Pós-Graduação em Ciência da Computação

Marcus Vinicius Duarte dos Santos

UMA ABORDAGEM BASEADA EM METAHEURÍSTICAS PARA EXPLORAÇÃO DO ESPAÇO DE PROJETO DE MEMÓRIAS CACHE MULTINÍVEL EM PLATAFORMAS MULTI-CORES PARA APLICAÇÃO ESPECÍFICA

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2017

Marcus Vinicius Duarte dos Santos

UMA ABORDAGEM BASEADA EM METAHEURÍSTICAS PARA EXPLORAÇÃO DO ESPAÇO DE PROJETO DE MEMÓRIAS CACHE MULTINÍVEL EM PLATAFORMAS MULTI-CORES PARA APLICAÇÃO ESPECÍFICA

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientadora: *Edna Natividade da Silva Barros*

RECIFE
2017

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S237a Santos, Marcus Vinicius Duarte dos
Uma abordagem baseada em metaheurísticas para exploração do espaço de projeto de memórias cache multinível em plataformas multi-cores para aplicação específica / Marcus Vinicius Duarte dos Santos. – 2017.
133 f.: il., fig., tab.

Orientadora: Edna Natividade da Silva Barros.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências.

1. Engenharia da computação. 2. Sistemas embarcados. 3. Otimização. I. Barros, Edna Natividade da Silva (orientadora). II. Título.

621.39 CDD (23. ed.) UFPE- MEI 2017-142

Marcus Vinicius Duarte dos Santos

UMA ABORDAGEM BASEADA EM METAHEURÍSTICAS PARA EXPLORAÇÃO DO ESPAÇO DE PROJETO DE MEMÓRIAS CACHE MULTINÍVEL EM PLATAFORMAS MULTI-CORES PARA APLICAÇÃO ESPECÍFICA

Tese de doutorado apresentada ao programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação

Aprovado em: 09/03/2017.

Orientadora: Profa. Dra. Edna Natividade da Silva Barros

BANCA EXAMINADORA

Prof. Dr. Manoel Eusébio de Lima
Centro de Informática / UFPE

Prof. Dr. Abel Guilhermino da Silva Filho
Centro de Informática / UFPE

Prof. Dr. Adriano Lorena Inácio de Oliveira
Centro de Informática / UFPE

Prof. Dr. Ivan Saraiva Silva
Departamento de Computação / UFPI

Prof. Dr. Edward David Moreno Ordonez
Departamento de Computação/UFS

Dedico este trabalho a todos que de alguma forma me deram suporte e tornaram possível a sua concretização.

Agradecimentos

Obrigado a minha orientadora Edna N. S. Barros pelo incentivo, paciência e credibilidade no trabalho. Obrigado a Maria Milet Pinheiro, Gabriela Pinheiro Duarte e Cecília Pinheiro Duarte pelo incentivo e compreensão nos momentos difíceis e no dia-a-dia. Obrigado ao prof. André Aziz pelo apoio teórico em todas as fases do trabalho, nas publicações submetidas e nas reuniões de acompanhamento.

Agradeço aos colegas de pós-graduação Max Santana, Rafael Macieira, Marília Lima, Dário Dias, Pyetro Ferreira, Paulo Guedes e Lucas Cambuim pelos debates que me ajudaram bastante a trilhar o caminho do meu trabalho. Obrigado também aos prof. George Lima e Max Santana por esclarecimentos em escrita dos artigos submetidos a conferências.

Agradeço também aos funcionários do CIN-UFPE Suilan Dias, Roberto Mariano, Nadja Lins, Maria do Socorro Oliveira, Carlos Melo e Márcia Porto, pelo apoio em assuntos administrativos e de suporte.

Gostaria também de agradecer ao Instituto Federal de Pernambuco (IFPE), campus Caruaru, pela concessão de meu afastamento de minhas atividades de professor para realização do trabalho de doutorado com dedicação exclusiva por dois anos, entre setembro de 2014 e agosto de 2016.

"O que dá o verdadeiro sentido ao encontro é a busca, e é preciso andar muito para se alcançar o que está perto."

