View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositório Institucional da UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
BACHARELADO EM ENGENHARIA MECATRÔNICA

ELOI LUIZ GIACOBBO FILHO

MELHORANDO O SUPORTE A UNIDADES DE MONITORAMENTO DE DESEMPENHO NO EPOS

Joinville

2016

ELOI LUIZ GIACOBBO FILHO

MELHORANDO O SUPORTE A UNIDADES DE MONITORAMENTO DE DESEMPENHO NO EPOS

Trabalho de Conclusão de Curso (Graduação) apresentado como requisito parcial para obtenção do Grau de Bacharel em Engenharia Mecatrônica da Universidade Federal de Santa Catarina.
Orientador: Prof. Dr. Giovani Gracioli

Joinville

2016

Giacobbo, Eloi Luiz Filho

MELHORANDO O SUPORTE A UNIDADES DE MONITORAMENTO DE DESEMPENHO NO EPOS / Eloi Luiz Giacobbo Filho; Orientador: Prof. Dr. Giovani Gracioli – Joinville, 2016.

97 p.

Trabalho de Conclusão de Curso (Graduação) – Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville. Bacharelado em Engenharia Mecatrônica.

Inclui referências

1. Sistemas embarcados. 2. Sistemas operacionais de tempo real. 3. Processadores multinúcleo. 4. Unidades de monitoramento de desempenho. I. Orientador: Prof. Dr. Giovani Gracioli. II. Universidade Federal de Santa Catarina. III. Centro Tecnológico de Joinville. IV. Bacharelado em Engenharia Mecatrônica. V. Título: MELHORANDO O SUPORTE A UNIDADES DE MONITORAMENTO DE DESEMPENHO NO EPOS.

Eloi Luiz Giacobbo Filho

MELHORANDO O SUPORTE A UNIDADES DE MONITORAMENTO DE DESEMPENHO NO EPOS

Este trabalho de conclusão de curso foi julgado adequado para obtenção do Título de “Engenheiro Mecatrônico”, aprovado em sua forma final pelo Curso de Graduação em Engenharia Mecatrônica da Universidade Federal de Santa Catarina.

Joinville, 2 de dezembro de 2016.

Prof. Dr. Diego Santos Greff
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Giovani Gracioli
Orientador

Prof. Dr. Guilherme Piegas Koslovski
Membro 1

Me. Claudio Eduardo Soares
Membro 2

Joinville
2016

AGRADECIMENTOS

A toda minha família que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida.

Ao professor Giovani pela paciência n orientação incentivo q tornaram possível conclusão desta monografia.

Ao LISHA e seus integrantes, pelas trocas acadêmicas e pelo suporte oferecido durante o desenvolvimento deste trabalho.

Aos meus colegas de curso que me acompanharam durante estes últimos anos em longas noites e finais de semana de estudos.

RESUMO

O EPOS é um sistema operacional de tempo real voltado ao desenvolvimento de sistemas embarcados. Sistemas embarcados fazem parte de nosso dia a dia e podem ser definidos como quaisquer sistemas computacionais dedicados ao monitoramento ou controle de aplicações específicas. Muitas vezes, os requisitos de desempenho em um sistema embarcado incluem restrições de tempo real. Independente de sua carga total de execução ou da ocorrência de falhas, todo sistema de tempo real deve cumprir seus requisitos de tempo. Para garantir o cumprimento de tais prazos, previsibilidade é uma característica ainda mais importante do que velocidade de processamento de dados e isto torna essencial o uso de sistemas operacionais de tempo real (RTOS). Em aplicações modernas, RTOSs são utilizados no processamento de elevadas cargas de dados, como no processamento de imagens, aplicações multimídia, entre outros. Neste cenário, processadores multinúcleos representam uma ótima alternativa à redução de custos em determinados projetos. No entanto, o uso de multiprocessadores traz consigo uma série de fatores inconvenientes. Processadores e periféricos são conectados à hierarquia de memória, barramentos, *buffers* e diversos outros recursos complexos. O compartilhamento destes recursos permite que operações desenvolvidas por uma unidade de processamento possam ocasionar contenções e gerar atrasos imprevisíveis na execução de tarefas num outro processador. Neste contexto, sabe-se que a hierarquia de memória utilizada atualmente é um dos principais fatores de incerteza num processador multinúcleo e, por esta razão, atualmente existem diversos estudos voltados à melhoria de sua previsibilidade. Entre eles, o uso de unidades de monitoramento de desempenho (PMU) se mostra como um recurso capaz de auxiliar na tomada de decisões em processos de escalonamento de tarefas e auxiliar na tarefa de tornar mais previsível as operações sobre a hierarquia de memória de sistemas multiprocessados. Sendo assim, este trabalho busca aprimorar o suporte do sistema operacional EPOS a unidades de monitoramento de desempenho. Como parte desta proposta, um mecanismo capaz de auxiliar no processo de escalonamento de tarefas conforme as informações fornecidas pelo monitoramento do sistema é implementado. O impacto de tal recurso na execução de tarefas de tempo real é avaliado ao longo deste trabalho, demonstrando sua capacidade de auxiliar o sistema no cumprimento de requisitos de tempo real.

Palavras-chave: Sistemas embarcados. Sistemas operacionais de tempo real. Processadores multinúcleo. Unidades de monitoramento de desempenho.

ABSTRACT

EPOS is a real-time operating system aimed to the development of embedded systems. Embedded systems are part of our everyday life and can be defined as any computer systems dedicated to the monitoring or control of specific applications. Often, performance requirements in an embedded system include real-time constraints. Regardless of the total workload or the occurrence of failures, every real-time system must meet it's time requirements. To ensure compliance with such deadlines, predictability is an even more important feature than processing speed and this makes the usage of real-time operating systems (RTOS) essential. In modern applications, RTOSs are used to process memory intensive workloads, such as in image processing, multimedia applications, among others. In this scenario, multi-core processors represent a great alternative for reducing costs in certain projects. However, the use of multiprocessors brings with it a number of inconvenient factors. Processors and peripherals are connected to the memory hierarchy, buses, buffers, and various other complex resources. Sharing these features allows operations performed by one processing unit to cause contention and generate unpredictable delays in performing tasks on another processor. In this context, it is known that the memory hierarchy currently used is one of the main factors of uncertainty in a multicore processor and, for this reason, there are currently several studies aimed at improving its predictability. Among them, the use of performance monitoring units (PMU) is shown as a resource capable of assisting decision making in task scheduling processes and assisting in the task of making operations on the memory hierarchy of multiprocessed systems more predictable. Therefore, this work improves the EPOS operating system support to performance monitoring units. As part of this proposal, a mechanism capable of assisting in task scheduling processes according to the information provided by the performance monitoring units is implemented. The impact of such a resource in the execution of real-time tasks is evaluated throughout this thesis, demonstrating its ability to assist the system in complying with real-time requirements.

Keywords: Embedded systems. Real-time operating systems. Multicore processors. Performance monitoring units.

LISTA DE FIGURAS

Figura 1 – Exemplo de hierarquia de memória em processadores multinúcleo. 14
Figura 2 – Exemplo de hierarquia de memória. 18
Figura 3 – Escalas de execução produzidas pelos critérios (a) EDF e (b) RM. 22
Figura 4 – Diagrama de blocos de um escalonador multinúcleo particionado. 23
Figura 5 – Diagrama de blocos de um escalonador multinúcleo global. 24
Figura 6 – Exemplo de configuração para um contador de desempenho da PMU Intel. . 28
Figura 7 – Exemplo de leitura para um contador de desempenho da PMU Intel. 29
Figura 8 – Esquema de memória do recurso DS Area. 30
Figura 9 – Diagrama de classe UML dos componentes de escalonamento de tempo real no EPOS: Thread, Criterion, Scheduler e Scheduling_List. 34
Figura 10 – Diagrama de classe UML dos componentes de escalonamento de tempo real no EPOS: subclasses de Criterion. 35
Figura 11 – Mediador de *hardware* IA32_MMU. 36
Figura 12 – Diagrama de classe UML dos componentes de regiões e segmentos de endereços de memória. 37
Figura 13 – Exemplo de utilização do recurso page_coloring na alocação de memória. . 38
Figura 14 – Diagrama de classe UML da família de mediadores PMU. 39
Figura 15 – Diagrama de classe UML do componente de monitoramento Perf_Mon. . . 40
Figura 16 – Método para configuração de PMC0 utilizando o componente Perf_Mon. . . 40
Figura 17 – Método para leitura de PMC0 utilizando o componente Perf_Mon. 41
Figura 18 – Diagrama de classe UML da família de mediadores PMU proposta. 43
Figura 19 – Diagrama de sequência para o processo de inicialização da PMU no núcleo 0 da CPU (continua). 45
Figura 20 – Diagrama de sequência para o processo de inicialização da PMU no núcleo 0 da CPU (conclusão). 46
Figura 21 – Diagrama de sequência para o processo de inicialização da PMU nos núcleos da CPU, exceto no núcleo 0. 47
Figura 22 – Definições em *Traits* da classe IA32_PMU. 48
Figura 23 – Diagrama de sequência para o tratamento global de interrupções da PMU (continua). 50
Figura 24 – Diagrama de sequência para o tratamento global de interrupções da PMU (conclusão). 51
Figura 25 – Diagrama de sequência para o processo configuração de eventos e monitoramento de *Threads*. 53
Figura 26 – Seção de código utilizada pelas tarefas de tempo real não críticas. 56
Figura 27 – Diagrama de organização do ambiente de testes utilizado. 58

Figura 28 – Interface da aplicação desenvolvida para o gerenciamento do ambiente de testes. 59
Figura 29 – Fator de escala de WCET por conjunto de tarefas. 61
Figura 30 – Fator médio de escala de WCET. 61
Figura 31 – Atraso de deadline médio. 62
Figura 32 – Percentual de deadlines perdidos por tipo de tarefas. 63
Figura 33 – Tempo de execução médio das aplicações. 64
Figura 34 – Seção de código responsável por chamar a inicialização da PMU no núcleo 0 da CPU. 73
Figura 35 – Seção de código responsável por chamar inicialização da PMU nos núcleos da CPU, exceto no núcleo 0. 74
Figura 36 – Seção de código responsável inicializar a PMU no EPOS (continua). 75
Figura 37 – Seção de código responsável inicializar a PMU no EPOS (continuação). . . 76
Figura 38 – Seção de código responsável inicializar a PMU no EPOS (continuação). . . 77
Figura 39 – Seção de código responsável inicializar a PMU no EPOS (continuação). . . 78
Figura 40 – Seção de código responsável inicializar a PMU no EPOS (conclusão). . . . 79
Figura 41 – Seção de código responsável pelo tratamento global de interrupções da PMU (continua). 80
Figura 42 – Seção de código responsável pelo tratamento global de interrupções da PMU (conclusão). 81
Figura 43 – Seção de código responsável pela configuração de eventos para o monitoramento de *Threads*. 82

LISTA DE TABELAS

Tabela 1 – Especificações técnicas do processador Intel i7-2600 57
Tabela 2 – Parâmetros de definição do conjunto de tarefas 0. 83
Tabela 3 – Parâmetros de definição do conjunto de tarefas 1. 84
Tabela 4 – Parâmetros de definição do conjunto de tarefas 2. 85
Tabela 5 – Experimento sobre o conjunto de tarefas 0 utilizando CAP-GS sem ação da PMU. 86
Tabela 6 – Experimento sobre o conjunto de tarefas 1 utilizando CAP-GS sem ação da PMU. 87
Tabela 7 – Experimento sobre o conjunto de tarefas 2 utilizando CAP-GS sem ação da PMU. 88
Tabela 8 – Experimento sobre o conjunto de tarefas 0 utilizando CAP-GS com ação da PMU. 89
Tabela 9 – Experimento sobre o conjunto de tarefas 1 utilizando CAP-GS com ação da PMU. 90
Tabela 10 – Experimento sobre o conjunto de tarefas 2 utilizando CAP-GS com ação da PMU. 91
Tabela 11 – Experimento sobre o conjunto de tarefas 0 utilizando BFD sem ação da PMU. 92
Tabela 12 – Experimento sobre o conjunto de tarefas 1 utilizando BFD sem ação da PMU. 93
Tabela 13 – Experimento sobre o conjunto de tarefas 2 utilizando BFD sem ação da PMU. 94
Tabela 14 – Experimento sobre o conjunto de tarefas 0 utilizando BFD com ação da PMU. 95
Tabela 15 – Experimento sobre o conjunto de tarefas 1 utilizando BFD com ação da PMU. 96
Tabela 16 – Experimento sobre o conjunto de tarefas 2 utilizando BFD com ação da PMU. 97

SUMÁRIO

1 INTRODUÇÃO . . . 12
1.1 OBJETIVO GERAL . . . 16
1.2 OBJETIVOS ESPECÍFICOS . . . 16
1.3 ORGANIZAÇÃO DO DOCUMENTO . . . 16

2 FUNDAMENTAÇÃO TEÓRICA . . . 18
2.1 HIERARQUIA DE MEMÓRIA . . . 18
2.1.1 Memória Cache . . . 19
2.2 ESCALONADORES DE TEMPO REAL . . . 20
2.2.1 Escalonadores de tempo real para sistemas de um único núcleo . . . 21
2.2.2 Escalonadores de tempo real para sistemas multinúcleo . . . 22
2.3 UNIDADES DE MONITORAMENTO DE DESEMPENHO . . . 25
2.3.1 Monitoramento de Desempenho em Processadores Baseados na Microarquitetura Intel *Sandy Bridge* . . . 26

3 EMBEDDED PARALLEL OPERATING SYSTEM . . . 32
3.1 ESCALONAMENTO DE TEMPO REAL MULTIPROCESSADO . . . 33
3.2 PARTICIONAMENTO DE CACHE . . . 35
3.3 SUPORTE DO SISTEMA OPERACIONAL EPOS À PMU . . . 38
3.4 CONSIDERAÇÕES PARCIAIS . . . 41

4 MELHORANDO O SUPORTE À PMU NO EPOS . . . 43
4.1 INICIALIZAÇÃO DA PMU . . . 44
4.2 ROTINA GLOBAL DE SERVIÇO PARA INTERRUPÇÕES DA PMU . 49
4.3 AÇÃO DA PMU NO ESCALONAMENTO DE TAREFAS . . . 52
4.4 CONSIDERAÇÕES PARCIAIS . . . 54

5 AVALIAÇÃO DO MECANISMO DE MONITORAMENTO PROPOSTO 55
5.1 DESCRIÇÃO DO EXPERIMENTO . . . 55
5.2 AMBIENTE DE TESTES . . . 57
5.3 AVALIAÇÃO DO MECANISMO DE AÇÃO DA PMU . . . 59
5.3.1 Fator de Escala do WCET . . . 60
5.3.2 Atraso de Deadline . . . 62
5.3.3 Perda de Deadline . . . 63
5.3.4 Tempo de Execução . . . 64

6 CONSIDERAÇÕES FINAIS 65

REFERÊNCIAS 67

APÊNDICE A – PROCESSO DE INICIALIZAÇÃO DA PMU NO EPOS 73

APÊNDICE B – TRATADOR GLOBAL DE INTERRUPÇÕES PARA A PMU NO EPOS 80

APÊNDICE C – AÇÃO DA PMU NO PROCESSO DE ESCALONAMENTO DE TAREFAS 82

APÊNDICE D – PARÂMETROS DOS CONJUNTOS DE TAREFAS GERADOS 83

APÊNDICE E – RESULTADOS DA AVALIAÇÃO DO MECANISMO DE AÇÃO DA PMU 86

1 INTRODUÇÃO

O *Embedded Parallel Operating System* (EPOS) é um sistema operacional de tempo real multiplataforma, orientado a objetos e baseado em componentes, desenvolvido através de uma metodologia para projeto de sistemas embarcados orientados pela aplicação (FRÖHLICH, 2001; EPOS, 2016).

O EPOS é um sistema voltado ao desenvolvimento de sistemas embarcados e tem sido utilizado em diversos projetos de pesquisa acadêmica e industrial nos últimos anos, como o desenvolvimento de um sistema de rádio definido por *software* (MÜCK; FRÖHLICH, 2011), redes de sensores sem fio (FRÖHLICH; WANNER, 2008), implementação de escalonadores de tempo real uniprocessados em *hardware* (MARCONDES et al., 2009), aplicações com uso eficiente de energia (FRÖHLICH, 2011) e a implementação de um suporte para aplicações multiprocessadas (GRACIOLI, 2014).

Sistemas embarcados fazem parte de nosso dia a dia e podem ser definidos como quaisquer sistemas computacionais dedicados ao monitoramento ou controle de aplicações específicas (BERGER, 2002). Sua interface com o ambiente ocorre, tipicamente, através de sensores e atuadores ao invés de interfaces homem-máquina (SHAW, 2001). Exemplos destes dispositivos incluem eletrodomésticos, celulares, sistemas de geração e distribuição de energia, satélites e componentes de sistemas automotivos.

Muitas vezes, os requisitos de desempenho em um sistema embarcado incluem restrições de tempo real. Segundo Farines, Fraga e Oliveira (2000), um Sistema operacional de tempo real, do inglês *Real-Time Operating Systems* (RTOS), é um sistema computacional capaz de reagir a estímulos oriundos do seu ambiente segundo prazos específicos. Esta definição sugere uma característica extremamente importante nestes sistemas: não apenas os resultados de operações são importantes na execução de tarefas, também são observadas restrições relacionadas a quando estes valores são obtidos.

Em sistemas de tempo real, tarefas são caracterizadas por prazos de execução (*deadlines*) que devem ser atendidos pelo RTOS. Um pensamento comum relaciona a capacidade de um sistema quanto ao cumprimento de prazos com sua velocidade de processamento. Porém, possuir um tempo de resposta inferior não garante que as restrições de tempo para dada aplicação serão realmente cumpridas. De acordo com Halang et al. (2000), previsibilidade é uma característica ainda mais importante para sistemas de tempo real do que sua velocidade no processamento de instruções.

Independente de sua carga total de execução ou da ocorrência de falhas, todo RTOS busca cumprir seus requisitos de tempo. Conforme Farines, Fraga e Oliveira (2000), em sistemas onde as noções de tempo e de concorrência são tratadas explicitamente, o escalonamento de tarefas se

torna ponto central na previsibilidade do comportamento de sistemas de tempo real. Para tal, se faz necessário descrever cada tarefa em termos de seu pior tempo de execução, normalmente referido como *Worst Case Execution Time* (WCET), custo computacional e, eventualmente, período de execução. Através destes parâmetros torna-se possível avaliar em tempo de projeto se existe um método de escalonamento capaz de garantir tais pré-requisitos.

Adicionalmente, tarefas de tempo real podem ser classificadas em duas categorias: tarefas de tempo real críticas, *hard real-time tasks* (HRT), ou tarefas de tempo real não críticas, *soft real-time tasks* (SRT). Basicamente, se o resultado de uma operação possui utilidade mesmo após a perda de seu *deadline* esta é considerada uma tarefa não crítica, caso contrário ela será chamada de tarefa crítica (KOPETZ, 2011). Num cruzamento entre uma estrada e uma ferrovia, por exemplo, uma catástrofe pode ocorrer caso a sinalização de tráfego não interrompa a circulação pela estrada antes de que um trem chegue.

Em aplicações modernas, sistemas de tempo real são utilizados no processamento de elevadas cargas de dados, como no processamento de imagens, aplicações multimídia, simulações cientificas, entre outros. Neste cenário, processadores multinúcleos representam uma ótima alternativa para a redução de custos em determinados projetos.

Para Calandrino e Anderson (2008), a elevada capacidade de processamento de dispositivos multinúcleos pode permitir que um mesmo dispositivo processe maiores cargas de trabalho e ainda seja capaz de desempenhar tarefas de utilidade adicionais. Automóveis modernos, por exemplo, possuem mais de 100 unidades eletrônicas de controle (ECU) que desempenham cada vez mais operações de comodidade e conforto (PITCHER, 2012). Com o uso de microcontroladores multiprocessados este número de ECUs pode ser reduzido consideravelmente, o que implicaria numa redução de custo nesses sistemas.

Por outro lado, o uso de multiprocessadores traz consigo uma série de fatores inconvenientes. Para elevar sua taxa média de desempenho, multiprocessadores apresentam diversos mecanismos na redução de atrasos de acesso em seus componentes. Processadores e periféricos são conectados à hierarquia de memória, barramentos, *buffers* e diversos outros recursos complexos que impactam diretamente na definição do pior caso de execução de uma tarefa.

O compartilhamento destes recursos permite que operações desenvolvidas por uma unidade de processamento possam ocasionar contenções e gerar atrasos imprevisíveis na execução de tarefas num outro processador (GRACIOLI, 2014). Por contenções entende-se situações onde elementos do sistema competem entre si para utilizar determinado recurso compartilhado e necessitam aguardar numa fila até serem servidos.

A hierarquia de memória utilizada atualmente é considerada um dos principais fatores de incerteza num processador multinúcleo (GRACIOLI et al., 2015). Sabe-se que existe certo nível de dependência entre os núcleos de um processador devido à organização de sua hierarquia de memória. Além de compartilhar um único barramento de memória RAM, algumas arquiteturas

de processadores podem compartilhar um ou mais níveis da memória cache. Por consequência deste compartilhamento, o desempenho da memória em plataformas multinúcleos pode variar significativamente dependendo de como os dados estão organizados e como os bancos de memória são compartilhados em um determinado momento (YUN et al., 2014).

Através da Figura 1 um exemplo de hierarquia de memória utilizado em processadores multinúcleos é apresentado. Neste diagrama esquemático é possível observar as conexões entre os núcleos de um processador com diferentes níveis de memória cache e a memória RAM do sistema. Ainda neste diagrama temos o terceiro nível da memoria cache (L3) compartilhado entre todos os núcleos do processador.

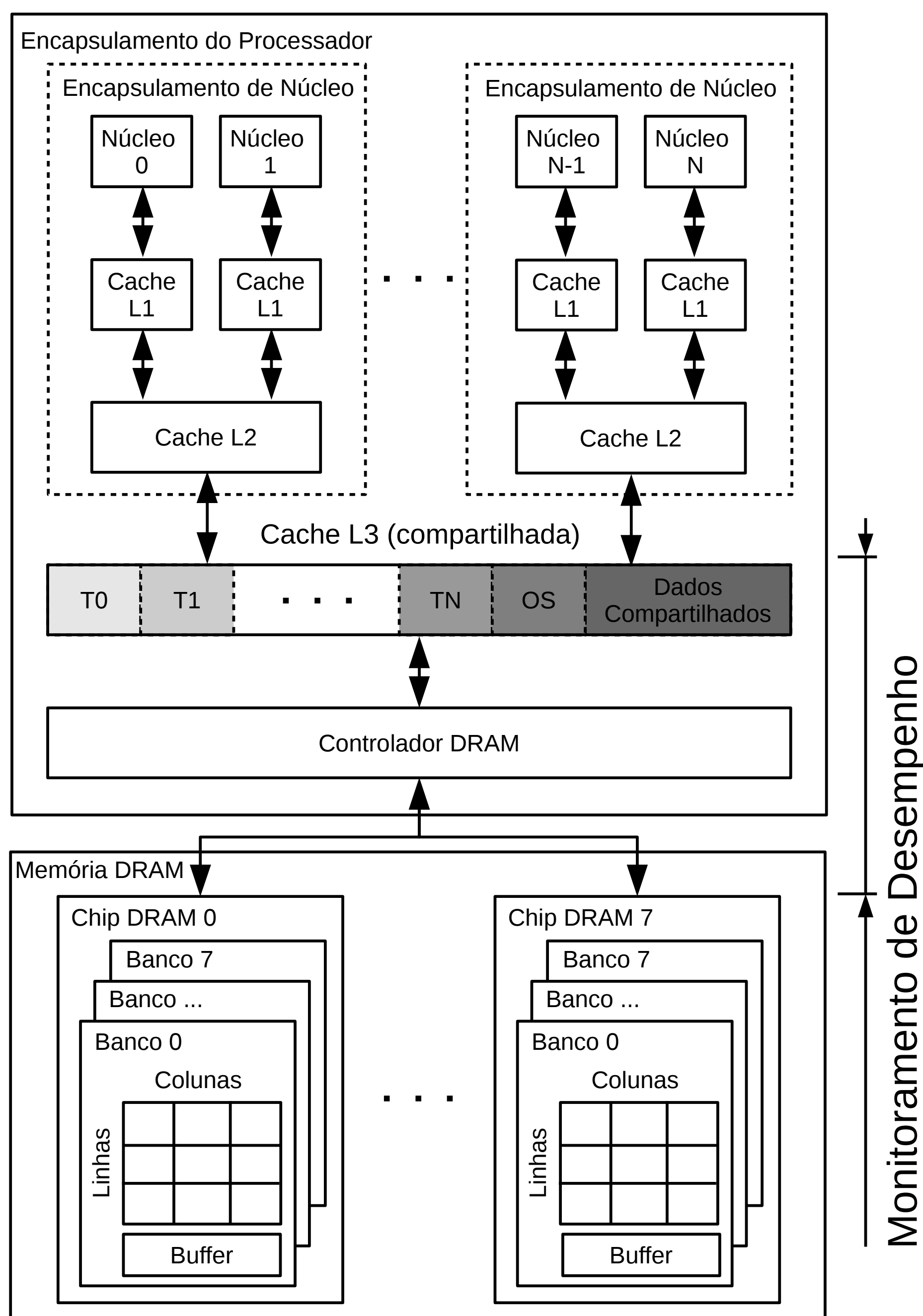
De acordo com Jacob (1999), o uso da memória memória cache se mostra inapropriado para sistemas embarcados de tempo real, uma vez que esta fornece um aumento de desempenho probabilístico. Uma dada informação pode ou não estar contida nesta região de memória e determinar quando estes dados se fazem presentes é uma tarefa extremamente difícil, isto dificulta ao determinar o pior caso de execução de tarefas. No entanto, esta não é uma visão aceita por muitos pesquisadores. Na maior parte dos programas, o pior caso de execução de tarefas é muito maior quando se faz uso da memória cache, mesmo que determinado dado ou instrução seja utilizado apenas uma única vez (LIEDTKE; HÄRTIG; HOHMUTH, 1997).

Todos esses fatores afetam a previsibilidade temporal da hierarquia de memória em sistemas de tempo real. Por este motivo, existe uma necessidade cada vez maior de soluções de gerenciamento de banda de memória e memória cache que garantam determinada qualidade de serviço (YUN et al., 2013).

Atualmente existem diversos tópicos de estudo voltados à atenuação de tais efeitos sobre a hierarquia de memória, como: a utilização de técnicas de particionamento de cache; cache *locking*; ou até mesmo estudos que fazem uso de registradores de desempenho como um recurso de monitoramento. O particionamento de cache é utilizado para reservar regiões de memória ao armazenamento de dados de tarefas específicas, conforme representado na Figura 1. A técnica de cache *locking* bloqueia determinada linha ou região da memória cache e impede a retirada de seus dados.

Unidades de monitoramento de desempenho, ou *Performance Monitoring Unit* (PMU), são compostas de registradores responsáveis por monitorar e contabilizar a ocorrência de determinados eventos durante a execução de um programa. Através destes eventos, torna-se possível verificar o funcionamento interno do processador à medida com que o código de uma aplicação é executado, oferecendo informações que auxiliam o RTOS em sua tomada de decisões durante a execução de programas.

Figura 1 – Exemplo de hierarquia de memória em processadores multinúcleo.



Fonte: Autor, 2017.

A PMU pode ser utilizada, por exemplo, para o cálculo de métricas de desempenho do sistema através da medição de um ou mais eventos ou até para analisar características de regiões de código específicas ao registrar o local onde ocorrem os eventos monitorados (INTEL CORPORATION, 2010; BITZES; NOWAK, 2014). Para tratar dos problemas que afetam a previsibilidade da hierarquia de memória, PMUs podem contabilizar as operações em memória realizadas pelo sistema e auxiliar na tomada de decisões em processos de escalonamento de tarefas. Ainda na Figura 1, temos destacada a região onde tais operações de memória ocorrem.

A presente monografia dá continuidade ao trabalho desenvolvido por Gracioli (2014) em "*Real-Time Operating System Support For Multicore Applications*" que desenvolve uma infraestrutura de suporte para aplicações de tempo real multiprocessadas. Sua implementação se deu no RTOS chamado *Embedded Parallel Operating System* (EPOS).

Este trabalho busca implementar novas funcionalidades ao modelo de escalonador para sistemas de tempo real apresentado por Gracioli (2014). Sua proposta demonstra como o uso de registradores de desempenho é capaz de oferecer informações importantes para seu escalonador e auxiliá-lo em determinadas tomadas de decisão. Como objetivo principal deste trabalho, o suporte do sistema operacional EPOS a unidades de monitoramento de desempenho é aprimorado. Sendo assim, este trabalho apresenta as principais expansões propostas para os mediadores de *hardware* da PMU do EPOS. Como parte deste trabalho, um mecanismo capaz de auxiliar no processo de escalonamento de tarefas conforme as informações fornecidas pelo monitoramento do sistema é implementado. O impacto de tal recurso na execução de tarefas de tempo real é avaliado ao longo deste trabalho, demonstrando sua capacidade de auxiliar de forma dinâmica em operações de escalonamento.

1.1 OBJETIVO GERAL

Aprimorar o suporte do sistema operacional EPOS à unidades de monitoramento de desempenho e avaliar sua operação em conjunto a um mecanismo de particionamento de cache e escalonamento de tempo real na execução de tarefas sintéticas em *hardware* real.

1.2 OBJETIVOS ESPECÍFICOS

O presente trabalho tem por objetivos específicos:

- Aprimorar o suporte do sistema operacional EPOS à unidades de monitoramento de desempenho;
- Adquirir prática no desenvolvimento de sistemas embarcados de tempo real multiprocessados;
- Estudar o modelo de escalonador para sistemas de tempo real proposto por Gracioli (2014);
- Integrar o mecanismo de monitoramento proposto com o recurso de particionamento de memória cache desenvolvido por Gracioli (2014);
- Avaliar a implementação proposta através da execução de diferentes conjuntos de tarefas sintéticas em *hardware* real.

