UNIVERSIDADE FEDERAL DE SANTA CATARINA

HEURÍSTICA PARA ESCALONAMENTO MULTICORE DE TEMPO REAL VISANDO DETERMINISMO TEMPORAL

José Luis Conradi Hoffmann

Florianópolis - SC

2018 / 2

UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

HEURÍSTICA PARA ESCALONAMENTO MULTICORE DE TEMPO REAL VISANDO DETERMINISMO TEMPORAL

José Luis Conradi Hoffmann

Trabalho De Conclusão De Curso apresentado na Universidade Federal de Santa Catarina (UFSC) como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Florianópolis – SC

2018 / 2

José Luis Conradi Hoffmann

HEURÍSTICA PARA ESCALONAMENTO MULTICORE DE TEMPO REAL VISANDO DETERMINISMO TEMPORAL

Trabalho De Conclusão De Curso apresentado na Universidade Federal de Santa Catarina (UFSC) como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Prof. Dr. Antônio Augusto Fröhlich

Orientador

INE / UFSC

Prof. Dr. Giovani Gracioli

Coorientador

ICPSPE / TUM

Banca Examinadora

Dr. Tiago Rogério Mück

ARM

Prof. Dr. Lucas Francisco Wanner

IDC / UniCamp

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE REDUÇÕES

MSR	Model Specific Register
PMU	Performance Monitoring Unit
PMC	Performance Monitoring Counter
EPOS	Embedded Parallel Operating System
IOT	Internet Of Things
CPU	Central Process Unit
DVFS	Dynamic Voltage and Frequency Scaling
RAPL	Running Average Power Limit
PEBS	Processor Event Based Sampling
SO	Sistema Operacional
CSV	Comma Separated Values

RESUMO

Com a evolução dos chips processadores, busca-se cada vez mais otimizar seu desempenho e consumo energético. Tal otimização, vem crescendo tanto pelo Hardware, através do aumento da quantidade de recursos disponíveis no chip, quanto pelo Software, onde se busca utilizar da melhor maneira possível esses recursos. Dentro do contexto de monitoramento de performance, várias pesquisas vêm surgindo com o objetivo de aprender sobre a execução através de canais de monitoramento providos pela Unidade de Monitoramento de Performance (PMU) do processador. Ao adquirir conhecimento sobre as características da execução atual, é possível influenciar nas decisões tomadas durante o escalonamento das tarefas. Neste trabalho, o foco é desenvolver uma heurística utilizando a contagem de eventos de performance, através da PMU. Sendo os dados utilizados para controlar o escalonamento dinâmico de voltagem e frequência (DVFS), onde a heurística desenvolvida consegue executar tarefas críticas relacionadas com um alto uso da hierarquia de memória e/ou com um alto uso de recursão, sem perder a característica de determinismo temporal, ao mesmo tempo que reduz o consumo dos núcleos processador em até 15%.

Palavras chave: Multi-core, Tempo Real, Heurística, DVFS, PMU, MSR, IA-32, Sensores de Hardware, Determinismo Temporal, *Energy Aware*.

1. INTRODUÇÃO

O conceito de determinismo temporal na computação ganhou grande importância com o advento de sistemas de tempo real, onde ao criar um sistema que deve executar não somente buscando a corretude lógica da execução, mas também a corretude temporal, foi imposta a necessidade de que os momentos de início e fim determinados para a execução de uma tarefa fossem respeitados [14]. Em alguns sistemas de tempo real, como por exemplo um *player* de vídeo, a falta de sincronia entre os *frames* pode ser incômoda, mas não chega a ser algo crítico para o sistema, em um outro caso, como por exemplo o controle de um carro, se o freio não responder no momento correto pode ocasionar uma fatalidade, sendo, este último, caracterizado como um sistema crítico. Esse e muitos outros exemplos de *Deadlines* que não foram respeitadas em um sistema crítico, que motivam pesquisas na comunidade de computação para garantir o determinismo temporal, ao mesmo tempo em que busca-se otimizar tanto o desempenho, quanto o consumo energético.

Os processadores, ao longo do tempo, foram cada vez mais sendo projetados para prover informações sobre a execução, começando com simples contadores, como Instruções finalizadas, até uma gama de mais de 200 tipos de eventos nas implementações mais atuais [4]. O uso de uma Unidade de Monitoramento de Performance (PMU), e de sensores de hardware, para obter dados sobre o comportamento dos Cores, a fim de utilizá-los para entender e até mesmo tentar predizer os acontecimentos dentro de uma CPU, é uma área que vem crescendo nos últimos anos. Sendo possível estimar o desempenho e o gasto energético de uma carga de trabalho

dinâmica em tempo de execução através dos contadores de hardware disponíveis na arquitetura, como demonstrado em Run-DMC [16].

Um outro exemplo é o sistema operacional SPARTA [15], que busca melhorar a eficiência energética e o desempenho de processadores manycores heterogêneos, através de uma caracterização em tempo de execução das tarefas, para então atribuir prioridades na alocação de recursos. O sistema SPARTA obteve uma redução do consumo energético de até 23%, o que gerou uma grande motivação para este trabalho.

Pesquisas atuais mostram que um bom controle do uso da hierarquia de cache da máquina, em sistemas multi-core de tempo real, diminui atrasos em *Deadlines* [19], o que afeta diretamente o determinismo temporal do sistema. E também, através de contadores de hardware, é possível detectar problemas de coerência de cache [20], o que pode aumentar a corretude do sistema, se tratados devidamente.

A ideia que deu início a este TCC foi a de utilizar um sistema de baixa interferência (EPOS) para realizar várias vezes uma mesma execução, onde em cada uma delas, um novo conjunto de eventos é utilizado na PMU. Isso abre a possibilidade de agrupar todos os eventos como se fossem capturados numa única execução, que nas versões atuais dos processadores é impossível devido a limitação de registradores em hardware para a unidade de monitoramento. Com esses dados agrupados, é possível a aplicação de técnicas de Data Mining, como algoritmos de Machine Learning (e.g. PCA - *Principal Component Analysis* [18]), para então verificar correlações entre as variáveis capturadas, buscando um grupo que possa ser utilizado durante a execução descrevendo um comportamento do sistema que deseja-se monitorar. Esse processo pode ser realizado para vários conjuntos de tarefas distintos, fazendo então um refinamento das correlações.

O foco é de encontrar um conjunto de eventos correlacionado com perdas de deadline quando o sistema está afetado por DVFS, com isso, utilizar este conjunto de eventos para predizer uma perda de deadline para que então, seja controlado de forma preventiva o uso da técnica DVFS, visando diminuir o consumo e manter a corretude temporal, ou seja, tratando o problema antes que ele ocorra através da análise do conjunto de variáveis estabelecido.

1.1. OBJETIVOS

1.1.1. OBJETIVOS GERAIS

Utilizando o sistema operacional EPOS [1], desenvolvido pelo LISHA [2] (Laboratório de integração Software/Hardware) funcionando em uma arquitetura Intel IA-32 em um servidor fornecido pelo laboratório, coletar dados, de contadores de eventos e termais, durante a execução do sistema, a fim de aplicar algoritmos de Machine Learning, tais como PCA (*Principal Component Analysis*), com o objetivo de encontrar correlações entre os dados que descrevem eventos durante a execução. Com estas correlações estabelecidas, desenvolver uma heurística que diminua o consumo de energia do processador, através de *Dynamic Voltage and Frequency Scaling* (DVFS) mantendo a característica de determinismo temporal do sistema.

Vale citar também, que a heurística tem enfoque no sistema de escalonamento PEDF, pois dessa forma podemos tomar decisões mais precisas sobre o comportamento de um núcleo, visto que o mesmo mantém as mesmas tarefas ao longo do tempo.

1.1.2. OBJETIVOS ESPECÍFICOS

- Desenvolver, em conjunto com o discente Leonardo Passig Horstmann (15103030), um sistema não intrusivo de captura de dados de monitoramento de performance em tempo de execução, além de um sistema de envio para o banco de dados inteligentes de IoT Lisha [5].
- Busca de correlação dos dados capturados, através de algoritmos de Machine Learning, que consigam predizer eventos de perda de deadlines durante a execução de um determinado conjunto de tarefas.
- Desenvolver uma heurística de escalonamento, através das correlações encontradas, que permita um melhor uso energético ao longo da execução e que mantenha o determinismo temporal do sistema.

2. FUNDAMENTAÇÃO TEÓRICA E MATERIAIS UTILIZADOS

2.1. INTRODUÇÃO

Neste capítulo serão apresentados os principais conceitos necessários para o entendimento de sistemas multicore de tempo real, focando no sistema EPOS e Intel IA-32, sistemas não intrusivos, Data Mining, determinismo temporal, DVFS, e os materiais utilizados no desenvolvimento deste trabalho.

2.2. COMPUTAÇÃO MULTICORE

Um Chip Multi-Processador (CMP) usa processadores *single-thread* relativamente simples para explorar apenas quantidades moderadas de paralelismo dentro de uma thread qualquer, enquanto executa várias threads em paralelo através de múltiplos núcleos de processamento [9].

Outro tipo de sistema multicore é o processador *multithread* simultânea (SMT), também conhecidos como processadores *HyperThread* [10], estes por sua vez, se assemelham muito com processadores com múltiplos cores, mas eles implementam cada núcleo virtual com uma combinação de unidades funcionais dentro de um processador tradicional.

Sistemas Operacionais atuais, prestam suporte a esse tipo de tecnologia, além de prover ao usuário uma maneira relativamente simples para gerenciar os recursos compartilhados por tarefas executadas paralelamente, assim garantindo um alto poder computacional ao mesmo tempo que reduz a necessidade de uma velocidade de processamento muito alta, o que implica na redução do consumo de energia necessária.

2.3. SISTEMAS DE TEMPO REAL

Em computação de tempo real, a corretude do sistema não depende apenas do resultado lógico da computação, mas também no tempo em que esse resultado é produzido [11].

Esta limitação temporal existente nos sistema de tempo real é chamada de *deadline,* ou seja, um prazo limite para a entrega. Porém, a computação de tempo real não é simplesmente uma computação rápida, mas sim obedecer os requisitos de tempo individual de cada tarefa, o sistema deve ser previsível [11]. Outras variáveis importantes em um sistema de tempo real são Tempo de liberação (Release Time), WCET (Worst Case Execution Time), ou seja, o maior tempo necessário para concluir a tarefa, e Tempo de execução (AET – Actual Execution Time), o tempo que a tarefa leva para executar.

Um sistema de tempo real pode ser definido como *Hard* (crítico) ou *Soft* (não crítico):

- Em um sistema Hard, uma perda de prazo (*deadline*) pode ocasionar em um grande impacto em seu contexto, como por exemplo o freio de um carro demorar para responder em um momento crítico, logo ela DEVE ser atendida no tempo correto, nem antes e nem depois do que foi definido [12].
- Em um sistema Soft, por sua vez, uma perda de prazo não tem um impacto tão grande [12].

Uma outra característica importante é que para atender um sistema de tempo real, os escalonadores das tarefas devem prestar suporte as variáveis citadas acima, garantindo a corretude da execução. Escalonadores de tempo real trabalham,

normalmente, atribuindo prioridades às tarefas, estas, por sua vez, podem ser dinâmicas ou estáticas.

2.3.1. ESCALONAMENTO EDF – PRIORIDADE DINÂMICA

EDF é um algoritmo de escalonamento de prioridade dinâmica, e como o nome sugere, quanto mais próximo de encontrar seu prazo de execução, maior a prioridade de uma thread, ou seja, maior a chance de que ela seja escolhida para entrar em execução.

O algoritmo de escalonamento EDF tem um limite de 100% de utilização, se esse limite for ultrapassado, perdas de prazos irão ocorrer no sistema [12]. Para sistemas *Muilticore*, o EDF possui 3 (três) versões, que adaptam a fila de escalonamento, sendo elas:

- Global (GEDF): A fila de escalonamento do EDF se mantém única, porém com uma “cabeça” de fila para cada núcleo do processador.
    - É caracterizado por explorar melhor o paralelismo, por não limitar qual núcleo cada thread pode rodar, porém, a troca de contexto pode gerar mais atraso (*overhead*).
- Cluster (CEDF): A fila de escalonamento do EDF é dividida em clusters de núcleos, ou seja, existe uma fila para cada cluster e uma “cabeça” de fila para cada núcleo do cluster.
    - É caracterizado por diminuir um pouco a exploração do paralelismo se a divisão de tarefas for infeliz, mas diminui o atraso gerado pela troca de contexto (overhead).

- Particionado (PEDF): A fila de escalonamento do EDF é dividia em uma por núcleo, ou seja, aplica um EDF para cada núcleo.
    - É caracterizado por gerar o mínimo de interferência na troca de contexto, mas explora pouco o paralelismo se a divisão de tarefas for infeliz.

2.4. EPOS

EPOS [1], ou Sistema Operacional Paralelo Embarcado (*Embedded Parallel Operating System*), consiste de um sistema operacional desenvolvido no Laboratório de Integração Software/Hardware - LISHA [2]. O EPOS conta com o Método de Design de Sistemas Embarcados Dirigido - Aplicação (ADESD) proposto pelo professor Antônio Augusto Fröhlich para projetar e implementar componentes de software e hardware que podem ser adaptados automaticamente para atender aos requisitos de aplicações específicas [3].

Um grande ganho ao se utilizar o EPOS, é de que o sistema apresenta uma baixíssima quantidade de interferência, o baixo overhead e alto desempenho são alcançados por uma implementação cuidadosa que faz uso de técnicas de programação generativa, incluindo a metaprogramação estática [3].

O sistema operacional EPOS pode ser configurado para executar tanto em um processador Intel, quanto em um processador ARM. Neste trabalho o foco será em sua versão para Intel IA-32, a qual presta suporte aos recursos aqui necessários.

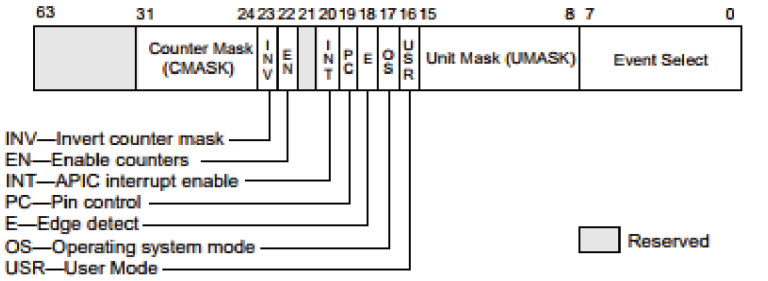
2.5. INTEL IA-32 E PMU

Intel Corporation (ou apenas Intel) (https://www.intel.com.br) é uma empresa multinacional desenvolvedora de chips semicondutores, e uma das principais

desenvolvedores de processadores no mundo. No presente trabalho, foi utilizado para o seu desenvolvimento, um processador intel da arquitetura IA-32, mais especificamente, da arquitetura Sandy Bridge.

A arquitetura Intel IA-32 apresenta vários recursos para monitorar o desempenho do sistema. Entre os principais está a *Performance Monitoring Unit* (PMU), um conjunto de vários registradores (MSR's), que permitem a configuração de contadores (PMC's) para monitorar e medir eventos, como instruções finalizadas e *Cache Misses*, que ocorrem durante a execução do sistema [4].

Figura 2.1 – Exemplo de *Layout* de um MSR.



Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual (2018)

2.5.1. VERSÕES

Com o lançamento de novas arquiteturas (Core Solo e Duo, Core 2 Duo, Atom, Skylake, Kabylake, etc.), várias versões da PMU também foram desenvolvidas. Dentre os aprimoramentos, vale citar o aumento da gama de eventos monitoráveis e também a quantidade de MSR's disponíveis para monitorar paralelamente.

A versão mais atual da PMU é a versão arquitetural 4, sendo que cada distribuição possui caraterísticas não arquiteturais, ou seja, eventos de uma distribuição não

necessariamente estarão contidos em outra distribuição mesmo que ambas se baseiam na mesma versão arquitetural.

O EPOS presta suporte para as versões arquiteturais 1, 2 e 3, e para a distribuição Sandy Bridge (versão arquitetural 3) [4]. No desenvolvimento deste trabalho, foi utilizado a versão Sandy Bridge, porém o sistema pode ser facilmente portado para as outras versões.

Durante uma execução, podemos capturar apenas 4 (quatro) canais configuráveis e 3 (três) canais fixos por thread na versão da PMU presente no Sandy Bridge, limitados pela quantidade de registradores físicos. Canais fixos sempre monitoram os mesmos eventos, sendo eles *Instructions Retired, CPU Clock Unhalted e CPU Clock Unhalted Ref-TSC* [4], enquanto os canais configuráveis monitoram qualquer outro evento disponível na versão utilizada. Uma lista contendo todos os eventos cobertos pela implementação da versão Sandy Bridge no EPOS pode ser encontrada no Anexo “A1 Tabela de Eventos Intel Sandy Bridge no EPOS“.

2.6. NÃO INTRUSIVIDADE

Um comportamento não intrusivo é quando o comportamento comum de um usuário (ou sistema) não é afetado. ou que tenha um impacto mínimo, para realizar as operações desejadas [21].

No escopo deste trabalho, o monitoramento não pode afetar os resultados do sistema, ou seja, uma execução com e sem o sistema de monitoramento habilitado deve ser muito similar (sendo a diferença algo que possa ser desconsiderado), para que os dados de capturados durante a execução não sejam afetados.

2.7. DATA MINING

Data Mining é um processo iterativo, com métodos manuais ou automáticos, sobre um grande conjunto de dados. Estes algoritmos são caracterizados por extrair conhecimento útil “escondido” dentro desses dados, o qual é muito útil quando não existe nenhuma noção predeterminada sobre o que seria um bom resultado [17].

Esse conhecimento pode ser um padrão ou uma correlação entre as variáveis do conjunto de dados. Para se obter um melhor resultado, deve se encontrar um equilíbrio com o conhecimento de um especialista de domínio e os resultados obtidos, para assim melhorar os parâmetros de cada iteração [17].

2.7.1. PCA - PRINCIPAL COMPONENT ANALYSIS

PCA (*Principal Component Analysis*) consiste de um algoritmo de Data Mining, que visa encontrar padrões dentro de um conjunto de dados. O ponto inicial da análise é uma matriz de dados que consiste de N linhas e K colunas, onde cada linha N corresponde a um objeto (ou observação) e cada coluna K corresponde a uma variável da captura [18].

2.7.2. WEKA 3

Weka é um software para tarefas de Data Mining desenvolvido pelo grupo de Machine Learnign da universidade de Waikato (https://www.cs.waikato.ac.nz/ml/weka/index.html), na linguagem JAVA. Consiste de uma coleção de algoritmos de Machine Learning que podem ser aplicados diretamente a um conjunto de dados. Weka será utilizado para aplicar o algoritmo PCA nos dados coletados durante a execução deste TCC.

2.8. DETERMINISMO TEMPORAL

O conceito de determinismo temporal baseia-se em assumir que o comportamento do sistema pode ser determinístico, ou seja, que ele seja previsível ao longo do tempo. No caso de sistemas computacionais multicore de tempo real, o determinismo temporal significa que as deadlines previamente estabelecidas devem ser respeitadas durante a execução, para isso, assume-se que as latências do sistema são constantes ou insignificantes.

Porém, garantir essas propriedades raramente é possível ou é muito custoso. Em sistemas multithread, existem interferências desde o uso das caches até uma preempção de troca de contexto, o que acaba gerando uma interferência total que não é insignificante [14].

2.9. DYNAMIC VOLTAGE AND FREQUENCY SCALING - DVFS

DVFS, ou Escalonamento Dinâmico de Voltagem e Frequência, consiste de uma técnica utilizada para diminuir a potência e consumo de microprocessadores, por meio da redução da frequência e/ou voltagem de operação do mesmo [24]. A técnica apresenta redução no consumo, pois processadores com alta dissipação de potência tendem a esquentar, o que aumenta a taxa de falha e aumenta a complexidade do sistema de resfriamento.

Porém, em sistemas de tempo real, aplicar o DVFS pode prejudicar o desempenho, o que por sua vez, tem uma chance muito alta de causar atrasos nos prazos das tarefas, ou seja, se aplicado sem nenhum cuidado, o sistema tende a perder sua corretude temporal.

Atualmente, já existem técnicas para a aplicação de DVFS em sistemas de tempo real, Uma das técnicas mais conhecidas, definida por Pilai e Shin [25], é o Escalonamento de Voltagem Estático (Static Voltage Scaling), no qual é selecionada a mais baixa frequência de operação para a qual seja possível cumprir todas as deadlines no escalonador selecionado. A técnica então consiste em recalcular o WCET das tarefas baseado na frequência que se deseja aplicar, dado que seu período e deadline são mantidos. Feito isso, basta o teste de escalabilidade do algoritmo selecionado.

Uma outra abordagem proposta em [25] é o algoritmo Cycle-conserving RT-DVS, este algoritmo, por sua vez, tem como objetivo usar a técnica de DVFS de modo a diminuir a frequência quando as tarefas usam menos que o WCET para executar, e aumentá-la quando ocorre a necessidade do pior caso.

Por fim, um outro exemplo seria a técnica Look-Ahead RT-DVS, também descrita em [25], sendo está uma abordagem considerada mais agressiva, pois busca obter o máximo de economia de energia. Sendo no início da execução a frequência configurada para o mínimo necessário, mas vai aumentando ao longo da execução, sempre que necessário.

2.10. MATERIAIS UTILIZADOS

Para realização deste trabalho foi utilizado um computador rodando o sistema operacional EPOS na sua versão 2.1. As configurações do computador relevantes para este trabalho são:

- Processador: Intel Core i7-2600 3.40 GHz;
- Memória RAM: 4GB 1333 MHz;
- Placa Mãe: PCWARE IPMH61R3

- Fonte: ALLIED SL-8180 BTX
- Gabinete: CPU 65L3900

O processador presente na máquina provém da segunda geração dos processadores Intel Core i7, isto significa que a versão da PMU presente na máquina é relativa a versão arquitetural 3, mais especificamente a versão Sandy Bridge.

Para realização das análises de consumo, foi utilizado o analisador de qualidade de energia e potência Fluke 435 Series-II [22], medindo o consumo direto da fonte do computador. A configuração utilizada foi de 3 alicates, sendo um em cada fio do cabo de alimentação (Fase, Neutro e Terra) em uma das pontas do cabo, e na outra ponta, dois alicates i430flex, sendo um no Neutro e outro no Fase. Já sobre as medições, foi utilizado a função *Logger,* a qual realiza uma captura a cada 250 milissegundos, com adição dos medidores de consumo energético.

Além do analisador Fluke, também foi utilizado os contadores da interface *RAPL* da intel (descrito no tópico “3.3. Leitura de Consumo Energético”), internos ao próprio processador, fornecendo assim uma visão do consumo isolado.

3. IMPLEMENTAÇÃO

Neste capítulo, é descrita a implementação do sistema não intrusivo de captura de dados em tempo de execução, dentro e fora do sistema EPOS. Esta etapa foi desenvolvida em conjunto com o discente, também da UFSC, Leonardo Passig Horstmann (15103030).

Uma descrição mais detalhada da implementação, contendo parte do estudo feito e algumas conclusões que levaram a mudanças e evoluções que trouxeram o sistema até o estado atual, podem ser encontradas em *"Performance Monitoring with EPOS"* [7].

3.1. LEITURA DE TEMPERATURA

De acordo com o manual de desenvolvedores da Intel [4], na arquitetura IA-32, a temperatura de um processador pode ser lida usando o MSR IA32_THERM_STATUS. No tópico 14.7.5.2 "Reading the Digital Sensor" (2018, vol 3B, capítulo 14, p. 2697), foi definido o seguinte método para leitura do valor da temperatura:

Primeiramente, é preciso lembrar que, diferente de um leitor analógico de temperatura, a saída do sensor térmico digital é uma temperatura relativa a máxima temperatura permitida para operação do processador:

- Digital Readout (bits 22:16, RO) - Leitura da temperatura digital em 1 grau Celsius relativos ao temperatura de ativação do TCC (Thermal Control Circuitry - circuito de controle térmico).
    - 0: Temperatura de ativação do TCC;
    - 1: (Ativação do TCC - 1).

Neste sentido, a leitura de um valor baixo do campo Digital Readout (bits 22:16) indica uma temperatura, que na verdade, é alta. Então, para calcular a temperatura atual do processador é necessário identificar o TJunction, ou seja, a temperatura em que é ativado o Thermal Throttling, que pode ser adquirido realizando uma leitura no MSR IA32_TEMPERATURE_TARGET (Endereço 0x1a2). Logo, a temperatura atual é:

- IA32_TEMPERATURE_TARGET (bits 23:16) - IA32_THERM_STATUS (bits 22:16);

3.2. MODULAÇÃO DE CLOCK

De acordo com o Manual de Desenvolvimento de Software da Intel [4], modulação de clock é um método de controle térmico disponível para o processador. A modulação de Clock é executada ligando e desligando rapidamente os relógios em um duty cycle que deve reduzir a dissipação de energia em cerca de 50%. No tópico 14.7.3 “Software Controlled Clock Modulation” (2018, vol. 3B, cap. 14, p. 3179), o MSR IA32_CLOCK_MODULATION, provê meios para o sistema operacional implementar uma política de gerenciamento de energia para reduzir o consumo do processador.

Segundo a definição da Intel [4, seção “14.7 THERMAL MONITORING AND PROTECTION”, página 3177], “duty cycle" não se refere ao ciclo de trabalho real do sinal de clock, e sim ao período de tempo durante o qual o sinal de clock pode acionar o chip do processador. Usando o mecanismo de parada do clock para controlar com que freqüência o processador é “cronometrado” (recebe os sinais do clock), o consumo de energia do processador pode ser modulado.

Figura 3.1 – *Layout* do MSR IA32_CLOCK_MODULATION

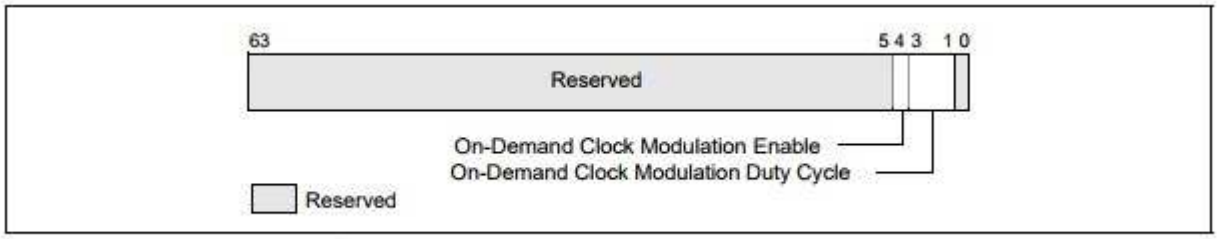


Figure 14-25. IA32_CLOCK_MODULATION MSR

Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual (2018)

- *On-Demand Clock Modulation Enable*, bit 4 — Habilita o controle de software de modulação de clock por demanda quando em 1 e o desabilita quando 0.
- *On-Demand Clock Modulation Duty Cycle*, bits 1 ao 3 — Seleciona o *duty cycle* da modulação de clock por demanda (ver Figura X). Esse campo só é válido quando o bit 4 (descrito acima) está habilitado.

Em um sistema com múltiplos núcleos físicos, cada núcleo pode modular seu *duty cycle* de forma independente. Isso traz ao EPOS, a possibilidade de executar operações de DVFS, quando executando na arquitetura Intel.

Figura 3.2 – Descrição dos *Duty Cycles*

Table 14-3. On-Demand Clock Modulation Duty Cycle Field Encoding

Duty Cycle Field Encoding	Duty Cycle
000B	Reserved
001B	12.5% (Default)
010B	25.0%
011B	37.5%
100B	50.0%
101B	63.5%
110B	75%
111B	87.5%

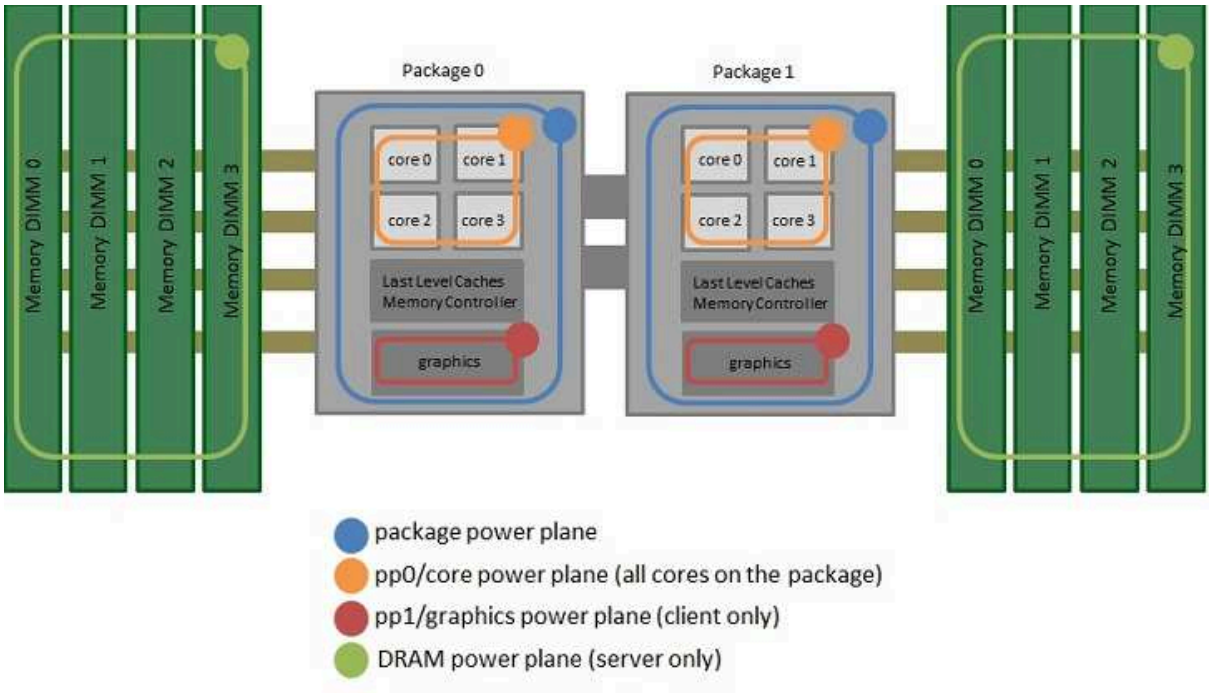
Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual (2018)

É importante ressaltar que em sistemas multicore, cada core controla seu duty cycle, porém quando se usa a tecnologia de Hyper-threading, o duty cycle do core físico será configurado para o valor mais alto (menor frequência) entre as threads.

3.3. LEITURA DE CONSUMO ENERGÉTICO

Mesmo com um medidor de consumo energético, é possível, através da interface RAPL da Intel [4], acompanhar o consumo de forma mais específica. A interface provê suporte para leitura do Consumo do Pacote (*PKG*), e das partes internas do mesmo, além do consumo energético da DRAM. Sendo as partes internas do *PKG* os núcleos de processamento (*PP0*) e dos núcleos de processamento de vídeo da *GPU* integrada (*PP1*).

Figura 3.3 – Descrição da divisão utilizada pela Interface RAPL.



Fonte:Intel® Power Governor (2012) [23].

Porém, o suporte a todas essas medidas muda de acordo com a versão da arquitetura do processador, logo, sendo a versão do processador presente na máquina de testes utilizada neste trabalho a versão Sandy Bridge, o suporte é apenas para informações de consumo no nível de PKG e PP0 (http://web.eece.maine.edu/~vweaver/projects/rapl/rapl_support.html).

Figura 3.4 – MSR da Interface RAPL que provém as informações da unidade de Energia medida, além da unidade de potência e tempo.

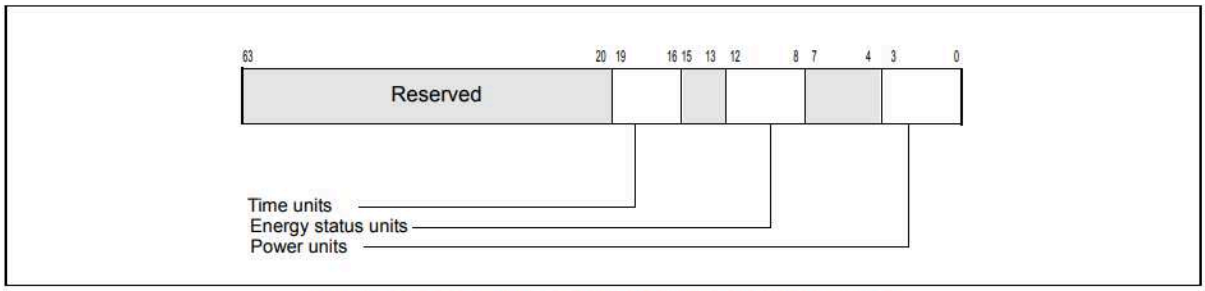


Figure 14-31. MSR_RAPL_POWER_UNIT Register

Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual (2018)

Figura 3.5 – MSR da Interface RAPL que contém a contagem de energia total consumida pelo PKG.

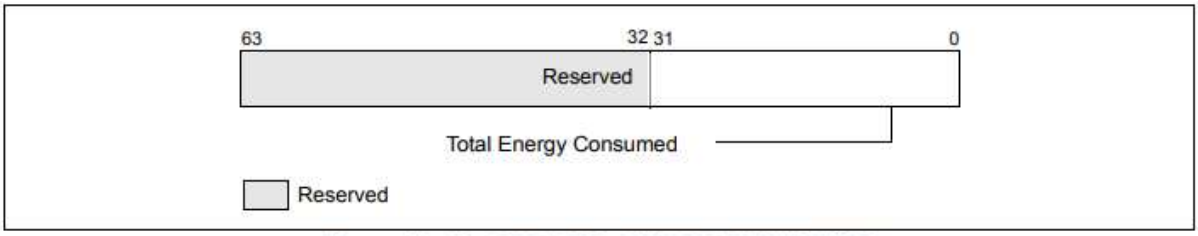


Figure 14-33. MSR_PKG_ENERGY_STATUS MSR

Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual (2018)

Figura 3.6 – MSR da Interface RAPL que contém a contagem da energia total consumida pelo PP0 ou pelo PP1.

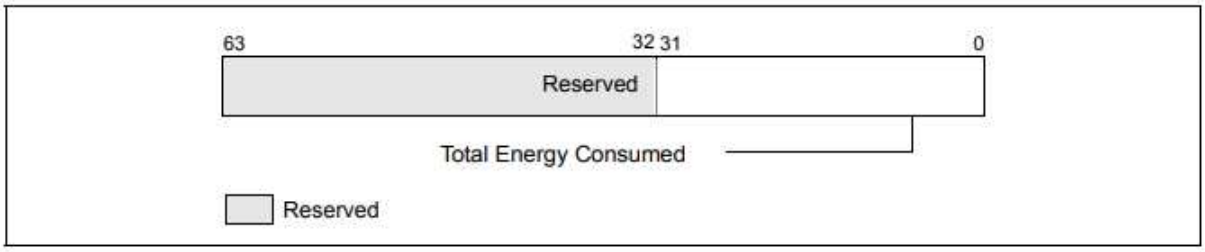


Figure 14-37. MSR_PP0_ENERGY_STATUS/MSR_PP1_ENERGY_STATUS MSR

Fonte: Intel® 64 and IA-32 Architectures Software Developer's Manual (2018)

Para se aferir consumo do PKG, deve-se ler o conteúdo presente nos 32 bits menos significativos do MSR MSR_PKG_ENERGY_STATUS (Figura 3.5) e dividir pela potência de 2 elevado ao valor lido no campo ENERGY STATUS UNIT (bits 8 ao 12) do MSR MSR_RAPL_POWER_UNIT (Figura 3.4), transformando o valor de energía consumida para a unidade Joules. Para obter o consumo em Watts, basta dividir o valor encontrado na operação pelo tempo de execução em segundos (Watts = Joules/s).

- Consumo em Joules =

  (ENERGY_STATUS1 - ENERGY_STATUS0) / ($2^{ENERGY\_STATUS\_UNIT}$);

- Consumo em Watts =

  Consumo em Joules / (Tempo1 - Tempo0); Sendo o tempo em segundos.

O processo para cálculo do consumo para PP0 e PP1 é similar, sendo necessário apenas substituir o MSR MSR_PKG_ENERGY_STATUS (Figura 3.5) pelo MSR_PPx_ENERGY_STATUS (Figura 3.6), onde x representa qual o PP desejado.

É importante observar que o contador apresentado possui limite de contagem de 32 bits, quando o mesmo ultrapassa o valor máximo, ele zera e reinicia a contagem, este momento é chamado pela Intel de wraparound (14.9.3 Package RAPL Domain - Intel sdm [4]). Logo, se uma execução for longa o suficiente, deve se tomar certas precauções para

os momentos em que este contador realiza o *reset*. A Intel prevê um tempo de 60s para o wraparound com uso alto, mas o tempo pode aumentar conforme o uso diminui.

No sistema utilizado aqui, as capturas acontecem ao longo da execução, e os cálculos acima apresentados são realizados apenas com o *log* da mesma. Desta forma, podemos também calcular um consumo médio e outra estatísticas.

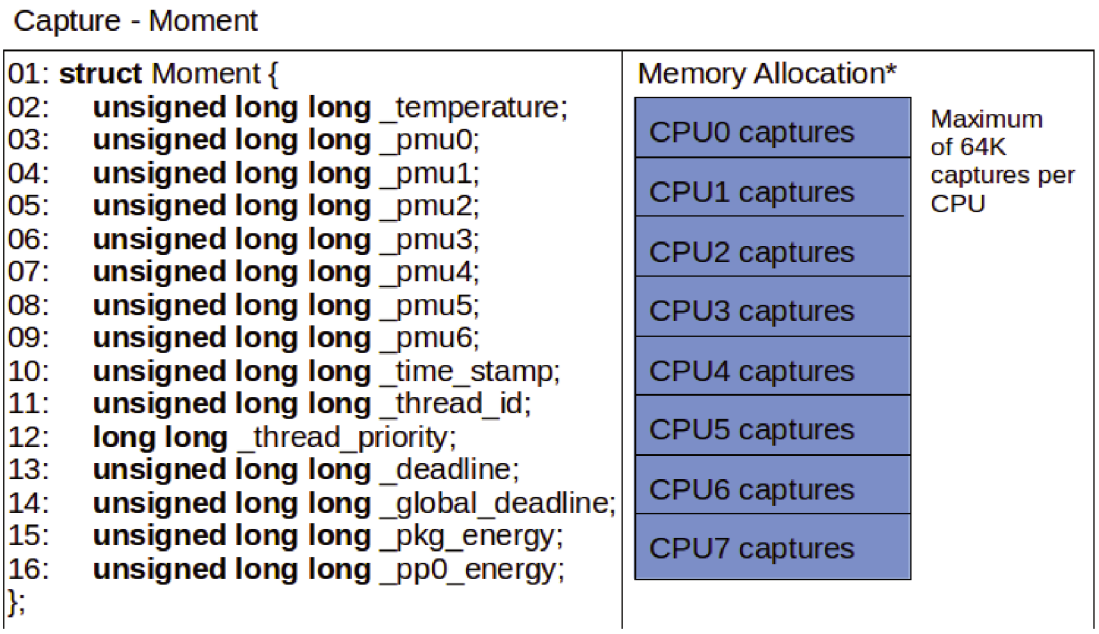
3.4. SISTEMA NÃO INTRUSIVO DE CAPTURAS

Uma captura tem como objetivo gravar informações do sistema em um determinado momento. Tais informações podem vir a ser utilizadas para uma análise detalhada de uma execução e do comportamento do Software e do Hardware.

Ao efetuar uma captura, o “Momento” atual do sistema é armazenado. Este momento é composto pelo *timestamp* da leitura, seguido da temperatura, os três canais fixos da PMU, os quatro configuráveis, o id, a prioridade e o número de *deadline misses* da thread (que está executando ou passará a executar), o número de *deadline misses* em todo o sistema e as informações de consumo energético.

Para lidar com tais variáveis, foi escolhida uma *struct* em C++, composta por 15 variáveis, que por sua vez, descrevem o momento atual do sistema operacional e do Hardware, de acordo com os critérios configurados na inicialização (canais da PMU monitorados).

Figura 3.7 – Estrutura de um Moment



As capturas ocorrem sempre que é executado o *dispatch* de uma thread, ou seja, quando uma troca de contexto acontece. Ao realizar uma captura, podem ser coletadas as informações, tanto da *Thread* que está entrando, quanto da *Thread* que está saindo da execução. Após vários testes, para este trabalho, se provou mais útil capturar apenas da *Thread* que está saindo da execução, mas agrupando a quantidade de perdas de deadline da *Thread* que está saindo e da que está entrando.

É importante ressaltar que as leituras feitas dos registradores da PMU e de temperatura estão relacionados diretamente ao núcleo que executa o código.

3.4.1. NÃO INTRUSIVIDADE

Um dos principais requisitos do sistema, é que ele não gere interferência na execução de uma task-set, ou seja, o tempo final da execução deve ser, idealmente, o mesmo do que quando o sistema operacional executa sem o sistema de capturas

habilitado. Porém como manter o mesmo tempo de execução é impossível, uma interferência mínima é exigida.

Com o objetivo de manter o sistema com sua característica de ser não intrusivo, é necessário uma boa escolha de onde e quando as capturas e a inicialização dos componentes ligados a elas devem ocorrer, com este fim, realizamos estudos (leitura da documentação e do código) sobre a inicialização e funcionamento do sistema EPOS.

A partir dos resultados dos estudos, foram delimitados locais específicos para realizar estas operações, garantindo um impacto baixíssimo no desempenho do sistema. Sendo elas a de inicializar o sistema de capturas junto com a inicialização do SO, mais especificamente ao inicializar o sistema de threads do SO, e a de capturar os dados durante uma troca de contexto a fim de evitar uma nova interrupção no sistema.

Para evitar interrupções indesejadas, um bom método de armazenamento era necessário, dado o fato de que as execuções podem gerar um grande volume de dados, deixando inviável realizar a impressão dos dados na tela ou enviá-los via porta serial durante a execução, pois mesmo isso sendo realizado em um período de *idle* do sistema, dependendo da utilização do task-set sendo executado pode não haver tempo suficiente para o consumo dos dados sem que ocorra perda. Outro ponto é o paralelismo de escrita, pois se o recurso utilizado não for cercado por uma tranca, as impressões se misturam quando duas ou mais CPUs estão realizando esta tarefa.

Logo, um grande espaço em memória deve ser alocado para armazenar todas as capturas até o final da execução. Para evitar o uso de uma tranca na escrita deste vetor, o mesmo foi dividido em 8 partes, uma para cada CPU, como pode ser observado na Figura

3.7 - Estrutura de um Moment. Sendo assim, garantindo confiabilidade das impressões e principalmente não intrusividade neste aspecto.

3.4.2. ESCALONADORES PARA MONITORAMENTO

Como já citado, as inicializações dos componentes ligados ao sistema de captura foram feitas juntamente a inicialização do sistema. Para concretizar a inicialização da PMU, portanto, foi modificado o método de inicialização do escalonador, selecionando-o através das *traits* da aplicação que se deseja executar, incubindo ao escalonador a configuração do monitoramento do sistema. Vale lembrar que, a alteração não modifica o comportamento do escalonador, foi apenas uma alternativa para configurar o monitoramento em tempo de compilação.

Tal método foi implementado, originalmente vazio, criando uma pequena modificação na estrutura já existente. Assim, para habilitar o monitoramento da PMU, basta preencher este método com os comandos de inicialização e configuração. Além disso, optou-se por adicionar uma variável de checagem a ser alterada, a qual foi atribuído o nome de “monitoring”. Todo o código de monitoramento usa estruturas de checagem e, portanto, desativá-lo significa apenas mudar o escalonador no traits antes da compilação.

Para diferenciar os escalonadores nos quais o método init contém a configuração e inicialização da PMU e a atribuição de valor verdade para variável condiciona, os que foram configurados segundo esse princípio foram renomeados para M<nome do escalonador>.

3.4.3. CONFIGURAÇÃO DOS EVENTOS PMU

A partir das configurações citadas no tópico acima, quando um critério de escalonamento que habilita o monitoramento é inicializado, ele inicializa os canais de monitoramento da PMU. Para selecionar quais canais configuráveis serão monitorados durante a presente execução, uma variável estática foi definida na própria classe PMU do EPOS, com a função de definir a posição do evento escolhido no *enum* presente na classe.

Visando facilitar a configuração do sistema para execuções de testes em batch (lote), foi adotado como padrão que a variável definida na classe PMU representa apenas a posição do primeiro evento. Como consequência disso, ao configurar a PMU na inicialização do escalonador, se inicializam os eventos configuráveis a partir da posição descrita pela variável da classe PMU e as 3 posições seguintes no vetor.

3.4.4. ENVIO PARA O BANCO

Como já descrito, um dos objetivos desta implementação, era de que os dados capturados fossem enviados para uma plataforma IoT da UFSC. Para tanto foram analisadas as seguintes possibilidades:

- Envio em tempo real:
    - envio do dado assim que capturado;
    - bufferização dos dados para envio em intervalos;
- Envio após execução;

Com a análise das opções e dos impactos de cada uma, e buscando manter a ideia de um sistema de monitoramento com o mínimo possível de jitter/interferência na execução (não intrusividade), foi escolhida a estratégia de envio após a execução.

Partindo da escolha do método de envio, foi implementado um conjunto de aplicações para administrar o funcionamento do EPOS, além de algumas funções para impressão de dados dentro do próprio SO.

3.4.4.1 METADADOS DE ENVIO

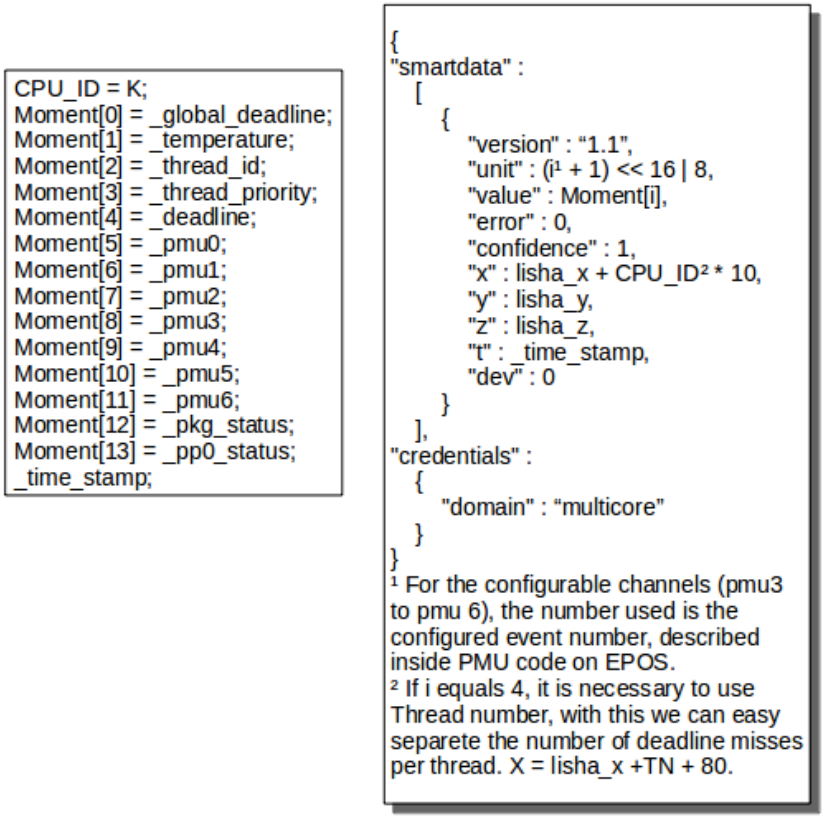
O envio dos dados para IoT segue uma estrutura definida por series e smartdatas, na qual as series representam a série temporal sob a qual os dados estarão associados, e as smartdatas representam os dados em si, juntamente com alguns metadados inerentes a eles, que ajudam na identificação da series a qual está associado [6].

A partir disto, são necessários alguns dados do sistema, além dos capturados, para executar o envio a plataforma IoT UFSC [5], são eles:

- Id's das threads configuradas no teste;
- Series a serem criadas (são geradas antes da execução pois contém os timestamps inicial e final);
- Lista de units (unit é um dos parâmetro das series) utilizadas (varia conforme configuração dos eventos não estáticos);
- Timestamps iniciais e finais.

Destes, os únicos que dependem da execução das tasks e, portanto, são impressos após a execução, são os id's das threads.

Figura 3.8 – Exemplo de montagem de um SmartData e de uma Series.



3.4.4.2 SISTEMA EXTERNO

Visando enviar os dados capturados, é necessário, primeiramente, um algoritmo para monitorar a execução do EPOS e salvar o conteúdo impresso em um arquivo, doravante chamado de log, através da leitura constante de uma porta serial durante a execução. Após o término da execução é preciso, então, de um algoritmo capaz de ler o log e organizar os dados e metadados de maneira a poder realizar o envio das series e das smartdatas. Por fim para o envio das series e smartdatas são necessários scripts configurados conforme a plataforma.

3.4.4.3 SISTEMA INTERNO

Para possibilitar o envio dos dados, capturados durante a execução, para IoT, foram necessárias as implementações de funções para impressão dos dados, em si, e de alguns metadados para facilitar o processo de envio.

Tendo em mente as questões ligadas ao funcionamento e ao desempenho do sistema, o melhor lugar para realizar a impressão dos metadados e dados relacionados a execução era após a conclusão da mesma, enquanto os dados e metadados independentes da execução são impressos antes do início da execução das tasks. Logo, o único impacto durante a execução é o do processo de captura. A partir disso, o sistema começa a etapa de impressão logo após perceber o fim da execução da última thread que, no EPOS, acontece na primeira verificação feita ao entrar em idle, antes do comando que ordena o desligamento do computador.