—JOSÉ SARAMAGO

Resumo

A evolução dos computadores tem nos mostrado que, com o passar dos anos, esses equipamentos têm evoluído em diversas características, como novas tecnologias em uso, redução de tamanho, redução de custo, aumento do desempenho, e redução do consumo de energia. Entre essas melhorias destacamos como fundamentais para projetos de sistemas embarcados as melhorias em desempenho de aplicação específica e a melhoria em consumo. Em um sistema microprocessado, um dos principais responsáveis pelo consumo de energia é a hierarquia de memória cache, que pode ser responsável por até 50% da energia consumida pelo sistema completo. Nesse trabalho é apresentada uma abordagem para exploração do espaço de projeto de memórias cache em plataformas MPSoCs de aplicação específica que utiliza como base o algoritmo ABCs (Colônia Artificial de Abelhas) adaptado para multi-objetivo (melhoria de desempenho e de consumo de energia) e utilizando técnicas de DoE (*Design of Experiments*) para tornar a busca global do algoritmo mais eficiente, reduzindo seu tempo total de execução. O algoritmo ABC modificado foi denominado de algoritmo AbcDE. Nos experimentos avaliamos a abordagem AbcDE executando algumas aplicações dos *benchmark* Splash2 (fft, radix e matrix) e o ParMibench (Djikstra) para um nível de cache (L1) e foi obtido um conjunto de configurações da cache L1 dentro do *Pareto front* reduzindo o tempo de exploração em uma média de 42,3%. O número de simulações da plataforma MPSoC foi reduzida em 40,4% quando comparado com o uso do algoritmo ABC original em multi-objetivo. Os resultados foram obtidos para uma plataforma MPSoC baseada em NoC com 4 processadores. Também avaliamos a abordagem AbcDE executando as aplicações dos *benchmarks* previamente citados em conjunto com as aplicações do benchmark ParMibench (Sha, Stringsearch e Basicmath) para hierarquia de cache em multinível (L1 e L2). Foram obtidas configurações de cache dentro do *Pareto Front* apresentando uma quantidade média de execuções da plataforma MPSoC em cerca de 37,14% menor que o algoritmo ABCMOP, e em cerca de 37,10 % menor que o algoritmo MOPSO (considerando todas as aplicações dos experimentos). Mesmo obtendo uma melhoria significativa em termos eficiência, comparado aos algoritmos ABCMOP e MOPSO, o algoritmo AbcDE não degradou sua precisão. O algoritmo AbcDE, em termos de hipervolume, foi em média inferior ao algoritmo ABCMOP em apenas 0,91%, e foi em média superior ao algoritmo MOPSO em apenas 0,66%. Verificamos que o algoritmo AbcDE conseguiu obter resultados ótimos para configurações de cache multi-nível com eficiência e sem degradar sua precisão, simulando apenas cerca de 0,13% do espaço do projeto total da hierarquia de cache.

Palavras-chave: *Exploração de Espaço de Projeto. Inteligência de Enxame. Metaheurística. MPSoC. Algoritmo ABC. Projeto de Experimentos.*

Abstract

The computer's evolution has shown over the years these devices have evolved in several features such as new technologies in use, size reduction, cost reduction, increased performance, and reduced energy consumption. Among these improvements we highlight as fundamental to embedded system design, the improvements in performance and energy consumption. In a microprocessor-based system, the major contributor to the energy consumption is the cache hierarchy, which can account for up to 50% of the energy consumed by the entire system. This work introduces the AbcDE, a cache design space exploration approach to application-specific MPSoC platforms. The AbcDE uses the algorithm ABC (Artificial Bee Colony) in multi-objective mode (improvement of performance and energy consumption simultaneuosly) and using DoE (Design of Experiments) techniques to improve the efficiency of algorithm global search, reducing the execution time. In the experiments we evaluated the AbcDE approach to some applications of Splash2 benchmark (fft, radix and matrix multiplication) and ParMiBench benchmark (Djikstra) and was obtained a L1 cache configurations set into the Pareto front with a reduction of 42.3% in the exploration time. The mean number of platform executions is 40.4% lower when compared with the original multi-objective ABC algorithm. All results were obtained for a NoC-based MPSoC platform using four processors. We also evaluated the AbcDE approach by executing the previously cited benchmark applications in conjunction with the benchmark applications Sha, Stringsearch and Basicmath (ParMibench benchmark) for multilevel cache hierarchy (L1 and L2). Cache configurations within Pareto Front were obtained and it was obtained a mean number of MPSoC platform simulations at about 37,14 % smaller than the ABCMOP algorithm, and about 37,10 % smaller than the MOPSO algorithm (Considering all applications of the experiments). Although obtaining a significant improvement in efficiency terms, compared to the ABCMOP and MOPSO algorithms, the AbcDE algorithm did not degrade its accuracy. The AbcDE algorithm, in terms of hypervolume metric, obtained on average less than the ABCMOP algorithm by only 0.91%, and obtained on average superior to the MOPSO algorithm by only 0.66%. The AbcDE algorithm was able to achieve optimal results for multi-level cache configurations efficiently and without degrading its accuracy, simulating only about 0.13 % of the total design space of the cache hierarchy.

