UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

José Luis Conradi Hoffmann

# Optimizing Energy Consumption of Multicore Real-Time Embedded Systems using Machine Learning

Florianópolis
2020

José Luis Conradi Hoffmann

# Optimizing Energy Consumption of Multicore Real-Time Embedded Systems using Machine Learning

Dissertação submetida ao Programa de Pós-Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de  Mestre em Ciência da Computação.
Orientador: Prof. Antônio A. M. Fröhlich, PhD.

Florianópolis
2020

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

```
Hoffmann, José Luis Conradi
   Optimizing Energy Consumption of Multicore Real-Time
Embedded Systems using Machine Learning / José Luis
Conradi Hoffmann ; orientador, Antônio Augusto Medeiros
Fröhlich, 2020.
   128 p.


   Dissertação (mestrado) - Universidade Federal de Santa
Catarina, Centro Tecnológico, Programa de Pós-Graduação em
Ciência da Computação, Florianópolis, 2020.

   Inclui referências.

   1. Ciência da Computação. 2. Sistemas Multicore
Embarcados de Tempo Real. 3. Otimização Energética. 4.
Aprendizado de Máquina. 5. Monitoramento Não Intrusivo. I.
Fröhlich, Antônio Augusto Medeiros. II. Universidade Federal
de Santa Catarina. Programa de Pós-Graduação em Ciência da
Computação. III. Título.
```

José Luis Conradi Hoffmann

**Optimizing Energy Consumption of Multicore Real-Time Embedded Systems using Machine Learning**

O presente trabalho em nível de Mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Mestre em Ciência da Computação.

Documento assinado digitalmente
Vania Bogorny
Data: 11/01/2021 08:30:59-0300
CPF: 684.586.570-15

Prof. Vania Bogorny, PhD.
Coordenadora do Programa de
Pós-Graduação

Documento assinado digitalmente
Antonio Augusto Medeiros Frohlich
Data: 08/01/2021 16:12:31-0300
CPF: 500.696.400-68

Prof. Antônio A. M. Fröhlich, PhD.
Orientador

Florianópolis, 2020.

This work is dedicated to my dear parents, Cezar and Ivonete, and my dear sister Munick.

**ACKNOWLEDGEMENTS**

First, I would like to thank my Father, Cezar Nilo Hoffmann, and my Mother, Ivonete Terezinha Conradi Hoffmann, my sister, Munick Conradi Hoffmann, and my brother in law and great friend, Leonardo do Nascimento, for all the emotional support and encouragement provided.

I would like to thank my advisor Antônio Augusto Medeiros Fröhlich, for the scientific support and helpful discussions that lead to this work conclusion and for the opportunity to be a member of LISHA. I also would like to thank my dear friend, Leonardo Passig Horstmann, for the scientific discussions that helped me during the development of this work and the encouraging words. Many thanks also to all colleagues at LISHA.

I am also grateful to Prof. Rafael de Santiago, Prof. Marcus Völp, and Prof. Nikil Dutt for serving on my thesis committee.

Last but not least, many thanks to CAPES for financial support during two years in the graduate program.

## ABSTRACT

Modern multicore processors must combine a large variety of architectural features to cope with the growing demands imposed by applications, often featuring heterogeneous cores, SIMD units, and application-specific accelerators, all interconnected by Network-on-Chip (NoC) technology. Such extreme variability usually requires some level of self-adaptation to attain the expected performance of real-time scenarios while managing energy consumption to fit with the constraints imposed by Embedded Environments. In this way, this work combines a careful Non-intrusive Monitoring and a Non-Intrusive energy optimizer design supported by Machine Learning techniques to enable safe actuation by learning the system demands using the data such systems produce as they operate. The Monitoring design proposed abstracts the available sensors and actuators in such platforms through a lean, architecture-independent API. The implementation focuses on non-intrusiveness, with a measured overhead of at most 0.0718% and maximum added jitter of less than 40$\mu s$. The Non-intrusive Monitoring design is then used to explore the architectural behavior impact over performance aspects in a multicore platform, enabling Machine Learning techniques to build awareness of the current performance demands and the impact of the Dynamic Voltage-Frequency Scaling regimen on critical tasks, thus, guiding the energy optimizer, voting and actuation. To enable adaptation to the system variability, the energy optimizer encompasses Online Learning capabilities, implemented through an Artificial Neural Network design. The proposed ANN model is supported by feature selection, which provides the most relevant variables to describe shared resource contention in the selected multicore architecture. They are used at runtime to produce a performance trace that encompasses sufficient information for the ANN model to predict the impact of a frequency change on the performance of tasks. A migration heuristic encompassing a weighted activity vector is combined with the ANN model to dynamically adjust frequencies and also to trigger task migrations among cores, enabling further optimization by solving resource contentions and balancing the load among cores. The proposed solution achieved energy-savings of 24.97% on average when compared to the run-to-halt approach, and it did it without compromising the criticality of any single task. The overhead incurred in terms of the execution time was 0.1791% on average. Each prediction added 15.3585$\mu s$ on average, and each retraining cycle triggered at frequency adjustments was never larger than 100$\mu s$.
**Keywords**: Real-time systems and embedded systems, Multi-core/single-chip multiprocessors, Scheduling and task partitioning, Machine learning, Energy-aware systems.

## RESUMO

Os processadores multicore modernos devem combinar uma grande variedade de recursos arquitetônicos para lidar com as demandas crescentes impostas pelos aplicativos, muitas vezes apresentando núcleos heterogêneos, unidades SIMD e aceleradores específicos do aplicativo, todos interconectados pela tecnologia *Network-on-Chip (NoC)*. Essa extrema variabilidade geralmente requer algum nível de autoadaptação para suprir o desempenho esperado de cenários em tempo real enquanto gerencia o consumo de energia para se ajustar às restrições impostas pelos ambientes embarcados. Desta forma, este trabalho combina um monitoramento não intrusivo cuidadoso e um otimizador de energia não intrusivo apoiado por técnicas de aprendizado de máquina para permitir uma atuação segura, aprendendo as demandas do sistema usando os próprios dados que esses sistemas produzem enquanto operam. O monitoramento proposto abstrai os sensores e atuadores disponíveis em tais plataformas por meio de uma API enxuta e independente de arquitetura. A implementação é focada na não intrusão, com um overhead medido de no máximo 0.0718% e jitter adicionado máximo inferior a $40\mu s$. O design de monitoramento não intrusivo é então usado para explorar o impacto do comportamento arquitetural sobre os aspectos de desempenho em uma plataforma multicore, permitindo que as técnicas de aprendizado de máquina criem consciência das demandas de desempenho atuais e do impacto do regime Dynamic Voltage-Frequency Scaling em tarefas críticas, orientando assim o votação e atuação do otimizador de energia. Além disso, para permitir a adaptação à variabilidade do sistema, o otimizador de energia engloba recursos de Aprendizado Online, implementado por meio de um design de uma Artificial Neural Network. O modelo ANN proposto é apoiado pela seleção de *features*, resultando nas variáveis mais relevantes para descrever a contenção de recursos compartilhados na arquitetura multicore selecionada. As variáveis são usadas em tempo de execução para produzir um *trace* de desempenho que engloba informações suficientes para o modelo ANN prever o impacto de uma mudança de frequência no desempenho das tarefas. Uma heurística de migração que abrange um vetor de atividade ponderada é combinada com o modelo ANN para ajustar dinamicamente as frequências e também para acionar migrações de tarefas entre os núcleos, permitindo uma otimização adicional resolvendo contenções de recursos e balanceando a carga entre os núcleos. A solução proposta alcançou economia de energia média de 24,97% quando comparada com a abordagem *run-to-halt*, e sem comprometer a criticidade de nenhuma tarefa. O overhead incorrido em termos de tempo de execução foi de 0.1791% em média. Cada previsão adiciona 15.3585 $\mu s$ em média e cada ciclo de retreinamento disparado nos ajustes de frequência nunca foi maior que 100 $\mu s$.

**Palavras-chave**: Sistemas de Tempo-Real e Sistemas Embarcados, processadores Multi-core/single-chip, Escalonamento e Migração de tarefas, Aprendizado de Máquina, Sistemas conscientes de energia.

## RESUMO EXPANDIDO

### Introdução

Em sistemas embarcados a expansão das funcionalidades exigidas por novos aplicativos (SARMA et al., 2015), particularmente para ambientes críticos, como aqueles que lidam com a visão computacional em veículos autônomos, aumenta as demandas de desempenho da plataforma embarcada destino para que seja capaz lidar com as restrições de tempo. No entanto, aumentar o desempenho de uma plataforma embarcada também aumenta seu consumo de energia, o que é um desafio considerando as restrições de energia de tais ambientes. Aumentar o número de núcleos nessas plataformas é uma solução para acomodar as crescentes demandas de tais aplicações. Processadores embarcados multicore modernos combinam uma grande variedade de recursos arquitetônicos, incluindo núcleos heterogêneos, unidades SIMD e aceleradores específicos de aplicação, estes interconectados por uma tecnologia Network on Chip. Todavia, essas arquiteturas complexas fazem uso intenso de paralelismo e mecanismos de ocultação de latência que causam variações no desempenho das tarefas em execução (CRAEYNEST et al., 2012; KEDAR et al., 2017).

Essas plataformas, no entanto, são sistemas ciberfísicos altamente instrumentados que podem ser monitorados e controlados com base nos dados que produzem durante a operação (MÜCK et al., 2018). Os contadores de desempenho de hardware (HPC) foram introduzidos por fabricantes de chips de processador por meio do componente Performance Monitoring Unit (PMU). PMUs são usadas para amostrar ocorrências de eventos de desempenho durante a execução, fornecendo informações sobre o uso da arquitetura, como Taxa de Instruções Finalizadas, Acesso à Memória, Branches Executados e Ciclos de Estagnação. Assim, para fornecer um nível mais alto de confiabilidade, garantir a exatidão do tempo de tarefas críticas e economizar energia, conjuntos de tarefas específicas requerem que os aspectos arquitetônicos sejam constantemente avaliados para atingir o nível esperado de determinismo, avaliando as necessidades imediatas de recursos da execução das tarefas a fim de prever o impacto das medidas de economia de energia em suas restrições em tempo real.

No entanto, lidar com a otimização de energia em cenários críticos em plataformas multicore vai além de modelar o desempenho do sistema manualmente e garantir estimativas Worst Case-Execution Time. A complexidade de novas aplicações críticas, com várias tarefas e vários núcleos, como é o caso dos veículos autônomos, exige controle de desempenho, temperatura e consumo de energia de forma mais automatizada. O problema de encontrar uma configuração ideal, que forneça desempenho suficiente com o menor consumo de energia possível, geralmente requer conhecimento especialista do conjunto de tarefas e da plataforma. Nesse sentido, o Aprendizado de Máquina

(ML) se destaca como uma alternativa para automatizar o processo de aquisição do conhecimento necessário sobre os fenômenos arquitetônicos expressos por rastreamentos de desempenho e estatísticas de Sistemas Operacionais (OS).

Técnicas de aprendizado de máquina podem ser utilizadas para automatizar o processo de mapear rastros da execução de tarefas obtidas através do uso arquitetural em informações úteis para suportar otimização de tempo de execução. Por exemplo, em (LAHIRI et al., 2007; JUNG; PEDRAM, 2010) os autores usam modelos de Artificial Neural Network (ANN) focando em estatísticas de tempo do conjuntos de tarefas para otimizar a frequência da CPU. Outros trabalhos exploram modelos de Reinforcement Learning para encontrar a melhor configuração Dynamic Voltage-Frequency Scaling com base em características temporais (ISLAM et al., 2018) e a contagem de ciclos de processamento (DAS et al., 2015). Outros trabalhos (CHEN et al., 2018; JUNG; PEDRAM, 2010; RAI et al., 2010) combinam características temporais e contadores de desempenho, como a taxa de instruções finalizadas e a taxa de acessos a elementos-chave na hierarquia de memória para fornecer informações mais detalhadas para algoritmos de aprendizado supervisionado a fim de prever o impacto das operações de economia de energia no desempenho. No entanto, a realização de otimizações baseadas em aprendizado de máquina em tempo de execução junto com a execução de tarefas críticas requer que otimizadores garantam o mesmo nível de determinismo temporal definido para as tarefas. Portanto, eles devem coletar dados e executar algoritmos sem interferir nas tarefas em execução, e as otimizações propostas, que podem incluir ajustes de frequência e migrações de tarefas, não devem prejudicar a execução de nenhuma tarefa crítica.

Portanto, para que otimizações energéticas em sistemas embarcados em tempo real sejam suportadas em uma plataforma multicore, eles devem ser capazes de construir autoconsciência da utilização atual e de garantir otimizações de forma a não prejudicar a execução das tarefas, permitindo que o sistema alcance o equilíbrio de desempenho/potência esperado sem prejudicar nenhuma tarefa crítica. Combinando monitoramento de desempenho não intrusivo e técnicas de aprendizado de máquina, tal sistema deve ser capaz de aprender um comportamento de conjunto de tarefas, gerando indicadores em tempo de execução para o sistema se adaptar. Assim, um sistema embarcado multicore em tempo real pode executar a atuação guiada por aprendizado de máquina seguindo uma margem de segurança de reserva de recursos, permitindo a execução de forma segura de medidas de economia de energia (por exemplo, ajuste de frequência, migração de tarefa e gerenciamento de energia) para atingir o nível exigido de desempenho enquanto provê economia de energia.

**Objetivos**
Considerando em sistemas embarcados multicore em tempo real os desafios acima mencionadas e as capacidades de aprendizagem das técnicas de aprendizado de máquina, o objetivo principal deste trabalho é definido como **para projetar, implementar e validar um otimizador de energia de tempo de execução não intrusivo, baseado em ANN que abrange a migração de tarefas e Dynamic Voltage-Frequency Scaling.** O otimizador de energia é construído sobre o monitoramento de desempenho, usando a ANN como um preditor para orientar a economia de energia sem prejudicar a execução de qualquer tarefa crítica. Isto é feito por meio dos seguintes objetivos específicos:

- Realizar uma revisão do estado da arte sobre os conceitos de Tempo Real e Aprendizado de Máquina, com foco em plataformas multicore em tempo real e Aprendizado de Máquina aplicado ao gerenciamento de energia.
- Projetar um sistema de monitoramento de desempenho não intrusivo e implementá-lo para as arquiteturas Intel e ARM.
- Projetar uma coleção de rastros de uso arquitetural por meio de conjuntos de tarefas em tempo real, sintéticos e específicos à arquitetura, que incluem todos os contadores de desempenho disponíveis e estatísticas do sistema operacional, que também permita a análise de contenção de recursos compartilhados.
- Projetar uma seleção de *features* de forma off-line para filtrar o conjunto de recursos mais expressivo dos dados coletados em relação ao desempenho das tarefas e contenção de recursos compartilhados.
- Projetar uma estratégia de treinamento offline para um modelo inicial de ANN de baixa intrusão para prever o impacto das medidas de economia de energia no desempenho da tarefa.
- Elaborar um treinamento incremental online não intrusivo para o modelo ANN se adaptar em tempo de execução ao conjunto de tarefas corrente.
- Projetar um algoritmo de migração de tarefas baseado no monitoramento de desempenho para equilíbrio de atividades das CPUs.
- Projetar e implementar o processo de atuação do otimizador de energia não intrusivo, baseado em ANN, permitindo que os sistemas embarcados em tempo real atinjam uma configuração de frequência e distribuição de tarefas otimizada sem prejudicar o desempenho de tarefas críticas, incluindo otimizações de distribuição de carga e prevenção de contenção de recursos compartilhados.

## Metodologia

A metodologia deste trabalho caracteriza-se como uma pesquisa aplicada de abordagem quantitativa, implementada por meio de pesquisa exploratória composta por pesquisa bibliográfica e procedimentos experimentais.

Para tal, uma coleção de dados é executada a fim de extrair dados de uma plataforma multicore embarcada de tempo real executando benchmarks sintéticos, específicos de arquitetura (VENKATA et al., 2009; GRACIOLI et al., 2019) representando aplicativos embarcados relevantes para estimular os fenômenos arquitetônicos da plataforma alvo sob várias configurações. O conjunto de dados incluí todos os contadores de desempenho disponíveis na plataforma de destino e estatísticas úteis do sistema operacional, onde a coleta é realizada através de um sistema de monitoramento não intrusivo. Por meio dos dados coletados, uma seleção de features é realizada para encontrar o conjunto de features mais relevantes que será usado para construir o preditor. O preditor utilizado neste trabalho é um regressor ANN que toma como entrada as features selecionadas e prevê a utilização de uma tarefa em uma configuração de frequência mais baixa. A ANN é inicialmente treinada offline, evitando assim problemas de *cold start* e possibilitando o ajuste de sua topologia. Além disso, o treinamento offline da ANN segue uma abordagem de aprendizado incremental, onde a métrica de avaliação de desempenho utilizada para o processo de ajuste leva em consideração a quantidade, em média, necessária de treinamentos incrementais para a ANN se adaptar a um novo cenário considerando um limite de desvio. O preditor é subseqüentemente treinado em tempo de execução sempre que a frequência é ajustada ou uma tarefa é migrada, assim, liberando-o do conjunto de tarefas sintéticas inicial. Por fim, para permitir ainda mais otimizações de energia junto com o controle de frequência, uma heurística de migração é proposta como um vetor de atividade ponderada com base nos mesmos recursos usados pela ANN.

O otimizador de energia implementado é avaliado por meio de três conjuntos de tarefas diferentes, compostos por benchmarks relevantes. A avaliação se da pela análise do overhead adicionado à execução do sistema embarcado, a precisão e adaptabilidade do modelo e a economia de energia proporcionada pela solução proposta.

## Resultados e Discussão

Os principais resultados provenientes das análises de desempenho da implementação de prova de conceito do otimizador energético proposto são os seguintes:

- O otimizador energético proposto é avaliado em questão de overhead adicionado ao sistema, impacto no tempo de execução das tarefas, adaptação em tempo de execução do preditor produzido, e economia de energia proporcionada, sendo todos as análises apresentadas provindas de uma implementação de prova de conceito sob uma plataforma multicore real, mais especificamente, um processador quadcore Cortex-A53.

- O otimizador energético proposto é avaliado inicialmente sobre três diferentes conjuntos de tarefas, sendo o primeiro um cenário mais simples, o segundo, um cenário com diferentes fases de execução das tarefas, e o terceiro, um cenário com alta taxa de contenção por recursos compartilhados sobre as tarefas executadas em paralelo.
- A análise de overhead é dividida em três etapas:
  - o sistema de monitoramento: sendo esta uma avaliação de sua implementação em um processador Intel i7-2600, apresenta um baixo impacto no tempo de ociosidade do sistema (cerca de 43 ms em 1 minuto, aproximadamente 0.0718%) e uma baixa intrusividade no tempo de ativação das tarefas, apresentado um jitter máximo menor que $40\mu s$.
  - O otimizador energético em execução: o mesmo apresenta um baixo nível de intrusão no tempo ocioso do sistema, cerca de $15\mu s$ por ativação do sistema de votação e atuação, e de no máximo $92\mu s$ para ativações que envolvem o treinamento online do preditor, onde tais ativações ocorrem de acordo com a configuração de gatilho do otimizador energético, no caso dos cenários apresentados, o hyper-período do conjunto de tarefas.
  - O impacto na execução das tarefas: tarefas de uso intenso de CPU não sofreram nenhum impacto no seu tempo de execução, tarefas de uso médio de CPU e Memória sofreram, em média, um impacto próximo a 0.036% em seu tempo de execução, e tarefas de uso intenso de memória sofreram, em média, um impacto próximo a 0.18%
- A economia de energia proporcionada pelo otimizador energético é comparada com o consumo energético dos mesmos conjuntos de tarefas executando sobre o mesmo sistema operacional de tempo real (EPOS), o qual implementa uma política run-to-halt, onde a frequência da CPU é maximizada a fim de maximizar o tempo ocioso do sistema, e também comparando os resultados obtidos em relação ao sistema operacional Linux executando o mesmo conjunto de tarefas sob três políticas de otimização de energia distintas, run-to-halt, OnDemand, e Conservative. O otimizador energético obteve sempre o melhor resultado quando comparado com os cenários citados, e o fez sem comprometer a execução de nenhuma tarefa crítica, utilizando ambas as técnicas de DVFS e migração de tarefas. Os resultados obtidos foram os seguintes: 24.97% de redução, em média, no consumo energético quando comparado ao EPOS sob a politica run-to-halt, e

68.91%, 64.70% e 65.13%, em média, quando comparado ao Linux executando sob a política run-to-halt, OnDemand, e Conservative, respectivamente.

- A avaliação do otimizador energético proposto também incluí uma análise dos desvios das predições em tempo de execução do modelo ANN proposto e os resultados do respetivo retreinamento online do modelo. Tal avaliação é demonstrado em uma análise de cenários com alta variabilidade na execução das tarefas, concluindo que o modelo é capaz de rapidamente se adaptar a novos cenários sem comprometer a execução de tarefas críticas.

O projeto do otimizador de energia foi avaliado por meio de uma implementação de prova de conceito. A implementação teve como foco o design não intrusivo e os recursos da API de monitoramento e atuação proposta neste trabalho. Quando aliado à baixa intrusão apresentada por um RTOS (Embedded Parallel Operating System (EPOS)), o design da API cria uma estrutura de um otimizador limpa e confiável com recursos poderosos de monitoramento e atuação. Com o RTOS focado no manuseio de um determinado conjunto de tarefas, o monitoramento de desempenho e o projeto de atuação podem ser controlados sem prejudicar o determinismo da execução de qualquer tarefa.

Para implementar o otimizador de energia, primeiro, um preditor de utilização é necessário e, para cumprir o design proposto, um preditor com recursos de aprendizado online é desejado. Portanto, um modelo de ANN treinado incrementalmente é proposto juntamente com a exploração de perfilamento de fenômenos arquitetônicos por meio do monitoramento de desempenho. Além disso, o projeto da ANN está focado em dois momentos de aprendizagem: Um treinamento off-line usando um conjunto de tarefas sintéticas para ajuste de arquitetura e prevenção de problemas de inicialização a frio. E o treinamento online, para tornar o modelo livre do conjunto de tarefas sintéticas usado no treinamento offline, levando em conta a variabilidade de desempenho do conjunto de tarefas, proveniente tanto das tarefas quanto dos fenômenos arquitetônicos.

A exploração de features é realizada através de uma análise de contadores de desempenho disponíveis na arquitetura, com foco em identificar os mais relevantes no que se diz a fenômenos arquiteturais relacionados ao desempenho das tarefas, especialmente problemas decorrentes da contenção de recursos compartilhados gerados pela arquitetura multicore. Com o objetivo de tornar este processo completamente independente de conhecimento especialista e análises manuais, uma abordagem de mineração de dados para seleção de features é implementada. O método proposto inclui um fluxo de trabalho de pré-processamento juntamente com a combinação de três diferentes abordagens de seleção de features. Este processo é combinado com a remoção de redundância para melhorar a cobertura do conjunto de features resultante. A cobertura do conjunto de features resultante inclui contadores de desempenho cobrindo o uso de memória e o uso da CPU de tarefas, no qual a memória constitui o principal recurso compartilhado na arquitetura alvo. As informações de desempenho extraídas por meio de tais contadores foram suficientes para construir o modelo de aprendizado de máquina previsto para controlar a atuação do DVFS e orientar a migração da tarefa, o que corroborou a eficácia da abordagem utilizada para extração de features, o conjunto de tarefas sintéticas utilizadas juntamente com o sistema de monitoramento não intrusivo e o processo de seleção de features.

**Considerações Finais**

Neste trabalho, um otimizador de energia não intrusivo baseado em uma Artificial Neural Network (ANN) para arquiteturas multicore de tempo-real embarcadas que atua em tempo de execução é proposto. O otimizador de energia é capaz de prover otimizações energéticas sem comprometer os rigorosos requisitos de tempo de tarefas críticas. Os recursos do otimizador de energia incluem Dynamic Voltage-Frequency Scaling (DVFS) e migrações de tarefas, as quais atuam com base nas saídas do modelo ANN e de uma heurística baseada no conceito de vetor de atividades ponderado. A ANN é um componente do otimizador de energia que visa fornecer previsões sobre o impacto que um ajuste da frequência de execução de uma CPU terá no desempenho da tarefa, usando contadores de desempenho como entrada. O otimizador de energia considera cada tarefa em execução em cada núcleo do processador para conceber uma atuação, compondo uma predição de tempo de ociosidade disponível e uma margem de segurança de atuação definida pelo usuário. O modelo ANN é construído sobre rastros de execução coletados de contadores de desempenho de hardware e estatísticas do sistema operacional, selecionados através de algoritmos de extração de features offline. O processo de extração de recursos visa expor as variáveis mais relevantes relacionadas ao desempenho usando conjuntos de tarefas sintéticos específicos da arquitetura. Os rastros de desempenho também são usados para construir um treinamento offline para ajustar a configuração da ANN. O preditor é então treinado em tempo de execução sempre que a frequência é ajustada, liberando-o do conjunto de tarefas sintéticas inicial. A migração da tarefa é baseada em um conceito de vetor de atividade ponderado e usa os mesmos contadores de desempenho usados pela ANN. O objetivo da migração de tarefas é reduzir a variação da atividade entre CPUs em um estilo de balanceamento de carga, incluindo migração de tarefas e trocas de tarefas entre CPUs para obter uma melhor distribuição de carga e resolver contenções sobre recursos compartilhados . Os pesos usados pelo algoritmo são traçados de antemão através da técnica de Gradiente Descendente. Com isso, os resultados obtidos pela implementação de prova de conceito demonstram que a solução foi capaz de otimizar

o consumo da plataforma alvo eficientemente sem prejudicar o desempenho de nenhuma tarefa crítica através de um solução com suporte a adaptação online do modelo de aprendizado de máquina utilizado de forma não intrusiva ao sistema.

**Palavras-chave**: Sistemas de Tempo-Real e Sistemas Embarcados, processadores Multi-core/single-chip, Escalonamento e Migração de tarefas, Aprendizado de Máquina, Sistemas conscientes de energia.

## LIST OF FIGURES

## LIST OF TABLES

# CONTENTS

# 1 INTRODUCTION

In the realm of embedded systems, the expanding functionalities required by new applications (SARMA et al., 2015), particularly for critical environments such as those handling computer vision in autonomous vehicles, increases the performance demands of the target embedded platform in order to cope with time constraints. Nevertheless, increasing the performance of an embedded platform also increases its energy consumption, which is challenging considering the energy constraints of such environments. Increasing the number of cores in those platforms is a solution to accommodate the growing application demands. Modern embedded multicore processors combine a large variety of architectural features, including heterogeneous cores, SIMD units, and application-specific accelerators interconnected by NoC technology. These complex architectures make intense use of parallelism and latency hiding mechanisms that cause variations in the performance of running tasks (CRAEYNEST et al., 2012; KEDAR et al., 2017).

These platforms, however, are themselves highly instrumented cyber-physical systems that can be monitored and controlled based on the data they produce during operation (MÜCK et al., 2018). Hardware Performance Counters (HPC) have been introduced by processor chip manufacturers through a Performance Monitoring Unit (PMU) component. PMUs are used to sample performance events occurrences during execution, providing information regarding the architectural usage, such as Committed Instructions Rate, Memory Access, Branches Executed, and Stalls Cycles. Thus, to provide a higher level of reliability, guarantee the timing correctness of critical tasks, and save energy, specific task-sets are often required to look after such architectural aspects to attain the expected level of determinism, assessing the immediate resource needs of running tasks and predicting the impact of power-saving measures on their real-time constraints.

However, handling energy optimization in critical scenarios on multicore platforms goes beyond modeling system performance manually and assuring Worst Case-Execution Time (WCET) estimations. The complexity of new critical applications, with several tasks and several cores, as is the case for autonomous vehicles, requires handling performance, temperature, and energy consumption in a more automated way. The problem of finding an optimal configuration, which provides sufficient performance with the lowest energy consumption possible, often requires expert knowledge about the task-set and the platform. In this sense, Machine Learning stands up as an alternative to automate the process of acquiring the necessary knowledge about the architectural phenomena expressed by performance traces and OS statistics.

Machine Learning (ML) techniques are being used to automate the process of mapping performance traces provided by the architecture into useful information to support runtime optimizations. For instance, works like (LAHIRI et al., 2007; JUNG; PEDRAM, 2010) use ANN models in combination with timing specs of task-sets to optimize the CPU frequency. Other works envision Reinforcement Learning (RL) models to find the best Dynamic Voltage-Frequency Scaling configuration based on timing (ISLAM et al., 2018) and cycle counting (DAS et al., 2015). Other works (CHEN et al., 2018; JUNG; PEDRAM, 2010; RAI et al., 2010), combine with timing specs performance counters, such as the rate of instructions committed and the rate of accesses to key elements in the memory hierarchy to supply more detailed information for supervised learning algorithms to predict the impact of energy-saving operations on performance. However, performing ML-based optimizations at runtime along with the execution of critical tasks requires optimizers to secure the same level of temporal determinism defined for the tasks. Therefore, they must collect data and run algorithms without interfering with running tasks, and the proposed optimizations, which may include frequency scaling and task migrations, must not impair the execution of any critical task.

Therefore, for real-time embedded systems to endure in a multicore platform, they must build self-awareness of the current performance and be able to ensure optimizations, enabling the system to reach the expected performance/power balance without impairing any critical task. Combining non-intrusive performance monitoring and ML techniques, the system is able to learn a task-set behavior, yielding indicators at runtime for the system to adapt itself. Thus, a real-time multicore embedded system can run the ML-guided actuation matching a safety margin of resource reservation, enabling power-saving measures (e.g., frequency scaling, task migration, and power management) to achieve the desired performance level while optimizing energy-savings.

## 1.1 GOALS

Considering the aforementioned constraints of real-time multicore embedded systems and the learning capabilities of ML-techniques, the main goal of this work is defined as **to design, implement and validate a non-intrusive, ANN-Based, run-time energy optimizer that encompasses task migration and DVFS.** The energy optimizer is built upon performance monitoring, using the ANN as a predictor to guide energy-savings without impairing the execution of any critical task.

A top-level depiction of the proposed system is presented in Figure 1. The ANN model takes execution traces of individual tasks as input to predict their performance in case the core where they are running is subjected to an energy-saving actuation. Traces are built at runtime out of hardware performance counters, sensors, and OS variables, selected by offline feature extraction algorithms. The predictor aims at initial offline training using a synthetic task-set to tune its configuration and avoid cold start issues, and is subsequently trained at runtime whenever the frequency is scaled or a task is migrated, thus, setting it free from the initial synthetic task-set. Besides the predictor, the optimizer makes use of a task migration heuristic that encompasses a notion of resource reservation as an additional measure to avoid impairing critical tasks.

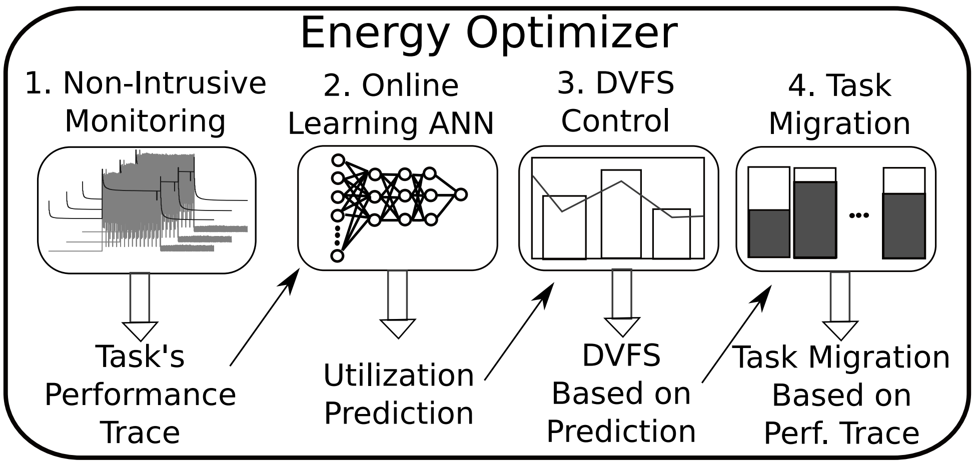


Figure 1 – Envisioned energy-optimizer solution overview.

### 1.1.1 Specific Goals

The proposed energy optimizer is designed and implemented corroborating with the following specific goals:

- Perform a state-of-the-art review over Real-Time and Machine Learning concepts, focused on real-time multicore platforms and Machine Learning applied for energy management.
- Design a non-intrusive performance monitoring system and implement it for Intel and ARM architectures.
- Design a performance trace collection through real-time, synthetic, architecture-specific task-sets, that includes every performance counter available and OS statistics, and also enables shared resource contention analysis.
- Design an offline feature selection to filter the most expressive feature-set from the collected data regarding tasks' performance and shared resource contention.
- Design an offline training strategy for an initial, low-intrusive, ANN model to predict the impact of power-saving measures at task performance.
- Design a non-intrusive online incremental training for the ANN model to adapt itself at run time to the current task-set.
- Design a task migration algorithm based on performance monitoring for activity balance of CPUs.

- Design and implement the non-intrusive, ANN-based, energy optimizer actuation process, enabling the real-time embedded systems to achieve optimal frequency configuration without impairing critical tasks, including load distribution optimizations and shared resource contention avoidance.

## 1.2 PREVIOUS RELATED WORKS BY THE GROUP

This master's project was developed at the Software/Hardware Integration Laboratory (LISHA) at the Federal University of Santa Catarina (UFSC). Over the last decades, several research projects have been conducted in the group on the topic of Real-Time Multicore Systems, Performance Monitoring, and Power Management. These works granted the group the required perspective and experience that led to the idea of the Online Learning Energy Optimizer. What follows is a list in chronological order of the main past contributions with their main authors that directly influenced the present work:

- Antônio Augusto Fröhlich, Application-Oriented Operating Systems (FRÖHLICH, A. A., 2001) - For the design of the Embedded Parallel Operating System (EPOS) and the first version of the Scheduler Framework;

- Arliones Stevert Hoeller Junior (JUNIOR, 2007), Application Oriented Power Management in Embedded Systems (free translation of *"Gerência do Consumo de Energia Dirigida pela Aplicação em Sistemas Embarcados"*) - For the perspective and notions of Power Management in embedded systems;

- Hugo Marcondes, A Hybrid Hardware and Software Component Architecture for Embedded System Design (MARCONDES, 2009) - For the real-time implementation of the Scheduler Framework;

- Gustavo Roberto Nardon Meira, Real-Time Dynamic Voltage and Frequency Scaling on EPOS System (MEIRA, 2011) - For the perspective and notions of DVFS in real-time systems;

- Giovani Gracioli, Real-Time Operating System Support for Multicore Applications; (GRACIOLI, 2014) - For the perspective and notions of real-time multicore systems, and the multicore implementation of the multicore Scheduler Design;

## 1.3 METHODOLOGY

The methodology of this work is characterized as applied research on a quantitative approach, implemented through exploratory research composed of bibliographic research and experimental procedures.

It includes the building of a database by extracting data from a real embedded multicore platform running synthetic, architecture-specific, benchmarks (VENKATA et al., 2009; GRACIOLI et al., 2019) representing embedded applications to stimulate the architectural phenomena of the target embedded multicore platforms under several configurations. The dataset will include every available performance counter in the target platform and useful OS statistics, where the collection will be done using a non-intrusive monitoring design. Through the collected, a feature selection will be performed to find the most relevant feature-set that will be used to build the predictor. The predictor used in this work is an ANN regressor that takes as input the selected features and predicts the utilization of a task in a lower frequency configuration. The ANN is initially trained offline, thus, avoiding cold start issues and enabling its topology tuning. Moreover, the ANN offline training follows a incremental learning approach, where the performance evaluation metric used for the tuning process takes into account the amount of incremental retrains that are necessary on average for the ANN to adapt to a new scenario considering a deviation threshold. The predictor is subsequently trained at runtime whenever the frequency is scaled or a task is migrated, thus, setting it free from the initial synthetic task-set. Lastly, to further enable energy optimizations alongside the frequency control, a migration heuristic is proposed as a weighted activity vector based on the very same features used by the ANN.

The implemented energy optimizer is evaluated through three different task-sets composed of relevant benchmarks, where the added overhead to the embedded system execution, model accuracy and adaptability, and energy consumption reductions achieved are measured to evaluate the proposed solution.

## 1.4 CONTRIBUTIONS

This work contributes to the Computing Science area as it proposes a novel solution for DVFS and Task migration on Multicore Real-Time Embedded Systems through a lean and non-intrusive approach that builds on Machine Learning predictors. Specifically to the aforementioned area, the following contributions are achieved by this work:

- The design, implementation, and evaluation of a Non-intrusive Monitoring API.
- The design of collection and feature selection process that accounts for architectural phenomenal that is agnostic of the target task-set.
- The usage of adaptable predictors to enable safe run-time actuation even in unforeseen scenarios, thus, not disrupting the criticality of any task in the system during actuation.

- The Energy Optimizer design that encompasses machine learning based DVFS and Task migration control at run-time on a non-intrusive design.
- The proof of concept implementation and evaluation of the proposed non-intrusive Energy optimizer.

## 1.5 OVERVIEW

Chapter 2 presents an overview of the state-of-the-art solutions for energy optimization in Multicore Scenarios, focusing on Machine Learning techniques for DVFS and Task Migration. Chapter 3 presents the design of the non-intrusive Monitor API, a lean and architecture-independent monitoring API that abstracts Performance Monitoring Unit counters, OS statistics, and Sensors. The Monitor API also provides tools for run-time actuation based on monitored data. Chapter 4 presents the energy optimizer design, including the modeling of data collection, predictor design and integration, online training and actuation, and the DVFS voting system alongside the migration heuristic proposed. Moreover, Chapter 4 also includes the design of the feature extraction proposed, the offline Machine Learning Model design and its online learning integration.

Chapter 5 presents the proof of concept implementation over a Cortex-A53 architecture, including the implementation of the data analysis, online monitoring, and actuation processes proposed in this work. The energy optimizer performance is then evaluated over three aspects: (i) Energy-savings achieved, by measuring the energy consumption on the platform used for the proof of concept implementation, and comparing the results to a run-to-halt approach and Linux's Ondemand and Conservative Power Governors; (ii) Machine learning capabilities, which includes evaluating the final DVFS and Allocation configuration achieved for each scenario and measuring the predictions deviations before and after online training; (iii) Non-intrusiveness, where the proposed solution is evaluated regarding the added overhead and the impact on tasks' execution time. Moreover, a discussion covering the complete design and implementation and the presented results concludes the Chapter 5.

The work is concluded in Chapter 6, with a recapitulation of the main contributions and the presentation of the final remarks for this work.