A fim de sintetizar este tópico, podemos então descrever o funcionamento do sistema da seguinte maneira:

1. O sistema de captura é configurado junto a inicialização do sistema de Threads do EPOS.
2. A PMU é configurada no início da execução, junto da inicialização do sistema de escalonamento;
3. As capturas são habilitadas assim que a aplicação inicia sua execução, e também são desabilitadas junto ao fim da aplicação.
4. Sempre que o sistema reescalona uma Thread, uma captura do sistema naquele momento é realizada e armazenada.

5. Antes do sistema operacional finalizar, os momentos armazenados são enviados via porta serial para outro computador.
6. Neste outro computador um log é gerado, e em sequência é consumido pelo algoritmo de envio dos dados para a plataforma IoT do LISHA no formato de Series e SmartDatas.

3.4.5. TESTE DE NÃO INTRUSIVIDADE

Para avaliar a não intrusividade do sistema, foi realizado uma comparação de uma execução com e sem o sistema habilitado, usando como base apenas o tempo de execução da aplicação, ou seja, apenas do período em que ocorrem capturas. Os resultados foram positivos, apresentando um aumento de aproximadamente 0.000002% em relação a execução sem o sistema de capturas. Os cálculos foram feitos da seguinte maneira:

- O tempo de execução pode ser descrito como o período de uma Thread vezes a quantidade de iterações que ela executa durante a iteração. No exemplo utilizado, todas as threads foram configuradas para executar por 2 minutos (120 segundos), sendo a Thread utilizada para calcular o tempo teórico tinha o período de 31000us, logo ela deveria executar 3900 vezes para executar por aproximadamente 2 minutos.
    - Tempo = Iterações * período;
    - Tempo teórico = 3900 * 31000 us = 120900000 us;
    - Tempo de execução sem o sistema de Monitoramento = 122122579 us;
    - Tempo de execução com o sistema de Monitoramento = 122122803 us;
    - Diferença = 224 us = 0.000224 seg = 0.000001834%.

Tabela 3.1 – Conjunto de Tarefas Utilizado para os testes de não intrusividade, CEDF - 15 Threads.

Período (us)	Deadline (us)	WCET (us)	CPU (Cluster)
4000	4000	435	0
14000	14000	1084	0
20000	20000	1094	0
5000	5000	84	0
23000	23000	2377	1
31000	31000	2513	1
25000	25000	1849	1
25000	25000	2547	2
22000	22000	1838	2
20000	20000	1277	2
30000	30000	440	2
15000	15000	1425	3
6000	6000	556	3
22000	22000	1018	3
23000	23000	810	3

4. GERAÇÃO E ANÁLISE DOS DADOS

A fim de cumprir os objetivos deste TCC, se viu necessário a geração de um conjunto de dados de execuções para análise, pois, dado este conjunto, será possível entender mais sobre o funcionamento do processador durante uma execução, via os contadores de eventos. Sendo então, o sistema não intrusivo de capturas, descrito no tópico “3.4. Sistema Não Intrusivo de Capturas”, utilizado para realizar tal tarefa.

O resultado das execuções é um log contendo todos os dados descritos na Figura 3.7, de todos os núcleos do processador, além dos metadados descritos no tópico “3.4.4.1 Metadados de Envio”. Estes logs estão disponíveis, para análise, no banco de dados IoT, via a interface de SmartData e Series, e também estão disponíveis localmente.

Sendo o foco da futura análise o comportamento do sistema em momentos em que ocorre uma perda de deadline, os dados gerados devem representar tal momento, ou seja, deve se tomar o cuidado ao gerar os conjuntos de tarefas para que perdas de deadline ocorram. Para solucionar isto, foi então escolhido uma abordagem na qual o sistema executa sobre um conjunto de tarefas que exige um uso maior que o fornecido, sendo o DVFS aplicado de forma a gerar as perdas de deadline, possibilitando também, indicar possíveis pontos de corte para a heurística que se busca desenvolver. Outra possibilidade é de que a cada uma certa quantidade de operações, o uso é aumentado, causando então um sobreuso da CPU e gerando perdas de deadline.

Lembrando também que, com a versão Sandy Bridge da PMU Intel, possuímos apenas 4 contadores de eventos configuráveis e 3 contadores de eventos fixos, e para poder analisar todos os contadores, deve ser gerado um grupo de execuções, sendo que cada uma cobre 4 eventos distintos. Isto é possível através das características de não

intrusividade tanto do EPOS, quanto do próprio sistema de capturas, podendo-se considerar os dados como se fossem de uma mesma execução, permitindo então encontrar correlações entre estes eventos.

4.1. CÓDIGOS PARA ESTÍMULO DOS CONTADORES

Uma importante definição para a geração dos dados, além do que já foi descrito no tópico anterior, é como estimular os contadores, ou seja, quais tarefas devem ser executadas ao longo de uma execução, para que seja possível gerar perdas de deadline ao mesmo tempo que os contadores demonstram variabilidade no seu comportamento. Visto que, tarefas diferentes estimulam partes diferentes de um processador, como, por exemplo, tarefas iterativas e matemáticas focam seu uso na ULA (Unidade Lógica Aritmética), e tarefas de cópias de regiões de memória focam seu uso em acesso à hierarquia de memória do computador, deve-se então, gerar alguns grupos de execuções para análise.

Os dois tipos de tarefas que serão utilizados para a geração da heurística aqui desenvolvida, tem relação com a memória, sendo que um deles também envolve tarefas matemáticas e o outro foca apenas em tarefas de acesso a memória.

O primeiro grupo de execuções realiza o algoritmo de cálculo do Fibonacci recursivo, muito conhecido na área de computação. Sendo o Fibonacci na, sua implementação recursiva, também lembrado por ter um uso alto do processador, graças a sua característica de alta recursividade, o que implica numa complexidade exponencial. A implementação aqui utilizada é descrita na Figura 4.1 (a), sendo a tarefa executada pela Thread Periódica descrita na Figura 4.1 (b).

Figura 4.1 (a) – Código de execução da tarefa de Fibonacci Recursivo.

```
1: int fib(int pos) {
2:     if (pos == 1 || pos == 0) {
3:         return 1;
4:     } else {
5:         return (fib(pos - 1) + fib(pos - 2));
6:     }
7: }
```

Figura 4.1 (b) - Tarefa de controle da Thread Periódica.

```
01: #define FIB_REC_BASE_TIME 600
02: int fib_recursive (int id) {
03:     int pos = 2;
04:     //int run = 0;
05:     float ret = 1.33;
06:     int max = (int) ((threads_parameters[id][2]/FIB_REC_BASE_TIME));
07:     for (int i = 0; i < iterations[id]; i++) {
08:         //run = max;
09:         //if (!(i % 15)) {
10:         //     run *= 3;
11:         //}
12:         // for (int x =0; x < run; x++) {
13:         for (int x = 0; x < max; x++) {
14:             ret *= fib(25);
15:             //pos++;
16:         }
17:         Periodic_Thread::wait_next();
18:     }
19:     return int(ret);
20: }
```

O segundo grupo de execuções realiza o algoritmo de Cópias de Regiões de Memória [29], também conhecido como Memory Copy (MemCpy), sendo este um algoritmo que também estimula o processador, mas neste caso em acesso a memória, estimulando, principalmente, os contadores de acesso aos níveis de cache, a própria memória RAM e operações de Flush (atualização) das tabelas de memória. Lembrando que o EPOS já possui uma implementação da função de Cópia de Regiões de Memória.

Figura 4.2 – Código de execução e controle da tarefa de MemCpy.

```
01: #define BASE_MEMCPY_T 30
02: int vectors[Traits<Build>::CPUS][32768];
03: //Inicialização dos vetores -- Ocorre na main, antes da criação das demais threads
04:   //for (int m = 0; m < Machine::n_cpus(); m++) {
05:   //     for (int i = 0; i < 32768; i++) {
06:   //         vectors[m][i] = i * m - 2 * m;
07:   //     }
08:   // }
09:
10: int mem_cpy (int id_thread) {
11:     Random * rand;
12:     float ret = 1.33;
13:     int max = (int) ((threads_parameters[id][2]/BASE_MEMCPY_T));
14:
15:     for (int i = 0; i < iterations[id_thread]; i++) {
16:         Periodic_Thread::wait_next();
17:         rand->seed(clock.now());
18:         //run = max;
19:         //if (!(i % 15)) {
20:         //     run *= 3;
21:         //}
22:         // for (int x =0; x < run; x++) {
23:         for (int x = 0; x < max; x++) {
24:             memcpy(reinterpret_cast<void *>(&vectors[(x*3)%3]),reinterpret_cast<void
*>(&vectors[rand->random() % Machine::n_cpus()]),sizeof(vectors[0]));//size = 32768 * 4 = 128KB
25:         }
26:     }
27:     return int(ret);
28: }
```

Para definir os parâmetros relacionados a quantidade de repetições por período da Thread, apresentados nas Figuras 4.1 e 4.2, foi utilizado um cronômetro do próprio EPOS para identificar quantas vezes era necessário repetir as operações, a fim de atingir um tempo próximo do especificado como Tempo de Execução no Pior Caso (WCET - Worst Case Execution Time) pelo conjunto de tarefas, sendo que a tarefa em si não pode ocupar um grande período de tempo, pois o WCET pode ser pequeno, por esse motivo que foi inserido o laço de repetição na execução dos algoritmos.

4.1.1. CONFIGURAÇÃO DOS TASK-SETS UTILIZADOS

Outra definição importante a ser realizada antes de gerar os dados é qual a configuração do conjunto de tarefas. Para tal, foi utilizado o algoritmo descrito por Gracioli em Real-Time Operating System Support for Multicore Applications (2014) [28] chamado de Particionamento de Tarefas, este, por sua vez, gera o conjunto de tarefas para uma arquitetura Multi-core em um sistema de Tempo Real.

Neste trabalho, o enfoque foi no algoritmo de escalonamento PEDF, o qual separa as tarefas por CPU, e não realiza troca de tarefas entre as CPUs, facilitando a análise dos contadores, visto que o comportamento da CPU ao longo da execução é mais determinístico. Logo, os conjuntos de trabalho, gerados pelo algoritmo de Particionamento de Tarefas, que foram utilizados para gerar os dados e realizar as análises, são os seguintes:

Tabela 4.1 – Configuração do Conjunto de Tarefas 1 - 12 Threads.

Período*	Deadline*	WCET*	CPU
25000	25000	13958	0
100000	100000	15135	1
200000	200000	136986	2
50000	50000	25923	3
25000	25000	11637	4
100000	100000	20072	5
50000	50000	30484	6
200000	200000	25220	7
200000	200000	23924	7
100000	100000	31920	1
50000	50000	18343	5

50000	50000	19205	7

*Tempo em Microssegundos

Tabela 4.2 - Configuração do Conjunto de Tarefas 2 - 14 Threads.

Período*	Deadline*	WCET*	CPU
50000	50000	27098	0
25000	25000	5504	1
100000	100000	68919	2
100000	100000	64664	3
50000	50000	9310	4
200000	200000	105758	5
200000	200000	29326	6
200000	200000	67222	7
50000	50000	21151	6
50000	50000	6757	4
50000	50000	34329	1
50000	50000	8203	4
100000	100000	44566	7
25000	25000	8853	4

*Tempo em Microssegundos

Para melhorar o entendimento dos conjuntos, segue o uso de cada CPU:

Tabela 4.3 - Uso de cada CPU nos Conjuntos 1 e 2.

CPU	Conjunto 1 (Tabela 4.1)	Conjunto 2 (Tabela 4.2)
0	55,832%	54,196%
1	47,055%	90,674%
2	68,493%	68,919%
3	51,846%	64,664%

4	46,548%	83,952%
5	56,758%	52,879%
6	60,968%	56,965%
7	62,982%	78,177%

Observando a Tabela acima (Tabela 4.3), podemos notar que o conjunto 2 apresenta, na maioria dos casos, um uso mais alto que o conjunto 1, principalmente se for levado em conta o fato de que o processador utilizado na máquina em que os dados serão gerados, faz uso a técnica de Hyper-threading, onde as CPUs lógicas se encontram em grupos de 2 em uma CPU física, a CPU 0 e 1, por exemplo, compartilham a mesma CPU física, logo, o uso da CPU física é mais alto no conjunto 2. Essa conclusão fez com que, na maioria dos casos, os dados gerados fossem mais focados no conjunto 2, e, mais a frente, a heurística também fosse testada com o conjunto 1 para uma análise de desempenho.

Vale também citar que, como descrito tópico “4. Geração e Análise dos dados”, foi estabelecido que os dados seriam gerados com o DVFS aplicado para representar, de forma simples, uma heurística básica em ação, e, além disso, gerar perdas de deadline ao longo da execução. Agora com o uso estabelecido, foi então selecionado reduzir a frequência para 62,5% (fator aplicado no Duty Cycle de 5/8), o que a princípio geraria perdas de deadline em cinco das oito CPUs do conjunto 2 e em duas das oito no conjunto 1. Já em relação a possibilidade de gerar perdas de deadline por meio de um sobreuso com a frequência inalterada, foram gerados dados em que a cada 15 iterações, o uso era triplicado por aquela iteração, e após isso, volta ao normal.

4.2. ANÁLISE DOS DADOS

A análise dos dados começa com a busca de correlações entre os contadores de eventos provenientes das execuções e a ocorrência de uma perda de deadline na mesma. Para realizar tal tarefa é necessário recuperar os dados gerados, ou via IoT, realizando Get's nas series, ou via os próprios logs disponíveis na máquina após serem gerados. Por motivos de simplicidade, foi utilizado diretamente os logs, mas um algoritmo para recuperar os dados através de um get na IoT também foi desenvolvido, porém consome mais tempo.

Tendo os dados recuperados, como o objetivo é de utilizar estes dados em um dos algoritmos de correlação da ferramenta WEKA, foi optado por utilizar o padrão CSV (Comma Separated Value) para a realização das análises, sendo estes arquivos gerados a partir dos logs, via um script python. Os arquivos contém os eventos capturados (três fixos e os quatro configuráveis) junto das informações de perda de deadline, alinhadas pelo tempo em relação ao início da execução.

4.2.1. TRATAMENTO DOS DADOS

Neste ponto, com os dados em mãos no formato CSV, são removidas as capturas que possuem pouco espaço de tempo entre elas, cerca de menos de 0.3 milissegundos (300 microssegundos), pois ao realizar um primeiro tratamento, outliers são gerados.

Os primeiros pontos do tratamento são de deixar equivalentes os valores entre as capturas, pois capturas com maior distância temporal apresentam maior crescimento nos contadores. Dessa forma, o valor do contador é subtraído da captura anterior, possibilitando o trabalho com diferenças, ou seja, o crescimento do contador naquele período de tempo. Após isso, esse valor é dividido pela diferença do tempo da captura

atual para a captura anterior, podendo então, trabalhar com o crescimento do contador. Logo, temos uma análise que não é afetada por capturas que ocorreram com uma grande diferença de tempo em relação as demais.

O próximo passo trata das perdas de deadline, onde as mesmas sofrem um tratamento parecido com o do passo anterior, porém sem a divisão pelo tempo, ou seja, apenas temos a diferença da quantidade de deadlines no momento em relação a captura anterior. Após isso, valores positivos são transformados em 1 (um), e valores negativos ou zero são transformados em 0, fazendo com que a variável represente a ocorrência ou não de uma deadline, e não a quantidade específica de deadlines que ocorreram ou que estavam acumuladas.

Tabela 4.4 - Exemplo de CSV gerado para uma CPU.

Time	pmu0	pmu1	pmu2	pmu12	pmu13	pmu14	pmu15	deadlineM
1798640	3057	2693	3392	0	0	67	0	0
1802598	3096	2693	3392	0	0	96	0	1

Enquanto no nível de análise, como o nome que representa o evento é normalmente grande, foi decidido pelo uso de um número que representa o evento numa lista de eventos na PMU (Anexo A1), mas em momentos em que o evento é importante para o entendimento, será utilizado também o seu nome de acordo com a sua descrição no manual de desenvolvedores da Intel [4].

4.2.2. ESTUDO DE CORRELAÇÕES

Com os CSVs de uma execução prontos, é possível iniciar uma análise das correlações no WEKA. Para tal, foi escolhido o algoritmo *Correlation Attribute Evaluation* [27], o qual atribui um peso ao atributo de acordo com o cálculo de correlação de Pearson

entre aquele atributo e a classe selecionada, que neste caso é o atributo de perda de deadline. O valor retornado sempre está entre -1 e 1, sendo -1 uma correlação totalmente negativa, 1 uma correlação totalmente positiva e 0 sem correlação. Os resultados são similares a Matriz gerada pelo algoritmo PCA, mas focando apenas em um atributo, que no caso são as perdas de deadline.

No entanto, o método utilizado não foi feito para séries temporais, que é o caso dos dados gerados. Para tentar contornar esse problema, uma das alternativas é realizar um *forward*, ou seja, deslocar em uma ou mais posições o atributo que representa as perdas de deadline. Isto serve para possivelmente alinhar a perda de deadline com o evento que a gerou, pois, como descrito anteriormente, o objetivo da heurística é de evitar o acontecimento de uma perda de deadline, logo, precisamos encontrar eventos que mudaram seu comportamento alguns momentos antes de uma deadline ocorrer.

As análises começaram com a busca por correlação no conjunto de tarefas descrito na Tabela 4.2 executando a tarefa Fibonacci Recursivo, descrito nas Figuras 4.1 (a) e (b) com o DVFS aplicado em 62,5%. O foco da análise então foram as CPUS 6 e 7, de modo que a CPU 7 apresenta perdas de deadline e a CPU 6 não, de acordo com o uso do conjunto de tarefas, vide Tabela 4.3.

Os primeiros eventos a apresentarem uma correlação relevante foram os fixos, sendo eles INSTRUCTION_RETIRED, CLOCK e DVS_CLOCK. Onde o primeiro conta o número de instruções finalizadas, o segundo conta os ciclos de clock que não estão em pausa (*halt*) e o terceiro conta os ciclos de clock seguindo um clock de referência, ou seja, a contagem no evento CLOCK é afetada pelo DVFS e o DVS_CLOCK não.

Ainda sobre estes eventos e seu comportamento, o crescimento sempre ocorre durante a execução, sendo que quando a CPU está em *idle* eles apresentam um

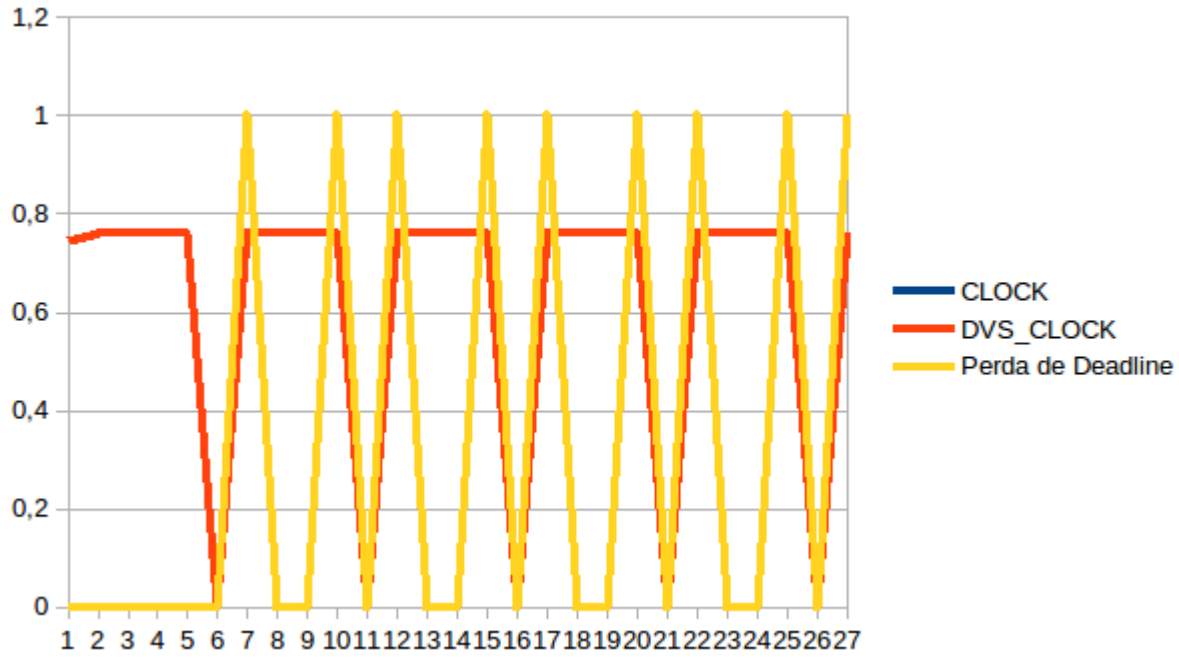
crescimento ínfimo. As correlações apresentadas pelo Weka destes eventos com a perda de deadline está apresentada na tabela abaixo.

Tabela 4.5 - Correlação apresentada pelo Weka dos eventos fixos com a perda de deadline.

Evento	Correlação
INSTRUCTIONS_RETIRED	0,37987
CLOCK	0,39259
DVS_CLOCK	0,3926

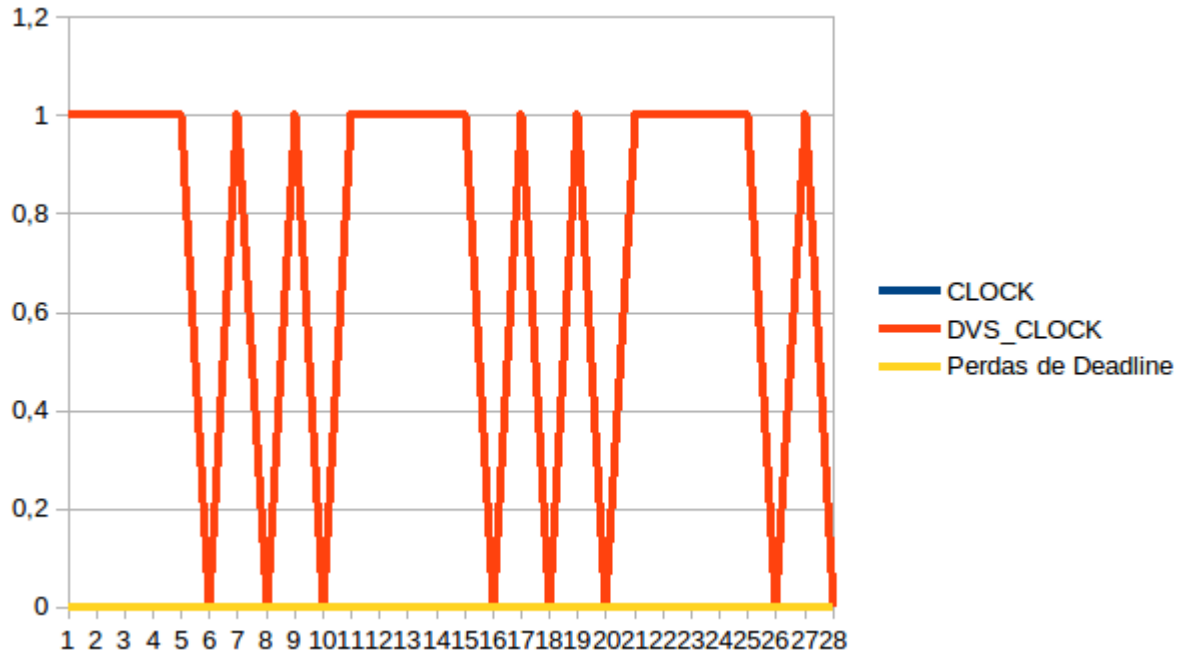
Sendo então a correlação mais alta com o evento DVS_CLOCK, seguida pelo CLOCK e depois o INSTRUCTIONS_RETIRED. Porém, analisando os valores diretamente no log, podemos identificar que os valores de CLOCK e DVS_CLOCK tem crescimento praticamente constante, visto que o DVFS só é aplicado no começo, logo a frequência de execução não muda e o contador DVS_CLOCK conta a uma frequência fixa, logo também não muda. Isto pode ser verificado na Figura abaixo (Figura 4.3 e 4.4).

Figura 4.3 - Valores dos eventos normalizados ao longo de um trecho da execução na CPU 7, onde ocorrem perdas de deadline.



A princípio, os eventos parecem estar correlacionados, porém, ao levar em conta que o evento só apresenta tais baixas pois o sistema acabará de entrar em um estado de idle, a correlação perde o sentido. Um outro ponto que pode reforçar a esta conclusão é que quando o sistema não apresenta perdas de deadline, o comportamento dos contadores muda apenas pela diferença no uso dessas duas CPUS, e não acompanha as perdas de deadline, como demonstrado na Figura abaixo.

Figura 4.4 - Valores dos eventos normalizados ao longo de um trecho da execução na CPU 6, onde não ocorrem perdas de deadline.



Este tipo de comportamento também acontece com outros eventos, como o LONGEST_LAT_CACHE_MISS, que conta a quantidade de misses para referências a cache do último nível, que apresentou uma correlação de 0,3939. Este evento, por sua vez, poderia possuir sim a correlação desejada, pois aponta um evento que a princípio degradaria o desempenho, mas o comportamento apontado como correlacionado quando ocorrem perdas de deadline, também acontece quando não existem tais perdas. Isto pode ser verificado na Figura 4.5 (a) e (b).

Figura 4.5 (a) - Valores do evento normalizados ao longo de um trecho da execução na CPU 7, onde ocorrem perdas de deadline.

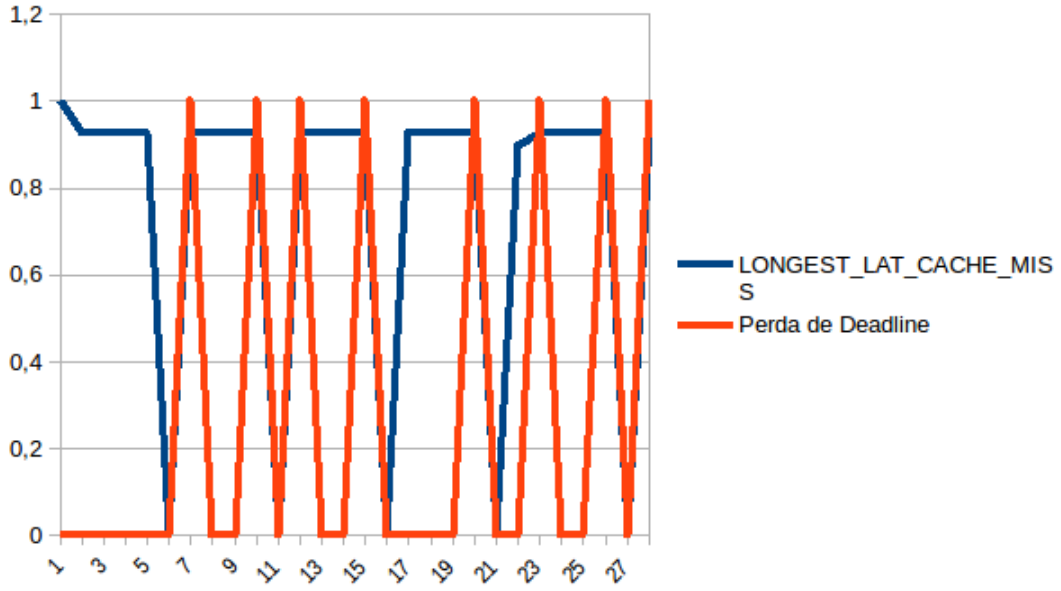
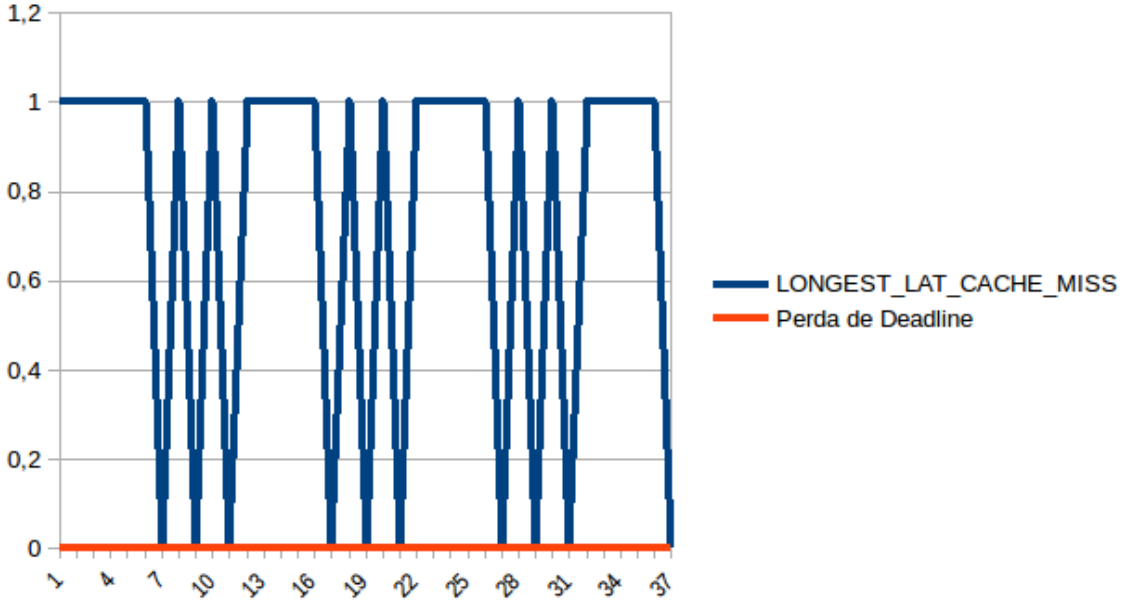


Figura 4.5 (b) - Valores do evento normalizados ao longo de um trecho da execução na CPU 6, onde não ocorrem perdas de deadline.



Sendo mais um contador que apresenta a queda no crescimento nos momentos em que a CPU está em idle, e apresentou correlação apenas pela característica periódica das tarefas.

Continuando a análise, foi possível verificar que o evento INSTRUCTIONS_RETIRED não segue este comportamento de variar apenas quando entra ou sai de um momento de idle, pois o mesmo apresenta um crescimento sempre que o sistema aumenta seu uso, ou seja, finaliza mais instruções no tempo. Este sim pode ser um indicativo mais confiável de uma possível perda de deadline, como podemos verificar na Figura 4.6 (a) e (b), onde o contador apresenta um aumento no crescimento momentos antes de uma perda de deadline, e a ausência desse crescimento quando não existem perdas de deadline.

Figura 4.6 (a) - Valores dos eventos normalizados ao longo de um trecho da execução na CPU 7, onde ocorrem perdas de deadline.

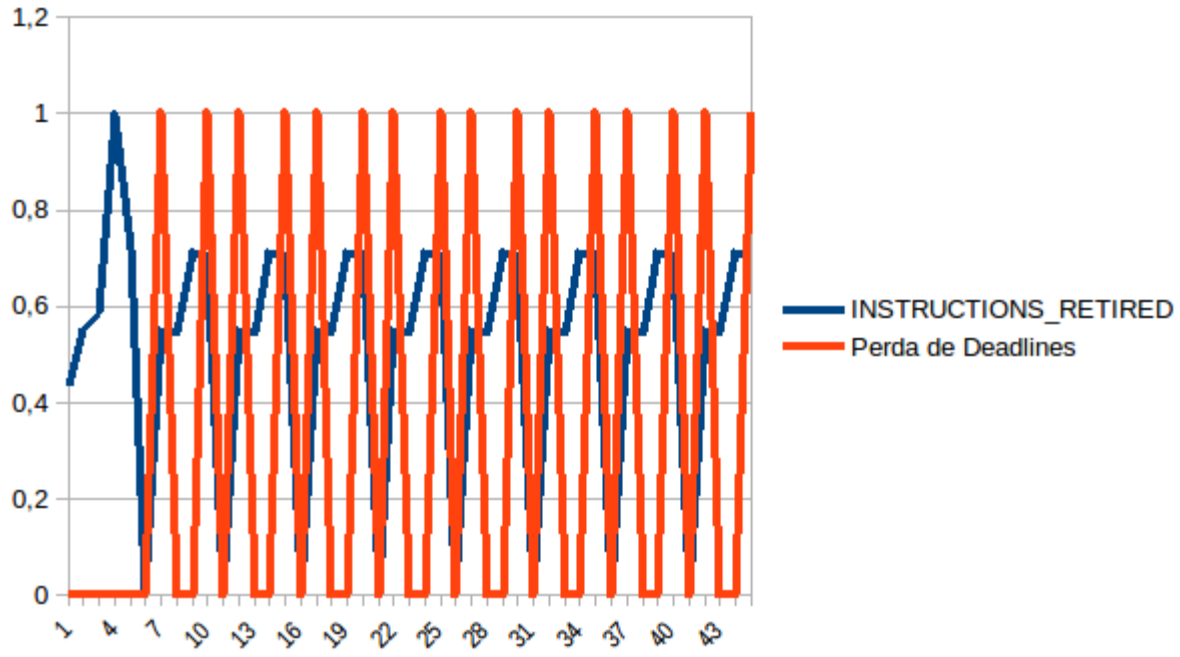
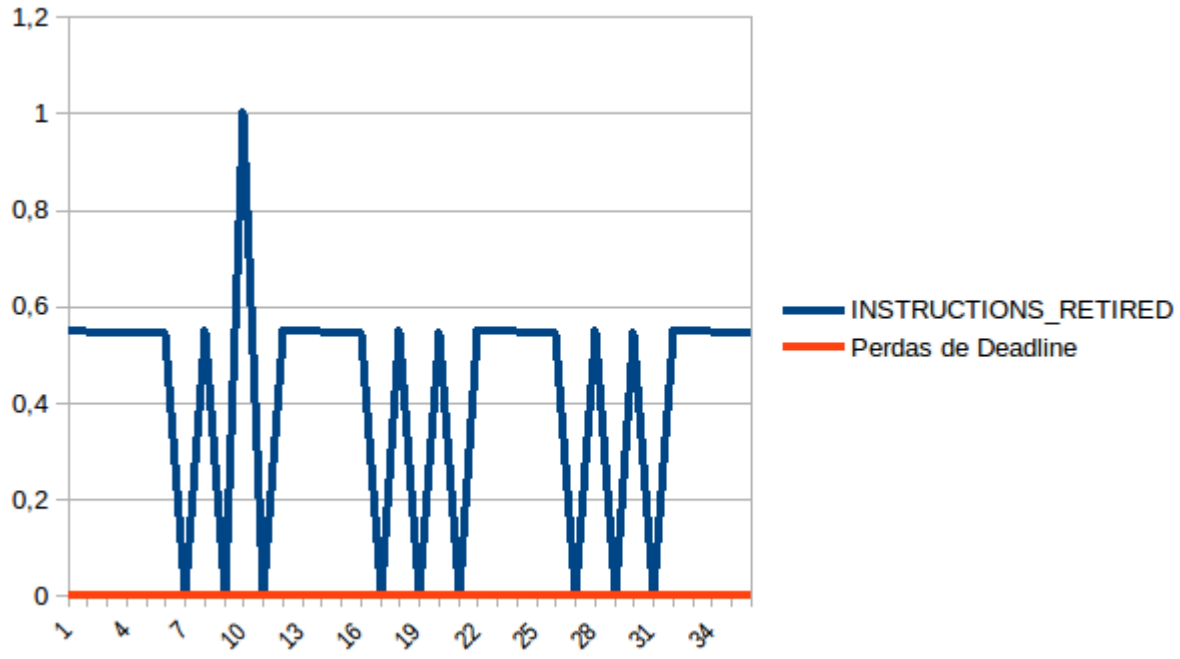


Figura 4.6 (b) - Valores dos eventos normalizados ao longo de um trecho da execução na CPU 6, onde não ocorrem perdas de deadline.



Uma explicação para a alta correlação de variáveis que na verdade não possuem uma correlação real, é que a Correlação de Pearson avalia se há evidência estatística de uma relação linear entre os mesmos pares de variáveis na população [26], logo, ele ignora a característica temporal dos dados. Sendo assim, algumas correlações na verdade estão mais ligadas ao fato de que a variação no evento de perda de deadline é baixa, pois na maioria das capturas o valor se encontra em zero (não ocorrência de uma perda de deadline). Além disso, alguns canais que apresentam uma maior variabilidade ao longo da execução, possuem uma correlação de certa forma baixa, porém essa variabilidade pode sim corresponder a uma futura perda de deadline.

Um ponto que também foi analisado, foi a possibilidade de gerar muitas deadlines em uma execução, fazendo com que os canais que antes apresentavam correlação por não variar deixassem de apresentar a correlação. Porém, isso coloca o sistema em um

ponto de saturação, o que de certa forma piora ainda mais as correlações com os canais que realmente apresentariam uma correlação, devido ao fato de que seu comportamento ali apresentado não corresponde ao sistema numa execução normal ou com a futura heurística habilitada. Neste caso, apenas os momentos iniciais seriam úteis, ou seja, antes do sistema entrar num estado de saturação.

4.2.3. ANÁLISE MANUAL GUIADA PELAS CORRELAÇÕES

A solução encontrada foi de aliar as buscas de correlações uma análise manual, como já foi demonstrado acima, comparando as correlações do Weka com os logs das execuções. Ou seja, os arquivos CSVs que continham os eventos que apresentaram as maiores correlações, ou que esperava-se um certo grau de correlação devido a características arquiteturais, foram avaliados. Assim, removendo da análise eventos que não apresentavam variação, ou que a mesma não estava ligada a ocorrência de perdas de deadline. Mesmo que esse método possa deixar passar alguns eventos que teriam uma boa relação com as perdas de deadline, ele apresentou resultados mais convincentes que apenas utilizando a análise pelo algoritmo do WEKA.

Para realizar esta análise, foram escolhidos alguns parâmetros, o evento que seria analisado deveria ter apresentado uma correlação entre as maiores das apresentadas no WEKA (também considerando correlação com o *forward* em alguns eventos) e/ou ter uma relação arquitetural com a tarefa executada.

A partir disso, os eventos foram analisados manualmente nos CSVs. Nessa análise, foi realizada uma busca de relações entre as variações do crescimento das ocorrências do evento e a ocorrência ou não de uma perda de deadline. Na maioria dos casos, o evento é analisado em uma CPU em que não ocorre, ou que ocorrem poucas,

perdas de deadline, e em uma CPU em que ocorrem várias perdas de deadline. Dentro da mesma CPU também, quando possível, são analisados os eventos em períodos de tempo em que não ocorrem perdas de deadline e em períodos que ocorrem. Buscando assim filtrar o eventos ao mesmo tempo em que já é identificado algum possível ponto de corte da heurística.

Em alguns casos, quando um evento sozinho não consegue prever todas as ocorrências de perda de deadline, mas prevê a maioria, é realizada uma análise combinada dele e de mais algum outro evento que também teve uma boa relação com as perdas de deadline, tentando encontrar um conjunto que se complementa ao longo da execução.

A análise dos pontos de corte será realizada no tópico “5. Implementação da Heurística”, além de uma análise focada no comportamento das heurísticas e as ações tomadas em cada passo.

4.2.3.1 FIBONACCI RECURSIVO

Continuando aqui a análise dos dados das execuções do algoritmo Fibonacci Recursivo, agora realizando as análises de correlações no WEKA junto da análise manual.

Tabela 4.6 - Resumo dos eventos selecionados pelas correlações no WEKA, nas execuções de Fibonacci Recursivo.

Evento	Forward 0	Forward 1
LD_BLOCKS_NO_SR	05,733%	12,3616%
LONGEST_LAT_CACHE_MISS	39,39%	-54,901%

O evento LONGEST_LAT_CACHE_MISS já foi citado no tópico “4.2.2 Estudo de Correlações” como um evento que aparecia com uma alta correlação quando analisado, mas ainda é interessante citar o mesmo, visto que quando analisado com o *Forward* em 1 ele apresenta uma correlação negativa muito alta, comprovando que não tem uma característica de previsão de perda de deadlines tão boa assim.

Sobre o evento LD_BLOCKS_NO_SR, ele apresentava uma correlação baixa quando analisado sem o uso do forward, mas aumentou consideravelmente esta correlação quando analisado utilizando o *Forward* em 1, o que significa que o mesmo pode apresentar pontos de previsão de uma perda de deadline, o que levou a sua análise manual. O evento conta quantas vezes foi realizado um bloqueio em uma operação de carga dividida (*Split Load*) devido a recurso não disponível. Sendo um evento, que por sua vez, apresenta relação com a tarefa realizada, visto sua característica de alta recursividade, ao realizar muitas chamadas recursivas, o algoritmo faz aumentar o tamanho da pilha de execução.

Com a análise manual. foi possível identificar alguns pontos de controle que podem vir a ser úteis na heurística, pois o evento, na maioria das vezes, apresenta um aumento no seu crescimento antes da execução apresentar uma perda de deadline. Outro ponto importante, que reafirmou a utilidade do evento, é que na CPU 6, onde não aconteceram perdas de deadline nesta execução, o mesmo manteve seu valor baixo. As imagens abaixo demonstram este comportamento.

Figura 4.7 (a) - Valores do evento LD_BLOCKS_NO_SR normalizados ao longo de um trecho da execução na CPU 7, onde ocorrem perdas de deadline.

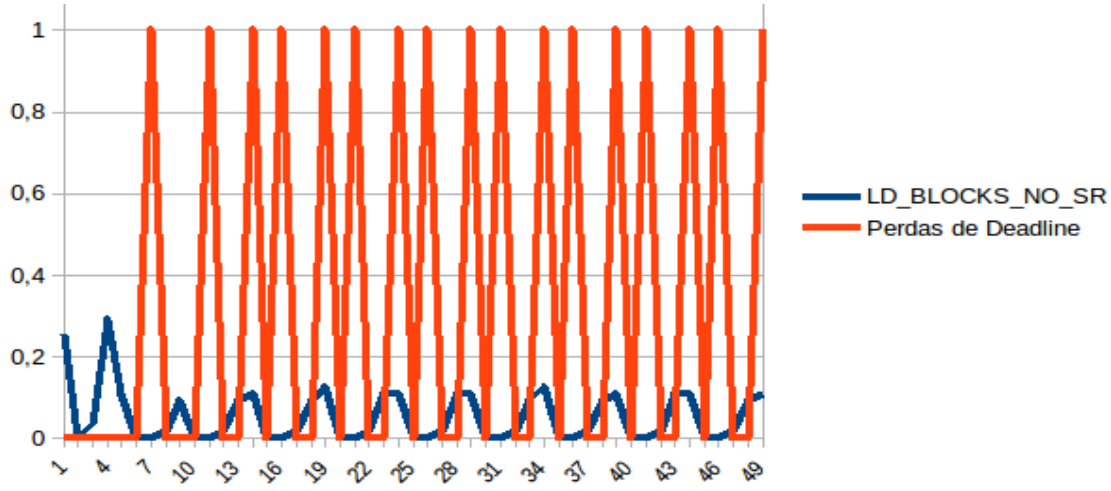
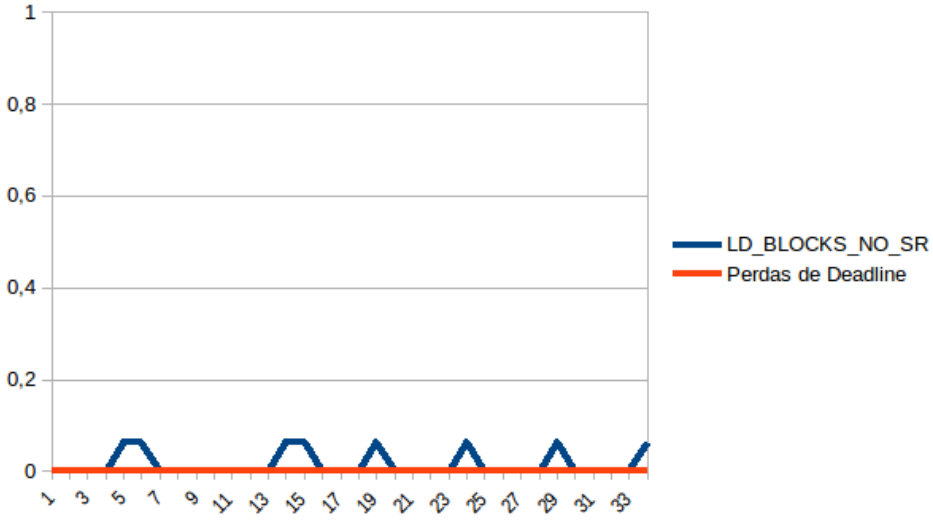


Figura 4.7 (b) - Valores do evento LD_BLOCKS_NO_SR normalizados ao longo de um trecho da execução na CPU 6, onde não ocorrem perdas de deadline.



4.2.3.2 CÓPIA DE REGIÕES DE MEMÓRIA

Com o canal utilizado para o algoritmo Fibonacci Recursivo definido, agora será realizada a análise para o algoritmo de Cópias de Região de memória. Sendo então primeiro analisadas as correlações no WEKA e após isso, os canais são selecionados ou

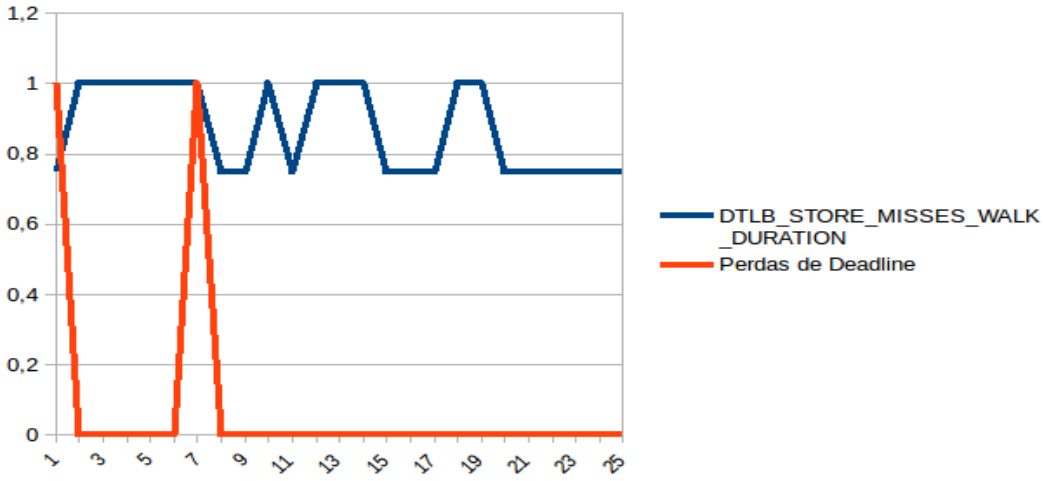
não pela análise manual, da mesma forma que na análise dos eventos para o Fibonacci Recursivo.

Tabela 4.7 - Resumo dos eventos selecionados nas correlações do WEKA nas execuções de Cópia de Regiões de Memória.

Evento	Forward 0	Forward 1
LD_BLOCKS_NO_SR	09,31%	08,38%
DTLB_STORE_MISSES_WALK_DURATION	23,844%	13,014%
ITLB_ITLB_FLUSH	09,85%	09,1803%
OFFCORE_REQUESTS_DEMAND_DATA_RD	09,67%	07,8002%

O evento DTLB_STORE_MISSES_WALK_DURATION, apresenta uma alta correlação, se comparado com os outros eventos apresentados na tabela, e também foi um dos eventos que apresentou a maior correlação durante a análise sem utilizar o *forward* dos dados (Forward 0 na tabela), porém, ao analisá-lo de forma manual, o mesmo não tem uma correlação tão boa assim, sendo que apresenta a variação mais lenta que outros eventos, o que gerou uma correlação com as perdas de deadline. O ponto principal é que em alguns períodos ele aparenta prever a perda da deadline, diminuindo seu crescimento, mas em outros não, como demonstrados na figura abaixo.

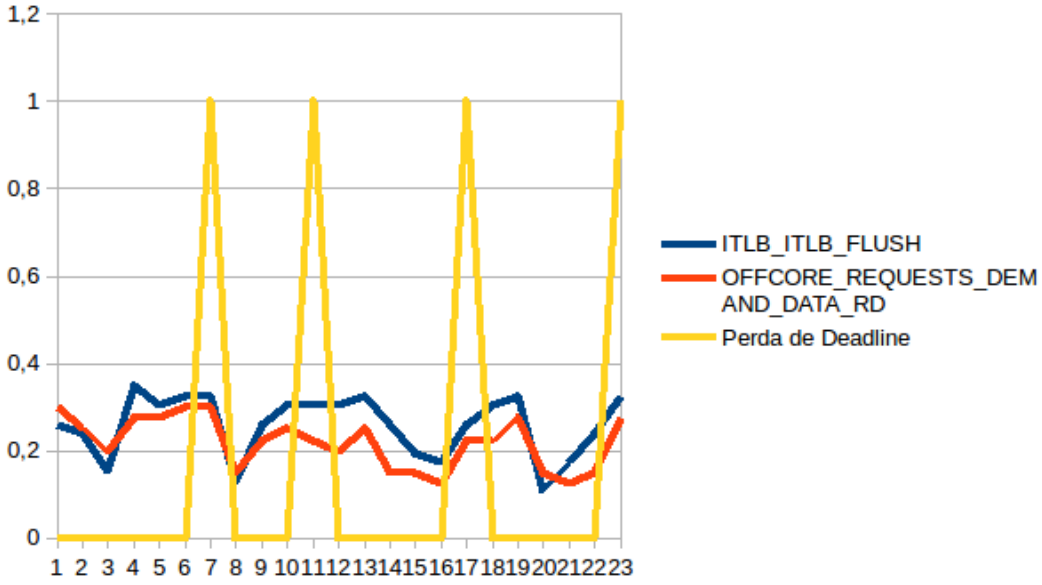
Figura 4.8 - Valores do evento DTLB_STORE_MISSES_WALK_DURATION normalizados ao longo de um trecho da execução.



Mesmo sendo um evento que tinha características em comum com a tarefa executada, seu comportamento não consegue descrever tão bem o momento de ocorrência de uma perda de deadline. Isto também aconteceu para mais alguns eventos ao longo da análise, mas o evento citado acima representa bem o problema.