Keywords: *Design Space Exploration. Swarm Intelligence. Metaheuristic. MPSoC. ABC Algorithm. Design of Experiments.*

Lista de Figuras

1.1 Crescimento no desempenho dos processadores desde 1970. 19
1.2 Diferença de desempenho entre Processador e Memória DRAM. 19
1.3 Diferença de densidade entre Processador e Memória DRAM. 20
1.4 Consumo de Energia do Xeon 4 (Usando os benchmark suites SPEC-CPU2006, SPEC-OMP2001 e NAS-NPB). 20
1.5 Análise de Pareto para diferentes configurações de cache. 21
1.6 Fluxo de Síntese de Sistema Embarcado dados a plataforma e o mapeamento. . . . 24

2.1 Velocidade e tamanho de memórias. 27
2.2 Hierarquia de memórias. 27
2.3 Funcionamento da memória cache. 29
2.4 Mapeamento Associativo (totalmente Associativo). 30
2.5 Mapeamento Associativo por conjunto (2-*way* associativo). 31
2.6 L3 - Cache de último nível (LLC). 37
2.7 Exemplo de população balanceada. 42
2.8 Exemplo de DoE com parâmetros A, B e C. 45
2.9 Exemplo de DoE com parâmetros A, B e C incluindo cálculo de $y_{i..}$, $y_{.j.}$, $y_{..k}$ e $y_{...}$ e seus valores quadrados. 47
2.10 Valores médios, máximos e mínimos de tempo de execução para os tamanhos de cache de 1Gb, 2Gb e 4Gb. 50
2.11 Resultados Conjunto Pareto 1, Conjunto Pareto 2 e o Pareto Ótimo. 51
2.12 Hipervolume do Conjunto Pareto 1 e do Conjunto Pareto 2. 51
2.13 Diferença do Hipervolume do Conjunto Pareto 1, do Conjunto Pareto 2 e do Pareto Ótimo. 51

3.1 Classificação das Técnicas de Cache Tuning (ZANG; GORDON-ROSS, 2013). . . 54
3.2 Visão Geral da Operação do USPACS. 57
3.3 *Speedup* de tempo de simulação do U-SPaCS comparado ao Dinero. 58
3.4 Arquitetura MPSoC. 59
3.5 Sistema de Exploração de Espaço de Projeto. 62
3.6 Exemplo de Uso dos Simuladores de Hardware(hSim) no Nivel 2 da Hierarquia de Cache. 62
3.7 Interface e Estrutura do Módulo de *Hardware* - hSim. 62
3.8 Resultados do experimentos sobre o sistema A: (a) selecionada a configuração de tamanho para as caches $C_{i,j}$; (b) resultados de $T_{i,j,k_{min}}$ para as caches $C_{i,j}$ como verificado no algoritmo; T_{TOT}, configuração obtida em cada passo da iteração. . . . 64

3.9 Resultados do experimentos sobre o sistema B: (a) selecionada a configuração de tamanho para as caches $C_{i,j}$; (b) resultados de $T_{i,j,k_{min}}$ para as caches $C_{i,j}$ como verificado no algoritmo; T_{TOT}, configuração obtida em cada passo da iteração. . . . 65
3.10 Hierarquia de cache de dois-níveis em arquitetura MPSoC. 66
3.11 Fluxo de simulação do DIMSim. 67
3.12 Arquitetura SoC com Hierarquia de memória. 72
3.13 ABCMOP versus Simulação Exaustiva para as seguintes aplicações do MIBench: (a)-Bitcount, (b)-CRC, (c)-Sha, (d)-Djpeg, (e)-Mpeg2dec, (f)-H263enc. 74
3.14 Tempo de Simulação ABCMOP versus Tempo de Simulação Exaustiva. 75
3.15 Número de Simulações necessárias para cada Algoritmo e aplicação. 75