# 2 LITERATURE REVIEW

Real-time embedded systems manages tasks through a description of their timeliness characteristics, often represented by periodicity and deadlines. Such characteristics is used to create a notion of priority, which is a notion that varies according to scheduling policies like Earliest Deadline First (EDF), Rate Monotonic (RM), and Deadline Monotonic (DM). Moreover, by assessing a task performance demands, like its Worst Case-Execution Time (WCET) enables a offline feasibility analysis of the system scheduling. However, the intense use of parallelism and latency hiding mechanisms present in complex architectures, like modern multicore platforms, incurs into variations on the performance of running tasks (CRAEYNEST et al., 2012). Thus, offline solutions, which can only assess timing specs of tasks (i.e., Worst-Case Execution Time estimation through extensive profiling), must ensure a large resource reservation to guarantee real-time constraints. Such a large reservation ensure performance by wasting computational power and energy, requiring expensive multicore processors running at high-frequency levels to cope with the requirements imposed by only relying on offline estimations.

Several works have investigated the representativeness of PMU events monitoring on providing information of the current architectural usage, avoiding relying only on worst-case estimation. For instance, Singh et al. (SINGH et al., 2009), aiming at estimating power consumption, uses cache misses and hits, Committed Instructions rate, and Stalls. Similarly, Eyerman et al. (EYERMAN; EECKHOUT, 2010) proposes a profitability estimation of DVFS effects over a counter architecture to estimate energy consumption given a power budget. The energy consumption estimation is achieved by analyzing tasks' pipe-lined time, which is affected by DVFS, and the non-pipe-lined time, which is not. As a relevant trend, performance monitoring can lend essential information from the current task's demands. When combining meaningful performance counters with OS statistics, a broader analysis can be achieved when estimating the performance of tasks in the near future or a different configuration. In this sense, Machine Learning stands up as an alternative to automate the process of acquiring the necessary knowledge about the architectural phenomena expressed by performance traces and OS statistics, that, when provided with sufficient information, result in a reliable prediction of an outcome for actuation, and thus, enabling the definition of an actuation plan to safely optimize the Multicore Real-Time embedded system at run-time.

Controlling the power consumption of a multicore processor is an essential step to enable a balance between performance and energy. The Power Dissipation in a CMOS chip can be approximated as $P_{core} = P_{static} + P_{dynamic}$ (RABAEY; PEDRAM, 2012), where $P_{static}$ (also called leakage) is the power loss due to transistor leakage currents, and $P_{dynamic}$ is the dynamic power consumption based on the current configuration of the processor core, which can be approximated as $P_{dynamic} = CV^2f$, where $C$ is the capacitance, $V$ is the Voltage, and $f$ is the frequency of the processor core. So, DVFS comes as a powerful and widespread actuation strategy to improve energy-saving in a multicore platform, reducing the operating frequency and voltage of a core to reduce its power consumption (HSU, 2003). Therefore, by monitoring the system's behavior, it is possible to apply DVFS safely, and reach a better performance and energy ratio (DEHMELT, 2014). Similarly, Dynamic Power Management (DPM) mechanisms, implemented by the underlying OS, can also be used for optimizing energy consumption. They consist of putting hardware components in low-power states when they are idle (FRÖHLICH, A. A., 2011). Since bringing a component back to an operational state requires additional energy and time, this strategy must evaluate the overall energy-saving and performance impact before actuating. Moreover, alongside DVFS, an optimal task allocation can enhance both performance and energy-savings, by avoiding shared resource contention between tasks running in parallel, and by achieving a balanced load distribution between CPUs.

## 2.1 BACKGROUND

This section is dedicated to summarize the aspects that basis this work. It starts with a discussion of relevant Real-Time concepts, including task definition and criticality, and unicore and multicore scheduling. Next, relevant Machine Learning concepts for this work are discussed, including supervised learning and feature selection.

### 2.1.1 Real-Time Scheduling

A Real-Time system is characterized by a set of tasks with timing constraints, where the system behavior depends not only on the logical correctness of their execution but also on the time they are executed. A Real-Time task can be defined as a set of related jobs that provide some system function, where each job is an instance of the task that is scheduled according to the task requirements (LIU, J. W. S., 2000).

Real-Time tasks can be classified into three main task models (GRACIOLI, 2014): periodic (LIU, C. L.; LAYLAND, 1973), sporadic (MOK, 1983), and aperiodic. A periodic task model defines that each task $T_i \in \{T_1, T_2, ..., T_n\}$, a set of tasks $\tau$, releases a job at every period $p_i$, thus, every job $J_{i,j}$, the $j^{th}$ job of task $T_i$, has a release time $r_{i,j}$, which can be expressed as $r_{i,j} = p_i * j$. OS operations like reschedule, priority update, and dispatch, are implemented in order to handle job scheduling, where job releases can be triggered through device interrupts or timers (GRACIOLI, 2014). Sporadic tasks, on the other hand, defines the period $p_i$ of a task $T_i$ as a lower bound on job separation (MOK, 1983). A Sporadic task model can be analyzed as a periodic task model if $p_i$ is considered the minimum time interval between jobs. In the aperiodic task model, the tasks work in an event-driven fashion instead of a time-triggered one. Thus, they can be released at any time without the specification of a period or a minimum period interval (GRACIOLI, 2014).

Moreover, in this work, it is considered that a Real-Time system task can be classified into two groups regarding their timing constraints: **hard real-time** and **soft real-time**. Tasks that have a hard deadline must never miss a deadline to be considered correct, thus, representing critical scenarios where the loss of a deadline can cause catastrophic damage (i.e., loss of human lives and money). On the other hand, on tasks that have a soft deadline, the loss of a deadline is tolerable, limited by a Quality of Service (QoS) constraint, thus, representing non-critical scenarios where only the QoS is affected, like on multimedia applications. In this way, the deadline of a task is a parameter defined to represent it's correct behavior on the system and is extended to each job $j$ of the task $T_i$ according to its release time, providing an absolute deadline notion. For instance, in a Periodic task model, the deadline of the $j^{th}$ job of the task $T_i$ is defined as $d_{i,j} = r_{i,j} + d_i$, thus, if the time the $j^{th}$ job finishes its execution is $\leq d_{i,j}$, the job meet its deadline. In a hard real-time scenario, the correctness of a task can be expressed by $R_j \leq d_i$, where $R_j$ is the maximum response time of the task's jobs (BRANDENBURG, 2011).

As this work aims at handling energy optimizations on critical systems, the periodic task model is assumed, focusing on hard real-time scenarios. Thus, a complete definition of the periodic task model is defined below.

- **Task**: In a periodic task model, each task has three basic parameters $p_i, d_i, c_i$. $p_i$ is the period of $T_i$ and represents the distance in time between each job of the task $T_i$. Thus, at every $p_i$ units of time, a job of task $T_i$ is released by the OS and inserted into the Scheduling Queue. The state of the job at its release is defined by the scheduler according to the scheduling policy implemented. $d_i$ is the relative deadline of the job and represents its timing constraint. $c_i$ represents the Worst-Case Execution Time (WCET) estimation of $T_i$, and provides a notion of CPU time required by the task, where $c_i > 0$. Worth mentioning that $c_i$ is a static estimation and naturally depends on the hardware platform speed. Furthermore, this measure is affected by DVFS. For the WCET estimation to be accurate, it requires extensive profiling, especially for multicore platforms, where the parallel access to share resources highly affects a job execution time (GRACIOLI, 2014). On the other hand, $p_i$ and $d_i$ are machine-independent parameters, defined by the application design itself (BRANDENBURG, 2011).

- **Jobs**: A job is considered to be in *Ready* state when is release time $r_{i,j}$ is reached, and will be set to *Running* according to the scheduling policy implemented. In a periodic task model, a job has an absolute deadline $d_{i,j}$ that must be met for its execution to be considered correct, taking at most $c_i$ time units to run. The completion time $f_{i,j}$ defines its correctness, respecting both the task period, where $f_{i,j} \geq r_{i,j} | r_{i,j} = p_i * j, j \geq 0$, and the task relative deadline $f_{i,j} \leq d_{i.j}$. The maximum response time $R_i$ of a task $T_i$ is obtained as follows: $R_{i,j} = f_{i,j} - r_{i,j}$, $R_i = max(R_{i,j})$.

- **Deadlines**: The acceptable range of response times of a task $T_i$ is defined by the task relative deadline $d_i$. If the job completion time $f_{i,j} > d_{i,j}$, the absolute deadline of a job, it means that the job has missed a deadline. A task deadline can be defined as implicit, constrained. A implicit deadline is defined as $d_i = p_i$. A constrained deadline means that $d_i \leq p_i$. Moreover, a task-set $\tau$ is classified as implicit if every task $T_i \in \tau$ is implicit. The same is valid for constrained task-sets. If a task-set is neither implicit nor constrained, it is classified as arbitrary (GRACIOLI, 2014).

- **Utilization and Density**: $U_i$ is the utilization of task $T_i$, and is defined as $\frac{c_i}{p_i}$, representing the amount of time a periodic task keeps the processor busy (LIU, J. W. S., 2000). The utilization of the task-set can be approximated at design time as the sum of each task utilization: $U_\tau = \sum\limits_{T_i \in \tau} U_i$. Considering constrained task-sets, the density measure of a task-set is a useful information in the schedulability analysis. The density $\delta_i$ corresponds to the rate of execution of a task, and is obtained by normalizing the task utilization by its relative deadline. The density of a task $T_i$ is defined as $\delta_i = \frac{c_i}{min(d_i, p_i)}$ (GRACIOLI, 2014).

- **Deadline Definition Limitation**: In this work, as the goal is to reduce the execution frequency of a task-set to achieve energy savings while still meeting every deadline of critical tasks, it is assumed that the task-set has implicit deadlines, as the metric used for voting and actuation is the very-own task-set measured utilization.

Considering the aforementioned definitions, several scheduling policies have been defined for real-time scenarios. Those algorithms are based on the task-set parameters of $p_i, d_i, c_i$ to provide a prioritization of tasks, enabling ordering the jobs dispatching in such a way that their timing constraints are met. In this way, a design-time evaluation of the system correctness, at least when considering a static configuration (e.g., a static processor speed, and for multicore platforms, a static task allocation), is a concept presented by the literature as schedulability, where every scheduling algorithm has a respective schedulability test that aims at evaluating if the task-set is schedulable or not following its scheduling policy.

A task-set is considered **schedulable** under some policy if the scheduling algorithm is capable of providing an ordering that meets every task's jobs $d_{i,j}$. A task-set is said **feasible** if there exists a scheduling algorithm A such that $\tau$ is schedulable under

A (GRACIOLI, 2014). For instance, an easy and straightforward schedulability test is that the total utilization of a task-set must fit in the computational power available (i.e., $U_\tau \leq m$, where m is the number of processors available). Moreover, a schedulability test can be *pessimistic* and may state that a task-set is not schedulable, when in fact, it is schedulable (BRANDENBURG, 2011).

Some common assumptions for real-time scheduling algorithms are the following (BRANDENBURG, 2011):

- **Tasks are independent**: tasks do not share any kind of resources besides the processor.
- **Jobs do not self-suspend**: jobs are always ready to execute when allocated to a processor by the scheduler.
- **Jobs are preemptive**: at any time, the scheduler may replace the executing job with a higher priority job.
- **Jobs respects their periods**: jobs release are separated by their period $p_i$.
- **Run-time overhead is negligible**: the RTOS run-time overhead, such as the time for context-switch between jobs is negligible.

In this sense, some of those assumptions somehow simplify the scheduling problem, as its the case of run-time overhead being negligible and tasks being independent. OS interference caused at run-time by I/O and timer interruptions, or even the very own hardware accelerators impact (e.g., memory caches), and especially the shared resource contention present on complex multicore platforms, can create additional overhead that is complex to be modeled offline. The other assumptions are safe to be assumed in a real-time scenario, like job releases respecting their periods, self-suspension nonexistence, and jobs being preemptive.

In a non-real-time scenario, several algorithms have been proposed aiming at keeping the CPU busy at all times and deliver acceptable response times for all programs. For instance, scheduling algorithms like Round-Robin (TANENBAUM; BOS, 2015) promotes a time-slicing of the CPU between tasks in the scheduling queue. Round-Robin is a widespread scheduling algorithm that provides to the user interacting with the system a notion of multitasking even in a unicore processor, which is achieved by selecting a sufficient small slice size (called slot or quantum). Thus, at the end of a quantum, if the task has not been completed yet, the scheduler under the round-robing policy will preempt the task currently running to dispatch the next one at the head of the scheduling queue, reinserting the preempted one at the end. Priority-based algorithms, on the other hand, assigns priority to each task according to its policy and selects the one with the highest priority in the system.

In the realm of priority based-scheduling, scheduling algorithms can be classified into two types: those that assign *static* priority and those that enable *dynamic* priority assign. Real-time schedulers fit into the priority-based schedulers and can be either static or dynamic. A static algorithm relies on a more predictable scenario, where tasks priorities are assigned offline based on the task-set configuration, assuming release and execution times for all jobs in the system. The extensive offline analysis and validation available for static priority algorithms is an advantage. However, they became inflexible to variability into release and execution times (LIU, J. W. S., 2000), requiring a deterministic behavior of the execution. Dynamic schedulers, on the other hand, enables priority update at run-time at scheduling decisions. However, due to the priority changes at run-time, reaching an optimal scheduling configuration is harder. For both scenarios, a scheduling decision can be time-triggered or priority-driven, encompassing reschedule, priority updates, periodically (time-triggered), job completion, and job releases (priority-driven).

#### 2.1.1.1 Unicore Real-Time Scheduling

Unicore architectures, due to the nonexistence of parallel executions, and the shared resource contention caused by them, has a lower scheduling complexity than multicore platforms, making an offline analysis of WCET more reliable as the only effect caused by multiple tasks execution is reasonably represented by it. For instance, the WCET of a task is often its first job execution, where the data and instruction caches are loaded for the first time, or a job execution where additional overhead is generated by cache lines eviction due to preemptions or the very own sequential execution of other tasks.

In this way, unicore (or uniprocessor) real-time scheduling is a well studied and formalized concept in the literature. The first work to address unicore real-time for a periodic task model was proposed by Liu and Layland (LIU, C. L.; LAYLAND, 1973), presenting scheduling policies for fixed priority and for job-level fixed priority, a class of scheduling policies in which each job of a task can have different priorities. A priority-based scheduler in a unicore scenario can be seen as a queue manager, where the queue is sort descending. Whenever a job is released, it is inserted into the queue in a position corresponding to its priority as follows: $P(J_i) \leq P(J_{i-1} and P(J_i) \geq P(J_{i+1}$, where $J_{i,0}$ is the highest priority job and will be selected as the next job to run at the processor.

One of the fixed priority algorithms was rate monotonic (RM) (LIU, C. L.; LAYLAND, 1973), a widespread scheduling policy that assigns priority according to the task period, in inverse relation, where the task with the shortest period has the highest priority. An example of Rate Monotonic scheduling is depicted in Figure 2. In this scenario, the task-set is composed of three hard real-time tasks: *T1* is a task that must run for 2 units of time every 6, *T2* is a task that must run for 2 units of time every 8, and *T3* is a task that must run for 3 units of time every 9. This task-set presents a total utilization $U = 0.9167$ (91.67%), and by following Rate Monotonic scheduling, the highest priority task is *T1*, followed *T2* and *T3*, respectively. In this way, whenever a job of *T1* is released, it preempts any other job currently running, as it has the highest priority on the system. In the example depicted in Figure 2, at time 0 every task has a job released, and *T1* is scheduled to run for 2 units of time consecutively, as it has the highest priority. At time 2, when *T1* job finishes its execution, *T2* job is scheduled and runs until it finishes at time 4. Then, *T3* job, which requires 3 units of time for completion, is scheduled and runs for until time 6, where a new job from *T1* is released, incurring into a context switch between *T1* and *T3*, as *T1* has a highest priority. In this way, *T1* runs until it finishes at time 8, where a new job of task *T2* is released and subsequently scheduled to the CPU as it has a priority higher than *T3*. As *T2* job runs for 2 units of time, it ends at time 10, while the job of *T3* misses a deadline at time 9, as it still needed one unit of time of execution to finish. In this way, this task-set is not schedulable under the RM scheduling policy.

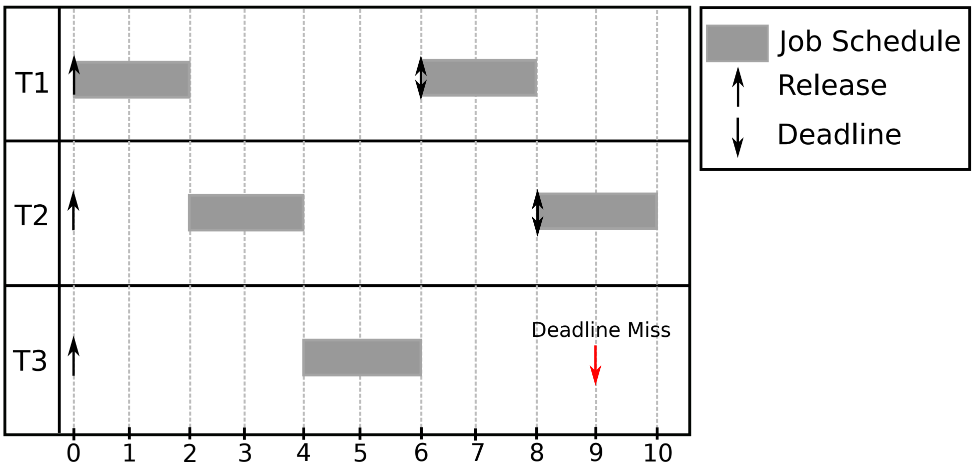


Figure 2 – Rate Monotonic Scheduling for a task-set with 91.67% of usage, presenting a deadline miss.

Liu and Layland (LIU, C. L.; LAYLAND, 1973) proposed a schedulability test for RM, where for a given task-set $\tau$ with size $n$ and tasks with implicit deadline, $\tau$ is schedulable under RM if $U_\tau \leq n(2^{1/n} - 1)$. In this sense, considering the limit of this function when $n \to \infty$, the upper bound converges to $ln(2) \approx 0.69$, restraining the utilization to such limit of 69.00%. For the example depicted in the previous paragraph, following the schedulability test proposed by Liu and Layland, the maximum utilization RM will be capable to handle, following this schedulability test, with $n = 3$ will be $3 * (2^{(1/3)} - 1) \approx 0.7798$.

Earliest Deadline First (EDF) (LIU, C. L.; LAYLAND, 1973) is the most important and widespread job-level priority fixed scheduling algorithm (GRACIOLI, 2014). On

EDF, the priority of a job of a task is given by its absolute deadline $d_{i,j}$, and similarly to RM, the priority is an inverse relation, where, as the scheduling algorithm name suggest, the task with the earliest deadline has the highest priority. Moreover, EDF has been proved to have maximum utilization of 100.00%. Thus, every periodic task-set $\tau$ with implicit deadlines is schedulable under EDF on a unicore system if and only if $U_\tau \leq 1$ (LIU, C. L.; LAYLAND, 1973). An example of the EDF scheduling with the same task-set presented in Figure 2 for the RM scheduling policy is presented for EDF in Figure 3. In this scenario, as EDF assigns priorities for jobs according to their absolute deadlines $d_{i,j}$, at time 6, when the second job of *T1* is released, its priority is assigned as 12 ($d_{i,j} = p_i * j + d_i$, with $p_i = 6$, $j = 1$, and $d_i = 6$), while T3 first job has a priority of 9 ($d_{3,0} = 9*0+9$), and finishes its execution at time 7, without missing any deadline. In this scenario, as the utilization of this task-set is 0.9167 (< 1), through the schedulability test proposed by Liu and Layland (LIU, C. L.; LAYLAND, 1973), the task-set is schedulable under EDF scheduling policy.

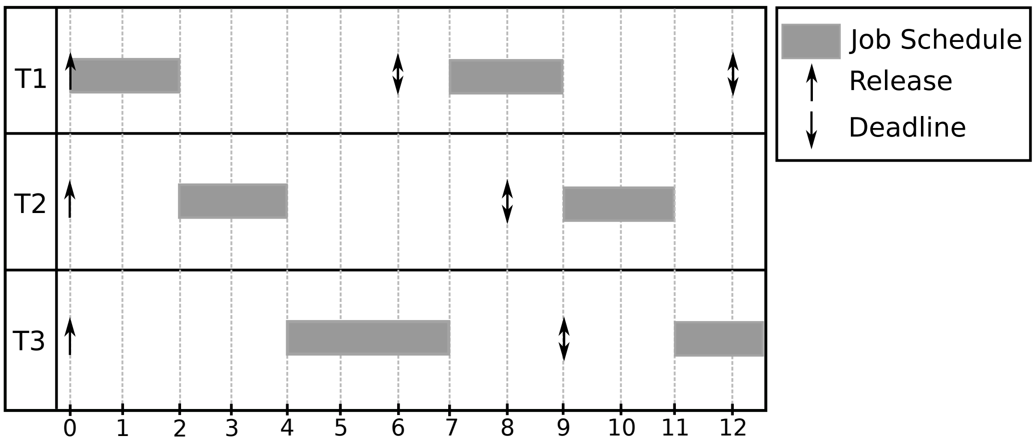


Figure 3 – Earliest Deadline First Scheduling for a task-set with 91.67% of usage, without presenting deadline misses.

### 2.1.1.2 Multicore Real-Time Scheduling

Multicore architectures create a more complex scenario, with an increase of the interference due to shared resource contention and the management of several CPUs executions. As previously explained, an offline analysis over the schedulability of a task-set assumes no interference between tasks, thus simplifying the analysis. In this way, critical application designers must take into account the variability of multicore architectures through WCET estimations and resource reservations, which requires extensive profiling to achieve a measure closer to reality, especially for complex scenarios with several tasks and several CPUs, as it is the case for autonomous vehicles handling computer vision applications.

Considering a real-time scheduler as a queue manager, which orders such a queue by using tasks (or jobs) priority, the scheduling policies described in the previous section can be extended for multicore scheduling by extending the queue design. In this way, one can reduce the problem of multicore scheduling into a set of *m* simpler unicore schedulers. The scheduling in a multicore platform can be classified into three queue configurations: Global, Partitioned, and Clustered (CARPENTER et al., 2004). On a unicore configuration, there is only one `Ready` queue and one Head element, where the first element of the queue is the second element with the highest priority on the system, currently awaiting for the resource, and the `HEAD` is the one with the highest priority, currently using the target resource (also called running or chosen()). A depiction of the Queue configuration for the unicore scenario is presented in Figure 4.

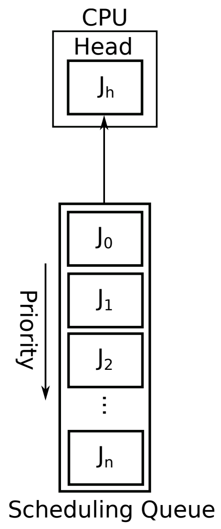


Figure 4 – Unicore Scheduling Queue.

One of the extensions for multicore scenarios is to create a partitioned configuration where every CPU in the multicore platform has its `Ready` queue and `HEAD`. This scenario fits the aforementioned configuration where the problem of multicore scheduling is reduced to a set of simpler unicore schedulers. A depiction of the Queue configuration for the multicore partitioned scenario is presented in Figure 5. However, this solution can only be applied if the problem of task allocation has been previously solved. Task allocation will be addressed later in this section.

Global scheduling, on the other hand, does not require a task allocation algorithm to take place at design time. Instead, in a Global scheduling scenario, one can increase the number of `HEAD`s a scheduling queue has, more specifically, one head for each available CPU. In this way, the *m* highest priority jobs will run in parallel. A depiction of the Queue configuration for the multicore global scenario is presented in Figure 6. In this scenario, when a new job is released with a priority higher than one of the *m* running jobs, and if the scheduling policy is preemptive, the lowest priority job in the set of running jobs will be preempted for the new job with the highest priority to run. The schedulability tests for Global scheduling depends on the scheduling policy selected. One of the disadvantages of global scheduling is that the cache affinity of a

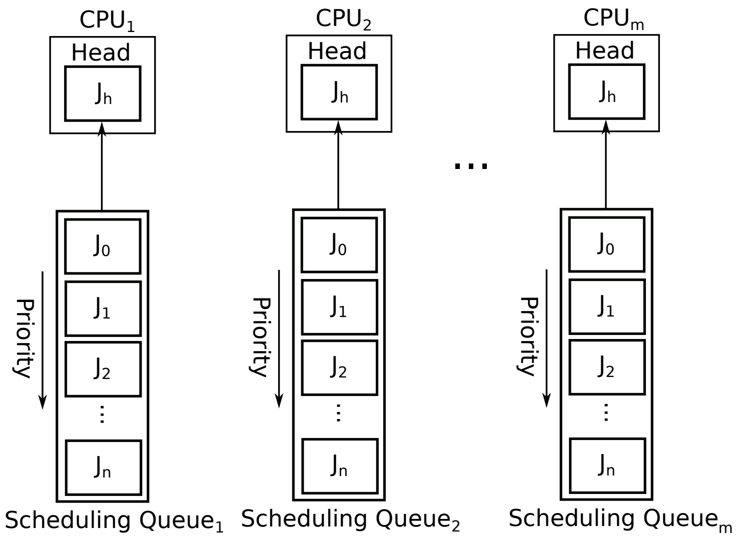


Figure 5 – Multicore Scheduling Queue - Partitioned Configuration.

preempted job that is resumed in another CPU is broken, creating variability in the job execution. The same is valid for job releases from the same task running on different tasks. Moreover, shared resource contention avoidance requires more complex handling as the jobs running in parallel can change from one task's job to another, adding more variability to the task behavior. This work focuses on partitioned scheduling, and thus, the schedulability analysis of global scheduling is out of the scope. For a review of global schedulability tests see the surveys from Davis and Burns (DAVIS; BURNS, 2011), Ismail et al. (ISMAIL et al., 2015), and Maiza et al. (MAIZA et al., 2019).

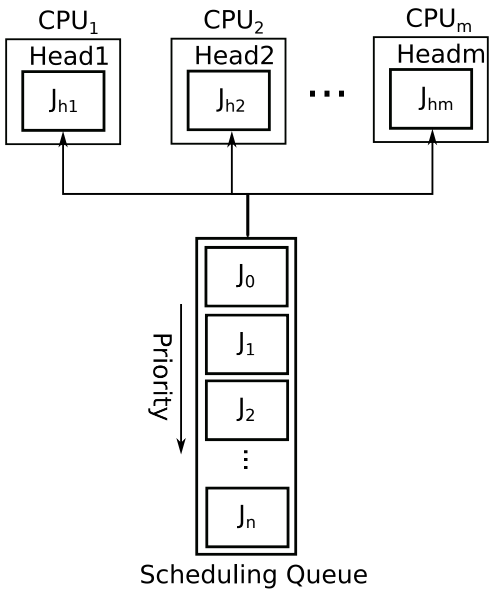


Figure 6 – Multicore Scheduling Queue - Global Configuration.

Clustered scheduling consists of a mix of both partitioned and global scheduling and is typically implemented to match the cache topology of a specific architecture (GRACIOLI; FRÖHLICH, A. A., 2015). In this scenario, a set of scheduling queues (clusters) with multiple HEADs are used for scheduling. For instance, in a multicore architecture with 8 logical CPUs and 4 physical CPUs (i.e., hyperthreading), a clustered configuration can be composed of 4 queues, representing the 4 physical CPUs, where every queue has 2 HEADs, thus encompassing the 8 CPUs. A depiction of the Queue configuration for the multicore clustered scenario with a cluster size of 2 (number of HEADs per cluster is 2, with *m*/2 clusters) is presented in Figure 7. As the clustered scheduling still requires the definition of partitions, this scheduling approach also assumes a pre-defined task allocation. The schedulability analysis for clustered scheduling has no specific test and involves the partitioned and global tests. Thus, a task-set must be partitioned through a task partitioning algorithm and pass in a global schedulability test (GRACIOLI, 2014).

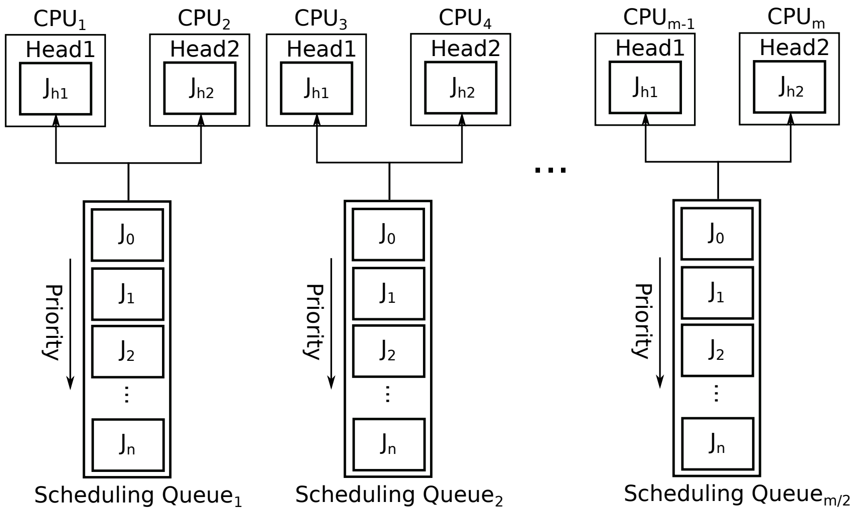


Figure 7 – Multicore Scheduling Queue - Clustered Configuration.

Many works in the realm of multicore platforms have addressed optimal task allocation and task migration. A task allocation algorithms, and also an online task migration, could be thought of as a bin-packing problem. In this scenario, each bin represents a CPU with a specific maximum capacity (e.g., 100% considering EDF schedulability bound), and a set of tasks $\tau$ with a pre-defined utilization is the set of elements that must be allocated into the available bins without breaching their utilization capacity. If there is a packing that fits such requirements, the task-set is feasible (BRANDENBURG, 2011). The best allocation is the one that better suits the system requirements. When focusing on load balancing, the search approach must aim for the lowest standard deviation between bins. When looking to reduce the number of active CPUs, the search approach must look for the final configuration that yields the highest amount of empty CPUs.

Searching for the best task allocation is a known high complexity problem (bin-packing). Several heuristics aiming at solving the bin-packing problem with a lower complexity have been explored in the literature, for instance, First-Fit, Best-Fit, and

Worst-Fit, and their respective sorted versions, First-Fit Decreasing (FFD), Best-Fit Decreasing (BFD), and Worst-Fit Decreasing (WFD). The literature of bin packing heuristic is extensive and out of the scope of this work. Thus, only the ones most relevant in the real-time community will be reviewed. Note that, as handling larger objects is more difficult (smaller ones are more likely to fit the remaining capacity), the decreasing sorting improves the performance of the heuristics, making their final results closer to the optimal solution (GRACIOLI, 2014). Worth mentioning that Worst-Fit Decreasing is one of the heuristics that achieves results more similar to the optimal (ALENAWY; AYDIN, 2005).

Whenever an element does not fit in any of the available bins, a new bin is created. This assumption is valid for each of the three aforementioned heuristics. The First-fit heuristic searches for the first bin in which the object fits into its capacity. According to Johnson (JOHNSON, 1973), the First fit heuristic requires not more than twice the number of bins of the optimal solution. The Best-fit algorithm, on the other hand, searches for the bin with minimal remaining capacity that fits the object, and thus, this heuristic is more suitable for scenarios that aim to concentrate the load into a reduced number of CPUs. Both FFD and BFD heuristics require at most 1.22 times the number of bins in the optimal solution (1.5 if the number of bins is smaller than 4) (JOHNSON, 1973). The Worst-fit heuristic does the opposite of the Best-fit, searching for the bin with the maximum remaining capacity that fits the object. Thus, the worst-fit heuristic is more suitable for scenarios that aim for a more balanced distribution (GRACIOLI, 2014). WFD improves the Worst-fit heuristic, using at most 1.25 times the number of bins used by the optimal solution (JOHNSON, 1973).

Lastly, such heuristics are limited for task-set with heavy tasks, as some task-sets cannot be partitioned as a task load cannot be divided between two or more CPUs in this scenarios (GRACIOLI, 2014). Another approach presented by the literature is Semi-Partitioned scheduling, which extends partitioned scheduling by allowing a small number of task jobs to migrate between ready queues, using a timeshare of multiple partitions, thereby improving schedulability (BASTONI et al., 2011) but possibly increasing variability due to caches and other resources affinity. Semi-Partitioned scheduling, however, is out of the scope of this work. For more details over the schedulability of semi-Partitioned scheduling algorithms, see Sheckar et al. (SHEKHAR et al., 2012) work.

### 2.1.2 Machine Learning

AI techniques have been explored over the years in several scenarios, from business intelligence to image recognition and embedded systems. They have been conceived as a way to mimic human learning using computational resources. Machine Learning methods are AI techniques concerned with automatically improve themselves with experience (MITCHELL, 1997). Moreover, to learn meaningful patterns from information previously collected and adapt its structure to achieve better results. Machine Learning methods have limitations, but for certain types of learning tasks, effective algorithms have already been developed (MITCHELL, 1997).

According to Mitchell (MITCHELL, 1997), a learning problem can be specified over three parameters: (i) The Task $T$, the goal of the learning problem. (ii) a Performance measure $P$ that defines how effective is the learning algorithm. (iii) The Experience $E$, the training experience that will be provided to the algorithm.

Mitchell (MITCHELL, 1997) presents several examples of the learning problems definitions, for instance, the problem of handwritten recognition, where $T$ is recognizing and classifying handwritten words within images, $P$ is the percent of words correctly classified, and $E$ is a database of handwritten words with given classifications. In this way, the approach selected in this work to implement the energy optimizer with machine learning revolves around the following learning problem specification:

- **Task** $T$: predict the utilization of a task in a lower frequency configuration.
- **Performance measure** $P$: the average deviation in a utilization prediction.
- **Experience** $E$: a database of performance traces of tasks in different frequency configurations with given average utilization in a lower frequency.

This learning problem can be seen as a function approximation problem, more specifically the approximation of an unknown function $F(X)$ that maps the input $X$, performance traces of task execution, into a value $Y$ that represents its performance demands in a new configuration, thus, approximating $F(X) = Y$ or $F : X \mapsto Y$ based on the input $X$ with corresponding label $Y$ (PAGANI et al., 2020). Function approximation problems can be split into two classes: classification and regression problems (NISSEN, 2007).

***Classification problems*** are problems where the output is discrete (NISSEN, 2007). Algorithms focused on Classification problems aim at classifying an input into one or more groups. For instance, an algorithm focused on Classification can learn from a labeled data-set (i.e., one that maps inputs into classes) to differentiate each class based on the patterns presented on the data-set, like decision trees (QUINLAN, 1986).

***Regression problems*** are problems where the output is a real value (NISSEN, 2007). Algorithms focused on Regression problems works in a similar way to the Classification, learning to map an input into a correspondent output, but in this case, a smooth real number. Artificial Neural Network (ANN) (MCCULLOCH; PITTS, 1943) is an example of a machine learning algorithm for Regression problems. Moreover, ANNs are not limited to Regression problems and can be used for Classification too (NISSEN, 2007).

A classification problem is often implemented by the learning algorithm as a set of binary outputs, one for each of the classes, determining if the input belongs to that class or not. Such approximation can be done more aggressively, taking into account only a few features of the input (NISSEN, 2007). On the other hand, regression problems usually require a more smooth approximation, for instance, fitting a mathematical function for real values.

Following the presented specification, it is clear that the approach selected in this work is classified as a Regression Problem and not a Classification problem. Moreover, the envisioned learning specification assumes that training samples will be coupled with a label, more specifically, with the measured utilization in a lower frequency configuration obtained in a real platform. This assumption classifies the problem as supervised training, where the learning algorithm learns the task $T$ through labeled samples $E$. Furthermore, other methods, like unsupervised learning (e.g., k-means), are focused on defining a hidden structure for unlabeled data (ISLAM et al., 2018), which does not fit our scope.

Learning the impact of performance losses, especially for non-specific solutions, is not trivial, as it lends on several aspects of task-sets and multicore architecture. Thus, due to the aforementioned complexity and variability of multicore architectures, the machine learning solution envisioned here must account for model adaptability through online learning capabilities. Thus, the Machine Learning Model can safely actuate in unexplored scenarios.

In this way, reinforcement learning comes as a suitable solution for online adaptability. RL techniques mimic the most conventional and natural learning for humans, a trial-and-error approach that learns at run-time the effectiveness of actions based on a reward function (NISSEN, 2007). In a RL solution, the system is described through a State x Action mapping, where each element is ranked based on its previous rewards. For instance, a widely-used RL implementation is Q-Learning (PAGANI et al., 2020). Given a State set $S$ and an Action set $A$, Q-Learning stores a $Q_{map}$ relating each state $s \in S$ to each action $a \in A$ with a rank, representing the effectiveness of the pair $Q\{s, a\}$. Two methods can be used for selecting an action: (i) selecting the highest ranked action, also called "winner takes it all" or (ii) randomize the selection considering a probability of selection for each action. In the first case, the learning process will start selecting at random or following a specific initial ranking, and the first mapping to present good results will have the highest rank, which could not be optimal. The second approach avoids stopping into a local optimal by applying a probability based on the rank for each action, where the actions with the highest ranks are more commonly selected, another action that may not have been tested yet has a chance to be selected and update their ranks. Such convergence towards the optimal solution is dependent on the extrapolation and exploitation process, where during extrapolation, the model must try different actions to avoid falling into a local optima (SUTTON; BARTO, 2018). Moreover, Q-Learning has three main phases at each activation: (i) account for the reward for the last action selected; (ii) update the system state; (iii) select the next action.

Nevertheless, reinforcement learning solutions suffer from cold start issues, requiring a high convergence time until it learns a new behavior. Moreover, the lack of information in the initial extrapolation phase can lead to severe damages in a critical scenario. Thus, in this work, motivated by the non-linear regression capabilities of ANNs, their capability to extrapolate to unforeseen scenarios (PAGANI et al., 2020), and its suitability to online learning (i.e., incremental training) without suffering from cold start issues, we have selected ANN as our ML model. Moreover, ANNs are reliable and widespread ML methods for performance tracing-based predictions (RAI et al., 2009; CHEN et al., 2018; YE; XU, 2012; SHEN et al., 2013; MARINAKIS et al., 2019).

#### 2.1.2.1 Artificial Neural Networks

One of the most remarkable Machine Learning techniques is Artificial Neural Networks (ANN) (MCCULLOCH; PITTS, 1943), a widely used machine learning algorithm for non-linear regression and classification (KRAWCZAK, 2013). The most used kind of ANN is the multilayer feedforward ANN (NISSEN, 2007). They are composed of layers of artificial neurons connected via connections that can only go forward between layers. ANN architectures encompass three layers: the input layer, composed of the input data, the hidden layer, which is often extended to a multilayer structure, and the output layer, which provides the final ANN result, and can be composed of multiple neurons. Figure 8 depicts an example of a multilayer feedforward ANN fully connected.