Já sobre os eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD, que também estão ligados as operações de cópia de regiões de memória, apresentaram uma correlação linear não tão alta. Ao analisá-los manualmente, apresentaram bons resultados, pois em períodos em que uma Thread sofre uma perda de deadline, ambos apresentavam uma queda no crescimento. A correlação aqui não é trivial, e somente após identificar alguns padrões de funcionamento, que foi possível entender que estes canais tinham uma correlação com as perdas de deadline, os gráficos apresentados nas figuras abaixo ilustram isso.

Figura 4.9 - Valores dos eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD normalizados ao longo de um trecho da execução.



Estes eventos foram identificados como complementares em alguns momentos, dado ao fato que foram originalmente gerados na mesma execução (e então estavam no mesmo arquivo CSV), ao realizar a análise manual, foi possível identificar a correlação entre os mesmos. O ponto principal é que quando o crescimento de ambos diminui mais que um determinado valor, as deadline misses aconteciam, logo atuar sobre essa queda vale a tentativa, podemos identificar esses detalhes como na tabela abaixo.

Tabela 4.8 -Trecho do CSV da execução de Cópia de Regiões de Memória onde acontece perda de deadline.

Tempo	ITLB_ITLB_FLUSH	OFFCORE_REQUESTS_DEMAND_DATA_RD	Perda de deadline
2004990	9	8	0

2007572	5	6	0
2015175	4	4	1

Mas essa relação só é válida, quando esses contadores estão com o crescimento baixo, pois essa queda não apresentava uma perda de deadline nos momento seguintes, como apresentado na tabela abaixo.

Tabela 4.9 -Trecho do CSV da execução de Cópia de Regiões de Memória onde existe a queda no crescimento dos contadores, mas não acontece perda de deadline.

Tempo	ITLB_ITLB_FLUSH	OFFCORE_REQUESTS_DEMAND_DATA_RD	Perda de deadline
4697860	38	36	0
4707853	13	9	0
4717847	11	5	0
4727598	10	5	0
4736835	15	11	0

O outro problema agora é identificar os momentos em que a CPU estava executando com folga, mesmo com o DVFS ativo, ou seja, identificar um ponto que o DVFS poderia ser intensificado sem causar uma perda de deadline. A relação encontrado foi de que, se os dois eventos aumentassem seu crescimento em conjunto, e o evento

LD_BLOCKS_NO_SR estivesse baixo, a frequência poderia diminuir. Foi adicionado aqui o evento LD_BLOCKS_NO_SR na tentativa de evitar pontos em que não deveria ser feito uma redução da frequência, mas apenas os dois eventos descritos até agora permitiriam.

Tabela 4.10 -Trecho do CSV da execução de Cópia de Regiões de Memória adicionando o evento LD_BLOCKS_NO_SR na análise.

Tempo	LD_BLOCKS_NO_SR	ITLB_ITLB_FLUSH	OFFCORE_REQUESTS_DEMAND_DATA_RD	Perda de deadline
8255700	91	9	5	0
8265694	90	10	5	0
8275688	88	10	5	0
8277104	148	16	10	0
8286347	242	18	13	1

Logo, se fosse levado em consideração apenas os eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD, na quarta linha da tabela a frequência poderia ser reduzida, porém a relação deveria ser contrário, logo, considerando também o evento LD_BLOCKS_NO_SR, que tem aumento no seu crescimento no momento da decisão, podemos evitar a tomada de decisão errônea. Claro que, ao longo da execução, já sendo tomadas outras decisões de aumentar ou diminuir a frequência, o caso demonstrado pode ser diferente, por este motivo, no próximo Tópico (“5. Implementação da heurística”), estes pontos de corte são aprimorados.

5. IMPLEMENTAÇÃO DA HEURÍSTICA

Neste tópico será demonstrado o processo iterativo da busca pelos pontos de corte da heurística, ou seja, pontos de mudança no crescimento da contagem dos eventos que descrevem momentos em que se torna necessário aumentar a frequência da execução e de momentos em que é possível diminuir a frequência, de forma a diminuir o consumo do processador.

Além disso, também será apresentado o código da heurística desenvolvida para cada tipo de tarefa e a forma na qual ambas foram combinadas.

5.1. BUSCA PELOS PONTOS DE CORTE

Os pontos de corte iniciais de cada heurística foram definidos a partir dos próprios logs utilizados no Tópico “4.2.3 Análise Manual Guiada pelas Correlações”. Porém, estes pontos não funcionam tão bem quando a CPU não está com o DVFS ativado no fator 5/8, ou seja, apenas 62,5% da frequência total da CPU. Logo, o primeiro processo para ambas as heurísticas foi de observar o comportamento dos eventos enquanto o sistema estava funcionando com mais ou menos frequência.

Após esta primeira etapa, algumas definições comuns as heurísticas foram estabelecidas. Sendo a primeira uma nova variável, o *slowdown*, ou seja, toda vez que a frequência sobe, o sistema ativa um *slowdown* que afeta as decisões das duas próximas vezes que a CPU entrar em um *dispatch* ou *wait_next* (momentos de uma possível troca de contexto e o momento do fim de uma execução da thread periódica, respectivamente), de forma que nestas duas ocasiões, a frequência não pode ser reduzida. Isso se fez necessário para que a CPU possa se recuperar antes de uma perda de deadline

acontecer, visto que o sistema toma as decisões tentando prever a ocorrência de perdas de deadline, a decisão de subir a frequência necessita de pelo menos uma rodada de slowdown para surtir efeito. Após várias tentativas, duas rodadas foi o mais eficiente. Lembrando que, isso afeta apenas as decisões para diminuir a frequência, se por acaso a decisão for de subir a frequência novamente, o período do *slowdown* é renovado.

A segunda característica presente nas duas heurísticas é inerente do DVFS em sistemas com Hyper-threading. Ambas as CPUs lógicas possuem uma cópia do registrador, mas se o valor de *Duty-cycle* for diferente entre as CPUs lógicas, o valor utilizado é o maior Duty-cycle (como já explicado no tópico “3.2. Modulação de Clock”). Logo, para obter um melhor controle do DVFS, principalmente para respeitar casos em que uma CPU lógica possui maior uso que a outra CPU lógica que compartilha a CPU física com ela, quando uma CPU diminui sua frequência, ela é limitada a baixar no máximo um nível do fator aplicado de Duty-cycle a outra.

Nos próximos tópicos será descrito a busca dos pontos de corte da heurística em cada um dos algoritmos, porém o método de busca destes pontos foi similar. A essência do método foi a experimentação, onde então foram aplicados os pontos iniciais e iterativamente aprimorando-os, de forma a analisar o resultado visando uma baixa suficiente no fator para o uso do conjunto de Tarefas estabelecido, ao mesmo tempo que se evita a ocorrência de perda de deadlines.

5.1.1. FIBONACCI RECURSIVO

Iniciando o desenvolvimento da heurística para as tarefas de Fibonacci Recursivo, os pontos identificados na análises de seleção do evento foram selecionados, visto que os valores apresentados perto de uma perda Deadline com a frequência mais baixa, eram

sempre maiores que 1, foi imposta a necessidade de trabalhar com o valor do contador na sua forma de ponto flutuante (*Float* na linguagem C++). Dito isto, para os momentos de frequência reduzida, fatores 5 e 6 no Duty Cycle (62,5%% e 75%, respectivamente), foram estabelecidos pontos de corte entre zero e um no crescimento do contagem do evento.

Como apresentado na tabela abaixo, os valores realmente ficam entre 0 e 1 se ainda está longe de uma perda de deadline, sendo que quando não acontecem perdas de deadline na execução com o fator em 5, o valor fica sempre abaixo de 1. Sendo para o fator 6, os valores um pouco mais altos no pontos sem perdas de deadline, mas ainda entre 0 e 1.

Tabela 5.1 - Comportamento em fator 5 durante a execução, com perdas de deadline.

Tempo	LD_BLOCKS_NO_SR	Perdas de Deadline
4835162	0,08	0
4898514	0,06	0
4935072	1,21	0
4983912	5,81	0
5030128	6,38	1

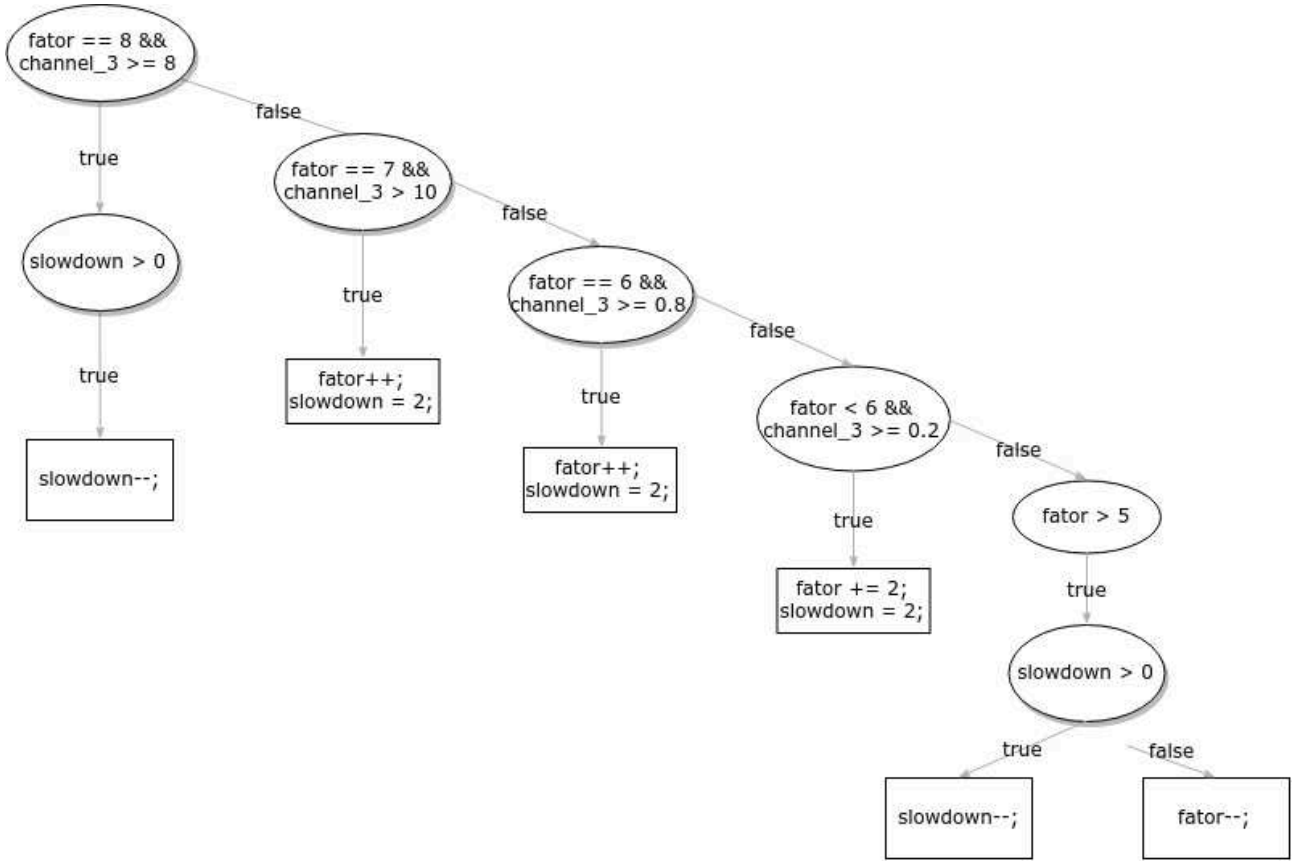
Além da especificação dos pontos de corte na frequências mais baixas, também foi estipulado um limite de fator, ou seja, no máximo o fator vai até 5, apresentando uma redução máxima até 62,5% da frequência de execução padrão da CPU. Isso foi estabelecido visto que recuperar a execução fica cada vez mais complicado quando a frequência está muita baixa, pois isto causa atrasos que, na maioria das vezes, não podem mais ser recuperados. além disso, também causaria um impacto grande no consumo, ter de retornar a frequência direto para 100% a fim de recuperar a execução a

tempo de não perder uma deadline. Então foi selecionado como limite o fator 5, pois durante os testes, foi o que melhor se comportou no quesito de recuperação analisando apenas o crescimento das ocorrências do evento.

Para estabelecer os pontos quando a frequência está alta, fatores 7 e 8 (82,5% e 100% respectivamente), foram executados testes com estes fatores aplicados no DVFS, nos quais, devido às características dos conjuntos de tarefas, não apresentavam perdas de deadline. Logo, apenas foram estabelecidos os valores padrões de comportamento em momentos sem perdas de deadline, ou seja, talvez estes pontos de corte estabelecidos no primeiro momento, não cubram os casos de quando o sistema aumenta a frequência a fim de recuperar uma execução que estava com a frequência mais baixa. Isto pode acarretar em uma redução repentina após o período de slowdown terminar, fazendo com que ainda aconteçam perdas de deadline. Para solucionar estas dúvidas a heurística foi sendo testada iterativamente até entrar em um comportamento suficiente, tanto com as características de determinismo temporal, quanto às características de consumo.

A versão final da heurśitica segue o seguinte diagrama de decisão (Figura 5.1), sendo então a versão final do código desenvolvido apresentado nas linhas 13 até 35 da Figura 5.5.

Figura 5.1 - Diagrama de tomada de decisão desenvolvido para a Heurística de Algoritmos Recursivos.



Para o fator 8 (100% da frequência), apenas é possível baixar a frequência, diminuindo o fator em 1, ou seja, diminuir o fator para 7, ou manter o fator em 8. O valor estabelecido foi 8 na contagem do evento LD_BLOCKS_NO_SR, onde o valor original foi dividido pelo tempo, logo seriam 8 ocorrências por microssegundo. Sendo menor que 8 representando uma baixa na frequência e maiores ou iguais a 8 a frequência é mantida, Isto pode ser verificado no “if” da linha 14 (Figura 5.5).

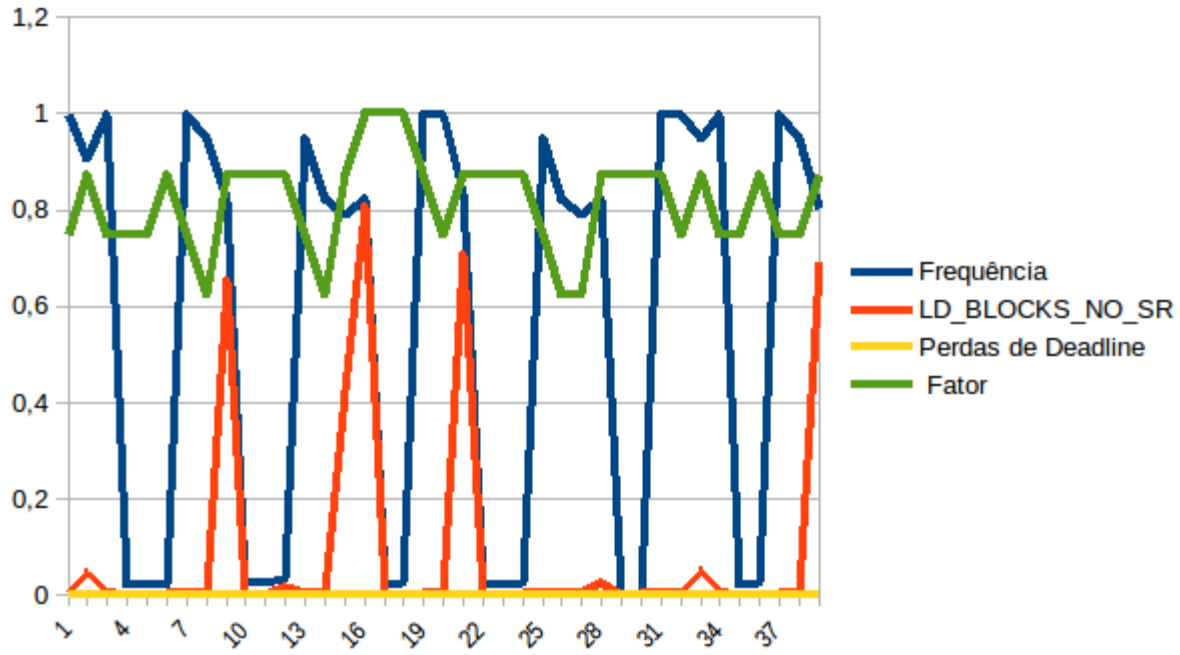
Para o fator 7, foi estabelecido que valores maiores que 10 representam uma necessidade de aumento na frequência, e valores menores ou iguais a 10 representam uma possibilidade de baixar a frequência. Isto pode ser verificado no “if” da linha 18 (Figura 5.5).

Já para o fator 6, o valor final foi de 0.8, sendo um valor maior ou igual a 0.8 representando uma necessidade de aumento na frequência, e valores menores ou iguais a 0.8 representando a possibilidade de baixar a frequência. Isto pode ser verificado no if da linha 22 (Figura 5.5).

E então, para o fator 5, o valor selecionado foi de 0.2, sendo um valor maior ou igual a 0.2 representando uma necessidade de aumento da frequência, e valores menores ou iguais a 0.2 representando que o fator pode ser mantido. Isto pode ser verificado no “if” da linha 26 (Figura 5.5).

Por fim, o código também é controlado pelo slowdown (descrito no tópico anterior) em todas as condições, além do controle da diferença entre os valores do fator nas CPUs que compartilham o mesmo core físico, feito nas linhas 97 até 105 (Figura 5.5).

Figura 5.2 - Gráfico Fator, frequência e perdas de deadline ao longo de um trecho da execução da Heurística sobre Fibonacci Recursivo.



Como pode-se observar, o comportamento da heurística ao longo da execução segue o que foi descrito no código, quando a linha vermelha apresenta valores baixos, o fotor é reduzido, e volta a crescer quando a linha vermelha tem um pico. Vale lembrar que, devido ao controle das diferenças de fator entre as CPUs que compartilham o mesmo core físico, e o slowdown, algumas vezes em que o fator poderia ser reduzido, a ação não aconteceu.

5.1.2. CÓPIA DE REGIÕES DE MEMÓRIA

No caso da heurística para as tarefas de Cópia de Regiões de Memória, por usar mais eventos para controle, na busca das correlações foi analisando o valor da contagem dos três eventos em conjunto. O básico da busca pelos pontos de corte já foi descrito no tópico “4.2.3.2. Cópia de Regiões de Memória”, principalmente nas análises dos contadores, onde já foi estipulado o comportamento dos mesmos em conjunto. Sendo que, quando o LD_BLOCKS_NO_SR apresenta um crescimento baixo, e os canais ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD apresentam um aumento no crescimento em relação a captura anterior, isto significa, para os casos analisados, que a frequência pode diminuir. Já quando os contadores ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD apresentam uma queda no crescimento em relação a captura anterior, isto significa, na frequência analisada até agora, que a frequência deve aumentar.

Da mesma forma realizada na heurística descrita no tópico anterior, também foi necessário analisar o comportamento destes contadores em frequências mais altas que o fator 5 (62,5% da frequência), ou seja, analisar os fatores 6, 7 e 8, para poder aferir os pontos de corte, e posteriormente, adaptar estes pontos para cumprir os objetivos da

heurística. Sendo esta operação feita de forma iterativa, onde em cada execução se observa os pontos de erro da heurística (ocorrência de perda de deadline) ou a perda de possibilidades de não diminuir a frequência.

Com os pontos já bem definidos, ocorreu um problema que não havia aparecido anteriormente, onde a heurística funcionava bem para o conjunto de tarefas mais leve (Tabela 4.1), porém ainda causava perdas de deadline no conjunto de tarefas mais pesado (Tabela 4.2). As ocorrências de perda de deadline no conjunto de tarefas mais pesado, ocorriam apenas nas CPUs 1 e 4, que apresentam os maiores usos de todo o conjunto, 90,674% e 83,952% respectivamente (Tabela 4.3).

Visto que a CPU 7 apresenta um uso de 78,177%, e a mesma não apresenta perdas de deadline, e que o primeiro nível de DVFS, o Duty Cycle de fator 7, reduz a frequência para 87,5% e o segundo, fator 6, reduz para 75%, a heurística teria de ser mais rígida nos pontos de corte, o que faria com que, agindo sobre as CPUs com o uso abaixo dos apresentados, não teriam ganhos na redução do consumo coerentes com seu uso, fazendo com o que a heurística perca seu desempenho. A alternativa para tal problema foi de limitar o uso das tarefas nas CPUs que estão utilizando a heurística, o que, após alguns testes foi observado como um limite de aproximadamente 80% no uso da CPU. Com isso, o conjunto de tarefas descrito na tabela 4.2, foi adaptado para o seguinte conjunto:

Tabela 5.2 - Conjunto de Tarefas 2 limitado a 80% do uso, 15 Threads.

Período*	Deadline*	WCET*	CPU
50000	50000	27098	0
25000	25000	5504	1
100000	100000	68919	2

100000	100000	64664	3
50000	50000	9110	4
200000	200000	105758	5
200000	200000	29326	6
200000	200000	67222	7
50000	50000	21151	6
50000	50000	700	6
50000	50000	6757	4
50000	50000	29329	1
50000	50000	8000	4
100000	100000	44566	7
25000	25000	8000	4

*Tempo em Microssegundos

Sendo também, adicionada uma nova thread na CPU 6, visando aumentar seu uso. Os novos usos são:

- CPU1: 5504/25000 + 29329/50000 = 80,674%
- CPU4: 9110/50000 + 6757/50000 + 8000/50000 + 8000/25000 = 79,734%
- CPU6: 29326/200000 + 21151/50000 + 700/50000 = 58,365%

Sendo a heurística desenvolvida para tarefas com alto uso de memória descrita no diagrama de decisões abaixo, e o código apresentado na Figura 5.5, da linha 37 até a linha 89. Um dos detalhes a serem destacados, que não é aplicada a heurística quando a CPU volta de um período de idle, pois ambos os contadores são 0, o que geraria problema ao compará-los com os valores anteriores.

Figura 5.3 (a) - Diagrama de tomada de decisão desenvolvido para a Heurística de alto uso de memória parte 1.

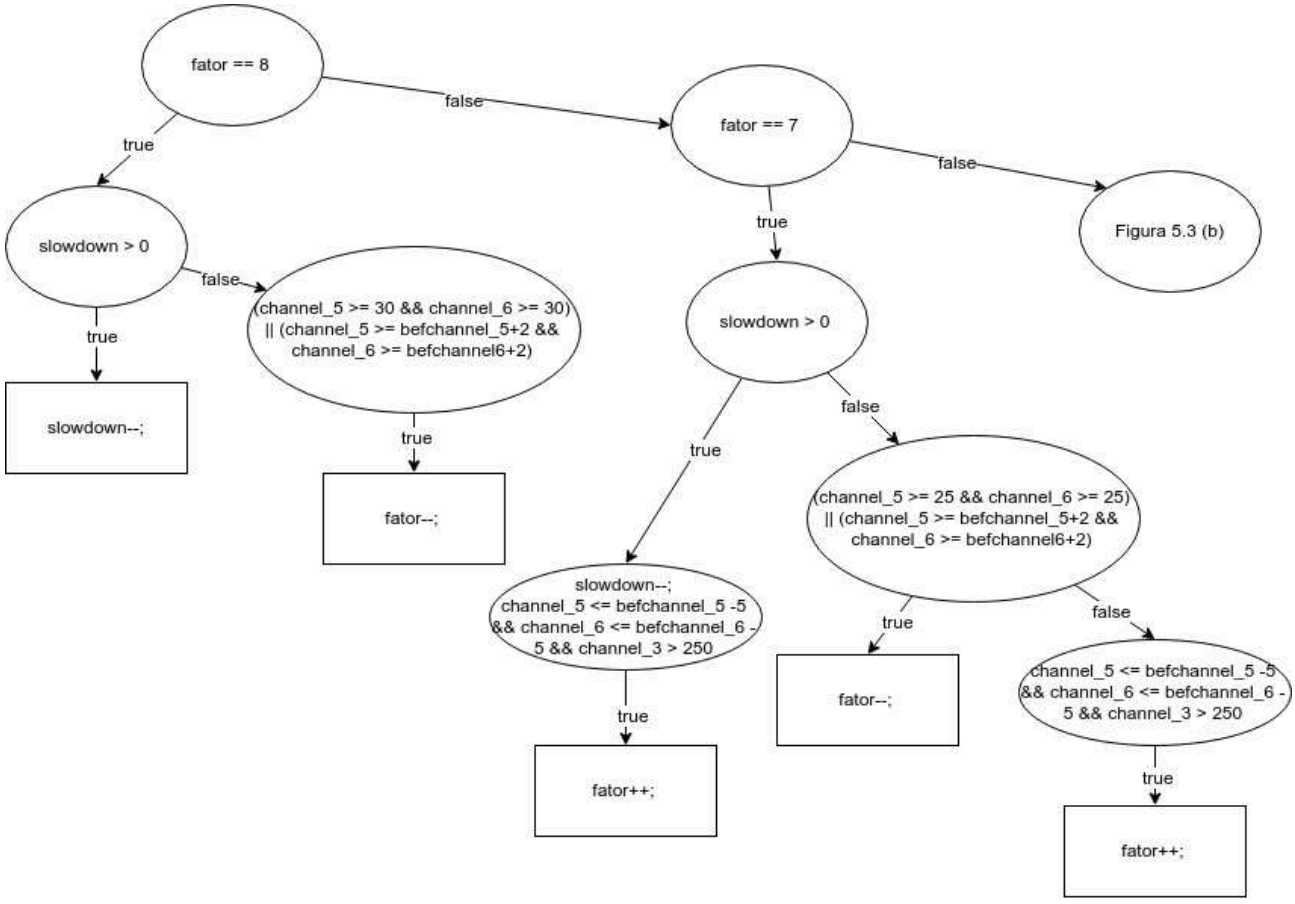
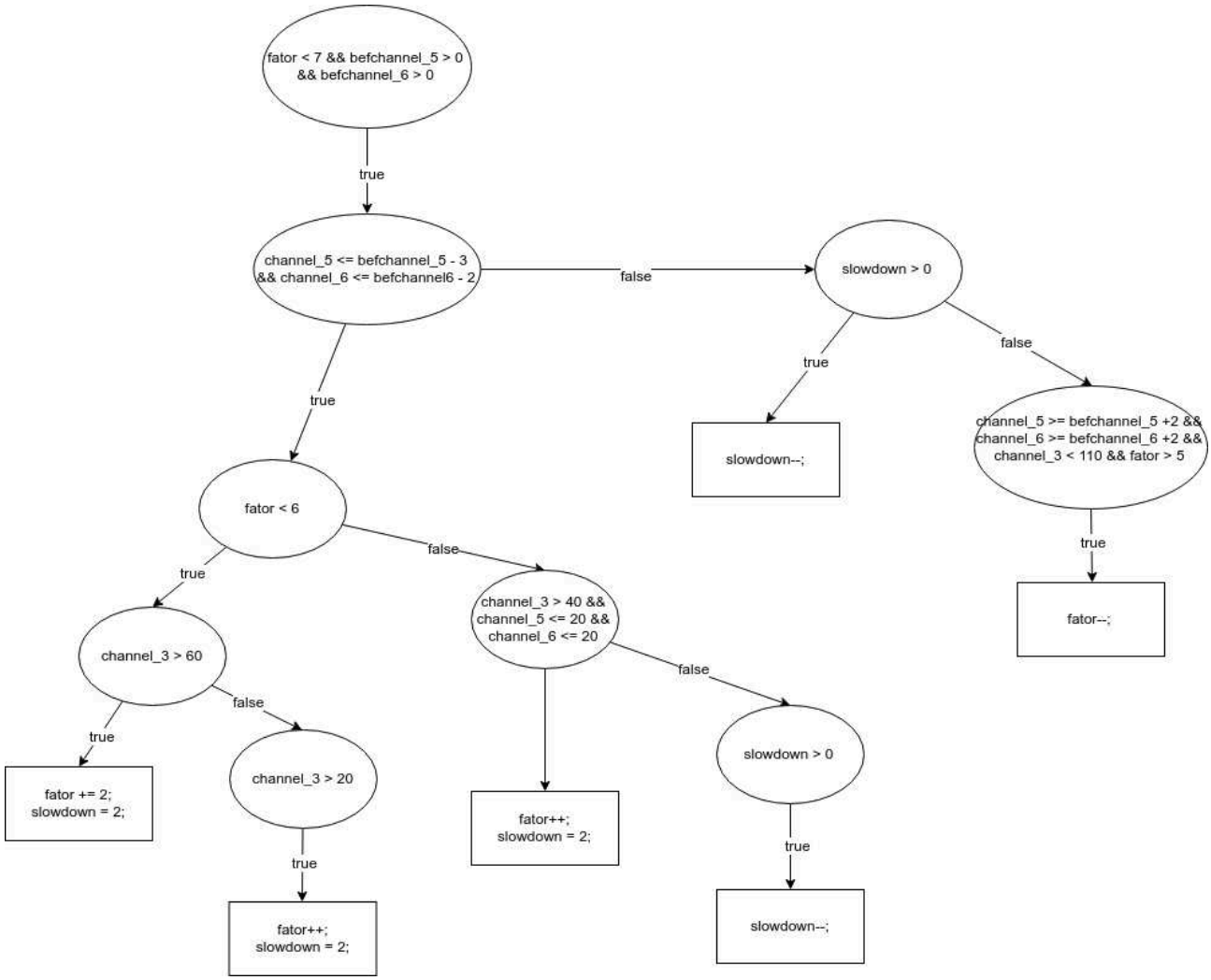


Figura 5.3 (b) - Diagrama de tomada de decisão desenvolvido para a Heurística de alto uso de memória parte 2.



Os valores selecionados para quando a CPU está utilizando 100% de sua frequência (fator 8), foi os contadores dos eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD apresentarem, ambos, um valor maior que 30, como foi explicado no tópico “4.2.3.2. Cópia de Regiões de Memória”, que se o crescimento destes contadores está alto, o sistema está executando, na maioria das vezes, com folga, e principalmente no fator 8, que apresenta o máximo do uso da frequência da CPU. Além disso, se os contadores estiverem abaixo de 30, mas ainda assim apresentarem um aumento maior ou igual a 2 em ambos os crescimentos, em relação a captura interior, também é possível reduzir o fator, e por consequência a frequência. Isto pode ser identificado no código nos “ifs” das linhas 40, 43 e 44.

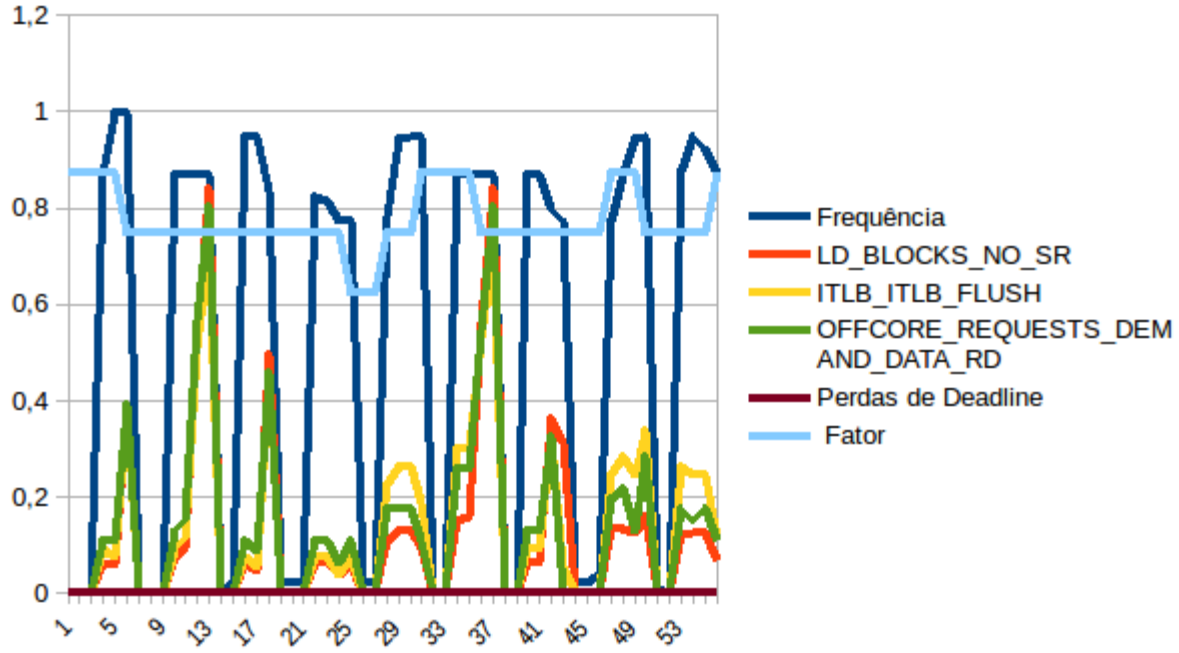
Para o fator 7, após o processo iterativo de adaptações dos valores, foi decido por subir a frequência se ambos os contadores dos eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD obtiverem uma queda de 5 ou mais em seu crescimento, e o contador do evento LD_BLOCKS_NO_SR apresente um crescimento de mais de 250. Para ocorrer um decréscimo na frequência, o slowdown deve estar desativado, e os contadores dos eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD devem ser maiores ou iguais a 25 ou apresentarem um aumento de 2 ou mais em relação ao crescimento da última captura . Isto pode ser verificado no código da linha 47 até a 59.

Já para o fator 6 e 5, se ocorrer uma queda de 3 ou mais no crescimento do contador do evento ITLB_ITLB_FLUSH e uma queda de 2 ou mais no crescimento do contador do evento OFFCORE_REQUESTS_DEMAND_DATA_RD, pode ser necessário aumentar a frequência. Logo, para decidir se a frequência irá subir ou não, o contador do evento LD_BLOCKS_NO_SR é utilizado, que no caso do fator 5, se apresentar um valor maior que 60, o fator é incrementado em 2, e se apresentar um valor maior que 20 e menor que 60, o fator é incrementado em 1. No caso do fator 6, se o crescimento do contadores dos eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD forem menores que 20 e o crescimento do contador LD_BLOCKS_NO_SR for maior que 40, o fator é incrementado em 1. Agora, se a primeira condição não for verdade, ou seja, não houve queda no crescimento dos contadores dos eventos ITLB_ITLB_FLUSH e OFFCORE_REQUESTS_DEMAND_DATA_RD, mas na verdade houve um aumento de 2 ou mais e o crescimento apontado pelo contador do evento LD_BLOCKS_NO_SR for

menor que 110, o fator pode ser decrementado. Isto pode ser verificado no código apresentado na Figura 5.5, da linha 60 até a 89.

Por fim, o código também é controlado pelo slowdown quando sobe para o fator 6 ou fator 7 (descrito no tópico “5.1 Busca Pelos Pontos de Corte”), além do controle da diferença entre os valores do fator nas CPUs que compartilham o mesmo core físico, feito nas linhas 97 até 105.

Figura 5.4 - Gráfico Fator, Frequência e Perdas de Deadline ao longo de um trecho da execução da Heurística sobre Cópia de Regiões de Memória.



Como pode ser observado no gŕafico acima, o fator somente é reduzido quando a linha vermelha está baixa, e a verde e amarela aumentaram. Vale lembrar que, devido ao controle das diferenças de fator entre as CPUs que compartilham o mesmo core físico, e o slowdown, algumas vezes em que o fator poderia ser reduzido, a ação não aconteceu.

5.2. CÓDIGO FINAL DA HEURÍSTICA

Com ambas as heurísticas desenvolvidas se torna interessante a ideia de combiná-las de forma a permitir que algumas CPUs possam executar algoritmos similares ao comportamento do Fibonacci Recursivo enquanto outras CPUs executam algoritmos similares a Cópia de Regiões de Memória, e todas com a heurística habilitada, realizando a diminuição no consumo, ao mesmo tempo que garante que o sistema não apresente perdas de deadline.

Para isso ser possível, é também necessário relembrar que, devido às características inerentes a versão arquitetural da PMU, apenas 7 eventos podem ser capturados por vez, ou seja, 3 eventos fixos e 4 eventos configuráveis. Porém, como as heurísticas utilizam no total apenas 3 eventos distintos, esse ponto não é um impedimento para combiná-las.

A fim de não prejudicar nenhuma das heurísticas, um ponto que diferencia qual delas deve ser executada precisa ser identificado. Dessa forma, foi utilizado a diferença no comportamento do evento LD_BLOCKS_NO_SR, que apresenta valores bem mais baixos para a heurística de algoritmos que exaltam a característica de recursividade, se comparados com os valores apresentados pela heurística de algoritmos que utilizam como principais tarefas as relacionadas com o uso da memória. Sendo assim, uma análise foi feita, para identificar os pontos de maior valor da contagem do evento executando o algoritmo Fibonacci Recursivo e os menores valores para o algoritmo do Cópia de Regiões de Memória.

Um valor que aparentemente pareceria não afetar o desempenho de ambas, e principalmente não faria com que perdas de deadline acontecessem foi de 30, visto que

esse valor quase nunca era atingido pelo Fibonacci Recursivo, e quando era, a heurística já teria tomado uma decisão quando o valor estava aumentando até chegar a 30. E para a heurística de Cópia de Regiões de Memória valores próximos a 30 são específicos de quando a CPU está com a frequência baixa, mas valores menores ou iguais a 30 são raros. Para chegar no valor apresentado na heurística final, também foram testados valores próximos a 30, de forma que o mais eficiente nos testes foi o 20. Logo, com este valor, as heurísticas apresentaram o mesmo desempenho que se executadas sozinhas.

Figura 5.5 - Código Final - Heurísticas Combinadas

```
001: if (heuristic) {
002:    time_last_capture = _thread_monitor->lastCapture(Machine::cpu_id(), 7);
003:    diff_time = thread_monitor->time() - time_last_capture;
004:      if (diff time > 999) {
005:      // LD_BLOCKS_NO_SR
006:      channel_3 = (PMU::read(3) - _thread_monitor->lastCapture(Machine::cpu_id(), 3) * 1.0)
/ diff_time;
007:      // ITLB_ITLB_FLUSH
008:      channel_5 = (PMU::read(3) - _thread_monitor->lastCapture(Machine::cpu_id(), 3)) /
diff_time;
009:      // OFFCORE_REQUESTS_DEMAND_DATA_RD
010:      channel_6 = (PMU::read(3) - _thread_monitor->lastCapture(Machine::cpu_id(), 3)) /
diff_time;
011:      // Recursive Tasks Heuristic
012:      if (channel_3 <= 20) {
013:        // Factor 8, DVFS Disable, Frequency 100%
014:        if (channel_3 >= 8 && _clock_factor[Machine::cpu_id()] == 8) {
015:          if (_slowdown[Machine::cpu_id()] > 0)
016:             _slowdown[Machine::cpu_id()]--;
017:        // Factor 7, Frequency in 87,5%
018:        } else if (channel_3 > 10 && _clock_factor[Machine::cpu_id()] == 7) {
019:          _clock_factor[Machine::cpu_id()]++;
020:          _slowdown[Machine::cpu_id()] = 2;
021:        // Factor 6, Frequency in 75%
022:        } else if (channel_3 >= 0.8 && _clock_factor[Machine::cpu_id()] == 6) {
023:          _clock_factor[Machine::cpu_id()]++;
024:          _slowdown[Machine::cpu_id()] = 2;
025:        // Factor 5, Frequency in 62,5%
026:        } else if (channel_3 >= 0.2 && _clock_factor[Machine::cpu_id()] < 6) {
027:          _clock_factor[Machine::cpu_id()]+= 2;
028:          _slowdown[Machine::cpu_id()] = 2;
029:        } else if (_clock_factor[Machine::cpu_id()] > 5) {
030:          if (_slowdown[Machine::cpu_id()] > 0)
031:            _slowdown[Machine::cpu_id()]--;
032:          else {
```

```
033:           _clock_factor[Machine::cpu_id()]--;
034:         }
035:       }
036:     }
037:     // Memory Copy Heuristic
038:     else if (channel_6 > 0 || channel_5 > 0) {
039:        // Factor in 8, DVFS Disable, Frequency in 100%
040:        if (_clock_factor[Machine::cpu_id()] == 8) {
041:          if (_slowdown[Machine::cpu_id()] > 0) {
042:            _slowdown[Machine::cpu_id()]--;
043:          } else if ((channel_5 >= 30 && channel_6 >= 30) ||
044:            (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 && channel_6 >=
_bef_channel6[Machine::cpu_id()]+2)) {
045:            _clock_factor[Machine::cpu_id()]--;
046:          }
047:        // Factor 7, DVFS enable, Frequency in 87,5%
048:        } else if (_clock_factor[Machine::cpu_id()] == 7) {
049:          if (_slowdown[Machine::cpu_id()] > 0) {
050:            _slowdown[Machine::cpu_id()]--;
051:            if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 5 && channel_6 <=
_bef_channel6[Machine::cpu_id()] - 5 && channel_3 > 250) {
052:              _clock_factor[Machine::cpu_id()]++;
053:            }
054:          } else if ((channel_5 >= 25 && channel_6 >= 25) ||
055:            (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 && channel_6 >=
_bef_channel6[Machine::cpu_id()]+2)) {
056:            _clock_factor[Machine::cpu_id()]--;
057:          } else if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 5 && channel_6 <= 061:
_bef_channel6[Machine::cpu_id()] - 5 && channel_3 > 250) {
058:            _clock_factor[Machine::cpu_id()]++;
059:          }
060:         // Factor less than 7
061:         } else if (_clock_factor[Machine::cpu_id()] < 7 &&
(_bef_channel5[Machine::cpu_id()] > 0 || _bef_channel6[Machine::cpu_id()] > 0)) {
062:           if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 3 && channel_6 <=
_bef_channel6[Machine::cpu_id()] - 2) {
063:            // Factor 5, Frequency in 62,5%
064:            if (_clock_factor[Machine::cpu_id()] < 6) {
065:              if (channel_3 > 60) {
066:                _clock_factor[Machine::cpu_id()]+= 2;
067:                _slowdown[Machine::cpu_id()] = 2;
068:              } else if (channel_3 > 20) {
069:                _clock_factor[Machine::cpu_id()]++;
070:                _slowdown[Machine::cpu_id()] = 2;
071:              }
072:            // Factor 6, Frequency in 75%
073:            } else if (channel_3 > 40 && channel_6 <= 20 && channel_5 <= 20) {
074:               _clock_factor[Machine::cpu_id()]++;
075:               _slowdown[Machine::cpu_id()] = 2;
076:            } else if (_slowdown[Machine::cpu_id()] > 0) {
077:              _slowdown[Machine::cpu_id()]--;
078:            }
079:          } else {
```

```
080:             if (_slowdown[Machine::cpu_id()] > 0) {
081:                _slowdown[Machine::cpu_id()]--;
082:             } else if (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 && channel_6 >=
083:               _bef_channel6[Machine::cpu_id()]+2 && channel_3 < 110 &&
084:               _clock_factor[Machine::cpu_id()] > 5) {
085:                 _clock_factor[Machine::cpu_id()]--;
086:             }
087:           }
088:         }
089:       }
090:       // Update the old values of channel 5 and 6
091:       _bef_channel6[Machine::cpu_id()] = channel_6;
092:       _bef_channel5[Machine::cpu_id()] = channel_5;
093:     }
094:   }
095:   // Set the difference between both Logical CPUs that share the same Physical Core to a
096:   // maximum of 1 factor.
097:   if (Machine::cpu_id() % 2) {
098:     if ((_clock_factor[Machine::cpu_id()-1] - 1) > _clock_factor[Machine::cpu_id()]) {
099:       _clock_factor[Machine::cpu_id()] = _clock_factor[Machine::cpu_id()-1] -1;
100:     }
101:   } else {
102:     if ((_clock_factor[Machine::cpu_id()+1] - 1) > _clock_factor[Machine::cpu_id()]) {
103:       _clock_factor[Machine::cpu_id()] = _clock_factor[Machine::cpu_id()+1] -1;
104:     }
105:   }
106:   // Apply the Duty Cycle Factor
107:   CPU::clock((CPU::clock()/8 * _clock_factor[Machine::cpu_id()]));
108: }
```

6. RESULTADOS

Uma das maneiras de medir o consumo para validar a heurística, agora completa, é comparar o consumo de uma execução idêntica, com e sem a heurística habilitada. Para esta tarefa, foram selecionadas duas maneiras para aferir o consumo, uma delas é através do aparelho Fluke, descrito no Tópico “2.10. Materiais Utilizados”, e a segunda é através da interface RAPL, descrita no Tópico “3.3. Leitura de Consumo Energético”.

Neste tópico, o foco é demonstrar a comparação do consumo em ambas as alternativas, de forma a validar a heurística desenvolvida. Nos tópicos seguintes é apresentado a comparação dos consumo dos dois conjuntos de tarefas descritos nas Tabelas 4.1 e 5.2, sendo duas análises para cada, uma executando as operações de Fibonacci Recursivo, e uma executando as operações de Cópia de Regiões de Memória.

6.1. ANÁLISE DE CONSUMO FLUKE

Utilizando o sistema Fluke para realizar as aferições de consumo obtém-se a diferença do consumo da máquina como um todo, devido ao fato de que o Fluke mede o consumo a partir da fonte do computador. Logo, mesmo o foco da heurística sendo principalmente o processador, ainda existe um impacto nos demais componentes, além do fato de que o processador representa boa parte consumo de um computador.

Além disso, o sistema Fluke também fornece relatórios das medições, além de algumas estatísticas, aqui utilizadas para avaliar os ganhos da heurística.

6.1.1. FIBONACCI RECURSIVO

- Relatórios do Fluke sobre o conjunto descrito na Tabela 4.1 executando com a heurística desativada:

Figura 6.1 - Consumo ao longo da execução Fibonacci, Conjunto 1 e Heurística desativada (mínimo do gráfico de 60W e máximo de 110W).

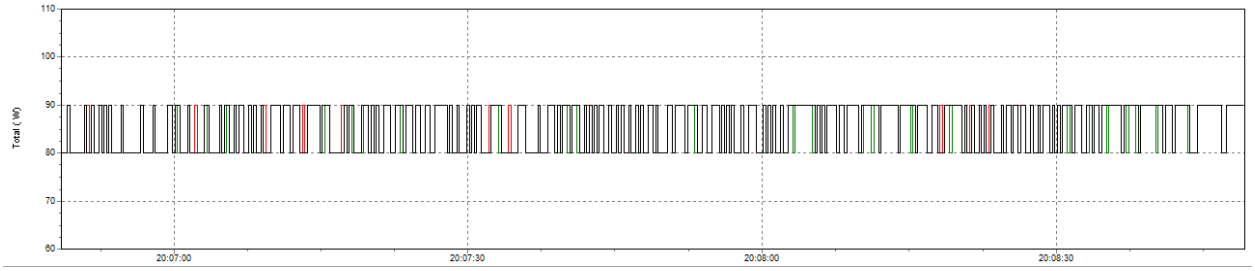


Figura 6.2 - Frequência das medições da execução Fibonacci, Conjunto 1 e Heurística desativada.

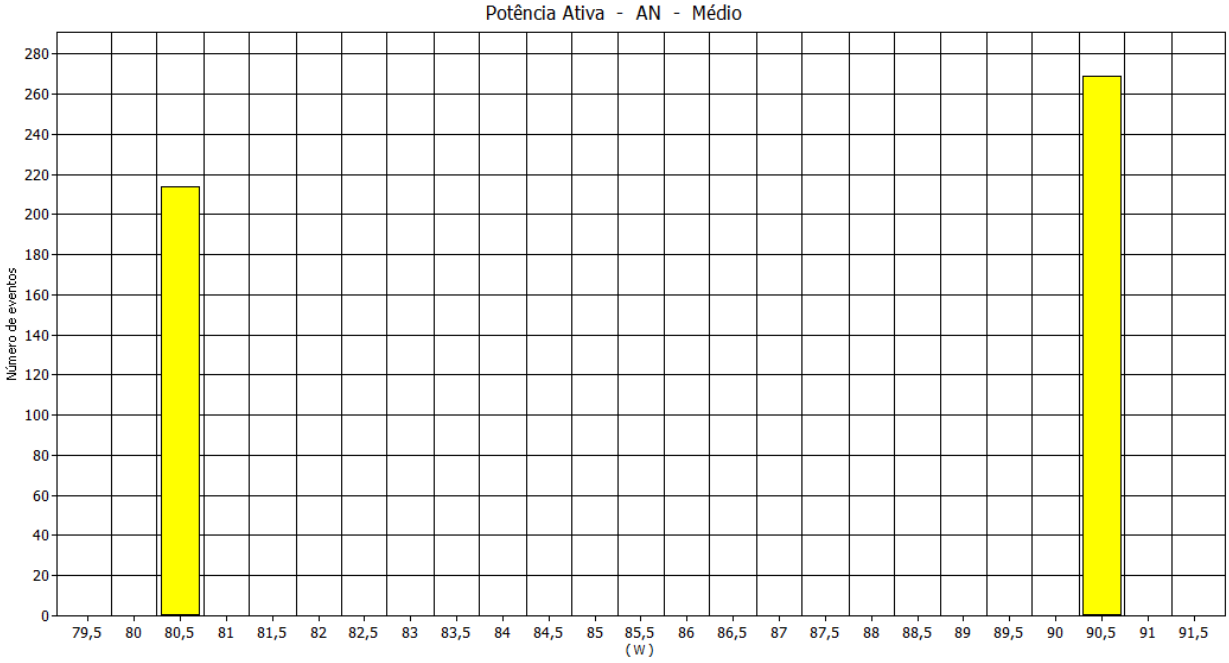


Figura 6.3 - Estatísticas das medições da execução Fibonacci, Conjunto 1 e Heurística desativada.

Sumário

De	03/10/2018 20:06:48
Para	03/10/2018 20:08:49
Valor máximo	90 W
Em	03/10/2018 20:06:49
Valor mínimo	80 W
Em	03/10/2018 20:06:48
μ	85,5694 W
s	4,97263 W
5% percentil	80 W
95% percentil	90 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Relatórios do Fluke sobre o conjunto descrito na Tabela 4.1 executando com a heurística ativada:

Figura 6.4 - Consumo ao longo da execução Fibonacci, Conjunto 1 e Heurística ativada (mínimo do gráfico de 60W e máximo de 110W).