4.1 Exemplo de cálculo do Indicador de Taxa de Dominação Mútua. 82
4.2 Diagrama em blocos do AbcDE. 85
4.3 Definição do total de execuções do Algoritmo AbcDE através da análise do Tamanho da Amostra, utilizando a métrica de número de simulações da plataforma. 91
4.4 População Balanceada de configurações de cache (*SimulatedFoods*). 96
4.5 População Desbalanceada de configurações de cache (*SimulatedFoods*). 97
4.6 Fluxograma de seleção de parâmetro e nível para obter a fonte de alimento candidata v_{id}. 99
4.7 Modelo de energia para sistemas *multicore*. 101
4.8 Modelo de energia para cache de apenas um nível (L1). 102
4.9 Modelo de energia para cache de dois níveis. 103

5.1 Plataforma MPSoC *Infinity* 105
5.2 Diagrama do sistema em exploração de espaço de projeto AbcDE em interação com a plataforma Infinity. 107
5.3 Consumo de energia versus Desempenho - ABC Multi-Objetivo, Algoritmo AbcDE e Exaustivo para as aplicações Matrix, Radix, FFT e Djikstra. 112
5.4 Resultados para tempo médio de execução, Simulações da Plataforma, e ADRS. . . 113
5.5 Distribuição de publicações de algoritmos de metaheurística. 115
5.6 Áreas de aplicação publicadas com uso do MOPSO. 115
5.7 Consumo de energia versus Desempenho - Algoritmo AbcDE, MOPSO e ABCMOP para as aplicações Matrix, Radix, FFT e Djikstra. 117
5.8 Consumo de energia versus Desempenho - Algoritmo AbcDE, MOPSO e ABCMOP para as aplicações Sha, StringSearch e Basicmath. 118
5.9 Resultados para número médio de simulações da Plataforma para aplicações Matrix, Radix, FFT e Djikstra (a) e Sha, Stringsearch e Basicmath (b). 120
5.10 Resultado de redução de simulações da plataforma devido ao reuso de simulações. . 120

5.11 Tempo de execução do algoritmos AbcDE, ABCMOP e MOPSO a- para as aplicações Matrix, Radix, FFT e Dijkstra, e b- para as aplicações Sha, Stringsearch e BasicMath. 121
5.12 Resultados de hiper-volume médio para execução das aplicações Matrix, Radix, FFT e Djikstra (a) e Sha, Stringsearch e Basicmath (b). 123
5.13 Uso de memória na execução dos algoritmos AbcDE, ABCMOP e o MOPSO utilizando a aplicação Matrix. 123

Lista de Tabelas

2.1 Exemplo Kruskal-Wallis 48
2.2 Exemplo Kruskal-Wallis com os postos de cada elemento. 49
2.3 Valores de Z para principais intervalos de confiança 52
2.4 Amostra Populacional exemplo. 52

3.1 Porção explorada do espaço de projeto comparado com trabalho prévio de Nawinne (NAWINNE et al., 2014) 63
3.2 Análise Comparativa do Estado da Arte. 76

4.1 Teste de Kruskal-Wallis para números de fontes de alimento usando a aplicação Matrix. 87
4.2 Teste de Kruskal-Wallis para números de fontes de alimento usando a aplicação Radix. 87
4.3 Teste de Kruskal-Wallis para números de fontes de alimento usando a aplicação FFT. 88
4.4 Teste de Kruskal-Wallis para o parâmetro Limite o usando a aplicação Matrix. . . . 89
4.5 Teste de Kruskal-Wallis para o parâmetro Limite o usando a aplicação Radix. 90
4.6 Teste de Kruskal-Wallis para o parâmetro Limite o usando a aplicação FFT. 90
4.7 Total de execuções do algoritmo para cada aplicação. 92
4.8 Matriz utilizada para geração de população balanceada de 81 fontes de alimento. . . 94

5.1 Dados do Pareto set exibidos nas figuras 5.7 e 5.8 119
5.2 Dados de valores médios e desvio padrão das execuções dos algoritmos AbcDE. ABCMOP e MOPSO para as simulações das aplicações Matrix, Radix, FFT, Dijkstra, Sha, StringSearch e BasicMath na plataforma MPSoC. 122
5.3 Análise Comparativa do Estado da Arte com o ABcDE. 124

