBURG, 2011). In the quantum-centric method, each task's WCET is inflated to ensure the completion given a lower bound on the effective quantum length. In the task-centric method, interrupts extend each job's actual execution time and its WCET is inflated accordingly. In the processor-centric accounting, task parameters remain unchanged and interrupts reduce the processing capacity available to tasks. The preemption-centric approach modifies the task-centric interrupt accounting to be less pessimistic regarding the number of preemptions. The processor-centric method is difficult to integrate in EDF analyses and pessimistic with heavy tasks (BRANDENBURG, 2011). Usually, IPI latency is treated as self-suspension (i.e., jitter since it happens at the beginning of the job's period). However, G-EDF HRT schedulability tests cannot take self-suspension into account (BRANDENBURG, 2011). Thus, IPI latency is accounted for in G-EDF as an extra computation time by inflating the WCET. Hence, we use the preemption-centric interrupt accounting method in this thesis.

The preemption-centric interrupt accounting for EDF-based schedulers transforms each $T_i$ parameters to account for all sources of run-time overhead, including interrupts (see Table 5) (BRANDENBURG, 2011). Let $Q$ be the period of a timer interrupt, each $e_i$ is inflated as

**Table 5: Summary of sources of overhead and our notation (BRANDENBURG, 2011).**

| Notation | Overhead Description |
|---|---|
| $\Delta^{cxs}$ | Time interval to switch the context between two threads |
| $\Delta^{scd}$ | Time interval to choose a thread to run |
| $\Delta^{rel}$ | Time interval to release all threads that have reached their release time |
| $\Delta^{tck}$ | Time interval to count a tick in a periodic timer interrupt |
| $\Delta^{ipi}$ | Delay until IPI is received (IPI latency) |
| $\Delta^{cpd}$ | Delay caused by the loss of cache affinity (CPMD) |

follows[4]:

$$e_i^{'} = \frac{e_i + 2\times(\Delta^{scd}+\Delta^{cxs})+\Delta^{cpd}}{1-u_0^{tck}} + 2\times c^{pre} + \Delta^{ipi} \tag{2}$$

Where $u_0^{tck} = \frac{\Delta^{tck}+\Delta^{rel}}{Q}$ and $c^{pre}$ as defined by the following Equation:

$$c^{pre} = \frac{\Delta^{tck}+\Delta^{rel}}{1-u_0^{tck}} \tag{3}$$

We assume that on every timer interrupt ($Q$ time units), there is the overhead of counting a tick and releasing all threads that have reached their release time at the same interrupt.

[4] We consider the original parameters $\Delta^{ev}$ (delay until an interrupt service routine starts) and $\Delta^{cid}$ (loss of cache affinity due to an interrupt service routine) as negligible.

# 3 RELATED WORK

This chapter presents previous work on the fields related to this thesis. We organize the related work in four main topics: (i) memory management; (ii) real-time operating systems; (iii) run-time performance monitoring; and (iv) multicore real-time scheduling. For each of these topics, we provide a classification of the published research works.

The chapter is organized as follows: Section 3.1 presents the related work on memory management mechanisms for (multicore) real-time systems. Section 3.2 presents the main real-time operating systems that offer support for multicore processors. Section 3.3 summarizes the works that use HPCs to improve the predictability and performance of a multicore system. Finally, Section 3.4 presents the recent advances on the real-time multicore scheduling area.

## 3.1 MEMORY MANAGEMENT

The current cache memory hierarchy is one of the main factors of unpredictability in a multicore processor[1]. The execution time of a real-time task in a multicore processor can be affected by a number of different types of interference, depending on the behavior of the cache hierarchy:

- **Intra-task interference:** intra-task interference occurs when tasks have working set sizes larger than a specific cache level, or, in general, when two memory entries in the working set are mapped in the same cache set. The consequence in this case is that a task evicts its own cache lines. Intra-task interference also happens in single-core systems.

- **Intra-core interference:** intra-core interference happens locally in a core. Specifically, when a running task evicts a preempted task's cached data. As a result, the preempted task will experience an increase in its memory access time (and thus a de-

[1] The contents of this section appear in a preliminary version in the following submitted paper:
G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, A Survey on Cache Management Mechanisms for Predictable (hard) Real-Time Embedded Systems, to be submitted to the ACM Computing Surveys, 2014.

lay) as soon as it is rescheduled. The severity of the experienced delay depends on the particular cache line replacement policy implemented by the cache, as well as the length of the preemption and the memory access pattern of the preempting task (REINEKE *et al.*, 2007; GRUND; REINEKE, 2010; KIM *et al.*, 2013).

- **Inter-core interference:** inter-core interference is present when tasks running on different cores concurrently access a shared level of cache (KIM *et al.*, 2013). The memory bandwidth and bus are also sources of performance bottleneck in the inter-core interference. Moreover, two or more tasks may access the same cache line due to true or false sharing. Consequently, the cache coherence protocol will act to keep the data coherent in all private caches that have the shared cache line(s).

  Inter-core interference can also occur due to Symmetric Multi-Threading (SMT), when two or more hardware threads share the same private cache levels (L1 and/or L2). Since this type of interference is suffered by tasks that can run in parallel, an exact analysis would depend on all the possible interleaving of their executions. This combinatorial problem is clearly intractable, thus, inter-core interference is extremely difficult to integrate into a WCET static analysis framework (GUAN *et al.*, 2009).

Furthermore, memory accesses originating from a core can be classified as either demand accesses, if required by instructions executed on the core, or prefetch accesses, if speculatively issued by a hardware prefetcher unit. This leads to a further classification of cache interference under three different categories:

- **Demand-demand interference:** similarly to the discussion above, demand (load/store) requests from one core can interfere with requests from a different core.

- **Prefetch-prefetch interference:** prefetches from one core can delay or displace prefetches from another core, causing contention (EBRAHIMI *et al.*, 2009).

Although important for the average performance of the system, it is common practice in real-time systems to disable hardware prefetchers to eliminate the latter two categories of interference, thus making the processor more predictable.

**WCET derivation**. The main objective of cache management schemes for real-time embedded systems is to deal with the cited problems, simplifying the estimation of the tasks' WCET.

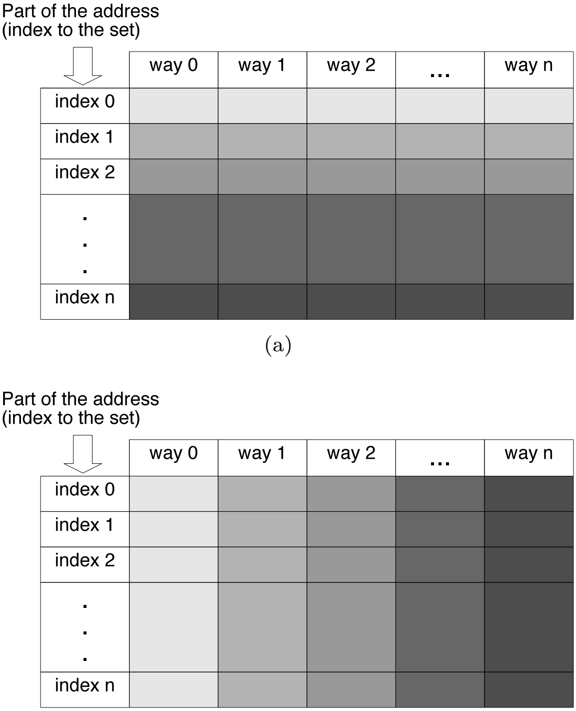
WCET estimation typically follows one of two main approaches: static analysis or empirical measurement (WILHELM *et al.*, 2008). In the former, a tool attempts to provide WCET by analyzing the application binary code without executing it directly on the hardware. Generally, WCET estimation tools are available only for simple processors, due to sophisticated hardware features, such as caches, branch predictors and pipelines, which make the static analysis extremely difficult or overly pessimistic (WILHELM *et al.*, 2008; LIANG; MITRA, 2008). In the latter approach, the application binary code is executed on the hardware platform for multiple times, and an estimation of the WCET is extracted from these executions. Moreover, in measurement-based approaches, the resulting value of WCET is inflated further with an error margin in order to account for unobserved conditions that can delay the execution (SEHLBERG *et al.*, 2006). Consequently, measurement-based approaches can also lead to final values of WCET that represent an overestimation of the real WCET.

The use of caches makes both static and measurement-based WCET estimation more complex, because the execution time of an instruction may vary depending on: (i) the data/instruction location in the memory hierarchy; (ii) if a memory access results in a miss or a hit in any of the cache levels; (iii) which cache coherence protocol is being used in case of true or false sharing; and (iv) which cache replacement policy is implemented in the cache controller (WILHELM *et al.*, 2008; ZHURAVLEV *et al.*, 2012). Note that while several real-time cache analysis frameworks have been provided for single-core systems (WILHELM *et al.*, 2008), current static analysis methodologies provide highly pessimistic bounds on cache misses for shared caches. Furthermore, to the best of our knowledge no existing static analysis technique is able to account for the effects of the coherence protocol.

Note that, even if WCET estimation is possible, the presence of inter-core interference greatly complicates validation and certification of multicore real-time systems. In domains such as avionics (Aeronautical Radio, Inc, 2013), separate applications are deployed as a collection of real-time software partitions, also known as Integrated Modular Avionics (IMA) partitions. The ARINC-653 (Aeronautical Radio, Inc, 2013) standard defines a hierarchy of certification levels for applications, ranging from level-A (highest certification level) to level-F (level with lowest criticality, or best-effort). The standard requires that applications certified at level-A for safety-critical operations run in partitions, that are isolated from any other partition from both a spatial and a temporal point of view. This implies that any module running in a different IMA partition should not be able to affect the behavior of any given level-A task. It also implies that it must be possible to define level-A IMAs independently from each other, so that the conformity of their parameters at production time is guaranteed regardless the final workload of the system. Unfortunately, due to inter-core interference, allowing the hardware components of commodity multicore platforms to operate in an unrestricted manner makes it significantly harder to provide said guarantees. In fact, if no predictable arbitration mechanism for shared resources is in place, the pessimism on task WCET can easily result to be unacceptable.

**Existing solutions**. Several works have been proposed to cope with the described cache memory hierarchy problems in the context of real-time embedded systems and to tighten the WCET estimation. In general, such works rely on enforcing timing isolation between sub-components in the system: the execution time of one sub-component should not depend on the behavior of other sub-components. Such isolation can be implemented (i) at the level of a single task, where sub-components are functions or basic blocks in the task's code; (ii) at the level of a core, where sub-components are tasks; or (iii) at the multicore system level, where sub-components are individual cores or software partitions running on each core.

**Cache partitioning**. Two are the approaches that are mostly used to enforce a more deterministic behavior on CPU caches. The first approach is cache partitioning, which divides the cache in partitions and assigns specific partitions to tasks or cores (WOLFE, 1994; LIEDTKE *et al.*, 1997; CHOUSEIN; MAHAPATRA, 2005; GUAN *et al.*, 2009; MANCUSO *et al.*, 2013; KENNA *et al.*, 2013; KIM *et al.*, 2013; GRACIOLI; FRÖHLICH, 2013). The objective is to reduce inter-core interference, which increases predictability and facilitates WCET estimation. There are two ways to perform cache partitioning: index-based or way-based partitioning. In the former, partitions are formed as an aggregation of associative sets in the cache. In the latter, partitions are formed as an aggregation of individual cache ways. Figure 27 shows an example of the two cache partitioning approaches in an $n$-way set-associative cache. In Figure 27(a), each set is considered a different and isolated partition (horizontal slicing). One or more sets are individually assigned to a task or core and all memory allocations performed by this task or core are mapped to the assigned set(s). In Figure 27(b), each way is considered an individual partition and one or more ways can be assigned to a task or core (vertical slicing). Cache partitioning can be further divided in hardware- or software-based approaches. Moreover, software-based approaches may need compiler or OS support.



(a)

(b)

**Figure 27: Classification of cache partitioning approaches: (a) overview of the index-based partitioning. (b) overview of the way-based partitioning.**

**Cache locking**. The second cache management mechanism for real-time embedded systems is cache locking. The central idea behind cache locking is to lock a portion of the cache in order to exclude the contained lines from being evicted by the cache replacement policy and by intra-task, intra-core, or inter-core interferences. Cache locking is a hardware-specific feature, which is typically done at a granularity of a single way or line. Figure 28 presents two cache locking variations. Figure 28(a) shows the locking of an entire way. The lock of a whole way means that the contents within that way across all sets cannot be evicted. Locking a whole way has not been explored deeply, because the number of ways is usually limited (in the range from 4 to 32) (MANCUSO *et al.*, 2013). Figure 28(b) shows the locking of an individual way or cache line. The cache line locking strategies provided by most of the current commercial embedded platforms are non-atomic. This makes it difficult to predict what is cached and what is not. Moreover, multicore shared caches are usually physically indexed and tagged. Thereby, if no manipulation is enforced on the physical addresses of the locked entries (usually by the OS), in the worst-case scenario no more than $n$ locked lines can be kept at the same time (MANCUSO *et al.*, 2013).

**Memory allocator**. Both cache locking and partitioning can be made available to applications through the memory allocator. A memory allocator is responsible for managing free blocks of memory in a large pool of memory (usually called *heap*), thus serving the application requests for memory spaces. The *malloc* libc function is an example of a user memory allocator. Two main concerns are common to memory allocators: performance and fragmentation. Performance with respect to how free blocks of memory are organized such that a search for a block is performed efficiently. Fragmentation with respect to how to avoid gaps between allocated memory blocks (BORG *et al.*, 2006). Besides performance and fragmentation, a memory allocator for real-time embedded systems must also be predictable, i.e., the time to allocate a memory block, despite its size, must be known. Furthermore, a memory allocator can also provide means for applications to use a cache partitioning or locking mechanism in a transparent and simple way. In this thesis, we provide an overview of memory allocators that are either predictable or cache-aware.

The common denominator for all the aforementioned techniques is to improve the predictability of real-time systems deployed on top of cache-based architectures, in order to provide better isolation for real-time embedded applications. In the following sections, we present the main memory management mechanisms for real-time systems, from the first studies of the field in 1990 up to the latest research published in

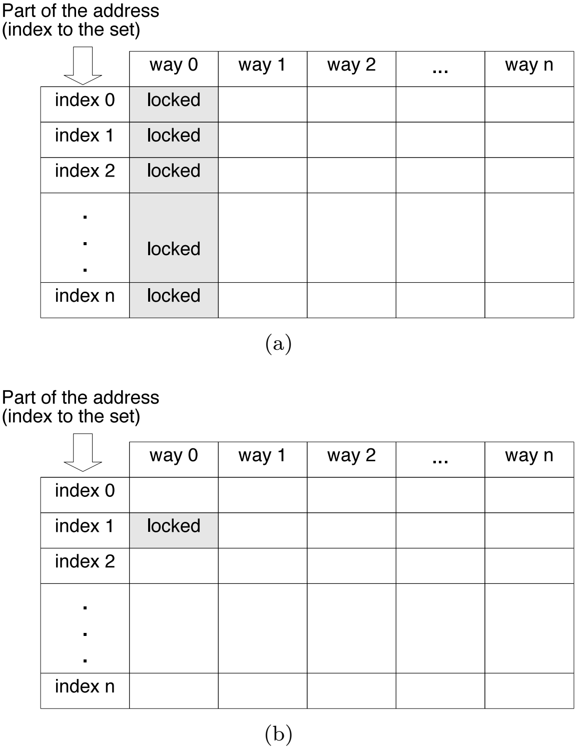


**Figure 28: Examples of cache locking variations: (a) overview of the way locking. (b) overview of the cache line locking.**

2013. The objective is to categorize each approach and to provide a detailed comparison in terms of similarities and differences. In summary, in this section we provide:

- A classification of each related work into one of the following categories: index-based cache partitioning, way-based cache partitioning, cache locking, or OS memory allocator. Moreover, we discuss related works that deal with the CPMD.

- A discussion of the main characteristics of the most relevant works for each category. In particular, we detail: (i) whether the approach is implemented in hardware or software, and in the latter case, whether it requires compiler or OS support; (ii) what type of isolation it provides, i.e., whether it addresses intra-task, intra-core, or inter-core interference; (iii) the level of isolation, i.e., how effective the approach is at preventing the addressed source of interference; and (iv) any further limitation or key assumptions in the work that might limit its applicability, in particular in an industrial context.

### 3.1.1 Index-based Cache Partitioning Methods

There are two index-based cache partitioning categories: hardware- (KIRK; STROSNIDER, 1990; LIU *et al.*, 2004; IYER, 2004; RAFIQUE *et al.*, 2006; SUHENDRA; MITRA, 2008; SRIKANTAIAH *et al.*, 2008) and software-based (WOLFE, 1994; LIEDTKE *et al.*, 1997; CHOUSEIN; MAHAPATRA, 2005; GUAN *et al.*, 2009; LU *et al.*, 2009; MURALIDHARA *et al.*, 2010; KIM *et al.*, 2013) techniques. The former requires special hardware support, such as specialized implementations that are not available in most of the current commercial processors. The latter has the advantage of being fully transparent to applications without demanding special hardware support. Software-based partitioning is further divided in those that require OS or compiler support.

**Hardware index-based partitioning.** Several hardware index-based cache partitioning have been proposed to improve the QoS of general-purpose applications. The Shared Processor-Based Split L2 cache organization assigns cache sets according to the CPU ID (LIU *et al.*, 2004). The L2 cache controller (configurable by the OS) keeps a table that maps the CPU ID to its sets. Every CPU memory access looks first at its available sets, but can subsequently look at other CPU sets before going to the main memory. Upon a miss, the requested memory block is allocated only into the available CPU sets. Set pinning is a similar approach, where cache sets are associated with owner processors (SRIKANTAIAH *et al.*, 2008). However, in set pinning, memory accesses that would lead to inter-core interference are redirected to a small processor owned private (POP) cache. Each processor has its POP cache, storing memory blocks that would cause inter-core cache misses. The objective is to reduce these cache misses and improve the average system performance.

CQoS classifies applications memory accesses in priorities and then assigns more set partitions to higher priority applications (IYER, 2004). The CQoS framework also implements a selective cache allocation in which it counts the number of lines occupied in the cache at a given priority level and probabilistically allocates or rejects cache line allocation requests.

Rafique *et al.* proposed a hardware implementation of a quota enforcement mechanism (RAFIQUE *et al.*, 2006). The quota is enforced at a set-level for different tasks/applications that access the shared cache. The mechanism requires the maintenance of an ownership information for each cache block, which is performed by adding a Tag-owner-ID along with each tag in the cache address bits.

Strategic Memory Allocation for Real-Time (SMART) was the first hardware-based implementation of a cache mechanism designed to provide predictability for (uniprocessor) real-time systems (KIRK; STROSNIDER, 1990). SMART divides the cache into M segments. These segments are allocated to N tasks ($N \geq M or N \leq M$). One specific partition referred as shared partition, which is formed by a set of segments, services tasks that do not require private partitions and provides coherent caching of shared data structures. Performance critical tasks have private partitions. Private partitions are protected and all tasks can access the shared partition. A cache location address is divided in a segment and cache ID fields that identify how many segments a task owns and which they are. A hardware flag is used to determine whether the shared partition or the private partition is active. During the memory access, the set address for the cache is combined with the cache ID and hardware flags. The cache ID, segment count field, and hardware flags are part of each task context and are loaded on every context switch performed by the OS. The authors presented a SMART designed for the MIPS R3000 processor.

Chousein and Mahpatra proposed a hardware-based cache partitioning for fully associative cache architectures (CHOUSEIN; MAHAPATRA, 2005). As reviewed in Section 2.1.2, a fully-associative cache places a memory block in any cache line. This means that the cache partitioning mechanism should provide a mean to search and isolate a cache line efficiently. To this end, multiple memory entries are aggregated together to associate them with a single tag entry to create a partition segment in the cache. Each tag entry has been built using content addressable memory (CAM) cells in conjunction with a few bits of ternary content addressable memory (TCAM) cells. The use of CAM and TCAM reduces miss ratio as demonstrated by Kohonen (KOHONEN, 1987).

Suhendra and Mitra proposed a cache partitioning mechanism able to perform three different partitioning strategies: (i) no partition, where a cache block may be occupied by any task, scheduled on any core; (ii) task-based partitioning, where each task is assigned a portion of the cache; or (ii) core-based partition, where each core is assigned a portion of the cache, and each task scheduled on that core may occupy the whole portion while it is executing (SUHENDRA; MITRA, 2008). Cache partitioning is then combined with a static cache locking mechanism (cache content remains unchanged throughout execution) or a dynamic scheme (cache content can be reloaded at run-time). The experimental evaluation performed in a dual-core processor (simulated) with 2-way set-associative L1 and L2 caches resulted in a set of guiding design principles for real-time systems: (i) if one wants to use a static cache locking, the best cache partitioning method is core-based partition; (ii) core-based partitioning emerges as overall winner independent of locking strategy; and (iii) dynamic cache locking is better than static cache locking only for tasks with a large number of hot regions and for smaller shared cache size.

**Software index-based partitioning.** The most common software-based cache partitioning technique is page coloring (TAYLOR *et al.*, 1990; LIEDTKE *et al.*, 1997; TAM *et al.*, 2007; GUAN *et al.*, 2009). Page coloring explores the logical to physical page address translations presented in current MMUs at OS-level, when caches are physically-indexed. Page addresses are mapped to pre-defined cache regions, avoiding the overlap of cache regions. Figure 29 illustrates the physical addresses from the cache and OS point of views for the hardware platform used in our experiments (Intel i7-2600, see Table 12). By controlling the colored bits of the set-associative cache number, the OS can change the mapping of 4 KB pages in the physical memory and thus the cache location. Our platform has an 8 MB 16-way set-associative shared L3-cache with 64-bytes per line. Each of the 16 ways can store a cache line. There are $2^{13}$ sets in the cache (8 MB/16ways × 1way/64 B). Thus, the first 6 bits in the cache address access a cache line, the next 13 bits access a set, and the next 13 bits define a line from one of the 16 ways (Tag in Figure 29).

The idea of page coloring is to assign color 0 to page 0, color 1 to page 1, and so on, starting again from color 0 after reaching the

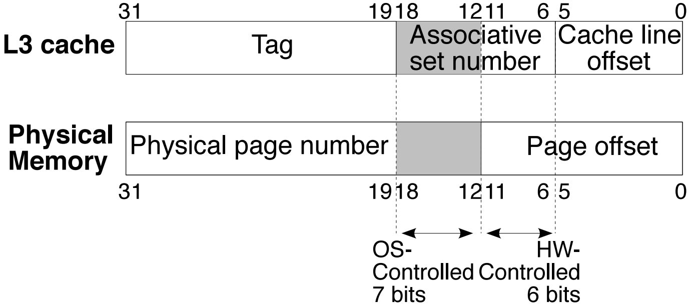


**Figure 29: Physical address view from the cache (on top) and from the OS (bottom)**

maximum color number. Figure 30 shows the mapping of physical pages to cache locations in our platform. There are 64 cache lines in each 4 KB page. Each of these lines has a different cache set index. Since 7 bits are available to page coloring (cache size / number of ways / page size), page 128 maps to the same color as page 0. Therefore, it is possible to partition the cache by assigning different colors to tasks.

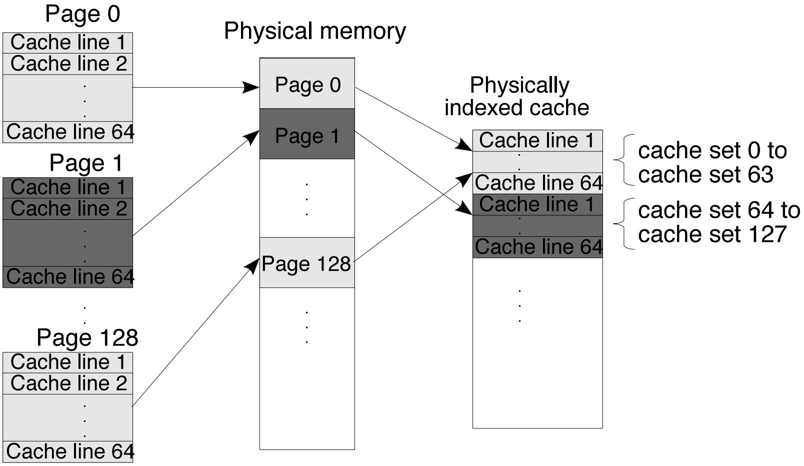


**Figure 30: Mapping physical pages to cache locations.**

For practical reasons, the cache mapping algorithms implemented in real multicore processors produce synonyms (*i.e.*, they are not fully associative), so different memory addresses can be mapped to the same cache line. This leads to *false sharing*, which is a source of interference indistinguishable from *true sharing* from the perspective of page coloring.

Page coloring was first implemented in the MIPS OS to improve performance stability (TAYLOR *et al.*, 1990). Then, page coloring was used to evenly distribute cache accesses across the whole cache and subsequently reduce cache misses within a single application (KESSLER; HILL, 1992; ROMER *et al.*, 1994; SHERWOOD *et al.*, 1999). More recently, several shared cache partitioning mechanisms based on page coloring for general-purpose systems have been proposed (TAM *et al.*, 2007; LIN *et al.*, 2008; ZHANG *et al.*, 2009; GUAN *et al.*, 2009).

Tam *et al.* used information collected from HPCs at run-time to estimate the size of each shared L2 cache partition (TAM *et al.*, 2007). Experimental results have shown that cache partitioning can recover up to 70% of degraded instruction per cycles due to cache contention (TAM *et al.*, 2007). Lin *et al.* implemented a page coloring mechanism in the Linux kernel for IA-32 processors (LIN *et al.*, 2008). The authors changed the Linux memory management to support multiple lists, each one linking free pages with the same color. When a page fault occurs, the kernel searches for free pages in the free colored lists in a round-robin fashion. Two cache partitioning policies were supported: static and dynamic. A static cache partitioning policy predetermines the amount of cache blocks allocated to each program at the beginning of its execution. In the dynamic policy, a page recoloring is enforced whenever a process increases its WSS. The kernel performs page recoloring by rearranging the virtual-physical memory mapping of the process. This approach involves allocating physical pages of the new color, copying the memory contents, and freeing the old pages. Although page recoloring achieved good QoS for general-purpose applications, the cost of copying pages in a real-time application may result in deadline misses. Zhang *et al.* proposed a hot-page coloring approach which enforces coloring on only a small set of frequently accessed (or hot) pages for each process (ZHANG *et al.*, 2009). Hot pages are identified by checking the page table entry access bit. Whenever a page is accessed, its access bit is automatically changed by the processor. By periodically checking and clearing the access bit, it is possible to estimate each page's access frequency. This approach also supports page recoloring for dynamic execution environments.

Several works proposed software-based cache partitioning to increase predictability of (multicore) real-time systems (WOLFE, 1994; MUELLER, 1995; LIEDTKE *et al.*, 1997; BUI *et al.*, 2008; GUAN *et al.*, 2009; KIM *et al.*, 2013; KENNA *et al.*, 2013; GRACIOLI; FRÖHLICH, 2013). We classify these works in two categories: those that are implemented at the OS-level and those that are implemented at the compiler-level. We discuss both categories below.

**Real-time OS index-based partitioning.** OS-controlled cache partitioning for real-time systems was first proposed by Wolfe (WOLFE, 1994). Wolfe proposed a mechanism similar to page coloring to provide predictable execution times for low-priority tasks in a preemptive uniprocessor system. In fact, his approach alters the address decomposition into tag, index, and offset as in the page coloring partitioning. The technique requires the task set to be static and all programs to be compiled by the same compiler prior to the system start. Code and data of a task are logically restricted to memory portions that map into cache lines assigned to the task. Liedtke *et al.* extended the Wolfe's work and used page coloring to provide predictability for uniprocessor real-time systems (LIEDTKE *et al.*, 1997). Bui *et al.* considered the cache partitioning problem as an optimization problem whose objective is to minimize the worst-case system utilization under the constraint that the sum of all cache partitions cannot exceed the total cache size (BUI *et al.*, 2008). The authors proposed a genetic algorithm to solve the optimization problem. The cache partitioning mechanism is based on page coloring. Experimental evaluation has shown an improvement on the schedulability of single-core systems.

Kim *et al.* stated that cache partitioning based on page coloring suffers from two problems: (i) the memory co-partitioning and (ii) the limited number of partitions (KIM *et al.*, 2013). Then, the authors proposed a cache management scheme in which they assign to each core a set of private partitions to avoid inter-core cache space interference. However, tasks within each core can share cache partitions. Although this can result in intra-core interference, it solves the aforementioned problems. In other words, each task can now have a larger number of partitions which could improve its execution performance. In addition, memory partitions can be utilized by all tasks. They bound the penalties due to the sharing of cache partitions by accounting for them as cache-related preemption delays when performing the schedulability analysis.

$MC^2$ treats the management of cache lines as synchronization and scheduling problems (KENNA *et al.*, 2013). The proposed approach uses page coloring and real-time multiprocessor locking protocol together. The OS associates a set of colors to each task. With locking mechanism, a job must acquire a lock for each color it needs before execution, and it releases this lock when it finishes execution. Hence, the entire job execution is treated as a critical section. This non-preemptive execution of critical sections can create a long priority inversion blocking. To mitigate this problem, the authors proposed period splitting and job splitting as two ways to reduce the detrimental effect of lengthy critical sections. However, this may cause jobs to reload their working sets. Unlike the locking mechanism described above, the cache colors can be treated as preemptive resources in which the concurrent accesses can be mediated by scheduling. Instead of dealing with a scheduling problem over two preemptive resources – the processor and the cache – the authors reduce the problem into uniprocessor scheduling. They define a logical *cache processor* to whom they assign tasks that share either cache colors or processor. They evaluated the schedulability of each cache processor as a uniprocessor, and apply known schedulability tests. The authors compared both techniques in terms of schedulability of the P-RM algorithm and concluded that cache locking approach is better than the scheduling approach. In our work, we extended the page coloring performance analysis to P-EDF, C-EDF and G-EDF algorithms and considered also the impact of the OS overhead on the WCET. Furthermore, unlike $MC^2$, which is implemented using a real-time patch for Linux, we evaluated the cache partitioning and the real-time algorithms on an RTOS designed from scratch, with less interference (GRACIOLI; FRÖHLICH, 2013).

**Real-time compiler index-based partitioning.** Mueller was the first to introduce compiler support for cache partitioning in the context of real-time systems (MUELLER, 1995). The compiler receives the cache size and the partition size of a task as additional input. The output is separate object files for each code partition and each data partition. Then, the object files of all tasks are combined into an executable by the linker. To deal with code partitions larger than the partition size, the compiler inserts an unconditional jump to the next code partition at the end of each code partition. Each partition is stored in a separate object file, which may be padded with no-ops at the end to extend it to the exact size given by the cache partition size. Global data is split into memory partitions of the data cache partition size. The compiler ensures that no data structure spans multiple partitions. If the size of a data structure exceeds the cache partition size, it is split over multiple partitions and the compiler needs to transform the access to the data structure. Local data on the stack is split into partitions by manipulating the stack pointer whenever necessary. Dynamic allocation on the heap is supported as long as memory requests do not exceed the cache partition size. OS and libraries are also treated as separated partitions. The used cache partition mechanism is based on page coloring and the approach targets uniprocessor real-time systems.

Bugnion *et al.* extended the Mueller's approach and implemented page coloring support in the SUIF parallelizing compiler (BUGNION *et al.*, 1996). The objective was to improve performance of general-purpose multiprocessor systems. The proposed technique, named compiler-directed page coloring (CDPC), uses information available within the compiler to predict the access patterns of a compiler-parallelized application, such as array access patterns. This information is then used to customize the application's page mapping strategy by the OS, i.e., it is used to perform cache partitioning through a page coloring mechanism. Basically, the OS generates preferred color for each virtual page. Additionally to the compiler information, the OS also receives machine-specific parameters, such as the number of processors, cache configuration, and page size. Then, the OS tries to honor all received information as much as possible.

Vera *et al.* used compiler techniques together with cache partitioning and locking to improve the predictability in preemptive multitasking uniprocessor systems in the presence of data caches (VERA *et al.*, 2003b). The proposed predictable framework works with both hardware or software cache partitioning schemes. The compiler technique uses loop tiling, and padding to reduce capacity and conflict misses, that is, it reduces intra-task interference. Loop tiling reorders accesses in such a way that reuse distance is shortened and padding modifies the data layout of arrays and data structures. Cache locking is used to lock cache lines that are accessed by more than one task. The authors compared the proposed framework with static cache locking in which all tasks share the whole cache in terms of worst-case performance of a multitasking system. The results indicated that the framework was able to schedule tasks that need a high throughput.

### 3.1.2 Way-based Cache Partitioning Methods

Way-based partitioning has two main advantages. First, it requires limited changes to the set-associative cache organization which do not have a dramatic impact on the overall structure. Second, this partitioning scheme keeps requests for the different compartments isolated from each other. Thus, there is no contention for the cache ways at the cores. However, the main drawback with this approach is that the number of partitions, as well as the granularity of the allocations, is limited by the associativity of the cache. Increasing the associativity of a cache is not always feasible or efficient since a higher-order associativity determines an increase in the cache access time and tag storage space.

Ranganathan *et al.* proposed a cache architecture that allows dynamic reconfiguration of partitions (RANGANATHAN *et al.*, 2000). Specifically, the traditional structure of a set-associative cache is adapted to allow dynamic definition of partitions which can be assigned to given address ranges. The dynamic reconfiguration can be performed at runtime by software. In the proposed architecture, each cache partition can be the result of the aggregation of one or more cache ways. Additional cache logic is required to differentiate between address spaces or partitions; to multiplex the tag comparators; and to generate different miss/hit signals for each addressed partition. However, the additional logic and wiring has been proven to minimally affect the cache access time, with a variable overhead that is proportional to the ratio $\frac{\#\ of\ partitions}{cache\ size}$. Although the described scheme has not been designed for real-time systems, a straightforward extension could be designed for multi-core systems, allowing to bind the address space of a task running on a given core to a given cache partition. This would eliminate the inter-core interference in an exclusively owned partition. However, intra-task interference (self-evictions) is still possible.

Chen *et al.* proposed a cache management scheme that partitions each cache set into shared and private ways (CHEN *et al.*, 2009). Assume that the associativity of the shared cache is A, and that all cache lines are classified into shared and private. Given a quota Q for shared cache lines, the cache controller partitions each cache set such that shared cache lines take Q ways in each set while private lines take A-Q. The main idea is to assign different partitions to shared and private data and thus reduce the LLC miss rate and increase QoS and fairness.

Similarly to Chen, Sundararajan *et al.* proposed RECAP: an architecture for a set-associative cache that can be partitioned across cores at the granularity of a single cache way (SUNDARARAJAN *et al.*, 2013). Different cache ways are assigned to shared and private data based on the observation that the majority of memory accesses (around 80%) for parallel applications is performed in shared memory, while above 90% of cache data belong to private memory regions. Moreover, RECAP performs an automatic arrangement of the partitioned ways, so that private data assigned to specific cores are allocated on ways that are positioned on the left side of the cache, while shared data are allocated on ways starting from the right end of the cache. The enforced data-alignment keeps the unused ways in the center of the cache and thus allowing dynamic power-saving by powering down unused ways. Partition arrangement for each core is performed in order to maximize utilization of the cache ways: a heuristic algorithm monitors the number of misses for each core and determines how many cache blocks need to be assigned in order to capture a given fraction of cache misses. Finally, the computed ways-to-cores assignment is enforced on the proposed cache architecture by programming a series of configuration registers. The results have shown a 15% average performance increase for large applications, with a reduction of power consumption which is above 80% (for both dynamic and static energy). However, even though this approach is effective to limit inter-core interference, its scope is beyond predictability, making it suitable for SRT rather than HRT applications. First because intra-core/intra-task interference is not addressed; second due to the heuristic approach that is being used to perform ways-to-cores assignment at run-time.

Varadarajan *et al.* proposed the idea of "molecular caches": CPU caches that are organized as a series of molecules (VARADARAJAN *et al.*, 2006). In this architecture, molecules are small in size (8-32KB) and reflect the structure of a direct-mapped cache. Molecules are grouped into tiles that can be assigned statically or dynamically to cores. Thus, by definition, per-core partitions are defined at the granularity of tiles, that in turn represent an aggregation of cache ways. Moreover, in order to address intra-core interference, individual regions can be defined internally to each tile and assigned to a specific application. In order to do this, each molecule of the same region inside a given partition exposes a configuration register which can be programmed with a unique identifier of a running application. In this way, all the molecules of a given region will be exclusively used by the matching application. The proposed scheme results effective in isolating cache misses from different running applications on the same or different cores, but the complexity of the resulting structure may not scale well with cache size.

Since molecules (and thus regions) can be dynamically assigned to applications (or unassigned), a high degree of run-time reconfiguration is possible. As such, assignment decision need to be made by a software module that runs periodically or in an adaptive manner. The metric used in the assignment controller is keeping the experienced number of caches misses as close as possible to a threshold that is statically defined for each application. Even though inter-task interference is not addressed by such a partitioning scheme, molecular cache represent an effective solution to prevent inter/intra-core interference, while providing the cache with a fine-grained partitioning mechanism. Moreover, the particular choice of associativity and size for the molecules leads to a circuit design that enables a reduction in energy dissipation. However, due to its complexity, a hardware implementation of molecular caches is not available to date. The applicability of such an architecture to real-time applications would be broad, assuming that the assignment controller employs a predictable allocation strategy.

Qureshi *et al.* proposed a utility-based approach to dynamically partition the cache (QURESHI; PATT, 2006). The main insight is that differences in the working set size as well as memory addressing pattern of applications affect the way they benefit from cache assignment. Applications that exhibit low spatial and temporal locality in their memory access patterns, combined with a large memory size (in comparison with the size of the cache), benefit less from cache resources than applications with opposite characteristics. Thus, the former can be classified as a low-utility application, while the latter can be considered as a high-utility application from the cache assignment point of view. To account for cache utility, the authors modified the cache controller by adding a low-overhead circuit. The new circuit consists of a number of utility monitors (one per each core accessing the shared cache) and a single circuit that runs the partitioning algorithm relying on data collected at the utility monitors. The partitioning algorithm reads all the hit counters from the different utility monitors and computes a cache partitioning at the granularity of a way that minimizes the overall number of misses suffered by all the applications. Evaluation results have shown that such partitioning approach is able to provide fairness and performance speedup if compared to using LRU and static partitioning (*e.g.,* an even split of the cache in two halves for two cores). The hardware overhead is relatively low when applied to LRU caches, but can significantly increase for different cache replacement policies. However, the inherently heuristic approach employed in the partitioning algorithm is not directly suitable for HRT purposes, since the amount of assigned cache strictly depends on the workload on other cores. This problem could be mitigated allowing a minimal assignment of cache per each core or application.

### 3.1.3 Cache Locking Methods

Cache locking prevents the eviction of cache lines by marking them as locked until an unlock operation is performed. Cache locking is a hardware-specific feature that it is not present in all of the modern multicore processors. There are two ways to lock a cache content: (i) through an atomic instruction to fetch and lock a given cache line into the cache, or (ii) defining, for each single CPU in the system, the lock status of every cache way. The last mechanism is called *lockdown by master* in multicore systems (MANCUSO *et al.*, 2013). Some embedded multicore platforms feature only an atomic instruction, such as Freescale P4040 and P4080 platforms, while other platforms, such as TI OMAP4460 and OMAP4430, Nvidia Tegra 2 and 3, Xilinx Zynq-7000, and Samsung Exynos 4412, implement a lockdown by master mechanism.

To exemplify the applicability of a cache locking mechanism, consider a dual-core platform with a 2-way set-associative cache and a cache controller that implements a lockdown by master mechanism. We could set up the hardware so that way 1 is unlocked for core 1 and locked for core 2, while way 2 is locked for core 1 and unlocked for core 2 (MANCUSO *et al.*, 2013). This means that a task running on core 1 would deterministically allocate blocks on way 1 and blocks allocated on way 1 could never be evicted by a task running on core 2. The same situation occurs on way 2 referring to core 2. This assignment can be easily changed at run-time by manipulating a set of registers provided by the cache controller interface (MANCUSO *et al.*, 2013). If the platform provides an atomic instruction to fetch and lock a cache line, a software procedure that realizes a mechanism functionally equivalent to the lockdown by master can be easily built (MANCUSO *et al.*, 2013). Hence, cache locking provides a more predictable and controllable access to shared caches, easing the WCET estimation and improving the performance of real-time applications. Cache locking avoids/reduces intra-task, intra-core, and inter-core interferences.

We classify the works that use cache locking as hardware-only, when a specific hardware design is proposed, or hardware-software approaches, when the hardware available on current multicore processors is used by a software layer (*e.g.,* RTOS, compiler, or specific algorithms). We further state if the work provides a cache locking only for instruction caches, data caches, or both caches.

**Hardware-only approaches.** Asaduzzaman *et al.* introduced a miss table (MT) based cache locking scheme at L2-cache to improve timing predictability for real-time applications (ASADUZZAMAN *et al.*, 2010). The MT holds information of block addresses of the current application that cause most cache misses if not locked. The cache miss for each block address is obtained by the Heptane simulation tool (AVILA; PUAUT, 2014). Then, the MT sorts the block addresses in descending order of the miss numbers. The proposed MT-based scheme was evaluated by simulating an 8-core processor with 2 levels of cache using the MPEG4 and H.264 decoding and FFT algorithms. The results have shown a predictability improvement in all algorithms.

Sarkar *et al.* used cache locking to provide a predictable task migration scheme applicable to the PFair scheduling algorithm (SARKAR *et al.*, 2011). The authors proposed several cache migration models in the presence of a cache locking mechanism to deterministically bound the migration delay of tasks. The work uses the push model hardware feature, where every cache controller has a push logic block and each cache line has a PID associated with it. When a task migrates, the new core is initialized to start pushing the cache lines with a push request. Then, the push block identifies locked lines pertaining to the migrated task through hardware enhancements and uses the cache line PID to correctly manage locked cache lines. Simulation results have shown a reduction of the migration cost of up to 56%.

**Hardware-software approaches.** Campoy *et al.* were the first authors to use cache locking in the context of uniprocessor real-time systems (CAMPOY *et al.*, 2001). The authors proposed a genetic algorithm that selects which instruction blocks are loaded and locked in the cache in a preemptive, multitasking system. In such a system, the locking of a cache line from one task affects the other tasks. The genetic algorithm provides the set of blocks, an estimation of the WCET of each task executing in a locked cache with the set of blocks loaded and locked, and the response time of all tasks considering the WCET estimated using the locking cache. Experimental results indicated that the proposed algorithm allows the calculation of the response time of tasks in a preemptive scheduler with negligible overestimation and without performance loss (CAMPOY *et al.*, 2001).

As in (CAMPOY *et al.*, 2001), Puaut and Decotigny also explored the use of static cache locking of instruction caches in uniprocessor multitasking systems, addressing intra-task and intra-core interferences (PUAUT; DECOTIGNY, 2002). The authors proposed two algorithms to select the instruction cache blocks that should be locked. In contrast to (CAMPOY *et al.*, 2001), the proposed algorithms select the contents of the locked cache in a non blind manner, by using the tasks memory access patterns of the instruction flow. The first algorithm aims at minimizing the CPU utilization of the task set, while the second aims at reducing the intra-core interference using a fixed-priority scheduler. A worst-case performance analysis of the proposed algorithms on two task sets has shown that they perform better than static cache analysis for large instruction caches and caches with a large degree of associativity (PUAUT; DECOTIGNY, 2002).

Vera *et al.* combined compile time cache analysis with data cache locking to improve the WCET estimation and performance of uniprocessor real-time systems (VERA *et al.*, 2003a). A compile time algorithm identifies those regions of code where a static analysis does not precisely predict their behavior. The algorithm uses a reuse vector to perform cache locality analysis and to select data to be loaded and locked in the cache. Static analysis is used in the other code regions. Also, the compile time algorithm inserts beyond the lock instructions, unlock and load instructions and compute the WCET in presence of k-way set-associative data caches. The authors implemented the algorithm in the SUIF2 compiler and obtained a more predictable cache behavior with minimal performance loss.

Falk *et al.* proposed a technique that also explored static locking of instruction caches at compile time to minimize the WCET in uniprocessor real-time systems (FALK *et al.*, 2007). Differently from the previous works, the technique explicitly takes the worst-case execution path into account in order to apply cache locking. An optimization algorithm takes a compiled and linked binary executable for the ARM920T processor as input. Then, the algorithm calls a static WCET analysis tool (aiT), which extracts WCET-relevant data of the binary's function. From this data, the optimization algorithm selects those functions to be locked in the instruction cache. Experimental results reported a reduction between 54% and 73% in the WCET of some benchmarks running on the ARM920T processor.

Liang and Mitra proposed two cache locking algorithms to improve the average-case instruction cache performance for uniprocessor systems (LIANG; MITRA, 2010). Both algorithms rely on two phases: (i) a profiling phase creates the temporal reuse profile (TRP) for each memory block by simulating or executing the application on the target platform; and (ii) a locking phase determines the cache sets to be block to maximize the number of cache hits. An optimal algorithm searches a binary decision tree, which represents the entire search space. Each level in the binary search tree corresponds to locking decision for one memory block in the set. A heuristic algorithm maximizes the number of cache hits by individually analyzing each cache set and computing the number of hits in that cache set. The authors performed an experimental evaluation using the SimpleScaler simulation framework with different cache sizes and associativities. The obtained results have shown an improvement of up to 24% in the instruction cache miss rate. Although not proposed for real-time systems, the profiling technique could be integrated into a WCET analysis framework to indicate each cache lines are evicted or not during the application execution.

Aparicio *et al.* proposed a method to select the best cache lines to be loaded and locked (*i.e.,* dynamic cache locking) into the instruction cache at each context switch, taking into account both intra-task and intra-core interferences (APARICIO *et al.*, 2011). The method is based on an integer Linear Programming (ILP) method, named Lock-MS (Maximize Schedulability), and targets uniprocessor real-time systems. The ILP-method obtains the cache lines to be loaded and locked based on the requirements of a single task and the effects of interferences between tasks. At run-time, a dynamic cache locking method preloads the cache contents at every context switch, providing a solution that minimizes the worst overall cost of a preemption. The authors compared Lock-MU with a static locking mechanism (PUAUT; DECOTIGNY, 2002) and obtained better performance results.

Mancuso *et al.* proposes a memory framework that uses profiling techniques to analyze the memory access pattern of tasks and obtain the most frequently accessed memory pages (MANCUSO *et al.*, 2013). Then, cache locking and page coloring are used to provide isolation among tasks and increase predictability. The framework was implemented and evaluated in the Linux kernel.

### 3.1.4 OS Memory Allocators

Programmers usually allocate data with little or any concern for cache memory hierarchy. Hence, the resulting data allocation in the cache memory hierarchy may interact poorly with the program's memory access pattern (CHILIMBI *et al.*, 2000), incurring in intra-task, intra-core, and inter-core interferences. In a multicore real-time system, this bad memory allocation may cause the loss of deadlines and it is not tolerable. Thus, OS memory allocators for real-time systems must be predictable (despite the memory size being allocated) and cache-conscious in order to allocate memory efficiently. Basically, we can divide OS memory allocators for real-time systems in three categories: (i) those that are cache-aware, i.e., provide a cache partitioning or cache locking mechanism to ensure predictability; (ii) those that are predictable, i.e., the time to allocate memory blocks is bounded; (iii) those that are predictable and cache-aware. In this section we review OS memory allocators that are designed to be predictable and/or cache-aware.

**Cache-aware allocators.** Chilimbi *et al.* proposed a memory allocator (named *ccmaloc* – cache-conscious malloc) that receives the requested bytes and a pointer to an existing object that the program is likely to access contemporaneously with the element to be allocated (CHILIMBI *et al.*, 2000). The allocator attempts to allocate the requested data in the same cache block as the existing item, thus improving data locality, cache hit rate, and the average execution time. The authors also integrated the memory allocator with a data structure reorganizer. The reorganizer basically transforms a pointer structure layout into a linear memory layout and maps its elements to reduce cache conflicts using page coloring. The main drawback of this approach is the need for copying data to a new linear memory region and the absence of a predictable memory allocation time.

Cache-Index Friendly (CIF) also tries to improve the average execution time, instead of improving the predictability, by explicitly controlling the cache-index position of allocated memory blocks (AFEK *et al.*, 2011). The central idea in CIF is to insert small spacer regions into the array of blocks within the allocator to better distribute block indices, thus disrupting the regular ordering of block addresses, returned by the allocator. The authors performed a set of experiments to show that CIF reduces intra-task and inter-core interferences. CIF, however, is not designed for real-time systems.

**Time-predictable allocators.** The Half-fit algorithm was the first dynamic memory allocator to perform allocation/deallocation in a constant and predictable time (OGASAWARA, 1995). In Half-fit, free blocks of size in the range $[2^i, 2^{i+1})$ are grouped into a free list indexed by $i$. When a block is deallocated, it is immediately merged with neighboring free blocks. After merging, the index of the new free block of size $r$ in a free list bit vector can be found using $i = \lfloor log_2\ r \rfloor$. The bit vector is a word-length variable used to keep track of which free lists are empty and which are not. The logarithm operation in a bit vector can be executed in one cycle in several 32-bit CPUs. Half-fit eliminates the search on free lists for allocation by calculating the index $i$ and taking a free block of memory from the list $i$. If the list $i$ is empty, the allocation algorithm takes a block from the subsequent non-empty list whose index is closest to $i$. If allocated blocks are larger than requested sizes, the allocated block is split in two parts: one is returned to the requester and another one is relinked on the free list. Both allocation and deallocation have time complexity of O(1) (OGASAWARA, 1995).

Two-Level Segregated Fit memory allocator (TLSF) uses a segregated fit mechanism to implement a good-fit policy, which returns the smallest chunk of memory big enough to hold the requested memory block (MASMANO *et al.*, 2004). TLSF limits the size of memory to be allocated in 16 bytes, to store inside the free blocks, all information needed to manage them, including the pointers to free blocks. The segregated fit mechanism uses an array of free lists, in which each list (accessed by a specific array position) holds free blocks within a size class. The array of lists is organized in two-levels to speed up the access to free blocks and to reduce fragmentation. The first-level divides free blocks in classes that are power of two (16, 32, 64, 128, etc) and the second-level sub-divides each first-level class linearly, where the number of divisions is a user configurable parameter. As in Half-fit, a bitmap marks which lists are empty and which ones contain free blocks. TLSF uses the boundary tag technique (KNUTH, 1997) to easily coalesce free blocks. The boundary tag adds a pointer to the beginning of the same block to each free or used block. Thus, when a block is released, the pointer of the previous block (which is located one word before the released block) is used to access the head of the previous physical block to check whether it is free or not and merge both accordingly (MASMANO *et al.*, 2004). Hence, each free block is linked in the segregated list and in a list ordered by physical address. Both allocation and deallocation have time complexity of O(1) (MASMANO *et al.*, 2004).

Sun *et al.* proposed the TLSF-I, an improved version of the TLSF algorithm (SUN *et al.*, 2007). TLSF-I aims at improving the efficient in allocating small blocks and reduce fragmentation of the original TLSF algorithm. The authors used different strategies to allocate blocks of memory depending on their sizes. When the size of a block is small, TLSF-I uses exact-fit polity, instead of good-fit. To maintain an exact fit table, TLSF-I uses a two-level occupancy bitmap. When a memory block to be allocated is small, the first task is to get the map index of that size in order to verify if that block fits in a free list. If it does not, the algorithm searches in the bitmap tree to find an available block. In general, the TLSF-I performs better than TLSF algorithm and leads to a low fragmentation (SUN *et al.*, 2007).

Compact-FIT (CF) is a memory allocation/deallocation system that provides predictable memory fragmentation and response times that are constant or linear in the size of the memory request (CRACIUNAS *et al.*, 2008). Available memory in CF is partitioned in 16 KB pages. Each page is an instance of a size-class, which partitions a page further into same-sized page-blocks. Data is always allocated in a page of the smallest-size size class whose page-blocks still fit the data. The allocation of data larger than 16 KB is not supported. The main idea of CF is to keep the memory size-class compact at all times. At most one page of each size-class may be not-full at a given time while all other pages of the size-class are always kept full. When a memory is released, the data in the not-full page is moved to take its place and thus maintain the invariant (CRACIUNAS *et al.*, 2008). There are two implementations of CF, named moving and non-moving. In the moving CF implementation, page-blocks are mapped directly to physically contiguous pieces of memory, which requires moving data memory for compaction. In this implementation, allocation takes constant time and deallocation takes linear time if compaction occurs. In the non-moving implementation, a block table is used to map page-blocks into physical block-frames that can be located anywhere in memory. Compaction in this case is performed by re-programming the block table rather than moving data. Both allocation and deallocation in the non-moving implementation take liner time. The authors compared CF with TLSF in terms of fragmentation. CF has presented a more controlled and predictable fragmentation than TLSF. In terms of allocation/deallocation time, Half-fit and TSLF are faster than CF due to compaction activities.

A completely different approach for time analysis of dynamic memory allocation is proposed by Herter and Reineke (HERTER; REINEKE, 2009). The authors propose algorithms to compute static allocation for programs that use dynamic memory allocation. The algorithms strive to produce static allocations that lead to minimal WCET times in a subsequent WCET analyses (HERTER; REINEKE, 2009). The approach of transforming dynamic allocation to static allocation relies on further assumptions. For instance, all loop bounds and block sizes that are requested must be statically known (HERTER *et al.*, 2011). Good performance can only be achieved by this approach when the program's allocation behavior can be statically derived.

Puaut presented a performance analysis of several general purpose memory allocators (first-fit, best-fit, btree-best-fit, fast-fit, quick-fit, buddy-bin, and buddy-fibo) with respect to real-time requirements (PUAUT, 2002). The author compared the worst-case behavior obtained analytically with the worst timing behavior observed by executing real and synthetic workloads, considering the allocation and deallocation of memory blocks. The study has concluded that for applications with low allocation rates, the analytically worst-case allocation/deallocation times do not have an excessive impact on the application execution times for the most predictable allocators (buddy systems and quick-fit).

More recently, Masmano *et al.* compared the first-fit, best-fit, binary-buddy (KNUTH, 1997), DLmalloc (Lea, D., 1996), Half-fit, and TLSF memory allocators for real-time applications (MASMANO *et al.*, 2006). The results indicated that TLSF and Half-fit have a stable and bounded response time, which make them suitable for real-time applications. In contrast, algorithms designed to optimize average execution times, such as DLmalloc and binary-buddy, are not suitable for real-time applications. Moreover, Half-fit achieves bounded response times wasting more memory than TLSF (MASMANO *et al.*, 2006).

**Cache-aware and time-predictable allocators.** Cache-Aware Memory Allocator (CAMA) is the first memory allocator that combines constant time (predictability) with cache-awareness (HERTER *et al.*, 2011). Predictability is obtained by managing free blocks in segregated free lists, which allows for constant look-up times and hence constant response times. Internal fragmentation is reduced using a multi-layered segregated-list similar to TLSF's approach. Cache-awareness is obtained by adding an additional parameter to allocation requests: the first parameter is the original requested block size and the second and new parameter is the cache set that the block's memory address shall map to. Single segregated list contains all memory blocks within the same size and whose memory addresses map to the same cache set. CAMA uses cache-aware splitting and coalescing techniques to keep external fragmentation low. A bit vector for each cache set is responsible for signalizing that a block is empty or not. To provide guarantees about which cache sets may be accessed during allocation requests, CAMA stores descriptor blocks in the segregated lists, instead of free blocks directly. Each allocated block then stores a pointer to its descriptor block instead of a pointer to its appropriate free list. The free lists contain the descriptor blocks of free memory blocks instead of the free blocks themselves (HERTER *et al.*, 2011).

In this thesis we propose a memory allocation mechanism that overloads the C++ new and delete operators, enabling colored memory allocation/deallocation. This mechanism has the advantage of being non-intrusive (there is no need to create a new function as in CAMA) and to support user- and OS-centric approaches. We do not provide a worst-case timing behavior analysis of our memory allocator. However, in our experiments, we always allocate memory before the execution of the application. Hence, worst-case bounded time is not an issue. We detail our memory allocator in Chapter 4.

### 3.1.5 Cache-related Preemption and Migration Delay

Cache-related Preemption and Migration Delay (CPMD) is the delay caused by the loss of cache affinity after a preemption/migration. Basically, there are two ways of estimating CPMD: through offline static analyses and through online empirical experiments. Static analysis techniques estimate CPMD by analyzing the program code/data of the preempted and preempting tasks. Then, the analysis determines which data or instructions are reused after a preemption (NEGI *et al.*, 2003; STäRNER; ASPLUND, 2004; STASCHULAT; ERNST, 2005; YAN; ZHANG, 2008; HARDY; PUAUT, 2009; ALTMEYER *et al.*, 2012).

*Schedule-Sensitive* and *Synthetic* are two methods to measure CPMD through experimentation (BASTONI *et al.*, 2010a). In the first method, the system records the delays online by executing tests and collecting the measured data through the use of a TSC. The drawback of this method is the impossibility of controlling when a preemption or migration happens, which causes many useless data to be collected (BASTONI *et al.*, 2010a). The second method tries to overcome this problem by explicitly controlling preemptions and migration of a task, and thus measuring the delays. The evaluation shows that the CPMD in a system under load is only predictable for working set sizes that do not trash the L2 cache (BASTONI *et al.*, 2010a). It is possible to use a CPMD estimation approach together with a cache partitioning or locking mechanism (reviewed in the previously subsections). Thus, the estimation of the CPMD becomes easier, because of the isolation provided by those cache management strategies, which decrease intra-task, intra-core, and inter-core interferences.

In our work, we measure the CPMD by using hardware performance counters online through an RTOS in an 8-core modern processor. We use the obtained CPMD values to compare P-EDF, C-EDF, and G-EDF through the weighted schedulability metric (BASTONI *et al.*, 2010a). For high CPMD, P-EDF, C-EDF, and G-EDF tend to have the same performance. It is important to highlight that our objective in this work is not to propose a new method to measure the CPMD. We believe that CPMD is a complicated topic and still an open problem. Analyzing still-cached data after a short preemption and data cached in the several cache levels, including the effects of the coherence protocol on migrated data, is especially complex. On the one hand, specific methods relying on offline static analysis add a lot of constraints and/or requirements that may not be satisfied by all benchmarks, architectures, or applications used by industry. On the other hand, empirical experiments rely on a lot of assumptions, such as specific scheduling algorithms or the knowledge about the task sets, to perform the evaluation.

### 3.1.6 Summary

We now summarize the memory management mechanisms reviewed during this section. We present a comparative table for each of the following categories: index-based cache partitioning, way-based cache partitioning, cache locking, and memory allocators.

**Index-based cache partitioning.** Table 6 summarizes the discussed index-based cache partitioning works that rely on hardware-specific implementations. The works proposed by (LIU *et al.*, 2004), (SRIKANTAIAH *et al.*, 2008), (IYER, 2004), and (RAFIQUE *et al.*, 2006) although focused on multicore systems, were not proposed for real-time systems and, consequently, predictability is not the main concern. Instead, they aim at improving the QoS and the average execution time. SMART was the first hardware-based implementation of a cache mechanism designed to provide predictability for uniprocessors (KIRK; STROSNIDER, 1990). The SMART approach consists of dividing the cache into $M$ segments and assigning these segments individually to tasks. A cache location address is divided in a segment and cache ID fields that identify how many segments a task owns and which they are. Chousein and Mahpatra were the first authors to propose a hardware-based implementation of a cache partitioning mechanism to improve the predictability of multicore systems (CHOUSEIN; MAHAPATRA, 2005). Their approach focused on physical changes, using different technologies (CAM and TCAM cells) to implement cache memories. Suhendra and Mitra were the first authors to combine a cache partitioning mechanism, that can be implemented in hardware or software, with cache locking (SUHENDRA; MITRA, 2008). In their work, the cache partitioning mechanism performs task-based and core-based partitioning.

Table 7 summarizes the discussed index-based cache partitioning works that use a software approach, either implemented in the OS or in the compiler. The first works to use cache partitioning implemented in the OS did not focus on multicore real-time systems (TAYLOR *et al.*, 1990; KESSLER; HILL, 1992; ROMER *et al.*, 1994; SHERWOOD *et al.*, 1999). Instead, their objectives were to improve the average performance in uniprocessors. Also, several works improved the average performance in multicore systems (TAM *et al.*, 2007; LIN *et al.*, 2008; ZHANG *et al.*, 2009). Wolfe was the first author to use a cache partitioning mechanism to improve the predictability in uniprocessors (WOLFE, 1994). The work proposed by Liedtke et al. was the first to use page coloring, also in the

**Table 6: Comparative table of the reviewed state-of-the-art mechanisms of index-based cache partitioning with hardware-specific implementations.**

| Cache partitioning | Features | | | | | |
|---|---|---|---|---|---|---|
| | **Proposed to RTS?** | **Inst. or data caches** | **HW or SW?** | **Technique** | **Use cache locking?** | **Multicore or singlecore?** |
| (LIU *et al.*, 2004) | no | both | HW | cache controller table | no | multicore |
| (SRIKANTAIAH *et al.*, 2008) | no | both | HW | POP cache | no | multicore |
| CQoS (IYER, 2004) | no | both | HW | selective cache alloc. | no | multicore |
| (RAFIQUE *et al.*, 2006) | no | both | HW | quota enforcement | no | multicore |
| SMART (KIRK; STROSNIDER, 1990) | yes | data | HW | M segments | no | singlecore |
| (CHOUSEIN; MAHAPATRA, 2005) | yes | data | HW | CAM and TCAM cells | no | multicore |
| (SUHENDRA; MITRA, 2008) | yes | data | HW | HW or SW partitioning | yes | multicore |

context of uniprocessor real-time systems (LIEDTKE *et al.*, 1997). Bui et al. proposed a genetic algorithm to optimize the worst-case system utilization by finding the best cache partitioning assignment for a task set (BUI *et al.*, 2008). Kim et al. used page coloring to propose a cache management scheme that assigns to each core a set of partitions (KIM *et al.*, 2013). Kenna et al. proposed the cache management strategy called $MC^2$, which treats the management of cache lines as scheduling and synchronization problems (KENNA *et al.*, 2013). All the discussed works so far used an OS implementation of a cache partitioning mechanism based on page coloring. Other works proposed changes in the compiler to support cache partitioning at the compile time. Mueller was the first author to introduce a compiler support for cache partitioning in the context of uniprocessor real-time systems (MUELLER, 1995). Bugnion et al. extended the Mueller's approach and implemented page coloring support in the SUIF parallelizing compiler targeting multicore processors (BUGNION *et al.*, 1996). Both Bugnion et al and Mueller's approaches used page coloring to partition the cache. Finally, Vera et al. used compiler techniques together with cache partitioning and locking to improve the predictability in preemptive multitasking uniprocessor systems in the presence of data caches (VERA *et al.*, 2003b). The mechanism relies on a hardware or software cache partitioning and was the first compiler-based mechanism to combine cache partitioning with cache locking.