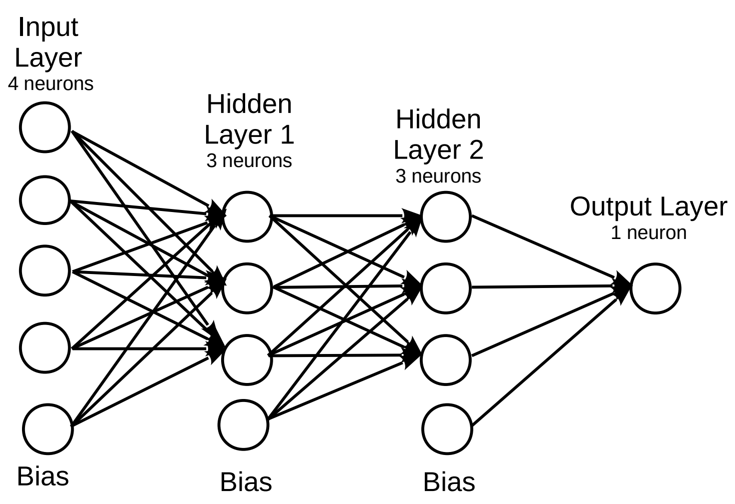


Figure 8 – A multilayer feedforward ANN fully connected with two hidden layers

Neurons are structures that assemble a set of connections $X$, a bias $b$, a weight vector $W$ composed of the weights for each incoming connection, and a specific activation function $\sigma$. Inputs are forwarded layer by layer according to the connectivity of the ANN (e.g., every value from the previous layer is forwarded to every neuron in the next layer if the ANN is fully connected). For each active connection, the neurons in the next layer assemble the incoming data and the connection weight to the bias and apply the activation function, creating a new piece of information composed of all previous data, forwarding it to the next layer. This process can be expressed as a function of the connections as represented in the following formula:

$$y(X) = \sigma(b + \sum_{i=1}^{|X|} W_i * X_i) \tag{1}$$

The activation function $\sigma(x)$ is a function that smooths the neuron output based on the sum of the input. An activation function can a threshold returning 0 or 1, or another function that output values between 0 and 1 (or -1 and 1). Some examples of widespread activation functions are threshold (i.e., $1 if x + t > 0 else 0$), Sigmoid (i.e. $\sigma(x) = \frac{1}{1+e^{-2s(x+t}}$ and Hyperbolic Tangent (i.e., $\sigma(x) = tanh(s(x + t))$, where t is a value that pushes the center of the activation function away from zero and s is a steepness parameter (NISSEN, 2007). The bias $b$ presented in (1), is a neuron that lies in every layer connected to the next layer, and never to the previous, and it always emits 1, thus, working as the $t$ parameter present in the activation functions (NISSEN, 2007). Many ANN implementations focused on reducing the model complexity (e.g., Fast Artificial Neural Network - FANN - http://leenissen.dk/fann), provide a linear approximation of activation functions, like Linear Sigmoid and Linear Hyperbolic Tangent, improving the performance for training and run-time execution at the cost of accuracy.

In this way, the knowledge acquired by an ANN through training is stored at the weights of its connections. Thus, ANN training focuses on minimizing the deviation from the output layer to the desired output, more specifically, the regression or classification error. The most used training algorithm for ANN is backpropagation (WERBOS, 1974). The backpropagation algorithm aims at propagating the error from the output layer to the input layer, updating the weights of the connections accordingly.

Two popular backpropagation strategies are Incremental and Batch training. Incremental training updates the weights at each input pattern, which provides a simpler training method and hardly sticks into local minima. However, global optimizations available with batch approaches aren't explored during Incremental training (NISSEN, 2007). In this scenario, to achieve continuous learning for online specialization to the running task-set and further optimize the model accuracy, incremental training has been selected as the backpropagation strategy, as the online training is naturally incremental learning. According to Steffen Nissen (NISSEN, 2007), the implementation of the backpropagation algorithm can be described as follows:

The input must be first propagated from the input layer to the output layer, and thus, the error for this input prediction is $e_k = d_k - y_k$, where $d_k$ is the desired output and $y_k$ is the calculated output of neuron $k$. The error value is used to calculate $\delta_k$ as follows:

$$\delta_k = e_k \sigma'(y_k) \tag{2}$$

Where $\sigma'$ is the derived activation function. Provided with $\delta_k$ the $\delta_j$ values for preceding layers can be calculated iteratively. The $\delta_j$ of the previous layer can be calculated as follows:

$$\delta_j = \lambda \sigma'(y_j) \sum_{k=0}^{K} \delta_k W_{jk} \tag{3}$$

Where K is the number of neurons in these layers, and $\lambda$ is the learning rate. The final $\Delta W$ used for adjusting the weights can be calculated using these $\delta$ values as follows:

$$\Delta W_{jk} = \delta_j y_k \tag{4}$$

and the weights are adjusted by:

$$W_{jk} = W_{jk} + \Delta W_{jk} \tag{5}$$

This process is repeated for the next inputs until the ANN error is sufficiently small, usually expressed as a threshold of the measured mean squared error of the training data. Other advanced approaches regarding batch training described in the literature, like *iRPROP* (IGEL; HÜSKEN, 2000), and *QuickPROP* (FAHLMAN, 1989), are not explored here and are out of the scope of this work.

#### 2.1.2.2 Feature Selection

Feature selection techniques can help to improve the quality of machine learning models. According to Mark A. Hall (HALL, 1999), the representation and quality of the example data are the first and foremost factor that affects the success of machine learning of a given task. Moreover, multicore platforms often present limitations regarding the number of Performance Counters that can be simultaneously monitored. To overcome this restriction and to both improve the quality of the final model, a feature selection can be performed to evaluate the expressiveness of each variable and select the most relevant ones according to the learning task (i.e., utilization prediction).

The feature selection process encompasses relevance analysis and redundancy removal, thus, reducing the dimensionality of the data-set and allowing the learning algorithm to operate faster (HALL, 1999), corroborating with the non-intrusiveness and the hardware limitations (i.e., number of PMU registers available). Statistical analysis of a data-set has been extensively explored in the literature, for instance, the correlation analysis of a data-set. Features that are correlated to the learning task are considered relevant features (GENNARI et al., 1989). Otherwise, it is considered irrelevant. Nevertheless, mutually correlated features, even though relevant to the learning task, only add redundant information to the model. Thus, a feature set must contain highly correlated features that are uncorrelated with each other (HALL, 1999). A widespread solution to measure the correlation (linear correlation) between two features or a feature and the learning task is Pearson's Correlation Coefficient (PCC), which evaluates the correlation between a pair of numerical features given a data-set containing both features (see Section 4.2.2 for more a detailed description of PCC algorithm and correlation analysis).

Feature selection techniques can be classified into three main models (BOLÓN-CANEDO et al., 2012): Filter Methods, which executes as a preprocessing step relying on the general characteristics of the training data (e.g., Pearson's Correlation Coefficient, Information Gain, and Relief). Wrapper Methods, which involves optimizing a predictor as a part of the selection process (e.g., Wrapper C4.5, and Recursive Feature Elimination). And Embedded Methods, which perform feature selection as a part of the training process and are usually specific to a learning method (e.g., Lasso and FS-Perceptron).

Moreover, there is not a so-called "best method" for feature selection (BOLÓN-CANEDO et al., 2012), as there is a multitude of feature selection algorithms, and their performance and the quality of their results, however, are strongly dependent on the data-set characteristics and particularities. Thus, more reliable and no technique-dependent solutions can be achieved by combining results from multiple feature selection methods (MOLINA et al., 2002). In this way, we aim at using multiple feature selection techniques to overcome technique specificities and acquire a more reliable merged feature-set, composed of the most expressive features for the selected architecture using runtime collected data.

## 2.2 RELATED WORKS

In the following sections, we present the literature review regarding the state-of-the-art solutions for real-time multicore scheduling and for energy and performance management in multicore platforms, focusing on the combination of machine learning and performance counter analysis. A table summarizing the main conceptual differences and advantages of the work proposed here to the most relevant state-of-art-solutions is presented at the end of this Section (Table 1).

### 2.2.1 Real-time Multicore Scheduling

This section is mostly dedicated to the aspects of real-time multicore scheduling that are more directly connected to the contributions of this work, task migration in particular.

In this work, to avoid unnecessary migrations and increase the cache-affinity of tasks, partitioned scheduling is selected as a scheduling scheme. As the design proposed here also accounts for DVFS actuation, which impacts the execution time of tasks based on its CPU-boundness, the original allocation of tasks can become sub-optimal. Thus, the approach breaches the partitioned scheduling regimen in a controlled manner looking forward to improve the current system configuration. Furthermore, the migration strategy proposed here does not account for job partitioning, as presented in the Semi-Partitioned regimen. A full description of the migration algorithm proposed is depicted in Section 4.1.3.1.

### 2.2.2 Contention-aware Task Allocation

Based on the classical best-fit bin-packing algorithm, Synchronized Partitioning Algorithm (SPA) (LAKSHMANAN et al., 2009), Blocking-Aware Partitioning Algorithm (BPA) (NEMATI et al., 2010), and Resource Oriented partitioning (ROP) (YANG et al., 2019), are three popular approaches presented in the literature for static task allocation (AKRAM et al., 2019). Both SPA and BPA are priority based algorithms, where SPA extends best-fit decreasing bin-packing algorithm, and BPA uses a heuristic that weights tasks based on their utilization and then calculate cost of remote blocking based on priority ceiling protocols. Both approaches group tasks sharing resources into macro tasks. These macro-tasks are allocated on an available core based on their utilization. Macro-tasks that are unable to fit entirely on a CPU are split into two parts and are allocated accordingly. In BPA, a task attraction is calculated for each pair of tasks, and the least attracted tasks are the ones split. ROP approach focuses on split tasks according to its phase. A phase defines the periods a task accesses shared resources. Task phases that access shared resources follow a priority ceiling protocol. Otherwise, it follows a first-fit bin-packing algorithm. SRTA (AKRAM et al., 2019) algorithm differs from SPA and BPA by considering inter-resource affinity of the shared resources on tasks with multiple shared resources, and the used split algorithm is based on the duration of shared resource sections. In a Page-coloring approach, Cache-Aware task Partitioning (CAP) (GRACIOLI; FROHLICH, 2014) show that grouping tasks that share the same cache line (same color) within the same core can increase the performance and reduce inter-core interference. After grouping tasks that share at least one color, CAP order groups by utilization and apply WFD partitioning, assuming the summed utilization of tasks that share a color is lesser than 100% and that the sum of the memory required by those tasks does not exceed a partition. However, such solutions assume prior knowledge of tasks' behavior.

### 2.2.3 Task Migration

Considering energy constraints, one can combine frequency scaling and migrations to achieve better results. In this way, task migrations aiming at load balance can be useful to overcome hardware limitations like DVFS domain restrictions in homogeneous scenarios. The usage of OS variables, like the idle time of CPUs and the execution time of tasks, has been explored in in heterogeneous scenarios too. For instance, Kim et al. (KIM et al., 2014) propose an optimization to Linux's Load Balance algorithm focused on big.LITTLE architectures. By monitoring the CPU idle time along with task's priority and utilization, they estimate the utilization in the near future, and evaluate the benefits of migrations from big to LITTLE core, and vice-versa, regarding their current execution frequency, which is controlled by the OnDemand Linux Power Governor. In an offline analysis of the target task-set, Rupaneti et al. (RUPANETTI; SALAMY, 2019) proposes an energy-efficient scheduling scheme, with an offline analysis consisting of a first-fit algorithm that prioritizes core shutdown before DVFS and a Genetic Algorithms (GA) to look for useful migrations assuming a semi-partitioned scheduling. Therefore, the GA evaluates a possible migration regarding its energy consumption based on an estimated power model. It considers that a task $T_i$ has a migratable portion $\delta_i$, and builds the initial set of possible configurations starting from the first-fit algorithm, allocating the migratable portion of tasks into randomly select processors and randomly selected speed rates. The algorithm assumes a EDF scheduling and discards configurations where the Utilization of the target CPU is > 1. In their solution, the scheduling limit of a CPU is assumed to be a ratio between the current utilization and the CPU frequency. Moreover, the frequency scaling assumed affects only the migratable portions of a task, where every migration and scaling are established offline. In complex scenarios, like modern multicore critical systems, using only timing characteristics of a task to estimate its performance in a different architecture requires extensive profiling of the task-set. The very-own memory hierarchy, I/O, and Bus Access priority and configurations can create performance variability. In this case, the techniques the authors proposes would require indicators of performance loss/gain based on the task's characteristic to better estimate migration profitability.

In a non-ML approach, but combining OS statistics and performance counters, Merkel et al. (MERKEL et al., 2010) proposes a co-scheduling approach considering shared resources contention, implemented over Linux. The approach bases on monitoring Memory Bus Access, L2 Cache Access, and Committed Instructions rate to build an activity vector for each task. Their heuristic uses task migration aiming at increasing the activity vector variance between cores, subsequently reducing the amount of shared resource contention. Moreover, they merge the information acquired via the activity vector to co-schedule tasks with different activities, a proposal with high computational complexity according to the number of CPUs and a variety of tasks.

Other works covering task migration while focusing on energy optimization, like Run-DMC (MÜCK et al., 2015) and SPARTA (DONYANAVARD et al., 2016), are covered in the next section.

## 2.3 MACHINE LEARNING FOR ENERGY OPTIMIZATION IN MULTICORE SYSTEMS

This section is mostly dedicated to the aspects of multicore systems energy optimizations guided by Machine Learning Techniques that are more directly connected to the contributions of this work, performance prediction in particular.

### 2.3.1 Supervised Learning approaches for Energy Optimization

Many works have explored various supervised learning techniques to achieve an optimal frequency configuration. For instance, Lahiri et al. (LAHIRI et al., 2007) present a ANN model for frequency prediction on multicore environments, reducing energy consumption up to 19% on video decoding benchmarks. The authors recognize the importance of a performance counter analysis along with timing characteristics to obtain a more refined model. However, they do not explore performance counters in their ANN design. Similar to the approach proposed here, their solution accounts for incremental training of the ANN, which promotes online learning, but they did not evaluate such property. Their model is limited to OS statistics, namely average idle time, current frequency, task periodicity, and two task's ID, one from the preempted task, and one from the task will be dispatched. The two tasks ID are used to represent the different behaviors regarding the scheduling order in a CPU, however, this feature still ignores the parallel execution impact. Nevertheless, only relying on the ID to depict the performance impact of the sequential execution limits the model to a single behavior expectation to each task, as only the task ID does not provide enough information when considering the natural variability of multicore platforms and the very own variability of tasks' behavior. The approach proposed in this work, the energy optimizer design, accounts for an analysis of performance counters to build a trace that depicts the current performance of a task in more detail.

Jung et al. (JUNG; PEDRAM, 2010), on the other hand, present a Learning-Based Power Management Framework using a Bayesian classifier for DVFS control. Their classifier predicts the output class probability based on a set of discretized input, where the classifier performance relies on the extensiveness of the training set to map the probability of each output based on the input. The possible output classes encompass three discretized levels of power dissipation and execution time, which are predicted based on task type, queue occupancy, and job release rate. Given the prediction for the arriving task at a CPU, a state-action mapping is evaluated to provide

$$Performance = 1 - PF_{loss} = 1 - \left(\frac{T_{fn} - T_{fmax}}{T_{fmax}}\right) \tag{6}$$

*Equation describing the performance of a task extracted from (CHEN et al., 2018).*

the cheapest action. However, the paper does not address real-time scenarios and relies on the user to define the task type. In their approach, a core runs faster when high-priority tasks with medium or high job release rates arrive under low or medium occupancy and slower for low-priority tasks. Other than the real-time support, the work proposed here also accounts for performance counters to describe in more details a task behavior in such complex scenarios, addressing task migration and frequency control.

Focusing on performance counters to achieve a more detailed performance description for an offline learning DVFS control, Chen et al. (CHEN et al., 2018) uses a Counter Propagation Network (CPN) to predict the best CPU frequency regarding energy consumption and a user-defined performance factor for non-real-time multicore scenarios. Their actuation includes three levels of configuration: (i) per-chip DVFS, where the DVFS affects all the CPUs in a chip; (ii) per-core DVFS, where the DVFS decisions are taken individually; (iii) Cluster DVFS, where the DVFS decision affects a set of CPUs. Similarly, the actuation design proposed here handles different DVFS domains through a voting system, where the CPUs that share a DVFS domain must accord an actuation decision for it to be executed, thus, controlling the actuation with no restriction to the number of CPUs that compose different DVFS domains in the same chip.

Chen et al. (CHEN et al., 2018) approach uses Committed Instructions, Cache misses, and a user-defined performance score. As it does not address real-time, the performance score is a computational performance factor based on a relation of instructions executed and memory access performed by the application in the highest frequency. In simple terms, the author defines a measure of performance of a task as: $T_{fmax}$ is the execution time of the task at the maximum frequency available, where by definition, the task has the highest performance, and $T_{fn}$ is the execution time of the task at a given frequency. As memory usage usually is not affected by frequency scaling, there is a portion of the execution time of a task that is not affected by the CPU frequency. In this sense, to obtain a more accurate measure of performance loss, the authors propose a performance score $\mu$, ranging from 0%-100%, by approximating the weights of Committed Instructions Count and Memory Access as $\mu = \rho * N_{instr} + \sigma * N_{mem}$. Therefore, the weights were approximated by profiling the target platform over a given benchmark, and they observed that for some tests, a performance score of 100% was achieved in lower frequencies, depending on the task memory access time.

From data sampled from a benchmark running on a real multicore platform, the authors couple each sample with both a performance score input and a profiled output frequency to train the CPN model offline, thus, training the CPN model to correlate the performance counters and the performance scores to the profiled output frequencies. However, a fully offline learning solution lacks adaptability, requiring extensive executions to profile the best CPN configuration for accurate and expressive offline training results. Moreover, the counter-set was limited to three counters selected by hand (Instructions and Data Cache misses and Committed Instructions). The solution proposed here, on the other hand, includes a relevance analysis of all available performance counters through feature selection, and also differs by addressing task migration in combination with DVFS. Additionally, their solution does not comprise a real-time scenario, lending to the user the definition of the optimization limit. In this work, the energy optimizer focuses on automatically finding such a limit through a combination of offline and online learning, making the solution proposed here more adaptable to new task-sets and requiring less extensive offline training.

Looking to improve the modeling of run-time and power consumption, and to avoid limiting the analysis to expert knowledge, Wu and Taylor (WU; TAYLOR, 2016) present a method to identify performance counters relevant for such models. The approach is composed of Spearman correlation and Principal Component Analysis (PCA), applied on a data-set of performance counters acquired from a real platform running a set of specific applications. In their work, the models are application-specific and are used to guide code optimizations instead of DVFS. They use non-negative multivariable regression analysis to generate the models based upon the counters relevance and CPU frequency. In this work, we address the counter selection through a combination of three different feature selection techniques. We also differ by using an ANN instead of linear regression, where the ANN model is adapted online through incremental training to fit the running task-set and address variability.

Run-DMC (MÜCK et al., 2015) is a run-time dynamic performance and power estimator for Heterogeneous Multicore platforms. Run-DMC is designed for non-real-time scenarios and is implemented over a Linux kernel. The method aims at thread-level prediction for both Instructions per Cycle (IPC) and Dynamic Power (PD) based on Least Square Method over performance counters, accounting for 10 performance counters, namely: active cycles $cy_{active}$ and Committed Instructions $I_{total}$ (i.e., used to provide IPC measurements), share of memory ($I^{t_i,c_j}_{mshare}$), branch ($I^{t_i,c_j}_{bshare}$), floating-point ($I^{t_i,c_j}_{fpshare}$), branch predictor ($mr^{t_i,c_j}_{Br}$), L1 instruction cache ($mr^{t_i,c_j}_{L1I}$), L1 data cache ($mr^{t_i,c_j}_{L1D}$), instruction TLB ($mr^{t_i,c_j}_{ITLB}$), and data TLB ($mr^{t_i,c_j}_{DTLB}$) miss rates. In this sense, the Run-DMC estimators can account for shared resources impact at thread performance, private caches, and CPU usage. However, such an approach requires expertise knowledge from the architecture as it does not provide an automate way to select the most relevant performance counters. The Run-DMC IPC and Power estimators are shown in

(7) and (8), respectively. The relation between every pair of heterogeneous core $c_l, c_j$ is depicted by the set of coefficients in the linear regression expression, thus, every combination of $c_l, c_j$ requires a individual training. Furthermore, the frequency ratio between $c_l$ and $c_j$ is included in the estimation to evaluate frequency scaling.

$$\begin{aligned} IPC_{t_i}^{c_l} = x_0 + x_1 * \frac{F^{c_l}}{F^{c_j}} + x_2 * mr_{L1I}^{t_i,c_j} + x_3 * mr_{L1D}^{t_i,c_j} + x_4 * mr_{ITLB}^{t_i,c_j} + x_5 * mr_{DTLB}^{t_i,c_j} + \\ x_6 * I_{mshare}^{t_i,c_j} + x_7 * I_{fpshare}^{t_i,c_j} + x_8 * I_{bshare}^{t_i,c_j} + x_9 * mr_{Br}^{t_i,c_j} + x_{10} * IPC^{t_i,c_j} \end{aligned} \tag{7}$$

*IPC estimator linear equation extracted from Run-DMC (MÜCK et al., 2015).*

$$\begin{aligned} PD_{t_i}^{c_l} = y_0 + y_1 * F^{c_l} * (V^{c_l})^2 + y_2 * I_{mshare}^{t_i,c_j} + \\ y_3 * I_{fpshare}^{t_i,c_j} + y_4 * I_{bshare}^{t_i,c_j} + y_5 * IPC^{t_i,c_j} \end{aligned} \tag{8}$$

*Power estimator linear equation extracted from Run-DMC (MÜCK et al., 2015).*

The Run-DMC thread mapping algorithm is implemented as a search algorithm that aims at maximizing energy efficiency by maximizing the IPS/Power ratio. The estimators train and evaluation is implemented over *gem5 simulator* (BINKERT et al., 2011) and shows an average improvement of 51% in energy consumption when compared to the unmodified Linux Kernel, and adding less than 1% of overhead to the system execution (up to 869$\mu s$ every 100$ms$ for a 16 threads scenario). As the approach does not account for online adaptability, the performance of the estimator relies on the extensiveness of the data-set used to build the predictor. In this scenario, if the training data-set does not include shared resources contention examples, the model will not be able to capture this relation at run-time and could mislead a decision-taking process.

Donyanavard et al. (DONYANAVARD et al., 2016) addresses heterogeneous manycore scenarios in a similar approach to Run-DMC and proposes SPARTA, a run-time, throughput-aware, energy-efficient task allocation system integrated to the Linux scheduler. SPARTA relies on a thread-level bin-based prediction of IPS and Power consumption. The predictions are taken based on performance counters sampled at the same rate as the Linux Scheduler. SPARTA accounts for a correlation analysis of performance counters to IPS, using Pearson's Correlation Coefficient. According to this analysis, memory-boundness and compute-boundness metrics are built to implement the bin-based predictor. In their implementation, the memory-boundness *mb* of a task is given by the miss rates of the first and second level of caches ($mb = mr_{L1I} + mr_{L1D} + mr_{L2}$), while the compute-boundness is represented in two levels: The first, $cb_1$, is given by branch misprediction ($cb_1 = mr_{Br}$), and the second level is the sampled IPS ($cb_2 = mr_{IPS}$).

A bin-based predictor is created for all combinations of core types and frequencies. The predictor training is done by creating bin-layers for each metric ($mb, cb_1$, and $cb_2$), where at each layer, equally sized bins are created, splitting the training data between the bins. At the last layer, the average value of each bin is taken as the final prediction. The process is similar to a decision tree, but the tree architecture is built manually by the authors. Even though supported by correlation analysis, the process of defining the number of layers, and the metrics $mb$, $cb_1$, and $cb_2$ used at the respective layers, is done manually, and can vary depending on the platform specification. Different from Run-DMC, SPARTA assumes a target throughput for each task and limits the optimization problem to the set of core configurations that achieve the throughput demands of the task. The allocation algorithm prioritizes tasks with a partial subset of cores, followed by unachievable target throughput, and then, achievable target throughput in all cores. Tasks with achievable target throughput are allocated to a core with maximum IPS/Watt that achieves target throughput. Tasks with unachievable target throughput are allocated to cores that maximize throughput. The complexity of the allocation algorithm is bound by $\mathcal{O}(|T| * |C|)$, where T is the task-set, and C is the core-set. SPARTA focuses on task migration and relies on Linux Power Governor to handle frequency scaling. Due to the lack of runt-time adaptability, the modeled bin-based predictor quality relies on the expressiveness of the training data-set. The energy optimizer proposed here, on the other hand, uses the Machine Learning model with online learning support to guide the DVFS actuation through a prediction of task utilization in a lower frequency configuration, while relying on linear regression to weight the activity vector used for task migration.

### 2.3.2 Reinforcement Learning approaches for Energy Optimization

Many works addressed DVFS control using RL. Islam et al. (ISLAM et al., 2018) presented an Online RL (Q-Learning implementation) to select the best configuration of voltage and frequency and also the best DVFS technique to be used regarding energy consumption. Their RL is based on Core id $c$, System Utilization $SU$ and Dynamic Slack $DS$ for the technique selection Q-Map, and Core id $c$, utilization of the scheduled task on core $\mu_c$, and the frequency of the core c $f_c$ for the Frequency selector Q-Map. Both Q-Maps use a power model to define the learning reward. To accommodate real-time constraints, a *lowest frequency controller* is required by this approach to avoid selecting a frequency that would lead to a deadline miss. The authors implement the *lowest frequency controller* based on scheduling information and the technique selected, assuming that a frequency scaling factor will have a linear effect on the execution time of tasks (i.e., if the frequency is reduced by 10% the execution time of tasks will increase by 10%). This assumption is not always true for every task-set when considering complex modern multicore architectures, for instance, when facing shared resource contention.

The proposed solution was only evaluated through simulations with an estimated power-consumption, which might not endure the variability of modern out-of-order, speculative, multicore architectures under a strong DVFS regimen. Another design issue of RL techniques described by the authors is the space complexity of the Q-Mapping. For instance, in their model, the number of State x Action pairs in the first RL model, the one focused on the technique selection, has a space complexity of $\mathcal{O}(|SU| * |DS| * |core| * N)$, where $N$ is the number of DVFS techniques available, and $|x|$ indicates the number of steps (or bins) for the values of the variable $x$. Adding performance counters to the state description can exponentially increase the size of the system Q-Map, and subsequently increase the length and complexity of the extrapolation phase (i.e., where the State x Action pairs are evaluated according to the reward function). The same is valid for the Frequency Q-Map.

Similarly, Anup Das et al. (DAS et al., 2015) modeled the RL algorithm in a global fashion by using the sum of Cycle Count from all active CPUs. They actuate with DVFS and shutting down CPUs considering global scheduling. Their solution achieves an average reduction of 22% using utilization and power consumption estimation as rewards for the learning. Biswas et al. (BISWAS et al., 2017) also presented a RL (Q-Learning implementation) for DVFS application, achieving up to 16% better results when compared with Linux on-demand governor. The authors used a Cycle Count prediction to profile the task-set and improve the state/action selection. The learning is achieved by evaluating the average scheduling slack on the CPU.

Zeppenfeld and Herkersdorf (ZEPPENFELD; HERKERSDORF, 2011) achieved energy consumption reductions using a Learning Classifier Table, a very similar approach to RL, for workload management and DVFS application in an FPGA platform implementation, using both hardware and software to control the autonomic behavior. The learning in this approach is done through a rewarding action based on its effectiveness regarding a predefined optimal system utilization measured by cycle count, that aims at reducing the slack time of the cores through DVFS and migration. The work presented here differs not only by the technique implemented but also by providing a full software solution while considering a more detailed performance definition at each decision step.

The main advantage of RL approaches when compared to supervised learning solutions is that they do not require previous knowledge from the system. However, it suffers from cold start issues, requiring a higher convergence time when compared to an already trained Supervised Learning approach, especially for scenarios with a large number of states and actions, usually requiring a heuristic to discard impossible or invalid configuration based on the problem domain (e.g., limiting the frequency scaling based on tasks utilization).

### 2.3.3 Summary of the Literature Review

Table 1 – Qualitative Comparison to Related Works Over the Main Concepts Addressed by This Work

| Work | DVFS Control | Task Migration | Real-Time | Machine Learning | PMU Counters | Contention Aware | Online Adaptability |
|---|---|---|---|---|---|---|---|
| Eyerman et al. | ✓ | | | | ✓ | | |
| Rupanetti et al. | ✓ | ✓ | ✓ | ✓ | | | |
| Kim et al. | | ✓ | | | | | |
| Merkel et al. | | ✓ | | | ✓ | ✓ | |
| Lahiri et al. | ✓ | | | ✓ | | | ✓ |
| Jung et al. | ✓ | | | ✓ | | | |
| Chen et al. | ✓ | | | ✓ | ✓ | | |
| Wu and Taylor | | | ✓ | ✓ | ✓ | ✓ | |
| Mück et al. | | ✓ | | ✓ | ✓ | ✓ | |
| Donyanavard et al. | | ✓ | | ✓ | ✓ | ✓ | |
| Islam et al. | ✓ | | ✓ | ✓ | | | ✓ |
| Das et al. | ✓ | | | ✓ | ✓ | | ✓ |
| Biswas et al. | ✓ | | | ✓ | ✓ | | ✓ |
| Zeppenfeld et al. | ✓ | ✓ | | ✓ | ✓ | | ✓ |
| This work | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Table 1 presents a qualitative comparison between the present work and the related works presented in this section, highlighting the main concepts addressed by this work, especially the criticality of the target scenario, represented by its real-time characteristics, the Online Adaptability of the Machine Learning solution, and the combination of DVFS and Task Migration to optimize the overall energy consumption of the platform. Few works in the literature comprise all of the aforementioned concepts at the same time, while the majority of related works only addresses a single optimization level, a non-real-time scenario, or a non-adaptable model. Except for the Lahiri et al. work, only Reinforcement Learning approaches address run-time adaptability, but they lack initial knowledge, suffering from cold start issues, requiring warm-up rounds. Moreover, most state-of-the-art approaches in Reinforcement learning narrows their approach to a small set of features in order to avoid the technique space and extrapolation complexity.

Lahiri et al. work comprises an incremental training of an ANN, enabling online learning. However, their model is limited to OS statistics and soft real-time scenarios, and they do not include performance counters in their analysis neither evaluate the model online training capabilities. Chen et al., on the other hand, focus on extensive offline training and on an user-defined threshold to optimize the system. They also do not address task migration and DVFS at the same time. Rupanetti et al. encompasses both techniques but relies only on an offline approach using timing characteristics, which might not endure the real platform shared resources contention, thus, requiring such knowledge to be acquired beforehand. Donyanavard et al. approach comprise DVFS optimizations along with task migration, but the DVFS control is delegated to the Linux OnDemand Power Governor, while in this work the energy optimizer itself handles the frequency scaling to guarantee real-time constraints. Mück et al. (Run-DMC) focus on a linear regression solution, while Donyanavard et al. (SPARTA) focus on a bin-based approach, both relying on extensive profiling of the architecture to create their models. Here, we explore the ANN incremental learning capabilities at run-time, setting it free from the initial synthetic task-set. Moreover, the proposed approach enable the model to account for variability without requiring extensive profiling in an offline analysis. Worth mentioning that, from the works that included performance counters into their models, Wu and Taylor and Donyanavard et al. included feature selection as one of their accomplishments, an important step to improve the final model accuracy considering that it will provide a more refined set of performance counters. However, Donyanavard et al.'s work still requires expert knowledge to build the metrics for their bin-based predictor, while in this work, the feature selection process combines three different techniques results to provide the final feature set in a process that does not require expert knowledge.

# 3 NON-INTRUSIVE MONITORING

In a multicore real-time embedded system, task-sets are meant to periodically execute specific jobs that demand an amount of performance from the processor, represented by its deadline and capacity, where capacity is commonly expressed as a worst-case scenario approximation. Such critical systems are handled by an Operating System (OS), which manages the periodic job releases by timed control loop operations like reschedule, priority update, and dispatch (for context-switches). Thus, to guarantee criticality, those OS operations must not impair the execution of any critical task for the generated overhead to be considered non-intrusive.

In this way, to build an adaptive embedded system capable of optimizing its operation, especially to become mindful of performance variations while preserving the criticality of tasks and the quality of captured data, a Non-Intrusive Monitoring API design is proposed here. The Monitoring API builds on sensors and event counters present in modern hardware platforms and on variables kept by the operating system to capture run-time data that are subsequently subjected to ML tools to produce scheduling heuristics targeting specific optimization goals. It provides non-intrusive mechanisms to collect such data while the system runs task-sets, and it does without impairing the execution of tasks. It abstracts Performance Monitoring Unit, thermal sensing, energy monitoring, and Dynamic Voltage-Frequency Scaling available on such platforms through a lean, architecture-independent API.

The Monitoring API (HORSTMANN et al., 2019) is designed as an extension of the Scheduler Framework first proposed by Fröhlich in EPOS design (FRÖHLICH, A. A., 2001), and later extended by Marcondes et al. (MARCONDES et al., 2009), and Gracioli et al. (GRACIOLI; FRÖHLICH, A. A., 2015). The Scheduler Framework is a flexible, parameterized framework that can be used to instantiate virtually any kind of scheduler[1]. Therefore, the scheduler framework is extended to include monitoring capabilities in order to provide a clean and non-intrusive Monitoring API for scheduler designers. An overview of the framework and the extensions proposed here is depicted in Figure 9.

## 3.1 SCHEDULER FRAMEWORK

The `Scheduler` parameterized class models traditional scheduling operations in a manner that is agnostic on the objects being scheduled (which are given as a parameter) and also to scheduling policies (provided by specializations of class `Criterion`). For this discussion, threads are the objects of interest, and logical CPU cores are

[1] Previous versions of the framework, without clerk and monitor, have been used to produce a variety of schedulers based on empirical knowledge about the underlying architecture, not taking into account the run-time data used in this work (MARCONDES et al., 2009; GRACIOLI; FRÖHLICH, A. A., 2015).

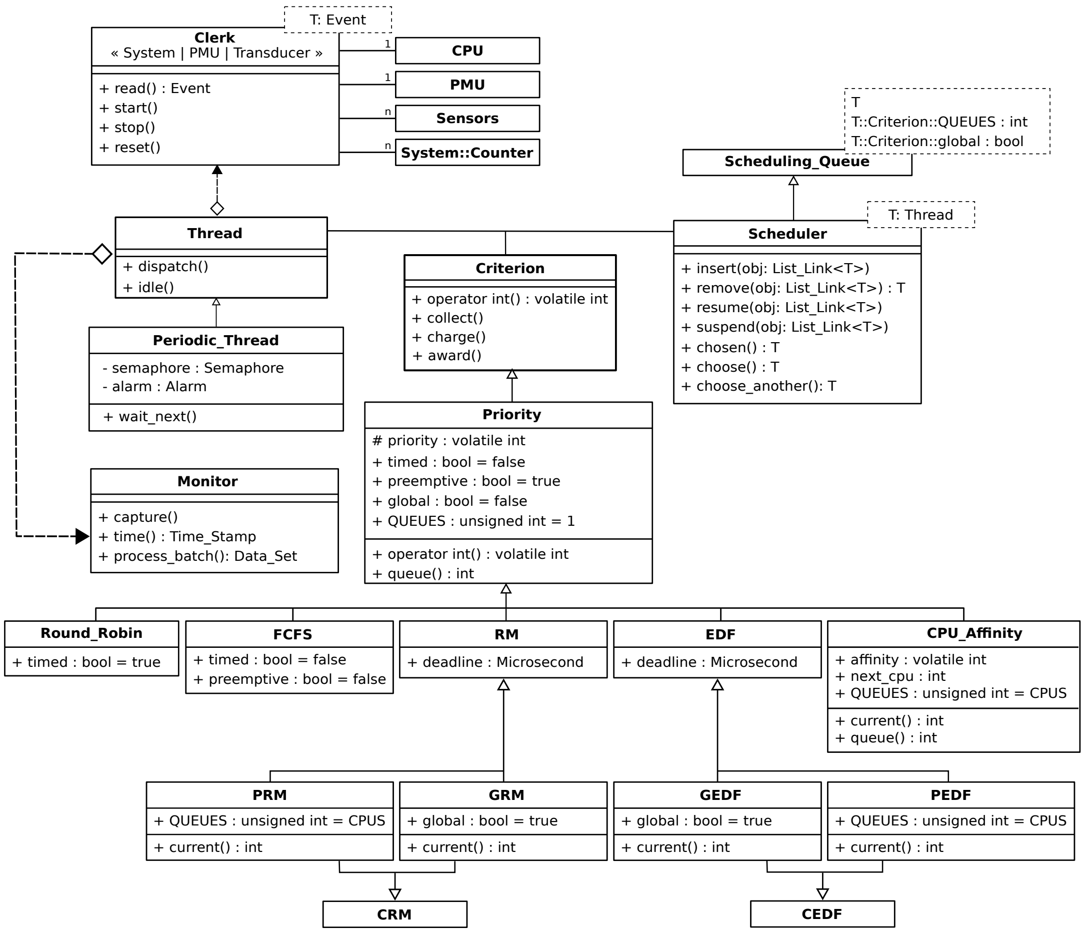


Figure 9 – Scheduler Framework UML class diagram with the Monitoring API extension.

the relevant resources, but the framework could be used to schedule disks, networks, energy, and any other kind of resource.

The `Scheduler` parameterized class owes its implementation to the powerful parameterized class `Scheduling_Queue` (GRACIOLI; FRÖHLICH, A., 2013), which, besides taking the object's type as parameter, adapts its behaviour to handle scenarios in which multiple ordered queues are needed. The `choose()` method triggers a scheduling decision, while `chosen()` remembers that decision. In this way, objects never leave the scheduler during normal operation[2]. The method `choose_another()` is used to implement operations such as `Thread::yield()`, which forces `choose()` to select an object that is not ranked highest in the queue.

Scheduling policies are modeled separately, outside the `Scheduler` class, by the `Criterion` class. This class hierarchy provides `Scheduling_Queue` with a ranking algorithm through `operator int()` so that queues are kept ordered considering the ordering of integer numbers. Scheduling policies are, therefore, implemented by providing a mapping onto such ordered set through `operator int()`.

[2] This design decision, in combination with methods `suspend()` and `resume()`, which temporarily make an object unschedulable, enables efficient hardware implementations of schedulers with High-Level Synthesis tools.

Considering Real-time Multicore scheduling, the framework groups policy-related algorithms in two classes: global and partitioned (CARPENTER et al., 2004). In a global scheduler, there is only one `Ready` queue (`Priority::QUEUES` in the figure), and the scheduler assigns the highest-ranked thread to any available core. Conversely, partitioned algorithms are individually applied to each partition, which can either correspond to a single core or to a cluster of cores that are typically grouped to match the cache's topology (CALANDRINO et al., 2007; GRACIOLI; FRÖHLICH, A. A., 2015). Each partition has its own `Ready` queue (e.g. `PRM::QUEUES` and `CEDF::QUEUES` in the figure). This design relies on the scheduling queue's ability to adjust its behavior to mimic multiple queues for partitioned algorithms and multi-head queues for global and clustered ones. A multi-head scheduling queue is a queue that has more than one chosen object at a time. For multicores, method `chosen()` will return a different thread for each core on a global scheduling policy and a different thread for each partition on a clustered scheduling policy.