Lista de Acrônimos

ABC	Artficial Bee Colony
AbcDE	Artficial Bee Colony Design space Explorer
ABCMOP	Artficial Bee Colony for Multi-Objective Problems
ACO	Ant Colony Optimisation
ADRS	Average Distance from Reference Set
ANOVA	ANalysis Of VAriance
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DE	Differential Evolution
DoE	Design of Experiments
DRAM	Dynamic Random Access Memory
EA	Evolutionary Algorithm
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
GA	Genetic Algorithm
GRASP	Greedy Ramdomized Adptative Procedure
LLC	Last Level Cache
MDRI	Mutual Domination Rate Indicator
MOGA	Multi-Objective Genetic Algorithm with elitism
MOPSO	Multi-Objective Particle Swarm Optimization
MPSoC	MultiProcessor System-on-Chip
NSGA	Non-dominated Sorting Genetic Algorithm II
NoC	Network-on-Chip
pMOS	P-type Metal Oxide Semiconductor
nMOS	N-type Metal Oxide Semiconductor
PSO	Particle Swarm Optimization
RDAT	Requested Data Access Time
RHRT	Requested HeadRoom Time

DIMSim	Dual Independent Modular Simulation
SDS	Stochastic Diffusion Search
SOA	Swarm-based Optimisation Algorithms
SoC	System-on-Chip
SRAM	Static Random Access Memory

Sumário

1 Introdução 18
1.1 Motivação 18
1.2 Objetivo do Trabalho 25
1.3 Estrutura do Documento 25

2 Conceitos Básicos 26
2.1 Sistema de Cache 26
2.1.1 Funcionamento das memórias Cache 28
2.1.2 Parâmetros de Configuração de cache 28
2.1.2.1 Tamanho Total da Cache 28
2.1.2.2 Mapeamento de Cache 29
2.1.2.3 Tamanho do Bloco da Cache 30
2.1.2.4 Política de Escrita 31
2.1.3 Consumo de Energia 32
2.1.4 Tuning de Cache 33
2.1.4.1 Tuning Estático de Cache 34
2.1.4.2 Tuning Dinâmico de Cache 34
2.2 O problema de exploração de espaço de projeto com múltiplos processadores conectados por NoC 35
2.2.1 Aumento do Espaço de Projeto de Cache 36
2.2.2 Coerência de Cache 36
2.2.3 Caches de Último Nível compartilhadas 36
2.2.4 Uso de Interconexão por NoC 37
2.3 Introdução a Metaheurística e Algoritmo ABC (Colônia Artificial de Abelhas) 38
2.3.1 Algoritmo ABC (Colônia Artificial de Abelhas) 39
2.4 DoE - Projeto de Experimentos 41
2.4.1 DoE - Uso de População Balanceada 41
2.4.2 DoE - Análise Estatística do Modelo de Efeitos 43
2.5 Teste de Kruskal-Wallis 46
2.6 Indicador de Hipervolume 50
2.7 Determinação de Tamanho de Amostra 51

3 Trabalhos Relacionados . . . 54
3.1 Configuração com Base em Simulação . . . 55
3.1.1 U-SPaCS - Uma Metodologia de Simulação de Cache em passo Único para Caches de Dois-Níveis Unificadas . . . 56
3.1.2 Nawinne2015 - Explorando Hierarquias de Cache Multinível em MPSoCs de Aplicação Específica . . . 59
3.1.3 DIMSim - Uma Abordagem Rápida de Simulação de Cache de Dois Níveis para MPSoC baseado em Deadline . . . 66
3.1.3.1 Estágio-1: Configuração de cache L2 de dados que satisfaça o RDAT . . . 68
3.1.3.2 Estágio-2: Trace secundário para o simulador da cache L1 . . . 68
3.1.3.3 Estágio-3: Configurações de cache L1 para a cache L2 compartilhada . . . 69
3.1.4 ABCMOP - Algoritmo de Colônia de Abelhas aplicado a Exploração de Arquitetura de Memória para Redução de Energia . . . 72
3.2 Análise Comparativa do Estado da Arte . . . 76