1.3 ORGANIZAÇÃO DO DOCUMENTO

Este documento é organizado da seguinte forma: inicialmente, o capítulo 2 relaciona uma série de conceitos que formam a base deste trabalho; o capítulo 3 apresenta o sistema operacional EPOS, plataforma da qual o algoritmo proposto foi implementado, bem como algumas de suas principais funcionalidades; para o capítulo 4 o projeto e implementação propostos são abordados, seguido no capítulo 5 de uma avaliação dos resultados obtidos; e, por fim, o capítulo 6 conclui este trabalho através de suas considerações finais e propõe temas de trabalhos futuros para o projeto.

2 FUNDAMENTAÇÃO TEÓRICA

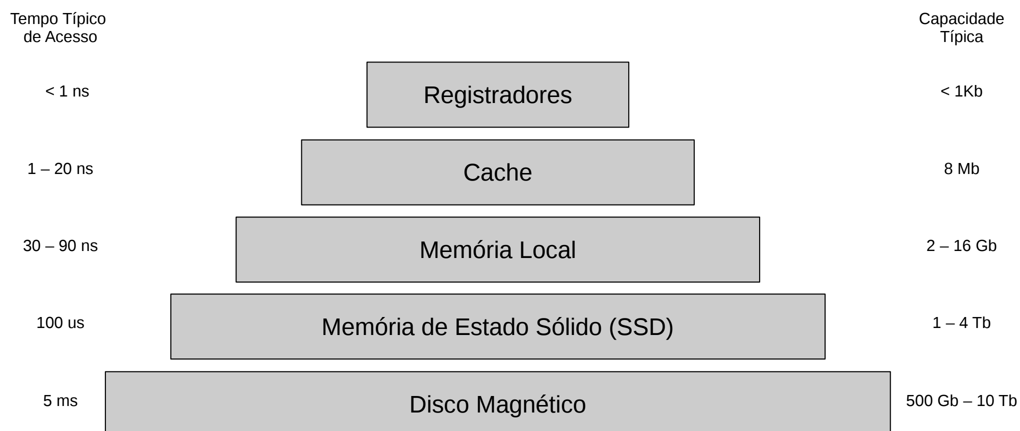
Para atingir os objetivos propostos por este trabalho, se faz necessário compreender uma série de conceitos relacionados a sistemas operacionais de tempo real e à arquitetura de processadores modernos. Sendo assim, este capítulo trás uma visão geral sobre a hierarquia de memória presente em processadores modernos e introduz conceitos básicos sobre escalonamento de tarefas de tempo real e unidades de monitoramento de desempenho.

2.1 HIERARQUIA DE MEMÓRIA

A fim de garantir um bom aproveitamento dos recursos disponíveis num sistema computacional, dispositivos de memória devem ser capazes de acompanhar sua velocidade de processamento (TANENBAUM, 2009a). A medida com que um processador executa instruções, é indesejável que ele faça uma pausa à espera por instruções ou conjuntos de dados.

Há muito tempo se reconhece que para atingir um bom desempenho aplicações requerem acesso rápido à dados e instruções (STALLINGS, 2010a). A solução para este dilema não é contar com um único componente de memória, mas empregar uma hierarquia de memória, conforme representado pela Figura 2.

Figura 2 – Exemplo de hierarquia de memória.



Fonte: Adaptado de Rustad (2013) e Tanenbaum (2009a).

De um modo geral, este modelo é composto pelos registradores internos do próprio processador (CPU), dispositivos de memória cache, memória RAM, discos rígidos, entre outros. Conforme cada um destes dispositivos se aproximam a base desta hierarquia sua capacidade de memória é aumentada assim como seu tempo de acesso, enquanto seu custo de fabricação é reduzido. A chave para o sucesso desta organização é promover uma redução na frequência que o processador precisa acessar sua memória principal.

2.1.1 Memória Cache

Processadores possuem memórias cache em sua arquitetura para acelerar a execução de aplicações em geral. A memória cache para um determinado dispositivo de armazenamento só precisa ser grande o suficiente para manter o conjunto de trabalho, conjunto de instruções e arquivos de dados que determinado aplicativo está executando para ser eficaz. Assim, a maior parte dos acessos em memória serão satisfeitos pela cache e o tempo de acesso geral será o de cache, muito mais rápido do que o dispositivo de armazenamento principal.

O princípio de funcionamento da memória cache é relativamente simples. Toda instrução executada pela CPU que faz referência à memória é primeiramente buscada na cache. Se encontrada, ocorre o chamado acerto de cache, ou cache *hit*, e o dado é enviado imediatamente ao processador. Porém, caso a instrução ou dado não seja localizado na cache um erro de acesso chamado de cache *miss* é apresentado. Neste caso, a informação requisitada é lida da memória principal, ou de um nível mais alto de cache, e armazenada em cache para uso futuro.

O problema com esse arranjo ocorre principalmente em estado estacionário, quando os dispositivos de cache se encontram cheios de dados importantes (JACOB, 1999). Qualquer referência a um objeto que não esteja armazenado neste nível de memória retorna uma informação que já foi referenciada para um nível de memória inferior para dar espaço à informação desejada.

Segundo Stallings (2010b), ainda mais crítica é a situação de sistemas multiprocessados. É comum que tais sistemas possuam um ou dois níveis de memória cache associados a cada processador. Este método, no entanto, cria um problema conhecido como coerência de cache.

Mancuso et al. (2013) caracteriza quatro tipos principais de interferência que ocorrem sobre a cache: quando uma tarefa substitui suas próprias linhas de cache ao carregar dados da memória principal; quando uma tarefa substitui linhas de cache de uma outra tarefa escalonada em sua CPU; quando o sistema operacional (SO) involuntariamente polui a cache durante a execução de suas funções (por exemplo, durante o tratamento de interrupções); ou quando as tarefas que executam simultaneamente em diferentes núcleos acessam uma linha de cache que foi modificada por um ou mais deles. Enquanto os três primeiros tipos de interferência ocorrem em ambos os sistemas com um ou mais núcleos, o último cria uma dependência entre núcleos.

Caso estes processadores estejam autorizados a atualizar suas cópias livremente é possível que haja um conflito entre seus dados, a não ser que seus dispositivos de memória monitorem as demais CPUs do sistema ou recebam uma notificação sobre a alteração de seus dados. Recursos que desempenham esta tarefa são chamados de protocolos de coerência de cache e estão presentes em todo sistema multiprocessado. Sua implementação pode apresentar diferentes níveis de complexidade, variando de protocolos que mantém a coerência em *hardware* até políticas de *software* que previnem a existência de cópias de dados compartilhados (STENSTROM, 1990).

De uma forma geral, prever qual instrução de um programa será executada é uma tarefa extremante difícil, complicando o processo de selecionar quais informações devem ser mantidas

em memória cache. Isto faz com que o armazenamento em memória cache possa ser visto como prejudicial para aplicações em tempo real.

De acordo com Jacob (1999), o problema com o uso em sistemas de tempo real é que elas fornecem um aumento de desempenho probabilístico. Enquanto sistemas de uso geral geralmente visam o aumento médio em sua velocidade de processamento, sistemas de tempo real estão preocupados com a previsibilidade de execução. Uma dada informação pode ou não estar contida nesta região de memória e determinar quando estes dados se fazem presentes é uma tarefa extremamente difícil, o que dificulta ao determinar o pior case de execução de tarefas. Por consequência, aplicações em tempo real podem até mesmo desabilitar os dispositivos de cache de um processador para aumentar sua previsibilidade de acesso em memória (LIEDTKE; HÄRTIG; HOHMUTH, 1997).

Apesar destes problemas, na maior parte dos programas o pior caso de execução de tarefas é muito menor quando se faz uso da memória cache, mesmo que determinado dado ou instrução seja utilizado apenas uma única vez (LIEDTKE; HÄRTIG; HOHMUTH, 1997). Por este motivo existem diversos estudos voltados à atenuação de tais efeitos no comportamento da memória cache.

Conforme apresentado por Gracioli et al. (2015), duas abordagens são mais comumente utilizadas para impor um comportamento mais determinista sobre a memória cache. A primeira abordagem, voltada ao isolamento temporal de tarefas, é o particionamento de cache. Esta técnica divide a cache em partições e as atribui para tarefas ou núcleos específicos. O particionamento de cache isola cargas de trabalho de tarefas que interferem umas com as outras, aumentando assim a previsibilidade do sistema (MANCUSO et al., 2013).

A segunda técnica que tem sido investigada é o bloqueio de cache, ou *cache locking*. Bloqueio de cache se baseia em recursos de hardware presentes em muitas plataformas embarcadas. Especificamente, é possível sinalizar uma determinada linha de cache, ou uma sequência de linhas, como bloqueada, impedindo assim que seu conteúdo seja retornado a memória principal até que uma operação de desbloqueio sucessivo seja realizada.

Porém, mesmo estes recursos não oferecem soluções capazes de tornar as operações em memória cache totalmente previsíveis. Aspectos como *prefetchers* de *hardware*, dispositivos de entrada e saída, protocolos de coerência e arquitetura de processadores ainda são considerados como tópicos de estudo para melhorar a previsibilidade de memória cache atual (GRACIOLI et al., 2015).

2.2 ESCALONADORES DE TEMPO REAL

Frequentemente um computador é utilizado para o processamento de diversas tarefas, ou *threads*, simultaneamente. Nessa situação, quando duas ou mais destas se encontram prontas para serem executadas num mesmo instante, uma escolha tem de ser feita para determinar qual

tarefa será executada a seguir. O componente do sistema operacional que realiza esta escolha é chamado de escalonador, o qual segue uma dada política de escalonamento.

Além de determinar a sequência de execução de tarefas, o escalonador também deve se preocupar em fazer uso eficiente da CPU. A troca de contexto entre *threads*, por exemplo, é um processo custoso para a CPU e deve ocorrer de forma controlada (TANENBAUM, 2009b). Além disso, trocas de contexto normalmente invalidam parte ou até toda a informação de uma tarefa que estava armazenada em memória cache, forçando que estes dados sejam recarregados da memória principal. De uma forma geral, deve-se evitar a ocorrência de grandes taxas de troca de contexto, pois este processo custa um tempo de processamento considerável.

Políticas de escalonamento definem critérios, ou regras, para a ordenação de tarefas (FARINES; FRAGA; OLIVEIRA, 2000). No contexto de sistemas de tempo real, políticas de escalonamentos são definidas com o objetivo de cumprir com os prazos de execução de tarefas, prezando, obviamente, pela previsibilidade do sistema. Estas estratégias de escalonamento atribuem prioridades às tarefas de acordo com suas restrições temporais.

2.2.1 Escalonadores de tempo real para sistemas de um único núcleo

Os critérios *Rate Monotonic* (RM) e *Earliest Deadline First* (EDF) estão entre os critérios de escalonamento mais antigos e melhores conhecidos, propostos por Liu e Layland (1973).

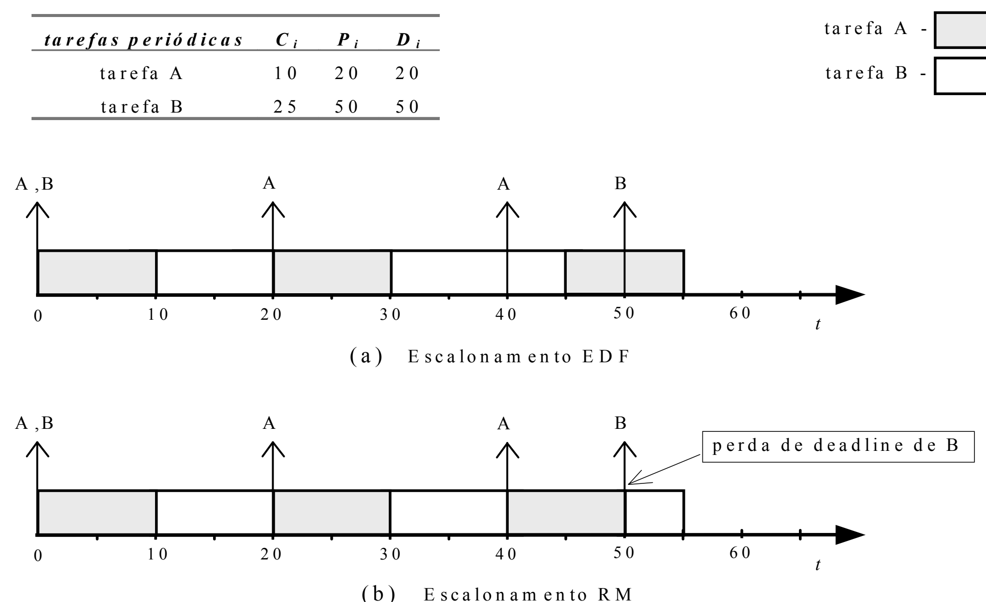
RM é caracterizado por uma política preemptiva e orientada por prioridades, voltado ao escalonamento de tarefas periódicas. Seu esquema obedece uma relação de prioridades fixa, o que o define como um escalonador estático. Sua atribuição de prioridades é definida linearmente com a taxa de execução de cada tarefa, ou seja, quanto menor seu período, maior a prioridade resultante. Liu e Layland (1973) demonstra que a máxima utilização de CPU alcançável para RM converge para apenas 69% de sua capacidade conforme o número de tarefas aumenta. Uma sequência de escalonamento gerada para um conjunto de tarefas com custo computacional superior a este limite irá ocasionar em falhas no atendimento de *deadlines*.

Já a política de *Earliest Deadline First* não se limita a tarefas periódicas, além de utilizar atribuições dinâmicas de prioridades. Em seu algoritmo, a prioridade de cada tarefa é determinada pelo tempo absoluto até seu próximo *deadline*. A tarefa com a *deadline* mais próxima terá sempre a maior prioridade. Sabe-se que o EDF é um critério ótimo para tarefas periódicas, e que ter uma utilização de CPU inferior ou igual a 100% é uma condição necessária e suficiente para a EDF ser capaz de escalonar tarefas (PELOQUIN et al., 2010).

Para esses critérios, Farines, Fraga e Oliveira (2000) apresenta um ótimo exemplo na representação de suas operações, conforme pode ser visto na Figura 3. O conjunto de trabalho é composto pelas tarefas A e B. Cada tarefa apresenta um custo, ou tempo, de processamento C_i, um período de execução P_i e um *deadline* D_i. Sabe-se ainda que a taxa de utilização da CPU neste caso é de 100%.

Assim, esta figura realiza uma comparação entre as escalas de processamento resultante das políticas RM e EDF através de um gráfico de linha do tempo. Através das flechas verticais são sinalizados os instantes de ativação de cada tarefa e, por consequência, de suas *deadlines*.

Figura 3 – Escalas de execução produzidas pelos critérios (a) EDF e (b) RM.



Fonte: Farines, Fraga e Oliveira (2000).

É possível observar como o escalonador RM sempre prioriza a execução da tarefa A, devido ao seu menor período de execução, independente dos prazos de execução das tarefas. Desta forma, a escala de execução apresentada em (b) falha em cumprir com os requisitos desta aplicação. Isso demonstra ainda o quão importante é a definição do critério de escalonamento mais adequado à aplicação desejada.

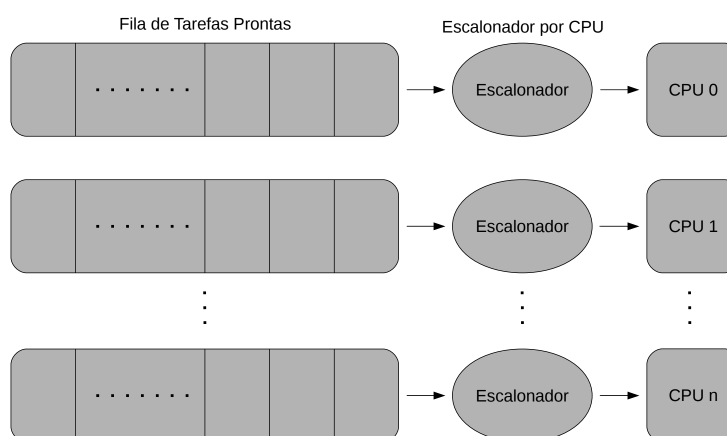
2.2.2 Escalonadores de tempo real para sistemas multinúcleo

Segundo Brandenburg (2011), existem duas formas de se escalonar tarefas em processadores multinúcleos de memória compartilhada: tarefas podem ser escalonadas entre os núcleos de processamento através um único escalonador e uma única fila execução é compartilhada entre todos os núcleos; ou então os núcleos podem ser subdivididos em conjuntos menores com escalonadores próprios e só então filas de execução são atribuídas a cada conjunto. Esta primeira abordagem é chamada de escalonamento global, enquanto a segunda pode ser chamada de escalonamento particionado ou escalonamento agrupado, ou em *clusters*, conforme o número de núcleos por conjunto.

O escalonamento particionado é a abordagem de escalonamento de tempo real multiprocessado mais utilizada na prática em sistemas críticos (BRANDENBURG, 2011). Cada partição é composta por apenas um processador que possui um escalonador e um conjunto de tarefas

próprios. O grande apelo destes critérios decorre do fato de que cada partição pode ser escalonada e analisada utilizando técnicas existentes para processadores de um único núcleo, como as técnicas RM e EDF citadas anteriormente (DAVIS; BURNS, 2011). A Figura 4 exemplifica a forma de organização deste escalonador, onde cada processador possui uma fila de tarefas e escalonador exclusivos.

Figura 4 – Diagrama de blocos de um escalonador multinúcleo particionado.



Fonte: Autor, 2017.

Distribuir a carga de processamento entre todos os núcleos é preferível por várias razões. Em primeiro lugar, há pouco benefício para deixar processadores disponíveis não utilizados. Mesmo para os núcleos em uso, sempre é necessário reservar parte de sua capacidade para compensar *overheads*. Portanto, o uso de técnicas de escalonamento particionado pode fornecer maior flexibilidade na execução das tarefas de um sistema.

No entanto, a reutilização das teorias de escalonamento para uniprocessadores possui um preço: dividir o conjunto total de tarefas em partições menores e garantir que nenhum núcleo de processamento seja sobrecarregado não é um processo simples (BRANDENBURG, 2011). Na prática, algoritmos de empacotamento (*Bin Packing*) são utilizados para encontrar a melhor distribuição de tarefas por núcleo.

Encontrar uma distribuição ótima de conjuntos geralmente é um problema intratável. Porém, as heurísticas de busca disponíveis na literatura são capazes de encontrar soluções quase ideais que possuem desempenhos relativamente bons (GRACIOLI, 2014). Para conjuntos de tarefas que podem ser particionados em um dado número de processadores, geralmente é possível encontrar uma distribuição válida de tarefas, particularmente para conjuntos de tarefas que não utilizem de toda a capacidade de processamento disponível.

Na literatura é possível encontrar diversas alternativas de algoritmos de empacotamento. Neste trabalho, no entanto, apenas dois deles serão abordados: *Best Fit Decreasing* (BFD) e *Color-Aware task Partitioning with Group Splitting* (CAP-GS). No algoritmo BFD a cada alocação todas as tarefas e núcleos disponíveis são avaliados. Seu particionamento é um processo

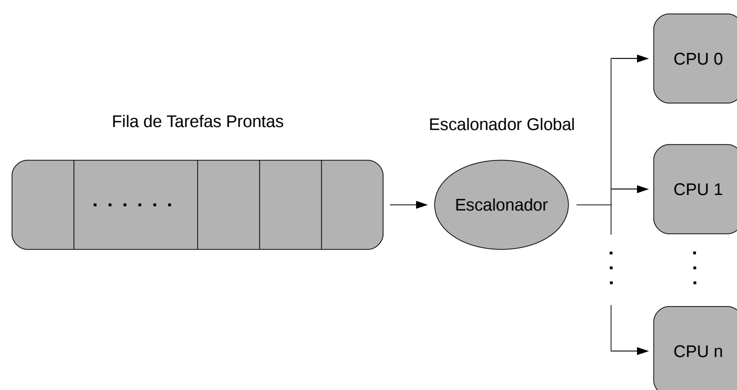
iterativo, onde a tarefa de maior utilização é selecionada e alocada no núcleo com maior nível de utilização, sem ultrapassar o limite de utilização definido pelo critério de escalonamento adotado (MARTELLO; TOTH, 1990; MARANHãO, 2009).

Já o algoritmo CAP-GS assume que cada tarefa é atribuída a uma partição de memória cache que serve como parâmetro durante seu processo de empacotamento. A proposta deste algoritmo é apresentada através de Gracioli e Fröhlich (2014) e Gracioli (2014). Tarefas que pertencem as mesmas partições (cores) são agrupadas e todo o grupo é atribuído ao mesmo núcleo de processamento.

CAP-GS permite ainda que um grupo de tarefas tenha uma utilização maior do que a capacidade de um processador, considerando o limite de utilização de uma política de escalonamento específica. Quando isso acontece, a tarefa com menor utilização é removida do grupo e o particionamento do grupo é executado novamente. Essa etapa é repetida até que o grupo seja particionado corretamente ou após a verificação de todas as suas tarefas, buscando agrupar o maior nível de utilização de tarefas possível de uma determinada partição sobre o mesmo núcleo. Maiores informações sobre o particionamento de cache são apresentadas na Seção 3.2.

Em um sistema de escalonamento global existe apenas uma única fila de tarefas prontas para serem executadas, compartilhada entre todos os núcleos de processamento. Através desta o escalonador determina quais das tarefas disponíveis serão executadas nos processadores disponíveis em um dado momento, de acordo com sua política de escalonamento e prioridades. Isto elimina a necessidade de distribuir tarefas em conjuntos distintos e soluciona o problema de empacotamento, o qual é o principal responsável pela perda de capacidade de processamento em técnicas de escalonamento particionado (BRANDENBURG, 2011). A Figura 5 exemplifica a forma de organização deste escalonador, onde todos os processadores compartilham uma única fila de tarefas e escalonador.

Figura 5 – Diagrama de blocos de um escalonador multinúcleo global.



Fonte: Autor, 2017.

Escalonadores globais e particionados podem ser vistos como casos especiais do chamado escalonador agrupado. Esta técnica de escalonamento combina algumas características de

ambos escalonadores. Quando um único núcleo compõe um *cluster* este se torna equivalente ao sistema de escalonador particionado, enquanto quando todos os *n* núcleos são agrupados num único *cluster* este se torna um escalonador global (BRANDENBURG, 2011). Assim como nos escalonadores particionados, tarefas são atribuídas a diferentes *clusters* de acordo com suas heurísticas de particionamento. Entre as tarefas de cada *cluster*, no enquanto, estas são escalonadas de forma "global" entre seus processadores ao longo de sua execução.

2.3 UNIDADES DE MONITORAMENTO DE DESEMPENHO

Outro recurso essencial para o desenvolvimento deste trabalho são as Unidades de Monitoramento de Desempenho, ou *Performance Monitoring Unit* (PMU). Essas fazem parte do projeto de processadores modernos atuais e são capazes de auxiliar na identificação de gargalos do sistema (INTEL CORPORATION, 2015). Processadores fabricados pela Intel, por exemplo, apresentam PMUs desde a família P6 de processadores Pentium, enquanto a AMD oferece PMUs desde a série de processadores Athlon (BANDYOPADHYAY, 2004). Monitores de desempenho estão presentes até mesmo em certas arquiteturas de microcontroladores, como pode ser observado em processadores ARM e Freescale (ARM, 2008; FERNáNDEZ; CAZORLA, 2014).

Unidades de monitoramento de desempenho são compostas de registradores responsáveis por monitorar e contabilizar a ocorrência de determinados eventos durante a execução de um programa. A partir de uma lista, a qual depende do fabricante e modelo de processador utilizado, o usuário é capaz de selecionar eventos a serem monitorados, como o número de instruções executadas, o número de acertos ou erros de acesso na cache, número de substituições de linhas da cache, entre outros (FERNáNDEZ; CAZORLA, 2014; GRACIOLI; FRÖHLICH, 2011). Através destes eventos, torna-se possível verificar o funcionamento interno do processador à medida com que o código de uma aplicação é executado.

As PMUs também proporcionam recursos para gerar interrupções de *hardware* após a ocorrência de um certo número de eventos. Este recurso pode ser utilizado, por exemplo, para o cálculo de métricas de desempenho do sistema através da medição de um ou mais eventos. Esta metodologia é muitas vezes referida como *profiling*, ou a identificação do perfil de execução de uma dada aplicação (INTEL CORPORATION, 2010). Através deste recurso também se torna possível analisar as características de regiões de código específicas ao registrar o local onde ocorrem os eventos monitorados (BITZES; NOWAK, 2014).

Contudo, esses recursos muitas vezes sofrem de um conjunto comum de problemas. Devido a quantidade de contadores disponíveis, um número limitado de eventos podem ser monitorados num dado momento. Esta é uma restrição severa, uma vez que detectar gargalos de desempenho pode requerer um conhecimento detalhado sobre diversos componentes do processador (AZIMI; STUMM; WISNIEWSKI, 2005; SPRUNT, 2002). Outra característica importante a ser notada nas PMUs se refere ao registro do ponteiro de instrução. Mesmo utilizando recursos de interrupção, é provável que este ponteiro tenha progredido durante o

intervalo entre a ocorrência de um evento e a interrupção efetiva do processamento de dados, levando assim a uma localização imprecisa da região de código que disparou o evento (BITZES; NOWAK, 2014).

Os eventos que podem ser monitorados pelas PMUs são tipicamente de baixo nível e específicos da microarquitetura de um processador. Como resultado, a correta interpretação de tais resultados se torna uma tarefa difícil sem um conhecimento detalhado da CPU. De um modo geral, mesmo os eventos mais conhecidos como o número de ciclos de processamento por instrução, a taxa de erros de acesso na cache ou contenções no barramento de memória só podem ser obtidos através da medição da frequência com que uma combinação de diferentes eventos ocorrem (AZIMI; STUMM; WISNIEWSKI, 2005).

De acordo com a microarquitetura observada, processadores Intel, por exemplo, podem variar em número de registradores de desempenho, assim como suas características de operação e eventos monitorados. Por este motivo, este projeto tomou como principal referência os recursos de processadores baseados na microarquitetura Intel *Sandy Bridge*. Este modelo faz parte de uma plataforma moderna de processadores e se encontra disponível no Laboratório de Integração *Software/Hardware* para o desenvolvimento deste trabalho. Com base no manual de desenvolvimento de *software* para as arquiteturas Intel® 64 e IA-32 (INTEL CORPORATION, 2015), suas principais características são apresentadas na seção a seguir.

2.3.1 Monitoramento de Desempenho em Processadores Baseados na Microarquitetura Intel *Sandy Bridge*

Os processadores Intel, dependendo da microarquitetura (por exemplo, Nehalem, Core, Atom, etc), possuem diferentes versões de PMU. Cada versão oferece diferentes características e um número variável de contadores de desempenho. Por exemplo, a PMU versão 2 possui dois contadores de desempenho por CPU que podem ser configurados com eventos de propósito geral e três contadores de desempenho fixos que contam eventos específicos. A PMU versão 3, por sua vez, estende a versão 2 e fornece suporte para multiprocessamento simultâneo (*hyper-threading*), até 8 contadores de desempenho de propósito geral e contadores de eventos precisos. No entanto, todas as três versões compartilham um conjunto de eventos pré-definidos da própria arquitetura, como a contagem do número de ciclos de processamento, número de erros de acesso no último nível da cache e o número de instruções suspensas. Atualmente já foi desenvolvida até mesmo uma PMU versão 4, utilizada em microarquiteturas mais recentes.

Sistemas operacionais executados por estes processadores podem fazer uso da instrução chamada "CPUID" para verificar a disponibilidade de recursos arquitetônicos de cada processador. Suas variáveis de retorno são capazes de indicar o suporte de um processador ao monitoramento de desempenho, sua versão, número de contadores, comprimento de contagem, entre outros recursos adicionais.

O processo de configurar um evento na unidade de monitoramento de desempenho envolve a programação de registradores de seleção de eventos, que são registradores específicos de modelo. Esta expressão se refere aos "*Model-Specific Registers*" (MSR), conjunto de registradores responsáveis por definir o modo de operação de uma CPU. O processo de seleção de um evento envolve a configuração dos registradores chamados de seletores de eventos de desempenho (IA32 PERFEVTSELx) correspondentes a um contador de desempenho físico específico (IA32_PMCx). Por exemplo, para configurar o PMC0 para contar o número de ciclos de processamento executados, o PERFEVTSEL0 deve ser configurado com a máscara de evento CPU_CLK_UNHALTED_THREAD_P (0x00) e duas máscaras de unidade que definem as condições em que o evento deve ser contado.

Outro importante registrador é chamado de IA32_PERF_GLOBAL_STATUS. Este MSR fornece uma série de *bits* de *status* capaz de indicar ao *software* se algum dos contadores de desempenho excedeu seu limite de contagem, evento denominado de *overflow*. A PMU, por padrão, bloqueia a contagem de eventos referente a um dado contador enquanto seu respectivo *bit* indicativo de *overflow* se encontra ativo. O MSR IA32_PERF_GLOBAL_OVF_CTL, por sua vez, permite que cada um destes *bits* sejam reiniciados individualmente pelo sistema. Esta ação é essencial para que o *hardware* retorne a operar normalmente e continue contando a ocorrência dos eventos selecionados.

Para completar o conjunto de registradores que controlam os contadores de desempenho PMC, temos o chamado MSR_PERF_GLOBAL_CTRL. Este MSR fornece uma série de *bits* responsáveis por habilitar a operação de cada contador. A partir do momento em que o seletor de eventos para um dado contador foi devidamente configurado e seu *overflow* verificado, seu respectivo *bit* de habilitação pode ser acionado para dar início ao processo de monitoramento de desempenho.