In this architecture, real-time periodic threads (`Periodic_Thread` in Figure 9) are modeled by endowing aperiodic threads (`Thread`) with timing (`Alarm`) and synchronization (`Semaphore`) mechanisms. When a periodic thread is instantiated, the associated semaphore is initialized with 0, and the associated alarm is configured to trigger a `v()` operation at each period. The constructor of `Periodic_Thread` is a variadic function template and was omitted from Figure 9 for simplicity. It takes, among other parameters, the thread's period, deadline, and, in some cases, worst-case execution time. A task's job is finished by invoking the method `wait_next()`, which implicitly calls `p()` on the associated semaphore, thus causing the thread to block until the next period. Jobs are, therefore, periodically released by this alarm, which is triggered by a high-priority hardware interrupt.

Dynamic scheduling policies, like EDF, provide an `update()` method to update the object's priority every time a new job is released. This method is invoked before releasing the job (i.e., unblocking the thread) through the invocation of `v()` on the associated semaphore by the alarm, so the corresponding thread is inserted in the `Ready` queue considering the ordering defined by the updated rank (viz. priority). Scheduling criteria can also provide specific methods to account for the resources consumed by the thread that is leaving the CPU (`Criterion::charge()`) and adjust the resource budget of the thread entering the CPU (`Criterion::award()`). Figure 10 presents the classic thread state transition diagram enriched with the actions taken on behalf of the scheduler. Both `charge()` and `award()` are triggered by `Thread::dispatch()` while transitioning from Ready to Running. The other transitions shown in the figure will be discussed later in this section.

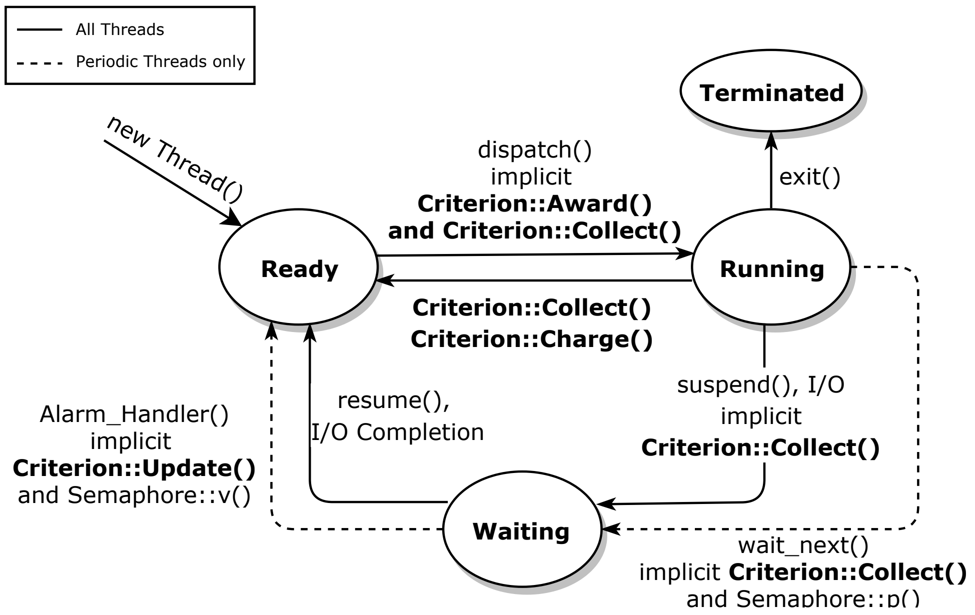


Figure 10 – Thread state diagram and the associated methods of class Criterion.

## 3.2 NON-INTRUSIVE MONITOR DESIGN

The constructs described in the previous section can be used to implement virtually any scheduling policy, including those described as feedback scheduling algorithms (STANKOVIC et al., 1999). However, some algorithms rely on data collected at run-time to dynamically adjust the rank of objects in the scheduling queues. In this context, the Monitoring API aims at collecting such data and building historical series. These series, besides being used for scheduling, can be used for profiling and Machine Learning as well.

The `Monitor` class shown in Figure 9 cooperates with class `Thread` to deliver a powerful run-time monitoring engine that can collect data from sensors, from the CPU, the PMU, and the OS, and make them available as time-series to be used during run-time or exported during idle time or at the end of the execution. The data of interest for each policy is modeled within the specialization of `Criterion`, with the `collect()` method being used to sample them. This method is implicitly invoked by `Thread` whenever a thread leaves a CPU (transitions departing from Running in Figure 10), thus, adding a little execution latency but limiting the interference on the running threads (more about the non-intrusiveness of the monitor in Section 3.2.1).

The run-time monitor gathers data sampled by `collect()` whenever `capture()` is speculatively called by the *idle* thread. Since *idle* has the lowest priority of all threads, samples gathering only happens when no other threads are ready to run and therefore has virtually no side-effect on the running system besides spoiling cache locality a bit. The method `process_batch()` is used to externalize raw data, eventually saving them to disk or sending them over the network. It can be called at the end of a task set execution or by the idle Thread on a best-effort principle during execution.

Looking forward to simplifying data acquisition, the Monitoring API framework abstracts different sources through an entity called the `Clerk`. It provides methods to configure, control, and sample variables of interest, independently of where they have been produced (if by sensors, by the CPU, by the PMU, or by the OS). Tracking an event is achieved by instantiating the parameterized class `Clerk` with the event type as a parameter and instantiating an object thereof with an event enumerator as a parameter. For instance, `Clerk<Transducer>::Clerk(CPU_TEMPERATURE)` creates a clerk to keep track of the temperature of the CPU in which it was instantiated, `Clerk<PMU>::Clerk(INSTRUCTIONS _RETIRED)` gives access to the respective PMU counter, and `Clerk<OS>::Clerk(FREE _MEMORY)` gives access to the respective OS variable. The method `read()` is used to get immediate samples of data, while `start()`, `stop()`, and `reset()` are used on a cumulative fashion. Clerks are hard to implement by platform providers since they abstract a large variety of data sources, many of which requiring specific low-level coding, but that very same code would have to be handled by anyone designing a scheduler based on that source. Clerks provide an elegant API and also guide the pursuit of such low-level coding. Figure 11 presents an example of data extracted at run-time by the monitoring API during ten seconds of execution, covering three instances of the same application running in three CPU cores of an ARM Cortex-A53 architecture. Note that, for ARM Cortex-A53, the PMU is limited to six events simultaneously monitored. Thus, two executions were required to collect the seven events presented in the figure. The monitored events are: Committed Instruction rate (b), Branches executed (c), Immediate Branches executed (d), Branch misses (e), L1 Cache Hits (f), Unaligned Loads and Stores executed (g), Data Memory Access (h), and L1 Cache Misses (i).

Due to its lean and simple design, the Clerk abstraction can be extended to any available source of information in the system. For instance, a hardware accelerator or from another processor chip like in a distributed system. However, heuristics that required a more complex sampling must account for the different sampling rates and other limitations from such sources.

### 3.2.1 Non-Intrusiveness

The primary goal of collecting run-time data from sensors, counters, and variables while designing a new scheduler is to detect patterns and produce models. If the monitoring system itself disrupts such patterns, the confidence and the reliability of the resulting models are compromised. Additionally, several hardware platforms have limitations on accessing sensors and performance counters, making run-time data acquisition even more complicated. The number of channels of a PMU, for example, limits the number of events that can be monitored simultaneously, thus requiring the same application to be executed multiple times to collect a broader, meaningful set of variables. Therefore, the Monitor API was conceived to be non-intrusive, incurring in as little overhead and interference as possible.

Assuming that the target task set does not exceed the platform's capacity, that is, its per-core utilization does not exceed 100%, we address non-intrusiveness through the following design decisions:

1. The monitoring system is initialized along with OS, before applications start, adding on booting time, but not on applications' execution time;
2. `Clerk::read()` acting on behalf of `Criterion::collect()` uses polling instead of interrupts, thus adding a little, constant latency to thread dispatching time only;
3. Data sampled by `Criterion::collect()` using `Clerk::read()` are stored in contention -free circular buffers of fixed size that are statically allocated as part of each thread's context;
4. The data collected in the circular buffers are gathered by the idle thread (using `Monitor::capture()`) when no application threads are running;
5. Data gathered by `Monitor::capture()` are packed and saved to disk or sent over the network by the idle thread on a best-effort principle, eventually after the completion of the task set execution.

In this way, data acquisition is mostly non-intrusive, and the quality of the acquired data becomes a matter of the utilization imposed on the platform by application tasks. If utilization is very high, then circular buffers will overflow, and the Monitor will capture partial data, which might not be enough to produce accurate models.

### 3.2.2 Per Task Monitoring Functionality

For some scenarios, a specific sampling is required, as it is the case for some Machine Learning approaches that depend upon data to be sampled per task and not per CPU. The Monitor API functionalities can be extended to a per-task sampling by splitting the circular buffers per task and by configuring `Criterion::collect()` to be executed whenever a reschedule incurs in a context-switch.

In this way, when a job is scheduled, the `Criterion::collect()` does not require anymore checking for the sampling rate. Instead, it verifies if the running job will change. In a positive case, the method samples the monitored clerks. The sampling is done by accumulating the clerks for the task job leaving the CPU, capturing the clerks' counter growth during its execution. Moreover, the `Criterion::collect()` can be personalized as required by the scheduler design, for instance, accumulating the values per job or following a secondary sampling rule (e.g., hyper period). Every other operation performed by the Monitor works in the same way as before, maintaining the low intrusive design proposed. The data structures used for storing sampled data also remains the same, but now in a different configuration (split per task). Moreover, in this configuration, the sampling rate is adaptable to the very-own task-set context-switching, thus, adding no interruption to the execution and performing the lowest amount of sampling as possible to maintain the necessary accuracy in the current configuration.

### 3.2.3 Actuation Design

Schedulers designed using our framework will eventually actuate on the platform to achieve fine-grained control over the resources at hand. The main actuation mechanisms available in the framework are thread migration and suspension, DVFS, and DPM. If used by a given scheduler, these mechanisms are typically executed before dispatching a thread to a CPU in method `Criterion::award()`, after `Criterion::charge()` has already been applied to the thread leaving the CPU. They may, however, be also used in `Criterion::charge()` and in `Criterion::update()`, which updates a thread's priority at each job release (transaction from Waiting to Ready in Figure 10) whenever a dynamic priority policy is in force. These methods also have plain access to the data collected by the monitor so they can base their actuation on the historical data available.

Thread suspension is attained with `Thread::suspend()` and `Thread::resume()` (not shown in Figure 9). Thread migration is done by invoking `Scheduler::remove()`, adjusting the thread's `Criterion` to designate the target CPU and then calling `Scheduler::insert()`. The core conducting the migration will automatically send an Inter-processor Interrupt (IPI) to the target CPU core, so a reevaluation of the scheduling policy is conducted there. While thread suspension is usually only applicable to best-effort tasks, thread migration yields a powerful mechanism to schedulers aiming to ensure timeliness of critical tasks, either alleviating the load of the core where they are currently running or moving them to a less loaded one.

DVFS is abstracted in our framework through the `CPU::clock()` method (not shown in Figure 9), which takes a target frequency in Hertz. Indeed, today's processors use quite different strategies to implement DVFS that often cannot be directly mapped into arbitrarily chosen frequencies. Therefore, scheduler designers are required to read back the actual CPU's frequency after invoking `clock()`. Dynamic Power Management (DPM) mechanisms implemented by the underlying OS can also be used by schedulers aiming at optimizing energy consumption.

### 3.2.4 A Generalized Learning Strategy

The data collected by the Monitoring API can be used to profile task-sets while they are executed, to assess the system's behavior concerning specific metrics, and also to support the design of novel scheduling heuristics. Concerning this last goal, a set of recurrent steps are here listed to model them as a workflow to learn from collected data. The steps presented in this learning workflow are based on data mining recurrent steps (MANNILA, 1996), like understanding the domain, preprocessing data, discovering patterns, and putting results into use. It comprises the following:

1. **Define the goal** for the aimed heuristic and list a set of variables and events related to it. For instance, a heuristic targeting the optimization of the energy consumed by a task set would probably benefit from overall, per-core, and per-task power consumption.

2. **Collect the Data** while running representative task sets to stimulate the events related to the goal defined on the first step.

3. **Preprocess the Data** so they can be subjected to learning algorithms. Commonly preprocessing tasks include:

    a) Data alignment: as described in Section 3.2, the acquisition of some sorts of data, such as PMU counters, is limited by the hardware to a small number of events at a time. Therefore, task sets must be executed multiple times to capture a meaningful data set. If the OS is a low interference one, simply aligning the data by their time-stamps might be enough, but a Unix-like OS might require more sophisticated alignment algorithms, such as hash-table indexing (e.g., BLAST), suffixes/prefixes tries, and merge sorting (e.g., Slider and Slider II) (LI; HOMER, 2010).

    b) Outlier detection and removal: even if the collected data comes mostly from digital sensors and counters, errors and overflows might occur. In some cases, the conditions that led to the outlier are very relevant to the heuristic. For instance, a very low increment on the retired instructions counter can be interpreted as an outlier, but it can also denote stalls and thus should not be ignored by the analysis.

    c) Redundancy and dependency handling: some events and variables may represent the same feature under certain conditions. Eliminating them reduces processing time and, in some cases, prevent model corruption due to overvaluing a feature (RADHA; MURALIDHARA, 2015). For instance, events directly driven by the CPU's clock become synonyms when the power manager interrupts clock modulation.

    d) Scaling: counters and sensed data may have different ranges. For instance, the temperature will range in extreme conditions from -150ºC to 150ºC, while the Committed Instructions rate is in the range of millions or billions for each capture.

e) Encoding: grouping ranges of values of interest variables in classes can improve the results of classification algorithms such as decision trees. For instance, deadline-misses can be grouped as true and false, or energy consumption readings can be grouped as high, medium, and low for a power-cap heuristic.

4. **Select the features** that best describe the behavior of an event of interest. If the heuristic relies on a predictor, feature selection can considerably reduce the working set on platforms with many accessible variables. It can also reduce the number of resources needed to run the predictor within the scheduler by producing the simplest, less accurate, yet effective models. Some useful algorithms at this stage have been previously depicted in Section 2.1.2.2.

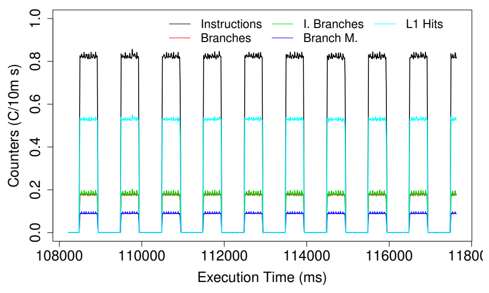
5. **Learn from the Data** by applying machine learning or data mining tools. Some of the algorithms we use in this stage are Artificial Neural Networks (MCCULLOCH; PITTS, 1943), K-NN, the C4 Decision-Tree Generator, and Google's TensorFlow.

6. **Test the model** while capturing data to corroborate it. Since the proposed monitoring infrastructure is non-intrusive, collecting data during validation is fundamental to make small adjustments.

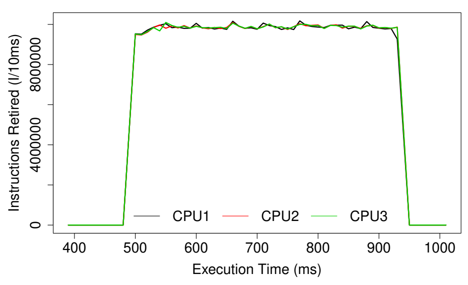
The learning steps described above are applied in this work to achieve a predictor for the task's performance demands in a new frequency configuration, based on their performance traces and the current frequency. Each of these steps is covered in the next sections along with the description of the energy optimizer design.

## 3.3 SUMMARY OF THE NON-INTRUSIVE MONITOR

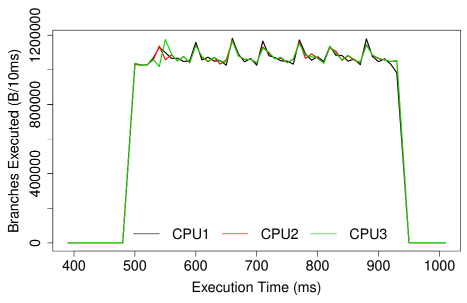
The proposed Monitor API is carefully designed to achieve negligible intrusiveness. The non-intrusiveness of the Monitor API is evaluated in Section 5 alongside the evaluation of the non-intrusiveness of the whole energy optimizer. The Monitor API design enables capturing high-quality data for ML tools to detect patterns and produce models. The data is captured within an architecture-independent API (`Clerk`), and the framework infrastructure allows the implementation of several types of actuators like DVFS and task migration. The framework defines a powerful mechanism to model scheduling policies through the specification of an ordering scheme (`Criterion`) elegantly interfaced via the `operator int()` in C++, making it simple to develop any new policy or heuristic that uses data generated at run-time by the running tasks. Moreover, the Clerk abstractions can also include data sources from hardware accelerators or external to the processor chip, however, the complexity inherent to such data acquisition must be accounted for at the scheduling policy or heuristic development.



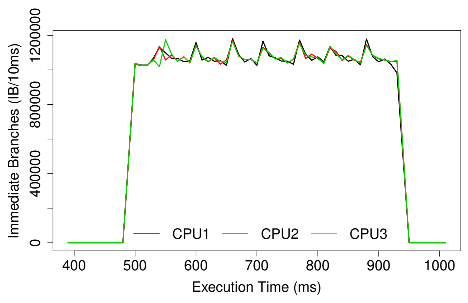
(a) Sample of events (10s).



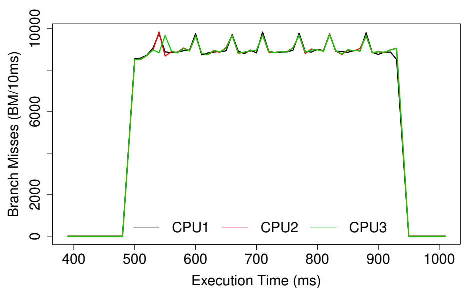
(b) Committed Instruction rate.



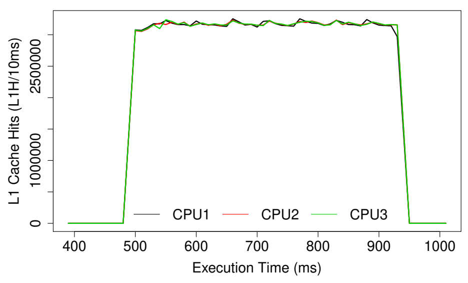
(c) Branches executed.



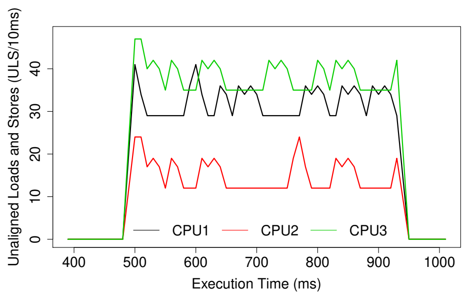
(d) Immediate Branches executed.



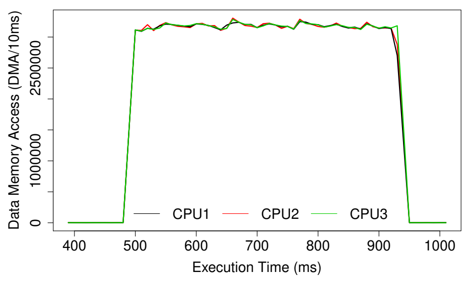
(e) Branch misses.



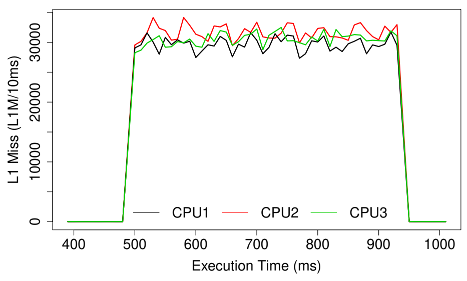
(f) L1 Cache Hits.



(g) Unaligned Loads and Stores executed.



(h) Data Memory Access.



(i) L1 Cache Misses.

Figure 11 – Example of run-time data collected by the Monitoring system.

# 4 ENERGY OPTIMIZER

Handling energy optimizations in critical scenarios while guaranteeing that tasks meet their deadlines is not trivial, requiring automation of the process and adaptability to achieve the desired level of optimization. Actuate over such a critical platform is a cautious process and requires safety margins to avoid failing timeliness. The complexity of such architecture increases due to new functionalities and the increasing demand for more performance, for instance, on autonomous vehicles handling computer vision and several other critical tasks. Therefore, for such scenarios, profiling becomes a task that demands too much experimentation to acquire a sufficient data-set in order to build reliable models, which, for some scenarios is unfeasible due to project specificities or budget limitations. Thus, such a critical system usually relies on an overestimated safety margin, requiring more processors with more computational power to accommodate the system functionalities while disabling multicore features to increase the execution determinism, increasing the hardware costs and the energy consumption of the architecture. In this sense, the approach proposed here aims at automating such an optimization process while attending to online adaptations. The design of the Energy Optimizer is described in the next sections, taking as its main concerns the capabilities of non-intrusiveness, system criticality awareness, and adaptability to new scenarios.

## 4.1 ENERGY OPTIMIZER DESIGN

The energy optimizer is a combination of three main components designed as implementations to each one of the Criterion functions describe in Section 3.2, namely (`Criterion::collect()`) for data collection, (`Criterion::charge()`) for task utilization prediction, and (`Criterion::award()` for actuation. The energy optimizer assumes a scenario where a task deadline is equal to its period and the scheduling algorithm is assumed to be partitioned, avoiding unnecessary migration and increasing tasks cache affinity (GRACIOLI; FRÖHLICH, A. A., 2015). A diagram depicting the Energy Optimizer is presented in Figure 12.

Two actuation methods are implemented by the energy optimizer: DVFS and Task migration. The DVFS actuation has priority over Task migration and is supported by a predictor of task utilization when facing a frequency scaling. A frequency scaling is considered suitable for the current task-set demands if, according to the predictions, for every $CPU_i \in DVFS\_Domain$ the equation $CPU_i.idle\_time > SM$ must be true, where $SM$ is the Safety Margin established for the task-set. The idle time of a CPU is acquired as $1 - \sum t.prediction \ \forall \ t \in CPU_i.Scheduling\_Queue$. The design of the predictions of task utilization is here implemented through the learning approach, more specifically, an Online learning Artificial Neural Network, that, provided with a task's utilization, performance trace, and the current frequency, predicts the task utilization in

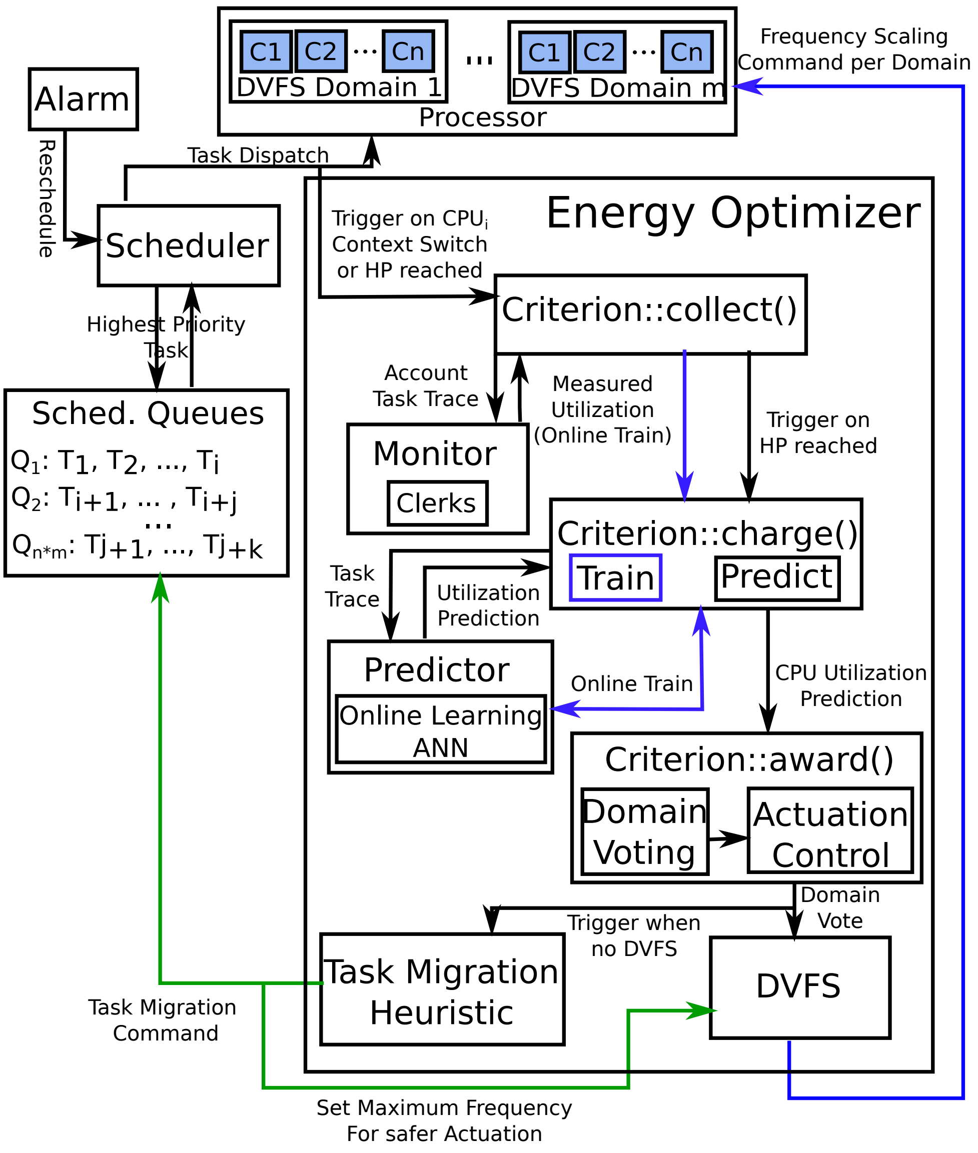


Figure 12 – The Energy Optimizer Design Diagram. The green arrows represent the execution flow for a task migration actuation, while the blue arrows, the execution flow for a DVFS actuation, which also includes an online learning section.

a scenario after a frequency scaling. The performance trace is composed of the most relevant performance counters for the architecture, enabling the ANN model to account for shared resource contention and the system variability.

Embedded systems can benefit from load balanced task-set to overcome DVFS limitations (e.g., when the voltage and frequency scaling applies to all CPUs at the same time). To abstract the DVFS domain specificities of each processor-chip and make the energy optimizer agnostic of a single configuration, the actuation performed by the energy optimizer handles different DVFS domains through voting. CPUs that share a DVFS domain must accord for the actuation to be executed (i.e., frequency scaling), thus, controlling the actuation with no restriction to processor-chip DVFS domain restrictions, as depicted by the `Criterion::award()` actuation in Figure 12. Thus, the energy optimizer can handle different configurations of DVFS domains at the same time inside a processor-chip, like individual cores DVFS and cluster DVFS.

The predictor output provides the capability of safely managing the DVFS actuation. Furthermore, the DVFS energy-savings can be improved when the CPUs achieve a balance of workload, as it helps to avoid temperature hot-spots, which due to the small size of modern multicore processors chips, can influence the energy consumption of the other CPUs and contributes to equal aging, enduring embedded environment life time (PAGANI et al., 2020).

The ANN design and the performance counters selection will be discussed further in the next section (Section 4.2). Furthermore, the energy optimizer design is not limited to the ANN predictor and the Task migration described here, and can be extended to any technique that follows the non-intrusiveness and criticality awareness requirements.

By following the Criterion design, no interruption is added to the system in any of the aforementioned procedures, as each one of them is taken implicitly during ordinary OS actuation. Moreover, the same idea of contention-free buffers used for storing the monitored data is applied for handling the ANN infrastructure and the necessary data structures used by the actuation, which is replicated for each CPU. In this sense, to improve non-intrusiveness, the energy optimizer proposed here assumes that CPU0 is a CPU reserved for management operations only (e.g., Interrupt Handling) and does not run any task from the tasks-set. `Criterion::charge()`, the predictions handling, remains as a per-CPU operation, while CPU0 is the one that periodically checks if every CPU inside a DVFS domain has finished voting at `Criterion::award()`, issuing the actuation commands regarding DVFS control and the task migration heuristic. The main motivation to handle award partially at CPU0 is to avoid the time complexity a task migration heuristic could add to the system. Such algorithms often require iterating over the tasks and CPUs, verifying suitable combinations of tasks in other CPUs. For instance, a migration algorithm could check the performance of every task in every CPU, evaluating the overall performance and selecting the best migration (i.e., one specific task to one specific CPU) according to some metric. This procedure can be seen as bound in time complexity by $\mathcal{O}(|T| * |C|)$.

### 4.1.1 Collecting

In a multicore scenario, different periods between tasks that share resources can incur different execution times for each respective job from the same task. In this sense, to correctly measure the behavior of a task, encompassing the variability of performance in its jobs, the energy optimizer data collection must accumulate the performance counters until a different behavior or a phase completion is found. Once this trigger is reached, or after a certain amount of time of execution (e.g., a maximum

**Algorithm 1** Data collection

```
 1: procedure collect ($t\_prev$, $t\_next$, $CPU_i$)
 2:     for each $c \in Monitor::Clerks$ do
 3:         $t\_prev.clerks[c]$ += $c.read() - t\_prev.clerks\_aux[c]$
 4:         $t\_next.clerks\_aux[c] = c.read()$
 5:     end for
 6:     if $time.now() \geq HP$ then
 7:         for each $t \in CPU_i.Scheduling\_Queue$ do
 8:             for each $c \in Monitor::Clerks$ do
 9:                 $t.clerks\_last\_hp[c] = t.clerks[c]$
10:                 $t.clerks[c] = 0$
11:             end for
12:         end for
13:     end if
14: end procedure
```

actuation period), the energy optimizer prediction and actuation are executed to analyze, through the predictor, the collected data, thus, building awareness of the task demands in a different configuration. On more traditional real-time system, the hyper-period of a task-set can be used to trigger the energy optimizer predictor and actuator, more specifically, the global hyper-period *HP* (hyper-period accounting for all CPUs hyper-periods) can be used. A hyper-period is the smallest interval of time after which the periodic patterns of all the tasks are repeated (RIPOLL; BALLESTER-RIPOLL, 2013). The hyper-period of the ith CPU $HP_{CPU_i}$ is calculated following (9), where *LCM* is the least common multiple of $HP_{CPU_i}$ and *t.period*, $T_{CPU_i}$ is the set of tasks of the ith CPU ($CPU_i$), starting from 1.

$$HP_{CPU_i} = LCM(HP_{CPU_i}, t.period), \forall t \in CPU_i.Scheduling\_Queue \tag{9}$$

While for more dynamic real-time systems, the concept of phases of executions can be used to trigger the energy optimizer. Thus, the OS can be configured to monitor such behavior and trigger the energy optimizer accordingly.

Thus, for the energy optimizer to encompass sufficient information for prediction and actuation, the Monitor comprises the PMU performance counters, task execution time (represented as task utilization), and CPU idle time. They are sampled following the `Criterion ::collect()` behavior described in Section 3.2.2, accumulating the Clerks' growth during the task's jobs execution inside a global hyper-period to provide a per-task vision of the system. Algorithm 1 describes the data collection performed by the energy optimizer.

**Algorithm 2** Charging

```
1: procedure charge ($CPU_i$, $actuated$, $DT$, $TT$)
2:     if not $actuated$ then
3:         for each $t \in CPU_i.Scheduling\_Queue$ do
4:             for each $c \in Monitor::Clerks$ do
5:                 $t.input[c] = t.clerks\_last\_hyper\_period[c]$
6:             end for
7:             $t.prediction = Predictor.predict(t.input)$
8:         end for
9:     else
10:        $to\_train \leftarrow \emptyset$
11:        for each $t \in CPU_i.Scheduling\_Queue$ do
12:            if $|t.prediction - t.clerks\_last\_hp[Execution\_Time]| \geq DT$ then
13:                $to\_train \leftarrow to\_Train \cup t$
14:            end if
15:        end for
16:        $done$ = false
17:        while $TT > 0$ and not $done$ do
18:            $done$ = true
19:            for each $t \in to\_train$ do
20:                $Predictor.train(t.input, t.clerks\_last\_hp[Execution\_Time])$
21:            end for
22:            for each $t \in to\_Train$ do
23:                $prediction = Predictor.predict(t.input)$
24:                $done = done$ and
25:                    $(prediction - t.clerks\_last\_hp[Execution\_Time]| < DT)$
26:            end for
27:            $TT = TT - 1$
28:        end while
29:    end if
30: end procedure
```

### 4.1.2 Charging

The energy optimizer charge runs at each global hyper-period to account for the whole task-set, and aims at providing an adaptable and accurate model for tasks' utilization prediction in a lower frequency scenario. Algorithm 2 depicts `Criterion::charge()` functionalities in the energy optimizer.

When a global hyper-period *HP* is reached, each task accumulated clerk value in the respective global hyper-period is accounted by `Criterion::collect()` (described in Section 4.1.1). With clerks data for each task available, each CPU runs `Criterion ::charge()`. This method runs a predictor to provide an estimation of each task utilization in a new configuration, more specifically, their estimated utilization if the frequency of the CPU is decreased in one level. The utilization $U_{t_i}$ of a task $t_i$ is the ratio of its execution time *t.clerks_last_hp*[*Execution_Time*] and the global hyper-period *HP*

($U_{t_i} = \frac{t.clerks\_last\_hp[Execution\_Time]}{HP}$), represented in the range {0,1}. The predictor is a Learning-based algorithm that models the architectural phenomena through the performance trace of a task and estimates the frequency scaling impact in the utilization of the task. Thus, it is expected that the learning-based algorithm used is capable of building awareness of such relation from both an initial offline model provided by a synthetic task-set, and from online learning, which focuses on the model specialization to the current task-set. This strategy avoids cold start issues at the same time it avoids extensive offline training and profiling, reducing the model design complexity and making it agnostic of a task-set. In this work, a Artificial Neural Network is used as a predictor, enabled via backpropagation-based incremental training.

In this way, through a set of selected features, composed of OS statistics and PMU counters, the offline training is expected to capture the relation between the selected features and tasks utilization, a knowledge that will be enhanced at run-time to better suit unforeseen scenarios other than the synthetic ones used to stimulate the system more generically. So, as long as the same set of features used for offline trained is continuously monitored in the target architecture, the online training can teach the ANN to fit to new behaviors and architectural variability. For further adaptation, the energy optimizer can be configured to also account for different contexts of execution by reconfiguring the energy optimizer parameters, like the Safety Margin *SM* to be used on more restrained phases or even a set of predictors specifically design for a set of defined contexts.

Whenever a prediction is made, the pair input-output is stored in a historical buffer. If a DVFS actuation is taken by `Criterion::award()`, a flag denoting it will be raised. In the next `Criterion::charge()` execution, instead of running a prediction, the predictor accuracy will be evaluated, calculating its output deviations from the tasks' measured utilization after actuation. If a prediction deviates from the measured utilization for more than a configurable deviation threshold $DT$, the input from the last hyper period is retrieved and the predictor is trained to match the measured utilization. For the ANN predictor implemented here, the deviations are used for incremental training, back-propagating the error accordingly for every train round. The online training is repeated until the predictor fits each input-output pair set or a limit of tries $TT$[1] is reached. Even though the predictor already encompasses initial knowledge of the architectural phenomena, it is not expected to present a remarkable accuracy in its initial form. In this way, the process modeled here focuses on learning the task-set characteristics at the highest frequencies to be able to safely actuate when the task-set gets closer to its utilization Safety Margin *SM* threshold.

[1] The amount of retrains the predictor can execute online is based on the amount of overhead the online training creates and how much it is acceptable for the current task-set.

### 4.1.3 Actuation

The energy optimizer actuation is executed at each global hyper-period, after the execution of `Criterion::charge()`. The actuation is done by the `Criterion::award()`, handling DVFS domain voting based on the predictions done at `Criterion::charge()`, and actuating through DVFS and task migration accordingly. Algorithm 3 depicts the `Criterion::award()` functionalities in the energy optimizer.

**Algorithm 3** Actuating

```
 1: procedure award ($CPU_i$, $DVFS\_domain$, $SM$, $F\_level$, $MT$)
 2:     if $CPU.idle\_time < SM$ then
 3:         $cpu::clock(cpu::clock() + F\_level)$
 4:         $Vote\_is\_Ready_{CPU_i}$ = false
 5:     else
 6:         $U_{CPU_i} = 0$
 7:         for each $t \in CPU_i.Scheduling\_Queue$ do
 8:             $U_{CPU_i}$ += $t.prediction$
 9:         end for
10:         $vote_{CPU_i} = 1 - U_{CPU_i} > SM$
11:         $vote\_is\_ready_{CPU_i}$ = true
12:     end if
13:     if $\forall\ CPU \in DVFS\_domain : vote\_is\_ready_{CPU}$ == true then
14:         $actuate = \forall\ CPU \in DVFS\_domain : vote_{CPU}$ == true
15:         if $actuate$ then
16:             $cpu::clock(cpu::clock() - F\_level)$
17:             $actuated$ = true
18:         else
19:             $actuated$ = false
20:             $evaluate\_migration(DVFS\_domain, MT)$
21:         end if
22:     end if
23: end procedure
```

The utilization predictions are composed to represent the utilization of each CPU. The utilization of the *ith* CPU is accounted as $U_{CPU_i}$ following (10), where $N$ is the number of tasks inside $CPU_i.Scheduling\_Queue$. As the collected data already accounts for OS overhead and shared resources contention (i.e., through utilization and performance counters), it is safe to assume that the sum of all task predictions represents the total CPU utilization.

$$U_{CPU_i} = \sum_{j=0}^{N} t_j.prediction \tag{10}$$

Every CPU inside the DVFS domain runs `Criterion::award()` periodically at each global hyper-period $HP$, computing their utilization and voting for frequency decrease. The voting for frequency decrease is done by calculating the available idle time of every CPU inside a DVFS domain using the predictions done at `Criterion::charge`. The vote of a CPU can be defined as follows:

$$vote_{CPU_i} = 1 - U_{CPU_i} > SM \tag{11}$$

A safety margin for resource reservation *SM* must be defined to ensure the criticality of the system. A safety margin is a well-known concept of critical systems, which aims at ensuring a critical-safety scenario. To ensure safety, such margin is often overestimated when based on an offline estimation of WCET and task-set profiling. In the design proposed here, the predictor, when fitted to the current scenario, provides safety to the actuation, thus, enabling the usage of a smaller safety margin. Moreover, a safety margin *SM* can be an adaptive configuration that follows the tasks behavior variation, considering, in its definition, tasks with variable behavior and guaranteeing the necessary resource reservation by limiting the actuation in the presence of a lower utilization of the tasks, and reducing the margin when in the presence of a higher utilization of the tasks. For instance, a task that periodically checks for a sensor reading and only actuates on a specific scenario, will have a small utilization during most of the execution, but under the specific scenario, will increase its usage. In this way, the system can model a resource reservation represented through the *SM* to assure the task timing correctness. So, during a lower utilization phase, the energy optimizer will be limited by *SM*, which will be increased with the task variability. During a higher utilization phase, the utilization of the task can be deduced from *SM*, thus, the energy optimizer will not undo its optimizations.