4 Abordagem Proposta . . . 78
4.1 Introdução . . . 78
4.2 Algoritmo ABC modificado para Otimização MultiObjetivo . . . 78
4.2.1 Análise de Dominância de Pareto . . . 79
4.2.2 Cálculo da função de adequação (Fitness) . . . 79
4.2.3 Critério de Parada Automático . . . 81
4.3 O Algoritmo proposto AbcDE . . . 83
4.3.1 Definição dos parâmetros do Algoritmo AbcDE . . . 86
4.3.1.1 Definição do Número de Fontes de Alimento e População de abelhas . . . 86
4.3.1.2 Definição do Limite . . . 88
4.3.1.3 Definição do Total de Execuções do Algoritmo para cada aplicação . . . 89
4.3.2 Otimização da Busca Global usando DoE . . . 92
4.3.2.1 Uso de População Balanceada . . . 93
4.3.2.2 Uso da análise estatística do modelo de efeitos . . . 93
4.3.2.3 Cálculo do impacto dos níveis de cada parâmetro de cache . . . 95
4.3.2.4 Seleção de fonte de alimento com base em impacto . . . 98
4.4 Modelo de Energia . . . 100
4.4.1 Modelo de energia para cache de apenas um nível (L1) . . . 100
4.4.2 Modelo de energia para cache multinível . . . 101

5 Resultados . . . 104
5.1 Ambiente de Simulação . . . 105
5.2 Plataforma de Simulação . . . 105
5.2.1 Reuso das configurações previamente simuladas . . . 107

5.3 Aplicações Selecionadas para Simulação na Plataforma MPSoC 108
5.3.1 Aplicação Multiplicação de Matrizes - Matrix 108
5.3.2 Aplicação Radix 109
5.3.3 Aplicação FFT 109
5.3.4 Aplicação Dijkstra 109
5.3.5 Aplicação Sha 109
5.3.6 Aplicação StringSearch 110
5.3.7 Aplicação BasicMath 110
5.4 Resultados do Algoritmo AbcDE aplicado a Cache de Único Nível - L1 . . . 110
5.5 Resultados do Algoritmo AbcDE aplicado a Cache Multinível 111
5.6 Análise Comparativa com o Estado da Arte 124

6 Conclusão 126

Referências 129

1 Introdução

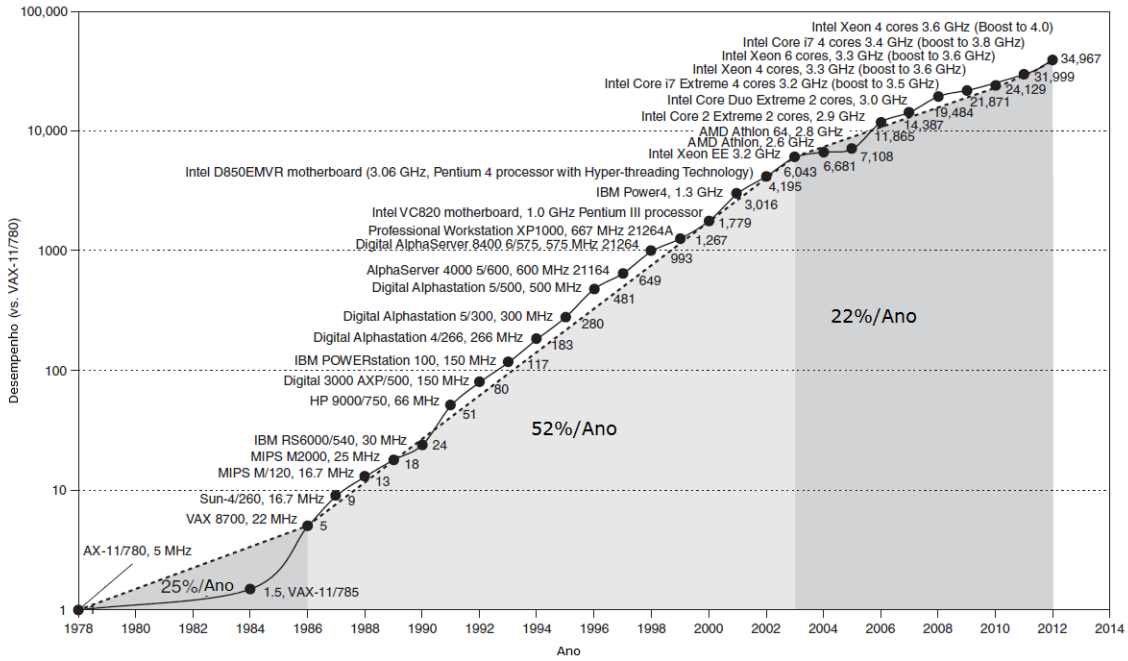
O propósito deste capítulo é descrever as principais motivações para a realização deste trabalho e de oferecer uma visão global dos caminhos traçados para realização do mesmo. No decorrer do capítulo são apresentadas, além da motivação, o objetivo principal e os objetivos secundários para a realização do trabalho proposto.