Qualquer contador de desempenho pode ser configurado para gerar uma interrupção na ocorrência de um *overflow*. Este recurso é capaz de parar o processamento de determinado processador e chamar o tratador de sua rotina de serviço. Desta forma, pode-se então coletar informações sobre o estado do processador ou programa logo após a ocorrência de um dado evento. Tal recurso conta ainda com o registrador chamado OVF_PMI para determinar qual dos contadores foi o responsável por gerar uma interrupção, recurso essencial para quando vários contadores são configurado para gerar interrupções.

Em resumo, pode-se citar uma série de passos que devem ser desempenhados para a correta configuração de eventos utilizando PMU:

- O primeiro passo para configurar os registradores da PMU é desabilitar suas interrupções do sistema. Este passo somente se aplica quando o recurso de interrupções da PMU é utilizado;
- Em seguida, o contador a ser configurado deve ser desabilitado;

- Após estes passos pode-se definir o valor inicial do PMC desejado;
- Somente então deve-se selecionar o evento a ser monitorado;
- Por fim, o contador em questão e o recurso de interrupções da PMU podem ser novamente habilitados.

O processo de leitura dos PMCs é realizado de forma similar:

- O primeiro passo para realizar a leitura dos contadores da PMU é desabilitar suas interrupções do sistema. Novamente, este passo somente se aplica quando o recurso de interrupções da PMU é utilizado;
- Em seguida, o contador a ser verificado deve ser desabilitado;
- Após estes passos pode-se ler o registador responsável por armazenar a contagem de um evento previamente configurado;
- E, por fim, o contador em questão e o recurso de interrupções da PMU podem ser novamente habilitados.

Para melhor representar este processo, o exemplo a seguir foi elaborado. Nos trechos de código apresentados nas Figuras 6 e 7 os registradores descritos ao longo desta seção são manipulados.

Figura 6 – Exemplo de configuração para um contador de desempenho da PMU Intel.

```
1 void count_cpu_llc_miss(long long event_initial_value)
2 {
3     // Desabilita as interrupções do sistema
4     CPU::int_disable();
5
6     // Desabilita a contagem do evento configurado em PMC0
7     *IA32_PERF_GLOBAL_CTRL = *IA32_PERF_GLOBAL_CTRL & ~PMC0_ENABLE;
8
9     // Define um valor inicial de contagem para PMC0
10    *IA32_PMC0 = event_initial_value;
11
12    // Define o evento a ser monitorado por PMC0, habilitando sua interrupção e contagem de eventos
13    *IA32_PERFEVTSEL0 = (ENABLE | OS | USR | LLC_MISSES);
14
15    // Habilita a contagem do evento configurado em PMC0
16    *IA32_PERF_GLOBAL_CTRL = *IA32_PERF_GLOBAL_CTRL | PMC0_ENABLE;
17
18    // Habilita as interrupções do sistema
19    CPU::int_enable();
20 }
```

Fonte: Autor, 2017.

Na Figura 6 é possível observar a atribuição da variável *event_initial_value* ao contador PMC0 através da operação realizada na linha 10, definindo assim o valor inicial de contagem

do respectivo PMC. A seleção do evento a ser monitorado pelo PMC0 (LLC_MISSES) e a habilitação de seu recurso de interrupção de contagem são ainda realizados na linha 13. Já o código apresentado na Figura 7 realiza a leitura do valor armazenado no PMC0 e o retorna através da variável *counter_value* através das operações nas linhas 13 e 22, respectivamente.

Em ambos trechos de código a função “CPU::int_disable()” foi utilizada para desabilitar as interrupções do sistema durante o processo de manipulação dos registradores da PMU, enquanto “CPU::int_enable()” reabilita este recurso. Da mesma forma, o registrador IA32_PERF_GLOBAL_CTRL é utilizado para desabilitar a contagem de PMC0 enquanto seu contador é configurado/lido e reabilitar sua operação ao fim destas operações.

Processadores Intel oferecem ainda um recurso chamado *Debug Store Save Area* (DS *Area*). Através do registrador IA32_DEBUGCTL se torna possível utilizar um mecanismo responsável por capturar registros dos caminhos de execução, interrupções e exceções e salvá-las numa região de memória específica. A este processo é dado o nome de *Branch Trace Store* (BTS). O monitoramento dos caminhos, interrupções e exceções que ocorrem durante a execução de um dado programa podem fornecer informações úteis para rotinas de depuração de código, possibilitando a criação de um método para determinar o caminho e as decisões tomadas para alcançar um local de código particular. Porém, apesar de oferecer um conjunto de informações de grande utilidade para a depuração do sistema operacional, o monitoramento através do *DS Area* pode reduzir significativamente o desempenho do processador e deve ser utilizado com cuidado.

Figura 7 – Exemplo de leitura para um contador de desempenho da PMU Intel.

```
1  unsigned long long read_pmc0()
2  {
3      // Define uma variável temporária para o armazenamento de PMC0
4      unsigned long long counter_value;
5
6      // Desabilita as interrupções do sistema
7      CPU::int_disable();
8
9      // Desabilita a contagem do evento configurado em PMC0
10     *IA32_PERF_GLOBAL_CTRL = *IA32_PERF_GLOBAL_CTRL & ~PMC0_ENABLE;
11
12     // Realiza a leitura do valor armazenado em PMC0
13     counter_value = *IA32_PMC0;
14
15     // Habilita a contagem do evento configurado em PMC0
16     *IA32_PERF_GLOBAL_CTRL = *IA32_PERF_GLOBAL_CTRL | PMC0_ENABLE;
17
18     // Habilita as interrupções do sistema
19     CPU::int_enable();
20
21     // Retorna o valor lido em PMC0
22     return counter_value;
23 }
```

Fonte: Autor, 2017.

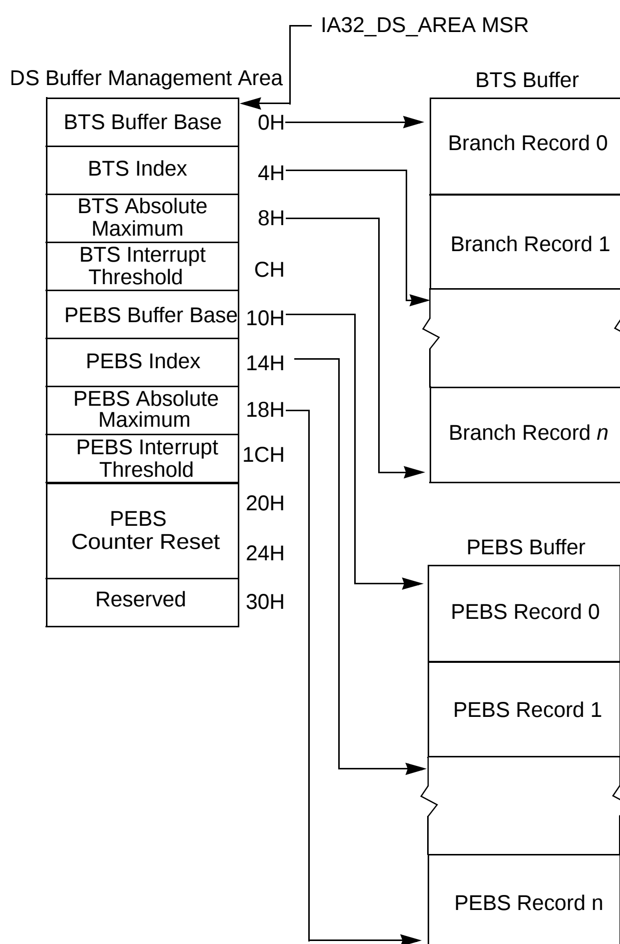
O mecanismo de *Debug Store* oferece ainda um conjunto de contadores de eventos precisos (PEBS). Este conjunto especial de contadores são utilizados juntamente do recurso de

interrupções para ativar as operações de *DS Area* e armazenar o estado de seu processador na ocorrência do evento selecionado.

DS Save Area é dividido em três partes: sua área de gerenciamento de *buffer*, seu *buffer* para armazenamento dos dados de BTS e o *buffer* de PEBS. Tais regiões de memória podem ser verificadas através do esquema de organização de memória da Figura 8. A área de gerenciamento de *buffer* é usada para definir a localização e o tamanho dos *buffers* de BTS e PEBS. O processador utiliza então a área de gerenciamento de *buffer* para controlar os registros de caminhos e/ou PEBS em seus respectivos *buffers*. O endereço do primeiro *byte* da área de gerenciamento de *buffer* do *DS Area* é especificado através do registrador IA32_DS_AREA.

O uso dos recursos de *DS Store* requer sua devida configuração, além da usual seleção dos eventos de PEBS quando necessário. Seu mecanismo de operação necessita de uma região específica de memória para armazenar os dados obtidos durante a execução de programas. A localização e tamanho desta região de memória deve ser estabelecida pelo desenvolvedor do sistema conforme o esquema apresentado pela Figura 8.

Figura 8 – Esquema de memória do recurso DS Area.



Fonte: (INTEL CORPORATION, 2015).

Internamente, o recurso *Debug Store* utiliza três campos da área de gerenciamento para percorrer os *buffers* de BTS e PEBS, estes são chamados de *Base*, *Index* e *Absolute Maximum* são responsáveis por indicar o endereço inicial de cada *buffer*, o endereço inicial da próxima região de armazenamento disponível e o endereço final absoluto de seu *buffer*, respectivamente. Uma vez que a capacidade de armazenamento dos *buffers* de *DS Area* pode variar conforme sua implementação, a área de gerenciamento conta ainda com um campo específico para a geração de interrupções. A partir do momento em que o índice de BTS ou PEBS ultrapassa o endereço definido neste campo (*Interrupt Threshold*), a interrupção da PMU é acionada.

Finalmente, a configuração do recurso *DS Store* pode ser resumida através da sequência a seguir:

- Definir uma região de memória para o gerenciamento do recurso DS Area conforme o esquema apresentado pela Figura 8;
- Escrever o endereço base da área de gerenciamento no registrador IA32_DS_AREA_MSR;
- Configurar a entrada do contador de desempenho no vetor de interrupções do sistema;
- Escrever uma rotina de serviço de interrupção para lidar com a interrupção de PMCs;
- Atribuir o endereço do tratador de interrupção de PMCs desenvolvido a sua entrada no vetor de interrupções.

Os passos relacionados à configuração do sistema de interrupções somente se aplicam quando PEBS é utilizado e/ou quando se deseja monitorar a capacidade de armazenamento disponível nos *buffers* de *DS Area*.

Maiores informações sobre as unidades de monitoramento de desempenho dos processadores Intel podem ser obtidas através do manual técnico: "*Intel® 64 and IA-32 Architectures Software Developer's Manual: system programming guide*", conforme Intel Corporation (2015). Neste manual cada registrador e operação relacionado ao uso de monitoradores de desempenho são detalhados, incluindo a definição dos parâmetros utilizados pelas Figuras 6 e 7 (PMC0_ENABLE, ENABLE, OS, USR e LLC_MISSES).

3 EMBEDDED PARALLEL OPERATING SYSTEM

Em sua grande maioria, os processadores produzidos hoje em dia são desenvolvidos para sistemas computacionais dedicados que executam um único aplicativo ou um pequeno conjunto de aplicações previamente conhecidas. Em contraste com sistemas computacionais genéricos, esses sistemas possuem requisitos de execução muito específicos, como baixo consumo de energia e restrições de memória, que não são devidamente cumpridos por sistemas operacionais de propósito geral (MALL, 2009).

A impossibilidade de prever quais aplicações serão executadas por determinado sistema operacional genérico fazem com que estes disponibilizem um amplo conjunto de serviços, destinados a tornar todos os seus recursos disponíveis para qualquer aplicação executada. Porém, oferecer uma grande variedade de recursos nem sempre auxilia no cumprimento de certos requisitos.

Segundo Fröhlich (2001), a metodologia de sistemas embarcados orientados pela aplicação, *Application-Driven Embedded System Design*, ou simplesmente ADESD, surge como uma alternativa a este cenário. Esta técnica propõe uma estratégia sistemática para construir sistemas como arranjos de componentes de *software* adaptáveis. Ao invés de possuírem propriedades padrões de *hardware* e/ou *software*, os recursos oferecidos por tal sistema são definidos diretamente pelos requisitos da aplicação, permitindo que este seja personalizado de acordo com necessidades específicas.

O EPOS toma por base a metodologia ADESD para orientar o desenvolvimento tanto de componentes de *software* quanto de *hardware*, totalmente adaptáveis para o cumprimento dos requisitos de aplicações singulares. Essa estratégia, além de reduzir drasticamente o número de abstrações exportadas, permite que programadores expressarem facilmente as necessidades de aplicações relacionadas com o sistema computacional a ser gerado (FRÖHLICH, 2001). Em outras palavras, temos uma plataforma de *hardware* e *software* com suporte exclusivo aos recursos associados a aplicação desejada.

Sua interface de aplicação é composta por dois tipos de estruturas: abstrações (ou componentes)[1] e mediadores de *hardware*. O suporte específico às plataformas é implementado através de mediadores de *hardware*, que são funcionalmente equivalentes aos *drivers* de dispositivo.

O EPOS suporta uma diversidade de arquiteturas, variando entre 8 e 64 bits. Entre elas, pode-se citar: IA32, AVR8, PPC32, MIPS e ARM7. O EPOS também oferece estruturas de dados comuns como listas, vetores e tabelas de dispersão.

[1] Componentes no EPOS são classes C++ com interfaces e comportamentos bem definidos, independentes de plataforma.

Conforme apresentado na introdução deste documento, alguns elementos do EPOS se fazem essenciais para o desenvolvimento do trabalho proposto. Estes são sua arquitetura de escalonamento de tarefas, sua capacidade para atuação sobre a hierarquia de memória cache (particionamento de cache) e seu suporte ao uso de unidades de monitoramento de desempenho presente no microprocessador em questão (Intel i7-2600, microarquitetura *Sandy Bridge*).

Sendo assim, a Seção 3.1 apresenta a estrutura dos componentes de escalonamento de tempo real multiprocessado no EPOS e suas principais funcionalidades. Em seguida, os mediadores de *hardware* responsáveis pelo gerenciamento de memória do EPOS e seu mecanismo voltado ao particionamento de cache são descritos na Seção 3.2. Por fim, este capítulo é encerrado pela Seção 3.3 ao explorar o suporte do EPOS ao monitoramento de desempenho dos processadores Intel.

3.1 ESCALONAMENTO DE TEMPO REAL MULTIPROCESSADO

O EPOS oferece uma série de políticas de escalonamento, desde escalonadores tradicionais, como Round-Robin e escalonamento por prioridade de tarefas, até escalonadores de tempo real multiprocessados. Esse sistema operacional utiliza um único escalonador que opera de acordo com o critério de escalonamento definido durante sua configuração (EPOS, 2016).

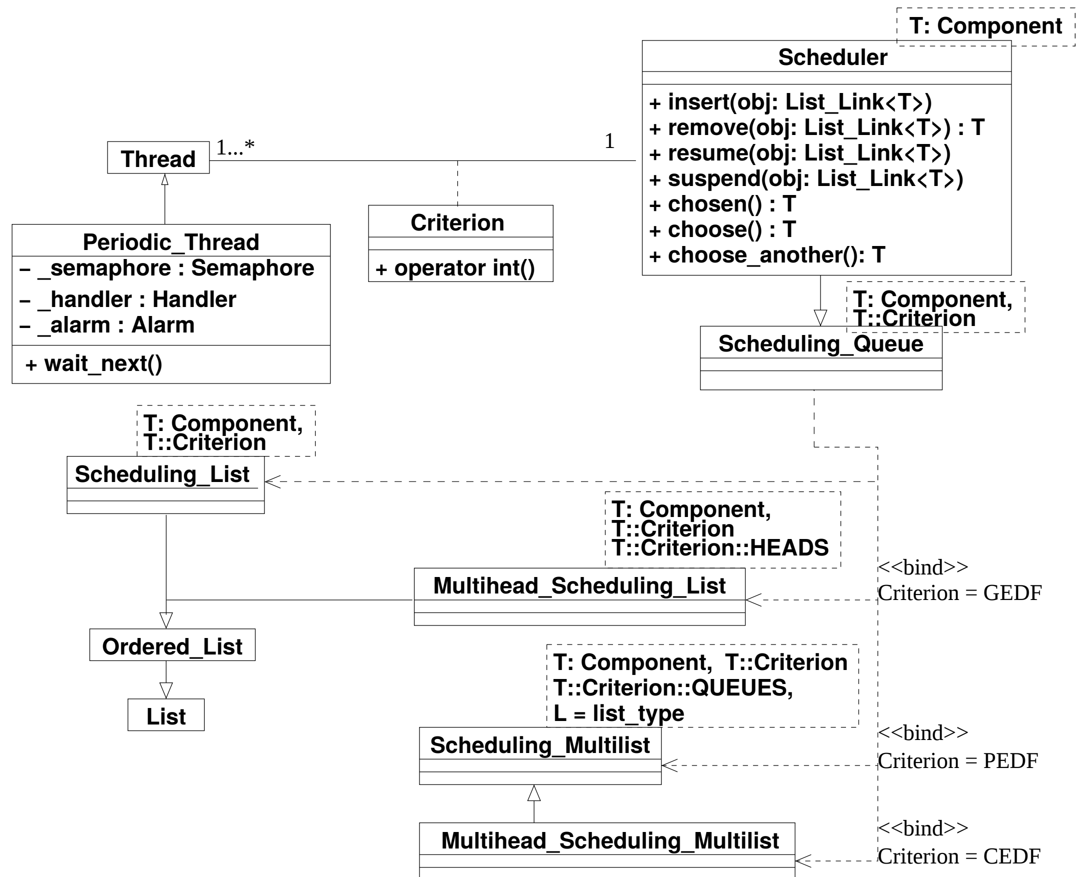
A Figura 9 mostra os três principais componentes responsáveis pelo suporte de escalonamento de tempo real no EPOS: Thread, Scheduler e Criterion. A classe *Thread* representa uma tarefa aperiódica e define seu fluxo de execução, com seu próprio contexto e pilha. A classe implementa as funcionalidades de *threads* tradicionais, como as operações para suspender, retomar, “dormir” e “acordar”.

A classe *Periodic_Thread* estende a classe *Thread* para fornecer suporte para tarefas periódicas e agregar mecanismos relacionados à re-execução de tarefas. O método *wait_next* executa uma operação *p* em um semáforo que força a *thread* a dormir durante o período definido. A implementação do Semáforo segue o conceito tradicional de semáforos proposto por Dijkstra (1968). Quando uma interrupção do *timer* ocorre, seu manipulador (um alarme) executa uma operação *v* no semáforo para liberar e “acordar” a tarefa. O construtor da *thread* periódica é responsável por criar esse alarme.

Em tempo de compilação, o critério de escalonamento é definido na classe *Trait*[2] (por exemplo, *“typedef Scheduling_Criteria::GEDF Criterion“* define o critério de escalonamento como G-EDF). Como o EPOS utiliza recursos de metaprogramação estática em C++, todas as dependências são resolvidas em tempo de compilação, sem nenhuma sobrecarga de tempo de execução.

[2] Um *Trait* é um artefato de metaprogramação que processa as características de um componente em tempo de compilação, de modo que outros componentes e metaprogramas possam trabalhar sobre ele.

Figura 9 – Diagrama de classe UML dos componentes de escalonamento de tempo real no EPOS: Thread, Criterion, Scheduler e Scheduling_List.

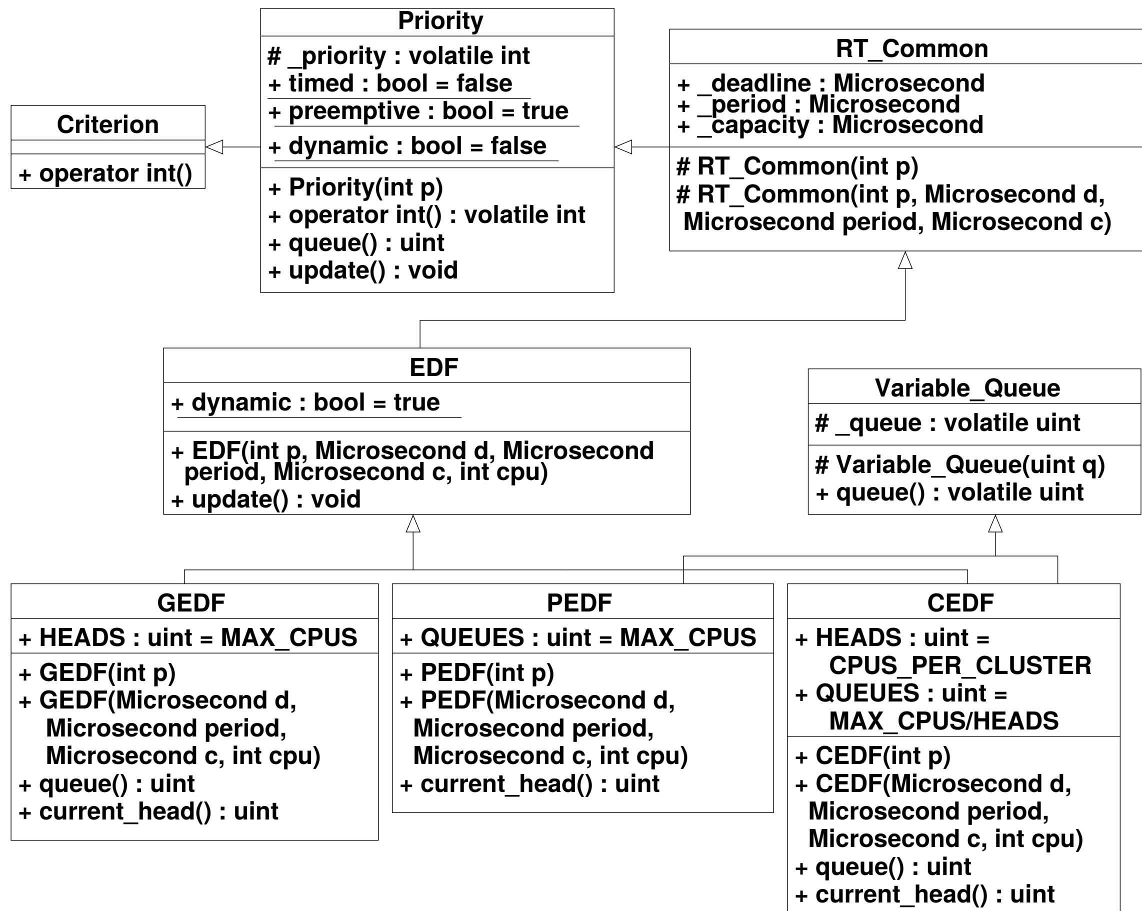


Fonte: (GRACIOLI; FRÖHLICH, 2011).

A classe *Scheduler* representada na Figura 9 define a estrutura que realiza o escalonamento de tarefas em conjunto com as subclasses *Criterion* da Figura 10. O EPOS reduz a complexidade de manutenção dessa hierarquia e promove a reutilização de código ao remover as políticas de escalonamento (aqui representadas pelas sub-classes *Criterion*) de seu mecanismo (por exemplo, implementações de estrutura de dados como listas e pilhas). A estrutura de dados na classe *Scheduler* usa o critério de programação definido para ordenar suas tarefas. Cada classe de critério basicamente define a prioridade de uma tarefa, que é mais tarde usada pelo escalonador na tomada de decisões, e outras características de sua política, como preempção e temporização, por exemplo.

A Figura 10 apresenta a hierarquia de classes de alguns dos critérios implementados no EPOS. Os critérios G-EDF, P-EDF e C-EDF herdam da classe *RT_Common*, que define o *deadline*, período e WCET de uma tarefa. Cada critério define duas variáveis estáticas que serão usadas mais tarde pela fila de escalonamento: *QUEUES* e *HEADS*. *QUEUES* define o número de listas e *HEADS* define o número de ”cabeças“ em cada lista de escalonamento. A cabeça de uma lista de escalonamento representa uma tarefa em execução.

Figura 10 – Diagrama de classe UML dos componentes de escalonamento de tempo real no EPOS: subclasses de Criterion.



Fonte: (GRACIOLI; FRÖHLICH, 2011).

Por exemplo, o critério G-EDF tem apenas uma lista de agendamento (QUEUS é um) e HEADS é igual ao número de CPUs no processador atual. P-EDF tem *QUEUES* igual ao número de CPUs e HEADS igual a um. O C-EDF combina características dos critérios G-EDF e P-EDF: *QUEUES* é igual ao número de *clusters* e *HEADS* é igual ao número de CPUs em cada *cluster*. A Figura 10 mostra apenas os critérios baseados em EDF, no entanto, o mesmo princípio é usado para outros escalonadores, como os já citados RM, Round-Robin, por prioridades, entre outros.

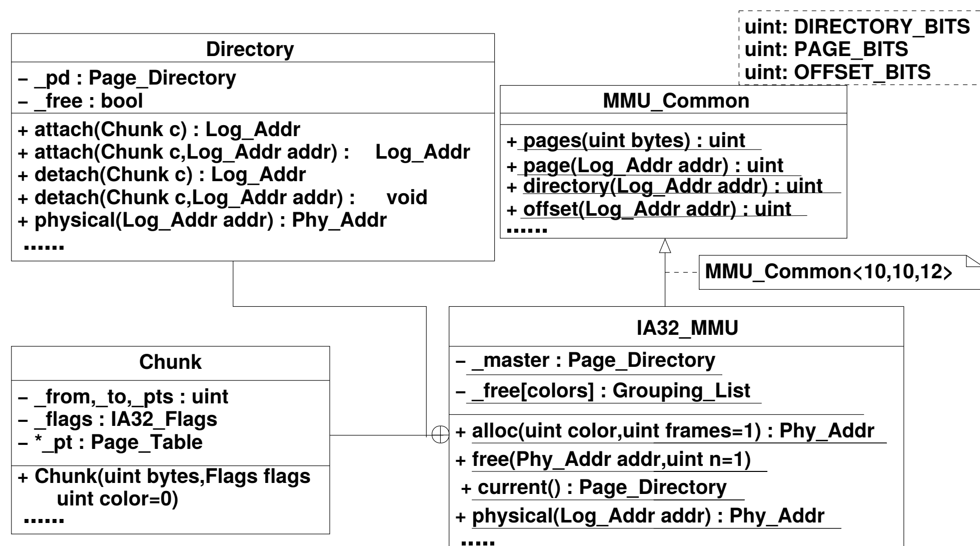
3.2 PARTICIONAMENTO DE CACHE

Sistemas operacionais portáteis enfrentam um desafio em termos de gerenciamento de memória: algumas plataformas de computação possuem unidades de gerenciamento de memória (MMU) sofisticadas, enquanto outras plataformas não oferecem suporte para mapear e proteger regiões de endereçamento (GRACIOLI; FRÖHLICH, 2011). A arquitetura do EPOS agrupa detalhes referente à proteção de espaço de endereçamento, tradução e alocação de memória física nos mediadores de *hardware* da MMU, o que permite que os componentes de gerenciamento de me-

mória sejam altamente portáteis em praticamente qualquer plataforma, desde microcontroladores simples até processadores *multicore* complexos (POLPETA; FRÖHLICH, 2004).

A Figura 11 mostra parte da família de mediadores de MMU. A classe Common_MMU fornece funções básicas e comuns a todas as arquiteturas. Diferentes classes especializam a classe base implementando funções dependentes de arquitetura. A classe IA32_MMU implementa o suporte para paginação de 32-bits disponível em processadores Intel x86, incluindo a manipulação de tabelas de páginas e diretórios. Uma lista agrupada implementa a estratégia de alocação de memória e mantém um registo da memória física disponível. Cada elemento da lista representa uma região de memória física livre. O método de inicialização da MMU inicializa a lista agrupada de acordo com a memória disponível.

Figura 11 – Mediador de *hardware* IA32_MMU.



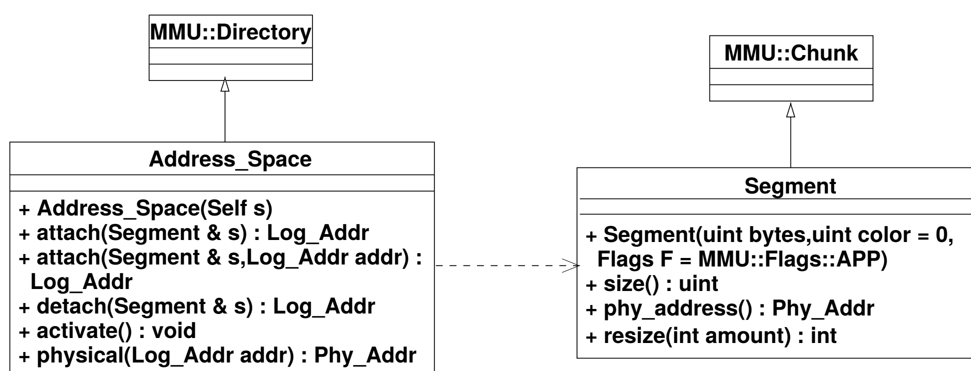
Fonte: (GRACIOLI; FRÖHLICH, 2011).

A Figura 12 descreve os componentes do sistema que fornecem a memória disponível principal para as aplicações. *Segment* representa um intervalo, ou *Chunk*, de memória que armazena código e dados arbitrários. Quando um componente de sistema cria um segmento, a classe MMU::Chunk aloca e mapeia a quantidade de memória solicitada. O componente *Address_Space*, ou espaço de endereços, é um contêiner para segmentos de memória responsável por gerenciar um espaço de memória física correspondente.

No entanto, aplicações não utilizam diretamente os componentes *Address_Space* e *Segment* na alocação de memória. Em vez disso, duas instâncias de classe *heap* diferentes usam os componentes de gerenciamento de memória descritos e fornecem memória livre para o sistema operacional e aplicativos. As alocações dinâmicas de memória do sistema operacional, como a criação de pilhas de *threads* e objetos do sistema, vão para a *heap* do sistema, enquanto os pedidos da aplicação vão para a *heap* de aplicações.