When the last CPU inside a DVFS domain finishes its voting round, it also evaluates the frequency scaling for that DVFS domain. Thus, if, and only if, every CPU inside the DVFS domain voted for frequency scaling, it is executed. The flag *actuated* is set to (true) for the DVFS domain if the frequency was scaled. Otherwise, it is set to false, and the frequency remains untouched. Whenever no further frequency scaling fits the current scheduling slack, the migration heuristic is executed.

In this sense, if an isolated core is being responsible for the energy optimizer, a pooling strategy can be easily implemented to check for the DVFS domain vote availability. In response, the actuation command will be issued by the isolated core, triggering a frequency scaling. For task migrations, the isolated core will set a flag to be checked by the target CPU when the next global hyper-period *HP* is reached in the DVFS domain, avoiding issuing a migration in an unappropriated moment.

#### 4.1.3.1 Task Migration

The task migration heuristic proposed here consists of minimizing the activity variance between CPUs. The task migration heuristic is capable of migrating tasks in two different ways: a simple migration, sending a task from one CPU to another, or by swapping tasks between CPUs. The activity of a CPU is calculated using the concept of a weighted activity vector, a composition of the very same features used by the predictor, scaled to the range {0,1}. The activity $A$ of each $CPU_i$ is calculated as follows:

$$A_{CPU_i} = \sum_{j=0}^{N} \sum_{k=0}^{E} t_j.activity[k] * W_k * t_j.U \tag{12}$$

Where $N$ is the number of tasks inside the *Scheduling_Queue* of $CPU_i$, $E$ is the number of features being monitored, $t_j.U$ is the utilization of the task $t_j$, and $W_k$ is the weight of the feature $k$.

The proposed heuristic is also supported by a recent memory concept composed of the last activity prediction $LA_{CPU}$ (CPU activity estimated for migration) and the lowest frequency reached until now $LF$. Whenever a migration incurs in a worse configuration, the heuristic revokes it, and the task is no longer allowed to migrate to that CPU until the system state changes. Such behavior can result from an underestimated migration or due to a resource contention that didn't exist before. However, if the last execution was a swap, and in the current state, the best migration revolves around sending one task from the CPU negatively affected to the one positively affected, that means the heuristic has solved a contention, and hence, it is allowed to maintain the last migration. If the lowest frequency is not met, the heuristic undoes the two previous steps. At every undoing, the migration threshold is also increased.

The optimal weight $W_k$ of each feature $k$ is determined by profiling the task-set, thus avoiding a cold start. The profiling is done by initially setting the feature weights to 0.5 and updating them online after each migration using Gradient Descent of the activities of each CPU $A_{CPU_i}$ considering the chosen migration and the measured value in the next hyper-period $HP$, as depicted in Algorithm 6. The implementation is based on (ROHAN PAUL, 2020) implementation of multivariate gradient descent. After the profiling execution is done the weights are saved and used for the normal execution of the task-set.

The heuristic scores a task migration according to its reduction in the activity variance between CPUs. Moreover, only migrations that do not exceed the maximum activity of each feature, i.e., $W_k > (\sum_{j=0}^{T} t_j.activity[k] * W_k * t_j.U)\ \forall k \in activity$, are evaluated. If the migration with the lowest activity variance presents a minimization higher than the current migration threshold, the migration is executed. In the case it fails to meet threshold, task swaps between every CPU pair are evaluated in the same way. Before the execution of the migration, a reset of CPU frequency takes place, setting it to the maximum value to prevent that a migration exceeds the established Safety Margin (e.g., due to an underestimation of the migration impact). Algorithms 4 and 5 depicts the idea of the migration heuristic for the procedure of finding and executing the

**Algorithm 4** Activity Control - Auxiliary Functions

```
1: procedure fit (CPU, t, W)
2:     fit = true
3:     for each $k \in Activity\_Features$ do
4:         $fit = fit$ and $(CPU.activity[k] + t.activity[k] * t.U * W_k) < W_k$
5:     end for
6:     return fit
7: end procedure
8:
9: procedure update_activity (DVFS_domain, t, W)
10:    for each $CPU \in DVFS\_domain$ do
11:        $CPU.activity = 0$
12:        for each $t \in CPU.Scheduling\_Queue$ do
13:            for each $k \in Activity\_Features$ do
14:                $CPU.activity[k]$ += $t.activity[k] * W_k * t.U$
15:            end for
16:        end for
17:    end for
18: end procedure
19:
20: procedure remove (CPU, t, W)
21:    for each $k \in Activity\_Features$ do
22:        $CPU.activity[k]$ -= $t.activity[k] * W_k * t.U$
23:    end for
24: end procedure
25:
26: procedure add (CPU, t, W)
27:    for each $k \in Activity\_Features$ do
28:        $CPU.activity[k]$ += $t.activity[k] * W_k * t.U$
29:    end for
30: end procedure
31:
32: procedure variance (DVFS_domain)
33:    $average = 0$
34:    $variance = 0$
35:    for each $CPU \in DVFS\_domain$ do
36:        $average$ += $\frac{A_{CPU}}{|DVFS\_domain|}$
37:    end for
38:    for each $CPU \in DVFS\_domain$ do
39:        $variance$ += $\frac{(A_{CPU} - average)^2}{|DVFS\_domain|}$
40:    end for
41:    return variance
42: end procedure
```

**Algorithm 5** Task Migration

```
1: procedure evaluate_migration (DVFS_domain, Threshold)
2:     if Last_Frequency < cpu::clock() then
3:         Last_migrate.block()
4:         schedule_migration(Last_to, Last_from, Last_migrate)
5:         Threshold += increase_factor
6:         cpu::clock(Maximum_Frequency)
7:         return true
8:     else
9:         update_activity(DVFS_domain, W)
10:        variance = variance(DVFS_domain)
11:        if variance < Threshold then
12:            return false
13:        end if
14:        from_cpu = null
15:        to = null
16:        migrate = null
17:        Threshold_aux = Threshold
18:        for each CPU $\in$ DVFS_domain do
19:            for each t $\in$ CPU.Scheduling_Queue do
20:                remove(CPU, t)
21:                for each CPU_target $\in$ (DVFS_domain − CPU) do
22:                    if not t.blocked(CPU_target) and fit(CPU_target, t) then
23:                        add(CPU_target, t)
24:                        $variance_{new}$ = variance(DVFS_domain)
25:                        if $variance_{new}$ < variance − (variance $*$ Threshold_aux) then
26:                            variance = $variance_{new}$
27:                            from_cpu = CPU
28:                            to = CPU_target
29:                            migrate = t
30:                            Threshold_aux = 0
31:                        end if
32:                    end if
33:                end for
34:                add(CPU, t)
35:            end for
36:        end for
37:        if from_cpu $\neq$ null then
38:            Last_Frequency = cpu :: clock()
39:            Last_from_cpu = from_cpu
40:            Last_to = to
41:            Last_migrate = migrate
42:            cpu::clock(Maximum_Frequency)
43:            $schedule_{m}igration$(from_cpu, to, migrate)
44:            return true
45:        end if
46:        return false
47:    end if
48: end procedure
```

**Algorithm 6** Gradient Descent weight update

```
1: procedure update_weights (DVFS_domain, Learning_Rate, W)
2:     for each $CPU \in DVFS\_domain$ do
3:         $error_{CPU} = A_{CPU} - LA_{CPU}$
4:     end for
5:     for each $k \in Activity\_Features$ do
6:         $adjustment = 0$
7:         for each $CPU \in DVFS\_domain$ do
8:             $adjustment$ += $LA_{CPU}.activity[k] * error_{CPU}$
9:         end for
10:        $adjustment$ /= $DVFS\_domain.size()$
11:        $W_k$ -= $Learning\_Rate * adjustment$
12:    end for
13:    return W
14: end procedure
```

best migration (non-swap) or revoking a migration. A swap is only evaluated when all possible non-swap migrations were discarded. The search method for swap incurs on adding one more *for loop* inside the *for loop* of lines 21-33 to evaluate every task inside the *CPU_target* as a possible swap. The process requires removing the task from the *CPU_target* activity vector and checking the *fit* for both CPUs, and adding it back to the *CPU_target* after evaluation.

## 4.2 LEARNING WORKFLOW: FEATURE EXPLORATION AND ONLINE LEARNING ANN

In this section, the learning process is depicted in details. First, the process envisioned for feature exploration is presented, followed by the ANN design description. The feature exploration method focuses on stimulating the architecture and collecting every performance counter available. In this way, it is possible to find the counter-set that better represent the system performance, especially considering counters that also can depict shared resource contention in the multicore platform, thus, providing information regarding the architectural phenomena that impacts the performance of tasks. Such information can be used both to guide the aforementioned task migration heuristic and to serve as input for the ANN model predicting the impact a DVFS actuation have into a task performance. Such a counter-set is highly dependent on the architecture itself, where the number of counters necessary to represent the architectural phenomena will vary depending on the complexity of the architecture. Hence, the process described for the feature exploration is designed to be extendable for several architectures, especially the feature selection process. However, the final counter-set will vary according to the target architecture.

Shared resources in multicore processors, when requested by two cores simultaneously, will create access contention. For instance, two tasks running in parallel requesting data from the last level cache or main memory will create such behavior, where one will suffer from contention waiting for the required resource to be released. Moreover, the performance increase provided by faster memory access (e.g., caches) is one of the main concerns when addressing performance variability in a multicore scenario, where two or more tasks running in parallel can evict cache lines from each other, impacting their execution time.

Tracing tasks in order to identify counters that can provide information regarding architectural usage of shared resources, and then using them to depict contention behavior, can benefit migration heuristics, which can migrate such task to avoid the contention. Nevertheless, if the contention is somehow unavoidable, the system must build awareness of the contention impact and consider this information when applying frequency scaling and other kinds of actions. Henceforth, a model created to predict execution time using timing characteristics or classical counters (i.e., Committed instructions rate, Cache Access, and Cycle Count) without a proper relevance analysis for the target architecture, will have limited expressiveness in a complex scenario, like when facing shared resource contention, decreasing the model accuracy for such scenarios.

The data-set must include scenarios with and without contention. A correlation analysis of performance counters and CPU utilization without considering contention scenarios will indicate that most of the performance counters are proportional to CPU utilization. For instance, when not considering shared resources contention, the classical counter Committed Instruction is proportional to the task's performance, as depicted through Task T1 in Figure 13. Figure 13 presents the behavior of a task under three different conditions. T1 is the task running in a no contention scenario, T3 is the task running in a low contention scenario, and T2 is the task running in a high contention scenario, which presents a performance loss in its execution time of approximately 10%, and most notable, making its utilization disproportional to the Committed Instructions rate. In this way, when adding shared resources implications, like in the behavior T2, this counter alone is not sufficient to accurately represent performance, as the relation previously established for the counter Committed Instructions and Task Utilization is not valid anymore. A model built upon this counter alone, without considering the contention behavior, will mislead a prediction to a wrong actuation.

Exploring every performance counters during the feature extraction process can reveal meaningful information about the architectural behavior. A combined performance counter-set, acquired through feature selection techniques can yield sufficient information to build a prediction model accounting for contention. For instance, Counters like Bus Access for Memory Write provides a better description of the scenario presented in Figure 13. Through the depiction presented in Figure 14, it is clear that the

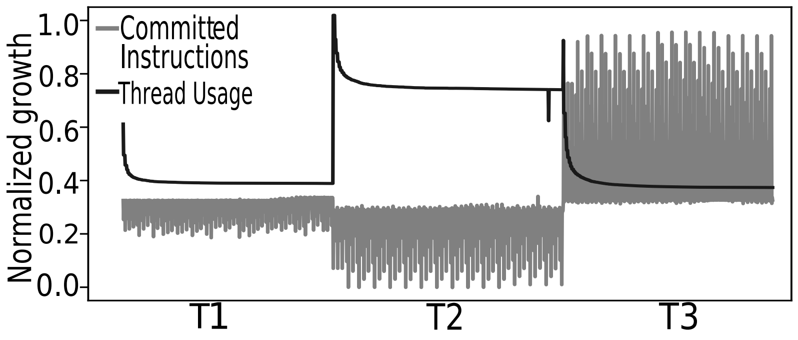


Figure 13 – Committed Instructions trace of a task in three different scenarios: T1 runs in a no contention scenario, T2 runs in a high contention scenario, and T3 runs in a low contention scenario.

number of Bus Access for write operations better depicts the architectural phenomenon that created the worsened performance than the Committed Instructions. Where in task T2 the counter growth highly increases when compared to the other scenarios, as it is facing contention over the Memory Bus by sharing it with other tasks running parallel in the platform.

Not only memory hierarchy but any resources shared between the CPUs can impact the performance of a task in a multicore platform. Thus, a combination of performance counters is expected to yield a more reliable prediction model for CPU frequency demand prediction by also considering performance losses.

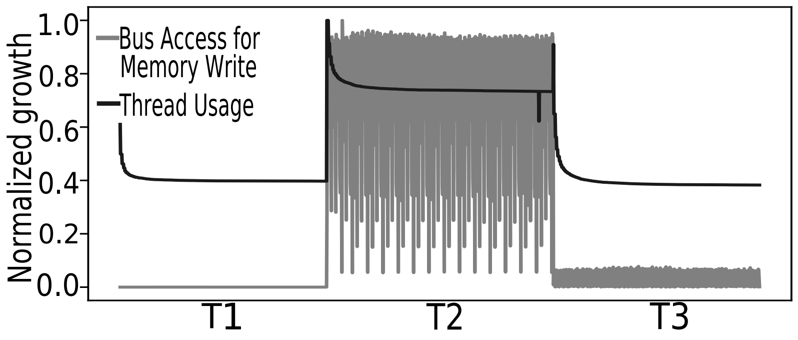


Figure 14 – Bus Access for Memory Write trace of a task in three different scenarios: T1 runs in a no contention scenario, T2 runs in a high contention scenario, and T3 runs in a low contention scenario.

Considering the aforementioned, the feature exploration designed here relies on the data collection performed through the execution of a representative task-set, using benchmark tasks to stimulate architectural phenomena. In this way, the chosen task-set must include examples with and without shared resources contention for the feature selection techniques to be able to correlate the counters to the utilization of the modeled tasks. The Monitoring API (see Section 3) is then configured for collecting the performance counters and OS statistics for the selected task-set. Moreover, to enable a complete overview of the architecture, every performance counter available is expected to be sampled. To do so, several execution will be required, as PMUs are limited regarding the amount of registers available for monitoring.

The Monitor API configuration also includes the definition of a sampling rate. Defining an optimal sampling rate requires profiling for each task-set, and must consider the problem requirements (e.g., samples in a specific form, like per CPU or Task), the very own specificities of the task-set (e.g., by synchronizing the sampling rate with rescheduling operations), and the monitored features specificities (e.g., a sensor that has a low sample rate). For instance, if a too high sampling rate is selected, the collection of sensors and performance counters with a low occurrence rate can mislead a further analysis, as sampling them in a too high frequency will result in several captures with zero-occurrences, possibly classifying the monitored event occurrence as an outlier. On the other hand, selecting a too low sampling rate can create trashing of the collected data, mixing two or more behaviors into the same capture (e.g., job rescheduling or mixing it with other tasks or even idle time).

Therefore, as the architectural phenomena stimulation is here assumed to be done through a synthetic task-set, a recommendation is to sample the counters at least once for each job, aiming for samples that are composed of a single task trace. This enables a correlation analysis between counters growth and the utilization of a specific task. A synthetic task-set ease the profiling of the architecture, as the as the architectural phenomena stimulation can be done in a controllable scenario. For instance, configuring jobs with the same priority provides more control over the parallelism present in the execution of the task-set (enables periods of known shared resource contention) at the same time it avoids undesirable task's preemption that could mix behaviors on the same sample.

### 4.2.1 Pre-processing

Following the learning strategy proposed in Section 3.2.4, the learning goal and the data collection configuration have already been defined in the previous sections. The next step is the pre-processing of the data collected, which is described below.

- Data alignment: Given PMU simultaneous monitoring limitations, several executions are necessary to capture all performance counters. Due to the low intrusiveness added by the monitor design, and assuming a low intrusive RTOS (e.g., EPOS (UFSC/LISHA, 2019)), the data captured through the executions will be equivalent regarding timing characteristics[2]. Along with the fact that no feature has a limitation over its sampling rate (performance counters are not limited to

[2] In the case the target OS does not provide low intrusive RTOS capabilities, a more sophisticated data alignment process is required. For such, the steps recommended in Section 3.2.4 can be used to implement the data alignment considering, for instance, interpolation and sampling rate synchorinization

sampling), no misalignment between features will be present (e.g., a temperature sensor can have its sampling rate lower than the PMU counters, which would create the necessity of interpolation). Thus, to gather the data-set from the executions, only time-stamp alignment is necessary. After alignment, the data-set collected by the Monitor API per CPU is concatenated, resulting in a single data-set with all PMU events and OS statistics as features. Lastly, the samples are transformed from cumulative count to per capture growth (the difference from a current count to its previous).

- Outlier detection and removal: Due to the occurrence of performance counter overflows (32-bit or 64-bit registers depending on the architecture, limited to $2^{32}-1$ or $2^{64}-1$ event occurrences count), some outliers are present in the data sampled. However, as this is known behavior of performance counters (event occurrences count can only grow and not decrease), they can be detected whenever $Current\_Sample < Previous\_Sample$. Thus, instead of removing the samples, the original value of the counter growth can be reconstructed as follows: $Counter\_growth = Counter\_Limit - Previous\_Count + Current\_Count$. Moreover, as every collected data comes from digital counters and OS statistics, no other outlier removal technique was necessary (e.g., remove sensor sampling errors).

- Redundancy and dependency handling: Redundancy and dependency are handled during feature selection using Pearson's Correlation Coefficient between each pair of counters, and is further detailed in Section 4.2.2.

- Scaling: Each counter has been scaled to the range {0,1} using Min-Max normalization (SHALABI et al., 2006), where to scale each counter, its respective minimum and maximum values at the data-set where used as the minimum and maximum of the algorithm.

- Encoding: Data discretization was applied as a separated pre-processing from the main pre-processing workflow, specifically to be used as input for some feature selection techniques that require discretized data (i.e., Information Gain Ratio). The discretization process subdivides every feature into ten folds as follows: class 0 represents values ranging from 0 to 0.1, 1 from 0.1 to 0.2, 2 from 0.2 to 0.3, and so forth. Moreover, the dependent variable, *Task Execution Time*, was discretized into two classes, with and without shared resource contention.

- Data Split: The last pre-processing step applied to the data-set was a simple split per task using the Running Thread ID as a parameter. Traces from tasks that represent the same benchmark were merged to compose the different behaviors of each stimulated configuration.

### 4.2.2 Feature Selection

The feature selection process combines three different feature selection algorithms, each one using a different technique to filter the data-set and acquire the more relevant counters. The final feature-set is a merged analysis of the three techniques, aiming at increasing the model accuracy (prediction of impacts caused by frequency reductions at tasks' performance). The combination of feature selection techniques leads to a more reliable assessment of feature relevance (MOLINA et al., 2002), as by merging different analysis, we do not limit the feature-set to a specific learning technique. Focusing on a single technique can mislead the selection if the final learning model has different limitations and metrics (e.g., linear regression vs. ANN non-linear model). The merging method applied here is based on selecting the most relevant counters in each technique while filtering those with a high Pearson's Correlation Coefficient with each other, thus, avoiding redundancy in the final ML model. The method starts by selecting the counters with the highest correlation to the dependent variable that is present in each of the three techniques results. If the counter is not present in the three techniques, but a redundant counter to this one is present (i.e., a counter with a high Pearson's Correlation Coefficient (PCC) value to the one selected by the other techniques). The merge is done following the PCC output order.

The three approaches presented here include Correlation and Entropy (Filter Methods) and Linear Regression with regularization (Embedded Method) feature selection techniques. Each algorithm is described below:

- **Pearson's Correlation Coefficient (PCC)**: PCC is a classical statistical method used to calculate the relationship between two attributes within a data-set. It indicates how much a variation of one feature is related to the other's using a variance (13) and covariance (14) analysis (BENESTY et al., 2009), where a PCC between two features, $X$ and $Y$, can be calculated following (15). A PCC between two features can result in a value in the range {-1,1}. A PCC of 1 represents a perfect positive linear relationship. A PCC of 0 represents that the two variables are independent. And a PCC of -1 represents a perfect negative relationship. Correlation coefficients are very useful to identify the more relevant features to the dependent one (i.e., Task execution time). Nevertheless, redundancy removal of highly correlate features is also another possible result of this analysis, reducing the data-set multicollinearity. In this sense, for the sake of simplicity, we use the absolute values of PCC between two features for both feature selection and redundancy removal procedures. PCC can handle both continuous and discrete data. Thus, no extra preprocessing was necessary.

$$\sigma^2 = \frac{\sum_{n=1}^{N}(\overline{X} - X_n)^2}{N} \tag{13}$$

$$Cov(X,Y) = \frac{\sum_{n=1}^{N}(X_n - \overline{X})(Y_n - \overline{Y})}{N} \tag{14}$$

$$PCC(X,Y) = \frac{Cov(X,Y)}{\sqrt{\sigma(X)^2} * \sqrt{\sigma(Y)^2}} \tag{15}$$

- **Information Gain Ratio (IGR)**: IGR (QUINLAN, 1986) is a filter algorithm that scores each feature based on entropy, a statistical measure that shows how much uncertainty is inherited by the feature possible outcomes. The Information Gain Ratio of an attribute is calculated using (19), it represents the entropy of the class $H(Y)$ (16) minus the conditional class entropy, given the value of the feature $X$ ($H(Y|X)$) (17), divided by the intrinsic value of the feature (18). As Information Gain Ratio is meant to handle discrete data, the preprocessing step *Encoding*, described in the previous section (Section 4.2.1), is applied to the data-set.

$$H(Y) = -\sum_{v \in Y}(p(v)\log_2 p(v)) \tag{16}$$

$$H(Y|X) = -\sum_{v \in Y}\left(\frac{|\{x \in X | x.class = v\}|}{|X|} * H(\{x \in X | x.class = v\}\right) \tag{17}$$

$$IV(X,Y) = \sum_{v \in Y}\left(\frac{|\{x \in X | x.class = v\}|}{|X|}\log_2\frac{|\{x \in X | x.class = v\}|}{|X|}\right) \tag{18}$$

$$IGR(X,Y) = \frac{H(Y) - H(Y|X)}{IV(X,Y)} \tag{19}$$

- **Lasso CV**: Least Absolute Shrinkage and Selection Operator with Cross Validation (TIBSHIRANI, 1996) is a linear regression method that can be used for feature selection, considered well suited for multicollinearity data-sets, like performance counters. Lasso Regression uses shrinkage, a technique to shrunk data values towards a central point like the mean, and can be calculated as a minimization of formula presented in 20. The regression performs L1 Regularization, adding a penalty equal to the absolute value of coefficient magnitude to a feature, which can lead to zero coefficients. Cross-Validation is used to find the best $\lambda$ configuration for the Lasso model, selecting the model with the lowest error in the validation data-set.

$$\sum_{i=1}^{N}\left(y_i - \sum j = 1^K x_i j * \beta_j\right)^2 + \lambda \sum j = 1^K |\beta_j|) \tag{20}$$

### 4.2.3 ANN Design

Learning the impact of performance losses, especially for non-specific solutions, is not trivial, as it lends on several aspects of task-sets and multicore architecture. The capability of approximating an unexplored non-linear function and the model extrapolation to unforeseen scenarios provided by Multi-Layer ANN (PAGANI et al., 2020) motivated the choice of ML predictor for the performance demands of tasks based on their traces. Moreover, ANNs are reliable and widespread ML methods for performance tracing-based predictions (RAI et al., 2009; CHEN et al., 2018; YE; XU, 2012; SHEN et al., 2013; MARINAKIS et al., 2019). They also avoid trial-and-error and the high convergence time of Reinforcement Learning solutions. Furthermore, other learning methods, like unsupervised learning (e.g., k-means), are focused on defining a hidden structure for unlabeled data, which does not fit our scope.

ANN are complex methods which can require high computational costs and, if not carefully designed, can create undesired contention into cache locality in a multicore platform. To afford an ANN in a non-intrusive design, some restrictions were imposed: (i) The ANN must be limited to linear activation functions (i.e., Stepwise linear approximation to symmetric Sigmoid) to reduce its computational cost. (ii) The ANN offline tuning process must focus on finding a simple architecture with sufficient performance, thus, reducing the ANN computational complexity. (iii) A contention-free design must be accounted for, where every CPU must have its own ANN predictor structure.

The ANN learning here is guided by a backpropagation-based training (for more details over backpropagation algorithm see Section 2.1.2.1). The error measure used to train the ANN is Mean Square Error (MSE), measuring the mean squared deviation between the regression result and the input label, where the training aims at minimizing the MSE. Moreover, MSE is a widespread algorithm used for measuring regression error, where the error is backpropagated from the output layer to the input layer updating the neurons connections weights accordingly. In this scenario, to achieve continuous learning for online specialization to the running task-set, and further optimize the model accuracy, we have selected to train the ANN following incremental training method, as the online training is naturally incremental learning. Thus, at run-time, each time the ANN runs a prediction, the input pair must be stored to be evaluated after actuation. Whenever a utilization prediction deviates for more than a deviation threshold ($DT$) from the measured utilization, the input from the last actuation is retrieved, and the ANN is trained incrementally, back-propagating the error accordingly for each input-output pair.

The ANN model must first be trained offline, using pre-collected data to tune the ANN hidden layer configuration and create an initial base-knowledge to avoid cold start issues. The data collection process revolves around sampling the selected counters and OS statistics on every frequency configuration available on the platform. Thus, the labeling of each sample can be done by coupling samples with their corresponding measured utilization after one level of frequency scaling. For this purpose, the Monitor API configured with its *Per Task Monitoring Functionality* can be used to collect samples for the task-set in the same configuration proposed in Section 4.1.1, achieving clean and adaptable sampling of task traces. In this sense, the ANN topology, in its first layer, the input layer, is composed of the final feature-set acquired through the feature selection process along with the task utilization and current CPU frequency. The last layer, the output layer, is composed of one neuron and returns the prediction for the task utilization at a lower frequency level. The hidden layer topology and the learning rate of the ANN are parameters that require tuning.

**Algorithm 7** ANN Topology Evaluation

```
1: procedure configuration_performance (train, test, validation, topology)
2:     ANN.topology = topology
3:     ANN.incremental_train(train, test)
4:     average = 0
5:     for each $v \in validation$ do
6:         error = |ANN.predict(v) – v.label|
7:         count = 0
8:         while error > DT do
9:             ANN.incremental_train(v)
10:            error = |ANN.predict(v) – v.label|
11:            count +=1
12:        end while
13:        average +=count
14:    end for
15:    average = average / validation.size()
16:    return average
17: end procedure
```

To avoid over-fitting the ANN, a second data-set, collected from a different task-set configuration, can be used as a validation data-set. Other than the validation data-set, the evaluation of the ANN topology and the learning rate is done by simulating the online training, considering as performance metric the average number of rounds the ANN requires to fit a sample from the validation data-set, following the maximum deviation threshold *DT* desired. A description of the performance evaluation of a ANN topology considering its adaptation is presented in Algorithm 7, using as parameters the training data-set subdivided into *train* and *test* data-sets, the validation data-set, and the topology to be evaluated. In the aforementioned algorithm, the function `incremental_train(v)` (line 9) is an implementation of the standard backpropagation algorithm as described in Section 2.1.2.1, and the function `incremental_train(train, test)` (line 3) is the same algorithm, but executed sequentially for the set of inputs presented in the set train instead of a single sample, while the test set is used for validation purposes.

The tuning process is here implemented by iteratively modifying (e.g., increasing and decreasing) the topology of the hidden layers (number of layers and number of neurons per layer) and the learning rate, starting from a simple topology with one neuron at the hidden layer and expanding the number of neurons one by one. Whenever adding more neurons at one layer does not improve the model performance (or even decrease it), a new layer is added. In the same way, if adding a new layer does not improve the model performance, the process stops. For instance, if adding a new neuron to the first hidden layer of a topology with two hidden layers with two neurons each does not improve the model performance, the process will try adding the neuron to the second layer instead. If the performance did not improved again, the process of adding new neurons to the current layers stops, and a new layer is added with one neuron. If the model performance did not improve by adding the new layer, the process stops. This method execution time can be reduced if a performance threshold is provided, stopping the process when the threshold of performance is met.

## 4.3 SUMMARY OF THE RUN-TIME ENERGY OPTIMIZER DESIGN

The runtime energy optimizer for multicore embedded architectures is designed to cope with the stringent time requirements of critical tasks. The energy optimizer capabilities include DVFS and task migration. It actuates based on the predictor (ANN) output, that aims at predicting the impact of frequency scaling into the performance of a task based on its performance trace. The energy optimizer accounts for every task currently running at each CPU to conceive an actuation, considering the available idle time and a user-defined safety margin. Moreover, the predictor model is built upon runtime traces collected from hardware performance counters and OS variables selected using offline feature extraction algorithms. The feature extraction process aims at exposing the most relevant variables related to performance using synthetic architecture-specific task-sets. The traces collected are also used to build offline training to tune the ANN topology. The predictor is then trained at runtime whenever the frequency is scaled, setting it free from the initial synthetic task-set.

The proposed design abstracts heterogeneity by splitting the actuation voting and migration heuristic per DVFS domain. In this way, the ANN used by each CPU inside the DVFS domain can learn the specificities of each DVFS domain architecture, adapting to each one of them. However, the migration heuristic presented in this section is limited to migrations internal to the DVFS domain. Other works that handle task migration between heterogeneous cores, like Run-DMC (MÜCK et al., 2015), depict the performance impact of a migration in a heterogeneous scenario by estimating the Instructions per Cycle through a linear regression implemented to each heterogeneous combination. In their solution, a ratio between the frequency of the target core and the current core is used to model the effects of migrating a task between different

DVFS domains. Following this same principle, such a ratio can be used to extend the migration algorithm proposed here for scenarios with multiple DVFS domains. The ratio would act as an overall weight of the activity vector, weighing the task activity when considering a migration between CPUs with different frequencies. For instance, $t_j.activity = t_i.activity * (W_f * \frac{f_j}{f_l})$, where $t_i.activity$ is the sum of the activity vector of task $t_i$, $W_f$ is the weight profiled for the frequency ratio, and $f_j$ and $f_l$ is the target CPU frequency and source CPU frequency, respectively.

The ANN model proposed here is focused on predicting the utilization of a task in a lower frequency configuration. The focus of the ANN can be extended to predict task utilization into different configurations instead. By adding two new inputs to the architecture design, one covering the target CPU frequency and another one representing the target CPU architectural configuration, the ANN training can be extended with information of the execution of the tasks in several different configurations, and then learn the proportionality between the scenarios, both offline and online. Another approach closer to the Run-DMC (MÜCK et al., 2015) solution will be to profile the weights of each feature offline for each core combination and then apply such weights at the ANN inputs. However, this approach does not support online adaptation.

# 5 PROOF OF CONCEPT IMPLEMENTATION

In this section, the proof of concept implementation of the energy optimizer is presented. The selected platform tor this proof of concept implementation is a Cortex-A53 processor, a widely used processor for embedded applications. The Cortex-A53 has four homogeneous cores with a configurable execution frequency of 0.6GHz to 1.2GHz, an 8-stage pipeline with two issued instructions per cycle, a coherent Level 1 (L1) private Cache of 32KB (16KB for Instructions and 16KB for Data), a shared coherent L2 cache of 512KB. The processor has support to ARM PMUv3 architecture (ARM, 2016) providing six configurable channels and over 50 performance event counters, which will be used in this experiment to gather data in order to profile the tasks and define a sane performance state.

The platform was mediated by EPOS (UFSC/LISHA, 2019), a low interference Real-Time Operating System (RTOS) for embedded applications with support to multicore architectures. The Scheduler Framework and the Monitor API design presented in Section 3 is currently implemented in EPOS and were used to perform the data collection in this work. EPOS also presents support for Raspberry PI 3B (RASPBERRY PI FOUNDATION, 2019), a single-board computer that uses ARM Cortex-A53 processor, the platform which will be used for the evaluation of this work.

## 5.1 DATA COLLECTION

Following the goal of predicting tasks' performance demands in a new frequency configuration based on their current performance demand and their architectural performance trace, three OS counters have been monitored for timing trace: Running Thread ID, Thread Execution Time, and CPU Execution Time. The idea is to correlate the tasks' execution time into different scenarios (with and without shared resource contention) to map the more relevant performance counters in the architecture regarding its architectural phenomena that impact the execution time. From the hardware performance counters, all the 54 events available on Cortex-A53 PMU were monitored (e.g., Cycle Count, Committed Instructions, Branches Taken and Missed, Unaligned Loads and Stores, L1 (Instruction and Data) and L2 Cache Misses and Hits, Data Memory Access, L1 and L2 Cache Writebacks, Bus Accesses). A full list of the available counters can be found at Cortex-A53 Technical Reference Manual (ARM, 2016).

To make a cleaner and more reliable data collection, CPU0 is reserved for management operations only (i.e., Interrupt Handling), as the management operations can make the trace dirty and possibly lead to a misinterpretation further at the ANN model training. Thus, the platform is configured with two logical clusters of processors, the first one containing the Core 0, and the second cluster containing the remaining three cores scheduled with a partitioned algorithm (P-EDF).

### 5.1.1 Benchmarks

Table 2 – Task-sets configuration.

| Task-set | CPU | Period/WCET | Task |
|---|---|---|---|
| 1 | 1 | 500*ms*/100*ms* | T0 Bandwidth |
| | | 500*ms*/100*ms* | T1 Disparity |
| | 2 | 500*ms*/100*ms* | T2 Disparity |
| | | 500*ms*/100*ms* | T3 CPU Hungry |
| | 3 | 500*ms*/100*ms* | T4 CPU Hungry |
| | | 500*ms*/100*ms* | T5 Disparity |
| 2 | 1 | 250*ms*/50*ms* | T0 Bandwidth |
| | | 500*ms*/100*ms* | T1 Disparity |
| | 2 | 1000*ms*/200*ms* | T2 Disparity |
| | | 250*ms*/35*ms* | T3 CPU Hungry |
| | 3 | 125*ms*/20*ms* | T4 CPU Hungry |
| | | 250*ms*/100*ms* | T5 Disparity |
| 3 | 1 | 100*ms*/10*ms* | T0 Bandwidth |
| | 2 | 100*ms*/5*ms* | T1 Bandwidth |
| | | 1000*ms*/400*ms* | T2 Disparity |
| | 3 | 100*ms*/30*ms* | T3 CPU Hungry |
| | | 500*ms*/100*ms* | T4 Disparity |
| | | 250*ms*/60*ms* | T5 CPU Hungry |
| | | 1000*ms*/100*ms* | T6 Bandwidth |

To acquire a data-set for the multicore real-time embedded system, the proposed approach aims at a synthetic but representative task-set. The first task-set will be used for data collection for offline analysis of the architectural phenomena and the ANN train. This task-set is composed of distinct performance behavior to stimulate architectural features. We aim at analyzing the performance impact considering resource sharing contention between tasks when running parallel in a multicore embedded environment. The following tasks are considered:

- **Bandwidth** is a benchmark implementation based on (HEECHUL YUN, 2019). Bandwidth focuses on memory stressing and is tailored to constantly perform read and write operations in a data structure with at least the size of L2 Cache (512*KB* in our platform).

- **Disparity map** is a task from San-Diego Visual Benchmark Studio (VENKATA et al., 2009), representing a real workload task of embedded systems. Disparity Map is a widely used task for embedded vision applications in autonomous vehicles, like cruise control, pedestrian tracking, and collision control.

- **CPU Hungry** is a loop function executing mathematical operations using ALU. Our implementation is based on iterative Fibonacci.

Bandwidth and Disparity map were also used by Gracioli et al. in (GRACIOLI et al., 2019) to depict shared resource contention. Our main difference to Gracioli's approach is that we added a mathematical CPU-bound task. The synthetic task-set is composed of the three applications, varying the tasks that run in parallel to capture traces that depict the contention for a shared resource. Two combinations of parallel executions have been configured, one composed of Bandwidth, Disparity Map, and CPU Hungry, and the other one composed of 2 Disparity Maps and one CPU Hungry. A depiction of the first task-set is presented in Table 2.

In this sense, the performance variability can be captured during the different phases of the task-set execution, modeling variability with two different scenarios, one with high shared resource contention, and the other with low shared resource contention. The variability in the Disparity Map execution is also accounted for at the very own variability from task T1 and T5. T1 starts its execution earlier than T5 due to T0, its previous task, execution close to 10 ms less than T4. Thus, they will run most of their execution in parallel, but in slightly different phases of execution, creating different behaviors that are expressed by the performance counters (see Figure 16 (d), (f), and (g)).

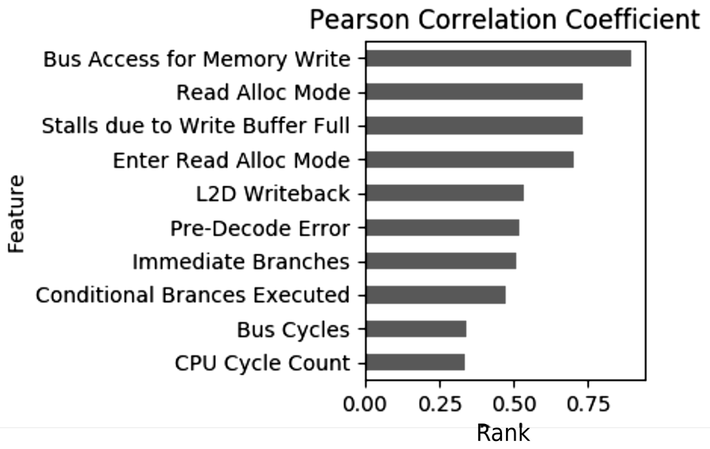
For the offline feature analysis, the monitor sampling rate selected for this task-set was $33Hz$, providing approximately three collections per job. This value has been selected as the architectural phenomena are being stimulated in a controllable scenario with small jobs. Additionally, as every task has the same priority (same period/deadline in P-EDF scheduling criteria), a task job is not preempted during its execution, avoiding trashing a capture.

A more variable scenario is presented for task-sets 2 and 3, the ones that will be used as validation scenarios for the proposed solution later. In task-set 2, for instance, due to the different periodicity of tasks, a single job of T2 will present different phases of execution as its execution is partially parallel to a bandwidth task, which will create a high contention scenario during the parallel execution. The same is true for T5 in the same task-set. For task-set 3, a more complex scenario is presented due to the high contention of the parallel execution of three memory-intensive tasks (the bandwidth tasks T0, T1, and T6). Moreover, in those task-sets, the concept of WCET, when extracted from the task running alone on the platform, is not representative when facing shared resource contention, with the utilization of tasks, and subsequently of a CPU, highly increasing (e.g., close to 4x T0 normal utilization in task-set 3 when running in parallel with T1). Thus, modeling all possible configurations manually is a timing consuming task, and some times infeasible for applications with several tasks and different phases of execution.