In this work we design and implement a cache partitioning mechanism based on page coloring in EPOS, which allows the assignment of individual cache partitions to the internal OS data structures. We use this mechanism to evaluate how the RTOS affects the execution of real-time tasks and how different scheduling algorithms benefit from cache partitioning.

**Way-based cache partitioning.** All the reviewed way-based cache partitioning mechanisms use a hardware-specific implementation. They are summarized in Table 8. None of the way-based cache partitioning works were originally proposed to improve the predictability of real-time systems, although some of the proposed techniques could be adapted in the context of SRT systems.

**Cache locking.** Table 9 summarizes the discussed cache locking mechanisms. In (ASADUZZAMAN *et al.*, 2010) and (SARKAR *et al.*, 2011), two hardware-only cache locking mechanism are proposed. The first targets both multicore and single-core systems, while the second

**Table 7: Comparative table of the reviewed state-of-the-art mechanisms of index-based cache partitioning implemented in software.**

| Cache partioning | Features | | | | | |
|---|---|---|---|---|---|---|
| | **Proposed to RTS** | **Inst. or data caches** | **HW or SW** | **Technique** | **Use cache lock.** | **Multicore or singlecore** |
| (TAYLOR *et al.*, 1990; KESSLER; HILL, 1992; ROMER *et al.*, 1994; SHERWOOD *et al.*, 1999) | no | data | SW-OS | page coloring | no | singlecore |
| (TAM *et al.*, 2007; LIN *et al.*, 2008; ZHANG *et al.*, 2009) | no | data | SW-OS | page coloring | no | multicore |
| (WOLFE, 1994) | yes | data | SW-OS | similar to page coloring | no | singlecore |
| (LIEDTKE *et al.*, 1997) | yes | data | SW-OS | page coloring | no | singlecore |
| (BUI *et al.*, 2008) | yes | data | SW-OS | genetic alg. with page col. | no | singlecore |
| (KIM *et al.*, 2013) | yes | data | SW-OS | page coloring | no | multicore |
| (KENNA *et al.*, 2013) | yes | data | SW-OS | page coloring | no | multicore |
| (MUELLER, 1995) | yes | both | SW-compiler | compiler with page col. | no | singlecore |
| (BUGNION *et al.*, 1996) | no | both | SW-compiler | CDPG | no | multicore |
| (VERA *et al.*, 2003b) | yes | data | SW-compiler | HW or SW cache partitioning | yes | singlecore |

**Table 8: Comparative table of the reviewed state-of-the-art way-based cache partitioning mechanisms.**

| **Cache partitioning** | **Features** | | | | | |
|---|---|---|---|---|---|---|
| | **Proposed to RTS** | **Inst. or data caches** | **HW or SW** | **Techn.** | **Use cache lock.** | **Multi-core or single-core** |
| (RANGANATHAN *et al.*, 2000) | no | both | both | dynamic reconfiguration | no | singlecore |
| (CHEN *et al.*, 2009) | no | data | HW | quota enforcement | no | multicore |
| RECAP (SUNDARARAJAN *et al.*, 2013) | no | data | HW | specific mem. arch. | no | multicore |
| (VARADARAJAN *et al.*, 2006) | no | data? | both | molecular cache | no | multicore |
| (QURESHI; PATT, 2006) | no | data | HW | utility-based approach | no | multicore |

is proposed only for multicore systems. The rest of the works does not propose a hardware-specific mechanism to lock the cache. Instead, they use the cache locking mechanisms available on current processors. Campoy *et al.* were the first to study the effects of locking lines in instruction caches of uniprocessor real-time systems (CAMPOY *et al.*, 2001). The authors proposed a genetic algorithm to select the best lines to be locked. Following the same research line, in (PUAUT, 2002) the authors proposed two algorithms to also select best lines to be locked in an instruction cache. In (VERA *et al.*, 2003a) and (FALK *et al.*, 2007), the authors proposed changes in the compiler to extract information regarding the data/instruction access pattern by tasks. Then, this information is used to lock cache lines. Liang and Mitra proposed a profiling technique to extract the instruction access pattern of an application (LIANG; MITRA, 2010). Then, a cache locking algorithm analyzes the pattern and chooses which cache lines should be locked. Although not proposed for real-time systems, similar profiling techniques are used by WCET estimation tools. Suhendra and Mitra (this work was discussed in Section 3.1.1) were the first authors to evaluate the combination of cache partitioning and cache locking in the context of multicore real-systems (SUHENDRA; MITRA, 2008). Specifically to cache locking, the authors used the algorithms proposed in (PUAUT, 2002) to select the cache lines to be locked. Unlike this work, Mancuso *et al.* were the first authors to evaluate the combination of cache partitioning and locking using a real hardware and OS in the context of multicore real-time systems.

In this work we use cache partitioning to reduce inter-core interference. A cache locking mechanism is a complementary approach to cache partitioning, but it requires a hardware support that ist not present in many of the modern multicore processors.

**OS memory allocators.** Table 10 summarizes the OS memory allocators that are time-predictable or/and cache-aware. ccmalloc and CIF do not have real-time systems as the main concern, although support page coloring and index-based cache management techniques, respectively. The dynamic to static allocation translation has the disadvantage of requiring additional assumptions about the program's allocation behavior, such as the size of memory allocation requests and loop bounds. Half-fit, TLSF, and TLSF-I provide a bounded time for allocation and deallocation activities. TLSF has lower fragmentation than Half-fit. TLSF-I improves the TLSF allocation for small memory blocks and reduces its fragmentation. CF was designed to have a low fragmentation rate due to an implemented compaction mechanism. Thus, CF has lower fragmentation than TLSF and Half-fit. However, CF is slower than TLSF and Half-fit for allocation/deallocation tasks. Finally, CAMA is the only memory allocator that provides a cache memory technique together with bounded allocation/deallocation time behavior. Our approach, the C++ new operator overload technique, does not provide a worst-case timing behavior analysis, but is the first work to provide page coloring allocation for internal OS memory pages. We detail our memory allocator in Chapter 4.

## 3.2 REAL-TIME OPERATING SYSTEMS

Currently, a huge variety of RTOSes is being developed and used in the embedded system area. For instance, the (incomplete) Wikipedia's list of RTOSes has about one hundred active projects (WIKIPEDIA, 2014a). The main reason for this diversity is the minimal set of features required by some embedded systems. Because of that, small run-time

**Table 9: Comparative table of cache locking state-of-the-art mechanisms.**

| **Cache locking** | **Features** | | | | | |
|---|---|---|---|---|---|---|
| | **Proposed to RTS** | **Inst. or data caches** | **HW-only** | **HW-SW** | **SW approach** | **Multicore or singlecore** |
| (ASADUZZAMAN *et al.*, 2010) | yes | both | yes | no | - | both |
| (SARKAR *et al.*, 2011) | yes | both | yes | no | - | multicore |
| (CAMPOY *et al.*, 2001) | yes | instruction | no | yes | genetic algorithm | singlecore |
| (PUAUT, 2002) | yes | instruction | no | yes | two alg. to select locked cache lines | singlecore |
| (VERA *et al.*, 2003a) | yes | data | no | yes | compile time alg. | singlecore |
| (FALK *et al.*, 2007) | yes | instruction | no | yes | compile time alg. | singlecore |
| (LIANG; MITRA, 2010) | no | instruction | no | yes | profile and locking alg. | singlecore |
| (APARICIO *et al.*, 2011) | yes | instruction | no | yes | ILP method | singlecore |
| (SUHENDRA; MITRA, 2008) | yes | both | no | yes | cache partitioning and locking | multicore |
| (MANCUSO *et al.*, 2013) | yes | both | no | yes | profiling, cache partitioning and locking | multicore |

**Table 10: Comparative table of OS memory allocators.**

| **Memory Allocator** | **Features** | | | |
|---|---|---|---|---|
| | **Proposed to RTS** | **Time-predictable** | **Cache-aware** | **Cache management technique** |
| ccmaloc (CHILIMBI *et al.*, 2000) | no | no | yes | clustering and page coloring |
| CIF (AFEK *et al.*, 2011) | no | no | yes | index-based |
| Dynamic translation (HERTER; REINEKE, 2009) | yes | yes | no | - |
| Half-fit (OGASAWARA, 1995) | yes | yes | no | - |
| TLSF (MASMANO *et al.*, 2004) | yes | yes | no | - |
| TLSF-I (SUN *et al.*, 2007) | yes | yes | no | - |
| CF (CRACIUNAS *et al.*, 2008) | yes | yes | no | - |
| CAMA (HERTER *et al.*, 2011) | yes | yes | yes | index-based |
| **Overload of the C++ new and delete operators** | **yes** | **no** | **yes** | **page coloring** |

environments are designed and implemented for specific embedded systems, and are called as “RTOS”. In fact, a 2013 embedded market study has pointed out interesting facts about the development and usage of RTOSes (UBM, 2013):

- 68% of the current embedded system projects use an OS, RTOS, kernel, software executive, or scheduler of any kind. The majority of the other 32% does not use an RTOS or kernel just because the project does not need it;

- 68% of the current embedded system projects have real-time capabilities;

- 12% of the current embedded system projects use a new or different OS/RTOS;

- Considering only those projects that use an RTOS, 35% use a commercial version, 34% use an open-source version, 19% use an internally developed or in-house version, and 12% use a commercial distribution of an open-source version. Real-time capability is the most important feature that influences developers in choosing a commercial RTOS;

- 24% of the current embedded system projects use an in-house or custom RTOS version, that is, real-time embedded developers prefer to design and implement their on RTOSes. The FreeRTOS, for instance, is used by 13% of the projects and is in the fourth place of the list.

The embedded system market study ensures that there is a proliferation of embedded RTOSes, some of them with the same characteristics and design choices. However, the majority of RTOSes of this kind do not support multicore processors and thus are not relevant for this thesis. Nevertheless, even if we consider those RTOSes with any type of shared-memory multiprocessor support, a large number of RTOSes still remains.

**Interrupt latency.** Interrupt latency is an important characteristic or metric commonly used to discuss and compare RTOSes. The interrupt latency describes how quickly the RTOS reacts to interrupts. If an OS disables interrupts or suppresses calls to the scheduler for long periods, it tends to have high interrupt latency (BRANDENBURG, 2011). From the schedulability point of view, the interrupt latency is a blocking time for a job, because the job that should be scheduled is delayed until the interrupt is handled. The release overhead described in Section 2.5 includes the interrupt latency in the schedulability analysis.

In the following subsections we present an overview of current multiprocessor-capable RTOSes. We group similar RTOSes into categories and for each category we present a description of the main RTOSes. A survey of RTOSes for uniprocessors and distributed-memory multiprocessors can be found in (STANKOVIC; RAJKUMAR, 2004).

### 3.2.1 Embedded Real-Time Operating Systems

The main objective of an embedded RTOS is to minimize resource usage to fit the system in a resource-constrained platform, which is usually used by embedded systems. In this category, the OS kernel and tasks execute in the same address space. Thus, a task can access a device without any restriction, maximizing the performance.

Multiprocessor support is rare in this category, but does exists, often in the form of message passing (each processor sends and receives messages to/from another processor) (BRANDENBURG, 2011). The scheduling approach used in this case is the partitioned scheduling. Global access to resources is usually not supported. Typical RTOS support for this category includes FP scheduling, mutex, and semaphores with optional priority inheritance (BRANDENBURG, 2011).

Commercial embedded RTOSes with multiprocessor support include RTXC/mp (message passing, preemptive round-robin scheduling, fixed and dynamic priorities, and priority inheritance) (QUADROS, 2005), ThreadX (preemptive FP scheduling with 32 priorities and priority inheritance) (EXPRESS, 2013), Nucleus RTOS (preemptive FP partitioned scheduler with 1024 priority levels and priority inheritance) (MENTOR, 2013), and OSE (FP partitioned scheduler with message passing) (ENEA, 2013).

RTEMS (Real-Time Executive for Multiprocessor Systems) is the most representative open-source embedded RTOS that supports multiprocessors (RTEMS, 2013). RTEMS provides a FP scheduling with 256 priority levels and do not support the EDF dynamic policy. The RTEMS kernel and the user applications are compiled together into one binary image for each processor. Thus, RTEMS is also able to run in a distributed system. In each processor, there is a *multiprocessing server*, which is responsible for handling requests from all remote tasks (tasks that run in other processors). The RTOS also supports priority inheritance protocols, such as the Stack Resource Policy (SRP).

Other embedded RTOSes focus on fulfilling safety and security requirements for specific areas, such as automotive and avionics systems. These RTOSes are designed to isolate software components to avoid that an error in one component be propagate to other components. Hence, it is possible to certify the RTOS in terms of safety and security features. Due to the certification process, this type of embedded RTOS generally has low complexity and implements few kernel services, such as the ciclic executive scheduler. Multicore support on this type of RTOS is usually absent, so we do not review this class in details. An example of a kernel used in the Airbus A380, Boeing 777, and Boeing 787 aircrafts is the Green Hill's INTEGRITY kernel (Green Hills Software, 2011).

Research embedded RTOSes have also been proposed in the last years, although most of these academic projects focused on uniprocessors and distributed systems. One of the first research RTOS to provide multicore support was the Spring OS (STANKOVIC; RAMAMRITHAM, 1991). The Spring kernel was the first to use a dedicated processor to handle interrupts. In a dedicated interrupt handling scheme, one processor (named system processor) is responsible for handling interrupts, while the remaining processors execute the user processes. The system processor is also responsible for releasing tasks and making scheduling decisions. This technique is called *interrupt shielding*, which prevents real-time tasks to be disturbed by interrupts and OS overheads (BRANDENBURG, 2011). The result is a more predictable OS. Another research RTOS is the Fiasco microkernel (HÄRTIG; ROITZSCH, 2006). Recent Fiasco versions support FP partitioned scheduling. Also, interrupts are not processed in the kernel space. Instead, they are sent to user space processes through IPC messages, which handle the interrupts accordingly. This technique is known as *split interrupt handling* and allows interrupt to be safely deferred by assigning low priorities to the "driver" processes (BRANDENBURG, 2011).

The OS used in this thesis, EPOS, can be classified as an embedded RTOS. EPOS originally did not supported real-time multicore scheduling algorithms. In Chapter 4, we present the design and implementation of the real-time multicore support for EPOS, which enabled it to be the first open-source RTOS to provide FP and dynamic partitioned, clustered, and global schedulers in a shared-memory multiprocessor.

### 3.2.2 POSIX-Like Real-Time Operating Systems

This category encompasses RTOSes that implement the complete or parts of the POSIX-API and match the expectations of the POSIX real-time profiles. The Portable Operating System Interface (POSIX) is a set of standards created to maintain the compatibility of

OSes. POSIX also defines a set of real-time extensions, which includes periodic timers, FP scheduling, semaphores, and thread interface (IEEE, 2003). Like a General-Purpose OS (GPOS), a POSIX-like RTOS provides high-level functionalities, such as process and threads, dynamic process creation, filesystems, full TCP/IP communication protocol stack, and so on. The most important RTOSes of this category are the Research in Motion's QNX Neutrino and WindRiver's VxWorks. We briefly discuss both RTOSes below.

QNX Neutrino is the first POSIX-like RTOS to support multiprocessors (since 1997) (QNX, 2013). QNX Neutrino is a microkernel fully compliance with the POSIX standard and requires an MMU to work properly and to have address space separation between applications and kernel. As every POSIX-like OS, QNX Neutrino has 256 priority levels, FP scheduling (if priorities are equals, two tasks are queued in FIFO order or round-robin – SCHED_FIFO and SCHED_RR in the POSIX nomenclature). The RTOS provides multiprocessor support through *processor affinity masks*. Tasks have an affinity mask, which indicates in which processor they may execute. Thus, QNX Neutrino implements a FP scheduling with processor affinity: when a real-time task is ready to execute, all processors on which it may execute (given by its affinity) are checked sequentially, and if there is an idle processor or a lower priority task, the task is resumed on that processor. If there is no lower priority task, the newly task is enqueued in the ready queue until one of the processors becomes available. The processor affinity scheme can be used to implement clustered scheduling, and hence also global and partitioned scheduling (depending on the cluster configuration, see Section 2.4.2.3) (BRANDENBURG, 2011). QNX Neutrino does not support any dynamic scheduling policy, such as EDF and LLF.

VxWorks is another well-known POSIX-like RTOS that supports SMP systems (Wind River, 2013). The real-time multicore scheduling in VxWorks is similar to the QNX Neutrino. Tasks have affinity masks and are scheduled only on those processors that are in the affinity mask. The scheduler uses FP and is preemptive. Also, the RTOS supports mutual exclusion for tasks running on different processors. VxWorks is able to assign and configure specific processor to handle interrupts.

Both QNX Neutrino and VxWorks are popular in the real-time embedded system domain and are not used in systems that face severe resource limitations (*i.e.,* deeply embedded systems) or requires particular certification requirements. The 2013 embedded system market states that VxWorks is used in 7% of all embedded system projects, while QNX is the choice in 2% of the projects (UBM, 2013).

### 3.2.3 General-Purpose Operating Systems with Real-Time Extensions

There is a trend in the academic real-time community nowadays to, instead of building an RTOS from scratch, use a GPOS as base and to add real-time features to it (*e.g.,* real-time schedulers, memory partition, or lowered interrupt latency) (BRANDENBURG, 2011). The most advantages of this technique are the reduced development time and the wide support for different multicore architectures. The disadvantages are the inherent non real-time behavior of such GPOSes, which impact the real-time tasks that are executed, and the "fixed" design choices, which affect the range of applicable real-time extensions. Also, there is the requirement of minimal resources; some embedded systems may not have them and thus may not be able to execute the GPOS.

The most common base GPOS with real-time extensions is Linux. Dozens of commercial and academic "real-time Linux" exist. They can be divided into two groups: native kernels and para-virtualized kernels (BRANDENBURG, 2011). In a native kernel, the Linux kernel is the only present and it is responsible for providing real-time features. In this group, real-time tasks are treat as regular Linux processes. In a para-virtualized Linux kernel, there is a real-time microkernel or hypervisor between the hardware and the original Linux kernel. The real-time microkernel has a higher priority than the original kernel (*i.e.,* it is always executed before the original kernel), and is responsible for scheduling real-time tasks (not Linux processes) and to handle interrupts. Traditional Linux processes are executed in the original kernel. In the following, we classify the related work in either native or para-virtualized real-time Linux extensions and discuss the main projects in each category.

**Para-Virtualized Real-Time Linux Extensions**. All stable Linux versions prior to 2.4 executed every system call and interrupt with interrupts disabled, that is, as a long non-preemptive section. In the context of real-time systems, long non-preemptive sections insert unacceptable blocking time. Thus, real-time extensions applied to early Linux versions have chosen to design a work around to this problem and to place a microkernel or a hypervisor between the hardware and the Linux kernel. The Linux kernel is executed as a task of the real-time microkernel or hypervisor and does not have full control of the hardware and can be preempted at any time. The advantages of this design are the low interrupt latency and the relative small changes required in the Linux kernel (BRANDENBURG, 2011). The limitation is that real-time tasks cannot access Linux services nor directly interact with Linux processes. However, some para-virtualized kernels, such as the RTLinux (BARABANOV, 1997), enable the communication between real-time tasks and Linux processes by using non-blocking queues and buffers. In summary, para-virtualized approaches try to co-exist with the original Linux kernel, without modifying it. The provided real-time support encompasses P-FP scheduling with priority inheritance protocols. Besides the RTLinux, two other well-known para-virtualized real-time Linux extension are the Real-Time Application Interface (RTAI) (CLOUTIER *et al.*, 2000), which focuses on industrial applications, and Xenomai, which is also used by industrial applications but has a compatibility layer for legacy applications (GERUM, 2008).

**Native Real-Time Linux Extensions**. A native real-time Linux extension directly modifies the Linux kernel to enhance its real-time capabilities. Usually, the modifications are applied to the kernel through a patch or the modified source code is made available to download. Kansas University Real-Time Linux (KURT Linux) was one of the first projects to introduce real-time capabilities to Linux kernel (SRINIVASAN *et al.*, 1998). KURT Linux introduced the concept of high resolution software timers, which is implemented in current Linux versions as *hrtimers* (GLEIXNER; NIEHAUS, 2006). KURT Linux also supported cyclic executive HRT schedulers, the load of static scheduler as modules at run-time, and the ability to only schedule real-time tasks (no background Linux processes). The last KURT Linux patch was released in 2002 for the Linux version 2.4.18.

Advanced Real-Time Linux (ART Linux) is a real-time Linux for robotics applications (ISHIWATA; MATSUI, 1998). ART Linux has been used in the Open Humanoid Robotics Platform and humanoid robot HRP-4C (KANEKO *et al.*, 2009). According to Brandenburg (BRANDENBURG, 2011), ART Linux is not well-documented nor well-known by the real-time community. The Real-Time and Embedded Linux (RED Linux) is a pioneer real-time patch for Linux to provide a flexible infrastructure to design and implement uniprocessor real-time schedulers (WANG; LIN, 1998). The RED project did not target multiprocessors and it seems to be closed.

Oikawa and Rajkumar proposed the Linux/Resource Kernel (Linux/RK), which is an implementation of the Portable RK in the Linux kernel (OIKAWA; RAJKUMAR, 1999). Portable RK is a framework that abstracts the resource management (*e.g.,* CPU, network, and disk bandwidth reservation) from a specific OS. Moreover, Portable RK provides an API to users, thus a developer when porting the framework for a different OS, only needs to deal with the differences in the OS APIs (OIKAWA; RAJKUMAR, 1999). However, for implementing the Portable RK in the Linux kernel, the authors inserted a number of *callback hooks* to send relevant scheduling events to the Portable RK framework. The authors measured several run-time overhead sources in the framework, but did not provide a comparison with other Linux-based real-time implementations. Recent research topics in Linux/RK include multicore processors, integration with real-time java, and resource reservation strategies. The last Linux/RK version was released in early 2007.

Other real-time patches focused on improving the QoS and providing SRT support in the Linux kernel. QLinux (SUNDARAM *et al.*, 2000), Linux-SRT (CHILDS; INGRAM, 2001), and Adaptive Quality of Service Architecture (AQuoSA) are examples of this type of patch. QLinux provides a resource reservation scheme for the processor, and network and disk bandwidth. Linux-SRT provides several resource reservations schemes similarly to QLinux. AQuoSA is a patch for Linux that allows the interception of scheduling events and consequently, the implementation of external schedulers (PALOPOLI *et al.*, 2009). A resource reservation module implements different algorithms, such as (CBS), IRIS and GRUB. The module also implements an EDF scheduler to manage the internal queues.

Several real-time patches for Linux have been proposed recently (for Linux 2.6 or superior). PREEMPT_RT modifies Linux to make it more deterministic and reduce the average of scheduling latency (FU; SCHWEBEL, 2014). SCHED_DEADLINE is an implementation of a real-time scheduling class using the Linux scheduling class mechanism (FAGGIOLI *et al.*, 2009). The scheduling class supports EDF (P-EDF, C-EDF, and G-EDF) and CBS (ABENI; BUTTAZZO, 2004) algorithms to provide temporal isolation among tasks. Furthermore, SCHED_DEADLINE uses the control groups (*cgroups*) API to natively support multicore platforms and hierarchical scheduling.

ExSched is a scheduler framework that enables the implementation of different (real-time) schedulers as external plugins without modifying the OS (ÅSBERG *et al.*, 2012). ExSched hides all OS dependencies, which allows different OSes to reuse the same plugin implementation. An API is responsible for all communication between user applications and the target OS, incurring in more execution time overhead. The authors implemented some multicore real-time scheduling algorithms, such as partitioned and global FP schedulers, and compared the run-time performance of ExSched multicore implementations to the SCHED_FIFO Linux scheduler in terms of schedulability ratio of generated task sets. For some cases, FP schedulers in ExSched had worse performance than the Linux SCHED_FIFO scheduler.

Advanced Interactive Real-time Scheduler (AIRS) is another real-time extension for Linux designed on top of SCHED_DEADLINE scheduling class (KATO, 2012). AIRS increases the system performance when multiple interactive real-time applications, like multimedia applications, run on a multicore processor. AIRS proposes two new concepts: Flexible CBS (F-CBS) that improves the CPU bandwidth reservation and the Window-constrained Migration and Reservation (EDF-WMR) scheduler that improves the absolute CPU bandwidth available for multicore real-time applications (KATO, 2012). In a comparison carried out by the authors, AIRS delivered higher quality to simultaneous multiple videos than the existing real-time Linux extensions (*e.g.,* SCHED_DEADLINE, Linux/RK, and $LITMUS^{RT}$) (KATO, 2012).

### 3.2.4 LITMUS$^{\mathbf{RT}}$

**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems (LITMUS$^{RT}$) is a native real-time extension to Linux. Due to its importance for this work (it is used in the comparative experiments[2]), LITMUS$^{\text{RT}}$ is described in more detail in a separated section. LITMUS$^{\text{RT}}$ is implemented as a plugin (*i.e.,* a patch) and allows different multiprocessor scheduling algorithms to be implemented as plugin components using the available Linux infrastructure (CALANDRINO *et al.*, 2006). The current LITMUS$^{\text{RT}}$ version (2013.1) is based on Linux

[2] We choose $LITMUS^{RT}$ to compare our work, because it has been used in several research work recently and has an activity community.

3.10.5 and supports G-EDF, P-EDF, C-EDF, P-FP, and PFair scheduling algorithms.

Figure 31 shows an overview of the LITMUS$^{\mathrm{RT}}$ architecture. It has four main parts: core infrastructure, scheduler plugins, userspace interface, and userspace library and tools. The core infrastructure adds a new and highest priority scheduling class to the traditional Linux scheduler. As a consequence, LITMUS$^{\mathrm{RT}}$ always executes the highest priority jobs in advance of the Linux original scheduler (BRANDENBURG *et al.*, 2008). The core infrastructure also changes the Linux scheduler to invoke the plugin initialization functions, scheduling and tick handlers at run-time. At each timer interrupt interval (1 ms), the processor invokes the tick handler. Linux invokes the scheduler handler at every scheduling decision to select the next task to be executed (BRANDENBURG; ANDERSON, 2009). The original Linux scheduler is not modified and it is called when the LITMUS$^{\mathrm{RT}}$ plugin is disabled.

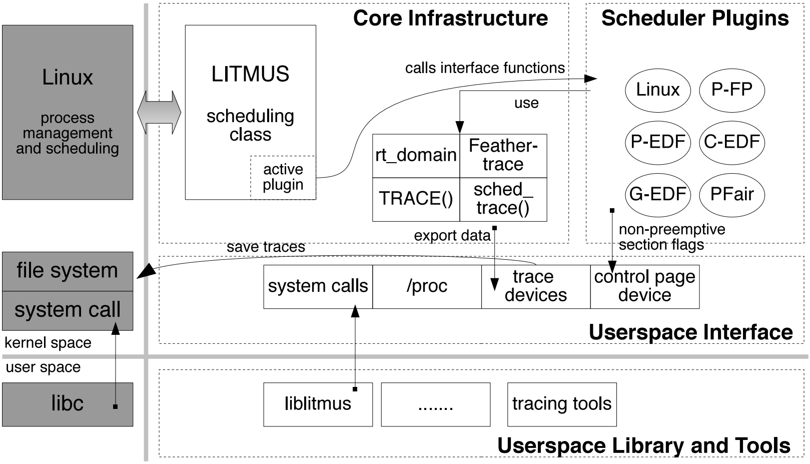


**Figure 31: Digram of the LITMUS$^{\mathrm{RT}}$ main components. Adaptation from (BRANDENBURG, 2011).**

To ease the development of new scheduling policies, LITMUS$^{\mathrm{RT}}$ implement a scheduler plugin interface, which allows the implementation of new scheduling policies without the need to know the complete Linux scheduling infrastructure. The developer chooses a scheduler plugin before running real-time tasks. This scheduler plugin becomes the active plugin, which will receive the forwarded kernel messages. The interface has 13 functions, such as *schedule* to select the next process to be scheduled, and *tick* to trigger the scheduler.

The userspace interface adds system calls and several virtual character devices. Among the system calls, there are calls to get and set task parameters and processor assignments, job control for real-time processes, and measurement of system calls overhead (BRANDENBURG, 2011).

The *liblitmus*, a userspace library, provides an API to create and control real-time tasks. Initially, tasks are created as non real-time and specific liblitmus functions are called to initialize real-time settings and per-task data structures and put a task in real-time mode. Additionally, LITMUS$^{\mathrm{RT}}$ also provides a tool, named Feather-Trace (BRANDENBURG; ANDERSON, 2007), to collect and store events at run-time, such as scheduling and release overheads. Feather-Trace adds the smallest possible interference to the system. The tracing tool implements a wait-free (no locks) FIFO-buffer to store events, which are collected by the TSC, and is multiprocessor safe (*i.e.,* allows multiple writers). There is an addition of only one instruction to check if a traced event is enabled or not. If an event is disabled, then only one instruction is added compared to the case when there is no tracing code. The total code size for obtaining and storing a timestamp in Feather-Trace is 61 instructions (for instance, about 0.017 µs in the platform used in our experiments – Intel i7-2600 – assuming 1 cycle per instruction) (BRANDENBURG; ANDERSON, 2007).

### 3.2.5 Summary

Generally, all the RTOSes reviewed in this sections that support multicore processors are either POSIX-like or GPOS with real-time extensions. Few embedded RTOSes are designed for multicore processors. Among those that have multicore support, the majority is proprietary and difficult to get more details about their design and implementation.

The real-time community tends to use a GPOS with real-time extensions. All of "real-time Linux" projects suffer from the inherent non real-time aspect of Linux, higher run-time overhead (mainly the OS-independent frameworks), and are tightly coupled to the Linux infrastructure. Global JLFP or JLDP schedulers are supported only by SCHED_DEADLINE and LITMUS$^{\mathrm{RT}}$ patches. The support for memory management mechanisms, such as cache partitioning, on those works are limited by the Linux API (*i.e., brk* system call) and difficult to use. Differently, our work shows how an RTOS designed from scratch allows better code reuse, smaller run-time overhead, and easier memory management mechanisms than Linux-based approaches. With the real-time support introduced by this work, EPOS is the first RTOS designed from scratch to support partitioned and global FP, JLFP, and JLDP multicore schedulers.

## 3.3 RUN-TIME PERFORMANCE MONITORING

Run-time performance monitoring through the use of HPCs has received much more attention in recent years, mainly because PMUs are offering a vast number of events that can be monitored and are useful while performing online optimizations. Next, we review performance monitoring APIs and tools in Section 3.3.1, and the main works that use HPCs to improve OS decisions at run-time in Section 3.3.2.

### 3.3.1 Performance Monitoring APIs and Tools

The Performance API (PAPI) is the most used open source cross-platform interface for HPCs (MUCCI *et al.*, 1999; DONGARRA *et al.*, 2003). PAPI consists of a machine-dependent layer, a high-level interface that provides access to read, start, and stop counters, and a low-level interface that provides additional features. In this work, we design and implement an API to manipulate HPCs. It is not our target, however, to make a comparison between PAPI and our implementation. PAPI supports a wide range of platforms, was designed for general-purpose computing systems, and has been under development for more than 10 years. Instead, our interface is designed for embedded applications, which usually have their requirements known at design time. Thus, it is possible to generate only the needed code for the application and nothing else.

Linux abstracts the usage of HPCs through a performance counter subsystem. A performance monitoring tool (*perf*) uses this subsystem and allows developers to obtain and analyze the performance of their applications. The tool also supports a command to record data into files which can be later analyzed using a *report* command. Other tools such as Intel Vtune (MALLADI, 2010) and AMD CodeAnalyst (DRONGOWSKI, 2008) offer a friendly interface to monitor performance of processors and applications through the use of HPCs. However, embedded systems usually do not have such a control interface.

### 3.3.2 Run-Time OS Decisions

Several works use HPCs as an alternative to easily detect sharing pattern among threads and help scheduling decisions in multicore processors. Bellosa and Steckermeier were the first to suggest using HPCs to dynamically co-locate threads onto the same processor (BELLOSA; STECKERMEIER, 1996). Weissman uses HPCs to guide thread scheduling in SMPs (WEISSMAN, 1998). His method is based on a model of the shared cache state that takes as input the number of cache misses, measured by HPCs during a scheduling quantum, and a graph representing the current shared state dependencies among threads. This graph is induced by users annotations in the source code. The objective is to compute the effects of all threads' working sets in the cache of a processor. Based on the model, two scheduling policies are proposed. Nevertheless, they do not consider the invalidation effects of data sharing between threads running in different processors and do not provide real-time guarantees.

Tam *et al.* use HPCs to monitor the addresses of cache lines that are invalidated due to cache coherence activities and to construct a summary data structure for each thread, which contains the addresses of each thread that are fetching from the cache, for ccNUMA architecutres (TAM *et al.*, 2007). The scheduling mechanism is divided in four phases. First, there is an association between the processor stall time with the reasons that caused the processor to stall. The average number of cycles per instructions of an application is split in completed cycles (*i.e.,* at least one instruction was completed) and cycles that cause the processor to stall (*i.e.,* cycles that none instruction is completed). The cycles that cause the processor to stall are subdivided according to a specific reason, such as misses in the Translation Lookaside Buffer (TLB) and data cache misses. The data cache misses are divided in their sources: *local*, representing the access to the cache carried out by a core within the same chip, and *remote*, when there is a cache access in another chip. In the second phase, when the number of remote accesses reaches 20% of one billion of cycles, the scheduling mechanism detects data sharing among threads. The cache lines that are invalidated due to the cache coherence protocol are monitored and the summary data structure for each thread is built. The detection of shared cache lines was implemented in the IMB Power5 processor, which allows the collection of the last address that caused a L1 data cache miss. Based on the information from this data structure, the scheduler builds a cluster composed of a set of threads, and allocates a cluster to a specific core. The last phase is ordering the groups according to the amount of data. Thus, the group that has more data is allocated in the core that has less threads. The main drawback of this method is the dependency of a specific hardware event to detect the last address that generated a L1 data cache miss, which is not available in most of the current processors. Moreover, this approach does not provide real-time guarantees.

Tam *et al.* extended the described work to support cache partitioning and detection of cache regions with bad performance (AZIMI *et al.*, 2009). The used cache partitioning method was page coloring. HPCs determine the size of each partition at run-time by using the RapidMRC technique, which uses the cache miss rate to estimate the memory consumption of threads (TAM *et al.*, 2009). The cache miss and hit rates are used to build a map showing the access pattern to pages. Based on this map, pages that are less accessed are mapped to other cache regions (using page coloring).

West *et al.* (WEST *et al.*, 2010) proposed an online technique based on a statistical model to estimate per-thread cache occupation online through the use of HPCs. The main contribution of this work is the monitoring of cache hit numbers that represent the access frequency to cache lines. Thus, the authors achieved a more precise estimation using associative caches that implement the LRU cache line replacement policy. However, data sharing among cores and real-time guarantees are not considered by the authors.

Muralidhara *et al.* proposed a dynamic software-based partitioning technique that distributes the shared cache space among threads of a given application (MURALIDHARA *et al.*, 2010). At the end of each 15 ms interval, the dynamic cache partitioning scheme uses hardware performance counters, such as cache hit/misses, cycle, and instruction counts for each thread, to allocate different cache regions based on individual thread performance. The objective is to speed up the critical path, that is, the thread that has the slowest performance and, consequently, improve the overall performance of the application. Cache partitioning is performed either by using reconfigurable caches, where the cache hardware structures are modified at run-time, or by modifying the cache replacement algorithm used by the shared cache, where there is no sudden reconfiguration but a gradual move towards the intended partition. In this case, when a thread suffers a cache miss and the number of cache ways that belong to it is less than the thread's assigned cache partition ways, a cache line belonging to some other thread is chosen for replacement. If the number of cache ways belonging to the thread is greater than or equal to the assigned number of ways, a cache line belonging to the same thread is chosen for replacement. This way, the cache is incrementally partitioned via the replacement policy. Experimental results have shown a performance gain of up to 23% over a statically partitioned cache (MURALIDHARA *et al.*, 2010). The work does not provide real-time guarantees.

Another work to address shared resource contention via scheduling was proposed by Zhuravlev *et al.* (ZHURAVLEV *et al.*, 2010). Their approach identifies the main problems that can cause contention in shared multicore processors (*e.g.,* memory controller contention, memory bus contention, hardware prefetching, and cache space contention). The authors propose two scheduling algorithms (*Distributed Intensity* - DI, and *Distributed Intensity Online* - DIO). DI uses a threads' memory pattern classification as input and distributes threads across caches such that the miss rates are distributed as evenly as possible. DIO uses the same idea, but it reads the cache misses online, through HPCs. DIO performed better than the Linux Completely Fair Scheduler (CFS) both in terms of average performance as well as execution time stability from different executions (ZHURAVLEV *et al.*, 2010). Zhuravlev *et al.* also provided a survey of scheduling techniques that address shared resources in multicore processors to improve the average execution time of general-purpose applications (ZHURAVLEV *et al.*, 2012).

Calandrino and Anderson use HPCs to count the number of cache misses and estimate the WSS of each thread (CALANDRINO; ANDERSON, 2009). Based on the WSSs, a cache-aware real-time scheduler selects threads to be scheduled in such a way that co-runner threads do not thrash the shared cache memory.

### 3.3.3 Summary

In general, the above related works that propose a performance monitoring API or tool neither use nor design an efficient API to handle the complexity of HPCs in embedded systems. In this thesis, we propose a simple and lightweight API for a common PMU family available in today's multicore processors (the Intel family of PMUs). We describe our API in Chapter 4.

Considering the related works that use HPCs to improve OS decisions at run-time, only the work proposed by Tam *et al.* (TAM *et al.*, 2007) considers a scenario where threads/tasks share data (true or false sharing), but does not provide real-time guarantees. The contention for shared cache lines influences the application's execution time and lead to performance degradation and deadline losses. In Chapter 7, we propose a task partitioning algorithm that considers true and false sharing of tasks to improve the schedulability of hard real-time applications. In Chapter 8, we propose an online scheduling algorithm that uses HPCs to dynamically detect the access to shared cache lines among real-time and non real-time tasks and to discard non real-time tasks when they interfere with real-time tasks.

## 3.4 MULTICORE REAL-TIME SCHEDULING

In this section we review the recent multicore real-time scheduling works. Section 3.4.1 presents multicore real-time scheduling algorithms that extend global or partitioned approaches, which were reviewed in Section 2.4.2. Section 3.4.2 presents recent works that investigated how the OS run-time overhead impacts the schedulability of hard and/or soft real-time applications.

### 3.4.1 Scheduling Algorithms

Several scheduling algorithms have been proposed to provide real-time guarantees for multicore applications. They are usually either global or partitioned approaches. Considering the P-EDF algorithm, some works propose new task partitioning, splitting, or admission control techniques (KATO; YAMASAKI, 2007; MASRUR *et al.*, 2010; BURNS *et* *al.*, 2012). Additionally, semi-partitioning algorithms combine characteristics from partitioning and global scheduling: they allow few tasks to migrate between the processors to improve the system utilization (ANDERSON *et al.*, 2005; KATO; YAMASAKI, 2008; BLETSAS; ANDERSSON, 2009). Other works compare different real-time schedulability tests for G-EDF (GOOSSENS *et al.*, 2003; BAKER, 2005a; BAKER, 2003; BARUAH, 2007; BAKER; BARUAH, 2009; BERTOGNA *et al.*, 2005; BERTOGNA; CIRINEI, 2007; BARUAH *et al.*, 2009). In general, the performance evaluation in these works considers the ratio of schedulable task sets. For example, Bertogna and Baruah compared the main G-EDF schedulability tests for HRT scenarios up to eight processors (BERTOGNA; BARUAH, 2011). Baker compared three G-EDF schedulability tests with P-EDF approaches also in terms of the ratio of schedulable task sets (BAKER, 2005b).

In this work, we extended the G-EDF versus P-EDF comparison made by previous related works. We compare eight G-EDF schedulability tests with three P-EDF partitioning techniques (FFD, BFD, and WFD) using synthetic task sets composed of different periods and utilizations in a scenario with 100 processors.

Global schedulers based on different concepts have also been proposed. Cho *et al.* proposed a new abstraction for task execution on multiprocessors, named the time and local remaining execution-time plane (T-L plane) (CHO *et al.*, 2006). The entire scheduling over time is the repetition of T-L planes of various sizes. Other global scheduling algorithms are based on the proportional fairness, such as PFair (BARUAH *et al.*, 1996) and its variants, ER-Fair (ANDERSON; BLOCK, 2000), QRfair (ANDERSON *et al.*, 2003), $PD^2$ Pfair (SRINIVASAN, 2003), e DP-Wrap (LEVIN *et al.*, 2010). PFair-based schedulers are optimal, since they correctly schedule any feasible intra-sporadic task system on $m$ processors. However, PFair-based algorithms incur more run-time overhead, because of more complex scheduling decisions and migration rates. Their task models consider only periodic or sporadic tasks and do not deal with shared data among tasks (SRINIVASAN *et al.*, 2003).

Other real-time schedulers focus on different processor architectural aspects, such as memory hierarchy. Calandrino and Anderson proposed several heuristics to improve the performance on the cache usage for SRT systems (CALANDRINO; ANDERSON, 2008). In their approach, tasks as organized as tasks with multiple threads (MTT). Each MTT consists in periodic and sequential tasks with different execution times, but with the same period. Each MTT has a WSS that is shared among all threads within that MTT. All cores are symmetric, share a chip-wide L2 cache, and each core supports one hardware thread. The main idea is to encourage the co-scheduling of jobs that do not thrash the L2-cache, i.e., the sum of the WSS of each MTT must be less than the shared cache size. To achieve this for the G-EDF scheduler, the deadline of a job is promoted to allow it to execute before another job with a largest WSS that would cause cache thrashing. The proposed heuristics determine which job should be promoted and thresholds inform when a job should be promoted during a scheduling quantum. The heuristics and thresholds are the following (CALANDRINO; ANDERSON, 2008):

- Cache-aware: this heuristic is used when cache utilization reaches the cache utilization threshold. The scheduler chooses an MTT to promote based on the remaining "un-utilized" cache C and "free" cores N.

- Lost-cause: this heuristic is used when cache utilization reaches the lost-cause threshold. Once this threshold is reached, it is assumed that poor cache performance is inevitable during the current quantum. Then, three different policies are used: (i) revert to G-EDF; (ii) promote a MTT with the largest WSS; and (iii) promote a MTT with the largest ratio between the WSS and the number of threads in the MTT.

- Phantom tasks: if the system is not fully utilized, then it may be possible to idle one or more cores to prevent thrashing. Thus, the scheduler can choose to idle cores by promoting jobs of phantom tasks, which are single-threaded tasks that have a period equal to the hyperperiod of the real-time workload, an execution cost of one, and a WSS of zero.

An experimental evaluation concluded that the best performance, when comparing the heuristics with the original G-EDF scheduler, was achieved by using an utilization threshold of 0%, the cache-aware heuristic that promotes the job with smaller WSS, the lost-cause heuristic that uses G-EDF and phantom tasks. However, there are some limitations in the proposed approach. First, the WSS of each task must be known as well as the shared cache size. Second, the scheduler tries to schedule tasks within the same MTT that share the same memory region. Consequently, when two threads from the same MTT access the same data, the cache coherence protocol will delay the MTT's execution time. This delay is not considered by the authors.

Later, Calandrino and Anderson extended the proposed work by implementing the best heuristic (as described above) in the Linux kernel and using HPCs to estimate the WSS of each MTT (CALANDRINO; ANDERSON, 2009). A cache profiler measures the total cache miss number during a scheduling quantum, divides the cache miss number by the number of jobs scheduled in that quantum, and multiplies the obtained value by the cache line size. Thus, the profiler obtains an estimative of the WSS for one MTT. However, two assumptions are made: (i) the $i^{th}$ job of a task $T$ and the $j^{th}$ job of a task $U$, from the same MTT, do not share data; and (ii) the monitored jobs do not thrash the shared cache. The implemented algorithm was compared with G-EDF and obtained an improvement of up to 7% in the WCET of a multimedia application. The two assumptions limit the usage of the proposed approach, because applications may consume more memory than the shared cache size and tasks may share data.

$FP_{CA}$ is another cache-aware scheduling algorithm that divides the shared cache space into partitions (GUAN *et al.*, 2009). The used cache partitioning mechanism is page coloring. Tasks are scheduled in a way that, at any time, any two running tasks' cache spaces (*e.g.,* a set of partitions) are non-overlapping. A task can execute only if it gets an idle core and enough cache partitions. The authors proposed two schedulability tests, one based on a linear program (LP) problem and another one as an over-approximation of the LP test. Tasks are not preemptive and the algorithm is blocking, i.e., it does not schedule lower priority ready jobs to execute in advance of higher priority even though there are enough available resources.

A well-known metric to evaluate partitioning algorithms is the approximation ratio, which compares the number of bins needed by a bin packing heuristic A to pack a set of items, with the number of bins needed by an optimal algorithm. Coffman *et al.* presents a survey of bounds on approximation ratios of many bin packing heuristics (COFFMAN JR. *et al.*, 1997). Hochbaum and Shmoys have designed a polynomial-time approximation scheme (PTAS) that partitions any task system that can be partitioned upon a given platform by an optional algorithm by augmenting resources (in terms of faster processors) as compared to the resources available to the optimal algorithm (HOCHBAUM; SHMOYS, 1987). However, this algorithm has very poor implementation efficiency (BARUAH, 2013). Chattopadhyay and Baruah adapted the PTAS algorithm to obtain a better implementation efficiency (CHATTOPADHYAY; BARUAH, 2011). Baruah proposed the use of the speed up factor in conjunction with utilization bounds to better characterize the performance of different partitioning algorithms (BARUAH, 2013). The speedup factor of an algorithm $A$ is the smallest number $f$ such that any task system that can be partitioned by an optimal algorithm upon a particular platform can be partitioned by $A$ in a platform in which each processor is $f$ times as fast (BARUAH, 2013). The results indicated that the best algorithms are any of the reasonable allocation algorithms that first sort the tasks according to non-increasing order of utilization.

Starke and Oliveira proposed a cache-aware partitioning algorithm that assigns tasks to processors according to their WCET and CPMD (STARKE; OLIVEIRA, 2013). The utilization of a task takes the CPMD into consideration: $U_i^* = \frac{e_i + \Delta_i^{cpd}}{p_i}$. Tasks are sorted by non-increasing $U_i^*$ and assigned to processors according to the BFD heuristic and using the DM scheduling policy. The authors added the CPMD to the RTA test and used the test as the capacity bound of a processor. The CPMD ($\Delta_i^{cpd}$) as well as the other task parameters were randomly generated. The evaluation considering these generated parameters has shown a better schedulability ratio than the original BFD heuristic.

Lindsay proposed a partitioning algorithm for SRT systems that evenly distributes the WSS of the tasks on all cores in order to reduce cache conflicts and capacity-related cache misses (LINDSAY, 2012). The algorithm is named the Largest Working set size First, Grouping (LWFG). LWFG is based on the next fit decreasing bin packing heuristic, ordering the tasks by their decreasing WSS. The author relies on the notion of cache distance of NUMA architectures to further optimize the partitioning. This is only used for tasks that share common memory area(s) and are gathered into groups. However, it is unclear how the common memory area defines a group. If the next-fit heuristic fails to find a core with sufficient capacity to partition the group, the task that shares the least memory with the first task is removed from the group, and partitioning is retried with the smaller group. The evaluation using a real-time patch in the Linux kernel (ChronOS Linux) has shown that in some cases, LWFG has an improvement of instructions per cycles of up to 15% and decreases the maximum tardiness by up to 60% when compared to common bin packing partitioning heuristics (WFD, FFD, and BF). LWFG fails to provide HRT guarantees, because it divides tasks of a group that shares memory and partitions these tasks in different cores. Consequently, the cache coherence protocol may invalidate shared cache lines, thus delaying the tasks' execution time.

Nemati *et al.* also proposed a partitioning algorithm that groups tasks with shared resources and assigns the entire group to a same core (NEMATI *et al.*, 2009). Resource sharing among tasks is identified by a matrix. By analyzing this matrix, the algorithm calculates a task weight through a cost function. Then, tasks are ordered according to their weight and inserted into a task group if it satisfies a schedulability test. The proposed approach was not evaluated, neither simulated nor in a real hardware.

Suzuki *et al.* proposed two algorithms, one based on solving a mixed-integer linear problem and another one based on solving a variant of the knapsack problem, for assigning tasks to processor to decrease conflicts at the cache and DRAM bank levels (SUZUKI *et al.*, 2013). The proposed algorithms use cache and bank coloring together. The objective is to optimize the assignment of cache colors to tasks and bank colors to processors and avoid cache and bank interference among tasks. The authors have shown by experimentation that cache and bank coloring together increases the system performance. However, the allocation algorithms were not evaluated in a real hardware nor in terms of real-time guarantees. Moreover, transitive color sharing seems not to be treated by the proposed memory interference model. Our page coloring mechanism could be easily extended to also support bank coloring.

In this work, we use HPCs to detect when non real-time tasks (BE tasks) interfere with real-time ones. Then, we prevent BE tasks from running to decrease the contention for shared cache lines and to allow real-time tasks to meet their HRT and SRT deadlines. We also propose a color-aware task partitioning algorithm, which groups tasks that use the same color and partitions each group into available cores to provide HRT guarantees. Chapter 7 describes the partitioning algorithm and Chapter 8 describes the dynamic scheduler.

### 3.4.2 Run-time Overhead and Implementation Tradeoffs

Several multicore real-time schedulers were evaluated considering run-time overhead using $LITMUS^{RT}$ (see Section 3.2.4 for an overview). Calandrino *et al.* measured run-time overheads of G-EDF, P-EDF, and two variants of the PFair algorithm (CALANDRINO *et al.*, 2006). Brandenburg *et al.* investigated the scalability in terms of number of processors in partitioned and global schedulers (BRANDENBURG *et al.*, 2008). Brandenburg and Anderson discussed how the implementation of the ready queue, quantum-driven versus event-driven scheduling, and interrupt handling strategies affect a global real-time scheduler, considering the different run-time overhead sources in each implementation (BRANDENBURG; ANDERSON, 2009). The results indicated that implementation issues can impact schedulability as much as scheduling-theoretic tradeoffs. Furthermore, in their case study, the system achieved the best performance by using a fine-grained heap, event-driven scheduling, and dedicated interrupt handling. An empirical comparison of G-EDF, P-EDF, and C-EDF on a 24-core Intel platform, assuming run-time overhead (*e.g.,* release, context switch, and scheduling) and CPMD, has concluded that P-EDF outperforms the other evaluated algorithms in HRT scenarios (BASTONI *et al.*, 2010a). Moreover, the same study suggests the use of "less global" approaches (P-EDF and C-EDF-L2, which cluster at the cache level 2) in contrast of "more global" approaches (G-EDF and C-EDF-L3) for HRT applications. Bastoni *et al.* investigated implementation tradeoffs in semi-partitioned schedulers (BASTONI *et al.*, 2011). Mollison and Anderson proposed a userspace scheduler implemented on top of Linux that supports C-EDF (MOLLISON; ANDERSON, 2012). The authors measured OS overhead, including releasing, scheduling and context switching, and compared the obtained values with LITMUS$^{RT}$. They concluded that the overheads of both implementations are roughly comparable.

Following the same research line, Lelli *et al.* compared the performance of partitioned, clustered, and global RM and EDF scheduling algorithms on top of Linux, focusing on SRT applications (LELLI *et al.*, 2012). The authors concluded that the migration overhead is not more costly than a context switch, in an AMD Opteron NUMA platform with 48 cores. In addition, the clustered variants were more efficient than the global approaches mainly due to reduced run-time overhead, as was also noted by (BASTONI *et al.*, 2010b). Other studies created micro-benchmarks to quantify context switch overhead on specific processors and/or situations, e.g., hardware interrupts, program data size, and cache performance (MOGUL; BORG, 1991; DAVID *et al.*, 2007; LI *et al.*, 2007; TSAFRIR, 2007).

In general, the related works that measured run-time overhead in multicore real-time schedulers used Linux as base OS. We extended these works by comparing LITMUS$^{\mathrm{RT}}$ (that uses Linux) with EPOS RTOS in terms of run-time overhead and schedulability ratio. We choose LITMUS$^{\mathrm{RT}}$, because several works that measure the run-time overhead were implemented on top of LITMUS$^{\mathrm{RT}}$ and it has an active community. To the best of our knowledge, EPOS is the first open-source research RTOS that supports global, partitioned, and clustered schedulers in the context of multicore real-time systems. We believe that this OS can be extensively and easily used to conduct research in the area, due to higher predictability and smaller overhead, obtaining more precise results for HRT scenarios. Chapter 5 shows the run-time overhead comparison between EPOS and LITMUS$^{\mathrm{RT}}$. In the next section we present a summary of the analyzed multicore (real-time) scheduling algorithms.

### 3.4.3 Summary

Table 11 summarizes the main characteristics of the analyzed multicore (real-time) scheduling algorithms. The works based on proportional fairness, although optimal, are difficult to implement in practice and have a larger run-time overhead due to higher scheduling decision and migration rates (SRINIVASAN *et al.*, 2003). The $FP_{CA}$ algorithm provides HRT guarantees, but the used task model is simples and difficult to be found in real systems.

Considering the usage of HPCs, in general, the related works use the shared cache miss number to help the OS to estimate the cache space when performing cache partitioning at run-time. Furthermore, the most used cache partitioning method is page coloring. The work proposed by Tam *el al.* is the only that deals with the delay caused by the cache coherence protocol. However, it does not provide real-time guarantees. Only the scheduler proposed by Calandrino and Anderson (CALANDRINO; ANDERSON, 2008) uses the possibility of stopping a core ("idle") to execute real-time tasks for a while to avoid cache thrashing. Among the real-time schedulers, *none of them explicitly deals with shared data among real-time tasks.*

In this work, we use HPCs to detect when non real-time tasks interfere with real-time tasks. Then, the scheduler avoids to co-schedule real-time tasks with non real-time tasks that interfere with each other. The objective is to guarantee the execution of real-time tasks within their deadlines. We also propose a task partitioning algorithm for task sets composed only of HRT tasks. Our task partitioning approach is similar to the LWFG algorithm. However, our partitioning algorithm assumes that a cache partitioning mechanism based on page coloring is available and each task uses a set of colors. Thus, it is possible to determine which tasks share cache lines by inspecting the colors of each task. In contrast, in the LWFG approach, it is not clear how the algorithm is aware of data sharing among tasks. Moreover, LWFG is proposed for SRT system, since whenever the next fit heuristic fails to partition a task group, the task that shares the least memory with the first task is removed from the group, and partitioning is retried with the smaller group. Thus, inter-core interference is not avoided, which may incur in deadline losses. Our partitioning algorithm limits the utilization of a task group in the processor capacity given by the scheduling algorithm to avoid inter-core interference. Furthermore, the performance evaluation of LWFG was carried out using a real-time patch for the Linux kernel, which may limit the observed gains, because the inherent non real-time behavior of Linux, as will be shown in Chapter 5. Instead, we evaluate our algorithm using an RTOS designed from scratch, without the excessive run-time overhead introduced by a GPOS. Chapter 7 presents our partitioning algorithm and its evaluation, while Chapter 8 presents the design, implementation, and evaluation of the dynamic scheduler.

**Table 11: Comparative table of the features of the analyzed multicore (real-time) scheduling algorithms.**

| **Algorithm** | **Features** | | | | |
|---|---|---|---|---|---|
| | *Deal with shared memory* | *Memory partitioning* | *HPCs Usage* | *Real-time guarantees* | *Idle core* |
| Tam *et al.* (TAM *et al.*, 2007) | Yes | No | L1 data cache misses and number of remote accesses | No | No |
| Azimi *et al.* (AZIMI *et al.*, 2009) | Yes | Yes | Miss rate (`RapidMRC`) | No | No |
| Zhuravlev *et al.* (ZHURAVLEV *et al.*, 2010) | No | No | Cache miss rate | No | No |
| West *et al.* (WEST *et al.*, 2010) | No | No | Cache misses and hits | No | No |
| `Pfair` (BARUAH *et al.*, 1996) and variations (ANDERSON; BLOCK, 2000; ANDERSON *et al.*, 2003; SRINIVASAN, 2003; LEVIN *et al.*, 2010) | No. Tasks are independent | No | No | Hard | No |
| Cache-aware scheduler (CALANDRINO; ANDERSON, 2008) | No Try to schedule tasks that share data at the same time | Use information from the WSS | No | Soft | Yes |
| Cache-aware scheduler with HPC (CALANDRINO; ANDERSON, 2009) | No | Schedule jobs that do not thrash the cache | Cache misses | Soft | Yes |
| $FP_{CA}$ (GUAN *et al.*, 2009) | No Tasks are independent | *Page coloring* | No | Hard | No |

# 4 REAL-TIME SUPPORT ON EPOS

In this chapter[1], we present our extensions to the Embedded Parallel Operating System (EPOS) to support and improve the predictability of multicore real-time applications. EPOS is a multi-platform, object-oriented, component-based, embedded system framework designed following the Application-Driven Embedded System Design (ADESD) methodology (FRÖHLICH, 2001; EPOS, 2014). EPOS has been used in several academic and industrial research and development projects in the last years, such as software-defined radio (MÜCK; FRÖHLICH, 2011), wireless sensor networks (WANNER; FRÖHLICH, 2008), digital TV (LUDWICH; FRÖHLICH, 2011), uniprocessor real-time schedulers implemented in hardware (MARCONDES *et al.*, 2009), and energy-efficient applications (FRÖHLICH, 2011).

The main contributions of this chapter are:

- We propose a HPC API for monitoring hardware events specifically designed for multicore (real-time) embedded systems. The API is designed following the ADESD concepts (FRÖHLICH, 2001) and it is able to provide to OSes a simple and lightweight interface for handling the communication between applications and PMUs. Through an API usage example, in which a hardware event is used by the OS while making scheduling decisions, we were able to identify the main drawbacks of the current PMUs. As a consequence, we propose a set of guidelines, such as features for monitoring address space intervals and OS trap generation, that can help hardware designers to improve the PMU capabilities in the future, considering the embedded operating system's point of view.

- We show how a well-designed object-oriented component-based

[1] Contents of this chapter appear in the following published papers:
G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, Implementation and evaluation of global and partitioned scheduling in a real-time OS, Real-Time Systems, v. 49, p. 1-48, 2013.
G. Gracioli and A. A. Fröhlich, An Experimental Evaluation of the Cache Partitioning Impact on Multicore Real-Time Schedulers, In Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013.
G. Gracioli and A. A. Fröhlich, An embedded operating system API for monitoring hardware events in multicore processors, Workshop on Hardware-support for parallel program correctness, 2011.

RTOS (EPOS) allows code reuse of system components (*e.g.,* scheduler, thread, semaphore, etc) and easy global, clustered, and partitioned real-time scheduling extensions. To the best of our knowledge, EPOS is the first open-source RTOS that supports global and clustered schedulers. We believe that EPOS can be used to conduct research in multicore HRT due to higher predictability and smaller overhead compared to real-time patches for Linux (this comparison is done in Chapter 5).

- We design and implement an original cache partitioning mechanism in EPOS. The mechanism is able to assign partitions to the OS internal data structures and does not rely on any specific hardware support. Additionally, two different memory allocation approaches are supported that define from which partition data should be allocated: the user-centric approach, in which the developer inserts code annotations to define the partition, and the OS-centric approach, in which EPOS chooses the partition based on a task ID.

The rest of this chapter is organized as follows. Section 4.1 shows the hardware mediator concept and examples, as well as the design and implementation of an API of a PMU. Section 4.2 presents the design and implementation of the real-time multicore support on EPOS. Finally, Section 4.3 presents the design and implementation of a memory management (*i.e.,* page coloring) mechanism in EPOS that provides shared cache partitioning for real-time applications.

## 4.1 HARDWARE MEDIATORS

In EPOS, platform-independent system components implement traditional OS services, such as threads and semaphores. Hardware mediators implement platform-specific support (POLPETA; FRÖHLICH, 2004). Hardware mediators are functionally equivalent to device drivers in Unix, but do not build a traditional HAL. Instead, hardware mediators sustain the interface contract between software and hardware by means of static metaprogramming techniques and inlining code that "dilute" mediator code into components at compile time (no calls, no layers, no messages; mostly embedded assembly). Figure 32 illustrates the process of "diluting" the mediators code at compile time. It is important to note that the final system image only contains the code of the used mediators (POLPETA; FRÖHLICH, 2004).

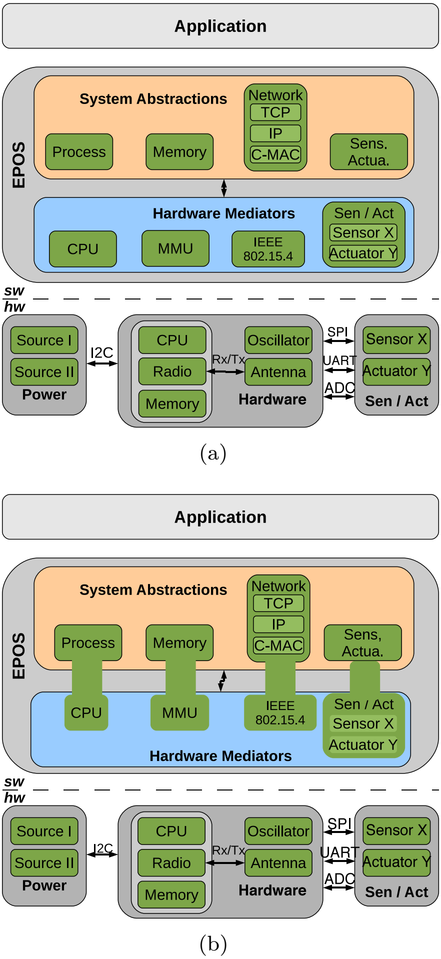


**Figure 32: During the system compilation, hardware mediators are diluted into components. (a) hardware mediators before compilation and (b) hardware mediators after compilation.**

Figure 33 shows part of the CPU hardware mediator family. The IA32 mediator declares the *cas* (compare and swap) method as inline.

```
 1 class CPU_Common
 2 {
 3       ...
 4     static int cas(volatile int & value, int compare, int
            replacement) {
 5         int old = value;
 6         if(value == compare) {
 7             value = replacement;
 8         }
 9         return old;
10     }
11       ...
12 }
13
14 class IA32: public CPU_Common
15 {
16       ...
17     static int cas(volatile int & value, int compare, int
            replacement) {
18       ASMV(”lock cmpxchgl %2, %3\n”
19       : ”=a”(compare)
20       : ”a”(compare), ”r”(replacement), ”m”(value)
21       : ”memory”);
22       return compare;
23     }
24       ...
25 };
```

**Figure 33: A part of the IA32 CPU hardware mediator. The *cas* method is declared as inline.**

The compiler inserts the *cas* code into the components that use the method at compile time, as exemplified by Figure 32, without any calls or layers. Note that if the target processor does not support bus locking (*e.g.,* the lock instruction at line 18), the CPU_Common class provides a generic implementation that can be reused in uniprocessor systems.