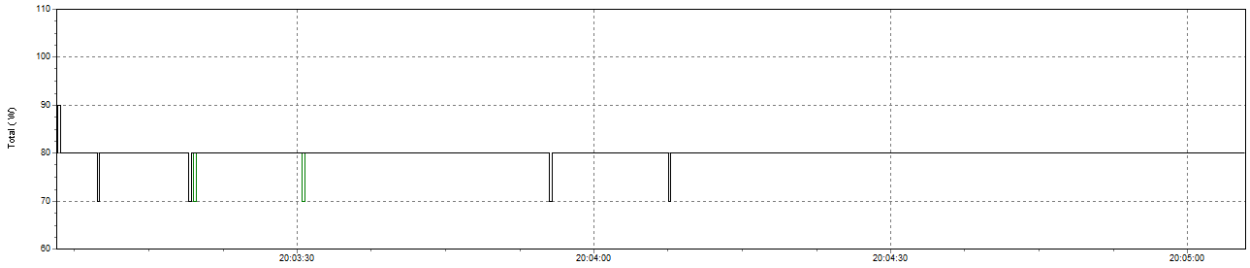


Figura 6.5 - Frequência das medições da execução Fibonacci, Conjunto 1 e Heurística ativada.

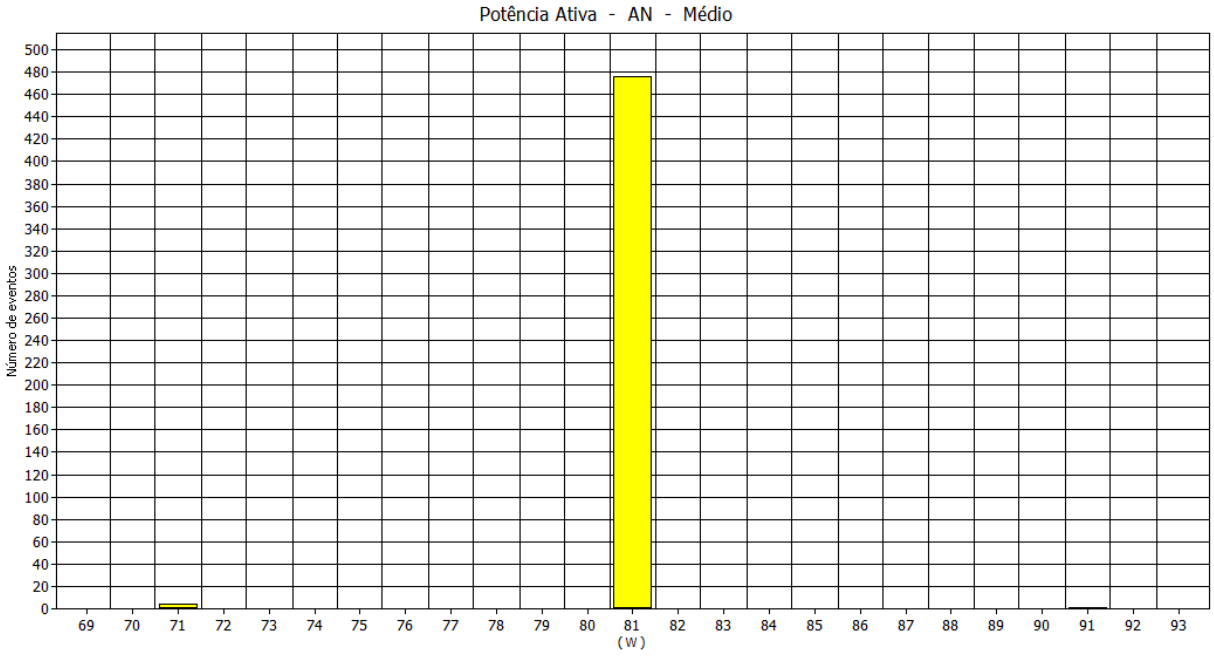


Figura 6.6 - Estatísticas das medições da execução Fibonacci, Conjunto 1 e Heurística ativada.

De	03/10/2018 20:03:05
Para	03/10/2018 20:05:05
Valor máximo	90 W
Em	03/10/2018 20:03:06
Valor mínimo	70 W
Em	03/10/2018 20:03:10
μ	79,9376 W
s	1,01871 W
5% percentil	80 W
95% percentil	80 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Comparativo de desempenho para o Conjunto 1:

Tabela 6.1 - Comparativo Consumo com e sem Heurística, Fibonacci, Conjunto de Tarefas 1.

	Média	Máximo	Mínimo
Sem Heurística	85,5694 W	90,0 W	80,0 W
Com Heurística	79,9376 W	90,0 W	70,0 W

Diferença	5.6318 W	0 W	10,0 W

É possível observar que, sobre o consumo médio da máquina, a heurística teve um impacto de 5,6318 W em 2 minutos de execução, gerando um ganho de aproximadamente 6.5% (6.581558460815899%). Além disso, fez com que a execução quase nunca atingisse os 90W de consumo, apenas atingindo esse valor no início, onde ainda não havia impactado no sistema, visto que ela diminui no máximo um do fator por cada vez que uma tarefa é executada ou o alarme é disparado.

- Relatórios do Fluke sobre o conjunto descrito na Tabela 5.2 executando com a heurística desativada:

Figura 6.7 - Consumo ao longo da execução Fibonacci, Conjunto 2 e Heurística desativada.

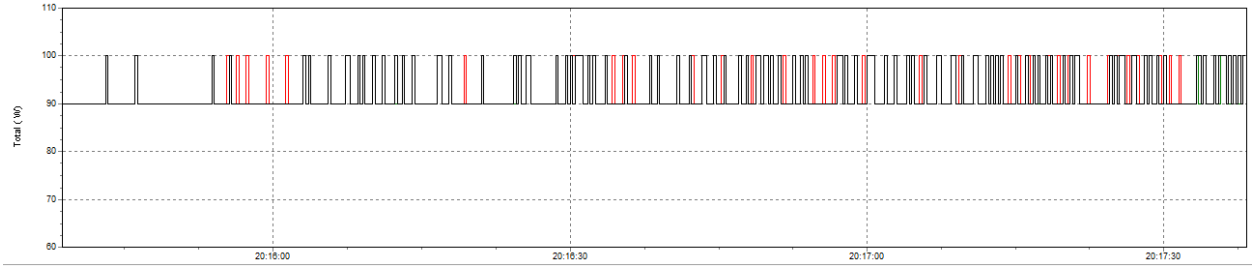


Figura 6.8 - Frequência das medições da execução Fibonacci, Conjunto 2 e Heurística desativada.

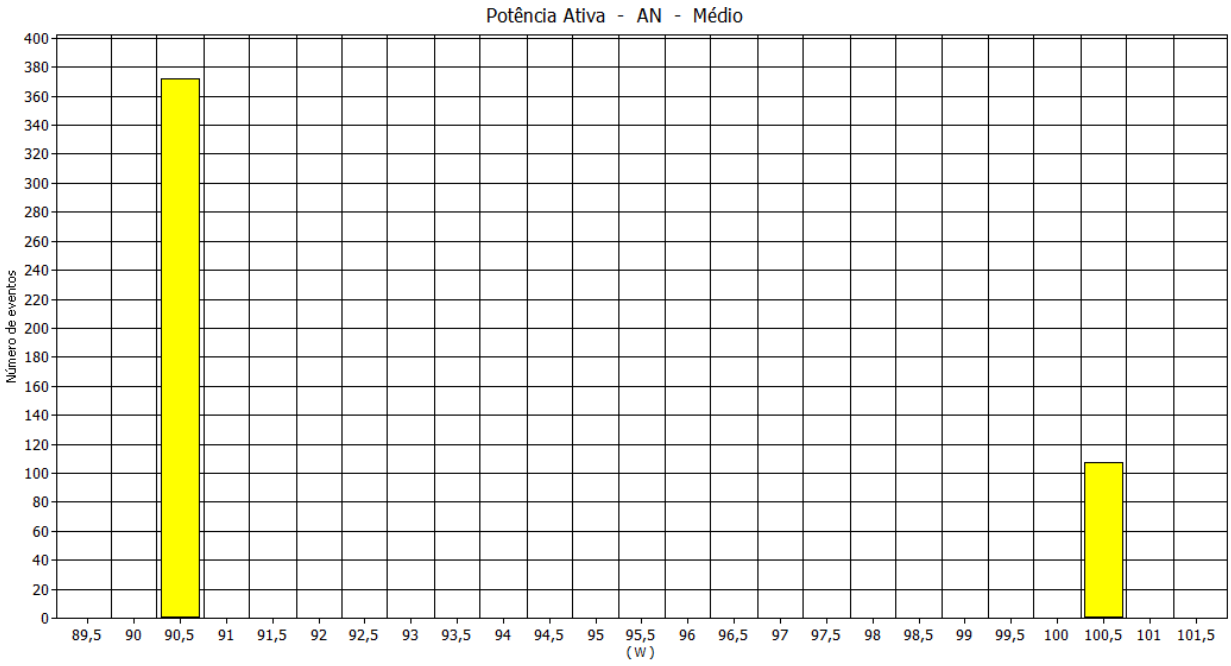


Figura 6.9 - Estatísticas das medições da execução Fibonacci, Conjunto 2 e Heurística desativada.

Sumário

De	03/10/2018 20:15:38
Para	03/10/2018 20:17:38
Valor máximo	100 W
Em	03/10/2018 20:15:43
Valor mínimo	90 W
Em	03/10/2018 20:15:38
μ	92,2338 W
s	4,16948 W
5% percentil	90 W
95% percentil	100 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Relatórios do Fluke sobre o conjunto descrito na Tabela 5.2 executando com a heurística ativada:

Figura 6.10 - Consumo ao longo da execução Fibonacci, Conjunto 2 e Heurística ativada.

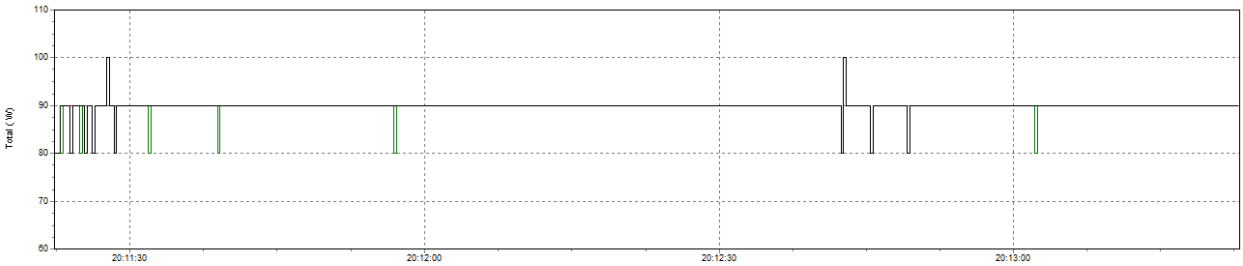


Figura 6.11 - Frequência das medições da execução Fibonacci, Conjunto 2 e Heurística ativada.

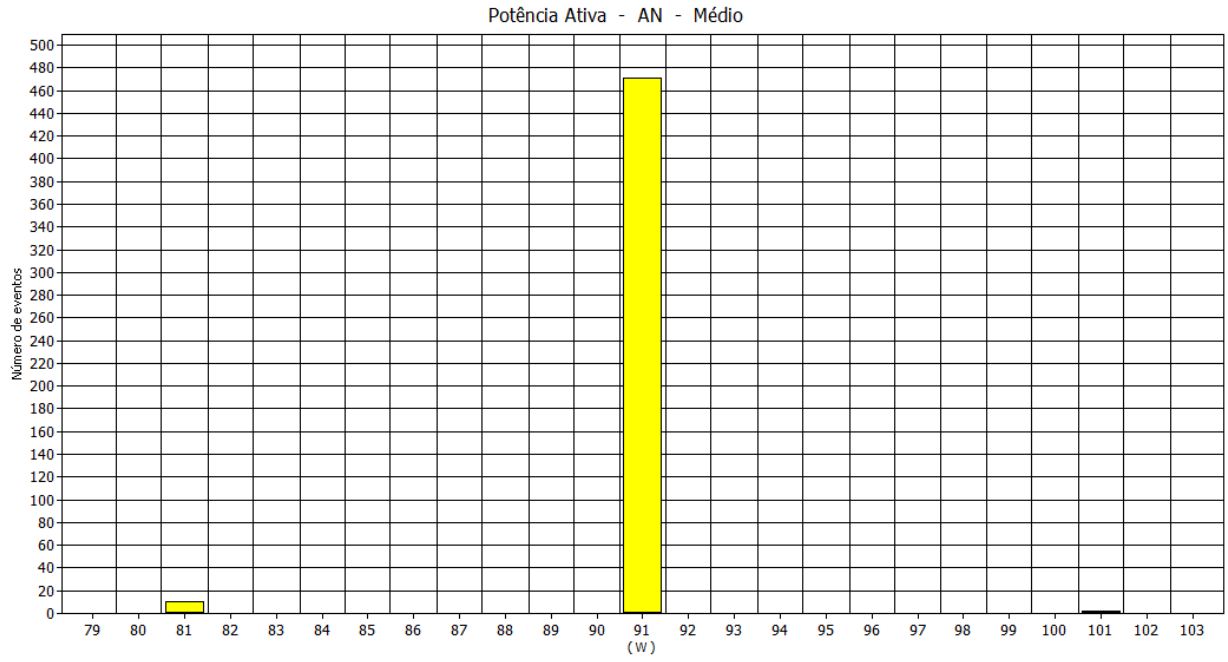


Figura 6.12 - Estatísticas das medições da execução Fibonacci, Conjunto 2 e Heurística ativada.

De	03/10/2018 20:11:22
Para	03/10/2018 20:13:22
Valor máximo	100 W
Em	03/10/2018 20:11:27
Valor mínimo	80 W
Em	03/10/2018 20:11:22
μ	89,8344 W
s	1,56912 W
5% percentil	90 W
95% percentil	90 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Comparativo de desempenho para o Conjunto 2:

Tabela 6.2 - Comparativo Consumo com e sem Heurística, Fibonacci, Conjunto de Tarefas 2.

	Média	Máximo	Mínimo
Sem Heurística	92,2338 W	100,0 W	90,0 W
Com Heurística	89,8344 W	100,0 W	80,0 W
Diferença	2.3994 W	0 W	10,0 W

Pode se concluir então, que a heurística executando um conjunto de tarefas de uso mais alto (Conjunto descrito na Tabela 5.2), obteve uma redução de aproximadamente 2.4W em dois minutos de execução, o que remete em uma redução de aproximadamente 2.6% (2.601432446673562%) no consumo da máquina como um todo. Sendo também reduzidas a quantidade de vezes em que o valor mais alto do consumo é atingido, ou seja, a heurística evita, na maioria dos casos, um consumo instantâneo alto desnecessário, lembrando que isto também ocorreu para conjunto 1 de tarefas, apresentado mais acima neste mesmo tópico.

Isso leva à conclusão de que os ganhos na redução do consumo são melhores, para a heurística executando tarefas recursivas, quando existe uma folga maior no uso do conjunto de tarefas, permitindo que a mesma explore esta característica.

6.1.2. CÓPIA DE REGIÕES DE MEMÓRIA

- Relatórios do Fluke sobre o conjunto descrito na Tabela 4.1 executando com a heurística desativada:

Figura 6.13 - Consumo ao longo da execução de Cópia de Regiões de Memória, Conjunto 1 e Heurística desativada.

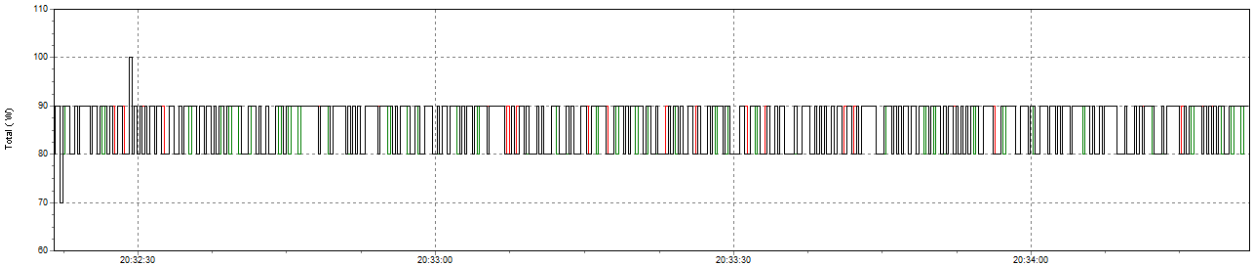


Figura 6.14 - Frequência das medições da execução de Cópia de Regiões de Memória, Conjunto 1 e Heurística desativada.

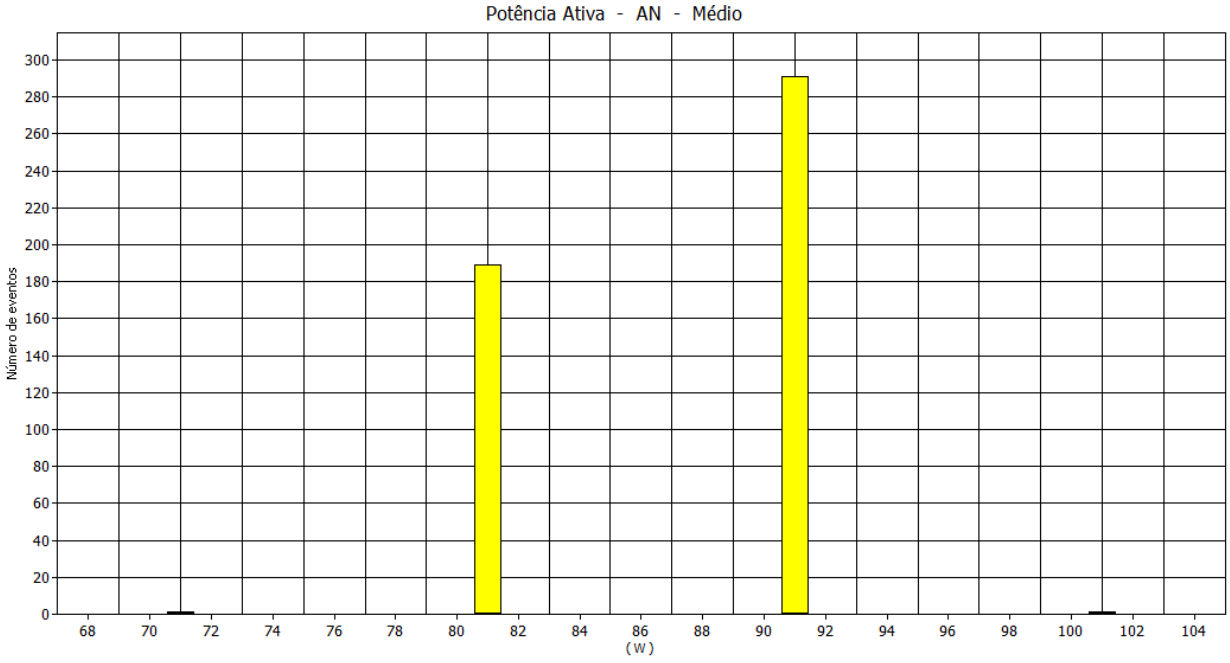


Figura 6.15 - Estatísticas das medições da execução de Cópia de Regiões de Memória, Conjunto 1 e Heurística desativada.

Sumário

De	03/10/2018 20:32:21
Para	03/10/2018 20:34:22
Valor máximo	100 W
Em	03/10/2018 20:32:29
Valor mínimo	70 W
Em	03/10/2018 20:32:22
μ	86,0581 W
s	4,97611 W
5% percentil	80 W
95% percentil	90 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Relatórios do Fluke sobre o conjunto descrito na Tabela 4.1 executando com a heurística ativada:

Figura 6.16 - Consumo ao longo da execução de Cópia de Regiões de Memória, Conjunto 1 e Heurística ativada.

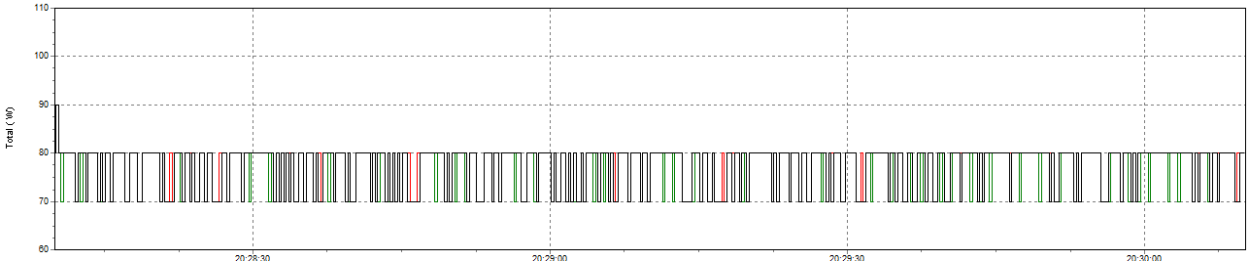


Figura 6.17 - Frequência das medições da execução de Cópia de Regiões de Memória, Conjunto 1 e Heurística ativada.

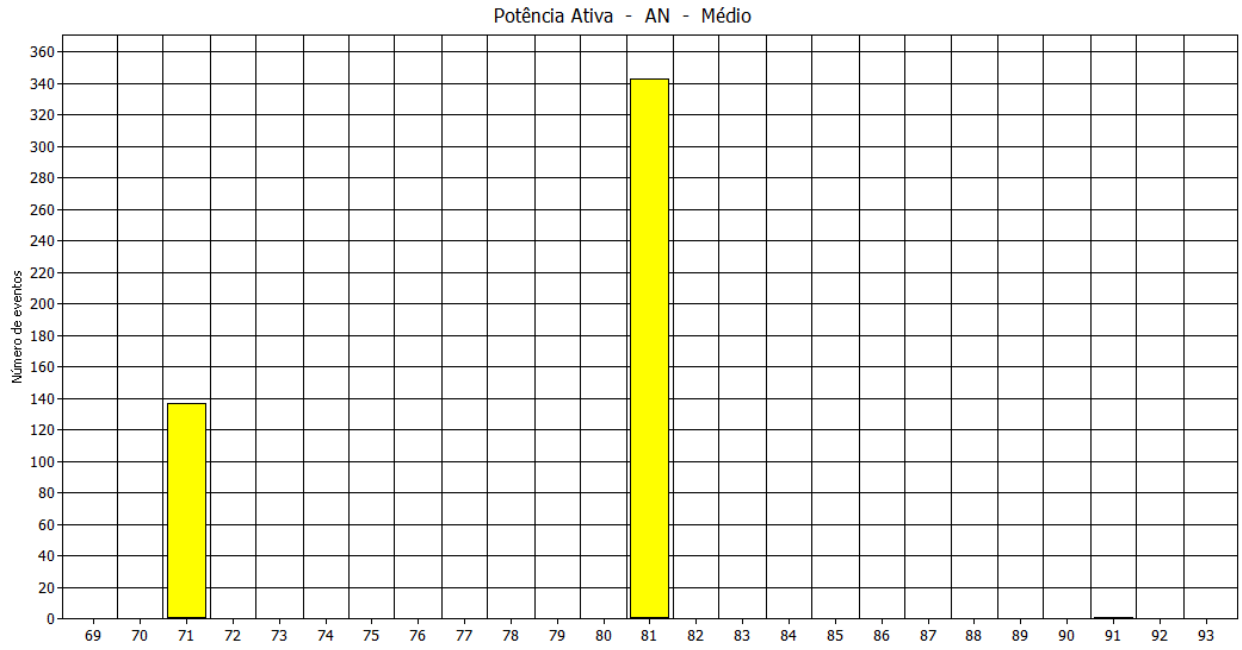


Figura 6.18 - Estatísticas das medições da execução de Cópia de Regiões de Memória, Conjunto 1 e Heurística ativada.

Sumário

De	03/10/2018 20:28:09
Para	03/10/2018 20:30:10
Valor máximo	90 W
Em	03/10/2018 20:28:10
Valor mínimo	70 W
Em	03/10/2018 20:28:12
μ	77,1726 W
s	4,554 W
5% percentil	70 W
95% percentil	80 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Comparativo de desempenho para o Conjunto 1:

Tabela 6.3 - Comparativo Consumo com e sem Heurística, Cópia de Regiões de Memória, Conjunto de Tarefas 1.

	Média	Máximo	Mínimo
Sem Heurística	86,0581 W	100,0 W	70,0 W
Com Heurística	77,1726 W	90,0 W	70,0 W

Diferença	8,8855 W	10,0 W	0 W

Pode se perceber que, a heurística apresentou um bom ganho na redução do consumo, mesmo analisando o consumo total do computador. O ganho apresentado foi de 8,8855 W na média de dois minutos de execução, o que representa aproximadamente um ganho de 10,3% (10.32500136535666%), devido, principalmente, da redução do consumo máximo de 100 W para 90 W.

- Relatórios do Fluke sobre o conjunto descrito na Tabela 5.2 executando com a heurística desativada:

Figura 6.19 - Consumo ao longo da execução de Cópias de Regiões de Memória, Conjunto 2 e Heurística desativada.

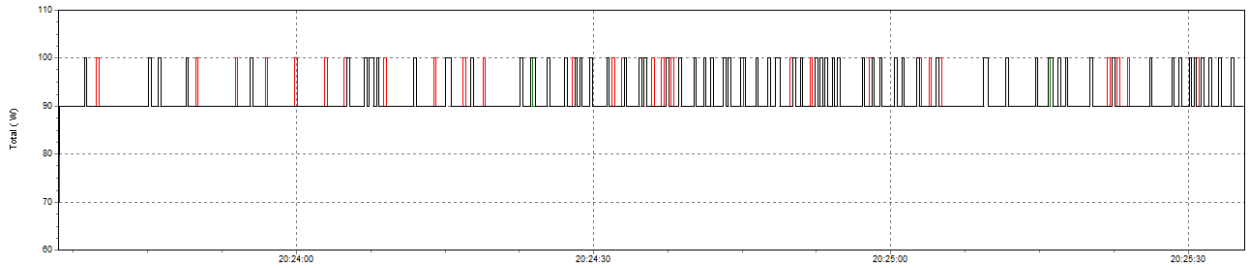


Figura 6.20 - Frequência das medições da execução de Cópias de Regiões de Memória, Conjunto 2 e Heurística desativada.

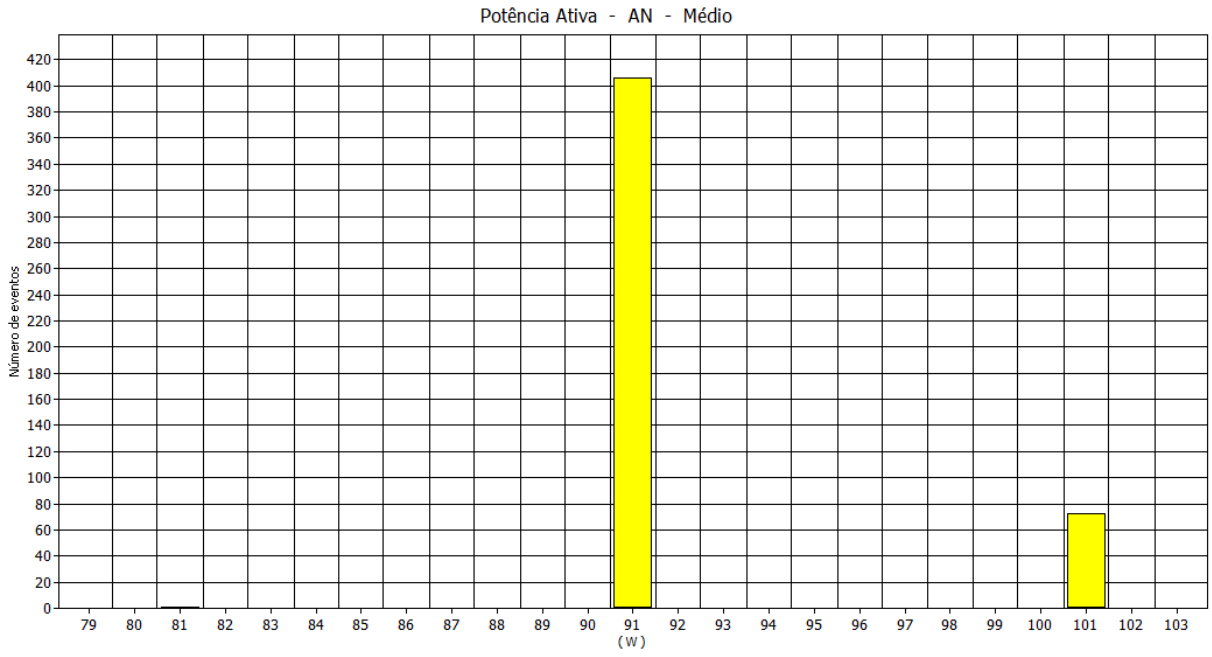


Figura 6.21 - Estatísticas das medições de da execução de Cópias de Regiões de Memória, Conjunto 2 e Heurística desativada.

Sumário

De	03/10/2018 20:23:36
Para	03/10/2018 20:25:35
Valor máximo	100 W
Em	03/10/2018 20:23:38
Valor mínimo	80 W
Em	03/10/2018 20:23:36
μ	91,4823 W
s	3,61529 W
5% percentil	90 W
95% percentil	100 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Relatórios do Fluke sobre o conjunto descrito na Tabela 5.2 executando com a heurística ativada:

Figura 6.22 - Consumo ao longo da execução de Cópias de Regiões de Memória, Conjunto 2 e Heurística ativada.

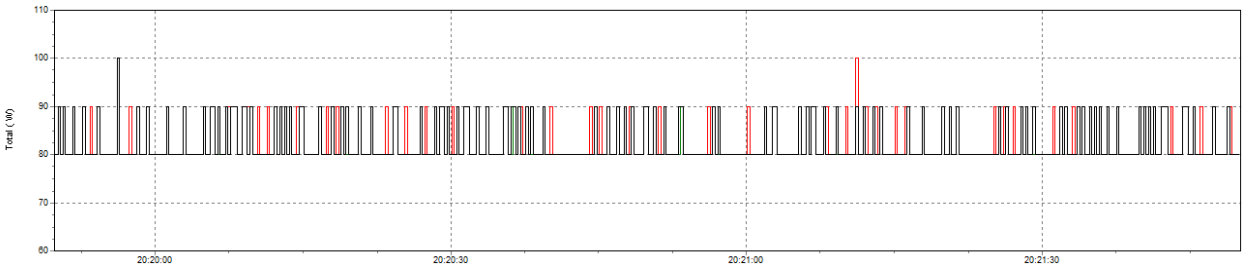


Figura 6.23 - Frequência das medições da execução de Cópias de Regiões de Memória, Conjunto 2 e Heurística ativada.

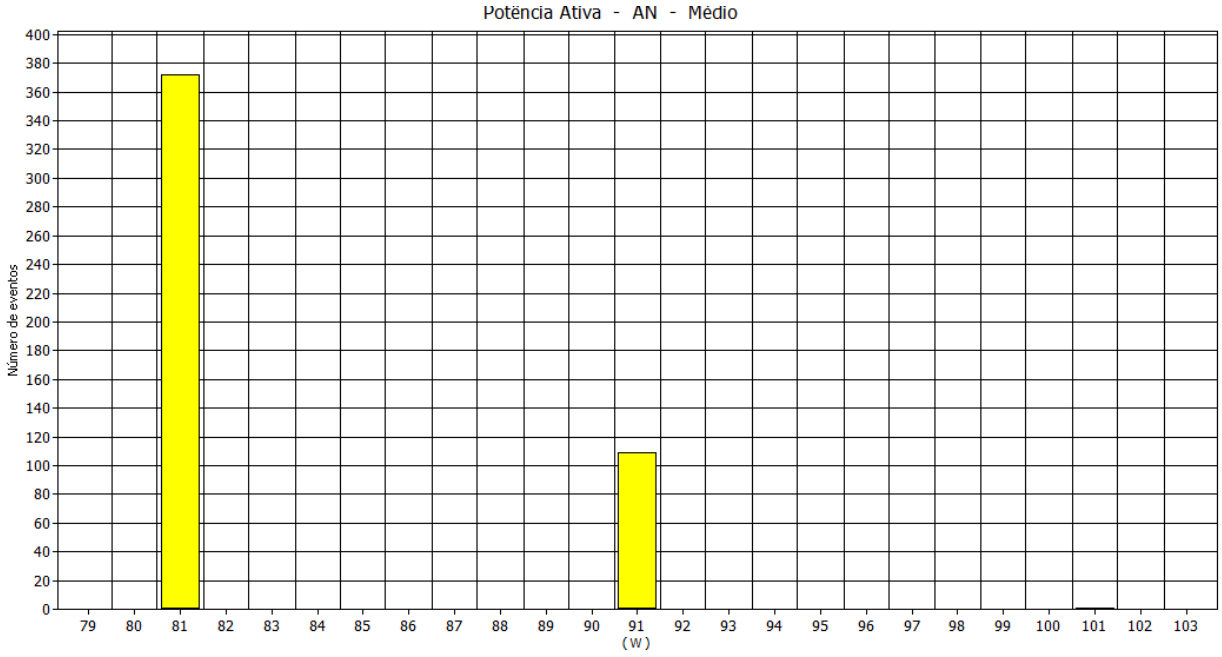


Figura 6.24 - Estatísticas das medições de da execução de Cópias de Regiões de Memória, Conjunto 2 e Heurística ativada.

Sumário

De	03/10/2018 20:19:49
Para	03/10/2018 20:21:50
Valor máximo	100 W
Em	03/10/2018 20:19:56
Valor mínimo	80 W
Em	03/10/2018 20:19:49
μ	82,3029 W
s	4,26361 W
5% percentil	80 W
95% percentil	90 W
% [85% - 110%]	0%
% [90% - 110%]	0 %

- Comparativo de desempenho para o Conjunto 2:

Tabela 6.4 - Comparativo Consumo com e sem Heurística, Cópia de Regiões de Memória, Conjunto de Tarefas 2.

	Média	Máximo	Mínimo
Sem Heurística	91,4823 W	100,0 W	80,0 W
Com Heurística	82,3029 W	100,0 W	80,0 W
Diferença	9,1794 W	0 W	0 W

Como demonstrado na tabela acima, o ganho na redução de consumo foi de 9,1794 W no período de 2 minutos de execução, ou seja, cerca de 10% (10.034072164779417%) do consumo médio sem heurística foi reduzido. Isto demonstra que a heurística diminui o gasto exacerbado de energia de forma desnecessária para executar as tarefas especificadas.

Além disso, a diferença no consumo demonstra que a baixa no consumo é mais significativa quando executando tarefas muito relacionadas com a memória, se comparada com as reduções no consumo das tarefas recursivas, apontando então que esta parte da heurística está melhor desenvolvida, além também de apresentar um desempenho mais constante, aproximadamente 10% em ambos os conjuntos de tarefas.

6.2. ANÁLISE DE CONSUMO INTERFACE RAPL

Como descrito nos Tópicos “2.10. Materiais Utilizados” e em “3.3. Leitura de Consumo Energético”, uma alternativa às aferições de consumo feitas com o Fluke é o uso da interface RAPL, disponibilizada pela Intel. Por se tratar de registradores internos do processador, é possível então realizar a análise de consumo energético do

processador isoladamente. Sendo aqui analisado o consumo do processador como um todo, através do MSR MSR_PKG_ENERGY_STATUS, e o consumo apenas dos núcleos, através do MSR MSR_PP0_ENERGY_STATUS.

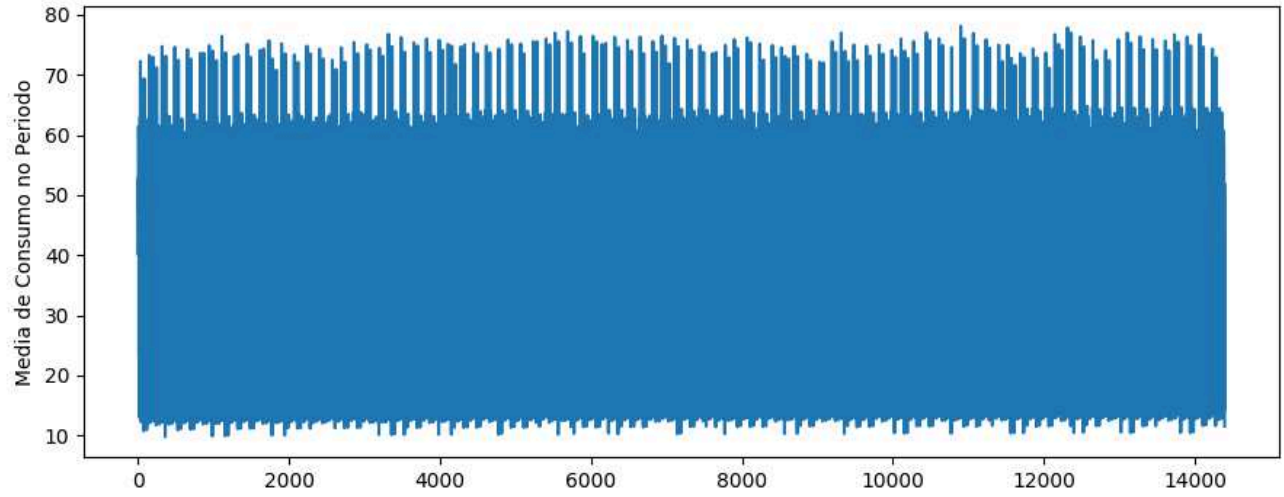
Outro detalhe é de que a análise aqui é feita utilizando os logs de execução, aplicando a fórmula descrita no tópico “3.3. Leitura de Consumo Energético” para calcular o consumo médio de um período da execução, sendo então as leituras feitas em uma granularidade muito maior que pelo próprio Fluke, como exemplo, em dois minutos de execução, são realizadas aproximadamente 15000 capturas na CPU analisada (varia um pouco com o uso e a quantidade de Threads associadas a mesma), onde o Fluke realiza cerca de 480 capturas (120 seg/0.25 seg). Logo, a análise dos logs se dá por um código python, onde a biblioteca “matplotlib.pyplot” é utilizada para realizar o *plot* dos gráficos.

Por fim, antes de partir para as análises, vale citar que nesta etapa a análise do consumo foi feita apenas para o conjunto de Tarefas 1 (Tabela 4.1), pois o mesmo foi o que obteve os melhores resultados de redução de consumo no Fluke.

6.2.1. FIBONACCI RECURSIVO

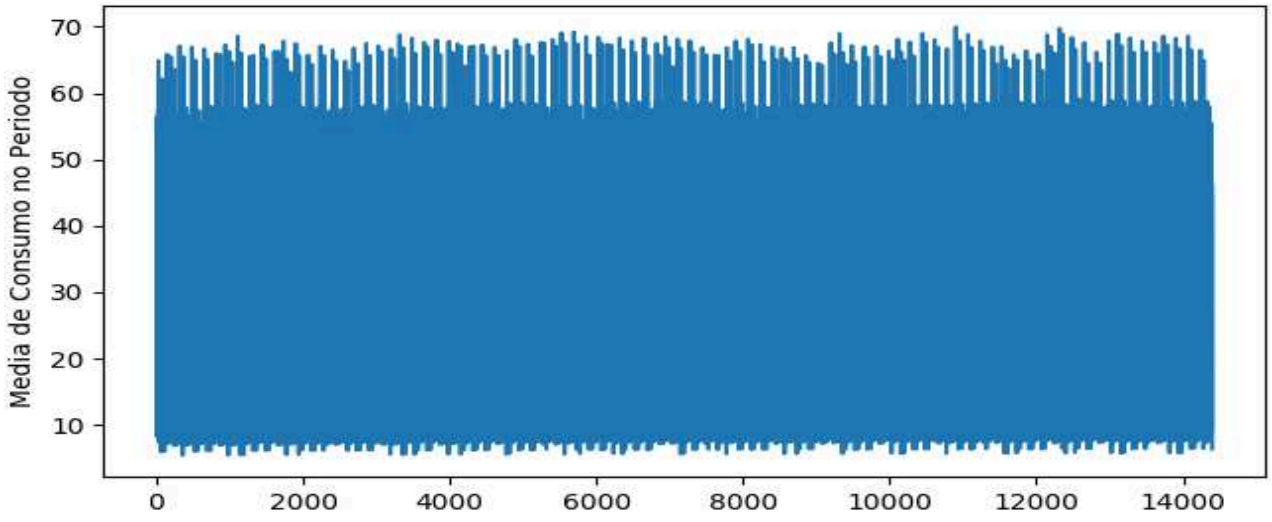
As estatísticas geradas pela execução do Fibonacci Recursivo com a heurística desativada foram as seguintes:

Figura 6.25 - Consumo energético médio PKG por períodos ao longo da execução de Fibonacci Recursivo sem Heurística



- Consumo Médio do PKG ao longo da execução: 42.8 W
- Comparativo com o consumo observado com o Fluke:
    - 85,5694 W (Fluke) - 42.8W (RAPL) = 42.7694.
    - Ou seja, o consumo do processador foi próximo da metade do consumo total do computador.

Figura 6.26 - Consumo energético médio PP0 por períodos ao longo da execução de Fibonacci Recursivo sem Heurística.

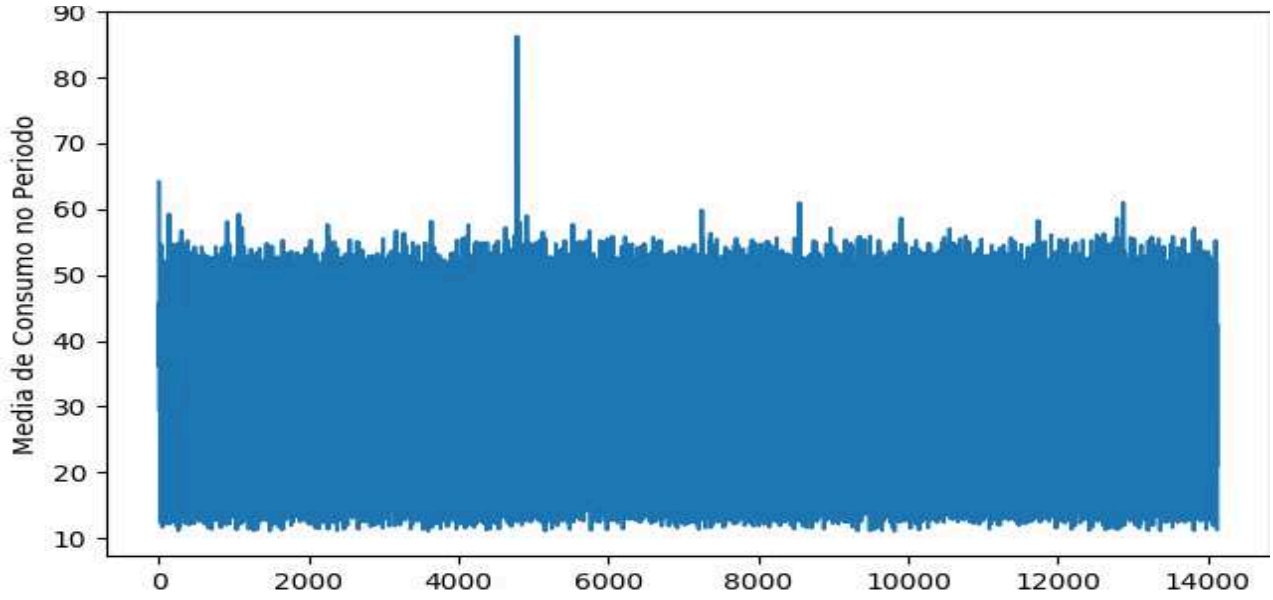


- Consumo Médio do PP0 ao longo da execução: 37.6 W
- Comparativo com o consumo observado com o Fluke:

- 85,5694 W (Fluke) - 37.6 W (RAPL) = 47.9694 W.
- Ou seja, o consumo do Núcleos do processador foi próximo de 44% do consumo total do computador.

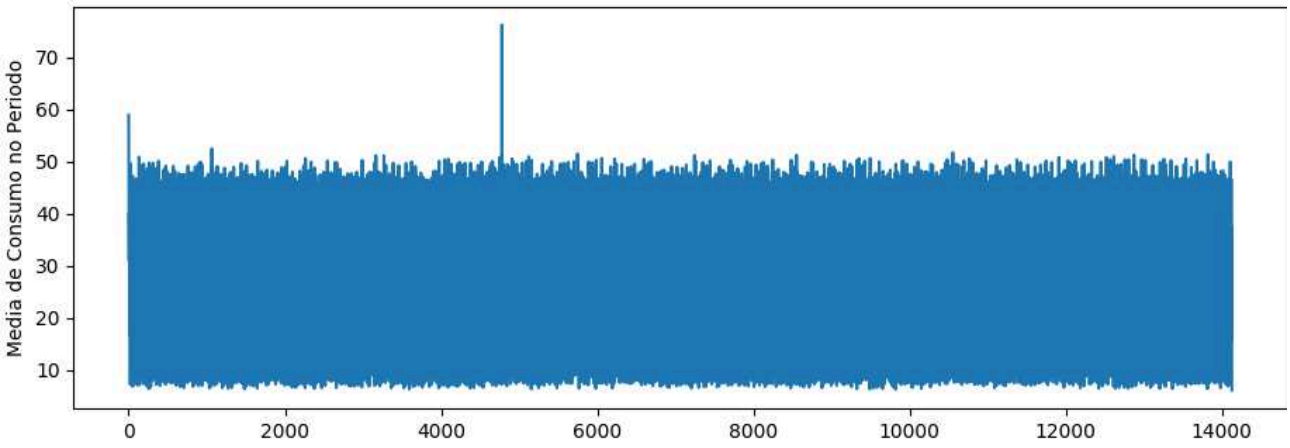
As estatísticas geradas pela execução do Fibonacci Recursivo com a heurística habilitada foram as seguintes:

Figura 6.27 - Consumo energético médio PKG por períodos ao longo da execução de Fibonacci Recursivo com Heurística habilitada.



- Consumo Médio do PKG ao longo da execução: 39.7 W
- Comparativo com o consumo observado com o Fluke:
  - 79,9376 W (Fluke) - 39.7W (RAPL) = 40.2376.
  - Ou seja, o consumo do processador foi próximo da metade do consumo total do computador.

Figura 6.28 - Consumo energético médio PP0 por períodos ao longo da execução de Fibonacci Recursivo com Heurística habilitada.



- Consumo Médio do PP0 ao longo da execução: 34.5 W
- Comparativo com o consumo observado com o Fluke:
    - 79,9376 W (Fluke) - 34.5W (RAPL) = 45.4376.
    - Ou seja, o consumo dos núcleos do processador foi próximo de 43% do consumo total do computador.

Tabela 6.5 - Comparativo Consumo com e sem Heurística, Fibonacci Recursivo, Conjunto de Tarefas 1.

	Média PKG	Média PP0
Sem Heurística	42.8 W	37.6 W
Com Heurística	39.7 W	34.5 W
Diferença	3.1 W	3.1 W

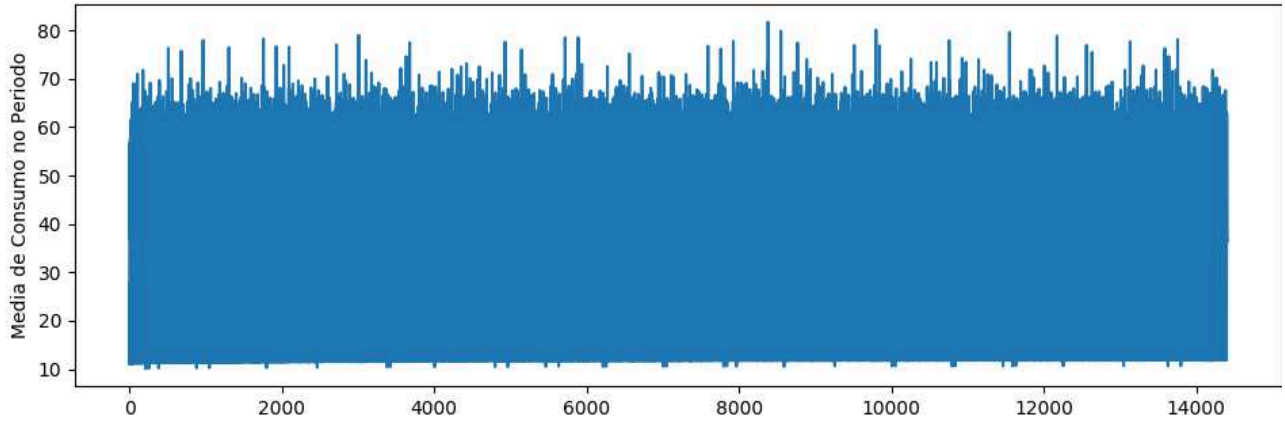
Como a diferença do consumo foi igual no PKG e no PP0, e o PP0 está incluído no consumo do PKG, podemos concluir que a diferença no consumo foi diretamente no PP0, ou seja, a heurística afetou diretamente o consumo núcleos de processamento. Sendo então uma redução de 3.1 W na média de consumo da execução, ou seja,

aproximadamente 8.25% de redução no consumo das CPUs. O que já apresenta um resultado mais significativo que com a análise do consumo total do computador realizada pelo Fluke, onde a redução foi de 6.5% do consumo energético.

6.2.2. CÓPIA DE REGIÕES DE MEMÓRIA

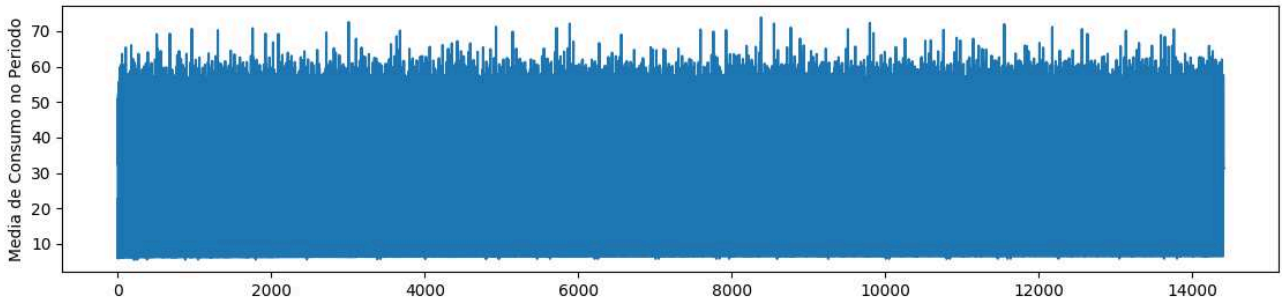
As estatísticas geradas pela execução de Cópia de Regiões de Memória com a heurística desativada foram as seguintes:

Figura 6.29 - Consumo energético médio PKG por períodos ao longo da execução de Cópia de Regiões de Memória sem Heurística.