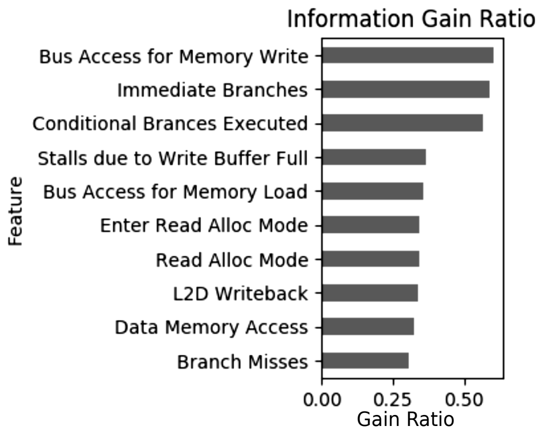
To further evaluate the proposed energy optimizer adaptability, explicit changes into task execution performance are also modeled as more variable scenarios. In those evaluation cases, called here *variable contention scenarios*, the bandwidth task behavior will periodically change from executing read and write operations into an array with the size of L2 cache to an array with the size of L1 cache. This will further stimulate the variability in the task-set execution (more details over these task-sets are given in Section 5.2.2), especially due to the fact that the memory hierarchy is the main source of shared resource contention in the target architecture.

### 5.1.2 Feature Selection Results

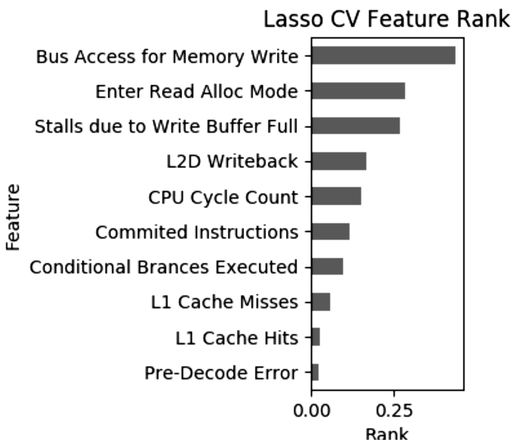
Each of the three algorithms considered uses a slightly different relevance metric, so the identification of the feature-set that most closely expresses changes in performance is not directly derivable from a simple intersection of their results. The 10 most relevant performance counters according to each applied technique are depicted in Figure 15 (a), (b), and (c), for Pearson's Correlation Coefficient, Information Gain Ratio, and Lasso CV, respectively. As a PCC inverse correlation is as relevant as a positive correlation, the absolute PCC value was selected for analysis.



(a) Pearson's Correlation Coefficient Results.



(b) Information Gain Ratio Results.



(c) LASSO Results.

Figure 15 – Ten most relevant features according to the feature selection techniques used.

The first conclusion is the unanimity that the number of **Bus Access for Memory write** is the most related feature to the performance loss presented. Following the analysis, **Stalls due to Write Buffer Full** and Read Alloc Mode counters (Enter Read Alloc Mode and Read Alloc Mode) are present in all three feature selection techniques as one of the most relevant counters. However, they present a high PCC to each other, about 95% correlation between the two Read Alloc Mode counters, and 92% between Read Alloc Mode counters and Stalls due to Write Buffer Full. In this way, only one will be selected to avoid multicollinearity and increase the amount of information available for the ANN model by possibly adding another counter. A graphical representation of the three counters is depicted in Figure 16 (a), (b) and (c).

The **L2D Writeback** counter also presented a high ranking in all three feature selection methods and a PCC of approximately 85% to Bus Access for Memory Load. Thus, only L2D Writeback has been selected. The performance trace of L2D Writeback is depicted in Figure 16 (d). The next counter to be evaluated is **Immediate Branches** (depicted in Figure 16 (g)), which presented a high information gain ratio rank and is also relevant through PCC. Moreover, PCC also pointed to a correlation of approximately 92% between Immediate Branches and Committed Instructions, counters that presents lower values in contention scenarios. Besides Lasso CV not presenting Immediate Branches as one of the best features, Committed Instructions was, thus, corroborating with the importance of this counter for the desired prediction model. In this way, only **Immediate Branches** has been selected.

**CPU Cycle Count** and Bus Cycles are both relevant features on PCC selection. However, only CPU Cycle Count is present on Lasso CV selection, which is justified by a correlation greater than 99% of those counters on PCC. However, the counters can still be useful on a composed analysis of the counter set, for instance, as in Lasso CV. CPU Cycle Count has already been used for performance prediction in other works (DAS et al., 2015). The performance trace of CPU Cycles is depicted in Figure 16 (e). The last counter to be selected is **L1 Cache Hits**, which can be useful in a composed analysis to depict tasks that request for shared memory (low L1 Cache Hits accounted). As L1 Cache Misses are not accounted for if a subsequent miss also happens in cache L2, and L2 Writeback already contributes with L2 Cache information (approximately 88% correlated through PCC to L2 Cache Misses). Thus, even though it presents a higher ranking in Lasso CV, we have selected L1 Cache Hits instead. Nonetheless, L1 Cache Hits is also highly correlated to Data Memory Access, approximately 99% through PCC, a counter selected through Information Gain Ratio. The performance trace of L1 Cache Hits is depicted in Figure 16 (f).

The resultant set of features is composed of Bus Access for Memory write operations Figure 14, Stalls due to Write Buffer Full Figure 16 (a), L2D Writeback Figure 16 (d), CPU Cycle Count Figure 16 (e), L1 Cache Hits Figure 16 (f), and Immediate

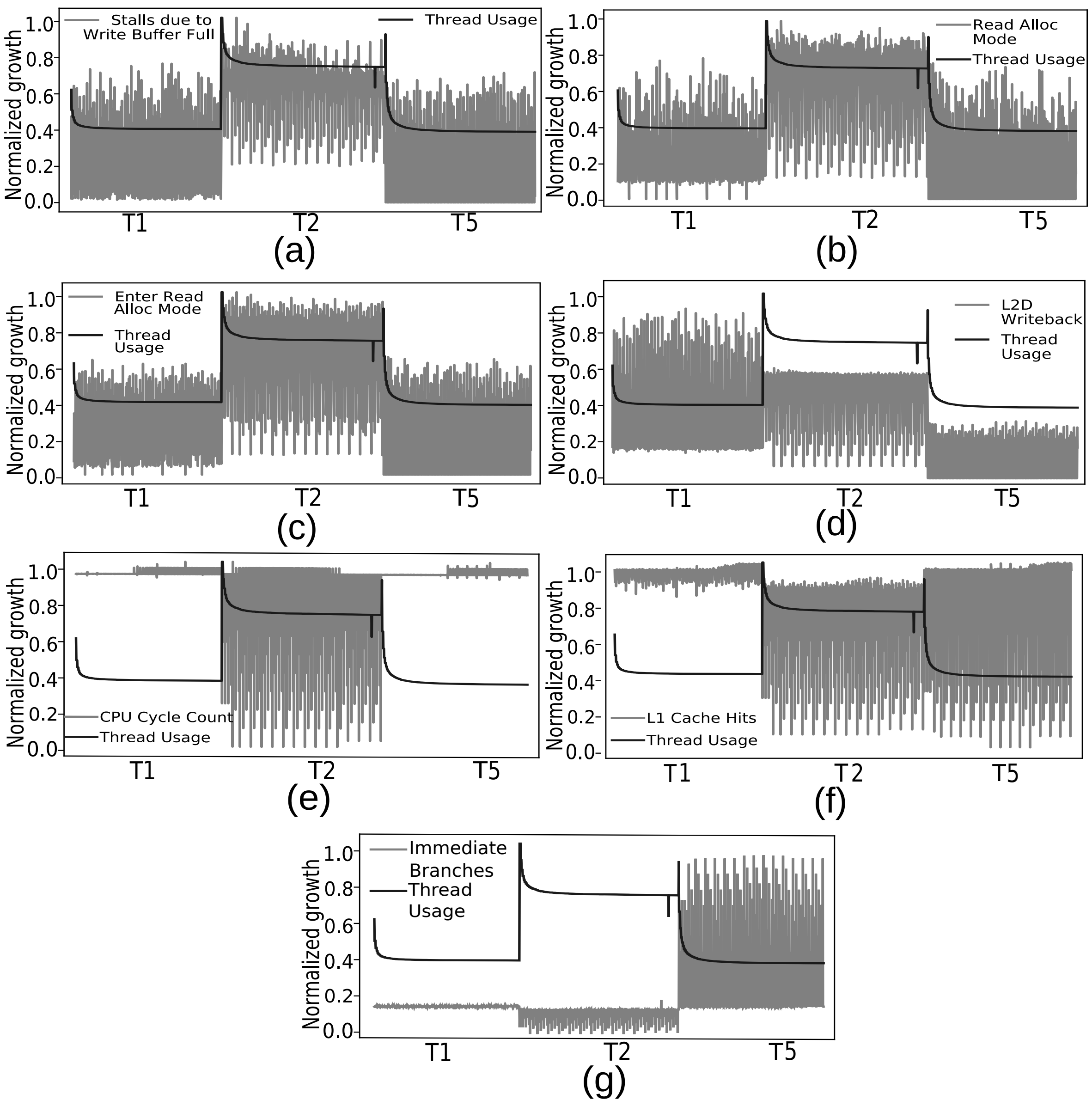


Figure 16 – Stalls due to Write Buffer Full and Read Alloc Mode performance Counters trace (highly correlated counters following PCC): (a) Stalls due to Write Buffer Full, (b) Read Alloc Mode, and (c) Enter Read Alloc Mode. And Performance Counters trace of (d) L2D Writeback, (e) CPU Cycles, (f) L1 Cache Hits, and (g) Immediate Branches.

Branches Figure 16 (g).

The selected performance counters were chosen for their ability to capture the current utilization of tasks from the perspective of the performance expressiveness. Counters like CPU Cycle Count and Immediate Branches, for instance, can represent the throughput demands in a CPU regarding CPU-bound behavior. Memory hierarchy related counters (e.g., Bus Access for Memory Write, Stalls due to Write Buffer Full, L2D Writeback, and L1 Cache Hits) can be used to describe non-CPU-bound behavior, for instance, providing information regarding access to high latency resources that can be caused by shared resource contention from both intra- and inter-CPU tasks. A composite analysis is expected to provide sufficient information to depict performance losses, and subsequently, CPU's frequency demands. For instance, an abrupt decrease in CPU-bound related counters along with an increase in Memory Hierarchy access. Thus, they provide representative information over performance for a ML technique (i.e., ANN) to be capable of learning to accurately predict the impact a frequency scaling will have on the performance of a task, guiding the DVFS to enable energy-savings accounting for real-time constraints.

### 5.1.3 ANN Configuration

The proposed ANN model serves as a predictor for the impact of a frequency scaling at the resource utilization of tasks, using as input the performance trace of the task and its utilization. The predictor outputs are taken per CPU and analyzed by the energy optimizer for DVFS control.

For instance, the predictor is meant to learn that, even though a CPU idle time seems to be suitable for a frequency reduction, a performance trace composed by a lower growth of CPU Cycle Count, Immediate Branches and/or L1 Cache Hits, along with higher growth of Bus Access for memory write operations, Stalls due to Write Buffer Full, and/or L2D Writebacks points to a low throughput due to shared resource contention, and, given the current frequency and task execution time, it can more accurately estimate that the performance demands for this task in a lower frequency level will differ from one with higher growth of CPU Cycle Count, which has higher throughput. Thus, our proposed ANN can learn to differentiate the task behavior by encompassing the growth of each selected performance counters and the current performance demands. Therefore, we aim at both online and offline learning, whereby offline learning we achieve an initial broader knowledge for initial stages and a tuning of the configuration of the nodes, avoiding the high overhead of complex ANN architectures and cold start issues. And through online learning, we achieve a continuous and adaptable optimization for the running task-set.

The validation task-set (task-set 2 at Table 2), was used to evaluate the model accuracy during the tuning process. More specifically, the model capability of adaption to new scenarios. Thus, the validation process is a simulation of the online learning (see Algorithm 7), using previously collected data and re-training each CPU ANN model whenever the current prediction deviates from the measurements by more than a user-defined deviation threshold *DT* (i.e., 2% of deviation). The tuning process evaluates both the ANN topology and learning rate, selecting the model that requires fewer re-training rounds to adapt to the new scenario. The best results have been achieved with a simpler topology and a small learning rate, as by increasing any of both configurations, the ANN model rapidly over-fits to the training data-set. For more details over the tuning of the ANN topology, see Section 4.2.3. The configuration that lends better results during validation is depicted in Figure 17 and explained in detail below. The selected model used a learning rate of 0.35 and provided an offline performance as follows: a minimum accuracy of 91.8% (a maximum of 35 trains required) for a deviation threshold *DT* of 2% and an average accuracy of 95% considering the validation data-set.

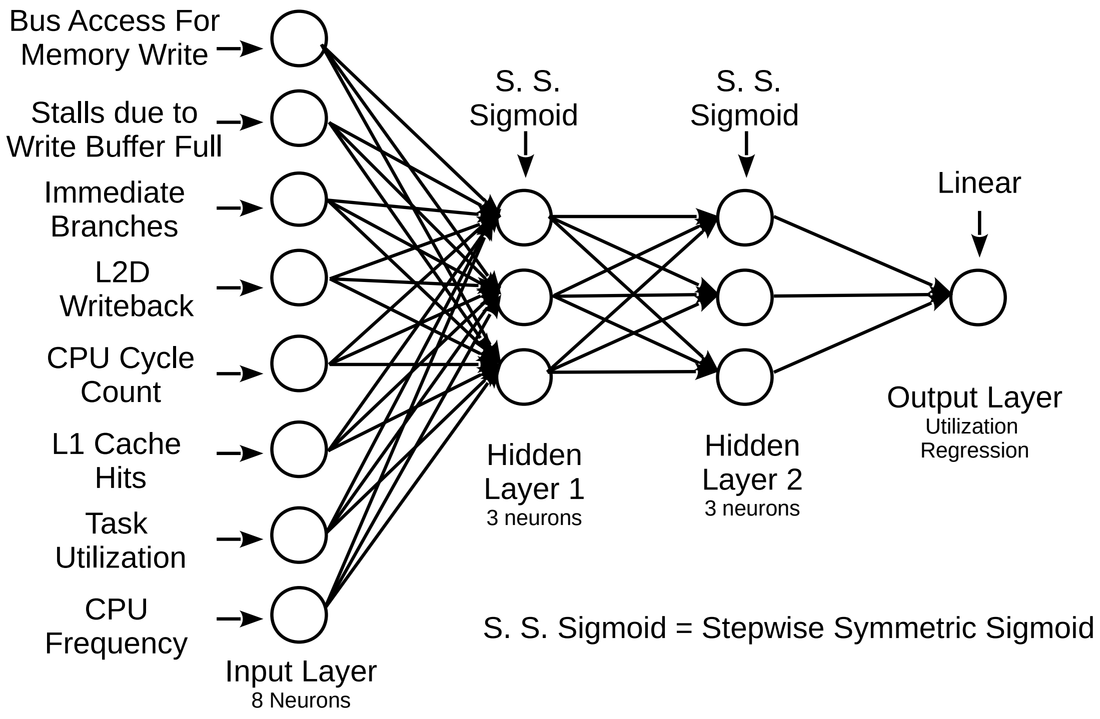


Figure 17 – Artificial Neural Network final architecture.

- **Input Layer** is composed of eight elements: six performance counters, representing the task's performance trace and utilization in the last global hyper-period of execution and the current frequency level. Also, the inputs are equally normalized as an average to account for different tasks periods and represented on the same scale as the output {0,1} with Min-Max Normalization. In our experiments, we have empirically selected a normalization period of 100*ms*.
- **Output layer** is composed of one output neuron representing the utilization prediction for the task if the frequency is scaled, accounting in this prediction, the shared resource contention through the input variables. Moreover, the OS overhead is naturally accounted into the task trace and execution time measuring. The output neuron result will be represented in the range {0,1}, directly mapping to the task utilization between 0% and 100%.
- **Hidden Layer** is composed of two layers, with three neurons each, using the Step-wise Symmetric Sigmoid activation function.

#### 5.1.3.1 ANN Implementation Library

The ANN library chosen for this project was Fast Artificial Neural Network (FANN) (STEFFEN NISSEN, 2019). FANN neural network library, which implements multilayer artificial neural networks in C with support to the necessary functions required by the proposed ANN design, especially the incremental training and the linear approximations of activation functions. The ANN model was first trained offline with the runtime data collected with the Monitor, and then the resultant model was ported to EPOS.

For this to be possible, the necessary structure for FANN to execute was linked to the EPOS as part of the Monitor to be used for `Criterion::charge()` implementation. The ported methods were `FANN::fann_run()` and every method used to execute a FANN, along with an adaptation of `FANN::fann_create_from_file()`, that creates a FANN with the results of the training, called `FANN::create_from_config()`. As the ported FANN model is required to adapt at run-time through incremental training, the method `FANN::fann_train_data_incremental()` and every method used by this function were ported to EPOS too. The EPOS code with the FANN port is available at https://gitlab.lisha.ufsc.br/epos/epos/tree/ann-energy-optimizer.

Once the methods were ported to EPOS, the final model configuration is ported by parsing the configured network parameters and then configuring the FANN at compile time to be initialized as an EPOS component. Moreover, as the ANN is trained with scaled data, a Min-Max scaling function was added as a preprocessing of the input vector.

## 5.2 PROPOSAL EVALUATION

In this section, the evaluation of the proposed energy optimizer, including non-intrusiveness analysis, method effectiveness regarding **energy consumption**, and the **ANN model predictions and online adaptations**. The evaluation uses three case studies composed of different configurations, especially to evaluate the ANN adaptability through online learning and the task migration heuristic proposed. Moreover, the evaluation is done in a real multicore embedded platform, a Cortex-A53 processor, the same used for feature extraction and offline training. Each task-set configuration is depicted in Table 2.

First, an overhead analysis of our approach is presented to corroborate the claim of low-intrusiveness for the Monitor API and the energy optimizer. The overhead has been measured by comparing executions with and without the Monitor API and energy optimizer enabled. Then, the analysis of the energy consumption of the three task-sets is presented, comparing the proposed approach to a scenario with no optimization and with Linux's Power Governors. Next, an overview of the energy optimizer actuation is presented, covering the ANN predictions and online learning, and the task migration execution. Lastly, the three task-sets are extrapolated regarding their behavioral variation to evaluate the online learning capabilities of the energy optimizer approach, as they do not necessarily represent realistic variations on embedded systems, like autonomous vehicles task-sets (GRACIOLI et al., 2019).

### 5.2.1 Overhead Analysis

The energy optimizer design accounted for non-intrusiveness from the features monitoring up to the actuation execution. The main design decisions that lead to the low-intrusion are the following: the initialization along with the OS, contention-free monitoring buffers, polling instead of interruptions for both monitoring and actuation (collection, charging, and award), contention-free predictor structures, and limited actuation windows, where the energy optimizer actuates only at global hyper-period to avoid misleading a prediction or affecting a task in the middle of its execution due to a frequency scaling or preempting it due to migration.

The monitoring overhead was evaluated first during the first implementation of the monitor (HORSTMANN et al., 2019), in an Intel i7-2600 processor with 8 logical cores with 2-way hyperthreading (i.e., 4 physical cores), L1 cache with 4 x 64KB 8-way set associative, L2 cache (non-inclusive) with 4x 256KB 8-way set associative and L3 cache (inclusive) with 8MB 16-way set associative. The platform was mediated by EPOS, with the OS configured to produce no additional overhead besides the one caused by the scheduler. The monitor was configured to sample: two OS counters, deadline misses (OS) and idle time (OS); two PMU counters, committed instructions per core (PMU) and per-thread (PMU); and two hardware sensor, the PP0 and PKG, core and entire chip energy consumption sensors, obtained through Intel RAPL interface (INTEL CO., 2019). The sampling rate was set to 100Hz, incurring into a maximum observed overhead of 43 ms for a set of executions lasting 1 minute (corresponding to 0.0718%). Moreover, the jitter imposed in the architecture was measured per tasks by capturing the time-stamp of each thread release (at `Semaphore::v()` in Figure 9) and another one when it effectively resumes execution (after `Periodic_Thread::wait_-next()`). The variation was calculated using these time-stamps aligned for both cases, with and without the monitor. The maximum observed difference on the jitters was of 38.81$\mu$s, the minimum was 0.78$\mu$s, and the standard deviation was 11.22$\mu$s. Thus, the overhead was non-intrusive into tasks' execution behaviors.

Considering such a low intrusion, in the Cortex-A53 platform, the Monitor API is considered non-intrusive, especially when running in its *Per Task Monitoring Functionality* (see Section 3.2.2), where a collection is only executed when a thread leaves the CPU. In this sense, the overhead of the energy optimizer will consider its impact on tasks' execution time, mostly incurred by the low impact on data and instruction cache's locality. The overhead was evaluated through three aspects: (i) ANN run and train overhead; (ii) energy optimizer online overhead; (iii) Impact on Tasks execution time.

The ANN average overhead, for both prediction and training, was measured on each available frequency level through 260000 iterations. The time necessary to run a prediction ranged from 0.9027$\mu s$ to 2.5657$\mu s$ on average at the highest and lowest frequency, respectively. And the necessary time to execute the incremental train online over one sample ranged from 1.8073$\mu s$ to 5.1318$\mu s$ on average at the highest and lowest frequency, respectively. Moreover, the necessary time to run an iteration of the ANN is related to its topology (i.e., number of layers and neurons per layer). In a similar scenario, using a more complex ANN over the same Monitor API design, but focusing on anomaly detection based on performance traces (HOFFMANN et al., 2019), the same ANN port and the same hardware platform were used in a different configuration. On an ANN using the same activation functions, but with 5 neurons on the input layer, two hidden layers with 10 neurons each, and one neuron at the output layer, the ANN predictions required approximately 4$\mu s$ to run in the highest frequency possible. Even though 4$\mu s$ is still a small overhead, the increase of complexity in the ANN model must be carefully done, as in this case, by increasing the number of neurons in approximately 1.73 times (from 15 to 26) increased the prediction run time in approximately 4 times.

For the energy optimizer online actuation overhead, a limit of 8 rounds of training at each training section (every time a frequency scaling occurs) was selected. This is a configurable feature of the energy optimizer, and can be customized considering the affordable overhead the system can manage. In this configuration, the maximum measured overhead during training rounds was 92$\mu s$ (in a global hyperperiod of 500000$\mu s$), which includes `Criterion::collect()` complete execution (Algorithm 1) and `Criterion::charge()` training execution (Algorithm 2 from lines 9 to 29). For voting rounds, which includes `Criterion::collect()` complete execution (Algorithm 1), `Criterion::charge()` prediction execution (Algorithm 2 from lines 2 to 8), and `Criterion::award()` voting (Algorithm 3 lines 2 to 11) presented an average overhead added per activation of 15.3585$\mu s$. All the aforementioned measurements were captured in a scenario with two tasks per CPU.

Lastly, to measured the impact of the energy optimizer execution, the actuation was disabled to provide the same scenario with and without the presence of the energy optimizer procedures, thus, providing a measure of the intrusiveness added by the energy optimizer into each task type execution time. For CPU-Hungry tasks, which are completely CPU-bound tasks, no intrusion was added to their execution time, thus highlighting the low impact into instruction caches locality the energy optimizer adds. For Disparity, the task representing the class of applications with a mid-term behavior regarding CPU-/Memory-Boundness, the average overhead added to its execution time was 0.0365%. And for bandwidth, the task representing fully memory-bound tasks, the average overhead added to its execution time was 0.1791%.

### 5.2.2 Experimental Results

To provide a baseline for the evaluation of the proposed energy-optimizer, the energy consumption of the task-sets presented in Table 2 was first measured in a run-to-halt approach over the same OS (EPOS), where the task-set is also limited to run only at the CPUs 1, 2 and 3. The run-to-halt approach is focused on maintaining the highest frequency to increase the time the CPU stays in the halt state, thus, saving energy with DPM. The isolation of CPU 0 from this comparison was made as a design decision (see Section 4.1). Furthermore, this CPU has a low utilization and is only responsible to issue the DVFS commands and to run the Task Migration heuristic, where the Task Migration heuristic is an algorithm with a complexity dependent on the number of Tasks and the number of cores inside a DVFS domain. Nevertheless, the energy consumed by CPU 0 is also accounted for, as the energy consumption of the platform (Raspberry Pi 3b) is measured as a whole for evaluation purposes. Due to its low utilization, a solution to avoid the isolation of 1 CPU could be using a micro-controller to handle the command issues and the Task Migration Heuristic.

To build a more extensive baseline the energy optimizer performance is also compared to RaspberryPi OS with Linux Kernel version 5.4. In this scenario, the Linux energy consumption is measured by computing the same workload computed on EPOS in three different configurations: run-to-halt and CPUFreq's OnDemand and Conservative. On Linux, to provide the same computing power in both scenarios, only 3 out of the 4 available CPUS were used. The CPUFreq's OnDemand and Conservative power-governor policies were used. Both policies check the CPU-utilization statistics over the last period and then set the CPU Frequency accordingly (DOMINIK BRODOWSKI, 2017). The Conservative policy increases and decreases the CPU frequency more slightly than OnDemand, which jumps between the minimum and maximum frequencies. To provide an equivalent scenario in Linux, the task-sets were scheduled using Linux's SCHED-DEADLINE, to provide a real-time environment, and the amount of CPUs in which the task-set can run was restricted to CPUs from 1 to 3, using Linux's cpuset.

The proposed energy optimizer effectiveness is evaluated through the energy consumption reductions achieved and the ANN online adaptation capability. Figure 18 presents a comparison of the energy consumption on each scenario, with the Linux OS in a run-to-halt approach, along with OnDemand and Conservative power governors presented as a baseline. The energy optimizer is implemented in a Real-Time OS (RTOS) (i.e., EPOS) considering a Safety Margin of 5%. The RTOS is focused on handling the given task-set and managing the performance monitoring and actuation, thus, presenting a lower energy consumption when compared to Linux. During the experiments, the energy optimizer always reaches the lowest frequency possible for each task-set and achieves, on average, 24.97% of energy consumption reduction when compared to the RTOS, and 68.91%, 64.70%, and 65.13% when compared to Linux run-to-halt, OnDemand, and Conservative, respectively.

Moreover, the energy measurements were taken using an energy consumption monitor at the platform power-source, thus, accounting for the whole platform energy consumption while the energy optimizer actuation only accounts for CPU energy consumption optimizations. In this way, to reduce this element impact at the energy consumption comparisons, a baseline of energy consumption for the platform was measured using an execution with idle tasks only. Thus, every CPU remains halted during execution, being waked-up periodically by the system alarm. The energy consumption was measured for the same amount of time the task-set used for evaluation execute, and presented a consumption of 81.702$J$. Then, for every analysis presented here using EPOS, this was considered the system base energy consumption.

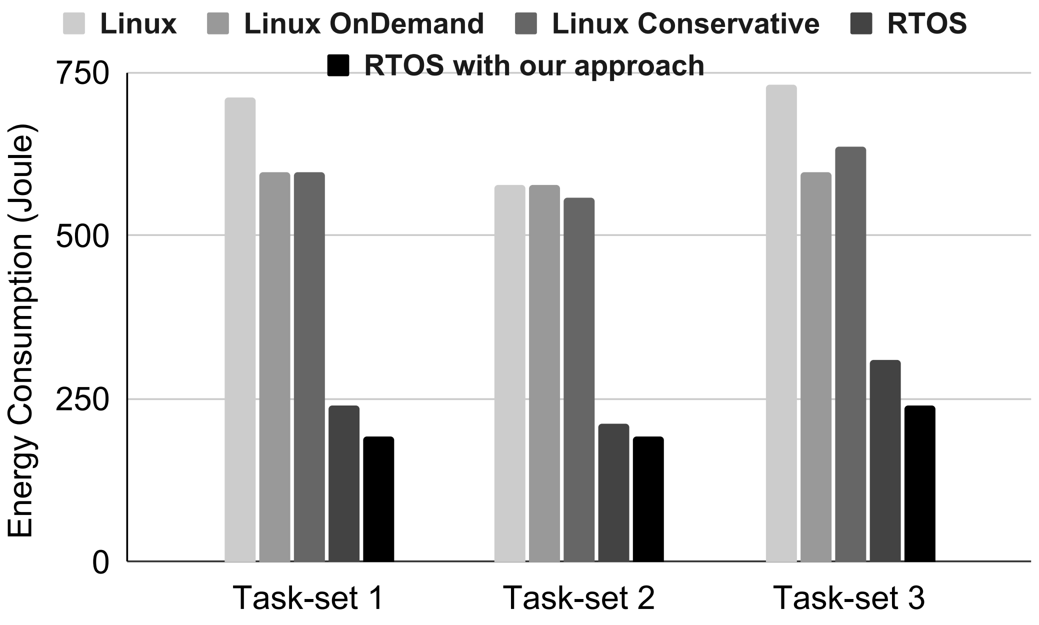


Figure 18 – Task-sets energy consumption under different optimizers.

For the first task-set, the same one the ANN has been trained with the energy optimizer presents an average energy saving of 30.30%. A depiction of the task-set execution is presented in Figure 19, where the bar plot in the background of the figure represents the CPU cluster frequency. The lines represent the CPUs' utilization related to the right axis, and the stars the utilization prediction of the CPUs' if a frequency scaling is executed. The task-set converges to a stable configuration at the lowest frequency possible in this architecture (i.e., 0.6 GHz), without requiring any migration. The CPU with the highest load presents a slack of 19.65% of utilization. The variability in the multicore architecture on its own demanded online training of the ANN model

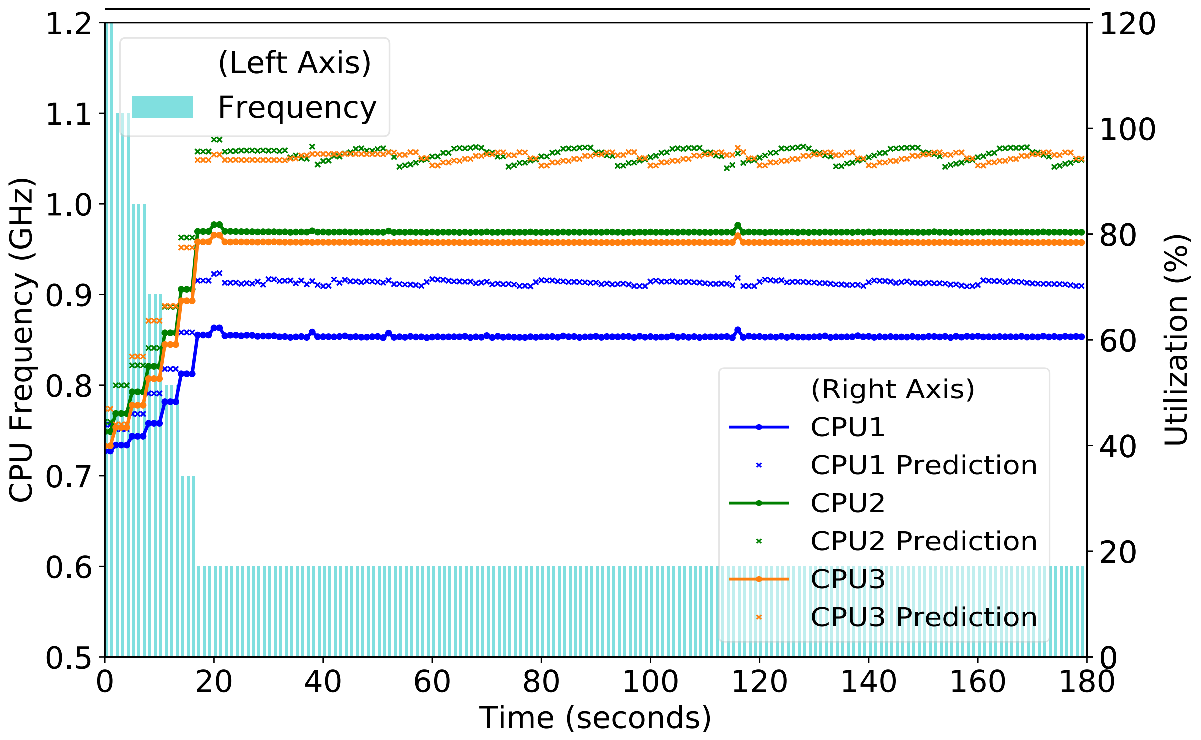


Figure 19 – Energy Optimizer actuation on Task-set 1.

even for the task-set it was originally trained offline. The ANN started with a deviation in the utilization prediction of 3.85%, 1.59%, and 3.52%, for CPU1, CPU2, and CPU3, respectively. The deviations at the last actuation were 0.49%, 1.14%, and 1.07%. The ANN was retrained online whenever a prediction deviates for more than 2%, this fitting the model to the current task-set behavior.

For the second task-set, the energy optimizer presents an average energy saving of 14.81% and improve the load balance in the task-set, reducing the standard deviation of the CPUs' utilization by 22.74%. A depiction of the task-set execution and the ANN predictions is presented in Figure 20, using the same semantics described for Figure 20, now with the addition of the fourth axis at the top. The dashes and numbers at this axis represent the migrations executed, described in the legend at the bottom. The task-set converges to a stable configuration at the lowest frequency possible in this architecture (i.e., 0.6 GHz) after two migrations. In the final configuration, the CPU with the highest load presents a slack of 5.2% of utilization, respecting the safety margin of 5.0%. The ANN started with a deviation in the utilization prediction for CPU1, CPU2, and CPU3 of 8.05%, 0.62%, and 3.58%, respectively. At the last actuation, the ANN presented deviations of 1.73%, 3.77%, and 7.01% deviation. It is worth mentioning that, after the first migration, the deviation increased up to 6.57% for CPU1, 7.93% for CPU2, and 12.2% for CPU3. The ANN online training improved the prediction deviation, fitting the model to the task-set behavior, especially after being affected by migrations, for instance, because a shared resource contention was solved.

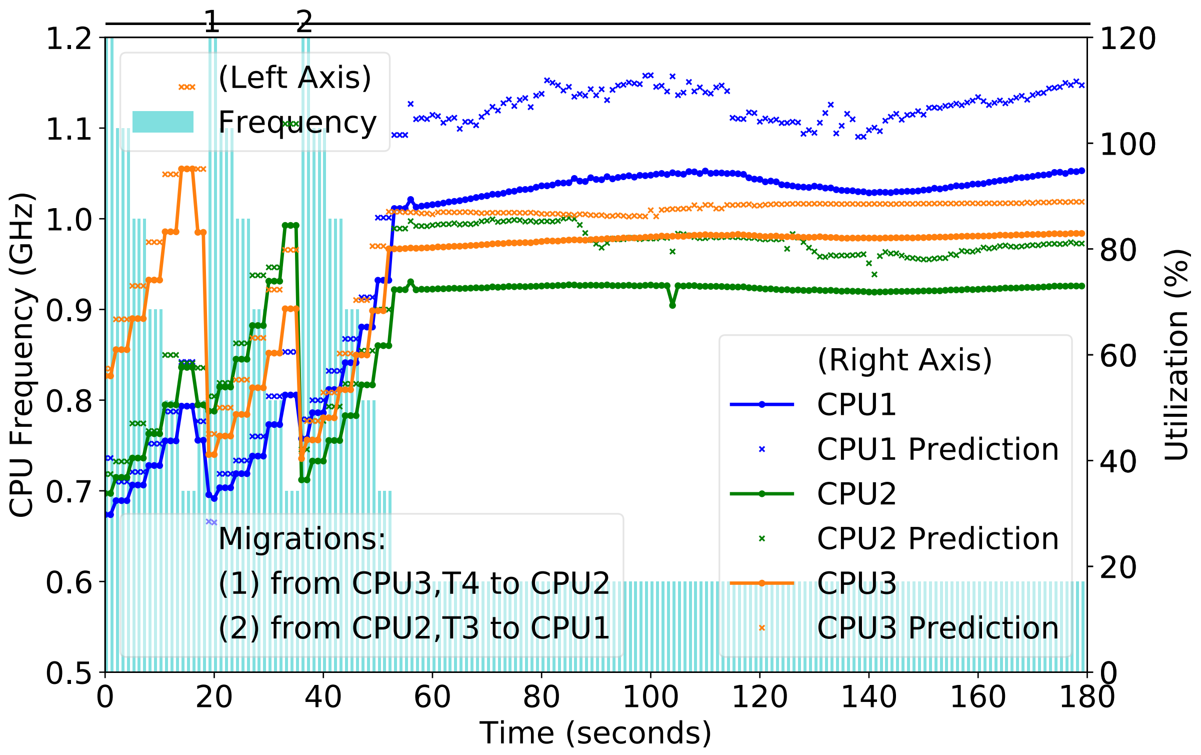


Figure 20 – Energy Optimizer actuation on Task-set 2.

For the third task-set, which represents a more complex scenario in terms of shared resources contention (due to the three bandwidth tasks), the energy optimizer presents an average energy saving of 14.81% and improve load balancing, reducing the standard deviation of the CPUs' utilization by 68.00%. A depiction of the task-set execution and the ANN predictions is presented in Figure 21 using the same semantics described for Figure 20. The task-set converged to a stable configuration at the lowest frequency possible for this task-set (i.e., 0.8 GHz) after four migrations and one revocation. The safety margin *SM* of 5% was preserved, as the CPU with the highest load presented a slack of 16.12%. The ANN started with a deviation in the utilization prediction for CPU1, CPU2, and CPU3 of 0.16%, 21.64%, and 11.66%, respectively. At the last actuation, the deviations decreased to 4.70%, 0.25%, and 4.71% (the deviation for the last actuation was 17.91%, but the training log reported that the error was reduced to 4.71% after the online training). It is worth to mention that after each migration that affected the behavior of the tasks, the deviation at the related CPUs increased. The maximum deviation for CPU1, CPU2, and CPU3 after the initial actuation were 16.52%, 15.30%, and 17.91%, respectively, and the ANN quickly adapted to the new behavior.

The importance of activity vector weight profiling (described in Section 4.1.3.1) for critical scenarios such as the one represented by this task-set 3 is illustrated in Figure 22. It depicts the task-set profiling and shows undesirable migrations based on the initial weights of 0.5. For instance, near 50 seconds of execution, migration can be seen to have caused a deadline miss. Certainly, increasing the safety margin beyond the 5% used in this case could also prevent faulty migrations, but only profiling will allow the system to learn bad combinations of tasks that must be avoided during ordinary operation. The activity weights profiling usually also reduces the number of migrations, reducing the overhead while leading to an optimal task allocation.

