Um método de otimização da relação desempenho/consumo de energia para arquiteturas multi-cores heterogêneas em FPGA

Bruno de Abreu Silva

Bruno de Abreu Silva

Um método de otimização da relação desempenho/consumo de energia para arquiteturas multi-cores heterogêneas em FPGA

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Vanderlei Bonato

USP – São Carlos
Maio de 2016

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

S586u

Silva, Bruno de Abreu
Um método de otimização da relação desempenho/consumo de energia para arquiteturas multi-cores heterogêneas em FPGA / Bruno de Abreu Silva; orientador Vanderlei Bonato. -- São Carlos -- SP, 2016.
121 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2016.

1. Tese. 2. Consumo de energia. 3. Desempenho. 4. FPGA. 5. *Multi-cores* heterogêneos. I. Bonato, Vanderlei, orient. II. Título.

Bruno de Abreu Silva

A method to optimize performance/energy consumption relation for heterogeneous multi-core architectures on FPGA

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Vanderlei Bonato

USP – São Carlos
May 2016

AGRADECIMENTOS

O processo de doutoramento foi, de fato, o maior desafio intelectual o qual tive a coragem de enfrentar até o momento em que escrevo este documento. Evidentemente, seria impossível concluí-lo sem que diversas pessoas que tive o prazer de conhecer ajudassem de alguma forma a potencializar minha energia, criatividade e produtividade. Ao concluir, portanto, esta etapa de minha vida, tenho o dever de agradecer a essas pessoas. Por antecipação, peço desculpas àqueles que se sentirem injustiçados caso não encontrem o seu nome aqui. Saiba que, a despeito das falhas que a memória humana pode cometer, minha gratidão será eterna a todas as pessoas que fazem ou fizeram parte da minha história. Minha gratidão para:

- Deus, por sempre me dar forças para enfrentar os desafios inerentes à vida;
- Minha mãe, Neuza, guerreira que, além de me ensinar valores e princípios importantíssimos, sempre fez o impossível para me ver bem sem nunca pedir nada em troca;
- Meu pai, Gilson (*in memoriam*), que sempre foi a principal referência sobre como uma pessoa deve ser;
- Meu ”pai de criação”, Marcus, e minha irmã Alyne, por existirem;
- Minha noiva Natália e meu enteado Henrique, por tornarem a minha vida ainda mais especial;
- Minha família;
- Amigos e colegas, em especial, aos pertencentes ao Laboratório de Computação Reconfigurável (LCR), por todos os momentos de descontração como nos churrascos épicos, nos cafés evolutivos e no próprio laboratório e também por toda ajuda durante o desenvolvimento deste trabalho;
- Prof. Dr. Vanderlei Bonato, por ter me orientado e me ensinado muito, não só sobre o assunto deste trabalho, mas também sobre diversos outros temas em nossas conversas e reuniões;
- Professores Doutores membros da banca julgadora deste trabalho pelas contribuições e valiosas sugestões de melhoria;

- Todos os professores e funcionários do Instituto de Ciências Matemáticas e de Computação, da USP de São Carlos, e de outras instituições que contribuíram para minha formação acadêmica;
- Agradecimento especial à FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) que, por meio do processo 2011/10163-9, apoiou e viabilizou o desenvolvimento, a conclusão e a publicação dos resultados desta tese.

RESUMO

SILVA, B. DE A.. **Um método de otimização da relação desempenho/consumo de energia para arquiteturas multi-cores heterogêneas em FPGA**. 2016. 121 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Devido às tendências de crescimento da quantidade de dados processados e a crescente necessidade por computação de alto desempenho, mudanças significativas estão acontecendo no projeto de arquiteturas de computadores. Com isso, tem-se migrado do paradigma sequencial para o paralelo, com centenas ou milhares de núcleos de processamento em um mesmo *chip*. Dentro desse contexto, o gerenciamento de energia torna-se cada vez mais importante, principalmente em sistemas embarcados, que geralmente são alimentados por baterias. De acordo com a Lei de Moore, o desempenho de um processador dobra a cada 18 meses, porém a capacidade das baterias dobra somente a cada 10 anos. Esta situação provoca uma enorme lacuna, que pode ser amenizada com a utilização de arquiteturas *multi-cores* heterogêneas. Um desafio fundamental que permanece em aberto para estas arquiteturas é realizar a integração entre desenvolvimento de código embarcado, escalonamento e hardware para gerenciamento de energia. O objetivo geral deste trabalho de doutorado é investigar técnicas para otimização da relação desempenho/consumo de energia em arquiteturas *multi-cores* heterogêneas *single-ISA* implementadas em FPGA. Nesse sentido, buscou-se por soluções que obtivessem o melhor desempenho possível a um consumo de energia ótimo. Isto foi feito por meio da combinação de mineração de dados para a análise de softwares baseados em *threads* aliadas às técnicas tradicionais para gerenciamento de energia, como *way-shutdown* dinâmico, e uma nova política de escalonamento *heterogeneity-aware*. Como principais contribuições pode-se citar a combinação de técnicas de gerenciamento de energia em diversos níveis como o nível do hardware, do escalonamento e da compilação; e uma política de escalonamento integrada com uma arquitetura *multi-core* heterogênea em relação ao tamanho da memória *cache* L1.

Palavras-chave: Tese, Consumo de energia, Desempenho, FPGA, *Multi-cores* heterogêneos.

ABSTRACT

SILVA, B. DE A.. **Um método de otimização da relação desempenho/consumo de energia para arquiteturas multi-cores heterogêneas em FPGA**. 2016. 121 f. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Due to the growing need for high-performance computing along with higher volume of data to process, important changes are happening in computer architecture design. Parallel computing processors having hundreds or thousands of processing cores in a single chip are becoming a common solution, even for embedded systems. Power management becomes increasingly important, especially for mobile systems. A key challenge remaining open for these architectures is to perform the integration of application code, runtime scheduling and hardware control for power management. This thesis aims to present a method able to integrate these three aspects, by investigating techniques for optimizing performance versus power consumption in single-ISA heterogeneous multi-cores architectures implemented on FPGA. Our approach applies a data mining technique to analyze the application source-code, traditional techniques for power management, and an heterogeneity-aware scheduling policy. The main contributions are the combination of power management techniques at hardware, scheduling and compilation levels; a new scheduling policy along with a heterogeneous multi-core architecture relative to its L1 cache memory size determined offline and online.

Key-words: Thesis, Power consumption, Performance, FPGA, Heterogeneous multi-core architectures.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADN ..... Ãcido Desoxirribonucleico
AMCD ... *Asymmetric Multi-Core Design*
API ..... *Application Programming Interface*
CAMP ... *Comprehensive Scheduler for Asymmetric Multicore Processors*
CMOS ... *Complementary Metal-Oxide-Semiconductor*
CPLD ... *Complex Programmable Logic Device*
CPUs .... *Central Processing Units*
DAMICORE *Data Mining of Code Repositories*
DDR3 ... *Double Data Rate 3*
DFS ..... *Dynamic Frequency Scaling*
DPM .... *Dynamic Power Management*
DSPs .... *Digital Signal Processors*
DTM .... *Digital Thermal Management*
DVFS .... *Dynamic Voltage/Frequency Scaling*
DVS ..... *Dynamic Voltage Scaling*
ECC ..... Error-Correcting Code
ESHMP .. *Efficient and Scalable Heterogeneous Multi-core Processors*
FN ..... *Fast algorithm of Newman*
FPGAs .. *Field-Programmable Gate Arrays*
FPU ..... *Floating-Point Unit*
GCC ..... *GNU Compiler Collection*
GPL ..... *GNU Public License*
GPU ..... *Graphics Processing Unit*
HAL ..... *Hardware Abstraction Layer*
HASS .... *Heterogeneity-Aware Signature-Supported*
HCFs .... *Hardware Configuration Files*
HDTV ... *High-Definition Television*
HIF ..... *Hardware Information File*
HMP .... *Heterogeneous Multi-core Processor*
IoT ..... *Internet of Things*
IPC ..... *IPC-driven algorithm*

ISA ..... *Instruction Set Architecture*
LRR ..... *least-recently-replaced*
LRU ..... *least-recently-used*
MLQ .... *Multi-Level Queue*
MPSoC .. *Multi-processor System-on-Chip*
NCD ..... *Normalized Compression Distance*
NJ ..... *Neighbor Joining*
NoC ..... *Network-on-Chip*
PA ..... *Parallelism-Aware*
POSIX ... *Portable Operating System Interface*
QDR ..... *Quad Data Rate*
RTL ..... *Register-Transfer Level*
SECDED *Single Error Correction, Double Error Detection*
SIF ..... *Software Information File*
SMP ..... *Symmetric Multi-Processing*
SO ..... Sistema Operacional
SoC ..... System on-Chip
SRAM ... *Static Random-Access Memory*
SSRAM .. *Synchronous Static Random-Access Memory*
TLB ..... *Translation Lookaside Buffer*
TLP ..... *Thread-Level Parallelism*
TMR .... Triple Modular Redundancy
VHDL ... *Very High Speed Integrated Circuits Hardware Description Language*
VLSI ..... *Very Large Scale Integration*

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

Desde o final do século XX até o início do século XXI, o ganho de desempenho dos processadores ocorreu principalmente por meio do aumento da velocidade de chaveamento do transistor e dos avanços nos componentes da microarquitetura dos processadores. Porém, ao final da primeira década do século XXI, verificou-se uma diminuição da escalada por velocidade devido às limitações físicas da tecnologia criando novos desafios para a contínua busca por maior desempenho (fim da *Dennard Scaling* (EECKHOUT, 2015)). Como resultado, a frequência de *clock* dos processadores tem crescido mais lentamente, tendo o consumo de energia como um forte limitador do desempenho (BORKAR; CHIEN, 2011).

Por outro lado, muitas aplicações possuem demandas crescentes de poder computacional (KIRK; HWU, 2010). Por exemplo, codificação e manipulação de áudio e vídeo para o padrão *High-Definition Television* (HDTV); interfaces homem-máquina baseadas em voz e visão computacional; *touch screen* de alta resolução; aplicações para pesquisas biológicas em nível molecular, onde o aumento do desempenho é essencial para viabilizar a modelagem e simulação de grandes sistemas biológicos e, ainda assim, obter os resultados em tempo aceitável; simulação de dinâmica de fluidos; modelagem e previsão de condições climáticas; aplicações para as *cidades inteligentes* (*smart cities*) (BATTY *et al.*, 2012) e também para a Internet das Coisas, *Internet of Things* (IoT) (PERERA *et al.*, 2014; ATZORI; IERA; MORABITO, 2010).

Diante deste panorama, mudanças severas nas arquiteturas dos computadores têm ocorrido. Os projetos utilizam paralelismo em larga escala, *cores* heterogêneos e aceleradores para atingir maior desempenho e consumo de energia reduzido se comparados às abordagens de projeto tradicionais (BORKAR *et al.*, 2005; BORKAR; CHIEN, 2011). Muito esforço tem sido feito também para se desenvolverem pesquisas para exploração do espaço de projeto de arquiteturas heterogêneas e compilação de softwares para estas

arquiteturas (BELWAL; SUDARSHAN, 2015).

Porém, a integração de múltiplos *cores*, necessária em arquiteturas *multi-cores* ou *many-cores*, dificulta a manutenção de condições adequadas de temperatura e consumo de energia. A Figura 1 ilustra uma projeção de como seria alto o consumo de energia em arquiteturas *multi-cores* caso os projetistas de *chips* simplesmente adicionassem mais *cores* conforme a capacidade de integração se tornasse disponível e operasse o *chip* na frequência mais alta possível. Então, para que o consumo de energia permaneça dentro de limites aceitáveis, os projetistas devem limitar a frequência de funcionamento e o número de *cores*. Entretanto, isso limita severamente o aumento do desempenho dos microprocessadores.

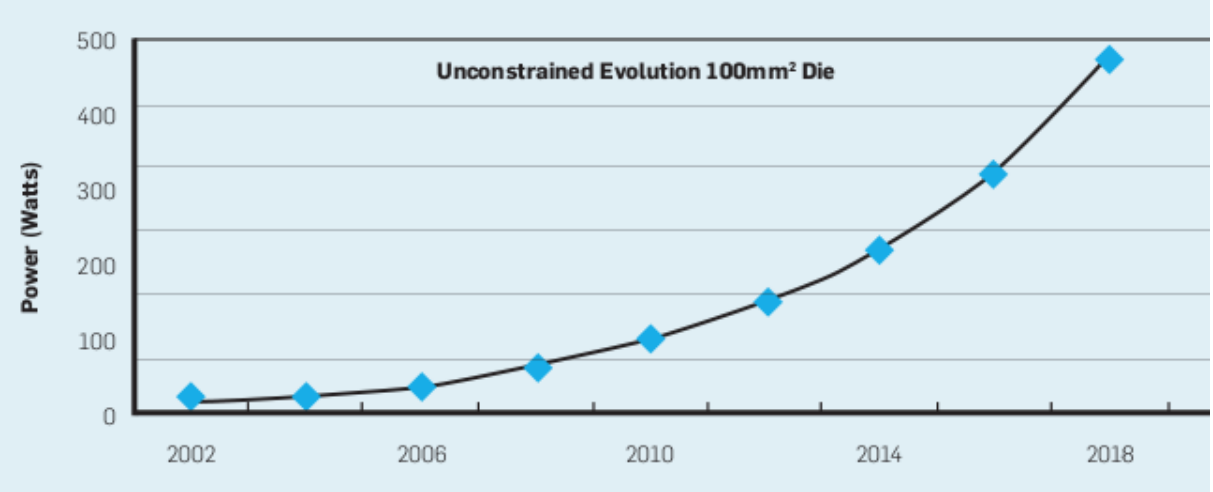

Figura 1 – Evolução irrestrita de um microprocessador resulta em consumo de energia excessivo (BORKAR; CHIEN, 2011)

A Figura 2 ilustra as principais lacunas (do inglês, *gap*) existentes na indústria de computadores comparando a diferença na evolução de seus principais componentes. O *power reduction gap*, acontece devido à diferença da taxa de crescimento do desempenho dos processadores, do desempenho da transmissão de dados e da quantidade de energia que uma bateria pode fornecer. De acordo com a lei de *Shannon*, o desempenho da transmissão de dados dobra a cada 8,5 meses. A lei de *Moore* diz que o desempenho dos processadores dobra a cada 18 meses. Entretanto, a capacidade das baterias dobra apenas a cada 10 anos. Consequentemente, para reduzir esta lacuna entre a capacidade das baterias e a capacidade de processamento, técnicas eficientes de gerenciamento de energia são essenciais, principalmente para sistemas embarcados que são comumente alimentados por baterias.

Algumas técnicas têm sido utilizadas para gerenciar o consumo de energia dos *chips*. *Digital Thermal Management* (DTM) é o conjunto de técnicas usado para evitar que tarefas executem em elementos de processamento localizados em regiões do *chip* com temperatura elevada. *Dynamic Power Management* (DPM) é uma estratégia em que o recurso gerenciador desliga os *cores* que estão ociosos ou ligam os *cores* desligados quando novas tarefas estão prontas para execução. *Dynamic Voltage/Frequency Scaling* (DVFS)

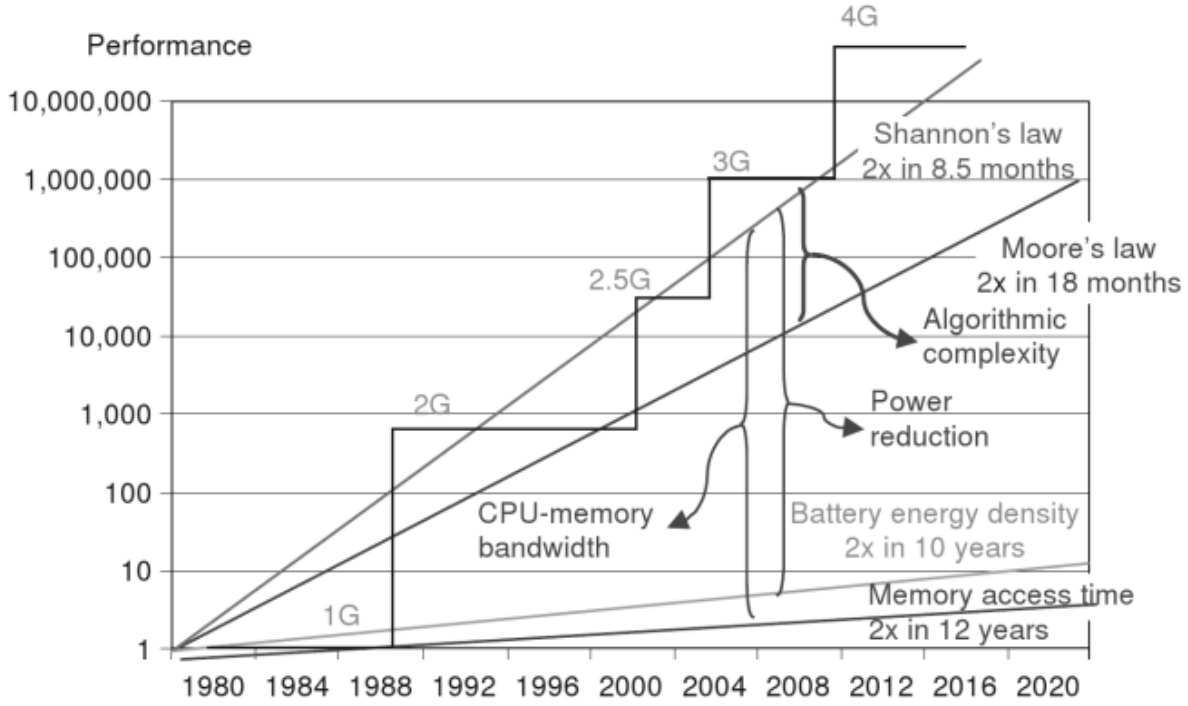

Figura 2 – Lacunas entre diferentes tecnologias (SHEARER, 2007)

é a fusão das técnicas *Dynamic Frequency Scaling* (DFS) e *Dynamic Voltage Scaling* (DVS). Como o nome sugere, DVFS permite que o hardware tenha a sua frequência e tensão de funcionamento alteradas dinamicamente com o objetivo de economizar energia. Pode-se citar o *Xeon* da Intel, *Crusoe da Transmeta*, processadores móveis da AMD como exemplos de processadores que utilizam DVFS. Diferentes aplicações possuem diversos requisitos em relação ao desempenho e consumo de energia. DVFS tenta otimizar essa relação enquanto o sistema está ocioso e previne também violações de temperatura durante o alto processamento. As tecnologias emergentes, como os *Field-Programmable Gate Arrays* (FPGAs), são restritas no aumento das margens de fornecimento de tensão, que é o componente chave por trás do DVFS. Portanto, novas técnicas são necessárias para otimizar a relação desempenho/consumo de energia nestas tecnologias. Algumas tendências de longo prazo (até 2030 aproximadamente) apontadas por Borkar e Chien (2011) para alcançar o consumo eficiente de energia em arquiteturas *multi-cores* são:

- *organização lógica*: seleção e escalonamento preditivo dos *cores* para otimizar o consumo de energia e minimizar o movimento dos dados, pois a energia gasta em movimento de dados é um fator crítico no desempenho atingível;

- *organização de memória*: a configuração da memória *cache* é também crítica para o desempenho e consumo de energia, pois é possível reduzir o consumo de energia e aumentar o desempenho consideravelmente dependendo da maneira como se implementa a estrutura das *caches* (JHENG; DUH; LAI, 2010; DING; WANG;

ZHANG, 2011);

- *circuitos*: aplicação de DVFS, circuitos com tensão de operação próxima do limiar e circuitos eficientes para gerenciamento de energia;

- *software*: novos algoritmos, linguagens, análises e técnicas que exploram características do hardware (o fim da escalada de desempenho em uma única *thread* já significa maiores desafios para o software; a mudança para o paralelismo criou um grande desafio para o software na história da computação (CATANZARO *et al.*, 2010; CHIEN, 2007). A demanda por consumo de energia eficiente tem levado ao desenvolvimento de pesquisas sobre uso de *cores* e aceleradores heterogêneos, aumentando ainda mais o desafio para a geração e compilação de software.

De acordo com essas tendências, para que o potencial de economia de energia se torne realidade, são necessários avanços em nível de aplicação, compiladores, ambientes de execução e sistemas operacionais para entender e predizer o comportamento das aplicações e da carga de trabalho (BIENIA *et al.*, 2008; FLINN; SATYANARAYANAN, 2004). Em relação aos sistemas operacionais especificamente, o escalonador desempenha um papel importante para atingir consumo de energia eficiente (SAEZ *et al.*, 2010a).

Alternativamente, pode-se utilizar arquiteturas reconfiguráveis, como por exemplo FPGAs, para implementar sistemas *multi-cores*. Dessa forma, é possível a implementação de *multi-cores* heterogêneos capazes de fornecer melhor relação desempenho/consumo de energia às aplicações. O potencial dos FPGAs nesse contexto consiste na possibilidade de se gerar arquiteturas heterogêneas que sejam customizadas para um dado conjunto de aplicações. *Softcores*, como o LEON3 (GAISLER, 2015), podem ser usados para implementar arquiteturas *multi-cores* em FPGA.

1.1 Objetivos

O objetivo geral deste trabalho é propor e avaliar um novo método para a otimização da relação desempenho/consumo de energia em arquiteturas *multi-cores* heterogêneas em FPGA que sejam específicas para um dado conjunto de aplicações. A heterogeneidade destas arquiteturas se dá principalmente por diferentes tamanhos de *cache* L1. Para otimizar a relação desempenho/consumo de energia nestas arquiteturas, serão combinadas técnicas aplicadas em tempo de compilação (*offline*) por meio da análise do código-fonte e *profiling* com técnicas aplicadas em tempo de execução (*online*), como mapeamento, escalonamento e redimensionamento dinâmico de *cache* - *dynamic way-shutdown*.

Os objetivos específicos são:

- integração vertical de diversos níveis de desenvolvimento: arquitetura, sistema operacional e compilação;
- integração de técnicas de otimização do consumo de energia realizadas em tempo de compilação (*offline*) e tempo de execução (*online*);
- integração de técnicas de otimização já conhecidas na literatura com técnicas novas propostas neste trabalho;
- implementação de uma política de escalonamento no Sistema Operacional (SO) utilizado para viabilizar a exploração das arquiteturas heterogêneas;
- implementação de um método para a análise *offline* do código da aplicação e gerar como saída os arquivos de configuração do hardware e mapeamento e escalonamento inicial do software;
- propor um método que seja pouco dependente do conhecimento de especialistas;
- propor um método que possa ser aplicado com relativa facilidade em áreas importantes para sistemas embarcados, como tolerância a falhas;
- propor um método que possa ser utilizado para arquiteturas *multi-cores* com a heterogeneidade representada por características além do tamanho da memória *cache* L1, como a presença ou não de Unidade de Ponto Flutuante, do inglês *Floating-Point Unit* (FPU).

1.2 Justificativa

As tendências no desenvolvimento das tecnologias dos processadores de um lado e as diversas novas aplicações que têm surgido acentuam o chamado *Power Wall* (ASANOVIC *et al.*, 2009) (ilustrado na Figura 2) e os desafios para enfrentá-lo. Aumentar a eficiência no consumo de energia é de vital importância para o futuro dos sistemas computacionais, sejam eles sistemas embarcados, centros de processamento ou mesmo *desktops* (EECKHOUT, 2015). Um desafio fundamental citado por Kornaros (2010) e que, segundo Eeckhout (2015), permanece em aberto para arquiteturas heterogêneas é realizar a integração entre desenvolvimento de código embarcado, escalonamento e hardware autônomo para gerenciamento de energia.

Considerando esta questão em aberto, neste trabalho é apresentado um método capaz de gerar mapeamento das aplicações em tempo de compilação para execução em arquiteturas *multi-cores* heterogêneas. Além disso, em tempo de execução serão realizadas otimizações por meio da aplicação de técnicas tradicionais aliadas a uma política de escalonamento que leva em consideração a heterogeneidade do sistema. A vantagem de se

combinar técnicas em tempo de compilação e em tempo de execução é a possibilidade de reduzir o consumo de energia se baseando tanto em informações do perfil do código quanto em informações sobre o comportamento do sistema perante um determinado conjunto de dados de entrada para uma dada aplicação. Na literatura, em geral os trabalhos realizam somente um dos tipos de otimização: ou *offline* quando realizada em tempo de compilação ou *online* quando realizada em tempo de execução.

Em relação à análise *offline* usada para a definição da configuração da arquitetura, mapeamento e escalonamento das aplicações, muitos métodos podem ser aplicados quando se conhece o comportamento de todas as aplicações. Entretanto, poucos métodos foram propostos para esta análise que podem ser usados mesmo quando não existe este conhecimento. O método apresentado neste trabalho é proposto com a intenção de ser usado em situações onde não se sabe o comportamento e os requisitos necessários para a execução de todas as aplicações.

Outra questão é relacionada às características exploradas para gerar arquiteturas *multi-cores* heterogêneas baseadas em um mesmo processador. Em muitos trabalhos presentes na literatura, o mais comum é explorar *cores* com diferentes frequências uma vez que a maioria das plataformas *multi-cores* existentes até o momento em que estes trabalhos foram desenvolvidos possuem microarquitetura homogênea (FEDOROVA *et al.*, 2009; BLAKE; DRESLINSKI; MUDGE, 2009). Com a utilização de *softcores* é possível explorar outras características para estas arquiteturas, como por exemplo, configuração de *cache*, que é uma memória que consome a maior parte da potência que alimenta o processador (ASADUZZAMAN, 2011), e unidade de ponto flutuante que também pode exercer influência no consumo de energia e no desempenho da arquitetura em função da aplicação.

Destaca-se a importância do estudo, pesquisa e desenvolvimento nesta área onde as arquiteturas, técnicas e ferramentas poderão ser utilizadas no desenvolvimento de sistemas embarcados em robôs móveis, sistemas críticos, de alto desempenho e de tempo real, além de celulares, *smartphones*, computadores embarcados, *netbooks*, *tablets*, simuladores de diversos tipos, entre outras aplicações.

1.3 Organização

O restante deste documento está organizado da seguinte maneira: no Capítulo 2 são apresentados os principais conceitos e estado da arte sobre arquiteturas *multi-cores* com enfoque maior para sistemas embarcados, como definições básicas, técnicas de otimização e gerenciamento do consumo de energia, exemplos de arquiteturas *multi-cores* para sistemas embarcados e políticas de escalonamento para *multi-cores* heterogêneos; o Capítulo 3 contém a descrição do método e dos materiais utilizados para se atingir os objetivos

deste trabalho; no Capítulo 4 a ferramenta proposta neste trabalho e as implementações realizadas são apresentadas; em seguida, no Capítulo 5, os experimentos realizados assim como seus resultados são apresentados; e por fim, a conclusão é apresentada no Capítulo 6 seguida pelas referências bibliográficas.

CAPÍTULO

2

REVISÃO BIBLIOGRÁFICA

Arquiteturas *multi-cores* integradas em um único *chip* - *System on-Chip* (SoC) - têm grande importância para uma ampla gama de aplicações que necessitam de alto desempenho. O uso de SoCs *multi-cores* é favorecido pelo fato de o custo de seu desenvolvimento ser, em geral, menor do que desenvolver um único processador com desempenho equivalente (KORNAROS, 2010). Isto acontece porque a complexidade de desenvolver vários processadores funcionando a uma frequência moderada é menor do que projetar um único processador com arquitetura mais complexa e que funcione a uma frequência muito mais alta.

Além disso, existe uma intensa busca por sistemas que utilizam de forma eficiente a quantidade de energia disponível (*energy-efficient systems*). Esta filosofia é semelhante aos *green systems* (ou *green computing*), que buscam minimizar o desperdício de energia para que, entre outras questões, ajudem a reduzir a poluição e degradação ambiental, emissões de carbono, etc. Seguindo essa filosofia, o projeto de *multi-core* permite um melhor gerenciamento do consumo de energia, pois, devido à modularização da arquitetura, facilita a implementação de diferentes circuitos para fornecimento de energia e diferentes domínios, onde cada domínio pode ser ativado, desativado ou ter sua frequência/tensão de operação alterada independentemente, de acordo com as necessidades da aplicação. Em um *smartphone*, por exemplo, o *core* responsável por acelerar o processamento gráfico dos jogos pode ser desativado durante uma ligação, tornando o sistema mais eficiente em relação ao uso da energia disponível.

Neste capítulo, serão apresentadas as definições de alguns conceitos importantes para o entendimento do trabalho e do contexto ao qual se insere e será apresentada uma visão geral das pesquisas relacionadas a arquiteturas *multi-cores* com um enfoque maior para sistemas embarcados e técnicas para otimizar a relação desempenho/consumo de energia nestas arquiteturas.

2.1 Definições

Existem diversas formas de se classificar as arquiteturas dos processadores e, no caso dos *multi-cores*, na literatura não há um consenso com relação à classificação. Alguns autores utilizam por exemplo a palavra híbrido para classificar sistemas que utilizam diversos processadores, como *Graphics Processing Unit* (GPU), integrados com processadores de propósito geral e *Digital Signal Processors* (DSPs).

Uma das possíveis classificações é com relação ao número de processadores. Outra é com relação à arquitetura, ou seja, ao conjunto de instruções que os processadores que compõem a arquitetura possuem. E uma terceira é com relação à microarquitetura dos processadores. A seguir será apresentada uma classificação baseada em Eeckhout (2015), Zhuravlev *et al.* (2013) e Kornaros (2010).

Assim, é possível classificar as arquiteturas de processadores em três tipos diferentes:

- *Single processor*: neste caso, a arquitetura possui somente um *core*, podendo também possuir instruções customizadas ou até mesmo um coprocessador;
- *Multi-cores* homogêneos *on-chip* ou também *Multi-processor System-on-Chip* (MPSoC) homogêneo: este tipo de arquitetura possui mais de um *core*, porém todos os *cores* são iguais;
- *Multi-cores* heterogêneos:
  - *Single-ISA*: Possuem o mesmo conjunto de instruções (*Instruction Set Architecture* (ISA)), mas diferem em outros aspectos como o número de estágios de *pipeline*, a habilidade para realizar predição de desvio, a configuração e tamanho da *cache*, a frequência de funcionamento dos *cores* e a implementação das instruções;
  - *Multi-ISA*: possuem diferentes conjuntos de instruções, vários processadores, hierarquias de memória e interfaces de comunicação irregulares além de possuírem DSPs e processadores de propósito específico. Neste tipo de arquitetura, o processo de compilação e particionamento hardware/software é mais complexo.

Como se pode perceber, em *multi-cores* heterogêneos *single-ISA*, os *cores* podem se diferenciar em relação ao desempenho, área ocupada e consumo de energia.

Algumas arquiteturas homogêneas podem implicitamente se tornarem heterogêneas quando, por exemplo, a heterogeneidade surge devido às variações no processo de fabricação dos *cores* (HUMENAY; TARJAN; SKADRON, 2007).

As arquiteturas *many-cores* se encaixam nas mesmas definições dos *multi-cores* apresentadas acima, porém os *many-cores* contém centenas ou até milhares de elementos

de processamento, enquanto os *multi-cores* possuem, no máximo, algumas dezenas de acordo com Borkar *et al.* (2005).

Com relação à memória, podemos classificar sua organização e estrutura de acordo com o mecanismo de compartilhamento de memória que pode ser centralizada e compartilhada, distribuída e compartilhada ou sem memória compartilhada (troca de mensagens). Uma característica presente nas aplicações para sistemas embarcados que trabalham com muitos dados é a grande quantidade de acessos à memória externa feitos por cada *core*. Assim, uma boa oportunidade para melhorar o desempenho e reduzir o consumo de energia é otimizar o acesso à memória em nível de aplicação. De acordo com (KORNAROS, 2010), o desenvolvimento de novas estratégias e técnicas é necessário para diminuir o número de acessos à memória.

Um grande número de redes de interconexão de memórias com processadores e de processadores com processadores foi explorado na literatura (HENNESSY; PATTERSON, 2011). Embora a interconexão baseada em barramento centralizado não seja eficiente quando existem muitos processadores, memórias e periféricos no sistema, este tipo de interconexão ainda é bastante utilizado. Isso acontece, pois em boa parte dos sistemas, a quantidade de processadores e periféricos normalmente é pequena. Para sistemas onde a quantidade de processadores e periféricos é maior, um tipo de interconexão que tem recebido muita atenção da comunidade científica é a *Network-on-Chip* (NoC). Várias arquiteturas para NoC foram propostas na literatura, como pode ser visto em Ivanov e Micheli (2005).

Projetar *multi-cores* que atendam satisfatoriamente os requisitos de todas as aplicações é uma tarefa difícil. Consequentemente, de acordo com Ishebabi e Bobda (2009), surgiu um interesse dos pesquisadores em usar FPGA para construir *multi-cores* específicos para determinados conjuntos de aplicações (BOBDA *et al.*, 2007; CHANG; WAWRZYNEK; BRODERSEN, 2005; SALDANA *et al.*, 2006).

Quatro etapas complexas estão envolvidas na implementação de *multi-cores* específicos para as aplicações:

- Projeto dos componentes que irão compor a arquitetura;
- Realizar o mapeamento da aplicação para os diversos *cores*;
- Integrar o sistema;
- Sintetizar a arquitetura para configurar o FPGA.

Em Ishebabi e Bobda (2009) é apresentado um método para síntese automatizada de arquiteturas para programas paralelos em sistemas *multi-cores* implementados com FPGA. Este método é baseado em otimização combinatória e são definidos a quantidade

de *cores*, a rede de interconexão, topologia e o mapeamento do software na arquitetura. Entretanto, o sistema trabalha somente em tempo de compilação e não trabalha no nível do escalonamento e não emprega técnicas para redução de consumo de energia em tempo de execução.

2.2 Gerenciamento e otimização do consumo de energia

Com a integração dos circuitos eletrônicos em *chips* cada vez menores devido à redução do tamanho e mudança da geometria do transistor, a quantidade de energia a ser dissipada por área tem aumentado. Algumas técnicas para otimizar o seu efeito têm ganhado força, como *clock gating*, otimização da árvore de *clock*, DPM e gerenciamento do sistema em tempo de execução.

O consumo de energia em um *chip* é dividido em estático e dinâmico. O consumo estático ocorre quando o transistor não está sendo chaveado, ou seja, não está ocorrendo computação. O consumo dinâmico ocorre quando os transistores mudam de estado entre aberto e fechado (chaveamento *on/off*), ou seja, quando o circuito está realizando computação.

De uma maneira um pouco mais detalhada, a energia dissipada por um circuito *Complementary Metal-Oxide-Semiconductor* (CMOS) pode ser representada pela seguinte equação:

$$E = E_{sw} + E_{leak} + E_{sc} \tag{2.1}$$

E_{sw} e E_{sc} representam o consumo de energia dinâmica causado pela carga ou descarga de componentes capacitivos (*sw - switching power*) e pela energia dissipada por causa dos curtos circuitos (*sc - short circuit*) que acontecem por pequenos períodos de tempo durante o chaveamento do circuito, respectivamente. E_{leak} é o consumo de energia estática que é dissipada devido à corrente de fuga do circuito (*leak - leakage current*).

As diferentes técnicas para otimização do consumo de energia tentam reduzir por meio de uma ou mais variáveis da Equação 2.1. A atividade de chaveamento dos transistores no circuito é responsável por aproximadamente 90% do consumo total em um *chip* (SCHMITZ; AL-HASHIMI; ELES, 2004). Sabe-se que uma maneira de reduzir o consumo de energia dinâmica é por meio da redução da tensão de alimentação do circuito (KORNAROS, 2010). Portanto, os *chips* estão sendo projetados para funcionar com tensões cada vez mais baixas. Porém, para que seja possível reduzir a tensão de alimentação, é necessário reduzir também a tensão de limiar do transistor para manter a alta velocidade de chaveamento do circuito (RABAEY; CHANDRAKASAN; NIKOLIC, 2003). O problema da redução da tensão de limiar do transistor é que ela gera um aumento

do consumo de energia estática. Deste modo, as técnicas que reduzem a energia estática também são importantes nos circuitos modernos.

Diversas técnicas para gerenciamento do consumo de energia (estática e/ou dinâmica) têm sido desenvolvidas ao longo dos anos e muitas são encontradas em dispositivos comerciais, como estas apresentadas a seguir.

2.2.1 Dynamic voltage/frequency scaling

A potência (P_{sw}) e a energia dissipada (E_{sw}) de um circuito devido à atividade de chaveamento de seus transistores por uma dada tarefa computacional pode ser representada pelas Equações 2.2 e 2.3 (SCHMITZ; AL-HASHIMI; ELES, 2004):

$$P_{sw} = \alpha \times C_L \times V_{dd}^2 \times f \tag{2.2}$$

$$E_{sw} = N_c \times \alpha \times C_L \times V_{dd}^2 \tag{2.3}$$

Onde N_c é o número de ciclos de *clock* para executar a tarefa, α é a atividade de chaveamento calculada ou estimada para cada circuito e tarefa em particular (também chamada de *toggle rate* - taxa de transições lógicas por unidade de tempo), C_L é a capacitância do circuito V_{dd} é a tensão de alimentação e f é a frequência de *clock*.