- Consumo Médio do PKG ao longo da execução: 45.2 W
- Comparativo com o consumo observado com o Fluke:
    - 86,0581 W (Fluke) - 45.2 W (RAPL) = 40.8581.
    - Ou seja, o consumo do processador foi próximo de 52% do consumo total do computador.

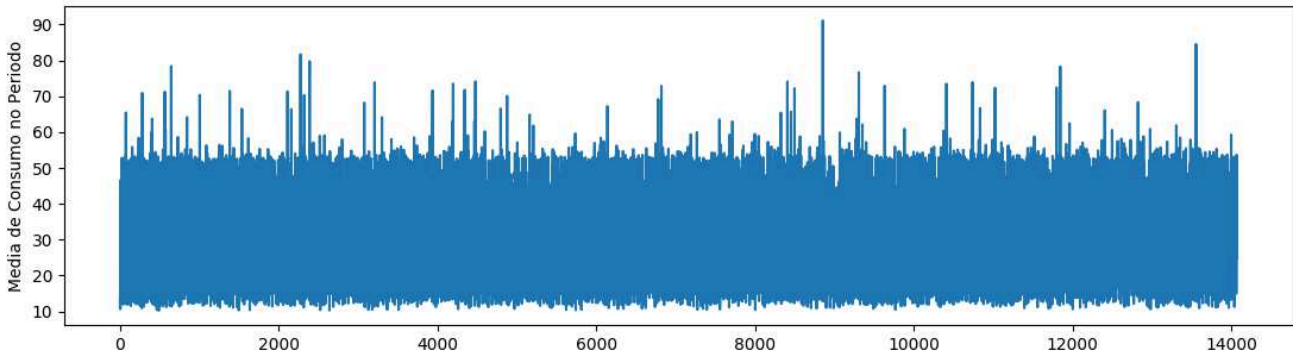
Figura 6.30 - Consumo energético médio PP0 por períodos ao longo da execução de Cópia de Regiões de Memória sem Heurística.



- Consumo Médio do PP0 ao longo da execução: 39.9 W
- Comparativo com o consumo observado com o Fluke:
    - 86,0581 W (Fluke) - 39,9 W (RAPL) = 46,1581.
    - Ou seja, o consumo dos Núcleos do processador foi próximo de 46% do consumo total do computador.

As estatísticas geradas pela execução de Cópia de Regiões de Memória com a heurística habilitada foram as seguintes:

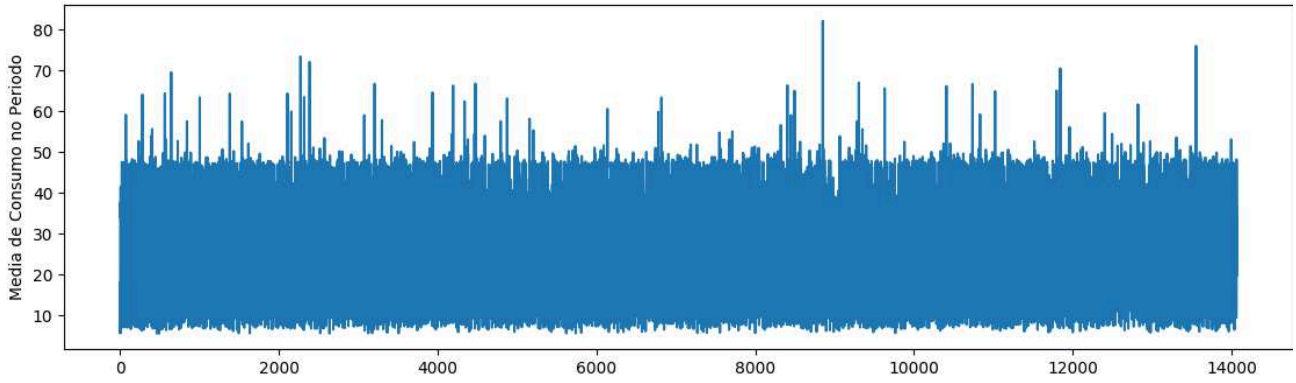
Figura 6.31 - Consumo energético médio PKG por períodos ao longo da execução de Cópia de Regiões de Memória com Heurística habilitada.



- Consumo Médio do PKG ao longo da execução: 38,8 W
- Comparativo com o consumo observado com o Fluke:
    - 77,1726 W (Fluke) - 38.8W (RAPL) = 38,3726.

- Ou seja, o consumo do processador foi próximo da metade do consumo total do computador.

Figura 6.32 - Consumo energético médio PP0 por períodos ao longo da execução de Cópia de Regiões de Memória com Heurística habilitada.



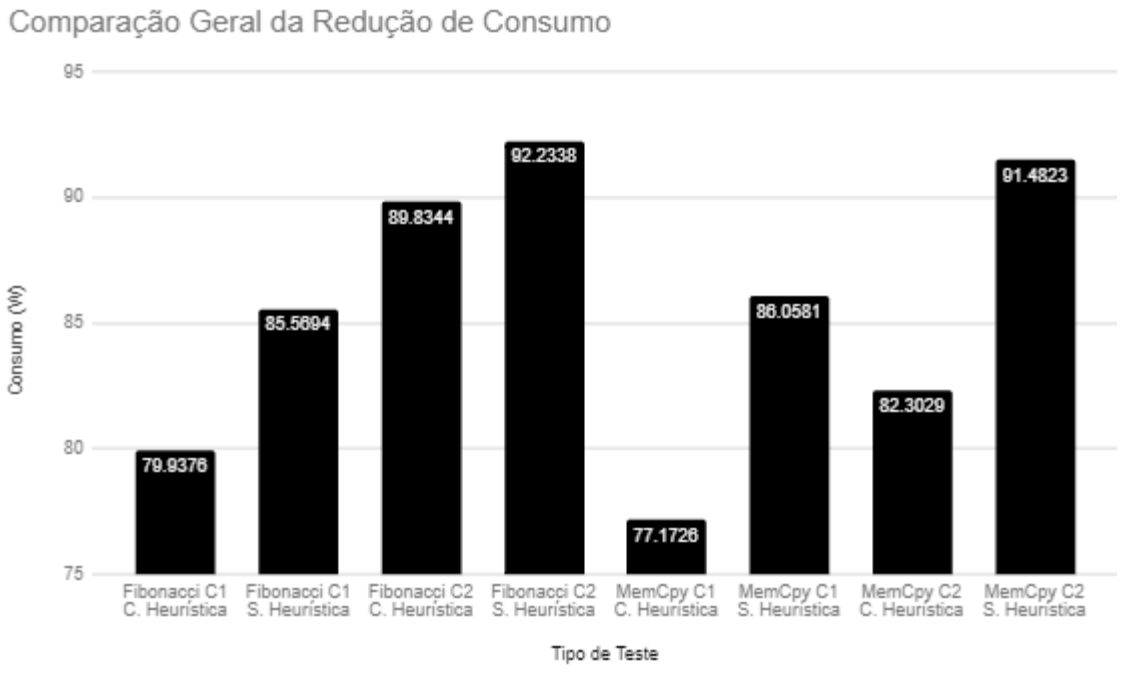
- Consumo Médio do PP0 ao longo da execução: 33.7 W
- Comparativo com o consumo observado com o Fluke:
  - 77,1726 W (Fluke) - 33.7 W (RAPL) = 43,4726.
  - Ou seja, o consumo dos Núcleos do processador foi próximo de 44% do consumo total do computador.

Tabela 6.6 - Comparativo Consumo com e sem Heurística, Cópia de Regiões de Memória, Conjunto de Tarefas 1.

	Média PKG	Média PP0
Sem Heurística	45,2 W	39,9 W
Com Heurística	38,8 W	33,7 W
Diferença	6,4 W	6,2 W

Sendo então, a maior parte da redução de consumo de energia presente no PKG provém do PP0, logo, é possível afirmar que a heurística teve maior impacto nas CPUs do processador. Obtendo um ganho de redução de consumo energético médio de 6.2 W, que representa aproximadamente 15% do consumo energético do PP0. Logo, a heurística teve um impacto mais significativo isolando o consumo das CPUs, que é onde ela realmente opera, se comparada com a redução de consumo gerada pela a análise do Fluke, que obteve um ganho de 10% na redução do consumo do computador. Isto decorre do fato do computador apresentar um consumo basal, ou seja, um consumo que não será afetado pelo consumo do processador, logo, mesmo que o processador entre em estado de halt, ainda ocorre alimentação de energia dos demais componentes, que, pelas diferenças de consumo apresentadas acima, é de aproximadamente 40W.

Figura 6.33 - Comparação do consumo energético entre as análises de cada Teste executado com o Fluke.



Pode-se observar no gráfico acima que, dentre as duas partes da heurística, a que atende as operações de cópias de regiões de memória leva vantagem, visto que reduz de forma mais efetiva o consumo. Outra conclusão, é de que a parte que atende operações recursivas, consegue um melhor desempenho quando o uso do conjunto de tarefas é mais baixo.

Além disso, a diferença das medições do Fluke e do RAPL mostram que a aplicação do DVFS no processador também afetou outros componentes do computador, visto que a redução nas aferições do RAPL no caso das tarefas com alta ligação com a memória foi de 6.4W no conjunto 1, e pelas aferições do Fluke a redução foi de 8,8855 W, gerando uma diferença de 2.4885W, que provém dos demais componentes do computador.

7. CONCLUSÃO

O desenvolvimento deste Trabalho de Conclusão de Curso proporcionou um grande aprendizado sobre o comportamento de um processador durante a execução, principalmente sobre características arquiteturais e relacionadas com eventos que descrevem seu desempenho. Além disso, também foi possível adquirir um vasto conhecimento sobre sistemas operacionais através do EPOS, onde foi então estudado principalmente sobre escalonadores de tarefas e a inicialização do sistema operacional junto a inicialização de um computador. Outro conhecimento importante adquirido foi sobre métodos científicos de pesquisa, os quais vão auxiliar em uma futura carreira acadêmica.

O objetivo de desenvolver, em conjunto com o discente Leonardo Passig Horstmann (15103030), um sistema de captura de dados de execução não intrusivo, foi concluído com sucesso, proporcionando o desenvolvimento da heurística aqui apresentada. Além disso, o sistema também pode ser útil para vários pesquisadores da área que buscam analisar dados, sem interferência das capturas e do SO, do desempenho de um processador executando tarefas fisicamente, e não apenas em simuladores.

Sobre o desenvolvimento das heurísticas, foi possível concluir esta tarefa através da análise dos com algoritmos de Data Mining em conjunto com uma análise manual. Obtendo-se assim, resultados satisfatórios, onde a heurística desenvolvida consegue executar tarefas críticas relacionadas com um alto uso da hierarquia de memória e/ou com um alto uso de recursão, sem perder a característica de determinismo temporal, ao mesmo tempo que reduz o consumo dos núcleos processador em até 15%. A aplicação

da heurística abrange sistemas multi-core de tempo real, principalmente sistemas embarcados, onde um baixo consumo de energia e um alto desempenho são cada vez mais necessários.

8. TRABALHOS FUTUROS

Trabalhos futuros podem ser realizados a partir do sistema de captura de dados não intrusivo desenvolvido, pois, através de várias técnicas de mineração de dados, muito pode se descobrir sobre otimização em sistemas multi-core de tempo real, principalmente com dados sem interferência e gerados por uma execução real de um conjunto de tarefas.

Também pode se desenvolver otimizações sobre a heurística aqui desenvolvida, principalmente com o uso da PEBS, um sistema do próprio processador, que gera interrupções no sistema operacional a partir de gatilhos configuráveis nos canais da PMU monitorados, a fim de melhorar a precisão das tomadas de decisões sobre o estado atual da execução. É possível também, estender a ideia utilizada no desenvolvimento da heurística, a fim de aumentar a gama de tarefas na qual a mesma consegue abranger.

BIBLIOGRAFIA

[1] **“Embedded Parallel Operating System”**, LISHA, 2018. Disponível em: <https://epos.lisha.ufsc.br/HomePage>. Acesso em: 18/06/2018.

[2] **“Software/Hardware integration Laboratory”**, LISHA, 2018. Disponível em: <https://lisha.ufsc.br>. Acesso em: 20/11/2017.

[3] **“EPOS 2 User Guide”**, LISHA, 2018. Disponível em: <https://epos.lisha.ufsc.br/EPOS+2+User+Guide>. Acesso em: 18/06/2018.

[4] **“Intel® 64 and IA-32 Architectures Software Developer’s Manual”**, Intel, 2018. Disponível em: <https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>. Acesso em: 18/06/2018.

[5] “**Internet of Things at UFSC”**, LISHA, 2018. Disponível em: <https://iot.lisha.ufsc.br/HomePage>. Acesso em: 18/06/2018.

[6] “**IoT with EPOS”**, LISHA, 2018. Disponível em: <http://epos.lisha.ufsc.br/IoT+with+EPOS>. Acesso em: 18/06/2018.

[7] HOFFMANN, J. L. C., HORSTMANN, L. P. “**Performance Monitoring with EPOS**”., 2018. Disponível em: <http://epos.lisha.ufsc.br/Performance+Monitoring+with+EPOS>. Acesso em: 18/06/2018.

[8] HOFFMANN, J. L. C., HORSTMANN, L. P. “**Adaptive DVFS for EPOS Multicore Schedulers**”, dezembro de 2017. Disponível em:

<https://epos.lisha.ufsc.br/Adaptive+DVFS+for+EPOS+Multicore+Schedulers>. Acesso em: 20/05/2018.

[9] HAMMOND, L., NAYFEH, B. A., OLU-KOTUN, K., **"A Single-Chip Multiprocessor"**, Computer, vol. 30, no. 9, pp. 79-85, Setembro. 1997. Disponível em: https://ieeexplore.ieee.org/document/612253/. Acesso em: 17/06/2018.

[10] KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., HAHN, S., **"Using OS Observations to Improve Performance in Multicore Systems"**, IEEE Micro, vol. 28, no. 3, Maio-Junho 2008. Disponível em: https://ieeexplore.ieee.org/document/4550860/. Acesso em: 10/06/2018.

[11] STANKOVIC, J.A., "**Misconceptions about real-time computing: a serious problem for next-generation systems",** Computer, vol. 21, no. 10, Outubro - 1988. Dísponível em: <https://ieeexplore.ieee.org/document/7053/>. Acesso em: 15/06/2018.

[12] TANENBAUM, A. S., BOS, H., "**MODERN OPERATING SYSTEMS"**, Quarta edição, Pearson Education, 2015, Inc., Upper Saddle River, New Jersey.

[13] KUMAR, V. "**Real-Time Scheduling Algorithms".** Disponível em: <https://pdfs.semanticscholar.org/549a/2bff5d595e759156fb2cb9bad14d5a2fc892.pdf>. Acesso em: 18/06/2018.

[14] ÅRZÉN, K.E., CERVIN, A., "**CONTROL AND EMBEDDED COMPUTING: SURVEY OF RESEARCH DIRECTIONS"**, IFAC Proceedings Volumes, vol. 38, no. 1, 2005, Pág. 191-202. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1474667016370859>. Acesso em: 16/06/2018.

[15] DONYANAVARD, B.; MÜCK, T.; SARMA, S.; DUTT, N., “**SPARTA: Runtime task allocation for energy efficient heterogeneous manycores”**, Department of Computer Science, University of California, Irvine, USA, IEEE, 2016. Disponível em: <http://ieeexplore.ieee.org/document/7750975>. Acesso em: 29/09/2017.

[16] MÜCK, T.; SARMA, S.; DUTT, N.; “**Run-DMC: runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency”**, Amsterdam, The Netherlands, IEEE, 2015. Disponível em: <https://dl.acm.org/citation.cfm?id=2830859>. Acesso em: 29/09/2017.

[17] KANTARDZIC, M., **“Data Mining: Concepts, Models, Methods, and Algorithms”**, Second Edition, John Wiley & Sons, Inc., IEEE, 2011. Disponível em: <https://ieeexplore.ieee.org/xpl/ebooks/bookPdfWithBanner.jsp?fileName=6105629.pdf&bkn=6105606>. Acesso em: 18/06/2018.

[18] WOLD, S., ESBENSEN, K., GELADI, P., “**Principal component analysis”**, Amsterdam, Elsevier Science Publishers B.V, Chemometrics and Intelligent Laboratory Systems, no. 2, pág. 37-52, Agosto - 1987. Disponível em: <https://www.sciencedirect.com/science/article/pii/0169743987800849>. Acesso em: 14/06/2018.

[19] GRACIOLI, G., FRÖHLICH, A. A., “**Two-phase colour-aware multicore real-time scheduler**”, IET, vol. 11, no. 4, Junho - 2017. Disponível em: <https://ieeexplore.ieee.org/document/7956618/?arnumber=7956618>. Acesso em: 29/09/2017.

[20] GRACIOLI, G., FRÖHLICH, A. A., “**On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures**”, ACM SIGOPS

Operating Systems Review - Special Topics, vol. 49, no. 2, pág 2-16, Dezembro - 2015. Disponível em: <https://dl.acm.org/citation.cfm?id=2883594>. Acesso em: 29/09/2017.

[21] SHEN, X., EADES, P., HONG, S., MOERE, A.V., ”**Intrusive and Non-intrusive Evaluation of Ambient Displays**”, Proceedings of the 1st International Workshop on Ambient Information Systems, Colocated at Pervasive 2007, Toronto, Canada, Maio - 2007. Disponível em: <http://www.infoscape.org/publications/pervasive07_ambient_evaluation.pdf>. Acesso em: 18/06/2018.

[22] **“Analisador de energia e potência Fluke 435 série II”,** FLUKE CORPORATION, 2018. Disponível em: <https://www.fluke.com/pt-br/produto/teste-eletrico/os-analisadores-de-qualidade-de-energia/analisadores-da-qualidade-da-energia-trifasica/fluke-435-series-ii>. Acesso em: 20/07/2018.

[23] DIMITROV, M., STRICKLAND, C., KIM, S., KUMAR, K., DOSHI, K., **“Intel® Power Governor”**, Julho - 2012. Disponível em: <https://software.intel.com/en-us/articles/intel-power-governor >. Acesso em: 30/07/2018.

[24] SULEIMAN, D., IBRAHIM, M., HAMARASH, I., “**DYNAMIC VOLTAGE FREQUENCY SCALING (DVFS) FOR MICROPROCESSORS POWER AND ENERGY REDUCTION”**, Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=E6F55A8CE6B176124DC60E15141B65B5?doi=10.1.1.111.1451&rep=rep1&type=pdf>. Acesso em: 01/08/2017.

[25] PILLAI, P., SHIN, K. G., “**Real-time dynamic voltage scaling for low-power embedded operating systems”**. Disponível em: <http://sosp.org/2001/papers/pillai.pdf>. Acesso em: 15/06/2018.

[26] “**SPSS Tutorials: Pearson Correlation**”, Kent State University - University Libraries. Disponível em: <https://libguides.library.kent.edu/SPSS/PearsonCorr>. Acesso em: 16/05/2018.

[27] “**Correlation Attribute Eval**”, Weka SourceForge Disponível em: <http://weka.sourceforge.net/doc.dev/weka/attributeSelection/CorrelationAttributeEval.html >. Acesso em: 16/05/2018.

[28] GRACIOLI, G. **Real-Time Operating System Support For Multicore Applications.** 2014. Disponivel em: <http://www.lisha.ufsc.br/pub/Gracioli_PHD_2014.pdf>. Acesso em: 05/07/2018.

[29] “**Function memcpy**”, C Plus Plus. Disponível em: <http://www.cplusplus.com/reference/cstring/memcpy/>. Acesso em: 10/07/2018.

ANEXOS

A1. TABELA DE EVENTOS INTEL SANDY BRIDGE NO EPOS

Nome	Nome No EPOS	Nome na Documentação INTEL	Descrição Intel
pmu0	(FIXED) INSTRUCTION_RETIRED	Instructions Retired	Counts when the last uop of an instruction retires.
pmu1	(FIXED) CLOCK	UnHalted Core Cycles	Counts core clock cycles whenever the logical processor is in C0 state (not halted). The frequency of this event varies with state transitions in the core.
pmu2	(FIXED) DVS_CLOCK	UnHalted Reference Cycles1	Counts at a fixed frequency whenever the logical processor is in C0 state (not halted).
pmu3	BRANCH	Branch Instruction Retired	Counts when the last uop of a branch instruction retires.
pmu4	BRANCH_MISS	Branch Misses Retired	Counts when the last uop of a branch instruction retires which corrected misprediction of the branch prediction hardware at execution time.
pmu5	L1_HIT	MEM_LOAD_RETIRED.L1_HIT	Hit accesses to the L1 Cache
pmu6	L2_HIT	MEM_LOAD_RETIRED.L2_HIT	Hit accesses to the L2 Cache
pmu7	L3_HIT / LLC_HIT	MEM_LOAD_RETIRED.L3_HIT	Hit accesses to the L3 Cache
pmu8	L1_MISS	MEM_LOAD_RETIRED.L1_MISS	MISS accesses to the L1 Cache
pmu9	L2_MISS	MEM_LOAD_RETIRED.L2_MISS	MISS accesses to the L2 Cache
pmu10	L3_MISS	MEM_LOAD_RETIRED.L3_MISS	MISS accesses to the L3 Cache
pmu11	LLC_HITM	Hit accesses to the L3 Cache	Hit LLC "Modified"
pmu12	LD_BLOCKS_DATA_UNKNOWN	LD_BLOCKS.DATA_UNKNOWN	Blocked loads due to store buffer blocks with unknown data.
pmu13	LD_BLOCKS_STORE_FORWARD	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that

			cannot be forwarded.
pmu14	LD_BLOCKS_NO_SR	LD_BLOCKS.NO_SR	# of Split loads blocked due to resource not Available
pmu15	LD_BLOCKS_ALL_BLOCK	LD_BLOCKS.ALL_BLOCK	Number of cases where any load is blocked but has no DCU miss.
pmu16	MISALIGN_MEM_REF_LOADS	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.
pmu17	MISALIGN_MEM_REF_STORES	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.
pmu18	LD_BLOCKS_PARTIAL_ADDRESS_ALIAS	LD_BLOCKS_PARTIAL.ADDRES S_ALIAS	False dependencies in MOB due to partial compare on address.
pmu19	LD_BLOCKS_PARTIAL_ALL_STA_BLCOK	LD_BLOCKS_PARTIAL.ALL_STA _BLOCK	The number of times that load operations are temporarily blocked because of older stores, with addresses that are not yet known. A load operation may incur more than one block of this type.
pmu20	DTLB_LOAD_MISSES_MISS_CAUSES_A_WALK	DTLB_LOAD_MISSES.MISS_CA USES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.
pmu21	DTLB_LOAD_MISSES_MISS_WALK_COMPLETED	DTLB_LOAD_MISSES.WALK_CO MPLETED	Misses in all TLB levels that caused page walk completed of any size.
pmu22	DTLB_LOAD_MISSES_MISS_WALK_DURATION	DTLB_LOAD_MISSES.WALK_DU RATION	Cycle PMH is busy with a walk.
pmu23	DTLB_LOAD_MISSES_MISS_STLB_HIT	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.
pmu24	INT_MISC_RECOVERY_CYCLES	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears or JEClear. Set Cmask= 1.
pmu25	INT_MISC_RAT_STALL_CYCLES	INT_MISC.RAT_STALL_CYCLES	Cycles RAT external stall is sent to IDQ for this thread.
pmu26	UOPS_ISSUED_ANY	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.

pmu27	FP_COMP_OPS_EXE_X87	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed
pmu28	FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* double precision FP packed uops executed.
pmu29	FP_COMP_OPS_EXE_SSE_FP_SCALAR_SINGLE	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* single precision FP scalar uops executed.
pmu30	FP_COMP_OPS_EXE_SSE_PACKED_SINGLE	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* single precision FP packed uops executed.
pmu31	FP_COMP_OPS_EXE_SSE_SCALAR_DOUBLE	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE	Counts number of SSE* double precision FP scalar uops executed.
pmu32	SIMD_FP_256_PACKED_SINGLE	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floatingpoint Instructions.
pmu33	SIMD_FP_256_PACKED_DOUBLE	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floatingpoint Instructions.
pmu34	ARITH_FPU_DIV_ACTIVE	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of Divides.
pmu35	INSTS_WRITTEN_TO_IQ_INSTS	INSTS_WRITTEN_TO_IQ.INSTS	Counts the number of instructions written into the IQ every cycle.
pmu36	L2_RQSTS_DEMAND_DATA_RD_HIT	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.
pmu37	L2_RQSTS_ALL_DEMAND_DATA_RD	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.
pmu38	L2_RQSTS_RFO_HITS	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.
pmu39	L2_RQSTS_RFO_MISS	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.
pmu40	L2_RQSTS_ALL_RFO	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.
pmu41	L2_RQSTS_CODE_RD_HIT	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.
pmu42	L2_RQSTS_CODE_RD_MISS	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.

pmu43	L2_RQSTS_ALL_CODE_RD	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.
pmu44	L2_RQSTS_PF_HIT	L2_RQSTS.PF_HIT	Requests from L2 Hardware prefetcher that hit L2.
pmu45	L2_RQSTS_PF_MISS	L2_RQSTS.PF_MISS	Requests from L2 Hardware prefetcher that missed L2.
pmu46	L2_RQSTS_ALL_PF	L2_RQSTS.ALL_PF	Any requests from L2 Hardware prefetchers
pmu47	L2_STORE_LOCK_RQSTS_MISS	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines.
pmu48	L2_STORE_LOCK_RQSTS_HIT_E	L2_STORE_LOCK_RQSTS.HIT_E	RFOs that hit cache lines in E state.
pmu49	L2_STORE_LOCK_RQSTS_HIT_M	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state.
pmu50	L2_STORE_LOCK_RQSTS_ALL	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state.
pmu51	L2_L1D_WB_RQSTS_HIT_E	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.
pmu52	L2_L1D_WB_RQSTS_HIT_M	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.
pmu53	LONGEST_LAT_CACHE_REFERENCE	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.
pmu54	LONGEST_LAT_CACHE_MISS	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.
pmu55	CPU_CLK_UNHALTED_THREAD_P	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.
pmu56	CPU_CLK_THREAD_UNHALTED_REF_XCLK	CPU_CLK_THREAD_UNHALTED_REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.
pmu57	L1D_PEND_MISS_PENDING	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses

			every cycle. Set Cmask = 1 and Edge =1 to count Occurrences
pmu58	DTLB_STORE_MISSES_MISS_CAUSES_A_WALK	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).
pmu59	DTLB_STORE_MISSES_WALK_COMPLETED	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).
pmu60	DTLB_STORE_MISSES_WALK_DURATION	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk
pmu61	DTLB_STORE_MISSES_TLB_HIT	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.
pmu62	LOAD_HIT_PRE_SW_PF	LOAD_HIT_PRE.SW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.
pmu63	LOAD_HIT_PREHW_PF	LOAD_HIT_PRE.HW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.
pmu64	HW_PRE_REQ_DL1_MISS	HW_PRE_REQ.DL1_MISS	Hardware Prefetch requests that miss the L1D cache. A request is being counted each time it access the cache & miss it, including if a block is applicable or if hit the Fill Buffer for example.
pmu65	L1D_REPLACEMENT	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data Cache
pmu66	L1D_ALLOCATED_IN_M	L1D.ALLOCATED_IN_M	Counts the number of allocations of modified L1D cache lines.
pmu67	L1D_EVICTION	L1D.EVICTION	Counts the number of modified lines evicted from the L1 data cache due to replacement.
pmu68	L1D_ALL_M_REPLACEMENT	L1D.ALL_M_REPLACEMENT	Cache lines in M state evicted out of L1D due to Snoop HitM or dirty line replacement.
pmu69	PARTIAL_RAT_STALLS_FLAGS_MERGE_UOP	PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP	Increments the number of flags-merge uops in flight

			each cycle. Set Cmask = 1 to count cycles.
pmu70	PARTIAL_RAT_STALLS_SLOW_LEA_WINDOW	PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW	Cycles with at least one slow LEA uop allocated.
pmu71	PARTIAL_RAT_STALLS_MUL_SINGLE_UOP	PARTIAL_RAT_STALLS.MUL_SINGLE_UOP	Number of Multiply packed/scalar single precision uops allocated.
pmu72	RESOURCE_STALLS2_ALL_FL_EMPTY	RESOURCE_STALLS2.ALL_FL_EMPTY	Cycles stalled due to free list empty.
pmu73	RESOURCE_STALLS2_ALL_PRF_CONTROL	RESOURCE_STALLS2.ALL_PRF_CONTROL	Cycles stalled due to control structures full for physical registers.
pmu74	RESOURCE_STALLS2_BOB_FULL	RESOURCE_STALLS2.BOB_FULL	Cycles Allocator is stalled due Branch Order Buffer.
pmu75	RESOURCE_STALLS2_OOO_RSRC	RESOURCE_STALLS2.OOO_RSRC	Cycles stalled due to out of order resources full.
pmu76	CPL_CYCLES_RING0	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.
pmu77	CPL_CYCLES_RING123	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.
pmu78	RS_EVENTS_EMPTY_CYCLES	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread
pmu79	OFFCORE_REQUESTS_OUTSTANDING_DEMAND_DATA_RD	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count Cycles.
pmu80	OFFCORE_REQUESTS_OUTSTANDING_DEMAND_RFO	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.
pmu81	OFFCORE_REQUESTS_OUTSTANDING_ALL_DATA_RD	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count Cycles.
pmu82	LOCK_CYCLES_SPLIT_LOCK_UC_LOCK_DURATION	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.
pmu83	LOCK_CYCLES_CACHE_LOCK_DURATION	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.
pmu84	IDQ_EMPTY	IDQ.EMPTY	Counts cycles the IDQ is empty.
pmu85	IDQ_MITE_UOPS	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ

			from MITE path. Set Cmask = 1 to count cycles.
pmu86	IDQ_DSB_UOPS	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles
pmu87	IDQ_MS_DSB_UOPS	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS busy by DSB. Set Cmask = 1 to count cycles MS is busy. Set Cmask=1 and Edge =1 to count MS activations.
pmu88	IDQ_MS_MITE_UOPS	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS is busy by MITE. Set Cmask = 1 to count Cycles.
pmu89	IDQ_MS_UOPS	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.
pmu90	ICACHE_MISSES	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.
pmu91	ITLB_MISSES_MISS_CAUSES_A_WALK	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks.
pmu92	ITLB_MISSES_WALK_COMPLETED	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page Walks.
pmu93	ITLB_MISSES_WALK_DURATION	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.
pmu94	ITLB_MISSES_STLB_HIT	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.
pmu95	ILD_STALL_LCP	ILD_STALL.LCP	Stalls caused by changing prefix length of the Instruction.
pmu96	ILD_STALL_IQ_FULL	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.
pmu97	BR_INST_EXEC_COND	BR_INST_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired conditional branches
pmu98	BR_INST_EXEC_DIRECT_JMP	BR_INST_EXEC.TAKEN_DIRECT_JUMP	Taken speculative and retired conditional branches excluding calls and indirects.

pmu99	BR_INST_EXEC_INDIRECT_JMP_NON_CALL_RET	BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired indirect branches excluding calls and returns.
pmu100	BR_INST_EXEC_RETURN_NEAR	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_RETURN	Taken speculative and retired indirect branches that are returns.
pmu101	BR_INST_EXEC_DIRECT_NEAR_CALL	BR_INST_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired direct near calls.
pmu102	BR_INST_EXEC_INDIRECT_NEAR_CALL	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired indirect near calls.
pmu103	BR_INST_EXEC_NON_TAKEN	BR_INST_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired indirect branches excluding calls and returns.
pmu104	BR_INST_EXEC_TAKEN	BR_INST_EXEC.ALL_INDIRECT_NEAR_RETURN	Speculative and retired indirect branches that are Returns.
pmu105	BR_INST_EXEC_ALL_BRANCHES	BR_INST_EXEC.ALL_BRANCHES	Speculative and retired branches.
pmu106	BR_MISP_EXEC_COND	BR_MISP_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired mispredicted conditional branches
pmu107	BR_MISP_EXEC_INDIRECT_JMP_NON_CALL_RET	BR_MISP_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired mispredicted indirect branches excluding calls and returns.
pmu108	BR_MISP_EXEC_RETURN_NEAR	BR_MISP_EXEC.TAKEN_RETURN_NEAR	Taken speculative and retired mispredicted indirect branches that are returns.
pmu109	BR_MISP_EXEC_DIRECT_NEAR_CALL	BR_MISP_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired mispredicted direct near calls
pmu110	BR_MISP_EXEC_INDIRECT_NEAR_CALL	BR_MISP_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired mispredicted indirect near calls.
pmu111	BR_MISP_EXEC_NON_TAKEN	BR_MISP_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired mispredicted indirect branches excluding calls and returns.
pmu112	BR_MISP_EXEC_TAKEN	BR_MISP_EXEC.ALL_NEAR_CALL	Speculative and retired mispredicted direct near Calls.
pmu113	BR_MISP_EXEC_ALL_BRANCHES	BR_MISP_EXEC.ALL_BRANCHES	Speculative and retired mispredicted branches
pmu114	IDQ_UOPS_NOT_DELIVERED_CORE	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when

			there is no back-end stall.
pmu115	UOPS_DISPATCHED_PORT_PORT_0	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.
pmu116	UOPS_DISPATCHED_PORT_PORT_1	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.
pmu117	UOPS_DISPATCHED_PORT_PORT_2_LD		
pmu118	UOPS_DISPATCHED_PORT_PORT_2_STA		
pmu119	UOPS_DISPATCHED_PORT_PORT_2	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.
pmu120	UOPS_DISPATCHED_PORT_PORT_3_LD		
pmu121	UOPS_DISPATCHED_PORT_PORT_3_STA		
pmu122	UOPS_DISPATCHED_PORT_PORT_3	UOPS_DISPATCHED_PORT.PORT_3	
pmu123	UOPS_DISPATCHED_PORT_PORT_4	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.
pmu124	UOPS_DISPATCHED_PORT_PORT_5	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.
pmu125	RESOURCE_STALLS_ANY	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.
pmu126	RESOURCE_STALLS_LB	RESOURCE_STALLS.LB	Counts the cycles of stall due to lack of load buffers
pmu127	RESOURCE_STALLS_RS	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.
pmu128	RESOURCE_STALLS_SB	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync)
pmu129	RESOURCE_STALLS_ROB	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.
pmu130	RESOURCE_STALLS_FCSW	RESOURCE_STALLS.FCSW	Cycles stalled due to writing the FPU control word.
pmu131	RESOURCE_STALLS_MXCSR		
pmu132	RESOURCE_STALLS_OTHER		
pmu133	DSB2MITE_SWITCHES_COUNT	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches
pmu134	DSB2MITE_SWITCHES_PENALTY_CYCLES	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay

pmu135	DSB_FILL_OTHER_CANCEL	DSB_FILL.OTHER_CANCEL	Cases of cancelling valid DSB fill not because of exceeding way limit.
pmu136	DSB_FILL_EXCEED_DSB_LINES	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.
pmu137	DSB_FILL_ALL_CANCEL		
pmu138	ITLB_ITLB_FLUSH	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes; includes 4k/2M/4M pages.
pmu139	OFFCORE_REQUESTS_DEMAND_DATA_RD	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.
pmu140	OFFCORE_REQUESTS_DEMAND_RFO	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.
pmu141	OFFCORE_REQUESTS_ALL_DATA_RD	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and Prefetch).
pmu142	UOPS_DISPATCHED_THREAD	UOPS_DISPATCHED.THREAD	Counts total number of uops to be dispatched perthread each cycle. Set Cmask = 1, INV =1 to count stall cycles.
pmu143	UOPS_DISPATCHED_CORE	UOPS_DISPATCHED.CORE	Counts total number of uops to be dispatched percore each cycle.
pmu144	OFFCORE_REQUESTS_BUFFER_SQ_FULL	OFFCORE_REQUESTS_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this thread core.
pmu145	AGU_BYPASS_CANCEL_COUNT	AGU_BYPASS_CANCEL.COUNT	Counts executed load operations with all the following traits: 1. Addressing of the format [base + offset], 2. The offset is between 1 and 2047, 3. The address specified in the base register is in one page and the address [base+offset] is in another page.
pmu146	OFF_CORE_RESPONSE_0	OFF_CORE_RESPONSE_0	See Section 18.9.5, “Off-core Response Performance Monitoring”.
pmu147	OFF_CORE_RESPONSE_1	OFF_CORE_RESPONSE_1	See Section 18.9.5, “Off-core Response Performance Monitoring”
pmu148	TLB_FLUSH_DTLB_THREAD	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.

pmu149	TLB_FLUSH_STLB_ANY	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.
pmu150	L1D_BLOCKS_BANK_CONFLICT_CYCLES	L1D_BLOCKS.BANK_CONFLICT_CYCLES	Cycles when dispatched loads are cancelled due to L1D bank conflicts with other load ports.
pmu151	INST_RETIRED_ANY_P	INST_RETIRED.ANY_P	Number of instructions at retirement.
pmu152	INST_RETIRED_PREC_DIST	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution
pmu153	OTHER_ASSISTS_ITLB_MISS_RETIRED	OTHER_ASSISTS.ITLB_MISS_RETIRED	Instructions that experienced an ITLB miss.
pmu154	OTHER_ASSISTS_AVX_STORE	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.
pmu155	OTHER_ASSISTS_AVX_TO_SSE	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.
pmu156	OTHER_ASSISTS_SSE_TO_AVX	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.
pmu157	UOPS_RETIRED_ALL	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled Cycles.
pmu158	UOPS_RETIRED_RETIRE_SLOTS	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each Cycle.
pmu159	MACHINE_CLEARS_MEMORY_ORDERING	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.
pmu160	MACHINE_CLEARS_SMC	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section.
pmu161	MACHINE_CLEARS_MASKMOV	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.
pmu162	BR_INST_RETIRED_ALL_BRANCHES_ARCH	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.
pmu163	BR_INST_RETIRED_CONDITIONAL	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch

			instructions retired.
pmu164	BR_INST_RETIRED_NEAR_CALL	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.
pmu165	BR_INST_RETIRED_ALL_BRANCHES	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.
pmu166	BR_INST_RETIRED_NEAR_RETURN	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions Retired.
pmu167	BR_INST_RETIRED_NOT_TAKEN	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions Retired.
pmu168	BR_INST_RETIRED_NEAR_TAKEN	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.
pmu169	BR_INST_RETIRED_FAR_BRANCH	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.
pmu170	BR_MISP_RETIRED_ALL_BRANCHES_ARCH	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.
pmu171	BR_MISP_RETIRED_CONDITIONAL	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.
pmu172	BR_MISP_RETIRED_NEAR_CALL	BR_MISP_RETIRED.NEAR_CALL	Direct and indirect mispredicted near call instructions retired.
pmu173	BR_MISP_RETIRED_ALL_BRANCHES	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.
pmu174	BR_MISP_RETIRED_NOT_TAKEN	BR_MISP_RETIRED.NOT_TAKEN	Mispredicted not taken branch instructions retired.
pmu175	BR_MISP_RETIRED_TAKEN	BR_MISP_RETIRED.TAKEN	Mispredicted taken branch instructions retired.
pmu176	FP_ASSIST_X87_OUTPUT	FP_ASSIST.X87_OUTPUT	Number of X87 assists due to output value
pmu177	FP_ASSIST_X87_INPUT	FP_ASSIST.X87_INPUT	Number of X87 assists due to input value.
pmu178	FP_ASSIST_SIMD_OUTPUT	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.
pmu179	FP_ASSIST_SIMD_INPUT	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.
pmu180	FP_ASSIST_ANY	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.
pmu181	ROB_MISC_EVENTS_LBR_INSERTS	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by Hardware.

pmu182	MEM_TRANS_RETIRED_LOAD_LATENCY	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization. PMC3 only.
pmu183	MEM_TRANS_RETIRED_PRECISE_STORE	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.
pmu184	MEM_UOP_RETIRED_LOADS		
pmu185	MEM_UOP_RETIRED_STORES		
pmu186	MEM_UOP_RETIRED_STLB_MISS	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.
pmu187	MEM_UOP_RETIRED_LOCK	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.
pmu188	MEM_UOP_RETIRED_SPLIT	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline Boundary.
pmu189	MEM_UOP_RETIRED_ALL		
pmu190	MEM_UOPS_RETIRED_ALL_LOADS	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops
pmu191	MEM_LOAD_UOPS_RETIRED_L1_HIT	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data Sources.
pmu192	MEM_LOAD_UOPS_RETIRED_L2_HIT	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data Sources
pmu193	MEM_LOAD_UOPS_RETIRED_L3_HIT		
pmu194	MEM_LOAD_UOPS_RETIRED_HIT_LFB	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.
pmu195	XSNP_MISS	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops whose data source was an onpackage core cache LLC hit and cross-core snoop Missed.

pmu196	XSNP_HIT	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops whose data source was an onpackage LLC hit and cross-core snoop hits.
pmu197	XSNP_HITM	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops whose data source was an onpackage core cache with HitM responses.
pmu198	XSNP_NONE	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops whose data source was LLC hit with no snoop required.
pmu199	MEM_LOAD_UOPS_MISC_RETIRED_LLC_MISS	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops which data sources were data missed LLC (excluding unknown data source).
pmu200	L2_TRANS_DEMAND_DATA_RD	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache
pmu201	L2_TRANS_RFO	L2_TRANS.RFO	RFO requests that access L2 cache.
pmu202	L2_TRANS_CODE_RD	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions
pmu203	L2_TRANS_ALL_PF	L2_TRANS.ALL_PF	L2 or LLC HW prefetches that access L2 cache. //including rejects
pmu204	L2_TRANS_L1D_WB	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.
pmu205	L2_TRANS_L2_FILL	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.
pmu206	L2_TRANS_L2_WB	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.
pmu207	L2_TRANS_ALL_REQ_USESTS	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.
pmu208	L2_LINES_IN_I	L2_LINES_IN.I	L2 cache lines in I state filling L2
pmu209	L2_LINES_IN_S	L2_LINES_IN.S	L2 cache lines in S state filling L2.
pmu210	L2_LINES_IN_E	L2_LINES_IN.E	L2 cache lines in E state filling L2.
pmu211	L2_LINES_IN_ALL	L2_LINES_IN.ALL	L2 cache lines filling L2.
pmu212	L2_LINES_OUT_DEMAND_CLEAN	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.
pmu213	L2_LINES_OUT_DEMAND_DIRTY	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.
pmu214	L2_LINES_OUT_DEMAND_PF_CLEAN	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by L2 prefetch
pmu215	L2_LINES_OUT_DEMAND_PF_DIRTY	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by L2 prefetch.
pmu216	L2_LINES_OUT_DEMAND_DIRTY_ALL	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2. //Counting does not cover Rejects.
pmu217	SQ_MISC_SPLIT_LOCK	SQ_MISC.SPLIT_LOCK	Split locks in SQ.

A2. CÓDIGOS DESENVOLVIDOS

Os códigos desenvolvidos durante este TCC podem ser encontrados no repośitório online github, junto de um arquivo readme descrevendo as respectivas mudanças do código original. Sendo então o código desenvolvido em duas etapas, na primeira o foco foi no desenvolvimento do código do sistema não intrusivo de capturas (Tópico "3.4 Sistema Não Intrusivo de capturas"), e em sequência, o desenvolvimento do código apresentado no Tópico “5.2 Código Final da Heurística”.

Os códigos aqui produzidos, tiveram como base a versão 2.1 do sistema operacional EPOS, disponível a partir do site http://epos.lisha.ufsc.br/EPOS+Software, as versões finais de cada etapa estão melhor descritas no tópico a seguir.

A2.1 CÓDIGO DO SISTEMA DE CAPTURA NÃO INTRUSIVO

Como já citado anteriormente, este código tem como base a versão 2.1 do sistema operacional EPOS, acrescida também de códigos para leitura de sensores de hardware, como a temperatura e o consumo (interface RAPL), além da disponibilidade da opção de modulação de clock. Vale lembrar que este código foi desenvolvido em conjunto com o discente Leonardo Passig Horstmann, visto a utilidade do código para a pesquisa e desenvolvimento do TCC de ambos.