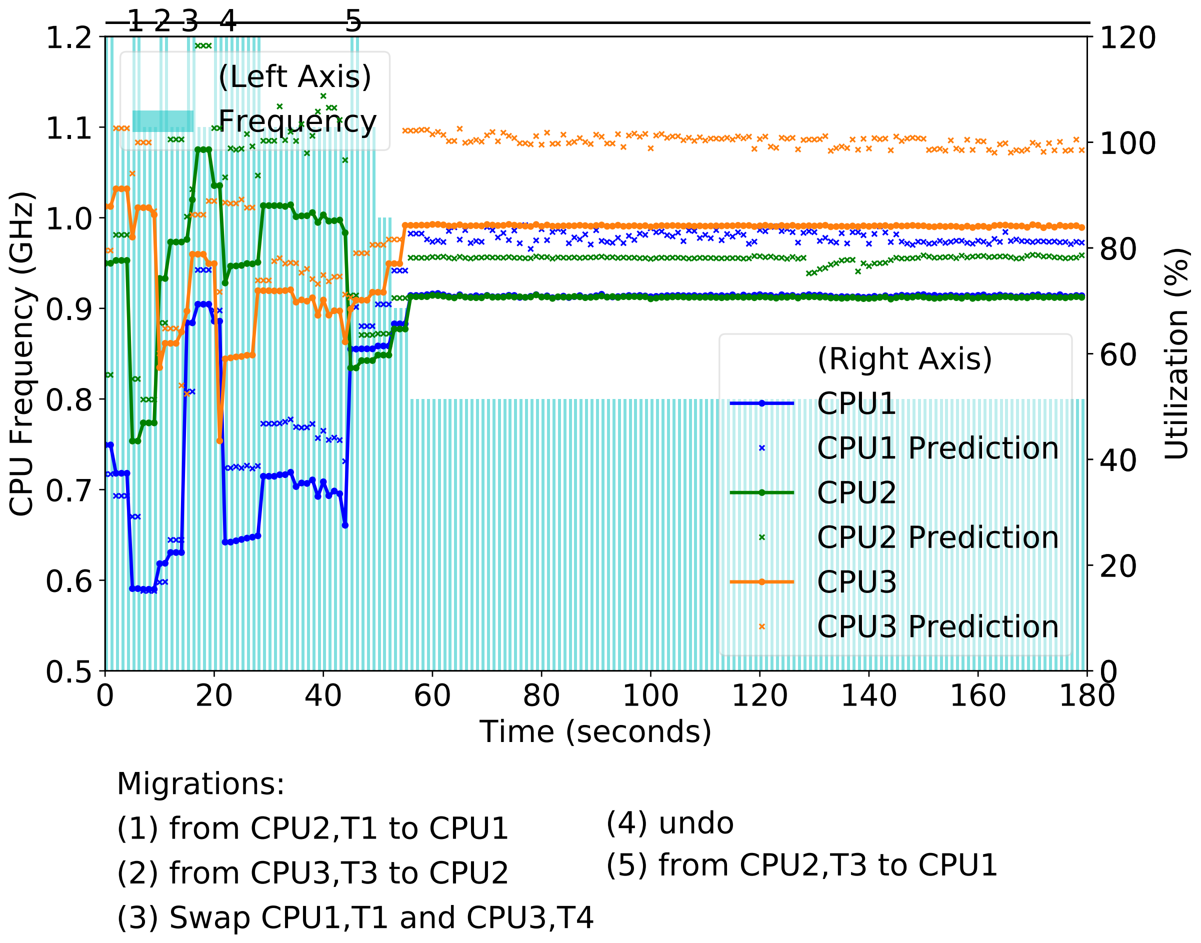
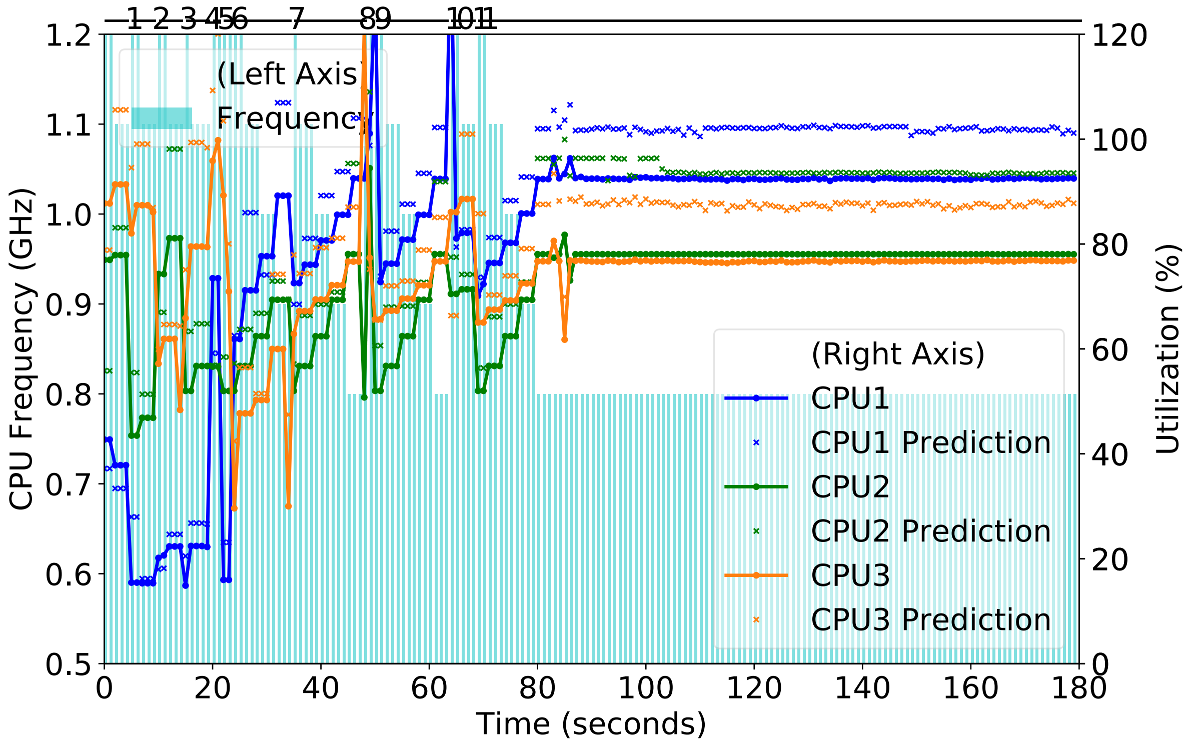


Figure 21 – Energy Optimizer actuation on Task-set 3.

To further evaluate the energy optimizer capabilities, each one of the task-sets has been executed with two additional configurations, especially to make the task-set vary its contention on Bandwidth and Disparity tasks. The two new configurations work as follows: i) the bandwidth task runs in a reduced data-set, performing a loop of reads and writes into a 16KB data structure (the size of the data cache level 1 in the Cortex-A53 available at the Raspberry Pi 3B), reducing the impact on shared resources with other CPUs, while still affecting the cache level 1 locality in the CPU it runs. ii) the bandwidth task changes its behavior periodically (every 90 jobs), providing an extremely variable scenario to evaluate the predictor and the migration technique capabilities. For those two new scenarios, the energy optimizer improved energy consumption by 19.59% on average. A depiction of the energy consumption for each case is depicted in Figure 23 for the task-sets with reduced Bandwidth data-set, and in Figure 24 for the task-sets with variable Bandwidth data-set.



Migrations:
(1) from CPU2,T1 to CPU1
(2) from CPU3,T3 to CPU2
(3) Swap CPU2,T2 and CPU3,T5
(4) Swap CPU1,T1 and CPU2,T3
(5) undo
(6) from CPU3,T2 to CPU1
(7) from CPU1,T0 to CPU3
(8) from CPU3,T6 to CPU2
(9) undo
(10) Swap CPU2,T3 and CPU3,T0
(11) undo

Figure 22 – Energy Optimizer actuation on Task-set 3 without predefined weights for migrations.

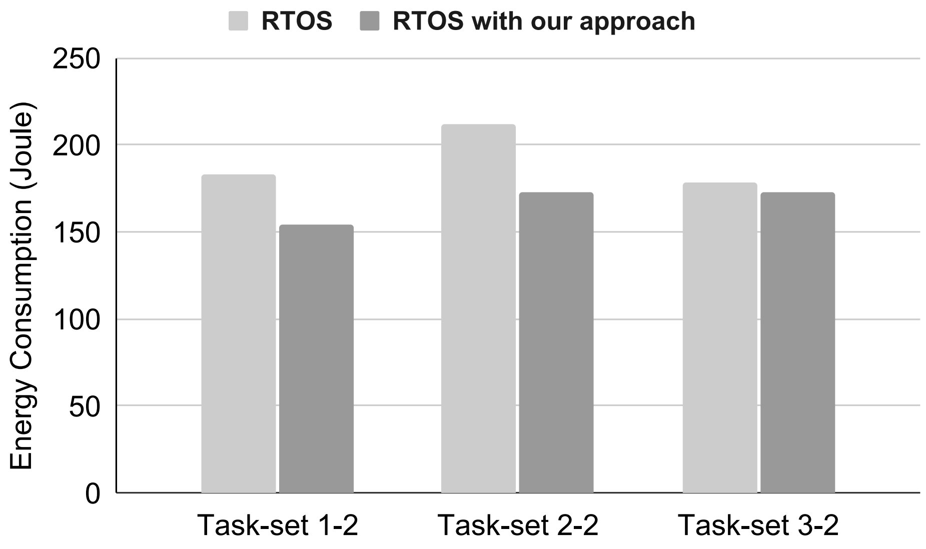


Figure 23 – Task-sets with reduced Bandwidth data-set energy consumption for the RTOS and the Energy Optimizer running on the RTOS.

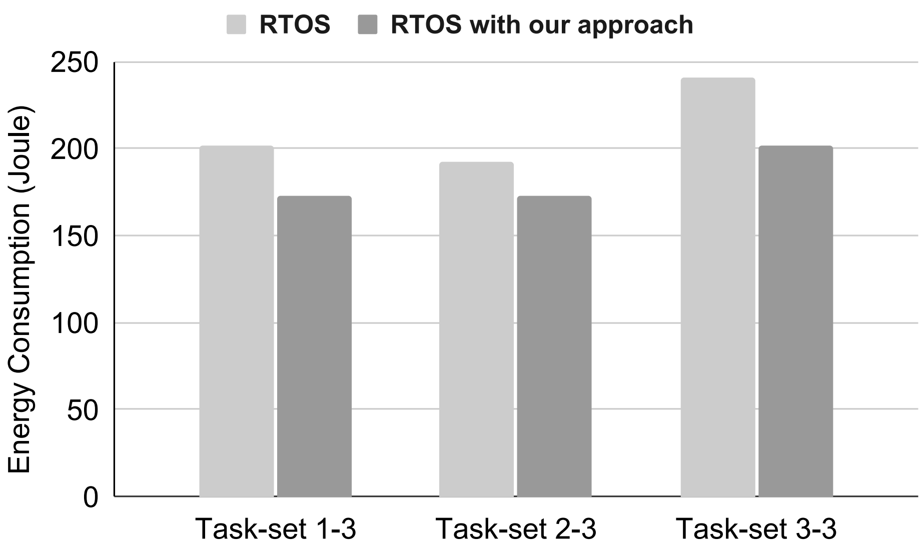


Figure 24 – Task-sets with Bandwidth configured with behavior variation data-set energy consumption for the RTOS and the Energy Optimizer running on the RTOS.

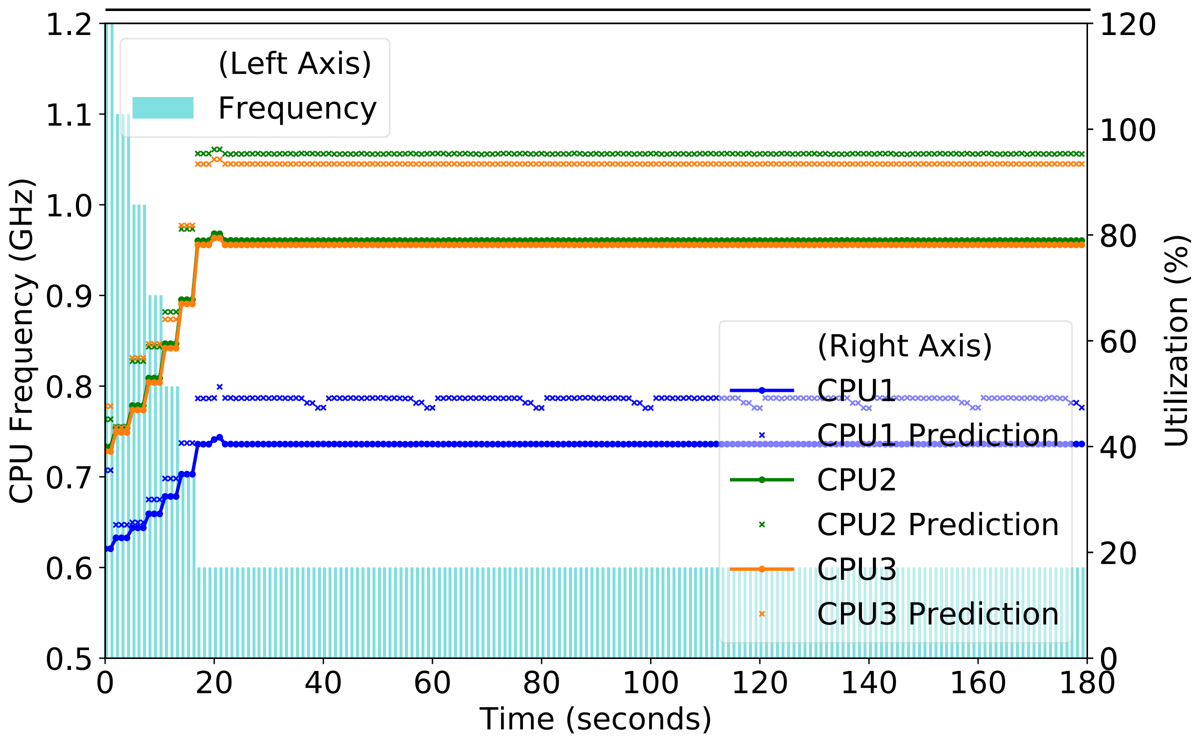


Figure 25 – Energy Optimizer actuation on Task-set 1 with Low Contention Bandwidth.

For the first task-set in the low contention configuration, the energy optimizer presented an average energy saving of 28.57%. A depiction of the task-set execution is presented in Figure 25, following the same semantics as the previous plots. The task-set runs in the same way as before, but now with a lower utilization in task T0 (Bandwidth), also incurring in a lower contention for the other tasks. This configuration quickly converges to a stable configuration at the lowest frequency possible in this architecture (i.e., 0.6GHz), and it does without requiring any migration. In the final configuration, the CPU with the highest load still presented a utilization slack of 21.05%. In this scenario, the maximum prediction deviation presented by the ANN was 12.78%, 4.00%, and 4.97%, for CPU1, CPU2, and CPU3 during the first and second actuation. Note that CPU1 prediction, the one with the new behavior inserted, deviated in 12.78% with no training to the new configuration, which decreased to 0.55% through the online learning in the next prediction, showing a high level of adaptability. The maximum deviations at the last actuation (from 0.7GHz to 0.6GHz) were 0.22%, 2.22%, and 3.65%. It is worth mentioning that for CPU2 and CPU3, the lowest prediction error was during the frequency scaling from 0.9 to 0.8 GHz, presenting a deviation of 0.57% and 0.81%. As this task-set has a lower utilization in general, even when setting T0 to vary its performance demands periodically, the task-set still maintains the lowest frequency possible for this architecture, as depicted by Figure 26, where, with this variable behavior, the average energy saving achieved by the energy optimizer in this scenario was 24.00%. After 90 seconds of execution, task T0 (Bandwidth) changes its behavior to the one presented in the original task-set configuration (Figure 19), in which, as the CPU slacks already fit into the current configuration, does not impacts the energy optimizer actuation. For this scenario, as the actuation was done during the first behavior of T0, the one equal to the previous case, the deviations were very similar. The maximum prediction deviation presented by the ANN was 13.00%, 4.06%, and 4.97%, for CPU1, CPU2, and CPU3, respectively. And the maximum deviations at the last actuation (from 0.7GHz to 0.6GHz) were 0.11%, 1.86%, and 3.58%.

For the second task-set in the low contention configuration, the energy optimizer presented an average energy saving of 17.39%. A depiction of the task-set execution is presented in Figure 27, following the same semantics as previous plots. In a scenario where CPU1, the CPU running the Bandwidth task, has a lower utilization when compared to the original task-set. In the final configuration, the CPU with the highest load still presented a utilization slack of 20.29%, which was only achieved by combining the migration heuristic and the ANN predictor. In this sense, instead of requiring two migrations to reach the optimal configuration, this task-set required only one. It is possible to see that T4, the CPU-Hungry task, initially allocated to CPU 4, was migrated to CPU1, providing an improved load balance between CPUs and achieving the lowest frequency configuration possible for the architecture (0.6 GHz). In this scenario, after the migration, the predictions deviations for CPU1 utilization increased to almost the same deviation presented at the beginning (13.57%), as the CPU1 now includes tasks from the three task types. The maximum prediction deviation presented by the ANN was 13.95%, 7.23%, and 4.04%, for CPU1, CPU2, and CPU3, respectively. And the maximum deviations at the last actuation (from 0.7GHz to 0.6GHz) were 6.13%, 3.86%,

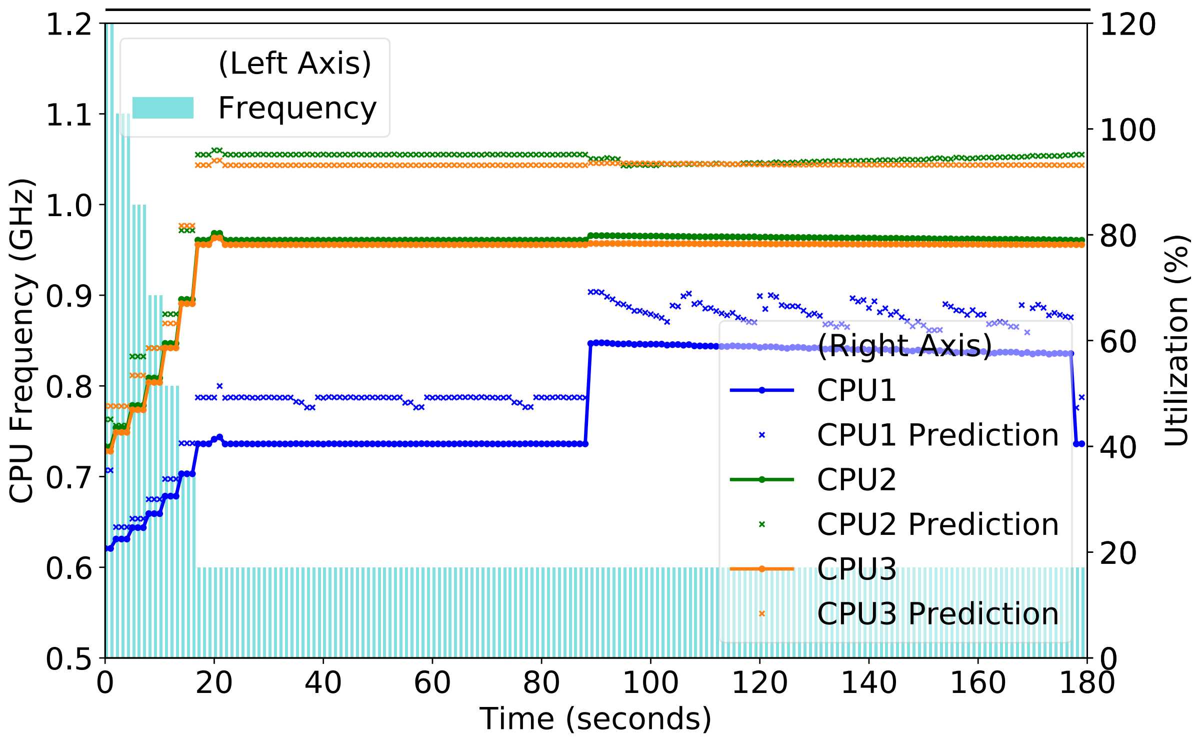


Figure 26 – Energy Optimizer actuation on Task-set 1 with variable contention Bandwidth.

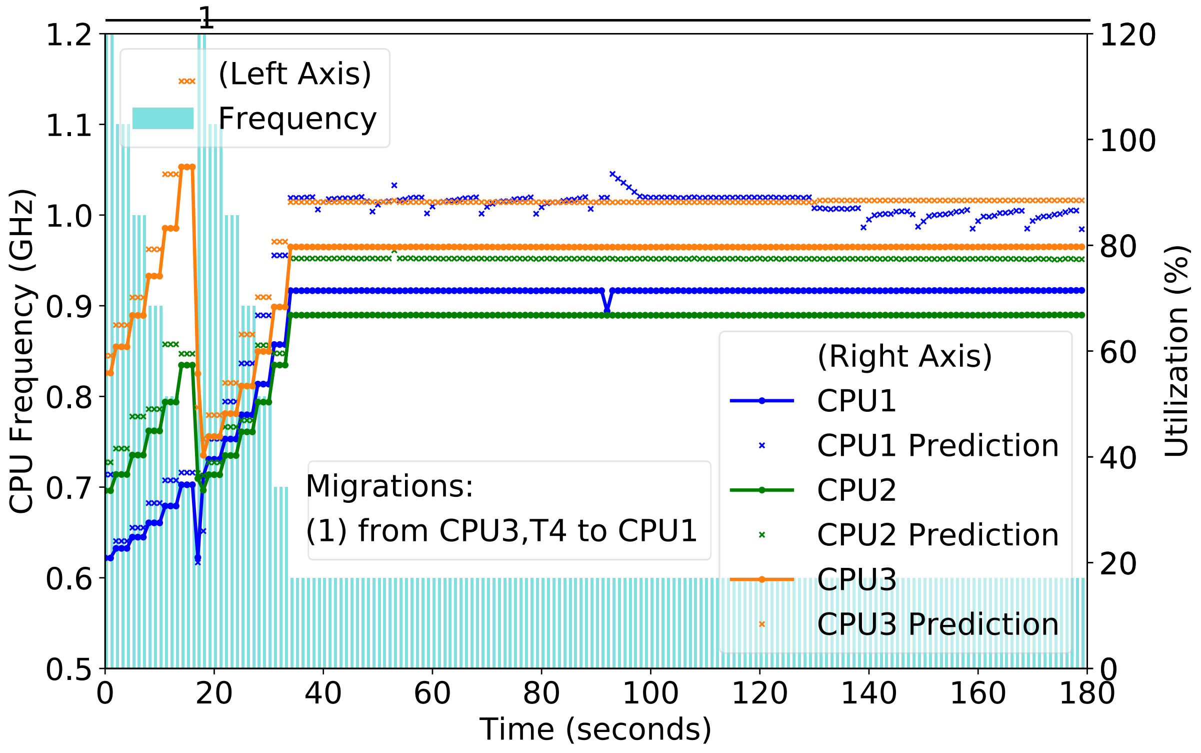


Figure 27 – Energy Optimizer actuation on Task-set 2 with Low Contention Bandwidth.

and 0.99% (CPU1 and CPU2 presented a deviation of 6.67% and 7.23% during the last actuation, which, following the training log, decreased to 6.13% and 3.86% through the online learning algorithm).

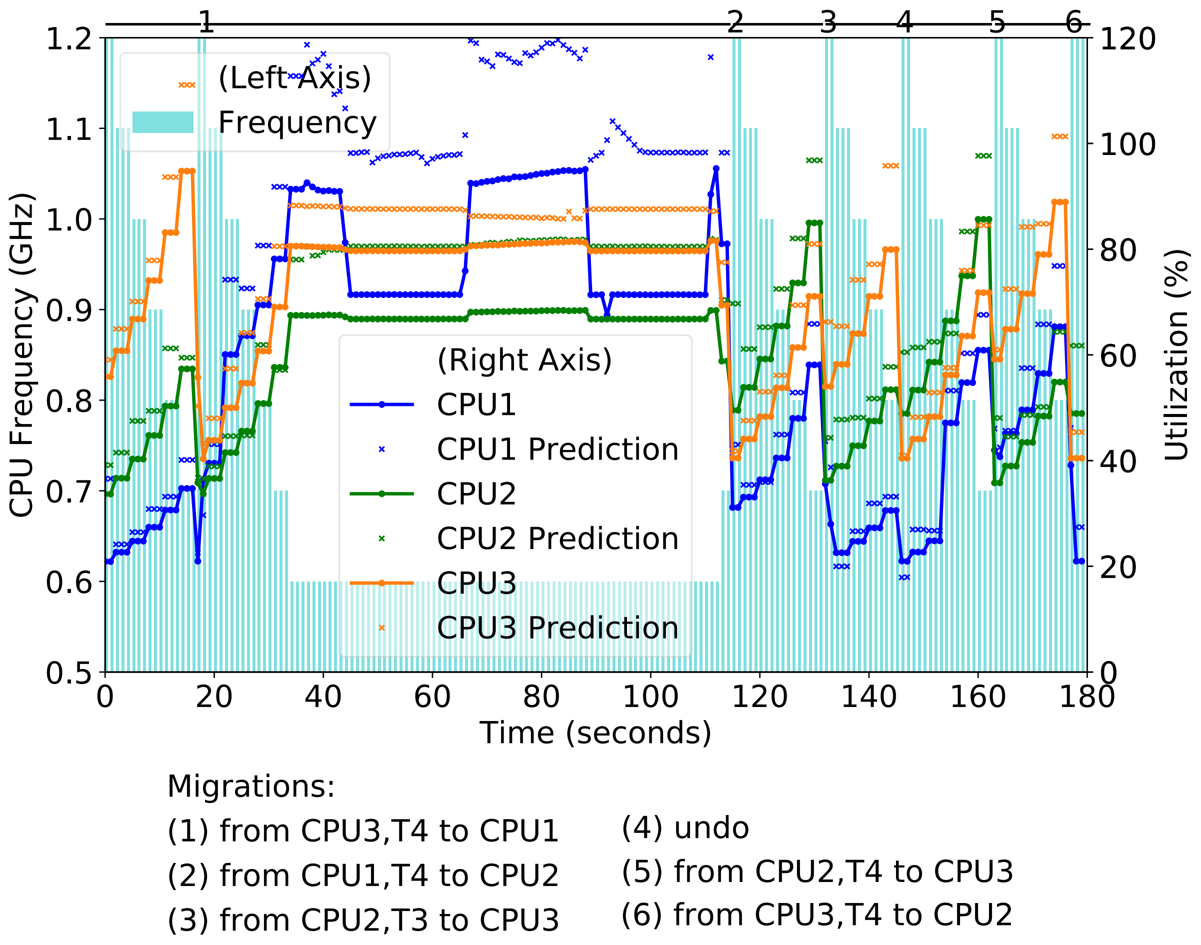


Figure 28 – Energy Optimizer actuation on Task-set 2 with variable contention Bandwidth.

Still analyzing task-set 2, but now for the task-set with Bandwidth configured with behavior variation, the energy-saving achieved by the energy optimizer was, on average, 17.39%. A depiction of the task-set execution is presented in Figure 28. In this scenario, even with Bandwidth's task utilization periodically changing, the task was still capable of reaching the lowest frequency (0.6GHz) while still under the 5% margin. Near 110 seconds of execution, due to the very own architectural variability, CPU 1 presented a slight increase in its utilization, breaking the 5% margin in 0.26%, not causing a deadline miss. In this way, by monitoring every CPU idle time, the energy optimizer identified the margin break and increased the CPUs frequency in one level to avoid disrupting time correctness. In this way, in the next prediction round, the voting concluded that CPU0 would again break the safety margin if the frequency is decreased, triggering the Task Migration heuristic to look for further optimizations, which executed four more migrations and one revocation. The behavior variation, when not fitted under the Safety Margin at the lowest frequency, will always trigger the behavior depicted in Figure 28, which can only converge if the migration threshold is not met by any migration or swap (see Section 4.1.3.1 and Algorithm 5 for further details regarding migration threshold). The maximum prediction deviation presented by the ANN was 16.95%, 15.85%, and 8.02%, for CPU1, CPU2, and CPU3, respectively. Note that such high deviations occurred mainly after migration and during behavior changes, where after the aforementioned maximum deviation, the ANN learned the new behavior, reducing the deviation up to 0.45%, 2.93%, and 2.78% for CPU1, CPU2, and CPU3 during the next actuation in the same behavior.

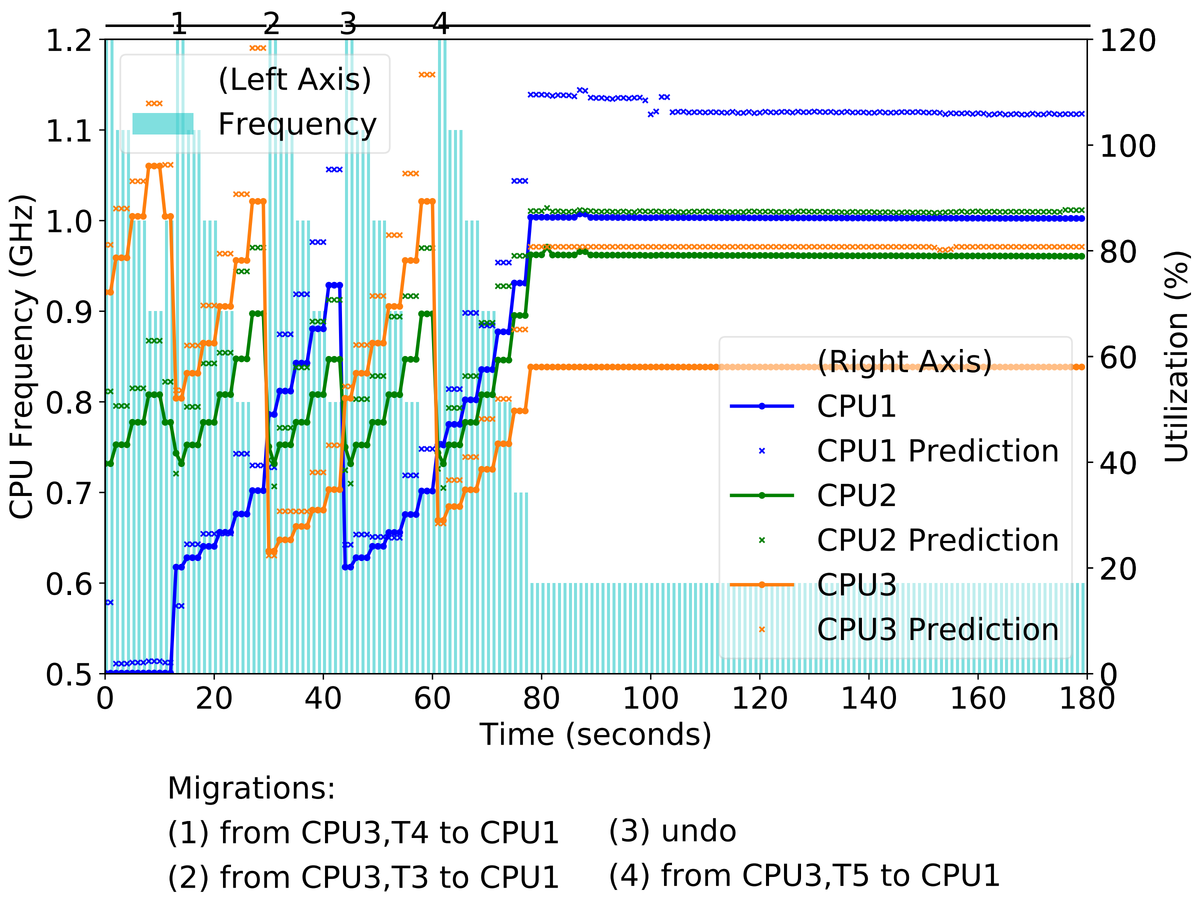


Figure 29 – Energy Optimizer actuation on Task-set 3 with Low Contention Bandwidth.

For the third task-set in the low contention configuration, the energy optimizer presented an average energy saving of 5.94%. A depiction of the task-set execution is presented in Figure 29. In this scenario, where the three Bandwidth tasks are configured with a small utilization, the energy optimizer successfully reached a stable configuration achieving the lowest frequency possible for the architecture (0.6 GHz) after three migrations and one revocation. The safety margin *SM* of 5% was preserved, as the CPU with the highest load presented a slack of 13.88%. The maximum prediction deviation presented by the ANN was 14.42%, 10.08%, and 7.09%, for CPU1, CPU2, and CPU3, respectively. Note that such high deviations occurred mainly after migration, which has been learned by the ANN during the next actuation. The prediction deviations at the last actuation were 6.87%, 0.19%, and 3.33% for CPU1, CPU2, and CPU3, respectively (the deviation for the last actuation was 7.09%, but the training log reported that the error was reduced to 3.33% after the online training).

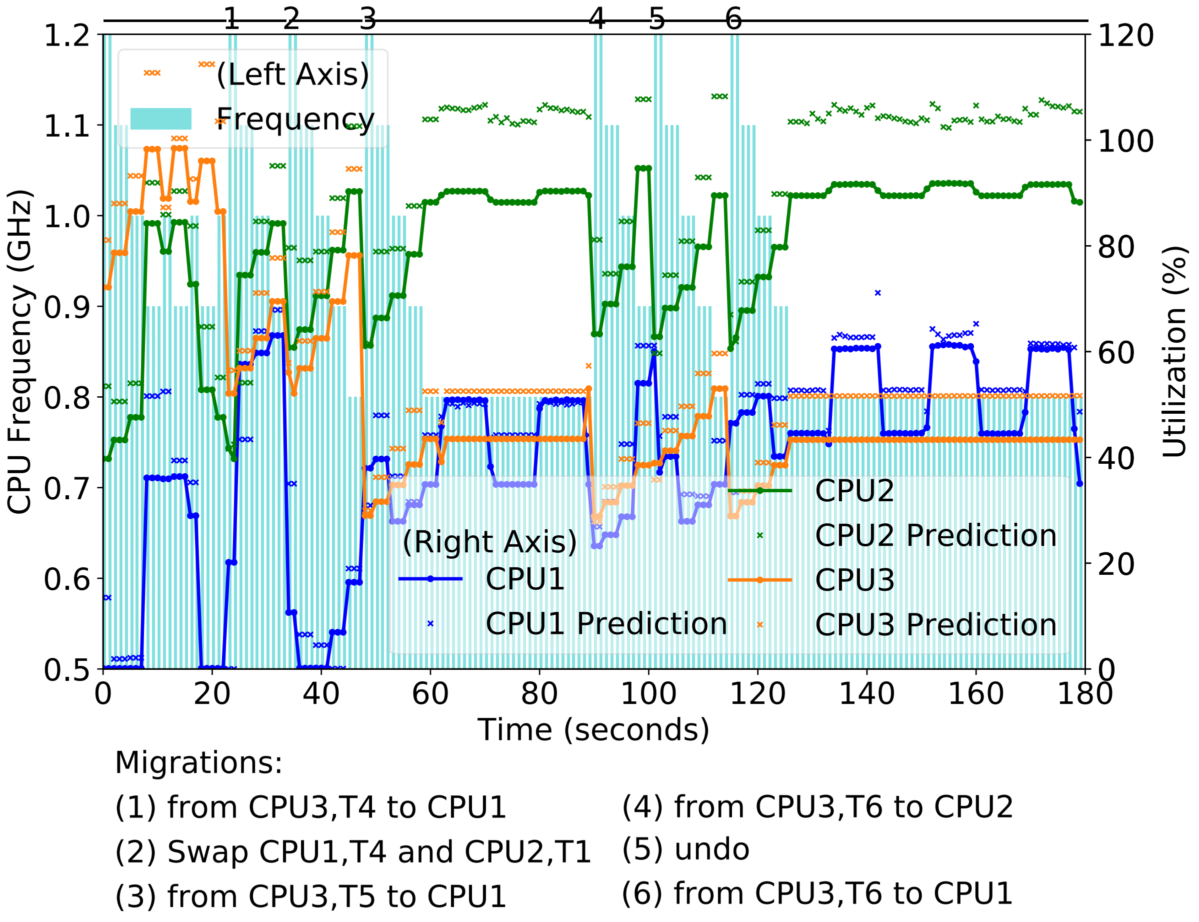


Figure 30 – Energy Optimizer actuation on Task-set 3 with variable contention Bandwidth.

For the task-set with Bandwidth configured with behavior variation in task-set 3, the energy optimizer presented an average energy saving of 24.24%. A depiction of the task-set execution is presented in Figure 30. In this scenario, where the three Bandwidth tasks are configured with behavior variation between the original and small utilization, the energy optimizer successfully reached a stable configuration. The final configuration achieved the same level of optimization presented for the original task-set (0.8GHz) after five migrations, where one of them was a swap between CPU1 and CPU2, and one a revocation. The safety margin *SM* of 5% was preserved in the final configuration, only being breached during the first moment of behavioral change, reaching a maximum of 98.44% of utilization (predicted as 93.23%). In the final configuration, the CPU with the highest load, CPU 2, presented a slack of 11.77% (9.3% during the execution of the heavy Bandwidth configuration) fitting the *SM* of 5%. The maximum prediction deviation presented by the ANN was 57.63%, 32.02%, and 11.2%, for CPU1, CPU2, and CPU3, respectively. Note that such high deviations occurred due to the behavior variation, especially for CPU1 when holding two bandwidth tasks, where after the aforementioned maximum deviation, the ANN learned the new behavior, reducing the deviation up to 0.83%, 0.39%, and 0.41% for CPU1, CPU2, and CPU3 during the next actuation in the same behavior.

## 5.3 DISCUSSION

The energy optimizer design was evaluated through a proof of concept implementation, which included the complete scheduler design presented in Figure 9. The implementation focused on the non-intrusive design and the capabilities of the monitoring and actuation API proposed in this work. When allied with the low intrusiveness presented by an RTOS (i.e., EPOS), the API design creates a clean and reliable scheduler framework with powerful monitoring and actuation capabilities. With the RTOS focusing on handling the given task-set, the performance monitoring and actuation design can be controlled without impairing the correctness of any task execution.

To implement the energy optimizer, first, a utilization predictor is required, and to fulfill the proposed design, a predictor with online learning capabilities is desired. So, an incremental trained ANN design is proposed along with feature exploration for architectural phenomena profiling through performance monitoring. Furthermore, the ANN design is focused on two learning moments: An offline training using a synthetic task-set for architecture tuning and cold start issues avoidance. And the online training, to make the model free from the synthetic task-set used for offline training, to account for performance variability of the task-set, incoming from both the tasks and the architectural phenomena. To further improve the ANN results, a tuning process including a validation data-set incoming from a secondary task-set is recommended, thus, simulating the ANN online training to evaluate the architecture regarding the number of training cycles required for adaptation. Moreover, to accommodate such a complex solution as an ANN into a non-intrusive design, some restrictions were imposed: (i) The ANN must be limited to linear activation functions to reduce its computational cost. (ii) The ANN offline tuning process must focus on finding a simple architecture with sufficient performance, thus reducing the ANN computational complexity. (iii) A contention-free design must be accounted for, where every CPU must have its own ANN predictor structure.