### 4.1.1 Performance Monitoring Unit

HCPs are special registers available in most of the modern microprocessors through a hardware PMU, which provides access to counting and/or sampling hardware events at run-time. These registers can be used together with OS techniques, such as scheduling and memory management, to monitor and identify performance bottlenecks and thus carry on dynamic optimizations (see Section 3.3.2 for a comprehensive review on recent works that have used HPCs to this purpose).

As the utilization of multicore processors in the embedded (real-time) system domain is increasing, an API for handling the complexity of HPCs specifically designed to this domain is desirable. The current HPCs APIs, initially proposed to general purpose computing, such as PAPI (DONGARRA *et al.*, 2003) and Intel perf (MALLADI, 2010), are not the most suitable for embedded systems, because they can use a substantial amount of resources, which makes the communication between software and hardware PMUs slower, or require the initialization and creation of lists or event sets that decrease the performance. Moreover, those APIs provide several functionalities that are not interesting for an embedded application, such as user-defined events or performance modeling metrics. Hence, an API specifically tailored for the application's needs delivers the meaningful functionality with a lower run-time overhead.

In this section, we propose a HPC API for monitoring hardware events specifically designed for embedded multicore systems. The proposed performance monitoring API was designed following the ADESD methodology. ADESD is a methodology to guide the development of application-oriented operating system from domain analysis to implementation (FRÖHLICH, 2001). ADESD is based on the well-known domain decomposition strategies found in *Family-Based Design* (FBD) (PARNAS, 1976), *Object-Oriented Design* (OOD) (BOOCH, 2004), *Aspect-Oriented Programming* (AOP) (KICZALES *et al.*, 1997), and *Component-Based Design* (CBD) (SANGIOVANNI-VINCENTELLI; MARTIN, 2001) in order to define components that represent significant domain entities.

Figure 34 shows an overview of the ADESD methodology. The problem domain is analyzed and decomposed into independent components or abstractions[2] that are organized as members of families, as defined by FBD. To reduce environment dependences and to increase abstractions re-usability, ADESD aggregates aspect separation from AOP to the decomposition process. Hence, it is possible to identify scenario variations and non-functional properties to model them as

[2] In EPOS, a component is a C++ class with a well-defined interface and behavior. Abstractions are user-visible components.

*scenario aspects* that crosscut the entire system. The *scenario adapter* wraps a component and applies a corresponding set of aspects to it (FRÖHLICH; SCHRÖDER-PREIKSCHAT, 2000).

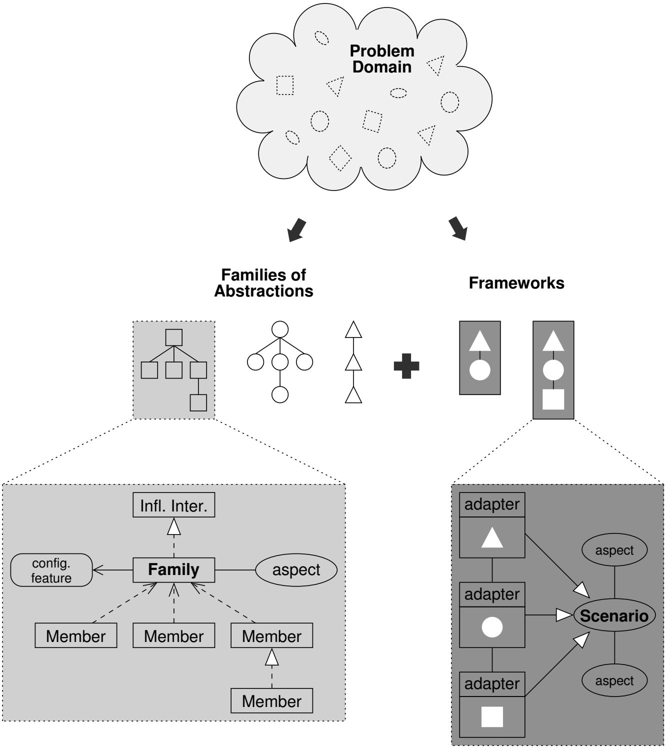


**Figure 34: ADESD methodology overview.**

Families of components are visible to application developers through *Inflated Interfaces* that export their members as an unique "super component". These *inflated interfaces* allow developers to postpone the decision about which component should be used until enough configuration knowledge is acquired. An automatic configuration tool is responsible for binding an *inflated interface* to one of the family members, choosing the appropriate member that realizes the required interface, independently of whether they are implemented in software or hardware.

The proposed monitoring infrastructure is composed by a PMU hardware mediator family and a platform-independent component. The interface of this component is used by application developers. Moreover, the component uses the hardware mediators in order to configure and read the HPCs. In the next subsections we present the hardware mediator family, the OS component, an example of how to use the API, and a practical use on OS scheduling.

### 4.1.2 PMU Hardware Mediator Family

We have designed a hardware mediator interface for the Intel PMU family. Figure 35[3] shows the UML class diagram of the interface. Intel processors, depending on the microarchitecture (*e.g.,* Nehalem, Core, Atom, etc), have different PMU versions. Each version provides different features and a variable number of HPCs. For example, the PMU version 2 has two performance counters that can be configured with general events and three fixed performance counters that count specific events, while the PMU version 3 provides additional support for simultaneous multi-threading, up to 8 general-purpose performance counters, and precise event based sampling (Intel Corporation, 2011). Yet, pre-defined architectural events, such as unhalted core cycles, last-level cache misses, and branch instruction retired, are shared by all the three versions.

Configuring an event involves programming performance event selection registers (IA32_PERFEVTSELx) corresponding to a specific physical performance counter (IA32_PMCx). In order to select an event, the PERFEVTSELx register must be written with the selected event, unit, and several control flags (also called as masks). The unit mask qualifies the condition that the selected event is detected. For example, to configure the PMC0 to count the number of snoop responses to bus transactions, the PERFEVTSEL0 must be written with the EXT_SNOOP event mask (0x77) and two unit masks that define the conditions when the event should be counted.

The designed PMU hardware mediator family represents the described Intel PMU organization. A base class IA32_PMU implements the Intel PMU version 1 and common services for both versions 2 and

[3] An underline in a UML class diagram defines a static attribute or method. A plus (+) signal defines a public attribute or method and a minus (-) signal a private attribute or method.

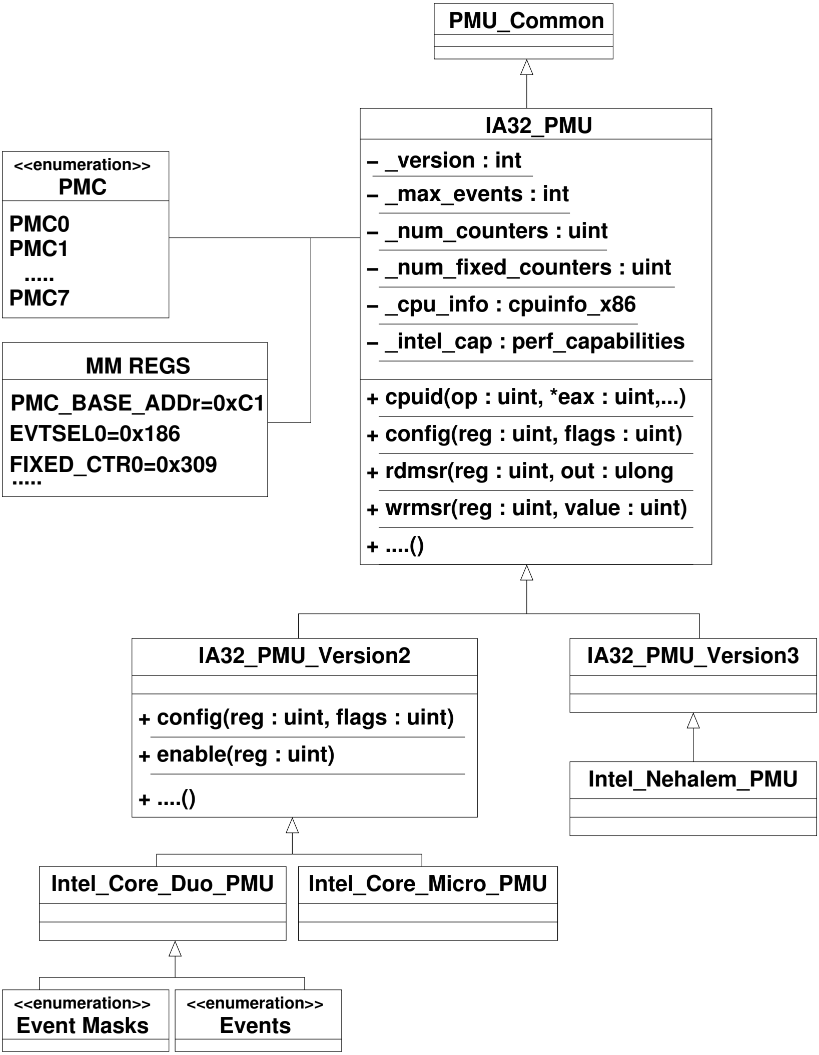


**Figure 35: UML class diagram for the proposed PMU hardware mediator API.**

3, including the pre-defined architectural events. Also, this class declares memory mapped registers and PMCs. The IA32_PMU_Version2 and IA32_PMU_Version3 extends the base class and implement specific services only available on those versions. Finally, available hardware events are listed by specific microarchitecture classes. For instance, Intel_Core_Micro_PMU and Intel_Nehalem_PMU list all available events masks for the Intel Core and Intel Nehalem microarchitectures, respectively.

The hardware mediator interface can be used by platform-independent components. Those components are responsible for implementing "intelligent" logic by using the mediators. For instance, event ratios such as Cycles per Retired Instruction (CPI), parallelization ratio, modified data sharing ratio, and bus utilization ratio, combine two or more hardware events in order to provide useful insight into the application performance issues (MALLADI, 2010). Moreover, a platform-independent component could also multiplex the hardware counters in order to overcome the limitation on the number of hardware counters. Multiplexing techniques divide the usage of counters over the time, providing to users a view that there exists more hardware counters than processors really support (DONGARRA *et al.*, 2003). We also propose a performance monitor abstraction, which is described below.

### 4.1.3 Performance Monitor

Figure 36 shows the performance monitor (`Perf_Mon`). It provides a set of methods to configure and read several event ratios and specific hardware events, such as last-level cache misses and L1 data cache snooped. The component hides from the users all the complexity of configuring, writing, and reading the hardware counters. Moreover, it also provides means for handling possible overflow in the counters.

The performance monitoring component uses the previously presented hardware mediator family. However, due to function inlining, the code of the mediators is dissolved into the component. For example, consider the code in Figure 37. There is a method for configuring the event CPU_CLK_UNHALTED_BUS[4] and another method for reading the event. The code of the

[4] This event counts the number of bus cycles while the core is not in the halt state. This event gives a measurement of the elapsed time while the core was not in the halt state, by dividing the event count by the bus frequency. The core enters the halt state when it is running the HLT instruction (Intel Corporation, 2011).

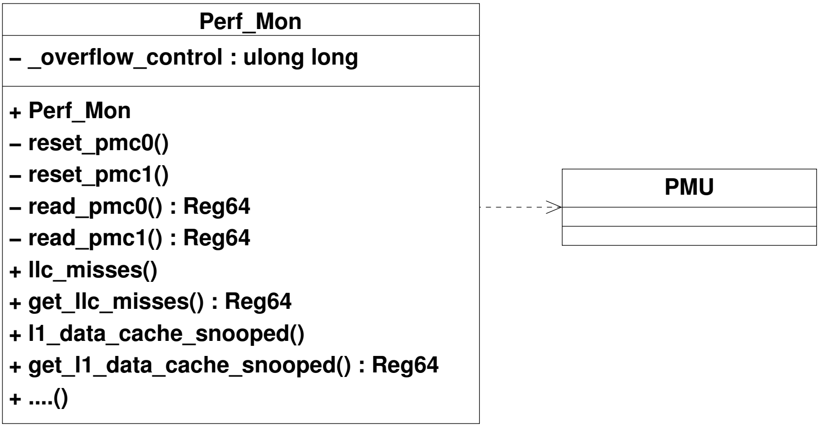

Perf_Mon
– _overflow_control : ulong long
+ Perf_Mon
– reset_pmc0()
– reset_pmc1()
– read_pmc0() : Reg64
– read_pmc1() : Reg64
+ llc_misses()
+ get_llc_misses() : Reg64
+ l1_data_cache_snooped()
+ get_l1_data_cache_snooped() : Reg64
+ ....()
PMU

**Figure 36: Performance monitoring OS component.**

mediator `Intel_Core_Micro_PMU::config()` is diluted into the `enable_cpu_clk_unhalted_bus()` method, thus there is no overhead associated to method calls.

At the same way, the code for reading the performance counter 0 is diluted into the `get_cpu_clk_unhalted_bus()` and no method calling or argument passing is generated in the final system image. Consequently, the generated image only aggregates the code needed by the application and nothing else. In the future, we plan to add into the component and the hardware mediator family a support for other PMU families, as those found in the ARM and PowerPC processors. To this end, we plan to make a complete domain analysis and extracted the common events of each PMU family. Thus, it will be possible to add a platform-independent layer for all PMUs. Moreover, we want to improve the PMU infrastructure, adding support for multiplexing and interrupt generation.

### 4.1.4 API Usage

In order to exemplify the API usage, we have designed a benchmark to generate shared cache memory invalidations in a multicore processor (Intel Core 2 Q9550). The benchmark is composed of two versions of an application: one sequential and another parallel. Both applications execute the same code (2 functions), but the parallel runs

```
 ...
class Perf_Mon
{
public:
 ...
void enable_cpu_clk_unhalted_bus() {
        //configure PMC0 to monitor the
              CPU_CLK_UNHALTED_BUS event
        Intel_Core_Micro_PMU::config(PMU::EVTSEL0, (
              Intel_Core_Micro_PMU::CPU_CLK_UNHALTED_BUS |
              PMU::USR | PMU::OS | PMU::ENABLE));
}

Reg64 cpu_clk_unhalted_bus() {
        return read_pmc0();
}
 ...
```

**Figure 37: Perf_Mon using the hardware mediator. The hardware mediator code is "dissolved" into the component at compile time.**

the two functions (in two different threads) in different cores at the same time and share two arrays of data. We also implemented a third version (best-case), in which both functions execute in parallel but do not share data. The objective is to demonstrate the utility of the proposed API in a multicore processor.

Figure 38 shows how the API is used by the parallel and best-case applications. At the beginning of a thread (func0), the performance monitoring component is created and the method for monitoring the number of snoops in the L1 data cache is started. At the end of the function, the hardware event is read and printed in the screen. For the sequential version, the performance monitoring component is created in the main function, since the two functions are executed in a sequential order in the same core. We choose this event because it represents memory coherence activities between the cores.

The benchmark, as well as the monitoring infrastructure, was implemented on top of EPOS. Each application was executed 10 times in the Intel Core 2 Q9550 processor, then we extracted the average number of snoops for each of them. The arrays' size was set to 8 MB (4 MB each). Each function has a loop with 10000 repetitions in which math operations are executed using the data in the arrays. Figure 39 shows

```
 ...
Semaphore s;
 ...
int func0(void)
{
        #ifndef __SEQUENTIAL
        Perf_Mon perf0;
        perf0.l1_data_cache_snooped();
        #endif
        register unsigned int sum0;
         ....

        #ifndef __SEQUENTIAL
        s.p();
        cout << ”\nL1 data cache snooped func0 = ” << perf0.
             get_l1_data_cache_snooped() << ”\n”;
        s.v();
        #endif
}
 ...
```

**Figure 38: An example of how to use the proposed API.**

the measured values. We can clearly see the difference among the three applications. The sequential and best-case applications have obtained a similar number of events. The parallel one obtained about 3 orders of magnitude more events and was slower than the sequential one. This confirms the software behavior – each function in the parallel application frequently reads/writes the same cache lines, generating snoops in the bus and cache line invalidations. The explanation for snoops in the sequential and best-case applications is the natural implementation of a multicore OS, where shared variables are used to guarantee mutual exclusion in some data structures. The hardware event correctly measures the bus activities and can be used by the OS to improve performance. The standard deviation for the sequential, parallel, and best-case application was 4.83%, 0.13%, and 5.05% respectively.

In order to compare the obtained values in terms of correctness, we ran the same three benchmark applications in the Linux 2.6.32 and used the *perf* Linux tool to read the number of snoops. We also ran each application for 10 times and extracted the average value. The evaluation was executed in the same Intel Q9550 processor. Figure 40 shows the obtained values. Linux has obtained in average 30% more

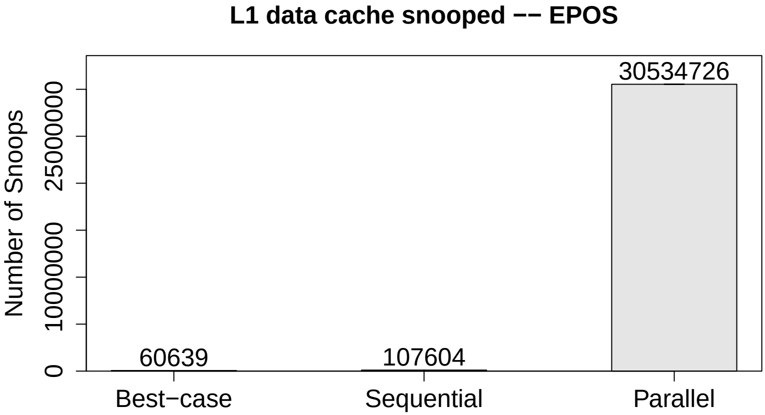


**Figure 39: API usage example: number of snoops in the L1 data cache for the three benchmark applications.**

snoops than EPOS and the standard deviation was also higher. The standard deviation for the sequential, parallel, and best-case application are 8.27%, 2.85%, and 9.96% respectively. As EPOS generates a system image composed of only the needed code, the influence of other OS parts in the execution of an application decreases. Consequently, the measured hardware events in EPOS are more precise than those obtained in Linux.

In order to demonstrate the performance of the proposed API, we have measured the memory overhead introduced by the API methods. The method for configuring the hardware counter (the same number of snoops hardware event as used above) occupied 32 bytes and 11 instructions and the method for reading the counter occupied 100 bytes and 40 instructions with no method calls, only inlined assembly code (Section 4.1.4 provides a code example of how these methods were implemented). Other APIs designed to general-purpose computing, such as PAPI (MUCCI *et al.*, 1999), require the initialization and creation of lists or event sets which decrease the performance. Polpeta and Fröhlich have compared EPOS hardware mediators to HALs implemented on eCos and $\mu$Clinux in terms of performance and memory consumption (POLPETA; FRÖHLICH, 2005). The authors have shown

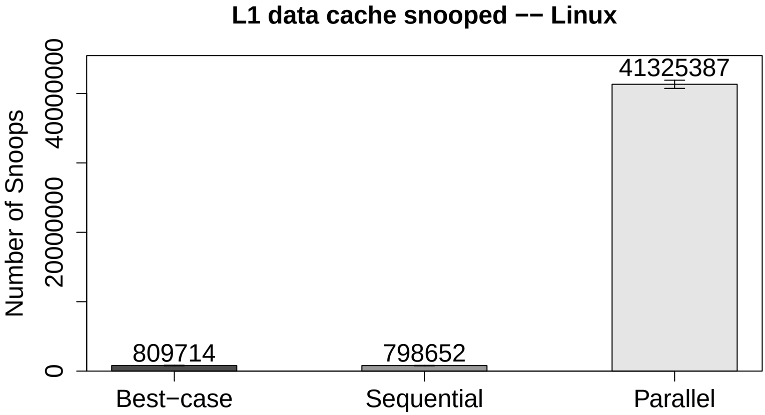


**Figure 40: Number of snoops in the L1 data cache for the three benchmark applications running in Linux.**

that hardware mediators have better results in both metrics.

The proposed interface could also be easily implemented to represent other PMU families, such those from the PowerPC and ARM processors. The API was implemented in C++ using the ADESD concepts in the EPOS operating system. EPOS is a component-based operating system, thus the same proposed PMU infrastructure could be used by other component-based operating systems without much implementation effort.

### 4.1.5 Applicability for Scheduling

As an example to demonstrate the usage of HPCs in OS implementation, we have used the L1 data cache snooped event to monitor the number of snoops at run-time and help the OS scheduling decisions. We have added to the EPOS reschedule method (see Section 4.2) a call to read the hardware event during a scheduling quantum (10 ms). By running the sequential and best-case applications, we observed that during a quantum, both applications obtained up to 100 events. By setting a threshold value (1000 in this case) we can detect when there is frequent snoops for cache lines and thus take a decision. Figure 41 shows the code exemplifying the changes. When the threshold value is reached, it is possible to move a thread to a core closer to the data (in case of a ccNUMA processor) and thus improving the application performance as demonstrated by Tam *et al.* (TAM *et al.*, 2007).

```
void Thread::reschedule(bool preempt)
{
    ...
    Thread * prev = running();
    Thread * next = _scheduler.choose();

    unsigned long long n_l1_data_snooped = _perf.
        l1_data_cache_snooped();

    if(n_l1_data_snooped > 1000) {
        // move a thread to another core
        ...
    }

    dispatch(prev, next);
    ...
}
```

**Figure 41: Modifying the OS scheduling.**

### 4.1.6 Guidelines for future PMU designs

Initially, hardware designers have added PMU capabilities into processors to collect information for their own use (TAM *et al.*, 2007). However, PMUs features have become useful for other performance measurements, such as energy consumption, memory partitioning, and scheduling. In consequence, hardware designers are now adding more functionality to PMUs, which can certainly help software developers even more. Below we provide a discussion about desired features that could help hardware designers to improve PMU features in multicore processors:

- **Data address registers:** storing data addresses that generated an event could certainly provide to the OS a powerful mean to perform optimizations. The IBM Power5 processor has a similar feature, but it could be improved. In this processor, the last data address accessed is stored into a special register. Thus, the last monitored event can be associated to the last address accessed. The work proposed by Tam et al. used this feature to estimate the memory consumption of the system threads and to help a cache memory partition mechanism, improving the system performance (TAM *et al.*, 2009). It would be interesting if data address registers were associated to events that are representative to an OS, such as last-level cache misses, bus snoops, and bus transactions.

Recent Intel processors have added support for precise event-based sampling (PEBS) that identifies instructions that cause key performance events and allows the developers to allocate a PEBS buffer in memory to hold the samples (*e.g.*, program counter and general-purpose registers values) collected during the program execution, which is extremely important for debugging mechanisms, such as tracing and replay (GRACIOLI; FISCHMEISTER, 2009).

- **Monitoring address space intervals:** in a multicore processor, several threads run in different cores and share the same address space. From the OS point of view, monitoring only the address spaces that are used by specific threads would allow a more precise view of the software behavior and consequently a more correct action could be taken by the OS.

- **Processing cycles spent in specific events:** especially for HRT applications, where deadlines must be always met, bus cycles spent in specific events are extremely important for estimating the execution time of tasks. In Intel processors, for example, there are events for measuring the bus cycles spent accessing the shared cache, bus cycles when data is sent on the front-side bus, bus cycles when the HIT and HITM pins are asserted in the bus, and so on. However, it is difficult to get the cycles for a specific event, as a cache miss or a bus snoop. Improving the PMU capabilities for providing the precise number of cycles in an event could ease the OS task of guaranteeing deadline constraints for real-time applications running in multicore processors.

- **OS trap generation:** PMUs could generate traps for the OS according to pre-defined numbers associated to events. This feature would allow the OS to be interrupted only when the number of hardware events is reached. Therefore, the OS could handle the exception and take a decision based on the the event that generated the trap.

### 4.1.7 Summary of the PMU support

In this section we proposed a PMU API specifically designed for embedded (real-time) multicore systems. The main focus of the API is on performance – to be fast and accurate. The results in Figures 39 and 40 have shown that our API is about 30% more accurate and has a smaller standard deviation than the perf Linux tool. We also briefly discussed a set of guidelines for future PMU designs to help software developers to have access to precise hardware event monitoring. The designed API is used by our dynamic scheduler in Chapter 8.

## 4.2 SCHEDULING

Figures 42 and 43 show the three main components responsible for the real-time scheduling support in EPOS. The `Thread` class represents an aperiodic task and defines its execution flow, with its own context and stack. The class implements traditional thread functionality, such as suspend, resume, sleep, and wake up operations. The `Periodic_Thread`[5] class extends the `Thread` class to provide support for periodic tasks by aggregating mechanisms related to the periodic task re-execution. The `wait_next` method performs a `p` operation on a semaphore that forces the thread to sleep during its defined period. The implementation of the `Semaphore` follows the traditional semaphore concept (DIJKSTRA, 1968). When a timer interrupt arrives, the timer handler (an `Alarm`) performs a `v` operation on the semaphore to release and wake up the task. The periodic thread constructor creates this alarm. We added two different handlers to the periodic thread: one static and another dynamic. At compile time, the

[5] An EPOS periodic thread is conceptually implemented as a real-time periodic task.

developer defines the scheduling criterion in the thread's `Trait`[6] class (for example, `typedef Scheduling_Criteria::GEDF Criterion` defines the scheduling criterion as G-EDF). If the criterion is dynamic, as EDF and LLF, the `Periodic_Thread` uses the dynamic handler. If the criterion is static, as RM and DM, the `Periodic_Thread` uses the static handler. The dynamic handler calls the `update` method from the defined scheduling criterion to update the task's priority. As we use C++ static metaprogramming (`typedef IF<Criterion::dynamic, Dynamic_Handler, Static_Handler>::Result Handler`), all dependencies are solved at compile time, without any run-time overhead. Section 4.2.1 shows more details about the activities performed on each periodic thread operation (sleep and wake up).

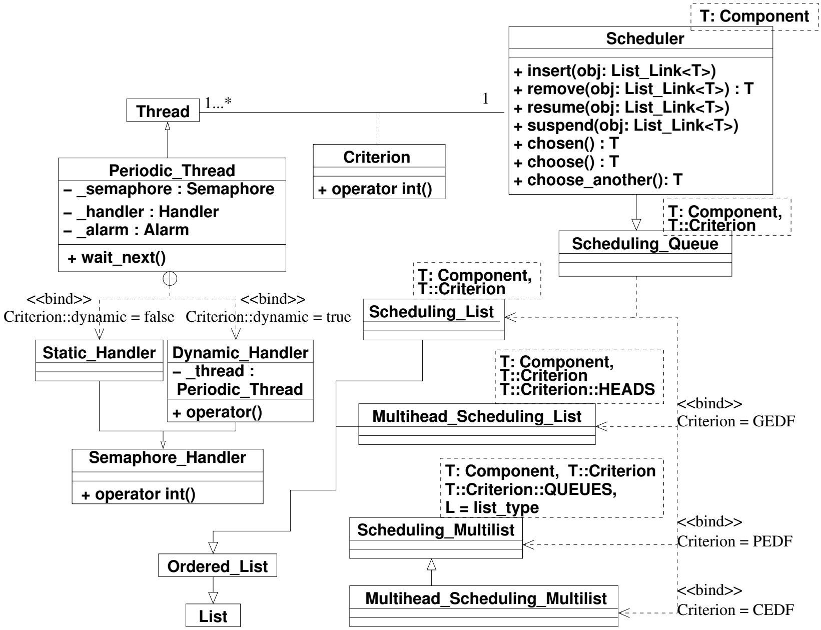

T: Component
Scheduler
+ insert(obj: List_Link<T>)
+ remove(obj: List_Link<T>) : T
+ resume(obj: List_Link<T>)
+ suspend(obj: List_Link<T>)
+ chosen() : T
+ choose() : T
+ choose_another(): T
Thread
1...*
1
Criterion
+ operator int()
Periodic_Thread
– _semaphore : Semaphore
– _handler : Handler
– _alarm : Alarm
+ wait_next()
T: Component, T::Criterion
Scheduling_Queue
T: Component, T::Criterion
Scheduling_List
<<bind>> Criterion::dynamic = false
<<bind>> Criterion::dynamic = true
Static_Handler
Dynamic_Handler
– _thread : Periodic_Thread
+ operator()
T: Component, T::Criterion T::Criterion::HEADS
Multihead_Scheduling_List
<<bind>> Criterion = GEDF
Semaphore_Handler
+ operator int()
T: Component, T::Criterion T::Criterion::QUEUES, L = list_type
Scheduling_Multilist
<<bind>> Criterion = PEDF
Ordered_List
List
Multihead_Scheduling_Multilist
<<bind>> Criterion = CEDF

**Figure 42: UML class diagram of the real-time scheduling components on EPOS: Thread, Criterion, Scheduler, and Scheduling list classes.**

The `Scheduler` class in Figure 42, and `Criterion` sub classes,

[6] A trait is a metaprogramming artifact that renders a component characterization at compile time, so other components and metaprograms can reason on it.

which are shown in Figure 43, define the structure that realizes task scheduling. Usually, object-oriented OS scheduler implementations use a hierarchy of specialized classes of an abstract scheduler class. Sub classes specialize the abstract class to provide different scheduling policies (MARCONDES *et al.*, 2009). EPOS reduces the complexity of maintaining such hierarchy and promotes code reuse by detaching the scheduling policy (here represented by the `Criterion` sub classes) from its mechanism (*e.g.,* data structure implementations as lists and heaps). The data structure in the scheduler class uses the defined scheduling criterion to order the tasks accordingly. Each criterion class basically defines the priority of a task, which is later used by the scheduler to choose a task (`operator ()`), and other criterion features, as preemption and timing, for instance.

The `Scheduler` consults the information provided by the criterion class to define the appropriate use of lists and operations. Figure 43 shows the G-EDF, P-EDF, and C-EDF criterion classes that inherit from the RT_Common class, which defines the deadline, period, and WCET of a task. Each criterion defines two static variables that are later used by the Scheduling_Queue: QUEUES and HEADS. QUEUES defines the number of lists and HEADS defines the number of heads in each scheduling list. A head in a scheduling list represents a running task. For example, the G-EDF criterion has only one scheduling list (QUEUS is one) and HEADS is equal to the number CPUs in the current processor. P-EDF has QUEUES equal to the number of CPUs and HEADS equal to one. C-EDF combines features from the G-EDF and P-EDF criteria: QUEUES is equal to the number of clusters and HEADS is equal to the number of CPUs in each cluster. The `Variable_Queue` class defines the current queue of a task. The method `queue` is used by the appropriate `Scheduling_List` to get the current queue and perform a scheduling decision correctly. For G-EDF, the current queue is the current CPU in which the task is executing. For P-EDF and C-EDF, the current queue is the partition or the cluster in which the task was assigned, respectively. Figure 43 only shows the EDF-based criteria, however, the same principle is used for other schedulers, such as LLF, RM, SJF, and DM. Note that the `Priority` class overloads the operator parenthesis, which returns the current priority of a task (an integer).

Figure 44 shows an example of the interaction among `Criterion`, `Trait`, and `Scheduler` classes. The `Scheduler` receives a component as

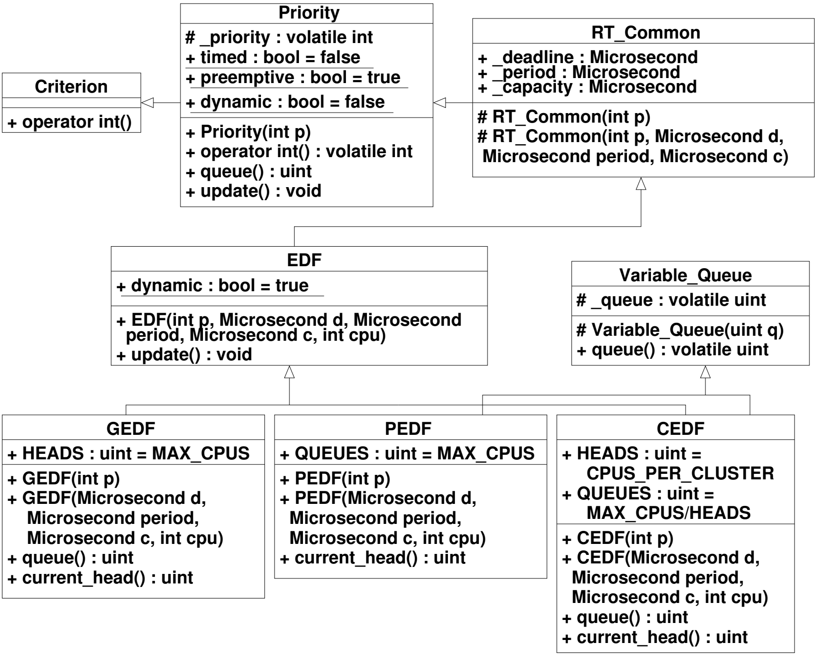

Priority
# _priority : volatile int
+ timed : bool = false
+ preemptive : bool = true
+ dynamic : bool = false
+ Priority(int p)
+ operator int() : volatile int
+ queue() : uint
+ update() : void
RT_Common
+ _deadline : Microsecond
+ _period : Microsecond
+ _capacity : Microsecond
# RT_Common(int p)
# RT_Common(int p, Microsecond d, Microsecond period, Microsecond c)
Criterion
+ operator int()
EDF
+ dynamic : bool = true
+ EDF(int p, Microsecond d, Microsecond period, Microsecond c, int cpu)
+ update() : void
Variable_Queue
# _queue : volatile uint
# Variable_Queue(uint q)
+ queue() : volatile uint
GEDF
+ HEADS : uint = MAX_CPUS
+ GEDF(int p)
+ GEDF(Microsecond d, Microsecond period, Microsecond c, int cpu)
+ queue() : uint
+ current_head() : uint
PEDF
+ QUEUES : uint = MAX_CPUS
+ PEDF(int p)
+ PEDF(Microsecond d, Microsecond period, Microsecond c, int cpu)
+ current_head() : uint
CEDF
+ HEADS : uint = CPUS_PER_CLUSTER
+ QUEUES : uint = MAX_CPUS/HEADS
+ CEDF(int p)
+ CEDF(Microsecond d, Microsecond period, Microsecond c, int cpu)
+ queue() : uint
+ current_head() : uint

**Figure 43: UML class diagram of the real-time scheduling components on EPOS: criterion sub classes.**

parameter (Thread in our case). This component defines a Criterion, which has specific features for each scheduling policy (*e.g.,* number of queues, preemption, timing, etc). A global criterion, such as G-EDF, uses a single scheduling list with multiple heads. A partitioned criterion, such as P-EDF, uses a scheduling list per processor, and a clustered criterion, such as C-EDF, uses a single scheduling list with multiple heads in each cluster. The Scheduler uses a Scheduling_Queue to specialize a specific implementation of a scheduling list, according to the selected criterion. Each scheduling list implements the functionalities used by the Scheduler, such as the insertion (insert) and removal (remove) of threads.

This new EPOS scheduling design makes the addition of any multicore scheduling policy straightforward and totally independent from other OS parts. For instance, a new scheduling policy must define the number of heads, the number of queues, add a new Scheduling_Queue specialization, and if it is dynamic, must implement the update

```
template <typename T, typename R = typename T::Criterion> //Base Scheduling_Queue class
class Scheduling_Queue: public Scheduling_List<T> {};

template <typename T> //Specialization for G-EDF
class Scheduling_Queue<T, Scheduling_Criteria::GEDF>: public Multihead_Scheduling_List<T> {};

template <typename T> //Specialization for P-EDF
class Scheduling_Queue<T, Scheduling_Criteria::PEDF>: public Scheduling_Multilist<T> {};

template <typename T> //Specialization for C-EDF
class Scheduling_Queue<T, Scheduling_Criteria::CEDF>: public Multihead_Scheduling_Multilist<T>{};
 .....
 .....
template <typename T>
class Scheduler: public Scheduling_Queue<T>
{
    typedef Scheduling_Queue<T> Base;
    typedef typename T::Criterion Criterion;
     ...
    void insert(T * obj) { Base::insert(obj->link()); }
    T * remove(T * obj) { return Base::remove(obj->link()) ? obj : 0; }
     ...
};

class Thread
{
        ...
        static Scheduler<Thread> _scheduler;
        ...
};
```

**Figure 44: An example about the interaction among the `Criterion`, `Trait`, and `Scheduler` classes.**

method. We use static metaprogramming techniques (CZARNECKI; EISENECKER, 2000), which does not incur in any run-time overhead.

### 4.2.1 Periodic Thread Operations

As stated before, the Periodic_Thread implementation uses an Alarm and a Semaphore to guarantee the re-execution of a thread. In this case, the Semaphore performs the sleep and wake up operations, instead of preventing concurrent accesses. After executing a code, the periodic thread calls the wait_next method to wait until the next period. Figure 45 shows an example of a periodic thread. The Periodic_Thread t is created in the main function, passing the func address as parameter to the constructor (line 5). Then, after each iteration, the periodic thread calls the wait_next method to wait until the next period (lines 12 to 14).

```
 1 void func();
 2
 3 int main()
 4 {
 5         Periodic_Thread *t = new Periodic_Thread(&func);
 6         t->join();
 7         delete t;
 8 }
 9
10 void func()
11 {
12         while(...) {
13                 do_work();
14                 Periodic_Thread::wait_next();
15         }
16 }
```

**Figure 45: Periodic thread code example.**

The wait_next method is a call to the p method of the periodic thread's Semaphore. Figure 46 depicts the UML sequence diagram of the sequence of calls starting from the wait_next method. The begin_atomic method prevents concurrent accesses by accessing a spinlock and disabling interrupts. The thread dispatch method releases the spinlock and enables the interrupts later. The Semaphore passes a Queue as an argument to the thread Sleep method. This method suspends the running thread, inserts this thread into the semaphore queue, chooses another thread to be ran by calling the Scheduler chosen method, and switch the context between them by calling the dispatch method. The scheduler suspend method removes and updates the head from the scheduling ordered list and the chosen method gets the new head. Note that the semaphore is private and is implemented as a common semaphore without priority inversion handling. EPOS originally has a semaphore that implements the immediate priority ceiling protocol, named IPC_Semaphore.

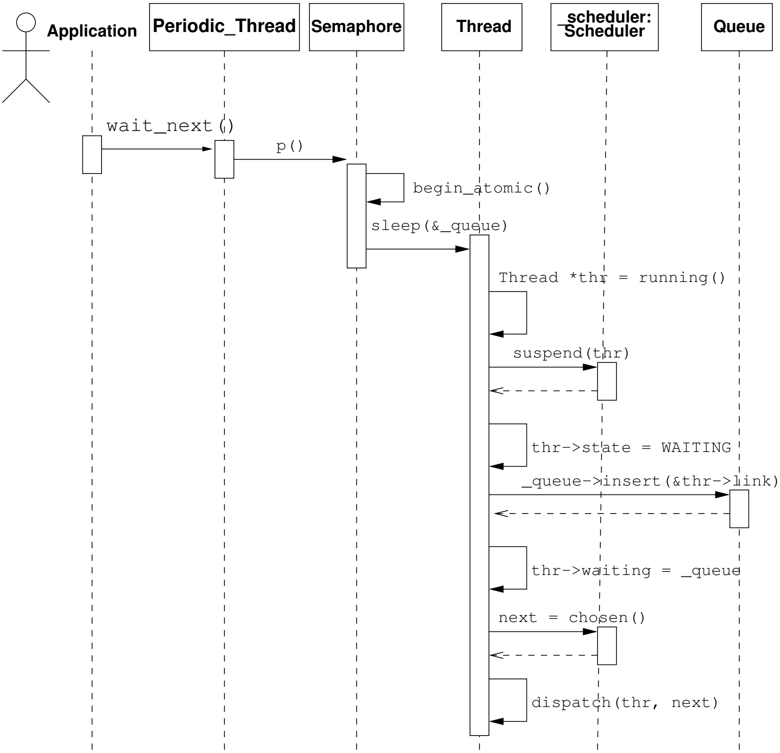


**Figure 46: UML sequence diagram of thread sleep method.**

The Alarm class is responsible for counting the time until a periodic thread can be released. To release a thread, the Alarm calls the v method of the Semaphore class, through the static or dynamic semaphore handler (see Figure 42). Figure 47 depicts the UML sequence diagram of the wake up operation. If a static criterion used, the update method is not called and the call from the Alarm goes directly to the Semaphore. Again, the begin_atomic method protects shared data by locking a spinlock and disabling interrupts. The **alt** label means an if/else condition and the **opt** label means an if clause. The thread dispatch method, called by the reschedule method, releases the spinlock and enables the interrupts. The Semaphore calls the wakeup method, which removes the Thread blocked on the semaphore's Queue and calls the scheduler to reinsert the thread into the scheduler list (resume method) according to the defined scheduling criterion (Criterion sub classes in Figure 43). Moreover, if the criterion is dynamic, the semaphore handler (Dynamic_Handler) calls the criterion update method to update the thread's priority before calling the thread's wakeup method. In the end, the wakeup method calls the thread reschedule to choose the highest priority task to be ran.

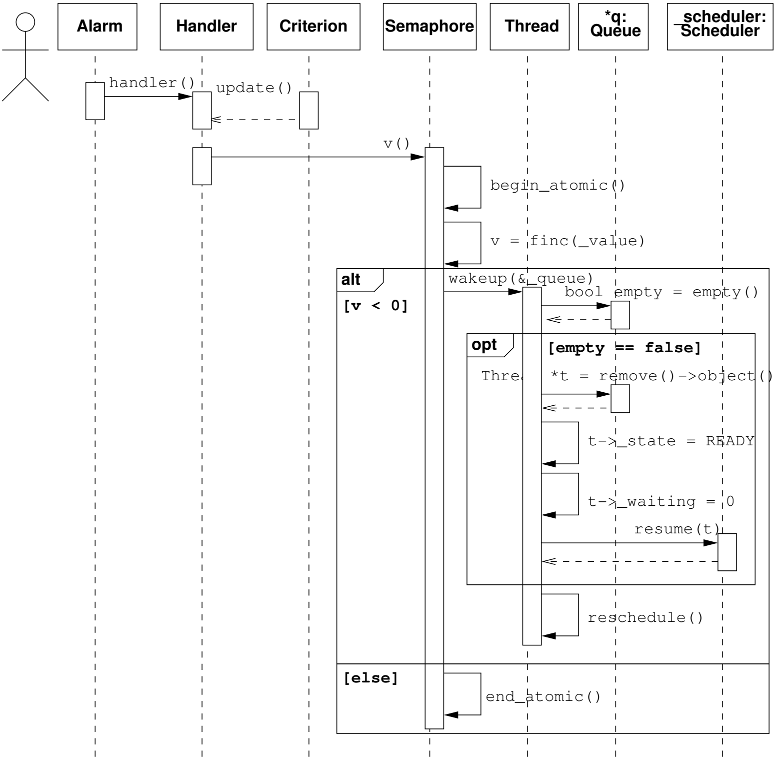


**Figure 47: UML sequence diagram of thread wake up method.**

Figure 48 shows the thread `reschedule` UML sequence diagram. We inserted a condition to test if an IPI is necessary in case of using a global scheduler. The system fires an IPI when the CPU that is executing the reschedule is not the CPU that is executing the lowest priority thread on the system. Then, the Interrupt Controller (IC) hardware mediator sends an IPI through the `ipi_send` method to switch the context on that CPU. A method in the `Scheduler` informs the lowest priority CPU. The CPU that receives the IPI message performs a thread rescheduling by calling the `choose` method of the `Scheduler` and then switching the context between the old and new thread. Moreover, we do not send an IPI when the running thread in the current CPU is an idle thread, because the scheduler may return another CPU that is also running another idle thread. It is important to highlight that the idle thread only calls the locking mechanism (*e.g.,* spinlock and disable interrupts) when there is a thread to be scheduled. Consequently, there is no influence in the system. If an IPI is not needed, the `reschedule` method just calls the `choose` method from the `Scheduler` to choose the next thread and switch the context between the running thread and the chosen thread. Note that the scheduling criterion statically sets a flag informing whether it is a global scheduler or not. As a consequence, the compiler processes the if condition in the `reschedule` method at compile time and thus does not incur in run-time overhead.

### 4.2.2 Context Switching

The thread `dispatch` method is responsible for switching context between the previously running and the chosen threads. The method verifies if a context switch needs to be performed by checking if the chosen thread is not the same as the running thread. If both threads are different, the method changes the state of the running thread to “ready”, the chosen thread state to “running”, and calls the CPU hardware mediator `switch_context` method to perform the context switch. Figure 49 presents the UML class diagram of the CPU hardware mediator. The mediator handles the most dependencies of process management. The inner class `CPU::Context` defines the execution context for each process architecture. The method `CPU::switch_context` is responsible for the switching context, receiving the old and new contexts. The CPU mediators also implement several functionalities as enabling and

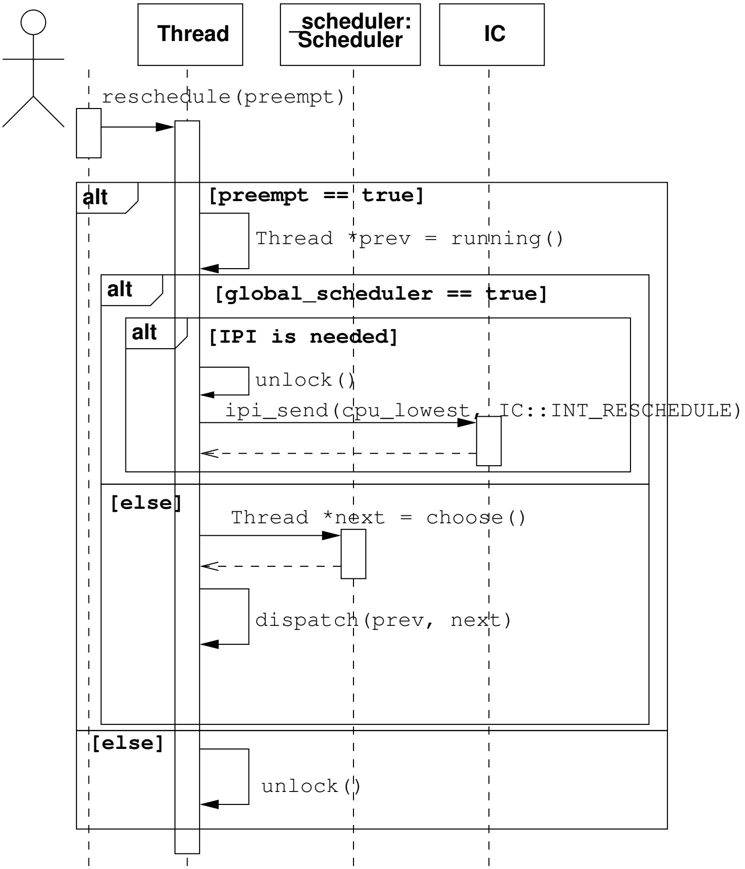


**Figure 48: UML sequence diagram of thread reschedule method.**

disabling interrupts and test and set lock operations. Each process architecture defines a set of registers and specific addresses, but the same interface remains. Thus, it is possible to keep the same operations for platform-independent components, such as threads, synchronizers, and alarms.

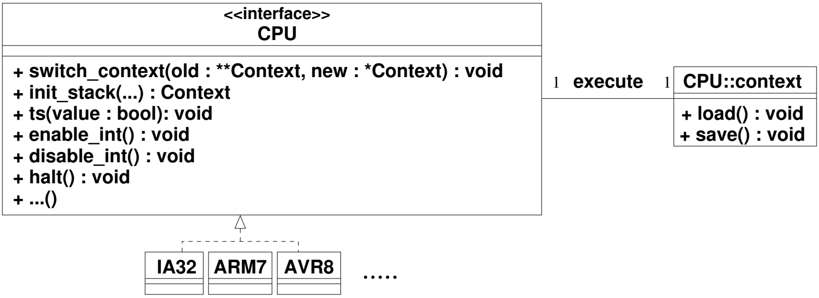


**Figure 49: CPU hardware mediator UML class digram.**

### 4.2.3 Alarm and Timer Interrupt Handler

The `Alarm` component handles timed-based events. The component uses a `Timer` hardware mediator class that abstracts the hardware timer. In a periodic event model, EPOS sets the hardware timer with a constant (configurable) frequency. When a new alarm event is registered, its interval is converted to timer *ticks*, with T = I / F, where T is the number of ticks, I is the desired interval, and F is the timer frequency (FRÖHLICH *et al.*, 2011). The `Timer` hardware mediator configures the hardware timer (*i.e.,* the Local-APIC timer configured in periodic mode on the IA32 architecture) during the system initialization. The mediator also supports the instantiation of several timers on top of the same physical timer (a multiplexing layer). The Local-APIC timer has a precision of microseconds (Intel Corporation, 2011). The `Alarm` instantiates a `Timer` during the system initialization. The alarm inserts all created events in an ordered and relative request queue. Thus, operations on the queue only affect its head, because all queue elements are relative to the first element. The `Alarm` component registers an interrupt handler that increments the tick counter, thus promoting every alarm in the event queue by a tick, at every hardware timer interrupt. The `Alarm` interrupt handler releases the head queue's event handler if there are no more ticks to count to that event. The event handler, when using a `Periodic_Thread`, releases the thread by calling the `v` semaphore method.

We changed the described alarm handler in two ways. First, we distributed the handler across all CPUs. Thereby, when the application creates a `Periodic_Thread`, the alarm constructor assigns the event handler of that `Periodic_Thread` to a specific CPU (the same CPU that schedules the periodic thread in case of a partitioned scheduler). Each different handler manages its own event list separately. Second, we changed the handler to release all events that reach 0 ticks in the same alarm interrupt handler. In this way, we guarantee that all events are released without any tick delay (*i.e.,* wait one or more ticks to be released) and that the OS always executes the $m$ highest priority threads.

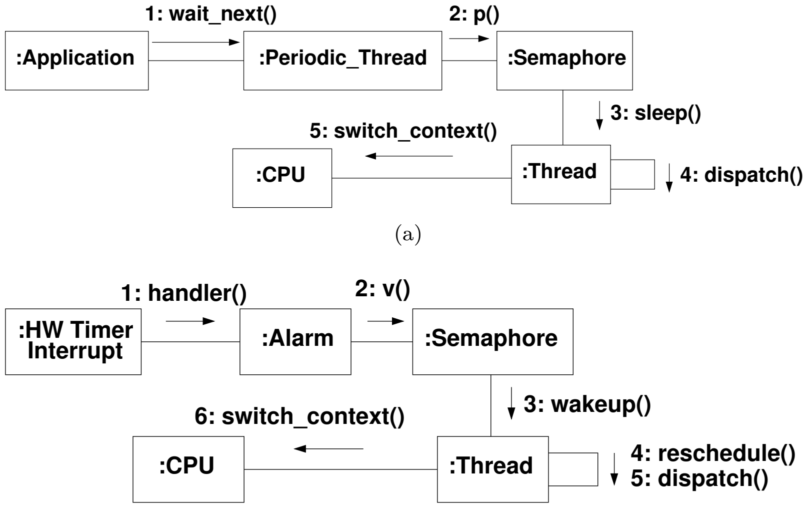
### 4.2.4 Summary of Real-Time Extensions and Overhead Sources

Figure 50 summarizes the sources of run-time overhead in EPOS. In Figure 50(b), message 1 and the `Alarm handler` method form the tick counting and thread release overheads. The `Thread dispatch` and `CPU::switch_context` methods constitute the context switching overhead. Finally, message 2 and the `Thread sleep` method in Figure 50(a) and messages 2 and 3 and `Thread wakeup/reschedule` methods in Figure 50(b) form the scheduling overhead.

In summary, we carried out the following extensions into the original EPOS real-time support:

- **Support for multicore real-time schedulers:** we designed and implemented a set of classes to support partitioned, clustered, and global schedulers. For each scheduler variant, a specialized scheduling list is chosen at compile time, depending on the scheduling criterion. For example, a partitioned scheduler uses individual scheduling lists for each processor, while global schedulers have only one global scheduling list. Clustered schedulers are a combination of partitioned and global approaches: each cluster is a partition and each partition has a global scheduling list. This design allows the addition of a scheduler variant of practically any scheduling policy;

- **Distributed alarm handler:** we have performed two modifications in the original EPOS alarm handler. First, we distributed the handler in all available CPUs. Each handler has its own private relative and ordered list. Second, we changed the handler to release all periodic threads that reached 0 ticks at the same



(a)

(b)

**Figure 50: UML communication diagram summarizing the sources of run-time overhead in EPOS. (a) Operations initiated by the periodic thread sleep operation. (b) Operations initiated by the hardware timer.**

interrupt. In this way, we are sure that the OS always executes the $m$ highest priority periodic threads;

- **IPI in global and clustered schedulers:** we have extended the IC hardware mediator to support IPI messages and added the IPI call into the thread `reschedule` method. The IPI message allows the implementation of virtually any global or clustered scheduler;

- **Semaphore dynamic handler:** we created a new semaphore handler for the `Periodic_Thread` class. The dynamic handler is responsible for updating the priority of a task when it is released, totally transparent to the rest of the system;

- **Separation of scheduling policy and mechanism:** we detached the scheduling policy from its mechanism by using static metaprogramming and creating several `Scheduling_Queue` specialization depending on the defined scheduling criterion (see

Figure 44). This design makes the creation of new schedulers straightforward;

- **New scheduling policies:** we implemented beyond the EDF, RM, and DM scheduling policies, the LLF and SJF scheduling policies. EPOS now supports global, partitioned, and clustered variants of EDF, RM, DM, LLF, and SJF. EPOS is the first RTOS designed from scratch that supports all of these policies with an original and elegant OOD.

In Chapter 5, we compare the run-time overhead of the EPOS multicore real-time infrastructure with LITMUS$^{RT}$. We also present a series of experiments demonstrating the schedulability ratio of several generated task sets inflated by the run-time overhead of EPOS and LITMUS$^{RT}$.

## 4.3 MEMORY MANAGEMENT

This and the next subsections present the memory management support in EPOS and the page coloring design and implementation.

Portable OSes face a challenge in terms of memory management: some computing platforms feature sophisticated MMUs, while other platforms do not provide any support to map and protect address spaces. EPOS careful design encapsulates details pertaining to address space protection, translation, and physical memory allocation into the MMU hardware mediators, which allows memory management components to be highly portable across virtually any platform, from simple microcontrollers to complex multicore processors (POLPETA; FRÖHLICH, 2004).

### 4.3.1 Original Memory Management in EPOS

Figure 51 shows part of the MMU mediators family. The `MMU_Common` parameterized class provides basic functions common to all architectures. Different classes specialize the base class implementing architecture-dependent functions. The `IA32_MMU` class implements the support for the 32-bit paging mode available on x86 processors, including the manipulation of page and directory tables. A grouping list implementing the buddy memory allocation strategy keeps track of the available physical memory. Each list element represents a free physical memory region. The MMU initialization method initializes the grouping list according to the available memory.

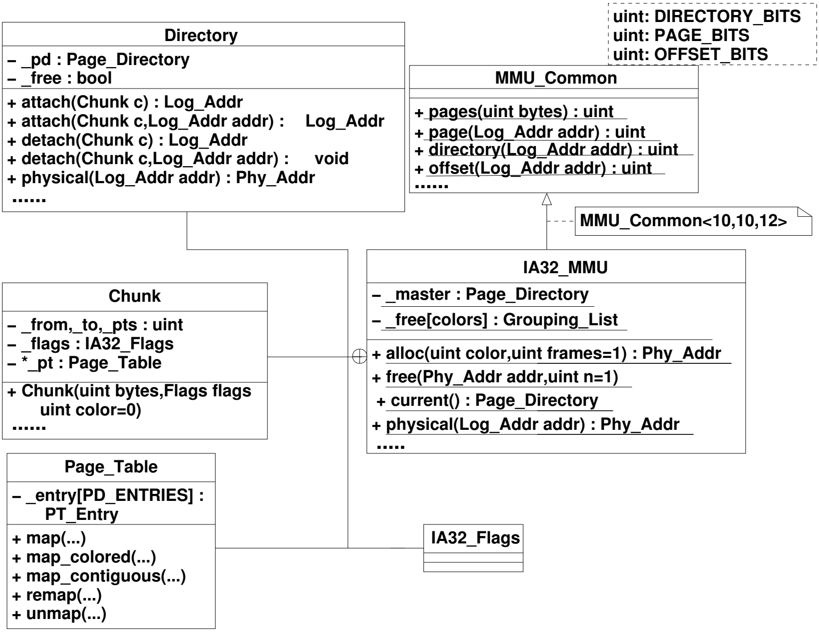

uint: DIRECTORY_BITS
uint: PAGE_BITS
uint: OFFSET_BITS
Directory
– _pd : Page_Directory
– _free : bool
+ attach(Chunk c) : Log_Addr
+ attach(Chunk c,Log_Addr addr) : Log_Addr
+ detach(Chunk c) : Log_Addr
+ detach(Chunk c,Log_Addr addr) : void
+ physical(Log_Addr addr) : Phy_Addr
......
MMU_Common
+ pages(uint bytes) : uint
+ page(Log_Addr addr) : uint
+ directory(Log_Addr addr) : uint
+ offset(Log_Addr addr) : uint
......
MMU_Common<10,10,12>
IA32_MMU
– _master : Page_Directory
– _free[colors] : Grouping_List
+ alloc(uint color,uint frames=1) : Phy_Addr
+ free(Phy_Addr addr,uint n=1)
+ current() : Page_Directory
+ physical(Log_Addr addr) : Phy_Addr
.....
Chunk
– _from,_to,_pts : uint
– _flags : IA32_Flags
– *_pt : Page_Table
+ Chunk(uint bytes,Flags flags
uint color=0)
......
Page_Table
– _entry[PD_ENTRIES] :
PT_Entry
+ map(...)
+ map_colored(...)
+ map_contiguous(...)
+ remap(...)
+ unmap(...)
IA32_Flags

**Figure 51: IA32 MMU hardware mediator.**

Figure 52 depicts the system components that deliver the main available memory to applications. A `Segment` is a chunk of memory that stores arbitrary code and data. When a system component creates a `Segment`, the `MMU::Chunk` inner class allocates the requested memory from the MMU grouping list and maps the allocated pages to corresponding page table entries. The `Address_Space` component is a container for memory chunks (*e.g.*, `Segments`), that manages the physical memory corresponding to a memory segment, thus keeping the `Segment` independent of a memory management policy. When a `Segment` is attached to an `Address_Space`, the `Address_Space` maps all previously allocated page tables to corresponding page directories through the `MMU::Directory` inner class.

Applications do not allocate memory directly from the

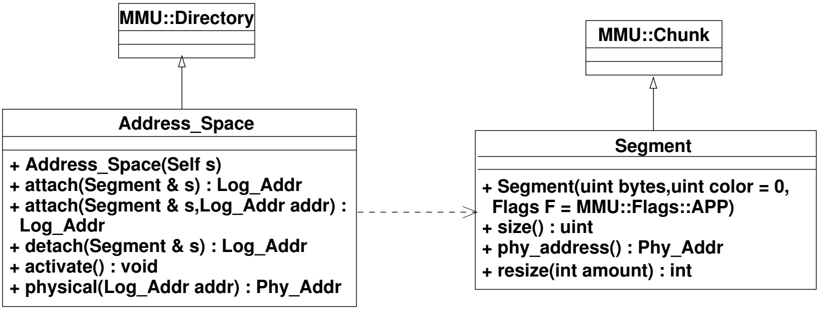

MMU::Directory
MMU::Chunk
Address_Space
+ Address_Space(Self s)
+ attach(Segment & s) : Log_Addr
+ attach(Segment & s,Log_Addr addr) : Log_Addr
+ detach(Segment & s) : Log_Addr
+ activate() : void
+ physical(Log_Addr addr) : Phy_Addr
Segment
+ Segment(uint bytes,uint color = 0, Flags F = MMU::Flags::APP)
+ size() : uint
+ phy_address() : Phy_Addr
+ resize(int amount) : int

**Figure 52: UML class diagram for address space and segment components.**

Address_Space and Segment components. Instead, two different heap class instances, which use the described memory management components, provide free memory for the OS and applications. Dynamic memory allocations from the OS, such as creation of thread stacks and system objects, go to the system heap, while requests from the application go to the application heap. During initialization, each heap frees its pre-defined (configurable) size by using the Address_Space and Segment components, as demonstrated in the next subsection.

### 4.3.2 Page Coloring Support

To support page coloring in EPOS, we changed the memory management system components, the MMU hardware mediator, and heap initialization. Figure 53 shows the new MMU trait class that enables page coloring in the system.

```
template <> struct Traits<IA32_MMU>: public Traits<void>
{
    static const bool page_coloring = true;
    static const bool user_centric = true; //false = OS centric
    static const unsigned int colors = 4;
};
```

**Figure 53: IA32 MMU trait class responsible for enabling page coloring.**

When the developer enables page coloring, the MMU grouping list becomes an array of colors grouping lists and the application heap becomes an array of colors heaps, where colors is the number of colors defined in the trait class (line 5 in Figure 53). If the defined number of colors is less than the maximum number of colors, then the colors are grouped, forming a super color[7] (LIN *et al.*, 2008).

The MMU mediator fills each grouping list with pages associated with the corresponding color during the initialization phase. In the same way, the Segment component provides, in its interface, a way to specify from which color (*i.e.*, from which MMU grouping list) a heap should allocate memory. Since each MMU grouping list has a set of physically mapped pages, with the same color, we make sure that each heap has a chunk of logical addresses that map to physical pages with the same color.

Figure 54 exemplifies the application heap initialization. The init process creates, for each color, a Segment with the heap size and attaches this Segment to an Address_Space. The free method receives the logical address returned by the attach method, and then initializes the heap free space. Moreover, the color method sets the heap color. To properly release a memory region, we use the defined color to find from which heap a memory address was allocated, as will be explained in the next subsection. We initialize the system heap in the same way, but we always allocate the memory for the system heap from the same MMU grouping list (only one *super color*). It is possible to include several system heaps with different colors as well, but this would require changes in the source code of the system components to define the appropriate color. Below, we propose two different approaches to specify the heap for a dynamic allocation: user-centric and OS-centric.

### 4.3.3 User-Centric Page Coloring

The C++ new and delete operators perform the dynamic memory allocation operations. We overload the EPOS new operator to include annotations (*i.e.*, colors) that define from which heap data should be allocated. Figure 55 shows how the new and delete operators use these annotations. An enumeration defines the available colors. We changed the new operator to receive, in addition to the requested num-

[7] Super color = page color % max. number of colors. We use color to refer to super color hereafter.