Observando as Equação 2.2 e 2.3, pode-se concluir que a redução da tensão causa uma redução quadrática no consumo de potência e de energia dinâmica. No caso da Equação 2.2, também pode-se observar que a frequência de *clock* influencia a potência de forma linear. Reduzir apenas a frequência não impacta em redução do consumo de energia dinâmica, mas um circuito que funciona em altas frequências, consumirá maior potência e trabalhará a uma temperatura mais elevada que, além de poder danificar o *chip*, é responsável pelo aumento do consumo de energia estática. Consequentemente, reduzir a frequência de *clock* de um circuito ajuda a reduzir a temperatura do *chip* e o consumo de energia estática.

Dynamic Voltage/Frequency Scaling (DVFS) foi introduzido na década de 90 (MAKEN *et al.*, 1990) com a promessa de reduzir consideravelmente o consumo de energia em sistemas digitais (processadores, bancos de memória, barramentos, etc) adaptando a tensão e a frequência dos sistemas de acordo com a mudança na carga de trabalho (ISCI *et al.*, 2006; SEMERARO *et al.*, 2002; SIMUNIC *et al.*, 2001; WU *et al.*, 2005). Os algoritmos para DVFS podem ser implementados em diversos níveis, como no nível da microarquitetura do processador (MARCULESCU, 2000), no escalonador do sistema operacional (ISHIHARA; YASUURA, 1998), ou até em algoritmos para compilação (HSU; KREMER, 2003; XIE; MARTONOSI; MALIK, 2003). Entretanto, DVFS tem sido limitado

pelos reguladores de tensão relativamente lentos que dificultam o ajuste de diferentes tensões em intervalos pequenos de tempo. E com o aumento da frequência de *clock*, fica cada vez mais difícil aplicar DVFS para obter economia de energia.

2.2.2 Clock gating

Clock gating é uma técnica que consiste em desabilitar os sinais de *clock* sempre que alguma lógica síncrona não está realizando computação (SHEARER, 2007). O objetivo é reduzir o consumo de energia dinâmica, pois cada vez que o sinal de *clock* muda, uma série de transistores chaveiam até que o circuito estabilize novamente.

Pode-se aplicar o *clock gating* em diferentes níveis: nível de registradores e *latches*; nível de unidade funcional, como em somadores e multiplicadores; nível de subsistema como bancos de *cache* ou várias unidades funcionais ligadas a uma mesma rede de distribuição; em nível de sistema, como por exemplo os processadores, onde o *clock* de cada processador em um *chip* pode ser desativado de forma independente, ou até mesmo nos casos onde um *chip* inteiro pode ter seu *clock* desativado em sistemas compostos por diversos *chips*.

O *overhead* associado à geração do sinal de ativação do *clock* deve ser considerado para garantir que a economia de energia irá ocorrer e este fato limita a granularidade do *clock gating*. Por exemplo, pode não ser vantajoso aplicar *clock gating* em um único registrador. Mas pode ser usado de uma forma mais agressiva, por exemplo, se em determinado momento da execução em uma unidade de inteiros, o inteiro não usa um conjunto de *bits*, pode-se economizar energia desabilitando todo esse conjunto.

2.2.3 Power gating

Power gating é uma técnica que objetiva eliminar a corrente de fuga, ou consumo de energia estática. Dessa forma, quando o dispositivo está ocioso (sem atividade de chaveamento) é possível desligá-lo para economizar energia.

Ao aplicar esta técnica, deve-se observar o *trade-off* entre a área ocupada pelo circuito responsável em realizar o *power gating*, a redução da corrente de fuga e o impacto no desempenho, pois uma vez desligado, o dispositivo pode demorar vários ciclos de *clock* para ser ligado novamente e voltar ao estado de operação. Para aplicar *power gating*, deve existir um controlador para decidir o que desligar e este controlador é projetado em conjunto com o circuito a ser controlado, normalmente na etapa *Register-Transfer Level* (RTL) do projeto (SHEARER, 2007).

2.2.4 Dynamic power management

Ainda de acordo com Shearer (2007), o princípio fundamental presente na aplicação do *Dynamic Power Management* (DPM) é colocar os recursos não usados ou ociosos em

estados de operação com níveis de desempenho reduzidos ou nulos. É feito dinamicamente de acordo com a carga de trabalho sob a qual o sistema está submetido.

Os dispositivos controlados pelo DPM podem ter seus estados alterados tanto por hardware quanto por software (ou ambos). O componente DPM é composto por:

- Um monitoramento da carga do sistema e seus recursos;
- Um controlador para enviar comandos para realizar as transições entre estados de funcionamento dos recursos do sistema;
- Uma política, que é um algoritmo de controle para decidir quando e como realizar as mudanças de estado (considerando as informações fornecidas pelo monitor).

Os algoritmos para DPM podem ser preditivos ou estocásticos. Os preditivos se baseiam em períodos anteriores de ociosidade e ocupação para predizer o tamanho de um período ocioso. O maior problema deste tipo de algoritmo é a baixa precisão se comparado aos algoritmos estocásticos. Os algoritmos estocásticos para geração e requisição de mudanças de estado dos dispositivos usam modelos estocásticos, como modelos semi-Markov, redes de Petri e modelos não-estacionários. Dessa forma, o gerenciamento de energia se torna um problema de otimização estocástico. As políticas estocásticas podem explorar os *trade-offs* entre economia de energia e desempenho, enquanto que as políticas preditivas não são tão robustas.

Esta técnica também pode incluir DVFS, pois alterar a frequência e tensão é equivalente a mudar de estado e muitas vezes é usado em conjunto com *clock gating* para conseguir maiores reduções no consumo de energia.

2.2.5 Smart caching

A memória *cache* possui uma influência significativa no consumo de energia. Em Powell *et al.* (2001), os autores utilizam as técnicas propostas anteriormente, *way prediction* (BATSON; VIJAYKUMAR, 2001; CALDER; GRUNWALD; EMER, 1996) e *selective direct mapping* (BATSON; VIJAYKUMAR, 2001) para reduzir o consumo de energia dinâmica da *cache* mantendo o desempenho alto. Outras técnicas como o *cache set prediction* e protocolos de coerência de *cache* são tipicamente pré-determinadas em tempo de projeto e não variáveis em tempo de execução. Mas estas técnicas podem ser usadas em conjunto com as técnicas realizadas em tempo de execução.

2.2.6 Considerações sobre as técnicas para redução do consumo de energia

De acordo com a tendência do avanço tecnológico e da necessidade das aplicações, segundo Borkar *et al.* (2005), os futuros processadores *multi-cores* serão compostos por centenas ou milhares de *cores* heterogêneos, alguns dos quais serão de propósito geral, alguns serão *cores* altamente especializados e alguns serão lógicas reconfiguráveis explorando paralelismo de granularidade fina, todos conectados a hierarquias de memórias *on-chip* por meio de redes *on-chip* de alta velocidade. Estes *cores* possuirão diferentes características de consumo de energia e desempenho. O desempenho e consumo de energia resultante é difícil de predizer devido ao comportamento imprevisível de parâmetros do sistema, como por exemplo, custo de comunicação, latência e faltas na memória, execução e ociosidade do *kernel*. Portanto, o monitoramento *online* de parâmetros do sistema e do gerenciamento dinâmico de energia se fazem necessários nos processadores *multi-cores*.

A temperatura também é um parâmetro importante a ser monitorado para reduzir a potência e o consumo da energia estática. Em Liu *et al.* (2007), técnicas para prevenir o aumento excessivo da temperatura, minimizar o custo de resfriamento e otimizar o desempenho do sistema por meio do uso de DVFS junto com o perfil térmico do sistema foram propostas. Foi formulado o problema da minimização da temperatura de pico da aplicação na presença de restrições de tempo real como um problema de programação não-linear. Entretanto, a otimização é estática e assume um conhecimento prévio sobre o perfil da tarefa. Técnicas para gerenciamento de energia e sensoriamento da temperatura estão sendo incorporadas nos processadores *multi-cores* emergentes.

2.3 *Multi-cores* heterogêneos *single-ISA*

Um processador *multi-core* heterogêneo - *Heterogeneous Multi-core Processor* (HMP) - é um conjunto de *cores* em que cada *core* possui diferentes características no que se refere ao desempenho, área ocupada e consumo de energia (NIE; DUAN, 2012). HMPs possuem grande potencial para obter maior desempenho e consumir menos energia do que processadores *multi-cores* homogêneos (HILL; MARTY, 2008).

Single-ISA HMPs também permitem a redução do consumo de energia (ZHURAVLEV *et al.*, 2013). Na literatura, este tipo de processador também costuma ser chamado de *Asymmetric Multi-Core Design* (AMCD) (ZHURAVLEV *et al.*, 2013; KUMAR *et al.*, 2004).

A maior vantagem de se utilizar o mesmo ISA é a possibilidade de se escalonar as tarefas para os *cores* mais adequados utilizando um único arquivo binário. Dessa forma, o processo de compilação se torna mais simples do que em *multi-cores* com diferentes ISAs,

mas ainda assim é possível explorar possibilidades de redução no consumo de energia e aumento do desempenho devido às diferenças existentes entre os *cores*. Ao explorar a heterogeneidade, é possível obter melhor desempenho se comparado aos *multi-cores* homogêneos (HILL; MARTY, 2008). Outra vantagem é que todos os *cores* podem executar todas as *threads*. Portanto, caso o *core* mais adequado para executar uma *thread* esteja ocupado, é possível escalonar esta *thread* para outro *core*, se assim for desejado.

Uma importante característica a ser considerada em HMPs para otimizar a relação entre desempenho e consumo de energia é o tamanho da memória *cache* dedicada a cada *core*, uma vez que memórias *cache* consomem a maior parte da potência que alimenta o processador (ASADUZZAMAN, 2011). O tamanho ideal da *cache* depende das características da aplicação. A quantidade exata de memória *cache* reduz o consumo de energia, entretanto uma *cache* muito pequena causa um excesso de *misses* e uma *cache* muito grande armazena muitos dados que possuem baixa taxa de reuso. Tanto a *cache* pequena quanto a grande são ineficientes na relação desempenho e consumo de energia. HMPs usando diferentes tamanhos de *cache* não têm sido largamente explorados por pesquisadores e nem pelos principais fabricantes de processadores *multi-cores* (BLAKE; DRESLINSKI; MUDGE, 2009).

A seguir serão apresentadas as principais políticas de escalonamento para AMCDs, uma vez que o escalonamento exerce um papel fundamental para explorar os benefícios deste tipo de arquitetura.

2.3.1 Políticas de escalonamento

Uma abordagem bem conhecida para explorar as vantagens dos *multi-cores* com arquiteturas homogêneas e microarquiteturas heterogêneas considera dois tipos de *cores*: rápido e lento. Durante uma execução, o sistema é monitorado continuamente para determinar a melhor atribuição das *threads* nos *cores*. No *IPC-driven algorithm* (IPC) (BECCHI; CROWLEY, 2006), isto é feito pelo cálculo de instruções por ciclo (IPC - *Instructions per cycle*) das *threads* em ambos os tipos de *core*. Depois disso, as *threads* que possuem uma maior taxa rápido-para-lento devem ser executadas nos *cores* rápidos permitindo atingir mais benefícios executando lá. A abordagem proposta em Kumar *et al.* (2004) é semelhante ao *IPC-driven algorithm*, exceto pelo método de amostragem que é mais robusto e a abordagem é focada em atribuições globalmente ótimas. A atribuição globalmente ótima amostra o desempenho de grupos de *threads* em vez de fazer decisões locais de troca de *threads*.

Usar estas políticas de escalonamento que fazem monitoramento *online* pode oferecer melhor desempenho do que as políticas que não consideram a heterogeneidade da microarquitetura. Embora estas abordagens demonstrem a vantagem dos *multi-cores* com microarquitetura heterogênea, eles foram avaliados somente em ambientes simulados

e alguns comportamentos não puderam ser observados devido à ausência de sistema operacional na simulação. Entretanto, em Saez *et al.* (2010b), é apresentada uma discussão sobre uma implementação real destes métodos e pode ser observado que o *overhead* associado à troca de *threads* para medir o IPC degrada o desempenho, afeta a escalabilidade e o balanceamento de carga do sistema.

Existe uma outra abordagem, a qual realiza *offline profiling* da carga de trabalho. Um exemplo de uma política de escalonamento que se encaixa nessa categoria é o *Heterogeneity-Aware Signature-Supported* (HASS) (SHELEPOV *et al.*, 2009). Nesta abordagem, a carga de trabalho é executada *offline* uma vez recebendo um dado conjunto de entrada representando o caso comum. A execução *offline* gera uma assinatura arquitetural, que é um resumo das características arquiteturais de uma aplicação. Em particular, no HASS, a assinatura arquitetural é uma tabela contendo as estimativas das taxas de *cache miss* em diferentes configurações de *cache*. Somente uma execução é suficiente para estimar a taxa de *cache miss* para cada arquitetura. Como a informação é coletada *offline*, não existe *overhead* de coletar a informação *online*. Entretanto, as mudanças dinâmicas de comportamento na execução da aplicação (fases da aplicação) não podem ser exploradas usando esta abordagem. Mesmo assim, políticas como HASS podem ser muito úteis porque elas são simples e escaláveis, embora possam ser menos precisas.

Em uma tentativa de obter escalabilidade e precisão ao mesmo tempo, a política *Efficient and Scalable Heterogeneous Multi-core Processors* (ESHMP) foi proposta (NIE; DUAN, 2012). O ESHMP elimina a necessidade de escalonar as *threads* em diferentes *cores* para estimar seu *speedup* relativo pelo cálculo do *Average Stall Time Per Instruction* (ASTPI).

Em vez de classificar as abordagens entre *online monitoring* ou *offline profiling*, pode-se dividir as abordagens baseado na especialização dos *cores*. *Efficiency specialization* garante que aplicações *CPU-intensive* executam nos *cores* rápidos, enquanto os *cores* lentos executam as aplicações *memory-intensive*. *Thread-Level Parallelism* (TLP) *specialization* garante que as *threads* (ou fases) sequenciais executam nos *cores* rápidos e os *cores* lentos são usados para executar *threads* (ou fases) paralelas da aplicação/carga de trabalho. Um exemplo desta especialização é o escalonador *Parallelism-Aware* (PA) (SAEZ *et al.*, 2010a). O primeiro escalonador a entregar *efficiency* e TLP *specialization* foi o *Comprehensive Scheduler for Asymmetric Multicore Processors* (CAMP) (SAEZ *et al.*, 2010b).

Todas as políticas descritas foram avaliadas em arquiteturas contendo somente frequências diferentes dos *cores*, pois na literatura ou se usa simulação ou processadores *multi-cores* com arquiteturas homogêneas (tais como Intel Xeon e AMD Opteron) e simulam a heterogeneidade utilizando a técnica DVFS. Arquiteturas heterogêneas *multi-cores single-ISA* com outras características diferentes, como por exemplo o tamanho das *caches*, também podem ser exploradas.

A Tabela 1 apresenta um comparativo entre essas políticas de escalonamento e suas principais características.

2.4 Considerações finais

Ao observar o estado da arte, algumas oportunidades podem ser identificadas. Uma delas é a integração de diferentes técnicas para gerenciamento do consumo de energia em uma mesma arquitetura. Mais especificamente, a maioria dos trabalhos utilizam *multi-cores* que possuem heterogeneidade somente com relação à frequência dos *cores*. Diversas outras características podem ser exploradas para gerar a heterogeneidade e obter melhorias. Além disso, nenhuma das políticas apresentadas, combinou o monitoramento *online* com o *offline profiling*. Alguns trabalhos sobre síntese automatizada de *multi-cores* específicos para a aplicação foram relacionados. Porém, estes trabalhos realizam otimizações somente em tempo de compilação, não trabalhando com o escalonamento. No entanto, a combinação da otimização em tempo de compilação com a otimização em tempo de execução por meio do escalonamento pode trazer melhorias significativas no consumo eficiente de energia.

Tabela 1 – Comparativo entre as políticas de escalonamento para *multi-cores* AMCD.

Políticas	**Princípio**	**Online ou offline**	**Especialização**	**Arquitetura e SO**	**Heterogeneidade**
IPC-driven por *thread* (BECCHI; CROWLEY, 2006)	calcula periodicamente IPC da *thread* fazendo *thread-swapping* nos *cores*	*online*	*eficiência*	simulação sem SO	frequência
IPC-driven por grupos de *threads* (KUMAR *et al.*, 2004)	calcula IPC por grupos e faz *thread-swapping* por grupos	*online*	*eficiência*	simulação sem SO	frequência
HASS (SHELEPOV *et al.*, 2009)	*workload* executado *offline* para gerar *architectural signature* (estimativa)	*offline*	*eficiência*	AMD Opteron Intel Xeon OpenSolaris	frequência
ESHMP (NIE; DUAN, 2012)	não executa as *threads* nos outros *cores*, calcula ASTPI	*online*	*eficiência*	AMD Opteron Intel Xeon Linux 2.6.21	frequência
PA (SAEZ *et al.*, 2010a)	verifica número de *threads* em execução	*online*	TLP	AMD Opteron OpenSolaris	frequência
CAMP (SAEZ *et al.*, 2010b)	introduz a métrica *Utility Factor* (UF) levando em conta as duas especializações: TLP e e *eficiência*	*online*	*eficiência* e TLP	AMD Opteron Intel Xeon OpenSolaris	frequência
Este trabalho	*architectural signature* Ranqueamento de Dominância DAMICORE Arquitetura *application-specific*	*offline* e *online*	*efficiency*	LEON3 FPGA e eCos	memória *cache* estática e dinâmica FPU

CAPÍTULO

3

MATERIAIS E MÉTODOS

Para atingir os objetivos geral e específicos deste trabalho, conforme enumerados no Capítulo 1, as seguintes atividades foram definidas como partes fundamentais do método proposto (detalhado no Capítulo 4):

1. Implementação de um processo de análise *offline* do código-fonte das aplicações que necessitasse de pouca informação sobre as aplicações e sobre as características que deveriam ser observadas nos códigos;

2. Implementação de parte de um sistema operacional capaz de receber dados resultantes da análise *offline* e que os utilizasse para auxiliar em sua decisão de escalonamento;

3. Implementação de uma arquitetura *multi-core* heterogênea *single-ISA* capaz de possuir *caches* de tamanhos diferentes, FPU e realizar *way-shutdown* dinâmico;

4. Utilização de uma placa robusta com capacidade suficiente de memória externa para viabilizar execução de grandes conjuntos de aplicações e com circuito medidor do consumo de potência.

A ferramenta *Data Mining of Code Repositories* (DAMICORE), descrita na Seção 3.1, foi utilizada como parte do processo descrito no item 1. Para o item 2, definiu-se o sistema operacional eCos, descrito na Seção 3.2, com as características mais desejáveis para este projeto. O processador LEON3, apresentado na Seção 3.3, e mais os outros componentes disponíveis em sua biblioteca foram usados para o item 3. E por fim, a placa Stratix IV GX foi definida para o item 4 e é descrita na Seção 3.4.

3.1 DAMICORE

A ferramenta DAMICORE é uma ferramenta para agrupamento de dados e reconhecimento de padrões desenvolvida pelo Professor Doutor Alexandre Cláudio Botazzo Delbem (Professor do ICMC – USP, pertencente ao mesmo laboratório de pesquisa do qual este projeto faz parte – Laboratório de Computação Reconfigurável) que foi utilizada em parceria neste projeto para agrupamento de aplicações com perfis de código semelhantes.

Para guiar a geração de uma configuração de hardware para a arquitetura HMP e para identificar os agrupamentos de aplicações adequados para cada *core*, foi usada uma abordagem para mineração de dados, chamada de DAMICORE. Neste caso específico, a entrada do DAMICORE é um conjunto de arquivos contendo o código-fonte das aplicações não necessitando a execução prévia de todas as aplicações. O DAMICORE consiste em uma combinação dos seguintes algoritmos: *Normalized Compression Distance* (NCD) (CILIBRASI; VITANYI, 2005) da Teoria da Informação, *Neighbor Joining* (NJ) da Filogenética (FELSENSTEIN, 2003) e *Fast algorithm of Newman* (FN) de Redes Complexas (NEWMAN, 2010). O NCD calcula a matriz de distância entre os códigos com independência do tamanho do código e da linguagem de programação. O NJ constrói uma árvore filogenética descrevendo os prováveis relacionamentos entre os códigos. Finalmente, o FN detecta comunidades nessa árvore (rede). Cada comunidade encontrada (*cluster*) corresponde a um conjunto de códigos similares, os quais são adequados para serem executados em um mesmo *core*.

Escolheu-se o uso do DAMICORE pela facilidade de uso e pela sua capacidade de agrupar códigos similares sem a necessidade de executá-los. As técnicas usadas pelo DAMICORE são baseadas nas pesquisas recentes nas áreas de Teoria da Informação resultando em um algoritmo capaz de agrupar grandes amostras de códigos sem a necessidade de definir características para agrupamento. Além disso, a ferramenta DAMICORE se mostrou promissora para identificar grupos de pequenas aplicações a fim de encontrar as mais adequadas para implementação em hardware (SANCHES; CARDOSO; DELBEM, 2011). Outros trabalhos que também utilizaram a ferramenta DAMICORE ou seus algoritmos internos, para diferentes propósitos, podem ser encontrados em: Moro (2015), Soares (2014) e Carvalho (2015).

Os maiores desafios enfrentados para a mineração de dados em códigos-fontes são: (i) grandes conjuntos de dados; (ii) grandes amostras; (iii) dados cujas características são desconhecidas. Para lidar com (i), a estratégia tem sido melhorar a eficiência dos algoritmos. O item (ii) pode ser enfrentado com técnicas para seleção de características. E para o item (iii), a *expertise* de profissionais da área sobre as características do problema podem ser usadas para definir as características que podem ser extraídas dos dados brutos. Uma vez que o conhecimento prévio de repositórios de códigos pode não estar disponível, as técnicas convencionais para análise de software não podem ser aplicadas com sucesso

neste contexto.

3.1.1 Normalized Compression Distance (NCD)

O NCD (CILIBRASI; VITANYI, 2005) pode ser usado para comprimir diversos tipos de dados. Trata-se de uma métrica que pode encontrar relacionamentos entre dados mesmo quando não é possível saber quais características podem ser usadas para representar adequadamente os objetos que estão sendo analisados. O princípio básico é que dois objetos x e y são considerados próximos se x pode ser significativamente comprimido usando a informação em y e *vice-versa*. Em outras palavras, se duas entidades são muito parecidas entre si, é mais fácil (e também mais eficiente) descrever a segunda entidade apenas apontando suas diferenças em relação à primeira. As propriedades desta compressão são as bases para teoria da Complexidade de Kolmogorov (LI; VITNYI, 2008). Entretanto, o cálculo da Complexidade de Kolmogorov é computacionalmente intratável. Cilibrasi e Vitanyi (2005) propuseram uma métrica baseada em compressão que aproxima a Complexidade de Kolmogorov em um tempo razoável. O NCD pode ser formalizado da seguinte forma (CILIBRASI; VITANYI, 2005):

$$NCD(x;y) = \frac{C(xy) - min\{C(x);C(y)\}}{max\{C(x);C(y)\}},$$

onde $C(x)$ é o comprimento da versão comprimida do arquivo x, y é outro arquivo e xy é o arquivo resultante da concatenação de x e y.

3.1.2 Neighbor Joining (NJ)

A Filogenia investiga o relacionamento entre objetos biológicos de diversas espécies, como Ácido Desoxirribonucleico (ADN) por exemplo. Os métodos filogenéticos constroem estruturas em árvore para descrever tais relações. Os dados podem ser discretos (como presença ou ausência de penas em animais) ou contínuos (como tamanho do crânio). Os métodos mais confiáveis para a reconstrução de árvores filogenéticas procura por árvores com maior verossimilhança em relação a modelos evolutivos existentes. Por outro lado, o *Neighbor Joining* usa modelos relativamente simples e pode construir rapidamente filogenias úteis e algumas vezes competitivas com filogenias obtidas pela maioria dos métodos confiáveis. Além disso, o NJ necessita apenas de uma matriz com as distâncias estimadas entre os objetos. Esta matriz viabiliza a generalização do uso do NJ para qualquer domínio de problema desde que a construção da árvore seja independente do domínio de onde a matriz foi estimada.

Outros métodos existem para a construção de agrupamentos hierárquicos a partir de uma matriz de distância, porém, o NJ enxerga cada ancestral (um nó interno de uma árvore) como a síntese de um *clado* (um agrupamento em potencial para objetos

biológicos). Então, o NJ recalcula a matriz de distância de uma forma que a divergência entre dois ancestrais relativa aos outros são descontadas das distâncias entre os objetos restantes. Esta característica é relevante quando o relacionamento entre os agrupamentos, não somente entre os objetos originais, é desejável.

A saída do NJ é uma árvore de relacionamentos e outro método deve ser aplicado para extrair os potenciais agrupamentos (*clusters*) - *clados* em Biologia - escondidos na topologia da árvore. Na área de Redes Complexas (NEWMAN, 2010), muitos algoritmos relevantes foram propostos para encontrar agrupamentos. Entre eles, o *Fast algorithm of Newman* tem sido utilizado para diversos contextos de redes em larga escala.

3.1.3 Fast algorithm of Newman (FN)

Uma vez encontrada a filogenia, deve-se verificar se existem clados (subárvores da árvore filogenética com alto grau de independência). Algoritmos da área de Redes Complexas focam em revelar comunidades (*clusters*) em redes grandes. Por exemplo, redes sociais e redes de transmissão e de distribuição elétrica são exemplos relevantes de áreas que aplicam algoritmos para detecção de comunidades. O FN pode lidar bem com este tipo de rede e opera adequadamente sobre dados arranjados em qualquer tipo de modelo de rede, possibilitando seu uso para árvores filogenéticas.

3.1.4 Mineração de dados

As comunidades encontradas pelo FN contem códigos, no caso deste trabalho, que apresentam maior similaridade entre si. Isto pode ser útil dependendo do problema a ser resolvido (por exemplo, aspectos de desempenho para um processador em uma arquitetura heterogênea).

A combinação de técnicas presente no DAMICORE também revela a hierarquia entre os agrupamentos permitindo que os agrupamentos mais próximos possam ser combinados para formar agrupamentos maiores. Consequentemente, os agrupamentos mais diferentes entre si, agrupamentos cuja união seja provavelmente inadequada, também podem ser determinados.

3.2 Sistema operacional eCos

O eCos (ECOS, 2016) é um sistema operacional *open source*, configurável e portável voltado para aplicações embarcadas de tempo real. Todos os aspectos do sistema são de livre acesso e podem ser examinados e modificados pelo usuário da maneira que for necessária de acordo com a *GNU Public License* (GPL). A principal vantagem do eCos

é a possibilidade de desenvolvedores criarem sistemas operacionais específicos para cada aplicação, eliminando assim qualquer funcionalidade não desejada no sistema.

Com relação ao suporte de aplicações *multi-threaded*, o *kernel* do eCos fornece as seguintes funcionalidades:

1. A habilidade de criar novas *threads* no sistema, seja durante a inicialização ou quando o sistema já está em execução;
2. Controle sobre as várias *threads* no sistema, por exemplo manipulando suas prioridades;
3. Mais de uma implementação de escalonador, determinando qual *thread* deve ser executada;
4. Um leque de primitivas de sincronização, permitindo que as *threads* interajam e compartilhem dados seguramente;
5. Integração com o suporte para interrupções e exceções do sistema.

Além da biblioteca de *threads* do próprio *kernel*, é possível utilizar um pacote de compatibilidade *Portable Operating System Interface* (POSIX) e programar em C usando a *Application Programming Interface* (API) Pthreads para programação *multi-threaded*. Esta característica é interessante, pois assim é possível utilizar códigos já existentes não sendo necessário utilizar a biblioteca de *threads* específica do eCos.

Diferente de outros sistemas, a alocação de memória e os *drivers* de dispositivos não são disponibilizados pelo *kernel*, mas sim por pacotes separados.

3.2.1 *Escalonador de* threads

A principal parte do *kernel* do eCos é o escalonador. Suas tarefas são selecionar a *thread* apropriada para execução, fornecer mecanismos para estas *threads* sincronizarem, e controlar o efeito das interrupções na execução das *threads*. Existe um contador dentro do escalonador que determina se o escalonador pode executar ou se ele está desativado. Se o contador é diferente de zero, ele está desativado, quando o contador retorna para zero, o escalonador volta a executar.

As políticas de escalonamento disponíveis para o eCos são: *Multi-Level Queue* (MLQ) e *bitmap*. O escalonador *bitmap* nada mais é do que uma fila de prioridades, onde as *threads* podem ter prioridade variando de 0 até 31, onde 0 é a maior prioridade. Deste modo, com essa política há um limite de 32 *threads* na fila e ela só pode ser usada em sistemas com um único processador. O escalonador MLQ permite a execução de várias *threads* em cada um de seus níveis de prioridade. O número de níveis de prioridade é uma

opção de configuração podendo variar de 0 a 31 onde 0 é a prioridade maior e 31 é a menor.

As prioridades são absolutas, ou seja, o *kernel* executará uma *thread* com prioridade menor somente se todas as *threads* de maior prioridade estiverem bloqueadas. O escalonador permite preempção entre os diferentes níveis de prioridade. Preempção é uma troca de contexto que pausa uma *thread* de baixa prioridade, permitindo que uma *thread* de maior prioridade execute.

O escalonador MLQ também permite *timeslicing* dentro de um nível de prioridade. *Timeslicing* permite que cada *thread* de uma dada prioridade execute por um período de tempo especificado por uma opção de configuração. A implementação da fila para o escalonador MLQ usa uma lista circular duplamente ligada para encadear as *threads* dentro de um nível de prioridade e *threads* de diferentes níveis de prioridade. O máximo de *threads* para o escalonador MLQ depende da quantidade de recursos de memória disponíveis.

3.2.2 Suporte a multi-processamento simétrico

Symmetric Multi-Processing (SMP) é uma arquitetura de computador que usa várias *Central Processing Units* (CPUs) para processar um programa. As CPUs compartilham um sistema operacional comum e um subsistema de memória. Isto permite que os processadores trabalhem juntos para compartilhar a carga de trabalho para fornecer mais desempenho do que um sistema com um único processador. O suporte a SMP é dividido em suporte no nível da *Hardware Abstraction Layer* (HAL) e no nível do *kernel*. SMP é suportado apenas com o escalonador MLQ. Algumas limitações para o hardware são impostas pelo suporte a SMP do eCos, incluindo:

- O número máximo de CPUs suportadas é oito, sendo que nos casos típicos são 2 ou 4 CPUs;
- O hardware deve fornecer um mecanismo de sincronização na forma de instruções *test-and-set* ou *compare-and-swap*. O *kernel* do eCos usa estas instruções de hardware para a implementação do *spinlock*;
- Ou não se usa memórias *caches*, e todos os processadores compartilham a mesma memória no sistema, ou o hardware mantem a coerência de *cache*. Isto previne que o eCos desempenhe operações de *flush* na *cache* entre cada acesso a memória;
- Todas as CPUs enxergam o mesmo espaço de endereçamento, podendo acessar toda a memória compartilhada;
- Todos os dispositivos são acessíveis por todas as CPUs;

- Um controlador de interrupção deve estar presente para rotear as interrupções para uma CPU específica;
- Para permitir que eventos em uma CPU façam um reescalonamento em outra CPU, um mecanismo é necessário para permitir que uma CPU no sistema interrompa outra CPU;
- O software que executa em uma CPU em particular, deve ser capaz de identificar em qual CPU ele está executando.

Para o LEON3, é disponibilizada uma versão do eCos com suporte a SMP na qual todos os requisitos listados acima já estão atendidos.

3.2.3 Inicialização do sistema

Inicialmente no eCos, é executado o *boot* da HAL e o *kernel* é inicializado por uma função chamada *cyg_start*, que é invocada ao final do *boot* da HAL. Esta função executa as rotinas de inicialização da plataforma, invoca os construtores e por fim chama a função *cyg_user_start*. Nesta função, o escalonador entra em execução e a aplicação do usuário inicia a sua execução.

A sequência de inicialização para sistemas SMP é um pouco diferente. Uma única CPU, chamada de primária, manipula a sequência de inicialização (*boot* da HAL e do eCos), enquanto as outras CPUs, chamadas de secundárias, permanecem suspensas. Depois que a função *cyg_scheduler_start* for chamada, as CPUs secundárias são inicializadas. A partir deste ponto, não há mais diferenciação entre as CPUs e as *threads* são atribuídas aos processadores de acordo com a política de escalonamento. E, nesse caso, o escalonador pode ser executado por qualquer *core*.

3.3 Processador LEON3

O LEON3 (GAISLER-A, 2015), disponibilizado pela Aeroflex Gaisler juntamente com uma biblioteca de componentes chamada GRLIB, é um modelo em *Very High Speed Integrated Circuits Hardware Description Language* (VHDL) de um processador 32 bits *open source* baseado na arquitetura SPARC V8 (SPARC, 1991). Projetado para aplicações embarcadas, combina alto desempenho, baixa complexidade e baixo consumo de energia. O LEON3 pode ser configurável por meio de parâmetros e é usado em projetos de SoCs. Algumas das principais características do processador LEON3 são:

- Mesmo conjunto de instruções do SPARC V8 com extensões V8e;
- *Pipeline* com sete estágios e predição de desvio;

- Unidades de multiplicação e divisão em hardware;
- Arquitetura Harvard com *cache* de instrução e dados separadas;
- *Cache* L1 parametrizável em tamanho e configuração;
- Suporte para *debug onchip*;
- Modo *power down* e *clock gating* (desligamento dos *cores* ociosos e de seus respectivos sinais de *clock*).

A Figura 3 apresenta o diagrama de bloco completo do processador. De acordo com a definição dos parâmetros, o processador pode conter menos funcionalidades do que apresentado nesta figura.

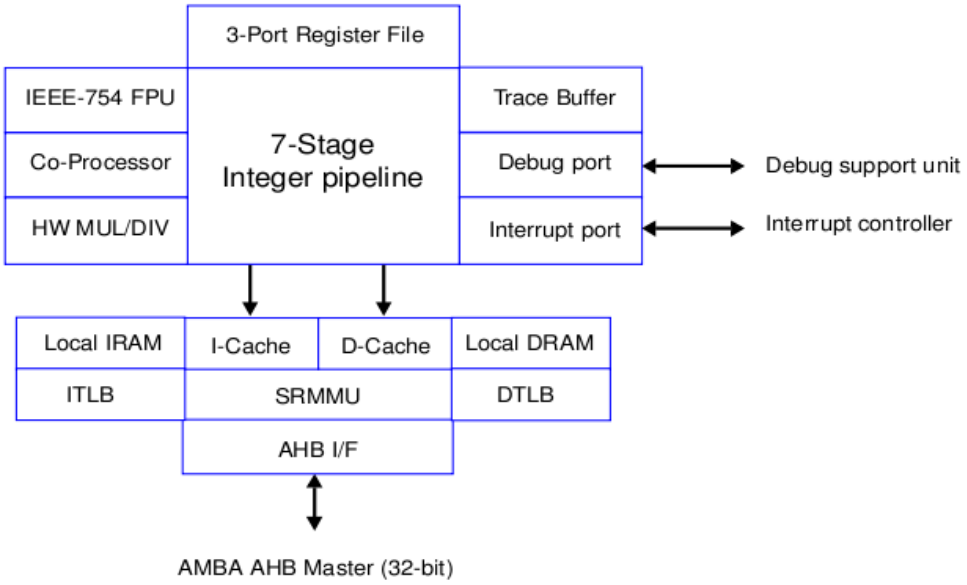

Figura 3 – Diagrama de bloco do processador LEON3 (GAISLER-B, 2015).

O sistema de *cache* do LEON3 é separado em *cache* de instruções e de dados e pode ser configurado em tempo de síntese do hardware. Ambas as *caches* podem ser configuradas com uma associatividade variando de 1 até 4 conjuntos, contendo de 1 até 256 kbytes por conjunto, 16 ou 32 bytes por linha. A *cache* de dados funciona de acordo com a política *write-through* e pode também realizar *bus-snooping* no barramento. Quando a associatividade é diferente do mapeamento direto (1 conjunto), a política de substituição dos dados nos conjuntos durante as leituras pode ser definida entre *random*, *least-recently-replaced* (LRR) e *least-recently-used* (LRU). O sistema de *cache* também implementa um AMBA AHB *master* para ler e escrever dados da *cache*. A interface está de acordo com o padrão AMBA-2.0.

Existe também uma interface para FPU. A própria Gaisler fornece uma unidade de ponto flutuante de alto desempenho chamada GRFPU. A FPU executa paralelamente

à unidade de inteiros e não bloqueia a operação a menos que haja uma dependência de dados entre as instruções em execução.

Opcionalmente, pode-se utilizar a unidade de gerenciamento de memória (SRMMU - SPARC V8 *Reference Memory Management Unit*). A SRMMU está de acordo com a especificação SPARC V8 MMU e fornece o mapeamento entre espaços de endereçamento virtuais de 32 bits e memória física de 36 bits. A MMU pode ser configurada com até 64 entradas em uma *Translation Lookaside Buffer* (TLB) *fully associative*.