Quando o sistema de particionamento de cache do EPOS é habilitado a lista agrupada e a *heap* da aplicação da MMU tornam-se vetores de partições ou ”cores“, uma vez que esta técnica é chamada de particionamento por coloração de páginas (*page coloring*). Durante o processo de inicialização do sistema, o mediador da MMU preenche cada lista com páginas associadas a uma cor correspondente. Da mesma forma, o componente *Segment* fornece, em sua interface, uma maneira de especificar a partir de qual cor (ou seja, a partir de qual lista agrupada da MMU) uma *heap* deve alocar memória. Para liberar adequadamente uma região de memória, a cor definida é utilizada para encontrar a partir de qual *heap* um endereço de memória foi alocado.

Figura 12 – Diagrama de classe UML dos componentes de regiões e segmentos de endereços de memória.



Fonte: (GRACIOLI; FRÖHLICH, 2011).

Os operadores *new* e *delete* executam as operações de alocação de memória dinâmica na linguagem C++. No EPOS este operador foi modificado para receber, além do número solicitado de *bytes*, o número da cor desejada para a alocação. Se a aplicação não especificar uma cor, o novo operador usará a cor 0. Além disso, o novo operador não aloca memória se a cor solicitada for maior que a cor máxima definida na classe *Traits*.

A partir destas modificações, toda operação de alocação de memória passa a alocar oito *bytes* (dois inteiros) a mais do que o tamanho solicitado: o primeiro inteiro contém o tamanho dos dados (o argumento *bytes*) e o segundo inteiro contém o número da cor. Assim, o operador *delete* se torna capaz de liberar o tamanho correto de dados no *heap* correspondente.

Um exemplo de uso do recurso *page coloring* é apresentado através da Figura 13. Conforme a configuração da MMU definida em *Traits*, o mecanismo de particionamento de cache pode ser habilitado. Através da operação desenvolvida pela linha 7 temos o operador *new* usual é utilizado na alocação de memória para um vetor de inteiros de tamanho *ARRAY_SIZE*. Com a utilização de page_coloring, no entanto, o processo de alocação de *array* realizado pelo trecho de código apresentado é desenvolvido pela operação da linha 4. Neste caso pode-se observar que um parâmetro adicional foi utilizado: COLOR_2. O valor representado por esta variável é definido pela estrutura *alloc_priority* e utilizado pela lista da *heap* de aplicação para determinar a partição de memória em que *array* será alocado.

Figura 13 – Exemplo de utilização do recurso page_coloring na alocação de memória.

```
1  // Alocação de memória para um vetor de inteiros de tamanho ARRAY_SIZE
2  if (Traits<IA32_MMU>::page_coloring)
3      // Caso page_coloring esteja habilitado, alocação é realizada na heap com cor COLOR_2
4      array = new (COLOR_2) int[ARRAY_SIZE];
5  else
6      // Caso page_coloring esteja habilitado, alocação é realizada através do operador new usual
7      array = new int[ARRAY_SIZE]
```

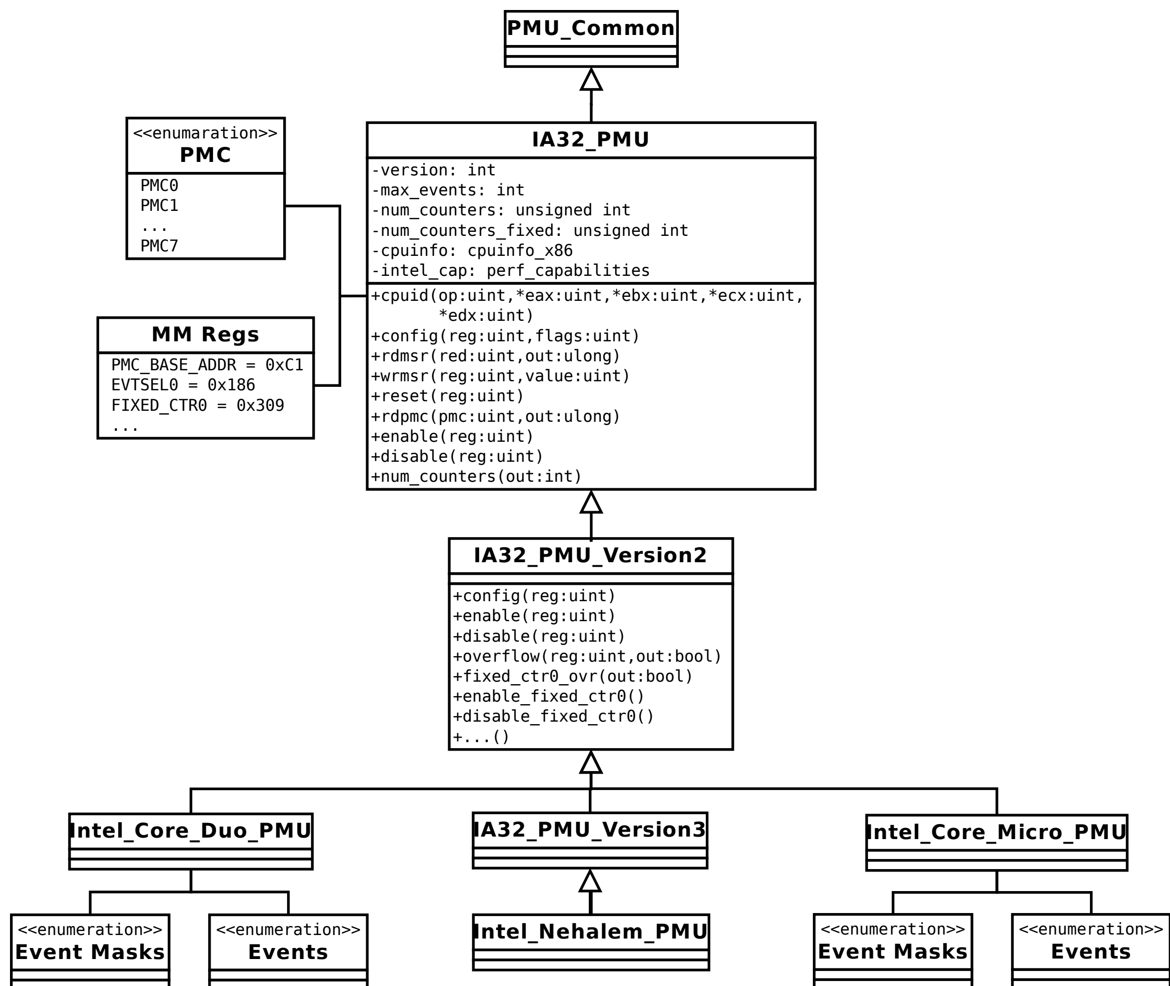
Fonte: Autor, 2017.

3.3 SUPORTE DO SISTEMA OPERACIONAL EPOS À PMU

Atualmente o EPOS possui dois mediadores de *hardware* para PMUs, sendo uma desenvolvida para a família de PMUs da Intel e outra para a arquitetura de processadores ARM. Seu gerenciamento se faz pelo componente chamado monitor de desempenho, que fornece um conjunto de métodos para configurar e ler eventos de *hardware* específicos. Sua utilização abstrai dos usuários a complexidade de configurar, escrever e ler os contadores de *hardware*. A Figura 14 mostra o diagrama de classes UML da família de mediadores para as PMUs dos processadores Intel.

Os mediadores de *hardware* projetados para a família PMU representam a organização de monitoradores de desempenho da Intel descritos na Seção 2.3.1, conforme proposto por (GRACIOLI; FRÖHLICH, 2011). A classe base IA32_PMU implementa a Intel PMU versão 1 e serviços comuns tanto para a versão 2 e 3, incluindo os eventos de arquitetura pré-definidos. Além disso, esta classe declara registradores e PMCs mapeados na memória dos processadores suportados. As classes IA32_PMU versão 2 e 3 estendem a classe base e implementam serviços específicos apenas disponíveis em cada uma destas versões, disponibilizando, por exemplo, funções voltadas ao gerenciamento de PMCs conforme o número de contadores disponíveis em casa versão. Finalmente, os eventos de *hardware* disponíveis estão listadas por classes de microarquiteturas específicas. Por exemplo, as PMUs Intel Core Micro e Intel Nehalem listam todos as máscaras de eventos disponíveis para as microarquiteturas Intel Core e Intel Nehalem, respectivamente.

Figura 14 – Diagrama de classe UML da família de mediadores PMU.

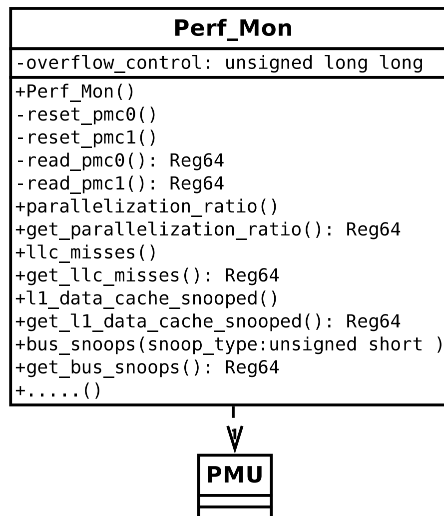


Fonte: Adaptado de (GRACIOLI; FRÖHLICH, 2011).

A interface do mediador de *hardware* pode ser utilizado por um componente independente da plataforma. Este componente é o responsável pelas configurações de "baixo nível" do sistema, como a definição dos eventos a serem monitorados e seus respectivos contadores. Além disso, um componente independente de plataforma também pode multiplexar os contadores de *hardware* a fim de ultrapassar a limitação do número de contadores disponíveis. Técnicas de multiplexação dividem o uso de contadores ao longo do tempo, proporcionando aos usuários uma visão de que existem mais contadores de *hardware* do que os processadores realmente são capazes de oferecer.

Por fim, a Figura 15 mostra o componente de monitoramento do desempenho (Perf_Mon). Ele fornece um conjunto de métodos para abstração de código, sendo responsável por configurar e ler várias taxas de eventos de *hardware* específicos. O componente esconde dos usuários toda a complexidade de configurar, escrever e ler os contadores de desempenho. Além disso, este também proporciona meios para a manipulação de um possível *overflow* nos contadores.

Figura 15 – Diagrama de classe UML do componente de monitoramento Perf_Mon.



Fonte: (GRACIOLI; FRÖHLICH, 2011).

De forma a demonstrar o uso dos componentes de monitoramento de desempenho do EPOS, os trechos de código apresentados pelas Figuras 6 e 7 foram reescritos na forma de métodos da classe Perf_Mon. O código apresentado nas Figuras 16 e 17 utiliza um conjunto de métodos implementados nos componentes da PMU e dispensa o acesso direto aos registradores do processador.

Figura 16 – Método para configuração de PMC0 utilizando o componente Perf_Mon.

```
1  void Perf_Mon::count_cpu_llc_miss(long long event_initial_value)
2  {
3      // Desabilita as interrupções do sistema
4      CPU::int_disable();
5
6      // Desabilita a contagem do evento configurado em PMC0
7      PMU::disable(PMU::EVTSEL0);
8
9      // Define um valor inicial de contagem para PMC0
10     PMU::wrpmc(PMU::PMC0, event_initial_value);
11
12     // Define o evento a ser monitorado por PMC0, habilitando sua interrupção e contagem de eventos
13     PMU::config(PMU::EVTSEL0, (PMU::ENABLE | PMU::OS | PMU::USR |
14                                PMU::LLC_MISSES));
15
16     // Habilita a contagem do evento configurado em PMC0
17     PMU::enable(PMU::EVTSEL0);
18
19     // Habilita as interrupções do sistema
20     CPU::int_enable();
21 }
```

Fonte: Autor, 2017.

Figura 17 – Método para leitura de PMC0 utilizando o componente Perf_Mon.

```
1  unsigned long long Perf_Mon::read_pmc0()
2  {
3      // Define uma variável temporária para o armazenamento de PMC0
4      unsigned long long counter_value;
5
6      // Desabilita as interrupções do sistema
7      CPU::int_disable();
8
9      // Desabilita a contagem do evento configurado em PMC0
10     PMU::disable(PMU::EVTSEL0);
11
12     // Realiza a leitura do valor armazenado em PMC0
13     counter_value = PMU::rdpmc(PMU::PMC0);
14
15     // Habilita a contagem do evento configurado em PMC0
16     PMU::enable(PMU::EVTSEL0);
17
18     // Habilita as interrupções do sistema
19     CPU::int_enable();
20
21     // Retorna o valor lido em PMC0
22     return counter_value;
23 }
```

Fonte: Autor, 2017.

Nesta nova versão, o processo de configuração do contador PMC0 é apresentado na Figura 16. A definição de seu valor inicial de contagem é realizada na linha 10 através da função “PMU::wrpmc(Reg32 counter, long long value)”, enquanto a configuração de seu evento de contagem é desenvolvida na linha 13 através de “PMU::config(Reg32 event_select_register, Reg32 config_flags)”. O código apresentado na Figura 17, por sua vez, realiza a leitura do valor de PMC0 através da função “PMU::rdmsr(Reg32 counter)”.

Novamente, as funções “CPU::int_disable()” e “CPU::int_enable()” são utilizadas para gerenciar a habilitação das interrupções do sistema durante o processo de configuração/leitura de PMC0. A habilitação de sua contagem, no entanto, passa agora a ser desenvolvida pela função “PMU::enable_PMC0()”.

3.4 CONSIDERAÇÕES PARCIAIS

Este capítulo apresentou uma visão geral sobre o RTOS EPOS. O conceito por trás de sua arquitetura foi apresentado, além de sua estrutura de escalonamento de tarefas e gerenciamento de memória. Em especial, este capítulo abordou o suporte do SO ao monitoramento de desempenho em processadores Intel, o qual é realizado através do mediador de *hardware* IA32_MMU e do componente de monitoramento Perf_Mon.

Conforme demonstrado ao longo desta seção, o uso de tais componentes oferece um conjunto de métodos que promovem a abstração de *hardware* do sistema. O uso da classe Perf_Mon dispensa ao usuário o acesso direto aos registradores do processador. As constantes e definições necessárias para a configuração e seleção dos eventos a serem monitorados, como PMC0,

EVTSEL0, ENABLE, OS, USR e LLC_MISSES, se fazem presentes no mediador de *hardware* que se refere à PMU utilizada. Desta forma o EPOS possibilita com que o monitoramento de desempenho de suas aplicações seja facilmente realizado.

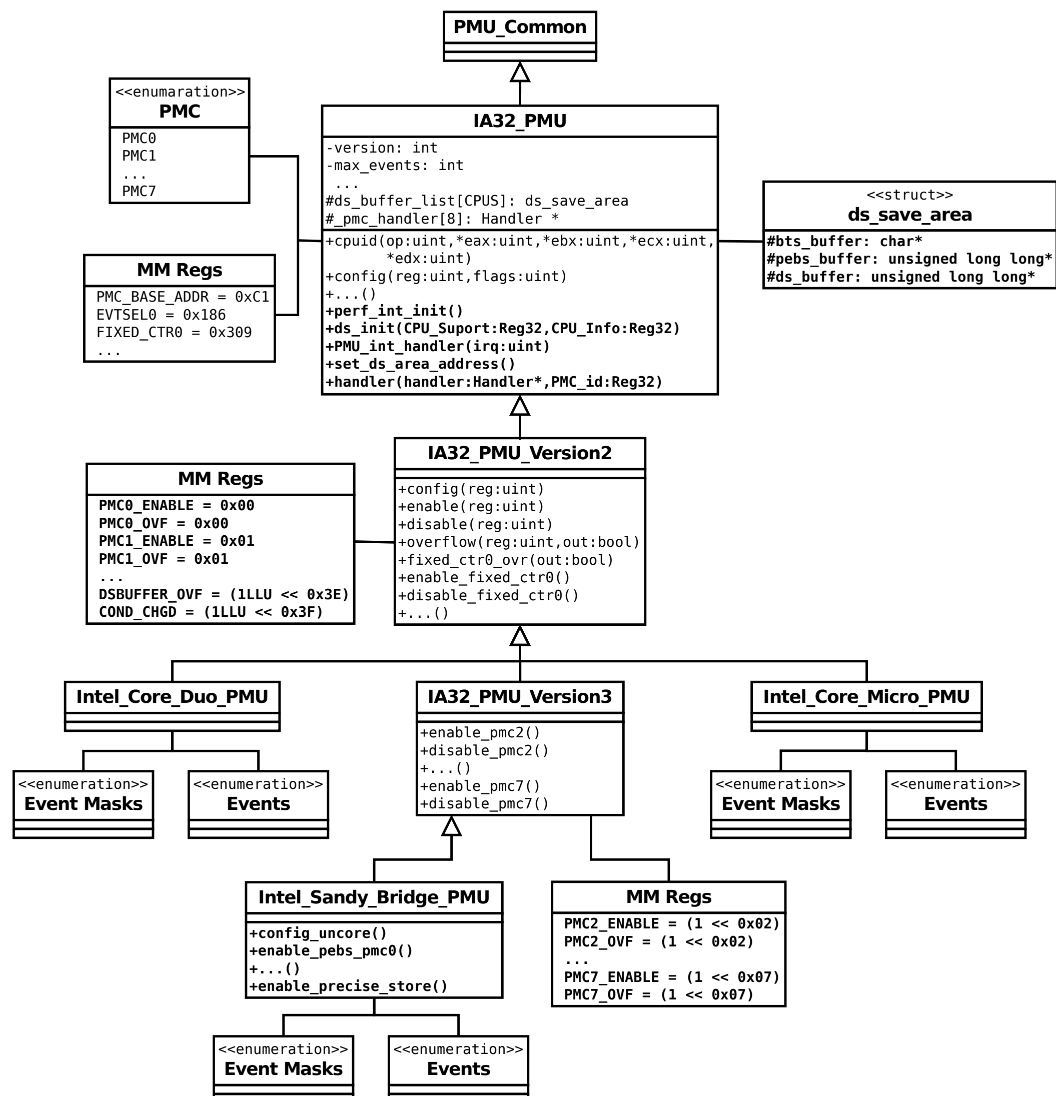
No entanto, embora a arquitetura atual do sistema possibilite o uso da PMU, ainda há margem para que este processo seja aprimorado. Por exemplo, novos métodos podem ser implementados em Perf_Mon para simplificar sua relação com as classes da PMU e melhorar a abstração deste componente. Se tratando de sua relação com processadores Intel, o EPOS ainda não é capaz de utilizar todos os recursos relacionados a suas unidades de monitoramento, como o uso de suas interrupções ou até mesmo do mecanismo de depuração *DS Area*. Estas oportunidades de melhoria ao suporte do EPOS ao monitoramento de desempenho fazem parte dos objetivos propostos para este trabalho e sua implementação é tratada no capítulo a seguir.

4 MELHORANDO O SUPORTE À PMU NO EPOS

Neste capítulo será apresentado um conjunto de extensões desenvolvidos para o EPOS ao longo deste trabalho com o objetivo de aprimorar seu suporte à família de PMUs Intel. De uma forma geral, a proposta deste trabalho apresenta uma revisão sobre os mediadores de *hardware* de sua PMU (IA32_PMU) e monitoramento de desempenho (Perf_Mon). A principal contribuição deste trabalho se dá pela implementação de rotinas que possibilitam o uso dos recursos de interrupção e *Debug Store* oferecidos por processadores Intel modernos.

Sendo assim, o diagrama de classes representado na Figura 18 traz as principais modificações propostas para os componentes da PMU, destacando em negrito os métodos e atributos adicionados ao sistema.

Figura 18 – Diagrama de classe UML da família de mediadores PMU proposta.



Fonte: Autor, 2017.

Entre as principais modificações propostas por este trabalho como um todo, temos:

- Implementação de métodos voltados ao gerenciamento e inicialização do recurso de interrupção da PMU;
- Implementação de mecanismo capaz de atuar no processo de escalonamento de tarefas;
- Implementação de métodos que promovem uma maior abstração de *hardware* à classe IA32_PMU;
- Definição dos registradores que compõem o recurso *DS Area*;
- Implementação de métodos voltados ao gerenciamento do recurso *DS Area*;

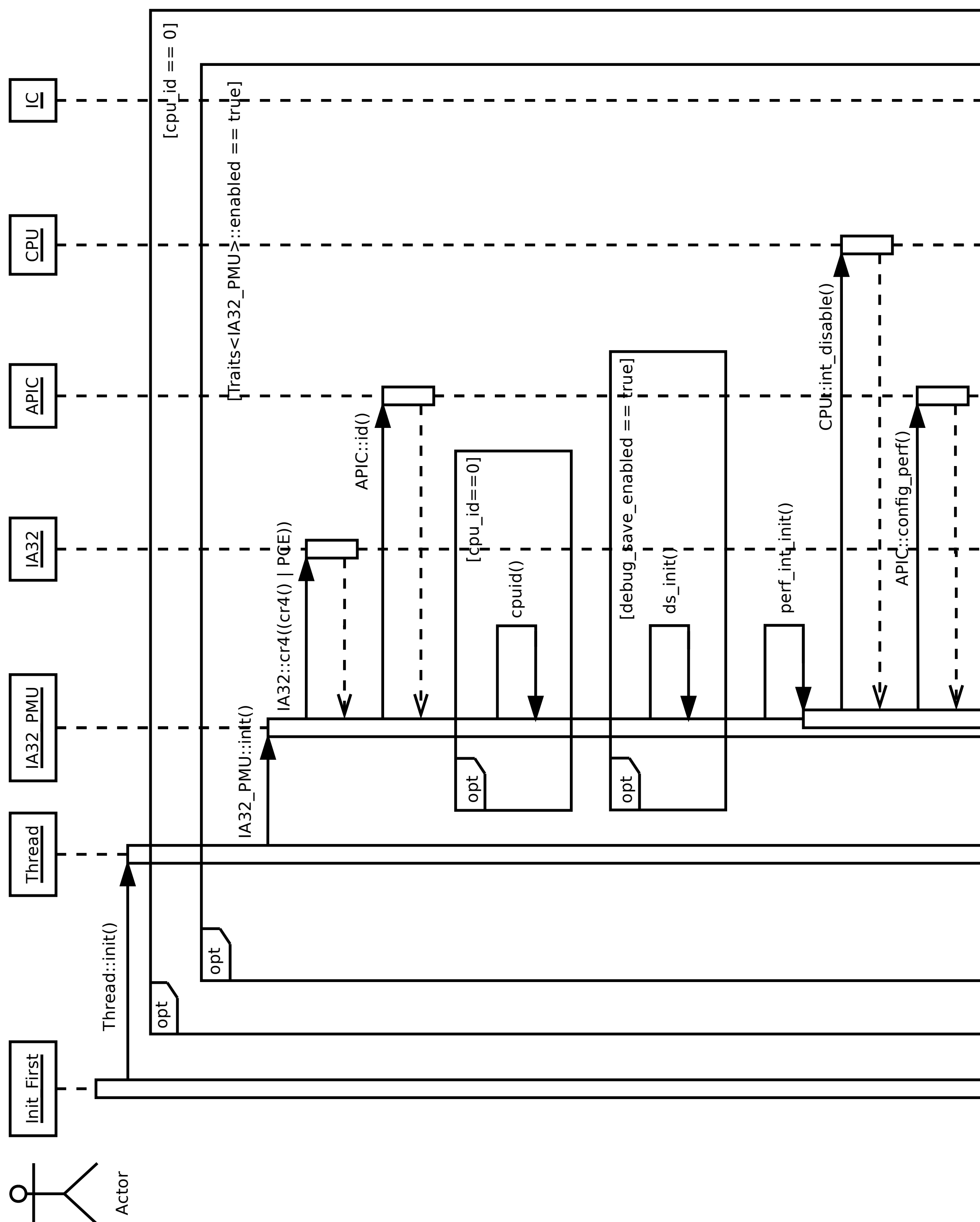
Para melhor abordar as melhorias do suporte à PMU no EPOS propostas por este trabalho, este capítulo foi dividido da seguinte forma: na Seção 4.1 o processo de inicialização da PMU no EPOS é apresentado, incluindo as modificações propostas; através da Seção 4.2 o método utilizado para o tratamento de interrupções da PMU é descrito; e, finalmente, o capítulo é encerrado ao explorar o mecanismo capaz de utilizar o monitoramento de desempenho para tomar ações sobre o processo de escalonamento de tarefas.

4.1 INICIALIZAÇÃO DA PMU

O uso dos contadores de desempenho em si implica na configuração de uma série de regiões criticas do sistema. A habilitação deste recurso como um todo deve ser realizada nos registradores que controlam a operação de cada núcleo do processador. Da mesma forma, a configuração de suas interrupções requer que a tabela de vetores de interrupção do sistema seja manipulada. Portanto, no EPOS optou-se por realizar o processo de inicialização da PMU durante o *boot* do próprio sistema operacional. Esta medida reduz o risco de modificar regiões importantes ao longo da operação do sistema.

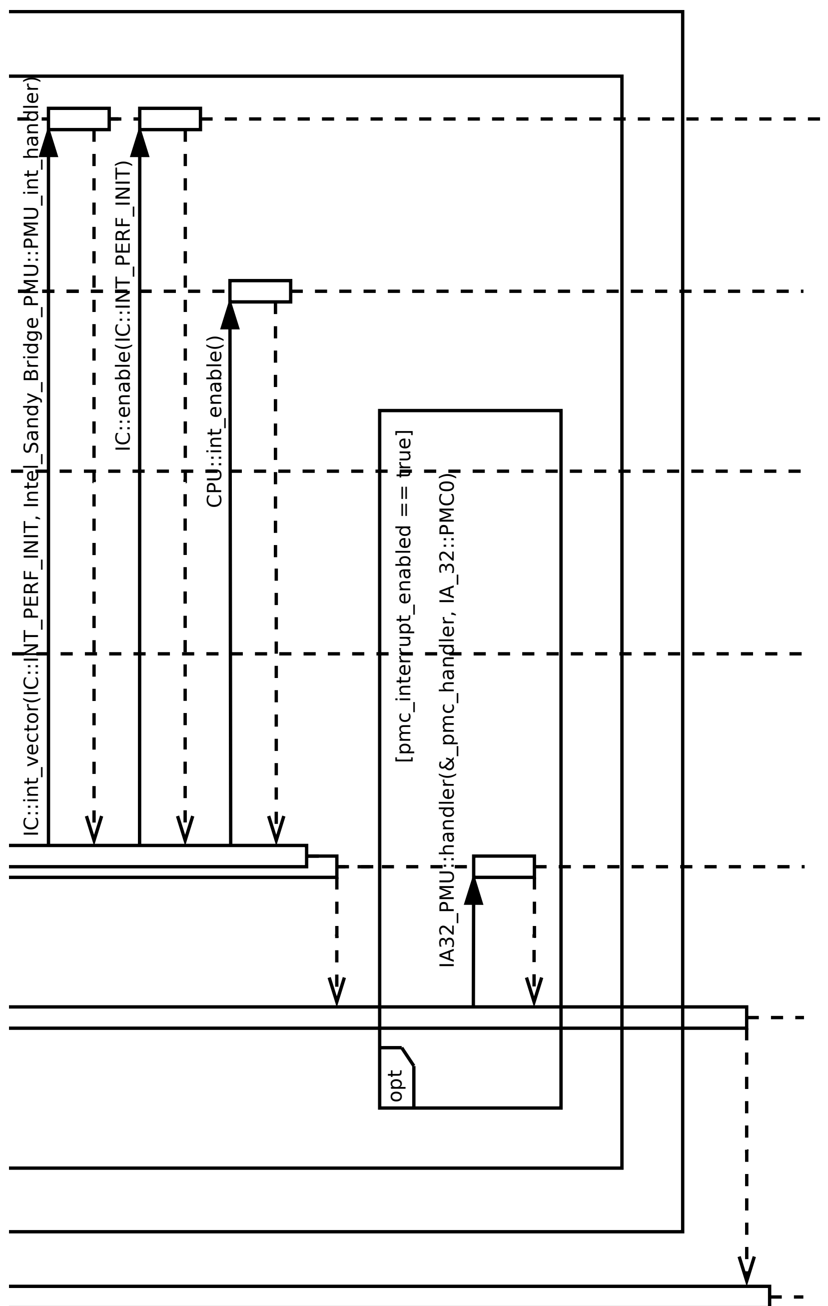
O processo de *boot* do EPOS é executado principalmente pela CPU 0, enquanto as demais CPUs do sistema são responsáveis por configurar apenas recursos exclusivos de seus núcleos. Por consequência, a função de inicialização das PMUs são chamadas em diferentes estágios da inicialização do sistema para estes núcleos: enquanto o núcleo 0 é chamado pela função que inicializa o mecanismo de *threads*, os demais núcleos são chamados pelo próprio componente de inicialização do EPOS (*Init_System*). Portanto, as Figuras 19, 20 e 21 representam a sequência de inicialização da PMU nos diferentes núcleos do processador.

Figura 19 – Diagrama de sequência para o processo de inicialização da PMU no núcleo 0 da CPU (continua).