The feature exploration design proposes a performance counter analysis focused on the architectural phenomena related to task performance, especially performance issues incurred from shared resource contention generated by the complex multicore architecture. Such a process usually requires an extensive manual analysis allied with expert knowledge to define the most relevant performance counters. Aiming at making it completely independent of expertise knowledge and manual analysis, similar to Wu and Taylor's (WU; TAYLOR, 2016) and Donyanavard et al.'s (DONYANAVARD et al., 2016) works, a data mining approach for feature selection is designed. Nevertheless, different from their works, which focus on a single method, namely PCA and PCC, respectively, the proposed method includes a preprocessing workflow along with the combination of three different feature selection approaches. This practice of combining multiple techniques, according to Molina et al. (MOLINA et al., 2002), provides a more reliable and extensive feature-set, making the resulting feature-set free from the limitations of using a single technique. This process is combined with redundancy removal to improve the coverage of the resulting feature-set. The resultant set of features is composed of Bus Access for Memory Write operations, Stalls due to Write Buffer Full, L2D Writeback, CPU Cycle Count, L1 Cache Hits, and Immediate Branches. The coverage of the resulting feature-set includes performance counters tracing the memory and CPU usage of tasks, where the memory comprises the primary shared resource in the target architecture. The performance information extracted through such counters was sufficient to build the envisioned Machine Learning model to control the DVFS actuation and guide the task migration, which corroborated the effectiveness of the approach used for feature extraction, the synthetic task-set along with the non-intrusive Monitoring API and the feature selection.

The energy optimizer evaluation over the three different task-sets, including three different configurations for each task-set, provided an overview of the proposed solution performance on a real platform. The idea of including variations of the same task-set was intended to promote an overview of the ANN architecture and Task Migration design under different levels of performance demands and contention. The variation in tasks configuration enabled a further evaluation of the model adaptability, which successfully reached the threshold of frequency configuration without breaking the safety margin established, and thus, not impairing the timing correctness.

The energy optimizer showed an average energy consumption reduction of 24.97%, up to 30.30%, for the original configuration, 17.30%, up to 28.57%, for the low contention scenario, and 21.88%, up to 24.24%, for the scenario with behavior variation. The comparison of energy consumption reductions was done by comparing the energy optimizer performance on an RTOS focusing on a run-to-halt approach, which aims at increasing the amount of halt time of the CPUs, and it does this by always running the tasks at maximum frequency.

As previously discussed, Dynamic Power Management (DPM) is also an actuation available on such embedded platforms. However, the DVFS control provided by the energy optimizer overcame the energy-savings of the run-to-halt approach in every scenario. The only scenario where the energy consumption of the run-to-halt approach was closer to the energy optimizer results was for task-set 3 in the low contention scenario, where the modified task was present in every CPU, reducing the overall performance demands of the task-set. In this scenario, the DPM strategy achieved good results, as one of the CPUs (CPU1) remained halted for approximately 99% of the execution, as the only task in its scheduling queue was one of the tasks affected by the variation, the Bandwidth task, presenting an utilization close to 1%. However, the energy optimizer optimizations still presented a reduction of 5.94% in the energy consumption. Additionally, as this work addresses real-time scenarios, the maximum energy-saving possible for a task-set is inversely proportional to the available idle time and the configured safety margin *SM* assumed for the task-set.

When comparing the energy consumption of the proposed approach, running on the RTOS, with Linux and its power governors, a reduction of 68.91%, 64.70%, and 65.13% were seen for Linux run-to-halt, OnDemand, and Conservative, respectively. Linux's OnDemand and Conservative power governors improved the energy consumption of the Linux Platform up to 16.22%, 3.33%, and 13.16%, for the three task-sets, respectively. When comparing the results in a proportional relation, the energy-savings of Linux Power Governors for the Linux baseline (i.e., run-to-halt), and the optimizations of the proposed approach for the RTOS baseline, the proposed approach still achieved better results in every scenario: 30.30% vs. 16.22% (This work vs. OnDemand and Conservative) in TS1, 14.81% vs. 3.33% (This Work vs. Conservative) in TS2, and 29.79% vs. 13.16% (This Work vs. OnDemand) in TS3.

As claimed by Chen et al. (CHEN et al., 2018), in some applications, not necessarily the lowest frequency configuration provides the lowest energy consumption. However, to not limit the energy optimizer to platforms supporting online power consumption measurement, and as a power model is out of the scope of this work, the actuation design here is focused on reaching the lowest level of frequency possible for the current configuration (comprised of CPU load and *SM*). In this sense, the energy optimizer will not reach the optimal energy-saving state for every scenario, resulting in sub-optimal energy-savings but still without impairing critical tasks. In the evaluated scenarios, task-sets 1, 2, and 3, under three different configurations, the proposed energy optimizer always reached the lowest frequency configuration possible combining the DVFS and the Task Migration heuristic, and it did so without impairing any critical task. As migrations impacts are harder to predict without previously established knowledge, especially due to shared resource contention, one of the limitations of the Task Migration heuristic is to initially profile the activity vector weights to avoid cold start issues, similar to the ANN requirement for offline training. The profiling process is depicted in Section 4.1.3.1.

The performance demand awareness for each different scenario, while maintaining a low ANN architecture complexity, was only achievable by accounting for online learning, here implemented as incremental learning with backpropagation. Increasing the offline data-set extensiveness will make the predictor proposed here closer to the models proposed by Chen et al. (CHEN et al., 2018), Donyanavard (DONYANAVARD et al., 2016), and Mück et al (MÜCK et al., 2015), creating a model with a more complex architecture that can provide better accuracy in its initial state. However, measuring the model coverage, including frequency changes, which can affect the tasks' parallel interactions, and aging effect has a complexity that scales with the platform complexity itself, as modern embedded multicore processors combine a large variety of architectural features, including heterogeneous cores, SIMD units, and application-specific accelerators interconnected by NoC technology, while making intense use of parallelism and latency hiding mechanisms.

Therefore, the first thought that made online learning an objective for this work was to enable the model to account for variability without requiring extensive profiling in an offline analysis, which encompasses both feature extraction and ANN training. Moreover, online learning makes the ANN a process independent of task-set, accounting for several scenarios with low prediction deviation. First of all, as previously explained, the feature extraction must include data from different scenarios to be capable of resulting in the most relevant feature-set regarding task performance and architectural phenomena coverage, a goal that is much easier to achieve through a synthetic task-set in a controllable scenario. Furthermore, the complexity of building a dataset for training an ANN with sufficient coverage for it to be able to extrapolate every possible outcome with high accuracy is infeasible for complex scenarios. The online learning approach reduces the complexity required for ANN offline training, enabling a much more budget-friendly analysis at the cost of a little adaptation during run-time.

The performance of the ANN and the effectiveness of the online learning can be observed, for instance, through the last task-sets' configuration, where the behavior variation of tasks was extrapolated to evaluate the adaptability of the ANN model in an extreme scenario. For Task-set 3 (depicted in Figure 30) during the first occurrence of behavior variation, the ANN prediction deviation increased, reaching 57.63% for CPU1, 32.02% for CPU2, and 11.2% for CPU3. For CPU1, its highest deviation occurred near 30 seconds of execution, reaching a deviation of 57.63% of utilization, which decreased to 16.35% and 0.83% in just two training sessions. The deviation increased again after the second migration, up to 34.87%, which was quickly learned in the next two training sessions, reducing it to 6.33%, and subsequently to 2.43%. Again, near 30 seconds of execution, CPU 2 reached 32.02% of deviation, which decreased in the following frequency scaling to 24.65%, and subsequently to 0.39%. And after the second migration, at 35 seconds, the deviation increased to 15.47%, decreasing to 6.69% and 0.28% in the next two learning sessions. In the remaining execution time, whenever the behavior of the task changed, the deviation increased again, but never that much higher, with the new behavior learned in two or fewer training sessions. For CPU3, the deviations were never higher than 11.2%. This occurrence only happened once and decreased to 0.41% after two learning sessions, as the task that presented behavior variation was migrated to CPU1 near 20 seconds of execution. Therefore, we can conclude that online learning is capable of quickly adapting to new scenarios without compromising tasks' timing correctness.

The energy optimizer results also corroborate the claim for low intrusiveness, adding less than 100μ*s* of overhead during training rounds in the worst-case scenario (limiting to 8 training rounds), and less than 16μ*s* on average per activation, with a maximum overhead on the tasks' execution time of 0.1791% on memory-bound tasks, the ones most affected. For the sake of comparison, the worst-case scenario evaluated for Run-DMC (MÜCK et al., 2015), a task-set with 16 threads, incurred an overhead of up to 869μ*s* per actuation (sensing, estimation and prediction, optimization, and thread mapping), and 158μ*s* in the simpler test case provided in their evaluation, with 4 threads. The ANN prediction and training overhead, when analyzed alone, presented an average run-time of 1.8073μ*s* for predictions and 5.1318μ*s* for one training iteration at the lowest execution frequency, also corroborating the claim for the ANN low-intrusive design. Moreover, the Monitor API was also evaluated alone, presenting a maximum overhead of 0.0718% for the sampling configuration composed of 6 Clerks at 100Hz, with a maximum jitter at job releases never higher than 40μ*s*, thus providing a non-intrusive monitoring design.

# 6 CONCLUSION

Real-time Multicore Embedded Systems, due to multicore architectural complexity, incur the necessity of careful handling of task execution in order to maintain execution determinism and to afford the expanding functionalities which demand more and more performance in such critical environments. With autonomous vehicles, for example, which must handle several computer vision applications in real time, critical application design in multicore embedded systems goes beyond modeling system performance manually and assuring Worst Case-Execution Time estimations. The complexity of such systems, combining a large variety of architectural features of multicores and the extensive usage of parallelism and latency hiding mechanisms, requires handling performance, temperature, and energy consumption in a more automated way. The platforms, however, are themselves highly instrumented cyber-physical systems that can be monitored and controlled based on the data they produce during operations. Therefore, Hardware Performance Counter provides a powerful tool for monitoring the most relevant architectural phenomena that have an impact on task execution. Machine Learning stands up as an alternative to automate the process of acquiring the necessary knowledge about the architectural phenomena expressed by performance traces and OS statistics.

In this work, a non-intrusive, ANN-based run-time energy optimizer for multicore embedded architectures is proposed. The energy optimizer is able to cope with the stringent time requirements of critical tasks. The energy optimizer capabilities include DVFS and task migrations, and it actuates based on the ANN model outputs. The ANN is a component of the energy optimizer that aims at providing predictions regarding the impact a frequency scaling will have on a task performance by using its performance trace as input. The energy optimizer accounts for every task currently running at each core to conceive an actuation, considering the available scheduling slack and a user-defined safety margin. The ANN model is built upon run-time traces collected from hardware performance counters and OS variables, selected using offline feature extraction algorithms. The feature extraction process aims at exposing the most relevant variables related to performance using synthetic architecture-specific task-sets. The traces are also used to build offline training to tune the ANN configuration. The predictor is then trained at run-time whenever the frequency is scale, setting it free from the initial synthetic task-set. The task migration is based on a weighted activity vector concept, and uses the same performance counters used by the ANN. The objective of the task migration is to reduce the activity variance between CPUs in a load-balancing fashion, including task migrations and task swaps between CPUs to achieve a better load distribution and solve contentions. The weights used by the algorithm are profiled beforehand via Gradient Descent.

A proof of concept implementation of the proposed energy optimizer design in a real platform, a Cortex-A53, a widespread quad-core embedded processor, is presented to evaluate the proposed design regarding the predictor capabilities, including its regression accuracy and adaptability and the achieved energy-savings. The evaluation encompassed three different task-sets over three different configurations, totaling 9 evaluation scenarios. The energy optimizer successfully achieved energy consumption reductions in each one of them, always reaching the lowest frequency configuration possible without impairing any critical task. The proposed solution achieved 24.97% energy consumption reductions on average when compared with the run-to-halt approach in the original task configuration, and it did so without impairing any critical task. In the low contention configuration of the task-sets, it achieved 17.30% energy consumption reductions on average. And in the scenario with behavior variability, it achieved 21.88% energy consumption reductions on average. The energy optimizer triggered task migrations with two of the three evaluated task-sets, further optimizing the frequency configuration. The last scenario, with behavior variability, was an extreme scenario made to evaluate the ANN adaptability to online training. The ANN was able to learn the new behavior at run-time, reducing prediction deviations from close to 50.0% to nearly 1.0% in two or fewer training sessions. The non-intrusiveness of the proposed design was evaluated through the poof of concept implementation, showing a maximum average overhead incurred in terms of the execution time on a task of 0.1791%, particularly to a Memory-bound task. For CPU-bound tasks, no overhead was added. In terms of system time overhead, each prediction added 15.3585$\mu s$ of overhead on average, and each retraining cycle triggered at frequency adjustments was never larger than 100$\mu s$.

## 6.1 FUTURE WORKS

In future works, the energy optimizer will be extended in terms of its task migration capabilities in order to include a Machine Learning approach for task migrations, especially for heterogeneous scenarios, where the Machine Learning model can be extended to predict the performance demands of a task on a completely different configuration, instead of only evaluating frequency scaling, for instance, on a big.LITTLE architecture. Other possible future works can investigate other ML algorithms and the combination of different ML algorithms for both Task Migration and DVFS control.

In this sense, the idea of using more complex techniques, or the need for more complex ANN models depending, which scales according to the very own system complexity, also put as a relevant future work the exploration of hybrid Software/Hardware implementations of the Energy Optimizer through FPGAs, like running the ML algorithms as a hardware component to improve the Energy Optimizer non-intrusiveness.

Moreover, the presented approach and analysis methodology can be extended to several different scenarios where the limiting agent is the performance impact over shared resources, and not only multicore embedded environments.

## REFERENCES

AKRAM, Naveed; ZHANG, Yangyang; ALI, Shahbaz; AMJAD, Hafiz Muhammad. Efficient Task Allocation for Real-Time Partitioned Scheduling on Multi-Core Systems. In: 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST). [S.l.]: IEEE, Jan. 2019. DOI: `10.1109/ibcast.2019.8667139`.

ALENAWY, T. A.; AYDIN, H. Energy-aware task allocation for rate monotonic scheduling. In: 11TH IEEE Real Time and Embedded Technology and Applications Symposium. [S.l.: s.n.], 2005. P. 213–223.

ARM. **ARM Cortex-A53 MPCore Processor**. [S.l.]: ARM, 2016.

BASTONI, A.; BRANDENBURG, B. B.; ANDERSON, J. H. Is Semi-Partitioned Scheduling Practical? In: 2011 23rd Euromicro Conference on Real-Time Systems. [S.l.: s.n.], 2011. P. 125–135.

BENESTY, Jacob; CHEN, Jingdong; HUANG, Yiteng; COHEN, Israel. Pearson Correlation Coefficient. In: NOISE Reduction in Speech Processing. [S.l.]: Springer Berlin Heidelberg, 2009. P. 1–4. DOI: `10.1007/978-3-642-00296-0_5`.

BINKERT, Nathan et al. The Gem5 Simulator. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 39, n. 2, p. 1–7, Aug. 2011. ISSN 0163-5964. DOI: `10.1145/2024716.2024718`. Available from: `https://doi.org/10.1145/2024716.2024718`.

BISWAS, Dwaipayan; BALAGOPAL, Vibishna; SHAFIK, Rishad; AL-HASHIMI, Bashir M.; MERRETT, Geoff V. Machine learning for run-time energy optimisation in many-core systems. In: DESIGN, Automation & Test in Europe Conference & Exhibition (DATE), 2017. [S.l.]: IEEE, Mar. 2017. DOI: `10.23919/date.2017.7927243`.

BOLÓN-CANEDO, Verónica; SÁNCHEZ-MAROÑO, Noelia; ALONSO-BETANZOS, Amparo. A review of feature selection methods on synthetic data. **Knowledge and Information Systems**, Springer Science and Business Media LLC, v. 34, n. 3, p. 483–519, Mar. 2012. DOI: `10.1007/s10115-012-0487-8`. Available from: `https://doi.org/10.1007/s10115-012-0487-8`.

BRANDENBURG, Björn B. **Scheduling and Locking in Multiprocessor Real-Time Operating Systems**. 2011. PhD thesis – The University of North Carolina at Chapel Hill. Ph. D. Thesis.

CALANDRINO, J.; ANDERSON, J.; BAUMBERGER, D. A hybrid real-time scheduling approach for large-scale multicore platforms. **19th Euromicro Conference on Real-Time Systems (ECRTS'07)**, v. 2, n. 19, p. 247–258, 2007. ISSN 1068-3070. Available from: `https://ieeexplore.ieee.org/document/4271698`.

CARPENTER, J.; FUNK, S.; HOLMAN, P.; SRINIVASAN, A.; ANDERSON, J.; BARUAH, S. **A categorization of real-time multiprocessor scheduling problems and algorithms**. [S.l.]: Chapman Hall/CRC, 2004.

CHEN, Yen-Lin; CHANG, Ming-Feng; YU, Chao-Wei; CHEN, Xiu-Zhi; LIANG, Wen-Yew. Learning-Directed Dynamic Voltage and Frequency Scaling Scheme with Adjustable Performance for Single-Core and Multi-Core Embedded and Mobile Systems. **Sensors**, MDPI AG, v. 18, n. 9, p. 3068, Sept. 2018. DOI: `10.3390/s18093068`. Available from: `https://doi.org/10.3390/s18093068`.

CRAEYNEST, Kenzo Van; JALEEL, Aamer; EECKHOUT, Lieven; NARVAEZ, Paolo; EMER, Joel. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In: 2012 39th Annual International Symposium on Computer Architecture (ISCA). [S.l.]: IEEE, June 2012. DOI: `10.1109/isca.2012.6237019`.

DAS, Anup; WALKER, Matthew J.; HANSSON, Andreas; AL-HASHIMI, Bashir M.; MERRETT, Geoff V. Hardware-software interaction for run-time power optimization: A case study of embedded Linux on multicore smartphones. In: 2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). [S.l.]: IEEE, July 2015. DOI: `10.1109/islped.2015.7273508`.

DAVIS, Robert I.; BURNS, Alan. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 43, n. 4, Oct. 2011. ISSN 0360-0300. DOI: `10.1145/1978802.1978814`. Available from: `https://doi.org/10.1145/1978802.1978814`.

DEHMELT, F. **Adaptive (Dynamic) Voltage (Frequency) Scaling—Motivation and Implementation**. [S.l.: s.n.], Mar. 2014. Available from: `http://www.ti.com/lit/an/slva646/slva646.pdf`.

DOMINIK BRODOWSKI. **Linux CPUFreq - CPUFreq Governors**. 2017. Available from: `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`. Visited on: 10 Sept. 2020.

DONYANAVARD, Bryan; MÜCK, Tiago; SARMA, Santanu; DUTT, Nikil. SPARTA: Runtime task allocation for energy efficient heterogeneous manycores. **2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)**, IEEE, p. 1–10, 2016.

EYERMAN, Stijn; EECKHOUT, Lieven. A Counter Architecture for Online DVFS Profitability Estimation. **IEEE Transactions on Computers**, Institute of Electrical and Electronics Engineers (IEEE), v. 59, n. 11, p. 1576–1583, Nov. 2010. DOI: `10.1109/tc.2010.65`. Available from: `https://doi.org/10.1109/tc.2010.65`.

FAHLMAN, Scott E. Faster-Learning Variations on Back-Propagation: An Empirical Study. In: TOURETZKY, David S.; HINTON, Geoffrey E.; SEJNOWSKI, Terrence J. (Eds.). **Proceedings of the 1988 Connectionist Models Summer School**. [S.l.]: San Francisco, CA: Morgan Kaufmann, 1989. P. 38–51.

FRÖHLICH, Antônio Augusto. A Comprehensive Approach to Power Management in Embedded Systems. **International Journal of Distributed Sensor Networks**, v. 2011, n. 1, p. 19, 2011. ISSN 1550-1477. Available from: `http://www.hindawi.com/journals/ijdsn/2011/807091/`.

FRÖHLICH, Antônio Augusto. **Application-Oriented Operating Systems**. 2001. S. 200. PhD thesis – Berlin: Technical University, Berlin. Ph. D. Thesis.

GENNARI, John H.; LANGLEY, Pat; FISHER, Doug. Models of incremental concept formation. **Artificial Intelligence**, Elsevier BV, v. 40, n. 1-3, p. 11–61, Sept. 1989. DOI: `10.1016/0004-3702(89)90046-5`. Available from: `https://doi.org/10.1016/0004-3702(89)90046-5`.

GRACIOLI, Giovani. **Real-Time Operating System Support for Multicore Applications**. 2014. S. 359. PhD thesis – Federal University of Santa Catarina, Florianópolis. Ph. D. Thesis.

GRACIOLI, Giovani; FROHLICH, Antonio Augusto. CAP: Color-aware task partitioning for multicore real-time applications. In: PROCEEDINGS of the 2014 IEEE Emerging

Technology and Factory Automation (ETFA). [S.l.]: IEEE, Sept. 2014. DOI: 10.1109/etfa.2014.7005118.

GRACIOLI, Giovani; FRÖHLICH, Antônio. Towards a Shared-data-aware Multicore Real-time Scheduler. In: REAL-TIME Scheduling Open Problems Seminar (RTSOPS). Paris, France: [s.n.], July 2013. Available from: http://www.lisha.ufsc.br/pub/Gracioli%5C_RTSOPS%5C_2013.pdf.

GRACIOLI, Giovani; FRÖHLICH, Antônio Augusto. On the Design and Evaluation of a Real-Time Operating System for Cache-Coherent Multicore Architectures. **ACM SIGOPS Operating Systems Review**, v. 49, n. 2, p. 2–16, 2015. ISSN 0163-5980. Available from: http://dl.acm.org/citation.cfm?id=2883594.

GRACIOLI, Giovani; TABISH, Rohan; MANCUSO, Renato; MIROSANLOU, Reza; PELLIZZONI, Rodolfo; CACCAMO, Marco. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. In: QUINTON, Sophie (Ed.). **31st Euromicro Conference on Real-Time Systems (ECRTS 2019)**. Dagstuhl, Germany: [s.n.], 2019. (Leibniz International Proceedings in Informatics (LIPIcs)), 27:1–27:25. DOI: 10.4230/LIPIcs.ECRTS.2019.27.

HALL, Mark Andrew. **Correlation-based Feature Selection for Machine Learning**. 1999. PhD thesis – The University of Waikato, New Zealand. Ph. D. Thesis.

HEECHUL YUN. **Misc micro-benchmarks & tools**. 2019. Available from: https://github.com/heechul/misc/.

HOFFMANN, J. L. C.; HORSTMANN, L. P.; FRÖHLICH, A. A. Anomaly Detection in Multicore Embedded Systems. In: 2019 IX Brazilian Symposium on Computing Systems Engineering (SBESC). [S.l.: s.n.], 2019. P. 1–8.

HORSTMANN, L. P.; HOFFMANN, J. L. C.; FRÖHLICH, A. A. A Framework to Design and Implement Real-time Multicore Schedulers using Machine Learning. In: 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). [S.l.: s.n.], 2019. P. 251–258.

HSU, Chung-Hsing. **Compiler-directed dynamic voltage and frequency scaling for cpu power and energy reduction**. 2003. S. 120. PhD thesis – Rutgers University, New Brunswick, NJ, USA. Ph.D Thesis.

IGEL, Christian; HÜSKEN, Michael. Improving the Rprop learning algorithm. English. In: PROCEEDINGS of the Second International Symposium on Neural Computation, NC'2000. [S.l.]: ICSC Academic Press, 2000. P. 115–121.

INTEL CO. **Intel 64 and IA-32 Architectures Software Developer's Manual**. [S.l.]: Intel Corporation, 2019.

ISLAM, F. M. M. ul; LIN, M.; YANG, L. T.; CHOO, K.-K. R. Task aware hybrid DVFS for multi-core real-time systems using machine learning. **Information Sciences**, v. 433-434, p. 315–332, 2018. ISSN 0020-0255. Available from: `https://www.sciencedirect.com/science/article/pii/S0020025517308897`.

ISMAIL, H.; JAWAWI, D. N. A.; ADHAM ISA, M. A weakly hard real-time tasks on global scheduling of multiprocessor systems. In: 2015 9th Malaysian Software Engineering Conference (MySEC). [S.l.: s.n.], 2015. P. 123–128. DOI: `10.1109/MySEC.2015.7475207`.

JOHNSON, David. **Near-optimal bin packing algorithms**. 1973. S. 401. PhD thesis – Massachusetts Institute of Technology, Massachusetts. Ph. D. Thesis.

JUNG, H.; PEDRAM, M. Supervised Learning Based Power Management for Multicore Processors. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 29, n. 9, p. 1395–1408, Sept. 2010. DOI: `10.1109/tcad.2010.2059270`.

JUNIOR, Arliones Stevert Hoeller. **Gerenciamento do Consumo de Energia Dirigido pela Aplicação em Sistemas Embarcados**. 2007. S. 87. MA thesis – Federal University of Santa Catarina, Florianópolis. M.Sc. Thesis.

KEDAR, G.; MENDELSON, A.; CIDON, I. SPACE: Semi-Partitioned CachE for Energy Efficient, Hard Real-Time Systems. **IEEE Transactions on Computers**, v. 66, n. 4, p. 717–730, 2017.

KIM, Myungsun; KIM, Kibeom; GERACI, James R.; HONG, Seongsoo. Utilization-aware load balancing for the energy efficient operation of the big.LITTLE processor. **2014 Design, Automation and Test in Europe Conference and Exhibition (DATE)**, p. 46–55, 2014. ISSN 1530-1591. Available from: `https://ieeexplore.ieee.org/document/6800437`.

KRAWCZAK, Maciej. **Multilayer Neural Networks**. [S.l.]: Springer International Publishing, 2013. DOI: `10.1007/978-3-319-00248-4`. Available from: `https://doi.org/10.1007/978-3-319-00248-4`.

LAHIRI, Anirban; BUSSA, Nagaraju; SARASWAT, Pawan. A Neural Network Approach to Dynamic Frequency Scaling. In: 15TH International Conference on Advanced Computing and Communications (ADCOM 2007). [S.l.]: IEEE, Dec. 2007. DOI: `10.1109/adcom.2007.123`.

LAKSHMANAN, Karthik; NIZ, Dionisio de; RAJKUMAR, Ragunathan. Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors. In: 2009 30th IEEE Real-Time Systems Symposium. [S.l.]: IEEE, Dec. 2009. DOI: `10.1109/rtss.2009.51`. Available from: `https://doi.org/10.1109/rtss.2009.51`.

LI, H.; HOMER, N. A survey of sequence alignment algorithms for next-generation sequencing. **2010 Briefings in Bioinformatics, Volume 11**, p. 473–483, 2010. ISSN 0020-0255. Available from: `https://doi.org/10.1093/bib/bbq015`.

LIU, C. L.; LAYLAND, James W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. **Journal of the ACM**, Association for Computing Machinery (ACM), v. 20, n. 1, p. 46–61, Jan. 1973. DOI: `10.1145/321738.321743`. Available from: `https://doi.org/10.1145/321738.321743`.

LIU, Jane W. S. **Real-Time Systems**. Upper Saddle River, NJ: Prentice Hall, 2000. Available from: `http://www.amazon.com/Real-Time-Systems-Jane-W-Liu/dp/0130996513`.

MAIZA, Claire; RIHANI, Hamza; RIVAS, Juan M.; GOOSSENS, Joël; ALTMEYER, Sebastian; DAVIS, Robert I. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 3, June 2019. ISSN 0360-0300. DOI: `10.1145/3323212`. Available from: `https://doi.org/10.1145/3323212`.

MANNILA, H. Data mining: machine learning, statistics, and databases. In: PROCEEDINGS of 8th International Conference on Scientific and Statistical Data Base Management. [S.l.: s.n.], 1996. P. 2–9. DOI: `10.1109/SSDM.1996.505910`.

MARCONDES, Hugo. **Uma Arquitetura de Componentes Híbridos de Hardware e Software para Sistemas Embarcados**. 2009. S. 92. MA thesis – Federal University of Santa Catarina, Florianópolis. M.Sc. Thesis.

MARCONDES, Hugo; CANCIAN, Rafael; STEMMER, Marcelo; FRÖHLICH, Antônio Augusto. On the design of flexible real time schedulers for embedded systems. In: INTERNATIONAL Symposium on Embedded and Pervasive Systems. Vancouver, Canada: [s.n.], Aug. 2009. P. 382–387.

MARINAKIS, Theodoros; KUNDAN, Shivam; ANAGNOSTOPOULOS, Iraklis. Meeting Power Constraints while Mitigating Contention on Clustered Multi-Processor System. **IEEE Embedded Systems Letters**, Institute of Electrical and Electronics Engineers (IEEE), p. 1–1, 2019. DOI: 10.1109/les.2019.2956990. Available from: https://doi.org/10.1109/les.2019.2956990.

MCCULLOCH, Warren S; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. **Bulletin of mathematical biology**, Bulletin of Mathematical Biophysics, v. 5, p. 115–133, 1943.

MEIRA, Gustavo Roberto Nardon. **Real-Time Dynamic Voltage and Frequency Scaling no sistema EPOS**. Florianópolis: [s.n.], 2011. P. 77. B.Sc. Thesis.

MERKEL, Andreas; STOESS, Jan; BELLOSA, Frank. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In: PROCEEDINGS of the 5th European Conference on Computer Systems. Paris, France: Association for Computing Machinery, 2010. (EuroSys '10), p. 153–166. DOI: 10.1145/1755913.1755930. Available from: https://doi.org/10.1145/1755913.1755930.

MITCHELL, Thomas M. **Machine Learning**. 1. ed. USA: McGraw-Hill, Inc., 1997. ISBN 0070428077.

MOK, Aloysius Ka-Lau. **Fundamental design problems of distributed systems for the hard-real-time environment**. 1983. PhD thesis – Massachusetts Institute of Technology, Cambridge, MA, USA. Ph. D. Thesis. Available from: http://hdl.handle.net/1721.1/15670.

MOLINA, L. C.; BELANCHE, L.; NEBOT, A. Feature selection algorithms: a survey and experimental evaluation. In: 2002 IEEE International Conference on Data Mining, 2002. Proceedings. [S.l.: s.n.], 2002. P. 306–313.

MÜCK, Tiago; FRÖHLICH, Antonio A.; GRACIOLI, Giovani; RAHMANI, Amir M.; REIS, João Gabriel; DUTT, Nikil. CHIPS-AHOy. In: PROCEEDINGS of the 18th International Conference on Embedded Computer Systems Architectures, Modeling, and Simulation - SAMOS'18. [S.l.]: ACM Press, 2018. DOI: `10.1145/3229631.3229642`. Available from: `https://doi.org/10.1145/3229631.3229642`.

MÜCK, Tiago; SARMA, Santanu; DUTT, Nikil. Run-DMC: Runtime Dynamic Heterogeneous Multicore Performance and Power Estimation for Energy Efficiency. In: PROCEEDINGS of the 10th International Conference on Hardware/Software Codesign and System Synthesis. Amsterdam, The Netherlands: IEEE Press, 2015. (CODES '15), p. 173–182.

NEMATI, Farhang; NOLTE, Thomas; BEHNAM, Moris. Partitioning Real-Time Systems on Multiprocessors with Shared Resources. In: LECTURE Notes in Computer Science. [S.l.]: Springer Berlin Heidelberg, 2010. P. 253–269. DOI: `10.1007/978-3-642-17653-1_20`.

NISSEN, Steffen. **Large Scale Reinforcement Learning using Q-SARSA(λ) and Cascading Neural Networks**. 2007. S. 264. MA thesis – University of Copenhagen, Denmark. M.Sc. Thesis.

PAGANI, Santiago; MANOJ, P. D. Sai; JANTSCH, Axel; HENKEL, Jorg. Machine Learning for Power, Energy, and Thermal Management on Multicore Processors: A Survey. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 39, n. 1, p. 101–116, Jan. 2020. DOI: `10.1109/tcad.2018.2878168`.

QUINLAN, J. R. Induction of decision trees. **Machine Learning**, Springer Science and Business Media LLC, v. 1, n. 1, p. 81–106, Mar. 1986. DOI: `10.1007/bf00116251`.

RABAEY, J.M.; PEDRAM, M. **Low Power Design Methodologies**. [S.l.]: Springer US, 2012. (The Springer International Series in Engineering and Computer Science). ISBN 9781461523079. Available from: `https://books.google.com.br/books?id=9IzuBwAAQBAJ`.

RADHA, R.; MURALIDHARA, S. Removal of redundant and irrelevant data from training datasets using speedy feature selection method. **International Journal of Computer Science and Mobile Computing, 2016**, p. 359–364, 2015. ISSN

2320-088X. Available from:
https://www.ijcsmc.com/docs/papers/July2016/V5I7201670.pdf.

RAI, Jitendra Kumar; NEGI, Atul; WANKAR, Rajeev; NAYAK, K.D. On Prediction Accuracy of Machine Learning Algorithms for Characterizing Shared L2 Cache Behavior of Programs on Multicore Processors. In: 2009 First International Conference on Computational Intelligence, Communication Systems and Networks. [S.l.]: IEEE, July 2009. DOI: 10.1109/cicsyn.2009.45.

RAI, Jitendra Kumar; NEGI, Atul; WANKAR, Rajeev; NAYAK, K.D. Performance Prediction on Multi-core Processors. In: 2010 International Conference on Computational Intelligence and Communication Networks. [S.l.]: IEEE, Nov. 2010. DOI: 10.1109/cicn.2010.125. Available from: https://doi.org/10.1109/cicn.2010.125.

RASPBERRY PI FOUNDATION. **Raspberry Pi 3 Model B**. 2019. Available from: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. Visited on: 22 June 2019.

RIPOLL, I.; BALLESTER-RIPOLL, R. Period Selection for Minimal Hyperperiod in Periodic Task Systems. **IEEE Transactions on Computers**, Institute of Electrical and Electronics Engineers (IEEE), v. 62, n. 9, p. 1813–1822, Sept. 2013. DOI: 10.1109/tc.2012.243.

ROHAN PAUL. **Vectorizing Gradient Descent — Multivariate Linear Regression and Python implementation**. 2020. Available from: https://medium.com/analytics-vidhya/vectorizing-gradient-descent-multivariate-linear-regression-and-python-implementation-e12758bc31b2.

RUPANETTI, D.; SALAMY, H. Energy Efficient Scheduling with Task Migration on MPSoC Architectures. In: 2019 IEEE International Conference on Electro Information Technology (EIT). [S.l.: s.n.], 2019. P. 156–161. DOI: https://doi.org/10.1109/EIT.2019.8833726.

SARMA, S.; DUTT, N.; GUPTA, P.; VENKATASUBRAMANIAN, N.; NICOLAU, A. Cyberphysical-System-on-Chip (CPSoC): A Self-Aware MPSoC Paradigm with Cross-Layer Virtual Sensing and Actuation. In: PROCEEDINGS of the 2015 Design, Automation and Test in Europe Conference and Exhibition. Grenoble, France: EDA Consortium, 2015. (DATE ’15), p. 625–628.

SHALABI, Luai Al; SHAABAN, Zyad; KASASBEH, Basel. Data Mining: A Preprocessing Engine. **Journal of Computer Science**, Science Publications, v. 9, n. 2, p. 735–739, 2006. ISSN 1549-3636.

SHEKHAR, M.; SARKAR, A.; RAMAPRASAD, H.; MUELLER, F. Semi-Partitioned Hard-Real-Time Scheduling under Locked Cache Migration in Multicore Systems. In: 2012 24th Euromicro Conference on Real-Time Systems. [S.l.: s.n.], 2012. P. 331–340. DOI: `10.1109/ECRTS.2012.27`.

SHEN, Hao; TAN, Ying; LU, Jun; WU, Qing; QIU, Qinru. Achieving autonomous power management using reinforcement learning. **ACM Transactions on Design Automation of Electronic Systems**, Association for Computing Machinery (ACM), v. 18, n. 2, p. 1–32, Mar. 2013. DOI: `10.1145/2442087.2442095`.

SINGH, Karan; BHADAURIA, Major; MCKEE, Sally A. Real Time Power Estimation and Thread Scheduling via Performance Counters. **ACM SIGARCH Computer Architecture News**, p. 46–55, 2009. ISSN 0163-5964. Available from: `https://doi.org/10.1145/1577129.1577137`.

STANKOVIC, J.A.; LU, Chenyang; SON, S.H.; TAO, Gang. The case for feedback control real-time scheduling. **11th Euromicro Conference on Real-Time Systems**, York, UK, p. 11–20, Aug. 1999. Available from: `https://doi.org/10.1109/EMRTS.1999.777445`.

STEFFEN NISSEN. **FANN:Fast Artificial Neural Network Library**. 2019. Available from: `http://leenissen.dk/fann/wp/`.

SUTTON, Richard S.; BARTO, Andrew G. **Reinforcement Learning: An Introduction**. Cambridge, MA, USA: A Bradford Book, 2018. ISBN 0262039249.

TANENBAUM, A.S.; BOS, H. **Modern Operating Systems**. [S.l.]: Pearson, 2015. (Always learning). ISBN 9780133591620. Available from: `https://books.google.com.br/books?id=9gqnngEACAAJ`.

TIBSHIRANI, Robert. Regression Shrinkage and Selection via the Lasso. **Journal of the Royal Statistical Society. Series B (Methodological)**, [Royal Statistical Society, Wiley], v. 58, n. 1, p. 267–288, 1996. ISSN 00359246.

UFSC/LISHA. **EPOS: Embedded Parallel Operating System)**. 2019. Available from: https://epos.lisha.ufsc.br/. Visited on: 22 June 2019.

VENKATA, Sravanthi Kota; AHN, Ikkjin; JEON, Donghwan; GUPTA, Anshuman; LOUIE, Christopher; GARCIA, Saturnino; BELONGIE, Serge; TAYLOR, Michael Bedford. SD-VBS: The San Diego vision benchmark suite. **2009 IEEE International Symposium on Workload Characterization (IISWC)**, p. 55–64, Oct. 2009.

WERBOS, Paul John. **Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences**. 1974. PhD thesis – Harvard University.

WU, Xingfu; TAYLOR, Valerie. Utilizing Hardware Performance Counters to Model and Optimize the Energy and Performance of Large Scale Scientific Applications on Power-Aware Supercomputers. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). [S.l.]: IEEE, May 2016. DOI: 10.1109/ipdpsw.2016.78. Available from: https://doi.org/10.1109/ipdpsw.2016.78.

YANG, Maolin; HUANG, Wen-Hung; CHEN, Jian-Jia. Resource-Oriented Partitioning for Multiprocessor Systems with Shared Resources. **IEEE Transactions on Computers**, Institute of Electrical and Electronics Engineers (IEEE), v. 68, n. 6, p. 882–898, June 2019. DOI: 10.1109/tc.2018.2889985. Available from: https://doi.org/10.1109/tc.2018.2889985.

YE, Rong; XU, Qiang. Learning-based power management for multi-core processors via idle period manipulation. In: 17TH Asia and South Pacific Design Automation Conference. [S.l.]: IEEE, Jan. 2012. DOI: 10.1109/aspdac.2012.6164929. Available from: https://doi.org/10.1109/aspdac.2012.6164929.

ZEPPENFELD, Johannes; HERKERSDORF, Andreas. Applying autonomic principles for workload management in multi-core systems on chip. In: PROCEEDINGS of the 8th ACM international conference on Autonomic computing - ICAC. [S.l.]: ACM Press, 2011.