O código pode ser encontrado no seguinte link:

- https://github.com/LeonardoHorstmann/Performance-Monitoring-on-EPOS

Junto do mesmo, se encontra um arquivo README que descreve as alterações realizadas. Seguem agora os códigos alterados:

app/test_pedf.cc (código da aplicação de teste)

```
#include <periodic_thread.h>
#include <utility/random.h>
#include <clock.h>
///*
#include <machine/pc/rtc.h>
#include <chronometer.h>
```

```
#include <utility/ostream.h>
#include <tsc.h>
//*/

using namespace EPOS;


#define BASE_MEMCPY_T 30
#define BASE_RECURSIVE_FIB_T 600
#define THREADS 12 //number of real-time tasks, usually, threads_parameters size
#define SECONDS 600 //execution time in seconds

OStream cout;
// period (microsecond), deadline, execution time (microsecond), cpu (partitioned)

unsigned int iterations[THREADS];
/*
unsigned int threads_parameters[][4] = { //heavy weight taskset
{ 50000 , 50000 , 27098 , 0 },
{ 25000 , 25000 , 5504 , 1 },
{ 100000 , 100000 , 68919 , 2 },
{ 100000 , 100000 , 64664 , 3 },
{ 50000 , 50000 , 9310 , 4 },
{ 200000 , 200000 , 105758 , 5 },
{ 200000 , 200000 , 29326 , 6 },
{ 200000 , 200000 , 67222 , 7 },
{ 50000 , 50000 , 21151 , 6 },
{ 50000 , 50000 , 6757 , 4 },
{ 50000 , 50000 , 34329 , 1 },
{ 50000 , 50000 , 8203 , 4 },
{ 100000 , 100000 , 44566 , 7 },
{ 25000 , 25000 , 8853 , 4 }
};
//*/
//*
unsigned int threads_parameters[][4] = { //light weight taskset
{ 25000 , 25000 , 13958 , 0 },
{ 100000 , 100000 , 15135 , 1 },
{ 200000 , 200000 , 136986 , 2 },
{ 50000 , 50000 , 25923 , 3 },
{ 25000 , 25000 , 11637 , 4 },
{ 100000 , 100000 , 20072 , 5 },
{ 50000 , 50000 , 30484 , 6 },
{ 200000 , 200000 , 25220 , 7 },
{ 200000 , 200000 , 23924 , 7 },
{ 100000 , 100000 , 31920 , 1 },
{ 50000 , 50000 , 18343 , 5 },
{ 50000 , 50000 , 19205 , 7 } };
//*/

//calculates number of iterations to be executed to accomplish execution time in SECONDS
int calc_iterations() {
    int sum = 0;
```

```
    //float base = 0;
    for (int i = 0; i < THREADS; i++) {
        if (iterations[i] == 0) {
    //        cout<<i<<" "<<threads_parameters[i][0]<<endl;
            iterations[i] = (SECONDS * 1000000) / threads_parameters[i][0];
        }
        sum+=iterations[i];
        cout<<threads_parameters[i][0]<<" "<<iterations[i]<<endl;
    }
    return sum;
}

//fibonacci recursive method
int fib(int pos) {
    if (pos == 1 || pos == 0) {
        return 1;
    } else {
        return (fib(pos - 1) + fib(pos - 2));
    }

}
/* //This is used to collect information about the time one execution */

/*unsigned int chronos[THREADS][ITERATIONS]; //capture time of each iteration
float mean_chronos[THREADS];
unsigned int pos_chronos[THREADS];
unsigned int total_chronos[THREADS];
unsigned int id_Thread[THREADS];

void mean() {
       cout << "Mean Execution Time per Iteration: " << endl;
       int greater = 0;
    for (int i = 0; i < THREADS; i++) {
       cout << "Thread[" << i << "] " << " - Cluster = " << threads_parameters[i][3]*2 << endl;
        mean_chronos[i] = 0;
        for (int j = 0; j < pos_chronos[i]; j++) {
              if (chronos[i][j] > greater)
                     greater = chronos[i][j];
                     cout << "Iter[" << j << "] = " << chronos[i][j] << endl;
            mean_chronos[i] += chronos[i][j];
        }
        mean_chronos[i] /= pos_chronos[i];
        cout << "Mean      = " << mean_chronos[i] << "us; Worst Case = " << greater << "us" <<
endl;
        cout << "Hard Mean = " << total_chronos[i]/ITERATIONS << "us" << endl;
        cout << "Max iter  = " << (threads_parameters[i][2]/BASE_MEMCPY_T)+1 << endl;
        cout << "---------------------------------------" << endl;
    }


}
//*/
```

```
//memcpy vector
int vectors[Traits<Build>::CPUS][32768];

//high priority threads vector (REAL-TIME)
Periodic_Thread * threads[THREADS];

//low priority threads vector
Thread *low_Threads[8];

//definition of clock variable used to calculate random seed
Clock clock;

//thread id vector
unsigned int threads_id[THREADS];

//method that configures iterative fibonacci execution
int iterative_fib_test (int id) {
    float ret = 1.33;
    int fib = 1;
    int temp = 1;
    int prev = 1;
    int max = (int) ((int(threads_parameters[id][2])));
    for (int i = 0; i < iterations[id]; i++) {
        Periodic_Thread::wait_next();
        for (int x = 0; x < max; x++) {
            fib = 1;
            prev = 1;
            for (int j = 1; j < 1000; j++) {
                temp = prev+fib;
                prev = fib;
                fib = temp;
            }
            ret *= fib;
        }
    }
    return int(ret);
}

//method that runs fibonacci recursively
int fib_test (int id) {
    float ret = 1.33;
    int max = (int) ((int(threads_parameters[id][2])/BASE_RECURSIVE_FIB_T));
    for (int i = 0; i < iterations[id]; i++) {
        Periodic_Thread::wait_next();
        for (int x = 0; x < max; x++) {
            ret *= fib(25);
        }
    }
    return int(ret);
}

//method that runs memcpy method
int test(int id) {
```

```
    Random * rand;
    float ret = 1.33;
    int max = (int) ((threads_parameters[id][2]/BASE_MEMCPY_T));
    for (int i = 0; i < iterations[id]; i++) {
        Periodic_Thread::wait_next();
        rand->seed(clock.now());
        for (int x = 0; x < max; x++) {
            memcpy(reinterpret_cast<void *>(&vectors[(x*3)%3]),reinterpret_cast<void
*>(&vectors[rand->random() % Machine::n_cpus()]),sizeof(vectors[0]));
        }
    }
    return int(ret);
}

//method to execute low_priority tasks, uncomment what you want to be executed
int low_priority() {
    int x = 0;

//*
    float result = 1.33;
    //cout << Machine::cpu_id() << endl;
    while (!Thread::_end_capture) {
        result = result*fib(25);
        x++;
    }
    return int(result); //*/
/*
    Random * rand;
    while (!Thread::_end_capture) {
        memcpy(reinterpret_cast<void *>(&vectors[(x*3)%3]),reinterpret_cast<void
*>(&vectors[rand->random() % Machine::n_cpus()]),sizeof(vectors[0]));
        memcpy(reinterpret_cast<void *>(&vectors[(x*2)%3]),reinterpret_cast<void
*>(&vectors[rand->random() % Machine::n_cpus()]),sizeof(vectors[0]));
        memcpy(reinterpret_cast<void *>(&vectors[(x)%3]),reinterpret_cast<void
*>(&vectors[rand->random() % Machine::n_cpus()]),sizeof(vectors[0]));
        x++;
    }
    return 0; //*/

/*
    int fib = 1;
    int prev = 1;
    int temp = 1;
    while (!Thread::_end_capture) {
        temp = fib+prev;
        prev = fib;
        fib = temp;
    }

    return fib; //*/
}
```

```
int main()
{
    Thread::_end_capture = true;
       // initializing the vectors used to stress CPU
    for (int m = 0; m < Machine::n_cpus(); m++) {
        for (int i = 0; i < 32768; i++) {
            vectors[m][i] = i * m - 2 * m;
        }
    }

    int numero = calc_iterations();
    Thread::_end_capture = false;
    // start
    unsigned long long init_time = ((TSC::time_stamp() * 1000000)/TSC::frequency());
    /*
    *task configuration, to execute memcpy operations change "fib_test" to "test",
    *for iterative method replace to "iterative_fib_test"
    */
    for(int i=0;i<THREADS;i++){
        threads[i] = new Periodic_Thread(RTConf(threads_parameters[i][0], iterations[i],
Thread::READY ,
            Thread::Criterion((RTC::Microsecond) threads_parameters[i][0], (RTC::Microsecond)
threads_parameters[i][1],
            (RTC::Microsecond) int(threads_parameters[i][2]), (threads_parameters[i][3]))),
&fib_test, i);
        threads_id[i] = reinterpret_cast<volatile unsigned int>(threads[i]);
    }
    /*
    for(int i=0;i<8;i++){
        Thread::Configuration   conf(Thread::READY,
                        Thread::Criterion(Thread::LOW,i)
                        );
        low_Threads[i] = new Thread(conf, &low_priority);
    }
    //*/
    // sync
    for(int i=0; i<THREADS;i++) {
        threads[i]->join();
    }

    //mean(); //use to calculate execution mean time

    // stop capturing
    Thread::_end_capture = true;

    /* uncomment this and line 238
    for(int i = 0; i < 8; i++)
       low_Threads[i]->join();
    //*/

    // used to print a series to current deadline misses, wich is thread bound, not cpu bound.
    cout<<"time = "<<(((TSC::time_stamp() * 1000000)/TSC::frequency()) - init_time)<<endl;
```

```
    //used to print thread_ids for sending algorithms
    /*
    cout << "<begin_tseries>" << endl;
    cout << "<";
    for (int i = 0; i < THREADS-1; i++)
        cout << threads_id[i] << ",";
    cout << threads_id[THREADS-1] << ">" << endl;
    cout << "<end_tseries>" << endl;
    //end */

    for(int i = 0; i < THREADS; i++)
        delete threads[i];

    return 0;
}
```

app/test_pedf_traits.h (configuração da aplicação de teste)

```
#ifndef __traits_h
#define __traits_h

#include <system/config.h>

__BEGIN_SYS

// Global Configuration
template<typename T>
struct Traits
{
    static const bool enabled = true;
    static const bool debugged = true;
    static const bool hysterically_debugged = false;
    typedef TLIST<> ASPECTS;
};

template<> struct Traits<Build>
{
    enum {LIBRARY, BUILTIN, KERNEL};
    static const unsigned int MODE = LIBRARY;

    enum {IA32, ARMv7};
    static const unsigned int ARCHITECTURE = IA32;

    enum {PC, Cortex};
    static const unsigned int MACHINE = PC;

    enum {Legacy_PC, eMote3, LM3S811, Zynq};
    static const unsigned int MODEL = Legacy_PC;

    static const unsigned int CPUS = 8;
    static const unsigned int NODES = 1; // > 1 => NETWORKING
};
```

```
// Utilities
template<> struct Traits<Debug>
{
    static const bool error   = true;
    static const bool warning = true;
    static const bool info    = false;
    static const bool trace   = false;
};

template<> struct Traits<Lists>: public Traits<void>
{
    static const bool debugged = hysterically_debugged;
};

template<> struct Traits<Spin>: public Traits<void>
{
    static const bool debugged = hysterically_debugged;
};

template<> struct Traits<Heaps>: public Traits<void>
{
    static const bool debugged = hysterically_debugged;
};


// System Parts (mostly to fine control debugging)
template<> struct Traits<Boot>: public Traits<void>
{
};

template<> struct Traits<Setup>: public Traits<void>
{
};

template<> struct Traits<Init>: public Traits<void>
{
};

// template<> struct Traits<Monitoring>: public Traits<void>
// {
// };

// Mediators
template<> struct Traits<Serial_Display>: public Traits<void>
{
    static const bool enabled = false;
    enum {UART, USB};
    static const int ENGINE = UART;
    static const int COLUMNS = 80;
    static const int LINES = 24;
    static const int TAB_SIZE = 8;
```

```
};

__END_SYS

#include __ARCH_TRAITS_H
#include __MACH_TRAITS_H

__BEGIN_SYS

// Components
template<> struct Traits<Application>: public Traits<void>
{
    static const unsigned int STACK_SIZE = Traits<Machine>::STACK_SIZE;
    static const unsigned int HEAP_SIZE = Traits<Machine>::HEAP_SIZE;
    static const unsigned int MAX_THREADS = Traits<Machine>::MAX_THREADS;
};

template<> struct Traits<System>: public Traits<void>
{
    static const unsigned int mode = Traits<Build>::MODE;
    static const bool multithread = (Traits<Application>::MAX_THREADS > 1);
    static const bool multitask = (mode != Traits<Build>::LIBRARY);
    static const bool multicore = (Traits<Build>::CPUS > 1) && multithread;
    static const bool multiheap = (mode != Traits<Build>::LIBRARY) ||
Traits<Scratchpad>::enabled;

    enum {FOREVER = 0, SECOND = 1, MINUTE = 60, HOUR = 3600, DAY = 86400, WEEK = 604800, MONTH
= 2592000, YEAR = 31536000};
    static const unsigned long LIFE_SPAN = 1 * HOUR; // in seconds

    static const bool reboot = true;

    static const unsigned int STACK_SIZE = Traits<Machine>::STACK_SIZE;
    //configuration of heap was changed to storing data captured
    static const unsigned int HEAP_SIZE = (6 * 8 * 4 * 1024 * 1024) +
(Traits<Application>::MAX_THREADS + 1) * Traits<Application>::STACK_SIZE;
};

template<> struct Traits<Task>: public Traits<void>
{
    static const bool enabled = Traits<System>::multitask;
};

template<> struct Traits<Thread>: public Traits<void>
{
    static const bool smp = Traits<System>::multicore;

    typedef Scheduling_Criteria::MPEDF Criterion;
    static const unsigned int QUANTUM = 5000; // us

    static const bool trace_idle = hysterically_debugged;
};
```

```
template<> struct Traits<Scheduler<Thread> >: public Traits<void>
{
    static const bool debugged = Traits<Thread>::trace_idle || hysterically_debugged;
};

template<> struct Traits<Periodic_Thread>: public Traits<void>
{
    static const bool simulate_capacity = false;
};

template<> struct Traits<Address_Space>: public Traits<void>
{
    static const bool enabled = Traits<System>::multiheap;
};

template<> struct Traits<Segment>: public Traits<void>
{
    static const bool enabled = Traits<System>::multiheap;
};

template<> struct Traits<Alarm>: public Traits<void>
{
    static const bool visible = hysterically_debugged;
};

template<> struct Traits<Synchronizer>: public Traits<void>
{
    static const bool enabled = Traits<System>::multithread;
};

template<> struct Traits<Network>: public Traits<void>
{
    static const bool enabled = (Traits<Build>::NODES > 1);

    static const unsigned int RETRIES = 3;
    static const unsigned int TIMEOUT = 10; // s

    // This list is positional, with one network for each NIC in Traits<NIC>::NICS
    typedef LIST<IP> NETWORKS;
};

//template<> struct Traits<ELP>: public Traits<Network>
//{
//    static const bool enabled = NETWORKS::Count<ELP>::Result;
//
//    static const bool acknowledged = true;
//};

template<> struct Traits<TSTP>: public Traits<Network>
{
    static const bool enabled = NETWORKS::Count<TSTP>::Result;
    static const unsigned int KEY_SIZE = 16;
};
```

```
// template<> template <typename S> struct Traits<Smart_Data<S>>: public Traits<Network>
// {
//     static const bool enabled = NETWORKS::Count<TSTP>::Result;
// };

template<> struct Traits<IP>: public Traits<Network>
{
    static const bool enabled = NETWORKS::Count<IP>::Result;

    enum {STATIC, MAC, INFO, RARP, DHCP};

    struct Default_Config {
        static const unsigned int  TYPE    = DHCP;
        static const unsigned long ADDRESS = 0;
        static const unsigned long NETMASK = 0;
        static const unsigned long GATEWAY = 0;
    };

    template<unsigned int UNIT>
    struct Config: public Default_Config {};

    static const unsigned int TTL  = 0x40; // Time-to-live
};

template<> struct Traits<IP>::Config<0> //: public Traits<IP>::Default_Config
{
    static const unsigned int  TYPE      = MAC;
    static const unsigned long ADDRESS   = 0x0a000100;  // 10.0.1.x x=MAC[5]
    static const unsigned long NETMASK   = 0xffffff00;  // 255.255.255.0
    static const unsigned long GATEWAY   = 0;           // 10.0.1.1
};

template<> struct Traits<IP>::Config<1>: public Traits<IP>::Default_Config
{
};

template<> struct Traits<UDP>: public Traits<Network>
{
    static const bool checksum = true;
};

template<> struct Traits<TCP>: public Traits<Network>
{
    static const unsigned int WINDOW = 4096;
};

template<> struct Traits<DHCP>: public Traits<Network>
{
};


__END_SYS
```

```
#endif
```

```
include/architecture/ia32/cpu.h:
novas funções para ler MSRs e para configurar a modulação de clock;
 - pp1_energy_status()
 - pp0_energy_status()
 - pkg_energy_status()
 - dram_energy_status()
 - rapl_power_unit()
 - rapl_energy_unit()
     Documentação da leitura de msrs em 14.9.3 “Package RAPL Domain” do Intel SDM
 - temperature()
 - clock (Reg64 clock)
```

```
// EPOS IA32 CPU Mediator Declarations

#ifndef __ia32_h
#define __ia32_h

#include <cpu.h>

__BEGIN_SYS

class CPU: public CPU_Common
{
...
    //check Intel SDM, 14.9.3 "Package RAPL Domain"
    static unsigned long long rapl_energy_unit() {
        Reg64 rapl_energy_read = rdmsr(0x606);
        rapl_energy_read >>= 8;
        rapl_energy_read &= (1ULL << 5) - 1;
        return (rapl_energy_read);
    }
    //indicates the value to be used while measuring energy
    static unsigned long long rapl_power_unit() {
        Reg64 rapl_power_read = rdmsr(0x606);
        rapl_power_read &= (1ULL << 4) - 1;
        return (rapl_power_read);
    }
    //energy consumed by the entire package
    static unsigned long long pkg_energy_status() {
        Reg64 pkg_energy_read = rdmsr(0x611);
        pkg_energy_read &= (1ULL << 32) - 1;
        return (pkg_energy_read);
    }
    //energy consumed by the dram, check for compatibility
    static unsigned long long dram_energy_status() {
        Reg64 dram_energy_read = rdmsr(0x619);
        dram_energy_read &= (1ULL << 32) - 1;
        return (dram_energy_read);
    }
    //energy consumed by the cores
```

```
    static unsigned long long pp0_energy_status() {
        Reg64 pp0_energy_read = rdmsr(0x639);
        pp0_energy_read &= (1ULL << 32) - 1;
        return (pp0_energy_read);
    }
    //energy on cache and integrated GPU, check for compatibility
    static unsigned long long pp1_energy_status() {
        Reg64 pp1_energy_read = rdmsr(0x641);
        pp1_energy_read &= (1ULL << 32) - 1;
        return (pp1_energy_read);
    }
...
   static unsigned int temperature() {
        Reg64 therm_read = rdmsr(THERM_STATUS);
        Reg64 temp_target_read = rdmsr(TEMPERATURE_TARGET);
        //temperature is measured by taking from the target the value on status
        int bits = 22 - 16 + 1;
        therm_read >>= 16;
        therm_read &= (1ULL << bits) - 1;
        bits = 23 - 16 + 1;
        temp_target_read >>= 16;
        temp_target_read &= (1ULL << bits) - 1;
        return (temp_target_read - therm_read);
    }
...
   static void clock(Reg64 clock) {
        // clock must be taken as Reg64 because Hertz is Reg32 is some configurations and
that's not enough for the comparisons bellow
        unsigned int dc;
        Reg64 cpu_clock_aux = _cpu_clock;
        if(clock <= cpu_clock_aux * 1875 / 10000)
            dc = 0b10011;   // Minimum duty cycle of 12.5 %
        else if(clock >= cpu_clock_aux * 9375 / 10000)
            dc = 0b01001;   // Disable duty cycling and operate at full speed
        else
            dc = 0b10001 | ((clock * 10000 / cpu_clock_aux + 625) / 625); // Dividing by 625
instead of 1250 eliminates the shift left
        wrmsr(CLOCK_MODULATION, dc);
    }
...
}
```

include/architecture/ia32/monitoring_capture.h (definicao do sistema de monitoramento)

```
#ifndef __ia32_monitoring_capture_h
#define __ia32_monitoring_capture_h

#include <machine.h>
#include <pmu.h>
#include <nic.h>
#include <tsc.h>
```

```
__BEGIN_SYS
struct Moment {
    unsigned long long _temperature;
    unsigned long long _pmu0;
    unsigned long long _pmu1;
    unsigned long long _pmu2;
    unsigned long long _pmu3;
    unsigned long long _pmu4;
    unsigned long long _pmu5;
    unsigned long long _pmu6;
    unsigned long long _time_stamp;
    unsigned long long _thread_id;
    long long _thread_priority;
    unsigned long long _deadline;
    unsigned long long _global_deadline;
    unsigned long long _pkg_energy;
    unsigned long long _pp0_energy;

    //"constructor"
    Moment() {}

    // used to make a copy of values
    // static void copy (Moment from, Moment *to) {
    //     to->_temperature = from._temperature;
    //     to->_pmu0 = from._pmu0;
    //     to->_pmu1 = from._pmu1;
    //     to->_pmu2 = from._pmu2;
    //     to->_pmu3 = from._pmu3;
    //     to->_pmu4 = from._pmu4;
    //     to->_pmu5 = from._pmu5;
    //     to->_pmu6 = from._pmu6;
    //     to->_time_stamp = from._time_stamp;
    //     to->_thread_id = from._thread_id;
    //     to->_thread_priority = from._thread_priority;
    //     to->_deadline = from._deadline;
    //     to->_global_deadline = from._global_deadline;

    // }
};

class Monitoring_Capture {

public:
       Moment * _mem_moment; //pointer to vector of moments
       unsigned int _init_pos[Traits<Build>::CPUS]; //initial positions of each cpu
    unsigned int _mem_pos[Traits<Build>::CPUS]; // number of stored captures of each cpu
    int _max_size; //max size of the cpu subvectors
       unsigned int _over[Traits<Build>::CPUS]; //number of captures unstored per cpu

    // smart_data_info
    unsigned long long _t0; //initial execution time
    unsigned long long _t1; //final execution time
    NIC::Address _mac; //mac address -> not used anymore
```

```
    unsigned long long _tsc_base; //base tsc time
    unsigned int _units[12]; //units vector (ignoring rapl)
    //coordinates
    int _x;
    int _y;
    int _z;
    unsigned int _r; //radius - distance from the central point (x,y,z)
    unsigned int _errorsmart; //smardata error value
    unsigned int _confidence; //confidence smartdata value

public:

    //constructor
    Monitoring_Capture(int size, Moment * init) {
       //storing config
        _max_size = size;
       for (unsigned int i = 0; i < Traits<Build>::CPUS; i++) {
              _mem_pos[i] = i * size;
              _init_pos[i] = _mem_pos[i];
              _over[i] = 0;
       }
        _mem_moment = init;
        //metadata config
        _t0 = RTC::seconds_since_epoch() * 1000000;
        _t1 = _t0 + (5*60*1000000);
        _tsc_base = _t0 - (TSC::time_stamp() * 1000000 / TSC::frequency());
        NIC nic;
        _mac = nic.address();
        for (int i = 0; i < 8; i++)
            _units[i] = (i+1) << 16 | 8;
        _units[8] = ((PMU::_channel_3+8)+1) << 16 | 8;
        _units[9] = ((PMU::_channel_4+8)+1) << 16 | 8;
        _units[10] = ((PMU::_channel_5+8)+1) << 16 | 8;
        _units[11] = ((PMU::_channel_6+8)+1) << 16 | 8;
        _x =  3746654;//3765.829 * 100000;
        _y = -4237592;
        _z = -293735;
        _r = 0;
        _errorsmart = 0;
        _confidence = 1;
        //print basic metadata before system execution
        print_smart_params();
        series();
    }

    //basic destructor
    ~Monitoring_Capture () {
        delete _mem_moment;
    }

    //stores a capture using the params
    void capture(unsigned int temperature, unsigned long long pmu0, unsigned long long pmu1,
unsigned long long pmu2, unsigned long long pmu3, unsigned long long pmu4, unsigned long long
```

```
pmu5,
                unsigned long long pmu6, unsigned int thread_id, long long thread_priority,
unsigned int cpu_id, unsigned int deadline, unsigned int global_deadline, unsigned long long
pkg, unsigned long long pp0) {
        if (_mem_pos[cpu_id] < ((cpu_id+1) * _max_size)) {
            unsigned int pos = _mem_pos[cpu_id];
            _mem_moment[pos]._temperature = (unsigned long long)temperature;
            _mem_moment[pos]._pmu0 = pmu0;
            _mem_moment[pos]._pmu1 = pmu1;
            _mem_moment[pos]._pmu2 = pmu2;
            _mem_moment[pos]._pmu3 = pmu3;
            _mem_moment[pos]._pmu4 = pmu4;
            _mem_moment[pos]._pmu5 = pmu5;
            _mem_moment[pos]._pmu6 = pmu6;
            _mem_moment[pos]._time_stamp = _tsc_base + (TSC::time_stamp() * 1000000 /
TSC::frequency());
            _mem_moment[pos]._thread_id = (unsigned long long)thread_id;
            _mem_moment[pos]._thread_priority = (long long)thread_priority;
            _mem_moment[pos]._deadline = (unsigned long long)deadline;
            _mem_moment[pos]._global_deadline = (unsigned long long)global_deadline;
            _mem_moment[pos]._pkg_energy = pkg;
            _mem_moment[pos]._pp0_energy = pp0;
            //Moment::copy(m, &_mem_moment[_mem_pos[cpu_id]]);
            _mem_pos[cpu_id]++;
           // if (!_circular_cpu_data[cpu_id]->next()) {
           //      send(cpu_id);
           // }
           // _circular_cpu_data[cpu_id]->insert(m);
        }
         /*else {
            _over[cpu_id]++;
        }//*/
    }

    //returns time of the system (reuse of the values)
    unsigned long long time () {
        return _tsc_base + (TSC::time_stamp() * 1000000 / TSC::frequency());
    }

    //returns the value of the last capture of a channel/event on a cpu
    unsigned long long last_capture(unsigned int cpu_id, unsigned int channel) {
        unsigned int pos = _mem_pos[cpu_id] - 1;
        if (pos < 0) {
            return 0;
        }
        switch (channel) {
            case 0:
                return _mem_moment[pos]._pmu0;
            case 1:
                return _mem_moment[pos]._pmu1;
            case 2:
                return _mem_moment[pos]._pmu2;
            case 3:
```

```
                return _mem_moment[pos]._pmu3;
            case 4:
                return _mem_moment[pos]._pmu4;
            case 5:
                return _mem_moment[pos]._pmu5;
            case 6:
                return _mem_moment[pos]._pmu6;
            default:
                return _mem_moment[pos]._time_stamp;
        }
    }

    void series();

    void datas();

    void print_smart_params();

    //returns the position of the storing of a cpu
    unsigned int  get_mem_pos(int cpu_id) {
        return _mem_pos[cpu_id];
    }

};

__END_SYS

#endif //monitoring_capture_h
```

src/architecture/ia32/monitoring_capture_send.cc (métodos de impressão dos dados)

```
#include <architecture/ia32/monitoring_capture.h>
#include <utility/ostream.h>
__BEGIN_SYS

OStream cout;
//printing captured data -> comment line 9 to print all the data
void Monitoring_Capture::datas() {
       unsigned long long final_time = RTC::seconds_since_epoch() *1000000;
       /*
       cout << "<end_capture>" << endl;
       for (unsigned int i = 0; i < Machine::n_cpus(); i++) {
              cout << "init_temp" << i << ": " << _mem_moment[_init_pos[i]]._temperature <<
endl;
              cout << "cap  CPU" << i << ": " << _mem_pos[i] - _init_pos[i] << endl;
              cout << "over CPU" << i << ": " << _over[i] << endl;
              cout << "deadlines " << i << ": " <<_mem_moment[_mem_pos[i]-1]._deadline <<
endl;
       }
       cout << "global_ddl_m" << ": " <<_mem_moment[_mem_pos[0]-1]._global_deadline << endl;
       //cout << "final time | Elapsed time: " << final_time << " | " << (final_time -
_t0)/1000000 << endl;
```

```
        while(1) cout<<"simulation ended"<<endl;
        //*/
        cout << "<begin_capture>" << endl;
        Moment m;
        for (unsigned int i = 0; i < Machine::n_cpus(); i++) {
                //continue; //discomment if you don't want to print the collected data
                for (unsigned int j = _init_pos[i]; j < _mem_pos[i]; j++) {
                        m = _mem_moment[j];
                        unsigned long long value[] = {m._temperature, m._pmu0, m._pmu1, m._pmu2,
m._pmu3, m._pmu4, m._pmu5, m._pmu6,
                                                m._time_stamp, m._thread_id, m._thread_priority, i,
m._deadline, m._global_deadline, m._pkg_energy, m._pp0_energy};
                        cout << "<";
                        for (int i = 0; i < 15; i++) {
                                cout << value[i] << ",";
                        }
                        cout << value[15] << ">" << endl;
                }
                //break;
        }
        cout << "<end_capture>" << endl;
        for (unsigned int i = 0; i < Machine::n_cpus(); i++) {
                cout << "cap  CPU" << i << ": " << _mem_pos[i] - _init_pos[i] << endl;
                cout << "over CPU" << i << ": " << _over[i] << endl;
        }
        cout << "final time | Elapsed time: " << final_time << " | " << (final_time -
_t0)/1000000 << endl;
        while(1) cout<<"simulation ended"<<endl;
}

//print series(metadata)
void Monitoring_Capture::series() {
    //*
    cout << "<begin_series>" << endl;
    for (unsigned int cpu = 0; cpu < Machine::n_cpus(); cpu++)
                for (int i = 0; i < 12; i++) {
                                if (i != 4) {
                                        cout << "+\n"
                                        << "{\n"
                                        << "\t\"series\" :\n"
                                        << "\t{\n"
                                        << "\t\t\"version\" : \"1.1\",\n"
                                        << "\t\t\"unit\" : " << _units[i] << ",\n"
                                        << "\t\t\"x\" : " << _x + cpu*10 << ",\n"
                                        << "\t\t\"y\" : " << _y << ",\n"
                                        << "\t\t\"z\" : " << _z << ",\n"
                                        << "\t\t\"r\" : " << _r << ",\n"
                                        << "\t\t\"t0\" : " << _t0 << ",\n"
                                        << "\t\t\"t1\" : " << _t1 << ",\n"
                                        << "\t\t\"dev\" : " << cpu << "\n"
                                        << "\t}\n"
                                        << "}\n"
                                        << "+" << endl;
```

```
                        }
                }
        cout << "<end_series>" << endl;//*/
}
//print the parameters to mount the smartdatas
void Monitoring_Capture::print_smart_params() {
        //*
        cout << "<begin_params>" << endl;
        cout << "<" << _mac << ",";
        for (int i = 0; i < 12; i++)
                cout << _units[i] << ",";
        cout << _t0 << ",";
        cout << _t1 << ">" << endl;
        cout << "<end_params>" << endl;
        //*/
}

__END_SYS
```

include/pmu.h
 - Adicionado a configuração dos eventos da PMU versão Sandy Bridge ao enum Event;
 - Adicionado variaveis para seleção dos canais da PMU a partir do enum Event;

```
// EPOS PMU Mediator Common Package

#ifndef __pmu_h
#define __pmu_h

#include <system/config.h>

__BEGIN_SYS

class PMU_Common
{
public:
    typedef unsigned int Channel;
    typedef long long int Count;

    enum Event {
        CLOCK, //0
        DVS_CLOCK, //1
        INSTRUCTION, //2
        BRANCH, //3
        BRANCH_MISS, //4
        L1_HIT, //5
        L2_HIT, //6
        L3_HIT, //7
        LLC_HIT = L3_HIT,
        CACHE_HIT = LLC_HIT,
        L1_MISS, //8
        L2_MISS, //9
        L3_MISS, //10
```

```
        LLC_MISS = L3_MISS,
        CACHE_MISS = LLC_MISS,
        LLC_HITM, //11
        //Sandy_Bridge_events
        LD_BLOCKS_DATA_UNKNOWN_C, //#12
        LD_BLOCKS_STORE_FORWARD_C,
        LD_BLOCKS_NO_SR_C,
        LD_BLOCKS_ALL_BLOCK_C,
        MISALIGN_MEM_REF_LOADS_C,
        MISALIGN_MEM_REF_STORES_C,
        LD_BLOCKS_PARTIAL_ADDRESS_ALIAS_C,
        LD_BLOCKS_PARTIAL_ALL_STA_BLCOK_C,
        DTLB_LOAD_MISSES_MISS_CAUSES_A_WALK_C,
        DTLB_LOAD_MISSES_MISS_WALK_COMPLETED_C,
        DTLB_LOAD_MISSES_MISS_WALK_DURATION_C,
        DTLB_LOAD_MISSES_MISS_STLB_HIT_C,
        INT_MISC_RECOVERY_CYCLES_C,
        INT_MISC_RAT_STALL_CYCLES_C,
        UOPS_ISSUED_ANY_C,
        FP_COMP_OPS_EXE_X87_C,
        FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE_C,
        FP_COMP_OPS_EXE_SSE_FP_SCALAR_SINGLE_C,
        FP_COMP_OPS_EXE_SSE_PACKED_SINGLE_C,
        FP_COMP_OPS_EXE_SSE_SCALAR_DOUBLE_C,
        SIMD_FP_256_PACKED_SINGLE_C,
        SIMD_FP_256_PACKED_DOUBLE_C,
        ARITH_FPU_DIV_ACTIVE_C,
        INSTS_WRITTEN_TO_IQ_INSTS_C,
        L2_RQSTS_DEMAND_DATA_RD_HIT_C,
        L2_RQSTS_ALL_DEMAND_DATA_RD_C,
        L2_RQSTS_RFO_HITS_C,
        L2_RQSTS_RFO_MISS_C,
        L2_RQSTS_ALL_RFO_C,
        L2_RQSTS_CODE_RD_HIT_C,
        L2_RQSTS_CODE_RD_MISS_C,
        L2_RQSTS_ALL_CODE_RD_C,
        L2_RQSTS_PF_HIT_C,
        L2_RQSTS_PF_MISS_C,
        L2_RQSTS_ALL_PF_C,
        L2_STORE_LOCK_RQSTS_MISS_C,
        L2_STORE_LOCK_RQSTS_HIT_E_C,
        L2_STORE_LOCK_RQSTS_HIT_M_C,
        L2_STORE_LOCK_RQSTS_ALL_C,
        L2_L1D_WB_RQSTS_HIT_E_C,
        L2_L1D_WB_RQSTS_HIT_M_C,
        LONGEST_LAT_CACHE_REFERENCE_C, //table 19-1 architectural event
        LONGEST_LAT_CACHE_MISS_C, //table 19-1 architectural event
        CPU_CLK_UNHALTED_THREAD_P_C, //table 19-1 architectural event
        CPU_CLK_THREAD_UNHALTED_REF_XCLK_C, //table 1901 architectural event
        L1D_PEND_MISS_PENDING_C, //counter 2 only - set cmask = 1 to count cycles
        DTLB_STORE_MISSES_MISS_CAUSES_A_WALK_C,
        DTLB_STORE_MISSES_WALK_COMPLETED_C,
        DTLB_STORE_MISSES_WALK_DURATION_C,
```

```
DTLB_STORE_MISSES_TLB_HIT_C,
LOAD_HIT_PRE_SW_PF_C,
LOAD_HIT_PREHW_PF_C,
HW_PRE_REQ_DL1_MISS_C,
L1D_REPLACEMENT_C,
L1D_ALLOCATED_IN_M_C,
L1D_EVICTION_C,
L1D_ALL_M_REPLACEMENT_C,
PARTIAL_RAT_STALLS_FLAGS_MERGE_UOP_C,
PARTIAL_RAT_STALLS_SLOW_LEA_WINDOW_C,
PARTIAL_RAT_STALLS_MUL_SINGLE_UOP_C,
RESOURCE_STALLS2_ALL_FL_EMPTY_C,
RESOURCE_STALLS2_ALL_PRF_CONTROL_C,
RESOURCE_STALLS2_BOB_FULL_C,
RESOURCE_STALLS2_OOO_RSRC_C,
CPL_CYCLES_RING0_C, //use edge to count transition
CPL_CYCLES_RING123_C,
RS_EVENTS_EMPTY_CYCLES_C,
OFFCORE_REQUESTS_OUTSTANDING_DEMAND_DATA_RD_C,
OFFCORE_REQUESTS_OUTSTANDING_DEMAND_RFO_C,
OFFCORE_REQUESTS_OUTSTANDING_ALL_DATA_RD_C,
LOCK_CYCLES_SPLIT_LOCK_UC_LOCK_DURATION_C,
LOCK_CYCLES_CACHE_LOCK_DURATION_C,
IDQ_EMPTY_C,
IDQ_MITE_UOPS_C,
IDQ_DSB_UOPS_C,
IDQ_MS_DSB_UOPS_C,
IDQ_MS_MITE_UOPS_C,
IDQ_MS_UOPS_C,
ICACHE_MISSES_C,
ITLB_MISSES_MISS_CAUSES_A_WALK_C,
ITLB_MISSES_WALK_COMPLETED_C,
ITLB_MISSES_WALK_DURATION_C,
ITLB_MISSES_STLB_HIT_C,
ILD_STALL_LCP_C,
ILD_STALL_IQ_FULL_C,
BR_INST_EXEC_COND_C,
BR_INST_EXEC_DIRECT_JMP_C,
BR_INST_EXEC_INDIRECT_JMP_NON_CALL_RET_C,
BR_INST_EXEC_RETURN_NEAR_C,
BR_INST_EXEC_DIRECT_NEAR_CALL_C,
BR_INST_EXEC_INDIRECT_NEAR_CALL_C,
BR_INST_EXEC_NON_TAKEN_C,
BR_INST_EXEC_TAKEN_C,
BR_INST_EXEC_ALL_BRANCHES_C,
BR_MISP_EXEC_COND_C,
BR_MISP_EXEC_INDIRECT_JMP_NON_CALL_RET_C,
BR_MISP_EXEC_RETURN_NEAR_C,
BR_MISP_EXEC_DIRECT_NEAR_CALL_C,
BR_MISP_EXEC_INDIRECT_NEAR_CALL_C,
BR_MISP_EXEC_NON_TAKEN_C,
BR_MISP_EXEC_TAKEN_C,
BR_MISP_EXEC_ALL_BRANCHES_C,
```

```
IDQ_UOPS_NOT_DELIVERED_CORE_C,
UOPS_DISPATCHED_PORT_PORT_0_C,
UOPS_DISPATCHED_PORT_PORT_1_C,
UOPS_DISPATCHED_PORT_PORT_2_LD_C,
UOPS_DISPATCHED_PORT_PORT_2_STA_C,
UOPS_DISPATCHED_PORT_PORT_2_C,
UOPS_DISPATCHED_PORT_PORT_3_LD_C,
UOPS_DISPATCHED_PORT_PORT_3_STA_C,
UOPS_DISPATCHED_PORT_PORT_3_C,
UOPS_DISPATCHED_PORT_PORT_4_C,
UOPS_DISPATCHED_PORT_PORT_5_C,
RESOURCE_STALLS_ANY_C,
RESOURCE_STALLS_LB_C,
RESOURCE_STALLS_RS_C,
RESOURCE_STALLS_SB_C,
RESOURCE_STALLS_ROB_C,
RESOURCE_STALLS_FCSW_C,
RESOURCE_STALLS_MXCSR_C,
RESOURCE_STALLS_OTHER_C,
DSB2MITE_SWITCHES_COUNT_C,
DSB2MITE_SWITCHES_PENALTY_CYCLES_C,
DSB_FILL_OTHER_CANCEL_C,
DSB_FILL_EXCEED_DSB_LINES_C,
DSB_FILL_ALL_CANCEL_C,
ITLB_ITLB_FLUSH_C,
OFFCORE_REQUESTS_DEMAND_DATA_RD_C,
OFFCORE_REQUESTS_DEMAND_RFO_C,
OFFCORE_REQUESTS_ALL_DATA_RD_C,
UOPS_DISPATCHED_THREAD_C,
UOPS_DISPATCHED_CORE_C,
OFFCORE_REQUESTS_BUFFER_SQ_FULL_C,
AGU_BYPASS_CANCEL_COUNT_C,
OFF_CORE_RESPONSE_0_C,
OFF_CORE_RESPONSE_1_C,
TLB_FLUSH_DTLB_THREAD_C,
TLB_FLUSH_STLB_ANY_C,
L1D_BLOCKS_BANK_CONFLICT_CYCLES_C,
INST_RETIRED_ANY_P_C, //table 19-1 architectural event
INST_RETIRED_PREC_DIST_C, //PMC1 only; must quiesce other PMCs
OTHER_ASSISTS_ITLB_MISS_RETIRED_C,
OTHER_ASSISTS_AVX_STORE_C,
OTHER_ASSISTS_AVX_TO_SSE_C,
OTHER_ASSISTS_SSE_TO_AVX_C,
UOPS_RETIRED_ALL_C,
UOPS_RETIRED_RETIRE_SLOTS_C,
MACHINE_CLEARS_MEMORY_ORDERING_C,
MACHINE_CLEARS_SMC_C,
MACHINE_CLEARS_MASKMOV_C,
BR_INST_RETIRED_ALL_BRANCHES_ARCH_C, //table 19-1
BR_INST_RETIRED_CONDITIONAL_C,
BR_INST_RETIRED_NEAR_CALL_C,
BR_INST_RETIRED_ALL_BRANCHES_C,
BR_INST_RETIRED_NEAR_RETURN_C,
```

```
        BR_INST_RETIRED_NOT_TAKEN_C,
        BR_INST_RETIRED_NEAR_TAKEN_C,
        BR_INST_RETIRED_FAR_BRANCH_C,
        BR_MISP_RETIRED_ALL_BRANCHES_ARCH_C, //table 19-1
        BR_MISP_RETIRED_CONDITIONAL_C,
        BR_MISP_RETIRED_NEAR_CALL_C,
        BR_MISP_RETIRED_ALL_BRANCHES_C,
        BR_MISP_RETIRED_NOT_TAKEN_C,
        BR_MISP_RETIRED_TAKEN_C,
        FP_ASSIST_X87_OUTPUT_C,
        FP_ASSIST_X87_INPUT_C,
        FP_ASSIST_SIMD_OUTPUT_C,
        FP_ASSIST_SIMD_INPUT_C,
        FP_ASSIST_ANY_C,
        ROB_MISC_EVENTS_LBR_INSERTS_C,
        MEM_TRANS_RETIRED_LOAD_LATENCY_C, //specify threshold in MSR 0x3F6
        MEM_TRANS_RETIRED_PRECISE_STORE_C, //see section 18.8.4.3
        MEM_UOP_RETIRED_LOADS_C,
        MEM_UOP_RETIRED_STORES_C,
        MEM_UOP_RETIRED_STLB_MISS_C,
        MEM_UOP_RETIRED_LOCK_C,
        MEM_UOP_RETIRED_SPLIT_C,
        MEM_UOP_RETIRED_ALL_C,
        MEM_UOPS_RETIRED_ALL_LOADS_C, // Supports PEBS. PMC0-3 only regardless HTT.
        MEM_LOAD_UOPS_RETIRED_L1_HIT_C,
        MEM_LOAD_UOPS_RETIRED_L2_HIT_C,
        MEM_LOAD_UOPS_RETIRED_L3_HIT_C,
        MEM_LOAD_UOPS_RETIRED_HIT_LFB_C,
        XSNP_MISS_C,
        XSNP_HIT_C,
        XSNP_HITM_C,
        XSNP_NONE_C,
        MEM_LOAD_UOPS_MISC_RETIRED_LLC_MISS_C,
        L2_TRANS_DEMAND_DATA_RD_C,
        L2_TRANS_RFO_C,
        L2_TRANS_CODE_RD_C,
        L2_TRANS_ALL_PF_C,
        L2_TRANS_L1D_WB_C,
        L2_TRANS_L2_FILL_C,
        L2_TRANS_L2_WB_C,
        L2_TRANS_ALL_REQ_UESTS_C,
        L2_LINES_IN_I_C,
        L2_LINES_IN_S_C,
        L2_LINES_IN_E_C,
        L2_LINES_IN_ALL_C,
        L2_LINES_OUT_DEMAND_CLEAN_C,
        L2_LINES_OUT_DEMAND_DIRTY_C,
        L2_LINES_OUT_DEMAND_PF_CLEAN_C,
        L2_LINES_OUT_DEMAND_PF_DIRTY_C, // last execution end
        L2_LINES_OUT_DEMAND_DIRTY_ALL_C, //#216 should not be used as parameter at _channel_3
-> use 214
        SQ_MISC_SPLIT_LOCK_C, //217
        EVENTS
```

```
    };

    enum Flags {
        NONE,
        INT
    };
    //first test 12. last normal test 212. last test 214
    static const unsigned int _channel_3 = 26; // sum 4 after each execution
    static const unsigned int _channel_4 = _channel_3+1;
    static const unsigned int _channel_5 = _channel_3+2;
    static const unsigned int _channel_6 = _channel_3+3;

protected:
    PMU_Common() {}
};

__END_SYS

#ifdef __PMU_H
#include __PMU_H
#endif

#endif
```

include/scheduler.h
- Configuração dos escalonadores com monitoramento habilitado;
- método init

```
// EPOS Scheduler Component Declarations

#ifndef __scheduler_h
#define __scheduler_h

#include <utility/list.h>
#include <cpu.h>
#include <machine.h>
#include <pmu.h>

__BEGIN_SYS

// All scheduling criteria, or disciplines, must define operator int() with
// the semantics of returning the desired order of a given object within the
// scheduling list
namespace Scheduling_Criteria
{
    // Priority (static and dynamic)
    class Priority
    {
        friend class _SYS::RT_Thread;

    public:
        enum {
```

```
            MAIN   = 0,
            HIGH   = 1,
            NORMAL = (unsigned(1) << (sizeof(int) * 8 - 1)) - 3,
            LOW    = (unsigned(1) << (sizeof(int) * 8 - 1)) - 2,
            IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1
        };

        static const bool timed = false;
        static const bool dynamic = false;
        static const bool preemptive = true;
        static const bool monitoring = false; // enables monitoring

    public:
        Priority(int p = NORMAL): _priority(p) {}

        operator const volatile int() const volatile { return _priority; }
        static void init() {} //created to configure and initialize pmu
        void update() {}
        unsigned int queue() const { return 0; }

    protected:
        volatile int _priority;
    };
...
     class MGEDF : public GEDF
      {
      public:
          static const bool monitoring = true;
      public:
          static void init() {
            PMU::stop(0);
            PMU::stop(1);
            PMU::stop(2);
            PMU::stop(3);
            PMU::stop(4);
            PMU::stop(5);
            PMU::stop(6);

            PMU::reset(0);
            PMU::reset(1);
            PMU::reset(2);
            PMU::reset(3);
            PMU::reset(4);
            PMU::reset(5);
            PMU::reset(6);

            PMU::write(0, 0);
            PMU::write(1, 0);
            PMU::write(2, 0);
            PMU::write(3, 0);
            PMU::write(4, 0);
            PMU::write(5, 0);
            PMU::write(6, 0);
```

```
            PMU::config(3, (PMU::Event)PMU::_channel_3);
            PMU::config(4, (PMU::Event)PMU::_channel_4);
            PMU::config(5, (PMU::Event)PMU::_channel_5);
            PMU::config(6, (PMU::Event)PMU::_channel_6);

            PMU::start(0);
            PMU::start(1);
            PMU::start(2);
            PMU::start(3);
            PMU::start(4);
            PMU::start(5);
            PMU::start(6);
          }
          MGEDF(int p = APERIODIC)
          : GEDF(p) {} // Aperiodic

          MGEDF(const Microsecond & d, const Microsecond & p = SAME, const Microsecond & c =
UNKNOWN, int cpu = ANY)
          : GEDF(d, p, c, cpu) {}

          static unsigned int queue() { return current_head(); }
          static unsigned int current_head() { return Machine::cpu_id(); }
      };
...
     class MPEDF : public PEDF
      {
       enum { ANY = Variable_Queue::ANY };
      public:
          static const bool monitoring = true;
          static const bool power_cap = true;
      public:
          static void init() {
            PMU::stop(0);
            PMU::stop(1);
            PMU::stop(2);
            PMU::stop(3);
            PMU::stop(4);
            PMU::stop(5);
            PMU::stop(6);

            PMU::reset(0);
            PMU::reset(1);
            PMU::reset(2);
            PMU::reset(3);
            PMU::reset(4);
            PMU::reset(5);
            PMU::reset(6);

            PMU::write(0, 0);
            PMU::write(1, 0);
            PMU::write(2, 0);
            PMU::write(3, 0);
```

```
            PMU::write(4, 0);
            PMU::write(5, 0);
            PMU::write(6, 0);

            PMU::config(3, (PMU::Event)PMU::_channel_3);
            PMU::config(4, (PMU::Event)PMU::_channel_4);
            PMU::config(5, (PMU::Event)PMU::_channel_5);
            PMU::config(6, (PMU::Event)PMU::_channel_6);

            PMU::start(0);
            PMU::start(1);
            PMU::start(2);
            PMU::start(3);
            PMU::start(4);
            PMU::start(5);
            PMU::start(6);
        }
         MPEDF(int p = APERIODIC)
         : PEDF(p) {} // Aperiodic

         MPEDF(const Microsecond & d, const Microsecond & p = SAME, const Microsecond & c =
UNKNOWN, int cpu = ANY)
         : PEDF(d, p, c, cpu) {}

         using Variable_Queue::queue;

         static unsigned int current_head() { return Machine::cpu_id(); }
     };
...
    class MCEDF : public CEDF
     {
       enum { ANY = Variable_Queue::ANY };
     public:
         static const bool monitoring = true;
     public:
         static void init() {
           PMU::stop(0);
           PMU::stop(1);
           PMU::stop(2);
           PMU::stop(3);
           PMU::stop(4);
           PMU::stop(5);
           PMU::stop(6);

           PMU::reset(0);
           PMU::reset(1);
           PMU::reset(2);
           PMU::reset(3);
           PMU::reset(4);
           PMU::reset(5);
           PMU::reset(6);

           PMU::write(0, 0);
```

```
            PMU::write(1, 0);
            PMU::write(2, 0);
            PMU::write(3, 0);
            PMU::write(4, 0);
            PMU::write(5, 0);
            PMU::write(6, 0);

            PMU::config(3, (PMU::Event)PMU::_channel_3);
            PMU::config(4, (PMU::Event)PMU::_channel_4);
            PMU::config(5, (PMU::Event)PMU::_channel_5);
            PMU::config(6, (PMU::Event)PMU::_channel_6);

            PMU::start(0);
            PMU::start(1);
            PMU::start(2);
            PMU::start(3);
            PMU::start(4);
            PMU::start(5);
            PMU::start(6);
         }
          MCEDF(int p = APERIODIC)
          : CEDF(p) {} // Aperiodic

          MCEDF(const Microsecond & d, const Microsecond & p = SAME, const Microsecond & c =
UNKNOWN, int cpu = ANY)
          : CEDF(d, p, c, cpu) {}
      };
}
...
template<typename T>
class Scheduling_Queue<T, Scheduling_Criteria::MGEDF>:
public Multihead_Scheduling_List<T> {};

template<typename T>
class Scheduling_Queue<T, Scheduling_Criteria::MPEDF>:
public Scheduling_Multilist<T> {};

template<typename T>
class Scheduling_Queue<T, Scheduling_Criteria::MCEDF>:
public Multihead_Scheduling_Multilist<T> {};
```

include/thread.h:
- adição de variáveis para o funcionamento do sistema de capturas;

```
// EPOS Thread Component Declarations

#ifndef __thread_h
#define __thread_h

#include <utility/queue.h>
#include <utility/handler.h>
#include <cpu.h>
```

```
#include <machine.h>
#include <system.h>
#include <scheduler.h>
#include <segment.h>
#include <architecture/ia32/monitoring_capture.h>
extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
...
public:
    volatile int _missed_deadlines; //it's not unsigned because of the calculating process
    int _times_p_count; //counts the number of executions to be used on deadline misses calc
    static Monitoring_Capture* _thread_monitor; //system monitor
    static volatile bool _end_capture; //stops capture when True
    static unsigned int _global_deadline_misses;
    static unsigned int _prev_global_deadline_misses; //used to calc the difference when using
heuristics

};
...
```

include/periodic_thread.h:
- calculates _times_p_count, ou seja, o número de vezes que já terminou uma execução.
- captura de um momento dentro do wait_next()

```
// EPOS Periodic Thread Component Declarations

// Periodic threads are achieved by programming an alarm handler to invoke
// p() on a control semaphore after each job (i.e. task activation). Base
// threads are created in BEGINNING state, so the scheduler won't dispatch
// them before the associate alarm and semaphore are created. The first job
// is dispatched by resume() (thus the _state = SUSPENDED statement)

#ifndef __periodic_thread_h
#define __periodic_thread_h

#include <utility/handler.h>
#include <thread.h>
#include <alarm.h>

__BEGIN_SYS

// Aperiodic Thread
typedef Thread Aperiodic_Thread;

// Periodic Thread
class Periodic_Thread: public Thread
{
...
```

```
template<typename ... Tn>
    Periodic_Thread(const Configuration & conf, int (* entry)(Tn ...), Tn ... an)
    : Thread(Thread::Configuration(SUSPENDED, (conf.criterion != NORMAL) ? conf.criterion :
Criterion(conf.period), conf.color, conf.task, conf.stack_size), entry, an ...),
      _semaphore(0), _handler(&_semaphore, this), _alarm(conf.period, &_handler, conf.times) {
        if (Criterion::monitoring) {
            _times_p_count = conf.times;
            _alarm_times = &_alarm;
        }
        if((conf.state == READY) || (conf.state == RUNNING)) {
            _state = SUSPENDED;
            resume();
        } else
            _state = conf.state;
    }

    const Microsecond & period() const { return _alarm.period(); }
    void period(const Microsecond & p) { _alarm.period(p); }

    static volatile bool wait_next() {

        Periodic_Thread * t = reinterpret_cast<Periodic_Thread *>(running());
        //calc ddl misses and capture
        if (Criterion::monitoring && !_end_capture) {
            unsigned int entry_temp = CPU::temperature();
            t->_missed_deadlines = t->_times_p_count - (t->_alarm_times->_times);

            unsigned long long pkg = CPU::pkg_energy_status();
            unsigned long long pp0 = CPU::pp0_energy_status();
            _thread_monitor->capture(entry_temp, PMU::read(0), PMU::read(1), PMU::read(2),
PMU::read(3), PMU::read(4), PMU::read(5),
                PMU::read(6), reinterpret_cast<volatile unsigned int>(t), t->priority(),
Machine::cpu_id(), t->_missed_deadlines, t->_global_deadline_misses, pkg, pp0);
        }
        //end capture code

        if(t->_alarm._times) {
            t->_semaphore.p();
            //db<Thread>(WRN)<<"times: "<<t->_alarm._times<<endl;
            t->_times_p_count--;
        }

        return t->_alarm._times;
    }
...
};
```

include/system/types.h:
- adicionados os escalonadores criados como um tipo.

```
// EPOS Internal Type Management System
```

```
typedef __SIZE_TYPE__ size_t;

#ifndef __types_h
#define __types_h
...
template<typename> class Scheduler;
namespace Scheduling_Criteria
{
    class Priority;
    class FCFS;
    class RR;
    class RM;
    class DM;
    class EDF;
    class GRR;
    class CPU_Affinity;
    class GEDF;
    class MGEDF; //added scheduler entry
    class PEDF;
    class MPEDF; //added scheduler entry
    class CEDF;
    class MCEDF; //added scheduler entry
    class PRM;
};
...
#endif
```

src/component/thread.cc:
- Captura de um momento dentro do dispatch;
- Mudança do código de finalização do sistema no método idle para imprimir os dados antes de desligar o computador;

```
// EPOS Thread Component Implementation

#include <machine.h>
#include <system.h>
#include <thread.h>
#include <alarm.h> // for FCFS

// This_Thread class attributes
__BEGIN_UTIL
bool This_Thread::_not_booting;
__END_UTIL

__BEGIN_SYS

// Class attributes
Monitoring_Capture* Thread::_thread_monitor;
unsigned int Thread::_global_deadline_misses;
unsigned int Thread::_prev_global_deadline_misses;
volatile unsigned int Thread::_thread_count;
volatile bool Thread::_end_capture;
```

```
...
void Thread::dispatch(Thread * prev, Thread * next, bool charge)
{

    if(charge) {
        if(Criterion::timed)
            _timer->reset();

        //Monitoring Capture
        if(Criterion::monitoring && !_end_capture) {
            unsigned int entry_temp = CPU::temperature();
            if (prev->priority() != IDLE && prev->priority() != MAIN) {
                prev->_missed_deadlines = prev->_times_p_count - (prev->_alarm_times->_times +
1);
            } else
                 prev->_missed_deadlines = 0;

            if (next->priority() != IDLE && next->priority() != MAIN) {
                next->_missed_deadlines = next->_times_p_count - (next->_alarm_times->_times +
1);
            } else
                next->_missed_deadlines = 0;

            if (prev->_missed_deadlines < 0)
                    prev->_missed_deadlines = 0;

            if (next->_missed_deadlines < 0)
                    next->_missed_deadlines = 0;

            unsigned long long ts = _thread_monitor->last_capture(Machine::cpu_id(), 7);
            unsigned long long ts_dif = _thread_monitor->time() - ts;
            if (ts == 0) {
                ts_dif = 5000;
            }
            unsigned int cpu = Machine::cpu_id();
            if (cpu % 2) {
                cpu--;
            }
            unsigned long long pkg = CPU::pkg_energy_status();
            unsigned long long pp0 = CPU::pp0_energy_status();
            //unsigned long long energy_unit = CPU::rapl_energy_unit(); // use only to know the
power unit value
            _thread_monitor->capture(entry_temp, PMU::read(0), PMU::read(1), PMU::read(2),
PMU::read(3), PMU::read(4), PMU::read(5),
            PMU::read(6), reinterpret_cast<volatile unsigned int>(prev), prev->priority(),
Machine::cpu_id(), prev->_missed_deadlines, prev->_global_deadline_misses, pkg, pp0);
            // capture next -> not necessary, it's possible to infer most of the data
            // _thread_monitor->capture(entry_temp, PMU::read(0), PMU::read(1), PMU::read(2),
PMU::read(3), PMU::read(4), PMU::read(5),
            // PMU::read(6), reinterpret_cast<volatile unsigned int>(next), next->priority(),
Machine::cpu_id(), next->_missed_deadlines, next->_global_deadline_misses);
        }
    }
```

```
    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;

        db<Thread>(TRC) << "Thread::dispatch(prev=" << prev << ",next=" << next << ")" << endl;
        db<Thread>(INF) << "prev={" << prev << ",ctx=" << *prev->_context << "}" << endl;
        db<Thread>(INF) << "next={" << next << ",ctx=" << *next->_context << "}" << endl;

        if(smp)
            _lock.release();

        if(multitask && (next->_task != prev->_task))
            next->_task->activate();

        CPU::switch_context(&prev->_context, next->_context);
    } else
        if(smp)
            _lock.release();

    // TODO: could this be moved to right after the switch_context?
    CPU::int_enable();
}

int Thread::idle()
{
    while(_thread_count > Machine::n_cpus()) { // someone else besides idles
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(CPU=" << Machine::cpu_id() << ",this=" <<
running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
        if(_scheduler.schedulables() > 0)// A thread might have been woken up by another CPU
            yield();
    }

    CPU::int_disable();
    if(Machine::cpu_id() == 0) {
        if (Criterion::monitoring) {
            db<Thread>(WRN) << "temp: "<< CPU::temperature() <<endl;
            _thread_monitor->datas();
        }
        db<Thread>(WRN) << "The last thread has exited!" << endl;

        if(reboot) {
            db<Thread>(WRN) << "Rebooting the machine ..." << endl;
            Machine::reboot();
        } else
            db<Thread>(WRN) << "Halting the machine ..." << endl;
    }
    CPU::halt();
```

```
    return 0;
}
...
```

src/component/thread_init.cc:
- aloca memória e inicializa o sistema de capturas

```
// EPOS Thread Component Initialization

#include <system.h>
#include <thread.h>
#include <alarm.h>
#include <clock.h>
#include <segment.h>
#include <mmu.h>

__BEGIN_SYS

void Thread::init()
{
    // The installation of the scheduler timer handler must precede the
    // creation of threads, since the constructor can induce a reschedule
    // and this in turn can call timer->reset()
    // Letting reschedule() happen during thread creation is harmless, since
    // MAIN is created first and dispatch won't replace it nor by itself
    // neither by IDLE (which has a lower priority)
    if(Criterion::timed && (Machine::cpu_id() == 0))
        _timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);

    // Install an interrupt handler to receive forced reschedules
    if(smp) {
        if(Machine::cpu_id() == 0)
            db<Thread>(WRN)<<"DATE:: "<<RTC::date()<<endl;
        if(Machine::cpu_id() == 0)
            IC::int_vector(IC::INT_RESCHEDULER, rescheduler);
        IC::enable(IC::INT_RESCHEDULER);
    }
    // Checks if monitoring execution is enable
    if (Criterion::monitoring) {
        // Only one CPU needs to initialize the Monitoring Capture System
        if (Machine::cpu_id() == 0) {
            _end_capture = true;
            // Allocating a memory region (117MB) for store the capture
            // For this to work, we increased heap size in 120MB. --> This can handle 1024000
captures
            _thread_monitor = new Monitoring_Capture(128000, new (SYSTEM) Moment[1024000]);
        }
    }

    Criterion::init();
}
```

```
__END_SYS
```