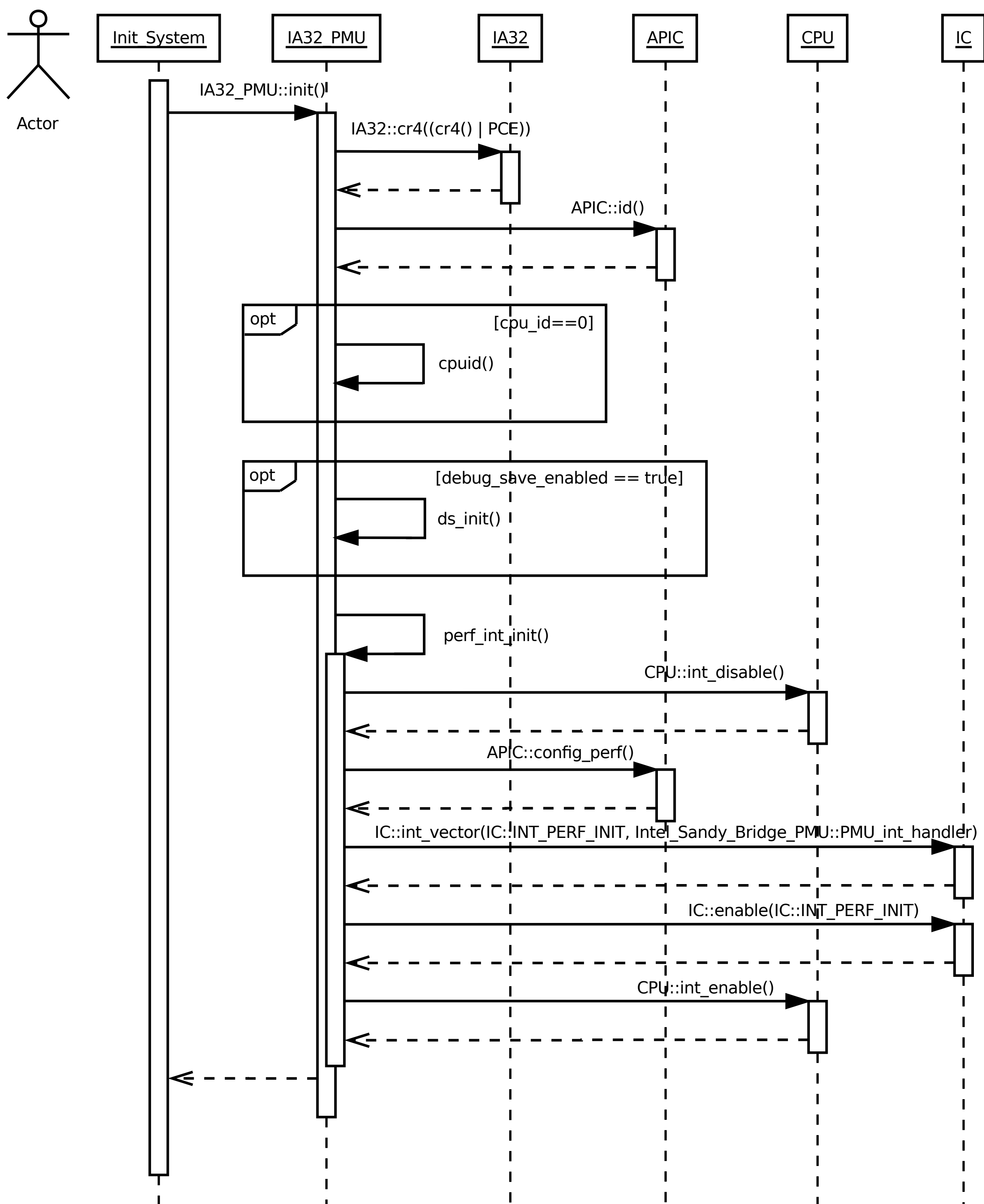
Fonte: Autor, 2017.

Figura 20 – Diagrama de sequência para o processo de inicialização da PMU no núcleo 0 da CPU (conclusão).



Fonte: Autor, 2017.

Figura 21 – Diagrama de sequência para o processo de inicialização da PMU nos núcleos da CPU, exceto no núcleo 0.



Fonte: Autor, 2017.

De acordo com os diagramas de sequência apresentados, o processo de inicialização da PMU pode ser resumido através dos seguintes passos:

- Primeiro, o recurso de monitoramento de desempenho é habilitado no registrador de controle de seu núcleo (*bit* 8 do registrador CR4);
- Em seguida a rotina de inicialização identifica qual núcleo está em execução;
- Caso este seja o núcleo 0, algumas leituras são realizadas sobre as características de *hardware* do processador para definir uma série de atributos da classe IA32_PMU que descrevem os recursos disponíveis no modelo de processador utilizado;
- O próximo passo verifica se o recurso *DS Area* foi habilitado por *Traits* e se o processador utilizado possui suporte ao recurso, para somente então realizar a inicialização de sua área de gerenciamento, conforme descrito na Seção 2.3.1;
- Por fim, a interrupção da PMU é inicializada, definindo sua entrada na tabela de vetores e atribuindo ao sistema o endereço da função de tratamento de interrupções deste componente.

Conforme mencionado ao longo deste documento, grande parte dos componentes implementados no EPOS são construídos em tempo de compilação conforme as definições realizadas na classe *Traits*. Devido à implementação destes novos componentes sobre os mediadores de *hardware* da PMU, a seção de código apresentada na Figura 22 foi adicionada ao arquivo de *Traits* referente a arquitetura Intel no EPOS. Suas definições indicam ao sistema quais dos recursos da PMU serão utilizados pela aplicação ou até mesmo se a PMU em si deverá ser inicializada pelo sistema.

Figura 22 – Definições em *Traits* da classe IA32_PMU.

```
1  template <> struct Traits<IA32_PMU>: public Traits<void>
2  {
3      static const bool enabled = true;
4
5      static const bool debug_save_enabled = false;
6      static const bool pmc_interrupt_enabled = true;
7  };
```

Fonte: Autor, 2017.

No Apêndice A encontram-se disponíveis os trechos de código responsáveis pela chamada e execução do processo de inicialização da PMU, desenvolvidos de acordo com os passos descritos nesta seção. Estes estão representados através das Figuras 34, 35, 36, 37, 38, 39 e 40.

4.2 ROTINA GLOBAL DE SERVIÇO PARA INTERRUPÇÕES DA PMU

Quando configurada para gerar interrupções, a PMU irá desviar o fluxo normal de execução do programa, chamando a função de tratamento de interrupções definida durante a inicialização do componente. Esta rotina deve verificar qual o evento responsável por sua chamada, tratar o processo de interrupção e realizar as ações definidas no sistema para o evento correspondente. Tal processo é detalhado ao longo desta seção e sua representação na forma de diagramas de sequência é apresentada através das Figuras 23 e 24.

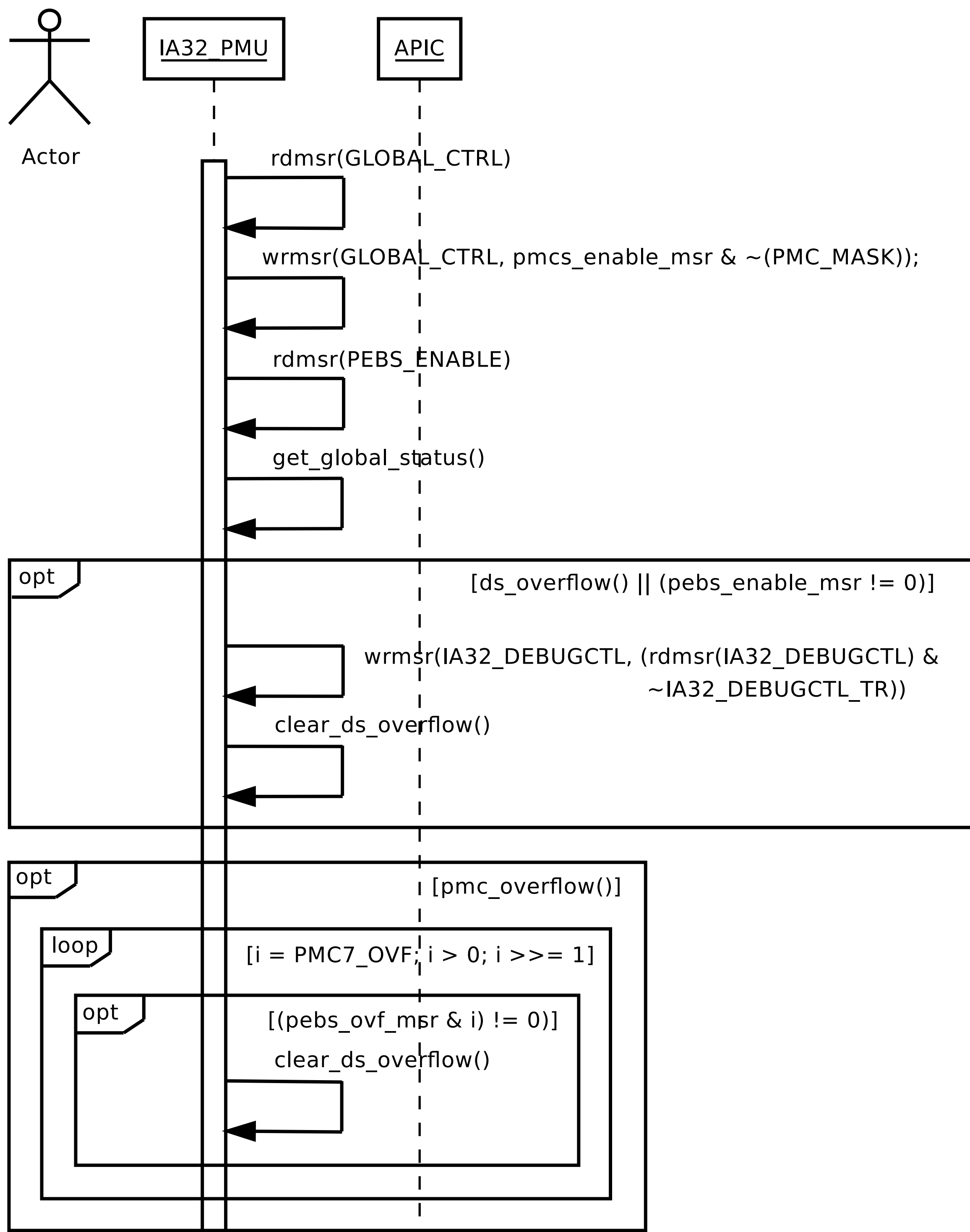
O tratamento de interrupções da PMU inicia por desabilitar a operação de seus contadores. Em seguida, verifica-se a fonte de chamada da interrupção entre o recurso *DS Area* e os PMCs. Quando a interrupção é chamada a partir de um evento de PEBS ou pela escrita em DS Area, a rotina de serviço reinicia os índices dos *buffers* BTS e PEBS, além de limpar o status de interrupção de *DS Area*.

Após isto, também é necessário verificar se a interrupção foi gerada a partir de algum contador de desempenho. Mesmo quando uma interrupção é gerada pelo *overflow* em PEBS, se faz necessário gerenciar seu contador como ocorre com qualquer outro PMC, isto torna necessário que esta verificação sempre seja realizada pelo tratador. Sendo assim, por padrão o EPOS desabilita todo contador com *status* de *overflow* e reinicia seu estado. Isto torna necessário que tais contadores sejam reconfigurados antes de retomarem o monitoramento de eventos.

Feito isto, a rotina de serviço reabilita os contadores que não sofreram *overflow*, reabilita as interrupções do sistema e somente então temos a chamada dos tratadores específicos de cada PMC. Esta sequência de ações foi estabelecida com o objetivo de reduzir ao máximo a interferência do processo de tratamento de interrupções sobre os demais periféricos do sistema. Embora será necessário que o processador continue com o processamento das ações de cada fonte de interrupção, isto não impedirá o monitoramento de eventuais eventos adicionais ou até mesmo no processamento de interrupções originadas de outros periféricos.

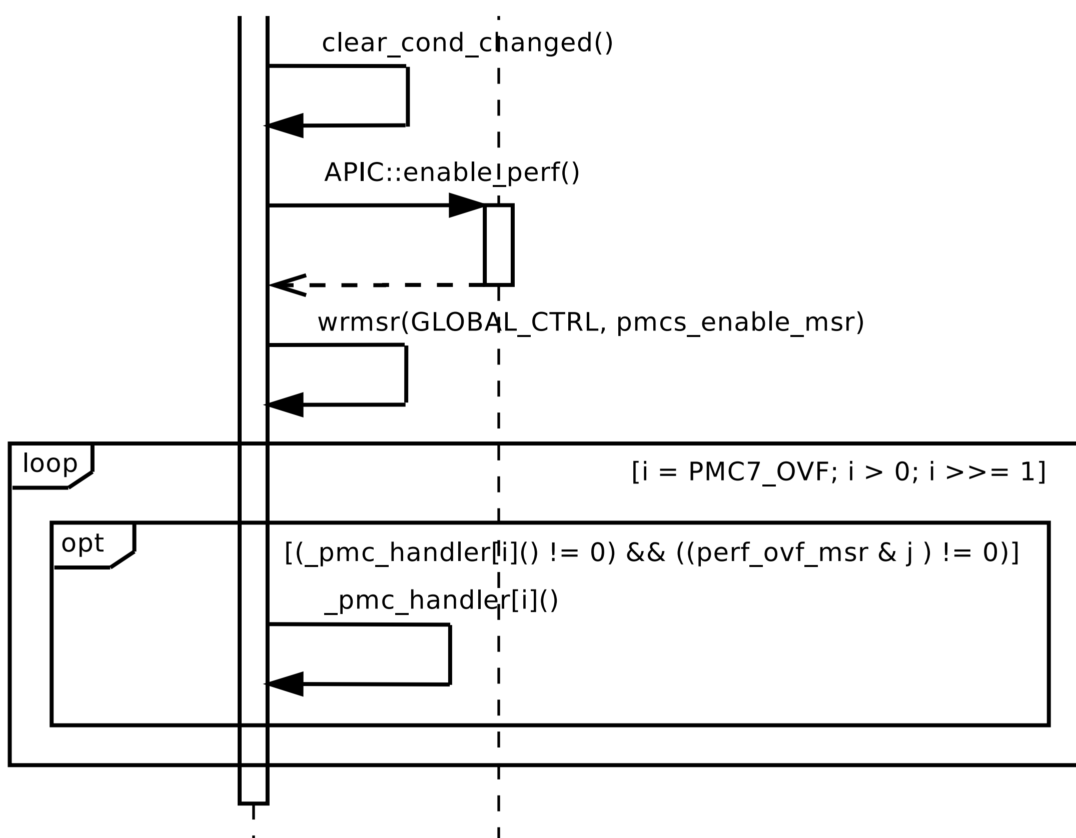
No Apêndice B se encontra disponível através das Figuras 41 e 42 o código desenvolvido para atuar como rotina global de serviço para as interrupções da PMU, desenvolvidos de acordo com os passos descritos nesta seção.

Figura 23 – Diagrama de sequência para o tratamento global de interrupções da PMU (continua).



Fonte: Autor, 2017.

Figura 24 – Diagrama de sequência para o tratamento global de interrupções da PMU (conclusão).



Fonte: Autor, 2017.

4.3 AÇÃO DA PMU NO ESCALONAMENTO DE TAREFAS

De acordo com Yun et al. (2013), existe uma necessidade cada vez maior de soluções de gerenciamento de banda de memória e memória cache que garantam determinada qualidade de serviço. Conforme demonstrado por Gracioli (2014), o uso de registradores de desempenho é capaz de oferecer os recursos necessários não apenas para registrar o perfil de acesso em memória da uma aplicação, mas também para servir de referência na tomada de decisões no processo de escalonamento de tarefas.

O autor comenta como é possível fazer uso de dados da PMU para, por exemplo, deslocar uma *Thread* para um núcleo diferente quando o componente de monitoramento de desempenho identifica um número excessivo de falhas de acesso na cache devido ao compartilhamento de dados entre núcleos. Estudos semelhastes foram desenvolvidos em: (INAM et al., 2014; INAM; SJODIN, 2014; TOBUSCHAT et al., 2013; YUN et al., 2015; YUN et al., 2013; MANCUSO et al., 2013). Sendo assim, neste trabalho é proposto um mecanismo capaz de atuar de forma dinâmica no processo de escalonamento de tarefas.

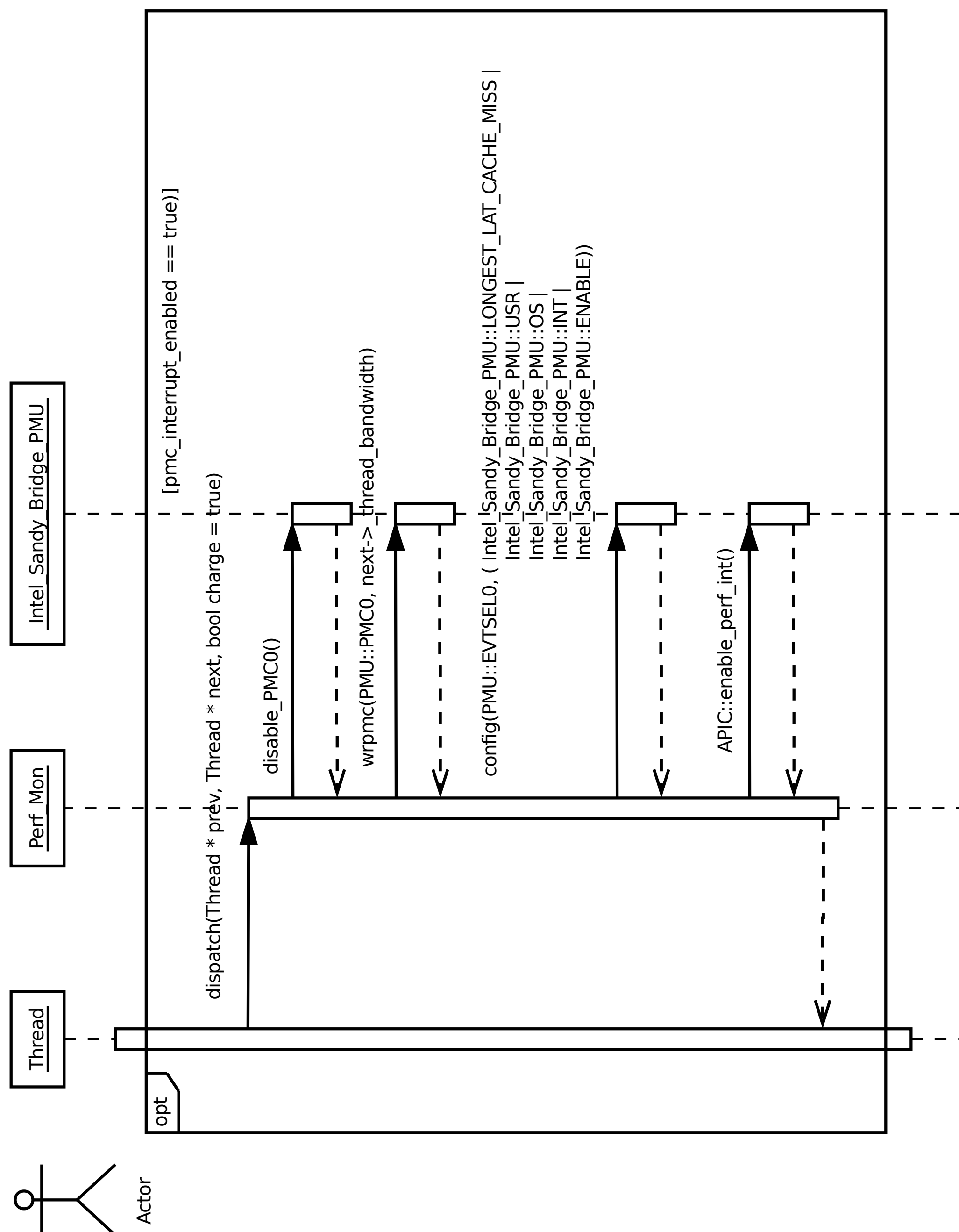
Na arquitetura do EPOS existe uma função utilizada especificamente na execução das operações de escalonamento: "Thread::dispatch(Thread * prev, Thread * next)". Esta função recebe como parâmetros de entrada ponteiros para tarefa em execução e para a próxima tarefa a ser executada, sendo responsável por iniciar sua troca de contexto.

Por esta razão, este método foi escolhido para realizar a configuração do evento de monitoramento das tarefas em execução, executados pela função "Perf_Mon::pmu_interrupt_config(long long event_number)". Desta forma, ao início de sua execução, toda *thread* configura o monitoramento do evento desejado. De acordo com o número inicial de contagem definido ao longo desta configuração, o momento em que a interrupção da PMU ocorre pode ser ajustado. Este processo torna possível que, por exemplo, o número de erros de acesso na cache seja monitorado (evento LLC_MISSES) e após a ocorrência de um número definido de eventos uma interrupção seja gerada, chamando assim o tratador do mecanismo de ação da PMU.

Este processo de configuração segue a sequência de ações descrita nas seções 2.3.1 e 3.3. Sua representação na forma de um diagrama de sequências pode ser observada através da Figura 25.

A definição do valor inicial de contagem, que neste caso pode ser visto como o número de eventos que ocasiona a ação do mecanismo, é realizada através do método "PMU::wrpmc(Reg32 counter, long long value)", enquanto a configuração do evento a ser monitorado é desenvolvida através de "PMU::config(Reg32 event_select_register, Reg32 config_flags)".

Figura 25 – Diagrama de sequência para o processo configuração de eventos e monitoramento de *Threads*.



Fonte: Autor, 2017.

Porém, nota-se que existem algumas diferenças nos métodos utilizados após a implementação das melhorias propostas para a PMU do EPOS. Por exemplo, mais um passo foi dado para melhorar a abstração de *hardware* nesta nova versão do sistema, através da implementação de métodos como "Intel_Sandy_Bridge_PMU::enable_PMC0()", responsável por habilitar operação do PMC0, e "Intel_Sandy_Bridge_PMU::disable_PMC0()", que, por sua vez, desabilita a operação do PMC0.

Também foi desenvolvido um método específico para o gerenciamento de interrupções nos componentes de monitoramento de desempenho: enquanto “APIC::enable_perf_int()” é capaz de habilitar as interrupções do sistema, “APIC::disable_perf_int()” realiza sua desabilitação. No entanto, estas operações somente são realizadas se o recurso de interrupções da PMU estiver ativo, caso contrário a execução destes comandos não se faz necessária.

Por fim, temos o último método responsável por compor este mecanismo de ação no escalonamento de tarefas: “Thread::pmc_interrupt_handler(unsigned int irq)”. Esta função nada mais é que um tratador de interrupções específico para desempenhar a operação de escalonamento desejada. A atribuição desta função à rotina de tratamento global é realizada durante a inicialização da PMU, conforme representado na Figura 20, enquanto sua chamada é desenvolvida pelo tratador global de interrupções da PMU, conforme representado na Figura 24.

No Apêndice C se encontra disponível através da Figura 25 o código desenvolvido para a configuração do evento responsável por monitorar a execução das *Threads* da aplicação e atuar no processo de escalonamento, de acordo com os passos descritos nesta seção.

4.4 CONSIDERAÇÕES PARCIAIS

Conforme descrito nas seções 2.3.1 e 3.3, embora o EPOS possibilite o uso da PMU de processadores Intel, a versão do RTOS anterior às melhorias propostas neste capítulo não é capaz de utilizar todos os recursos relacionados a suas unidades de monitoramento.

Sendo assim, através deste capítulo foram propostas uma série de modificações para o EPOS, buscando melhorar seu suporte a família de PMUs dos processadores Intel. A proposta deste trabalho apresentou uma revisão sobre a família de mediadores de *hardware* de sua PMU (IA32_PMU) e o componente de monitoramento de desempenho (Perf_Mon). Através destas alterações foram implementadas rotinas que possibilitam o uso dos recursos de interrupção e *Debug Store* oferecidos por processadores Intel modernos. Além disto, um mecanismo capaz de atuar no processo de escalonamento de tarefas foi proposto.

Com base neste mecanismo, aplicações desenvolvidas no EPOS poderão contar com um recurso capaz de monitorar determinado evento durante sua execução e desempenhar um conjunto de ações correspondente através do tratador de sua interrupção. Pode-se, por exemplo, limitar a taxa de erros de acesso em cache ao monitorar o evento LLC_MISSES, ou até mesmo reduzir sua taxa de contenções devido ao compartilhamento de dados entre núcleos ao monitorar o evento MEM_LOAD_UOPS_LLC_HIT_RETIRED_XSNP_HITM e utilizar de seus tratadores para suspender tarefas em conflito.

De modo a avaliar sua operação, o capítulo a seguir desenvolve uma série de experimentos sobre as modificações propostas ao longo deste trabalho, abordando o mecanismo de ação da PMU.

5 AVALIAÇÃO DO MECANISMO DE MONITORAMENTO PROPOSTO

No Capítulo 4 foram apresentadas as melhorias ao suporte da PMU para o sistema operacional EPOS propostas através deste trabalho. Sua utilização permite que desenvolvedores do EPOS obtenham mais informações sobre o perfil de execução de aplicações, podendo até mesmo utilizá-las na tomada de decisões durante o escalonamento de tarefas. Com isto, neste capítulo o desempenho do mecanismo de monitoramento proposto ao longo deste trabalho é avaliado utilizando o escalonador *Rate Monotonic* Particionado (P-RM), disponível no EPOS.

Este capítulo foi dividido da seguinte forma: num primeiro momento a aplicação utilizada para avaliação das implementações propostas é apresentada na Seção 5.1, descrevendo as considerações realizadas e seus parâmetros de execução; para possibilitar a execução destes experimentos um processo automatizado de testes foi desenvolvido em paralelo a este trabalho, apresentado na Seção 5.2; e, por fim, a avaliação propriamente dita das melhorias propostas é realizada na Seção 5.3.

5.1 DESCRIÇÃO DO EXPERIMENTO

Com o objetivo de avaliar as extensões propostas para os mediadores de *hardware* da PMU, uma aplicação sintética e intensiva em memória foi desenvolvida para a execução de três tipos de tarefas: tarefas de tempo real críticas, tempo real não-críticas e tarefas de melhor esforço (tarefas sem requisitos de tempo real). Todas estas tarefas realizam um volume intenso de operações de leitura e escrita em memória, principalmente as tarefas de melhor esforço, cujo principal objetivo é invalidar o maior número de linhas da cache possível, interferindo na execução das tarefas de tempo real.

Cada tarefa executa uma função responsável por alocar memória para seu vetor de trabalho, executando um laço que apenas contém operações de leitura e escrita sobre o mesmo, acessando posições aleatórias do vetor a cada operação. A título de exemplo, a Figura 26 apresenta a aplicação utilizada na execução de tarefas SRT.

É possível observar através da Figura 26 que o número de repetições do laço que realiza as operações de leitura e escrita no vetor de trabalho é o parâmetro responsável por determinar o WCET de cada tarefa. Isto indica uma importante característica desta aplicação, que se dá através de sua capacidade de permitir que um usuário defina os parâmetros das tarefas a serem executadas. Cada experimento pode executar um diferente número de tarefas, com tipos, grupos de execução (partições), períodos de execução, WCETs e até mesmo núcleos de processamento; parâmetros definidos pelo desenvolvedor do experimento durante o processo de compilação.

Figura 26 – Seção de código utilizada pelas tarefas de tempo real não críticas.

```
1  #define HRT_TASKS 8
2  #define WRITE_RATIO 4
3  #define ARRAY_SIZE KB_256
4  #define MEMORY_ACCESS 16384
5
6  int srt_func(int group, int id, int repetitions)
7  {
8      // Declaração de variáveis locais
9      int* array;
10     int sum = 0;
11     Chronometer c;
12     Pseudo_Random* rand;
13
14     // Criação de um objeto para geração de valores aleatórios
15     rand = new ((alloc_priority)(HRT_TASKS+group+2)) Pseudo_Random();
16
17     // inicialização do objeto rand
18     rand->seed(clock.now() + id);
19
20     // Criação do vetor de trabalho
21     array = new ((alloc_priority)(HRT_TASKS+group+2)) int[ARRAY_SIZE];
22
23     // Início do laço de execução da thread
24     for (int i = 0; i < ITERATIONS; i++)
25     {
26         ...
27         // Laço com operações de leitura e escrita
28         for (int j = 0; j < repetitions; j++)
29         {
30             for (int k = 0; k < MEMORY_ACCESS; k++)
31             {
32                 int pos = rand->random() % (ARRAY_SIZE - 1);
33                 sum += array[pos];
34                 if ((k % WRITE_RATIO) == 0)
35                     array[pos] = k + j;
36             }
37         }
38         ...
39     }
40
41     // Libera a memória alocada
42     delete array;
43     delete rand;
44
45     // Retorna um valor aleatório
46     return sum;
47 }
```

Fonte: Autor, 2017.

Os parâmetros que compõem os conjuntos de tarefas utilizados durante os experimentos deste projeto foram obtidos a partir da *software* Schedcat, ferramenta desenvolvida para testes e análise em experimentos com escalonadores de tarefas (BRANDENBURG, 2013). O *script* criado no Schedcat foi configurado para gerar três conjuntos de tarefas periódicas de tempo real (SRT e HRT) com parâmetros aleatórios, semelhante a abordagem utilizada por Ward et al. (2013) e Gracioli (2014).

O algoritmo de geração determina a quantidade de tarefas que compõem uma aplicação à medida com que a capacidade de processamento teórica do sistema é atingida (69% devido ao limite imposto pelo RM). Além destas tarefas, quatro tarefas de melhor esforço são adicionadas

em cada aplicação. Seu objetivo é simular a execução de funções de utilidade adicionais do sistema, sem quaisquer requisitos de tempo real ou prioridade de execução.

Cada conjunto de tarefas gerado foi avaliado em termos do algoritmo de empacotamento utilizado para sua distribuição entre núcleos do processador. Sendo assim, todo grupo de tarefas utilizado durante esta etapa de experimentos se apresenta através de duas combinações distintas, de acordo com os algoritmos selecionados. São eles: *Color-Aware Task Partitioning with Group Splitting* (CAP-GS) e *Best Fit Decreasing* (BFD).

Por fim, se faz necessário determinar as condições relativas ao particionamento de memória das tarefas executadas em cada aplicação. Desta forma, foi convencionado que tarefas críticas terão cores exclusivas, enquanto tarefas não críticas e de melhor esforço serão divididas entre quatro grupos de memória compartilhados.