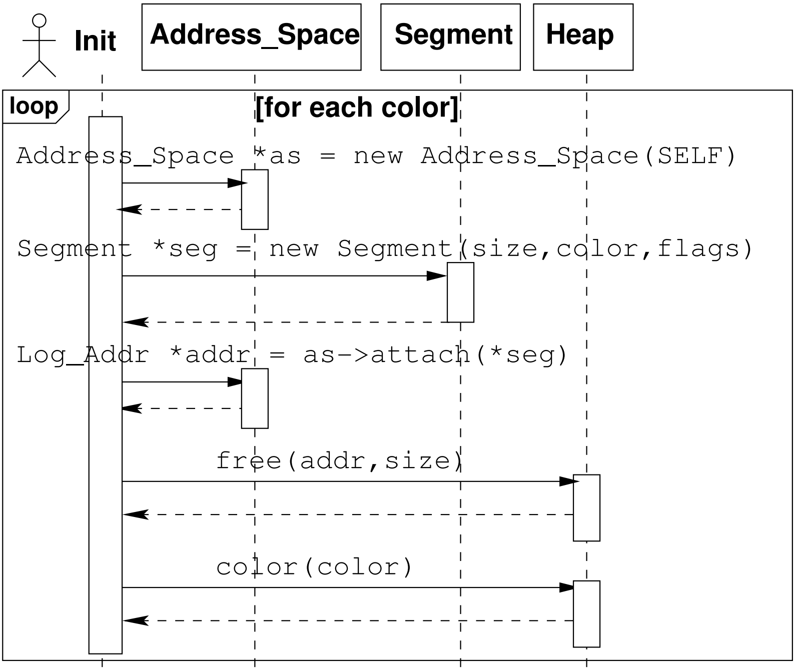

Init
Address_Space
Segment
Heap
loop
[for each color]
Address_Space *as = new Address_Space(SELF)
Segment *seg = new Segment(size,color,flags)
Log_Addr *addr = as->attach(*seg)
free(addr,size)
color(color)

**Figure 54: UML sequence diagram of the colored application heap initialization.**

ber of bytes, the color number. If the application does not specify a color, the `new` operator uses the color 0. Additionally, the `new` operator does not allocate memory if the requested color is greater than the maximum color, defined in the trait class (see Figure 53).

The heap allocates eight bytes (two integers) more than the requested size: the first integer contains the data size (the *bytes* argument) and the second integer contains the color number. Thus, the `delete` operator releases the allocated memory using the correct heap and size by reading the two integers. The overload of the `new` operator is part of the ISO C++ standard and is supported by any standard C++ compiler. Figure 53 shows the MMU trait class that defines the use of the user-centric approach.

```
typedef enum { //colors definitions
    COLOR_0,
     ...
    COLOR_128,
} colored_alloc ;
//overload of the new operator
void * operator new(size_t bytes, colored_alloc c = COLOR_0) {
     //perform memory allocation from the heap defined by c
}
//examples of dynamic memory allocation from the application
int *data = new (COLOR_1) int[50];
delete[] data;
```

**Figure 55: Overload of the EPOS `new` operator.**

### 4.3.4 OS-Centric Page Coloring

When the developer enables the OS-centric approach, the `new` operator ignores the annotated color (there is no need to pass a color). EPOS transparently chooses from which heap data should be allocated by using a thread ID to access a colored heap: ID % Traits<MMU>::colors. It is important to highlight that the choice of either user-centric or OS-centric is made at compile time, without any run-time overhead. The thread constructor assigns a unique ID to each thread; however, in this approach, the user should properly define the number of colors to avoid allocations from the same heap, giving consideration to the number of threads in the target application. The choice of the best number of colors for an application is not straightforward, because it depends on the WSS of each task and how each task uses its own WSS. It is important to highlight that our objective is not optimize the number of colors, but to provide an efficient mean to partition the shared cache. In Chapter 6, we empirically evaluate the impact of cache partitioning on multicore real-time schedulers.

### 4.3.5 Summary of Memory Management Extensions

In summary, we proposed the following extensions to the original EPOS memory management structure:

- **Different grouping list in the MMU class:** we created different MMU grouping lists, each of them composed of free pages with the same color. At the initialization phase, the MMU hardware mediator inserts all pages mapped to the same color into the appropriate grouping list, respecting the available memory size;

- **Colored application heaps:** we created $N$ different application heaps, where $N$ is the defined number of colors. The logical address for each application heap is mapped to a set of pages with the same colors. In practice, each application heap requests free memory from a unique MMU grouping list.

- **Colored system heap:** the system heap also requests memory from a specific grouping list. Thus, memory allocation requests from EPOS do not interfere with application data;

- **User- and OS-centric memory allocation:** we provide two mechanisms for applications to allocate memory. The user-centric approach overloads the C++ new and delete operators to allocate memory from a specific colored heap. The new operator receives the color from which memory should be allocated. The OS-centric approach automatically allocates memory from a colored application heap using the thread ID as color index.

In Chapter 6, we present an evaluation of the cache partitioning impact on partitioned, clustered, and global real-time schedulers using the proposed page coloring mechanism. We also analyze whether the RTOS interferes with real-time tasks by assigning isolated partitions to the internal OS data structures. We also use the proposed page coloring mechanism to evaluate our color-aware task partitioning algorithm in Chapter 7.

# 5 RUN-TIME OVERHEAD EVALUATION

In the previous chapter we designed and implemented an original multicore real-time infrastructure in the EPOS component-based RTOS. In this chapter[1], we compare EPOS to LITMUS$^{\mathrm{RT}}$ in terms of run-time overhead and the impact of both OSes on the schedulability ratio of generated task sets for G-EDF, C-EDF, and P-EDF schedulers. We measure the run-time overhead of EPOS and LITMUS$^{\mathrm{RT}}$ using a modern 8-core processor. In addition, we provide a comparison between the ideal (*i.e.,* no overhead) G-EDF and P-EDF using eight state-of-the-art G-EDF schedulability tests and three P-EDF partitioning heuristics for up to 100 processors for HRT systems.

In summary, the main contributions of this chapter are:

- We show that the EPOS run-time overhead, when incorporated into G-EDF, C-EDF, and P-EDF schedulability tests, can provide HRT guarantees close to the theoretical schedulability tests. Moreover, in some cases, the performance of G-EDF considering the overhead in EPOS is superior to P-EDF considering the overhead in LITMUS$^{\mathrm{RT}}$, which proves that the run-time overhead plays an important role on the G-EDF, C-EDF, and P-EDF schedulability analyses;

- A comparison in terms of task set schedulability ratio for HRT tasks between P-EDF, C-EDF and G-EDF, also considering the OS overhead. P-EDF has obtained the same or better performance than G-EDF and C-EDF for all analyzed scenarios. In our experiments, P-EDF, C-EDF, and G-EDF had the same behavior for task sets composed only of heavy tasks, mainly due to G-EDF's schedulability test bounds. We observed a slight improvement in G-EDF for this heavy tasks scenario compared to related work (CALANDRINO *et al.*, 2006), due to the use of up to date G-EDF schedulability tests (BERTOGNA; CIRINEI, 2007; BARUAH *et al.*, 2009).

[1] Contents of this chapter appear in a preliminary version in the following published paper:
G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, Implementation and evaluation of global and partitioned scheduling in a real-time OS, Real-Time Systems, v. 49, p. 1-48, 2013.

- We measure the CPMD on a modern 8-core processor, with shared L3 cache, using HPCs. We use the obtained values to compare P-EDF, C-EDF, and G-EDF through the weighted schedulability metric (BASTONI *et al.*, 2010b).

The rest of this chapter is organized as follows. Section 5.1 presents the experiments description. Section 5.2 to Section 5.8 show the results of the measured IPI latency, context switch, scheduling, tick counting, and thread releasing overheads of EPOS and LITMUS$^{\text{RT}}$. Section 5.9 presents the schedulability ratio of P-EDF, C-EDF, and G-EDF considering the run-time overhead in both OSes. Finally, Section 5.10 discusses the main results.

## 5.1 EXPERIMENTS DESCRIPTION

To measure the OS overhead and the schedulability ratio of G-EDF, C-EDF and P-EDF, we randomly generated task sets with different distributions similar to (BAKER, 2003; BAKER, 2005a) and (BRANDENBURG; ANDERSON, 2009). For generating tasks periods (all values are in ms), we used a uniform distribution between [3, 33] (short), [10, 100] (moderate), and [50, 250] (long). For generating tasks utilizations, we used a uniform distribution between [0.001, 0.1] (light), [0.1, 0.4] (medium), and [0.5, 0.9] (heavy), and a bimodal distribution (combining two uniform distribution) between [0.001, 0.1] and [0.5 ,0.9], with probabilities 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy), respectively. There are in total 18 combinations of periods and utilizations. Based on the generated task's period and utilization, we defined the task's WCET (before adding the OS overhead).

For measuring the overhead associated with OS activities, we fixed the number of tasks to 5, 15, 25, 50, 75, 100, and 125, and used the light uniform utilization with short periods to generate the tasks. Each task sums a local variable in a loop of 50 repetitions. The same task function code is used in EPOS and LITMUS$^{\text{RT}}$. The objective is to stress the OS. We used these numbers of tasks to measure the run-time overhead, because they represent the range of tasks in our generated synthetic task sets. We also added a warm-up code (*i.e.,* a loop performing a nop operation) to make sure that the scheduling and alarm data structures are in the same state before the beginning of the application. Then, we applied three G-EDF sufficient schedulability tests (GOOSSENS *et al.*, 2003; BAKER, 2005a; BERTOGNA *et al.*, 2005). A task set was considered feasible if it had passed in at least one test. We then used the generated task sets to measure the overhead of EPOS and LITMUS$^{\mathrm{RT}}$. We executed each task set for 100 times on the Intel i7-2600 processor (see Table 12) for LITMUS$^{\mathrm{RT}}$ and EPOS, and extracted the sampled WCET for each overhead from these executions.

**Table 12: Intel i7-2600 processor features and LITMUS$^{RT}$ version.**

| | |
|---|---|
| Clock speed | 3.4 Ghz |
| Cores | 4 |
| Hyper-threading (SMT) | 2 per core (8 logical cores) |
| L1 cache | 4 x 64 KB 8-way set associative (32 KB separate data and instructions caches) |
| L2 cache (non-inclusive) | 4 x 256 KB 8-way set-associative (unified) |
| L3 cache (inclusive) | 8 MB 16-way set-associative (unified) |
| LITMUS$^{\mathrm{RT}}$ version | 2012.1 kernel 3.0 |

To determine the schedulability of a task set that considers OS overhead, we first inflated the WCET of each task by adding the measured overheads, using the preemption-centric interrupt accounting method (BRANDENBURG, 2011) as described in Section 2.5, and then we verified if a task set is schedulable or not for the three schedulers (G-EDF, P-EDF, and C-EDF). The additional overhead for a task is dependent on the number of tasks in the task set. We verified the number of tasks and associated it with the measured overhead interval. For example, the additional overhead for a task set with 20 tasks is the measured overhead for 15 tasks, since the task number is between 15 and 25. Moreover, we used a step function to account for the overhead in the schedulability analysis. For instance, when accounting for the overhead of 50 tasks, and the overhead for 50 tasks is lower than the overhead for 25 tasks, we considered the overhead of 25 tasks.

A task set is considered to be schedulable in the G-EDF if it passes for at least one of the eight sufficient schedulability tests[2] (GOOSSENS *et al.*, 2003; BAKER, 2005a; BAKER, 2003; BARUAH, 2007; BAKER; BARUAH, 2009; BERTOGNA *et al.*, 2005; BERTOGNA; CIRINEI, 2007; BARUAH *et al.*, 2009). As in (BRANDENBURG; ANDERSON, 2009), we did not use the Baruah's test (BARUAH, 2007) for light uniform utilization due to high processing time caused by its pseudo-polynomial behavior. We created tasks until reaching a fixed utilization cap (from 2 to 8, in steps of 0.1). For each utilization, we defined a slack related to the utilization cap. For example, a slack of 0.05 specifies that a utilization cap $U$ of a task set $\tau$ is always between the interval $U$ - 0.05 and $U$. Thus, we make sure that the utilization values are always between two consecutive caps. The slacks used were 0.07, 0.07, and 0.1 for light, medium, and heavy utilization distributions, respectively. We generated 1000 task sets for each utilization cap.

For the P-EDF algorithm, we first partitioned the task set using three partitioning algorithms (FFD, BFD, and WFD) and then applied the EDF test (LIU; LAYLAND, 1973) for each partition (eight in total). A task set is schedulable if all the eight partitions pass the test and at least one partitioning algorithm correctly partitions the task set. For C-EDF, we first partitioned the task set using the same partitioning algorithms as in P-EDF, and then applied the same eight sufficient schedulability tests of G-EDF to each cluster. We defined four cluster with two processors in each cluster (sharing the L2 cache). A task set is schedulable if the four partitions pass in at least one of eight G-EDF sufficient tests and at least one partitioning algorithm correctly partitions the task set into the four clusters.

## 5.2 TRACING OVERHEAD

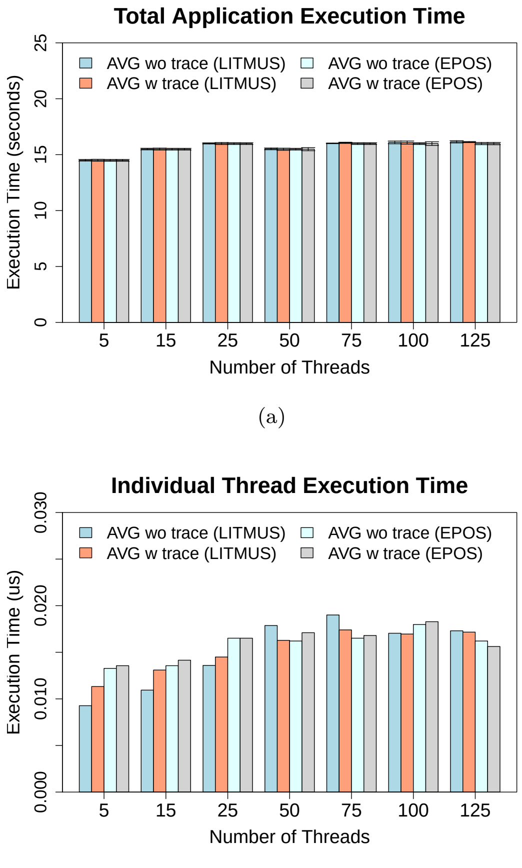
We measured the context switch, release, tick, and scheduling overheads and IPI latency for EPOS and LITMUS$^{RT}$ on the Intel i7-2600 processor. To record the overheads, in EPOS, we use the processor TSC and in LITMUS$^{RT}$, the tracing support accomplished by Feather-Trace (see Section 3.2.4) (BRANDENBURG; ANDERSON, 2007). First of

[2] We used the open-source implementation of the eight G-EDF schedulability tests available at `http://www.cs.unc.edu/~bbb/diss/`. We changed the code to allow the tests to be executed in parallel in a cluster and extended it to support also P-EDF and C-EDF partitioning heuristics and EDF uniprocessor test. The new code is also available online at `http://epos.lisha.ufsc.br`.

all, we designed two experiments to inspect whether the Feather-Trace introduces a considerable overhead to LITMUS$^{\text{RT}}$ or not and to verify the overhead to read and store the TSC in EPOS. We measured the time to complete all threads (we call the total application execution time) and the individual execution time of each thread (varying the number of threads from 5 to 125 as explained in the previous section) in LITMUS$^{\text{RT}}$ and EPOS with and without the tracing support. We used the TSC to measure the execution time both in LITMUS$^{\text{RT}}$ and EPOS. We executed each task set for 100 times in LITMUS$^{\text{RT}}$ and EPOS and extracted the average execution time from these executions. Figure 56(a) shows the average total application execution time. The total application execution time is similar for all number of threads. The variation observed among the number of threads comes from the different period lengths in each task set. Figure 56(b) shows the average execution time of each thread. We can note that the tracing support does not introduce a considerable overhead to the threads' execution time. We removed some outliers from LITMUS$^{\text{RT}}$. We considered outliers those values that increased the average by more than 1000%. It is important to highlight that our CPU-bound workload, described in the previous Section, may not induce all relevant overhead in Linux. Although it is not the focus of this work, impacts on the memory system when I/O buffers are full, such as disk and network buffers, can increase the WCET of a task.

## 5.3 CONTEXT SWITCH OVERHEAD EVALUATION

For measuring the context switch overhead in EPOS, we configured a test case composed of two threads *a* and *b*. Thread *a* sets the TSC before switching the context (Thread `dispatch` method, as described in Section 4.2.2) and thread *b* reads the TSC and calculates the difference, which represents the total context switch time for thread *a*. The context switch is performed on the same CPU. Thus, we can isolate the exact time that a context switch takes. Tick, releasing, and scheduling overheads are measured using the previously described methodology (*i.e.,* synthetic task sets generated using the light uniform distribution). These overheads account for all the overhead associated with the variation on the number of tasks in EPOS. We measured in



(a)

(b)

**Figure 56: Tracing overhead in LITMUS$^{RT}$ and EPOS. (a) Total application execution time and (b) individual thread execution time.**

total 5,000,000 context switches and extracted the worst-case[3] and average times from these executions. For LITMUS$^{RT}$, we measured the worst-case and average times by using Feather-Trace and running the task sets with fixed number of tasks, as presented previously. The to-

[3] From now on, whenever we refer to worst-case we mean the observed worst-case from the experiments.

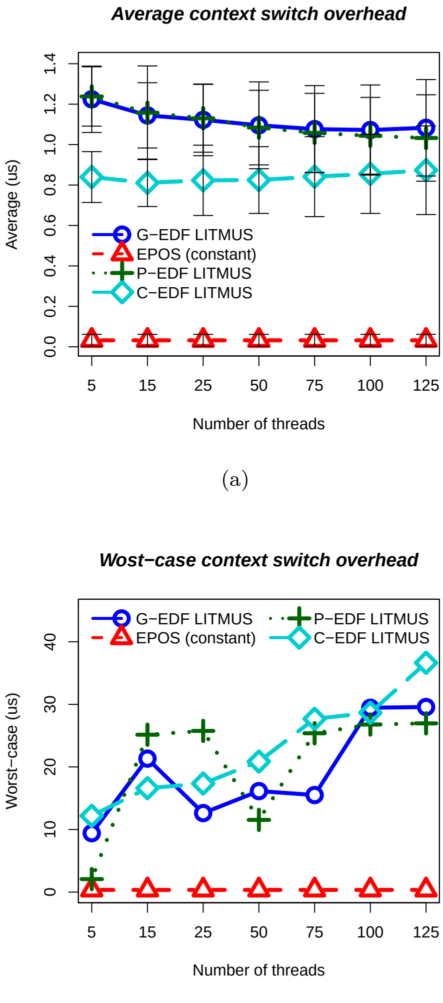
tal execution time is about 16 seconds, since the greatest period of all periodic tasks is 33 ms and each periodic thread is repeated 500 times.

Figure 57 shows the average and worst-case context switch overhead. The $x$-axis shows the number of threads and the $y$-axis the measured execution time in µs. The error bars represent the observed standard deviation. The average context switch overhead for EPOS is 0.03 µs and the worst-case is 0.3 µs. For LITMUS$^{\mathrm{RT}}$, the average context switch overhead is about 1.2 µs both for P-EDF and G-EDF and about 0.85 µs for C-EDF. The standard deviation ensures that the average context switch overhead is mostly the same for the three schedulers. For the worst-case context switch overhead, there is a high variation of the observed execution times, from 9.4 µs to 29.56 µs in G-EDF, from 2 µs to 26.95 µs for P-EDF, and from 12.19 µs to 36.63 µs for C-EDF.

EPOS performs a context switch up to 76.3 times faster than the LITMUS$^{\mathrm{RT}}$. For algorithms with a high rate of preemptions and context switches, such as fairness-based schedulers (BARUAH *et al.*, 1996; ANDERSON; BLOCK, 2000; ANDERSON *et al.*, 2003; SRINIVASAN *et al.*, 2003; LEVIN *et al.*, 2010), the use of an RTOS with low context switch overhead certainly improves the task set schedulability ratio, as will be shown in Section 5.9. However, it is important to highlight that we removed a few (from two to eight) outliers from LITMUS$^{\mathrm{RT}}$ measurements. For example, we once obtained a worst-case context switch time of up to 2.000 µs. As stated by (BRANDENBURG *et al.*, 2008), this outlier may be due to a set of factors, such as warm-up effects in the instrumentation code and non-deterministic aspects of Linux (BRANDENBURG *et al.*, 2008). This result corroborates the hypothesis that Linux causes interference for applications parts, harming the predictability needed in HRT systems (BRANDENBURG *et al.*, 2008), as will be observable in the next measurements.

## 5.4 IPI LATENCY EVALUATION

An Inter-Process-Interrupt (IPI) is issued to send reschedule requests from a core that is releasing a task to another core that must schedule that task. IPI rescheduling messages are common in global schedulers, such as G-EDF and C-EDF. In partitioned schedulers, such as P-EDF, if each core serves all release interrupts of those tasks that



(a)

(b)

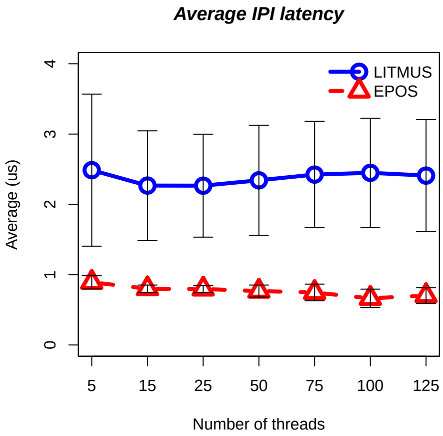
**Figure 57: Average (a) and worst-case (b) context switch overhead.**

have been assigned to it, there is no need to send IPI messages. We consider only the IPI latency for the G-EDF scheduler. Since the C-EDF scheduler uses IPI within each cluster, we use the IPI latency observed in G-EDF when accounting for IPI latency in the schedulability of C-EDF. Low IPI latency is important, because it delivers the interrupt message faster, which affects the preemption delay in the core that is receiving the interrupt message. The presence of memory transfers and other IPIs messages on the shared bus or on the on-chip point-to-point network can delay an IPI message. Furthermore, if the core that is receiving an IPI has disabled interrupts, the IPI is also delayed (IPI-fly-time) (BRANDENBURG, 2011).

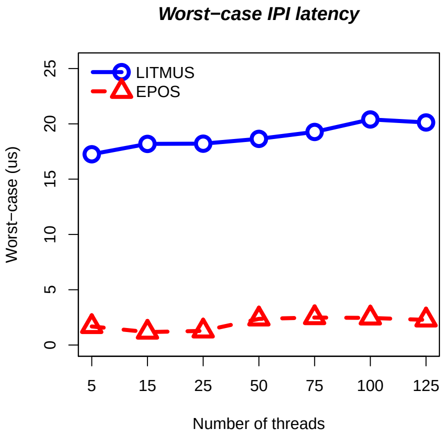
We measured the IPI latency in EPOS and LITMUS$^{RT}$ using the methodology described in Section 5.1 (varying the number of tasks). Figure 58 shows the obtained average and worst-case IPI latency for EPOS and LITMUS$^{RT}$. The $x$-axis shows the number of threads and the $y$-axis the measured IPI latency in µs. The error bars represent the observed standard deviation. We removed a few outliers from the obtained values for LITMUS$^{RT}$ (from two to seven). The worst-case IPI latency in EPOS is up to 15 times smaller than in LITMUS$^{RT}$ (for 15 tasks). This difference between EPOS and LITMUS$^{RT}$ is mainly caused by the `IC` hardware mediator and EPOS' design. The `ipi_send` method is "diluted" into the application code at compile time. There are no software layers between the application and OS, only embedded assembly code. Moreover, we used the EPOS library mode, in which the system is linked with the application, avoiding the overhead of system calls. EPOS also supports a kernel mode, which creates a system call layer between the application and OS. The system designer can choose the best configuration that fits the application requirements. Our focus is on embedded systems, which mostly are single application. That is the reason we chose the library mode. Another aspect that contributes to the smaller IPI latency is the smaller scheduling overhead (see Figure 59), in which interrupts are disabled. We can also note a slight decreasing in the average values for EPOS. However, the greater the number of tasks, the higher the standard deviation. In LITMUS$^{RT}$, we observed a standard deviation comparable to the average case values.

## 5.5 SCHEDULING OVERHEAD EVALUATION

We measured the G-EDF, C-EDF, and P-EDF scheduling overheads in EPOS and LITMUS$^{RT}$. The `sleep`, `wakeup`, and `reschedule` thread methods, including the list operations and thread



(a)



(b)

**Figure 58: Average (a) and worst-case (b) IPI latency.**

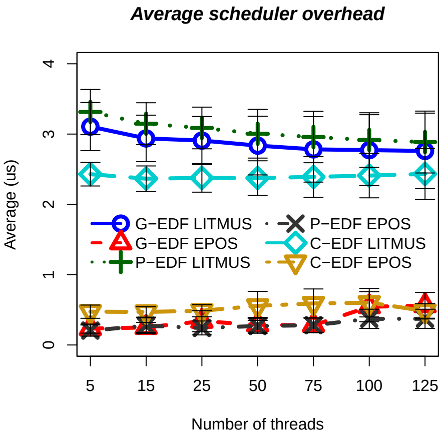
state changes, as demonstrated in the UML sequence diagrams of Section 4.2.1, are accounted for the scheduling overhead in EPOS. In EPOS, we measured the scheduling overhead using the processor's

TSC and in LITMUS$^{\mathrm{RT}}$, using the Feather-Trace.

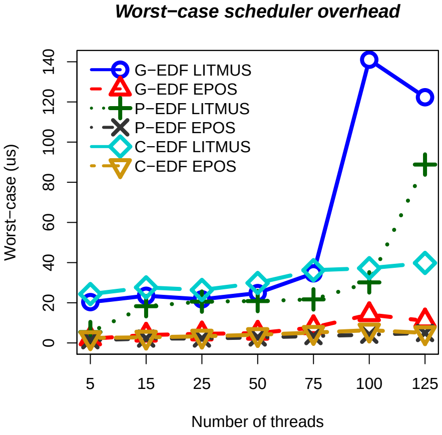
Figure 59 shows the average and worst-case scheduling overhead for both P-EDF, C-EDF, and G-EDF in EPOS and LITMUS$^{\mathrm{RT}}$. The $x$-axis shows the number of threads and the $y$-axis the measured execution time in µs. The error bars represent the observed standard deviation. For instance, the average scheduling overhead of EPOS and G-EDF scheduler for 100 tasks is about 0.5 µs with a standard deviation of about 0.2 µs. In Figure 59(b), the observed worst-case scheduling overhead for EPOS increases after 75 threads, mainly in G-EDF scheduler. This is due to the scheduling list: insertion and removal operations take more time, because there are more threads in the list (time complexity in the worst-case of *O(n)*). For P-EDF, on the other hand, as the threads are evenly distributed across the cores and each core has its own ready scheduling list, we did not observe a considerable variation in the overhead. C-EDF had a performance between P-EDF and G-EDF, as expected, because it uses global lists within each cluster. For LITMUS$^{\mathrm{RT}}$, as the number of threads increases, the unpredictability of Linux increases as well. C-EDF in LITMUS$^{\mathrm{RT}}$ had a similar behavior of C-EDF in EPOS. For G-EDF, EPOS was up to 11.10 times faster than LITMUS$^{\mathrm{RT}}$, for P-EDF it was up to 21.05 times faster, and for C-EDF it was up to 7.21 times faster. We again removed few outliers in the LITMUS$^{\mathrm{RT}}$ measurements, as explained before.

## 5.6 TICK COUNTING OVERHEAD EVALUATION

We measured the tick counting overhead in EPOS and LITMUS$^{\mathrm{RT}}$ using the methodology described in Section 5.1. The tick counting overhead in EPOS is formed by the call to the `Alarm handler` method performed by the `Timer` hardware mediator and by the increment of a variable and the access of register to get the appropriate CPU ID performed by the `Alarm handler`. Thus, the tick counting overhead in EPOS does not depend on the number of threads. Figure 60 shows the obtained values for EPOS and LITMUS$^{\mathrm{RT}}$. The $x$-axis shows the number of threads and the $y$-axis the measured execution time in µs. The error bars represent the observed standard deviation. The tick counting overhead in LITMUS$^{\mathrm{RT}}$ increases considerably after 75 threads. For C-EDF, the tick overhead surprisingly decreased. This corroborates the unpredictability of the Linux-based



(a)



(b)

**Figure 59: Average (a) and worst-case (b) scheduling overhead.**

real-time patches, as stated by Brandenburg (BRANDENBURG *et al.*, 2008). Also, the standard deviation in LITMUS$^{RT}$ is much higher than in EPOS. The worst-case overhead in EPOS is 0.248 µs and the average overhead is 0.051 µs with a standard deviation of 0.027 µs. The tick counting overhead in EPOS is up to 84 times faster than in LITMUS$^{RT}$.

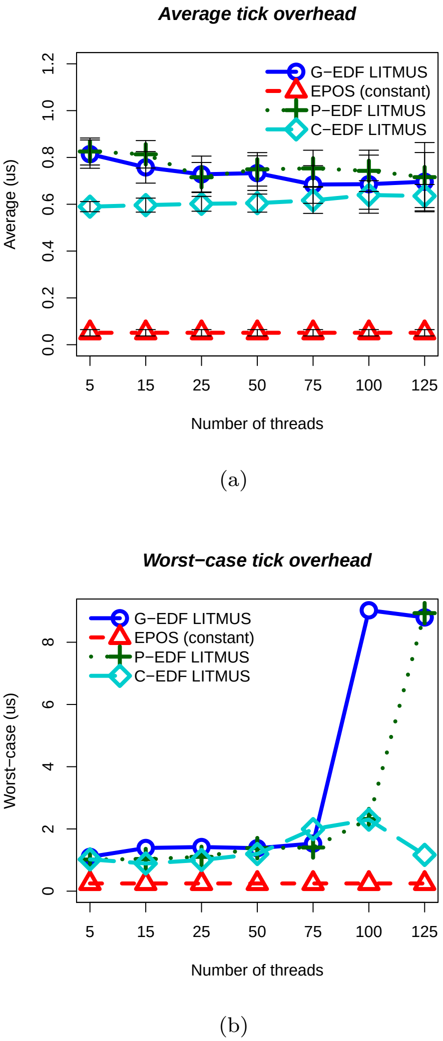


**Figure 60: Average (a) and worst-case (b) tick counting overhead.**

## 5.7 THREAD RELEASE OVERHEAD EVALUATION

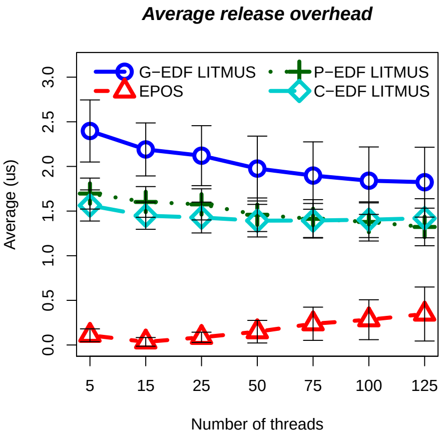
We measured the thread release overhead in EPOS and LITMUS$^{RT}$ using the methodology described in Section 5.1 (varying the number of tasks). In EPOS we used the TSC and in LITMUS$^{RT}$ the tracing support (feather-trace). The `Alarm handler` method performs all the operations related to the thread release overhead in EPOS (see Section 4.2.3). We did not measure separate overheads for the P-EDF, C-EDF, and G-EDF schedulers in EPOS, because thread releasing operations are the same for the three schedulers.

Figure 61 shows the average and worst-case release overhead. The $x$-axis shows the number of threads and the $y$-axis the measured execution time in µs. The error bars represent the observed standard deviation. For example, the average overhead for 125 threads in EPOS is about 0.34 µs with a standard deviation of about 0.6 µs. Considering the worst-case release overhead, in Figure 61(b), the observed worst-case times increase according to the number of threads in the system: the alarm `handler` releases more threads in the same handler, affecting the overhead. The G-EDF scheduler in LITMUS$^{RT}$ had a similar behavior, increasing the overhead as the number of threads also increases. The P-EDF scheduler did not present a considerable variation after 50 threads. The C-EDF scheduler in LITMUS$^{RT}$ had a similar behavior to G-EDF for the worst-case scenario, and a similar behavior to P-EDF for the average-case scenario.

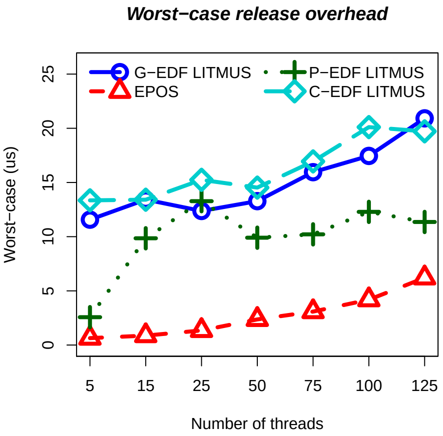
Furthermore, the use of a slower data structure in EPOS (list) against the LITMUS$^{RT}$ heap implementation is not critical, because EPOS performs less operations before and after inserting or removing elements to/from the data structure. As the number of threads increases, the standard deviation of EPOS increases as well. This is because the greater time difference between an interrupt that only releases one thread and an interrupt that releases several threads.

## 5.8 PREEMPTION/MIGRATION DELAY EVALUATION

We measured the CPMD using the EPOS hardware performance counter API, described in Section 4.1.1. We configured HPCs to count hardware events that together form three metrics, represented in Equations 1, 2, and 3. The metrics calculate the impact of L1, L2, and



(a)



(b)

**Figure 61: Average (a) and worst-case (b) thread release overhead.**

L3-cache misses in terms of cycles spent serving them (Intel Corporation, 2011; Intel Corporation, 2012). The Last-Level Cache Miss Impact (LLC MI) is the number of all memory accesses that missed the LLC multiplied by the number of cycles spent to serve one LLC miss (200):

$$\text{LLC MI} = 200 \times \text{mem load uops retired that miss LLC} \tag{1}$$

The Last-Level Cache Hit Impact (LLC HI) is the sum of all memory accesses that hit the LLC with no bus snoop needed, all memory accesses that hit the LLC and required a cross-core snoop hit, and all memory accesses that hit the LLC and had a hit modified response from another core multiplied by the processor cycles spent to serve each hardware event (31, 43, and 60 respectively):

$$\begin{aligned}\text{LLC HI} = {} & 31 \times \text{mem load uops retired that hit LLC} + \\ & 43 \times \text{mem load uops with LLC hit and snoop hit} + \\ & 60 \times \text{mem load uops with LLC hit and hitm response}\end{aligned} \tag{2}$$

The L2-cache Hit Impact (L2 HI) multiplies the number of all memory accesses that hit the L2-cache by the processor cycles spent in one hit (12 cycles):

$$\text{L2 HI} = 12 \times \text{mem load uops that hit L2} \tag{3}$$

The sum of the three equations gives us the total impact of all accesses that missed in the L1-cache. However, the platform presents a hardware limitation to perform this calculation. Intel i7-2600 processor offers only four programmable and three-fixed hardware counters per each core-thread, while the metrics need five counters. We eliminated the event that counts the LLC hit and had a cross-core snoop hit modified response from Equation 2, because there is no data sharing in our test application (see Figure 62) and this event only captures cache coherence activities.

Figure 62 shows part of the application code used to measure the CPMD. We vary the WSS from 4 KB to 10 MB (WSS variable in line 4), which provides a reasonable relation to the size of the three cache memory levels (see Table 12). The `Perf_Mon` component configures the hardware counters (line 9) and then reads them into a buffer (line 14). At every iteration, a periodic thread calls the `wbinvd` instruction to write back all modified cache lines to main memory and to invalidate the internal caches (line 8). Thus, we emulate a worst-case scenario (for

HRT applications) where a thread entirely loses its cache affinity after a preemption/migration. The application creates 16 periodic threads. Each thread iterates for 1000 times and the maximum period is 20 ms (total execution time of 20 seconds). To control a CPU migration, we used the `CPU_Affinity` scheduler and at every thread period we changed the thread's affinity to force a CPU migration.

```
int func(int factor, int id)
{
    Perf_Mon perf;
    int array[WSS];
    int sum = 0;
    for(int i = 0; i < ITERATIONS; i++) {
        Periodic_Thread::wait_next();
        asm(”wbinvd”);
        perf.cpmd();
        for(int k = 0; k < factor; k++) {
            for(int j = 0; j < WSS; j++)
              sum += array[j];
            if(k == 0)
              perf.get_cpmd(threads[id]->_buffer);
        }
    }
}
```

**Figure 62: CPMD application code.**

We executed the test application for ten times and extracted the worst-case values and the average cases for each WSS from these executions. We also considered a theoretical worst-case bound: the cache line size (64) divides the WSS, and the processor cycles to treat an LLC (200) multiplies the division resulting value. Figure 63 shows the calculated worst-case bound, the sampled worst-case, and the average CPMD for our test application. On the $x$-axis, we vary the WSS and on the $y$-axis, we present the CPMD in µs and in logarithm scale. Comparing the sampled worst-case with the calculated worst-case bound, the hardware pre-fetcher considerably improves the CPMD: it brings data to the LLC that is later accessed, which does not cause an LLC miss. For WSS of 10 MB, it is possible to observe a smaller difference between the sampled and calculated worst-case bound values, because the application thrashes the cache. Additionally, the CPMD difference between the average and sampled worst-case is not high, and the average cases have a low standard deviation (error bars that represent standard deviation are almost imperceptible). This shows that hardware performance counters can provide a correct view of the application behavior.

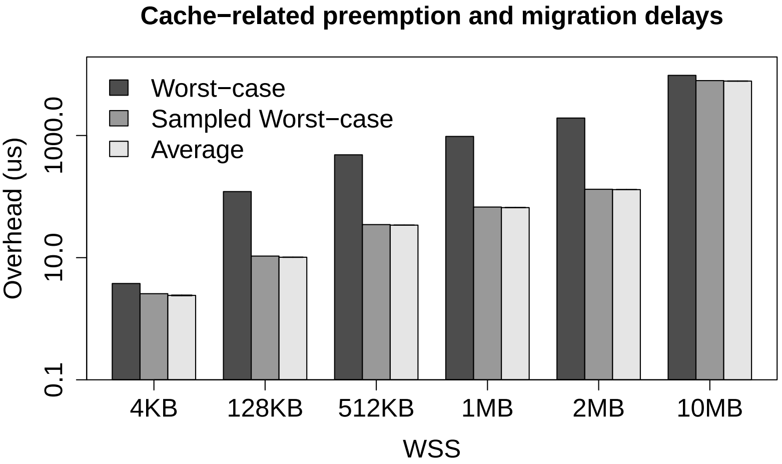


**Figure 63: Cache-related preemption and migration delay varying the WSS in microseconds. Note that the *y*-axis uses a logarithm scale.**

## 5.9 SCHEDULABILITY TESTS ANALYSIS

We present below the empirical comparison between G-EDF, C-EDF, and P-EDF. We first show the comparison considering the runtime overhead measured in eight processors and then we present the results of a comparison between P-EDF and G-EDF for 100 processors, considering only the ideal tests in both schedulers (*i.e.,* without overhead)[4].

[4] Recall that the experiments presented in this section were simulated. Specifically, in the experiments of Subsection 5.9.1, we inflated the WCET of each task with the OS overhead measured in the previous section and applied the schedulability tests accordingly, as described in Section 5.1.

### 5.9.1 Run-Time Overhead

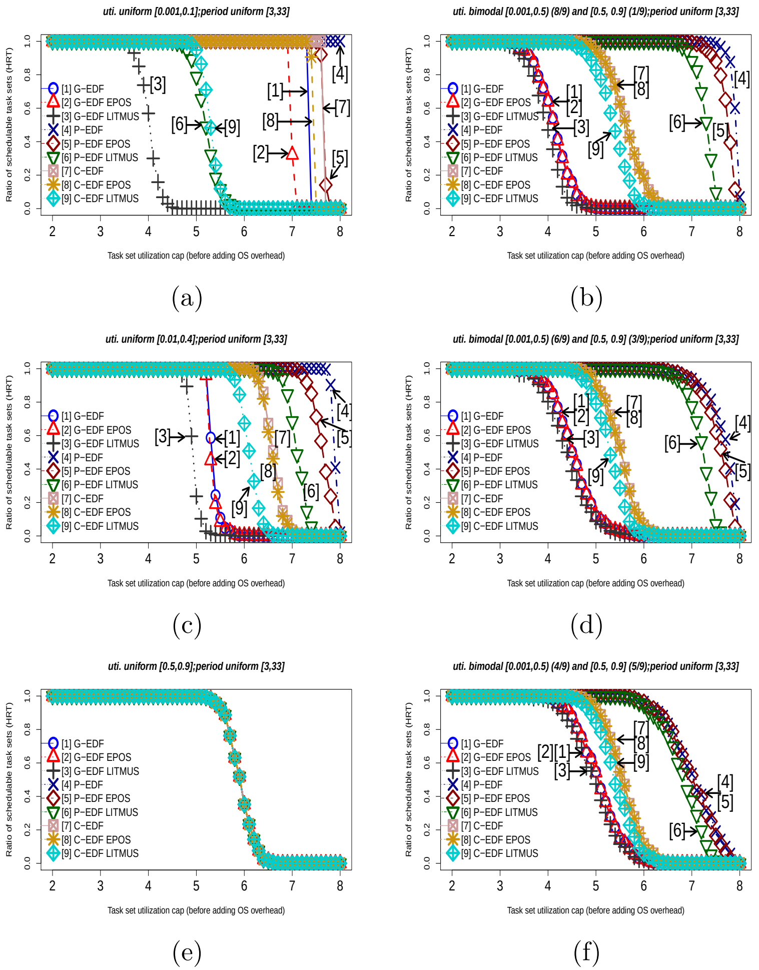
Figure 64 shows the task set schedulability ratio for short periods and the six combinations of uniform and bimodal utilization distributions. In the $x$-axis, we vary the utilization cap and in the $y$-axis, we present the ratio of schedulable task sets. A ratio of 0.6, for instance, means that 60% of the total generated task sets are schedulable.

In Figure 64(a), which shows the results for the light uniform utilization, P-EDF is able to partition and schedule all task sets. The partitioning heuristics performed well, because tasks utilizations are very low. For the bimodal light utilization, P-EDF did not have the same behavior due to few heavy tasks: for instance, in Figure 64(b), the P-EDF schedulability ratio starts to decrease around the utilization cap of 7.0.

For all analyzed distributions but the uniform heavy utilization, G-EDF is worse than P-EDF and C-EDF. When all tasks in a task set have utilizations between 0.5 and 0.9 (in the uniform heavy utilization), the partitioning heuristics can only partition a task set with a total number of tasks equal to the processor number (eight). Figure 64(f) exemplifies this situation, where around the utilization cap of 5.1, task sets have more than eight tasks. Consequently, the schedulability ratio starts to drop. For G-EDF, instead, the HRT bounds in the sufficient schedulability tests limit the schedulability ratio. In fact, the G-EDF schedulability tests are over pessimistic.

In the light uniform utilization, when the number of tasks in a task set is greater than in the medium and heavy utilization distributions, the run-time overhead impact is more significant. For example, in the uniform light utilization (Figure 64(a)), P-EDF considering the overhead in LITMUS$^{RT}$ is worse than G-EDF and G-EDF considering the overhead in EPOS. Also, P-EDF with the overhead in EPOS is similar to C-EDF without overhead. C-EDF with the overhead in EPOS is better than G-EDF without overhead. For the same light uniform utilization, we can observe an interesting fact in LITMUS$^{RT}$: C-EDF is better than P-EDF due to the smaller scheduler overhead for 125 threads (see Figure 59).

In general, P-EDF and C-EDF in EPOS had almost the same schedulability ratio as P-EDF and C-EDF without OS overhead, which demonstrates that a lightweight RTOS can provide HRT bounds close to the theoretical ones.



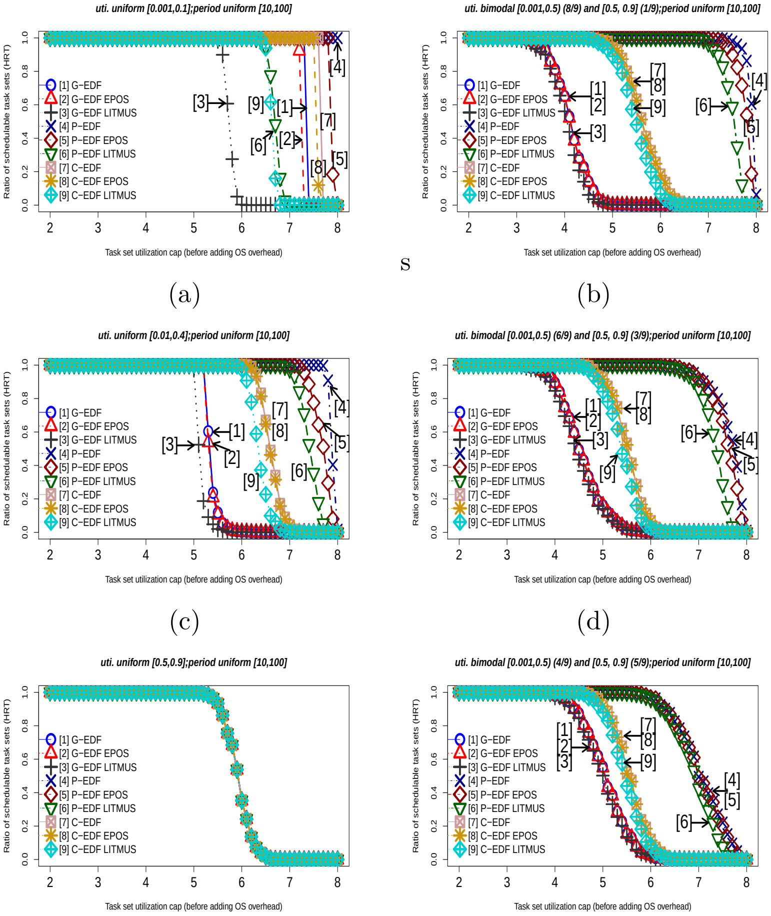
(a) (b)

(c) (d)

(e) (f)

**Figure 64: Comparison between G-EDF and P-EDF with short periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy.**

Figure 65 shows the task set schedulability ratio for moderate periods and the six combinations of uniform and bimodal utilization distributions. We observe a reduction in the overhead impact for all distributions compared to shorter periods: as the periods become larger, the proportion between periods and overheads becomes smaller. Figure 65(a) exemplifies this situation, where the lines for G-EDF with EPOS and LITMUS$^{RT}$overheads are closer to the theoretical G-EDF than the previous graph with short periods (Figure 64(a)).



s

(a) (b)

(c) (d)

(e) (f)

**Figure 65: Comparison between G-EDF and P-EDF with moderate periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy.**

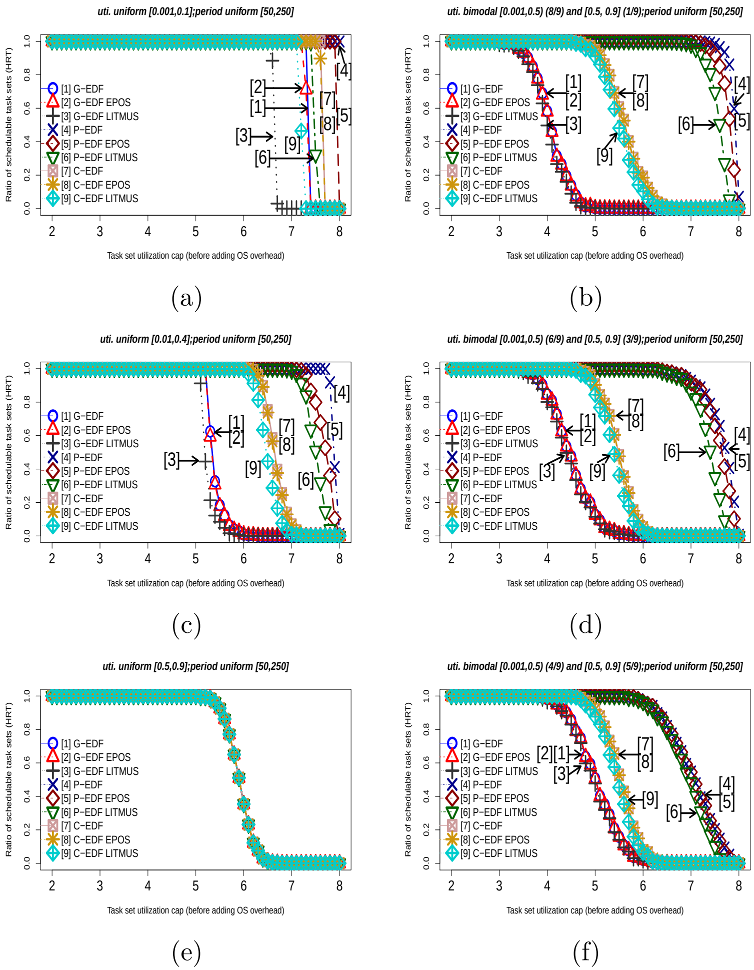
Moreover, for the same light uniform distribution, P-EDF with EPOS and LITMUS$^{RT}$ overheads also improved the schedulability ratio compared to the short period. For instance, the schedulability ratio of P-EDF with EPOS and LITMUS$^{RT}$ overheads in moderate periods start to decrease from 1 in the utilization caps of 7.9 and 6.6, respectively. For short periods, in contrast, the schedulability ratio of P-EDF inflated by the overhead in EPOS starts to drop in the utilization cap of 7.8 and the schedulability ratio of P-EDF considering the overhead in LITMUS$^{RT}$ in the utilization cap of 4.6. In the uniform light utilization (Figure 65(a)), P-EDF considering the overhead in LITMUS$^{RT}$ is still worse than G-EDF and G-EDF considering the overhead in EPOS. Also, the C-EDF with the overhead in LITMUS$^{RT}$ is now worse than the P-EDF with the overhead in LITMUS$^{RT}$. With short periods, the C-EDF in LITMUS$^{RT}$ had better performance than the P-EDF.

C-EDF with the overhead in EPOS is still close to the schedulability ratio of the C-EDF without overhead and better than G-EDF with and without overhead, although the difference between the schedulability ratio of C-EDF and G-EDF had a significant reduction.

Figure 66 shows the task set schedulability ratio for long periods and the six combinations of uniform and bimodal utilization distributions. Following the same trend, long period lengths reduce the overhead impact on the schedulability ratio: the proportion among periods and overheads becomes even smaller.

Furthermore, for the light uniform utilization (Figure 66(a)), P-EDF inflated by the overhead in LITMUS$^{RT}$ was better than the ideal G-EDF for the first time. C-EDF considering the overhead in LITMUS$^{RT}$ is still worse than the ideal G-EDF and G-EDF inflated by the overhead in EPOS. All scheduler variants improved the schedulability ratio. For example, the schedulability ratio in the light uniform utilization of the P-EDF inflated by the overhead in EPOS reaches 0% at utilization cap of 8, while for short periods it reaches 0% at utilization cap of 7.8. For P-EDF inflated by the overhead in LITMUS$^{RT}$, the schedulability ratio in the light uniform utilization reaches 0% at utilization cap of 7.6, while for short periods it reaches 0% at utilization cap of 6.2.

**Summary.** Table 13 summarizes the schedulability ratio results of P-EDF, C-EDF and G-EDF, considering the three utilization variants (light, medium, and heavy), and the uniform and bimodal distributions. G-EDF, C-EDF, and P-EDF have the same performance



(a) (b)

(c) (d)

(e) (f)

**Figure 66: Comparison between G-EDF and P-EDF with long periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy.**

for heavy uniform utilizations. Considering the different periods (short, moderate, and long), the biggest difference in terms of schedulability ratio between G-EDF, C-EDF, and P-EDF is in the bimodal light utilization. The presence of few heavy tasks profoundly affects the bound in the G-EDF schedulability tests.

Moreover, in the light uniform utilization and short periods scenario, the impact of the run-time overhead on the schedulability ratio is more significant, because the proportion between period and overhead is smaller. Furthermore, for the light uniform and short and moderate periods, G-EDF inflated by the overhead in EPOS is better than P-EDF inflated by the overhead in LITMUS$^{\mathrm{RT}}$, differently of the theoretical tests, in which P-EDF is always better than G-EDF (except for heavy uniform utilizations). Yet, in the light uniform utilization and short periods, C-EDF inflated by the overhead in LITMUS$^{\mathrm{RT}}$ is better than P-EDF also inflated by overhead in LITMUS$^{\mathrm{RT}}$, differently from the theoretical C-EDF and P-EDF as well, in which P-EDF is always better than C-EDF. In the bimodal heavy utilization, the run-time overhead only changes the schedulability ratio for P-EDF and short periods. For C-EDF, the schedulability ratio is almost the same for short, moderate, and long periods, and for G-EDF the schedulability ratio is the same.

### 5.9.2 Weighted Schedulability

We used the weighted schedulability to account for CPMD (BASTONI *et al.*, 2010a). As the schedulability ratio depends on two variables (*i.e.,* utilization cap and CPMD), the weighted schedulability reduces the results to a two-dimensional ploting without the use of the utilization cap.

Let $D_c$ be a maximum CPMD incurred by any job and $U$ be a utilization cap. $S(U, D_c)$ denotes the schedulability ratio for a given $U$ and $D_c$, which is in the interval [0, 1]. Let $Q$ be a set of utilization caps ($Q = \{2.0, 2.1, \ldots, m\}$). Then, Equation 4 defines the weighted schedulability for a $D_c$, $W(D_c)$ (BASTONI *et al.*, 2010a).

$$W(D_c) = \frac{\sum_{U \in Q} U \times S(U, D_c)}{\sum_{U \in Q} U} \tag{4}$$

We used the calculated worst-case bound CPMD values shown in Figure 63 as input to $D_c$. We assume that each task suffers the CPMD once — a single job is potentially preempted multiple times, but each job in the system can only cause one CPMD on one other job. We inflated the WCET $e_i$ of each task $T_i$ according to the preemption-centric interrupt accounting method, described in Section 2.5. Then,

Table 13: **Summary of the schedulability ratio comparison between P-EDF, C-EDF and G-EDF, considering also the run-time overhead in EPOS and LITMUS$^{RT}$.**

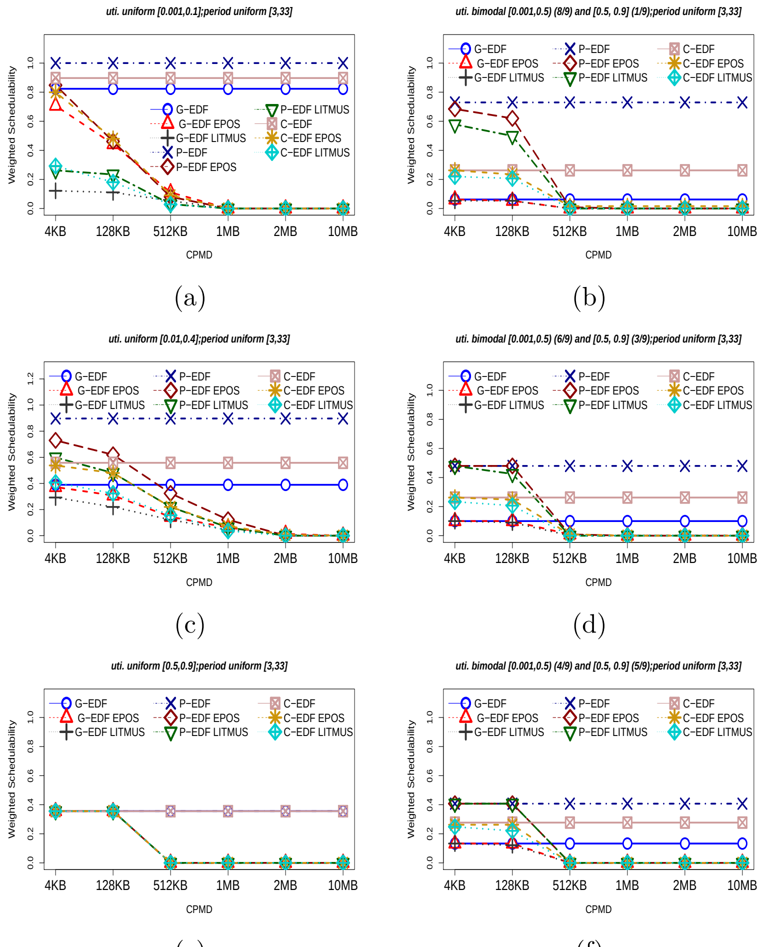
| | | P-EDF | C-EDF | G-EDF |
|---|---|---|---|---|
| Light Utilization | Uniform | P-EDF and P-EDF in EPOS have the best performance among all analyzed scenarios. | C-EDF in EPOS is better than G-EDF without overhead. C-EDF is better than P-EDF in LITMUS$^{RT}$ (short periods). | G-EDF with EPOS overhead is better than P-EDF with LITMUS$^{RT}$ overhead (moderate and short periods). |
| | Bimodal | All the three P-EDF variants are better than G-EDF. | C-EDF in EPOS is similar to C-EDF without overhead and better than in LITMUS$^{RT}$. | P-EDF is better than G-EDF. The biggest difference in the schedulability ratio between P-EDF and G-EDF. |
| Medium Utilization | Uniform | The run-time overhead is smaller in light utilization due to the lower number of tasks in the task sets. | C-EDF in LITMUS$^{RT}$ is worse than P-EDF in LITMUS$^{RT}$. C-EDF in EPOS is still better than G-EDF without overhead. | All G-EDF scenarios are worse than P-EDF. Moreover, G-EDF with the OS run-time overhead is equal or close to the ideal G-EDF. |
| | Bimodal | P-EDF in EPOS and in LITMUS$^{RT}$ is close to the ideal P-EDF. | The same behavior as in light utilization. | |
| Heavy Utilization | Uniform | G-EDF, C-EDF, and P-EDF have the same performance. | | |
| | Bimodal | The run-time overhead changes the schedulability ratio only for short periods. P-EDF is better than C-EDF and G-EDF. | The schedulability ratio is almost the same for short, moderate, and long periods. | The run-time overhead does not change the schedulability ratio. |

we computed the $S(U, D_c)$ varying the $D_c$ and $U$. The result is the schedulability ratio considering the CPMD.

We used the same previously generated task sets, varying the utilizations and periods. A task set is considerable schedulable in the G-EDF if it passes in at least one of eight sufficient schedulability tests (GOOSSENS *et al.*, 2003; BAKER, 2005a; BAKER, 2003; BARUAH, 2007; BAKER; BARUAH, 2009; BERTOGNA *et al.*, 2005; BERTOGNA; CIRINEI, 2007; BARUAH *et al.*, 2009). For the P-EDF algorithm, we first partitioned the task set using three partitioning algorithms (FFD, BFD, and WFD) and then applied the EDF test (LIU; LAYLAND, 1973) for each partition (eight in total). A task set is schedulable if all the eight partitions pass in the test and at least one partitioning algorithm correctly partitions the task set. For C-EDF, we first partitioned the task set using the same partitioning algorithms as in P-EDF, and then applied the same eight sufficient schedulability tests of G-EDF in each cluster. We defined four cluster with two processors in each cluster (sharing the L2 cache). A task set is schedulable if the four partitions pass in at least one of eight G-EDF sufficient tests and at least one partitioning algorithm correctly partitions the task set into the four clusters.

Figure 67 shows the obtained results for task sets with short periods. The $x$-axis defines the CPMD, while the $y$-axis presents the weighted schedulability metric. For instance, the weighted schedulability for a CPMD of 128 KB and P-EDF inflated by the overhead in EPOS is 0.82, which means that 82% of all generated task sets are schedulable. G-EDF, C-EDF and P-EDF lines serve as a reference for analyzing the results, since they have no overhead. For the uniform heavy utilizations distribution (Figure 67(e)), we observe the same trend as before: the three schedulers have the same performance for all CPMD values.

For small WSSs (4 KB and 128 KB), P-EDF is superior to C-EDF and G-EDF. As the WSS increases, the difference between P-EDF, C-EDF, and G-EDF decreases, which can be clearly seen in Figure 67(c). For WSSs greater than 512 KB, P-EDF, C-EDF, and G-EDF tend to be equal mainly due to the high CPMD values. Figures 67(a), (b), and (c) show the cases where the difference between the weighted schedulability of EPOS and LITMUS$^{RT}$ is more significant: there are more tasks in a task set, which favors EPOS due to the smaller run-time overhead. For light uniform utilization and short periods (Figure 67(a)), G-EDF, C-EDF, and P-EDF inflated by the overhead



(a) (b)

(c) (d)

(e) (f)

**Figure 67: Weighted schedulability for short periods.**

in EPOS have similar performance and they provide a considerable higher schedulability ratio than when they are inflated by the overhead in LITMUS$^{RT}$. G-EDF inflated by the overhead in EPOS is better than the three schedulers inflated by the overhead in LITMUS$^{RT}$ (4, 128, and 512 KB).

For moderate and long periods the graphs are similar to the previously ones: as the periods become larger, the proportion between CPMD and the period becomes smaller. As a consequence, the weighted schedulability ratio starts to drop only for higher CPMD values. For example, Figure 68 shows the weighted schedulability results for light uniform utilization and long periods. Compared to the light uniform utilization and short periods, the weighted schedulability reaches 0 only for WSS of 10 MB, instead of 1 MB as for short periods. Also, G-EDF inflated by the overhead in EPOS is worse than P-EDF inflated by the overhead in LITMUS$^{RT}$ for WSSs of 4 and 128 KB. For greater WSSs, G-EDF inflated by the overhead in EPOS provides a better weighted schedulability ratio.

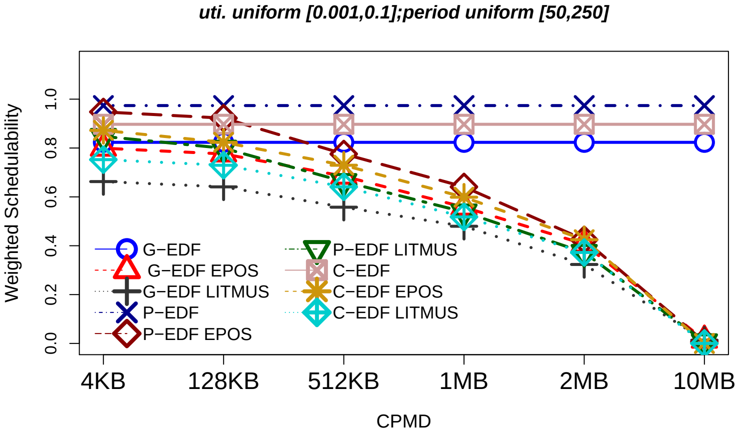


**Figure 68: Weighted schedulability for light uniform distribution and long periods.**

### 5.9.3 Schedulability Evaluation for 100 Processors

We present below the empirical comparison between the ideal (*i.e.*, without considering the run-time overhead) G-EDF and P-EDF for 100 processor. We did not compare C-EDF, because it is formed by a combination between G-EDF and P-EDF and, consequently, its behavior for HRT systems is always between both schedulers, as demonstrated by our previous results. We used the same period and utilization distributions as described early to generate the task sets. However, we used different utilization slacks: 0.5, 0.5, and 1 for light, medium, and heavy utilization distributions, respectively. Then, we executed the eight G-EDF sufficient schedulability tests and the three P-EDF partitioning techniques in the SHARCNET cluster (SHARCNET, 2012), varying the utilization cap from 2, 3, 4,…, to 100. The total computation time for all tests and task sets generation was more than 1.1 year.