- src/component/semaphore.cc:
  - register global_deadline_misses

```
// EPOS Semaphore Component Implementation

#include <semaphore.h>

__BEGIN_SYS

Semaphore::Semaphore(int v): _value(v)
{
    db<Synchronizer>(TRC) << "Semaphore(value=" << _value << ") => " << this << endl;
}


Semaphore::~Semaphore()
{
    db<Synchronizer>(TRC) << "~Semaphore(this=" << this << ")" << endl;
}


void Semaphore::p()
{
    db<Synchronizer>(TRC) << "Semaphore::p(this=" << this << ",value=" << _value << ")" <<
endl;

    begin_atomic();
    if(fdec(_value) < 1)
        sleep(); // implicit end_atomic()
    else
        end_atomic();
}


void Semaphore::v()
{
    db<Synchronizer>(TRC) << "Semaphore::v(this=" << this << ",value=" << _value << ")" <<
endl;

    begin_atomic();
    if(finc(_value) < 0)
        wakeup();  // implicit end_atomic()
    else {
        if (Thread::Criterion::monitoring)
            if(_value > 1) // if a daadline miss has occour in this alarm (one alarm per
thread)
                Thread::self()->_global_deadline_misses++; // increase the count of global
deadline misses (deadline misses in all threads)
        end_atomic();
    }
```

```
}

__END_SYS
```

src/architecture/ia32/pmu.cc:
- Configuração dos eventos númerados no vetor:
 - CPU::Reg32 Intel_Sandy_Bridge_PMU::_events[PMU_Common::EVENTS]

```
// Adds the address of the events in Intel Sandy Bredige Processors to the enum Events
const CPU::Reg32 Intel_Sandy_Bridge_PMU::_events[PMU_Common::EVENTS] = {
                        /* CLOCK              */ UNHALTED_CORE_CYCLES,
                        /* DVS_CLOCK          */ UNHALTED_REFERENCE_CYCLES,
                        /* INSTRUCTIONS       */ INSTRUCTIONS_RETIRED,
                        /* BRANCHES           */ BRANCH_INSTRUCTIONS_RETIRED,
                        /* BRANCH_MISSES      */ BRANCH_MISSES_RETIRED,
                        /* L1_HIT             */ 0,
                        /* L2_HIT             */ 0,
                        /* L3_HIT             */ LLC_REFERENCES,
                        /* L1_MISS            */ 0,
                        /* L2_MISS            */ 0,
                        /* L3_MISS            */ LLC_MISSES,
                        /*LD_BLOCKS_DATA_UNKNOWN_C*/
Intel_Sandy_Bridge_PMU::LD_BLOCKS_DATA_UNKNOWN,
                                              /*LD_BLOCKS_STORE_FORWARD_C*/
Intel_Sandy_Bridge_PMU::LD_BLOCKS_STORE_FORWARD,
                                              /*LD_BLOCKS_NO_SR_C*/
Intel_Sandy_Bridge_PMU::LD_BLOCKS_NO_SR,
                                              /*LD_BLOCKS_ALL_BLOCK_C*/
Intel_Sandy_Bridge_PMU::LD_BLOCKS_ALL_BLOCK,
                                              /*MISALIGN_MEM_REF_LOADS_C*/
Intel_Sandy_Bridge_PMU::MISALIGN_MEM_REF_LOADS,
                                              /*MISALIGN_MEM_REF_STORES_C*/
Intel_Sandy_Bridge_PMU::MISALIGN_MEM_REF_STORES,
                                              /*LD_BLOCKS_PARTIAL_ADDRESS_ALIAS_C*/
Intel_Sandy_Bridge_PMU::LD_BLOCKS_PARTIAL_ADDRESS_ALIAS,
                                              /*LD_BLOCKS_PARTIAL_ALL_STA_BLCOK_C*/
Intel_Sandy_Bridge_PMU::LD_BLOCKS_PARTIAL_ALL_STA_BLCOK,
                                              /*DTLB_LOAD_MISSES_MISS_CAUSES_A_WALK_C*/
Intel_Sandy_Bridge_PMU::DTLB_LOAD_MISSES_MISS_CAUSES_A_WALK,
                                              /*DTLB_LOAD_MISSES_MISS_WALK_COMPLETED_C*/
Intel_Sandy_Bridge_PMU::DTLB_LOAD_MISSES_MISS_WALK_COMPLETED,
                                              /*DTLB_LOAD_MISSES_MISS_WALK_DURATION_C*/
Intel_Sandy_Bridge_PMU::DTLB_LOAD_MISSES_MISS_WALK_DURATION,
                                              /*DTLB_LOAD_MISSES_MISS_STLB_HIT_C*/
Intel_Sandy_Bridge_PMU::DTLB_LOAD_MISSES_MISS_STLB_HIT,
                                              /*INT_MISC_RECOVERY_CYCLES_C*/
Intel_Sandy_Bridge_PMU::INT_MISC_RECOVERY_CYCLES,
                                              /*INT_MISC_RAT_STALL_CYCLES_C*/
Intel_Sandy_Bridge_PMU::INT_MISC_RAT_STALL_CYCLES,
                                              /*UOPS_ISSUED_ANY_C*/
Intel_Sandy_Bridge_PMU::UOPS_ISSUED_ANY,
                                              /*FP_COMP_OPS_EXE_X87_C*/
```

```
Intel_Sandy_Bridge_PMU::FP_COMP_OPS_EXE_X87,
                                        /*FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE_C*/
Intel_Sandy_Bridge_PMU::FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE,
                                        /*FP_COMP_OPS_EXE_SSE_FP_SCALAR_SINGLE_C*/
Intel_Sandy_Bridge_PMU::FP_COMP_OPS_EXE_SSE_FP_SCALAR_SINGLE,
                                        /*FP_COMP_OPS_EXE_SSE_PACKED_SINGLE_C*/
Intel_Sandy_Bridge_PMU::FP_COMP_OPS_EXE_SSE_PACKED_SINGLE,
                                        /*FP_COMP_OPS_EXE_SSE_SCALAR_DOUBLE_C*/
Intel_Sandy_Bridge_PMU::FP_COMP_OPS_EXE_SSE_SCALAR_DOUBLE,
                                        /*SIMD_FP_256_PACKED_SINGLE_C*/
Intel_Sandy_Bridge_PMU::SIMD_FP_256_PACKED_SINGLE,
                                        /*SIMD_FP_256_PACKED_DOUBLE_C*/
Intel_Sandy_Bridge_PMU::SIMD_FP_256_PACKED_DOUBLE,
                                        /*ARITH_FPU_DIV_ACTIVE_C*/
Intel_Sandy_Bridge_PMU::ARITH_FPU_DIV_ACTIVE,
                                        /*INSTS_WRITTEN_TO_IQ_INSTS_C*/
Intel_Sandy_Bridge_PMU::INSTS_WRITTEN_TO_IQ_INSTS,
                                        /*L2_RQSTS_DEMAND_DATA_RD_HIT_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_DEMAND_DATA_RD_HIT,
                                        /*L2_RQSTS_ALL_DEMAND_DATA_RD_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_ALL_DEMAND_DATA_RD,
                                        /*L2_RQSTS_RFO_HITS_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_RFO_HITS,
                                        /*L2_RQSTS_RFO_MISS_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_RFO_MISS,
                                        /*L2_RQSTS_ALL_RFO_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_ALL_RFO,
                                        /*L2_RQSTS_CODE_RD_HIT_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_CODE_RD_HIT,
                                        /*L2_RQSTS_CODE_RD_MISS_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_CODE_RD_MISS,
                                        /*L2_RQSTS_ALL_CODE_RD_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_ALL_CODE_RD,
                                        /*L2_RQSTS_PF_HIT_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_PF_HIT,
                                        /*L2_RQSTS_PF_MISS_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_PF_MISS,
                                        /*L2_RQSTS_ALL_PF_C*/
Intel_Sandy_Bridge_PMU::L2_RQSTS_ALL_PF,
                                        /*L2_STORE_LOCK_RQSTS_MISS_C*/
Intel_Sandy_Bridge_PMU::L2_STORE_LOCK_RQSTS_MISS,
                                        /*L2_STORE_LOCK_RQSTS_HIT_E_C*/
Intel_Sandy_Bridge_PMU::L2_STORE_LOCK_RQSTS_HIT_E,
                                        /*L2_STORE_LOCK_RQSTS_HIT_M_C*/
Intel_Sandy_Bridge_PMU::L2_STORE_LOCK_RQSTS_HIT_M,
                                        /*L2_STORE_LOCK_RQSTS_ALL_C*/
Intel_Sandy_Bridge_PMU::L2_STORE_LOCK_RQSTS_ALL,
                                        /*L2_L1D_WB_RQSTS_HIT_E_C*/
Intel_Sandy_Bridge_PMU::L2_L1D_WB_RQSTS_HIT_E,
                                        /*L2_L1D_WB_RQSTS_HIT_M_C*/
Intel_Sandy_Bridge_PMU::L2_L1D_WB_RQSTS_HIT_M,
                                        /*LONGEST_LAT_CACHE_REFERENCE_C*/
Intel_Sandy_Bridge_PMU::LONGEST_LAT_CACHE_REFERENCE,
```

```
                                        /*LONGEST_LAT_CACHE_MISS_C*/
Intel_Sandy_Bridge_PMU::LONGEST_LAT_CACHE_MISS,
                                        /*CPU_CLK_UNHALTED_THREAD_P_C*/
Intel_Sandy_Bridge_PMU::CPU_CLK_UNHALTED_THREAD_P,
                                        /*CPU_CLK_THREAD_UNHALTED_REF_XCLK_C*/
Intel_Sandy_Bridge_PMU::CPU_CLK_THREAD_UNHALTED_REF_XCLK,
                                        /*L1D_PEND_MISS_PENDING_C*/
Intel_Sandy_Bridge_PMU::L1D_PEND_MISS_PENDING,
                                        /*DTLB_STORE_MISSES_MISS_CAUSES_A_WALK_C*/
Intel_Sandy_Bridge_PMU::DTLB_STORE_MISSES_MISS_CAUSES_A_WALK,
                                        /*DTLB_STORE_MISSES_WALK_COMPLETED_C*/
Intel_Sandy_Bridge_PMU::DTLB_STORE_MISSES_WALK_COMPLETED,
                                        /*DTLB_STORE_MISSES_WALK_DURATION_C*/
Intel_Sandy_Bridge_PMU::DTLB_STORE_MISSES_WALK_DURATION,
                                        /*DTLB_STORE_MISSES_TLB_HIT_C*/
Intel_Sandy_Bridge_PMU::DTLB_STORE_MISSES_TLB_HIT,
                                        /*LOAD_HIT_PRE_SW_PF_C*/
Intel_Sandy_Bridge_PMU::LOAD_HIT_PRE_SW_PF,
                                        /*LOAD_HIT_PREHW_PF_C*/
Intel_Sandy_Bridge_PMU::LOAD_HIT_PREHW_PF,
                                        /*HW_PRE_REQ_DL1_MISS_C*/
Intel_Sandy_Bridge_PMU::HW_PRE_REQ_DL1_MISS,
                                        /*L1D_REPLACEMENT_C*/
Intel_Sandy_Bridge_PMU::L1D_REPLACEMENT,
                                        /*L1D_ALLOCATED_IN_M_C*/
Intel_Sandy_Bridge_PMU::L1D_ALLOCATED_IN_M,
                                        /*L1D_EVICTION_C*/
Intel_Sandy_Bridge_PMU::L1D_EVICTION,
                                        /*L1D_ALL_M_REPLACEMENT_C*/
Intel_Sandy_Bridge_PMU::L1D_ALL_M_REPLACEMENT,
                                        /*PARTIAL_RAT_STALLS_FLAGS_MERGE_UOP_C*/
Intel_Sandy_Bridge_PMU::PARTIAL_RAT_STALLS_FLAGS_MERGE_UOP,
                                        /*PARTIAL_RAT_STALLS_SLOW_LEA_WINDOW_C*/
Intel_Sandy_Bridge_PMU::PARTIAL_RAT_STALLS_SLOW_LEA_WINDOW,
                                        /*PARTIAL_RAT_STALLS_MUL_SINGLE_UOP_C*/
Intel_Sandy_Bridge_PMU::PARTIAL_RAT_STALLS_MUL_SINGLE_UOP,
                                        /*RESOURCE_STALLS2_ALL_FL_EMPTY_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS2_ALL_FL_EMPTY,
                                        /*RESOURCE_STALLS2_ALL_PRF_CONTROL_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS2_ALL_PRF_CONTROL,
                                        /*RESOURCE_STALLS2_BOB_FULL_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS2_BOB_FULL,
                                        /*RESOURCE_STALLS2_OOO_RSRC_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS2_OOO_RSRC,
                                        /*CPL_CYCLES_RING0_C*/
Intel_Sandy_Bridge_PMU::CPL_CYCLES_RING0,
                                        /*CPL_CYCLES_RING123_C*/
Intel_Sandy_Bridge_PMU::CPL_CYCLES_RING123,
                                        /*RS_EVENTS_EMPTY_CYCLES_C*/
Intel_Sandy_Bridge_PMU::RS_EVENTS_EMPTY_CYCLES,
                                        /*OFFCORE_REQUESTS_OUTSTANDING_DEMAND_DATA_RD_C*/
Intel_Sandy_Bridge_PMU::OFFCORE_REQUESTS_OUTSTANDING_DEMAND_DATA_RD,
                                        /*OFFCORE_REQUESTS_OUTSTANDING_DEMAND_RFO_C*/
```

```
Intel_Sandy_Bridge_PMU::OFFCORE_REQUESTS_OUTSTANDING_DEMAND_RFO,
                                        /*OFFCORE_REQUESTS_OUTSTANDING_ALL_DATA_RD_C*/
Intel_Sandy_Bridge_PMU::OFFCORE_REQUESTS_OUTSTANDING_ALL_DATA_RD,
                                        /*LOCK_CYCLES_SPLIT_LOCK_UC_LOCK_DURATION_C*/
Intel_Sandy_Bridge_PMU::LOCK_CYCLES_SPLIT_LOCK_UC_LOCK_DURATION,
                                        /*LOCK_CYCLES_CACHE_LOCK_DURATION_C*/
Intel_Sandy_Bridge_PMU::LOCK_CYCLES_CACHE_LOCK_DURATION,
                                        /*IDQ_EMPTY_C*/ Intel_Sandy_Bridge_PMU::IDQ_EMPTY,
                                        /*IDQ_MITE_UOPS_C*/
Intel_Sandy_Bridge_PMU::IDQ_MITE_UOPS,
                                        /*IDQ_DSB_UOPS_C*/
Intel_Sandy_Bridge_PMU::IDQ_DSB_UOPS,
                                        /*IDQ_MS_DSB_UOPS_C*/
Intel_Sandy_Bridge_PMU::IDQ_MS_DSB_UOPS,
                                        /*IDQ_MS_MITE_UOPS_C*/
Intel_Sandy_Bridge_PMU::IDQ_MS_MITE_UOPS,
                                        /*IDQ_MS_UOPS_C*/
Intel_Sandy_Bridge_PMU::IDQ_MS_UOPS,
                                        /*ICACHE_MISSES_C*/
Intel_Sandy_Bridge_PMU::ICACHE_MISSES,
                                        /*ITLB_MISSES_MISS_CAUSES_A_WALK_C*/
Intel_Sandy_Bridge_PMU::ITLB_MISSES_MISS_CAUSES_A_WALK,
                                        /*ITLB_MISSES_WALK_COMPLETED_C*/
Intel_Sandy_Bridge_PMU::ITLB_MISSES_WALK_COMPLETED,
                                        /*ITLB_MISSES_WALK_DURATION_C*/
Intel_Sandy_Bridge_PMU::ITLB_MISSES_WALK_DURATION,
                                        /*ITLB_MISSES_STLB_HIT_C*/
Intel_Sandy_Bridge_PMU::ITLB_MISSES_STLB_HIT,
                                        /*ILD_STALL_LCP_C*/
Intel_Sandy_Bridge_PMU::ILD_STALL_LCP,
                                        /*ILD_STALL_IQ_FULL_C*/
Intel_Sandy_Bridge_PMU::ILD_STALL_IQ_FULL,
                                        /*BR_INST_EXEC_COND_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_COND,
                                        /*BR_INST_EXEC_DIRECT_JMP_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_DIRECT_JMP,
                                        /*BR_INST_EXEC_INDIRECT_JMP_NON_CALL_RET_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_INDIRECT_JMP_NON_CALL_RET,
                                        /*BR_INST_EXEC_RETURN_NEAR_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_RETURN_NEAR,
                                        /*BR_INST_EXEC_DIRECT_NEAR_CALL_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_DIRECT_NEAR_CALL,
                                        /*BR_INST_EXEC_INDIRECT_NEAR_CALL_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_INDIRECT_NEAR_CALL,
                                        /*BR_INST_EXEC_NON_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_NON_TAKEN,
                                        /*BR_INST_EXEC_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_TAKEN,
                                        /*BR_INST_EXEC_ALL_BRANCHES_C*/
Intel_Sandy_Bridge_PMU::BR_INST_EXEC_ALL_BRANCHES,
                                        /*BR_MISP_EXEC_COND_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_COND,
                                        /*BR_MISP_EXEC_INDIRECT_JMP_NON_CALL_RET_C*/
```

```
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_INDIRECT_JMP_NON_CALL_RET,
                                        /*BR_MISP_EXEC_RETURN_NEAR_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_RETURN_NEAR,
                                        /*BR_MISP_EXEC_DIRECT_NEAR_CALL_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_DIRECT_NEAR_CALL,
                                        /*BR_MISP_EXEC_INDIRECT_NEAR_CALL_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_INDIRECT_NEAR_CALL,
                                        /*BR_MISP_EXEC_NON_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_NON_TAKEN,
                                        /*BR_MISP_EXEC_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_TAKEN,
                                        /*BR_MISP_EXEC_ALL_BRANCHES_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_EXEC_ALL_BRANCHES,
                                        /*IDQ_UOPS_NOT_DELIVERED_CORE_C*/
Intel_Sandy_Bridge_PMU::IDQ_UOPS_NOT_DELIVERED_CORE,
                                        /*UOPS_DISPATCHED_PORT_PORT_0_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_0,
                                        /*UOPS_DISPATCHED_PORT_PORT_1_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_1,
                                        /*UOPS_DISPATCHED_PORT_PORT_2_LD_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_2_LD,
                                        /*UOPS_DISPATCHED_PORT_PORT_2_STA_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_2_STA,
                                        /*UOPS_DISPATCHED_PORT_PORT_2_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_2,
                                        /*UOPS_DISPATCHED_PORT_PORT_3_LD_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_3_LD,
                                        /*UOPS_DISPATCHED_PORT_PORT_3_STA_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_3_STA,
                                        /*UOPS_DISPATCHED_PORT_PORT_3_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_3,
                                        /*UOPS_DISPATCHED_PORT_PORT_4_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_4,
                                        /*UOPS_DISPATCHED_PORT_PORT_5_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_PORT_PORT_5,
                                        /*RESOURCE_STALLS_ANY_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_ANY,
                                        /*RESOURCE_STALLS_LB_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_LB,
                                        /*RESOURCE_STALLS_RS_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_RS,
                                        /*RESOURCE_STALLS_SB_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_SB,
                                        /*RESOURCE_STALLS_ROB_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_ROB,
                                        /*RESOURCE_STALLS_FCSW_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_FCSW,
                                        /*RESOURCE_STALLS_MXCSR_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_MXCSR,
                                        /*RESOURCE_STALLS_OTHER_C*/
Intel_Sandy_Bridge_PMU::RESOURCE_STALLS_OTHER,
                                        /*DSB2MITE_SWITCHES_COUNT_C*/
Intel_Sandy_Bridge_PMU::DSB2MITE_SWITCHES_COUNT,
```

```
                                        /*DSB2MITE_SWITCHES_PENALTY_CYCLES_C*/
Intel_Sandy_Bridge_PMU::DSB2MITE_SWITCHES_PENALTY_CYCLES,
                                        /*DSB_FILL_OTHER_CANCEL_C*/
Intel_Sandy_Bridge_PMU::DSB_FILL_OTHER_CANCEL,
                                        /*DSB_FILL_EXCEED_DSB_LINES_C*/
Intel_Sandy_Bridge_PMU::DSB_FILL_EXCEED_DSB_LINES,
                                        /*DSB_FILL_ALL_CANCEL_C*/
Intel_Sandy_Bridge_PMU::DSB_FILL_ALL_CANCEL,
                                        /*ITLB_ITLB_FLUSH_C*/
Intel_Sandy_Bridge_PMU::ITLB_ITLB_FLUSH,
                                        /*OFFCORE_REQUESTS_DEMAND_DATA_RD_C*/
Intel_Sandy_Bridge_PMU::OFFCORE_REQUESTS_DEMAND_DATA_RD,
                                        /*OFFCORE_REQUESTS_DEMAND_RFO_C*/
Intel_Sandy_Bridge_PMU::OFFCORE_REQUESTS_DEMAND_RFO,
                                        /*OFFCORE_REQUESTS_ALL_DATA_RD_C*/
Intel_Sandy_Bridge_PMU::OFFCORE_REQUESTS_ALL_DATA_RD,
                                        /*UOPS_DISPATCHED_THREAD_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_THREAD,
                                        /*UOPS_DISPATCHED_CORE_C*/
Intel_Sandy_Bridge_PMU::UOPS_DISPATCHED_CORE,
                                        /*OFFCORE_REQUESTS_BUFFER_SQ_FULL_C*/
Intel_Sandy_Bridge_PMU::OFFCORE_REQUESTS_BUFFER_SQ_FULL,
                                        /*AGU_BYPASS_CANCEL_COUNT_C*/
Intel_Sandy_Bridge_PMU::AGU_BYPASS_CANCEL_COUNT,
                                        /*OFF_CORE_RESPONSE_0_C*/
Intel_Sandy_Bridge_PMU::OFF_CORE_RESPONSE_0,
                                        /*OFF_CORE_RESPONSE_1_C*/
Intel_Sandy_Bridge_PMU::OFF_CORE_RESPONSE_1,
                                        /*TLB_FLUSH_DTLB_THREAD_C*/
Intel_Sandy_Bridge_PMU::TLB_FLUSH_DTLB_THREAD,
                                        /*TLB_FLUSH_STLB_ANY_C*/
Intel_Sandy_Bridge_PMU::TLB_FLUSH_STLB_ANY,
                                        /*L1D_BLOCKS_BANK_CONFLICT_CYCLES_C*/
Intel_Sandy_Bridge_PMU::L1D_BLOCKS_BANK_CONFLICT_CYCLES,
                                        /*INST_RETIRED_ANY_P_C*/
Intel_Sandy_Bridge_PMU::INST_RETIRED_ANY_P,
                                        /*INST_RETIRED_PREC_DIST_C*/
Intel_Sandy_Bridge_PMU::INST_RETIRED_PREC_DIST,
                                        /*OTHER_ASSISTS_ITLB_MISS_RETIRED_C*/
Intel_Sandy_Bridge_PMU::OTHER_ASSISTS_ITLB_MISS_RETIRED,
                                        /*OTHER_ASSISTS_AVX_STORE_C*/
Intel_Sandy_Bridge_PMU::OTHER_ASSISTS_AVX_STORE,
                                        /*OTHER_ASSISTS_AVX_TO_SSE_C*/
Intel_Sandy_Bridge_PMU::OTHER_ASSISTS_AVX_TO_SSE,
                                        /*OTHER_ASSISTS_SSE_TO_AVX_C*/
Intel_Sandy_Bridge_PMU::OTHER_ASSISTS_SSE_TO_AVX,
                                        /*UOPS_RETIRED_ALL_C*/
Intel_Sandy_Bridge_PMU::UOPS_RETIRED_ALL,
                                        /*UOPS_RETIRED_RETIRE_SLOTS_C*/
Intel_Sandy_Bridge_PMU::UOPS_RETIRED_RETIRE_SLOTS,
                                        /*MACHINE_CLEARS_MEMORY_ORDERING_C*/
Intel_Sandy_Bridge_PMU::MACHINE_CLEARS_MEMORY_ORDERING,
                                        /*MACHINE_CLEARS_SMC_C*/
```

```
Intel_Sandy_Bridge_PMU::MACHINE_CLEARS_SMC,
                                        /*MACHINE_CLEARS_MASKMOV_C*/
Intel_Sandy_Bridge_PMU::MACHINE_CLEARS_MASKMOV,
                                        /*BR_INST_RETIRED_ALL_BRANCHES_ARCH_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_ALL_BRANCHES_ARCH,
                                        /*BR_INST_RETIRED_CONDITIONAL_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_CONDITIONAL,
                                        /*BR_INST_RETIRED_NEAR_CALL_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_NEAR_CALL,
                                        /*BR_INST_RETIRED_ALL_BRANCHES_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_ALL_BRANCHES,
                                        /*BR_INST_RETIRED_NEAR_RETURN_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_NEAR_RETURN,
                                        /*BR_INST_RETIRED_NOT_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_NOT_TAKEN,
                                        /*BR_INST_RETIRED_NEAR_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_NEAR_TAKEN,
                                        /*BR_INST_RETIRED_FAR_BRANCH_C*/
Intel_Sandy_Bridge_PMU::BR_INST_RETIRED_FAR_BRANCH,
                                        /*BR_MISP_RETIRED_ALL_BRANCHES_ARCH_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_RETIRED_ALL_BRANCHES_ARCH,
                                        /*BR_MISP_RETIRED_CONDITIONAL_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_RETIRED_CONDITIONAL,
                                        /*BR_MISP_RETIRED_NEAR_CALL_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_RETIRED_NEAR_CALL,
                                        /*BR_MISP_RETIRED_ALL_BRANCHES_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_RETIRED_ALL_BRANCHES,
                                        /*BR_MISP_RETIRED_NOT_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_RETIRED_NOT_TAKEN,
                                        /*BR_MISP_RETIRED_TAKEN_C*/
Intel_Sandy_Bridge_PMU::BR_MISP_RETIRED_TAKEN,
                                        /*FP_ASSIST_X87_OUTPUT_C*/
Intel_Sandy_Bridge_PMU::FP_ASSIST_X87_OUTPUT,
                                        /*FP_ASSIST_X87_INPUT_C*/
Intel_Sandy_Bridge_PMU::FP_ASSIST_X87_INPUT,
                                        /*FP_ASSIST_SIMD_OUTPUT_C*/
Intel_Sandy_Bridge_PMU::FP_ASSIST_SIMD_OUTPUT,
                                        /*FP_ASSIST_SIMD_INPUT_C*/
Intel_Sandy_Bridge_PMU::FP_ASSIST_SIMD_INPUT,
                                        /*FP_ASSIST_ANY_C*/
Intel_Sandy_Bridge_PMU::FP_ASSIST_ANY,
                                        /*ROB_MISC_EVENTS_LBR_INSERTS_C*/
Intel_Sandy_Bridge_PMU::ROB_MISC_EVENTS_LBR_INSERTS,
                                        /*MEM_TRANS_RETIRED_LOAD_LATENCY_C*/
Intel_Sandy_Bridge_PMU::MEM_TRANS_RETIRED_LOAD_LATENCY,
                                        /*MEM_TRANS_RETIRED_PRECISE_STORE_C*/
Intel_Sandy_Bridge_PMU::MEM_TRANS_RETIRED_PRECISE_STORE,
                                        /*MEM_UOP_RETIRED_LOADS_C*/
Intel_Sandy_Bridge_PMU::MEM_UOP_RETIRED_LOADS,
                                        /*MEM_UOP_RETIRED_STORES_C*/
Intel_Sandy_Bridge_PMU::MEM_UOP_RETIRED_STORES,
                                        /*MEM_UOP_RETIRED_STLB_MISS_C*/
Intel_Sandy_Bridge_PMU::MEM_UOP_RETIRED_STLB_MISS,
```

```
                                        /*MEM_UOP_RETIRED_LOCK_C*/
Intel_Sandy_Bridge_PMU::MEM_UOP_RETIRED_LOCK,
                                        /*MEM_UOP_RETIRED_SPLIT_C*/
Intel_Sandy_Bridge_PMU::MEM_UOP_RETIRED_SPLIT,
                                        /*MEM_UOP_RETIRED_ALL_C*/
Intel_Sandy_Bridge_PMU::MEM_UOP_RETIRED_ALL,
                                        /*MEM_UOPS_RETIRED_ALL_LOADS_C*/
Intel_Sandy_Bridge_PMU::MEM_UOPS_RETIRED_ALL_LOADS,
                                        /*MEM_LOAD_UOPS_RETIRED_L1_HIT_C*/
Intel_Sandy_Bridge_PMU::MEM_LOAD_UOPS_RETIRED_L1_HIT,
                                        /*MEM_LOAD_UOPS_RETIRED_L2_HIT_C*/
Intel_Sandy_Bridge_PMU::MEM_LOAD_UOPS_RETIRED_L2_HIT,
                                        /*MEM_LOAD_UOPS_RETIRED_L3_HIT_C*/
Intel_Sandy_Bridge_PMU::MEM_LOAD_UOPS_RETIRED_L3_HIT,
                                        /*MEM_LOAD_UOPS_RETIRED_HIT_LFB_C*/
Intel_Sandy_Bridge_PMU::MEM_LOAD_UOPS_RETIRED_HIT_LFB,
                                        /*XSNP_MISS_C*/ Intel_Sandy_Bridge_PMU::XSNP_MISS,
                                        /*XSNP_HIT_C*/ Intel_Sandy_Bridge_PMU::XSNP_HIT,
                                        /*XSNP_HITM_C*/ Intel_Sandy_Bridge_PMU::XSNP_HITM,
                                        /*XSNP_NONE_C*/ Intel_Sandy_Bridge_PMU::XSNP_NONE,
                                        /*MEM_LOAD_UOPS_MISC_RETIRED_LLC_MISS_C*/
Intel_Sandy_Bridge_PMU::MEM_LOAD_UOPS_MISC_RETIRED_LLC_MISS,
                                        /*L2_TRANS_DEMAND_DATA_RD_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_DEMAND_DATA_RD,
                                        /*L2_TRANS_RFO_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_RFO,
                                        /*L2_TRANS_CODE_RD_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_CODE_RD,
                                        /*L2_TRANS_ALL_PF_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_ALL_PF,
                                        /*L2_TRANS_L1D_WB_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_L1D_WB,
                                        /*L2_TRANS_L2_FILL_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_L2_FILL,
                                        /*L2_TRANS_L2_WB_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_L2_WB,
                                        /*L2_TRANS_ALL_REQ_UESTS_C*/
Intel_Sandy_Bridge_PMU::L2_TRANS_ALL_REQ_UESTS,
                                        /*L2_LINES_IN_I_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_IN_I,
                                        /*L2_LINES_IN_S_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_IN_S,
                                        /*L2_LINES_IN_E_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_IN_E,
                                        /*L2_LINES_IN_ALL_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_IN_ALL,
                                        /*L2_LINES_OUT_DEMAND_CLEAN_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_OUT_DEMAND_CLEAN,
                                        /*L2_LINES_OUT_DEMAND_DIRTY_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_OUT_DEMAND_DIRTY,
                                        /*L2_LINES_OUT_DEMAND_PF_CLEAN_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_OUT_DEMAND_PF_CLEAN,
                                        /*L2_LINES_OUT_DEMAND_PF_DIRTY_C*/
```

```
Intel_Sandy_Bridge_PMU::L2_LINES_OUT_DEMAND_PF_DIRTY,
                                        /*L2_LINES_OUT_DEMAND_DIRTY_ALL_C*/
Intel_Sandy_Bridge_PMU::L2_LINES_OUT_DEMAND_DIRTY_ALL,
                                        /*SQ_MISC_SPLIT_LOCK_C*/
Intel_Sandy_Bridge_PMU::SQ_MISC_SPLIT_LOCK
};
```

A2.2 CÓDIGO DA HEURÍSTICA DESENVOLVIDA

O desenvolvimento deste código está descrito no Tópico “5. Implementação da Heurística”, onde os detalhes sobre o processo e os valores finais utilizados são relatados e explicados. Além disso, os pontos de ação do código se dão em momentos de ocorrência de um *dispatch(),* durante uma troca de contexto, e em momentos de fim da execução em um período de uma thread periódica, ou seja, *o wait_next()*.

O código desenvolvido pode ser encontrado através do seguinte link:

- https://github.com/JoseHoffmann/Heuristic_TCC.

Junto do mesmo, se encontra um arquivo README que descreve as alterações realizadas.

Seguem agora os códigos alterados:

```
include/scheduler.h
- Criada a variável "heurístic", que representa se a execução vai fazer uso da heurística.
```

```
// EPOS Scheduler Component Declarations

#ifndef __scheduler_h
#define __scheduler_h

#include <utility/list.h>
#include <cpu.h>
#include <machine.h>
#include <pmu.h>

__BEGIN_SYS

// All scheduling criteria, or disciplines, must define operator int() with
// the semantics of returning the desired order of a given object within the
// scheduling list
namespace Scheduling_Criteria
{
```

```
    // Priority (static and dynamic)
    class Priority
    {
        friend class _SYS::RT_Thread;

    public:
        enum {
            MAIN   = 0,
            HIGH   = 1,
            NORMAL = (unsigned(1) << (sizeof(int) * 8 - 1)) - 3,
            LOW    = (unsigned(1) << (sizeof(int) * 8 - 1)) - 2,
            IDLE   = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1
        };

        static const bool timed = false;
        static const bool dynamic = false;
        static const bool preemptive = true;
        static const bool monitoring = false;
        static const bool heuristic = true;
    public:
        Priority(int p = NORMAL): _priority(p) {}

        operator const volatile int() const volatile { return _priority; }
        static void init() {}
        void update() {}
        unsigned int queue() const { return 0; }

    protected:
        volatile int _priority;
    };
...
```

include/thread,h
- Criado o vetor "_clock_factor", utilizado pela heurística para controlar o duty cycle aplicado na modulação de clock.
- Criado o vetor "_slowdown", utilizada pela heurística para controlar a variação do fator quando o mesmo é incrementado.
- Criados os vetores "_bef_channel6" e "_bef_channel5" utilizados pela heurística para armazenar o crescimento anterior do canal a fim de auxiliar a tomar decisões sobre o _clock_factor.

```
// EPOS Thread Component Declarations

#ifndef __thread_h
#define __thread_h

#include <utility/queue.h>
#include <utility/handler.h>
#include <cpu.h>
#include <machine.h>
#include <system.h>
#include <scheduler.h>
#include <segment.h>
```

```
#include <architecture/ia32/monitoring_capture.h>
extern "C" { void __exit(); }

__BEGIN_SYS

class Thread
{
...
public:
    volatile int _missed_deadlines; //it's not unsigned because of the calculating process
    int _times_p_count;
    static Monitoring_Capture* _thread_monitor;
    static volatile bool _end_capture;
    static unsigned int _global_deadline_misses;
    static volatile unsigned int _clock_factor[Traits<Build>::CPUS];
    static volatile unsigned int _clock_min[Traits<Build>::CPUS];
    static volatile unsigned int _slowdown[Traits<Build>::CPUS];
    static volatile float _bef_channel5[Traits<Build>::CPUS];
    static volatile float _bef_channel6[Traits<Build>::CPUS];
};
...
```

```
include/pmu.h
- Alterado o valor do "_channel_3" para 14.
- Alterado o valor do "_channel_5" para 138.
- Alterado o valor do "_channel_6" para 139.
(Sendo estes os eventos utilizados pela heurística)
```

```
// EPOS PMU Mediator Common Package

#ifndef __pmu_h
#define __pmu_h

#include <system/config.h>

__BEGIN_SYS

class PMU_Common
{
Public:
...
    static const unsigned int _channel_3 = 14; // sum 4 after each execution
    static const unsigned int _channel_4 = _channel_3+1;
    static const unsigned int _channel_5 = 138;
    static const unsigned int _channel_6 = 139;
...
};
```

```
include/periodic_thread.h:
- wait_next() -> adição do código da heurística.
```

```
   static volatile bool wait_next() {

        Periodic_Thread * t = reinterpret_cast<Periodic_Thread *>(running());
        t->_missed_deadlines = t->_times_p_count - (t->_alarm_times->_times);
        float channel_3 = 0;
       float channel_5 = 0;
       float channel_6 = 0;
        if (Criterion::heuristic) {
           if (t->_missed_deadlines > 0) {
               _clock_factor[Machine::cpu_id()] = 8;
           } else if (_thread_monitor->last_capture(Machine::cpu_id(), 0) > 0) {
               unsigned long long ts = _thread_monitor->last_capture(Machine::cpu_id(), 7);
               unsigned long long ts_dif = _thread_monitor->time() - ts;
               if (ts_dif > 999) {
                   channel_3 = (PMU::read(3) - _thread_monitor->last_capture(Machine::cpu_id(),
3)) / (ts_dif * 1.0);
                   channel_5 = (PMU::read(5) - _thread_monitor->last_capture(Machine::cpu_id(),
5)) / (ts_dif);
                   channel_6 = (PMU::read(6) - _thread_monitor->last_capture(Machine::cpu_id(),
6)) / (ts_dif);
                   if (channel_3 <= 20) {
                       if (channel_3 < 8 && _clock_factor[Machine::cpu_id()] == 8) {
                            if (_slowdown[Machine::cpu_id()] > 0)
                                _slowdown[Machine::cpu_id()]--;
                            else {
                                _clock_factor[Machine::cpu_id()]--;
                            }
                        } else if (channel_3 > 10 && _clock_factor[Machine::cpu_id()] == 7) {
                            _clock_factor[Machine::cpu_id()]++;
                        } else if (channel_3 >= 0.8 && _clock_factor[Machine::cpu_id()] == 6) {
                                _clock_factor[Machine::cpu_id()]++;
                        }  else if (channel_3 >= 0.2 && _clock_factor[Machine::cpu_id()] < 6) {
                                _clock_factor[Machine::cpu_id()]+= 2;
                        } else {
                            if (_clock_factor[Machine::cpu_id()] > 5) {
                                if (_slowdown[Machine::cpu_id()] > 0)
                                    _slowdown[Machine::cpu_id()]--;
                                else {
                                    if (Machine::cpu_id() % 2) {
                                        if (_clock_factor[Machine::cpu_id()-1] <=
_clock_factor[Machine::cpu_id()]) {
                                            _clock_factor[Machine::cpu_id()]--;
                                        }
                                    } else {
                                        if (_clock_factor[Machine::cpu_id()+1] <=
_clock_factor[Machine::cpu_id()]) {
                                            _clock_factor[Machine::cpu_id()]--;

                                        }
                                    }
                                }
                            }
                        }
```

```
                    } else if (channel_6 > 0 || channel_5 > 0) {

                        if (_clock_factor[Machine::cpu_id()] == 8) {
                            if (_slowdown[Machine::cpu_id()] > 0){
                                _slowdown[Machine::cpu_id()]--;
                            }
                            else if ((channel_5 >= 30 && channel_6 >= 30) ||
                             (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 && channel_6 >=
_bef_channel6[Machine::cpu_id()]+2)) {
                                _clock_factor[Machine::cpu_id()]--;
                             }

                        } else if (_clock_factor[Machine::cpu_id()] == 7) {
                             if (_slowdown[Machine::cpu_id()] > 0) {
                                 _slowdown[Machine::cpu_id()]--;
                                 if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 5 &&
channel_6 <= _bef_channel6[Machine::cpu_id()] - 5 && channel_3 > 250) {
                                     _clock_factor[Machine::cpu_id()]++;
                                 }
                             } else if ((channel_5 >= 25 && channel_6 >= 25) ||
                              (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 && channel_6 >=
_bef_channel6[Machine::cpu_id()]+2)) {
                                 _clock_factor[Machine::cpu_id()]--;
                             } else if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 5 &&
channel_6 <= _bef_channel6[Machine::cpu_id()] - 5 && channel_3 > 250) {
                                 _clock_factor[Machine::cpu_id()]++;
                             }

                         } else if (_clock_factor[Machine::cpu_id()] < 7 &&
(_bef_channel5[Machine::cpu_id()] > 0 || _bef_channel6[Machine::cpu_id()] > 0)) {
                             if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 3 && channel_6
<= _bef_channel6[Machine::cpu_id()] - 2) {
                                 if (_clock_factor[Machine::cpu_id()] < 6) {
                                     if (channel_3 > 60) {
                                         _clock_factor[Machine::cpu_id()]+= 2;
                                         _slowdown[Machine::cpu_id()] = 2;
                                     } else if (channel_3 > 20) {
                                         _clock_factor[Machine::cpu_id()]++;
                                         _slowdown[Machine::cpu_id()] = 2;
                                     }
                                 } else if (channel_3 > 40 && channel_6 <= 20 && channel_5 <=
20) {
                                     _clock_factor[Machine::cpu_id()]++;
                                     _slowdown[Machine::cpu_id()] = 2;
                                 } else if (_slowdown[Machine::cpu_id()] > 0) {
                                     _slowdown[Machine::cpu_id()]--;
                                 }
                             } else {
                                 if (_slowdown[Machine::cpu_id()] > 0) {
                                     _slowdown[Machine::cpu_id()]--;
                                 } else if (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 &&
channel_6 >= _bef_channel6[Machine::cpu_id()]+2
                                     && channel_3 < 110 && _clock_factor[Machine::cpu_id()] >
```

```
5){
                                    _clock_factor[Machine::cpu_id()]--;
                                }
                            }

                        }
                    }
                    _bef_channel6[Machine::cpu_id()] = channel_6;
                    _bef_channel5[Machine::cpu_id()] = channel_5;
                }
            }
            if (Machine::cpu_id() % 2) {
                if ((_clock_factor[Machine::cpu_id()-1] - 1) > _clock_factor[Machine::cpu_id()])
{
                    _clock_factor[Machine::cpu_id()] = _clock_factor[Machine::cpu_id()-1] -1;
                }
            } else {
                if ((_clock_factor[Machine::cpu_id()+1] - 1) > _clock_factor[Machine::cpu_id()])
{
                    _clock_factor[Machine::cpu_id()] = _clock_factor[Machine::cpu_id()+1] -1;
                }
            }
            CPU::clock((CPU::clock()/8 * _clock_factor[Machine::cpu_id()]));
        }

        Thread::_thread_monitor->capture(CPU::temperature(), PMU::read(0), PMU::read(1),
PMU::read(2), PMU::read(3), PMU::read(4), PMU::read(5),
            PMU::read(6), reinterpret_cast<volatile unsigned int>(t), 2, Machine::cpu_id(),
t->_missed_deadlines,  _clock_factor[Machine::cpu_id()]);
        if(t->_alarm._times) {
            t->_semaphore.p();
            t->_times_p_count--;
        }

        return t->_alarm._times;
    }
```