Com base nestas definições, critérios distintos foram adotados para a geração de tarefas HRT e SRT. Os períodos de cada tarefa HRT (com valores expressos em ms) foram selecionados uniformemente de 25; 50; 75 com utilização individual selecionada uniformemente entre o intervalo de [0,2; 0,35] até a utilização do sistema alcançar um valor entre 2,1 e 2,5. Tarefas SRT, por sua vez, foram divididas entre 4 grupos e geradas utilizando períodos de execução selecionados uniformemente de 100; 150; 200; 500 com utilização individual selecionada uniformemente entre o intervalo de [0,1; 0,3] até a utilização de cada grupo atingir um valor entre 0,6 e 0,7. O WCET de uma tarefa é definido de acordo com o período e utilização gerados.

5.2 AMBIENTE DE TESTES

A execução de tais experimentos se deu em *hardware* real: um *desktop* com um processador Intel i7-2600. Suas principais especificações estão dispostas na Tabela 1 a seguir:

Tabela 1 – Especificações técnicas do processador Intel i7-2600

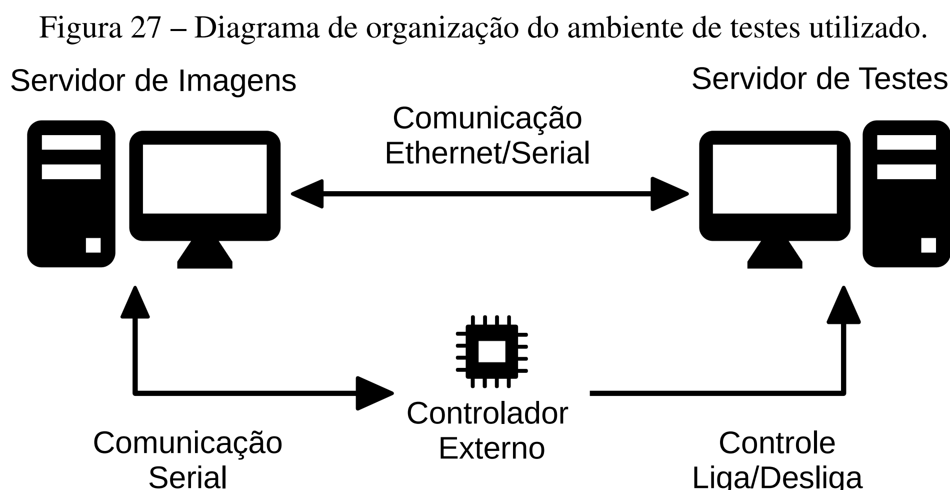
Velocidade de Processamento	3.4 GHz
Núcleos	4
Hyper-threading	2 por núcleo (8 núcleos lógicos)
Cache L1	4 x 64 KB
Cache L2	4 x 256 KB
Cache L3 (compartilhada)	8 MB

Fonte: Adaptado de Gracioli (2014).

Devido ao volume de testes e ao tempo necessário para executá-los, se fez necessário elaborar um processo automatizado de testes ao longo deste trabalho. O desenvolvimento desta tarefa foi realizado em paralelo a este trabalho e contou com a ajuda dos acadêmicos Breno Castro e Samuel Possamai. A Figura 27 representa o ambiente de testes utilizado: o servidor de testes propriamente dito, ou seja, o *hardware* responsável pela execução dos experimentos; um computador responsável por fornecer imagens do EPOS com a aplicação desejada; e um

dispositivo controlador externo responsável por ligar e desligar o servidor de testes, gerenciando assim o fluxo de testes a serem executados.

Figura 27 – Diagrama de organização do ambiente de testes utilizado.



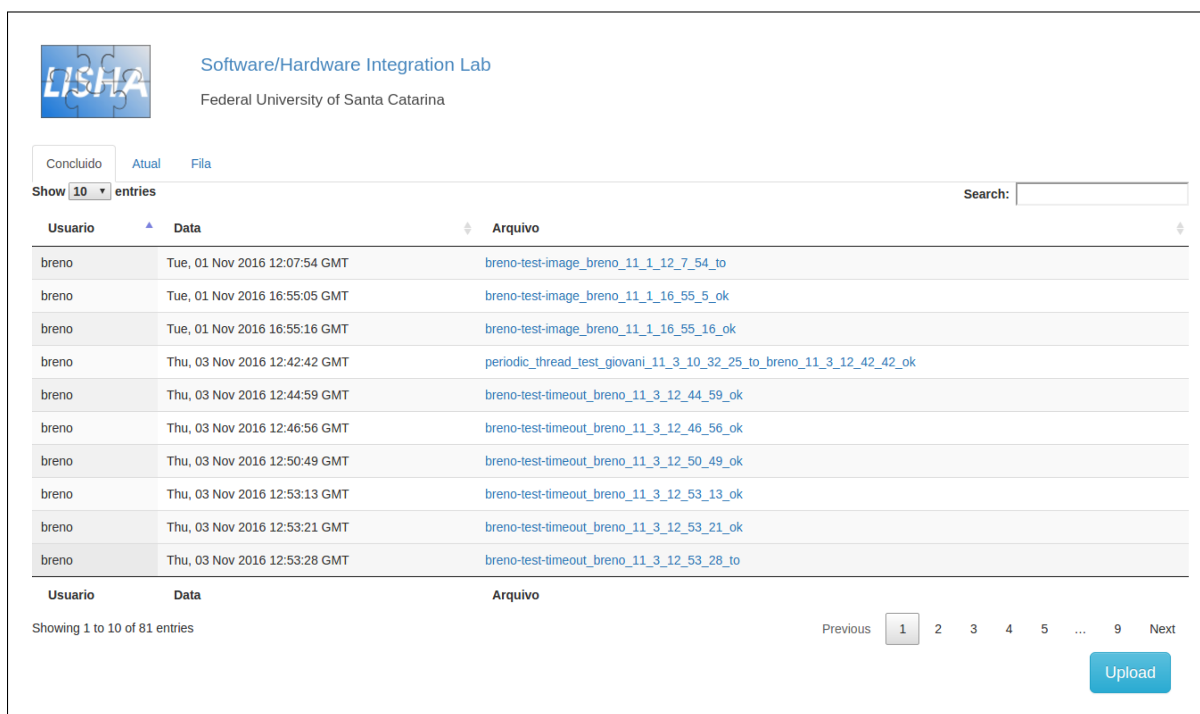
Fonte: Autor, 2017.

De uma forma geral, cada processo de testes segue a seguinte sequência de operações:

- Inicialmente o servidor de imagens seleciona a aplicação a ser executada e inicia o monitoramento de um ramal de comunicação serial com a máquina de testes. Através deste ramal são recebidos os resultados de execução dos experimentos;
- Em seguida o dispositivo controlador inicia a máquina de testes, a qual foi configurada para realizar seu processo de *boot* através da rede e requisitar ao servidor de imagens a aplicação a ser executada;
- Após sua inicialização, o servidor de testes executa a aplicação recebida, retornando através de sua porta serial seus resultados de execução;
- Ao final da execução de cada experimento o controlador de testes volta a desligar a máquina de testes e aguarda por novas instruções por parte do servidor de imagens;
- Por fim, o servidor de imagens armazena os resultados de execução recebidos e os disponibiliza ao usuário.

Este processo se repete enquanto houverem imagens disponíveis para execução no sistema do servidor de imagens. Adicionalmente, para facilitar todo este processo de envio de imagens e leitura de resultados, um primeiro protótipo de aplicação *WEB* foi elaborado. Sua interface é apresentada na Figura 28. Esta aplicação permite que seus usuários acessem o sistema de testes de qualquer lugar, permitindo o envio de imagens e leitura de resultados pela *internet*.

Figura 28 – Interface da aplicação desenvolvida para o gerenciamento do ambiente de testes.



Fonte: Autor, 2017.

5.3 AVALIAÇÃO DO MECANISMO DE AÇÃO DA PMU

De forma a avaliar o mecanismo proposto para tornar a PMU capaz de atuar sobre o processo de escalonamento de tarefas do sistema, quatro cenários foram considerados. São eles:

1. O conjunto de tarefas selecionado tem suas tarefas particionadas através do algoritmo CAP-GS e o recurso de ação da PMU desabilitado;
2. O conjunto de tarefas selecionado tem suas tarefas particionadas através do algoritmo CAP-GS e o recurso de ação da PMU habilitado;
3. O conjunto de tarefas selecionado tem suas tarefas particionadas através do algoritmo BFD e o recurso de ação da PMU desabilitado;
4. O conjunto de tarefas selecionado tem suas tarefas particionadas através do algoritmo BFD e o recurso de ação da PMU habilitado;

A execução de seus experimentos explora estes diferentes cenários através dos três conjuntos de tarefas apresentados no Apêndice D, sendo que estes foram gerados de acordo com as definições apresentadas na Seção 5.1. O mecanismo de ação da PMU, por sua vez, foi utilizado conforme apresentado no Capítulo 4. A ação a ser desenvolvida pelo tratador de interrupções

deste mecanismo, no entanto, foi definida como a chamada do método “(self())->suspend()”. Este método faz parte da classe *Thread* e é responsável por suspender a *thread* em execução por um tempo indeterminado.

Com relação ao evento monitorado pela PMU, manteve-se o mesmo que foi representado durante a apresentação do algoritmo: LLC_MISSES. Este evento é responsável por monitorar o números de erros de acesso no último nível de cache do processador. Em outras palavras, este evento é capaz de monitorar o número de acessos ao barramento de memória RAM.

O objetivo desta bateria de experimentos é, principalmente, observar a capacidade do mecanismo em gerar interrupções através da PMU e utilizar deste recurso para realizar uma operação de escalonamento de tarefas. Neste caso, ao atingir o limite de acessos determinado pela aplicação, a tarefa monitorada será retirada da fila de execução do escalonador e será impedida de executar novamente no experimento em questão.

Optou-se ainda por utilizar este mecanismo somente sobre as tarefas de melhor esforço, uma vez que estas são a principal fonte de interferência observada ao longo destes experimentos. Desta forma, após a suspensão das tarefas de melhor esforço é esperado que o conjunto de tarefas passe a ser executado conforme determinado por seus parâmetros de projeto, cumprindo com seus requisitos de período de WCET. Considerando que o objetivo de tal experimento é de apenas verificar capacidade de atuação do mecanismo de interrupção e reduzir ao máximo a interferência destas tarefas, seu parâmetro de limite de acesso a memória foi definido arbitrariamente como 9. Quando determinada tarefa de melhor esforço realizar 9 erros de acesso na cache, sua execução será suspensa.

A cada experimento foram executados um número de 200 iterações, ou seja, cada tarefa será executada por 200 períodos. Considerando que o período máximo de um conjunto de tarefas é de 500 ms, o tempo de execução esperado por aplicação é de 100 segundos. Por fim, foram executadas 100 repetições de cada experimento para tornar possível uma análise estatística dos resultados obtidos.

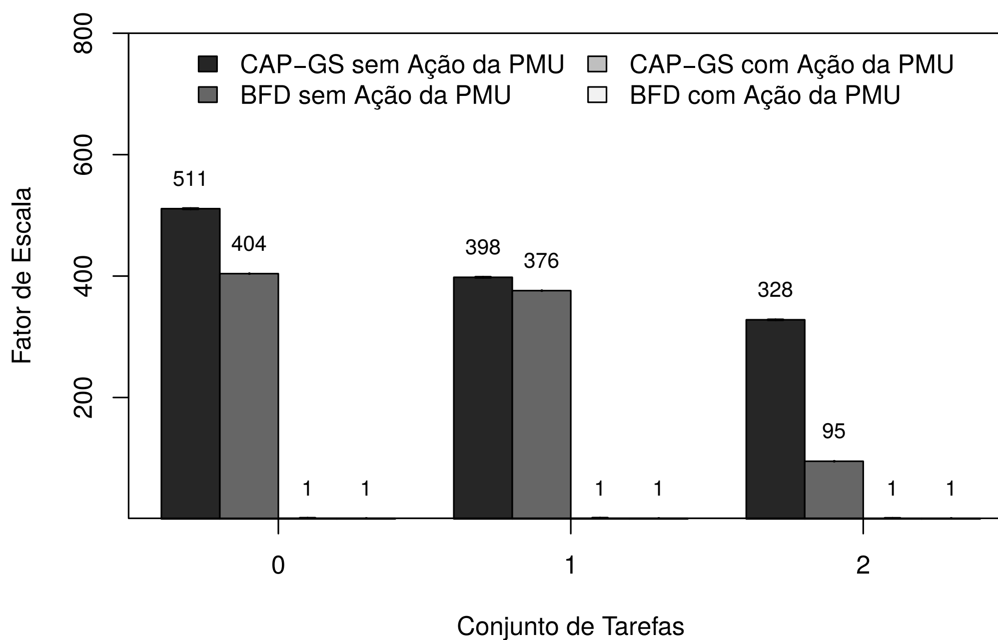
Dito isto, as seções a seguir apresentam os resultados obtidos através destes experimentos, avaliando o desempenho do algoritmo em termos da variação entre o WCET obtido e o projetado, seu percentual de perda de *deadlines*, seu tempo de atraso de deadlines e do tempo de execução da aplicação. No Apêndice E encontra-se ainda disponível uma relação completa dos resultados numéricos obtidos ao longo destes experimentos.

5.3.1 Fator de Escala do WCET

Ao fim da execução dos experimentos, o maior tempo de execução de cada cenário foi identificado, os quais podem ser observados na forma de fatores de escala, isto é, a razão média entre o WCET projetado e o WCET obtido por suas tarefas, através das Figuras 29 e 30. Enquanto a Figura 29 apresenta o fator de WCET médio entre as tarefas de cada conjunto de

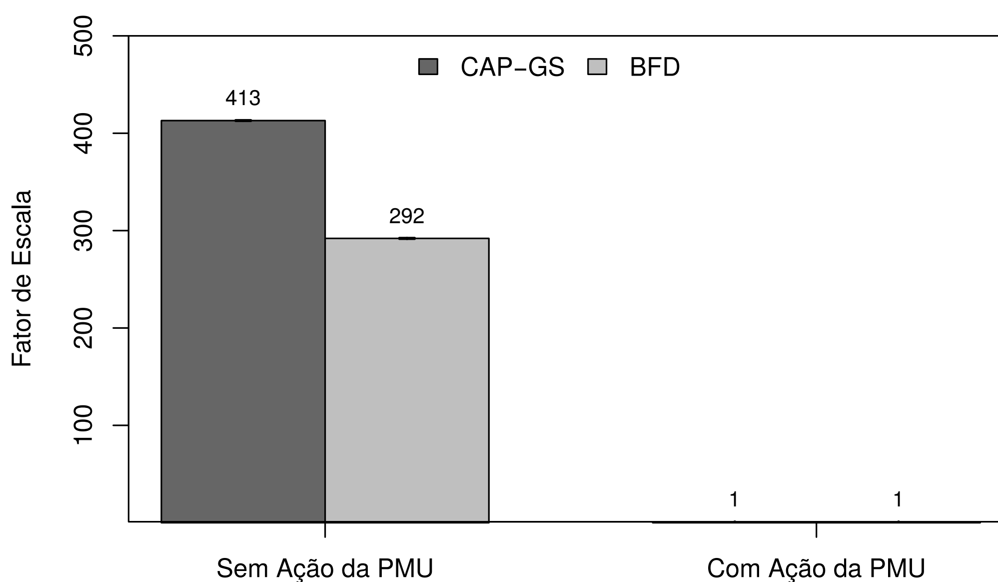
tarefas executado, a Figura 30 traz a média entre os WCETs dos experimentos como um todo. Em ambos os casos os resultados foram classificados entre os quatro cenários previstos para a aplicação.

Figura 29 – Fator de escala de WCET por conjunto de tarefas.



Fonte: Autor, 2017.

Figura 30 – Fator médio de escala de WCET.



Fonte: Autor, 2017.

Através destes gráficos pode-se constatar a operação do mecanismo proposto, bem como a severidade da interferência causada pelas tarefas de melhor esforço. Para os cenários em que a

ação da PMU foi habilitada, é possível observar que o sistema foi capaz de limitar o acesso em memória das funções de melhor esforço e impedir sua execução.

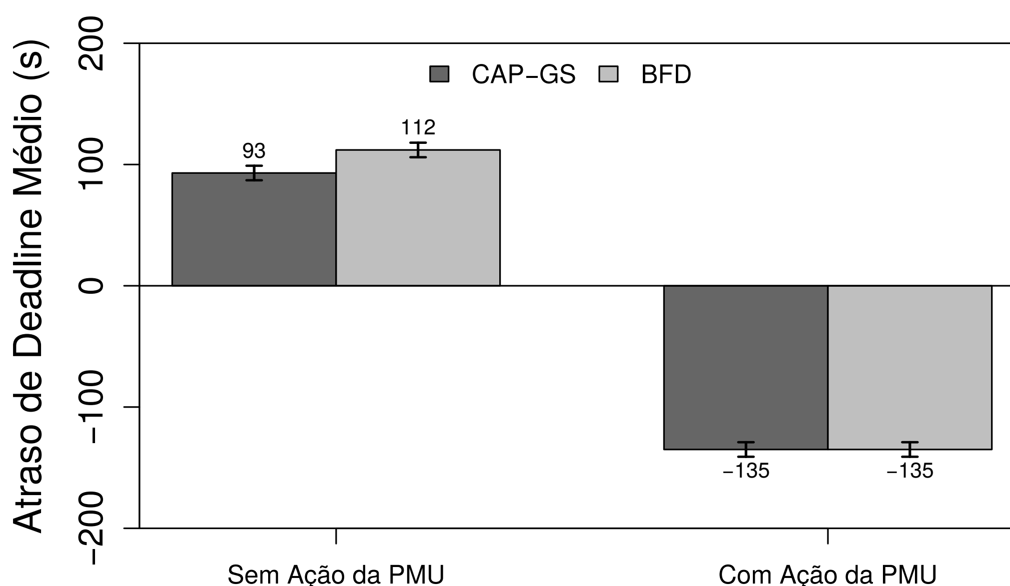
Isto pode ser afirmado porque nestes cenários a razão média de seus WCETs, sejam eles divididos por tarefas ou combinados como um todo, é aproximadamente igual a 1. Por outro lado, os cenários em que as funções poluidoras (de melhor esforço) foram executadas livremente apresentaram um grande nível de interferência entre suas tarefas. A menor razão entre os WCETs observada nestas aplicações indica que seu pior tempo de execução médio foi, aproximadamente, 95 vezes maior que o esperado. Além disso, pode-se observar ainda que o nível de interferência causado pelas funções de utilidade causou um impacto maior sobre os conjuntos de tarefas distribuídos através do algoritmo CAP-GS.

5.3.2 Atraso de Deadline

O atraso de deadline é outro parâmetro importante na avaliação de aplicações de tempo real. O atraso de uma tarefa é dado como a diferença entre o seu tempo de conclusão e prazo limite. Se uma tarefa de tempo real perde seu *deadline*, seu atraso é positivo. Já um atraso negativo significa que a tarefa concluiu sua execução antes do prazo e este é o comportamento esperado para tarefas com restrições de tempo real, especialmente tarefas HRT.

Dito isto, a Figura 31 nos traz o atraso de *deadline* médio dos experimentos realizados. Os valores apresentados neste gráfico representam a média dos tempos de execução de todas as tarefas executadas durante o conjunto de testes descrito neste capítulo. Estes valores são classificados de acordo com os cenários de execução propostos.

Figura 31 – Atraso de deadline médio.



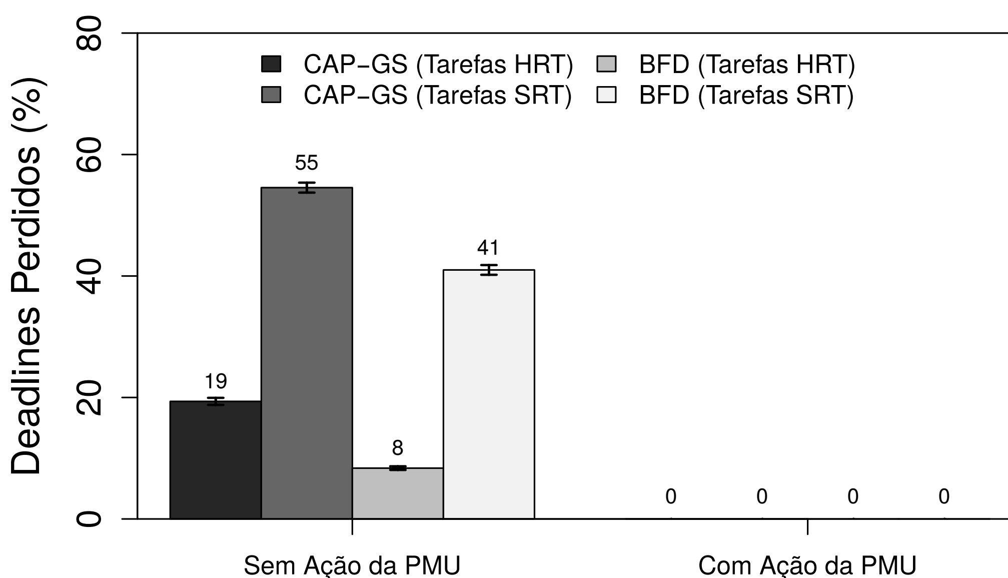
Fonte: Autor, 2017.

Como resultado é possível observar novamente o efeito causado pela ação do mecanismo de interrupções da PMU sobre a execução dos conjuntos de tarefas. Quando executadas sem a intervenção deste mecanismo, obteve-se um atraso médio de 93 s sobre a execução dos conjuntos de tarefas distribuídos pelo algoritmo CAP-GS e 112 s quando BFD foi utilizado. Através da ativação do recurso proposto, no entanto, ambos métodos de empacotamento obtiveram um atraso negativo de 135 s. Além de reduzir o atraso causado pelas funções de melhor esforço, tal resultado indica que, em média, o mecanismo de interrupção possibilitou com que as tarefas de tempo real concluíssem suas execuções antes de seus *deadlines*, auxiliando no cumprimento de seus requisitos de tempo real.

5.3.3 Perda de Deadline

Observou-se também o percentual de perda de *deadlines* ao longo da execução dos experimentos. Sendo assim, temos representado através da Figura 32 o percentual de perdas de *deadline* apresentado em cada um dos cenários previstos.

Figura 32 – Percentual de deadlines perdidos por tipo de tarefas.



Fonte: Autor, 2017.

Nos experimentos em que o mecanismo de interrupções foi habilitado não houve qualquer perda no cumprimento de *deadlines* ao longo de sua execução. Por outro lado, nos experimentos em que as funções de melhor esforço foram executadas normalmente um percentual de perdas de *deadline* de até 19% foi obtido sobre a execução de tarefas HRT e 55% sobre tarefas SRT.

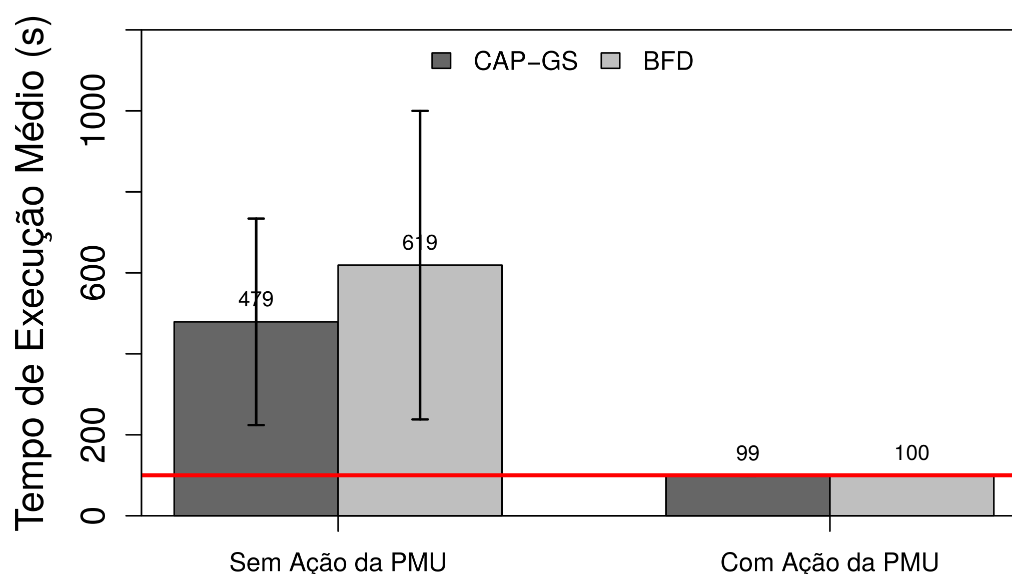
Este é mais um indicativo da intensidade com que estas funções interferem na execução das demais tarefas da aplicação. A avaliação deste parâmetro complementa as informações fornecidas através da análise do atraso de *deadlines*. O atraso sobre o tempo de execução apresentado pela Figura 31 se mostrou grande o suficiente para causar um elevado percentual de perda de *deadlines* quando não houve intervenção da PMU no processo de escalonamento. A

habilitação de seu mecanismo, por outro lado, impediu qualquer perda de *deadlines* durante seus experimentos, se mostrando de acordo com o atraso médio negativo observado.

5.3.4 Tempo de Execução

O último parâmetro monitorado durante a execução dos experimentos foi o tempo de execução de cada aplicação. Através da Figura 33 um gráfico que apresenta o tempo de execução médio das aplicações, contendo uma linha horizontal vermelha que indica o tempo de execução esperado para as aplicações (100 segundos) em comparação com a média de tempo obtida através da execução de cada um dos cenários apresentados.

Figura 33 – Tempo de execução médio das aplicações.



Fonte: Autor, 2017.

Como resultado, obteve-se que a execução das funções de utilidade foi capaz de prolongar o tempo médio necessário para a execução dos conjuntos de tarefas em, pelo menos, 4 vezes o tempo esperado. Os conjuntos de tarefas distribuídos através do algoritmo CAP-GS obtiveram um tempo de execução médio de 479 segundos, enquanto os que foram distribuídos através de BFD obtiveram um tempo de execução médio de 619 segundos. Ativar o uso do mecanismo proposto, por sua vez, fez com que suas aplicações fossem executadas em, aproximadamente, 100 segundos conforme esperado.

6 CONSIDERAÇÕES FINAIS

Em aplicações modernas, sistemas de tempo real são utilizados no processamento de elevadas cargas de dados, como no processamento de imagens, aplicações multimídia, simulações cientificas, entre outros. Neste cenário, processadores multinúcleos representam uma ótima alternativa para a redução de custos em determinados projetos. No entanto, o uso de multiprocessadores traz consigo uma série de fatores inconvenientes. Ao longo deste trabalho foram apresentadas algumas das dificuldades que se tem atualmente em utilizar processadores multinúcleos no processamento de sistemas de tempo real. Entre os tópicos de estudo voltados à atenuação de seus problemas, recursos de monitoramento de desempenho são utilizados na tomada de decisões durante o escalonamento de tarefas.

O presente trabalho foi desenvolvido com base nos seguintes objetivos: aprimorar o suporte do sistema operacional EPOS à unidades de monitoramento de desempenho; adquirir prática no desenvolvimento de sistemas embarcados de tempo real multiprocessados; estudar o modelo de escalonador para sistemas de tempo real proposto por Gracioli (2014); integrar o mecanismo de monitoramento proposto com o recurso de particionamento de memória cache desenvolvido por Gracioli (2014); avaliar a implementação proposta através da execução de diferentes conjuntos de tarefas sintéticas em *hardware* real; e apresentar os principais resultados obtidos.

Para tratar tais problemas uma série de melhorias sobre o RTOS EPOS foram propostas, buscando melhorar seu suporte a família de PMUs dos processadores Intel. Este trabalho apresentou uma revisão sobre os mediadores de *hardware* de sua PMU (IA32_PMU) e monitoramento de desempenho (Perf_Mon). Rotinas foram implementadas para possibilitar o uso dos recursos de interrupção e *Debug Store* oferecidos por processadores Intel modernos. Adicionalmente, um mecanismo capaz obter o perfil de execução de aplicações e utilizá-los na tomada de decisões durante o escalonamento de tarefas foi elaborado.

Este mecanismo foi avaliado em um *hardware* real através de uma aplicação sintética e intensiva em memória. A aplicação desenvolvida permite que cada experimento execute um diferente número de tarefas, com tipos, grupos de execução (partições), períodos de execução, WCETs e até mesmo núcleos de processamento; parâmetros definidos pelo desenvolvedor do experimento durante o processo de compilação. Desta forma cada conjunto de tarefas gerado foi avaliado em termos do algoritmo de empacotamento utilizado para sua distribuição entres núcleos do processador e utilizando o recurso de particionamento de memória cache desenvolvido por Gracioli (2014).

Por fim, utilizando conjuntos de tarefas gerados de forma aleatória foi demonstrado como a implementação proposta pode ser utilizada para monitorar determinado evento durante

a execução de uma aplicação e desempenhar um conjunto de ações correspondente através do tratador de sua interrupção. Durante a execução dos experimentos propostos, a ação do mecanismo de interrupção da PMU foi capaz de impedir a execução das tarefas de melhor esforço e garantir que suas aplicações fossem executadas de acordo com os parâmetros projetados.

Considerando as avaliações realizadas durante este trabalho, é importante enfatizar o uso de um RTOS voltado ao desenvolvimento de sistemas embarcados e a execução de suas tarefas em um processador real. No desenvolvimento dos experimentos propostos utilizou-se de cargas de trabalho sintéticas para explorar problemas de implementação, recursos de hardware e abordagens de agendamento. Entretanto, seria ainda mais interessante avaliar o mecanismo proposto utilizando cargas de trabalho reais.