O suporte ao *debug* é feito de forma não intrusiva no hardware. É possível atribuir *watchpoints* e verificar o conteúdo de todos os registradores do processador e das *caches*. O *trace buffer* interno pode monitorar e armazenar as instruções executadas, as quais podem ser lidas por meio da interface de *debug*.

Com relação às interrupções, o LEON3 suporta o modelo de interrupção do SPARC V8 com um total de 15 interrupções assíncronas. A interface de interrupção permite tanto gerar quanto receber interrupções.

O processador LEON3 implementa um modo *power-down*. Este modo paralisa o *pipeline* e a *cache* até a próxima interrupção. Esta característica é uma forma de minimizar o consumo de energia quando a aplicação está ociosa. Não é necessário nenhum suporte específico de ferramenta na forma de *clock gating*. Para implementar o *clock gating*, o processador possui um sinal de ativação do *clock*.

O suporte a multiprocessamento é feito atribuindo um *index* a cada processador. A política *write-through* das *caches* e o *snooping* garantem a coerência da memória em sistemas com memória compartilhada.

O LEON3 é configurável por meio de parâmetros (*VHDL generics*). Assim, é possível instanciar *cores* com diferentes configurações em um mesmo projeto. A atribuição de valores a estes parâmetros pode ser feita tanto pela atribuição direta no código VHDL ou por meio da ferramenta gráfica, também disponibilizada junto com a GRLIB. Nesta ferramenta, é possível configurar parâmetros não só do processador mas também dos periféricos *on-chip*, como controladores de memória e interfaces de rede. Os parâmetros que podem ser alterados e utilizados para atingir o objetivo deste trabalho são:

- Presença ou não de FPU;
- Parâmetros relacionados às *caches* de dados:
  - Política de substituição: LRU (*Least Recently Used*) ou LRR (*Least Recently Replaced*);
  - Número de conjuntos (*sets*) da *cache*, variando de 1 a 4;
  - Tamanho da linha da *cache*, em número de palavras, podendo ser 4 ou 8;

– Tamanho de cada conjunto da cache em kBytes, variando de 1 até 256.

A GRLIB foi projetada para construir sistemas centralizados em um barramento, ou seja, a maior parte dos componentes dos sistemas estarão conectados diretamente a um barramento AMBA2.0 - AHB/APB. Este barramento foi selecionado devido à sua ampla utilização no mercado, documentação disponível e pode ser utilizado gratuitamente sem restrições. Um exemplo de um sistema construído a partir da GRLIB pode ser visto na Figura 4.

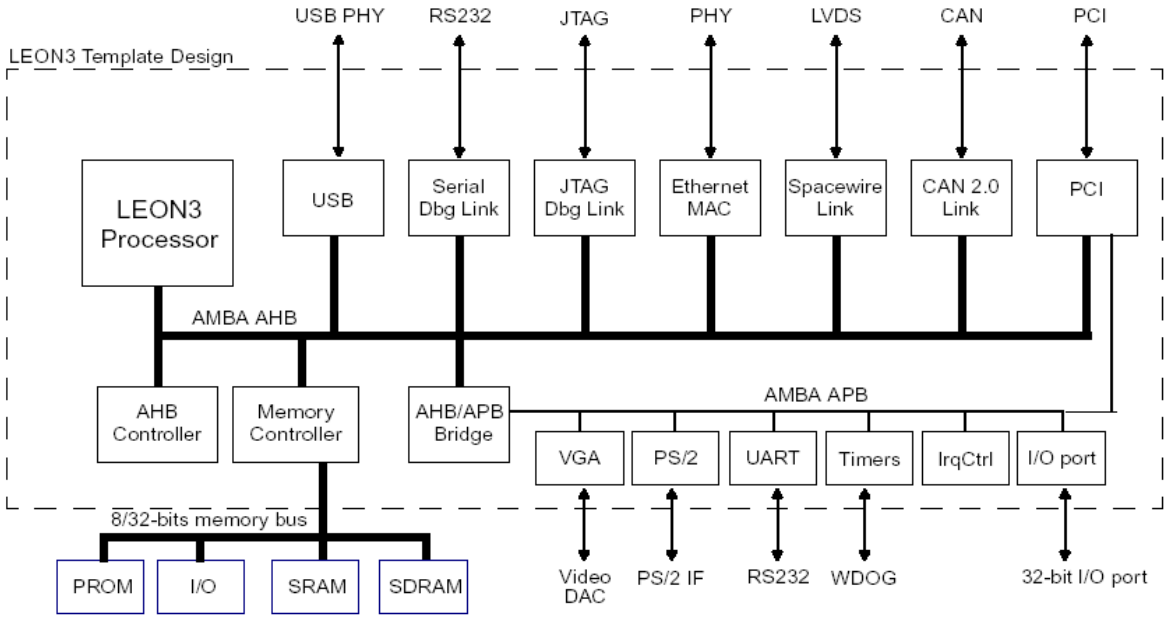

Figura 4 – Exemplo de um sistema com o LEON3 projetado com a GRLIB (GAISLER-A, 2015).

Após a construção dos arquivos que instanciam os componentes da arquitetura e definem os valores dos parâmetros, o hardware deve ser sintetizado. O sistema LEON3 pode ser sintetizado por meio de várias ferramentas de síntese, como Symplify, Synopsis DC, Mentor Precision, Xilinx XST, Altera Quartus II e Cadence RC.

Como o LEON3 possui o mesmo conjunto de instruções da arquitetura SPARC V8, os compiladores e *kernels* desenvolvidos para o SPARC V8 também podem ser usados para o LEON3. A Aeroflex Gaisler fornece o BCC, que é um compilador livre para C/C++ baseado no *GNU Compiler Collection* (GCC). No caso de aplicações *multi-threaded* com um ou mais processadores, é possível utilizar o *kernel* eCos para sistemas de tempo real. É possível instanciar até 16 processadores conectados ao barramento AMBA utilizando a GRLIB (GAISLER-A, 2015).

Demais características como o suporte ao Linux e *debugger* por meio do *gdb debugger* podem ser encontrados em Gaisler-a (2015).

3.4 Stratix IV GX Development Board

A placa de desenvolvimento *Stratix IV GX* da *Altera Corporation* foi utilizada neste trabalho. A placa fornece uma plataforma de hardware reconfigurável por meio de um FPGA da família Stratix IV, que permite o desenvolvimento e a prototipação de projetos complexos com um único *chip*. Além disso, possui memória externa e periféricos. As características mais importantes da placa para a execução deste trabalho são (ALTERA, 2012):

- FPGA Stratix IV EP4SGX230KF40;
- Controle do sistema por meio de um MAX II *Complex Programmable Logic Device* (CPLD) EPM2210;
- 2 *chips* de memória *Double Data Rate 3* (DDR3), com 512 MB e 128 KB, além de dois *chips* de 2 MB de memória QDRII+ (*Quad Data Rate* (QDR)) *Static Random-Access Memory* (SRAM), 64 MB de memória *flash* e 2 MB de *Synchronous Static Random-Access Memory* (SSRAM);
- Circuito para medição de potência *on-board*.

3.4.1 Controlador do sistema MAX II CPLD EPM2210

O MAX II CPLD EPM2210 é utilizado para os seguintes propósitos na placa *Stratix IV GX*, entre outros:

- Configuração do FPGA a partir da memória *flash*;
- Monitoramento da potência;
- Monitoramento da temperatura;
- Controle da ventoinha presente na placa para dissipação do calor.

A Figura 5 ilustra o diagrama de bloco do circuito que já vem configurado de fábrica no MAX II CPLD EPM2210 cujo projeto está disponível no site do fabricante.

O circuito medidor de potência presente na placa pode ser visto na Figura 6.

3.5 Considerações finais

Neste capítulo foram apresentadas inicialmente as atividades definidas como parte do método proposto neste trabalho para atingir os objetivos do mesmo. Em seguida, os materiais e ferramentas utilizados foram apresentados, evidenciando-se as características

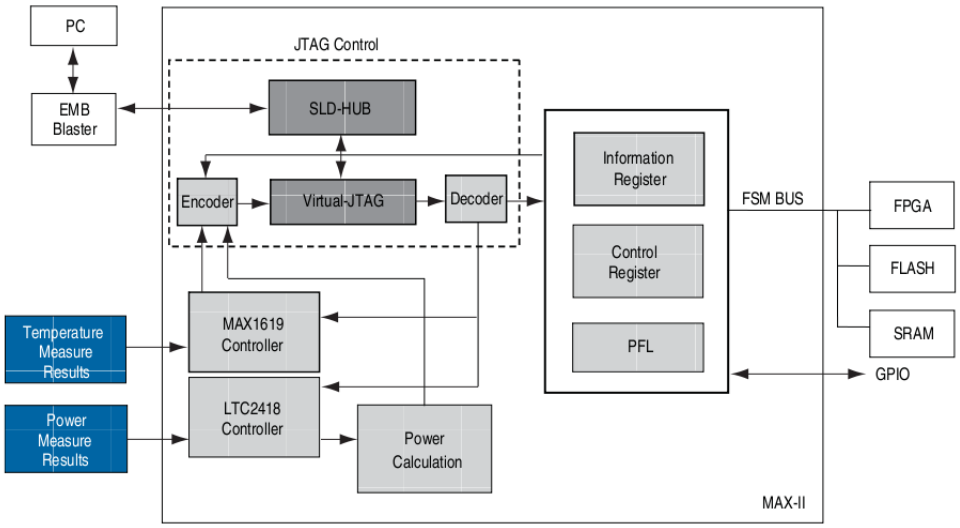

Figura 5 – Diagrama de bloco do CPLD controlador de sistema MAXII EPM2210 (ALTERA, 2012).

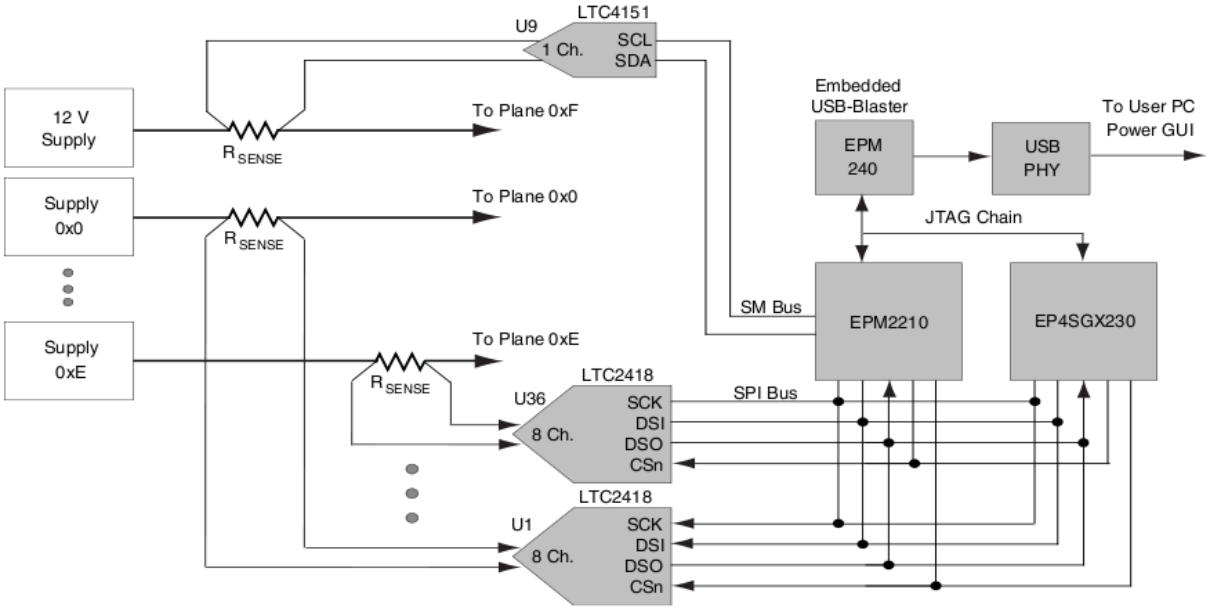

Figura 6 – Circuito medidor de potência (ALTERA, 2012).

mais relevantes em relação às necessidades deste trabalho. No próximo capítulo, é descrito o fluxo da ferramenta proposta para implementação do método, além de como os materiais e demais ferramentas citados neste capítulo foram adaptados e integrados neste trabalho.

CAPÍTULO

4

IMPLEMENTAÇÃO

A Figura 7 apresenta a ferramenta proposta neste trabalho de doutorado. Os blocos na cor cinza escuro foram implementados por completo, os blocos na cor cinza claro são blocos já existentes, mas que foram modificados para se adequarem às necessidades do trabalho e os blocos brancos já estavam implementados e foram utilizados sem modificações. O fluxo da ferramenta se divide em tempo de compilação (análise *offline*) e tempo de execução (monitoramento *online*). A parte *online* ocorre após a arquitetura ter sido definida e carregada na placa juntamente com o software a ser executado. Tudo que está dentro do retângulo que representa a *Stratix IV GX Development Board* pertence à parte *online*.

As subseções seguintes irão detalhar cada um dos blocos em cinza claro e cinza escuro, mostrando qual sua função, como foi implementado ou modificado e quais as dificuldades encontradas.

4.1 *Threads* de referência

As *threads* de referência (ou aplicações de referência)[1] são aplicações cujo comportamento já é previamente conhecido. Ou seja, cada aplicação de referência possui construída a sua assinatura arquitetural, contendo informações sobre sua taxa de *cache miss* e *hit*, consumo de energia e tempo de execução quando se variam as características da arquitetura. As aplicações de referência devem representar uma amostra representativa das características do conjunto total de aplicações. Por exemplo, no caso das taxas de *cache miss*, é interessante que as aplicações de referência apresentem diferentes taxas, variando seu comportamento entre *CPU-Intensive* e *Memory-Intensive*. São chamadas de aplicações de referência, pois servem para guiar o processo de geração da arquitetura

[1] Neste trabalho, as palavras *thread* e *aplicação* são usadas com o mesmo sentido, pois o sistema operacional utilizado implementa o paralelismo por meio de *multithreads* e cada *thread* neste trabalho implementa uma aplicação diferente, não possuindo dependências entre as demais *threads*.

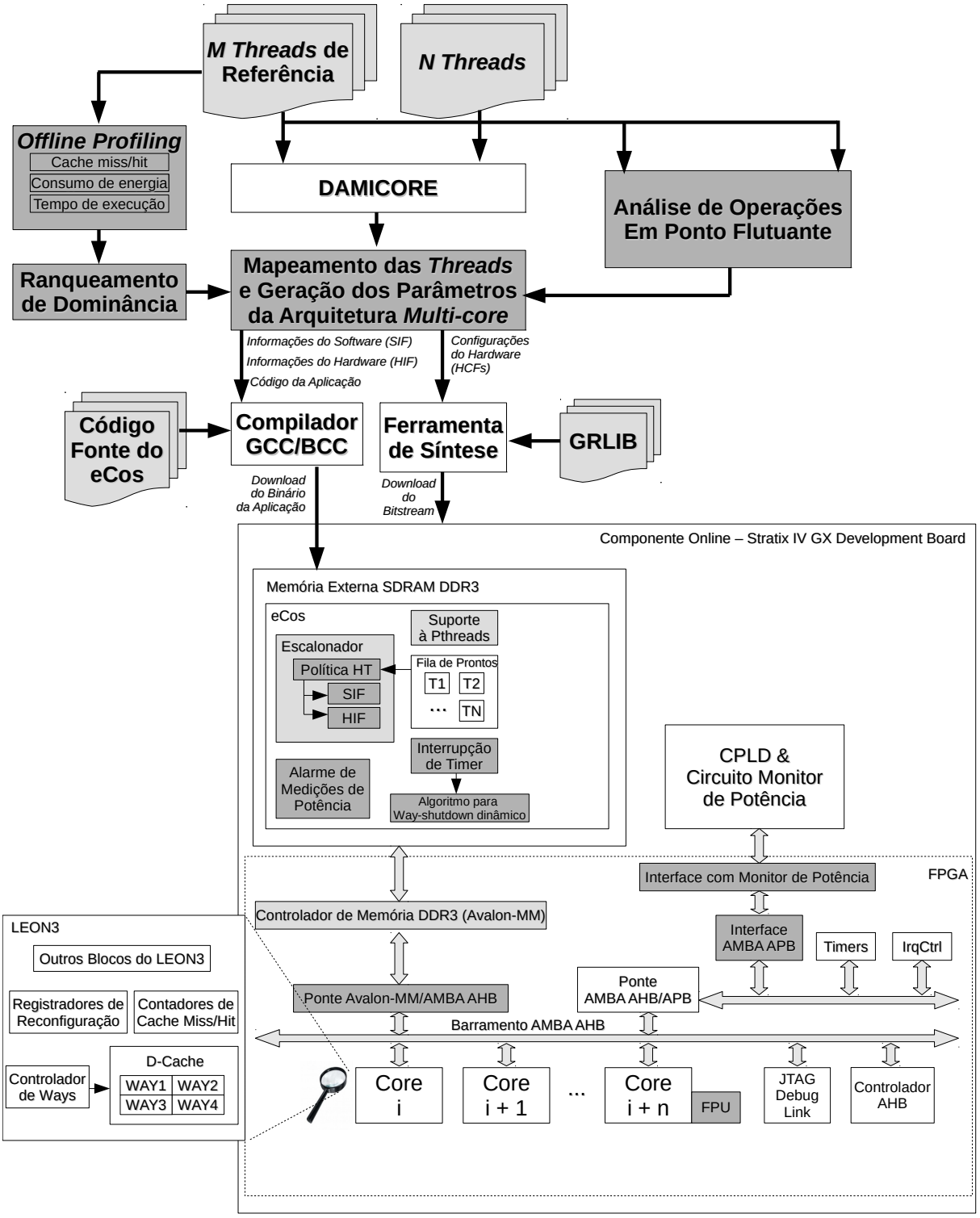

Figura 7 – Visão geral da ferramenta proposta. Em cinza claro estão os componentes que foram modificados, em cinza escuro estão os componentes que foram implementados por completo neste trabalho e os blocos brancos já estavam implementados e foram utilizados sem modificações.

multi-core heterogênea e evitam que seja necessário possuir a assinatura arquitetural de todas as aplicações que serão executadas na arquitetura. Nos experimentos, o conjunto foi composto por dez aplicações implementando algoritmos clássicos de ordenação, cálculos

matemáticos e manipulação de vetores e matrizes.

Como a principal técnica para realização da análise *offline* utilizada neste trabalho se baseia apenas no código-fonte de cada aplicação, dois fatores importantes foram considerados no desenvolvimento destes códigos: os códigos foram implementados na linguagem C e seguem uma padronização rigorosa em sua estrutura como declaração de variáveis, nomes de variáveis para que a diferença mais evidente no código-fonte seja em relação ao algoritmo em si; o tamanho dos dados manipulados pelas diferentes aplicações são, sempre que possível, próximos, se não forem iguais. Esse ajuste no tamanho dos dados foi necessário uma vez que a técnica proposta não explora o tamanho dos dados em sua análise. Dessa forma, por exemplo, um código de um filtro de imagens que utiliza como base de sua iteração uma janela 3x3 seria considerado igual (ou muito semelhante) a um código também do mesmo filtro, porém com uma janela 1000x1000, embora a necessidade de *cache* entre estas duas versões possa ser bastante distinta. Como esta diferença entre o tamanho dos dados manipulados foi reduzida ao máximo, seu impacto nos resultados também foi minimizado.

4.2 *Threads* não classificadas

As outras *threads* (aplicações) passadas como entrada para a ferramenta são aplicações em que não se tem nenhuma informação sobre seu comportamento em relação à taxa de *cache miss*, melhorias com uso de FPU, consumo de energia e tempo de execução. Foram selecionados 40 algoritmos pertencentes aos códigos da Texas Instruments (TMS320C64x Image/Video Processing Library e TMS320C64x DSP Library), além de algoritmos de ordenação e de cálculos matemáticos. Tais algoritmos foram selecionados por estarem presentes em diversas aplicações no contexto de sistemas embarcados, processamento de sinais, imagem e vídeo. Estas aplicações seguem o mesmo padrão definido para as aplicações de referência uma vez que também são submetidas ao processo de análise de código-fonte *offline*.

4.3 Offline profiling

O *offline profiling* é o processo ao qual as aplicações de referência são submetidas a fim de que se obtenha, por meio de execução real, seus perfis de comportamento em tempo de execução, caso estes ainda não estejam em mãos. A arquitetura básica utilizada para obter a taxa de *cache miss* e *hit*, melhoria no uso de FPU, consumo de energia e tempo de execução de uma aplicação pode ser vista na Figura 8. É uma arquitetura semelhante à apresentada na Figura 7, porém é uma arquitetura *single-core*, sem sistema operacional e que realiza o monitoramento do consumo de energia por meio de interrupção de *timer*. Esta interrupção ocorre a cada 140 milissegundos, pois este é o tempo necessário para que

ocorra uma leitura de potência instantânea feita pelo CPLD. Portanto, a cada 140ms, a rotina que trata a interrupção salva o valor da potência consumida naquele instante. Ao final da execução, é armazenado o tempo total de execução e com isso é possível fazer o cálculo da energia total consumida. Os valores de *cache miss* e *hit* são também acessados ao final da execução.

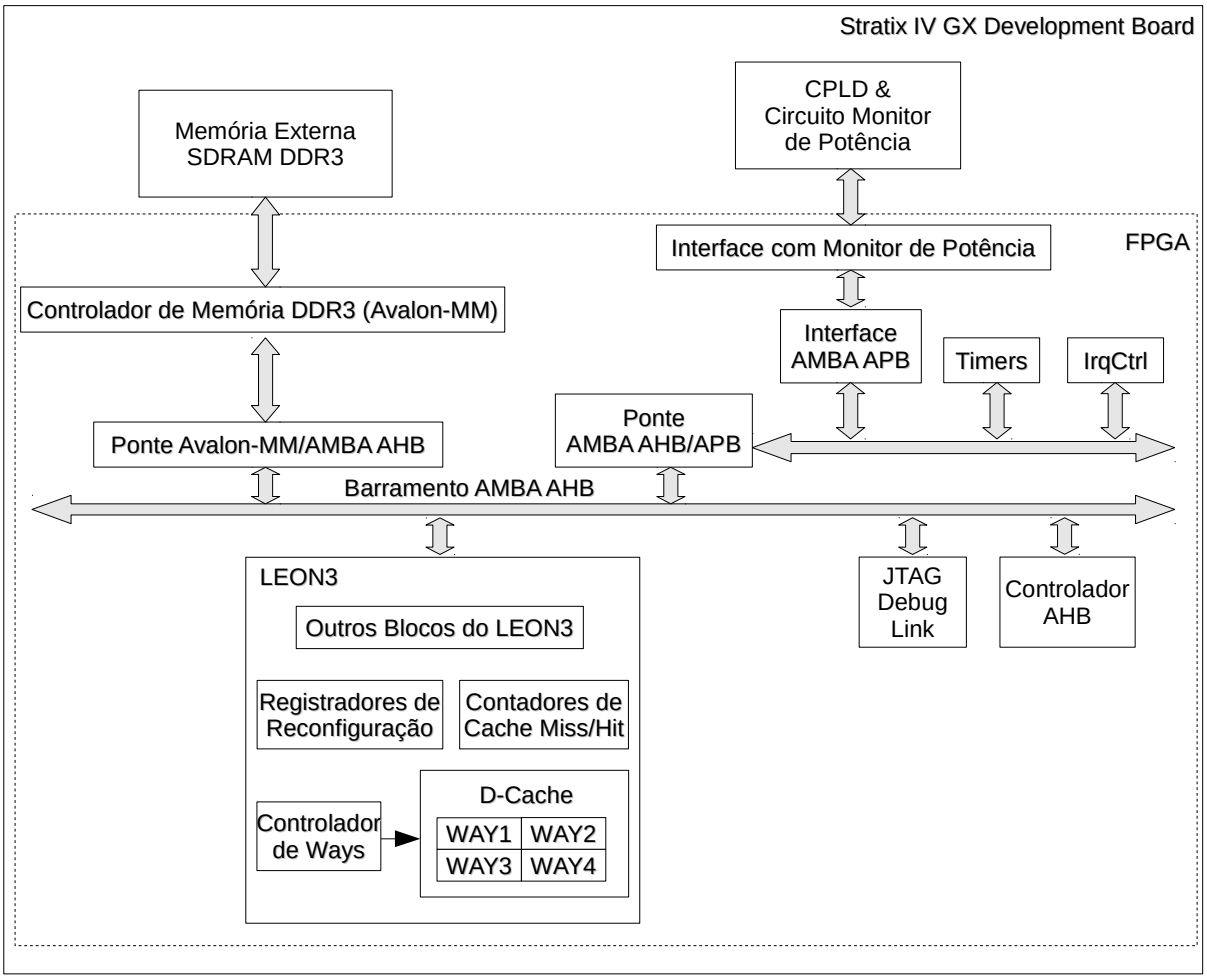

Figura 8 – Arquitetura *single-core* utilizada para realizar o *profiling* das aplicações de referência.

Como, para cada *core* contendo quatro *ways*, é possível configurar o tamanho total da *cache* em 4KB, 8KB, 16KB, 32KB, 64KB, 128KB e 256KB, sete arquiteturas *single-core* foram sintetizadas em FPGA para realizar as execuções das aplicações de referência. A Tabela 2, a Tabela 3 e a Tabela 4 mostram como exemplo os resultados do *offline profiling* para as aplicações de referência no caso da memória *cache*.

4.4 Ranqueamento de dominância

O Ranqueamento de dominância (FERREIRA, 2012) é um ranqueamento para comparação quantitativa de soluções em que, no caso particular deste trabalho, para cada critério uma arquitetura é comparada com todas as outras, par-a-par, e os ganhos e perdas

Tabela 2 – Taxa de *cache miss* para as aplicações de referência nas configurações de cache possíveis.

	Miss Rate (%)						
	4KB	8KB	16KB	32KB	64KB	128KB	256KB
autocor	25,05	25,05	23,02	0,62	0,62	0,62	0,62
bubble_sort	4,99	1,15	0,04	0,04	0,04	0,04	0,04
dotprod	99,97	99,97	99,97	99,97	91,18	3,81	3,82
fibonacci	37,53	37,54	37,55	37,55	26,89	0,43	0,43
matrix_mult	99,98	99,98	91,08	50,22	50,22	50,17	50,15
max	99,98	99,98	99,98	91,05	0,51	0,51	0,51
pi	7,45	7,56	7,59	7,59	8,17	9,19	9,73
vecsum	99,98	99,98	99,98	99,98	91,48	0,74	0,74
linear	5,7	5,34	4,49	2,27	0,84	0,84	0,84
jas_image_setbbox	46,18	46,18	46,18	46,18	1,51	1,27	1,27

Tabela 3 – Tempo de execução para as aplicações de referência nas configurações de cache possíveis.

	Tempo de execução (segundos)						
	4KB	8KB	16KB	32KB	64KB	128KB	256KB
autocor	4,2	4,2	4,1	3,4	3,4	3,4	3,4
bubble_sort	8,3	6,9	6,5	6,5	6,5	6,5	6,5
dotprod	8,4	8,4	8,4	8,4	8,1	5,3	5,3
fibonacci	3,6	3,6	3,6	3,6	3,1	1,6	1,6
matrix_mult	8,4	8,4	8,1	6,3	6,3	6,3	6,3
max	2,3	2,3	2,3	2,2	0,7	0,7	0,7
pi	0,7	0,7	0,7	0,7	0,7	0,7	0,7
vecsum	4,6	4,6	4,6	4,6	4,3	1,0	1,0
linear	1,0	1,0	1,0	1,0	0,9	0,9	0,9
jas_image_set	6,3	6,3	6,3	6,3	3,8	3,8	3,8

Tabela 4 – Consumo de energia para as aplicação de referência nas configurações de cache possíveis.

	Consumo de energia (Joules)						
	4KB	8KB	16KB	32KB	64KB	128KB	256KB
autocor	11,327	11,286	11,161	9,141	9,297	9,573	10,297
bubble_sort	23,321	19,371	18,003	18,359	18,857	19,782	21,365
dotprod	23,259	23,067	23,24	23,577	23,245	15,567	16,877
fibonacci	10,202	10,087	10,157	10,257	8,787	4,742	5,055
matrix_mult	23,5131	23,3585	22,4325	17,4753	17,8329	18,4769	19,7601
max	6,557	6,48	6,567	6,21	2,003	2,125	2,255
pi	2,037	2,013	2,023	2,045	2,096	2,173	2,356
vecsum	12,819	12,73	12,822	12,979	12,278	2,942	3,218
linear	2,724	2,698	2,686	2,654	2,7	2,792	3,007
jas_image_set	17,511	17,402	17,511	17,676	10,566	10,87	11,627

são contabilizados para determinação da dominância. A dominância de uma solução é o resultado da diferença entre o número de ganhos e o número de perdas. Portanto, quanto maior a dominância de uma solução, melhor será o seu posicionamento no ranqueamento em relação às demais.

Um dos problemas fundamentais que este trabalho objetiva resolver é o problema de definir configurações para uma arquitetura *multi-core* heterogênea de forma automática e que considere mais de um critério de seleção ao mesmo tempo como tempo de execução e consumo de energia. Trata-se de um problema tipicamente multiobjetivo. Várias técnicas podem ser usadas para resolver este tipo de problema, variando em sua complexidade. O ranqueamento de dominância é uma solução que, de certa forma, simplifica o problema e pode gerar resultados satisfatórios, como pode ser visto no Capítulo 5.

Dessa forma, após aplicar o ranqueamento de dominância uma vez, tem-se um *ranking* para cada um dos critérios. A partir da soma dos critérios individuais, é obtido um único ranqueamento que mostra quais as melhores configurações de hardware para cada aplicação considerando todos os critérios ao mesmo tempo. A Tabela 5 mostra o ranqueamento dos tamanhos de *cache* para cada aplicação de referência como exemplo, considerando os três critérios simultaneamente.

Tabela 5 – Ranqueamento de dominância dos tamanhos de *cache* para cada aplicação de referência. Os melhores tamanhos estão à esquerda e os piores, à direita.

	Tamanhos da *cache*						
autocor	32KB	64KB	256KB	128KB	16KB	8KB	4KB
bubble_sort	16KB	64KB	32KB	128KB	256KB	8KB	4KB
dotprod	128KB	256KB	64KB	8KB	16KB	4KB	32KB
fibonacci	128KB	256KB	64KB	8KB	16KB	4KB	32KB
matrix_mult	32KB	64KB	128KB	256KB	8KB	16KB	4KB
max	64KB	256KB	128KB	32KB	8KB	4KB	8KB
pi	4KB	8KB	16KB	32KB	64KB	128KB	256KB
vecsum	128KB	256KB	64KB	8KB	4KB	16KB	32KB
linear	64KB	32KB	128KB	16KB	256KB	8KB	4KB
jas_image_setbbox	128KB	256KB	64KB	8KB	16KB	4KB	32KB

Esta informação é utilizada então na etapa seguinte do fluxo da ferramenta proposta (Figura 7) para guiar a definição da arquitetura heterogênea e a definição da configuração da arquitetura.

4.5 Análise das operações em ponto-flutuante

Esta etapa também utilizou o ranqueamento de dominância para definir a forma de tratamento de operações em ponto flutuante e recebeu como entrada as aplicações de referência. O *offline profiling* então também inclui informações sobre o desempenho e consumo de energia quando as aplicações de referência são executadas com unidade de ponto flutuante. A etapa de ranqueamento de dominância também é feita da mesma forma para incluir esta característica.

As Tabelas 6 e 7 apresentam o *offline profiling* para o tempo de execução e o

consumo de energia das aplicações de referência para cada possibilidade de arquitetura no que se refere às operações de ponto flutuante.

Tabela 6 – Tempo de execução para as aplicações de referência em todas as configurações de FPU possíveis.

	Tempo de execução (segundos)		
	s/ FPU	GRFPU-lite	GRFPU
auto_corre	27,31	29,32	29,32
bubble_sort	105,63	108,08	108,08
dot_prod	30,39	28,75	28,76
fibonacci	13,96	15,75	15,75
matrix_mult	346,49	343,79	343,82
max	11,96	11,89	11,89
pi	15,07	39,28	39,28
vecsum	15,95	13,84	13,84
linear	8,54	0,60	0,46
jas_image_setbbox	31,82	33,30	33,30

Tabela 7 – Consumo de energia para as aplicações de referência em todas as configurações de FPU possíveis.

	Consumo de energia (Joules)		
	s/ FPU	GRFPU-lite	GRFPU
auto_corre	77,967	84,678	85,582
bubble_sort	306,402	315,81	320,23
dot_prod	86,903	82,487	83,857
fibonacci	40,327	45,858	46,554
matrix_mult	983,855	985,997	1002,992
max	34,4	34,158	35,026
pi	43,251	113,562	115,812
vecsum	45,888	39,794	40,713
linear	24,548	1,698	1,333
jas_image_setbbox	91,192	95,568	96,748

O ranqueamento de dominância é aplicado também para o caso das operações com ponto flutuante e é calculado a partir destas tabelas da mesma forma que foi feito para o tamanho de *cache*.

4.6 Geração dos parâmetros para a arquitetura *multi-core* e mapeamento das aplicações

Este módulo recebe duas informações como entrada: a saída do DAMICORE para as $N \quad Threads + M \quad Referencethreads$ e o ranqueamento de dominância. A partir destas duas entradas, este módulo aplica uma política para inferir a configuração para cada arquitetura baseada nas similaridades em relação às aplicações de referência e

concomitantemente define o número de *cores* que a arquitetura possuirá. Como saída este módulo gera os arquivos de configuração do hardware (*Hardware Configuration Files* (HCFs)) para sintetizar corretamente a arquitetura e também gera os arquivos para compilar o software (tanto arquivos da aplicação em si, quanto arquivos que configuram o sistema operacional, passando as informações sobre a configuração do hardware e o mapeamento das aplicações nos *cores* disponíveis - *Software Information File* (SIF) e *Hardware Information File* (HIF)).

4.7 Componente *online*

Esta etapa de implementação foi realizada ao longo do desenvolvimento deste trabalho de doutorado de forma incremental, uma vez que apresentava diversos desafios. Inicialmente, uma primeira etapa de implementação foi realizada que forneceu elementos para a comprovação da viabilidade técnica e prática de alguns aspectos da ideia proposta. Neste sentido, iniciou-se por uma implementação de um *single-core* sem sistema operacional (SO), passando por um *single-core* com SO, seguido por um *multi-core* com SO e posteriormente um *multi-core* com SO e com suporte a programação com *Pthreads*. Até então, todas as arquiteturas *multi-cores* eram com microarquiteturas homogêneas. Em seguida, microarquiteturas heterogêneas com diferentes tamanhos de *cache*, com SO e suporte a *Pthreads* foram implementadas.

A proposta deste trabalho é composta por uma arquitetura *multi-core* e uma ferramenta de análise *offline* que gera esta arquitetura de acordo com a aplicação. Por isso, decidiu-se começar a implementação pela arquitetura, para se ter um maior entendimento do que era de fato viável ou não, as limitações e oportunidades. Um dos componentes fundamentais da arquitetura é o próprio processador. Então, o primeiro passo foi estudar e dominar o fluxo de desenvolvimento de sistemas com o LEON3.

A Aeroflex Gaisler disponibiliza, juntamente com a GRLIB, uma série de projetos *templates* para diversas placas de FPGA. Infelizmente, não havia um *template* para a placa utilizada neste projeto (*Stratix IV GX Development Board*). Desse modo, o primeiro passo de implementação foi desenvolver um projeto básico com apenas um *core* que funcionasse na placa disponível. Não existem também, na GRLIB, controladores de memória DDR3 SDRAM. Portanto, a princípio optou-se por utilizar memória *onchip*. A versão original da GRLIB, permite que sejam criadas memórias *onchip* de até 512KB. Porém, como é possível acrescentar mais do que 512KB de memória *onchip* na Stratix IV, a implementação da GRLIB foi estendida para que fosse possível acrescentar memória *onchip* de até 1024KB. Outra alteração realizada na GRLIB foi feita com a ajuda do então aluno de Mestrado, também orientado pelo Professor Doutor Vanderlei Bonato, Lucas Albers Cuminato (CUMINATO, 2014). Tal alteração acrescentou contadores no controlador de memória

cache para contabilizar os *cache misses* e *cache hits* e os monitores de potência. Os valores dos contadores e da leitura de potência podem ser acessados por meio de instruções em *assembly*.

Após o desenvolvimento de uma arquitetura *single-core* com o LEON3, partiu-se para o estudo do sistema operacional eCos. A Aeroflex Gaisler fornece algumas versões do eCos já pré-compiladas para utilização em conjunto com o LEON3. Em princípio, alguns testes foram realizados com estas versões e constatou-se que seria mais interessante gerar versões customizadas do eCos, para suportar, por exemplo, programação com a *API PThreads*. Então, gerou-se uma versão do eCos capaz de suportar arquiteturas *multi-cores* e com compatibilidade com a *API Pthreads*.