Figure 69 shows the obtained task set schedulability ratio for uniform utilization distributions. Figures 69(a), (b), and (c) show the schedulability ratio for light uniform distribution. Figures 69(d), (e), and (f) for medium uniform distribution, and Figures 69(g), (h), and (i) for heavy uniform distribution. The $x$-axis defines the utilization cap and the $y$-axis the ratio of schedulable task sets. A ratio of 0 means that none of the task sets is schedulable, while a ratio of 1 means that all task sets are schedulable. P-EDF is always better than G-EDF, except for the heavy distribution. For the heavy distribution (Figures 69(g), (h), and (i)) G-EDF had the same results as P-EDF, for the same reason as explained early: the number of tasks in a task set affects the partitioning heuristics and the HRT bounds in the G-EDF sufficient tests limit the schedulability.

Figure 70 shows the obtained task set schedulability ratio for the bimodal distributions. The $x$-axis defines the utilization cap and the $y$-axis the ratio of schedulable task sets. P-EDF is again always better than G-EDF for HRT tasks. Moreover, for light bimodal utilization (Figures 70(a), (b), and (c)), the schedulability ratio of G-EDF reaches 0 when the utilization cap is 45. This clearly states the need for less pessimistic G-EDF sufficient schedulability tests, since 55% of the available processors are “wasted” due to HRT guarantees. For medium and heavy bimodal distribution, we can observe a decrease in the schedulability ratio for P-EDF due to that heavy tasks affect the partitioning algorithms. For G-EDF, we can note a slight increase in the schedulability ratio, because the sufficient schedulability tests provide better HRT bounds when there are more heavy tasks. In fact, few heavy tasks have a greater impact on the schedulability ratio of G-EDF.

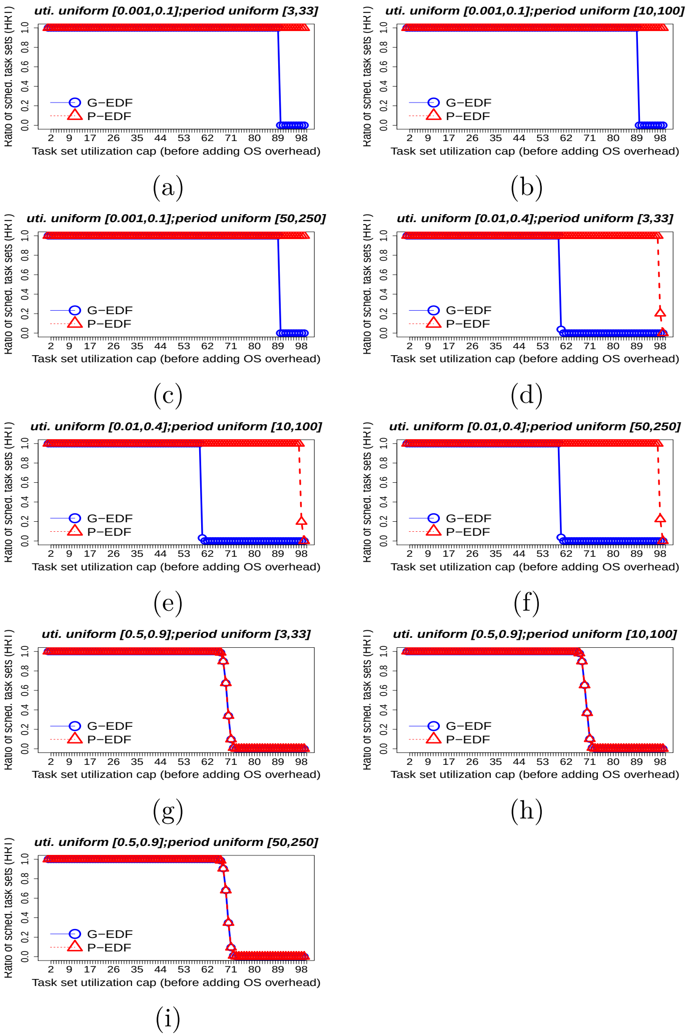
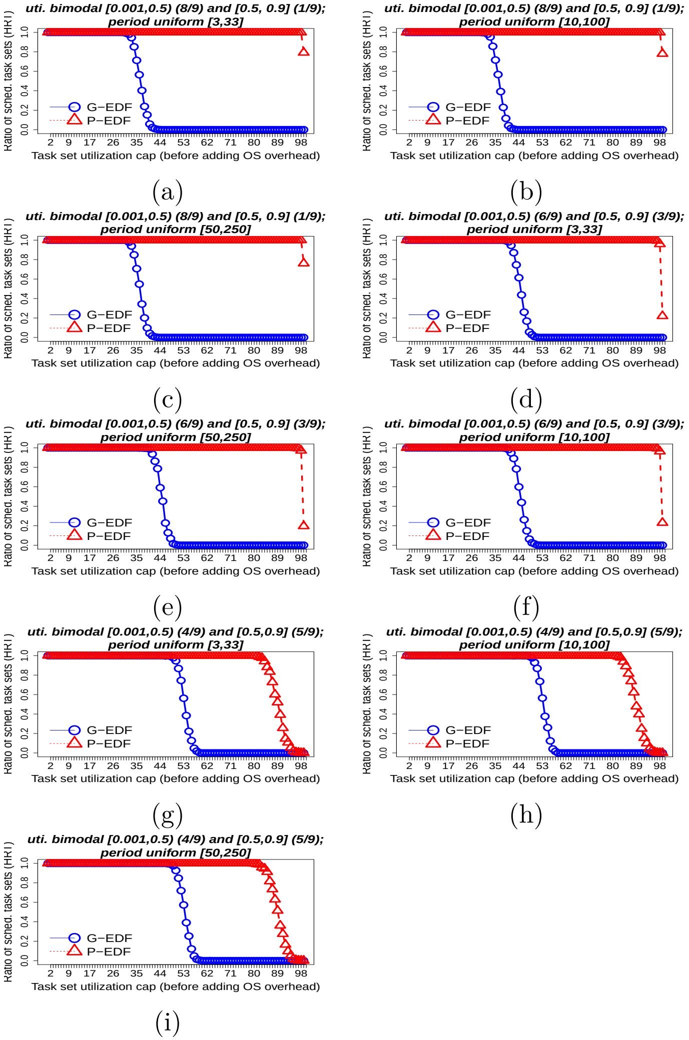


**Figure 69: Comparison between G-EDF and P-EDF using uniform utilizations: (a), (b), and (c) light uniform. (d), (e), and (f) medium uniform, and (g), (h), and (i) heavy uniform.**



uti. bimodal [0.001,0.5) (8/9) and [0.5, 0.9] (1/9); period uniform [3,33]

(a)

uti. bimodal [0.001,0.5) (8/9) and [0.5, 0.9] (1/9); period uniform [10,100]

(b)

uti. bimodal [0.001,0.5) (8/9) and [0.5, 0.9] (1/9); period uniform [50,250]

(c)

uti. bimodal [0.001,0.5) (6/9) and [0.5, 0.9] (3/9); period uniform [3,33]

(d)

uti. bimodal [0.001,0.5) (6/9) and [0.5, 0.9] (3/9); period uniform [50,250]

(e)

uti. bimodal [0.001,0.5) (6/9) and [0.5, 0.9] (3/9); period uniform [10,100]

(f)

uti. bimodal [0.001,0.5) (4/9) and [0.5,0.9] (5/9); period uniform [3,33]

(g)

uti. bimodal [0.001,0.5) (4/9) and [0.5,0.9] (5/9); period uniform [10,100]

(h)

uti. bimodal [0.001,0.5) (4/9) and [0.5,0.9] (5/9); period uniform [50,250]

(i)

**Figure 70: Comparison between G-EDF and P-EDF using bimodal utilizations: (a), (b), and (c) light bimodal, (d), (e), and (f) medium bimodal, and (g), (h), and (i) heavy bimodal.**

## 5.10 DISCUSSION

During our evaluation, we observed a set of interesting facts regarding aspects of the RTOS, processor architecture, and real-time scheduling:

- **OS design versus data structures:** we noticed that the OS design is as important as the internal data structures of the scheduling and task release functions. In comparison to LITMUS$^{RT}$, EPOS performs less operations before and after a scheduling decision and a task release, resulting in less overhead. Furthermore, we believe that changing the current EPOS lists to a data structure with better performance, such as heaps, can reduce the run-time overhead and consequently, reduce the impact on the schedulability ratio, as demonstrated by Brandenburg and Anderson (BRANDENBURG; ANDERSON, 2009).

- **RTOS versus general-purpose OS:** our results show that an RTOS designed from scratch considerably reduces the run-time overhead in comparison to general-purpose OSes with real-time patches. In scenarios composed of several light utilization tasks, we could note an improvement of about 46% in the task set schedulability ratio of G-EDF considering the overhead in EPOS in contrast to the G-EDF inflated by the overhead in LITMUS$^{RT}$ (Figure 64(a)). Considering P-EDF, each ready scheduling list has less tasks than the global scheduling list of the G-EDF, which reduces the scheduling run-time overhead. Nevertheless, for the same light uniform utilization scenario, P-EDF considering the overhead in EPOS was about 26% better than P-EDF considering the overhead in LITMUS$^{RT}$. For example, while P-EDF inflated by the overhead in LITMUS$^{RT}$ reaches 0% of schedulability ratio at the utilization cap of 6.2, P-EDF considering the overhead in EPOS reaches 0% at the utilization cap of 8. C-EDF inflated by the overhead in EPOS is also about 26% better than C-EDF inflated by the overhead in LITMUS$^{RT}$. For task sets composed of only heavy tasks, the influence of the run-time overhead on the schedulability ratio is less significant, because there are few tasks in the system. Thus, the scheduling and alarm lists (see Section 4.2) manage less elements, reducing the overhead.

- **Run-time overhead versus schedulability tests:** in our measurements, EPOS has presented a lower run-time overhead than LITMUS$^{RT}$, as demonstrated in Section 5.9.1. However, despite the higher run-time overhead, LITMUS$^{RT}$ achieved a schedulability ratio comparable with EPOS in all but non-uniform workloads. The main reason for this behavior is the pessimism of the available schedulability tests. Non-uniform task sets have few tasks with higher utilizations (between 0.5 and 0.9), which strongly affects the HRT guarantees provided by the G-EDF schedulability tests and the bin packing partitioning heuristics. For light uniform utilization and short periods, C-EDF was better than P-EDF with LITMUS$^{RT}$ overhead due to smaller scheduling overhead (see Figure 59). Also, our results corroborate previous studies (BASTONI *et al.*, 2010b) in the sense that more partitioned approaches are preferable for HRT systems.

- **P-EDF is always equal or better than G-EDF and C-EDF for HRT:** for all distributions, except the heavy utilization, P-EDF was superior to C-EDF and G-EDF. For task sets consisting of only heavy utilization tasks, P-EDF, C-EDF, and G-EDF had the same schedulability ratio. This is due to the bin packing problem limitation, in which the partitioning heuristics can only partition task sets with a task number equal to the number of processors, and due to the G-EDF schedulability bounds, which usually have a relation between the number of processor and the largest utilization or density (BERTOGNA; BARUAH, 2011). For task sets with few heavy tasks, as in the case of light bimodal utilization distribution, G-EDF presented the biggest difference in terms of task set schedulability ratio in comparison to P-EDF. This reinforces the need for better G-EDF schedulability tests for HRT systems with heavy utilization tasks (BRANDENBURG; ANDERSON, 2009).

- **Differences in task period length:** varying period lengths (short, moderate, and long) in our empirical evaluations did not affect the schedulability tests for the theoretical (*i.e.,* without overhead) G-EDF, C-EDF, and P-EDF schedulers. On the other hand, it has a significant impact on the run-time overhead. In short period distributions, the proportion between the period length and the overhead is higher than in long period distributions (see Figure 64(a), Figure 65(a), and Figure 66(a)). For example, for short periods and light uniform utilization, the schedulability ratio for G-EDF inflated by the overhead in LITMUS$^{RT}$ starts to drop at the utilization cap of 3.8, while for long periods and the same utilization distribution, the schedulability ratio starts to drop at the utilization cap of 6.9.

- **Outliers removal:** one difference between our approach and the related work (BRANDENBURG *et al.*, 2008) is about removing the outliers in the measured values for LITMUS$^{RT}$. In their approach, the authors discarded 1% of the greatest obtained values. By analyzing the collected tracing values, we noticed that discarding 1% of the values would not be fair for HRT applications. For example, in the context switch overhead for 125 tasks, the total number of collected events is 1188679. Discarding 1% of data (11886 values), would significantly decrease the observed worst-case value. By plotting a "boxplot", analyzing the outliers, and the standard deviation, we noticed that discarding a few outliers is sufficient. For instance, for the context switch overhead and 125 tasks, if we did not discard any outliers, the obtained standard deviation is 2.12 µs. Discarding the outliers (7 in this case), we obtained a standard deviation of 0.47 µs, which is similar to the standard deviation of the other cases (5, 15, 25, 50, 75, and 100 tasks). The same applies for the scheduling overhead and IPI latency.

- **Hardware performance counters:** hardware performance counters are useful for scheduling and memory management in RTOSs. Although each processor architecture supports different hardware events, different names for the same events, and presents different hardware limitations (*e.g.,* number of registers and features), well-designed OS APIs can abstract these differences for the rest of the system. Usually, hardware event names change but their meaning remain.

  Moreover, we believe that PMUs will support even more events and features in the near future. Examples of features that could be added by hardware designers into future PMUs are discussed in Section 4.1.6: (i) data address registers to store addresses that generated an event; (ii) monitoring address space intervals to provide more precise view of specific application address ranges; (iii) processing cycles spent in specific events, such as bus activities and memory coherence protocols; and (iv) OS trap generation according to pre-defined event numbers.

These features can improve scheduling decisions at run-time, providing a correct and precise view of the running applications. Also, such features improve shared memory partitioning algorithms, which are useful to avoid the overlap of shared cache spaces by tasks executing on different processors. Thus, cache partitioning reduces the contention for the shared cache and increases the system predictability (LIN *et al.*, 2008; SUHENDRA; MITRA, 2008; SRIKANTAIAH *et al.*, 2008; MURALIDHARA *et al.*, 2010).

- **Cache-related preemption and migration delay:** we measured the CPMD using HPCs and used the weighted schedulability to account for CPMD in the task set schedulability ratio. In our evaluations, P-EDF is better than G-EDF and C-EDF for WSSs of 4 KB and 128 KB. As the WSS increases, P-EDF, C-EDF, and G-EDF tend to be equal due to higher CPMD. For uniform and bimodal light and uniform moderate utilizations distributions, which have more tasks than the other generated task sets, the difference between EPOS and LITMUS$^{\mathrm{RT}}$ is higher. Moreover, P-EDF, C-EDF, and G-EDF had the same performance for task sets composed of only heavy tasks.

  A possible way to decrease the CPMD is the use of cache locking mechanisms (VERA *et al.*, 2003b; SUHENDRA; MITRA, 2008; APARICIO *et al.*, 2011; MANCUSO *et al.*, 2013). Cache locking prevents cache lines or ways to be evicted by the cache replacement policy during the program execution. The combination of cache partitioning and cache locking improves the system predictability (SUHENDRA; MITRA, 2008; MANCUSO *et al.*, 2013). However, most of the current processors do not support cache locking. Hardware designers should consider this feature for future processors.

- **Linux development support and HRT applications:** Linux has a complete infrastructure of libraries, benchmarks, and graphical interfaces to develop and test applications of any kind. Our focus, however, is on deeply real-time embedded systems, which usually do not even feature a graphical user interface. Consequently, an RTOS tailored for such applications can certainly improve their performance. Our main objective is to provide an environment for HRT research without the known interference of general-purpose OSes.

- **UMA x ccNUMA architectures:** we used a cache-coherent Non-Uniform Memory Access (ccNUMA)[5] architecture in our experiments. Previous studies on CPMD analysis (BASTONI *et al.*, 2010a) used a Uniform Memory Access (UMA) architecture. There are important differences between these two memory organizations. Usually, UMA processors implement the MESI cache-coherence protocol (HENNESSY; PATTERSON, 2006), while ccNUMA processors use MESIF (Intel Corporation, 2009) or MOESI (AMD, 2010) protocols. Also, the processor interconnect presents considerable differences. For example, the newer Intel processors, such as the Intel i7-2600, use the Intel's Quickpath Interconnect (QPI), while older Intel processors, such as the Intel Xeon 5030, use the Front-Side Bus (FSB). Compared to the FSB, QPI provides higher bandwidth and lower latency for NUMA-based architectures. Each processor has an integrated memory controller and features a point-to-point link (all processors are connected), allowing parallel data transfer and shortest snoop request completion (Intel Corporation, 2009). In consequence, ccNUMA-based processors have a faster communication among the cores, which decreases the CPMD.

[5] Remember here that the non-uniform memory refers to cache instead of the main memory as in the computer architecture point of view.

# 6 CACHE PARTITIONING EVALUATION

In Chapter 4, we proposed a page coloring mechanism for a component-based RTOS (EPOS) that is able to assign individual cache partitions to internal OS data structures and real-time tasks. In this chapter[1], we evaluate the performance of the cache partitioning mechanism using the G-EDF, C-EDF, and P-EDF schedulers on top of EPOS.

In summary, the main contributions of this chapter are:

- We evaluate the performance of the proposed cache partitioning mechanism using P-EDF, C-EDF, and G-EDF schedulers when they have total utilization close to the theoretical HRT bounds. Our evaluation is carried out on a modern 8-core processor, with shared L3 cache. Our results indicate that cache partitioning has different behavior depending on the scheduler and task's WSS. We also show an experimental upper-bound in terms of HRT guarantees provided by cache partitioning in each scheduler.

- By allocating a different cache partition to the internal EPOS data structures, we evaluate the cache interference caused by the RTOS. We show that a lightweight RTOS, such as EPOS, does not impact HRT tasks with separated partitions.

The rest of this chapter is organized as follow. Section 6.1 describes the experiment methodology. Sections 6.2, 6.3, and 6.4 show three cache partitioning evaluations. All experimental evaluations in this chapter use the Intel i7-2600 processor (see Table 12). Finally, Section 6.5 discusses the main results.

## 6.1 EXPERIMENT DESCRIPTION

We randomly generated task sets similar to (KENNA *et al.*, 2013). We selected the periods (all values are in ms) uniformly from {25, 50,

[1] Contents of this chapter appear in a preliminary version in the following published paper:
G. Gracioli and A. A. Fröhlich, An Experimental Evaluation of the Cache Partitioning Impact on Multicore Real-Time Schedulers, In Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013.

100, 200} and utilizations uniformly between [0.1, 0.7]. The WCET of a task is defined according to the generated period and utilization. For G-EDF, we generated tasks in a task set until the GFB schedulability test fails (GOOSSENS *et al.*, 2003). For P-EDF, we generated tasks in a task set until the WFD partitioning heuristic fails. For C-EDF, we first partitioned the tasks in each cluster using the WFD heuristic and then applied the GFB test in each cluster. P-EDF usually has a better system utilization for HRT than C-EDF and G-EDF, and thus will have a few more tasks in the task sets (proved in the previous chapter). The objective is to evaluate the performance of G-EDF, C-EDF, and P-EDF when the schedulers are close to theoretical HRT bounds and when tasks use a cache partitioning mechanism. We generated ten task sets for each scheduler.

Each task executes a function that allocates its WSS, reading and writing from/to the WSS (an array), in a loop, randomly. Figure 71 shows part of the function source code. We considered scenarios with different WSS similar to (CALANDRINO *et al.*, 2006): 32 KB, 64 KB, 128 KB, and 256 KB. We defined a write ratio of 20% (one write in a cache line after four readings – line 29). The number of reads and writes of a task varies depending on the WCET. We ran a task in an unloaded system and associated the obtained WCET to the WCET of the other tasks (the repetitions parameter in Figure 71, line 7). We executed some experiments following this methodology, and ensured that the WCET of each task was never greater than the generated WCET by passing a smaller value as the repetition parameter. With this approach, time is made available for OS overhead (scheduling, context switching, release, IPI latency, and tick counting) (GRACIOLI *et al.*, 2013) and preemptions. Thus, the total utilization of the task sets remains close to the generated utilization. We did not use a tool to automatically extract the WCET, because there is no tool for our processor due to its complexity. Tasks do not interfere on the L3-cache level using cache partitioning, but there might be loads from the main memory in case of L3 cache misses, which can inflate the WCET. However, in our experiments, the sum of the WSSs of each task in all task sets fits on the L3-cache, which reduces the impact of L3 cache misses on the WCET.

A task with the greatest period in a task set executes a function that reads from and writes to an array of 512 KB in a loop, emulating a periodic server that executes best-effort tasks. We choose a WSS

```
#define ITERATIONS 200
#define MEMORY_ACCESS 16384
#define WRITE_RATIO 4

static bool same_color = false;

int job(unsigned int repetitions, int id)
{
  int sum = 0;
  Pseudo_Random * rand;
  int *array;

  if(same_color) {
      rand = new (COLOR_2) Pseudo_Random();
      array = new (COLOR_2) int[ARRAY_SIZE];
  } else {
      rand = new (id+2) Pseudo_Random();
      array = new (id+2) int[ARRAY_SIZE];
  }

  rand->seed(clock.now() + id);

  for(int i = 0; i < ITERATIONS; i++) {
      Periodic_Thread::wait_next();
      for(int j = 0; j < repetitions; j++) {
          for(int k = 0; k < MEMORY_ACCESS; k++) {
              int pos = rand->random() % (ARRAY_SIZE - 1);
              sum += array[pos];
              if((i % WRITE_RATIO) == 0) array[pos] = k + j;
          }
      }
   }
}
```

**Figure 71: Part of the task function source code.**

of 512 KB, because when allocating the memory sequentially in this size, the task reads and writes using all cache lines (128 pages). We randomly choose the array position of the first read at the beginning of each period, and increment each reading and writing by 64 (cache line size) per loop iteration. All tasks repeat for 200 periods, which results in a total execution time of about 40 s.

## 6.2 INDIVIDUAL TASK EXECUTION TIME

To evaluate the different page coloring allocation mechanisms and the influence of the OS on the WCET of tasks, we considered three different memory allocation scenarios:

- **S1:** EPOS and each task allocate data from a different super color. Thus, we reduce the inter-core interference caused by tasks and RTOS running on different cores.
- **S2:** Each task allocates data from a different super color. EPOS allocates data from a non-colored, sequential heap. This creates interference between data allocated by EPOS and the data of each task, because EPOS can access a cache line of any color.
- **S3:** Each task allocates data from the same super color. EPOS allocates data from a different super color.

All tasks in the third scenario designedly allocate memory from the same super color, possibly sharing the same cache lines, reflecting a worst-case scenario. As explained in (KENNA *et al.*, 2013), safety-critical applications have security requirements in which the prevention of malicious tasks that evict cache lines from other tasks is desirable. The second scenario evaluates the interference of EPOS with the periodic tasks. Figure 71 shows part of the task's source code for the scenarios S1 and S3 (lines 13 to 19). The `same_color` boolean variable defines whether data is allocated from the colored heap or not. For the second scenario, we changed the MMU initialization (see Section 4.3) to construct a grouping list without any color (*i.e.,* sequential addresses) and then, the system heap allocates memory from this grouping list.

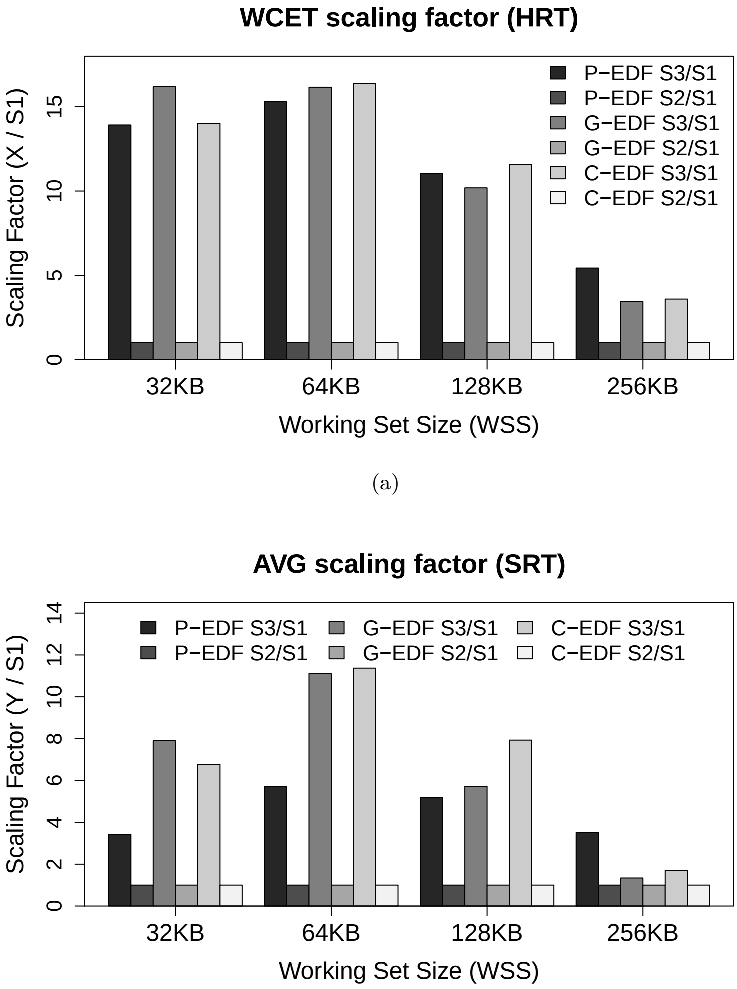
We executed the ten different task sets of G-EDF, P-EDF, and C-EDF in each scenario for 50 times, varying the WSS as described in the previous Section (more than 200 hours of tests using a real hardware and an RTOS). Then, we extracted the WCET and AVG execution time of each task for each scenario and WSS from these executions. We calculate the WCET scaling factor, $x_i^j$ / ($WCET_i^j$ in $S1$), where $i$ is a task in the task set $j$, and $x$ is the obtained WCET of $T_i$. Thus, $x_i^j$ represents the WCET of $T_i$ executing in either S2 or S3, divided by the obtained WCET for the same task $T_i$ executing in S1. We do the same for the AVG scaling factor: $y_i^j$ / ($AVG_i^j$ of $S1$), where $y_i^j$ represents the obtained AVG execution time of $T_i$ executing in either S2 or S3 and $AVG_i^j$ represents the AVG execution time of $T_i$ in S1. The AVG scaling factor is interesting for providing SRT guarantees.

We then calculate the average WCET and AVG scaling factors of each task set, varying the WSS. Figure 72(a) presents the obtained WCET scaling factors and Figure 72(b) shows the results for the AVG scaling factors. On the x-axis, we vary the WSS and, on the y-axis, we show the obtained scaling factors. Note that the scaling factors for S2 are equal to one, meaning that when EPOS uses an uncolored heap, it does not affect the tasks' execution time. EPOS interrupt service routines (ISRs) have a small fingerprint and use few bytes, generating very small interference on cache lines.

Additionally, for 32 and 64 KB, the WCET scaling factor for the G-EDF is greater than for P-EDF. As two logical cores share the same L2-cache with 128 KB, the contention for cache lines is more frequent and there are more inter-core communication (bus snooping caused by the cache coherence protocol). When setting a WSS to 32 KB, all data is possibly loaded to the L2-cache. When the WSS is 64 KB, the scaling factor increases due to higher number of misses on the L1-cache in S3 and the similar obtained WCET in S1 due to tasks isolation. We note a similar behavior for C-EDF. C-EDF had a better performance for 64 KB, possible because tasks executing in the same cluster can migrate and still access the same L2-cache, decreasing the number of misses when compared to G-EDF. When the WSS is 128 and 256 KB, the delay caused by the intra-task cache misses increases the AVG and WCET of tasks, thus decreasing the scaling factors. For 128 KB, all the three schedulers are alike: C-EDF had a slightly higher WCET scaling factor, taking advantage of migrations within the same cluster. For 256 KB, P-EDF has a greater WCET scaling factor than G-EDF and C-EDF due to the contention caused by two tasks running on the same logical core in S3, sharing the L1 and L2-caches. In the next section, we correlate the obtained scaling factors with deadline misses.

## 6.3 DEADLINE MISSES

Figure 73 presents the percentage of tasks that missed their deadlines in all task sets and varying the WSS. The x-axis presents the WSS and the y-axis the percentage of tasks that missed their deadli-



(a)

(b)

**Figure 72: (a) Obtained worst-case scaling factors (hard real-time). (b) Obtained average scaling factors (soft real-time).**

nes. We do not show S2, because it has the same behavior of S1. Our focus here is on HRT systems, consequently we show the percentage of all tasks that missed at least one deadline and not the total number of missed deadlines per task. Note that the utilization is close to the limit.

For WSS of 32 and 64 KB, the three schedulers in S1 do not miss any deadline. About 18% of tasks in G-EDF executing in S1 with 128 KB missed deadlines. This is mainly due to the cache contention caused by the MESIF cache coherence protocol. When a task is preempted and migrates to another core, it reloads the data that was on the original core. Thus, the cache controller causes invalidation on the cache lines used by the preempted task. The cost of an invalidation is comparable to access the main memory (Intel Corporation, 2012).

For P-EDF in S3, the percentage of tasks that missed deadline was 57.75%, 70.42%, 74.65%, and 93.02%, for 32, 64, 128, and 256 KB, respectively. For C-EDF in S3, the percentage was 80.56%, 83.33%, 88.89%, and 91.67%, and for G-EDF in S3, the percentage was 93%, 93.02%, 93.92%, and 97.67%. For 256 KB in S1, 54.93%, 72.22%, and 92% of tasks missed their deadline in P-EDF, C-EDF, and G-EDF, respectively. We can conclude that cache partitioning provides safe HRT bounds for WSSs of 32 and 64 KB for the three schedulers, and for WSS of 128 KB for P-EDF and C-EDF. The next section correlates the missed deadlines with the application execution time.

## 6.4 TOTAL EXECUTION TIME

Figure 74 shows the total application execution time obtained for S1 and S3 (*i.e.,* the total time to finish all tasks in a task set). Again, we do not show S2, because it has the same performance of S1. The x-axis shows the WSS and the y-axis the total execution time in seconds. As described early, each task iterates for 200 times, and the greatest possible period for a task is 200 ms, which gives us an expected execution time of 40 s (all task sets have at least one task with a period of 200 ms).

Figure 74 can be correlated to Figure 73 in terms of the frequency of deadline misses. For example, for WSS of 32 KB and S3 and G-EDF scheduler, 93% of tasks lost at least one deadline; however, we can see that total execution time is still 40 s. This shows that the occurrence of deadline misses is infrequent. As the WSS increases, the total application time also increases; tasks miss their deadlines

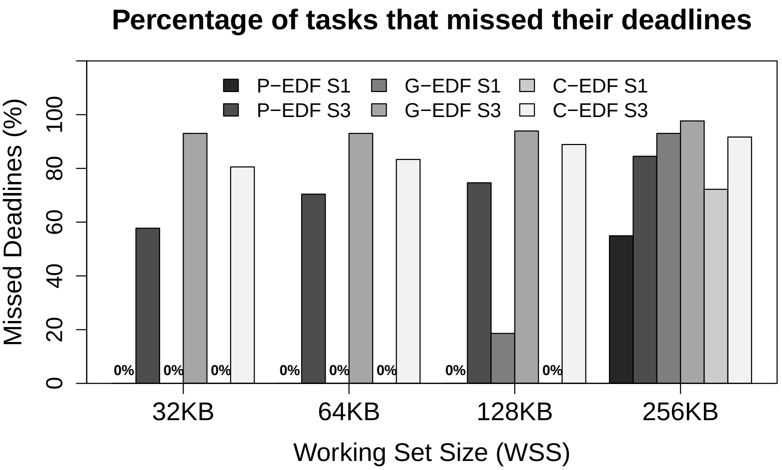


**Figure 73: Percentage of tasks that missed their deadline when varying the data size and using the P-EDF and G-EDF schedulers in S1 and S3.**

more frequently, and constantly overrun their periods. There is no handling mechanism for tasks that overrun their periods. As described in Chapter 4, the EPOS `Alarm` component releases a task with a `v` operation on a semaphore and the task waits for the next period by calling the `p` semaphore operation. When a task overrun its periods, the `p` operation found the semaphore with a value greater than one, and does not put the task to sleep. Instead, the `v` operation returns immediately and the task keeps executing.

For WSS of 128 KB in S1 with G-EDF, 18% of tasks lost their deadlines; however, the total application time is 40 s. When the system is overloaded (256 KB), global approaches (G-EDF and C-EDF) are able to handle the tasks more efficiently: a waiting task can execute as soon as a core is available (within the same cluster in case of C-EDF). The next section discusses the main observations found in the experimental evaluation.

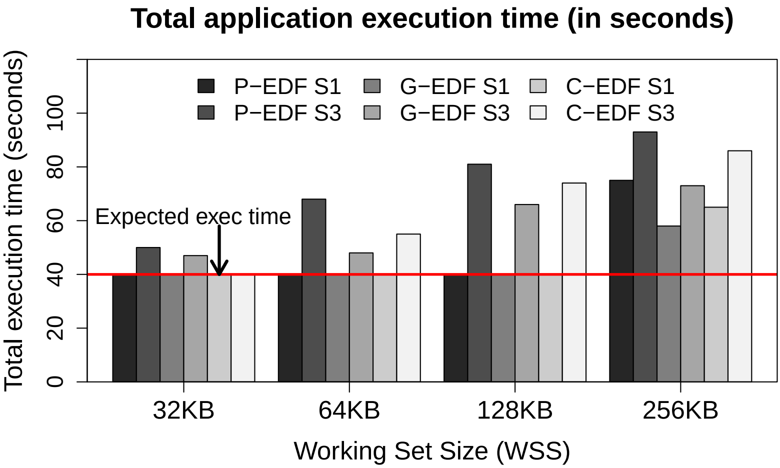


**Figure 74: Total application execution time when varying the data size and P-EDF and G-EDF with and without page coloring.**

## 6.5 DISCUSSION

Below, we summarize our main findings:

- **Cache hierarchy effects:** cache partitioning isolates task workloads and provide predictability for multicore real-time systems in terms of cache hierarchy. For P-EDF and C-EDF, page coloring supported up to 128 KB, and for G-EDF up to 64 KB. To support larger WSSs, cache partitioning could be used together with hardware techniques, such as cache locking. Even when tasks miss their deadlines with cache partitioning (128 KB for G-EDF, and 256 KB for the three schedulers), the advantage is predictability of cache accesses. It is possible to apply a data reuse method (JIANG *et al.*, 2010) and provide HRT guarantees during the theoretical schedulability analyses.
- **P-EDF, C-EDF, and G-EDF behaviors:** cache partitioning was more efficient in global approaches (G-EDF and C-EDF) for WSSs up to 64 KB, by helping to prevent inter-core communication through the cache coherence protocol. All data is able to fit in L2-cache and the invalidations in the L3-cache are reduced

when compared to S3, mainly for G-EDF. For 128 KB, page coloring was more efficient in C-EDF, because tasks can migrate inside the cluster and still access the same cache lines in L2, reducing the number of misses. For 256 KB page coloring was more efficient in P-EDF, because cache partitioning reduces contention for cache spaces when tasks are running on two logical cores at the same time.

Moreover, in an underloaded system, global approaches handle the tasks more efficiently than P-EDF, because tasks can migrate as soon as a core becomes available. As the WSS increases, there is more intra-task interference (cache misses in the same cache partitions), which increases the task execution time in S1 and reduces the scaling factor. Inter-core communication, caused by task migrations, has a considerable impact on HRT tasks (18% of missed deadlines), as shown in G-EDF with WSS of 128 KB.

- **Shared data among tasks:** when tasks execute in parallel on different cores and share data, they will access the same cache lines, causing invalidations handled by a bus snooping protocol. Cache partitioning does not solve the problem, but it helps to keep all data organized in memory. Providing a separate set of colors to shared data may improve the overall performance (CHEN *et al.*, 2009). Moreover, a shared-data-aware real-time scheduler can reduce the access serialization to the same cache line and saturation in the inter-core interconnection by avoiding the scheduling of tasks with shared data at the same time. We explore this possibility in the next two chapters.

- **Exclusive color for the RTOS:** EPOS is a lightweight RTOS. ISRs and scheduling operations in EPOS have a small footprint and use few bytes. Consequently, using an exclusive color for EPOS did not make any difference. However, ISRs of network and disk devices usually have large buffers and may benefit from having an exclusive color. Our page coloring mechanism is able to provide exclusive colors for different ISRs.

- **Pessimistic iterations:** we consider that threads always execute for the WCET. This may not be true for all applications. An extension is to incorporate a distribution method to define the number of repetitions during a period.

- **RTOS and general-purpose OS**: related work on cache partitioning and real-time systems usually use real-time patches applied to a general-purpose OS, such as Linux (MANCUSO *et al.*, 2013; KENNA *et al.*, 2013). Our evaluation was entirely carried out using an RTOS and real hardware. We believe that with the page coloring support and the new scheduling design proposed in this work, EPOS has improved its real-time support and temporal isolation among real-time tasks, providing a better multicore real-time open source research platform.

- **Cache coherence protocols and memory architectures**: we used a processor with ccNUMA memory architecture in our experiments. Usually, ccNUMA processors use MESIF or MOESI cache-coherence protocols, while UMA architectures use MESI protocol. The processor interconnect between these two architectures presents considerable differences. For example, the Intel's Quickpath Interconnect (QPI) provides higher bandwidth and lower latency for NUMA-based architectures than the Front-Side Bus (FSB), typically used in UMA architectures. Each processor has an integrated memory controller and features a point-to-point link, allowing parallel data transfer and shortest snoop request completion. In consequence, ccNUMA-based processors have a faster communication among cores. Thus, cache partitioning on UMA-based processors should theoretically be even more efficient.

# 7 STATIC COLOR-AWARE TASK PARTITIONING

In the previous chapter we have presented an evaluation of the cache partitioning impact on global, clustered, and partitioned real-time schedulers. The results have indicated that when real-time tasks share cache partitions, they may experience deadline losses mainly due to inter-core interferences: the observed WCET increased up to 15 times. The consequent variations on the execution time of real-time tasks made them miss up to 97% of their deadlines. Furthermore, the partitioned scheduler has missed less deadline compared to the global and clustered schedulers. Hence, it is desirable to avoid that two or more tasks that share same partition(s) (*i.e.,* same color(s) in case of a page coloring cache partitioning mechanism) access data at the same time. In practice this means that two or more tasks that use a same color should not be scheduled on different cores at the same time. By analyzing the results and behavior of the three schedulers in the last chapter, we propose a strategy to avoid inter-core interference when tasks share cache partitions.

In this chapter[1] we introduce a color-aware task partitioning algorithm that assigns tasks to cores according to the usage of cache memory partitions. Specifically, tasks that share one or more partitions are grouped together and the whole group is assigned to the same core using a bin packing heuristic, avoiding inter-core interference caused by the access to the same cache lines. We compare our partitioning strategy with the WFD heuristic in a real processor and using EPOS. The results indicate that our task partitioning mechanism is able to provide HRT guarantees that are not achieved by traditional task partitioning algorithms.

In summary the main contributions of this chapter are:

- We propose a Color-Aware task Partitioning (CAP) algorithm that partitions tasks to cores respecting the usage of shared cache memory partitions. Shared cache partitioning is performed by a page coloring mechanism. We assume that each task uses a set of colors that serve as input to our CAP algorithm. Then, tasks that

[1] Contents of this chapter appear in the following published paper:
G. Gracioli, A. A. Fröhlich, CAP: Color-Aware Task Partitioning for Multicore Real-Time Applications, In Proceedings of 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2014.

use the same colors are grouped together and the entire group is assigned to the same core using a bin packing heuristic.

- We compare the CAP approach with the WFD bin packing heuristic in terms of deadline misses of several generated task sets using the P-EDF scheduler. Our results indicate that it is possible to avoid inter-core interference and deadline misses by simply assigning tasks that access shared cache lines to the same core.

- We evaluate the partitioned task sets by running them on a modern 8-core processor, with shared L3 cache using a real RTOS (EPOS). The experimental evaluation on a real machine and RTOS demonstrates the effectiveness of our task partitioning mechanism.

The rest of this chapter is organized as follows. Section 7.1 presents the assumptions and notations used by the partitioning algorithm. Section 7.2 describes in details the proposed partitioning algorithm. Section 7.3 evaluates the proposed partitioning algorithm by comparing it to the WFD partitioning heuristic. Finally, Section 7.4 discusses the main findings.

## 7.1 ASSUMPTIONS AND NOTATIONS

We consider a system with a multicore processor. The processor has $m$ identical processors or cores running at a fixed clock speed. $M_{total}$ denotes the total size of the main memory available to the system. The processor has a unified last-level cache shared by all the $m$ cores. We adopted page coloring to partition the cache at software-level (there is no need for a special hardware support) —the cache is divided in $N_c$ (number of colors) partitions. The size of each partition depends on the available memory: $S_p = \frac{M_{total}}{N_c}$.

We assume a task set $\tau$ composed of $n$ periodic tasks. The $n$ tasks are scheduled using the EDF scheduling policy. We also assume that tasks share data by allocating memory from a same color set. All data that is not shared is allocated from a unique color set individually assigned to each task.

**Definition 7.1.1** *A task $T_i$ is represented as follows:*

$$T_i = \{e_i^{NC_i}, p_i, d_i, NC_i, M_{req}^i\}$$

- *$NC_i$ is the set of colors assigned to $T_i$;*
- *$e_i^{NC_i}$ is the WCET of $T_i$, when it runs with $|NC_i|$ colors and when it is also inflated by the sources of overhead (see Equation 2 in Section 2.5);*
- *$p_i$ is the period of $T_i$;*
- *$d_i$ is the relative deadline of $T_i$ ($d_i = p_i$);*
- *$M_{req}^i$ is the size of the memory region allocated to $T_i$.*

The minimum number of partitions $|NC_i|$ that minimizes the WCET $e_i^{NC_i}$ depends on how partitions/colors are assigned to tasks. We assume that the values of the set $NC_i$ are known at design time. For instance, a WCET analysis tool could estimate the value of $e_i^{NC_i}$ by varying the number of colors and returning the ideal $NC_i$ for each task $T_i$. It is important to mention that our focus is not to optimize the assignment of colors, but to provide a safe upper bound when tasks share colors. Furthermore, $e_i^{NC_i}$ is non-increasing with $NC_i$, which means that it can begin to converge when the number of partitions reaches a point in which adding more partitions does not reduce the $e_i^{NC_i}$ (KIM *et al.*, 2013).

The sum of the required memory from all tasks must be less than the available memory:

$$\sum_{i=1}^{n} M_{req}^i \leq M_{total} \tag{1}$$

**Definition 7.1.2** *Let $N_{total}$ be the set of all colors used in a task set $\tau$. $|N_{total}|$ must be less than the defined number of colors:*

$$|N_{total}| \leq N_c \tag{2}$$

**Definition 7.1.3** *Let $SC_i$ be a set of tasks that share colors (i.e., the set of tasks that allocate memory from a same color) with the task $T_i$:*

$$\forall\ T_i, T_j, T_k \in \tau.\ (((NC_i \cap NC_j) \neq \phi) \wedge ((NC_j \cap NC_k) \neq \phi)) \rightarrow i,k,j \in SC_i$$

We assume that the sum of the utilizations ($u_i = \frac{e_i^{NC_i}}{p_i}$) of all tasks that share color(s) is not greater than 100% (we call this restriction of utilizations). Formally defining:

$$\forall\ T_i \in \tau \sum_{j \in SC_i} u_j \leq 1 \tag{3}$$

Equation 3 restricts the utilization of one or more tasks that share the same colors to 100%. Note that the task $T_i$ is in the set $SC_i$ (Definition 7.1.3), which implies that its utilization is also accounted. Thus, two or more tasks that share colors can be assigned to the same core, preventing them from running in parallel on different cores, and consequently, preventing the access to the same cache lines, independently of whether the sharing was true or false. Since two or more tasks will be running on the same core, they suffer from the CPMD caused by the lost of cache affinity after preemptions. However, the WCET $e_i^{NC_i}$ already considers the run-time overhead, including the CPMD.

When tasks share a partition, the sum of the memory required by those tasks from the shared partition must not exceeded the size of a partition. Otherwise, some data would have to be allocated from other colors, incurring in cache interference. Equation 4 presents a sufficient and necessary condition for the restriction on the shared partition size to be met. For each partition $\rho$, the sum of per-partition usage of tasks that share a partition does not exceed the size of one memory partition (KIM *et al.*, 2013).

$$\sum_{\forall T_i:\ \rho \in NC_i} \frac{M_{req}^i}{|NC_i|} \leq S_p \tag{4}$$

## 7.2 COLOR-AWARE TASK PARTITIONING

We now describe a color-aware task partitioning able to provide HRT guarantees for the system model presented in the previous section, when tasks share data or colors. A partitioning algorithm is used to assign tasks in a task set to available cores (Section 2.4.2.1 has reviewed some partitioning heuristics, such as FFD, BFD, and WFD).

Figure 75 presents an overview of the CAP mechanism. A task set is composed of $n$ tasks. Each task $T_i$ has its own parameters $e_i^{NC_i}$,

$p_i$, $d_i$, $NC_i$, and $M^i_{req}$. The parameters of each task serve as input to the partitioning algorithm. The partitioning algorithm finds which tasks share colors by analyzing the set $NC_i$ of each task and uses the Equation 4 to ensure that these tasks meet the restriction of the shared partition size. The algorithm also checks whether the utilization constraints is met or not by using the Equation 3. Finally, the output of the partitioning algorithm is the assignment of those tasks that share partitions to the same core. Cache partitioning is performed at run-time by the page coloring mechanism described in Section 4.3.2. The run-time colored memory allocation should be either performed before the periodic task iteration or accounted for in the WCET. Thus, it is possible to perform run-time memory allocation without incurring in more run-time overhead. The next paragraphs describe in details each phase of the proposed partitioning approach.

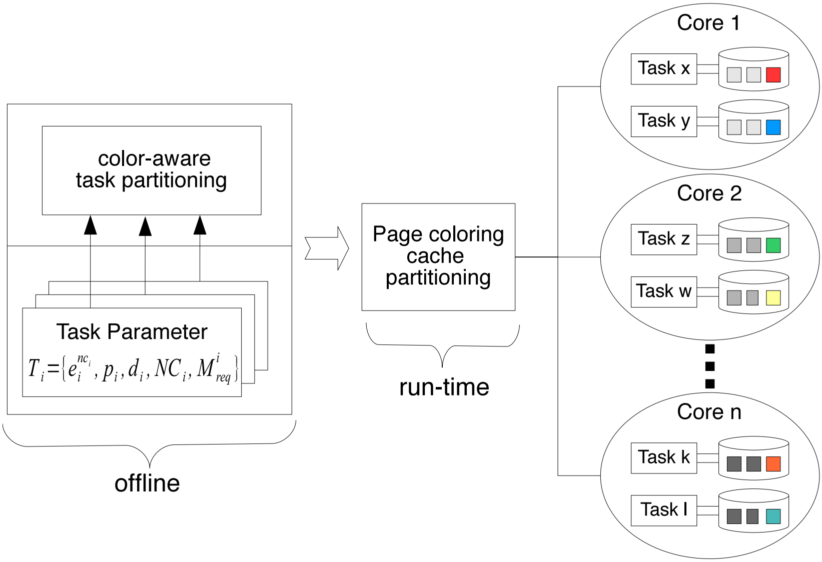

color-aware task partitioning
Task Parameter
$T_i=\{e_i^{nc_i}, p_i, d_i, NC_i, M^i_{req}\}$
offline
Page coloring cache partitioning
run-time
Core 1
Task x
Task y
Core 2
Task z
Task w
Core n
Task k
Task l

**Figure 75: Overview of the proposed color-aware task partitioning mechanism.**

We describe our partitioning algorithm in a top-bottom way. The Color-Aware Partitioning (CAP) algorithm is a variation of the BBF, FFD, NFD, or WFD. We use the WFD bin packing heuristic as an example to demonstrate the algorithm. Algorithm 1 shows the pseudo code for the CAP WFD variation. The algorithm receives a task set $\tau$, the number of available colors $N_c$, the total available memory $M_{total}$, and the number of cores $m$ as input. The output is a boolean (*partitioned*) informing whether the task set was partitioned or not and a task set assigned to each core. The algorithm begins initializing all task sets as empty and setting *partitioned* to true (lines 1 and 2). Then, the task set $\tau$ is partitioned into groups by calling the function $FindTasksWithSharedColors$, which returns the groups of tasks that share colors in the array *taskGroups* (line 3). The algorithm tests if each group of tasks satisfies the Equation 3 and 4 and returns false if a group does not satisfy them (lines 4 to 9). After guaranteeing both restrictions, the groups are sorted by decreasing order of utilizations[2] (line 10) and partitioned using the WFD heuristic (line 11). Finally, it creates and returns a task set for each partition by using the assigned groups. Note that it is possible to use any partitioning heuristic in line 11. The difference is that we are partitioning groups of tasks and not individual tasks.

**Algorithm 1** CAP_WorstFitDecreasing($\tau$, $N_c$, $M_{total}$, $m$)

**Input:** $\tau$: a task set of “n” tasks as described in Definition 7.1.1, $N_c$: available number of colors/partitions, $M_{total}$: available memory in the system, $m$: number of cores in the processor

**Output:** *partitioned*: a boolean if the WFD heuristic is able to partition the task set, *task_set*[$m$]: a task set per core

```
 1: task_set ← ∅                              ▷ Initializes all task sets as empty
 2: partitioned ← true
 3: taskGroups ← FindTasksWithSharedColors($\tau$)
 4: for each group of tasks in taskGroups do
 5:     if group does not satisfy Eq. 3 and Eq. 4 then
 6:         partitioned ← false
 7:         return {partitioned,task_set}
 8:     end if
 9: end for
10: Sort groups by decreasing order of utilization
11: Apply the WFD heuristic by groups
12: task_set ← Create task set per core
13: return {partitioned,task_set}
```

[2] A group utilization is the sum of the utilizations of all tasks in that group.

Algorithm 2 shows the pseudo code for the function *FindTasksWithSharedColors*. It receives a task set as input and returns a multi map data structure (*taskGroups*) representing the groups of tasks that share colors[3]. For example, *taskGroups*[0] contains all tasks that share at least one color in the group 0, *taskGroups*[1] contains all tasks that share at least one color in the group 1, and so on. Tasks in different groups do not share any color. The function first initializes all array positions with zero (line 1). Then, it compares the set of colors for each task and fills a specific array position whenever two tasks use at least one common color (lines 2 to 12). For instance, if $A_\tau[0][1]$ is equal to one, it means that task $T_1$ shares color with task $T_0$. If it is zero, $T_0$ and $T_1$ do not share color. The functions ends calling another function *MapTasksToGroups* to create the groups of tasks and returns the groups (lines 13 and 14).

**Algorithm 2** FindTasksWithSharedColors($\tau$)

**Input:** $\tau$: a task set of "n" tasks

**Output:** *taskGroups*: a multi map data structure representing groups of tasks that share colors

```
 1: $A_\tau[n][n] \leftarrow [0,0]$  ▷ An array representing the colors usage pattern among tasks. Initializes all positions with 0
 2: for $i \leftarrow 0$ to $n$ do
 3:     for $j \leftarrow 0$ to $n$ do
 4:         if $i \neq j$ then
 5:             $has\_shared\_data \leftarrow \tau[i].NC_i \cap \tau[j].NC_j$  ▷ $NC$ is the set of colors
 6:             if $has\_shared\_data == true$ then
 7:                 $A_\tau[i][j] \leftarrow 1$
 8:                 $A_\tau[j][i] \leftarrow 1$
 9:             end if
10:         end if
11:     end for
12: end for
13: taskGroups ← MapTasksToGroups($\tau$, $A_\tau$)
14: return taskGroups
```

[3] It is important to keep in mind that, although the *placement new operator* in C++ is used at run-time to dynamically allocate memory, its specialization to implement the page coloring mechanism in EPOS relies on constant color aliases that are know at compile-time.

The function *MapTasksToGroups* is depicted in Algorithm 3. It receives a task set and an array representing the colors usage pattern among tasks as input. The output is a multi map data structure *taskGroups* representing groups of tasks that share colors. For each task $T_i$ in the task set $\tau$, the function verifies if the current task $T_i$ is in a group and if it is not, a new group is created and $T_i$ is inserted in this group (lines 4 to 7). Then, for each task $T_j$ in $\tau$, if $T_i$ and $T_j$ share a color ($A_\tau[i][j] = 1$) the function calls *SolveShareColorsChain*, passing the multi map *taskGroups*, the current $T_j$ index ($j$), the task set $\tau$, the current *group*, and the array $A_\tau$ (lines 8 to 14). *SolveShareColorsChain* adds all tasks that share colors with $T_j$ recursively. Note that if $T_i$ does not share colors with another task, it is assigned to a new group without any other task. Finally, *taskGroups* is returned to *FindTasksWithSharedColors*, containing tasks mapped to specific groups and respecting the colors usage pattern among them.

**Algorithm 3** MapTasksToGroups($\tau$, $A_\tau$)

**Input:** $\tau$: a task set of “n” tasks, $A_\tau$: an array representing the colors usage pattern among tasks

**Output:** *taskGroups*: a multi map data structure representing groups of tasks that share colors

```
 1: $taskGroups \leftarrow \emptyset$                          ▷ Initializes all groups as empty
 2: $group \leftarrow -1$
 3: for each task $T_i$ in $\tau$ do
 4:     if $T_i$ is not in $taskGroups$ then
 5:         $group = group + 1$                         ▷ Create a new group of tasks
 6:         map $T_i$ into $taskGroup[group]$
 7:     end if
 8:     for each task $T_j$ in $\tau$ do
 9:         if $A_\tau[i][j] == 1$ then                  ▷ $T_i$ shares data with $T_j$
10:             if $T_j$ is not in $taskGroups$ then
11:                 SolveSharedColorsChain($taskGroups, j, \tau, group, A_\tau$)
        ▷ Insert all tasks that share colors into the same group
12:             end if
13:         end if
14:     end for
15: end for
16: return $taskGroups$
```

The last function, *SolveShareColorsChain*, is described in the pseudo code of Algorithm 4. It receives a multi map data structure with the task groups, the index of the current task being analyzed in the *MapTasksToGroups* function, a task set $\tau$, the current *group* id, and the array $A_\tau$ with the colors usage pattern among all tasks in the task set $\tau$. There is no output. The objective of *SolveShareColorsChain* is to add all tasks that share at least one color into the same group recursively. For each task $T_i$ in $\tau$, if $T_i$ shares a color with $T_{index}$ ($A_\tau[index][i] = 1$), the function performs two tests. First, it tests if $T_{index}$ already is in the current group, and inserts $T_{index}$ into *taskGroups*[*group*] when $T_{index}$ is not in the group. This test is necessary, because $T_{index}$ may have been inserted before (lines 3 to 5). The second test verifies if $T_i$ is in the *taskGroups*[*group*]. When the test fails, it means that $T_i$ has not been analyzed yet. Then, $T_i$ is added to *taskGroups*[*group*] and *SolveShareColorsChain* is called again, with $i$ as the new *index*. Thus, all tasks that share a color with $T_i$ are added to the same group. The process is repeated until finishing to add all tasks that share at least one color to the same task group.

**Algorithm 4** SolveShareColorsChain(*taskGroups*, *index*, $\tau$, *group*, $A_\tau$)

**Input:** $\tau$: a task set of "n" tasks, *index*: index of the task being analyzed, *group*: current group with tasks that share colors with $T_{index}$, $A_\tau$: an array representing the colors usage pattern among tasks
**Output:**

```
 1: for each task $T_i$ in $\tau$ do
 2:     if $A_\tau[index][i] == 1$ then
 3:         if $T_{index}$ is not in taskGroups then
 4:             map $T_{index}$ into taskGroup[group]
 5:         end if
 6:         if $T_i$ is not in taskGroups then
 7:             map $T_i$ into taskGroup[group]
 8:             SolveShareColorsChain($taskGroups, i, \tau, group, A_\tau$)
 9:         end if
10:     end if
11: end for
```

### 7.2.1 Example: partitioning a task set with CAP WFD

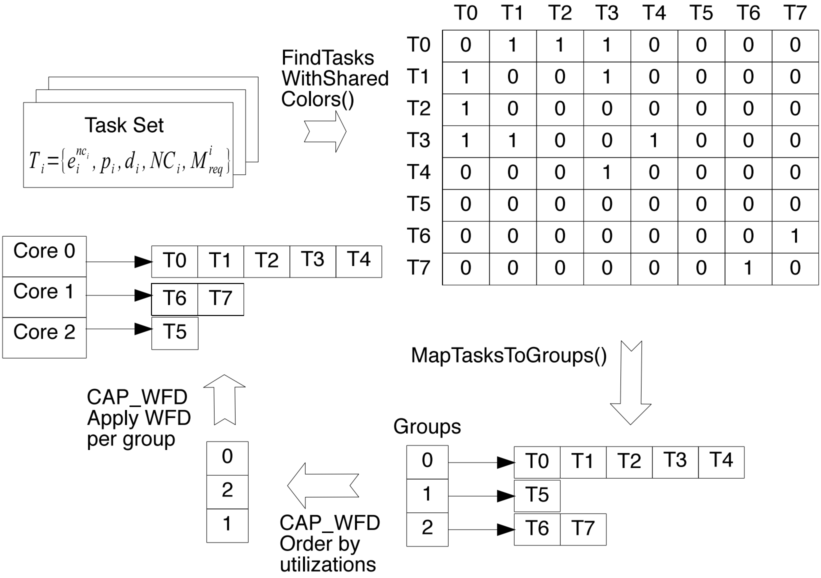
In this section, we present an example to demonstrate the CAP algorithm. Table 14 defines a task set with eight tasks and their parameters. $e$ is the WCET when tasks run with $NC$ colors, $p$ is the period, the deadline is implicit $(d = p)$, $u$ is the utilization, and $mem_{req}$ is the required memory in KB. Suppose that the target processor has three cores and the available memory $M_{total}$ is greater than the sum of the tasks' $mem_{req}$. The objective is to compare the partitioning carried out by the CAP WFD with the original WFD.

**Table 14: Parameters of a task set used to exemplify the CAP WFD algorithm.**

| $Task$ | $e$ | $p$ | $u$ | $mem_{req}$ (KB) | $NC$ |
|---|---|---|---|---|---|
| $T_0$ | 900 | 10000 | 0.09 | 1024 | {1,3} |
| $T_1$ | 6000 | 15000 | 0.4 | 64 | {1,4} |
| $T_2$ | 1500 | 5000 | 0.3 | 4 | {0,3} |
| $T_3$ | 500 | 5000 | 0.1 | 128 | {1,5} |
| $T_4$ | 1900 | 20000 | 0.095 | 32 | {5} |
| $T_5$ | 2600 | 5000 | 0.52 | 16 | {6} |
| $T_6$ | 2700 | 5000 | 0.54 | 512 | {7,8,9} |
| $T_7$ | 10000 | 50000 | 0.2 | 256 | {7,10} |

Figure 76 shows the phases of the CAP WFD algorithm, correlating them with the previously defined functions. The task set is the input for the $FindTasksWithSharedColors$ function, which mounts an array representing the colors usage pattern among tasks. For example, consider the tasks $T_0$ and $T_1$. Task $T_0$ uses the colors 1 and 3, while task $T_1$ uses colors 1 and 4. Consequently, the array positions [0][1] and [1][0] have the value of 1. The array is filled with "1s" whenever tasks share a color with each other and then it is passed to the $MapTasksToGroups$ function. $MapTasksToGroups$ creates task groups with all tasks that use the same color (set). For instance, $T_0$, $T_1$, $T_2$, $T_3$, and $T_4$ are placed in the group 0, because $T_0$, $T_1$, and $T_3$ use color 1, $T_1$, and $T_2$ use color 3, and $T_3$ and $T_4$ use color 5. $T_5$ is placed in the group 1, because it does not share any color with other tasks. $T_6$ and $T_7$ share the color 7, forming the group 2.

The next step in the algorithm is to order the groups by decrea-


Task Set
$T_i=\{e_i^{nc_i}, p_i, d_i, NC_i, M_{req}^i\}$
FindTasks WithShared Colors()
MapTasksToGroups()
Groups
0
1
2
T0 T1 T2 T3 T4
T5
T6 T7
CAP_WFD Order by utilizations
0
2
1
CAP_WFD Apply WFD per group
Core 0
Core 1
Core 2
T0 T1 T2 T3 T4
T6 T7
T5

| | T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 |
|---|---|---|---|---|---|---|---|---|
| T0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| T1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| T2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| T3 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| T4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| T5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| T6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| T7 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Figure 76: Example of the CAP WFD algorithm with eight tasks and three cores.**

sing order of utilizations and to verify if the groups satisfy Equations 3 and 4. Group 0 has utilization of 0.985, group 1 has utilization of 0.52, and group 2 of 0.74. Hence, all groups satisfy both equations and the groups order is defined as 0, 2, and 1. The last step is to assign the groups to "packs" (or cores in our case) using the WFD heuristic. Group 0 is assigned to core 0, group 2 is assigned to core 1, and group 1 is assigned to core 2. In summary, the CAP algorithm creates groups of tasks that share the same colors and assign each group to a specific core using a bin packing heuristic (FFD, BFD, NFD, or WFD).

The original WFD, in contrast, does not create groups of tasks. The algorithm simply order the task set in decreasing order of utilization and assigns each task to a core using the WFD heuristic. Figure 77 shows the task set of Table 14 partitioned by the original WFD. We can note that tasks that share colors are placed in different cores. For instance, $T_6$ and $T_7$ are in cores 0 and 1, respectively. Consequently, they would execute in parallel, causing additional overhead due to the contention for the same cache lines. This extra overhead could eventually lead to deadline losses. The next section compares the CAP WFD approach with the original WFD bin packing heuristic and shows its impact on preventing deadline misses using a real processor and RTOS.