src/component/thread.cc:
- dispatch() -> Adição do código da heurística.

```
void Thread::dispatch(Thread * prev, Thread * next, bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();

        //Monitoring Capture
        if(Criterion::monitoring && !_end_capture) {
            if (prev->priority() != IDLE && prev->priority() != MAIN) {
                prev->_missed_deadlines = prev->_times_p_count - (prev->_alarm_times->_times +
1);
            } else
```

```
                prev->_missed_deadlines = 0;

            if (next->priority() != IDLE && next->priority() != MAIN) {
                next->_missed_deadlines = next->_times_p_count - (next->_alarm_times->_times +
1);
            } else
                next->_missed_deadlines = 0;

            if (prev->_missed_deadlines < 0)
                    prev->_missed_deadlines = 0;

            if (next->_missed_deadlines < 0)
                    next->_missed_deadlines = 0;

            float channel_3 = 0;
            float channel_5 = 0;
            float channel_6 = 0;
            if ( next->priority() != IDLE && Criterion::heuristic) {
                if (prev->_missed_deadlines > 0 || next->_missed_deadlines > 0) {
                    _clock_factor[Machine::cpu_id()] = 8;
                } else if (_thread_monitor->last_capture(Machine::cpu_id(), 0) > 0) {
                    unsigned long long ts = _thread_monitor->last_capture(Machine::cpu_id(),
7);
                    unsigned long long ts_dif = _thread_monitor->time() - ts;
                    if (ts_dif > 999) {
                        channel_3 = (PMU::read(3) -
_thread_monitor->last_capture(Machine::cpu_id(), 3)) / (ts_dif * 1.0);
                        channel_5 = (PMU::read(5) -
_thread_monitor->last_capture(Machine::cpu_id(), 5)) / (ts_dif);
                        channel_6 = (PMU::read(6) -
_thread_monitor->last_capture(Machine::cpu_id(), 6)) / (ts_dif);
                        if (channel_3 <= 20 && prev->priority() != IDLE) {
                            //if (Machine::cpu_id() == 6)
                            //    db<Thread> (WRN) << 30 << endl;
                            if (channel_3 < 8 && _clock_factor[Machine::cpu_id()] == 8) {
                                if (_slowdown[Machine::cpu_id()] > 0)
                                    _slowdown[Machine::cpu_id()]--;
                                else {
                                    _clock_factor[Machine::cpu_id()]--;
                                }
                            } else if (channel_3 > 10 && _clock_factor[Machine::cpu_id()] == 7)
{
                                _clock_factor[Machine::cpu_id()]++;
                            } else if (channel_3 >= 0.8 && _clock_factor[Machine::cpu_id()] ==
6) {
                                _clock_factor[Machine::cpu_id()]++;
                            }  else if (channel_3 >= 0.2 && _clock_factor[Machine::cpu_id()] <
6) {
                                _clock_factor[Machine::cpu_id()]+= 2;
                            } else {
                                if (_clock_factor[Machine::cpu_id()] > 5) {
                                    if (_slowdown[Machine::cpu_id()] > 0)
                                        _slowdown[Machine::cpu_id()]--;
```

```
                                    else {
                                        if (Machine::cpu_id() % 2) {
                                            if (_clock_factor[Machine::cpu_id()-1] <=
_clock_factor[Machine::cpu_id()]) {
                                                _clock_factor[Machine::cpu_id()]--;
                                            }
                                        } else {
                                            if (_clock_factor[Machine::cpu_id()+1] <=
_clock_factor[Machine::cpu_id()]) {
                                                _clock_factor[Machine::cpu_id()]--;

                                            }
                                        }
                                    }
                                }
                            }
                        } else if (channel_6 > 0 || channel_5 > 0) {

                            if (_clock_factor[Machine::cpu_id()] == 8) {
                                if (_slowdown[Machine::cpu_id()] > 0){
                                    _slowdown[Machine::cpu_id()]--;
                                }
                                else if ((channel_5 >= 30 && channel_6 >= 30) ||
                                    (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 &&
channel_6 >= _bef_channel6[Machine::cpu_id()]+2)) {
                                    _clock_factor[Machine::cpu_id()]--;
                                }

                            } else if (_clock_factor[Machine::cpu_id()] == 7) {
                                if (_slowdown[Machine::cpu_id()] > 0) {
                                    _slowdown[Machine::cpu_id()]--;
                                    if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 5 &&
channel_6 <= _bef_channel6[Machine::cpu_id()] - 5 && channel_3 > 250) {
                                        _clock_factor[Machine::cpu_id()]++;
                                    }
                                } else if ((channel_5 >= 25 && channel_6 >= 25) ||
                                 (channel_5 >= _bef_channel5[Machine::cpu_id()]+2 && channel_6
>= _bef_channel6[Machine::cpu_id()]+2)) {
                                    _clock_factor[Machine::cpu_id()]--;
                                } else if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 5 &&
channel_6 <= _bef_channel6[Machine::cpu_id()] - 5 && channel_3 > 250) {
                                    _clock_factor[Machine::cpu_id()]++;
                                }

                            } else if (_clock_factor[Machine::cpu_id()] < 7 &&
(_bef_channel5[Machine::cpu_id()] > 0 || _bef_channel6[Machine::cpu_id()] > 0)) {
                                if (channel_5 <= _bef_channel5[Machine::cpu_id()] - 3 &&
channel_6 <= _bef_channel6[Machine::cpu_id()] - 2) {
                                    if (_clock_factor[Machine::cpu_id()] < 6) {
                                        if (channel_3 > 60) {
                                            _clock_factor[Machine::cpu_id()]+= 2;
                                            _slowdown[Machine::cpu_id()] = 2;
                                        } else if (channel_3 > 20) {
```

```
                                        _clock_factor[Machine::cpu_id()]++;
                                        _slowdown[Machine::cpu_id()] = 2;
                                    }
                                } else if (channel_3 > 40 && channel_6 <= 20 && channel_5
<= 20) {
                                    _clock_factor[Machine::cpu_id()]++;
                                    _slowdown[Machine::cpu_id()] = 2;
                                } else if (_slowdown[Machine::cpu_id()] > 0) {
                                    _slowdown[Machine::cpu_id()]--;
                                }
                            } else {
                                if (_slowdown[Machine::cpu_id()] > 0) {
                                    _slowdown[Machine::cpu_id()]--;
                                } else if (channel_5 >= _bef_channel5[Machine::cpu_id()]+2
&& channel_6 >= _bef_channel6[Machine::cpu_id()]+2
                                    && channel_3 < 110 && _clock_factor[Machine::cpu_id()]
> 5){
                                    //channel_4 = (PMU::read(3) -
_thread_monitor->last_capture(Machine::cpu_id(), 4)) / (ts_dif);
                                    //if (channel_4 < 350)
                                    _clock_factor[Machine::cpu_id()]--;
                                }
                            }

                        }
                    }
                    _bef_channel6[Machine::cpu_id()] = channel_6;
                    _bef_channel5[Machine::cpu_id()] = channel_5;
                }
            }
            if (Machine::cpu_id() % 2) {
                if ((_clock_factor[Machine::cpu_id()-1] - 1) >
_clock_factor[Machine::cpu_id()]) {
                    _clock_factor[Machine::cpu_id()] = _clock_factor[Machine::cpu_id()-1]
-1;
                }
            } else {
                if ((_clock_factor[Machine::cpu_id()+1] - 1) >
_clock_factor[Machine::cpu_id()]) {
                    _clock_factor[Machine::cpu_id()] = _clock_factor[Machine::cpu_id()+1]
-1;
                }
            }
            if (next->priority() == IDLE) {
                _bef_channel6[Machine::cpu_id()] = 0;
                _bef_channel5[Machine::cpu_id()] = 0;
            }
            CPU::clock((CPU::clock()/8 * _clock_factor[Machine::cpu_id()]));
        }
        _thread_monitor->capture(CPU::temperature(), PMU::read(0), PMU::read(1),
PMU::read(2), PMU::read(3), PMU::read(4), PMU::read(5),
        PMU::read(6), reinterpret_cast<volatile unsigned int>(prev), 1, Machine::cpu_id(),
prev->_missed_deadlines + next->_missed_deadlines, _clock_factor[Machine::cpu_id()]);
```

```
        }
    }

    if(prev != next) {
        if(prev->_state == RUNNING)
            prev->_state = READY;
        next->_state = RUNNING;

        db<Thread>(TRC) << "Thread::dispatch(prev=" << prev << ",next=" << next << ")" << endl;
        db<Thread>(INF) << "prev={" << prev << ",ctx=" << *prev->_context << "}" << endl;
        db<Thread>(INF) << "next={" << next << ",ctx=" << *next->_context << "}" << endl;

        if(smp)
            _lock.release();

        if(multitask && (next->_task != prev->_task))
            next->_task->activate();

        CPU::switch_context(&prev->_context, next->_context);
    } else
        if(smp)
            _lock.release();
    CPU::int_enable();
}
```

A3. ARTIGO PADRÃO SBC

Heurística Para Escalonamento Multicore De Tempo Real visando Determinismo Temporal

José Luis C. Hoffmann

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

{jose.hoffmann}@grad.ufsc.br

Abstract. *With the evolution of the processor chips, performance and energy consumption are increasingly seeking to be improved. Such optimization has been developed on Hardware, by increasing the number of resources available, and on Software, where the goal is to make the best use of these resources. Within the context of performance monitoring, several surveys have been developed to learn about the execution through monitoring channels provided by the processor's Performance Monitoring Unit (PMU). In this paper, the focus is on the developing of a heuristic using performance data count monitored through the PMU to learn about performance patterns, and use this knowledge to control dynamic voltage and frequency scaling (DVFS), maintaining the temporal determinism of critical tasks with a high use of memory hierarchy or high recursion use, while reducing the power consumption of processor cores up to 15%.*

Resumo. *Com a evolução dos chips do processador, o desempenho e o consumo de energia estão cada vez mais buscando melhorias. Essa otimização foi desenvolvida em hardware, aumentando o número de recursos disponíveis e em software, onde o objetivo é fazer o melhor uso desses recursos. Dentro do contexto de monitoramento de desempenho, várias pesquisas foram desenvolvidas para aprender sobre a execução através de canais de monitoramento fornecidos pela Unidade de Monitoramento de Desempenho (PMU) do processador. Neste trabalho, o foco é no desenvolvimento de uma heurística usando dados de desempenho monitorados através da PMU para aprender sobre padrões de desempenho e usar esse conhecimento para controlar o escalonamento dinâmico de voltagem e freqüência (DVFS), mantendo o determinismo temporal de tarefas críticas com alto uso da hierarquia de memória ou alto uso de recursão, enquanto é reduzido o consumo de energia dos núcleos de processador em até 15%.*

Palavras-chave: Multi-core, Tempo Real, Heurística, DVFS, PMU, MSR, IA-32, Sensores de Hardware, Determinismo Temporal, Energy Aware.

1. Introdução

O conceito de determinismo temporal na computação ganhou grande importância com o advento de sistemas de tempo real, onde ao criar um sistema que deve executar não somente buscando a corretude lógica da execução, mas também a corretude temporal, foi imposta a necessidade de que os momentos de início e fim determinados para a execução de uma tarefa fossem respeitados. Em alguns sistemas de tempo real, como por exemplo um player de vídeo, a falta de sincronia entre os frames pode ser incômoda, mas não chega a ser algo crítico para o sistema, em um outro caso, como

por exemplo o controle de um carro, se o freio não responder no momento correto pode ocasionar uma fatalidade, sendo, este último, caracterizado como um sistema crítico. Esse e muitos outros exemplos de Deadlines que não foram respeitadas em um sistema crítico, que motivam pesquisas na comunidade de computação para garantir o determinismo temporal, ao mesmo tempo em que busca-se otimizar tanto o desempenho, quanto o consumo energético.

Os processadores, ao longo do tempo, foram cada vez mais sendo projetados para prover informações sobre a execução, começando com simples contadores, como Instruções finalizadas, até uma gama de mais de 200 tipos de eventos nas implementações mais atuais. O uso de uma Unidade de Monitoramento de Performance (PMU), e de sensores de hardware, para obter dados sobre o comportamento dos Cores, a fim de utilizá-los para entender e até mesmo tentar predizer os acontecimentos dentro de uma CPU, é uma área que vem crescendo nos últimos anos. Sendo possível estimar o desempenho e o gasto energético de uma carga de trabalho dinâmica em tempo de execução através dos contadores de hardware disponíveis na arquitetura, como demonstrado em Run-DMC.

Um outro exemplo é o sistema operacional SPARTA, que busca melhorar a eficiência energética e o desempenho de processadores manycores heterogêneos, através de uma caracterização em tempo de execução das tarefas, para então atribuir prioridades na alocação de recursos. O sistema SPARTA obteve uma redução do consumo energético de até 23%, o que gerou uma grande motivação para este trabalho.

A ideia que deu início a este artigo foi a de utilizar um sistema de baixa interferência (EPOS) para realizar várias vezes uma mesma execução, onde em cada uma delas, um novo conjunto de eventos é utilizado na PMU. Isso abre a possibilidade de agrupar todos os eventos como se fossem capturados numa única execução, que nas versões atuais dos processadores é impossível devido a limitação de registradores em hardware para a unidade de monitoramento. Com esses dados agrupados, é possível a aplicação de técnicas de Data Mining, para então verificar correlações entre as variáveis capturadas, buscando um grupo que possa ser utilizado durante a execução descrevendo um comportamento do sistema que deseja-se monitorar. Esse processo pode ser realizado para vários conjuntos de tarefas distintos, fazendo então um refinamento das correlações.

O foco é de encontrar um conjunto de eventos correlacionado com perdas de deadline quando o sistema está afetado por DVFS, com isso, utilizar este conjunto de eventos para predizer uma perda de deadline para que então, seja controlado de forma preventiva o uso da técnica DVFS, visando diminuir o consumo e manter a corretude temporal, ou seja, tratando o problema antes que ele ocorra através da análise do conjunto de variáveis estabelecido.

2. Materiais

Os dados analisados foram capturados através de sensores e contadores de software e hardware, sendo eles vindos dos contadores da Unidade de Monitoramento de Performance da Intel (PMU) e dos sensores térmicos e energéticos, com leituras dos MSRs ligados a temperatura (IA32_THERM_STATUS, IA32_TEMPERATURE_TARGET) e da Interface RAPL (MSR_RAPL_POWER_UNIT, MSR_PKG_ENERGY_STATUS e MSR_PP0_ENERGY_STATUS), utilizada para controle e aferição de consumo energético.

O computador utilizado durante a fase de testes, geração de dados e configuração da heurística contava com um processador Intel® Core™ i7-2600, de arquitetura Sandy-Bridge, com 4 GB de memória RAM DDR3 de 1333MHz, placa mãe PCWARE IPMH61R3, com fonte de alimentação ALLIED SL-8180 BTX instalado sob corrente de 220V, disponibilizado pelo laboratório (LISHA) para captura de dados e testes das heurísticas. Para aferições energéticas, além

da interface RAPL, foi utilizado um analisador de potência e qualidade de energia Fluke 435 series II, para aferições de consumo energético.

Para as análises de Mineração de Dados através da aplicação de algoritmos de correlação de atributos, fora utilizado o software WEKA (https://www.cs.waikato.ac.nz/ml/weka/), um software de código aberto desenvolvido pelo grupo de Machine Learning da universidade de waikato.

3. Sistema Não Intrusivo de Capturas

Para coleta dos dados de performance, foi desenvolvido um sistema de monitoramento de performance não intrusivo em conjunto com o discente Leonardo Passig Horstmann (15103030), que armazena dados capturados durante a execução como um retrato de um momento da execução. Cada captura serve para adquirir informações sobre eventos de hardware e variáveis de sistema que possam ser úteis para a análise da execução.

Em síntese, uma captura é definida como uma descrição de um determinado momento do sistema, contendo: Os valores dos 7 contadores de eventos de Hardware (3 fixos e 4 configuráveis); O tempo em que a captura foi realizada (em formato Unix Time); A temperatura da CPU em que a captura foi realizada; A Thread que estava executando, sua prioridade e a quantidade de Deadlines perdidas por ela; A quantidade de Deadlines perdidas até o momento pelo sistema como um todo; A contagem de energia consumida pela interface RAPL, trazendo o consumo do PKG e do PP0.

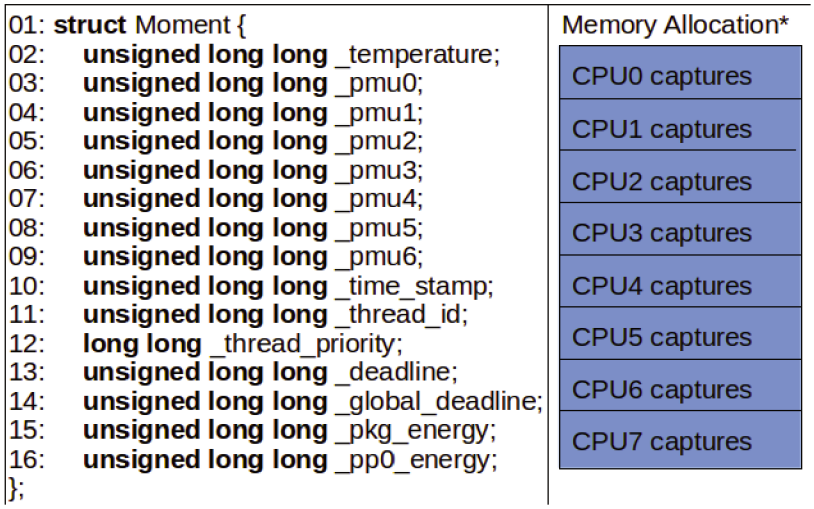


Figura 1. Estrutura de um Momento de captura e organização da alocação de memória.

Essas capturas são armazenadas em uma região da memória para cada uma das CPUs e consumidas somente no final da execução. Isto elimina a intrusividade de captura e consumo dos dados, visto que a escrita dos dados é paralelizada entre as CPUs e o consumo não tem influência na execução. Outro fator importante para evitar a intrusividade é realizar capturas somente em momentos de não intrusão, com estudos realizados em cima da estrutura do sistema EPOS, foi possível identificar os pontos de troca de contexto das threads como uma região propícia para realizar as capturas sem gerar aumento no tempo de execução. O sistema de captura funciona da seguinte maneira:

1. O sistema de captura é configurado junto a inicialização do sistema de Threads do EPOS.
2. A PMU é configurada no início da execução, junto da inicialização do sistema de escalonamento;
3. As capturas são habilitadas assim que a aplicação inicia sua execução, e também são desabilitadas ao fim da aplicação.

4. Sempre que o sistema reescalona uma Thread, uma captura daquele momento é realizada e armazenada.
5. Antes do sistema operacional finalizar, os momentos armazenados são enviados via porta serial para outro computador.
6. Neste outro computador um log é gerado, e em sequência é consumido pelo algoritmo de envio dos dados para uma plataforma IoT.

4. Geração de Tarefas e Pré-Processamento dos Logs de Dados

Com o sistema de capturas não intrusivo desenvolvido, o próximo passo então é gerar tarefas que estimulem o processador e, por consequência, a contagem dos eventos de desempenho feita pela PMU. Para isso, foram selecionadas dois tipos de tarefas, o algoritmo de Fibonacci Recursivo, que, por sua vez, calcula a série de Fibonacci até um determinado valor, modificando-o para executar também algumas operações de ponto flutuante, e o algoritmo de Cópia de Regiões de Memória (memcpy), que, por sua vez, estimula a hierarquia de memória do processador.

Além do tipo de tarefa, também é necessário selecionar um grupo de tarefas (task-set) para executar em um processador multicore, para isto, foi utilizado o método de geração de tarefas descrito por Gracioli em Real-Time Operating System Support for Multicore Applications (2014), chamado de Particionamento de Tarefas, que gera o conjunto de tarefas para uma arquitetura Multi-core em um sistema de Tempo Real. A utilização do conjunto de tarefas em cada CPU pode ser encontrado na Tabela 1.

As execuções, devido a limitação de 4 canais configuráveis para monitoramento simultâneo da PMU, foram organizadas em grupos, onde cada execução do grupo cobria 4 eventos configuráveis diferentes, e cada grupo executa sobre as mesmas configurações, ou seja, mesmo conjunto de tarefas e mesmo tipo de tarefa, com isso pode-se inferir correlações entre os canais e as respectivas perdas de deadline da execução.

Vale citar também que, como o objetivo é de descobrir eventos que mudem seu comportamento perto de uma perda de deadline, os dados foram capturados no sistema executando com um desempenho entregue menor do que o necessário, isto feito através da aplicação do método clock modulation (DVFS), reduzindo a frequência e gerando perda de deadline durante a execução, para ambos os casos, foram analisados os dados das CPUs 6 e 7, com o DVFS reduzindo a frequência para 62.5%.

Tabela 1. Uso total de cada CPU para os conjuntos de tarefas utilizadas para geração e análise dos dados.

CPU	Uso Conjunto 1	Uso Conjunto 2
0	55,832%	54,196%
1	47,055%	90,674%
2	68,493%	68,919%
3	51,846%	64,664%
4	46,548%	83,952%
5	56,758%	52,879%

6	60,968%	56,965%
7	62,982%	78,177%

Antes da análise dos dados, se torna necessário um tratamento dos logs para que os dados tenham utilidade nos algoritmos de correlação de atributos. Logo, o crescimento a ser analisado representa quanto o contador cresceu em relação a última captura, dividido pelo tempo decorrido desde a última captura, ou seja, a média de crescimento do evento por microssegundos. Além disso, foram removidas as capturas muito próximas, com diferença de tempo de menos de 300 microssegundos, visto que geraram outliers nos dados, e a contagem de perdas de deadline foi substituída por uma variável que representa se ocorreu uma perda de deadline entre as duas capturas naquela CPU (1) ou não (0). Um exemplo da saída deste pré-processamento pode ser encontrada na tabela abaixo (Tabela 2).

Tabela 2. Exemplo da organização dos dados na saída do tratamento dos logs.

Time	pmu0	pmu1	pmu2	pmu12	pmu13	pmu14	pmu15	deadlineM
1798640	3057	2693	3392	0	0	67	0	0
1802598	3096	2693	3392	0	0	96	0	1

5. Análise dos dados

A análise dos dados foi inicialmente apenas aplicando o método de correlação de atributos disponível no WEKA, onde é calculada a diferença entre a variação dos atributos em questão ao longo de todo o arquivo de dados. Porém, apenas o uso deste método não apresentou bons resultados, visto que suas limitações influenciam no resultado, principalmente o fato de que o algoritmo em questão não leva em conta a ordem temporal dos dados.

A validação das correlações foi feita a mão, analisando os dados nos logs, principalmente observando o comportamento em uma CPU com perdas de deadline e em uma CPU sem perdas de deadline. O caso mais comum entre os eventos com maior correlação pode ser ilustrado pela figura abaixo (Figura 2), onde o evento mantém um valor constante durante a execução, e apenas existe variação quando a CPU entra em idle, tendo o mesmo comportamento sem perdas de deadline, logo seu comportamento não está ligado às perdas de deadline.

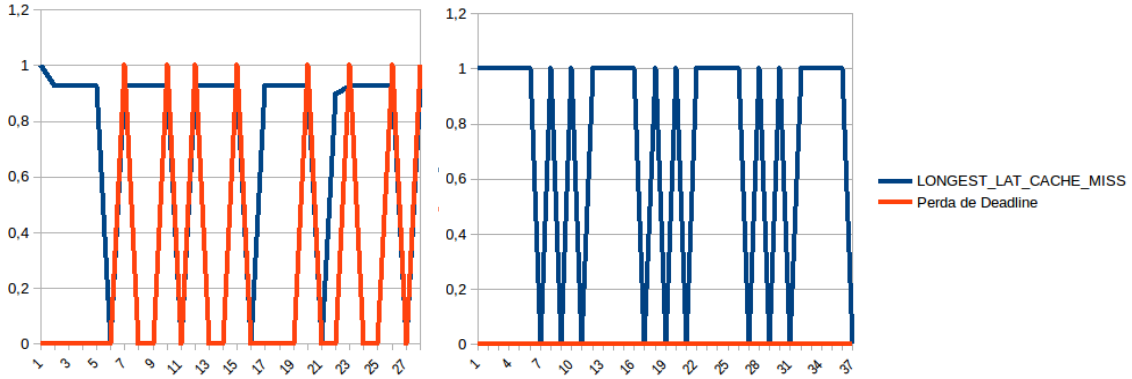


Figura 2. Comportamento da contagem do evento LONGEST_LAT_CACHE_MISS em uma CPU com perdas de deadline (esquerda) e em uma CPU sem perdas de Deadline (direita).

Utilizando-se da análise manual dos logs pré-processados, foi possível identificar comportamentos interessantes de alguns eventos em relação às perdas de deadline. O evento LD_BLOCKS_NO_SR, que representa a contagem de Split Loads bloqueados por recurso indisponível (loads que foram divididos em mais de um load, visto que a área em questão era maior que a cache de nível 1, e foram bloqueados porque o recurso estava ocupado) teve uma correlação de aproximadamente 5% com as perdas de deadline, porém utilizando-se da técnica de forward (que visa deslocar a "classe" para frente ou para trás nas capturas, para dar a ideia de correlacionar com o evento em períodos anteriores a sua ocorrência, ou seja, que podem ser os geradores da falha mais a frente) gerou uma correlação de aproximadamente 12%, que a princípio já se destacava em relação as demais correlações para as execuções do algoritmo de Fibonacci Recursivo, e, após uma análise manual do canal, foi identificado um comportamento de um pico no crescimento deste evento momentos antes de uma perda de deadline, isto pode ser verificado na figura abaixo (Figura 3).

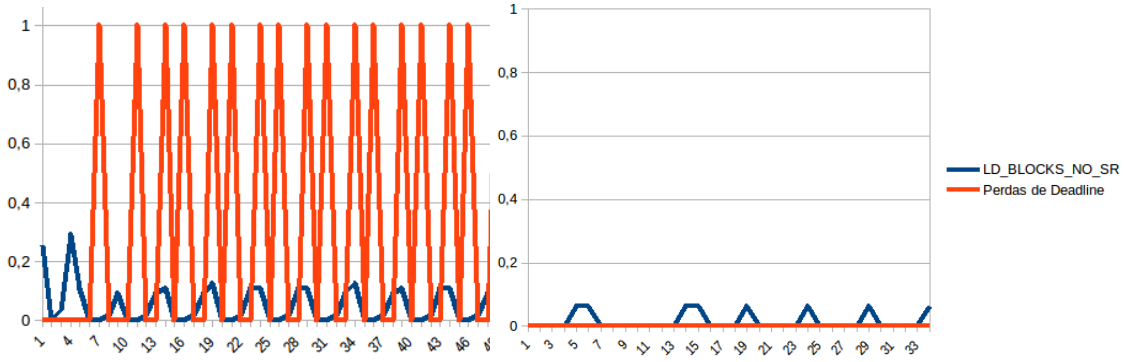


Figura 3. Comportamento da contagem do evento LD_BLOCKS_NO_SR em uma CPU com perdas de deadline (esquerda) e em uma CPU sem perdas de Deadline (direita).

Logo, a ideia é de que como o evento conta um "erro" no sistema, quando ele tem um pico representa uma queda no desempenho, o que poderia ser suprido com um aumento da frequência de execução, e quando o mesmo apresenta picos baixos, representa que o sistema está executando com mais folga, logo pode ser um momento propício para a redução da frequência de operação da CPU.

Já observando os dados provindos de execuções de Cópias de Regiões de Memória, dois eventos apresentaram um comportamento correlacionado com as perdas de deadline, principalmente na análise manual, visto que apresentavam valores relativamente baixos pelo algoritmo de correlação de atributos do WEKA. Estes eventos são ITLB_ITLB_FLUSH e OFFCORE_REQUEST_DEMAND_DATA_RD, que contam os flush na tabela de instruções e a quantidade de requisições de leitura de dados globais, respectivamente. Ambos contam eventos de um desempenho sadio, ou seja, não contam erros da execução, logo um crescimento alto é esperado, principalmente no evento de requisições de dados globais, visto que o vetor copiado no algoritmo era global.

Logo, a correlação encontrada foi quando ambos aumentam ou diminuem seu crescimento em conjunto, visto que quando ambos diminuem, representa uma queda no desempenho, que pode ser suprida por uma maior frequência de operação evitando a perda de deadlines, e quando ambos estão aumentando, representa um momento de folga do sistema, sendo propício para a redução da frequência de operação da CPU. Este comportamento pode ser identificado no gráfico abaixo, onde

quando ambos crescimentos diminuíam em relação a captura anterior, uma perda de deadline estava próxima.

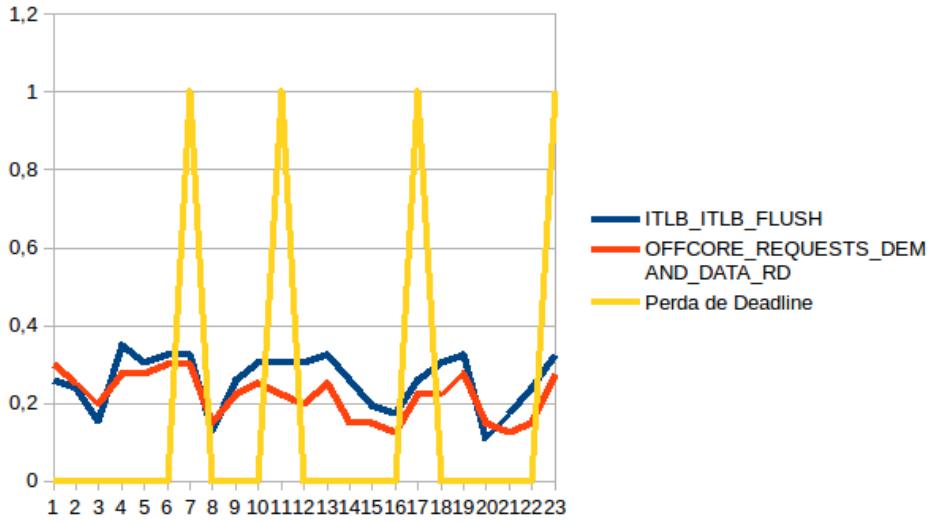


Figura 4. Comportamento da contagem dos eventos ITLB_ITLB_FLUSH e OFFCORE_REQUEST_DEMAND_DATA_RD em uma CPU com perdas de deadline.

Para complementar estes dois eventos, também foi adicionado a análise o evento LD_BLOCKS_NO_SR, visto que o mesmo complementava a heurística em alguns casos de falso positivo, ajudando na tomada de decisão de que somente é possível baixar a frequência quando o mesmo apresenta um crescimento baixo, e somente é necessário aumentar a frequência quando o mesmo apresenta um valor alto, claro, junto a análise do comportamento dos dois eventos já citados.

6. Heurística Desenvolvida

O desenvolvimento da heurística se deu de forma iterativa, utilizando inicialmente os comportamentos descritos no tópico anterior e uma aproximação do que seriam os pontos de tomada de decisão de aumentar ou diminuir a frequência. A cada iteração os pontos de tomada de decisão eram aprimorados, a fim de não ocorrerem perdas de deadline ao mesmo tempo que buscava-se manter a frequência mais baixa possível.

De acordo com as configurações disponíveis na arquitetura Intel Sandy Bridge em relação a modulação de clock, é possível realizar modulações em faixas de 12.5%, indo de 100% da frequência até 12.5% da frequência, sendo aqui as faixas chamadas de “fator”, sendo um fator aplicado a frequência da máquina, o cálculo segue a seguinte ideia: Frequência = (Frequência_Atual/8) * fator. Logo, diminuir ou aumentar a frequência, significa diminuir ou aumentar o fator.

Vale lembrar que em sistemas que disponibilizam a tecnologia de Hyperthreading, o fator aplicado a CPU física é o menor das duas CPUs lógicas, sendo assim, um controle do fator entre as duas CPUs lógicas de uma CPU física deve ocorrer, visto que se o uso for diferente, uma CPU aparenta poder baixar a frequência e isso afetaria o desempenho da que tem o maior uso. Além disso, como a heurística é preditiva, um “slowdown” é aplicado quando a frequência sobe, evitando que a mesma desça nas próximas duas entradas em pontos de checagem, para que surta o efeito de subir a frequência.

A heurística foi dividida em duas partes, uma para atender algoritmos recursivos, tais como o Fibonacci Recursivo, e uma para atender algoritmos com alto uso de memória, tais como o

algoritmo de Cópia de Regiões de Memória, a representação da heurística desenvolvida pode ser encontrada nas imagens a seguir (Figura 5, 6 e 7) na forma de árvore de decisão.

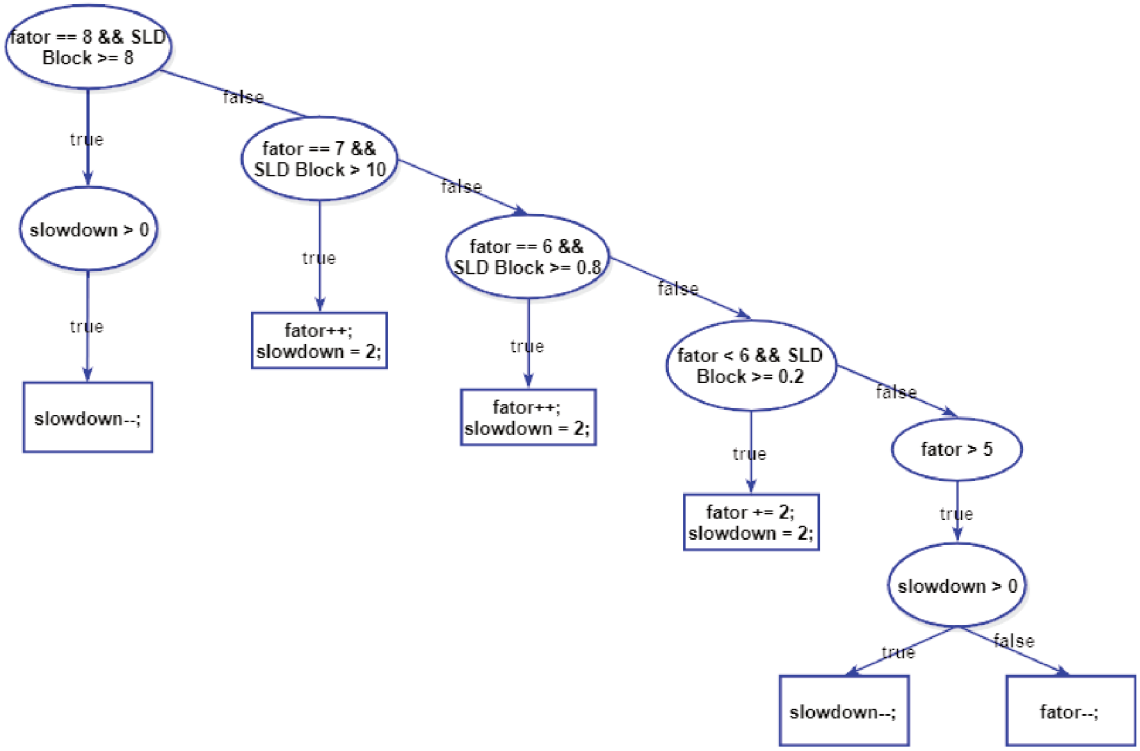


Figura 5. Árvore de decisão da primeira parte da heurística, para algoritmos recursivos, com comportamento similar ao Fibonacci Recursivo (SLD Block = LD_BLOCKS_NO_SR).

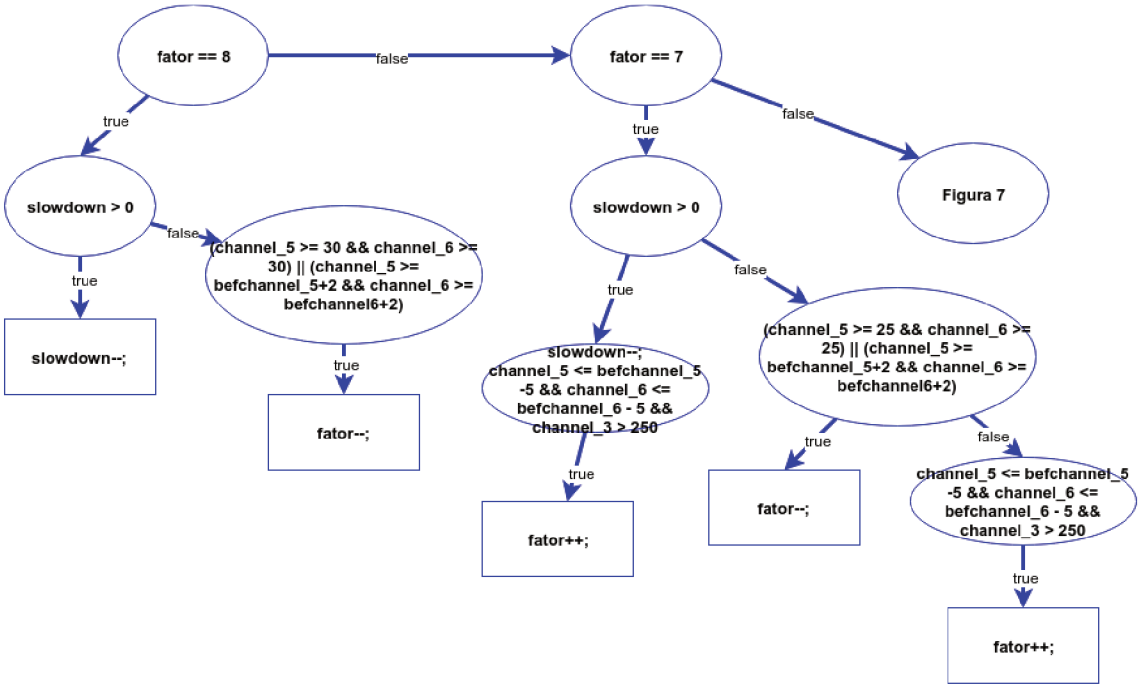


Figura 6. Árvore de decisão da segunda parte da heurística, para algoritmos com alto uso de memória global, com comportamento similar a Cópia de Região de Memória, em fatores 8 e 7 (Channel_3 = LD_BLOCKS_NO_SR; Channel_5 = ITLB_ITLB_FLUSH; Channel_6 = OFFCORE_REQUEST_DEMAND_DATA_READ; befchannel_(x) = crescimento captura anterior do channel x).

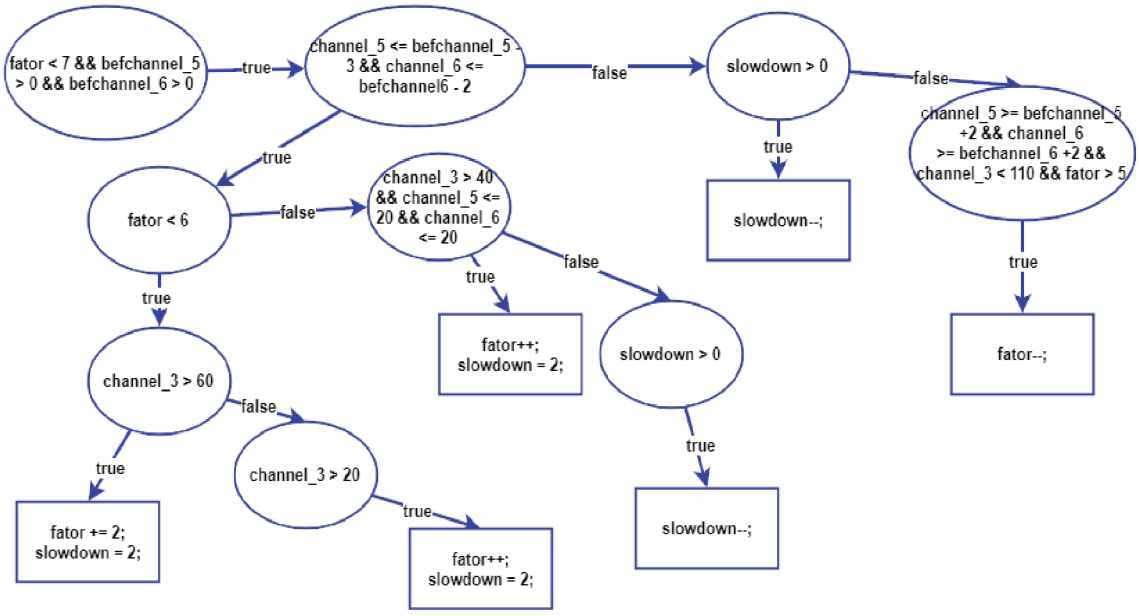


Figura 7. Árvore de decisão da terceira parte da heurística, para algoritmos com alto uso de memória global, com comportamento similar a Cópia de Região de Memória, em fatores 6 e 5 (Channel_3 = LD_BLOCKS_NO_SR; Channel_5 = ITLB_ITLB_FLUSH; Channel_6 = OFFCORE_REQUEST_DEMAND_DATA_READ; befchannel_(x) = crescimento captura anterior do channel x).

Sendo o ponto de divisão entre a parte 1 e 2 da heurística o valor correspondente ao crescimento apontado pelo contador do evento LD_BLOCKS_NO_SR, onde quando ele é menor que 20, provavelmente estamos em uma execução de Fibonacci Recursivo (parte 1), e se for acima de 20, estamos em uma execução de Cópia de Regiões de Memória (parte 2).

7. Resultados

Com as heurísticas polidas pelo processo iterativo, nenhuma perda de deadline ocorre para os testes atuais, basta então avaliar o desempenho em relação ao consumo energético do processador. Para isto, como citado no tópico 2 (Materiais), foi utilizado duas plataformas, uma delas a partir de análises do consumo na fonte do computador através do Fluke 435 series II, e a outra através da Interface disponibilizada em hardware pela própria Intel, o RAPL, o qual conta com o contador PP0, que contabiliza o consumo dos núcleos do processador a cada milissegundo, sendo este o principal afetado pela heurística.

Os testes realizados contam com uma execução com a heurística habilitada (C) e outra execução com a heurística desabilitada (S), e tiveram foco em execuções com os conjuntos de tarefas apresentados anteriormente na Tabela 1. Porém, foi necessário reduzir o uso de duas CPUs do conjunto 2, visto que apresentavam um uso muito alto, o qual a heurística não podia ter efeito, pois o primeiro nível de redução de frequência é de 82.5%, e isso já seria o suficiente para gerar uma perda de deadline, a não ser que os valores de tomada de decisão fossem modificados para identificar estes casos, mas isto diminuiria o ganho da heurística nas demais CPUs, visto que seria mais raro o caso de diminuir a frequência. As CPUs que tivera seu uso modificado então foram a 1 e a 4, onde o uso foi reduzido para 80%.

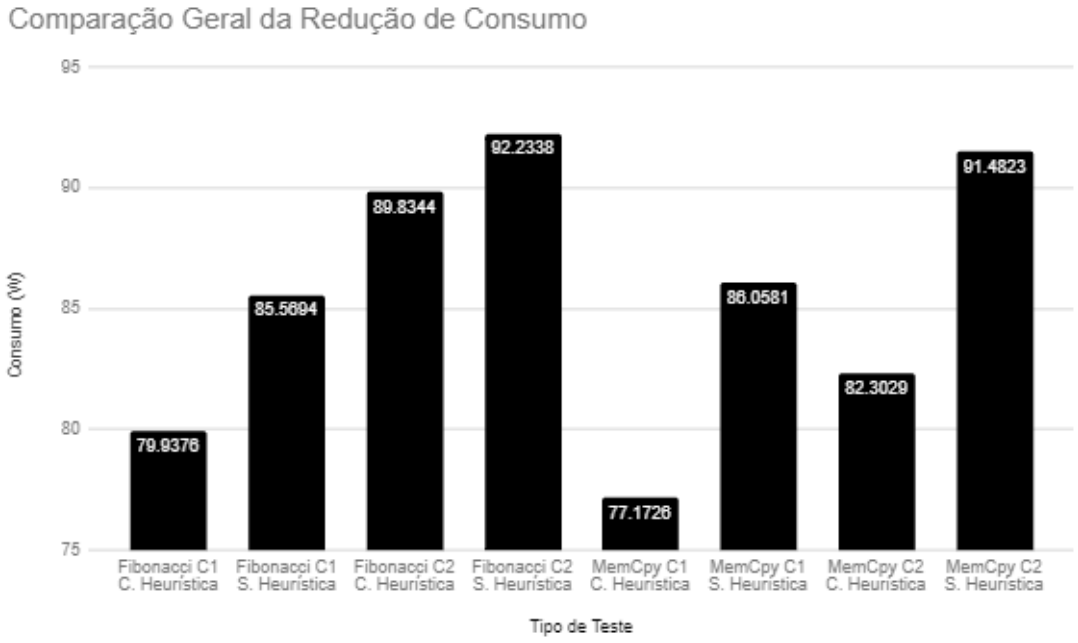


Figura 8. Comparação do consumo entre as execuções com e sem heurística, C1 é o conjunto de tarefas 1 e C2 é o conjunto de tarefas 2.

A figura acima representa a comparação do consumo médio entre as execuções, e como pode ser observado, a heurística teve resultados positivos para todos os casos testados, apresentando um resultado muito mais efetivo nos testes envolvendo a parte da heurística de cópia de regiões de memória, a qual usa mais canais para tomar as decisões. Uma análise mais detalhada do consumo pode ser verificada na Tabela a seguir (Tabela 3).

Tabela 3. Comparação de Consumo em valores entre execuções com e sem heursítica, C1 é o conjunto de tarefas 1 e C2 é o conjunto de tarefas 2.

Algoritmo e Conjunto de Tarefas	Sem Heurística	Com Heurística	Diferença
Fibonacci Rec. C1	85,5694 W	79,9376 W	5.6318 W (~6.5%)
Fibonacci Rec. C2	92,2338 W	89,8344 W	2.3994 W (~2,6%)
MemCpy C1	86,0581 W	77,1726 W	8,8855 W (~10,3%)
MemCpy C2	82,3029 W	91,4823 W	9,1794 W (~10%)
Fibonacci Rec. C1 PP0*	37,6 W	34,5 W	3,1 W (~6,5%)
MemCpy C1 PP0*	39,9 W	33,7 W	6,2 W (~15%)

*Interface RAPL da intel, contagem apenas do consumo dos núcleos processador.

Observando mais de perto as reduções do consumo, podemos confirmar um melhor desempenho para a segunda parte da heurística, principalmente se analisado o consumo a partir do registrador PP0 da interface RAPL, a qual apresenta a contagem do consumo apenas para os

núcleos do processador, a região foco da heurística, onde podemos encontrar uma redução de até 15% do consumo para o conjunto de tarefas 1.

Com isso, podemos verificar que os ganhos da heurística, sem perder nenhuma deadline durante a execução, foi de uma redução do consumo energético de até 6.5% para tarefas de Fibonacci Recursivo e de até 15% para tarefas de Cópias de Regiões de Memória (MemCpy). Vale citar que o sistema utilizado conta com a tecnologia de Hyper Threading, o que faz com que o uso de máximo de todas as CPUs não seja de 100%, o que implica que mesmo que uma CPU apresente um uso de 60%, o consumo não pode ser reduzido diretamente em 40% na mesma, o que reforça os bons resultados para alguns casos da heurística.

8. Conclusões e Trabalhos Futuros

O objetivo de desenvolver, em conjunto com o discente Leonardo Passig Horstmann (15103030), um sistema de captura de dados de execução não intrusivo, foi concluído com sucesso, proporcionando o desenvolvimento da heurística aqui apresentada. Além disso, o sistema também pode ser útil para vários pesquisadores da área que buscam analisar dados, sem interferência das capturas e do SO, do desempenho de um processador executando tarefas fisicamente, e não apenas em simuladores.

Sobre o desenvolvimento das heurísticas, foi possível concluir esta tarefa através da análise dos com algoritmos de Data Mining em conjunto com uma análise manual. Obtendo-se assim, resultados satisfatórios, onde a heurística desenvolvida consegue executar tarefas críticas relacionadas com um alto uso da hierarquia de memória e/ou com um alto uso de recursão, sem perder a característica de determinismo temporal, ao mesmo tempo que reduz o consumo dos núcleos processador em até 15%. A aplicação da heurística abrange sistemas multi-core de tempo real, principalmente sistemas embarcados, onde um baixo consumo de energia e um alto desempenho são cada vez mais necessários.

Trabalhos futuros podem ser realizados a partir do sistema de captura de dados não intrusivo desenvolvido, pois, através de várias técnicas de mineração de dados, muito pode se descobrir sobre otimização em sistemas multi-core de tempo real, principalmente com dados sem interferência e gerados por uma execução real de um conjunto de tarefas.

Também pode se desenvolver otimizações sobre a heurística aqui desenvolvida, principalmente com o uso da PEBS, um sistema do próprio processador, que gera interrupções no sistema operacional a partir de gatilhos configuráveis nos canais da PMU monitorados, a fim de melhorar a precisão das tomadas de decisões sobre o estado atual da execução. É possível também, estender a ideia utilizada no desenvolvimento da heurística, a fim de aumentar a gama de tarefas na qual a mesma consegue abranger.

Referências

Intel® 64 and IA-32 Architectures Software Developer's Manual, https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, Dezembro 2017.

Embedded Parallel Operating System, LISHA, 2018. https://epos.lisha.ufsc.br/HomePage, Junho, 2018.

Software/Hardware integration Laboratory, LISHA, 2018. https://lisha.ufsc.br, Janeiro, 2018.

HAMMOND, L., NAYFEH, B. A., OLU-KOTUN, K., "A Single-Chip Multiprocessor", Computer, vol. 30, no. 9, pp. 79-85, Setembro. 1997. https://ieeexplore.ieee.org/document/612253/, Junho 2018.

DONYANAVARD, B.; MÜCK, T.; SARMA, S.; DUTT, N., “SPARTA: Runtime task allocation for energy efficient heterogeneous manycores”, Department of Computer Science, University of California, Irvine, USA, IEEE, 2016. http://ieeexplore.ieee.org/document/7750975, Setembro 2017.

MÜCK, T.; SARMA, S.; DUTT, N.; “Run-DMC: runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency”, Amsterdam, The Netherlands, IEEE, 2015. https://dl.acm.org/citation.cfm?id=2830859, Setembro, 2017.

GRACIOLI, G., FRÖHLICH, A. A., “On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures”, ACM SIGOPS Operating Systems Review - Special Topics, vol. 49, no. 2, pág 2-16, Dezembro - 2015. https://dl.acm.org/citation.cfm?id=2883594, Setembro 2017.

SULEIMAN, D., IBRAHIM, M., HAMARASH, I., “Dynamic Voltage Frequency Scaling (Dvfs) For Microprocessors Power And Energy Reduction”, http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=E6F55A8CE6B176124DC60E15141B65B5?doi=10.1.1.111.1451&rep=rep1&type=pdf, Agosto 2017.