Já dominados os fluxos de desenvolvimento tanto do LEON3 quanto do eCos, o passo seguinte foi integrar o que havia sido desenvolvido. A partir desse ponto, foi possível gerar arquiteturas *multi-cores* homogêneas com até 1024 KB de memória *onchip* executando o sistema operacional eCos com suporte a programação com *Pthreads*. Posteriormente a interface com a memória externa e o hardware para a realização do *way-shutdown* dinâmico e suporte à FPU foram integrados à plataforma.

Nas próximas seções, as implementações de cada componente serão apresentadas em mais detalhes.

4.7.1 Código-fonte do eCos

O eCos originalmente não suporta arquiteturas heterogêneas, portanto, para se beneficiar das arquiteturas heterogêneas, este suporte foi implementado. Além disso, era necessário que o sistema operacional fosse capaz de identificar qual *core* é melhor para cada aplicação. Para adaptar o sistema operacional de acordo com o fluxo da ferramenta proposta, as seguintes características foram implementadas:

1. Capacidade de identificar uma *thread* por um código que fosse gerado na análise *offline* e que permanecesse o mesmo durante a execução;
2. Capacidade de saber qual o tamanho da *cache* e se há FPU para cada *core* na arquitetura;
3. Capacidade de saber qual o ranqueamento de dominância para cada *thread* a ser executada;
4. Uma política de escalonamento que, dado um *core* livre e uma fila de *threads* prontas, escolhesse a *thread* que mais se beneficia do *core* livre.

Para implementar o item 1, foi necessário alterar a implementação da função *pthread_create*, do padrão POSIX Threads. Originalmente no eCos, quando se cria uma

thread com a função ***pthread_create***, uma *thread* do *kernel* é criada e sua ID é gerada automaticamente pelo sistema. Entretanto, uma *thread* do *kernel* já possui um atributo chamado ***name***, que não é utilizado. Assim, a solução adotada neste trabalho foi modificar a função ***pthread_create*** para receber como argumento, além do que já recebia por padrão, também um nome para a *thread*. E no momento em que a *thread* do *kernel* é criada, este nome é passado também como argumento.

Após a geração da configuração da arquitetura, um arquivo chamado *Hardware Information File* (HIF) é criado. Um arquivo HIF nada mais é do que um arquivo que declara um vetor em C++ em que cada posição do vetor representa um *core* e o valor armazenado é o tamanho de sua *cache*. Outro arquivo em C++, chamado *Software Information File* (SIF) é criado contendo uma estrutura de dados que contém os nomes das *threads* e seu ranqueamento de dominância. O arquivo principal da aplicação (que declara a função *main* e cria todas as *threads*) é gerado automaticamente para definir os nomes de acordo com os nomes definidos no SIF. Tanto o HIF quanto o SIF são incluídos juntamente com os outros arquivos do sistema operacional de modo que suas variáveis são visíveis e acessíveis para o escalonador. Dessa forma, foi possível implementar os itens 2 e 3.

O eCos é um sistema operacional que executa a partir de uma memória compartilhada. Seu escalonador é uma função protegida por semáforo e a cada interrupção de *timer* específica um *core* executa o escalonador. Se o *core* está ocioso, ele deverá selecionar uma nova *thread* da fila de prontas. Originalmente, como o eCos não é *heterogeneity-aware*, a *thread* escolhida é simplesmente a primeira da fila. No caso da política implementada neste trabalho, o *core* ocioso consulta qual é o tamanho de sua *cache* no vetor definido pelo HIF e faz uma varredura na fila de prontas buscando a *thread* mais adequada para ser executada nele. Deste modo, para cada *thread* da fila de prontas faz-se uma consulta no SIF para saber qual é a dominância daquele tamanho de *cache* para aquela *thread*. A *thread* com maior dominância é a escolhida. A Figura 9 ilustra a ideia geral do algoritmo que aplica esta política de escalonamento.

```
1 while(!endOfReadyQueue()){
2     SIFpos = thread->get_name();
3     currentRank = SIF[SIFpos].dominance_ranking[HIF[CPU_THIS()].cache_size];
4     if(currentRank > bestRank){
5         bestRank = currentRank;
6         bestThread = step;
7     }
8 }
9 thread = readyQueue[bestThread];
```

Figura 9 – Política de escalonamento *heterogeneity-aware*.

A Figura 10 ilustra um exemplo do funcionamento do escalonador onde cinco aplicações foram divididas entre dois *clusters* (***A*** e ***B***). O ranqueamento de dominância

apresenta a pontuação para cada tamanho de *cache* para cada *cluster*. No escalonador, este ranqueamento é convertido em uma tabela de prioridade. Por exemplo, um *core* que possui uma *cache* de 32KB possui prioridade maior para executar aplicações do *cluster* A ($a1$, $a2$, e $a3$) do que do *cluster* B ($b1$ e $b2$). Com esta informação, cada *core* irá prioritariamente executar as aplicações mais adequadas para ele.

A Figura 10 também ilustra uma possível sequências de decisões de escalonamento para cinco instantes de tempo em um sistema com dois *cores*, CPU0 e CPU1, possuindo 32KB e 16KB de memória *cache*, respectivamente. No instante T1, a fila de prontos inclui todas as aplicações. Dado o ranqueamento de dominância e suas respectivas prioridades, CPU0 executa a aplicação $a1$, imediatamente seguido pela CPU1 no instante T2 que executa a aplicação $b1$. O escalonamento das aplicações então procede desta maneira para as demais aplicações. Entretanto, como no caso representado pelo instante T5, sempre que a fila de prontos não possuir mais aplicações adequadas para um dado *core*, o algoritmo seleciona para este *core* a melhor aplicação disponível (aquela com o menor valor de prioridade). A complexidade do escalonador é $O(n)$, onde n é o tamanho da fila de prontos. Porém, somente no pior caso é que todos os elementos da fila devem ser verificados pelo escalonador. Embora este algoritmo necessite de algumas comparações dos valores de prioridade, não foi verificado *overhead* significativo durante as execuções do escalonador.

Ranqueamento de Dominância

	4KB	8KB	16KB	32KB	64KB	128KB	256KB
Cluster A	-17	-13	-6	14	13	7	2
Cluster B	-18	-8	12	8	10	0	-4

Tabela de Prioridade (incluída no SIF) – Ex: CPU0 = 32KB; CPU1 = 16KB

	4KB	8KB	16KB	32KB	64KB	128KB	256KB
Cluster A	6	5	4	0	1	2	3
Cluster B	6	5	0	2	1	3	4

	Fila de Prontos	Decisão do Escalonador
T1	a1,a2,b1,a3,b2	CPU0 runs a1
T2	a2,b1,a3,b2	CPU1 runs b1
T3	a2,a3,b2	CPU1 runs b2
T4	a2,a3	CPU0 runs a2
T5	a3	CPU1 runs a3

Figura 10 – Exemplo do funcionamento do escalonador. O ranqueamento de dominância e a tabela de prioridades considerando dois *clusters* (A e B) e uma arquitetura com dois *cores* (CPU0 com 32KB e CPU1 com 16KB de memória *cache*). Decisões de escalonamento nos instantes $T1$ até $T5$.

O código fonte do eCos foi modificado também para incluir a informação sobre qual é o tipo de FPU que um *core* possui, caso a possua. Esta informação é visível também em nível de aplicação.

Dois arquivos binários de uma mesma aplicação são diferentes se um for compilado para usar FPU e outro para emular em software e um erro ocorre caso um binário compilado para usar FPU seja executado em um processador sem FPU. Várias soluções existem na literatura para este problema, como por exemplo reescalonar a aplicação para o *core* com FPU quando for necessário. Esta solução economiza espaço em memória por possuir apenas um código compilado, porém possui o *overhead* do reescalonamento. Outra solução é compilar dinamicamente a aplicação somente quando souber qual será o *core* que irá executá-la. Esta solução também possui grande *overhead*. Uma solução mais simples que ocupa mais espaço em memória mas que elimina este *overhead* é manter dois binários pré-compilados; e no momento da execução escolher o binário correto a executar. Está solução foi implementada neste projeto. Momentos antes de invocar a função a ser executada a aplicação verifica se o *core* em que ela está sendo executada possui FPU ou não e assim decide qual binário executar. Estes binários são acessíveis por meio de ponteiros para função, declarados em C.

4.7.2 Ponte Avalon-MM/AMBA AHB e controlador da memória DDR3

A Altera disponibiliza o controlador de memória DDR3 para a memória presente na placa. Para poder utilizar este mesmo controlador juntamente com o LEON3, foi necessário implementar uma ponte para traduzir os sinais do barramento Avalon-MM para AMBA AHB. A implementação da ponte em si não representa grandes dificuldades, entretanto o controlador disponibilizado pela Altera possui muitos blocos em que não se tem acesso ao código-fonte. Dessa forma, o entendimento do funcionamento do controlador e as suas particularidades de acesso é uma tarefa trabalhosa e complexa.

Os sinais do barramento Avalon-MM de um periférico do tipo escravo podem ser vistos na Figura 11 e os sinais de uma transição no barramento AMBA AHB são apresentados na Figura 12.

A diferença básica entre os dois barramentos é que o barramento AMBA AHB divide sua transação em duas fases - *address phase* e *data phase* – e o barramento Avalon-MM realiza a transação em uma única fase. A etapa *address phase* ocorre em um único ciclo de *clock*, entretanto a *data phase* pode durar vários ciclos. A ponte implementada basicamente trata esta diferença.

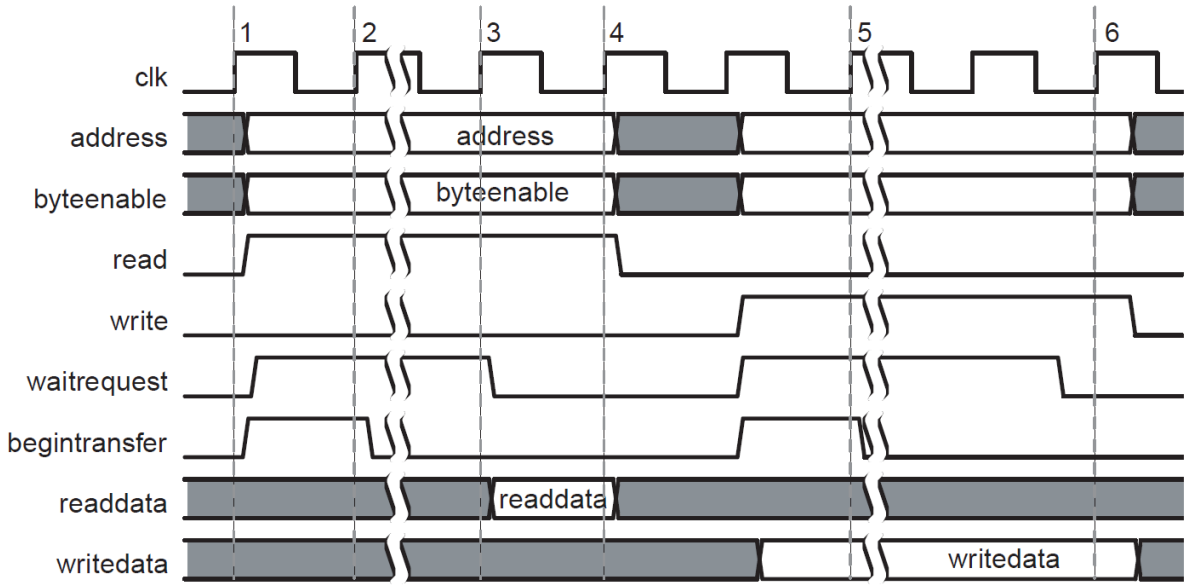

Figura 11 – Transação no barramento Avalon-MM

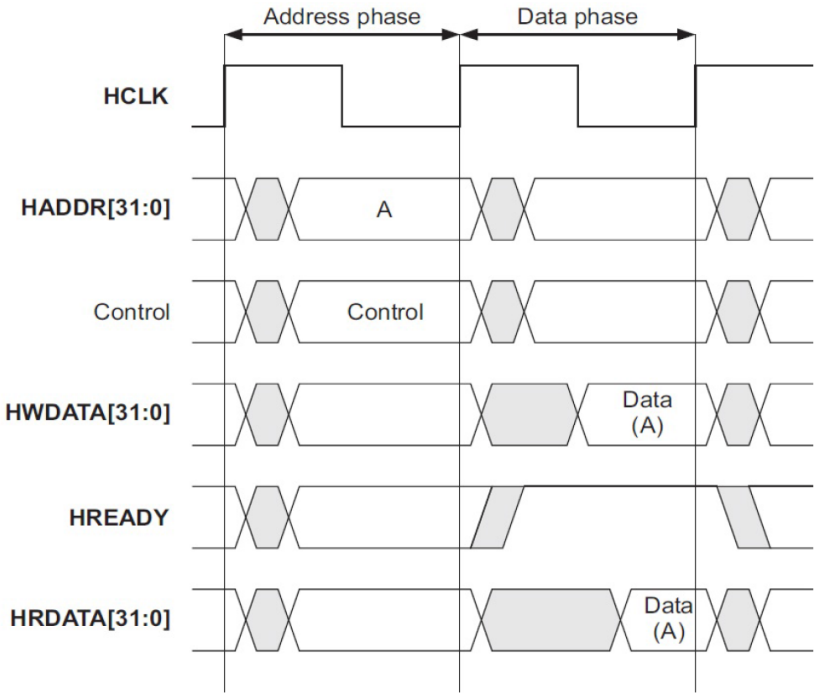

Figura 12 – Transação no barramento AMBA AHB.

4.7.3 Monitor de potência instantânea consumida e interface AMBA APB

A interface para acessar o CPLD que realiza a medição da potência instantânea é semelhante a uma interface de acesso à uma memória SRAM. Foi implementado então um bloco que se comunica com o CPLD por meio dos seguintes sinais:

- max_csn: *chip select*;
- max_wen: *write enable*;
- max_oen: *output enable*;
- max_ben : *byte write enable signal*;
- max_address: endereço;
- max_data: dados.

E finalmente, foi implementado um outro bloco para traduzir estes sinais para a interface *slave* do barramento AMBA AHB seguindo o padrão de transação ilustrado na Figura 12.

Este componente está mapeado na memória e o acesso aos seus valores é feito por meio de um *driver* que foi implementado em C, incluído como cabeçalho juntamente com o programa que necessita medir a potência. Como o valor lido é a potência, o software deve calcular a energia total consumida de acordo com o tempo de execução e os valores de potência instantânea lidos.

4.7.4 *Interrupção de* timer *para aplicar o algoritmo que realiza* way-shutdown *dinamicamente*

Além da otimização realizada durante a análise *offline* e da implementação da política de escalonamento, a arquitetura permite que os valores de *cache miss* e *hit* sejam monitorados em tempo de execução. De acordo com os valores lidos, pode-se aplicar um algoritmo que altera o número de *ways* da *cache* ativados. Para implementar este mecanismo, o principal objetivo foi deixar esta funcionalidade modularizada no sistema operacional e invisível para a aplicação propriamente dita. Adicionalmente, cada *core* deveria ser capaz de executar esta funcionalidade de forma independente. Para isso, foi utilizada a implementação de tratamento de interrupção presente no eCos.

A cada *core* então foi associada uma interrupção que a cada 140ms invoca uma função que aplica o algoritmo de *way-shutdown*. De uma maneira geral, este algoritmo monitora a taxa de *cache miss* atual e compara com a anterior e, dependendo dos valores, o algoritmo ativa ou desativa os *ways* da *cache* do *core* em questão. Os resultados obtidos com o intervalo entre cada interrupção definido em 140ms foram satisfatórios, entretanto este valor pode ser alterado, se necessário.

4.7.5 Alarme para registrar os valores de potência instantânea

O bloco de hardware responsável por salvar os valores de potência instantânea consumida pelo FPGA realiza nova leitura a cada 140ms. Desta maneira, é necessário um mecanismo que realiza a leitura desse registrador e salva seu valor em uma variável compartilhada visível no nível da aplicação. Qualquer *core* pode executar esta função, não existindo portanto a mesma restrição que há para o algoritmo de *way-shutdown* dinâmico. Optou-se pela utilização de um alarme por ser mais simples em termos de implementação do que uma interrupção de *timer*. Entretanto, seu funcionamento é similar.

4.7.6 Suporte à unidade de ponto flutuante na plataforma de hardware

A empresa Aeroflex Gaisler, que disponibiliza os códigos do processador LEON3, também disponibiliza gratuitamente as *netlists* das unidades de ponto flutuante disponíveis para o LEON3. O processo de inclusão de uma *netlist* e utilização em um projeto descrito em VHDL é relativamente simples. Duas variações de *netlists* interessantes para este projeto são disponibilizadas: uma versão eficiente (GRFPU) que ocupa maior área e também consome mais potência e uma versão mais simples, menos eficiente, mas que ocupa menos área e consome menos potência (GRFPU-lite).

Para a inclusão das *netlists* no projeto do hardware, basta alterar alguns parâmetros genéricos para que o hardware seja sintetizado com a FPU desejada. É importante observar também que ao compilar um software que irá utilizar a FPU, deve-se omitir a diretiva de compilação $msoft - float$.

4.8 Considerações finais

Neste capítulo foi apresentado o fluxo da ferramenta proposta juntamente com os detalhes referentes a cada etapa do fluxo, implementações e adaptações realizadas. Ao longo do processo de implementação, alguns experimentos foram realizados em um processo iterativo/incremental de implementação. Portanto, alguns resultados foram obtidos em uma primeira etapa, antes da conclusão da implementação da ferramenta. Após sua conclusão, a mesma foi submetida a diversos experimentos, caracterizando uma segunda etapa para os resultados. Tanto os resultados da primeira etapa quanto da segunda serão apresentados no próximo capítulo.

CAPÍTULO

5

RESULTADOS

Este capítulo apresenta os resultados alcançados nos experimentos realizados com o sistema descrito no Capítulo 4. Os experimentos foram incrementais e desenvolvidos ao longo deste trabalho de doutorado.

A Seção 5.1, apresenta um estudo inicial sobre os benefícios em se usar diferentes tamanhos de *cache* em uma arquitetura *multi-core*. Nesse estudo, o escalonamento foi realizado manualmente e a arquitetura era bastante limitada, possuindo um limite de quatro *cores*, memória *on-chip* (limitando o tamanho das memórias *cache*), contadores para *cache miss* e *hit* e sem monitor de potência. Quatro aplicações foram selecionadas com diferentes perfis de uso de memória para explorar os diferentes tamanho de *cache* em cada *core*.

A partir da Seção 5.2, são apresentados experimentos integrando a ferramenta DAMICORE. Na Seção 5.2.1, é apresentado um exemplo básico de utilização da ferramenta DAMICORE para os mesmo códigos da Seção 5.1. Porém, a partir deste experimento, é possível agrupar diversas aplicações para executar em um mesmo *core*. Na Seção 5.2.2, o objetivo foi ampliar a quantidade de aplicações submetidas ao DAMICORE, estimando os ganhos possíveis e verificando a influência do tamanho dos dados e da padronização dos códigos submetidos ao DAMICORE. As limitações da arquitetura e do escalonador ainda estão presentes nestes dois experimentos.

Nos experimentos mencionados, não havia variação dos valores obtidos tanto em relação ao tempo de execução quanto às taxas de *cache miss* e *hit* devido à aplicação de escalonamento manual e à simplicidade da arquitetura utilizada nos experimentos das Seções 5.1, 5.2.1 e devido aos resultados do experimento da Seção 5.2.2 se tratarem de uma estimativa. Por isso, não foi necessário realizar diversas execuções de um mesmo escalonamento e análise estatística dos dados.

O fluxo da ferramenta apresentado no Capítulo 3, Figura 7, começa a ser mais

adotado nos experimentos descritos a partir da Seção 5.2.3. Para estes experimentos, a arquitetura disponível já possuía memória externa DDR3, monitores de potência, *cache miss* e *hit*, controladores de *ways*, FPU, política de escalonamento *heterogeneity-aware* e todas as outras características descritas no Capítulo 4. Além disso, a partir da Seção 5.2.3 é que as aplicações foram padronizadas em relação ao código e tamanho dos dados utilizados.

Para melhor analisar os resultados encontrados a partir da Seção 5.2.3, uma análise estatística se fez necessária, uma vez que diversos fatores alteravam os resultados de uma execução, como por exemplo: ordem de criação das *threads* no sistema operacional, interrupções para ajustar *ways* ativos da *cache*, interrupções para medir potência, entre outros.

A seguir serão apresentados os resultados a partir dos experimentos mais simples até os experimentos de maior complexidade.

5.1 Estudo inicial dos benefícios de diferentes tamanhos de *cache*

O objetivo deste experimento foi investigar os benefícios de se usar diferentes tamanhos de *cache* em arquiteturas *multi-cores* e como o escalonador do sistema operacional pode explorar estes benefícios de acordo com a carga do sistema.

Neste experimento, foi sintetizada uma arquitetura *multi-core* e, usando um protótipo de um escalonador, quatro aplicações foram executadas em diferentes escalonamentos e foram verificadas as taxas de *cache miss* e *cache hit* da aplicação como um todo. Uma comparação da área ocupada pelas arquiteturas no FPGA também foi feita.

As arquiteturas geradas seguiram a plataforma básica desenvolvida, foi sintetizada para o *Stratix IV GX Development Kit*. A arquitetura básica pode ser vista na Figura 13.

A arquitetura é composta por quatro processadores LEON3 conectados via barramento AMBA e compartilham uma memória *onchip* de 1024KB.

5.1.1 Protótipo do escalonador

O protótipo desenvolvido é executado com base em dados coletados por meio de *offline profiling* e construção da assinatura arquitetural contendo o número de *cache misses* de cada aplicação em cada *core*. Baseado nestas medidas, foi proposto um método chamado *decreasing step*. O *decreasing step* indica quantos *cache misses* seriam economizados se a aplicação executasse em um *core* com mais *cache* em vez de executar em um *core* com menos *cache*.

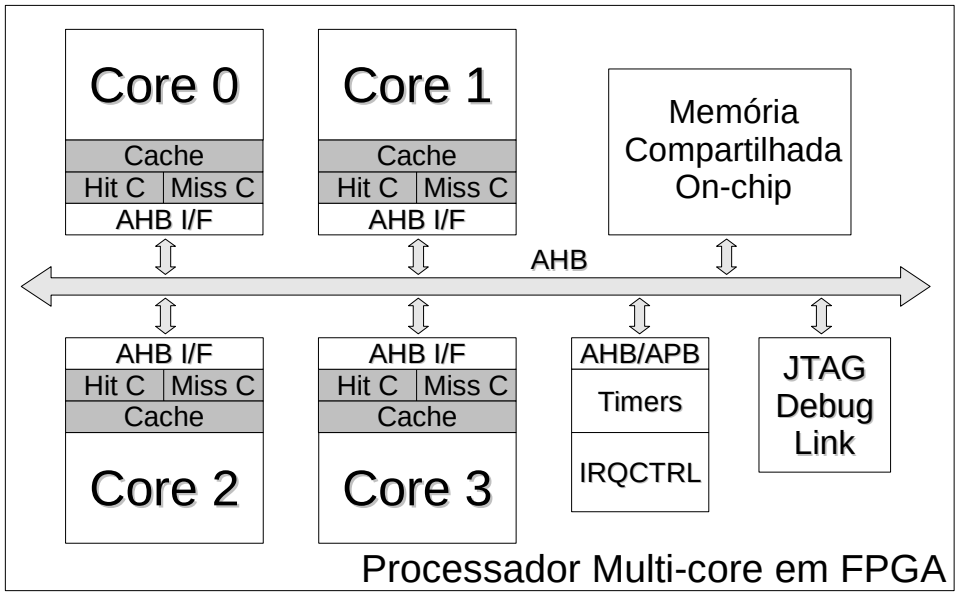

Figura 13 – Arquitetura *multi-core* básica usada no experimento.

Neste protótipo, foi utilizada uma abordagem gulosa para definir o escalonamento estático: inicia-se no primeiro *core* (aquele com a menor *cache*) e atribui-se a ele a *thread* que possui o menor *decreasing step* considerando o *core* atual e o próximo *core* (com *cache* maior). Este passo é feito até que todas as *threads* sejam atribuídas aos *cores*. A lógica por trás desta abordagem é que as *threads* que possuem menos *cache misses* que outras não tem prioridade para ser executadas nos *cores* com *caches* maiores.

Uma vez que o escalonamento estático foi definido, ele é escrito no código manualmente e a aplicação é recompilada para incluir o escalonamento. Este protótipo é uma versão modificada do MLQ, porém sem o *timeslice* e com o número de *threads* igual ao número de *cores*. Durante a execução propriamente dita, se uma *thread* inicia sua execução em um *core*, ela finalizará nele (desde que a *thread* não seja bloqueada). Sendo assim, não há *Round-Robin* e nem *thread-reassignment*.

5.1.2 Configuração do experimento

Nas arquiteturas geradas, foram implementadas *caches* utilizando o mapeamento direto e os tamanhos possíveis foram definidos como 1KB, 2KB, 4KB e 8KB. Desta forma, o HMP (chamado de HT) possui quatro *cores*, cada um contém um dos tamanhos de *cache* mencionados.

Uma aplicação *multi-thread* foi implementada na qual cada *thread* executa uma tarefa totalmente diferente. Então, tem-se quatro *threads*, cada uma executa os seguintes algoritmos: cálculo do pi (*pi*); o método *Jacobi-Richardson* para resolução de sistemas lineares (*jacobi*); e os métodos de ordenação *quick sort* e *bubble sort*. O *pi* é calculado por um algoritmo baseado em séries infinitas e realiza 80000 iterações para finalizar o

cálculo. A dimensão da matriz manipulada pelo método *Jacobi-Richardson* é 100 x 100 e o tamanho dos vetores manipulados pelo *bubble sort* e pelo *quick sort* são de 60000 e 10000, respectivamente.

5.1.3 Medições offline

As quatro *threads* foram executadas em cada *core* e o *cache miss* foi medido por meio dos contadores presentes em cada *core* da arquitetura. Baseado nos valores encontrados, foi definido um escalonamento estático considerando o *decreasing step*. Os resultados das medições *offline* podem ser vistos na Tabela 8.

Tabela 8 – Número de *cache misses* para diferentes tamanhos de *cache* (assinatura arquitetural) e o *Decreasing Step* que guia a política de escalonamento.

	Número de *Cache Miss*			
	pi	*jacobi*	*bubble sort*	*quick sort*
Cache 1KB	234	28.380.688	449.777.774	51.473.744
Cache 2KB	204	17.827.342	449.001.326	50.497.584
Cache 4KB	189	16.283.151	445.875.566	49.323.488
Cache 8KB	128	15.541.389	433.332.590	45.988.152
	Decreasing Step			
De 1KB para 2KB	**30**	10.553.346	776.448	976.160
De 2KB para 4KB	*15*	1.544.191	3.125.760	**1.174.096**
De 4KB para 8KB	*61*	**741.762**	12.542.976	*3.335.336*

Observando o *cache miss* de cada aplicação, é possível classificá-las entre *CPU-Intensive* e *Memory-Intensive*, onde o *pi* é mais *CPU-Intensive*, *jacobi* é menos *CPU-Intensive*, *quick sort* é menos *Memory-Intensive* e *bubble sort* é mais *Memory-Intensive*. Entretanto, independente se a aplicação é *CPU-Intensive* ou *Memory-Intensive*, cada aplicação reage diferentemente das outras quando o tamanho da *cache* muda. Algumas aplicações podem ser mais sensíveis às mudanças do que outras. Quanto mais uma aplicação é influenciada pelo aumento do tamanho da *cache*, maior é o benefício de executá-la em *caches* maiores.

Seguindo a abordagem do protótipo do escalonador, *pi* apresenta o menor *decreasing step* da *cache* de 1KB para a *cache* de 2KB. Portanto, o *pi* é atribuído para execução no *core* 0. Em seguida, o *quick sort* é atribuído ao *core* 1, o *jacobi* é atribuído ao *core* 2 e o *bubble sort* é atribuído ao *core* 3. A Tabela 9 apresenta o escalonamento considerando estes algoritmos.

Tendo em mãos o escalonamento estático, foram executados todos os escalonamentos possíveis na arquitetura *multi-core* e comparou-se as taxas de *cache miss* e *hit* para a execução completa.

Tabela 9 – Escalonamento estático considerando os diferentes tamanhos de *cache*.

HMP	Tamanho da *cache*	Aplicação
Core 0	1KB	*pi*
Core 1	2KB	*quick sort*
Core 2	4KB	*jacobi*
Core 3	8KB	*bubble sort*

5.1.4 Resultados e discussões

Para comparar a utilização dos recursos de memória no *multi-core*, cinco arquiteturas *multi-cores* foram sintetizadas. Quatro delas são arquiteturas *multi-cores* homogêneas usando os tamanhos de *cache* possíveis e a restante é com tamanhos de *cache* heterogêneo. Todas as arquiteturas implementadas neste experimento possuíam quatro *cores*.

Na Figura 14, a utilização dos recursos de memória é apresentada (quantidade de *bits* de blocos de memória do FPGA) para cada arquitetura *multi-core* gerada. As arquiteturas homogêneas (HM) possuem quatro *cores*, cada uma com tamanhos de *cache* variando de 1KB até 8KB. No *multi-core* com *caches* heterogêneas (HT), cada *core* possui um tamanho de *cache*: 1KB, 2KB, 4KB e 8KB, totalizando 15KB. Esta arquitetura usou menos recursos de memória do que a arquitetura com quatro *cores*, cada um contendo 4KB de *cache*, totalizando 16KB (HM4).

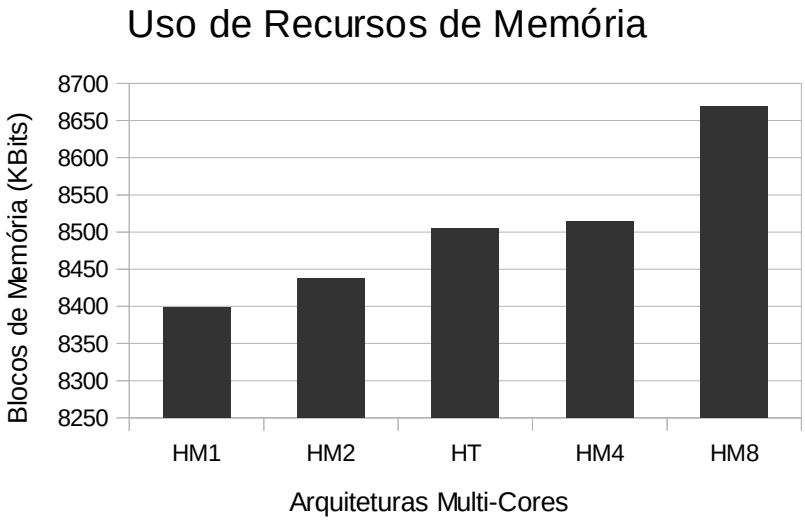

Figura 14 – Comparação da utilização de recursos de memória para diferentes arquiteturas *multi-cores*.

As taxas de *cache miss* e *hit* gerais foram calculadas para todos os possíveis escalonamentos no *multi-core* heterogêneo, como pode ser visto nas Figura 15 e 16, respectivamente.

O escalonamento com a menor taxa de *cache miss* e a maior taxa de *cache hit* (ou seja, o *best static scheduling*) foi o 1,4,2,3. O que significa que a *thread* 1 (*pi*) executa

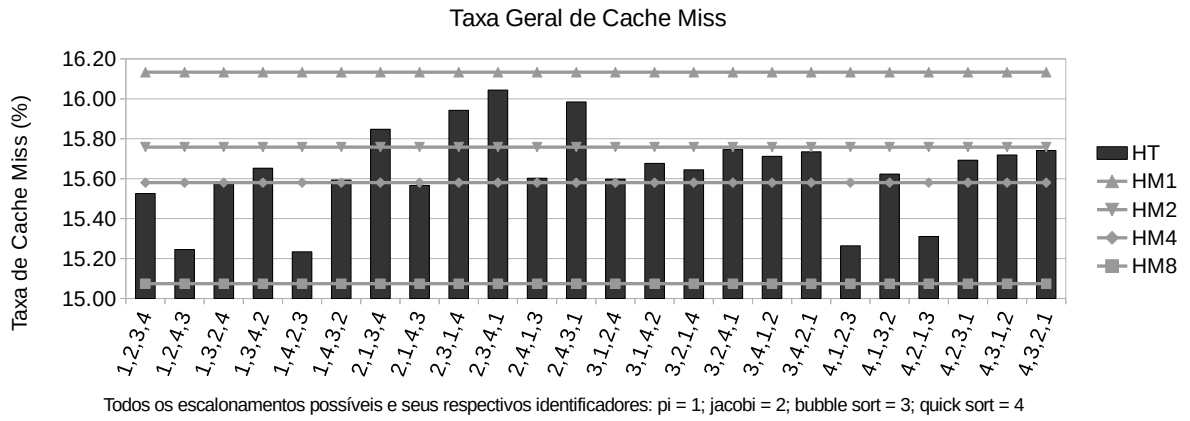

Figura 15 – *Cache miss* geral da aplicação executando nos processadores *multi-cores*

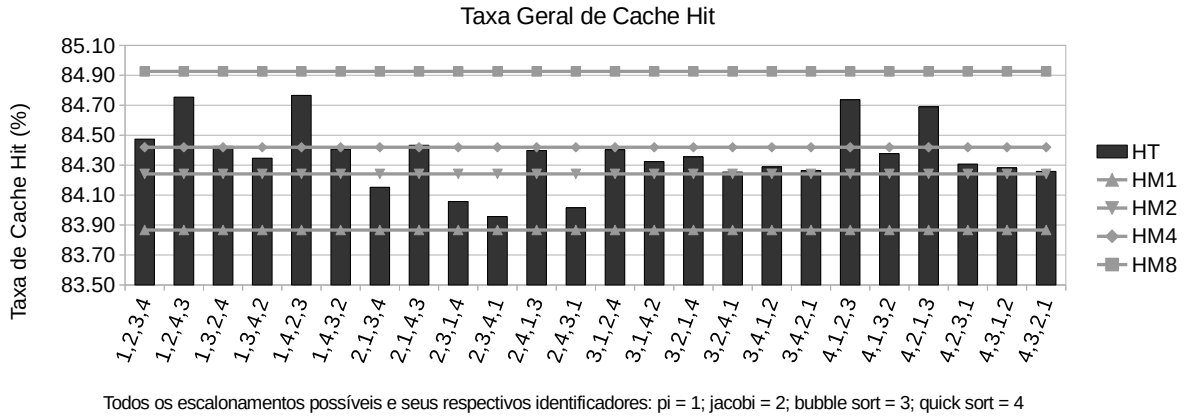

Figura 16 – *Cache hit* geral da aplicação executando nos processadores *multi-cores*.

no *core* 0, a *thread* 4 (*quick sort*) no *core* 1, a *thread* 2 (*jacobi*) no *core* 2 e a *thread* 3 (*bubble sort*) no *core* 3. Olhando novamente na Tabela 9, pode-se verificar que, neste caso, o escalonamento é exatamente o mesmo encontrado pelo *decreasing step* do *cache miss* obtido pela assinatura arquitetural (Tabela 8).

Pode-se observar que, qualquer que seja o escalonamento usado, a taxa de *cache miss* do HT nunca foi pior do que o HM1 e nunca foi melhor do que o HM8, como esperado. Porém, dentro destes limites, a taxa varia consideravelmente. Embora a diferença entre o melhor escalonamento (1,4,2,3) e o pior (2,3,4,1) seja somente de 0,81%, tal porcentagem representa 26.592.071 *cache misses*, uma vez que a aplicação realiza mais do que 3 bilhões de leituras na *cache*. Consequentemente, uma boa política de escalonamento é essencial para explorar os benefícios de diferentes tamanhos de *cache* em HMPs reduzindo a taxa global de *cache miss*.

A memória do sistema é limitada devido ao uso somente da memória *onchip*. Este fato limitou o experimento, pois não foi possível executar programas maiores e com mais dados. O passo seguinte foi incluir memória externa para verificar o comportamento em

problemas maiores.

Embora o protótipo de escalonador seja simples, sem *overhead* em tempo de execução e relativamente fácil de ser implementado, ele possui algumas limitações. Primeiro, deve-se assumir qual é o caso comum de dados de entrada. Se a aplicação muda radicalmente sua taxa de *cache miss* para diferentes dados de entrada, o *profiling* realizado de modo *offline* não irá obter essa informação. Uma possível solução seria executar a aplicação com vários dados de entrada e calcular a média aritmética das taxas de *cache miss*. Outro problema acontece sempre que uma *thread* muda seu comportamento em tempo de execução. Por exemplo, supondo que a taxa geral de *cache miss* de uma *thread* seja baixa, mas em um dado momento ela apresenta altas taxas de *cache miss* se comparada às outras *threads* em execução. Nestes casos, é importante haver um mecanismo de ajuste fino do tamanho da *cache*, como por exemplo redimensionamento dinâmico da *cache*.