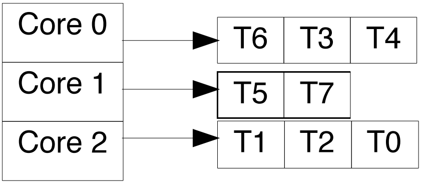


**Figure 77: The partition of the same task set as in Figure 76 with the original WFD heuristic.**

## 7.3 EVALUATION

This section describes the experimental evaluation of the CAP algorithm. The objective is to compare its performance in terms of HRT guarantees with that for traditional bin packing partitioning algorithms. We start describing the experiment methodology in Section 7.3.1. Then, we present the results of the evaluation in Section 7.3.2. We leave the discussion of the main findings for Section 7.4. All experiments in this section use the Intel i7-2600 processor (see Table 12) and EPOS RTOS.

### 7.3.1 Experiment Description

We randomly generated task sets similar to our previous experiment described in Section 6.1, adding a color number for each task. We selected the periods (all values are in ms) uniformly from {25, 50, 75, 100, 150, 200} and utilizations uniformly from the [0.1, 0.7] interval. The WCET of a task was defined according to the generated period and utilization. As our processor (Intel i7-2600) has eight cores, we fixed the number of task groups to eight and the number of colors to eight (one for each task group). For each task, we randomly selected a color between [1, 8]. In order to respect the utilization constraints in the CAP algorithm, we generated tasks of the same color until the accumulated utilization reached the [0.9, 1.0] interval. We used the P-EDF scheduling algorithm with implicit-deadlines. We also used the

CAP WFD variation, because with the WFD bin packing heuristic each core tends to have similar utilization. The CAP algorithm mounts task groups respecting their colors and assigns each task group to a core. The WFD heuristic assigns each task to a core despite its color. It it important to highlight that the generated task sets have total utilization close to HRT bounds. The objective was to run the same task sets using both partitioning approaches and then analyze whether deadlines were met or not.

Each task executes a function that allocates its WSS, reading and writing from/to the WSS (an array), in a loop, randomly. Figure 78 shows part of the function source code. Each task iterates for 200 times (ITERATIONS variable). We considered scenarios with different WSS (ARRAY_SIZE variable) similar to (CALANDRINO *et al.*, 2006): 32 KB, 64 KB, 128 KB, and 256 KB. We defined a write ratio of 20% (one write for each four readings) and 33% (one write for each two readings). By increasing the write ratio, we stimulate the cache coherence protocol to invalidate shared cache lines more often.

The number of reads and writes of a task varies depending on the WCET. To adjust the number of repetitions of each task, we used the same methodology as described in Section 6.1. Moreover, we always use the same value for the "repetitions" function argument in CAP WFD and WFD experiments. We also adjusted the repetitions parameter according to the WSS to account for the intra-task (self-evictions) and intra-core (preemption delay) cache interferences. When a task evicts its own cache lines, it increases its execution time and eventually misses a deadline as shown in Section 6.3. Each task receives a color as parameter and allocates memory for the array using that color.

### 7.3.2 Percentage of Missed Deadlines

To evaluate the ratio of missed deadlines of the CAP WFD and the original WFD heuristics, we generated ten task sets. The number of tasks in a task set varies from 23 to 30 tasks. The number of tasks in a task group varies from two to six. We then partitioned each task set using the CAP WFD and WFD algorithms. We executed the ten different partitioned task sets for 50 times, varying the WSS and write ratio as described in the previous section. We extracted the number of missed deadlines for each partitioned task set from these executions

```
#define ARRAY_SIZE KB_32 // or 64, 128, 256KB
#define ITERATIONS 200
#define MEMORY_ACCESS 16384
#define WRITE_RATIO 2 // or 4
int job(unsigned int repetitions, int id, int color)
{
    int sum = 0;
    Pseudo_Random * rand;
    int *array;
    rand = new (color) Pseudo_Random();
    array = new (color) int[ARRAY_SIZE];
    rand->seed(clock.now() + id);
    for(int i = 0; i < ITERATIONS; i++) {
        Periodic_Thread::wait_next();
        for(int j = 0; j < repetitions; j++) {
            for(int k = 0; k < MEMORY_ACCESS; k++) {
                int pos = rand->random() % (ARRAY_SIZE - 1);
                sum += array[pos];
                if((i % WRITE_RATIO) == 0) array[pos] = k + j;
            }
        }
    }
}
```

**Figure 78: Part of the task function source code.**

(over 88 hours of tests using a real hardware and an RTOS).

In EPOS, the `Alarm` component is responsible for releasing a task by calling a `v` operation of its `Semaphore` (see Section 4.2.3). Since we assume implicit-deadlines ($d_i = p_i$), we counted a missed deadline whenever the `v` operation was issued for the semaphore while its value was greater or equal to zero. In practice this means that a new task's job was released before the previous job had finished.

Figure 79 shows the percentage of missed deadlines for a WSS of 32 KB and write ratio of 33% (Figure 79(a)) and 20% (Figure 79(b)). On the x-axis, we vary the task set. On the y-axis, we present the percentage of missed deadlines for each task set. The Figures show missed deadlines only for the WFD heuristic, because *all tasks in all task sets partitioned by the CAP WFD algorithm were able to meet their deadlines.*

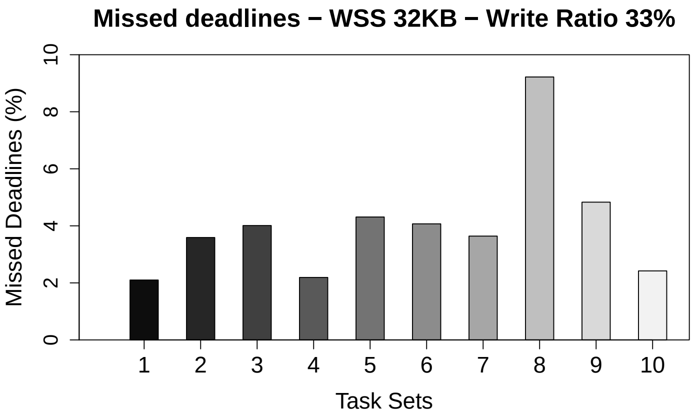
Note that Figure 79(a) and 79(b) have different scales for the y-axis, because decreasing the write ratio causes the cache coherence protocol to invalidate less cache lines, improving the system performance. However, for both write ratios, all task sets partitioned by the WFD algorithm lose deadlines: from 2.10% (task set 1) to 9.22% (task set 8) for write ratio of 33% and from 1.09% (task set 1) to 3.16% (task set 8) for write ratio of 20%.

In general, the task sets have similar performance for WSS of 32 KB. The task set 8, in special, have two task groups with five tasks in each group. When the task set is partitioned by the WFD algorithm, the tasks in each group are assigned to different cores, causing contention on the arrays and increasing their execution times. The CAP WFD, in contrast, avoids the contention and meets the application HRT constraints.

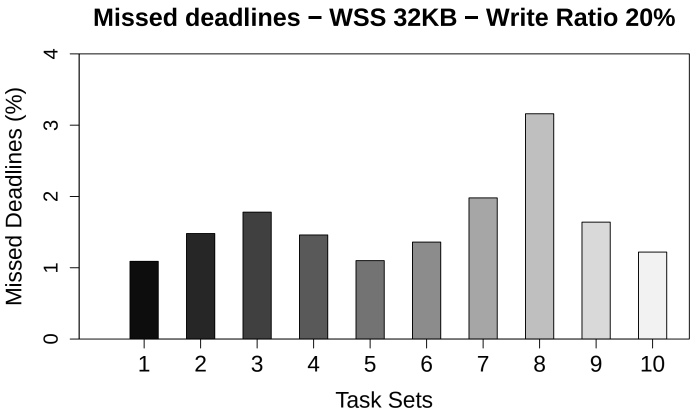
Figure 80 shows the percentage of missed deadlines for a WSS of 64 KB and write ratio of 33% (Figure 80(a)) and 20% (Figure 80(b)). On the x-axis, we vary the task set and on the y-axis, we present the percentage of missed deadlines for each task set. For write ratio of 33%, the deadline miss ratio varies from 5.70% to 34.27% and for write ratio of 20% it varies from 4.53% to 29.42%, both for the task sets 4 and 8, respectively.

Comparing the results in Figures 79 and 80, the percentage of missed deadlines increases due to the array size. Our processor has four L2-caches with 256 KB and 8-ways each, an 8 MB shared L3-cache with 16-ways, 64 bytes per cache line, and 8192 sets. Each set has 16-ways and each way can store a cache line. With a WSS of 32 KB, each task demands eight pages of the same color. With a WSS of 64 KB, each task demands 16 pages of the same color. Consequently, there is more contention for the cache ways both in L2- and L3-cache. The CAP WFD is able to decrease this contention by preventing tasks that share a color from running in parallel on different cores at the same time. Thus, the cache coherence protocol does not delay the tasks by invalidating cache lines due to the false sharing. The only delay a task may experience is the preemption delay, caused by the lost of cache affinity after a preemption, and intra-task cache misses in the L2-cache, caused by the task's own data accesses.

Figure 81 shows the percentage of missed deadlines for a WSS of 128 KB and write ratio of 33% (Figure 81(a)) and 20% (Figure 81(b)). On the x-axis, we vary the task set and on the y-axis, we present the percentage of missed deadlines for each task set. For write ratio of 33%, the deadline miss ratio varies from 48.17% (task set 5) to 78.84% (task



(a)

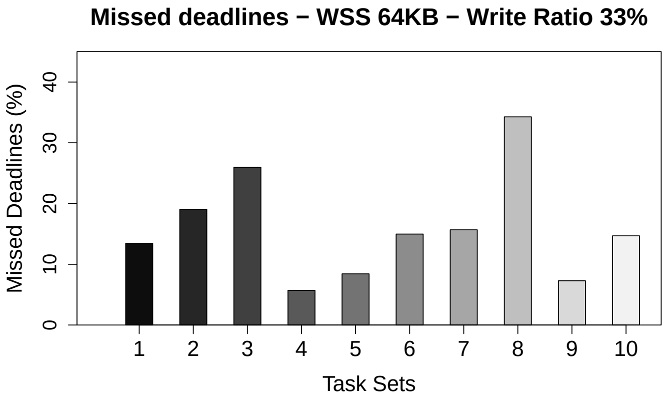


(b)

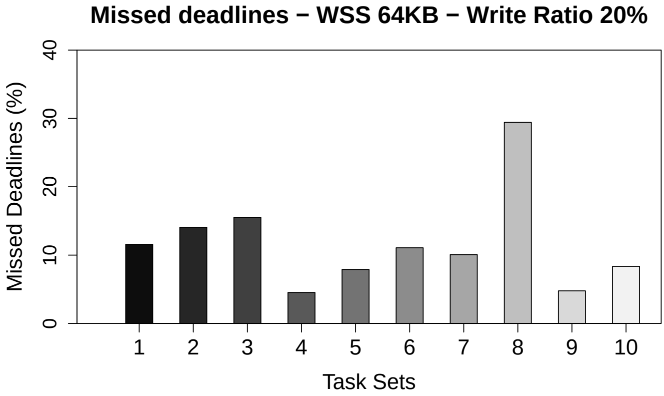
**Figure 79: Percentage of missed deadlines of the WFD heuristic. (a) WSS of 32KB and write ratio of 33%; (b) WSS of 32KB and write ratio of 20%.**

set 10) and for write ratio of 20% it varies from 43.39% (task set 5) to 72.60% (task set 10).

Comparing this behavior with the previous two experiments, the



(a)



(b)

**Figure 80: Percentage of missed deadlines of the WFD heuristic. (a) WSS of 64KB and write ratio of 33%; (b) WSS of 64KB and write ratio of 20%.**

percentage of missed deadlines has increased even more. This is due to the same reasons as for the increase from 32 to 64 KB observed in Figure 80. With WSS of 128 KB, each task demands 32 pages from the same color, causing more cache contention and more preemption delay due to cache lines that were evicted after preemptions. It is worth mentioning again that we adjusted the repetitions parameter (see Figure 78) of each task as we increased the array size in order to decrease the application execution time and account for the intra-task cache misses. Moreover, with 128 KB, and two tasks running on different hyperthreads, the L2-cache was more full and more L2-cache misses occurred.

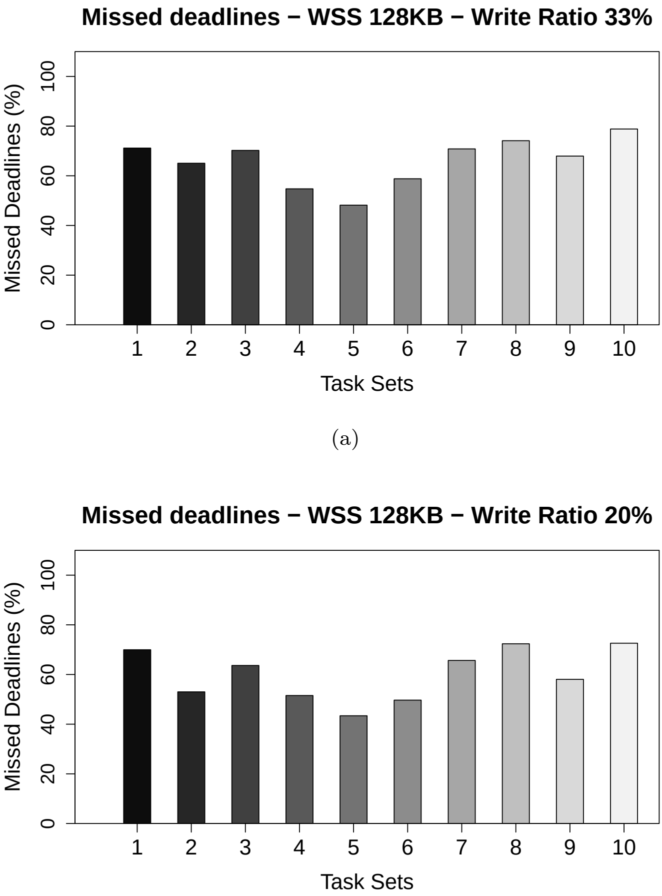
Finally, Figure 82 shows the percentage of missed deadlines for a WSS of 256 KB and write ratio of 33% (Figure 82(a)) and 20% (Figure 82(b)). On the x-axis, we vary the task set. On the y-axis, we present the percentage of missed deadlines for each task set. For write ratio of 33%, the deadline miss ratio varies from 8.96% (task set 4) to 24.94% (task set 8) and for write ratio of 20% it varies from 5.19% (task set 4) to 24.73% (task set 8). In comparison with the previous results (Figure 81), the percentage of missed deadline has decreased. With WSS of 256 KB, the preemption delay is greater, which means that we had to reduce the repetitions parameter of each task. Thus, there is less contention for the cache lines within the same color. However, the deadline miss ratio is similar to the Figure 80 when tasks ran with 64 KB.

## 7.4 DISCUSSION

During our work, we observed a number of interesting facts. Here we discuss our main observations on the experiments:

- **Color-Aware task Partitioning:** our Color-Aware task Partitioning (CAP) assigns tasks to processors respecting the colors (*e.g.,* memory partitions) usage by tasks. We generated 10 different task sets that were partitioned using the CAP WDF and the original WFD heuristics. We ran each partitioned task set for 50 times varying the WSS of each task from 32 to 256 KB. From these executions, we extracted the percentage of missed deadlines. By simply assigning tasks to processors, CAP prevented tasks from accessing shared cache lines and thus met the HRT application requirements without missing any deadline.

- **Working set size:** we analyzed the behavior of the multicore platform by varying the WSS (32, 64, 128, and 256 KB). With

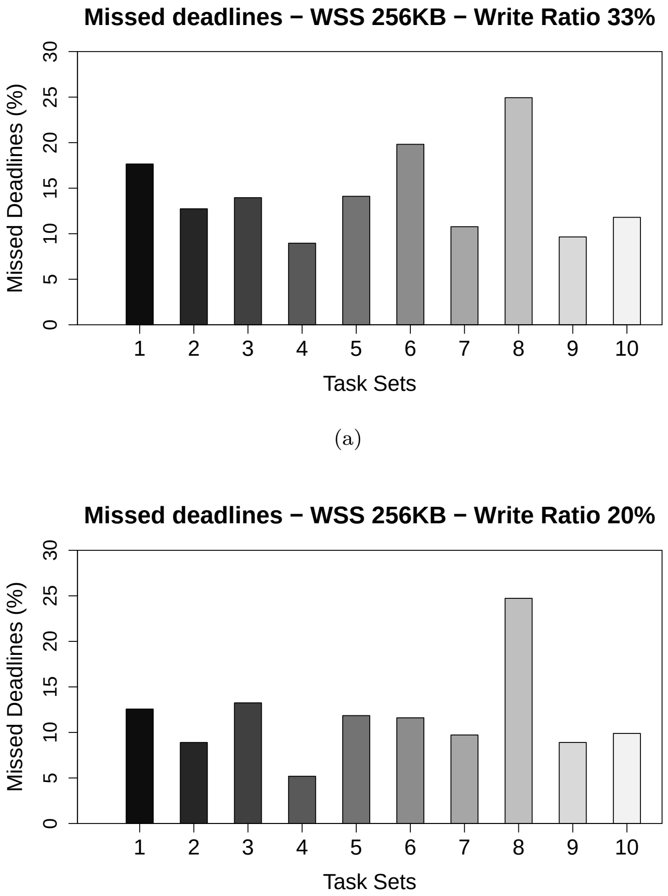


(a)

(b)

**Figure 81: Percentage of missed deadlines of the WFD heuristic. (a) WSS of 128KB and write ratio of 33%; (b) WSS of 128KB and write ratio of 20%.**

128 KB, the tasks have the greatest deadline miss ratio due to the more contention for shared cache lines and preemption delay. With 256 KB, the preemption delay is higher due to L2-cache misses. For 32 KB, mostly of the tasks data fits into the L2-



(a)

(b)

**Figure 82: Percentage of missed deadlines of the WFD heuristic. (a) WSS of 256KB and write ratio of 33%; (b) WSS of 256KB and write ratio of 20%.**

cache and the deadline miss ratio is smaller than in the other WSSs.

- **Write ratio:** we ran the partitioned task sets using two different write ratios (20 and 33%). By changing the write ratio, we stimulate the cache coherence protocol to invalidate more or less cache lines. For all WSS variation, the execution with the write ratio of 20% has missed less deadlines as expected.

- **True and false sharing:** the application used in our experiments did not explicitly share data (see Figure 78). Tasks allocate memory using a color received as argument. Some tasks allocate memory from the same color, as described in Section7.3.1, causing concurrent accesses to shared cache lines (false sharing). In an application with shared data, a mutual exclusion protocol, such as the multiprocessor priority ceiling protocol (MPCP) (RAJKUMAR *et al.*, 1988) and flexible multiprocessor locking protocol (FMLP) (BLOCK *et al.*, ), must be used to ensure correctness. In such applications, the blocking times caused by priority inversion and the implicit delay caused by the cache coherence protocol must also be taken into account while estimating the WCET. Such protocols can be easily accommodated under EPOS page coloring mechanism, which provides a straightforward API to allocate specific memory partitions to shared data.

- **Colors assignment:** in our task model (see Section 7.1), we assume that tasks have a set of colors that represents memory partitions. The process of assigning colors to tasks is a complex optimization problem. To assign the optimal combination of colors, one must analyze the data usage for each task and thus select the best combination of colors that minimizes the cache delays for a specific processor. However, once an heuristic is chosen, EPOS can easily incorporate it under its memory management abstractions, encapsulating the specificities of the underlying architecture in well-defined micro components. Memory partitioning is not restricted to applications. It can also be used for internal OS data structures, thus enabling a fine-grain color assignment.

- **Other scheduling policies:** we ran our experiment using the P-EDF scheduling policy. It is also possible to apply the CAP to any other scheduling policy, such as RM and DM. The difference would be restricted to the processor capacity given by each scheduling algorithm.

- **Other partitioning heuristics:** we executed our experiments using the WFD heuristic. However, as stated before, it is possible to apply any partitioning heuristic in CAP algorithm. The strategy would be forming groups of tasks that share colors and then applying a partitioning heuristic using the entire group instead of single tasks.

- **Run-time overhead:** since task partitioning is performed offline (*i.e.,* before the execution of the application), it does not add any overhead to the associated run-time overhead partitioned scheduler. Chapter 5 has shown that a partitioned scheduler (P-EDF) has a smaller run-time overhead than clustered (C-EDF) and global (G-EDF) approaches, which improves the schedulability ratio of HRT applications (BASTONI *et al.*, 2010b; GRACIOLI *et al.*, 2013).

- **Comparison with related work:** LWFG is the most similar partitioning algorithm (LINDSAY, 2012). However, our partitioning algorithm assumes that a cache partitioning mechanism based on page coloring is available and each task uses a set of colors. Thus, it is possible to determine which tasks share cache lines by inspecting the colors of each task. In contrast, in the LWFG approach, it is not clear how the algorithm is aware of data sharing among tasks. Moreover, LWFG is proposed for SRT system, since whenever the next fit heuristic fails to partition a task group, the task that shares the least memory with the first task is removed from the group, and partitioning is retried with the smaller group. Thus, inter-core interference is not avoided, which may incur in deadline losses. CAP limits the utilization of a task group in 100% in order to avoid inter-core interference. Furthermore, it is straightforward to incorporate the idea of task group splitting in the CAP algorithm, and thus provide similar performance for SRT systems as LWFG. Also, CAP is designed to be integrated to any bin packing heuristic, including the next fit decreasing, used by LWFG. The performance evaluation of LWFG was carried out using a real-time patch for the Linux kernel, which may limit the observed gains, because the inherent non real-time behavior of Linux, as shown in Chapter 5. Instead, we evaluated the CAP algorithm using an RTOS, without the excessive run-time overhead introduced by a GPOS.

  Nemati *et al.* also proposed a partitioning algorithm that groups tasks with shared resources and assigns the entire group to a same core (NEMATI *et al.*, 2009). Resource sharing among tasks is identified by a matrix. Although similar to our algorithm, the proposed approach was not evaluated, neither simulated nor in a real hardware, and it is not clear how one could construct the resource sharing matrix.

Suzuki *et al.* proposed two algorithms to decrease conflicts at the cache and DRAM bank levels (SUZUKI *et al.*, 2013). However, the allocation algorithms were not evaluated in a real hardware nor in terms of real-time guarantees. Moreover, transitive color sharing seems not to be treated by the proposed memory interference model. Our page coloring mechanism could be easily extended to also support bank coloring.

Other cache-aware real-time scheduler was proposed by Calandrino *et al.* (CALANDRINO; ANDERSON, 2008; CALANDRINO; ANDERSON, 2009). In their approach, the objective is to provide SRT guarantees. At run-time, HPCs are used to estimate the WSS of tasks. Then, tasks are scheduled in a way that cache thrashing is avoided. Two assumptions limit the usage of the proposed approach: (i) applications must not consume more memory than the shared cache size; and (ii) tasks do not share data. $FP_{CA}$ is another cache-aware scheduling algorithm that divides the shared cache space into partitions (GUAN *et al.*, 2009). Tasks are scheduled in a way that at any time, any two running tasks' cache spaces (*e.g.,* a set of partitions) are non-overlapped. A task can execute only if it gets an idle core and enough cache partitions. Thus, both proposed cache-aware multicore real-time schedulers use a task model that does not allow data sharing among tasks.

Kim *et al.* proposed a cache management scheme that assigns to each core a set of private cache partitions to avoid inter-core interference (KIM *et al.*, 2013). However, tasks within each core can share cache partitions. They bound the penalties due to the sharing of cache partitions by accounting for them as CPMD when performing the schedulability analysis. The main drawback of this approach is the difficulty of estimating the penalties incurred by the cache coherence protocol. In fact, to the best of our knowledge, no existing static analysis technique is able to account for the effects of the coherence protocol. Thus, the usage of proposed cache management scheme is restricted.

# 8 DYNAMIC COLOR-AWARE SCHEDULING ALGORITHM

In the previous chapter we have proposed the CAP algorithm to assign tasks to cores respecting their usage of colors (*i.e.,* memory partitions). However, the system model used in CAP has two main drawbacks:

- First, all tasks in a task set have HRT constraints. This scenario may not be true for every real-time application. In fact, emerging real-time workloads, such as multimedia and entertainment and some business computing applications, have highly heterogenous timing requirements and consist of HRT, SRT, and best-effort (BE) tasks (BRANDENBURG; ANDERSON, 2007). In such applications, there are few HRT tasks, which represent a small system utilization, and their deadlines must be guaranteed (RAJKUMAR, 2006). Also, deadline tardiness for SRT tasks and response times for BE jobs must be minimized (BRANDENBURG; ANDERSON, 2007).

- Second, the number of colors in a processor may be smaller than the number of real-time and BE tasks. Consequently, real-time and BE tasks may eventually share colors or memory partitions and the utilization of these tasks that share memory partitions may be greater than 100% (restriction of utilizations in the CAP system model). We have shown in the previous chapter that when tasks share colors, the contention for shared cache lines caused by the cache coherence protocol may lead to deadline misses. Thus, color sharing and the execution of tasks that use the same color in parallel on different cores should be avoided whenever possible.

In this chapter we propose a two-phase multicore real-time scheduler to deal with these two main drawbacks. The first phase is an extension to the CAP algorithm. HRT tasks are not allowed to share colors with any other task. SRT and BE tasks can share colors with each other. As in CAP, those tasks are also grouped together, but the group utilization is allowed to be greater than 100% or greater than the capacity given by the scheduling policy for each core. When a group utilization is greater than the processor capacity, we perform a group splitting similar to the LWFG approach (LINDSAY, 2012), removing the task with smaller utilization from the group and trying to partition the group again. The second phase is performed at run-time and consists of collecting tasks' information through the use of HPCs to detect when BE tasks are interfering with real-time ones.

In summary the main contributions of this chapter are:

- We propose a two phases scheduling algorithm able to deal with a heterogeneous real-time application. The system model considers a real-time application composed of HRT, SRT, and BE tasks. HRT tasks do not share colors with other HRT tasks nor with SRT or BE tasks. SRT tasks can share colors with other SRT tasks and BE tasks. The scheduling mechanism provides timing guarantees for HRT tasks, while minimizing deadline tardiness for SRT tasks and allowing the execution of BE tasks whenever possible.
- The first phase of the scheduling mechanism is an extension to the CAP algorithm, named Color-Aware task Partitioning with Group Splitting (CAP-GS). CAP-GS allows a group of tasks to have a utilization greater than the capacity of a processor considering a specific scheduling policy. When this happens, the task with smaller utilization is removed from the group and the group partitioning is performed again. This step is repeated until the group is partitioned or partitioning fails. Thus, we keep tasks that share colors in the same processor whenever possible, minimizing the contention for shared cache lines, while maintaing HRT deadlines.
- The second phase is performed at run-time. HPCs are used to collect information of cache coherence activities initiated by the cores. This information is then used by the scheduler to dynamically prevent the execution of BE tasks that share colors with SRT tasks. By doing so, we reduce the contention for shared cache lines at run-time and improve deadline tardiness of SRT tasks.
- We design and implement the dynamic phase of our scheduler in EPOS. We show how a component-based RTOS allows a straightforward integration of HPCs with scheduling algorithms.
- We compare the dynamic scheduler with CAP-GS without dynamic optimizations and BFD partitioning heuristic in terms of

deadline misses and tardiness of several generated task sets using the P-RM scheduler. Our results indicate that it is possible to improve deadline tardiness and to provide HRT guarantees by combining a color-aware task partitioning with dynamic scheduling optimizations.

- We evaluate the dynamic scheduler in a modern 8-core processor, with shared L3 cache using EPOS. The experimental evaluation on a real machine and RTOS demonstrates the effectiveness of our dynamic scheduler.

The rest of this chapter is organized as follows. Section 8.1 presents the assumption, notation, and task model used by the proposed multicore real-time scheduler. Section 8.2 describes in details the two phases of the algorithm. Section 8.3 evaluates the dynamic algorithm by comparing it to CAP and BFD partitioning approaches. Finally, Section 8.4 discusses the main findings.

## 8.1 ASSUMPTIONS AND NOTATIONS

The system model considered in this chapter is similar to that used by the CAP algorithm, which is described in Section 7.1. In this section, we describe the system model used by the dynamic scheduler, presenting its assumptions and notations.

We consider a system with a multicore processor. The processor has $m$ identical processors or cores running at a fixed clock speed. $M_{total}$ denotes the total size of the main memory available to the system. The processor has a unified last-level cache shared by all the $m$ cores. We adopted page coloring to partition the cache at software-level (there is no need for a special hardware support) —the cache is divided in $N_c$ (number of colors) partitions. The size of each partition depends on the available memory: $S_p = \frac{M_{total}}{N_c}$.

We assume a task set $\tau$ composed of $n$ periodic real-time tasks. The $n$ real-time tasks are scheduled concurrently with independent BE tasks on the $m$ processors. We assume that a periodic real-time task is either a *HRT task* or a *SRT task*. A HRT task must never miss a deadline. SRT tasks may experience deadline tardiness and BE tasks execute whenever a core is not executing a real-time task. HRT tasks have higher priorities than SRT tasks. To allow this priority difference between HRT and SRT tasks, we use the RM scheduling policy, which is a FP scheduling algorithm (see Section 2.4). Real-time and BE tasks are assigned to a core and do not migrate to another core.

We define priority ranges for HRT and SRT tasks. HRT tasks have priorities ranging from 0 (highest) to HRT_PRIORITY_LIMIT. SRT tasks have priorities ranging from HRT_PRIORITY_LIMIT + 1 to SRT_PRIORITY_LIMIT. BE tasks have priorities greater than SRT_PRIORITY_LIMIT.

We also assume that HRT tasks have private colors, *i.e.*, HRT tasks do not share colors with each other nor with SRT or BE tasks. SRT, instead, may share colors with each other and with BE tasks. Thus, we eliminate the restriction of utilizations presented in the CAP system model.

**Definition 8.1.1** *A task $T_i$ is represented as follows:*

$$T_i = \left\{e_i^{NC_i}, p_i, d_i, P_i, NC_i, M_{req}^i, Type_i\right\}$$

- *$NC_i$ is the set of colors assigned to $T_i$;*
- *$e_i^{NC_i}$ is the WCET of $T_i$, when it runs with $|NC_i|$ colors and when it is also inflated by the sources of overhead (see Equation 2 in Section 2.5);*
- *$p_i$ is the period of $T_i$;*
- *$d_i$ is the relative deadline of $T_i$ ($d_i = p_i$);*
- *$P_i$ is the priority of $T_i$.*
- *$M_{req}^i$ is the size of the memory region allocated to $T_i$;*
- *$Type_i$ is the type of the real-time task: HRT or SRT or BE.*

A BE task is assumed to have a utilization equal to zero. The priority $P_i$ of a HRT task $i$ is defined according to the RM policy: the shorter the period, the greater the priority. For a SRT task $j$, we add HRT_PRIORITY_LIMIT + 1 to $P_j$ to ensure that its priority is in the range [HRT_PRIORITY_LIMIT + 1, SRT_PRIORITY_LIMIT]. The priority of a BE task must be in the range of [SRT_PRIORITY_LIMIT + 1, $\infty$].

As in the CAP algorithm, we assume that the number of partitions $|NC_i|$ that minimizes the WCET $e_i^{NC_i}$ is known at design time.

Moreover, the WCET $e_i^{NC_i}$ already considers the run-time overhead. Also, the sum of the required memory from all tasks must be less than the available memory:

$$\sum_{i=1}^{n} M_{req}^{i} \leq M_{total} \tag{1}$$

**Definition 8.1.2** *Let $N_{total}$ be the set of all colors used in a task set $\tau$. $|N_{total}|$ must be less than the defined number of colors:*

$$|N_{total}| \leq N_c \tag{2}$$

When tasks share a partition, the sum of the memory required by those tasks from the shared partition must not exceeded the size of a partition. Otherwise, some data would have to be allocated from other colors, incurring in cache interference. Equation 3 presents a sufficient and necessary condition for the restriction of the shared partition size to be met. For each partition $\rho$, the sum of per-partition usage of tasks that share a partition does not exceed the size of one memory partition (KIM *et al.*, 2013).

$$\sum_{\forall T_i:\ \rho \in NC_i} \frac{M_{req}^{i}}{|NC_i|} \leq S_p \tag{3}$$

**Definition 8.1.3** *Let $\tau_{hrt}$ be the set of HRT tasks and $\tau_{srt}$ be the set of SRT tasks. $u_{hrt}$ denotes the utilization of HRT tasks and $u_{srt}$ denotes the utilization of SRT tasks:*

$$u_{hrt} = \sum_{T_i \in \tau_{hrt}} u_i \qquad \textit{and} \qquad u_{srt} = \sum_{T_i \in \tau_{srt}} u_i$$

As we are using the RM scheduling policy, the capacity of a processor is given by its schedulability test: $u_{sum} \leq n(2^{1/n} - 1)$, where $n$ represents the number of tasks in the task set. Note that when $n$ tends to a large number ($n \rightarrow \infty$), the upper bound $n(2^{1/n} - 1)$ converges to $\ln 2 \approx 0.69$.

We assume that the utilization of all HRT tasks is not greater than a constant $\alpha$ (defined at design time) multiplied by 0.69 (the processor capacity):

$$u_{hrt} \leq 0.69 \times \alpha \tag{4}$$

Thus, we know the available utilization for SRT and BE tasks at design time. Consequently, the utilization of all SRT tasks is defined as:

$$u_{srt} = u_{sum} - u_{hrt} \tag{5}$$

## 8.2 ALGORITHM DESCRIPTION

We now describe the two phases of the proposed dynamic scheduler. The first phase, named CAP-GS, is an extension of the CAP algorithm and it is described in Section 8.2.1. Section 8.2.2 demonstrates how CAP-GS partitions a task set. The second phase is performed at run-time and it is described in Section 8.2.3.

### 8.2.1 Color-Aware Task Partitioning with Group Splitting

The Color-Aware task Partitioning with Group Splitting (CAP-GS) extends the CAP algorithm described in the previous chapter. In CAP-GS, tasks that share colors (SRT and BE) are also grouped together, but they are allowed to have a utilization greater than the capacity of a processor given by the scheduling algorithm (*i.e.,* RM schedulability test). Algorithm 5 shows the pseudo code for the CAP-GS. The algorithm receives a task set $\tau$, the number of colors $N_c$, the total available memory $M_{total}$, and the number of cores $m$ as input. The output is a boolean (*partitioned*) informing whether the task set was partitioned or not and a task set assigned to each core.

The algorithm calls the function $FindTaskWithSharedData$ to find all tasks that share at least one color and group them together forming task groups that share colors (array *taskGroups* in the line 1). $FindTaskWithSharedData$ is the same function used by CAP depicted in Algorithm 2. Then, the task groups are sorted by decreasing order of utilization (line 2). From line 3 to line 13, CAP-GS partitions only HRT tasks using the FFD heuristic. The idea is to first assign

**Algorithm 5** CAP-GS($\tau$, $N_c$, $M_{total}$, $m$)

**Input:** $\tau$: a task set of “n” tasks as described in Definition 7.1.1, $N_c$: available number of colors/partitions, $M_{total}$: available memory in the system, $m$: number of cores in the processor

**Output:** *partitioned*: a boolean if the CAP-GS heuristic is able to partition the task set, *task_set*[$m$]: a task set per core

```
taskGroups ← FindTaskWithSharedData($\tau$)
Sort groups by decreasing order of utilization
for each group of tasks in taskGroups do
    if group does not satisfy Eq. 3 then
        partitioned ← false
    else if group has 1 HRT task then
        partitioned ← assigns this group to a core using FFD
    else
        Error: HRT task shares colors
    end if
    if partitioned is false then  ▷ could not partitioned HRT tasks
        return {partitioned, task_set}
    end if
end for
for each group of tasks in taskGroups do
    if group has 1 or more SRT or best-effort tasks then
        if group does not satisfy Eq. 3 then
            partitioned ← false
            return {partitioned, task_set}
        end if
        partitioned ← assigns this group to a core using BFD
        if partitioned is false then
            if group has only 1 task then
                return {partitioned, task_set}   ▷ partitioning failed
            end if
            taskGroups ← RemoveTask($\tau$, taskGroups, group)
            group ← group − 1          ▷ try again with the last group
        end if
    end if
end for
task_set ← Create task set per core
return {partitioned, task_set}
```

HRT tasks for the first available processors, decreasing the inter-core interference with HRT tasks caused by the color sharing among SRT and BE tasks. A task group formed by a HRT task has only one task, because it does not share any color with other HRT, SRT, or BE tasks. This condition is tested in the lines 6 to 9. In line 4, the algorithm verifies whether the HRT task satisfies or not the restriction of the partition size (Equation 3), aborting the partitioning process if the task does not satisfy the equation.

The next step in the algorithm is to partition SRT and BE tasks. BE tasks are executed whenever a processor is not executing any real-time task. Thus, from the task partitioning point of view, BE tasks are considered to have a utilization equal to zero and could be assigned to any processor. For each group formed by SRT and BE tasks, the algorithm verifies if the group satisfies the Equation 3 (from line 17 to 19) and if the group does, CAP-GS tries to partition the group using the BFD heuristic (line 21). If it is not possible to partition the whole group (line 22), the algorithm tests if the group has only one task (line 23), which means that the group possibly has been split before and the task set is not schedulable, *i.e.*, it is not possible to partition the task set. If the group has more than one task, the algorithm removes the task with smaller utilization from the group by calling the function *RemoveTask* (line 26), and tries to partition the same group again now that it has one less task and, consequently, a smaller utilization (line 27). Finally, CAP-GS creates a task set per each core (line 31) and returns the task set (line 32).

Algorithm 6 shows the pseudo code for the *RemoveTask* function. The objective of this function is to remove the task with the smaller utilization from a group. *RemoveTask* receives as input the task set $\tau$, the array with tasks separated in groups formed by all tasks that share at least one color (*taskGroups*), and the current group index which is being partitioned (*group*). The output is an array with tasks separated in groups that has a new group composed only by the task that has been removed from the current *group*.

In line 1, the function finds the task with smaller utilization within the task group. Then, this task is removed (line 2) and a new group is created with the tasks from the *taskGroup* (line 3) and adding the removed task to the new group (line 4). In the end, the new task group is returned (line 5).

**Algorithm 6** RemoveTask($\tau$, *taskGroups*, *group*)

**Input:** $\tau$: a task set of "n" tasks as described in Definition 7.1.1, *taskGroups*: a multi map with tasks organized in groups that share one or more colors, *group*: an integer representing the current group being partitioned

**Output:** *new_taskGroups*: a multi map with tasks organized in groups that will have one more group

```
find task with the smallest utilization within the $taskGroups[group]$
remove this task from the $taskGroups[group]$
$new\_taskGroups \leftarrow taskGroups + 1$    ▷ Creates a new group and keeps the old groups
insert the removed task in the new group $new\_taskGroups[group + 1]$
return $new\_taskGroups$
```

### 8.2.2 Example: partitioning a task set with CAP-GS

We present an example to demonstrate how the CAP-GS algorithm partitions a task set. Table 15 shows a task set composed of ten tasks and their parameters as in Definition 8.1.1: $e$ is the WCET when the task runs with $NC$ colors, $p$ is the period, $P$ is the priority, the deadline is implicit ($d = p$), $u$ is the utilization, $mem_{req}$ is the required memory in KB, and *type* is the class of the task (HRT, SRT, or BE). The target processor has three cores and the available memory $M_{total}$ is enough to accommodate all memory required by the tasks. The priorities for HRT tasks are defined according to the RM policy: the shorter the period, the greater the priority. For SRT we add 10001 (*i.e.,* the HRT_PRIORITY_LIMIT + 1) to the period and the resulting value is used as priority, and for BE tasks, their priorities are greater than 100000, which is greater than the lowest SRT task priority. Thus, we keep the RM policy proportional within each class. The objective is to illustrate the task partitioning performed by CAP-GS, CAP, and BFD (non-cache aware) heuristics. We define the capacity of each core as 0.75 (*i.e.,* 4 tasks per core – $4 \times (2^{1/4} - 1) \approx 0.75$).

Let us first present the partitioning with CAP-GS. The tasks that share at least one common color are grouped together. Tasks $T_3$, $T_5$, $T_6$, and $T_8$ form a group (with utilization of 0.80), because they share the color 7, and tasks $T_4$, $T_7$, and $T_9$ form another group (with

**Table 15: Parameters of a task set used to exemplify the CAP-GS algorithm.**

| $Task$ | $e$ | $p$ | $P$ | $u$ | $mem_{req}$ | $type$ | $NC$ |
|---|---|---|---|---|---|---|---|
| $T_0$ | 2000 | 10000 | 10000 | 0.2 | 16 | HRT | {1,4} |
| $T_1$ | 1500 | 5000 | 5000 | 0.3 | 12 | HRT | {2,3} |
| $T_2$ | 1000 | 10000 | 10000 | 0.1 | 8 | HRT | {0} |
| $T_3$ | 10000 | 50000 | 60001 | 0.2 | 32 | SRT | {6,7} |
| $T_4$ | 1500 | 5000 | 15001 | 0.3 | 16 | SRT | {5,8} |
| $T_5$ | 4800 | 12000 | 22001 | 0.4 | 64 | SRT | {7,9} |
| $T_6$ | 1000 | 10000 | 20001 | 0.1 | 8 | SRT | {7,11} |
| $T_7$ | 2000 | 20000 | 30001 | 0.1 | 4 | SRT | {8} |
| $T_8$ | 3000 | 15000 | 25001 | 0.1 | 20 | SRT | {7,10} |
| $T_9$ | - | - | 100000 | - | 64 | BE | {8,12} |

utilization of 0.4), because they share the color 8. Note that the task $T_9$ is a BE task and its utilization is assumed to be zero. Each of the other tasks form a group individually. The next step is to order the groups in decreasing order of utilizations. The groups order is defined as: $T_3$, $T_5$, $T_6$, and $T_8$ (group of SRT tasks), $T_4$, $T_7$, and $T_9$ (group of SRT tasks and a BE task), $T_1$ (HRT), $T_0$ (HRT), and $T_2$ (HRT). Then, CAP-GS partitions HRT tasks using the FFD heuristic and SRT tasks using the BFD heuristic.

Figure 83 shows the task set partitioned by CAP-GS. The HRT tasks are assigned to the core 0, respecting the order of decreasing utilization ($T_1$, $T_0$, and $T_2$). Then, CAP-GS tries to partition the group of SRT with utilization of 0.80 using the BFD heuristic. As the group utilization is greater than the processor capacity (0.75), the group is split and the task $T_8$ is removed forming a new group. The group of SRT tasks now has a utilization of 0.7 and it is assigned to core 1. Then, CAP-GS assigns the other group composed of two SRT and one BE tasks to core 3. Finally, the removed task $T_8$ is assigned to core 3. Note that with CAP-GS, the core 0 concentrates HRT tasks, while the inter-core communication caused by color sharing is minimized – only the task $T_8$ that shares the color 8 with tasks $T_3$, $T_5$, and $T_6$ may generate cache line invalidations, because it was assigned to a different core.

Partitioning the same task set with CAP is not possible, because the task group formed by the tasks $T_3$, $T_5$, $T_6$, and $T_8$ has a utilization

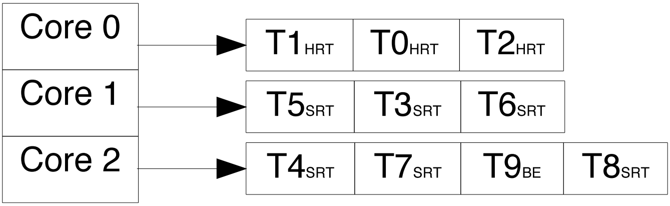


**Figure 83: The partition of the task set in Table 15 with the CAP-GS algorithm.**

of 0.80. As CAP employs a restriction of utilization, in which a group utilization must not be greater than the processor capacity, the task set is not partitioned under CAP.

The original BFD heuristic, in contrast, does not create group of tasks nor differentiate HRT, SRT, and BE tasks. It first order the tasks in decreasing order of utilizations. The order is $T_5$, $T_1$, $T_4$, $T_0$, $T_3$, $T_2$, $T_6$, $T_7$, $T_8$, and $T_9$. Then, the BF bin packing heuristic is applied to partition the task set. Figure 84 shows the task set partitioned by BFD. Tasks $T_5$, $T_1$, and $T_9$ are assigned to core 0, tasks $T_4$, $T_0$, and $T_3$ to core 1, and tasks $T_2$, $T_6$, $T_7$, and $T_8$ to core 2. Note that the color usage by tasks may be a problem in the BFD task assignment. For instance, the group of tasks that shares the color 7 ($T_3$, $T_5$, $T_6$, and $T_8$) are assigned to different cores. As shown in the previous chapter, this may result in deadline losses due to the delay caused by the false sharing of shared cache lines.

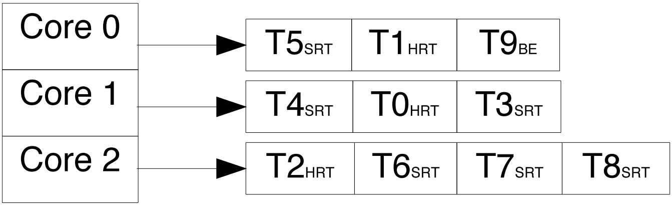


**Figure 84: The partition of the same task set as in Figure 83 with the original BFD heuristic.**

The next section describes the dynamic phase of our scheduling mechanism.

### 8.2.3 Dynamic Color-Aware Scheduling

The dynamic phase of the scheduling algorithm consists of measuring hardware events related to cache coherence activities and using the collected information to perform a scheduling decision. The main idea is to prevent the execution of BE tasks in order to avoid the delay caused by the access of data mapped to shared cache lines (true or false sharing).

We start presenting an analysis of hardware events that can be used by our scheduler in Section 8.2.3.1, then we show the design and implementation of the dynamic color-aware scheduling in Section 8.2.3.2.

#### 8.2.3.1 Analysis of Hardware Events

The dynamic color-aware scheduler uses HPCs to detect when a BE task should give the processor to another task. In this section, we present an analysis of the main HPCs events related to the cache hierarchy. Table 16 presents a list of selected hardware events related to data caches, cache line state, and cache snooping.

**Experiment description.** We used the HPC collection infrastructure implemented in EPOS to measure the selected events. We designed a HPC test application to force shared data access. Figure 85 shows part of the HPC test application. The *main* function creates an array in the partition 1 (*COLOR_1* - line 5) and two threads that execute *func0* and *func1* (lines 11 and 12), respectively. Note that the threads' stacks are allocated in an individual or a same color (lines 11 and 12). Consequently, local variables are also allocated in a separated partition if an individual color is used. Both threads can either access the array created by the *main* (line 22) or create their own arrays in different memory partitions (line 25), depending on the *sharing* boolean variable. We set *sharing* to true when we want the threads to share the array and to false when we want them to allocate the arrays from different memory partitions. After creating the arrays, both threads read and write from/to the array randomly (lines 29 to 37). Moreover, we also set the size of each array (SIZE variable) to different values (128 and 512 KB). We set the PMU sampling period to 1 s (*i.e.,* read current HPCs values at every second) and we monitor the threads for

**Table 16: Selected hardware events for the Sandy Bridge Intel microarchitecture (i7-2600 processor).**

| Event Name | Description |
|---|---|
| HITM | Counts demand loads that hit a cache line in the cache of another core and the cache line has been written to by that other core |
| HIT remote | Counts demand loads that hit a cache line in a cache of another core and the cache line has not been modified |
| Local HIT and remote MISS | Counts demand loads that hit the LLC and are assumed to be present also in a cache of another core but the cache line was already evicted from there |
| L2_LINES_IN.I | Counts the number of L2 cache lines in the invalid (I) state filling L2 |
| L2_LINES_IN.S | Counts the number of L2 cache lines in the shared (S) state filling L2 |

60 s. After 60 s, the main function sets the boolean variable *run* to false (line 3) and both threads stop executing (line 30). We use the P-RM scheduler, and assign thread 0 to core 1 and thread 1 to core 2. We configure the PMU to collect the selected events in all of the eight available processors (Intel i7-2600). We repeat the application for 100 times and extract the average number of measured events from these executions.

Figure 86 shows the collected average values for the HITM event. On the x-axis we vary the CPU, and on the y-axis we show the measured event number in logarithm scale. The bars represent the measured events for the two application cases: when threads allocate their arrays from different colors and when they access the same array. Error bars (almost imperceptible) represent the standard deviation. The CPU 1 is the processor where thread 0 is executed, CPU 2 is the processor where thread 1 is executed, and the other CPUs (0, 3, 4, 5, 6, and 7) execute EPOS idle threads. We plot the collected values only for CPU 7, because the other idle CPUs had similar behavior.

```
#define WRITE_RATIO 2
volatile bool run;
void alarm_handler() { run = false; }
int main() {
    unsigned int * array = new (COLOR_1) unsigned int[SIZE];
    run = true;
    //an alarm of 60 seconds; when it fires , set run to false
    Function_Handler handler(&alarm_handler);
    Alarm alarm(60000000, &handler, 1);
    thread0 = new (COLOR_1) Thread(&func0); //when sharing, both threads are created using the same color
    thread1 = new (COLOR_2) Thread(&func1);
    int status0 = thread0->join();
    int status1 = thread1->join();
     ...
}
int func0() {
    volatile unsigned int * array0;
    Pseudo_Random * rand;
    int sum0 = 0;
    if(sharing) {
        array0 = array;
        rand = new  (COLOR_1) Pseudo_Random();
    } else {
        array0 = new (COLOR_2) unsigned int[SIZE];
        rand = new  (COLOR_2) Pseudo_Random();
    }
    rand->seed(clock.now() + 1);
    while(1) {
        if(!run) break;
        for(int i = 0; i < MEMORY_ACCESS; i++) {
            int pos = rand->random() % (SIZE - 1);
            sum0 += array0[pos];
            if((i % WRITE_RATIO) == 0)
                array0[pos] = i + i;
        }
    }
    return sum0;
}
int func1() { } //same code of func0, but with a different color
```

**Figure 85: Part of the source code of the HPC test application.**

Figure 86(a) shows the values for a WSS of 128 KB and Figure 86(b) for a WSS of 512 KB. We can clearly note that when data sharing is disabled, the PMU measures few HITM events, from 0 to 99 in a period of 1 s. Also, when the arrays' size increases, the HITM event decreases. This behavior is expected, because the probability of accessing the same cache line is smaller than when the size is 128 KB. CPU 7, which does not execute any application thread, does not have any HITM event.

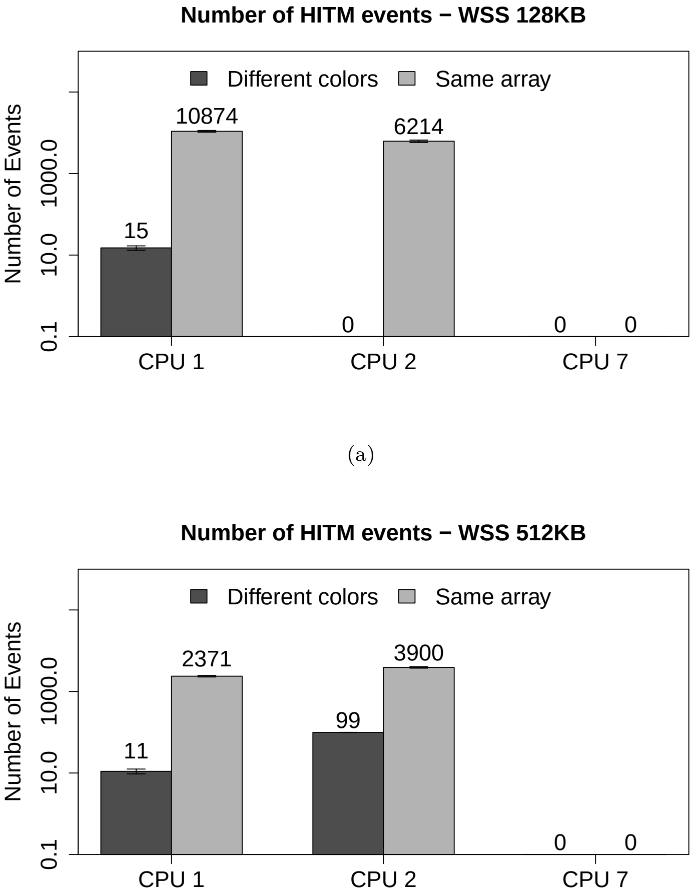
The collected values have shown that the HITM event is a good candidate to be used to detect when threads running on different cores at the same time access shared cache lines. If we normalize the collected events in a period of 10 ms, for instance, we would have less than 0.1 HITM event per period, considering the case of non data sharing. In this case, for example, a threshold of two could be safely used[1] as an approximation value for the scheduler threshold. The scheduler could then take an action based on the measured values and threshold.

Figure 87 shows the measured values for the HIT remote event and the same described scenarios. In Figure 87(a), the arrays' size is configured as 128 KB and in Figure 87(b) as 512 KB. The obtained values show a higher cross-core communication when threads access the same array. When the WSS increases, CPU 1 had more events and CPU 2 had less events. However, if we sum the obtained events of both CPUs and compare the two WSS variation, the obtained values are similar. CPU 7 had obtained few HIT remote events, mainly due to cache hits of internal EPOS data structures accessed by the idle thread.

Figure 88(a) shows the obtained values for the local HIT and remote MISS event and WSS of 128 KB and Figure 88(b) for WSS of 512 KB. The local HIT and remote MISS event shows that both threads are evicting each other's cache lines, because whenever a LLC hit is counted, the same cache line in the other's core cache was already evicted. This does not happen when threads have exclusive colors. The behavior when the threads access the arrays in different partitions/colors is mostly the same for the local HIT and remote MISS and HIT events. For CPU 7, there are some observed events, also caused by the access of internal OS data structures.

The obtained values for HITM, HIT remote, and local HIT and remote MISS events have shown that it is possible to detect when threads access shared cache lines at run-time easily. The cache partitioning

[1] We can only use integer here, because the number of events measured in an interval is obviously always integer. A value of one could also be used, but we set the value for two in this example to reduce the possibility of an OS intervention.

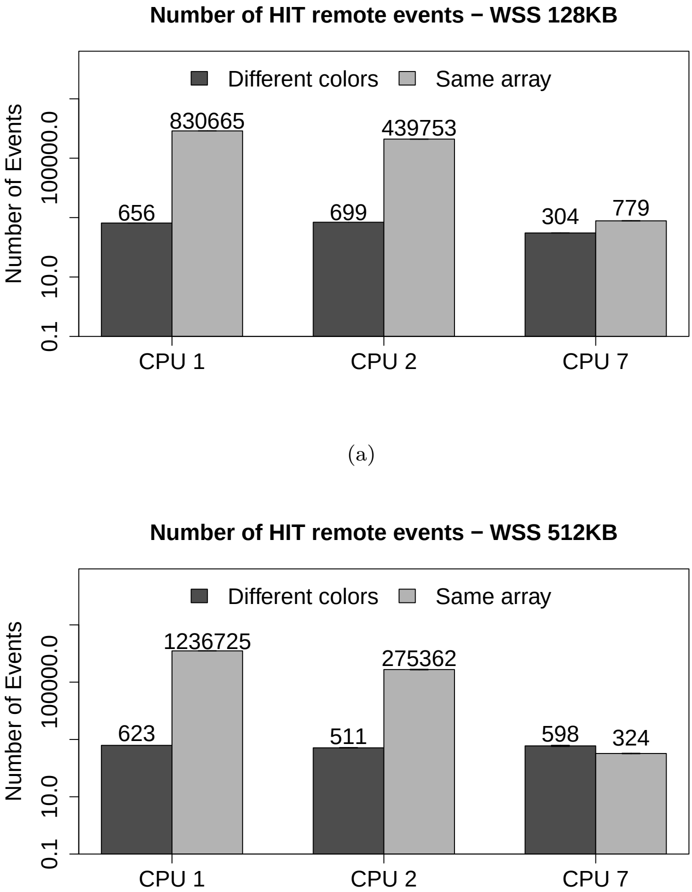


(a)

(b)

**Figure 86: Measured number of HITM events. (a) WSS of 128 KB (b) WSS of 512 KB.**

mechanism isolates task workloads that may interfere with each other, which prevents the access to shared cache lines and improves the correctness of measured hardware events. All the plotted values represent the average number of events in a period of 1 s. We could normalize



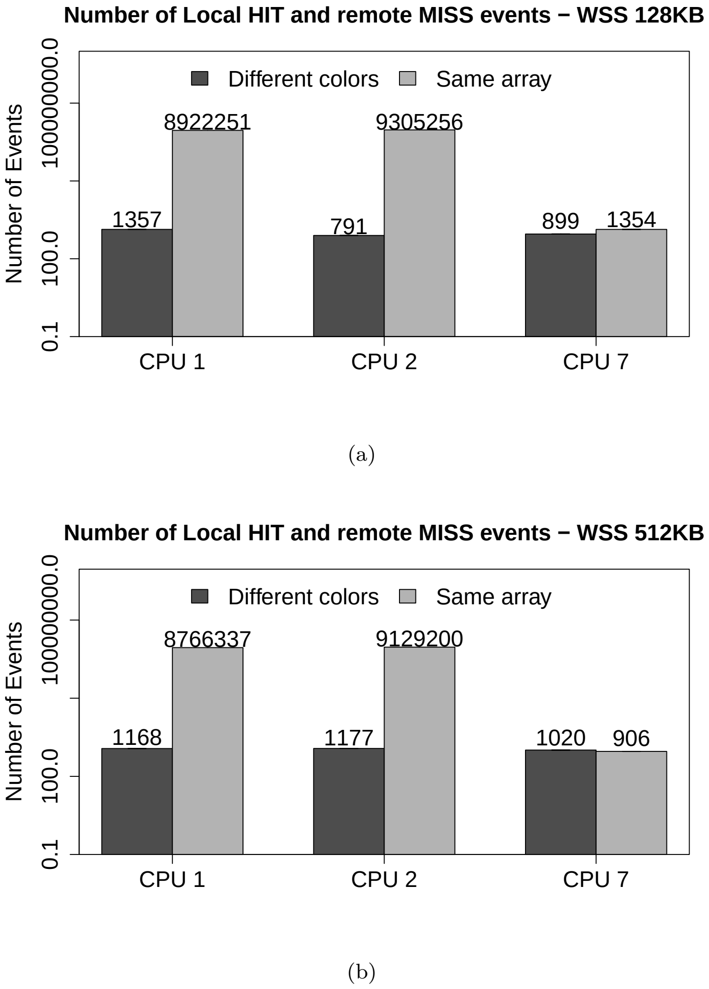
(a)

(b)

**Figure 87: Measured number of HIT remote events. (a) WSS of 128 KB (b) WSS of 512 KB.**

these values to a specific sampling period and use them as thresholds for triggering a scheduling decision. We discuss this idea in the next section.

The next two analyzed events are the number of cache lines in

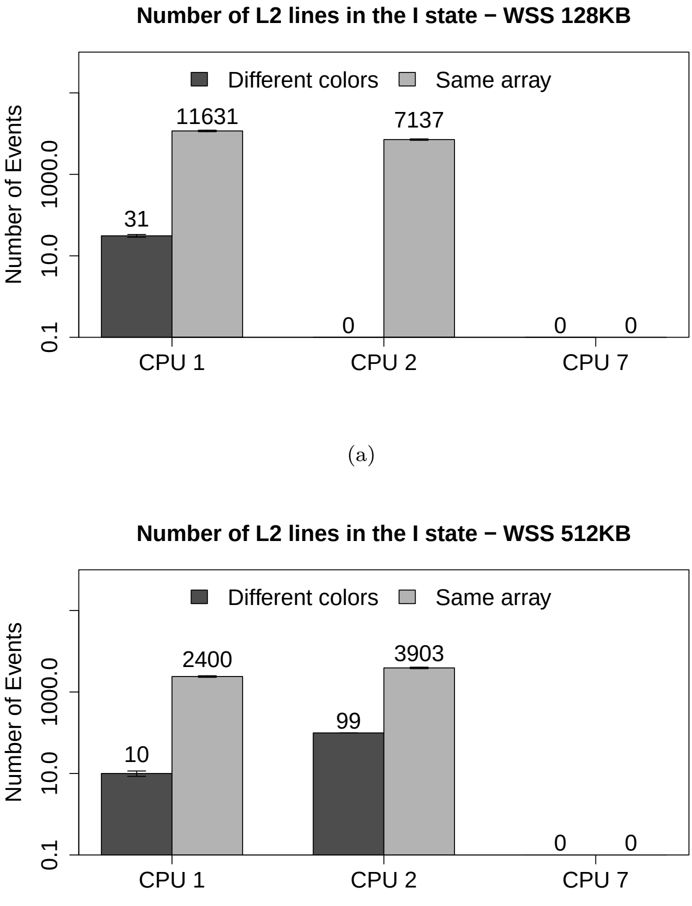


(a)

(b)

**Figure 88: Measured number of local HIT and remote MISS events. (a) WSS of 128 KB (b) WSS of 512 KB.**

the invalid (I) and the shared (S) states. A cache line in the I state indicates that the line does not hold a valid copy of data. The valid data copy can be located in the main memory or in another processor's cache. A cache line in the S state indicates that the cache line is shared by one or more caches and its state is valid. Section 2.1.4 reviews the main cache coherence protocols and their states.



(a)

(b)

**Figure 89: Measured number of L2 cache lines in the I state. (a) WSS of 128 KB (b) WSS of 512 KB.**

Figure 89 shows the number of L2 cache lines in the I state. On the x-axis we vary the CPU, and on the y-axis we show the number of L2 cache lines in logarithm scale. The bars represent the obtained values for the two application cases: when threads allocate their arrays from different colors and when they access the same array. Error bars (almost imperceptible) represent the standard deviation. As in the previous experiments, CPU 1 executes thread 0, CPU 2 executes thread 1, and CPU 7, as well as the other CPUs, executes an EPOS idle thread.

Considering the case when threads access the same array, we can note a reduction in the number of L2 cache lines in the I state when the WSS increases. This is the same behavior observed by the HITM event, and is caused by the same reason: the probability of accessing the same cache line with a bigger WSS is less with 512 KB than with 128 KB. CPU 7 had no L2 cache lines in the I state. When threads use private colors, the observed number of events varies from 0 to 99.