Sabe-se que existem limitações na execução de cargas de trabalho de tempo real em sistemas multiprocessados, principalmente devido a três razões (BRANDENBURG, 2011): aplicações HRT são geralmente construídas através de *hardware* específico, composto de vários sensores e atuadores de difícil reprodução em um laboratório; aplicações reais de tempo real embarcadas não são tornadas públicas por empresas devido a razões comerciais; e as cargas de trabalho de aplicações existentes ainda não exploram todos os gargalos de um processador multinúcleo, uma vez que a adoção de multiprocessadores pela indústria de sistemas embarcados ainda está numa fase inicial.

Existem vários *benchmarks* propostos para explorar os recursos de um processador *multicore*, como o PARSEC (BIENIA; LI, 2011) e o SPEC2000 (SPEC, 2016). No entanto, esses *benchmarks* não apresentam restrições de tempo real e, portanto, não são apropriados para avaliar os impactos em processos de escalonamento no contexto de sistemas em tempo real.

Este trabalho se mostra como um primeiro passo na utilização de recursos de monitoramento de desempenho em processos de escalonamento de tarefas de tempo real. Uma vez que este mecanismo foi elaborado, um estudo detalhado sobre os eventos de monitoramento disponíveis e a elaboração de heurísticas de atuação sobre o processo de escalonamento se faz necessário.

REFERÊNCIAS

ARM. *Application Note 195: ARM11 performance monitor unit*. 2008. Online. Disponível em: <https://goo.gl/hZtmY6>. Citado na página 25.

AZIMI, R.; STUMM, M.; WISNIEWSKI, R. W. Online performance analysis by statistical sampling of microprocessor performance counters. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2005. (ICS '05), p. 101–110. ISBN 1-59593-167-8. Disponível em: <https://goo.gl/TYKzYl>. Citado 2 vezes nas páginas 25 e 26.

BANDYOPADHYAY, S. A study on performance monitoring counters in x86-architecture. *Indian Statistical Institute*, v. 43, 2004. Disponível em: <https://goo.gl/c96mFR>. Citado na página 25.

BERGER, A. S. Embedded systems design: An introduction to processes, tools, and techniques. In: ____. Taylor & Francis, 2002. (CMP Books), cap. Introduction, p. 1–9. ISBN 9781578200733. Disponível em: <https://books.google.com.jm/books?id=3vY35UkvXrAC>. Citado na página 12.

BIENIA, C.; LI, K. *Benchmarking modern multiprocessors*. New York: Princeton University New York, 2011. Disponível em: <https://goo.gl/qDCrmC>. Citado na página 66.

BITZES, G.; NOWAK, A. The overhead of profiling using pmu hardware counters. *CERN openlab report*, July 2014. Disponível em: <https://goo.gl/fjuxO3>. Citado 2 vezes nas páginas 15 e 25.

BRANDENBURG, B. B. *Scheduling and locking in multiprocessor real-time operating systems*. Tese (Doutorado) — University of North Carolina, Chapel Hill, 2011. Disponível em: <http://goo.gl/9aH8xK>. Citado 4 vezes nas páginas 22, 23, 24 e 66.

BRANDENBURG, B. B. *Schedcat: the schedulability test collection and toolkit.* 2013. Online. Disponível em: <https://goo.gl/oKDqGo>. Citado na página 56.

CALANDRINO, J.; ANDERSON, J. Cache-aware real-time scheduling on multicore platforms: heuristics and a case study. In: IEEE. *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*. Prague: IEEE, 2008. p. 299–308. ISSN 1068-3070. Disponível em: <http://goo.gl/rRgtFl>. Citado na página 13.

DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM COMPUTING SURVEYS*, v. 43, n. 4, 2011. Disponível em: <http://goo.gl/AT2tFn>. Citado na página 22.

DIJKSTRA, E. W. Cooperating sequential processes. In: HANSEN, P. B. (Ed.). *The origin of concurrent programming*. New York: Springer, 1968. p. 65–138. Citado na página 33.

EPOS. *Welcome to the EPOS Project*. 2016. Disponível em: <http://goo.gl/1RshTC>. Citado 2 vezes nas páginas 12 e 33.

FARINES, J.-M.; FRAGA, J. S.; OLIVEIRA, R. S. Sistemas de tempo real. *Escola de Computação*, 2000. Disponível em: <http://goo.gl/LsgRRQ>. Citado 3 vezes nas páginas 12, 21 e 22.

FERNáNDEZ, M.; CAZORLA, F. J. *Multi-core PMCs: analysis and architectural definition*. Barcelona, 2014. Online. Disponível em: <https://goo.gl/y2OBHQ>. Citado na página 25.

FRÖHLICH, A. A. A comprehensive approach to power management in embedded systems. *International Journal of Distributed Sensor Networks*, Hindawi Publishing Corporation, v. 2011, p. 19, 2011. Disponível em: <https://goo.gl/Ao85m0>. Citado na página 12.

FRÖHLICH, A. A.; WANNER, L. F. Operating system support for wireless sensor networks. *Journal of Computer Science*, v. 4, n. 4, p. 272, 2008. Disponível em: <https://goo.gl/r5J5QE>. Citado na página 12.

FRÖHLICH, A. A. M. *Application Oriented Operating Systems*. Tese (Doutorado) — Technische Universität Berlin, Berlin, 2001. Disponível em: <http://goo.gl/cIXSCh>. Citado 2 vezes nas páginas 12 e 32.

GRACIOLI, G. *Real-Time Operating System Support for Multicore Applications*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2014. Disponível em: <https://goo.gl/whSjTU>. Citado 10 vezes nas páginas 12, 13, 15, 16, 23, 24, 52, 56, 57 e 65.

GRACIOLI, G. et al. A survey on cache management mechanisms for predictable real-time embedded systems. *ACM Computing Surveys*, v. 48, November 2015. Disponível em: <http://goo.gl/le2zzG>. Citado 2 vezes nas páginas 13 e 20.

GRACIOLI, G.; FRÖHLICH, A. A. An embedded operating system api for monitoring hardware events in multicore processors. In: *Workshop on hardware-support for parallel program correctness—IEEE Micro*. Porto Alegre: [s.n.], 2011. Disponível em: <https://goo.gl/pY01Cx>. Citado 8 vezes nas páginas 25, 34, 35, 36, 37, 38, 39 e 40.

GRACIOLI, G.; FRÖHLICH, A. A. Cap: Color-aware task partitioning for multicore real-time applications. In: IEEE. *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. 2014. p. 1–8. Disponível em: <https://goo.gl/Thb3nC>. Citado na página 24.

HALANG, W. A. et al. Measuring the performance of real-time systems. *Real-Time Systems*, v. 18, n. 1, p. 59–68, 2000. ISSN 1573-1383. Disponível em: <http://dx.doi.org/10.1023/A:1008102611034>. Citado na página 12.

INAM, R. et al. The multi-resource server for predictable execution on multi-core platforms. In: IEEE. *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. Berlin: IEEE, 2014. p. 1–12. ISSN 1080-1812. Disponível em: <http://goo.gl/R5OPgK>. Citado na página 52.

INAM, R.; SJODIN, M. Combating unpredictability in multicores through the multi-resource server. In: IEEE. *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. Barcelona: IEEE, 2014. p. 1–8. Disponível em: <http://goo.gl/FxdjWV>. Citado na página 52.

INTEL CORPORATION. *Intel® Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide: Nehalem core PMU*. [S.l.], 2010. Disponível em: <https://goo.gl/WPnyJm>. Citado 2 vezes nas páginas 15 e 25.

INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. [S.l.], 2014. Disponível em: <http://goo.gl/LatZ2Z>. Nenhuma citação no texto.

INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual: system programming guide*. [S.l.], 2015. Disponível em: <http://goo.gl/s5xtGC>. Citado 4 vezes nas páginas 25, 26, 30 e 31.

JACOB, B. Cache design for embedded real-time systems. p. 9, 1999. Disponível em: <http://goo.gl/9pQ4Y9>. Citado 3 vezes nas páginas 15, 19 e 20.

KOPETZ, H. Real-time systems: design principles for distributed embedded applications. In: ____. 2. ed. [S.l.]: Springer US, 2011. cap. The Real-Time Environment, p. 1–28. Citado na página 13.

LIEDTKE, J.; HÄRTIG, H.; HOHMUTH, M. Os-controlled cache predictability for real-time systems. In: IEEE COMPUTER SOCIETY TECHNICAL COMMITTEE ON REAL-TIME SYSTEMS. *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*. Montreal, Que.: IEEE, 1997. p. 213–224. Citado 2 vezes nas páginas 15 e 20.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, ACM, v. 20, n. 1, p. 46–61, 1973. Disponível em: <http://goo.gl/rt9zed>. Citado na página 21.

MALL, R. *Real-Time Systems: Theory and Practice*. Pearson Education, 2009. ISBN 9788131700693. Disponível em: <https://books.google.com.br/books?id=coPT7vaEjFsC>. Citado na página 32.

MANCUSO, R. et al. Real-time cache management framework for multi-core architectures. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. [s.n.], 2013. p. 45–54. ISSN 1080-1812. Disponível em: <https://goo.gl/tO1vIv>. Citado 3 vezes nas páginas 19, 20 e 52.

MARANHãO, V. T. de L. *Armazenamento de arquivos grandes em DVDs*. 2009. Online. Disponível em: <https://goo.gl/XMupuo>. Citado na página 23.

MARCONDES, H. et al. On the design of flexible real-time schedulers for embedded systems. In: *2009 International Conference on Computational Science and Engineering*. [s.n.], 2009. v. 2, p. 382–387. Disponível em: <https://goo.gl/E4H5Vv>. Citado na página 12.

MARTELLO, S.; TOTH, P. Knapsack problems: algorithms and computer implementations. In: ____. [S.l.]: John Wiley & Sons, Inc., 1990. cap. Bin-packing problem, p. 221–245. Citado na página 23.

MÜCK, T. R.; FRÖHLICH, A. A. Hyra: a software-defined radio architecture for wireless embedded systems. In: *10th international conference on networks, St. Maarten, The Netherlands Antilles*. [s.n.], 2011. p. 246–251. Disponível em: <https://goo.gl/GQnTsR>. Citado na página 12.

PELOQUIN, M. et al. A comparison of scheduling algorithms for multiprocessors. Citeseer, December 2010. Disponível em: <http://goo.gl/tIQDmf>. Citado na página 21.

PITCHER, G. Growing number of ecus forces new approach to cars electrical architecture. *New Electronics*, p. 27–28, September 2012. Disponível em: <https://goo.gl/LBLu21>. Citado na página 13.

POLPETA, F. V.; FRÖHLICH, A. A. Hardware mediators: A portability artifact for component-based systems. In: ____. *Embedded and Ubiquitous Computing: International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 271–280. ISBN 978-3-540-30121-9. Disponível em: <http://dx.doi.org/10.1007/978-3-540-30121-9_26>. Citado na página 35.

RUSTAD, E. *NumaConnect*. 2013. Online. 35 slides. Disponível em: <https://goo.gl/IsHmhM>. Citado na página 18.

SHAW, A. C. Real-time systems and software. In: ____. 1st. ed. New York, NY, USA: John Wiley & Sons, Inc., 2001. cap. Introduction. ISBN 0471354902. Citado na página 12.

SPEC. *SPEC's Benchmarks*. 2016. Disponível em: <https://goo.gl/gT2Bwu>. Citado na página 66.

SPRUNT, B. Pentium 4 performance-monitoring features. *IEEE Micro*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 22, n. 4, p. 72–82, July 2002. ISSN 0272-1732. Disponível em: <https://goo.gl/6WuLf2>. Citado na página 25.

STALLINGS, W. Computer organization and architecture design for performance. In: ____. 8. ed. New Jersey: Pearson Prentice Hall, 2010. cap. Cache Memory, p. 110–157. Citado na página 18.

STALLINGS, W. Computer organization and architecture design for performance. In: ____. 8. ed. New Jersey: Pearson Prentice Hall, 2010. cap. Parallel Processing, p. 110–157. Citado na página 19.

STENSTROM, P. A survey of cache coherence schemes for multiprocessors. *Computer*, IEEE, v. 23, n. 6, p. 12–24, June 1990. Disponível em: <https://goo.gl/VJR5Xs>. Citado na página 19.

TANENBAUM, A. S. Sistemas operacionais modernos. In: ____. 3. ed. São Paulo: Pearson Prentice Hall, 2009. cap. Processos e Threads, p. 324–378. Citado na página 18.

TANENBAUM, A. S. Sistemas operacionais modernos. In: ____. 3. ed. São Paulo: Pearson Prentice Hall, 2009. cap. Sistemas com Múltiplos Processadores, p. 324–378. Citado na página 21.

TOBUSCHAT, S. et al. Idamc: A noc for mixed criticality systems. In: IEEE. *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*. Taipei: IEEE, 2013. p. 149–156. ISSN 1533-2306. Disponível em: <http://goo.gl/jnUVKy>. Citado na página 52.

WARD, B. C. et al. Making shared caches more predictable on multicore platforms. In: IEEE. *2013 25th Euromicro Conference on Real-Time Systems*. Paris, France, 2013. p. 157–167. Citado na página 56.

YUN, H. et al. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In: IEEE. *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. Berlin: IEEE, 2014. p. 155–166. ISSN 1080-1812. Disponível em: <http://goo.gl/1iTtuy>. Citado na página 14.

YUN, H. et al. Memguard: memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: IEEE. *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. Philadelphia, PA: IEEE, 2013. p. 55–64. ISSN 1080-1812. Disponível em: <http://goo.gl/Vc8q90>. Citado 2 vezes nas páginas 15 e 52.

YUN, H. et al. Memory bandwidth management for efficient performance isolation in multi-core platforms. *Computers, IEEE Transactions on*, PP, n. 99, p. 1–1, April 2015. ISSN 0018-9340. Disponível em: <http://goo.gl/RFS9Y0>. Citado na página 52.

APÊNDICES

APÊNDICE A – PROCESSO DE INICIALIZAÇÃO DA PMU NO EPOS

Figura 34 – Seção de código responsável por chamar a inicialização da PMU no núcleo 0 da CPU.

```
1  void Thread::init()
2  {
3
4      ...
5
6      // Seção da inicialização da classe Thread executada apenas pelo núcleo 0
7      if(Machine::cpu_id() == 0)
8      {
9
10         ...
11
12         // Inicialização da PMU para o núcleo 0
13         if(Traits<IA32_PMU>::enabled)
14         {
15
16             IA32_PMU::init();
17
18             if(pmc_interrupt_enabled)
19                 IA32_PMU::handler(&pmc_interrupt_handler, IA32_PMU::PMC0);
20         }
21         else
22             db<Init, IA32_PMU>(WRN) << "PMU is disabled!\n";
23     }
24 }
```

Fonte: Autor, 2017.

Figura 35 – Seção de código responsável por chamar inicialização da PMU nos núcleos da CPU, exceto no núcleo 0.

```
1  class Init_System
2  {
3  public:
4
5    Init_System(void)
6    {
7
8      ...
9
10     // Seção da inicialização do sistema executada pelos núcleos diferentes do núcleo 0
11     if(Machine::cpu_id() != 0)
12     {
13
14         ...
15
16         // Inicializa a PMU, caso este recurso seja habilitado por Traits
17         if(Traits<IA32_PMU>::enabled)
18             IA32_PMU::init();
19
20         ...
21     }
22
23     ...
24
25   }
26
```

Fonte: Autor, 2017.

Figura 36 – Seção de código responsável inicializar a PMU no EPOS (continua).

```
1  // Inicialização do Mediador EPOS IA32_PMU
2
3  #include <pmu.h>
4  #include <machine.h>
5
6  __BEGIN_SYS
7
8  // @brief Função desenvolvida para realizar a inicialização da família de PMUs de processadores Intel
9  void IA32_PMU::init()
10 {
11     db<Init,IA32_PMU>(TRC) << "IA32_PMU::init()\n";
12
13     cpuid10_edx edx;
14     cpuid10_eax eax;
15     Reg32 ebx, ecx = 0;
16
17     Reg32 CPU_Support = rdmsr(IA32_MISC_ENABLE);
18
19     if (!(CPU_Support & PMU_UNAVAILABLE))
20     {
21         db<Init,IA32_PMU>(WRN) << "PMU_UNAVAILABLE!\n";
22         return;
23     }
24
25     // PCE - Performance-Monitoring Counter Enable (bit 8 of CR4)
26     // Habilita os contadores de desempenho na CPU
27     IA32::cr4((IA32::cr4() | PCE));
28
29     // Seção executada apenas na CPU0: caracterização da PMU
30     if (APIC::id() == 0)
31     {
32         // Verifica a identificação fabricante
33         cpuid(0x00000000,
34             (Reg32*)&_cpuinfo.cpuid_level,
35             (Reg32*)&_cpuinfo.x86_vendor_id[0],
36             (Reg32*)&_cpuinfo.x86_vendor_id[8],
37             (Reg32*)&_cpuinfo.x86_vendor_id[4]);
38
39         _cpuinfo.x86 = 4;
40
41         // Identificação dos recursos da arquitetura Intel
42         if (_cpuinfo.cpuid_level >= 0x00000001)
43         {
44             Reg32 junk = 0;
45             Reg32 tfms, cap0, misc;
46
```

Fonte: Autor, 2017.

Figura 37 – Seção de código responsável inicializar a PMU no EPOS (continuação).

```
47                  cpuid(0x00000001, &tfms, &misc, &junk, &cap0);
48                  _cpuinfo.x86 = (tfms >> 8) & 0xf;
49                  _cpuinfo.x86_model = (tfms >> 4) & 0xf;
50                  _cpuinfo.x86_mask = tfms & 0xf;
51
52                  if (_cpuinfo.x86 == 0xf)
53                      _cpuinfo.x86 += (tfms >> 20) & 0xff;
54                  if (_cpuinfo.x86 >= 0x6)
55                      _cpuinfo.x86_model += ((tfms >> 16) & 0xf) << 4;
56
57                  if (cap0 & (1 << 19))
58                  {
59                      _cpuinfo.x86_clflush_size = ((misc >> 8) & 0xff) * 8;
60                      _cpuinfo.x86_cache_alignment =
61                          _cpuinfo.x86_clflush_size;
62                  }
63              }
64
65              // Identificação dos recursos da PMU
66              cpuid(10, &eax.full, &ebx, &ecx, &edx.full);
67
68              _version = eax.split.version_id;
69              _num_counters = eax.split.num_counters;
70              _cntval_bits = eax.split.bit_width;
71              _cntval_mask = (1ULL << eax.split.bit_width) - 1;
72
73              if (_version > 1)
74              {
75                  if ((int)edx.split.num_counters_fixed) > 3)
76                  {
77                      _num_counters_fixed =
78                          (int)edx.split.num_counters_fixed;
79                  }
80                  else
81                  {
82                      _num_counters_fixed = 3;
83                  }
84              }
85
86              if (_version > 1)
87              {
88                  _intel_cap.capabilities = rdmsr(CAPABILITIES);
89              }
90          }
91
```

Fonte: Autor, 2017.

Figura 38 – Seção de código responsável inicializar a PMU no EPOS (continuação).

```
92          // Verificação do suporte ao recurso DS Area
93          cpuid(1, &eax.full, &ebx, &ecx, &edx.full);
94
95          // Caso o recurso esteja disponível e habilitado em traits, sua inicialização será realizada
96          if ((Traits<IA32_PMU>::debug_save_enabled == true) &&
97              (edx.full && (0x01 << 21)))
98          {
99              ds_init(CPU_Support, ecx);
100         }
101
102         // Inicialização das Interrupções da PMU
103         perf_int_init();
104     }
105
106     // @brief Função desenvolvida para realizar a inicialização do recurso DS Area em processadores Intel
107     // @param[in] CPU_Support Variável de informações ao suporte de recursos da CPU
108     // @param[in] CPU_Info Informações adicionais de suporte aos recursos do processador
109     void IA32_PMU::ds_init(Reg32 CPU_Support, Reg32 CPU_Info)
110     {
111         db<Init,IA32_PMU>(TRC) << "IA32_PMU::ds_init()\n";
112
113         // Identifica o número da CPU
114         int cpu_id = APIC::id();
115
116         // Inicializa os buffers do recurso DS Area
117         for (int i = 0; i < IA32_DS_BUFFER_SIZE; i++)
118             ds_buffer_list[cpu_id].ds_buffer[i] = 0;
119
120         for (int i = 0; i < (24 * IA32_BTS_BUFFER_SIZE); i++)
121             ds_buffer_list[cpu_id].bts_buffer[i] = 0;
122
123         for (int i = 0; i < (22 * IA32_PEBS_BUFFER_SIZE); i++)
124             ds_buffer_list[cpu_id].pebs_buffer[i] = 0;
125
126         // Configuração do BTS Buffer
127         if (!(CPU_Support & BTS_UNAVAILABLE))
128         {
129             db< Init, IA32_PMU >(TRC) << "Setting Up the BTS Buffer.\n";
130
131             // Para prevenir a interrupção do sistema ao trabalhar com buffers circulares, o SW ajustar
132             // o limiar de interrupção para um valor maior do que o BTS máximo absoluto
133
134             // Endereço inicial do BTS Buffer
135             ds_buffer_list[cpu_id].ds_buffer[0] =
136                 (unsigned long long)(&ds_buffer_list[cpu_id].
137                     bts_buffer[0]);
138
139             // Índice do BTS Buffer
140             ds_buffer_list[cpu_id].ds_buffer[1] =
141                 (unsigned long long)(&ds_buffer_list[cpu_id].
142                     bts_buffer[0]);
143
```

Fonte: Autor, 2017.

Figura 39 – Seção de código responsável inicializar a PMU no EPOS (continuação).

```
150             // Endereço de identificação de fim do BTS Buffer
151             ds_buffer_list[cpu_id].ds_buffer[2] =
152                 (unsigned long long)(&ds_buffer_list[cpu_id].
153                     bts_buffer[24 * IA32_BTS_BUFFER_SIZE]);
154
155             // Endereço para o buffer limiar de interrupção
156             ds_buffer_list[cpu_id].ds_buffer[3] =
157                 (unsigned long long)(&ds_buffer_list[cpu_id].
158                     bts_buffer[24]);
159
160             // Por padrão, o recurso BTS inicia desabilitado
161             wrmsr(IA32_DEBUGCTL, 0x00);
162         }
163         else
164             db<Init, IA32_PMU>(WRN) << "BTS_UNAVAILABLE!\n";
165
166         // Configuração de PEBS
167         if (!(CPU_Support & PEBS_UNAVAILABLE))
168         {
169             db<Init, IA32_PMU>(TRC) << "Setting Up the PEBS Buffer.\n";
170
171             // Endereço inicial de PEBS Buffer
172             ds_buffer_list[cpu_id].ds_buffer[4] =
173                 (unsigned long long)(&ds_buffer_list[cpu_id].
174                     pebs_buffer[0]);
175
176             // Índice de PEBS Buffer
177             ds_buffer_list[cpu_id].ds_buffer[5] =
178                 (unsigned long long)(&ds_buffer_list[cpu_id].
179                     pebs_buffer[0]);
180
181             // Endereço de identificação de fim do PEBS Buffer
182             ds_buffer_list[cpu_id].ds_buffer[6] =
183                 (unsigned long long)(&ds_buffer_list[cpu_id].
184                     pebs_buffer[22 * IA32_PEBS_BUFFER_SIZE]);
185
186             // Endereço para o buffer limiar de interrupção
187             ds_buffer_list[cpu_id].ds_buffer[7] =
188                 (unsigned long long)(&ds_buffer_list[cpu_id].
189                     pebs_buffer[22]);
190
191             // Valores iniciais para os contadores PEBS
192             ds_buffer_list[cpu_id].ds_buffer[8] = 0;
193             ds_buffer_list[cpu_id].ds_buffer[9] = 0;
194             ds_buffer_list[cpu_id].ds_buffer[10] = 0;
195             ds_buffer_list[cpu_id].ds_buffer[11] = 0;
196         }
197         else
198             db<Init, IA32_PMU>(WRN) << "PEBS_UNAVAILABLE!" << endl;
199     }
200
```

Fonte: Autor, 2017.

Figura 40 – Seção de código responsável inicializar a PMU no EPOS (conclusão).

```
184   // @brief Função desenvolvida para realizar a inicialização das interrupções da família de PMUs de
185   // processadores Intel
186   void IA32_PMU::perf_int_init()
187   {
188       // Desabilita as interrupções do sistema
189       CPU::int_disable();
190
191       // Inicializa a interrupção dos contadores de desempenho na APIC
192       APIC::config_perf();
193
194       // No CPU0, inicializa os ponteiros para tratadores de interrupção dos PMCs
195       if(APIC::id() == 0)
196       for (int i = 0; i < 8; i++)
197           IA32_PMU::_pmc_handler[i] = 0;
198
199       // Atribui o endereço do tratador de interrupções da PMU a tabela de interrupções do sistema
200       IC::int_vector(IC::INT_PERF_INIT,
201           Intel_Sandy_Bridge_PMU::PMU_int_handler);
202
203       // Habilita o recurso de interrupções da PMU
204       IC::enable(IC::INT_PERF_INIT);
205
206       // Habilita as interrupções do sistema
207       CPU::int_enable();
208   }
209
210   __END_SYS
211
212
```

Fonte: Autor, 2017.

APÊNDICE B – TRATADOR GLOBAL DE INTERRUPÇÕES PARA A PMU NO EPOS

Figura 41 – Seção de código responsável pelo tratamento global de interrupções da PMU (continua).

```
1   // @brief Function designed to act as PMU interrupt service routine
2   // @param[in] irq The irq
3   void Intel_Sandy_Bridge_PMU::PMU_int_handler(unsigned int irq)
4   {
5       db<Init,IA32_PMU>(TRC) << "IA32_PMU::PMU_int_handler()\n";
6
7       Reg64 perf_ovf_msr, pebs_enable_msr, pmcs_enable_msr;
8
9       // Identifica quais PMCs se encontram habilitados
10      pmcs_enable_msr = rdmsr(GLOBAL_CTRL);
11
12      // Desabilita todos os contadores
13      wrmsr(GLOBAL_CTRL, pmcs_enable_msr & ~(PMC_MASK));
14
15      // Identifica quais PEBS se encontram habilitados
16      pebs_enable_msr = rdmsr(PEBS_ENABLE);
17
18      // Identifica quais contadores e/ou buffers sofreram overflow
19      perf_ovf_msr = get_global_status();
20
21      // Seção de tratamento para o recurso DS Área
22      if (ds_overflow() || (pebs_enable_msr != 0))
23      {
24          db<Init,IA32_PMU>(TRC) << "IA32_PMU::ds_overflow();" << endl;
25
26          // Desabilita a opção Trace de DS Area
27          wrmsr(IA32_DEBUGCTL, (rdmsr(IA32_DEBUGCTL) & ~IA32_DEBUGCTL_TR));
28
29          // Identifica o processador em execução
30          int cpu_id = APIC::id();
31
32          // Reinicia o índie de BTS Buffer
33          ds_buffer_list[cpu_id].ds_buffer[1] =
34              (unsigned long long)(&ds_buffer_list[cpu_id].bts_buffer[0]);
35
36          // Reinicia o índice de PEBS Buffer
37          ds_buffer_list[cpu_id].ds_buffer[5] =
38              (unsigned long long)(&ds_buffer_list[cpu_id].pebs_buffer[0]);
39
40          // Limpa o indicador de overflow de DS Area
41          clear_ds_overflow();
42      }
```

Fonte: Autor, 2017.

Figura 42 – Seção de código responsável pelo tratamento global de interrupções da PMU (conclusão).

```
40
41          // Seção de tratamento para os PMCs
42          if (pmc_overflow())
43          {
44              db<Init, IA32_PMU>(INF) << "IA32_PMU::PMC_overflow("
45                                      << APIC::id() << ");" << endl;
46
47              // Laço responsável por desabilitar os contadores que sofreram overflow e limpar seu status
48              for (Reg32 i = PMC7_OVF; i > 0; i >>= 1)
49              {
50                  if ((perf_ovf_msr & i) != 0)
51                  {
52                      pmcs_enable_msr = pmcs_enable_msr & ~i;
53                      clear_pmc_overflow(i);
54                  }
55              }
56          }
57
58          // Limpa o registrador que indica a ocorrência de modificações em GLOBAL_STATUS
59          clear_cond_changed();
60
61          // Reabilita as interrupções dos PMCs
62          APIC::enable_perf();
63
64          // Reabilita os contadores em operação
65          wrmsr(GLOBAL_CTRL, pmcs_enable_msr);
66
67          // Chama os tratadores específicos de cada contador do PMC
68          Reg32 j = PMC7_OVF;
69          for (Reg32 i = 7; i < 8; i--)
70          {
71              // Verifica quais PMCs entraram em overflow e se seu tratador foi definido
72              if ((_pmc_handler[i] != 0) && ((perf_ovf_msr & j) != 0))
73              {
74                  // Então, chama seu tratador
75                  _pmc_handler[i]();
76              }
77
78              j >>= 1;
79          }
80      }
81
82      __END_SYS
```

Fonte: Autor, 2017.