Neste experimento, o FPGA foi usado para sintetizar arquiteturas *multi-cores* com diferentes tamanhos de *cache* e, por meio de um protótipo de escalonador, foram comparados os resultados apresentados pelos processadores *multi-cores* homogêneos com um heterogêneo. O *multi-core* heterogêneo com um total de 15KB de memória *cache* usou menos recursos de memória que a arquitetura HM4, que possui 4KB de *cache* (totalizando 16KB para o sistema completo). Ao aplicar o *best static scheduling*, o *multi-core* heterogêneo atingiu uma taxa de *cache miss* menor do que o HM4. Este fato mostra o potencial de se usar *multi-cores* com diferentes tamanhos de *cache* e políticas de escalonamento que considerem essa informação, pois além de reduzir o uso de recursos de memória do FPGA é também possível obter taxas menores de *cache miss* fornecendo um melhor relacionamento desempenho/consumo de energia.

5.2 Experimentos integrados com a ferramenta DAMICORE

Na seção anterior, foram apresentados resultados de um protótipo de escalonador *offline* que se aproveita das vantagens de uma arquitetura *multi-core* com diferentes tamanhos de *cache*. Entretanto, tal escalonador necessita de um *offline profiling* da aplicação, chamado de assinatura arquitetural. Esta assinatura é construída pela execução de todas as aplicações em todos os *cores* possíveis e realizando a medição das taxas de *cache miss* e *hit*.

No experimento descrito a seguir, uma técnica é apresentada para guiar a configuração da *cache* automaticamente de acordo com a aplicação. Esta técnica também é usada para fornecer informações para o escalonador mapear as aplicações nos *cores* de uma forma que melhore o desempenho e o consumo de energia. Foi observado que na maioria dos casos é possível obter uma taxa menor de *cache miss* usando menos recursos

de memória se comparado com processadores *multi-cores* tradicionais (usando a mesma configuração de *cache* para todos os *cores*).

5.2.1 Exemplo básico - 4 aplicações

A arquitetura de hardware usada neste experimento é a mesma apresentada na Figura 13 com quatro *cores* e monitores de *cache hit* e *miss* usada no experimento descrito na seção anterior.

Para melhor entendimento do uso da ferramenta DAMICORE, inicialmente foi realizado um experimento com poucas aplicações. Posteriormente, o experimento foi expandido e este será apresentado na próxima seção.

Foi usada a mesma aplicação *multi-threaded* possuindo quatro *threads* da seção anterior. No experimento anterior, a assinatura arquitetural foi obtida por meio da execução de todas essas threads em todos os *cores* contendo diferentes tamanhos de *cache*. Os tamanhos de *cache* possíveis foram definidos como 1KB, 2KB, 4KB e 8KB (Tabela 8).

O problema consiste em encontrar uma configuração de *cache* para a arquitetura *multi-core* que minimize o número de *cache misses* para a aplicação. Se todas as aplicações forem executadas em todos os *cores*, teremos a assinatura arquitetural completa. Mas encontrar a solução pode ser infactível sempre que o número de combinações for muito grande. A técnica proposta com este experimento, que justifica o uso da ferramenta DAMICORE, usa somente informações do código-fonte como guia para a geração da arquitetura.

O DAMICORE foi aplicado para analisar o código-fonte das quatro aplicações e dois agrupamentos foram encontrados conforme a Figura 17. O *quick sort* e o *bubble sort* formam o *Cluster 1* e as aplicações *pi* e *jacobi* formam o *Cluster 2*.

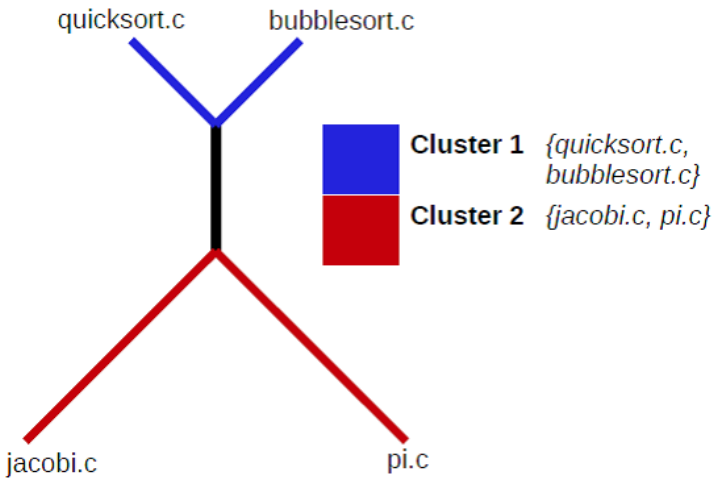

Figura 17 – Fornecendo as quatro aplicações para a entrada do DAMICORE gera um conjunto de *clusters* como saída. Neste caso, o *Cluster 1* é composto pelas aplicações *quick sort* e *bubble sort* e o *Cluster 2* é composto por *pi* e *jacobi*.

Tabela 10 – Estatísticas para a *cache* e o total de memória usado em cada arquitetura *multi-core*.

	Cache miss (%)	Cache hit (%)	Uso de memória (KB)
HM1	16.13	83.87	2
HM2	15.76	84.24	4
HT1	15.74	84.26	9
HM4	15.58	84.42	8
HT2	15.47	84.53	9
HM8	15.07	84.93	16

Para comparar a arquitetura gerada com esta abordagem, os *clusters* foram executados em quatro arquiteturas *multi-cores* homogêneas com dois *cores* cada: HM1 (1KB por *core*), HM2 (2KB por *core*), HM4 (4KB por *core*) e HM8 (8KB por *core*). A arquitetura heterogênea gerada (HT) possui dois *cores*, um com 1KB e outro com 8KB de *cache*. HT1 e HT2 são as mesmas arquiteturas, porém o mapeamento dos *clusters* é diferente: para HT1, o *Cluster 1* executa no *core* com 1KB e o *Cluster 2* executa no *core* com 8KB, para HT2 é o contrário.

O total de *cache hits* e *misses* para a execução dos *clusters* em cada arquitetura está na Figura 18. A Tabela 10 ilustra o total de *cache hit* e *miss* assim como o total de memória usado em cada uma das arquiteturas.

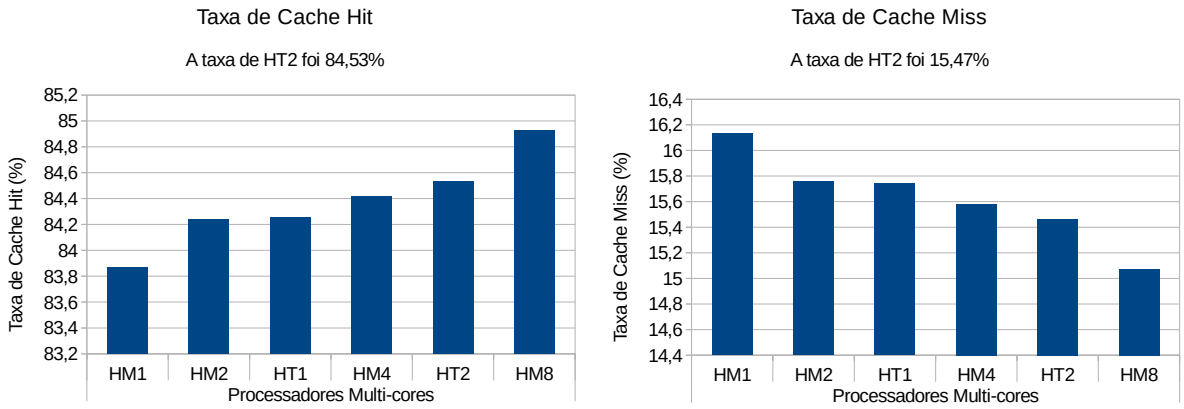

Figura 18 – Taxa de *cache* para a execução do *Cluster 1* (*quick sort* e *bubble sort*) e do *Cluster 2* (*pi* e *jacobi*) nas arquiteturas homogêneas (HM de 1 a 8) e na arquitetura heterogênea, com as duas possibilidades de mapeamento dos *clusters* nela (HT1 e HT2).

Observando a Figura 18, é possível notar que a arquitetura sugerida pela abordagem DAMICORE (HT2) possui proporcionalmente menos *cache miss* do que a arquitetura homogênea HM4 (15,58%) e mais do que HM8 (15,07%). Este exemplo básico ilustra o fluxo da proposta de se usar o DAMICORE e evidencia algumas questões importantes. Quando arquiteturas *multi-cores* com tamanhos diferentes de *cache* para os *cores* são usadas, dependendo das características da aplicação, é possível obter taxas de *cache miss* e *hit* melhores se comparadas às arquiteturas *multi-cores* homogêneas. Para isso, o mapeamento e escalonamento das aplicações é fundamental para obter um resultado que

Tabela 11 – Taxa de *cache miss* e *hit* para diferentes tamanhos de *cache* para 15 aplicações.

		1KB Cache	2KB Cache	4KB Cache	8KB Cache
pi	miss rate	0,0014%	0,0012%	0,0011%	0,0008%
	hit rate	99,9986%	99,9988%	99,9989%	99,9992%
jacobi	miss rate	10,84%	6,81%	6,22%	5,94%
	hit rate	89,16%	93,19%	93,78%	94,06%
bubble_sort	miss rate	15,48%	15,46%	15,35%	14,92%
	hit rate	84,52%	84,54%	84,65%	85,08%
quick_sort	miss rate	51,46%	50,48%	49,31%	45,97%
	hit rate	48,54%	49,52%	50,69%	54,03%
adpcm_code	miss rate	19,91%	18,35%	16,79%	16,79%
	hit rate	80,09%	81,65%	83,21%	83,21%
adpcm_deco	miss rate	11,95%	11,95%	10,07%	10,07%
	hit rate	88,05%	88,05%	89,93%	89,93%
autcor	miss rate	2,63%	2,63%	2,63%	2,63%
	hit rate	97,38%	97,38%	97,38%	97,38%
bubble_sort2	miss rate	1,61%	1,61%	1,61%	1,61%
	hit rate	98,39%	98,39%	98,39%	98,39%
dotprod	miss rate	100%	100%	100%	100%
	hit rate	0%	0%	0%	0%
fdct	miss rate	14,22%	13,64%	13,64%	13,64%
	hit rate	85,78%	86,36%	86,36%	86,36%
fibonacci	miss rate	40,22%	40,22%	40,22%	40,22%
	hit rate	59,78%	59,78%	59,78%	59,78%
max	miss rate	100%	100%	100%	100%
	hit rate	0%	0%	0%	0%
pop_cnt	miss rate	100%	100%	100%	100%
	hit rate	0%	0%	0% 0%	0%
sobel	miss rate	4,01%	4,01%	4,01%	4,01%
	hit rate	95,99%	95,99%	95,99%	95,99%
vecsum	miss rate	100%	100%	100%	100%
	hit rate	0 (0%)	0 (0%)	0 (0%)	0 (0%)

valha a pena. Com o DAMICORE, é possível agrupar as aplicações e, então, potencialmente economizar espaço e energia no sistema. Dessa forma, o número de *cores* e o número de aplicações mapeadas para cada um é uma questão importante também. Na próxima seção será apresentado um experimento com mais aplicações configurando uma situação mais complexa e realista.

5.2.2 Experimento com 15 aplicações

Neste experimento, foram definidas 15 aplicações conforme apresentado na Tabela 11. Esta tabela mostra as taxas de *cache hit* e *miss* medidas para cada aplicação nos tamanhos de *cache* possíveis para a arquitetura utilizada neste experimento.

O código-fonte destas aplicações foi então passado como entrada para a ferramenta DAMICORE e o agrupamento produzido como saída pode ser vista na Figura 19. O DAMICORE gerou cinco *clusters*. Se a mesma abordagem apresentada no exemplo básico for seguida, a arquitetura gerada possuirá cinco *cores*. Porém, neste experimento a arquitetura *multi-core* estava limitada a ter no máximo quatro *cores*. Consequentemente, um *cluster* deverá ser agrupado a outro *cluster*. Qualquer *cluster* poderia ser agrupado a outro *cluster*, considerando a distância entre eles. Porém, neste experimento, os *Clusters* 1 e 2 foram agrupados por serem aqueles que possuem menores taxas de *cache miss*, de acordo com a Tabela 11.

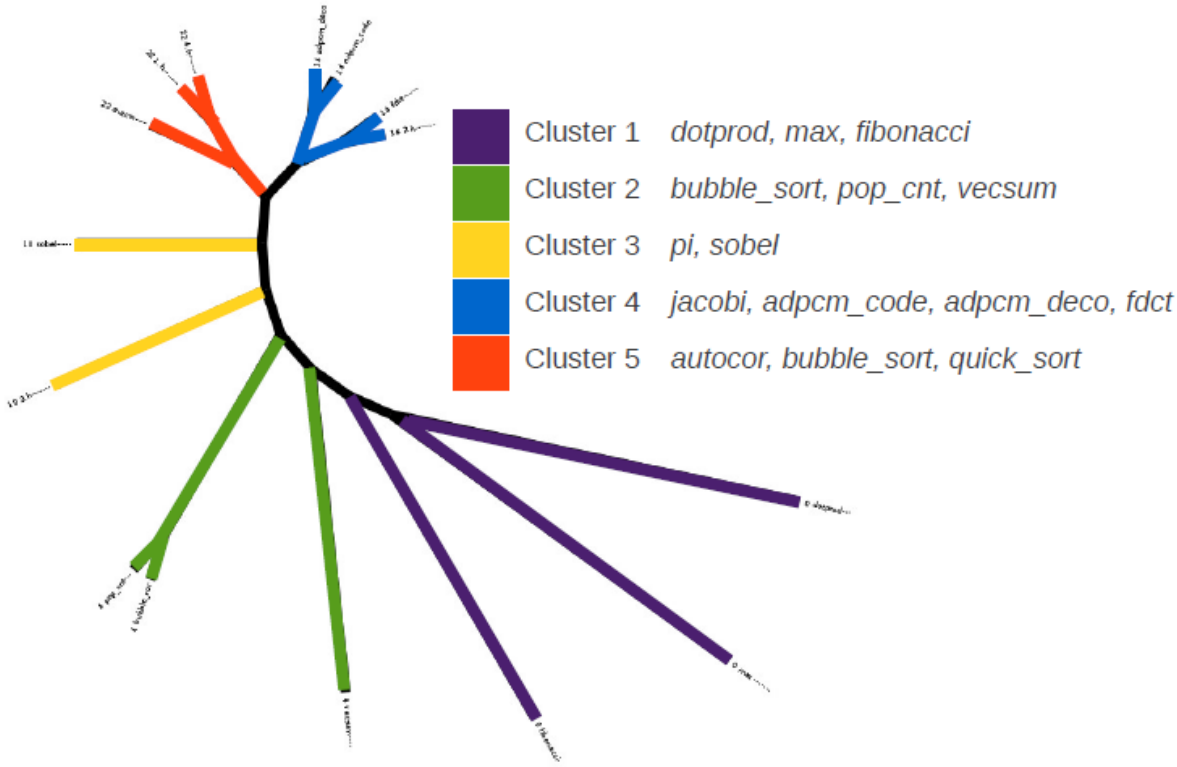

Figura 19 – Clusterização fornecida pela ferramenta DAMICORE para as aplicações da Tabela 11.

Tabela 12 – Mapeamento dos *clusters* na arquitetura *multi-core* heterogênea.

1KB cache	Clusters 1 and 2
2KB cache	Cluster 3
4KB cache	Cluster 4
8KB cache	Cluster 5

O próximo passo é definir o tamanho das memórias *cache* para cada *core*, lembrando que o *pi* foi usada como referência no exemplo básico. Neste experimento, tal conhecimento *a priori* foi considerado também. Entretanto, o *Cluster 1* e o *Cluster 2* necessitam de menos *cache* do que *pi*. Sendo assim o tamanho da *cache* para executar os *Clusters* 1 e 2 podem ser definidos como 1KB (o menor tamanho utilizado nesta arquitetura). A partir disso, seguindo a Figura 19, o *Cluster 3* é mapeado para um *core* com 2KB de *cache*, o *Cluster 4* é mapeado para um *core* com 4KB de *cache* e o *Cluster 5* é mapeado para um *core* com 8KB de *cache*. A Tabela 12 resume o mapeamento.

Considerando esta configuração da arquitetura, a taxa global de *cache miss* e *hit* é apresentada na Figura 20. Esta figura ilustra um gráfico de comparação entre as taxas globais de *cache miss* e *hit* da arquitetura *multi-core* heterogênea (HT) e das arquiteturas *multi-cores* homogêneas tradicionais (HM1, HM2, HM4 e HM8). É possível ver que a arquitetura HT obteve um resultado próximo da HM8 a qual possui as melhores taxas para os casos apresentados. Porém, enquanto a HM8 usa 32KB de memória *cache* (8KB * 4 *cores*), a HT usa somente 15KB (menos que a HM4).

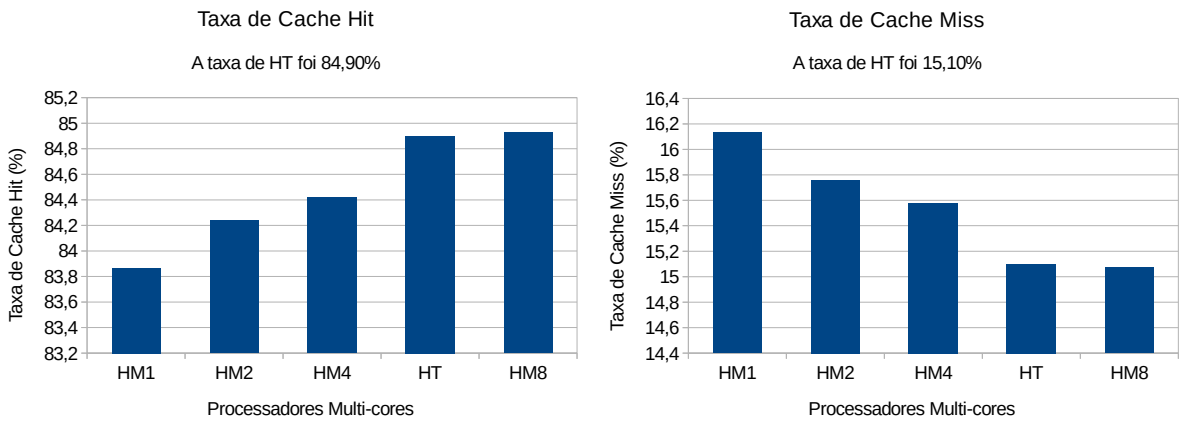

Figura 20 – Taxas de *cache* para a execução das 15 aplicações nas arquiteturas homogêneas (HM1 até HM8) e na heterogênea (HT).

Nesta seção, foi apresentado o fluxo da ferramenta proposta considerando o uso do DAMICORE. Neste fluxo, é feita uma análise *offline* de um conjunto de aplicações a fim de agrupá-las e gerar uma arquitetura *multi-core* heterogênea com tamanhos de *cache* adequados para executar as aplicações. Com os experimentos apresentados, foi possível perceber o potencial no uso da ferramenta DAMICORE e de HMPs com diferentes tamanhos de *cache*, podendo influenciar positivamente na relação desempenho e consumo de energia.

A próxima seção apresenta um experimento que amplia mais o conjunto de aplicações submetidas à ferramenta de análise *offline* e geração de código e compara a solução com um conjunto maior de arquiteturas.

5.2.3 Experimento com 50 aplicações

Este experimento avança o trabalho apresentado na seção anterior e apresenta uma técnica para guiar a definição da arquitetura *multi-core* mais adequada, em termos de consumo de energia e desempenho, para um conjunto de aplicações sem a necessidade de executar todas elas *a priori*. É possível então definir o número de *cores* e o tamanho da memória *cache* L1 para cada *core*.

5.2.3.1 Aplicações de referência

Escolher um bom conjunto para formar as aplicações de referência não é uma tarefa trivial uma vez que é necessário saber quais são as aplicações mais representativas em um dado conjunto. Entretanto, para o propósito deste experimento, foram escolhidas 10 aplicações como referência variando entre *CPU-Bound* e *Memory-Bound*, necessitando diferentes configurações de *cache* e cada *thread* executa um algoritmo diferente.

A Figura 21 apresenta a taxa de *cache miss* medida, o tempo de execução e

o consumo de energia para as aplicações de referência em uma arquitetura *single-core* variando entre sete possíveis configurações de *cache*: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB e 256KB.

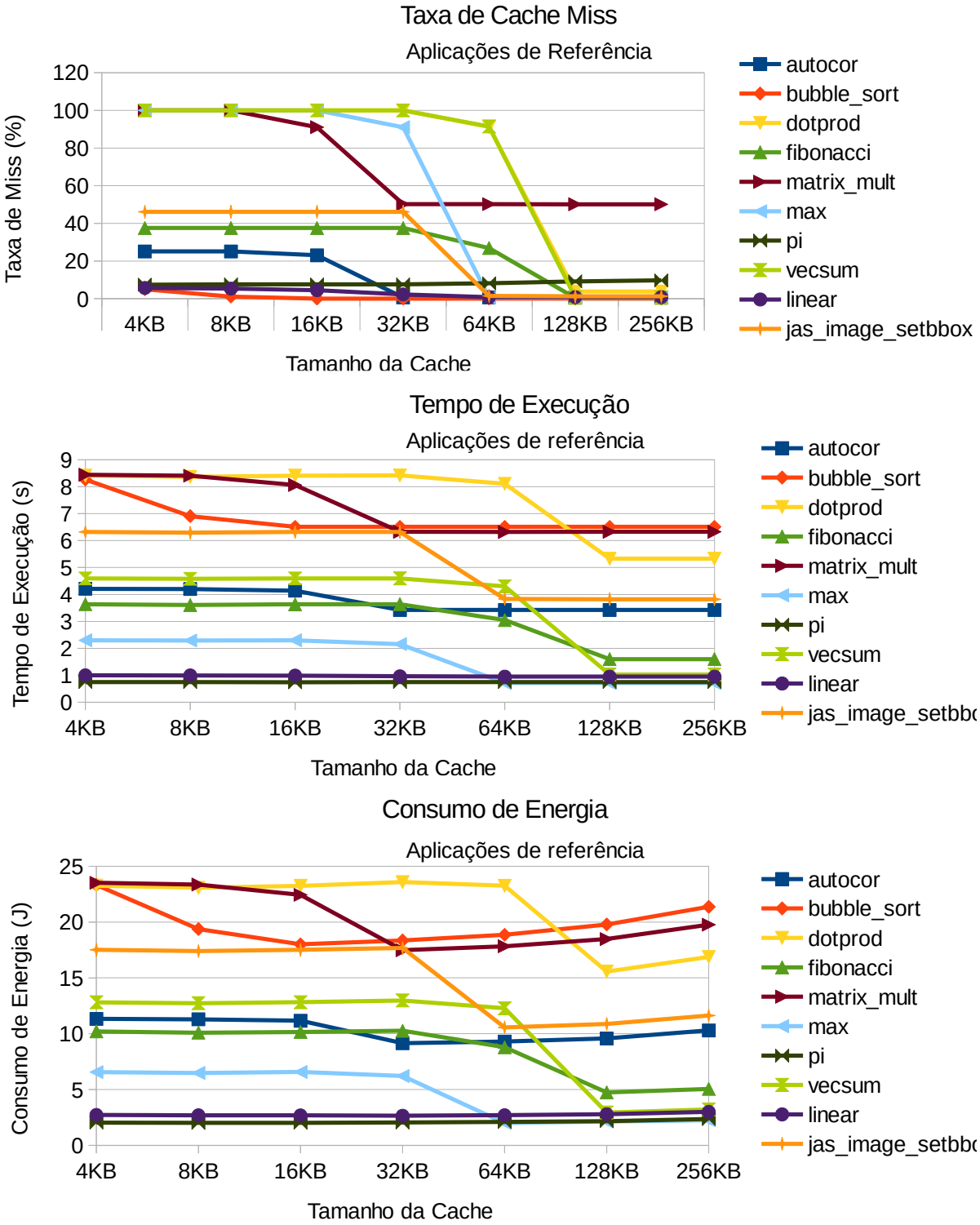

Figura 21 – Taxa de *cache miss*, tempo de execução e consumo de energia para diferentes tamanhos de *cache* das aplicações de referência.

Ao observar a Figura 21, é possível perceber que a taxa de *cache miss* das aplicações permanece aproximadamente a mesma conforme se aumenta o tamanho da *cache* até um

determinado ponto em que começa a cair. Este ponto depende do tamanho dos dados manipulados pelo algoritmo e da própria estrutura do algoritmo. Por exemplo, para o *vecsum*, a primeira queda significativa ocorre com 64KB de cache, já para *matrix_mult* ocorre com 16KB. Conforme se aumenta ainda mais a *cache*, a taxa de *cache miss* continua caindo até um ponto em que estabiliza. A partir da estabilização, já não é mais interessante aumentar o tamanho da *cache*, uma vez que uma *cache* maior sem reduzir a taxa de *miss* irá apenas consumir mais área e energia. Aplicações que são CPU-Bound não se beneficiam, ou se beneficiam pouco, do aumento da *cache*. Se observarmos a aplicação *pi*, que utiliza pouca memória, podemos ver que sua taxa de *cache miss* permanece estável mesmo quando se aumenta o tamanho da *cache*.

Fortemente correlacionados à taxa de *cache miss* estão o tempo de execução e o consumo de energia. Quando a taxa de *cache miss* cai, menos acesso é feito à memória externa e, portanto, o tempo de execução também cai, uma vez que o custo de tempo para acessar a memória externa é significativamente maior do que o tempo de se acessar um dado que já está na *cache*. Sendo assim, o menor tempo de execução coincide com a menor taxa de *cache miss*. Considerando o consumo de energia, o comportamento segue a mesma lógica, entretanto, a partir do momento em que a taxa de *cache miss* estabiliza e o tamanho da *cache* começa a aumentar, o consumo de energia também apresenta um aumento. Isso acontece porque uma *cache* maior consome mais energia e nesse ponto não se tem mais redução dos acessos à memória externa. Dessa forma, é possível definir qual é o tamanho ideal da *cache* para cada uma das aplicações de referência e estabelecer o ranqueamento de dominância, classificando as *caches* da melhor para a pior configuração.

5.2.3.2 Ranqueamento de dominância

Conforme já explicado anteriormente, o ranqueamento de dominância foi usado para representar qual é a melhor configuração de *cache* para cada aplicação. Esta técnica compara as configurações de *cache* contando quantas vezes cada configuração foi melhor e pior se comparada às outras configurações. A diferença entre as vezes em que uma configuração foi melhor e as vezes em que foi pior é a pontuação que uma dada configuração possui ao executar uma aplicação.

Por exemplo, na Figura 21, considerando o gráfico de tempo de execução para a aplicação *jas_image_setbbox*, a configuração de cache de 32KB é melhor do que as configurações de 4KB, 8KB e 16KB. Entretanto, a cache de 32KB é pior do que a de 64KB, 128KB e 256KB. Consequentemente, esta configuração ganhou três vezes e perdeu três vezes e, portanto, sua pontuação é três menos três, que é igual a zero. Agora, considerando a configuração de 64 KB também para a aplicação *jas_image_setbbox*, pode-se perceber que ela ganha 6 vezes e perde zero, possuindo 6 pontos. Baseando-se nestas pontuações, é possível afirmar que a *cache* de 64KB é melhor do que a de 32KB no que se refere

ao tempo de execução. O ranqueamento de dominância foi feito para cada configuração de *cache* considerando três critérios (taxa de *cache miss*, tempo de execução e consumo de energia). Então, para descobrir a pontuação para uma dada configuração de *cache*, considerando os três critérios ao mesmo tempo, basta somar a pontuação de cada um dos critérios. A Figura 22 apresenta o ranqueamento de dominância para cada um dos três critérios individualmente e também a soma deles. A vantagem em se usar este método é que todas as soluções podem ser ordenadas da melhor para a pior permitindo que sejam escolhidas soluções intermediárias se a melhor solução não está disponível em um determinado momento devido às restrições do hardware.

5.2.3.3 Clusters do DAMICORE e definição da arquitetura heterogênea

As dez aplicações de referência mais outras quarenta aplicações foram fornecidas como entrada para a ferramenta DAMICORE. Nestas cinquenta aplicações estão incluídos vários algoritmos de *benchmarks* da Texas Instruments (TMS320C64x Image/Video Processing Library e TMS320C64x DSP Library), e também algoritmos clássicos de ordenação e algoritmos matemáticos. A Figura 23 apresenta a saída do DAMICORE mostrada como um conjunto de *clusters*.

Tendo em mãos o ranqueamento de dominância das aplicações de referência e o conjunto de *clusters*, é possível iniciar a configuração do HMP. O primeiro passo é definir a configuração de *cache* para os *clusters* que possuem aplicações de referência. Neste passo, dois casos são considerados: *clusters* que possuem apenas uma aplicação de referência e *clusters* que possuem mais de uma aplicação de referência. Em seguida, o segundo passo é definir a configuração de *cache* para os *clusters* que não possuem nenhuma aplicação de referência. A Tabela 13 apresenta os *clusters* obtidos e as aplicações que os compõem. Iniciando pelo *Cluster 25*, que possui como aplicação de referência a aplicação *jas_image_setbbox*, ele deve executar em um *core* possuindo 128KB de *cache* preferencialmente (ou seja, a melhor configuração para esta aplicação considerando a taxa de *cache miss*, tempo de execução e consumo de energia ao mesmo tempo, de acordo com a Figura 22). Como o *Cluster 8* também possui somente uma aplicação de referência (*bubble_sort*), este *cluster* deverá ser executado em um *core* contendo 16KB de *cache*.

Para os casos em que um *cluster* possui mais do que uma aplicação de referência, a mesma ideia foi aplicada. Entretanto, em vez de se considerar o ranqueamento de dominância de apenas uma aplicação, é necessário considerar a composição do ranqueamento de todas elas, ou seja, a soma da pontuação. Dessa forma, para definir a configuração de *cache* para os *Clusters* 0, 61 e 76, o ranqueamento de dominância considerando todas as suas aplicações de referência juntas pode ser visto na Figura 24. Consequentemente, a configuração de *cache* em que os *Clusters* 0, 61 e 76 devem executar são, respectivamente, 64KB, 32KB e 128KB.

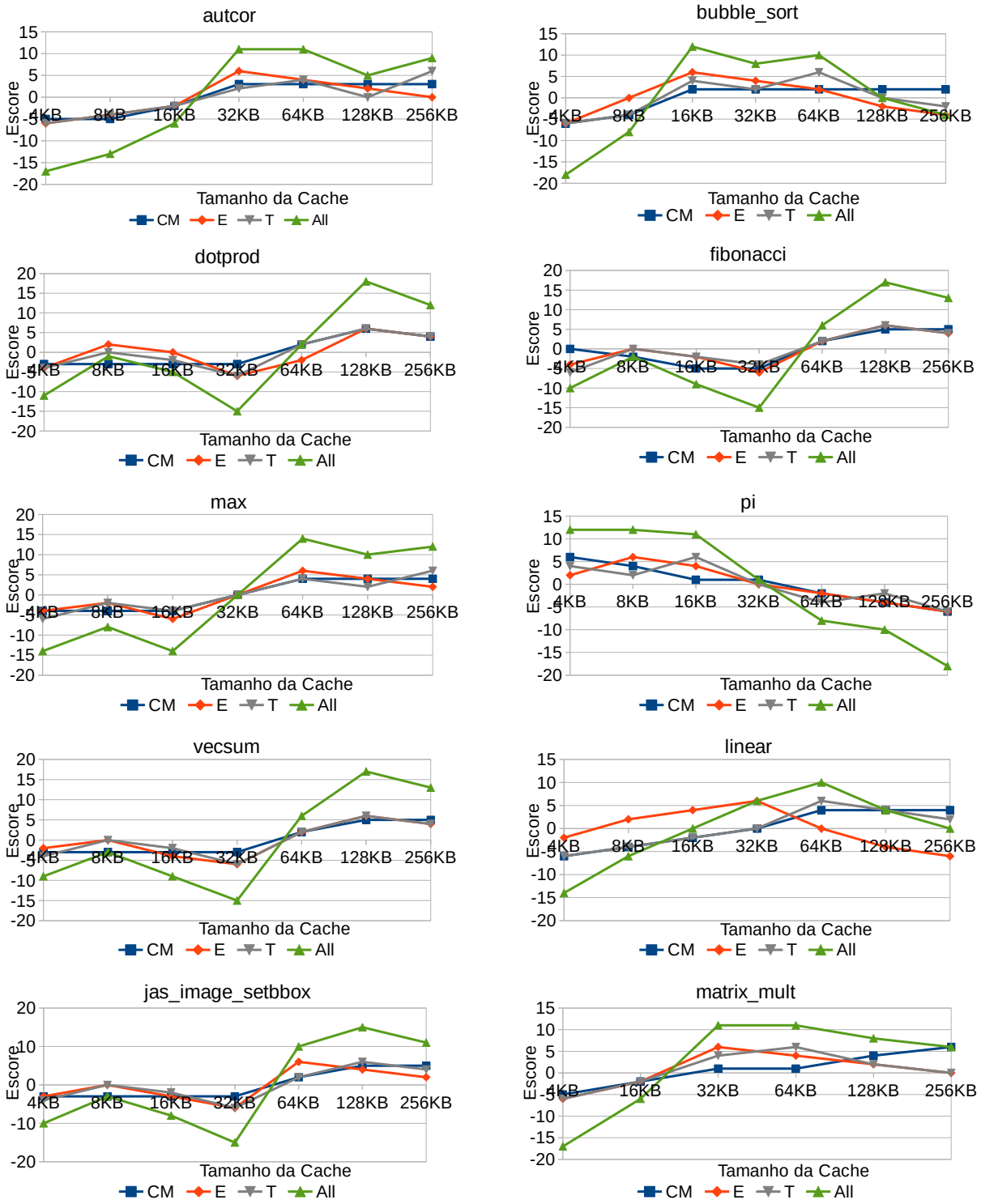

Figura 22 – Ranqueamento de dominância das aplicações de referência para os diferentes tamanhos de *cache* considerando os três critérios: taxa de *cache miss* (CM), tempo de execução (T) e consumo de energia (E).

Sabendo que aplicações mais próximas no agrupamento gerado pelo DAMICORE possuem maiores similaridades, a configuração de *cache* para os *clusters* que não possuem nenhuma aplicação de referência é definida como a mesma configuração de seu *cluster* vizinho mais próximo que possui aplicação de referência, de acordo com a Figura 23.

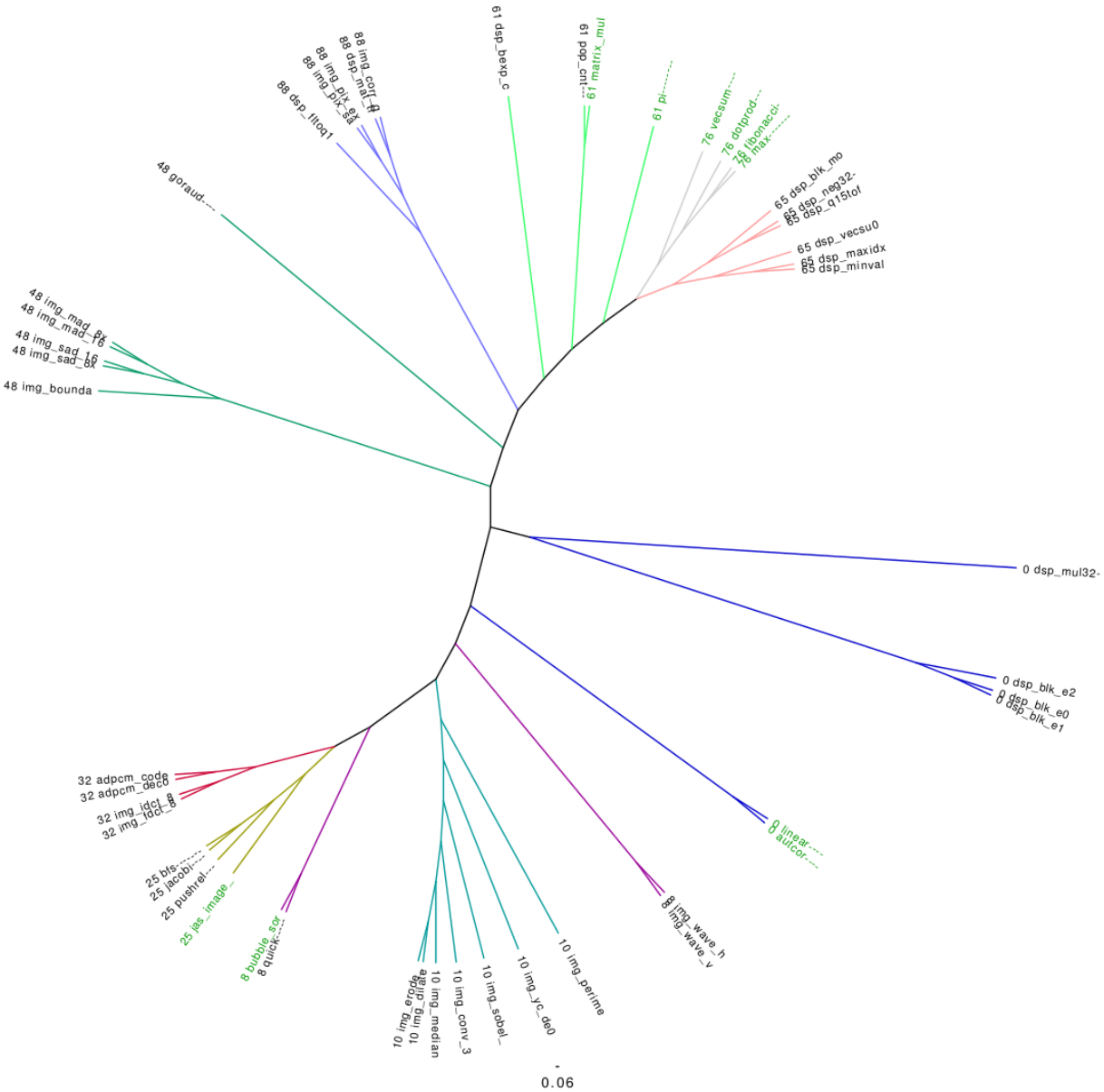

Figura 23 – Saída do DAMICORE para as cinquenta aplicações, incluindo as de referência. Grupos de aplicações apresentando alto grau de similaridade formam um *cluster*. O número antes de cada nome corresponde á identificação de cada *cluster*, identificado também pela cor das arestas.