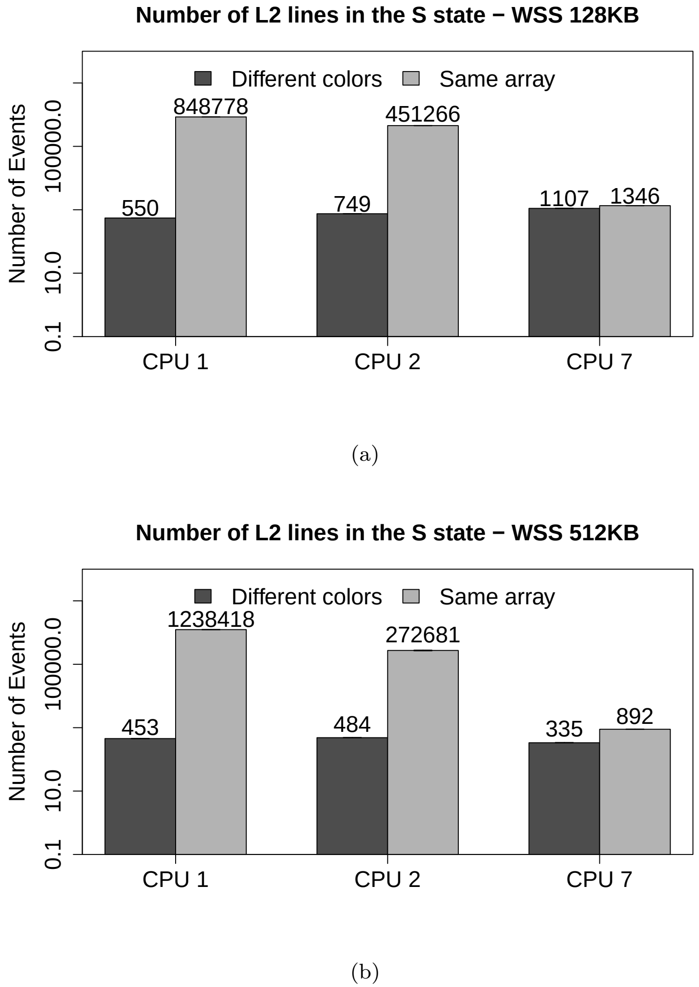
Figure 90(a) shows the number of L2 cache lines in the S state when the arrays' size is 128 KB and Figure 90(b) when the arrays' size is 512 KB. The number of events is similar to the HIT event. In fact, whenever a cache access is a hit, a cache line may be in the E, M, or S states (see Section 2.1.4). As the two threads are accessing the same array, most of the access find cache lines in the S state. CPU 7 had found some L2 cache lines in the S state also due to internal EPOS data structures.

We can conclude that the processor has several viable events that can be used by our dynamic scheduler. In special, the events that measure the number of L2 cache lines in I and S states have similar behavior with HITM and HIT events. In our evaluation presented in Section 8.3, we use the HITM event, because it had presented few values when threads have private partitions and no values when a processor does not execute any application thread.

#### 8.2.3.2 Design and Implementation

This section presents the design and implementation of the dynamic phase of the proposed multicore real-time scheduler.

**Changes in the EPOS scheduling mechanism**. We designed and implemented the dynamic color-aware scheduling algorithm to fit in EPOS. Figure 91 shows the UML class diagram of the new scheduling criterion added to the EPOS scheduling infrastructure (see Section 4.2) and the changes carried out in the `Priority` base class.



(a)

(b)

**Figure 90: Measured number of L2 cache lines in the S state. (a) WSS of 128 KB (b) WSS of 512 KB.**

The CA P-RM criterion is an extension of the `P-RM` criterion, which defines the number of scheduling queues to be equal to the CPUs number (*MAX_CPUS*). Every CPU has its own scheduling queue, as in the `P-EDF` scheduler described in Section 4.2. `P-RM` extends the `RM` class that defines a static criterion (the *dynamic* boolean is set to false). `CA_PRM` class sets the `timed` static boolean attribute to true, enabling the generation of a timer interrupt at every QUANTUM period and the initialization of a timer handler by the `Thread` component (described below). Thus, at every QUANTUM interval, the HPCs values are read and based on these values, a scheduling decision may be taken.

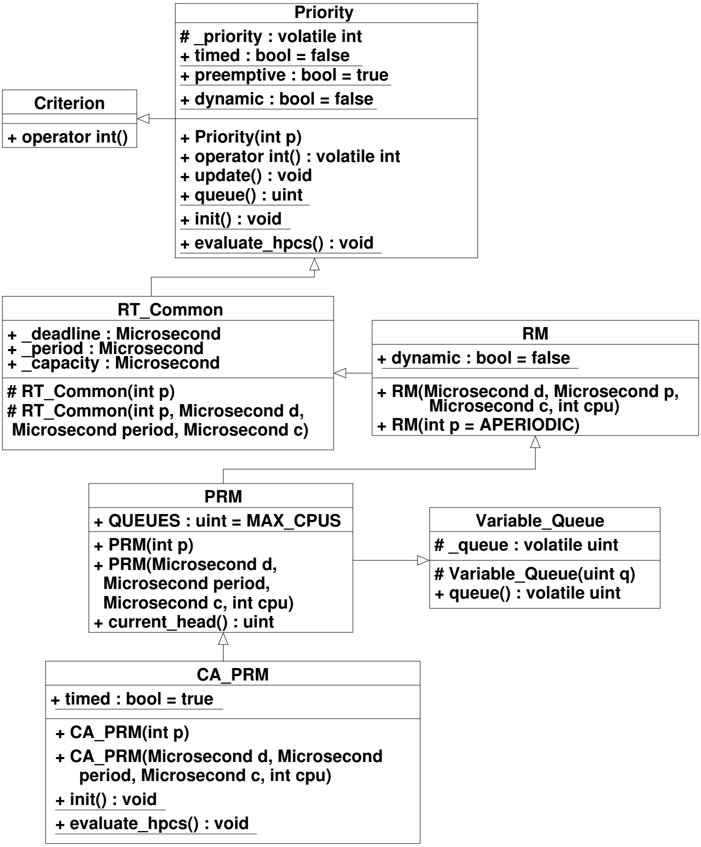


**Figure 91: UML class diagram of the Color-Aware P-RM scheduling policy.**

`CA_PRM` also defines two new static and public methods:

- **init():** this method initializes all PMU events used by the `CA_PRM`. The `Priority` base class has an empty implementation of it;

- **evaluate_hpcs():** this method reads the current HPCs values and informs whether the CPU should be given to other thread or not. This is done by comparing the read HPCs values to thresholds defined at compile-time (details are given below). The `Priority` base class always returns false in this method.

**Changes in the Traits classes.** The `Thread` component implements traditional thread functionalities, such as suspend, resume, sleep, and wake up operations. Section 4.2 has presented a complete overview of the `Thread` component. When a timed-based scheduling criterion is defined, a thread method, `time_slicer`, handles all interrupts generated by the timer at each scheduling quantum. The scheduling quantum as well as the scheduling criterion are defined at compile-time in the thread's trait class, as depicted by Figure 92. The developer defines the criterion by changing the criterion name in line 2. `Scheduling_Criteria` is a namespace in which all scheduling criteria are defined. The developer then chooses the criterion by writing the criterion's name. For example, *typedef Scheduling_Criteria::CA_PRM Criterion* defines the Criterion as `CA_PRM`. The `Thread` component later uses the chosen criterion to access its definition (*typedef Traits<Thread>::Criterion Criterion*). The developer chooses the scheduling quantum by setting the appropriate value to `QUANTUM` (line 4).

```
1 template <> struct Traits<Thread> : public Traits<void> {
2     typedef Scheduling_Criteria::CA_PRM Criterion;
3     static const bool smp = true;
4     static const uint QUANTUM = 10000; //us
5 }
```

**Figure 92: Traits of the Thread class.**

We added a new Trait class for enabling the definition of at least three static values, as demonstrated by Figure 93:

- **LOWER_PRIO_BOUND:** an integer that defines a lower priority bound. The bound is used to compare the priority of a running task when the measured HPCs detect a cache coherence activity (details on this are given below);

- **UPPER_PRIO_BOUND:** an integer that defines an upper priority bound that is also used to compare the priority of a running task;

- **THRESHOLDS:** one (or more) unsigned integer that defines the thresholds for the measured HPCs. There is one threshold for each HPC or for each metric (formed by two or more HPCs). When the collected HPC has a value greater than its threshold, a scheduling decision is performed (details on this are given below).

```
template <> struct Traits< Scheduler<Thread> > : public Traits< void> {
    static const int LOWER_PRIO_BOUND = X; // hypotetical lower priority bound
    static const int UPPER_PRIO_BOUND = Y; // hypotetical upper priority bound
    static const uint THRESHOLD1 = Z; // hypotetical threshold
    //other thresholds can be defined here
}
```

**Figure 93: New Traits class of Scheduler<Thread>.**

**Scheduler timer creation and HPCs initialization.** The scheduler timer used when a criterion is time-based, is created in the *Thread::init* method, depicted in Figure 94. When EPOS initializes, every CPU in the system executes the *Thread::init* method. The CPU id, returned by the *Machine::cpu_id* method, is used to identify the current CPU that is executing the code (line 3). Only one CPU is responsible for creating the scheduler timer (line 5). The scheduler timer is only created when the criterion sets the `timed` static boolean to true (line 4). When `timed` is false, the code block between the lines 3 and 5 is not inserted into the final system image. The period (`QUANTUM`) passed to the timer constructor is defined in the Thread's trait class (see Figure 92).

The *Thread::init* method also calls the `Criterion::init` method (line 8). Every active CPU executes this call to correctly initialize all HPCs used by the chosen criterion. The code of `Criterion::init` method is depicted in Figure 95, when `CA_PRM` is the defined criterion. In this Figure, only one event (HITM) is configured. After the execution of this method, the PMU counts all HITM events and stores the counting on the performance counter number 0 (defined by the `EVENTSEL0`).

**Scheduling decision.** The `Scheduler_Timer` calls the

```
void Thread:init() {
     ....
     if(Machine::cpu_id() == 0) { //only CPU 0 creates the timer
          if(timed)
               _timer = new Scheduler_Timer(QUANTUM, &time_slicer);
     }
     //each CPU calls this method
     Criterion :: init ();
     ....
}
```

**Figure 94: Part of the source code of the Thread init method.**

```
void CA_PRM:init() {
    PMU::config(PMU::EVENTSEL0, PMU::HITM | PMU::ENABLE);
}
```

**Figure 95: Example of the CA_PRM criterion init method.**

`Thread::time_slicer` at every `QUANTUM` microseconds. Figure 96 shows a possible implementation of the `Thread::time_slicer` method. It starts testing if the defined criterion is equal to the `CA_PRM` criterion (line 2). This test is actually a metaprogram and it is omitted to leave the Figure clearer. Thus, the code block between the lines 2 and 5 is only inserted into the final system image when the criterion is defined as `CA_PRM`. In the next step, `Criterion::evaluate_hpcs` is called and returns true if the collected HPC values are greater than the defined thresholds (line 3). Then, `Thread::yield` is called to give the CPU to another thread (including the idle thread).

The code of the `Criterion::evaluate_hpcs` method is depicted in Figure 97. The method starts disabling interrupts (line 2) and reading the hardware event value (line 3). After reading, the hardware event is reseted (line 4). Then, it compares the read hardware event value with the threshold value defined in the trait class (line 5). If the measured event is greater than its threshold, the current running thread on that CPU is retrieved (line 6) and its priority is stored in the *cur_prio* variable (line 7). The next step is to compare the running thread's priority with the lower and upper priority bounds defined in the trait class (see Figure 93). This is done in line 8. If the running thread's priority is in the range [LOWER_PRIO_BOUND, UPPER_PRIO_BOUND], the

```
void Thread::time_slicer() {
    if(Criterion == CA_PRM) {
        if(Criterion::evaluate_hpcs())
            yield();
    } else { //performs a tradition reschedule operation based on the
            quantum interval
      lock();
      reschedule(true);
    }
}
```

**Figure 96: Taking a scheduling decision at every PMU sampling period.**

method returns true (line 9). If not, interrupts are enabled again (line 11) and the method returns false (line 12).

```
 1 bool CA_PRM::evaluate_hpcs() {
 2     CPU::int_disable(); //disables interrupts on this CPU
 3     PMU::Reg64 count = PMU::rdpmc(PMU::PMC0); //reads
           performance counter 0
 4     PMU::reset(PMU::PMC0); //resets the performance counter 0
 5     if(count > Traits< Scheduler<Thread> >::THRESHOLD1) {
 6       Thread * current = running();
 7       int cur_prio = current->priority(); //gets current's priority
 8       if(cur_prio >= Traits< Scheduler<Thread> >::
             LOWER_PRIO_BOUND && cur_prio <= Traits<
             Scheduler<Thread> >::UPPER_PRIO_BOUND)
 9           return true;
10     }
11     CPU::int_enable(); //enables interrupts on this CPU
12     return false;
13 }
```

**Figure 97: Collecting the HPCs values at every sampling period.**

Note that the comparison of HPCs with thresholds can be modified to virtually any value and hardware event. Also, two or more hardware events can be combined to form a metric.

**Summary.** Figure 98 shows an overview of the proposed two-phase multicore real-time scheduler infrastructure. HRT tasks have exclusive colors, while SRT and BE tasks may share colors. The OS also has an exclusive color. HPCs are used to monitor cache coherence activities at run-time and to detect when tasks are interfering with each other, causing contention for shared cache lines. When this happens, the scheduler prevents the execution of tasks that are accessing shared cache lines concurrently.

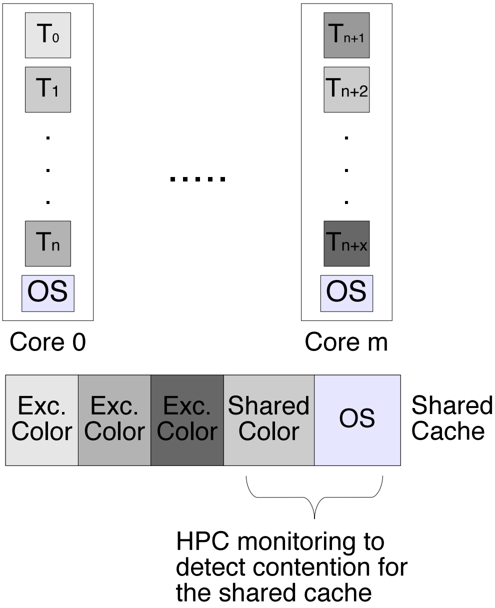


**Figure 98: Overview of the two-phase multicore real-time scheduler.**

Different scenarios can be decomposed from the proposed infrastructure: (i) HRT tasks may also share data (the same scenario as in CAP). In this case, HRT tasks are grouped together and the group utilization must not exceed the processor capacity. Thus, we guarantee that there is no inter-core interference and the HRT deadlines are met (see Chapter 7). (ii) malicious tasks may force the OS to often call the reschedule routine and to switch context, incurring in inter-core interference introduced by these OS activities. In this case, HPCs can detect the contention for the shared cache lines and to prevent the execution of these malicious tasks. (iii) it is possible to use a clustered-based scheduler, instead of a partitioned scheduler. Thus, a task group is assigned to a cluster and HPCs are used to monitor cache coherence activities within the cluster and to dynamically trigger task migrations to prevent cache contention. This approach is interesting mainly for SRT applications, which can have bounded tardiness, due to the better schedulability ratio provided by global schedulers for SRT applications (BASTONI *et al.*, 2010b).

## 8.3 EVALUATION

This section describes the experimental evaluation of the two-phase multicore real-time scheduler. We compare the proposed scheduler to CAP-GS without the dynamic phase (second phase) and BFD bin packing heuristic. We generate several task sets and collect the percentage of missed deadlines, deadline tardiness, and application execution time of these task sets running them on the Intel i7-2600 processor (see Table 12) and varying the scheduling approach (CAP-GS with dynamic optimization, CAP-GS without the second phase, and BFD heuristic).

We start describing the experiment methodology in Section 8.3.1. Section 8.3.2 presents the obtained results on the percentage of missed deadlines. Section 8.3.3 shows the average deadline tardiness, and Section 8.3.4 shows the obtained results for the total application execution time in the three scheduling approaches.

### 8.3.1 Experiment Description

We randomly generated task sets similar to our previous experiments described in Sections 7.3.1 and 6.1, adding a utilization cap for HRT tasks. We selected the tasks periods (all values are in ms) uniformly from {25, 50, 75, 100, 150, 200}. We generated HRT tasks selecting their utilization uniformly between [0.2, 0.35] until $u_{hrt}$ is between [2.1, 2.5] ($2.1 \leq u_{hrt} \leq 2.5$). Then, we set the number of SRT task groups (*i.e.*, groups of tasks that share memory partitions) according to the current $u_{hrt}$ as follows: $number\ of\ groups = ((0.69*8) - u_{hrt})/0.69$. Note that we define the processor capacity as 0.69, which is the limit given by the RM scheduling policy. We use the `CA_PRM` scheduler implemented in EPOS, as described in Section 8.2.3.2. Then, we generated SRT tasks for each group selecting their utilization uniformly between [0.1, 0.3] until $u_{group}$ is between [0.6, 0.7] ($0.6 \leq u_{group} \leq 0.7$). Also, we associated one BE task with each group and uniformly selected the processor for each BE task between {0, 1, …, 7} (the number of available cores in our processor). Each HRT task uses an individual color and each SRT group shares a color with all tasks within that group. We set the priorities of HRT, SRT, and BE tasks according to the priority range scheme described in Section 8.2.3.2.

The BE tasks simulate a server for aperiodic tasks, allocating memory from the same color as the associated SRT group. Figure 99 shows part of the BE task function source code. The function receives an ID as argument (line 3) and uses this ID to access a random class (`rand` variable) and an array (`pollute_buffer`), which were previously allocated by the main function using the generated color number for the task group. Then, the function reads from and writes to an array in a loop (lines 11 to 17), randomly choosing the first position (line 12) and incrementing the position in steps of 64 bytes (the size of a cache line – line 13). After executing the loop, the function gives the CPU to another task by calling the `Thread::yield` method. We set the array size to 512 KB (`POLLUTE_BUFFER_SIZE`) to stimulate BE tasks to interfere with SRT tasks. When all real-time tasks finish executing, the main function deletes the BE tasks, which makes them to stop their executions. We measured the execution time of the `best_effort_task` function loop (lines 11 to 17), when it is executed alone in the system for 100000 times. We obtained a WCET of 46.94 ms, an AVG execution time of 46.29 ms with a standard deviation of 0.49 ms.

HRT and SRT tasks execute a function that allocates its WSS, reading and writing from/to the WSS (an array), in a loop, randomly, as shown in Figure 78 (previous chapter). Each task iterates for 200 times (`ITERATIONS` variable), which give us an expected execution time of 40 s. We considered scenarios with different WSS (`ARRAY_SIZE` variable) similar to (CALANDRINO *et al.*, 2006): 32 KB, 64 KB, 128 KB, and 256 KB. We defined a write ratio of 20% (one write for each four readings) and 33% (one write for each two readings). The number of reads and writes of a real-time task varies depending on the WCET. To adjust the number of repetitions of each task, we used the same methodology as described in Sections 7.3.1 and 6.1. Moreover, we always use the same value for the "repetitions" function argument in the dynamic CAP-GS, CAP-GS, and BFD experiments. We also adjusted the repetitions parameter according to the WSS to account for the intra-task (self-evictions) and intra-core (preemption delay) cache interferences. When

```
#define POLLUTE_BUFFER_SIZE KB_512

int best_effort_task (int id)
{
  int sum = 0;

  rand[id]->seed(clock.now() + id);

  while(1) {

      for(int i = 0; i < ITERATIONS; i++) {
          for(int j = (rand[id]->random() % (
                POLLUTE_BUFFER_SIZE - 1)) % 1000;
                       j < POLLUTE_BUFFER_SIZE; j += 64) {
                pollute_buffer [id][j] = i % 64;
                sum += pollute_buffer[id][j];
          }
      }

      Thread::yield();
  }
  return sum;
}
```

**Figure 99: Part of the best-effort tasks function source code.**

a task evicts its own cache lines, it increases its execution time and eventually misses a deadline. Each task receives a color as parameter and allocates memory for the array using that color.

To evaluate the percentage of missed deadlines, deadline tardiness, and total application execution time, we generated ten task sets using the described methodology. We then partitioned the task sets using the CAP-GS and BFD heuristics. We used the same partitioned task sets to evaluated the dynamic CAP-GS and CAP-GS. The difference is that the `timed` boolean (see Section 8.2.3.2) was set to true for the dynamic CAP-GS. We set the `QUANTUM` period (*i.e.,* the PMU sampling rate) to 10 ms, because the WCET of a BE task is about 47 ms. Thus, a BE task may be interrupt by the scheduler timer up to four times during its execution, considering the case when a BE task is not interrupted by other real-time tasks. Considering the case when it is preempted by the release of a real-time task, the number of interruption caused by the scheduler timer may be greater than four. We discuss the variation on the quantum period in Section 8.4.

We also set the threshold for the HITM hardware event in the dynamic CAP-GS to four, because in our HPC experiments a processor that does not execute threads that share cache lines did not presented any HITM event in a period of 10 ms. Thus, we make sure that when the threshold is reached, it is really generated by threads that share memory partitions and not by the OS. We executed the ten partitioned task sets with the dynamic CAP-GS, CAP-GS, and BFD for 50 times, varying the write ratio and array size. We then extracted the result values for each evaluated metric from these executions (over 133 hours of tests using a real hardware and an RTOS).

### 8.3.2 Percentage of Missed Deadlines

This section presents the obtained percentage of missed deadlines in the three scheduling approaches, varying the experiment parameters as described in the previous section. In EPOS, the `Alarm` component is responsible for releasing a task by calling a `v` operation of its `Semaphore` (see Section 4.2 for a review of the EPOS periodic thread implementation). Since we assume implicit-deadlines ($d_i = p_i$), we counted a missed deadline whenever the `v` operation was issued for the semaphore while its value was greater or equal to zero. In practice this means that a new task's job was released before the previous job had finished.

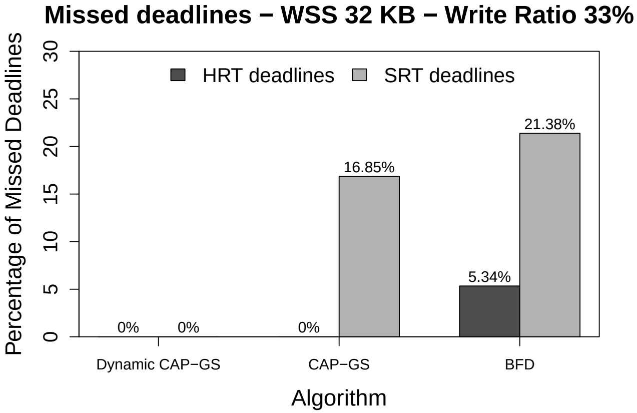
Figure 100 shows the percentage of missed deadlines for a WSS of 32 KB and write ratio of 33% (Figure 100(a)) and write ratio of 20% (Figure 100(b)). On the x-axis we vary the algorithm, and on the y-axis we show the average percentage of missed deadlines for all the ten task sets. The bars represent the percentage of HRT (darker grey) and SRT missed deadlines. *All real-time tasks in all task sets scheduled by the dynamic CAP-GS were able to meet their deadlines.* CAP-GS, instead, provided HRT guarantees and missed about 16% of SRT deadlines. This is due the interference caused by the BE tasks with SRT groups, since some BE tasks were assigned to different cores and may have executed in parallel with SRT tasks on different cores. Furthermore, some SRT groups were split and the contention for the shared cache lines from the tasks within the same group and BE tasks increased as well. This behavior was not observed in the dynamic CAP-GS, because the information collected by the HPCs at every sampling period allowed the scheduler to stop BE tasks before increasing the contention for the shared cache lines.

The BFD heuristic has lost about 5.3% of HRT and 22% of SRT deadlines. As BFD does not group tasks that share colors, the contention for shared cache lines is higher and, consequently, SRT tasks miss more deadlines. HRT tasks are affected by this contention as well, because the private L2 cache of the CPU that is currently running a HRT tasks must respond to snoop requests from other CPUs, thus affecting the time to access the cache and the main memory by HRT tasks. The variation on the write ratio has not affected the behavior of real-time tasks, mainly because the interference caused by the BE tasks that are constantly accessing data allocated from the same color as in SRT groups. Thus, the main source of contention is not the writings of real-time tasks, but the writings performed by BE tasks.

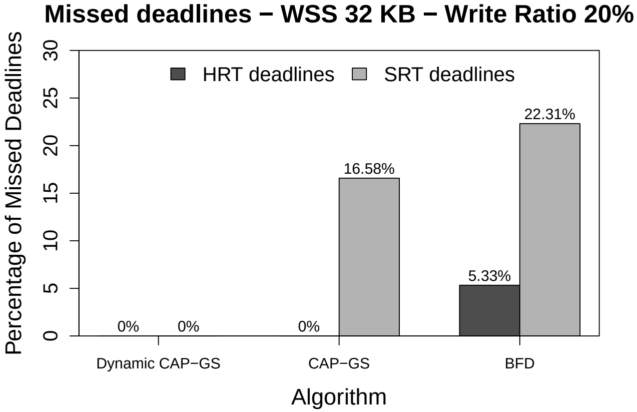
Figure 101(a) shows the percentage of missed deadlines for WSS of 64 KB and write ratio of 33% and Figure 101(b) for WSS of 64 KB and write ratio of 20%. On the x-axis we vary the algorithm, and on the y-axis we show the average percentage of missed deadlines for all the ten task sets. The bars represent the percentage of HRT (darker grey) and SRT missed deadlines. Comparing the obtained results with the previous graphs in Figure 100, we can note that CAP-GS has missed about 3% more SRT deadlines than before. The main reason for that is the higher delay caused by preemptions within groups of tasks assigned to the same cores (intra-core interference). However, CAP-GS still met all HRT deadlines due to the task grouping that reduces the inter-core interference caused by the cache coherence protocol.

The dynamic CAP-GS and BFD had practically the same behavior as before: dynamic CAP-GS did not miss any deadline, while BFD missed about 21% of SRT and 5% of HRT deadlines. Differently from the CAP-GS, the intra-core interference did not increase the percentage of missed deadline in BFD, because tasks that share memory partitions are not grouped together, and thus when a preempted task resumes its execution, it may find that some data is still in the cache (remembering that the L2 cache size in our platform is 128 KB). As observed before, increasing the write ratio did not affect the results (missed SRT deadlines in CAP-GS and BFD and HRT deadlines in BFD are roughly the same).

Figure 102(a) shows the percentage of missed deadlines for WSS of 128 KB and write ratio of 33% and Figure 102(b) for WSS of 128 KB



(a)

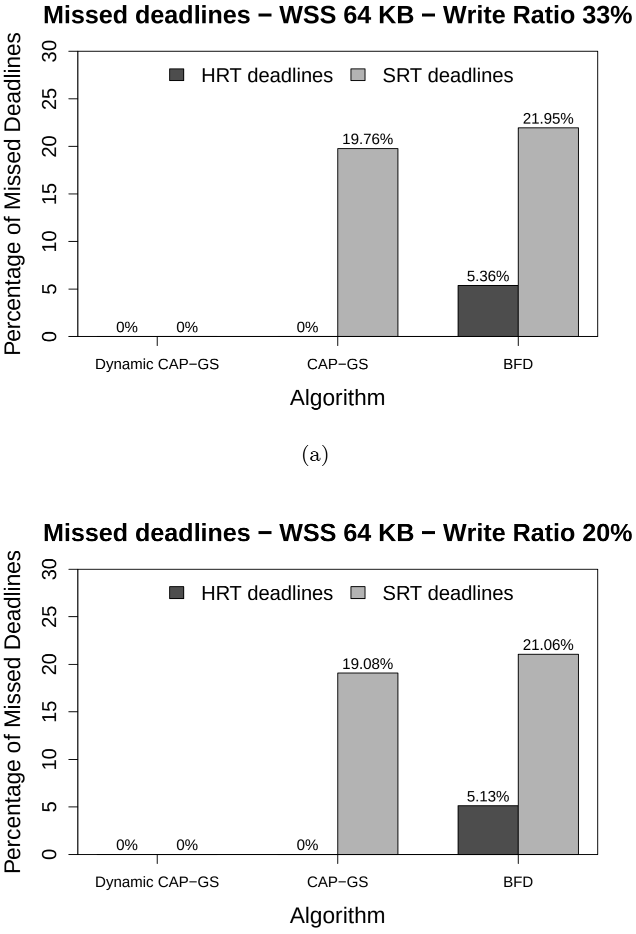


(b)

**Figure 100: Percentage of missed deadlines of a WSS of 32 KB. (a) Write ratio of 33%; (b) Write ratio of 20%.**

and write ratio of 20%. Comparing the obtained results with the previous two experiments, we can note two main differences.

First, we observed a decrease on the missed SRT deadlines in CAP-GS. This is mainly due to a bigger WSS, which decreases the



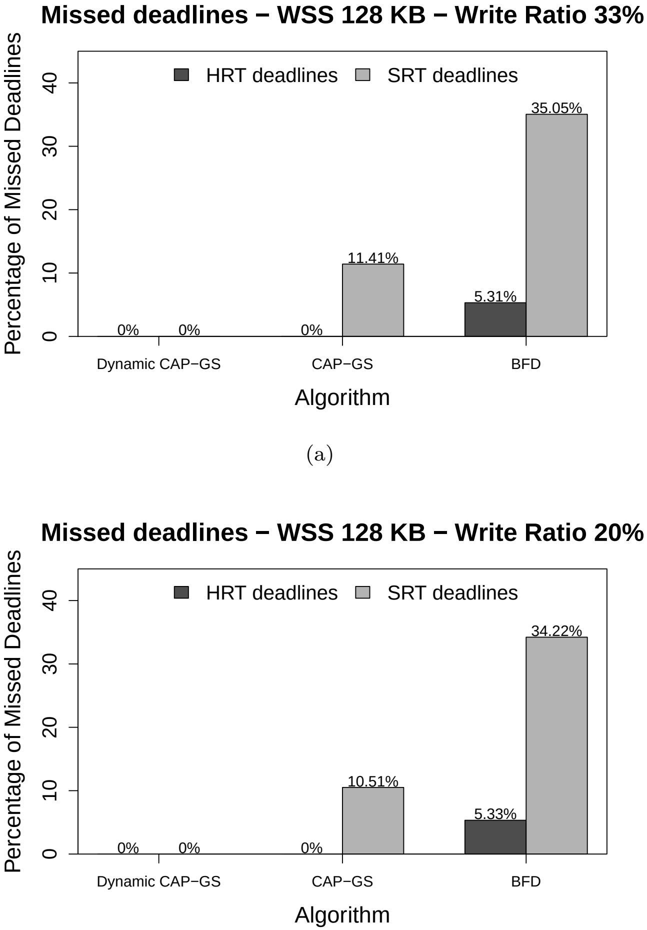
(a)

(b)

**Figure 101: Percentage of missed deadlines of a WSS of 64 KB. (a) Write ratio of 33%; (b) Write ratio of 20%.**

probability of accessing the same cache lines and, consequently, decreases the contention for shared cache lines within SRT task groups and BE tasks caused by the cache coherence protocol.

Second, we observed an increase on the missed SRT deadlines in



(a)

(b)

**Figure 102: Percentage of missed deadlines of a WSS of 128 KB. (a) Write ratio of 33%; (b) Write ratio of 20%.**

BFD. This is also due to the WSS, but a bigger WSS in BFD causes a different behavior. As BFD does not assign the tasks that share colors to the same core, the contention for shared cache lines (inter-core interference) and the preemption delay (intra-core interference) are greater than that with WSS of 32 and 64 KB. Once more, varying the write ratio did not affect the performance, for the same reasons as stated before. Also, the dynamic CAP-GS did not miss any deadline.

It is important to mention again that we adjusted the repetitions parameter of each task and executed the task sets with the same parameters using the three scheduling approaches. Thus, we make sure that the differences on the observed percentage of missed deadline comes from the task assignment performed by the task partitioning heuristics and the avoidance of cache contention by the CAP-GS task partitioning and by the dynamic phase of the proposed two-phase scheduler.

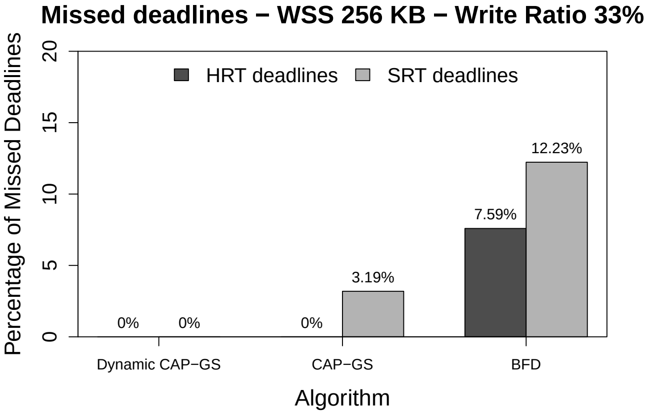
Finally, Figure 103(a) shows the percentage of missed deadlines for WSS of 256 KB and write ratio of 33% and Figure 103(b) for WSS of 256 KB and write ratio of 20%. The observed percentage of missed SRT deadlines in CAP-GS, if compared with the previous graphs, has decreased even more. The reason is the same: the likelihood of accessing the same cache lines in parallel during the task execution is smaller with a bigger WSS.

In BFD, we also observed a decrease on missed SRT deadlines due to the same reason as in CAP-GS. However, we can note an increase of about 2% on the missed HRT deadlines. Although the contention for shared cache lines has decreased, the intra-core interference increases. Consequently, the traffic between the cache levels and main memory is higher than before, which affect the execution of tasks assigned to two hyperthreads of the same core (each core has two logical hardware threads).

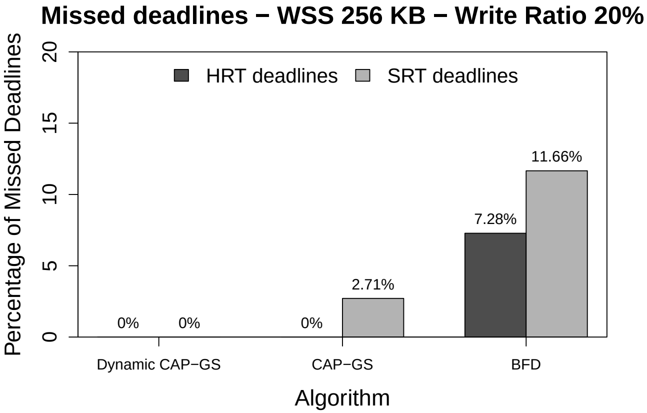
### 8.3.3 Deadline Tardiness

The tardiness of a real-time task is the difference between its completion time and deadline. If a real-time task misses its deadline, its tardiness is positive. A negative deadline tardiness means that the task completes its execution before the deadline, which is the expected behavior for every real-time task, specially those that have HRT constraints.

In this section we present the deadline tardiness obtained by the execution of the ten generated task sets using the three scheduling approaches. We measure the WCET of each task in each task set and calculate the tardiness as follows: *tardiness = observed WCET - dea-*



(a)



(b)

**Figure 103: Percentage of missed deadlines of a WSS of 256 KB. (a) Write ratio of 33%; (b) Write ratio of 20%.**

*dline*. We sum the tardiness of each task, resulting in the tardiness for the whole task set. Then, we calculate the average tardiness using the tardiness obtained in each task set, varying the scheduling approach, write ratio, and WSS.

Figure 104 shows the obtained average deadline tardiness for each scheduling approach. On the x-axis we vary the WSS. On the y-axis we show the obtained tardiness in ms. The bars represent the scheduling approach. Figure 104(a) shows the results for a write ratio of 33% and Figure 104(b) for a write ratio of 20%. The dynamic CAP-GS has presented a negative average tardiness, which was the expected behavior since the previous experiments have shown that it does not miss any deadline. Varying the WSS for CAP-GS did not result in a considerable difference of the tardiness. The tardiness observed in CAP-GS and BFD, instead, varies according to the WSS: as the WSS increases, the tardiness decreases. The worst-case scenario for BFD is when the WSS is 128 KB, which was the same scenario that it has missed the biggest percentage of missed deadlines (see Figure 102).

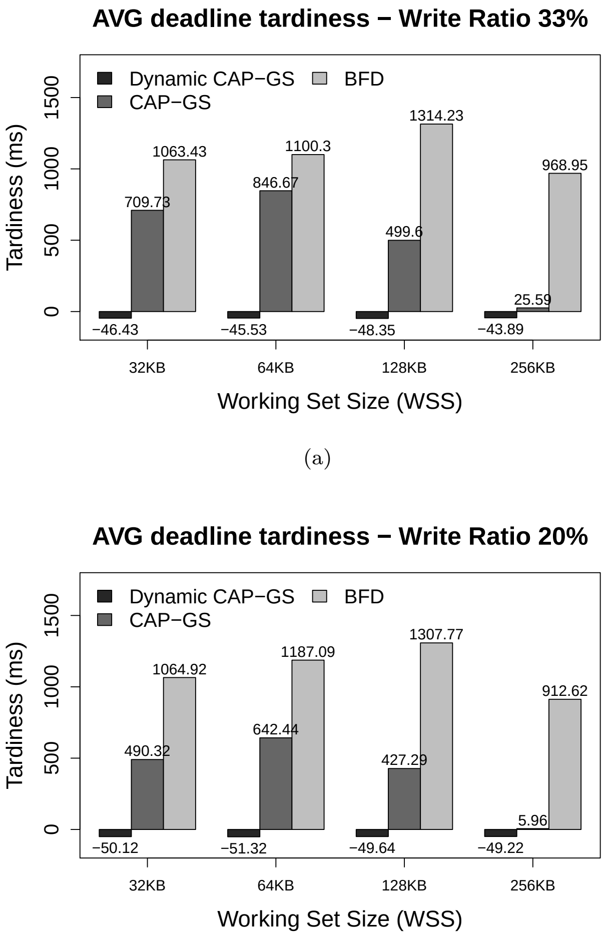
Comparing the tardiness results with the percentage of missed deadlines for CAP-GS, we can note that although the average deadline tardiness has reduced as the WSS increases, the percentage of missed deadline has not presented a great variation. We can conclude that the tasks are still executing above their deadlines, but with smaller WCETs. This is also proved by the results of the next subsection, which show the total application execution time.

Furthermore, the variation on the write ratio has shown a different behavior mainly for the CAP-GS and the dynamic CAP-GS: the observed tardiness has decreased. This is due to the reduction of the inter-core interference caused by the smaller ratio of writes to the shared cache lines. Specifically for the CAP-GS, this reduction did not improve the percentage of missed deadlines though.

### 8.3.4 Total Execution Time

The total application execution time is the time to finish all real-time tasks in a task set. We measured the total execution time for all task sets using the three scheduling approaches. Then, we extracted the average value from from these measurements. As the greatest period in a task set is 200 ms and the real-time tasks iterates for 200 times, the expected execution time is 40 s.

Figure 105(a) shows the total application execution time with a write ratio of 33% and Figure 105(b) with a write ratio of 20%. The application execution time for the dynamic CAP-GS is always 40 s,



(a)

(b)

**Figure 104: Average deadline tardiness. (a) Write ratio of 33%; (b) Write ratio of 20%.**

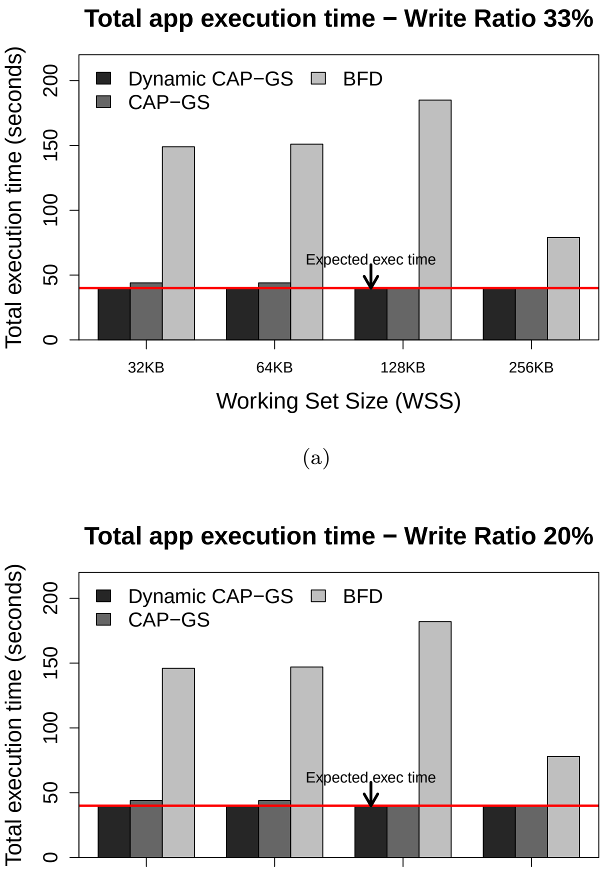
despite the size of the WSS or the write ratio. As the real-time tasks never miss deadlines, all task sets finish in the expected time. CAP-GS, instead, have a total execution time of 44 s when the WSS is 32 and 64 KB, for both write ratios of 33 and 20%. The obtained average tardiness in CAP-GS for WSS of 32 and 64 KB was also the greatest. This explains why for these for scenarios the total execution time exceeds 40 s. For WSS of 128 and 256 KB, CAP-GS finishes in the expected time, although it still misses deadlines.

The BFD always executes more than 40 s. The worst scenario is when the write ratio is 33% and the WSS is 128 KB (185 s), which is the scenario where we observed the greatest tardiness and percentage of missed SRT deadlines. Varying the write ratio did not present a considerable change in the execution time: it decreased about 2 s for BFD and did not change for CAP-GS.

### 8.3.5 Run-Time Overhead of the HPC Analysis

The additional run-time overhead introduced by the dynamic phase of the proposed scheduler is formed by the `Thread::time_slicer` and `Criterion::evaluate_hpcs` methods, which read the hardware event and perform a scheduling decision (Figures 96 and 97), and by the `Timer::handler` method, which handles a timer interrupt and calls the `Thread::time_slicer` method. These activities are performed at every `QUANTUM` period. Thus, they can be modeled as a higher priority task with period equal to the `QUANTUM` and execution time composed of the timer handler, `Thread::time_slicer`, and `Criterion::evaluate_hpcs` methods code.

As the call to the `Thread::yield` is only done when the current running thread is a BE task, the overhead introduced by this method does not have any influence on the execution time of real-time tasks. We measured the execution time of the timer interrupt handler and `Criterion::evaluate_hpcs` executing them for 100000 times. Table 17 shows the obtained WCET, AVG execution time, and standard deviation for the timer handler and `Criterion::evaluate_hpcs` methods (`Thread::time_slicer` only calls `Criterion::evaluate_hpcs`). Hence, in our experiments, the collection and analysis of HPCs can be modeled as a periodic task with period of 10 ms and WCET of 3.432 µs, with a utilization of 0.00343. As both methods access very few variable and registers, the CPMD can be considered as negligible. Thus, the proposed HPC monitoring infrastructure and the dynamic scheduling decision have a small impact on the real-time tasks.



(a)

(b)

**Figure 105: Total execution time of the test application. (a) Write ratio of 33%; (b) Write ratio of 20%.**

## 8.4 DISCUSSION

We now discuss our main observations on the experiments:

**Table 17: WCET, AVG execution time, and standard deviation (STD) of the timer handler and `Thread::collect_hpcs` methods. All values are in µs.**

| Method | WCET | AVG | STD |
|---|---|---|---|
| Timer::handler | 0.273 | 0.065 | 0.031 |
| Criterion::evaluate_hpcs | 3.159 | 1.025 | 0.652 |
| *Total* | *3.432* | *1.115* | *0.708* |

- **HRT constraints:** our two-phase dynamic scheduler combines the CAP-GS task partitioning with run-time scheduling decision based on the collected information from the HPCs. Thus, it is possible to decrease the inter-core interference and prevent BE tasks to interfere with SRT tasks. In our experiments, the two-phase dynamic scheduler was able to provide HRT and SRT guarantees for all tasks in all task sets. CAP-GS without the dynamic phase also provided HRT guarantees, because it decreases the inter-core interference that affects HRT tasks, which was observed in the BFD approach. *We can conclude that the assignment of exclusive colors to HRT tasks works only if followed by the color-aware task partitioning heuristic.*

- **Deadline tardiness:** the average deadline tardiness observed in the experiments gave us good hints about the behavior of the three compared scheduling approaches: (i) the dynamic scheduler has a negative tardiness, which means that all tasks completes their execution before the respective deadlines; (ii) although the tardiness for CAP-GS decreases as the WSS increases, the percentage of missed deadlines did not have a considerable change. This means that the SRT tasks are executing faster, but they are still completing their executions after the respective deadlines; and (iii) BFD had an average deadline tardiness of up to 153 times worse than the tardiness in CAP-GS (for WSS of 256 KB and write ratio of 20%). The worse result in terms of missed deadlines for BFD was with WSS of 128 KB and write ratio of 33%, mainly due to a bigger WSS which leads to a higher contention rate for shared cache lines and preemption delays. With 256 KB, the inter-core interference decreases, so as the percentage of missed deadlines.

- **QUANTUM period:** we set the QUANTUM period to 10 ms.

We choose this period, because it is within the execution time of the BE tasks (around 47 ms). Thus, we make sure that a BE task is interrupted before causing a considerable interference with SRT tasks. The QUANTUM period is extremely important, because it dictates when the HPCs are read and, consequently, when BE tasks are removed from the CPU. Thus, if one wants to set a PMU sampling period, we suggest to use a period that is within the execution time of the tasks that may be interrupted.

Some PMUs, however, generate an interrupt whenever an event reaches a pre-determined number. This feature simplifies the design of our dynamic scheduler, because a periodic interrupt is no longer required. Thus, the PMU could be configured to directly call `Thread::time_slicer` whenever needed, reducing the interference into the system and improving the responsiveness of scheduling decisions (*i.e.,* a scheduling decision in this case would not be limited by a periodic interrupt).

- **HPCs overhead:** the run-time overhead introduced by the dynamic phase of the proposed scheduler is formed by the code in the timer interrupt handler, and by the `Criterion::collect_hpcs` and `Thread::time_slicer` methods, which read the hardware event and perform a scheduling decision (Figure 96). We measured the run-time overhead of these three methods (`Thread::time_slicer` only performs a call to `Criterion::collect_hpcs`), and the observed WCET was 3.432 µs. As the call to the `Thread::yield` method is only performed when the current running thread is a BE task, the run-time overhead for the real-time tasks must not take the method execution time into account.

  In a real-time system, the `Criterion::evaluate_hpcs` and `Thread::time_slicer` methods could be modeled as a periodic task, with a period of 10 ms and a WCET of 3.432 µs, resulting in a utilization of 0.00342. This low overhead has two main factors: (i) the EPOS timer handler function is fast and access few data; and (ii) the EPOS PMU mediator allows the access to HPCs to be performed without any software layer, which also increases the speed. Furthermore, the WCET of this task must be also inflated by the source of overheads described in Section 2.5 (Equation 2).

- **HPCs thresholds:** in our experiments we use the HITM event with a threshold of four. With the created PMU sampling and scheduling infrastructure, it is possible to use any available event on the processor and any threshold. Also, it is possible to combine two or more events and form a metric to evaluate a specific application behavior.

- **Write ratio:** varying the write ratio almost did not change the percentage of missed deadlines in CAP-GS and BFD. Considering the tardiness, we could note a reduction on the WCET by SRT tasks, mainly in CAP-GS, but this reduction did not decrease the percentage of missed deadlines.

- **Working set size:** varying the WSS affects mainly the SRT tasks. We could note an increase in the missed deadlines from 32 KB to 128 KB in the BFD and from 32 KB to 64 KB in the CAP-GS. For a WSS of 256 KB, the intra-task interference (self-evictions) is higher than with the other WSS, which decreases the inter-core interference and the missed deadlines.

- **Responsiveness of BE tasks:** the BE tasks are the greatest source of inter-core interference. We choose to prioritize the SRT guarantees over the BE responsiveness. Allowing BE tasks to execute more than the sampling period, certainly improves their response times, but also increases the inter-core interference and deadline miss of HRT and SRT tasks.

- **Limitations:** our two-phase scheduling approach needs to be aware of cache coherence activities. We capture these activities by the use of hardware events available on modern multicore processors. Thus, a limitation of our approach is that the target multicore platform must offer support for measuring cache coherence activities at run-time.

  Another limitation is regarding the number of colors. HRT tasks have private colors, consequently, the number of colors available on the target processor must be enough to accommodate all HRT tasks as well as the colors to support possible SRT and BE tasks.

- **Summary of real-time extensions:** to support the collection of hardware events at run-time and scheduling decisions based on the collected information, we made a series of changes in the real-time support on EPOS, presented in Chapter 4:

(i) we added a new criterion class (`CA_PRM` – see Figure 91) that enables the generation of periodic PMU sampling interrupts (through the `timed` boolean) and defines two new methods (init and evaluate_hpcs);

(ii) we added a new trait class (Figure 93) that defines the lower and upper priority bounds and thresholds used to take a scheduling decision;

(iii) we changed the `Thread::init` method to call the `Criterion::init` method which configures the HPCs;

and (iv) we changed the `Thread::time_slicer` method to support the reading of HPCs at run-time. At every interrupt (the interrupt period is given by the `QUANTUM` definition), the `Thread::time_slicer` method calls the `Criterion::evaluate_hpcs` which reads the HPC and compares the value with the threshold defined in the trait class. If the value is greater than the threshold and the current running thread is within the lower and upper priorities bounds (also defined in the trait class), `Criterion::evaluate_hpcs` returns true and `Thread::time_slicer` removes the current thread from the CPU. In our case, we set the lower and upper bounds to the BE tasks priorities.

# 9 CONCLUSION

The main objectives of this research were: (i) to investigate how real-time tasks executing on a multicore platform and sharing cache lines due to true or false sharing could be efficiently supported by an RTOS; (ii) how an RTOS designed for deeply embedded systems affects cache partitioning mechanisms due to the use of its internal data structures; (iii) how different real-time scheduling algorithms affect the performance of cache partitioning mechanisms; and (iv) how the run-time overhead of an RTOS designed for deeply embedded systems impacts the schedulability of global, partitioned, and clustered real-time scheduling approaches and to investigate the differences in terms of run-time overhead between an RTOS and a GPOS patched with real-time extensions.

To investigate these issues, we have designed and implemented a multicore real-time infrastructure on top of EPOS. The infrastructure is composed of partitioned, clustered, and global scheduling variants of the EDF, RM, DM, and LLF real-time scheduling policies, a cache partitioning mechanism based on page coloring, a HPC monitoring API specifically designed for embedded (real-time) systems, a color-aware task partitioning (CAP) algorithm, and a two-phases color-aware multicore real-time scheduling algorithm. To the best of our knowledge, EPOS is the first RTOS designed from scratch to support all multicore real-time scheduling variants (partitioned, clustered, and global) of EDF, RM, DM, and LLF scheduling policies, and to provide a scheduling mechanism to deal with the contention for shared cache lines caused by the cache coherence protocol at run-time, combining task partitioning with information from the HPCs. In the following, we summarize our results (Section 9.1), present our closing remarks (Section 9.2), and discuss future work (Section 9.3).

## 9.1 SUMMARY OF CONTRIBUTIONS

Our research makes novel contributions in the following areas: (i) design and implementation of component-based RTOSes; (ii) cache partitioning evaluation in an RTOS; (iii) real-time task partitioning; and (iv) cache-aware multicore real-time scheduling. Here, we briefly recapitulate the key contributions from Chapter 4 to Chapter 8.

### 9.1.1 Real-Time Support on EPOS

The design and implementation of the multicore real-time support on EPOS is described in details in Chapter 4. The infrastructure of multicore real-time scheduling, HPCs, and page coloring created in this work closes the existing gap in the real-time system community, in which scheduling algorithms and cache partitioning strategies were evaluated using real-time patches applied to GPOSes. We believe that EPOS can be used to conduct research for multicore real-time related areas due to higher predictability and smaller overhead compared to real-time patches for Linux.

In the design of the multicore real-time scheduling of EPOS, we detached the scheduling policy from its mechanism by using static metaprogramming and creating several `Scheduling_Queue` specialization depending on the defined scheduling criterion (see Figure 44). For each scheduler variant, a specialized scheduling list is chosen at compile time. For example, a partitioned scheduler uses individual scheduling lists for each processor, while global schedulers have only one global scheduling list. Clustered schedulers are a combination of partitioned and global approaches: each cluster is a partition and each partition has a global scheduling list. This design allows the addition of a scheduler variant of virtually any scheduling policy. This structure of scheduling is unique and original.

Furthermore, we have performed several modifications on the EPOS structure: distribution of the alarm handler in all available processors; extension of the IC hardware mediator to support IPI messages; creation of the new dynamic semaphore handler for the `Periodic_Thread` class; and new scheduling policies (LLF and SJF).

Additionally, the page coloring mechanism designed and implemented in EPOS allows the assignment of individual cache partitions to internal EPOS data structures, which is not possible in GPOSes with real-time extensions due to their complexity. Thus, it is possible to achieve higher predictability by isolating the OS from the application. We also proposed two mechanisms for colored memory allocation: the user-centric approach in which the developer inserts code annotations to define the partition from which the data should be allocated, and the OS-centric approach in which EPOS chooses the partition based on a task ID.

### 9.1.2 Run-Time Overhead Evaluation

We measured the run-time overhead of EPOS and compare it with the run-time overhead in LITMUS$^{\mathrm{RT}}$. The evaluation is described in details in Chapter 5. The main contribution of the run-time overhead evaluation was to show that the RTOS run-time overhead, when incorporated into G-EDF, C-EDF, and P-EDF schedulability tests, can provide HRT guarantees close to the theoretical schedulability tests.

We generated several task sets, with different utilization and period distributions, and used these task sets to perform the run-time overhead evaluation. The main observations from these experiments were the following:

- LITMUS$^{\mathrm{RT}}$ achieved a schedulability ratio comparable with EPOS in all but non-uniform workloads. The main reason for this behavior is the pessimism of the available schedulability tests. Non-uniform task sets have few tasks with higher utilizations (between 0.5 and 0.9), which strongly affects the HRT guarantees provided by the G-EDF schedulability tests and the bin packing partitioning heuristics. For light uniform utilization and short periods, C-EDF with the overhead in EPOS was better than P-EDF with LITMUS$^{\mathrm{RT}}$ overhead due to smaller scheduling overhead (see Figure 59). Also, our results corroborate previous studies (BASTONI *et al.*, 2010b) in the sense that more partitioned approaches are preferable for HRT systems.

- P-EDF is always better than G-EDF and C-EDF for HRT in all workloads, except in the scenario of heavy utilization tasks. For task sets consisting of only heavy utilization tasks, P-EDF, C-EDF, and G-EDF had the same schedulability ratio. This is due to the bin packing problem limitation, in which the partitioning heuristics can only partition task sets with a task number equal to the number of processors, and due to the G-EDF schedulability bounds, which usually have a relation between the number of processor and the largest utilization or density (BERTOGNA; BARUAH, 2011). For task sets with few heavy tasks, as in the case of light bimodal utilization distribution, G-EDF presented the biggest difference in terms of task set schedulability ratio in comparison to P-EDF. This reinforces the need for better G-EDF schedulability tests for HRT systems with heavy utilization tasks (BRANDENBURG; ANDERSON, 2009).

- Differences in task period length did not affect the schedulability tests for the theoretical G-EDF, C-EDF, and P-EDF schedulers. On the other hand, it has a significant impact on the run-time overhead. In short period distributions, the proportion between the period length and the overhead is higher than in long period distributions, which decreases the schedulability ratio.

- The CPMD has a considerable impact on the schedulability ratio due to its high values, specially when the WSS of tasks are greater than 128 KB. As the WSS increases, P-EDF, C-EDF, and G-EDF tend to be equal due to higher CPMD. For uniform and bimodal light and uniform moderate utilizations distributions, which have more tasks than the other generated task sets, the difference between EPOS and LITMUS$^{RT}$ is higher, which makes evident the benefits of having an RTOS designed from scratch.

### 9.1.3 Cache Partitioning Evaluation

The main objectives of the cache partitioning evaluation, described in Chapter 6, were to show how the scheduling algorithm affects the gains obtained by cache partitioning and to analyze whether the RTOS impacts the cache partitioning behavior or not. To achieve these objectives, we evaluated the performance of the page coloring mechanism in EPOS using P-EDF, C-EDF and G-EDF schedulers in an eight-core processor, with shared L3 cache. We generated several task sets and executed the tasks using three cache partitioning strategies: (i) assigning different cache partitions to tasks and OS; (ii) assigning different cache partitions to tasks and making EPOS to allocate data from a non-colored heap; and (iii) assigning the same partition to tasks and giving an individual partition to EPOS. The main observations from the experiments were:

- For P-EDF and C-EDF, cache partitioning provided HRT bounds when tasks had WSS of up to 128 KB. For G-EDF, the achieved HRT bound was when tasks had WSS of up to 64 KB. To support larger WSSs, cache partitioning could be used together with hardware techniques, such as cache locking. Even when tasks miss their deadlines with cache partitioning (128 KB for G-EDF, and 256 KB for the three schedulers), the advantage is predictability of cache accesses. It is possible to apply a data reuse method (JIANG *et al.*, 2010) and provide HRT guarantees during the theoretical schedulability analyses.

- Cache partitioning was more efficient in global approaches (G-EDF and C-EDF) for WSSs up to 64 KB, by helping to prevent inter-core communication through the cache coherence protocol: all data is able to fit in L2-cache and the invalidations in the L3-cache are reduced, mainly for G-EDF. For 128 KB, page coloring was more efficient in C-EDF, because tasks can migrate inside the cluster and still access the same cache lines in L2, reducing the number of misses. For 256 KB page coloring was more efficient in P-EDF, because cache partitioning reduces the contention for cache spaces when tasks are running on two logical cores at the same time.
- In an underloaded system, global approaches handle the tasks more efficiently than P-EDF, because tasks can migrate as soon as a core becomes available. Inter-core communication, caused by task migrations, has a considerable impact on HRT tasks (18% of missed deadlines), as shown in G-EDF with WSS of 128 KB.
- When tasks execute in parallel on different cores and share cache lines (true or false sharing), they will access the same cache lines, causing invalidations handled by a bus snooping protocol. Cache partitioning does not solve the problem, but it helps to keep all data organized in memory. Providing a separate set of colors to shared data may improve the overall performance (CHEN *et al.*, 2009).
- Assignment different cache partitions to EPOS did not affect the performance of the three schedulers, because ISRs and scheduling operations in EPOS have a small footprint and use few bytes.

### 9.1.4 Static Color-Aware Task Partitioning

The cache partitioning results have indicated that when real-time tasks share cache partitions, they may experience deadline losses mainly due to inter-core interferences: the observed WCET increased up to 15 times. The consequent variations on the execution time of real-time tasks made them miss up to 97% of their deadlines. Furthermore, the partitioned scheduler has achieved the less percentage of missed deadline compared to the global and clustered schedulers. Hence, it is desirable to avoid that two or more tasks that share same partition(s) (*i.e.,* same color(s) in case of a page coloring cache partitioning mechanism) access data at the same time. In practice this means that two or more tasks that use a same color should not be scheduled on different cores at the same time. By analyzing the results and behavior of the three schedulers in Chapter 6, we proposed a color-aware task partitioning algorithm to avoid inter-core interference when tasks share cache partitions.

The color-aware task partitioning (CAP) is described in details in Chapter 7. The main idea of CAP is to group tasks that share the same colors and to assign the group to the same processor. To evaluate the CAP algorithm, we generated several task sets and partitioned these task sets using CAP and WFD bin packing heuristic. Then, we executed the partitioned task sets in a modern eight-core processor using EPOS, varying the WSS of each task (32, 64, 128, and 256 KB) and the write ratio (20 and 33%) to the WSS, and extracted the number of missed deadlines for each task. The main observations from these experiments were:

- CAP prevented tasks from accessing shared cache lines and provided HRT guarantees for all tasks in all executed task sets. The WFD heuristic, in contrast, missed up to 78% of deadlines.

- With WSS of 128 KB, the tasks have the greatest deadline miss ratio due to the more contention for shared cache lines and preemption delay. With 256 KB, the preemption delay is higher due to L2-cache misses. For 32 KB, mostly of the tasks data fits into the L2-cache and the deadline miss ratio is smaller than in the other WSSs.

- By changing the write ratio, we stimulate the cache coherence protocol to invalidate more or less cache lines. For all WSS variation, the execution with the write ratio of 20% has missed less deadlines, as expected.

### 9.1.5 Dynamic Color-Aware Scheduling Algorithm

The system model used in CAP has two main drawbacks: (i) all tasks in a task set have only HRT constraints. This scenario may not be true for every real-time application. In fact, emerging real-time workloads, such as multimedia and entertainment and some business computing applications, have highly heterogenous timing requirements and consist of HRT, SRT, and best-effort (BE) tasks; and (ii) the number of colors in a processor may be smaller than the number of real-time and BE tasks. Consequently, real-time and BE tasks may eventually share colors or memory partitions and the utilization of these tasks that share memory partitions may be greater than the processor capacity (restriction of utilizations in the CAP system model).

We have shown in Chapter 7 that when tasks share colors, the contention for shared cache lines caused by the cache coherence protocol may lead to deadline losses. Thus, color sharing and the execution of tasks that use the same color in parallel on different cores could be avoided whenever possible.

To deal with the two main drawbacks in the CAP system model, we proposed a two-phases color-aware multicore real-time scheduler, which is described in details in Chapter 8. The first phase of the scheduler is an extension to the CAP algorithm, named CAP-GS. Tasks that share colors (SRT and BE) are also grouped together, but the group utilization is allowed to be greater than the capacity given by the scheduling policy for each processor. When a group utilization is greater than the processor capacity, we perform a group splitting, removing the task with smaller utilization from the group and trying to partition the group again. HRT tasks do not share colors with any other task. The second phase is performed at run-time and consists of collecting tasks' information through the use of HPCs to detect when BE tasks are interfering with real-time tasks. When the collected hardware events are greater than the respective threshold and the running task is a BE task, the scheduler removes the CPU from the BE task, which prevents the access to the same cache lines and improves the tardiness of SRT tasks and ensures the HRT bounds of HRT tasks.

To evaluate the two-phases color-aware multicore real-time scheduler, we generated several task sets. We partitioned these task sets using the CAP-GS and BFD bin packing heuristic. Then, we executed the partitioned task sets using the dynamic scheduler, the CAP-GS without dynamic scheduling decisions, and the BFD using EPOS in a modern eight-core processor. We varied the WSS (32, 64, 128, and 256 KB) of each task as well as the write ratio (20 and 33%). From the experiments, we extracted the number of missed deadline and tardiness of each task. The main observations from the experiments were:

- The two-phases color-aware dynamic scheduler decreased the inter-core interference by combining a color-aware task partitioning with run-time scheduling decisions based on information collected by HPCs. In our experiments, the two-phases dynamic scheduler was able to provide HRT and SRT guarantees for all tasks in all task sets. CAP-GS without the dynamic phase also provided HRT guarantees, because it also decreased the inter-core interference that affects HRT tasks, which was observed in the BFD approach. We can conclude that the assignment of exclusive colors to HRT tasks works only if followed by the color-aware task partitioning heuristic.