APÊNDICE C – AÇÃO DA PMU NO PROCESSO DE ESCALONAMENTO DE TAREFAS

Figura 43 – Seção de código responsável pela configuração de eventos para o monitoramento de *Threads*.

```
1  // @brief Função desenvolvida para iniciar a troca de contexto entre as Threads prev e next
2  // @param prev Thread em execução
3  // @param next Próxima Thread na fila de prontos para execução
4  void Thread::dispatch(Thread * prev, Thread * next)
5  {
6      ...
7
8      // Se habilitado em Traits, configura o evento de monitoramento de Threads
9      if (Traits<IA32_PMU>::pmc_interrupt_enabled)
10         Perf_Mon::pmu_interrupt_config(next->_event_number);
11
12     ...
13 }
14
15 // @brief Função desenvolvida para configurar o evento de monitoramento de Threads
16 // @param[in] event_number Número inicial de eventos
17 // @note Para utilizar o recurso de interrupções, recomenta-se definir event_number com um valor negativo,
18 // este artifício facilita na determinação do número de eventos que precedem uma interrupção. Sabendo que o
19 // overflow ocorre quando ocorre uma transição por zero, event_number é dado por: (valor_negativo + 1).
20 // Por exemplo: para gerar uma interrupção imediatamente após a centésima ocorrência de um evento,
21 // (-100 + 1) == -99.
22 void Perf_Mon::pmu_interrupt_config(long long event_number)
23 {
24     // Desabilita as interrupções do sistema, caso o recurso de interrupções da PMU esteja ativo
25     APIC::disable_perf_int();
26
27     // Desabilita a contagem do evento configurado em PMC0
28     PMU::disable(PMU::EVTSEL0);
29
30     // Define um valor inicial de contagem para PMC0
31     Intel_Sandy_Bridge_PMU::wrpmc(PMU::PMC0, event_number);
32
33     // Define o evento a ser monitorado por PMC0, habilitando sua interrupção e contagem de eventos
34     Intel_Sandy_Bridge_PMU::config(PMU::EVTSEL0,
35             (Intel_Sandy_Bridge_PMU::LLC_MISSES |
36              Intel_Sandy_Bridge_PMU::USR | Intel_Sandy_Bridge_PMU::OS |
37              Intel_Sandy_Bridge_PMU::INT | Intel_Sandy_Bridge_PMU::ENABLE));
38
39     // Reabilita as interrupções do sistema
40     APIC::enable_perf_int();
41 }
42
43 // @brief Função desenvolvida para atuar como o tratador da interrupção gerada pelo monitoramento de
44 // Threads e realizar a ação desejada
45 void Thread::pmc_interrupt_handler(void)
46 {
47     // Operação de escalonamento
48 }
```

Fonte: Autor, 2017.

APÊNDICE D – PARÂMETROS DOS CONJUNTOS DE TAREFAS GERADOS

Tabela 2 – Parâmetros de definição do conjunto de tarefas 0.

Conjunto de Tarefas	Identificação da Tarefa	Grupo/Partição	Tipo	CPU		Período	WCET
				CAP-GS	BFD		
0	1	-1	0	0	2	25000	6578
	2	-1	0	0	4	75000	15648
	3	-1	0	1	6	75000	15417
	4	-1	0	4	4	25000	5736
	5	-1	0	6	0	50000	15901
	6	-1	0	1	0	25000	8504
	7	-1	0	5	4	25000	6151
	8	-1	0	2	1	75000	22828
	9	-1	0	2	3	50000	12678
	10	1	1	3	7	200000	23386
	11	1	1	7	1	150000	41672
	12	1	1	7	7	150000	20674
	13	1	1	6	6	150000	25114
	14	1	2	7	7	1	0
	15	2	1	3	7	200000	27816
	16	2	1	3	3	200000	33625
	17	2	1	5	3	100000	25901
	18	2	1	4	7	150000	20067
	19	2	2	6	1	1	0
	20	3	1	5	2	500000	134856
	21	3	1	4	5	500000	111678
	22	3	1	2	2	200000	29672
	23	3	1	1	6	150000	8785
	24	3	2	4	4	1	0
	25	4	1	0	5	150000	33584
	26	4	1	7	5	500000	112529
	27	4	1	1	6	500000	111213
	28	4	1	4	0	500000	14311
	29	4	2	5	5	1	0

Fonte: Autor, 2017.

Tabela 3 – Parâmetros de definição do conjunto de tarefas 1.

Conjunto de Tarefas	Identificação da Tarefa	Grupo/Partição	Tipo	CPU		Período	WCET
				CAP-GS	BFD		
1	1	-1	0	0	1	50000	15838
	2	-1	0	0	1	25000	7183
	3	-1	0	1	2	50000	13441
	4	-1	0	1	0	25000	8685
	5	-1	0	2	2	50000	14306
	6	-1	0	2	0	50000	16484
	7	-1	0	5	4	50000	12181
	8	-1	0	5	5	75000	17699
	9	1	1	6	6	200000	41717
	10	1	1	3	7	150000	18117
	11	1	1	3	4	500000	91466
	12	1	1	7	7	500000	62628
	13	1	1	6	6	200000	12489
	14	1	2	4	5	1	0
	15	2	1	7	3	100000	26399
	16	2	1	4	2	150000	19702
	17	2	1	6	3	200000	53249
	18	2	1	7	7	150000	5762
	19	2	2	5	3	1	0
	20	3	1	4	4	500000	124754
	21	3	1	4	6	200000	35099
	22	3	1	2	5	100000	21007
	23	3	1	7	1	150000	9739
	24	3	2	1	0	1	0
	25	4	1	4	7	100000	11791
	26	4	1	5	5	500000	107394
	27	4	1	3	3	150000	23809
	28	4	1	0	6	150000	31286
	29	4	2	4	4	1	0

Fonte: Autor, 2017.

Tabela 4 – Parâmetros de definição do conjunto de tarefas 2.

Conjunto de Tarefas	Identificação da Tarefa	Grupo/Partição	Tipo	CPU		Período	WCET
				CAP-GS	BFD		
2	1	-1	0	0	1	75000	23736
	2	-1	0	0	2	50000	14320
	3	-1	0	1	0	75000	24569
	4	-1	0	6	3	50000	12707
	5	-1	0	1	4	50000	11151
	6	-1	0	2	4	25000	5009
	7	-1	0	5	2	75000	21481
	8	-1	0	2	0	50000	16472
	9	1	1	4	5	200000	33390
	10	1	1	3	4	100000	23545
	11	1	1	3	6	150000	21816
	12	1	1	7	6	200000	30431
	13	1	2	7	1	1	0
	14	2	1	5	2	100000	11528
	15	2	1	4	5	200000	34507
	16	2	1	5	3	200000	50880
	17	2	1	7	6	150000	23667
	18	2	2	6	0	1	0
	19	3	1	4	7	200000	26663
	20	3	1	2	1	200000	59885
	21	3	1	6	7	150000	16938
	22	3	1	7	5	500000	77169
	23	3	2	4	0	1	0
	24	4	1	3	3	100000	17944
	25	4	1	1	5	150000	29240
	26	4	1	1	7	200000	23902
	27	4	1	0	6	500000	78288
	28	4	1	0	1	150000	7431
	29	4	2	2	2	1	0

Fonte: Autor, 2017.

APÊNDICE E – RESULTADOS DA AVALIAÇÃO DO MECANISMO DE AÇÃO DA PMU

Tabela 5 – Experimento sobre o conjunto de tarefas 0 utilizando CAP-GS sem ação da PMU.

Task Set	CAP-GS sem Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
0	1	32472	31965	30185	6785	9568613	20000	385	764
	2	229065	216355	105313269	14346	11619114	20000	2590	
	3	377253	304009	101368300	35623	11149710	20000	6638	
	4	35117	32654	394775	12611	11750218	20000	6243	
	5	179912	131065	135946989	37937	11115720	20000	11094	
	6	47501	47176	18515	11280	11426466	20000	1366	
	7	29643	29228	20523	7718	11438244	20000	309	
	8	48503	48185	14633	13800	4546569	20000	135	
	9	83312	78257	4176623	23478	9309121	20000	4398	
	10	300740	282781	32726139	54798	10084337	20000	611	
	11	6543662	3638754	42900533	525611	11542726	20000	19700	
	12	79358722	13552701	96967649	162220	11269202	20000	13362	
	13	76206057	62331001	72679246	544443	9333357	20000	17604	
	14	0	0	0	0	0	712899	0	
	15	80878826	41412012	53978960	419546	11407805	20000	18748	
	16	42472428	29229462	91560862	566616	11400819	20000	19755	
	17	215331	207990	32688865	94409	10944883	20000	12245	
	18	302058	286784	39351275	61064	10463348	20000	3040	
	19	0	0	0	0	0	883738	0	
	20	175346307	43228101	93401518	3798291	10443510	20000	19900	
	21	2163841	2141266	128654484	1041611	10435864	20000	19800	
	22	122559	105056	35179320	44643	11198193	20000	0	
	23	27436	26279	105201	15858	6049637	20000	11981	
	24	0	0	0	0	0	361475	0	
	25	64249	63808	118887	29419	6685584	20000	0	
	26	219652	168889	60251234	101270	9950101	20000	0	
	27	164388	151818	22444815	103999	11116917	20000	0	
	28	1894313	1119797	123368264	34797	11686954	20000	634	
	29	0	0	0	0	0	365406	0	

Fonte: Autor, 2017.

Tabela 6 – Experimento sobre o conjunto de tarefas 1 utilizando CAP-GS sem ação da PMU.

Task Set	CAP-GS sem Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
1	1	55898	42981	67751162	15649	12605571	20000	236	397
	2	40007	39453	39419	7301	5442705	20000	300	
	3	5215153	4933485	76603390	118889	10744282	20000	19900	
	4	29702	29394	16572	12217	14740606	20000	1967	
	5	140591	138881	1536786	41461	13530015	20000	9878	
	6	151610	148395	12504629	18224	10919297	20000	2875	
	7	17166	15875	430589	11876	9002521	20000	0	
	8	87866	86152	8024983	20020	10852284	20000	262	
	9	1010882	936530	110697072	230426	11791736	20000	15449	
	10	97473	97045	21560	58803	11017042	20000	0	
	11	42643407	42411601	56763170	1289436	9547248	20000	19900	
	12	251329052	77858269	80191386	1967951	10402948	20000	19547	
	13	156890	121877	60500606	79883	12187195	20000	0	
	14	0	0	0	0	0	212646	0	
	15	257207	256048	57496	150613	10839716	20000	9930	
	16	24739233	24671302	59148319	297317	10687275	20000	19584	
	17	60371307	59986300	108974581	826188	10157248	20000	19900	
	18	25799	25580	2844	16618	2708472	20000	0	
	19	0	0	0	0	0	447821	0	
	20	1099753	1096565	1895345	751065	9475385	20000	19900	
	21	22095881	21837929	93482766	471083	11561467	20000	19900	
	22	219050	217547	70155	136800	10442509	20000	10000	
	23	61341	60908	17102	45968	7583324	20000	4018	
	24	0	0	0	0	0	465579	0	
	25	91874	91248	49649	89565	1826844	20000	19900	
	26	54683673	42018930	132850653	474224	11566525	20000	19134	
	27	32915510	32778617	76579805	221086	9349474	20000	4118	
	28	1330369	1202840	37603699	258933	9657188	20000	16745	
	29	0	0	0	0	0	400	0	

Fonte: Autor, 2017.

Tabela 7 – Experimento sobre o conjunto de tarefas 2 utilizando CAP-GS sem ação da PMU.

Task Set	CAP-GS sem Ação da PMU								
	Thread ID	**WCET**	**WCET_m**	**WCET_var**	**ET_m**	**ET_var**	**Número de Execuções**	**Deadlines Perdidos**	**WCET (Experimento)**
2	1	211520	210030	497142	100580	9681127	20000	19900	275
	2	43167	23643	8654593	12806	8857326	20000	64	
	3	84454	67735	31720748	39340	11060578	20000	4635	
	4	27746	26645	76347	12930	8850650	20000	0	
	5	56295	52701	3472118	14550	11887642	20000	789	
	6	27719	27638	1644	9267	11482601	20000	3347	
	7	60516	57830	364509	30006	10199959	20000	0	
	8	36646	36005	104542	12323	10088520	20000	0	
	9	14214624	7876794	5291892	246515	11281693	20000	14438	
	10	188195	121049	27932048	38296	10081232	20000	4878	
	11	136621	87271	35204983	50844	10377656	20000	60	
	12	12616835	5711437	38355261	247230	10900607	20000	14431	
	13	0	0	0	0	0	187537	0	
	14	63009	62647	28432	25223	11813026	20000	0	
	15	62023121	16779956	29233482	554347	11196124	20000	15942	
	16	4205298	3860273	29257837	439991	10552196	20000	19526	
	17	134593	133622	562441	72405	11239840	20000	0	
	18	0	0	0	0	0	108253	0	
	19	188253	187485	111039	108270	11142121	20000	0	
	20	2962572	2817397	25949672	566925	11978123	20000	15016	
	21	107922	106409	401466	50407	11099202	20000	0	
	22	108528447	22127711	32403974	1367764	11619229	20000	19900	
	23	0	0	0	0	0	108038	0	
	24	206666	191355	3070778	50061	10994444	20000	5248	
	25	1649230	1391011	12798261	293834	9304150	20000	13299	
	26	58005188	12741832	22763015	585332	10877823	20000	15288	
	27	113702796	3068620	21499947	733419	11178627	20000	18665	
	28	97011	96663	14945	71419	9579389	20000	19900	
	29	0	0	0	0	0	326828	0	

Fonte: Autor, 2017.

Tabela 8 – Experimento sobre o conjunto de tarefas 0 utilizando CAP-GS com ação da PMU.

Task Set	CAP-GS com Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
0	1	6448	6408	389	4177	1601193	20000	0	100
	2	13818	13728	1237	6179	4977590	20000	0	
	3	15401	15329	641	7082	14269000	20000	0	
	4	5910	5849	638	2687	1485523	20000	0	
	5	16091	15987	1146	7749	16201655	20000	0	
	6	8305	8269	272	4416	3787504	20000	0	
	7	6039	6002	317	3815	1467437	20000	0	
	8	26063	19654	419203	9444	8618126	20000	0	
	9	12780	12729	331	5734	15720801	20000	0	
	10	20547	20389	19400	9479	10731318	20000	0	
	11	43141	42747	7960	19606	18263137	20000	0	
	12	19792	19414	14956	8756	17453658	20000	0	
	13	26173	25957	6094	8131	17653251	20000	0	
	14	0	0	0	0	0	0	0	
	15	27037	26729	15414	10244	4400840	20000	0	
	16	35684	35169	22801	14165	4578853	20000	0	
	17	26758	26682	1030	12659	17316470	20000	0	
	18	19481	19344	4956	7924	14431126	20000	0	
	19	0	0	0	0	0	0	0	
	20	139501	139211	7752	89557	7764429	20000	0	
	21	102470	102243	7777	76124	7225634	20000	0	
	22	31026	30897	1881	14488	19378195	20000	0	
	23	8684	8647	155	4455	3992478	20000	0	
	24	0	0	0	0	0	0	0	
	25	33030	32813	9726	17061	2282728	20000	0	
	26	113043	112906	4516	76643	15696810	20000	0	
	27	111014	110830	3183	72536	1312974	20000	0	
	28	15314	15239	842	3868	2577329	20000	0	
	29	0	0	0	0	0	0	0	

Fonte: Autor, 2017.

Tabela 9 – Experimento sobre o conjunto de tarefas 1 utilizando CAP-GS com ação da PMU.

Task Set	CAP-GS com Ação da PMU							
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
1	1	11676	11564	1662	7971	2324645	20000	0	99
	2	6536	6473	461	3565	1284331	20000	0	
	3	12791	12714	685	8425	7229974	20000	0	
	4	8326	8262	425	3523	2324788	20000	0	
	5	13221	13132	569	7301	8343689	20000	0	
	6	10852	10774	743	6432	4044790	20000	0	
	7	12961	12804	3284	10420	1565483	20000	0	
	8	15023	14749	3105	7585	7946896	20000	0	
	9	34149	34015	3176	22650	9590849	20000	0	
	10	17533	17430	1090	9733	6770949	20000	0	
	11	58579	58429	5039	37319	1285759	20000	0	
	12	45857	45443	36973	14870	1876432	20000	0	
	13	7453	7392	625	4918	797555	20000	0	
	14	0	0	0	0	0	0	0	
	15	23681	23594	549	16127	2489619	20000	0	
	16	19486	19223	15054	11292	11152741	20000	0	
	17	53866	53319	66891	22306	3273945	20000	0	
	18	5288	5253	191	3444	795630	20000	0	
	19	0	0	0	0	0	0	0	
	20	81168	80434	127200	45846	5557686	20000	0	
	21	26530	26333	17458	14491	21048930	20000	0	
	22	21097	20994	1630	11393	6114959	20000	0	
	23	9484	9408	1038	4862	5193438	20000	0	
	24	0	0	0	0	0	0	0	
	25	8451	8335	2334	4541	3206032	20000	0	
	26	116612	115799	88397	44721	11754756	20000	0	
	27	22928	22657	4369	9471	2077434	20000	0	
	28	32367	32068	13858	16100	7611827	20000	0	
	29	0	0	0	0	0	0	0	

Fonte: Autor, 2017.

Tabela 10 – Experimento sobre o conjunto de tarefas 2 utilizando CAP-GS com ação da PMU.

Task Set	CAP-GS com Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
2	1	22242	22125	1721	14352	9575332	20000	0	99
	2	15474	15347	1801	7350	9193575	20000	0	
	3	24526	24437	1639	14077	12480321	20000	0	
	4	12751	12686	598	7575	8165403	20000	0	
	5	10776	10722	459	7311	4998209	20000	0	
	6	4697	4668	125	2270	1080722	20000	0	
	7	20756	20721	293	16556	8325318	20000	0	
	8	15529	15369	3504	6544	12846772	20000	0	
	9	34297	33973	27224	18729	6009563	20000	0	
	10	22428	22339	618	9479	6824314	20000	0	
	11	21779	21682	1547	10769	12754409	20000	0	
	12	30310	30234	524	22605	11015169	20000	0	
	13	0	0	0	0	0	0	0	
	14	11091	11040	678	6446	10588840	20000	0	
	15	32991	32882	1510	26984	7529882	20000	0	
	16	51395	50958	69232	27907	10995410	20000	0	
	17	24188	24130	584	12362	1241554	20000	0	
	18	0	0	0	0	0	0	0	
	19	26595	26537	761	15485	18958494	20000	0	
	20	59915	59775	3294	32174	9066032	20000	0	
	21	16913	16828	829	9443	9490765	20000	0	
	22	77576	77283	76529	43863	2163511	20000	0	
	23	0	0	0	0	0	0	0	
	24	16945	16910	271	11436	14124708	20000	0	
	25	29492	29314	5468	17036	16846538	20000	0	
	26	23418	23317	972	17373	2509882	20000	0	
	27	83342	83065	10346	35912	11528373	20000	0	
	28	7374	7336	155	4798	4345293	20000	0	
	29	0	0	0	0	0	0	0	

Fonte: Autor, 2017.

Tabela 11 – Experimento sobre o conjunto de tarefas 0 utilizando BFD sem ação da PMU.

Task Set	BFD sem Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
0	1	29923	29540	25035	8336	3054108	20000	400	632
	2	122810	108666	11910097	17453	10561457	20000	1661	
	3	84526	84305	8307	17177	9512060	20000	897	
	4	133905	122037	50966967	12737	8390779	20000	9171	
	5	143952	142458	1063050	38963	10391015	20000	11193	
	6	56269	56000	10560	13220	5524223	20000	4997	
	7	64223	63897	13090	9774	8577909	20000	1062	
	8	44199	40331	885880	19122	6363465	20000	0	
	9	47815	46586	127421	16951	13141904	20000	0	
	10	48377631	36951390	110551073	317645	11173819	20000	16526	
	11	387840	365400	55162567	63339	11330014	20000	6407	
	12	742772	649126	56415765	103789	11405960	20000	10119	
	13	303128	217000	109281677	79474	11142058	20000	2111	
	14	0	0	0	0	0	599282	0	
	15	38862251	26378997	57711335	311100	11976211	20000	15854	
	16	8884107	8739070	71945995	300092	12288926	20000	19591	
	17	175810	174875	137101	95025	11766533	20000	8538	
	18	136603	135938	57746	74922	10592117	20000	0	
	19	0	0	0	0	0	672423	0	
	20	21607244	20017473	87664432	1341059	10258585	20000	19899	
	21	180967027	48737913	107313754	2519646	11335742	20000	19900	
	22	186881	185513	905974	135536	11746653	20000	0	
	23	149426	59207	47884226	38451	7489388	20000	15	
	24	0	0	0	0	0	559882	0	
	25	431417	425547	11521592	197754	10990677	20000	9996	
	26	481538893	100267365	55220062	2859205	10481783	20000	18995	
	27	6562984	3842563	119892592	499554	10737854	20000	17065	
	28	926207	516418	34021233	48597	10429361	20000	459	
	29	0	0	0	0	0	500	0	

Fonte: Autor, 2017.

Tabela 12 – Experimento sobre o conjunto de tarefas 1 utilizando BFD sem ação da PMU.

Task Set	BFD sem Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
1	1	95142	50463	76299731	13887	10174031	20000	303	994
	2	35061	34585	25296	6129	9670065	20000	857	
	3	99461	68370	46727653	22153	11331838	20000	4246	
	4	45404	45002	26574	7835	19664633	20000	305	
	5	51519	39317	6853992	12230	8186109	20000	4	
	6	118452	105776	57777474	12284	6397842	20000	920	
	7	52924	28652	24744435	9913	11714113	20000	4	
	8	50441	49506	1195216	16262	18522963	20000	0	
	9	16148791	864699	118511286	54275	10761624	20000	2260	
	10	88930	45986	73443851	16962	11304061	20000	0	
	11	479190817	114922512	57125499	4797945	10815145	20000	19568	
	12	1621317	337240	116425001	148641	10470837	20000	96	
	13	16159102	586958	105101826	17863	10873617	20000	925	
	14	0	0	0	0	0	871598	0	
	15	144490	137316	16437532	64345	9984529	20000	4735	
	16	392710	321148	89426497	81917	11525066	20000	4840	
	17	15465954	13386112	81721658	253621	11885093	20000	19900	
	18	22934	21749	59057	13859	7529698	20000	0	
	19	0	0	0	0	0	1036985	0	
	20	181851681	10269464	102324174	2474758	10651044	20000	19900	
	21	2914266	621322	115269956	170383	10174899	20000	14912	
	22	123067	115441	3402629	64287	11288471	20000	5413	
	23	228996	118081	86911455	63551	10633533	20000	24	
	24	0	0	0	0	0	1179900	0	
	25	28792	28539	32921	14663	12984801	20000	0	
	26	9794810	7042487	128559360	444715	10270774	20000	11367	
	27	10143499	2241958	135778091	97375	10522473	20000	8613	
	28	110418	93999	11540114	58586	10406403	20000	0	
	29	0	0	0	0	0	663	0	

Fonte: Autor, 2017.

Tabela 13 – Experimento sobre o conjunto de tarefas 2 utilizando BFD sem ação da PMU.

Task Set	BFD sem Ação da PMU								WCET (Experimento)
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	
2	1	49115	46900	1593649	24781	10758642	20000	0	232
	2	50861	34804	4588678	11006	7418194	20000	1	
	3	91771	55435	35687875	23553	12176991	20000	3922	
	4	26017	21852	4766785	10465	7404470	20000	0	
	5	51437	49995	16975603	9993	13801944	20000	102	
	6	22572	22226	8851	6183	12488935	20000	0	
	7	40198	27321	8881128	21922	3006373	20000	14	
	8	52789	51718	579554	13402	12323946	20000	96	
	9	16822791	9231644	39532940	172728	10878702	20000	12243	
	10	123550	98578	21201983	34038	10747417	20000	102	
	11	262145	174725	38233278	41899	10266482	20000	2974	
	12	1943659	1207403	5684558	130788	11924865	20000	10088	
	13	0	0	0	0	0	92203	0	
	14	81549	72996	1839851	28602	10944718	20000	83	
	15	5827424	3681012	6915397	186752	10954719	20000	12483	
	16	1862045	1700395	7878588	465144	12303786	20000	19895	
	17	220865	133528	40194782	57793	10788406	20000	48	
	18	0	0	0	0	0	107012	0	
	19	885170	883109	946736	159347	10773912	20000	12500	
	20	867898	864374	1654654	561317	10878169	20000	19880	
	21	113463	113071	29870	54574	11528932	20000	0	
	22	38024865	36978618	23913945	1119911	10948937	20000	19900	
	23	0	0	0	0	0	106259	0	
	24	89522	88301	425618	37093	10783154	20000	0	
	25	139526	132856	2238516	73136	10703826	20000	0	
	26	15363052	15066885	19029958	284592	9716104	20000	12400	
	27	26047879	21651716	2225456	1111815	11095486	20000	17551	
	28	35712	35575	2194	20712	14357025	20000	0	
	29	0	0	0	0	0	255579	0	

Fonte: Autor, 2017.

Tabela 14 – Experimento sobre o conjunto de tarefas 0 utilizando BFD com ação da PMU.

Task Set	BFD com Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
0	1	6256	6209	325	3140	2544154	20000	0	100
	2	15533	15478	415	9205	7541252	20000	0	
	3	15130	14942	5303	8063	12121378	20000	0	
	4	5292	5257	285	1717	950554	20000	0	
	5	15600	15517	1078	6381	13135708	20000	0	
	6	8540	8462	1053	3107	2470018	20000	0	
	7	6194	6139	388	3008	2504541	20000	0	
	8	22876	22811	866	12764	13530496	20000	0	
	9	12354	12292	673	4885	5885345	20000	0	
	10	24376	23617	62243	9607	9919328	20000	0	
	11	40067	39958	1068	23936	5690284	20000	0	
	12	20255	20166	1468	12085	13902908	20000	0	
	13	24611	24526	997	16543	6271673	20000	0	
	14	0	0	0	0	0	0	0	
	15	27647	27522	3875	11381	8539270	20000	0	
	16	36586	36217	30772	13831	10196448	20000	0	
	17	25337	25005	18434	11992	10623992	20000	0	
	18	19696	19592	2145	12041	11428982	20000	0	
	19	0	0	0	0	0	0	0	
	20	135962	135507	26143	76871	15575458	20000	0	
	21	120604	119887	33800	44666	4512182	20000	0	
	22	29407	29331	1112	14336	6168148	20000	0	
	23	8192	8133	366	4891	3000689	20000	0	
	24	0	0	0	0	0	0	0	
	25	34058	33844	7922	15066	656268	20000	0	
	26	111251	111072	4862	66340	15231989	20000	0	
	27	110917	110523	14116	66056	4062175	20000	0	
	28	14767	14652	1244	6070	1159243	20000	0	
	29	0	0	0	0	0	0	0	

Fonte: Autor, 2017.

Tabela 15 – Experimento sobre o conjunto de tarefas 1 utilizando BFD com ação da PMU.

Task Set	BFD com Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
1	1	16332	16270	635	10147	10889919	20000	0	100
	2	6849	6782	551	2901	938701	20000	0	
	3	13050	12923	3627	5390	6617851	20000	0	
	4	8094	8035	388	3704	2317631	20000	0	
	5	13692	13626	449	7930	14535414	20000	0	
	6	15544	15413	1129	9153	14498735	20000	0	
	7	12891	12803	1194	8175	10335346	20000	0	
	8	18363	18250	1820	8725	1972442	20000	0	
	9	43055	42903	3633	12794	1257364	20000	0	
	10	18777	18682	1061	7089	5142668	20000	0	
	11	98408	98087	20262	43534	19025655	20000	0	
	12	64307	63762	81730	21065	11010199	20000	0	
	13	11689	11573	2437	4361	5069928	20000	0	
	14	0	0	0	0	0	0	0	
	15	26617	26495	2370	14756	8499274	20000	0	
	16	20507	20393	1865	10840	14855830	20000	0	
	17	58490	58079	54863	21365	19006092	20000	0	
	18	5710	5631	665	4125	637379	20000	0	
	19	0	0	0	0	0	0	0	
	20	132783	131965	62519	80356	18496922	20000	0	
	21	32587	32500	1524	12711	11981143	20000	0	
	22	19675	19590	1177	9851	18670741	20000	0	
	23	9048	8991	618	4844	2270878	20000	0	
	24	0	0	0	0	0	0	0	
	25	11520	11439	572	5908	9269502	20000	0	
	26	116962	116194	25500	66506	11980745	20000	0	
	27	24183	24103	753	12962	7772804	20000	0	
	28	28891	28666	4370	17202	4668959	20000	0	
	29	0	0	0	0	0	0	0	

Fonte: Autor, 2017.

Tabela 16 – Experimento sobre o conjunto de tarefas 2 utilizando BFD com ação da PMU.

Task Set	BFD com Ação da PMU								
	Thread ID	WCET	WCET_m	WCET_var	ET_m	ET_var	Número de Execuções	Deadlines Perdidos	WCET (Experimento)
2	1	23761	23652	2155	15174	18571776	20000	0	99
	2	13979	13884	2728	9128	7213603	20000	0	
	3	24379	24052	16217	10751	6435983	20000	0	
	4	12421	12373	566	8709	11761451	20000	0	
	5	11360	11243	1384	7089	10081235	20000	0	
	6	5120	5068	443	1578	1018600	20000	0	
	7	21367	21318	526	13767	15889987	20000	0	
	8	15563	15442	2870	9665	1122097	20000	0	
	9	35872	34980	474029	20129	5531699	20000	0	
	10	24882	24674	4236	11062	11430720	20000	0	
	11	22046	21925	1683	6508	1385501	20000	0	
	12	30521	30389	5759	21319	19717558	20000	0	
	13	0	0	0	0	0	0	0	
	14	11127	10994	1749	5345	8491687	20000	0	
	15	34943	34817	4553	26472	15687195	20000	0	
	16	53155	52343	258152	36238	11709848	20000	0	
	17	23755	23616	2516	12026	21246975	20000	0	
	18	0	0	0	0	0	0	0	
	19	26478	26383	2248	16953	11223351	20000	0	
	20	59419	58767	85176	36370	12622438	20000	0	
	21	16976	16876	1062	11265	11164806	20000	0	
	22	77943	77817	4938	52344	2456824	20000	0	
	23	0	0	0	0	0	0	0	
	24	17571	17489	622	8765	13145610	20000	0	
	25	29653	29306	40296	11808	13883319	20000	0	
	26	23647	23521	1448	14829	2746230	20000	0	
	27	81293	79866	819998	50873	16469121	20000	0	
	28	6633	6569	1650	2554	1304688	20000	0	
	29	0	0	0	0	0	0	0	

Fonte: Autor, 2017.