Tabela 13 – Clusterização do DAMICORE para as 50 aplicações. As aplicações de referência estão em negrito. A numeração de cada *cluster* segue a numeração gerada automaticamente pela ferramenta DAMICORE.

Clusters	Aplicações
0	dsp_blk_e0, dsp_blk_e1, dsp_blk_e2, **linear**, **autcor**, dsp_mul32
8	img_wave_h, img_wave_v, **bubble_sor**, quick
10	img_yc_de0, img_sobel_, img_conv_3, img_median, img_dilate, img_erode, img_perime
25	jacobi—-, bfs———-, pushrel—, **jas_image**
32	img_fdct_8, img_idct_8, adpcm_deco, adpcm_code
48	img_sad_8x, img_sad_16, img_mad_16, img_mad_8x, img_bounda, goraud—-
61	**pi**———, pop_cnt—, **matrix_mul**, dsp_bexp_c
65	dsp_neg32, dsp_q15tof, dsp_blk_mo, dsp_maxidx, dsp_minval, dsp_vecsu0
76	**fibonacci-**, **max**———-, **dotprod**—, **vecsum**—-
88	img_pix_sa, img_pix_ex, dsp_mat_tr, img_corr_g, dsp_fltoq1

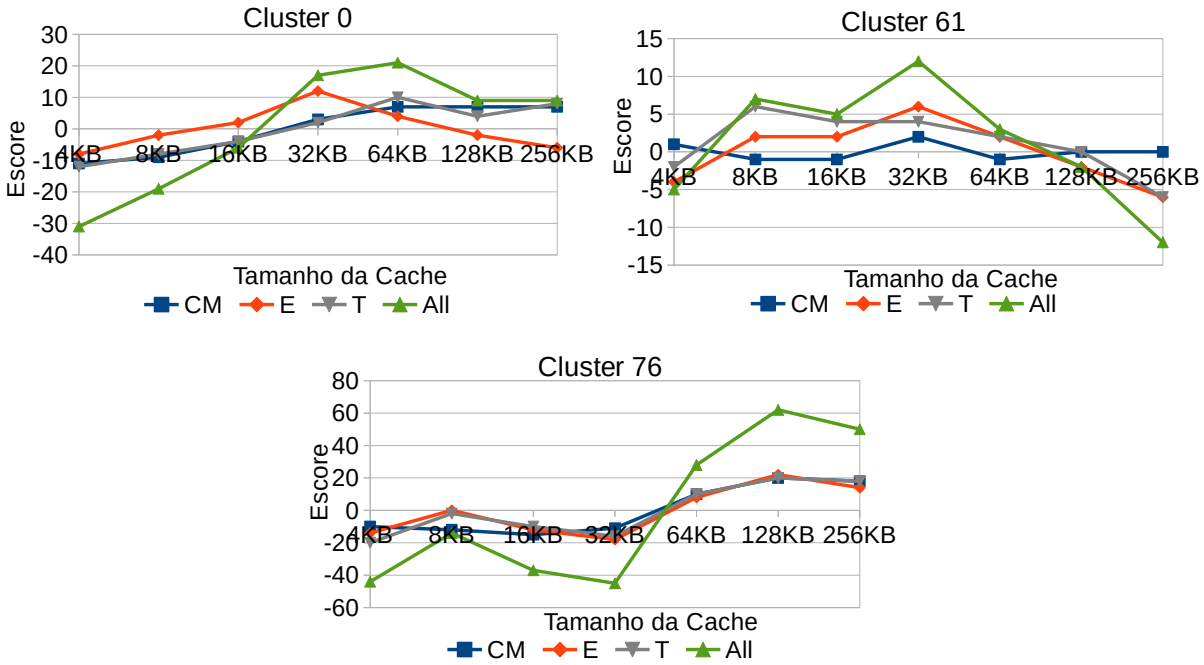

Figura 24 – Ranqueamento de dominância para *clusters* possuindo mais do que uma aplicação de referência considerando os três critérios: *cache miss* (CM), tempo de execução (T) e consumo de energia (E).

Consequentemente, como o *Cluster 10* está entre o *Cluster 8*, sua melhor configuração é também 16KB; o *Cluster 32* é 128KB (o vizinho mais próximo é o *Cluster 25*); o *Cluster 48* é 64KB (o vizinho mais próximo é o *Cluster 0*); o *Cluster* 88 é 32KB (o vizinho mais próximo é o *Cluster 61*) e, finalmente; o *Cluster 65* é 128KB (o vizinho mais próximo é o *Cluster 76*).

A solução HMP final é definida como o número mínimo de *cores* que satisfazem todas as configurações de *cache* encontradas para os *clusters*. Neste caso, quatro *cores* são suficientes contendo 16KB, 32KB, 64KB e 128KB de memória *cache*.

5.2.3.4 Política de escalonamento

Como já explicado anteriormente, a política de escalonamento padrão do eCos é a MLQ (Multi-Level Queue) e esta política considera todos os *cores* como iguais. Consequentemente, a MLQ não é capaz de explorar os benefícios de uma arquitetura HMP. Sendo assim, foi implementada uma nova política para o escalonador, que leva em consideração a configuração da arquitetura e a pontuação no ranqueamento de dominância. Então, o eCos é compilado e *linkado* à aplicação. Em tempo de execução, sempre que um *core* precisa escolher uma aplicação da fila de prontas, o escalonador consulta qual é a melhor para ser executada no *core* de acordo com o tamanho de sua *cache* e sua pontuação no ranqueamento de dominância.

5.2.3.5 Resultados experimentais

Para realizar as comparações neste experimento, foram sintetizadas em FPGA dez arquiteturas *multi-cores* homogêneas e duas heterogêneas, variando o número de *cores* e o tamanho das *caches*. As arquiteturas heterogêneas foram sintetizadas de acordo com a solução encontrada apresentada na seção anterior. As arquiteturas *multi-cores* homogêneas foram sintetizadas possuindo o mesmo número de *cores* das arquiteturas heterogêneas com o tamanho da *cache* sendo restringido por limitações físicas do FPGA e da ferramenta de síntese.

Os resultados comparando a execução das 50 aplicações em todas as arquiteturas de quatro *cores* sintetizadas podem ser vistos na Figura 25 e os resultados comparando todas as arquiteturas de cinco *cores* podem ser vistos na Figura 26. Os gráficos mostram para cada arquitetura a média de 10 execuções.

Nos gráficos da Figura 25 e da Figura 26, foi usada a seguinte convenção para os nomes das arquiteturas: HM 4C-32 (128), por exemplo, indica uma arquitetura *multi-core* homogênea (HM) que possui quatro *cores* (4C), onde cada *core* possui 32KB de *cache*, totalizando 128KB de memória em todo o sistema. HT indica uma arquitetura *multi-core* heterogênea e, neste caso, o tamanho da *cache* de cada *core* está no nome, também seguido pelo total de memória *cache* do sistema.

Analisando as arquiteturas *multi-cores* homogêneas de quatro e cinco *cores*, pode-se observar que, conforme o tamanho da *cache* aumenta para o mesmo número de *cores*, o tempo de execução e o consumo de energia diminui. Tal comportamento pode ser observado nas arquiteturas com quatro *cores* a partir de HM 4C-4 (16) até HM 4C-64 (256) para tempo de execução e a partir de HM 4C-4 (16) até HM 4C-32 (128) para consumo de energia. Similarmente, para as arquiteturas com cinco *cores*, a partir de HM 5C-4 (20) até HM 5C-128 (640) para tempo de execução e, considerando consumo de energia, a partir de HM 5C-4 (20) até HM 5C-32 (160). Quando o tamanho da *cache* aumenta, de um modo geral, mais dados podem ser armazenados na *cache* e menos acesso ocorre à memória externa. Consequentemente, menos ciclos de *clock* são necessários para ler um dado e menos energia é consumida pelo circuito de memória. Entretanto, a partir de um tamanho de *cache* em particular (64KB para quatro *cores* e 128KB para *cinco* cores) o consumo de energia tende a aumentar, embora o tempo de execução continua caindo (ou permanece estável). Isto significa que, para o conjunto de aplicações utilizado neste experimento, a partir destes pontos específicos, não é mais interessante aumentar a *cache* em termos de consumo de energia. Ou seja, o consumo de energia extra causado pelo aumento do tamanho da *cache* é maior do que a energia economizada pela redução de acessos à memória externa. Este é um comportamento típico de qualquer sistema que utiliza *cache*, porém encontrar este ponto ideal automaticamente pode não ser uma tarefa fácil. Se trata, especificamente, de um problema NP-Completo para o caso de arquiteturas

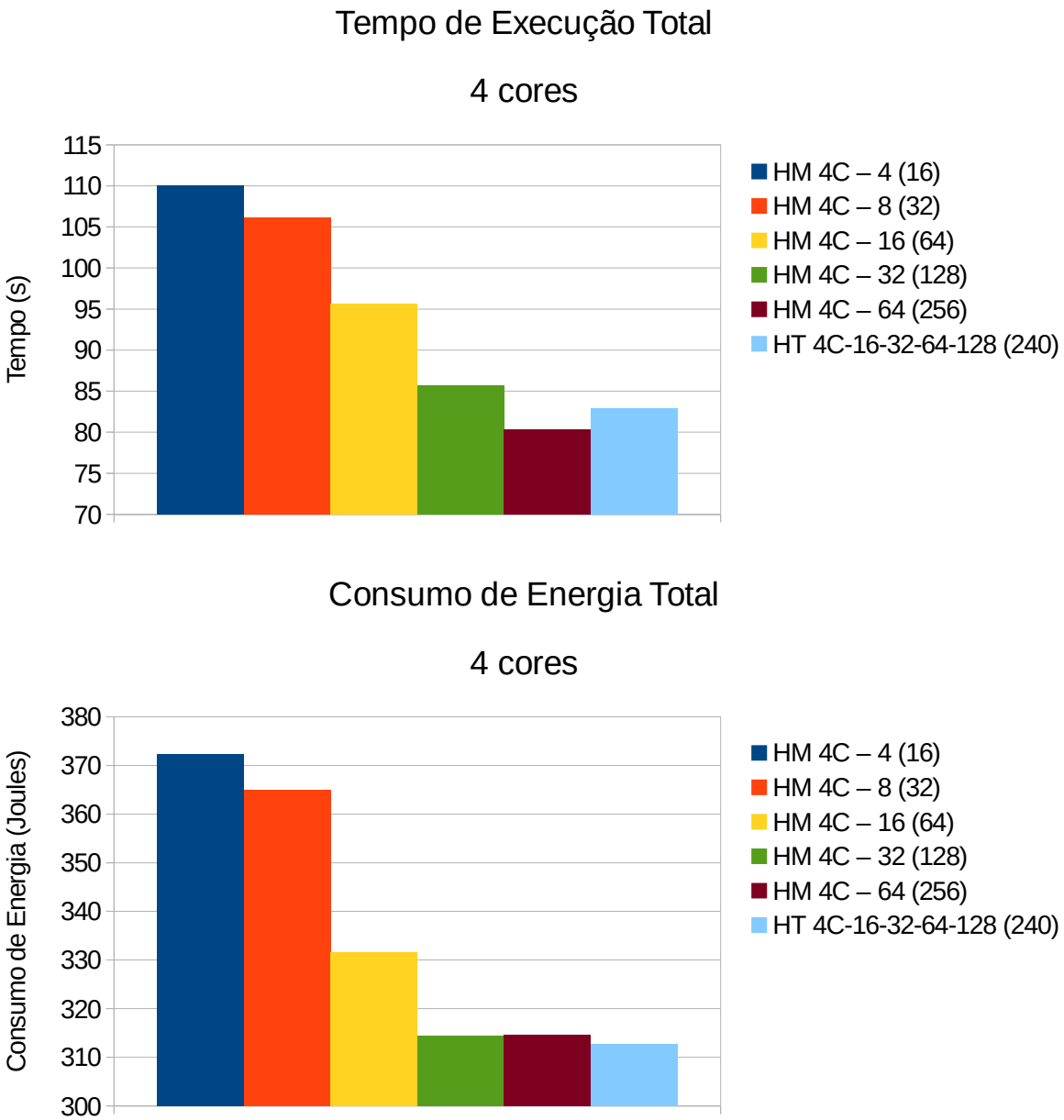

Figura 25 – Tempo de execução total (segundos) e consumo de energia (Joules) para as 50 aplicações executadas em todas as arquiteturas de quatro *cores* sintetizadas.

multi-cores (SOUISSI; ABDELHAKIM, 2013).

Como pode ser visto ainda na Figura 25, a HT 4C-16-32-64-128 (240) atingiu o menor consumo de energia para as arquiteturas de quatro *cores*, sendo 0,61% melhor do que a HM 4C-32 (128). Para o tempo de execução, o HT 4C-16-32-64-128 (240) apresentou um aumento de aproximadamente 3,3%, também comparada à HM 4C-64 (256). Dessa forma, o tempo de execução para o HT 4C-16-32-64-128 (240) foi o segundo melhor tempo. Entre as vantagens da arquitetura *multi-core* heterogênea encontrada pelo método proposto comparada às arquiteturas homogêneas de quatro *cores* está a capacidade de possuir pelo menos um *core* com *cache* de 128KB. Consequentemente, as aplicações que

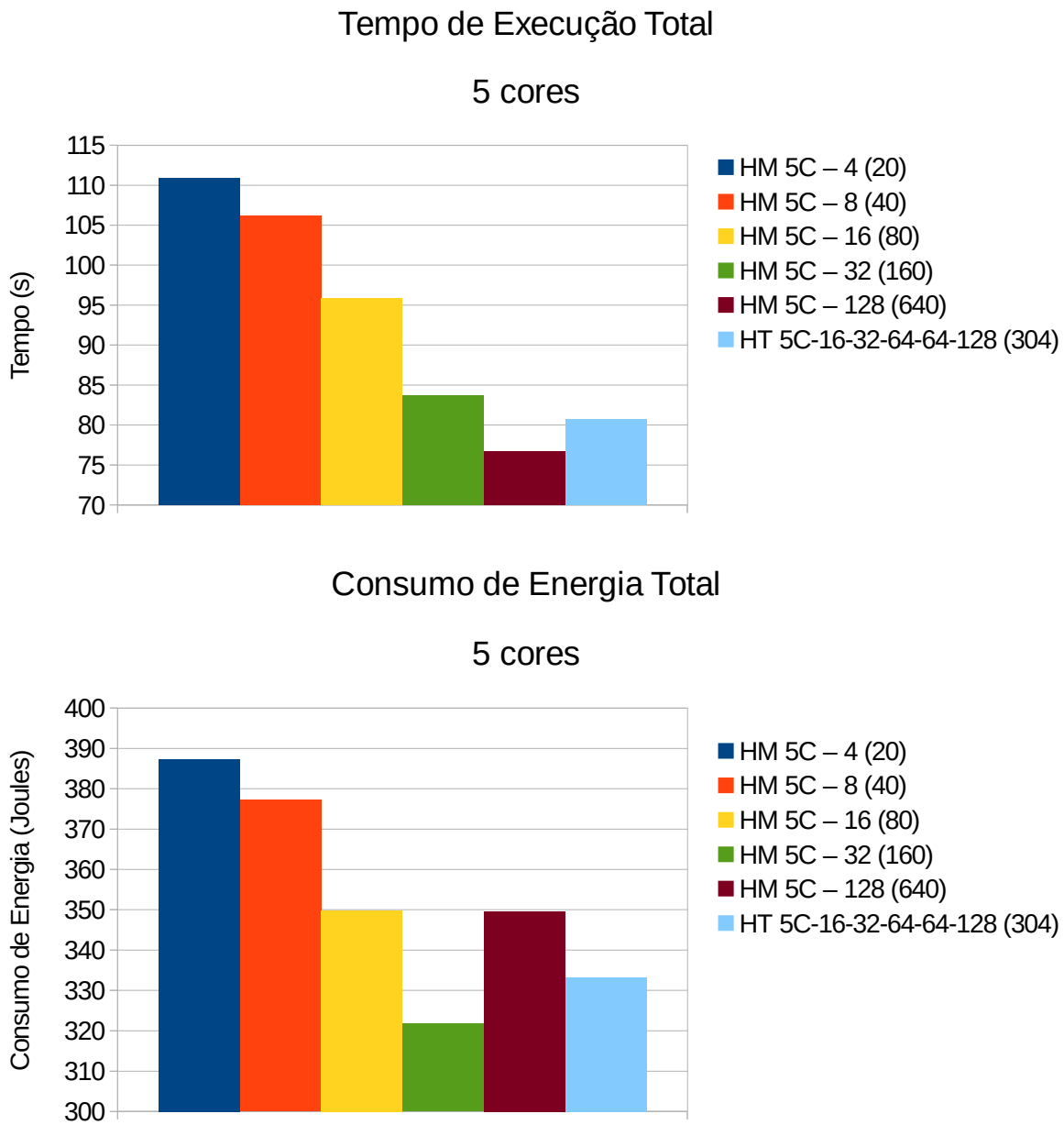

Figura 26 – Tempo de execução total (segundos) e consumo de energia (Joules) para as 50 aplicações executadas em todas as arquiteturas de cinco *cores* sintetizadas.

necessitam mais do que 64KB de *cache* (por exemplo, as aplicações no *Cluster 76* - Figura 23 e Tabela 13) podem ser escalonados para um *core* mais adequado. Além disso, a solução dada pelo método proposto apresentou uma leve redução no consumo de energia com pouca degradação no tempo de execução.

A Figura 26 apresenta os resultados para uma variação da solução encontrada pelo método proposto. Considerando o mapeamento final das aplicações, existem muitas aplicações que serão escalonadas para serem executadas em *cores* com *cache* de 128KB. Em uma tentativa de melhor balancear as aplicações entre os *cores*, é possível adicionar, por exemplo, outro *core* de 128KB no HMP. Entretanto, a ferramenta de síntese não conseguiu

gerar esta arquitetura heterogênea com um *core* extra de 128KB devido à impossibilidade de rotear fisicamente os componentes no FPGA. Sendo assim, foi adicionado um *core* extra de 64KB em vez de 128KB, uma vez que este tamanho também representa uma boa solução para tais aplicações de acordo com o ranqueamento de dominância (Figura 22). A arquitetura HT 5C-16-32-64-64-128 (304) atingiu o segundo melhor tempo de execução e o segundo menor consumo de energia se comparada a todas as arquiteturas de cinco *cores*. Para o tempo de execução, HT 5C-16-32-64-64-128 (304) é 3,54% melhor do que HM 5C-32 (160) e, para o consumo de energia, é 4,67% melhor do que HM 5C-128 (640). Portanto, esta solução heterogênea é um meio termo entre HM 5C-32 (160) e HM 5C-128 (640). A arquitetura homogênea de cinco *cores* contendo 64KB de *cache* por *core* provavelmente estaria localizada também entre HM 5C-32 (160) e HM 5C-128 (640), porém a arquitetura HM 5C-64 (320) também não pôde ser sintetizada devido às limitações da ferramenta de síntese. Como esta arquitetura não pôde ser sintetizada, usar a solução heterogênea é vantajoso para atingir uma solução que está no meio termo.

Comparando somente as arquiteturas heterogêneas, pode-se perceber que ambas as soluções são boas em termos de consumo de energia e tempo de execução, porém cada uma sobressai em um critério diferente. HT 4C-16-32-64-128 (240) foi 6,16% mais eficiente no consumo de energia e HT 5C-16-32-64-64-128 (304) foi 2,63% mais eficiente em tempo de execução.

A Figura 27 apresenta uma comparação entre todas as arquiteturas *multi-cores* sintetizadas para este experimento. A solução fornecida pela técnica proposta, HT 4C-16-32-64-128, atingiu o menor consumo de energia. Entretanto, considerando tanto consumo de energia quanto tempo de execução, as quatro melhores soluções são HM 4C-64 (256), HT 4C-16-32-64-128 (240), HM 5C-128 (640) e HT 5C-16-32-64-64-128 (304).

Com a técnica apresentada, é possível gerar arquiteturas *multi-cores* orientadas para a aplicação automaticamente, otimizando o consumo de energia e o tempo de execução. Embora a solução gerada não tenha sido a melhor, ela é relativamente boa. A solução heterogênea é interessante em contextos onde existem restrições no tempo de execução, consumo de energia, número de *cores* ou quantidade total de memória *cache*. Por exemplo, considerando que a memória *cache* é cara em termos de uso de recursos em um processador, como HT 4C-16-32-64-128 (240) usa 16KB a menos do que a melhor solução verificada (HM 4C-64 (256)), pode ser mais lucrativo utilizar a heterogênea em vez da solução ótima, mesmo sabendo que o tempo de execução terá uma leve degradação. O mesmo acontece com a HT 5C-16-32-64-64-128 (304), a qual usa menos do que metade da memória usada por HM 5C-128 (640) ao mesmo tempo em que possui uma eficiência no consumo de energia similar. Estas restrições estão presentes no projeto de diversos sistemas, tais como sistemas embarcados.

A técnica desenvolvida demonstrou ser eficiente para definir uma arquitetura HMP,

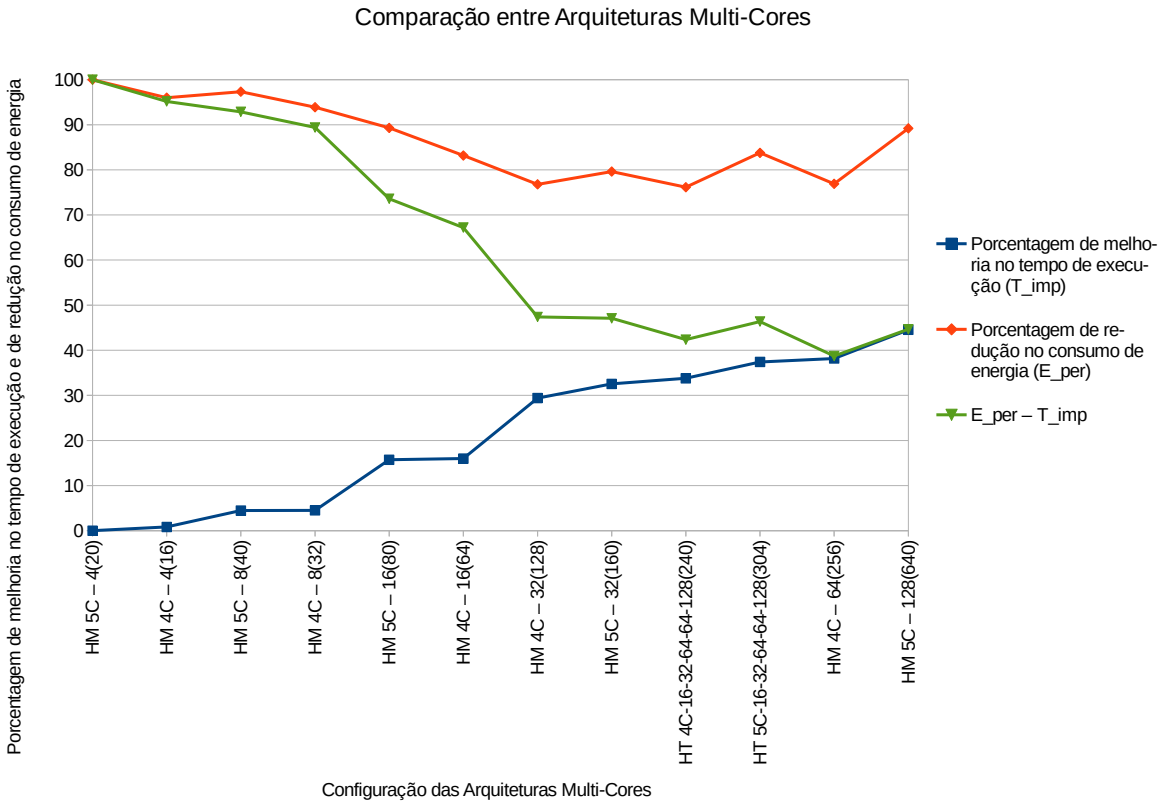

Figura 27 – Comparação entre as arquiteturas *multi-cores*. O eixo x apresenta todas as arquiteturas homogêneas e heterogêneas. O eixo y apresenta a porcentagem de melhorias para o tempo de execução e a porcentagem de redução no consumo de energia ambos comparados ao valor máximo obtido (neste caso, a pior arquitetura tanto em tempo de execução quanto em consumo foi HM 5C-4 (20)). A linha verde ($E\_per - T\_imp$) é apenas para representar a distância entre as porcentagens de redução no consumo de energia (E_per) e melhoria no tempo de execução (T_imp) para efeito de visualização. Para este experimento, quanto menor essa distância, melhor é uma dada arquitetura *multi-core*, uma vez que ela apresenta ao mesmo tempo baixo consumo de energia e alta melhoria no tempo de execução.

uma vez que não necessita de execução prévia de todas as aplicações para realizar a análise *offline*. Além disso, o agrupamento das aplicações em formato de *cluster* beneficia as decisões do escalonador em tempo de execução.

Os resultados demonstram que, para um conjunto de aplicações pertencendo a um conjunto de *benchmarks* de processamento de imagem, vídeo e sinais, melhorias foram atingidas em relação ao consumo de energia e recursos de memória *on-chip* comparados às arquiteturas *multi-cores* heterogêneas usando o mesmo número de *cores*. Este tipo de otimização é importante para a maioria dos sistemas computacionais, principalmente no cenário de sistemas embarcados onde melhorias na eficiência do consumo de energia é desejável, ou até mesmo fundamental.

Finalmente, este experimento demonstrou que a integração entre análise *offline* e o ambiente em tempo de execução é fundamental a fim de ser capaz de gerar arquiteturas apropriadas para as demandas da aplicação e, então, ser capaz de explorar tais recursos

durante a sua execução.

5.2.4 Análise estatística sobre o escalonamento e a arquitetura gerada para as 50 aplicações

Para realizar esta análise estatística, foram realizados experimentos com as 50 aplicações já mencionadas anteriormente e foram aleatoriamente definidas e sintetizadas uma amostra de 11 arquiteturas *multi-cores* homogêneas e outra amostra contendo 11 arquiteturas heterogêneas, incluindo a solução encontrada pela ferramenta apresentada neste trabalho. As configurações das arquiteturas variaram o número de *cores* entre 2 e 5 e os tamanhos das memórias *cache* até 128KB por *core*. O objetivo principal foi observar o comportamento médio da classe de arquiteturas heterogêneas usando o escalonador proposto neste trabalho comparado ao comportamento médio das classes de arquiteturas homogêneas. Além disso, foi analisado como o tempo de execução e o consumo de energia estão relacionados ao número de *cores* e ao total de capacidade da memória *cache*.

A Figura 28 apresenta uma comparação entre todas as arquiteturas divididas em três categorias: homogêneas com a implementação original do eCos (*HM*), heterogênea com a implementação original do eCos (*HT-original eCos*), e heterogênea com o eCos modificado que implementa a política de escalonamento proposta neste trabalho (*HT-modified eCos*). A Figura 28 revela que os *HMs* apresentam os piores resultados em média (102,29 segundos e 354,54 Joules), *HT-original eCos* apresenta em média uma solução intermediária (94,81 segundos e 339,61 Joules) e *HT-modified eCos* atingiu os melhores resultados em média (91,6 segundos e 325,33 Joules).

Para comparar estatisticamente os resultados para estas categorias de arquiteturas foi utilizado o teste de hipóteses *Wilcoxon signed-rank* (DEVORE, 2011) tanto para tempo de execução quanto para consumo de energia. Foram avaliadas duas hipóteses para as variações de arquitetura e escalonamento, que são: a hipótese nula H_0, significando que duas classes provavelmente possuem os mesmos valores de média; e a hipótese alternativa H_1 que significa que uma das classes possui a média menor do que a outra.

Tabela 14 – Valores de $p-value$ obtidos pelo teste de hipóteses *Wilcoxon signed-rank* comparando as arquiteturas homogêneas e heterogêneas em relação ao tempo e energia consumidos.

Comparação do $p-value$	**Tempo de execução**	**Consumo de energia**
HT-original eCos vs. HM	0,135	0,096
HT-modified eCos vs. HM	0,044	0,008

A Tabela 14 sumariza o teste de hipóteses primeiramente comparando *HT-original eCos* com *HM-original eCos* e, em seguida, comparando *HT-modified eCos* com *HM-original eCos*, ambos para um intervalo de confiança de 95% ($\alpha = 0,05$). Para a comparação entre *HM-original eCos* e *HT-original eCos*, como o valor de $p-value$ é maior do que

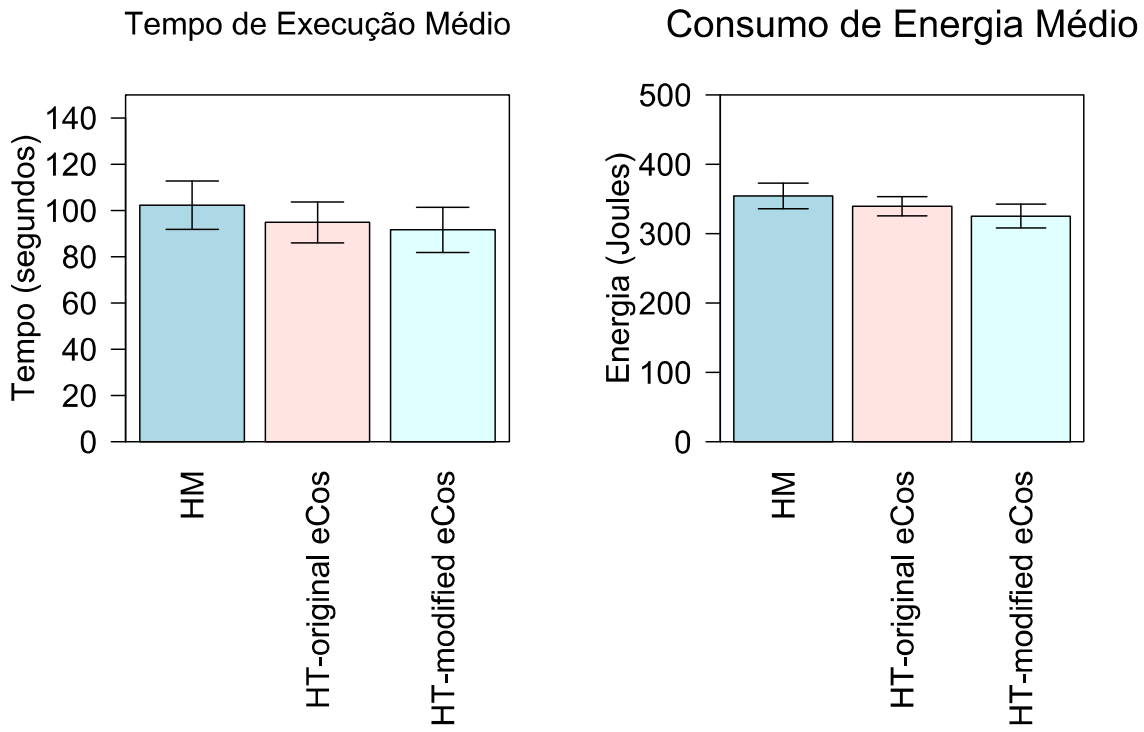

Figura 28 – Comparação entre três categorias de arquiteturas *multi-cores* em relação ao tempo de execução (esquerda) e o consumo de energia (direita).

α, não é possível descartar a hipótese nula e, consequentemente, é razoável concluir que ambas as médias são iguais (para tempo e energia). Entretanto, considerando a comparação entre *HM-original eCos* e *HT-modified eCos*, como $p-value$ é menor do que α, a hipótese nula pode ser descartada e a hipótese alternativa pode ser assumida como verdadeira. Em outras palavras, de acordo com o teste de hipóteses, a classe de arquiteturas heterogêneas é melhor do que a classe de arquiteturas homogêneas somente quando usa o escalonador proposto neste trabalho.

Note que de acordo com os resultados, se forem selecionadas aleatoriamente uma arquitetura *multi-core* heterogênea existe uma considerável probabilidade de que ela seja melhor do que um sistema *multi-core* homogêneo definido aleatoriamente, mesmo quando usado um escalonado *heterogeneity-unaware*. Este fato pode ser explicado pela natureza de cada categoria. Para arquiteturas homogêneas, o número de configurações possíveis é relativamente pequeno para cada número de *cores* e cresce linearmente conforme cresce o número de *cores*. Todavia, para as arquiteturas heterogêneas, o número de configurações possíveis cresce exponencialmente de acordo com o número de *cores*. Sendo assim, amostras de arquiteturas heterogêneas definidas por amostragem aleatória simples podem ter configurações com mais *cores*, mais *cache*, e potencialmente melhor desempenho do que amostras aleatórias simples de arquiteturas homogêneas.

A Figura 29 apresenta a distribuição estatística dos resultados de tempo e energia

consumidos pelas arquiteturas homogêneas e heterogêneas apresentadas neste estudo. Em geral, conforme o número de *cores* aumenta, o tempo de execução diminui (Figura 29, esquerda). A diminuição no tempo de execução também leva a uma diminuição na energia consumida até quatro *cores*, aumentado após isso já que a energia consumida pelas arquiteturas de cinco *cores* é maior do que a economia causada pela paralelização da execução (Figura 29, direita). O consumo de energia e o desempenho conforme o tamanho das *caches* aumenta exibem comportamento similar. *Caches* maiores tendem a levar a um decréscimo na taxa de *cache miss* e assim reduzem a energia consumida. Entretanto, quando o tamanho da cache é maior do que o necessário pelas aplicações, a taxa de *cache miss* estagna e a *cache* não utilizada consome energia sem trazer benefícios. Boas soluções, portanto, devem ser capazes de realizar este compromisso.

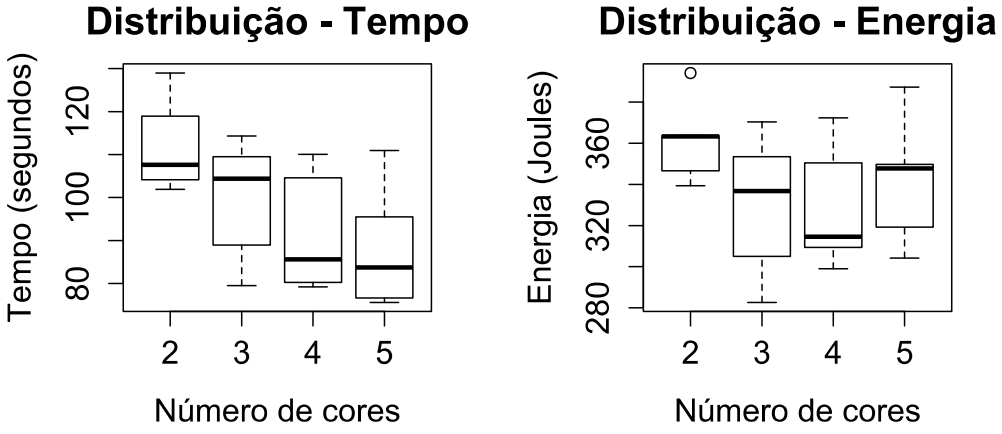

Figura 29 – Distribuição estatística dos resultados para tempo e energia *versus* número de *cores*.

Uma comparação entre todas as arquiteturas sintetizadas neste experimento em relação ao tempo e a energia pode ser vista na Figura 30 e 31, respectivamente. Foi usada a seguinte convenção para os nomes das arquiteturas: HM4-32 (128), por exemplo, indica uma arquitetura *multi-core* homogênea (HM) que possui quatro *cores* (4), onde cada *core* possui 32KB de *cache*, totalizando 128KB de memória em todo o sistema. HT indica uma arquitetura *multi-core* heterogênea e, neste caso, o tamanho da *cache* de cada *core* está no nome, também seguido pelo total de memória *cache* do sistema. A arquitetura heterogênea derivada usando a ferramenta proposta neste trabalho possui quatro *cores* com as *caches* L1 com as seguintes capacidades: 16, 32, 64 e 128 KBytes. As linhas horizontais nos gráficos representam, respectivamente, a média geral do tempo de execução de $96,95$ segundos e energia usada de $339,93$ Joules para todas as arquiteturas heterogêneas sintetizadas. Em termos de tempo de execução, HT5-128,64,64,32,32(320) atingiu a melhor média ($75,63$ segundos) e a arquitetura definida pela ferramenta proposta neste trabalho - HT4-16,32,64,128(240) - apresentou a quarta melhor média ($79,24$ segundos). A arquitetura HT3-16,64,128(208) atingiu o menor consumo de energia de $282,65$ Joules

e HT4-16,32,64,128(240) atingiu o segundo melhor consumo com uma média de 298,98 Joules.