- Although the tardiness observed in CAP-GS decreased as the WSS increased, the percentage of missed deadlines did not have a considerable change. This means that the SRT tasks were executing faster, but they were still completing their executions after the respective deadlines.

- BFD had an average deadline tardiness of up to 153 times worse than the tardiness in CAP-GS (for WSS of 256 KB and write ratio of 20%). The worse result in terms of missed deadlines for BFD was with WSS of 128 KB and write ratio of 33%, mainly due to a bigger WSS which leads to a higher contention rate for shared cache lines and preemption delays. With 256 KB, the inter-core interference decreased, so as the percentage of missed deadlines.

- The second phase of the dynamic scheduler can be modeled as a periodic task with a period of 10 ms and a WCET of 3.432 µs, resulting in a utilization of 0.00342. This low overhead has two main factors: (i) the EPOS timer handler function is fast and access few data; and (ii) the EPOS PMU mediator allows the access to HPCs to be performed without any software layer, which also increases the speed.

- Varying the write ratio almost did not change the percentage of missed deadlines in CAP-GS and BFD. Considering the tardiness, we could note a reduction on the WCET by SRT tasks, mainly in CAP-GS, but this reduction did not decrease the percentage of missed deadlines. BE tasks were the responsible for the greatest contention for shared cache lines.

- Varying the WSS affected mainly the SRT tasks. We could note an increase in the missed deadlines from 32 KB to 128 KB in the BFD and from 32 KB to 64 KB in the CAP-GS. For a WSS of 256 KB, the intra-task interference (self-evictions) is higher than with the other WSS, which decreased the inter-core interference and the missed deadlines.

## 9.2 CLOSING REMARKS

The use of multicore processors in real-time embedded systems can simplify the design of the hardware at the same time flexibility is improved. It can also replace hardware-specific implementations by software equivalents without compromising performance and real-time guarantees, as long as a proper run-time support system is available. Since multicore processors share resources, the predictability of real-time task execution is not straightforward. In particular, the cache memory hierarchy is one of the main sources for unpredictability. When real-time tasks access a same cache line due to true or false sharing, the delay caused by the cache coherence protocol may lead to deadline losses.

This research has studied how an RTOS can be designed to this scenario. Most importantly, by isolating HRT tasks workloads through the use of page coloring and combining it with partitioned FP scheduling, run-time scheduling decisions based on HPC information, and cache-aware task partitioning, *independent HRT tasks do not miss deadlines*, because task interference at the cache level is reduced. Moreover, *SRT tardiness is also reduced* even when a SRT task shares cache partitions with other SRT tasks and/or BE tasks. Thus, one of most important contributions of this research work is to demonstrate the viability of providing HRT guarantees in multicore processors at the RTOS level: deadlines of HRT tasks that share data in parallel real-time embedded applications are met as long as they are properly assigned to cores and do not demand more processing power than a single processor can deliver. Furthermore, the assignment of exclusive colors to HRT tasks must be performed whenever possible to reduce the delay caused by cache coherence protocol.

Our system model is mainly based on memory and scheduling isolation. Although CAP provides HRT bounds when parallel real-time embedded applications have utilizations smaller than a single processor's capacity, we did not deal with parallel real-time embedded applications that demand more processing power than an individual core can deliver.

Considering the evaluations performed during this work, a key advantage over the related work is the use of an RTOS designed for deeply embedded systems and the execution of tasks in a real processor. This allowed us to evaluate the run-time overhead, the cache partitioning impact, the CAP algorithm, and the two-phases color-aware scheduler in *practice*. Nevertheless, our evaluation methodology could be improved as discussed below.

**Realistic applications**. In our experiments, we used synthetic workloads to stress implementation issues, hardware features, and scheduling approaches. In future work it would be interesting to evaluate our contributions in realistic workloads. However, there are limitations in obtaining real-time workloads for multiprocessors, specially with HRT constraints, mainly due to three reasons (BRANDENBURG, 2011): (i) HRT applications are usually constructed using specific hardware, composed of several sensors and actuators that are difficult to reproduce in a lab; (ii) real embedded real-time applications are not made public by companies due to commercial reasons; and (iii) existing workloads do not yet stress all bottlenecks of a multicore processor, since the adoption of multicore systems by the embedded system industry is still at an early stage. There are several benchmarks proposed to stress all parts of a multicore processor, such as the PARSEC (BIENIA, 2011) and SPEC2000. However, these benchmarks do not have real-time constraints and thus are not appropriate to evaluate the impacts of real-time scheduling and cache partitioning techniques in the context of real-time systems.

**CPMD.** We measured the CPMD using HPCs and estimated the worst-case CPMD using information from the processor manual. An improvement would be use profiling tools to correct estimate the CPMD of applications and not always use the individual worst-case. A tradeoff, however, is that profiling tools for complex processor architectures, such as the one used in our experiments, do not exist or are very limited.

## 9.3 FUTURE DIRECTIONS

We now discuss future directions based on the contributions of this work.

**Parallel real-time embedded applications that demand more processing power than an individual core can deliver.** One limitation of our proposed two-phases color-aware multicore scheduling is the independence of HRT tasks, i.e., HRT tasks do not share data with any other task. Considering a scenario of bringing an algorithm implemented in hardware to software running on a multicore processor, data sharing among HRT tasks of this algorithm cannot be avoided. In this case, CAP has provided HRT bounds when the task group utilization is less than a single processor's capacity (using a partitioned scheduling). However, in this thesis, we did not consider a scenario where parallel real-time embedded applications, formed by several HRT tasks that share data, demand more processing power than an individual core can deliver. This subject is a challenge and will be investigate in the future. One possible idea is to isolate data sharing in specific code regions and use HPCs to monitor these regions and warning the scheduling.

**Implementation and analysis of scheduling and locking algorithms in EPOS.** The real-time support on EPOS can certainly be extended in several ways. Design, implementation, and evaluation of fairness-based and semi-partitioned schedulers and mutual exclusion protocols for multiprocessors in EPOS may lead to new results for the real-time system community.

Moreover, the investigation of how the locking of EPOS internal data structures and the use of faster data structures in the alarm and scheduler component may reduce the run-time overhead and, consequently, improve the schedulability of real-time tasks. Also, non-blocking data structure could be a good approach to decrease the time in which a processor is blocked until another processor finishes the execution of an OS mutual exclusion area.

Another interesting feature would be port EPOS to "more" embedded platforms, such as PandaBoard or Zynq, that feature a dual-core ARM processors and several co-processors, for instance. These platforms also feature a cache locking mechanism that can be used together with cache partitioning to improve the system predictability. Cache locking could be used at the OS-level to evaluate how it helps to reduce the OS run-time overhead, which has not been done yet.

**Specific hardware implementations.** Although it was not the focus of this thesis, the bus, pipeline, and other hardware features also affect the performance of a real-time application. These features in the current multicore processors are not designed to be predictable. Thus, the design of real-time cache coherence protocols, buses that provide real-time upper bounds, real-time memory controllers and prefetchers, are certainly a challenge.

**Cache partitioning.** In the context of cache partitioning, there is the lack of a color assignment algorithm. This color assignment algorithm would be responsible for analyzing the WSS of each task and the data sharing pattern among them, resulting in a set of colors for each task in a way that the WCET of each task is minimized.

**Dynamic scheduler.** The infrastructure created by this work, which is composed of HPCs and scheduling criteria, has opened several possible future directions. It is possible to investigate how new scheduling heuristics, such as sending an IPI message to stop a core when a specific number of hardware event is reached, help to prevent deadline misses. Also, investigate how other HPCs can collect information about data sharing and performance bottleneck to make the scheduling decision to be performed early, not only focused on decreasing the contention for shared lines, but also on improve other metrics, such as the deadline tardiness for SRT tasks.

**Multicore real-time benchmark.** There is an important lack in the real-time system community: there is no common benchmark available to evaluate works on the multicore real-time system area. A significant contribution would be to propose a benchmark for multicore real-time systems that could be used by researchers that want to evaluate their works.

# REFERENCES

ABENI, L.; BUTTAZZO, G. Resource reservation in dynamic real-time systems. **Real-Time Systems**, Kluwer Academic Publishers, Norwell, MA, USA, v. 27, n. 2, p. 123–167, jul. 2004. ISSN 0922-6443.

Aeronautical Radio, Inc. **Avionics Application Software Standard Interface: ARINC Specification 653 Part 0**. June 2013. Available from Internet: <https://www.arinc.com/cf/store-/catalog_detail.cfm?item_id=2039>.

AFEK, Y.; DICE, D.; MORRISON, A. Cache index-aware memory allocation. In: **Proceedings of the International Symposium on Memory Management**. New York, NY, USA: ACM, 2011. (ISMM '11), p. 55–64. ISBN 978-1-4503-0263-0.

AL-ZOUBI, H.; MILENKOVIC, A.; MILENKOVIC, M. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In: **Proceedings of the 42nd annual Southeast regional conference**. New York, NY, USA: ACM, 2004. (ACM-SE 42), p. 267–272. ISBN 1-58113-870-9.

ALBERS, K.; SLOMKA, F. Efficient feasibility analysis for real-time systems with EDF scheduling. In: **Proceedings of the Conference on Design, Automation and Test in Europe**. Washington, DC, USA: IEEE Computer Society, 2005. (DATE '05), p. 492–497. ISBN 0-7695-2288-2.

ALTMEYER, S.; DAVIS, R. I.; MAIZA, C. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. **Real-Time Systems**, Springer US, p. 1–28, 2012. ISSN 0922-6443.

AMD. **AMD64 Architecture Programmer's Manual Volume 2: System Programming**. 2010. June Publication # 24593. revision: 3.17.

ANDERSON, J. H.; BLOCK, A. Early-release fair scheduling. In: **In Proceedings of the 12th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2000. (ECRTS '00), p. 35–43.

ANDERSON, J. H.; BLOCK, A.; SRINIVASAN, A. Quick-release fair scheduling. In: **Proceedings of the 24th IEEE International Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2003. (RTSS '03), p. 130–. ISBN 0-7695-2044-8.

ANDERSON, J. H.; BUD, V.; DEVI, U. C. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In: **Proceedings of the 17th Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2005. (ECRTS '05), p. 199–208. ISBN 0-7695-2400-1.

APARICIO, L. C.; SEGARRA, J.; RODRíGUEZ, C.; VIÑALS, V. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. **Journal of Systems Architecture**, v. 57, n. 7, p. 695–706, 2011. ISSN 1383-7621.

ARM. **CortexTM-A9 MPCore Technical Reference Manual.** [S.l.: s.n.], 2010. Revision: r2p2.

ASADUZZAMAN, A.; SIBAI, F. N.; RANI, M. Improving cache locking performance of modern embedded systems via the addition of a miss table at the L2 cache level. **Journal of Systems Architecture**, v. 56, n. 4-6, p. 151–162, 2010. ISSN 1383-7621.

ÅSBERG, M.; NOLTE, T.; KATO, S.; RAJKUMAR, R. Exsched: An external cpu scheduler framework for real-time systems. In: **18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)**. [S.l.: s.n.], 2012.

AUDSLEY, N. C.; BURNS, A.; RICHARDSON, M. F.; WELLINGS, A. J. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In: **Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems**. [S.l.: s.n.], 1991.

AVILA, M.; PUAUT, I. **Hades Embedded Processor Timing ANalyzEr**. Jul 2014. Available from Internet: <http://raweb.inria.fr/rapportsactivite/RA2004/aces/uid43.html>.

AZIMI, R.; STUMM, M.; WISNIEWSKI, R. Online performance analysis by statistical sampling of microprocessor performance counters. In: **Proceedings of the 19th annual international con-**

**ference on Supercomputing**. New York, NY, USA: ACM, 2005. (ICS '05), p. 101–110. ISBN 1-59593-167-8.

AZIMI, R.; TAM, D.; SOARES, L.; STUMM, M. Enhancing operating system support for multicore processors by using hardware performance monitoring. **SIGOPS Operating System Review**, ACM, New York, NY, USA, v. 43, p. 56–65, April 2009. ISSN 0163-5980.

BAKER, T. P. Multiprocessor EDF and deadline monotonic schedulability analysis. In: **Proceedings of the 24th IEEE International Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2003. (RTSS '03), p. 120–. ISBN 0-7695-2044-8.

BAKER, T. P. An analysis of EDF schedulability on a multiprocessor. **IEEE Transactions on Parallel and Distributed Systems**, IEEE Press, Piscataway, NJ, USA, v. 16, n. 8, p. 760–768, ago. 2005. ISSN 1045-9219.

BAKER, T. P. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In: **Proceedings of the 2005 International Conference on Real-Time and Network Systems**. [S.l.: s.n.], 2005. (RTNS '05).

BAKER, T. P.; BARUAH, S. An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems. **Real-Time Systems**, Kluwer Academic Publishers, Norwell, MA, USA, v. 43, n. 1, p. 3–24, set. 2009. ISSN 0922-6443.

BAKER, T. P.; BARUAH, S. K. Schedulability analysis of multiprocessor sporadic task systems. In: **Handbook of Realtime and Embedded Systems**. [S.l.]: CRC Press, 2007.

BARABANOV, M. **A Linux-based Real-Time Operating System**. [S.l.], 1997. Master's thesis.

BARUAH, S. Techniques for multiprocessor global schedulability analysis. In: **Proceedings of the 28th IEEE International Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2007. (RTSS '07), p. 119–128. ISBN 0-7695-3062-1.

BARUAH, S. Partitioned EDF scheduling: a closer look. **Real-Time Systems**, Springer US, v. 49, n. 6, p. 715–729, 2013. ISSN 0922-6443.

BARUAH, S.; BONIFACI, V.; SPACCAMELA, A. M.; Stiller, S. Implementation of a speedup-optimal global EDF schedulability test. In: **Proceedings of the 21st Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2009. (ECRTS '09), p. 259–268. ISBN 978-0-7695-3724-5.

BARUAH, S.; BURNS, A. Sustainable scheduling analysis. In: **Proceedings of the 27th IEEE International Real-Time Systems Symposium**. [S.l.: s.n.], 2006. (RTSS '06), p. 159–168. ISSN 1052-8725.

BARUAH, S.; MOK, A.; ROSIER, L. Preemptively scheduling hard-real-time sporadic tasks on one processor. In: **Proceedings of the 11th Real-Time Systems Symposium**. [S.l.: s.n.], 1990. p. 182–190.

BARUAH, S. K.; COHEN, N. K.; PLAXTON, C. G.; VARVEL, D. A. Proportionate progress: A notion of fairness in resource allocation. **Algorithmica**, v. 15, p. 600–625, 1996.

BASTONI, A.; BRANDENBURG, B. B.; ANDERSON, J. H. Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In: **Proc. Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications**. Brussels, Belgium: [s.n.], 2010.

BASTONI, A.; BRANDENBURG, B. B.; ANDERSON, J. H. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In: **Proceedings of the 31st Real-Time Systems Symposium**. Washington, DC, USA: IEEE, 2010. (RTSS '10), p. 14–24. ISBN 978-0-7695-4298-0.

BASTONI, A.; BRANDENBURG, B. B.; ANDERSON, J. H. Is semi-partitioned scheduling practical? In: **Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2011. (ECRTS '11), p. 125–135. ISBN 978-0-7695-4442-7.

BASU, S. **Design Methods and Analysis of Algorithms**. PHI Learning, 2005. ISBN 9788120326378. Available from Internet: <http://books.google.com.br/books?id=SoDB4GI5JOYC>.

BELLOSA, F.; STECKERMEIER, M. The performance implications of locality information usage in shared-memory multiprocessors. **Journal of Parallel Distributed Computing**, Academic Press, Inc., Orlando, FL, USA, v. 37, p. 113–121, August 1996. ISSN 0743-7315.

BERTOGNA, M.; BARUAH, S. Tests for global EDF schedulability analysis. **Journal of Systems Architecture**, Elsevier North-Holland, Inc., New York, NY, USA, v. 57, n. 5, p. 487–497, maio 2011. ISSN 1383-7621.

BERTOGNA, M.; CIRINEI, M. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: **Proceedings of the 28th IEEE International Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2007. (RTSS '07), p. 149–160. ISBN 0-7695-3062-1.

BERTOGNA, M.; CIRINEI, M.; LIPARI, G. Improved schedulability analysis of EDF on multiprocessor platforms. In: **Proceedings of the 17th Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2005. (ECRTS '05), p. 209–218. ISBN 0-7695-2400-1.

BERTRAN, R.; GONZELEZ, M.; MARTORELL, X.; NAVARRO, N.; AYGUADE, E. A systematic methodology to generate decomposable and responsive power models for CMPs. **Computers, IEEE Trans. on**, v. 62, n. 7, p. 1289–1302, 2013. ISSN 0018-9340.

BIENIA, C. **Benchmarking Modern Multiprocessors**. Thesis (Ph.D) — Princeton University, January 2011.

BIRCHER, W.; JOHN, L. Complete system power estimation using processor performance events. **IEEE Transactions on Computers**, v. 61, n. 4, p. 563–577, 2012. ISSN 0018-9340.

BLETSAS, K.; ANDERSSON, B. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In: **Proceedings of the 2009 30th IEEE Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2009. (RTSS '09), p. 447–456. ISBN 978-0-7695-3875-4.

BLOCK, A.; LEONTYEV, H.; BRANDENBURG, B.; ANDERSON, J. A flexible real-time locking protocol for multiprocessors. In: **Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications**. [S.l.: s.n.]. (RTCSA '07).

BOOCH, G. **Object-Oriented Analysis and Design with Applications (3rd Edition)**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN 020189551X.

BORG, A.; WELLINGS, A.; GILL, C.; CYTRON, R. Real-time memory management: life and times. In: **Proceedings of the 18th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2006. (ECRTS '06), p. 11 pp.–250. ISSN 1068-3070.

BOYD-WICKIZER, S.; CLEMENTS, A. T.; MAO, Y.; PESTEREV, A.; KAASHOEK, M. F.; MORRIS, R.; ZELDOVICH, N. An analysis of linux scalability to many cores. In: **Proceedings of the 9th USENIX conference on Operating systems design and implementation**. Berkeley, CA, USA: USENIX Association, 2010. (OSDI'10), p. 1–8.

BRANDENBURG, B.; ANDERSON, J. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In: **Proceedings of the 19th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2007. (ECRTS '07), p. 61–70. ISSN 1068-3070.

BRANDENBURG, B. B. **Scheduling and Locking in Multiprocessor Real-Time Operating Systems**. Thesis (Ph.D) — The University of North Carolina at Chapel Hill, 2011.

BRANDENBURG, B. B.; ANDERSON, J. H. Feather-trace: A lightweight event tracing toolkit. In: **In Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications**. [S.l.: s.n.], 2007. (OSPERT '07), p. 61–70.

BRANDENBURG, B. B.; ANDERSON, J. H. On the implementation of global real-time schedulers. In: **RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2009. p. 214–224. ISBN 978-0-7695-3875-4.

BRANDENBURG, B. B.; CALANDRINO, J. M.; ANDERSON, J. H. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: **Proceedings of the 2008 Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2008. (RTSS '08), p. 157–169. ISBN 978-0-7695-3477-0.

BRANDENBURG, B. B.; LEONTYEV, H.; ANDERSON, J. H. An overview of interrupt accounting techniques for multiprocessor real-time systems. **Journal of Systems Architecture**, Elsevier North-Holland, Inc., New York, NY, USA, v. 57, n. 6, p. 638–654, jun. 2011. ISSN 1383-7621.

BUGNION, E.; ANDERSON, J. M.; MOWRY, T. C.; ROSENBLUM, M.; LAM, M. S. Compiler-directed page coloring for multiprocessors. In: **Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems**. New York, NY, USA: ACM, 1996. (ASPLOS VII), p. 244–255. ISBN 0-89791-767-7.

BUI, B.; CACCAMO, M.; SHA, L.; MARTINEZ, J. Impact of cache partitioning on multi-tasking real time embedded systems. In: **Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications**. [S.l.: s.n.], 2008. (RTCSA '08), p. 101–110. ISSN 1533-2306.

BURNS, A. Scheduling hard real-time systems: a review. **Software Engineering Journal**, v. 6, n. 3, p. 116–128, 1991. ISSN 0268-6961.

BURNS, A.; DAVIS, R.; WANG, P.; ZHANG, F. Partitioned EDF scheduling for multiprocessors using a c=d task splitting scheme. **Real-Time Systems**, Kluwer Academic Publishers, Norwell, MA, USA, v. 48, n. 1, p. 3–33, jan. 2012. ISSN 0922-6443.

BUTTAZZO, G. C. Rate monotonic vs. EDF: Judgment day. **Real-Time Systems**, Kluwer Academic Publishers, Norwell, MA, USA, v. 29, n. 1, p. 5–26, jan. 2005. ISSN 0922-6443.

CALANDRINO, J.; ANDERSON, J.; BAUMBERGER, D. A hybrid real-time scheduling approach for large-scale multicore platforms.

In: **Proceedings of the 19th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2007. (ECRTS '07), p. 247–258. ISSN 1068-3070.

CALANDRINO, J. M.; ANDERSON, J. H. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In: **Proceedings of the 2008 Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2008. (ECRTS '08), p. 299–308. ISBN 978-0-7695-3298-1.

CALANDRINO, J. M.; ANDERSON, J. H. On the design and implementation of a cache-aware multicore real-time scheduler. In: **Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2009. (ECRTS '09), p. 194–204. ISBN 978-0-7695-3724-5.

CALANDRINO, J. M.; LEONTYEV, H.; BLOCK, A.; DEVI, U. C.; ANDERSON, J. H. LITMUSRT: A testbed for empirically comparing real-time multiprocessor schedulers. In: **Proceedings of the 27th IEEE International Real-Time Systems Symposium**. Washington, DC, USA: IEEE Computer Society, 2006. (RTSS '06), p. 111–126. ISBN 0-7695-2761-2.

CAMPOY, M.; IVARS, A. P.; MATAIX, J. V. B. Static use of locking caches in multitask preemptive real-time systems. In: **Proceedings of IEEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)**. London, UK: [s.n.], 2001.

CHATTOPADHYAY, B.; BARUAH, S. A lookup-table driven approach to partitioned scheduling. In: **Proceedingsof the 17th Real-Time and Embedded Technology and Applications Symposium**. [S.l.: s.n.], 2011. (RTAS '11), p. 257–265. ISSN 1080-1812.

CHEN, Y.; LI, W.; KIM, C.; TANG, Z. Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy. In: **Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing**. [S.l.]: IEEE, 2009. (IPDPS '09), p. 1–11. ISBN 978-1-4244-3751-1.

CHILDS, S.; INGRAM, D. The linux-srt integrated multimedia operating system: bringing qos to the desktop. In: **Proceedings of the**

**7th Real-Time Technology and Applications Symposium**. [S.l.: s.n.], 2001. (RTAS '01), p. 135–140. ISSN 1080-1812.

CHILIMBI, T. M.; HILL, M. D.; LARUS, J. R. Making pointer-based data structures cache conscious. **Computer**, IEEE, Los Alamitos, CA, USA, v. 33, n. 12, p. 67–74, dez. 2000. ISSN 0018-9162.

CHO, H.; RAVINDRAN, B.; JENSEN, E. D. An optimal real-time scheduling algorithm for multiprocessors. In: **Proceedings of the 27th IEEE International Real-Time Systems Symposium**. [S.l.]: IEEE, 2006. (RTSS '06), p. 101–110. ISBN 0-7695-2761-2.

CHOUSEIN, A.; MAHAPATRA, R. N. Fully associative cache partitioning with don't care bits for real-time applications. **SIGBED Review**, ACM, v. 2, n. 2, p. 35–38, Apr 2005. ISSN 1551-3688.

CLOUTIER, P.; MANTEGAZZA, P.; PAPACHARALAMBOUS, S.; SOANER, I.; HUGHES, S.; YAGHMOUR, K. DIAPM-RTAI position paper. In: **Real-Time Linux Workshop Vol. 3**. [S.l.: s.n.], 2000.

COFFMAN JR., E. G.; GAREY, M. R.; JOHNSON, D. S. Approximation algorithms for NP-hard problems. In: HOCHBAUM, D. S. (Ed.). Boston, MA, USA: PWS Publishing Co., 1997. cap. Approximation Algorithms for Bin Packing: A Survey, p. 46–93. ISBN 0-534-94968-1.

CRACIUNAS, S. S.; KIRSCH, C. M.; PAYER, H.; SOKOLOVA, A.; STADLER, H.; STAUDINGER, R. A compacting real-time memory management system. In: **USENIX 2008 Annual Technical Conference on Annual Technical Conference**. Berkeley, CA, USA: USENIX Association, 2008. (ATC'08), p. 349–362.

CULLMANN, C.; FERDINAND, C.; GEBHARD, G.; GRUND, D.; MAIZA, C.; REINEKE, J.; TRIQUET, B.; WEGENER, S.; WILHELM, R. Predictability considerations in the design of multi-core embedded systems. **Ingénieurs de l'Automobile**, v. 807, p. 36–42, September 2010. ISSN 0020-1200.

CZARNECKI, K.; EISENECKER, U. W. **Generative programming: methods, tools, and applications**. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.

DAVID, F. M.; CARLYLE, J. C.; CAMPBELL, R. H. Context switch overheads for linux on arm platforms. In: **Proceedings of the 2007 workshop on Experimental computer science**. New York, NY, USA: ACM, 2007. (ExpCS '07). ISBN 978-1-59593-751-3.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 43, n. 4, p. 35:1–35:44, out. 2011. ISSN 0360-0300.

DENNING, P. J. Thrashing: Its causes and prevention. In: **Proceedings of the Fall Joint Computer Conference, Part I**. New York, NY, USA: ACM, 1968. (AFIPS '68), p. 915–922.

DEVI, U. An improved schedulability test for uniprocessor periodic task systems. In: **Proceedings of the 15th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2003. (ECRTS '03), p. 23–30.

DEVI, U. **Soft Real-Time Scheduling on Multiprocessors**. Thesis (Ph.D) — The University of North Carolina at Chapel Hill, 2006.

DEVI, U.; ANDERSON, J. Tardiness bounds under global EDF scheduling on a multiprocessor. In: **Proceedings of the 26th IEEE International Real-Time Systems Symposium**. [S.l.: s.n.], 2005. (RTSS '05), p. 12 pp.–341. ISSN 1052-8725.

DHALL, S. K.; LIU, C. L. On a real-time scheduling problem. **Operations Research**, v. 26, n. 1, p. 127–140, 1978.

DIJKSTRA, E. W. Cooperating sequential processes. In: GENUYS, F. (Ed.). **Programming Languages: NATO Advanced Study Institute**. [S.l.]: Academic Press, 1968. p. 43–112.

DONGARRA, J.; LONDON, K.; MOORE, S.; MUCCI, P.; TERPSTRA, D.; YOU, H.; ZHOU, M. Experiences and lessons learned with a portable interface to hardware performance counters. In: **Proceedings of the 17th International Symposium on Parallel and Distributed Processing**. Washington, DC, USA: IEEE Computer Society, 2003. (IPDPS '03), p. 289.2–. ISBN 0-7695-1926-1.

DRONGOWSKI, P. J. **An introduction to analysis and optimization with AMD CodeAnalyst Performance Analyzer**. [S.l.: s.n.], 2008.

EBRAHIMI, E.; MUTLU, O.; LEE, C. J.; PATT, Y. N. Coordinated control of multiple prefetchers in multi-core systems. In: **Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture**. New York, NY, USA: ACM, 2009. (MICRO 42), p. 316–326. ISBN 978-1-60558-798-1.

ENEA. **Enea Operating System Embedded (OSE)**. 2013. Available from Internet: <http://www.enea.com/solutions/rtos/ose/>.

EPOS. **EPOS website**. Feb 2014. Available from Internet: <http:/-/epos.lisha.ufsc.br>.

EXPRESS, L. **ThreadX**. Dec 2013. Available from Internet: <http:/-/rtos.com/products/threadx/>.

FAGGIOLI, D.; CHECCONI, F.; TRIMARCHI, M.; SCORDINO, C. An EDF scheduling class for the Linux kernel. In: **Proceedings of the Eleventh Real-Time Linux Workshop**. Dresden, Germany: [s.n.], 2009.

FALK, H.; PLAZAR, S.; THEILING, H. Compile-time decided instruction cache locking using worst-case execution paths. In: **Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis**. New York, NY, USA: ACM, 2007. (CODES+ISSS '07), p. 143–148. ISBN 978-1-59593-824-4.

FISHER, N.; GOOSSENS, J.; BARUAH, S. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. **Real-Time Systems**, Springer US, v. 45, n. 1-2, p. 26–71, 2010. ISSN 0922-6443.

FOTHERINGHAM, J. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. **Communications of the ACM**, ACM, New York, NY, USA, v. 4, n. 10, p. 435–436, out. 1961. ISSN 0001-0782.

FRÖHLICH, A. A. **Application-Oriented Operating Systems**. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. (GMD Research Series, 17).

FRÖHLICH, A. A. A Comprehensive Approach to Power Management in Embedded Systems. **International Journal of Distributed Sensor Networks**, v. 2011, n. 1, p. 19, 2011. ISSN 1550-1477.

FRÖHLICH, A. A.; GRACIOLI, G.; SANTOS, J. F. Periodic timers revisited: The real-time embedded system perspective. **Computers & Electrical Engineering**, Pergamon Press, Inc., Tarrytown, NY, USA, v. 37, n. 3, p. 365–375, maio 2011. ISSN 0045-7906.

FRÖHLICH, A. A.; SCHRÖDER-PREIKSCHAT, W. Scenario Adapters: Efficiently Adapting Components. In: **4th World Multiconference on Systemics, Cybernetics and Informatics**. Orlando, USA: [s.n.], 2000.

FU, L.; SCHWEBEL, R. **Real-time linux wiki. RT PREEMPT HOWTO**. Jul 2014. Available from Internet: <https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO>.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability; A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1990. ISBN 0716710455.

GERUM, P. **The Xenomai real-time system**. 2nd. ed. [S.l.]: O'Reilly Media, 2008. 365–385 p. Chapter 13.

GLEIXNER, T.; NIEHAUS, D. Hrtimers and Beyond: Transforming the Linux Time Subsystems. In: **Proceedings of the Linux Symposium**. Ottawa, Ontario, Canada: [s.n.], 2006. v. 1, p. 333–346.

GOOSSENS, J.; FUNK, S.; BARUAH, S. Priority-driven scheduling of periodic task systems on multiprocessors. **Real-Time Systems**, Kluwer Academic Publishers, Norwell, MA, USA, v. 25, n. 2-3, p. 187–205, sep 2003. ISSN 0922-6443.

GRACIOLI, G.; FISCHMEISTER, S. Tracing interrupts in embedded software. In: **Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems**. New York, NY, USA: ACM, 2009. (LCTES '09), p. 137–146. ISBN 978-1-60558-356-3.

GRACIOLI, G.; FRÖHLICH, A. A. An embedded operating system API for monitoring hardware events in multicore processors. In: **Workshop on Hardware-support for parallel program correctness - IEEE Micro 2011**. Porto Alegre, Brazil: [s.n.], 2011. ISBN 978-1-4503-1053-6.

GRACIOLI, G.; FRÖHLICH, A. A. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In: **Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications**. [S.l.]: IEEE Computer Society, 2013. (RTCSA '13), p. 72–81. ISSN 1533-2306.

GRACIOLI, G.; FRÖHLICH, A. A.; PELLIZZONI, R.; FISCHMEISTER, S. Implementation and evaluation of global and partitioned scheduling in a real-time OS. **Real-Time Systems**, Springer US, v. 49, n. 6, 2013. ISSN 1573-1383.

Green Hills Software. **Everything you need to develop embedded software for aerospace & defense**. 2011. Marketing material. Available from Internet: <http://www.ghs.com/AerospaceDefense.html>.

GRUND, D.; REINEKE, J. Precise and efficient FIFO-replacement analysis based on static phase detection. In: **Proceedings of the 22nd Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2010. (ECRTS '10), p. 155–164. ISBN 978-1-4244-7546-9. ISSN 1068-3070.

GUAN, N.; STIGGE, M.; YI, W.; YU, G. Cache-aware scheduling and analysis for multicores. In: **Proceedings of the 7th ACM International Conference on Embedded Software**. [S.l.]: ACM, 2009. (EMSOFT '09), p. 245–254. ISBN 978-1-60558-627-4.

HARDY, D.; PUAUT, I. Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches. In: **17th International Conference on Real-Time and Network Systems**. Paris, France: [s.n.], 2009. p. 45–54.

HÄRTIG, H.; ROITZSCH, M. Ten years of research on l4-based real-time systems. In: **8th Real-Time Linux Workshop**. [S.l.: s.n.], 2006.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. [S.l.]: Morgan Kaufmann, Fourth Edition, 2006. Paperback. ISBN 0123704901.

HERTER, J.; BACKES, P.; HAUPENTHAL, F.; REINEKE, J. CAMA: A predictable cache-aware memory allocator. In: **Proceedings of 23rd Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2011. (ECRTS '11), p. 23–32. ISSN 1068-3070.

HERTER, J.; REINEKE, J. Making dynamic memory allocation static to support WCET analyses. In: **Proceedings of 9th International Workshop on WCET Analysis**. [S.l.: s.n.], 2009.

HOCHBAUM, D. S.; SHMOYS, D. B. Using dual approximation algorithms for scheduling problems theoretical and practical results. **Journal of the ACM**, ACM, New York, NY, USA, v. 34, n. 1, p. 144–162, jan. 1987. ISSN 0004-5411.

HUM, H.; GOODMAN, J. **Forward state for use in cache coherency in a multiprocessor system**. jul. 2005. US Patent 6,922,756. Available from Internet: <http://www.google.com/patents/US6922756>.

IEEE. **IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP)**. [S.l.]: IEEE Computer Society, 2003. Number Std 1003.13-2003.

Intel Corporation. **An introduction to the Intel QuickPath Interconnect**. 2009. January Document Number: 320412-001US.

Intel Corporation. **Intel® 64 and IA-32 Architectures Software Developer's Manual**. [S.l.: s.n.], 2011.

Intel Corporation. **Intel® 64 and IA-32 Architectures Optimization Reference Manual**. [S.l.: s.n.], 2012.

ISHIWATA, Y.; MATSUI, T. Development of Linux which has advanced real-time processing function. In: **Proceedings of the 16th Annual Conference of Robotics Society of Japan**. [S.l.: s.n.], 1998. p. 355–356.

IYER, R. CQoS: A framework for enabling qos in shared caches of CMP platforms. In: **Proceedings of the 18th Annual International**

**Conference on Supercomputing**. New York, NY, USA: ACM, 2004. (ICS '04), p. 257–266. ISBN 1-58113-839-3.

JENSEN, E. D.; LOCKE, C. D.; TOKUDA, H. A time-driven scheduling model for real-time operating systems. In: **Proceedings of the 1985 Real-Time Systems Symposium**. [S.l.]: IEEE Computer Society, 1985. (RTSS '85), p. 112–122. ISBN 0-8186-0675-4.

JIANG, Y.; ZHANG, E. Z.; TIAN, K.; SHEN, X. Is reuse distance applicable to data locality analysis on chip multiprocessors? In: **Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction**. Berlin, Heidelberg: Springer-Verlag, 2010. (CC'10/ETAPS'10), p. 264–282. ISBN 3-642-11969-7, 978-3-642-11969-9.

JOHNSON, D. **Near-optimal Bin Packing Algorithms**. Thesis (Ph.D), 1973.

JOSEPH, M.; PANDYA, P. K. Finding response times in a real-time system. **The Computer Journal**, v. 29, n. 5, p. 390–395, 1986.

KANEKO, K.; KANEHIRO, F.; MORISAWA, M.; MIURA, K.; NAKAOKA, S.; KAJITA, S. Cybernetic human hrp-4c. In: **Proceedings of the 9th IEEE-RAS International Conference on Humanoid Robots**. [S.l.: s.n.], 2009. p. 7–14.

KATO, S. **AIRS website**. Oct 2012. Available from Internet: <http://www.ertl.jp/˜shinpei/airs/>.

KATO, S.; YAMASAKI, N. Real-time scheduling with task splitting on multiprocessors. In: **Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications**. Washington, DC, USA: IEEE Computer Society, 2007. (RTCSA '07), p. 441–450. ISBN 0-7695-2975-5.

KATO, S.; YAMASAKI, N. Portioned EDF-based scheduling on multiprocessors. In: **Proceedings of the 8th ACM international conference on Embedded software**. New York, NY, USA: ACM, 2008. (EMSOFT '08), p. 139–148. ISBN 978-1-60558-468-3.

KENNA, C.; HERMAN, J.; WARD, B.; ANDERSON, J. H. Making shared caches more predictable on multicore platforms. In: **Proceedings of the 25th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2013. (ECRTS '13), p. 157–167.

KESSLER, R. E.; HILL, M. D. Page placement algorithms for large real-indexed caches. **ACM Transactions on Computer Systems**, ACM, New York, NY, USA, v. 10, n. 4, p. 338–359, Nov 1992. ISSN 0734-2071.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C. V.; LOINGTIER, J.-M.; IRWIN, J. Aspect-oriented programming. In: **ECOOP'97**. [S.l.]: SpringerVerlag, 1997. p. 220–242.

KIM, H.; KANDHALU, A.; RAJKUMAR, R. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In: **Proceedings of the 25th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2013. (ECRTS '13), p. 80–89.

KIRK, D.; STROSNIDER, J. Smart (strategic memory allocation for real-time) cache design using the mips r3000. In: **Proceedings of the 11th Real-Time Systems Symposium**. [S.l.: s.n.], 1990. (RTSS '90), p. 322–330.

KNUTH, D. E. **The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89683-4.

KOHONEN, T. **Content-Addressable Memories**. Second. [S.l.]: Springer-Verlag, 1987.

Lea, D. **A Memory Allocator**. 1996. Unix/Mail, 6/96.

LEHOCZKY, J.; SHA, L.; DING, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In: **Proceedings of 10th the Real Time Systems Symposium**. [S.l.: s.n.], 1989. p. 166–171.

LELLI, J.; FAGGIOLI, D.; CUCINOTTA, T.; LIPARI, G. An experimental comparison of different real-time schedulers on multicore systems. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, USA, v. 85, n. 10, p. 2405–2416, out. 2012. ISSN 0164-1212.

LEONTYEV, H.; ANDERSON, J. H. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. In: **ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2008. p. 191–200. ISBN 978-0-7695-3298-1.

LEUNG, J. Y.-T.; WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. **Performance Evaluation**, v. 2, n. 4, p. 237 – 250, 1982. ISSN 0166-5316.

LEVIN, G.; FUNK, S.; SADOWSKI, C.; PYE, I.; BRANDT, S. DP-FAIR: A simple model for understanding optimal multiprocessor scheduling. In: **Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2010. (ECRTS '10), p. 3–13. ISBN 978-0-7695-4111-2.

LI, C.; DING, C.; SHEN, K. Quantifying the cost of context switch. In: **Proceedings of the 2007 workshop on Experimental computer science**. New York, NY, USA: ACM, 2007. (ExpCS '07). ISBN 978-1-59593-751-3.

LIANG, Y.; MITRA, T. Cache modeling in probabilistic execution time analysis. In: **Proceedings of the 45th ACM/IEEE Design Automation Conference**. [S.l.: s.n.], 2008. (DAC '08), p. 319–324. ISSN 0738-100X.

LIANG, Y.; MITRA, T. Instruction cache locking using temporal reuse profile. In: **Proceedings of the 47th ACM/IEEE Design Automation Conference**. [S.l.: s.n.], 2010. (DAC '10), p. 344–349. ISSN 0738-100X.

LIEDTKE, J.; HAERTIG, H.; HOHMUTH, M. OS-controlled cache predictability for real-time systems. In: **Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium**. [S.l.]: IEEE, 1997. (RTAS '97), p. 213–223. ISBN 0-8186-8016-4.

LIN, J.; LU, Q.; DING, X.; ZHANG, Z.; ZHANG, X.; SADAYAPPAN, P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In: **Proceedings of the 14th International Symposium on High Performance Computer Architecture**. [S.l.]: IEEE, 2008. (HPCA'08), p. 367–378.

LINDSAY, C. **LWFG: A Cache-Aware Multi-core Real-Time Scheduling Algorithm**. Dissertation (Masters) — Virginia Polytechnic Institute and State University, 2012.

LIU, C.; SIVASUBRAMANIAM, A.; KANDEMIR, M. Organizing the last line of defense before hitting the memory wall for CMPs. In: **Proceedings of the 10th International Symposium on High Performance Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2004. (HPCA '04), p. 176–. ISBN 0-7695-2053-7.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. **J. ACM**, ACM, New York, NY, USA, v. 20, n. 1, p. 46–61, jan. 1973. ISSN 0004-5411.

LIU, J. **Real-Time Systems**. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN 0130996513.

LU, Q.; LIN, J.; DING, X.; ZHANG, Z.; ZHANG, X.; SADAYAPPAN, P. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In: **Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques**. [S.l.: s.n.], 2009. (PACT '09), p. 246–257. ISSN 1089-795X.

LUDWICH, M. K.; FRÖHLICH, A. A. Abstracting hardware devices to embedded Java applications. In: **IADIS Applied Computing 2011**. Rio de Janeiro, Brazil: [s.n.], 2011. p. 371–378. ISBN 978-989-8533-06-7.

MALLADI, R. K. **Using Intel® VTuneTM Performance Analyzer Events/Ratios  Optimizing Applications**. [S.l.: s.n.], 2010.

MANCUSO, R.; DUDKO, R.; BETTI, E.; CESATI, M.; CACCAMO, M.; PELLIZZONI, R. Real-time cache management framework for multi-core architectures. In: **Proceedings of the 19th IEEE**

**Real-Time and Embedded Technology and Applications Symposium**. [S.l.: s.n.], 2013. (RTAS ’13), p. 45–54. ISSN 1080-1812.

MARCONDES, H.; CANCIAN, R.; STEMMER, M.; FRÖHLICH, A. A. On the design of flexible real-time schedulers for embedded systems. In: **Proceedings of the 2009 International Conference on Computational Science and Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (CSE ’09), p. 382–387. ISBN 978-0-7695-3823-5.

MARWEDEL, P. **Embedded System Design**. $2^{nd}$ edition. Berlin: Springer, 2006. ISBN 978-0-387-29237-3.

MASMANO, M.; RIPOLL, I.; CRESPO, A. A comparison of memory allocators for real-time applications. In: **Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems**. New York, NY, USA: ACM, 2006. (JTRES ’06), p. 68–76. ISBN 1-59593-544-4.

MASMANO, M.; RIPOLL, I.; CRESPO, A.; REAL, J. TLSF: a new dynamic memory allocator for real-time systems. In: **Proceedings of the 16th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2004. (ECRTS ’04), p. 79–88. ISSN 1068-3070.

MASRUR, A.; CHAKRABORTY, S.; FÄRBER, G. Constant-time admission control for partitioned EDF. In: **Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems**. Washington, DC, USA: IEEE Computer Society, 2010. (ECRTS ’10), p. 34–43. ISBN 978-0-7695-4111-2.

MASRUR, A.; DROSSIER, S.; FARBER, G. Improvements in polynomial-time feasibility testing for EDF. In: **Design, Automation and Test in Europe**. [S.l.: s.n.], 2008. (DATE ’08), p. 1033–1038.

MAY, J. Mpx: Software for multiplexing hardware performance counters in multithreaded programs. In: **Proceedings of the 15th International Parallel and Distributed Processing Symposium.** [S.l.: s.n.], 2001. p. 8 pp.

MENTOR, G. **Nucleos Real-Time Operating System**. 2013. Available from Internet: <http://www.mentor.com/embedded-software/nucleus/>.

MOGUL, J. C.; BORG, A. The effect of context switches on cache performance. **SIGOPS Operating System Review**, ACM, New York, NY, USA, v. 25, n. Special Issue, p. 75–84, abr. 1991. ISSN 0163-5980.

MOHAN, S.; CACCAMO, M.; SHA, L.; PELLIZZONI, R.; ARUNDALE, G.; KEGLEY, R.; NIZ, D. de. Using multicore architectures in cyber-physical systems. In: **Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components**. Michigan, USA: [s.n.], 2011.

MOK, A. K.-L. **Fundamental Design Problems of Distributed Systems for the Hard–Real–Time Environment**. Thesis (Ph.D), 1983.

MOLLISON, M.; ANDERSON, J. H. Utilization-controlled task consolidation for power optimization in multi-core real-time systems (to appear). In: **18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)**. [S.l.: s.n.], 2012. v. 1. ISSN 1533-2306.

MUCCI, P. J.; BROWNE, S.; DEANE, C.; HO, G. PAPI: a portable interface to hardware performance counters. In: **In Proceedings of the Department of Defense HPCMP Users Group Conference**. [S.l.: s.n.], 1999. p. 7–10.

MÜCK, T. R.; FRÖHLICH, A. A. HyRA: A Software-defined Radio Architecture for Wireless Embedded Systems. In: **10th International Conference on Networks**. St. Maarten, The Netherlands Antilles: [s.n.], 2011. p. 246–251. ISBN 978-1-61208-002-4.

MUELLER, F. Compiler support for software-based cache partitioning. In: **Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, and Tools for Real-time Systems**. New York, NY, USA: ACM, 1995. (LCTES '95), p. 125–133.

MURALIDHARA, S.; KANDEMIR, M.; RAGHAVAN, P. Intra-application cache partitioning. In: **Proceedings of the 2010**

**IEEE International Symposium on Parallel Distributed Processing**. [S.l.: s.n.], 2010. (IPDPS'10), p. 1 –12. ISSN 1530-2075.

NEGI, H. S.; MITRA, T.; ROYCHOUDHURY, A. Accurate estimation of cache-related preemption delay. In: **Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis**. New York, NY, USA: ACM, 2003. (CODES+ISSS '03), p. 201–206. ISBN 1-58113-742-7.

NEMATI, F.; BEHNAM, M.; NOLTE, T. Efficiently migrating real-time systems to multi-cores. In: **Proceeding of the 14th IEEE International Conference on Emerging Techonologies and Factory Automation**. [S.l.: s.n.], 2009. (ETFA '09), p. 1–8. ISSN 1946-0759.

OGASAWARA, T. An algorithm with constant execution time for dynamic storage allocation. In: **Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications**. Washington, DC, USA: IEEE Computer Society, 1995. (RTCSA '95), p. 21–. ISBN 0-8186-7106-8.

OIKAWA, S.; RAJKUMAR, R. Portable rk: A portable resource kernel for guaranteed and enforced timing behavior. In: **Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium**. Washington, DC, USA: IEEE Computer Society, 1999. (RTAS '99), p. 111–120.

PALOPOLI, L.; CUCINOTTA, T.; MARZARIO, L.; LIPARI, G. AQuoSA - adaptive quality of service architecture. **Software Practice and Experience**, John Wiley & Sons, Inc., New York, NY, USA, v. 39, n. 1, p. 1–31, jan. 2009. ISSN 0038-0644.

PAPAMARCOS, M. S.; PATEL, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. In: **Proceedings of the 11th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 1984. (ISCA '84), p. 348–354. ISBN 0-8186-0538-3.

PARNAS, D. On the design and development of program families. **IEEE Transactions on Software Engineering**, SE-2, n. 1, p. 1 – 9, march 1976. ISSN 0098-5589.

POLPETA, F. V.; FRÖHLICH, A. A. Hardware mediators: A portability artifact for component-based systems. In: **Embedded and Ubiquitous Computing**. [S.l.]: Springer Berlin Heidelberg, 2004. (Lecture Notes in Computer Science), p. 271–280.

POLPETA, F. V.; FRÖHLICH, A. A. On the automatic generation of soc-based embedded systems. In: **In Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation**. [S.l.: s.n.], 2005.

PUAUT, I. Real-time performance of dynamic memory allocation algorithms. In: **Proceedings of the 14th Euromicro Conference on Real-Time Systems**. [S.l.: s.n.], 2002. (ECRTS '02), p. 41–49. ISSN 1068-3070.

PUAUT, I.; DECOTIGNY, D. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In: **Proceedings of the 23rd IEEE Real-Time Systems Symposium**. [S.l.: s.n.], 2002. (RTSS '02), p. 114–123. ISSN 1052-8725.

QIAN, B. feng; YAN, L. min. The research of the inclusive cache used in multi-core processor. In: **Proceeding of the International Conference on Electronic Packaging Technology High Density Packaging**. [S.l.: s.n.], 2008. (ICEPT-HDP' 08), p. 1–4.

QNX. **QNX® Neutrino® RTOS**. 2013. Available from Internet: <http://www.qnx.com/products/neutrino-rtos/neutrino-rtos-.html>.

QUADROS. **RTXC Quadros Operating System**. 2005. Available from Internet: <http://www.vas.co.kr/products/datasheet/rtxc-quadros.pdf>.

QURESHI, M. K.; PATT, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In: **Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.]: IEEE, 2006. (MICRO 39), p. 423–432. ISBN 0-7695-2732-9.

RAFIQUE, N.; LIM, W.-T.; THOTTETHODI, M. Architectural support for operating system-driven CMP cache management. In: **Proceedings of the 15th International Conference on Parallel**

**Architectures and Compilation Techniques**. New York, NY, USA: ACM, 2006. (PACT '06), p. 2–12. ISBN 1-59593-264-X.

RAJKUMAR, R. Resource kernels: Why resource reservation should be the preferred paradigm of construction of embedded real-time systems. In: **18th Euromicro Conference on Real-Time Systems**. Dresden, Germany: [s.n.], 2006. Keynote talk. Available from Internet: <http://ecrts06.tudos.org/docs/RajRajkumar.pdf>.

RAJKUMAR, R.; SHA, L.; LEHOCZKY, J. Real-time synchronization protocols for multiprocessors. In: **Proceedings of the 9th Real-Time Systems Symposium**. [S.l.: s.n.], 1988. (RTSS '88), p. 259–269.

RANGANATHAN, P.; ADVE, S.; JOUPPI, N. P. Reconfigurable caches and their application to media processing. In: **Proceedings of the 27th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 2000. (ISCA '00), p. 214–224. ISBN 1-58113-232-8.

REINEKE, J.; GRUND, D.; BERG, C.; WILHELM, R. Timing predictability of cache replacement policies. **Real-Time Systems**, Kluwer Academic Publishers, Norwell, MA, USA, v. 37, n. 2, p. 99–122, nov. 2007. ISSN 0922-6443.

ROMER, T.; LEE, D.; BERSHAD, B. N.; CHEN, J. B. Dynamic page mapping policies for cache conflict resolution on standard hardware. In: **Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation**. [S.l.: s.n.], 1994. (OSDI '94), p. 255–266.

RTEMS. **RTEMS Real-Time Operating System**. 2013. Available from Internet: <http://www.rtems.org/>.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design & Test**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 18, p. 23–33, November 2001. ISSN 0740-7475.

SARKAR, A.; MUELLER, F.; RAMAPRASAD, H. Predictable task migration for locked caches in multi-core systems. In: **Proceedings**

**of the the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems**. New York: ACM, 2011. (LCTES'11), p. 131–140. ISBN 978-1-4503-0555-6.

SARKAR, A.; MUELLER, F.; RAMAPRASAD, H. Static task partitioning for locked caches in multi-core real-time systems. In: **Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems**. NY, USA: ACM, 2012. (CASES '12), p. 161–170. ISBN 978-1-4503-1424-4.

SEHLBERG, D.; ERMEDAHL, A.; GUSTAFSSON, J.; LISPER, B.; WIEGRATZ, S. Static WCET analysis of real-time task-oriented code in vehicle control systems. In: **Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation**. Washington, DC, USA: IEEE Computer Society, 2006. (ISOLA '06), p. 212–219. ISBN 978-0-7695-3071-0.

SHARCNET. **SHARCNET cluster website**. Jul 2012. Available from Internet: <https://www.sharcnet.ca>.

SHERWOOD, T.; CALDER, B.; EMER, J. Reducing cache misses using hardware and software page placement. In: **Proceedings of the 13th International Conference on Supercomputing**. New York, NY, USA: ACM, 1999. (ICS '99), p. 155–164. ISBN 1-58113-164-X.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 8th. ed. [S.l.]: Wiley Publishing, 2008. ISBN 0470128720.

SINGH, K.; BHADAURIA, M.; MCKEE, S. A. Real time power estimation and thread scheduling via performance counters. **SIGARCH Computer Architecture News**, ACM, New York, NY, USA, v. 37, n. 2, p. 46–55, 2009. ISSN 0163-5964.

SONG, T. **Cache Coherence Protocol**. Feb 2013. Parallel Architectures and Algorithms Course. Washington University in St. Louis. Available from Internet: <http://research.engineering.wustl.edu/˜-songtian/pdf/20130219-CoherenceProtocol.pdf>.

SORIN, D. J.; HILL, M. D.; WOOD, D. A. **A Primer on Memory Consistency and Cache Coherence**. 1st. ed. [S.l.]: Morgan & Claypool Publishers, 2011. ISBN 1608455645, 9781608455645.

SPRUNT, B. Pentium 4 performance-monitoring features. **IEEE Micro**, v. 22, n. 4, p. 72–82, Jul/Aug 2002. ISSN 0272-1732.

SRIKANTAIAH, S.; KANDEMIR, M.; IRWIN, M. J. Adaptive set pinning: managing shared caches in chip multiprocessors. In: **Proceedings of the 13th international conference on Architectural support for programming languages and operating systems**. [S.l.]: ACM, 2008. (ASPLOS XIII), p. 135–144. ISBN 978-1-59593-958-6.

SRINIVASAN, A. **Efficient and Flexible Fair Scheduling of Real-time Tasks on Multiprocessors**. 200 p. Thesis (PhD in Computer Science) — Department of Computer Science – University of North Carolina at Chapel Hill, 2003.

SRINIVASAN, A.; HOLMAN, P.; ANDERSON, J. H.; BARUAH, S. The case for fair multiprocessor scheduling. In: **IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing**. Washington, DC, USA: IEEE Computer Society, 2003. p. 114.1. ISBN 0-7695-1926-1.

SRINIVASAN, B.; PATHER, S.; HILL, R.; ANSARI, F.; NIEHAUS, D. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In: **Proceedings of the 4th Real-Time Technology and Applications Symposium**. [S.l.: s.n.], 1998. (RTAS '98), p. 112–119.

STANKOVIC, J.; RAJKUMAR, R. Real-time operating systems. **Real-Time Systems**, Kluwer Academic Publishers, v. 28, n. 2-3, p. 237–253, 2004. ISSN 0922-6443.

STANKOVIC, J.; RAMAMRITHAM, K. The spring kernel: a new paradigm for real-time systems. **IEEE Software**, v. 8, n. 3, p. 62–72, 1991. ISSN 0740-7459.

STARKE, R. A.; OLIVEIRA, R. S. de. Cache-aware task partitioning for multicore real-time systems. In: **Proceedings of the 3rd Brazilian Symposium on Computing System Engineering**. [S.l.: s.n.], 2013. (SBESC '13), p. 1–5. ISSN 2324-7886.

STäRNER, J.; ASPLUND, L. Measuring the cache interference cost in preemptive real-time systems. In: **Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems**. New York, NY, USA: ACM, 2004. (LCTES '04), p. 146–154. ISBN 1-58113-806-7.

STASCHULAT, J.; ERNST, R. Scalable precision cache analysis for preemptive scheduling. In: **Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems**. New York, NY, USA: ACM, 2005. (LCTES '05), p. 157–165. ISBN 1-59593-018-3.

SUHENDRA, V.; MITRA, T. Exploring locking & partitioning for predictable shared caches on multi-cores. In: **Proceedings of the 45th annual Design Automation Conference**. [S.l.]: ACM, 2008. (DAC' 08), p. 300–303. ISBN 978-1-60558-115-6.

SUN, X. H.; WANG, J.; CHEN, X. An improvement of TLSF algorithm. In: **Proceedings of the 15th IEEE-NPSS Real-Time Conference**. [S.l.: s.n.], 2007. p. 1–5.

SUNDARAM, V.; CHANDRA, A.; GOYAL, P.; SHENOY, P.; SAHNI, J.; VIN, H. Application performance in the QLinux multimedia operating system. In: **Proceedings of the Eighth ACM International Conference on Multimedia**. New York, NY, USA: ACM, 2000. (MULTIMEDIA '00), p. 127–136. ISBN 1-58113-198-4.

SUNDARARAJAN, K.; JONES, T.; TOPHAM, N. RECAP: Region-aware cache partitioning. In: **Proceedings of the 31st IEEE International Conference on Computer Design**. [S.l.: s.n.], 2013. (ICCD '13), p. 294–301.

SUZUKI, N.; KIM, H.; NIZ, D. d.; ANDERSSON, B.; WRAGE, L.; KLEIN, M.; RAJKUMAR, R. R. Coordinated bank and cache coloring for temporal protection of memory accesses. In: **Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering**. [S.l.]: IEEE, 2013. (CSE '13), p. 685–692. ISBN 978-0-7695-5096-1.

TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. **SIGOPS Operating System Review**, ACM, New York, NY, USA, v. 41, p. 47–58, March 2007. ISSN 0163-5980.

TAM, D. K.; AZIMI, R.; SOARES, L. B.; STUMM, M. RapidMRC: approximating l2 miss rate curves on commodity systems for online optimizations. In: **Proceeding of the 14th international conference on Architectural support for programming languages and operating systems**. New York, NY, USA: ACM, 2009. (ASPLOS '09), p. 121–132. ISBN 978-1-60558-406-5.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

TAYLOR, G.; DAVIES, P.; FARMWALD, M. The TLB slice - a low-cost high-speed address translation mechanism. In: **Proceedings of the 17th Annual International Symposium on Computer Architecture**. New York, NY, USA: ACM, 1990. (ISCA '90), p. 355–363. ISBN 0-89791-366-3.

TSAFRIR, D. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In: **Proceedings of the 2007 workshop on Experimental computer science**. New York, NY, USA: ACM, 2007. (ExpCS '07). ISBN 978-1-59593-751-3.

UBM, T. **2013 Embedded Market Study**. 2013. Available from Internet: <http://e.ubmelectronics.com/2013EmbeddedStudy/index.html>.

VARADARAJAN, K.; NANDY, S. K.; SHARDA, V.; BHARADWAJ, A.; IYER, R.; MAKINENI, S.; NEWELL, D. Molecular caches: A caching structure for dynamic creation of application-specific heterogeneous cache regions. In: **Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2006. (MICRO-39), p. 433–442. ISSN 1072-4451.

VERA, X.; LISPER, B.; XUE, J. Data cache locking for higher program predictability. In: **Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems**. New York, NY, USA: ACM, 2003. (SIGMETRICS '03), p. 272–282. ISBN 1-58113-664-1.

VERA, X.; LISPER, B.; XUE, J. Data caches in multitasking hard real-time systems. In: **Proceedings of the 24th Real-Time Systems**

**Symposium**. [S.l.]: IEEE, 2003. (RTSS ’03), p. 154–165. ISBN 0-7695-2044-8.

WANG, Y.-C.; LIN, K.-J. Enhancing the real-time capability of the linux kernel. In: **Proceedings of the 5th Real-Time Computing Systems and Applications**. [S.l.: s.n.], 1998. (RTCSA ’98), p. 11–20.

WANNER, L. F.; FRÖHLICH, A. A. Operating System Support for Wireless Sensor Networks. **Journal of Computer Science**, v. 4, n. 4, p. 272–281, 2008. ISSN 1549-3636.

WEHMEYER, L.; MARWEDEL, P. Influence of memory hierarchies on predictability for time constrained embedded software. In: **Proceedings of the conference on Design, Automation and Test in Europe**. Washington, DC, USA: IEEE Computer Society, 2005. (DATE ’05), p. 600–605. ISBN 0-7695-2288-2.

WEISSMAN, B. Performance counters and state sharing annotations: a unified approach to thread locality. **SIGOPS Operating System Review**, ACM, New York, NY, USA, v. 32, p. 127–138, October 1998. ISSN 0163-5980.

WEST, R.; ZAROO, P.; WALDSPURGER, C. A.; ZHANG, X. Online cache modeling for commodity multicore processors. **SIGOPS Operating System Review**, ACM, New York, NY, USA, v. 44, p. 19–29, December 2010. ISSN 0163-5980.

WIKIPEDIA. **List of real-time operating systems**. 2014. Available from Internet: <http://en.wikipedia.org/wiki/List_of_real-time_operating_systems>.

WIKIPEDIA. **MESI protocol**. Feb 2014. Available from Internet: <http://en.wikipedia.org/wiki/MESI_protocol>.

WIKIPEDIA. **MESIF protocol**. Feb 2014. Available from Internet: <http://en.wikipedia.org/wiki/MESIF_protocol>.

WIKIPEDIA. **MOESI protocol**. Feb 2014. Available from Internet: <http://en.wikipedia.org/wiki/MOESI_protocol>.

WILHELM, R.; ENGBLOM, J.; ERMEDAHL, A.; HOLSTI, N.; THESING, S.; WHALLEY, D.; BERNAT, G.; FERDINAND, C.;

HECKMANN, R.; MITRA, T.; MUELLER, F.; PUAUT, I.; PUSCHNER, P.; STASCHULAT, J.; STENSTRöM, P. The worst-case execution-time problem overview of methods and survey of tools. **ACM Transactions on Embedded Computing Systems**, ACM, New York, NY, USA, v. 7, n. 3, p. 36:1–36:53, May 2008. ISSN 1539-9087.

Wind River. **Wind River VcWorks RTOS**. 2013. Available from Internet: <http://www.windriver.com/products/vxworks/>.

WOLFE, A. Software-based cache partitioning for real-time applications. **Journal of Computing Software Engineering**, Ablex Publishing Corp., Norwood, NJ, USA, v. 2, n. 3, p. 315–327, Mar 1994. ISSN 1069-5451.

YAN, J.; ZHANG, W. WCET analysis for multi-core processors with shared L2 instruction caches. In: **Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium**. Washington, DC, USA: IEEE Computer Society, 2008. (RTAS '08), p. 80–89. ISBN 978-0-7695-3146-5.

ZHANG, X.; DWARKADAS, S.; SHEN, K. Towards practical page coloring-based multicore cache management. In: **Proceedings of the 4th ACM European Conference on Computer Systems**. [S.l.]: ACM, 2009. (EuroSys '09), p. 89–102. ISBN 978-1-60558-482-9.

ZHURAVLEV, S.; BLAGODUROV, S.; FEDOROVA, A. Addressing shared resource contention in multicore processors via scheduling. In: **Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems**. [S.l.: s.n.], 2010. (ASPLOS '10), p. 129–142. ISBN 978-1-60558-839-1.

ZHURAVLEV, S.; SAEZ, J. C.; BLAGODUROV, S.; FEDOROVA, A.; PRIETO, M. Survey of scheduling techniques for addressing shared resources in multicore processors. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 45, n. 1, p. 4:1–4:28, Dec 2012. ISSN 0360-0300.