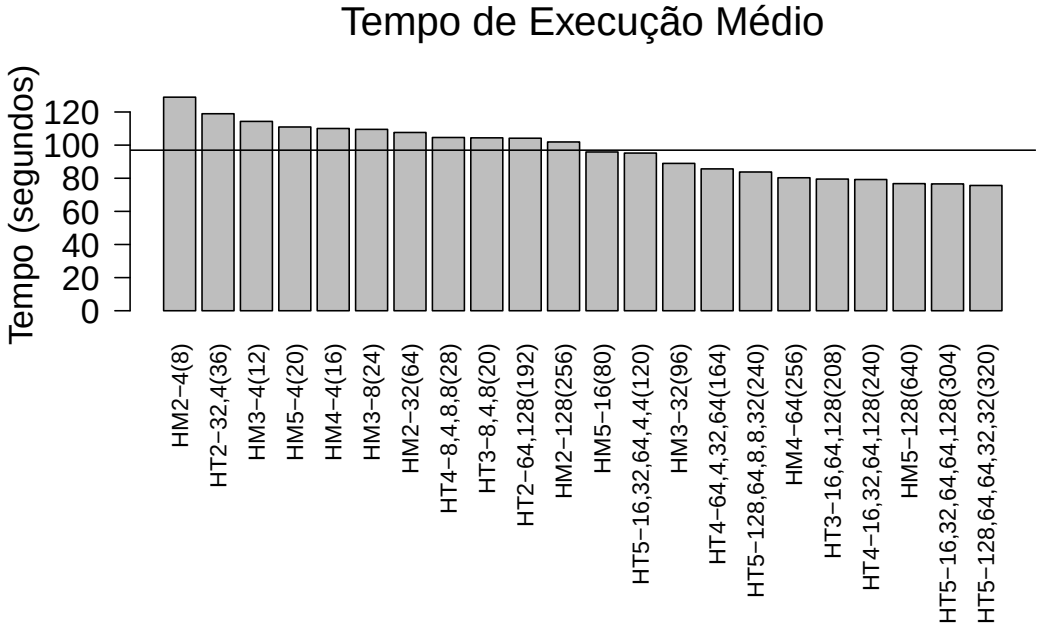

Figura 30 – Tempo de execução médio para as arquiteturas homogêneas com a implementação original do eCos e para as arquiteturas heterogêneas utilizando a implementação modificada do eCos. A linha horizontal no gráfico representa a média do tempo de execução dessas arquiteturas.

Estes resultados revelam que as arquiteturas heterogêneas derivadas usando a ferramenta para síntese de arquiteturas e o escalonamento de aplicações proposto neste trabalho atingiu uma redução de 18,27% no tempo de execução e 12% de redução no consumo de energia se comparado à média geral de todas as arquiteturas. A arquitetura HT4-16,32,64,128(240) é a quarta melhor em termos de tempo de execução e a segunda melhor em termos do consumo de energia. Comparando HT4-16,32,64,128(240) com a melhor média em tempo de execução, o teste de hipótese *Wilcoxon signed-rank* revela um $p-value$ de 0.06177 ($\alpha = 0.05$). Consequentemente, a hipótese nula (que considera que as duas médias são iguais) não pode ser descartada. Em relação ao consumo de energia, o $p-value$ obtido é de 0.004883, e, portanto, as duas médias podem ser consideradas diferentes. Sendo assim, a solução heterogênea derivada da ferramenta proposta neste trabalho é estatisticamente equivalente à melhor arquitetura em termos de tempo de execução atingindo o segundo melhor consumo de energia.

Comparar as classes de arquiteturas é importante para mostrar que arquiteturas heterogêneas combinadas a um escalonamento eficiente é capaz de obter uma eficiência melhor do que arquiteturas homogêneas. Entretanto, analisar arquiteturas heterogêneas comparadas com grupos de arquiteturas com números de *cores* e tamanhos de *caches* similares às heterogêneas pode revelar mais sobre seus comportamentos, como pode ser

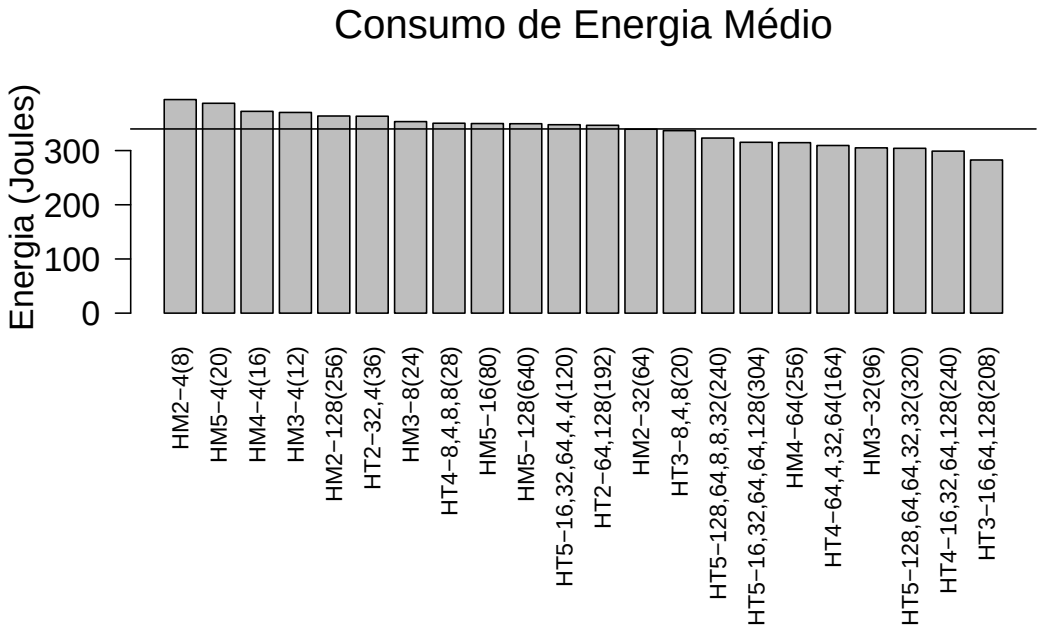

Figura 31 – Consumo de energia médio para as arquiteturas homogêneas com a implementação original do eCos e para as arquiteturas heterogêneas utilizando a implementação modificada do eCos. A linha horizontal no gráfico representa a média do consumo de energia dessas arquiteturas.

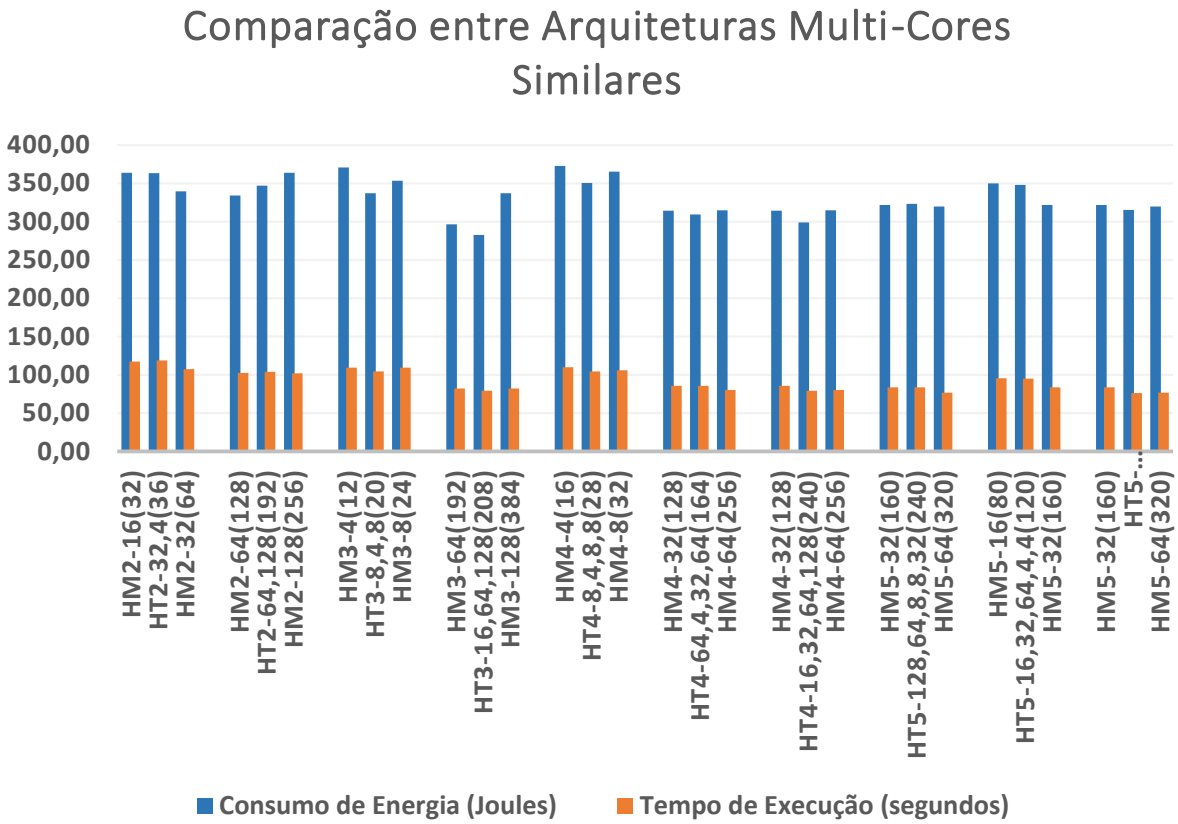

Figura 32 – Comparação entre as arquiteturas heterogêneas e homogêneas com número de *cores* e tamanhos de *cache* similares.

visto na Figura 32.

Cada arquitetura heterogênea foi comparada com as duas arquiteturas homogêneas mais próximas com o mesmo número de *cores*: um com menor *cache* e outro com maior. Em metade dos casos, as arquiteturas heterogêneas obtiveram melhor desempenho e menor consumo de energia que as duas arquiteturas homogêneas (HT3-8,4,8 (20), HT3-16,64,128 (208) HT4-64,4,32, 64 (164), HT4-16,32,64,128 (240) HT5-16,32,64,64,128 (304)). Consequentemente, uma exploração eficiente das arquiteturas heterogêneas feita pelo escalonador permite maior flexibilidade para definir a quantidade total de memória distribuída entre os *cores* de acordo com as necessidades da aplicação (economizando recursos de memória) obtendo melhores resultados que arquiteturas homogêneas em muitos casos.

5.2.5 Comparação com mais arquiteturas e way shutdown dinâmico como uma técnica de otimização complementar

Este experimento tem por objetivo trazer as seguintes contribuições em relação ao experimento anterior:

- Explorar melhor as análises estatísticas entre os dados, apresentando mais dados para comparação;
- Acrescentar a técnica para realizar *way-shutdown* dinâmico, além do processo utilizado no experimento anterior.

Foram geradas 28 arquiteturas homogêneas variando de um até seis *cores* e foram geradas 10 arquiteturas *multi-cores* heterogêneas de forma aleatória, além das duas arquiteturas que foram dadas como solução pelo método apresentado (com *way-shutdown* e sem *way-shutdown*), que são:

- Homogêneas:
  - 1 *core*: HM 1C – 4 (4), HM 1C – 8 (8), HM 1C – 16 (16), HM 1C – 32 (32), HM 1C – 64 (64), HM 1C – 128 (128);
  - 2 *cores*: HM 2C – 4 (8), HM 2C – 8 (16), HM 2C – 16 (32), HM 2C – 32 (64), HM 2C – 64 (128), HM 2C – 128 (256);
  - 3 *cores*: HM 3C – 4 (12), HM 3C – 8 (24), HM 3C – 32 (96), HM 3C – 64 (192);
  - 4 *cores*: HM 4C – 4 (16), HM 4C – 8 (32), HM 4C – 16 (64), HM 4C – 32 (128), HM 4C – 64 (256);
  - 5 *cores*: HM 5C – 4 (20), HM 5C – 8 (40), HM 5C – 16 (80), HM 5C – 32 (160), HM 5C – 128 (640);

  - 6 *cores*: HM 6C – 4 (24), HM 6C – 16 (96).

- Heterogêneas:
  - 2 *cores*: HT 2C 64-128 (192), HT 2C 32-4 (36);
  - 3 *cores*: HT 3C 16-64-128 (208), HT 3C 8-4-8 (20);
  - 4 *cores*: HT 4C 64-4-32-64 (164), HT 4C 8-4-8-8 (28);
  - 5 *cores*: HT 5C 16-32-64-64-128 (304), HT 5C 16-32-64-4-4 (120), HT 5C 32-16-2-2-8 (60), HT 5C 128-64-64-32-32 (320).
  - Solução com a ferramenta proposta neste trabalho:
    * HT 4C 16-32-64-128 (240) sem *way-shutdown* - HTDM;
    * HT 4C 16-32-64-128 (240) com *way-shutdown* - HTDM-WS.

No total, foram geradas 40 arquiteturas. Para cada arquitetura, foram realizadas 10 execuções das 50 aplicações, totalizando 400 execuções. A ordem de criação das aplicações definida de forma aleatória.

A Figura 33 apresenta um gráfico comparando a média dos valores de tempo de execução e consumo de energia para todas as 40 arquiteturas sintetizadas. Como o objetivo é minimizar tanto o consumo quanto energia, quanto menores estes valores, melhor. Portanto, os pontos mais próximos à origem do gráfico representam resultados mais interessantes. Isoladamente, com os piores resultados, estão as arquiteturas com apenas um *core*. Embora ocorra melhora quando se aumenta a *cache* até 32KB, todas as 50 aplicações são executadas sequencialmente. Após 32KB de *cache* já se observa um aumento no consumo de energia sem melhora significativa no tempo de execução. A grande maioria das arquiteturas apresenta consumo de energia entre 300 e 400 Joules e desempenho entre 80 e 120 segundos. A média geral entre todas as arquiteturas está representada no gráfico por um triângulo azul escuro (MÉDIA GERAL, na legenda do gráfico) e seus valores são de aproximadamente 112 segundos e 373 Joules para tempo de execução e consumo de energia, respectivamente. As arquiteturas que apresentaram consumo de energia menor do que 300 Joules foram as seguintes:

- HT 4C 16-32-64-128 (240) – *way-shutdown* = 296,7429;
- HM 3C – 64 (192) = 296,4375.

E as arquiteturas que apresentaram tempo de execução menor do que 80 segundos foram:

- HT 4C 16-32-64-128 (240) – *way-shutdown* = 78,403;

- HM 5C – 128 (640) = 76,755.

Se os valores para as arquiteturas *single-core* forem desconsiderados, então a média geral passa a ser 99,156 segundos para tempo de execução e 345,65 Joules para consumo de energia.

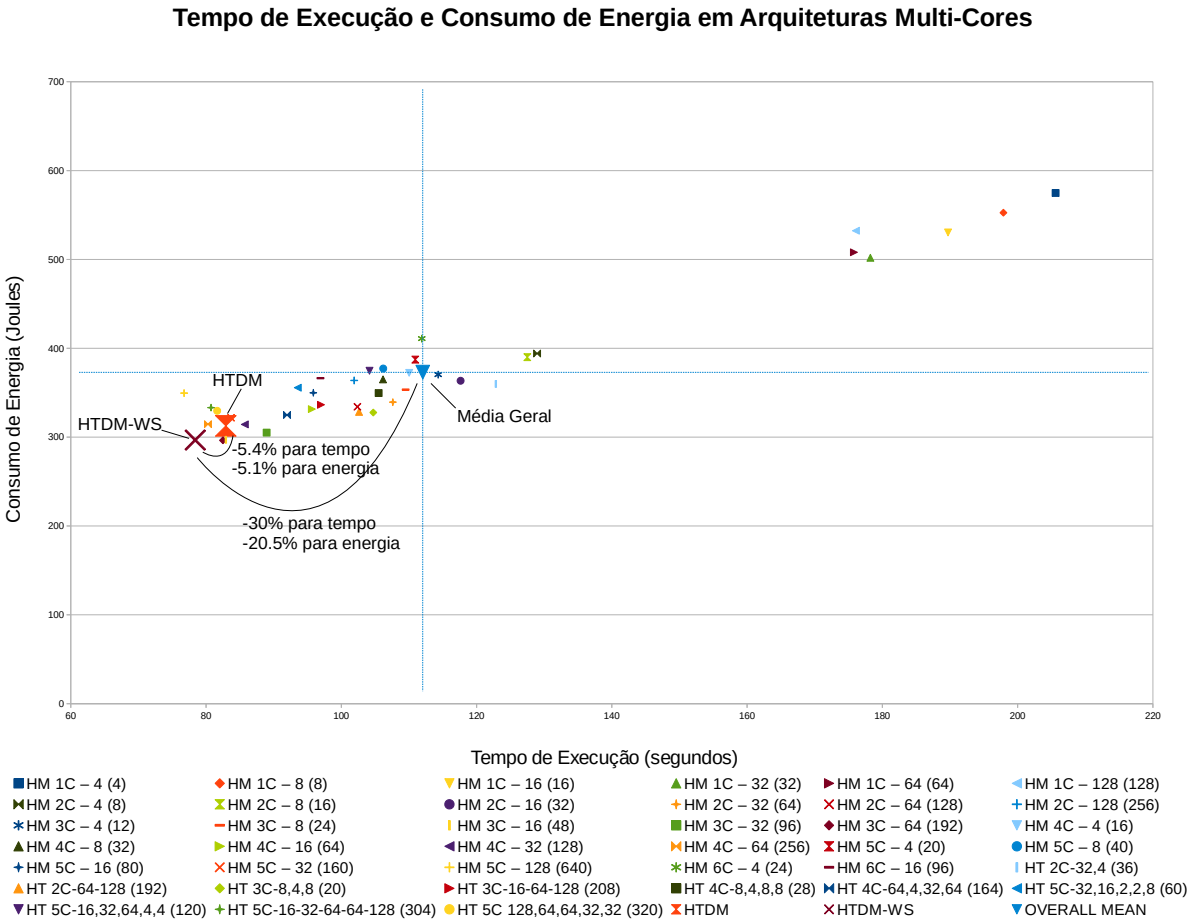

Figura 33 – Relação entre desempenho e consumo de energia para todas as arquiteturas sintetizadas.

A Figura 34 mostra apenas as doze arquiteturas que obtiveram valores melhores que a média geral desconsiderando as arquiteturas *single-core* tanto em relação ao consumo de energia quanto tempo de execução. Pode-se observar que as melhores arquiteturas são compostas por 3, 4 ou 5 *cores*, sendo que a maioria é composta por quatro *cores* e a divisão entre arquiteturas heterogêneas e homogêneas é bastante equilibrada. As arquiteturas HT 4C 16-32-64-128 (240) com e sem *way-shutdown* se posicionam entre as cinco melhores opções. Observando os dados, pode-se perceber que nem sempre mais *cores* representam uma solução melhor, pois a arquitetura HM 3C – 64 (192) apresenta resultado melhor do que várias arquiteturas com quatro e cinco *cores*. O mesmo pode ser observado para o tamanho da *cache*.

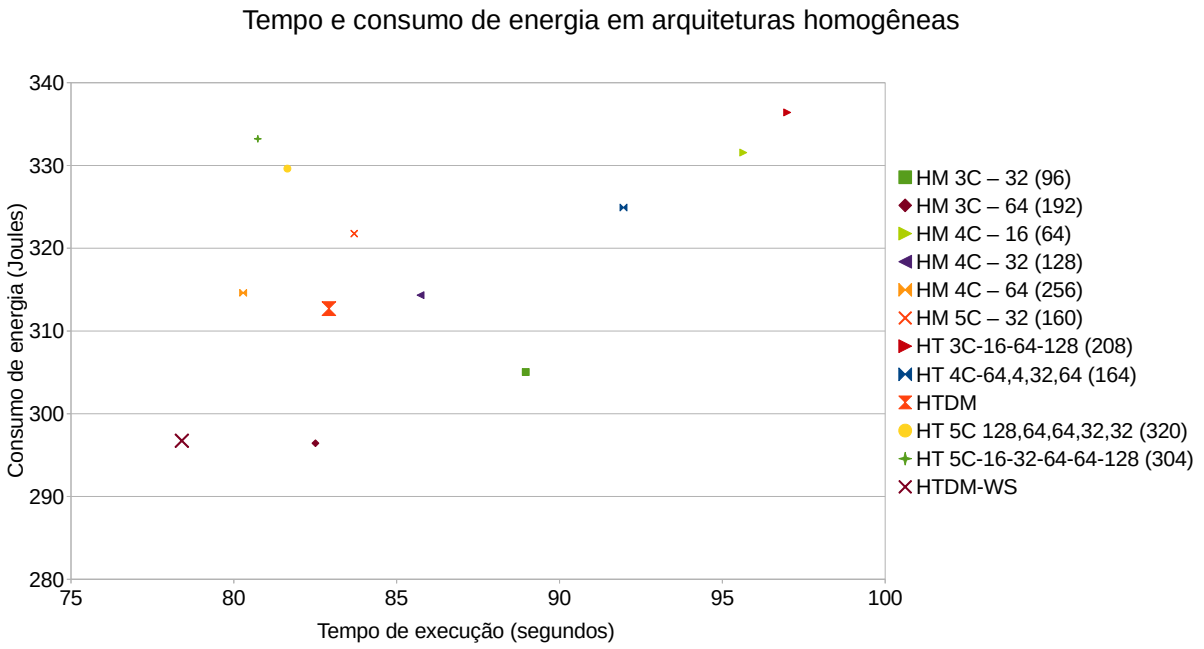

Figura 34 – Relação entre desempenho e consumo de energia para as 12 melhores arquiteturas sintetizadas.

A Figura 35 apresenta uma análise somente do tempo de execução para as doze melhores arquiteturas juntamente com os intervalos de confiança para cada média. Nesta Figura, observa-se que a melhor média para o tempo de execução foi obtida pela arquitetura HT 4C 16-32-64-128 (240) com *way-shutdown*. Entretanto, considerando também o intervalo de confiança, não é possível concluir se as arquiteturas HM 4C – 64 (256), HT 5C 128-64-64-32-32 (320) e HT 5C 16-32-64-64-128 (304) são piores do que a HT 4C 16-32-64-128 (240) com *way-shutdown*.

A Figura 36 apresenta o gráfico comparando as médias do consumo de energia para cada arquitetura juntamente com os respectivos intervalos de confiança. Assim como em relação ao tempo de execução, o menor consumo de energia foi obtido pela arquitetura HT 4C 16-32-64-128 (240) com *way-shutdown*. Entretanto, considerando também os intervalos de confiança, não é possível concluir se HT 4C 16-32-64-128 (240) com *way-shutdown* é melhor do que HM 3C – 32 (96), HM 3C – 64 (192), HT 4C 64-4-32-64 (164) e HT 4C 16-32-64-128 (240) sem *way-shutdown* no que diz respeito ao consumo de energia devido à existência de intersecção entre os intervalos de confiança.

Os intervalos de confiança apresentados na Figura 35 e na Figura 36 foram calculados considerando 95% de certeza.

A etapa de sintetização destas arquiteturas se mostrou demorada, pois algumas arquiteturas demoravam cerca de 6 horas para compilar e, ainda assim, a ferramenta não foi capaz de gerar certas arquiteturas devido às limitações físicas do FPGA.

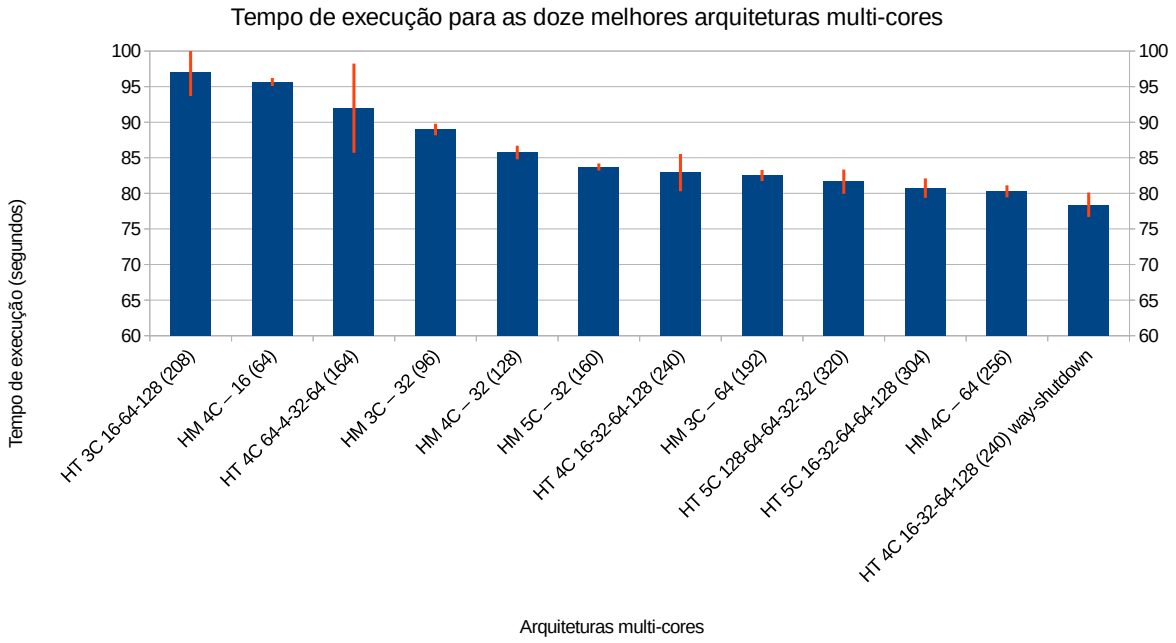

Figura 35 – Tempo de execução para as 12 melhores arquiteturas sintetizadas.

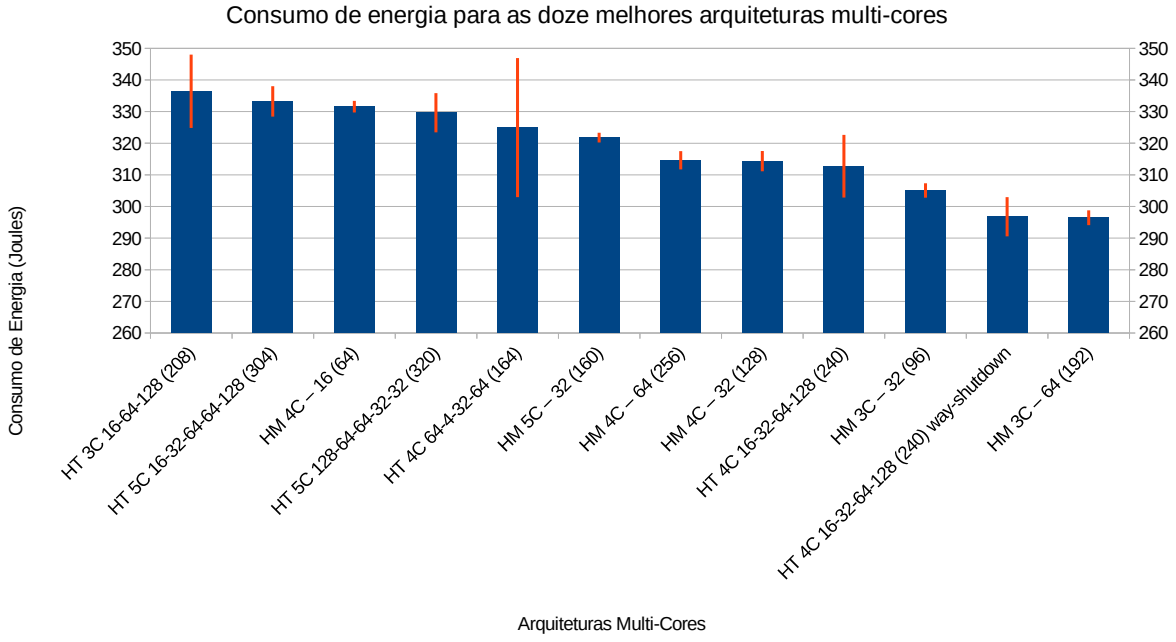

Figura 36 – Consumo de energia para as 12 melhores arquiteturas sintetizadas.

5.2.6 *Exploração do escalonador* heterogeneity-aware *para resiliência em* multi-cores

A diminuição dos tamanhos dos transistores *Very Large Scale Integration* (VLSI) tem permitido que os fabricantes modifiquem seus projetos para *multi-cores* em uma tentativa de manter o custo-benefício devido ao limite inerente do aumento das frequências

de *clock*. Como uma das consequências, os projetos têm exibido um aumento na corrente de fuga, na energia consumida e os erros momentâneos (ou *soft errors*) têm se tornado mais frequente.

Enquanto muito progresso tem sido feito na proteção de estruturas regulares tais como de armazenamento com técnicas de pareamento para a correção de erros em *bits* de memória e o chamado *dual error detection* (*Single Error Correction, Double Error Detection* (SECDED)), a maioria das estruturas internas dos processadores (lógica e até os registradores) são desprotegidos. Corrupção de dados silenciosa pode ocorrer nos registradores e em *caches* L1 (julgadas como muito caras para se proteger com as práticas atuais de proteção, como Error-Correcting Code (ECC)) e devem ser abandonadas uma vez que um único erro momentâneo nestas estruturas pode rapidamente se propagar para as memórias *cache* e registradores.

Uma técnica que de fato tem mitigado os efeitos de falhas necessita executar a mesma computação diversas vezes (redundância de software) gerando múltiplas saídas que são então comparadas para verificar a corretude. Se três cópias são executadas, a presença de um único erro pode ser corrigida em uma técnica conhecida como *software triple-modular-redundancy*, ou Triple Modular Redundancy (TMR). Além desta técnica em software, existem técnicas similares para o hardware onde estruturas específicas tais como *cores* e memórias podem ser replicados ou incluir nós para correção de erros para proteger os dados.

Este experimento tem por objetivo explorar a habilidade dos *multi-cores* para suportarem naturalmente as técnicas de software TMR pela execução de réplicas de uma dada aplicação em diferentes *cores*. Diferentemente de pesquisas anteriores, software TMR foi aplicada a uma arquitetura heterogênea em relação aos tamanhos das memórias *cache* L1 de cada *core*. Dessa forma, este experimento ilustra como a ferramenta proposta neste trabalho juntamente com seu escalonador poderiam ser usados em um contexto voltado para tolerância a falhas e como poderia contribuir para melhorar a relação entre desempenho e consumo de energia.

5.2.6.1 Resultados experimentais

Nesta seção serão apresentados os resultados do uso da resiliência e da ferramenta apresentada neste trabalho para avaliar os relativos méritos do TMR quando aplicado em arquiteturas homogêneas e heterogêneas em relação ao uso de memória *on-chip*, tempo de execução e consumo de energia. Além disso, foram avaliados os benefícios da exploração da informação usada pelo ranqueamento de dominância e o escalonamento das aplicações redundantes (geradas pelo TMR) na arquitetura heterogênea gerada pela ferramenta (arquitetura com 4 *cores* possuindo 16KBytes, 32KBytes, 64KBytes, e 128KBytes de memória *cache* L1, totalizando 240KBytes de *cache*) para a execução das 50 aplicações já

utilizadas nos experimentos anteriores. Para comparação, foi utilizada uma arquitetura homogênea com o mesmo número de *cores* com 64KBytes de memória *cache* L1 para cada *core*, totalizando em 256KBytes, sendo portanto comparável em termos de tamanho com a arquitetura heterogênea selecionada.

Foram desenvolvidas algumas variações da aplicação TMR de acordo com diferentes políticas de escalonamento implementadas no eCos como se segue:

- **original**: esta política corresponde à política original do escalonador do eCos e o código sem o TMR, ou seja, sem as réplicas das aplicações;
- **basic TMR**: uma política que aplica TMR mas as réplicas são mapeadas aos *cores* seguindo a política *First-In First-Out* (tanto para as arquiteturas homogêneas quanto heterogêneas);
- **dominance TMR**: uma política usando TMR mas as réplicas são mapeadas para os melhores *cores* disponíveis se baseando na informação fornecida pelo ranqueamento de dominância;
- **reverse dominance TMR**: uma política que mapeia cada réplica para o pior *core* disponível, fornecendo assim um cenário de *pior caso*.

Enquanto a **original** e **basic TMR** foram usadas tanto para a arquitetura homogênea quanto para as heterogêneas, as duas últimas variações fazem sentido somente para as arquiteturas heterogêneas.

Para implementar as políticas baseadas em TMR no sistema proposto, em tempo de compilação três diferentes instâncias de uma mesma aplicação são geradas, cada uma com um identificador diferente. Esta informação é *linkada* à biblioteca do sistema operacional e é usada pelo escalonador. Quando um *core* invoca o escalonador para selecionar uma aplicação da fila de prontos, ele usa uma tabela para detectar se este *core* ainda não executou nenhuma cópia dessa mesma aplicação antes. Caso tenha executado, o escalonador procura por outra cópia de outra aplicação na fila, até que encontre uma aplicação que ainda não tenha sido executada por este *core*. Para a arquitetura heterogênea, o escalonador usa o ranqueamento de dominância para definir qual é a melhor candidata de acordo com as características específicas do *core*.

As 50 aplicações (e suas réplicas) foram executadas 10 vezes para cada variação de arquitetura e política de escalonamento como descrito acima. A Figura 37 apresenta o tempo de execução médio (parte de cima) e o consumo de energia médio (parte de baixo) para um intervalo de confiança de 95% para a execução das aplicações em cada arquitetura (homogênea - HM e heterogênea - HT) e suas respectivas implementações de escalonador.

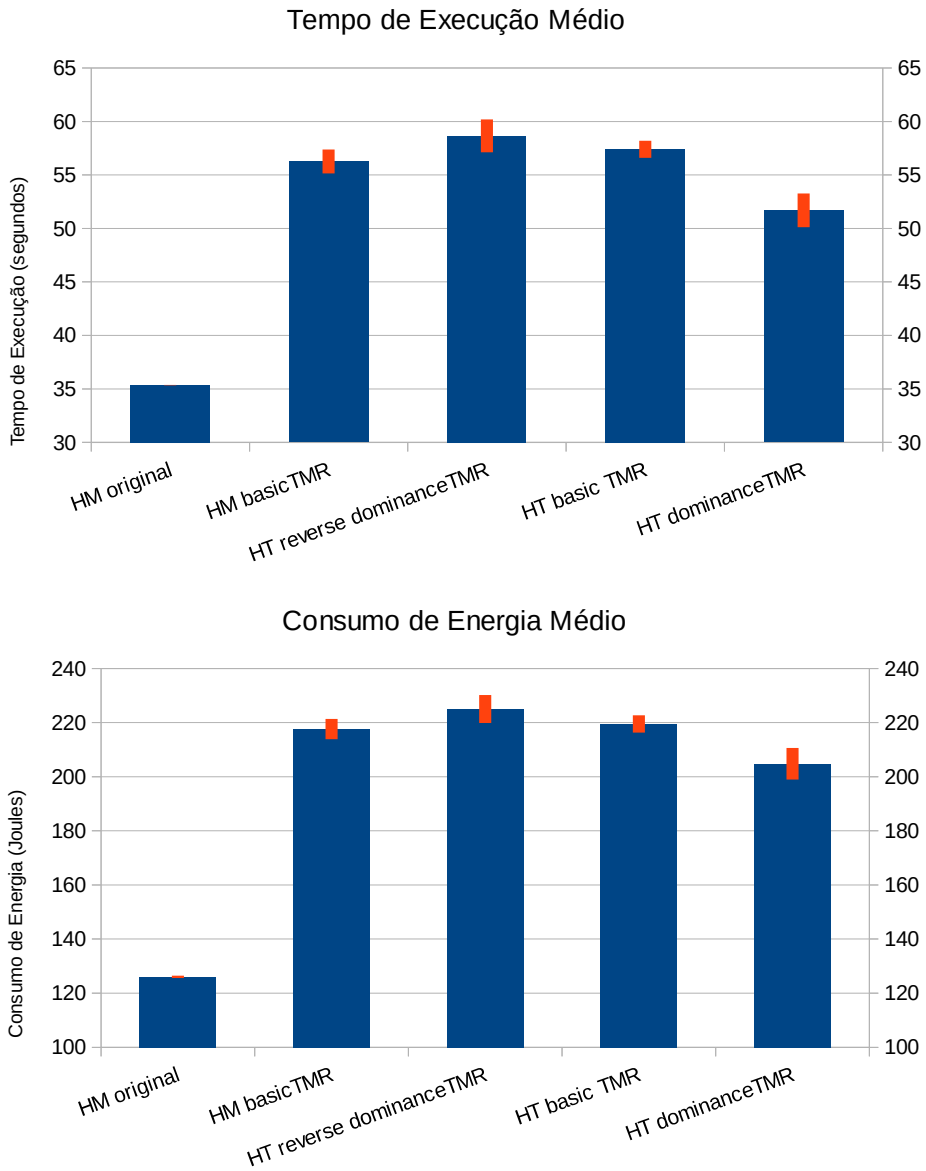

Figura 37 – Tempo de execução médio (acima) e consumo de energia médio (abaixo) para diferentes arquiteturas *multi-cores* para as diferentes variações do TMR e suas políticas de escalonamento.

Um comportamento similar pode ser observado tanto para o consumo de energia quanto para o tempo de execução. Os valores mais baixos são naturalmente obtidos pela arquitetura homogênea usando o escalonador original do eCos e sem o uso de TMR (**HM original** atingiu $35,35$ segundos and $126,02$ Joules). Pela comparação destes resultados com a execução do **basic TMR**, é possível observar os custos associados ao TMR. Como o sistema teve que executar 50×3 aplicações, naturalmente irá ocorrer um aumento no tempo e na energia consumida (**HM basicTMR** atingiu $56,27$ segundos e $217,61$ Joules).

A Tabela 15 apresenta o resultado das médias de tempo de execução e consumo de energia para as variações do TMR e arquiteturas.

Considerando agora a execução das variações de TMR nas arquiteturas heterogêneas,

Tabela 15 – Tempo de execução e consumo de energia médios para as variações de TMR e arquiteturas: homogêneas (HM) e heterogêneas (HT).

Aplicações e Variações de arquitetura	Tempo de execução Médio (segundos)	Consumo de energia Médio (Joules)
HM Original	35,35	126,02
HM basic TMR	56,27	217,61
HT reverse dominance TMR	58,65	225,04
HT basic TMR	57,40	219,49
HT dominance TMR	51,69	204,77

pode-se observar que o uso da informação do ranqueamento de dominância permite um decréscimo do tempo de execução e consumo de energia em relação ao **basic TMR**. Isto também é verdadeiro para a comparação do desempenho em relação à arquitetura homogênea.

Quantitativamente, o *dominance TMR* na arquitetura heterogênea selecionada atingiu uma redução de 8.14% no tempo de execução e uma redução de 5.9% no consumo de energia quando comparado ao TMR na arquitetura homogênea, mesmo com uma memória *cache* 6.25% menor. Além disso, conforme o esperado, o *reverse dominance TMR* obteve o pior desempenho em todas as métricas uma vez que escalonou as aplicações para os piores *cores* disponíveis.

Embora estes resultados foquem apenas em um conjunto específico de aplicações e consequentemente para uma arquitetura heterogênea customizada de acordo com os códigos das aplicações, eles revelam que tanto para tempo de execução quanto para consumo de energia, arquiteturas heterogêneas customizadas aliadas ao ranqueamento de dominância podem ser muito vantajosas em relação ao TMR simples implementado em uma arquitetura homogênea. Estas arquiteturas heterogêneas além de serem mais eficientes em termos de consumo de energia, também suportam técnicas baseadas em redundância tais como TMR.

5.2.7 *Estudo inicial para arquiteturas* multi-cores *heterogêneas em relação à FPU*

Nesta seção é apresentado um estudo inicial sobre o potencial em se explorar arquiteturas heterogêneas em relação à FPU. Neste experimento, foi realizado o *offline profiling* e cálculo do ranqueamento de dominância para 10 aplicações e, após isso, foi identificada a melhor configuração de FPU para cada aplicação. Estas 10 aplicações são as aplicações de referências já utilizadas nos experimentos com *caches* diferentes. Elas foram escolhidas para este experimento pois também há variação no uso de instruções de ponto flutuante. Para os resultados apresentados, foi feita uma estimativa de quais seriam os ganhos de eficiência da execução destas aplicações na melhor configuração possível.

Para realizar o *offline profiling* e calcular o ranqueamento de dominância, foram

Tabela 16 – Tempo de execução, em segundos, para 10 aplicações em diferentes formas de tratamento de instruções em ponto flutuante.

	Arquitetura *single-core* sem *cache*		
Aplicações	sem FPU	GRFPU-lite simple	GRFPU
auto_corre	27,31	29,32	29,32
bubble_sort	105,63	108,08	108,08
dot_prod	30,39	28,75	28,76
fibonacci	13,96	15,75	15,75
matrix_mult	346,49	343,79	343,82
max	11,96	11,89	11,89
pi	15,07	39,28	39,28
vecsum	15,95	13,84	13,84
linear	8,54	0,60	0,46
jas_image_setbbox	31,82	33,30	33,30

Tabela 17 – Consumo de energia, em Joules, para 10 aplicações em diferentes formas de tratamento de instruções em ponto flutuante.

	Arquitetura *single-core* sem *cache*		
Aplicações	sem FPU	GRFPU-lite simple	GRFPU
auto_corre	77,967	84,678	85,582
bubble_sort	306,402	315,81	320,23
dot_prod	86,903	82,487	83,857
fibonacci	40,327	45,858	46,554
matrix_mult	983,855	985,997	1002,992
max	34,4	34,158	35,026
pi	43,251	113,562	115,812
vecsum	45,888	39,794	40,713
linear	24,548	1,698	1,333
jas_image_setbbox	91,192	95,568	96,748

consideradas três variações de tratamento de execução de instruções em ponto flutuante disponíveis para o LEON3:

- sem FPU: arquitetura sem FPU e FPU é emulada em software;
- GRFPU-lite: implementação mais econômica da unidade de ponto flutuante;
- GRFPU: implementação mais eficiente da unidade de ponto flutuante.

As Tabelas 16 e 17 apresentam respectivamente os valores do *offline profiling* para tempo e consumo de energia nas três variações de tratamento de ponto flutuante. E a Tabela 18 apresenta o ranqueamento de dominância calculado para este caso.

Tabela 18 – Ranqueamento de dominância para 10 aplicações em diferentes formas de tratamento de instruções em ponto flutuante. Em negrito estão destacadas as maiores dominâncias.

	Ranqueamento de dominância		
Aplicações	sem FPU	GRFPU-lite simple	GRFPU
auto_corre	**4**	-1	-3
bubble_sort	**4**	-1	-1
dot_prod	-4	**4**	0
fibonacci	**4**	-1	-3
matrix_mult	0	**2**	-2
max	-2	**3**	-1
pi	**4**	-1	-3
vecsum	-4	**3**	1
linear	-4	0	**4**
jas_image_setbbox	**4**	-1	-3

5.2.7.1 Resultados

Para estimar os resultados, considerou-se uma arquitetura *multi-core* hipotética com um *core* para cada aplicação. Sendo assim, o consumo de energia total é a soma dos consumos individuais e o tempo total é o tempo da aplicação mais longa. Trata-se, portanto, de uma estimativa otimista. A Figura 38 apresenta o tempo de execução e a Figura 39 apresenta o consumo de energia.

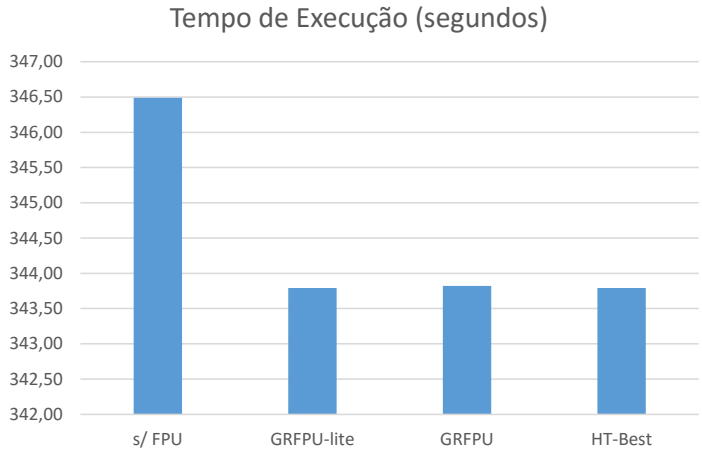

Figura 38 – Tempo de execução total para as aplicações de referência em arquiteturas com diferentes formas de tratamento de operações com ponto flutuante.

A HT-Best é a arquitetura heterogênea considerada a ideal para executar as aplicações de referência, considerando que a aplicação irá executar na arquitetura que apresentou maior valor para de dominância de acordo com a Tabela 18.

Como pode ser visto na Figura 38, o tempo de execução para HT-Best foi similar ao melhor tempo (GRFPU). Ao mesmo tempo, de acordo com a Figura 39, o consumo de energia foi 7% melhor que o segundo melhor consumo (s/ FPU).

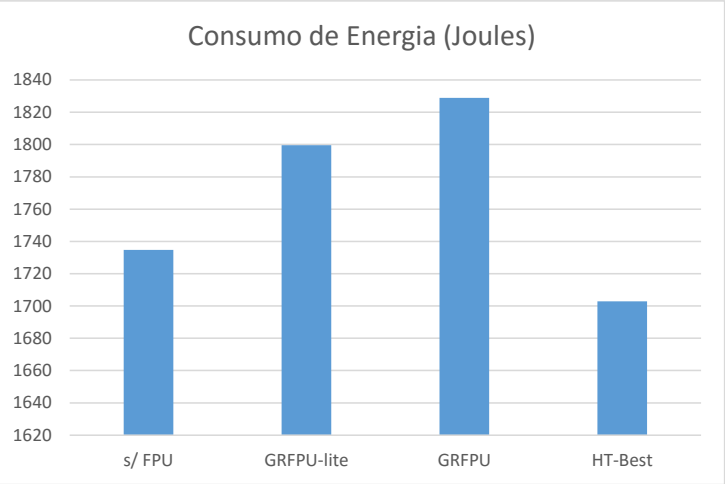

Figura 39 – Consumo de energia total para as aplicações de referência em arquiteturas com diferentes formas de tratamento de operações com ponto flutuante.

Portanto, o método proposto neste trabalho pode ser aplicado para realizar a análise do uso de arquiteturas heterogêneas em relação ao tratamento de operações de ponto flutuante. Para o caso hipotético apresentado nesta seção, é possível concluir que houve redução significativa no consumo de energia sem degradação do desempenho da aplicação como um todo. Porém, é necessário que haja uma investigação mais adequada, selecionando aplicações que façam uso efetivo da FPU em diferentes graus para se obter informações mais conclusivas a respeito do desempenho e do consumo de energia para estas arquiteturas.

5.3 Considerações finais

Este Capítulo apresentou os resultados para diversos experimentos realizados durante o desenvolvimento deste trabalho de doutorado. Ao longo da apresentação dos resultados, foram realizadas discussões sobre o que cada experimento trouxe de contribuição em relação ao anterior e também em relação ao objetivo desta tese. No próximo capítulo, será apresentada a conclusão deste documento, sintetizando as principais contribuições deste trabalho.

CAPÍTULO

6

CONCLUSÃO

Este trabalho de doutorado teve por objetivo investigar, propor e avaliar um novo método para a otimização da relação desempenho/consumo de energia em arquiteturas *multi-cores* heterogêneas *single-ISA* em FPGA que sejam específicas para um dado conjunto de aplicações. O método proposto compreende os aspectos *offline* e *online* de definição e uso de uma arquitetura. A definição ocorre em relação a quantidade de *cores*, tamanho de memória *cache* L1 para cada *core* e o uso ou não de FPU, sem haver a necessidade de execução de todas as aplicações *a priori*. E para o seu uso, é definido um conjunto de atividades que buscam a execução das aplicações de maneira eficiente em relação ao consumo de energia. As principais contribuições obtidas com este trabalho são apresentadas a seguir:

- A integração de técnicas *offline* com técnicas *online* é feita de forma automática e transparente para o usuário, e a otimização do consumo de energia é feita primeiramente pela definição de uma arquitetura *multi-core* ideal e, em seguida, as informações sobre esta arquitetura e como as aplicações devem ser executadas são usadas em tempo de execução pelo sistema operacional que, ao mesmo tempo, leva em consideração a carga do sistema;

- O método proposto, por utilizar a ferramenta DAMICORE como uma das partes do mesmo, é capaz de realizar a otimização com o conhecimento apenas das aplicações de referência, contribuindo para a diminuição da dependência do conhecimento de especialistas ao se utilizar a ferramenta implementada;

- Foi também realizada a mudança no sistema operacional eCos para suportar arquiteturas heterogêneas;

- Implementação da capacidade para realizar ajuste fino na redução do consumo de energia em tempo de execução por meio da técnica *way-shutdown* dinâmico;

- O método proposto viabilizou a aplicação de software TMR em arquiteturas heterogêneas com uma redução significativa no consumo de energia se comparado ao software TMR aplicado em arquiteturas homogêneas tradicionais;

- O método pode ser estendido para considerar outras características, como mostrou o estudo inicial apresentado sobre arquiteturas *multi-cores* heterogêneas em relação à forma de tratamento de operações com ponto flutuante;

- O trabalho apresenta diversos resultados de execuções em plataforma de hardware real, que servem para uma melhor compreensão do comportamento das soluções encontradas. Por meio do uso do FPGA e de análises estatísticas foi possível observar melhor o comportamento de muitas arquiteturas e, como existem poucas plataformas heterogêneas disponíveis comercialmente, este estudo revelou características ainda pouco observadas;

- Uma proposta de um método capaz de aplicar otimizações em diversos níveis de desenvolvimento: no nível da arquitetura, com a modificação do processador LEON3 para aplicar *way-shutdown* dinâmico, tamanhos de memória *cache* diferentes, presença ou não de FPU; no nível do sistema operacional, com a implementação de uma nova política *heterogeneity-aware*, os *drivers*, interrupções e alarme para se comunicar com os componentes de hardware que realizam medições de potência e controlam a *cache* e modificação no padrão POSIX *Threads* para suportar os *cores* heterogêneos; e no nível da compilação (*offline*) com a utilização de uma combinação de técnicas para conseguir gerar a configuração de uma arquitetura *multi-core* específica para um conjunto de aplicações e sugerir o mapeamento e escalonamento ideal para as aplicações nos *cores*.

Foram publicados quatro trabalhos diretamente derivados desta tese desde a proposta da ferramenta até os experimentos realizados. São eles: Silva e Bonato (2012), Silva, Cuminato e Bonato (2012), Silva *et al.* (2015a) e Silva *et al.* (2015b).

Uma das limitações deste trabalho é o pequeno número de características consideradas na heterogeneidade (número de *cores*, o tamanho da *cache*, a quantidade de *ways* ativos e o uso de unidade de ponto flutuante). Em ambiente simulado é possível trabalhar com diversas outras possibilidades. Além disso, apenas um estudo inicial foi feito com instruções em ponto flutuante e esta característica não foi explorada em conjunto com as demais. Dessa forma, os resultados apresentados para FPU revelam que existe potencial para redução do consumo pelo método proposto, mas não é possível concluir se poderá de fato trazer grandes benefícios. Outras técnicas para gerenciamento *online* do consumo de energia podem também ser integradas na arquitetura, como *clock gating* agressivo em nível de microarquitetura.

Muitas dificuldades foram enfrentadas ao longo do desenvolvimento deste trabalho. Entretanto, algumas merecem maior destaque, que são: o desenvolvimento da plataforma completa desde o hardware implementado no FPGA com o suporte da placa *Stratix IV GX* ao LEON3, que não existia, seus componentes para comunicação com a memória externa e o circuito medidor de potência até os componentes do sistema operacional; e o processo de compilação das arquiteturas *multi-cores* homogêneas e heterogêneas utilizadas para comparações é bastante demorado mesmo executando a ferramenta de síntese nos computadores mais recentes, variando entre 2 e 14 horas para concluir a compilação (com sucesso ou não) de apenas uma arquitetura. Esta última dificuldade ilustra a importância de um método como este apresentado neste trabalho. Uma vez que uma ferramenta automática é capaz de gerar uma arquitetura *multi-core* heterogênea eficiente para executar um dado conjunto de aplicações com apenas uma compilação, diversas horas e outros recursos podem ser economizados durante a etapa de desenvolvimento.

Como trabalhos futuros pode-se citar o estudo da influência de diferentes aplicações de referência na qualidade da solução gerada; a integração de mais características para a arquitetura heterogênea, como por exemplo diferentes frequências de *clock*; aplicação do método proposto para arquiteturas heterogêneas *multi-ISA*; e finalmente, avaliar o método para conjuntos de aplicações interdependentes, que possuam diversos pontos de sincronização.

REFERÊNCIAS

ALTERA. Stratix IV GX FPGA Development Board Reference Manual. 2012. Disponível em: <http://www.altera.com/literature/manual/rm_sivgx_fpga_dev_board.pdf>. Citado 3 vezes nas páginas 11, 53 e 54.

ASADUZZAMAN, A. **Cache Optimization for Real-Time Embedded Systems**. BiblioBazaar, 2011. ISBN 9781243688828. Disponível em: <https://books.google.com.br/books?id=d21CAwEACAAJ>. Citado 2 vezes nas páginas 28 e 39.

ASANOVIC, K.; BODIK, R.; DEMMEL, J.; KEAVENY, T.; KEUTZER, K.; KUBIATOWICZ, J.; MORGAN, N.; PATTERSON, D.; SEN, K.; WAWRZYNEK, J.; WESSEL, D.; YELICK, K. A View of the Parallel Computing Landscape. **Commun. ACM**, ACM, New York, NY, USA, v. 52, n. 10, p. 56–67, out. 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1562764.1562783>. Citado na página 27.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: A survey. **Computer Networks**, v. 54, n. 15, p. 2787 – 2805, 2010. ISSN 1389-1286. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1389128610001568>. Citado na página 23.

BATSON, B.; VIJAYKUMAR, T. N. Reactive-Associative Caches. In: **Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques**. Washington, DC, USA: IEEE Computer Society, 2001. (PACT '01), p. 49–60. ISBN 0-7695-1363-8. Disponível em: <http://dl.acm.org/citation.cfm?id=645988.674302>. Citado na página 37.

BATTY, M.; AXHAUSEN, K.; GIANNOTTI, F.; POZDNOUKHOV, A.; BAZZANI, A.; WACHOWICZ, M.; OUZOUNIS, G.; PORTUGALI, Y. Smart cities of the future. **The European Physical Journal Special Topics**, Springer-Verlag, v. 214, n. 1, p. 481–518, 2012. ISSN 1951-6355. Disponível em: <http://dx.doi.org/10.1140/epjst/e2012-01703-3>. Citado na página 23.

BECCHI, M.; CROWLEY, P. Dynamic thread assignment on heterogeneous multiprocessor architectures. In: **Proceedings of the 3rd conference on Computing frontiers**. New York, NY, USA: ACM, 2006. (CF '06), p. 29–40. ISBN 1-59593-302-6. Disponível em: <http://doi.acm.org/10.1145/1128022.1128029>. Citado 2 vezes nas páginas 39 e 42.

BELWAL, M.; SUDARSHAN, T. Intermediate representation for heterogeneous multicore: A survey. In: **VLSI Systems, Architecture, Technology and Applications (VLSI-SATA), 2015 International Conference on**. [S.l.: s.n.], 2015. p. 1–6. Citado na página 24.

BIENIA, C.; KUMAR, S.; SINGH, J. P.; LI, K. The PARSEC benchmark suite: characterization and architectural implications. In: **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. New York, NY, USA: ACM, 2008. (PACT '08), p. 72–81. ISBN 978-1-60558-282-5. Disponível em: <http://doi.acm.org/10.1145/1454115.1454128>. Citado na página 26.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A survey of multicore processors. **Signal Processing Magazine, IEEE**, IEEE, v. 26, n. 6, p. 26–37, nov. 2009. ISSN 1053-5888. Disponível em: <http://dx.doi.org/10.1109/MSP.2009.934110>. Citado 2 vezes nas páginas 28 e 39.

BOBDA, C.; HALLER, T.; MUEHLBAUER, F.; RECH, D.; JUNG, S. Design of adaptive multiprocessor on chip systems. In: **Proceedings of the 20th annual conference on Integrated circuits and systems design**. New York, NY, USA: ACM, 2007. (SBCCI '07), p. 177–183. ISBN 978-1-59593-816-9. Disponível em: <http://doi.acm.org/10.1145/1284480.1284531>. Citado na página 33.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Commun. ACM**, ACM, New York, NY, USA, v. 54, n. 5, p. 67–77, maio 2011. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1941487.1941507>. Citado 4 vezes nas páginas 11, 23, 24 e 25.

BORKAR, S. Y.; DUBEY, P.; KAHN, K. C.; KUCK, D. J.; MULDER, H.; PAWLOWSKI, S. S.; RATTNER, J. R. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. **Intel White Paper**, 2005. Disponível em: <ftp://download.intel.com/technology/computing/archinnov/platform2015/download/Platform_2015.pdf>. Citado 3 vezes nas páginas 23, 33 e 38.

CALDER, B.; GRUNWALD, D.; EMER, J. Predictive sequential associative cache. In: **Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 1996. (HPCA '96), p. 244–. ISBN 0-8186-7237-4. Disponível em: <http://dl.acm.org/citation.cfm?id=525424.822662>. Citado na página 37.

CARVALHO, H. L. de. **Método de análise para a coordenação dos processos de produção sob a ótica de redes de inovação colaborativas apoiado por agente inteligente evolutivo**. Tese de doutorado. Escola de Engenharia de São Carlos. Universidade de São Paulo, 2015. Citado na página 44.

CATANZARO, B.; FOX, A.; KEUTZER, K.; PATTERSON, D.; SU, B.-Y.; SNIR, M.; OLUKOTUN, K.; HANRAHAN, P.; CHAFI, H. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. **IEEE Micro**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 30, n. 2, p. 41–55, mar. 2010. ISSN 0272-1732. Disponível em: <http://dx.doi.org/10.1109/MM.2010.42>. Citado na página 26.

CHANG, C.; WAWRZYNEK, J.; BRODERSEN, R. W. BEE2: A High-End Reconfigurable Computing System. **IEEE Des. Test**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 22, n. 2, p. 114–125, mar. 2005. ISSN 0740-7475. Disponível em: <http://dx.doi.org/10.1109/MDT.2005.30>. Citado na página 33.

CHIEN, A. A. Pervasive parallel computing: an historic opportunity for innovation in programming and architecture. In: **Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming**. New York, NY, USA: ACM, 2007. (PPoPP '07), p. 160–160. ISBN 978-1-59593-602-8. Disponível em: <http://doi.acm.org/10.1145/1229428.1229467>. Citado na página 26.

CILIBRASI, R.; VITANYI, P. M. B. Clustering by compression. **IEEE Transactions on Information Theory**, v. 51, p. 1523–1545, 2005. Citado 2 vezes nas páginas 44 e 45.

CUMINATO, L. A. **Otimização de memória cache em tempo de execução para o processador embarcado LEON3**. Dissertação de mestrado. Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, 2014. Citado na página 62.

DEVORE, J. L. **Probability and Statistics for Engineering and the Sciences**. 8th. ed. [S.l.]: Brooks/Cole, 2011. ISBN-13: 978-0-538-73352-6. Citado na página 94.

DING, X.; WANG, K.; ZHANG, X. ULCC: a user-level facility for optimizing shared cache performance on multicores. In: **Proceedings of the 16th ACM symposium on Principles and practice of parallel programming**. New York, NY, USA: ACM, 2011. (PPoPP '11), p. 103–112. ISBN 978-1-4503-0119-0. Disponível em: <http://doi.acm.org/10.1145/1941553.1941568>. Citado 2 vezes nas páginas 25 e 26.

ECOS. ecos. 2016. Disponível em: <http://ecos.sourceware.org/>. Citado na página 46.

EECKHOUT, L. Heterogeneity in Response to the Power Wall. **Micro, IEEE**, v. 35, n. 4, p. 2–3, July 2015. ISSN 0272-1732. Citado 3 vezes nas páginas 23, 27 e 32.

FEDOROVA, A.; SAEZ, J. C.; SHELEPOV, D.; PRIETO, M. Maximizing power efficiency with asymmetric multicore systems. **Commun. ACM**, ACM, New York, NY, USA, v. 52, n. 12, p. 48–57, dez. 2009. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1610252.1610270>. Citado na página 28.

FELSENSTEIN, J. **Inferring Phylogenies**. [S.l.]: Sinauer Associates, 2003. Citado na página 44.

FERREIRA, E. J. **Método baseado em rotação e projeção otimizadas para a construção de ensembles de modelos**. Tese de doutorado. Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, 2012. Citado na página 58.

FLINN, J.; SATYANARAYANAN, M. Managing battery lifetime with energy-aware adaptation. **ACM Trans. Comput. Syst.**, ACM, New York, NY, USA, v. 22, n. 2, p. 137–179, maio 2004. ISSN 0734-2071. Disponível em: <http://doi.acm.org/10.1145/986533.986534>. Citado na página 26.

GAISLER-A, C. GRLIB IP Library User's Manual. 2015. Disponível em: <http://gaisler.com/products/grlib/grlib.pdf>. Citado 3 vezes nas páginas 11, 49 e 52.

GAISLER-B, C. GRLIB IP Core User's Manual. 2015. Disponível em: <http://gaisler.com/products/grlib/grip.pdf>. Citado 2 vezes nas páginas 11 e 50.

GAISLER, C. Leon3. 2015. Disponível em: <http://www.gaisler.com/index.php/products/processors/leon3>. Citado na página 26.

HENNESSY, J. L.; PATTERSON, D. A. **Computer architecture: a quantitative approach**. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 1-55860-596-7. Citado na página 33.

HILL, M. D.; MARTY, M. R. Amdahl's Law in the Multicore Era. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 41, n. 7, p. 33–38, jul. 2008. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2008.209>. Citado 2 vezes nas páginas 38 e 39.

HSU, C.-H.; KREMER, U. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In: **Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation**. New York, NY, USA: ACM, 2003. (PLDI '03), p. 38–48. ISBN 1-58113-662-5. Disponível em: <http://doi.acm.org/10.1145/781131.781137>. Citado na página 35.

HUMENAY, E.; TARJAN, D.; SKADRON, K. Impact of process variations on multicore performance symmetry. In: **Proceedings of the conference on Design, automation and test in Europe**. San Jose, CA, USA: EDA Consortium, 2007. (DATE '07), p. 1653–1658. ISBN 978-3-9810801-2-4. Disponível em: <http://dl.acm.org/citation.cfm?id=1266366.1266729>. Citado na página 32.

ISCI, C.; BUYUKTOSUNOGLU, A.; CHER, C.-Y.; BOSE, P.; MARTONOSI, M. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In: **Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2006. (MICRO 39), p. 347–358. ISBN 0-7695-2732-9. Disponível em: <http://dx.doi.org/10.1109/MICRO.2006.8>. Citado na página 35.

ISHEBABI, H.; BOBDA, C. Automated architecture synthesis for parallel programs on FPGA multiprocessor systems. **Microprocess. Microsyst.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 33, n. 1, p. 63–71, fev. 2009. ISSN 0141-9331. Disponível em: <http://dx.doi.org/10.1016/j.micpro.2008.08.009>. Citado na página 33.

ISHIHARA, T.; YASUURA, H. Voltage scheduling problem for dynamically variable voltage processors. In: **Proceedings of the 1998 international symposium on Low power electronics and design**. New York, NY, USA: ACM, 1998. (ISLPED '98), p. 197–202. ISBN 1-58113-059-7. Disponível em: <http://doi.acm.org/10.1145/280756.280894>. Citado na página 35.

IVANOV, A.; MICHELI, G. D. The network-on-chip paradigm in practice and research. **IEEE Des. Test**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 22, n. 5, p. 399–403, set. 2005. ISSN 0740-7475. Disponível em: <http://dx.doi.org/10.1109/MDT.2005.111>. Citado na página 33.

JHENG, G.-C.; DUH, D.-R.; LAI, C.-N. Real-time reconfigurable cache for low-power embedded systems. **IJES**, v. 4, n. 3/4, p. 235–247, 2010. Citado 2 vezes nas páginas 25 e 26.

KIRK, D. B.; HWU, W.-m. W. **Programming Massively Parallel Processors: A Hands-on Approach**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN 0123814723, 9780123814722. Citado na página 23.

KORNAROS, G. **Multi-Core Embedded Systems**. 1st. ed. 6000 Broken Sound Parkway NW, Suite 300, USA: CRC Press by Taylor and Francis Group, LLC, 2010. ISBN 978-1-4398-1161-0. Citado 5 vezes nas páginas 27, 31, 32, 33 e 34.

KUMAR, R.; TULLSEN, D. M.; RANGANATHAN, P.; JOUPPI, N. P.; FARKAS, K. I. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 32, n. 2, p. 64–, mar. 2004. ISSN 0163-5964. Disponível em: <http://doi.acm.org/http://dx.doi.org/10.1145/1028176.1006707>. Citado 3 vezes nas páginas 38, 39 e 42.

LI, M.; VITNYI, P. M. **An Introduction to Kolmogorov Complexity and Its Applications**. 3. ed. [S.l.]: Springer Publishing Company, Incorporated, 2008. ISBN 0387339981, 9780387339986. Citado na página 45.

LIU, Y.; YANG, H.; DICK, R. P.; WANG, H.; SHANG, L. Thermal vs Energy Optimization for DVFS-Enabled Processors in Embedded Systems. In: **Proceedings of the 8th International Symposium on Quality Electronic Design**. Washington, DC, USA: IEEE Computer Society, 2007. (ISQED '07), p. 204–209. ISBN 0-7695-2795-7. Disponível em: <http://dx.doi.org/10.1109/ISQED.2007.158>. Citado na página 38.

MAKEN, P.; DEGRAUWE, M.; PAEMEL, M. V.; OGUEY, H. A voltage reduction technique for digital systems. In: **Digest of Technical Papers IEEE International Solid-State Circuits Conference**. [S.l.: s.n.], 1990. p. 238–239. Citado na página 35.

MARCULESCU, D. On the use of microarchitecture-driven dynamic voltage scaling. In: **Proceedings of Workshop on Complexity-Effective Design, in Conjunction with International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2000. Citado na página 35.

MORO, L. F. de S. **Caracterização de alunos em ambientes de ensino online: estendendo o uso da DAMICORE para minerar dados educacionais**. Dissertação de mestrado. Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, 2015. Citado na página 44.

NEWMAN, M. **Networks: An Introduction**. New York, NY, USA: Oxford University Press, Inc., 2010. ISBN 0199206651, 9780199206650. Citado 2 vezes nas páginas 44 e 46.

NIE, P.; DUAN, Z. Efficient and scalable scheduling for performance heterogeneous multicore systems. **Journal of Parallel and Distributed Computing**, v. 72, n. 3, p. 353 – 361, 2012. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731511002358>. Citado 3 vezes nas páginas 38, 40 e 42.

PERERA, C.; ZASLAVSKY, A.; CHRISTEN, P.; GEORGAKOPOULOS, D. Context Aware Computing for The Internet of Things: A survey. **Communications Surveys Tutorials, IEEE**, v. 16, n. 1, p. 414–454, First 2014. ISSN 1553-877X. Citado na página 23.

POWELL, M. D.; AGARWAL, A.; VIJAYKUMAR, T. N.; FALSAFI, B.; ROY, K. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In: **Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 2001. (MICRO 34), p. 54–65. ISBN 0-7695-1369-7. Disponível em: <http://dl.acm.org/citation.cfm?id=563998.564007>. Citado na página 37.

RABAEY, J. M.; CHANDRAKASAN, A. P.; NIKOLIC, B. **Digital integrated circuits : a design perspective**. 2. ed. [S.l.]: Pearson Education, 2003. Paperback. (Prentice Hall electronics and VLSI series). ISBN 0130909963. Citado na página 34.

SAEZ, J. C.; FEDOROVA, A.; PRIETO, M.; VEGAS, H. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In: **Proceedings of the 7th ACM international conference on Computing frontiers**. New York, NY, USA: ACM, 2010. (CF '10), p. 31–40. ISBN 978-1-4503-0044-5. Disponível

em: <http://doi.acm.org/10.1145/1787275.1787281>. Citado 3 vezes nas páginas 26, 40 e 42.

SAEZ, J. C.; PRIETO, M.; FEDOROVA, A.; BLAGODUROV, S. A comprehensive scheduler for asymmetric multicore systems. In: **Proceedings of the 5th European conference on Computer systems**. New York, NY, USA: ACM, 2010. (EuroSys '10), p. 139–152. ISBN 978-1-60558-577-2. Disponível em: <http://doi.acm.org/10.1145/1755913.1755929>. Citado 2 vezes nas páginas 40 e 42.

SALDANA, M.; NUNES, D.; RAMALHO, E.; CHOW, P. Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI. **Reconfigurable Computing and FPGAs, International Conference on**, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 1–10, 2006. Citado na página 33.

SANCHES, A.; CARDOSO, J. M. P.; DELBEM, A. C. B. Identifying merge-beneficial software kernels for hardware implementation. In: **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2011. p. 74–79. ISSN 2325-6532. Citado na página 44.

SCHMITZ, M. T.; AL-HASHIMI, B. M.; ELES, P. **System-Level Design Techniques for Energy-Efficient Embedded Systems**. Norwell, MA, USA: Kluwer Academic Publishers, 2004. ISBN 1402077505. Citado 2 vezes nas páginas 34 e 35.

SEMERARO, G.; MAGKLIS, G.; BALASUBRAMONIAN, R.; ALBONESI, D. H.; DWARKADAS, S.; SCOTT, M. L. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In: **Proceedings of the 8th International Symposium on High-Performance Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2002. (HPCA '02), p. 29–. Disponível em: <http://dl.acm.org/citation.cfm?id=874076.876477>. Citado na página 35.

SHEARER, F. **Power Management in Mobile Devices**. Newton, MA, USA: Newnes, 2007. ISBN 0750679581, 9780750679589. Citado 3 vezes nas páginas 11, 25 e 36.

SHELEPOV, D.; ALCAIDE, J. C. S.; JEFFERY, S.; FEDOROVA, A.; PEREZ, N.; HUANG, Z. F.; BLAGODUROV, S.; KUMAR, V. HASS: a scheduler for heterogeneous multicore systems. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 43, n. 2, p. 66–75, abr. 2009. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1531793.1531804>. Citado 2 vezes nas páginas 40 e 42.

SILVA, B. A.; BONATO, V. Power/performance optimization in FPGA-based asymmetric multi-core systems. In: **22nd International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.: s.n.], 2012. p. 473–474. ISSN 1946-147X. Citado na página 112.

SILVA, B. A.; CUMINATO, L. A.; BONATO, V. Reducing the overall cache miss rate using different cache sizes for heterogeneous multi-core processors. In: **2012 International Conference on Reconfigurable Computing and FPGAs**. [S.l.: s.n.], 2012. p. 1–6. ISSN 2325-6532. Citado na página 112.

SILVA, B. A.; CUMINATO, L. A.; DELBEM, A. C. B.; DINIZ, P. C.; BONATO, V. Application-oriented cache memory configuration for energy efficiency in multi-cores. **IET Computers Digital Techniques**, v. 9, n. 1, p. 73–81, 2015. ISSN 1751-8601. Citado na página 112.

SILVA, B. A.; DELBEM, A.; BONATO, V.; DINIZ, P. C. Runtime mapping and scheduling for energy efficiency in heterogeneous multi-core systems. In: **2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)**. [S.l.: s.n.], 2015. p. 1–6. Citado na página 112.

SIMUNIC, T.; BENINI, L.; ACQUAVIVA, A.; GLYNN, P.; MICHELI, G. D. Dynamic voltage scaling and power management for portable systems. In: **Proceedings of the 38th annual Design Automation Conference**. New York, NY, USA: ACM, 2001. (DAC '01), p. 524–529. ISBN 1-58113-297-2. Disponível em: <http://doi.acm.org/10.1145/378239.379016>. Citado na página 35.

SOARES, A. H. M. **Algoritmos de estimação de distribuição baseados em árvores filogenéticas**. Tese de doutorado. Instituto de Ciências Matemáticas e de Computação. Universidade de São Paulo, 2014. Citado na página 44.

SOUISSI, O.; ABDELHAKIM, A. Two exact methods for mapping on heterogeneous CPU/FPGA architectures. In: **Networking, Sensing and Control (ICNSC), 2013 10th IEEE International Conference on**. [S.l.: s.n.], 2013. p. 520–525. Citado na página 90.

SPARC International Inc. The SPARC Architecture Manual. 1991. Disponível em: <http://www.gaisler.com/doc/sparcv8.pdf>. Citado na página 49.

WU, Q.; JUANG, P.; MARTONOSI, M.; CLARK, D. W. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In: **Proceedings of the 11th International Symposium on High-Performance Computer Architecture**. Washington, DC, USA: IEEE Computer Society, 2005. (HPCA '05), p. 178–189. ISBN 0-7695-2275-0. Disponível em: <http://dx.doi.org/10.1109/HPCA.2005.43>. Citado na página 35.

XIE, F.; MARTONOSI, M.; MALIK, S. Compile-time dynamic voltage scaling settings: opportunities and limits. In: **Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation**. New York, NY, USA: ACM, 2003. (PLDI '03), p. 49–62. ISBN 1-58113-662-5. Disponível em: <http://doi.acm.org/10.1145/781131.781138>. Citado na página 35.

ZHURAVLEV, S.; SAEZ, J.; BLAGODUROV, S.; FEDOROVA, A.; PRIETO, M. Survey of Energy-Cognizant Scheduling Techniques. **Parallel and Distributed Systems, IEEE Transactions on**, v. 24, n. 7, p. 1447–1464, July 2013. ISSN 1045-9219. Citado 2 vezes nas páginas 32 e 38.