1.1 Motivação

A evolução dos computadores tem nos mostrado que, com o passar dos anos, esses equipamentos têm melhorado em diversos aspectos, como tecnologias de fabricação, redução de tamanho, redução de custo, aumento do desempenho, e redução do consumo de energia. Os processadores são os principais componentes desses dispositivos e a figura 1.1 mostra o seu crescimento em termos de desempenho, desde os anos 70 até o dias atuais (HENNESSY; PATTERSON, 2017). Na referida figura podemos verificar que esse crescimento é mostrado em escala logarítmica e que houve três fases de crescimento no tempo: a primeira mostra uma evolução do desempenho a uma ordem de 25% ao ano (entre os anos 70 e 80); a segunda tem um crescimento na ordem de 50% ao ano (entre os anos 80 e início dos anos 2000); e uma terceira fase, em que a taxa de crescimento foi 22% ao ano.

Em um sistema computacional, como um sistema embarcado, existem diversos componentes que trocam informações para a execução de uma ou mais aplicações. Esses componentes são processadores, memória, dispositivos de entrada/saída, hardware dedicados, barramentos, etc. Entre eles destacamos o processador e a memória principal, como elementos fundamentais e de intensa troca de informação, que merecem atenção especial, pois trabalham trocando dados em altas velocidades e consumindo bastante tempo e energia nesse trabalho. Porém, como verificado na figura 1.2, existe uma diferença de desempenho bem considerável entre processadores e memória principal, o que levou à utilização de mais de um nível de memória rápida na hierarquia de memória, exatamente entre o processador e as memórias principais, para melhorar o tempo médio de acesso aos dados e instruções. Essas memórias rápidas, aplicadas entre o processador e a memória principal, são usadas até hoje e são chamadas de memórias cache (vide descrição

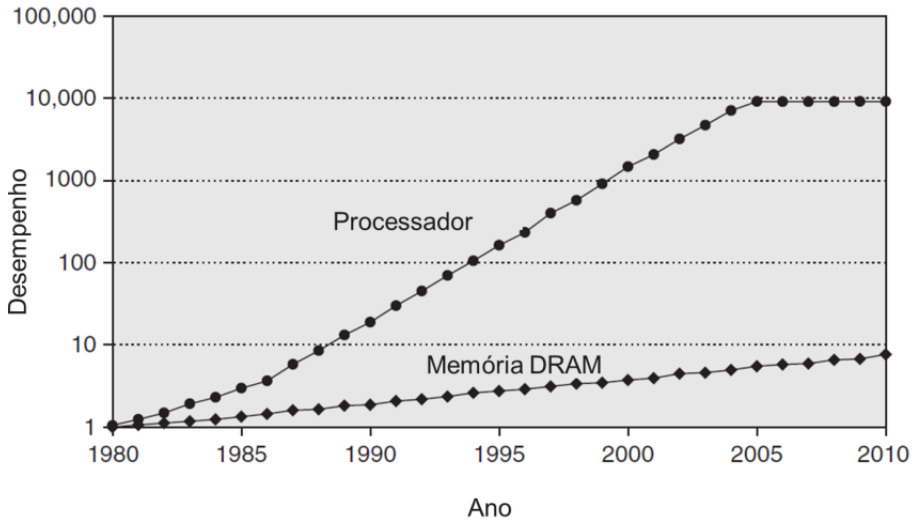
Figura 1.1: Crescimento no desempenho dos processadores desde 1970.



Fonte: HENNESSY; PATTERSON (2017).

sobre hierarquia de memória no capítulo 2).

Figura 1.2: Diferença de desempenho entre Processador e Memória DRAM.

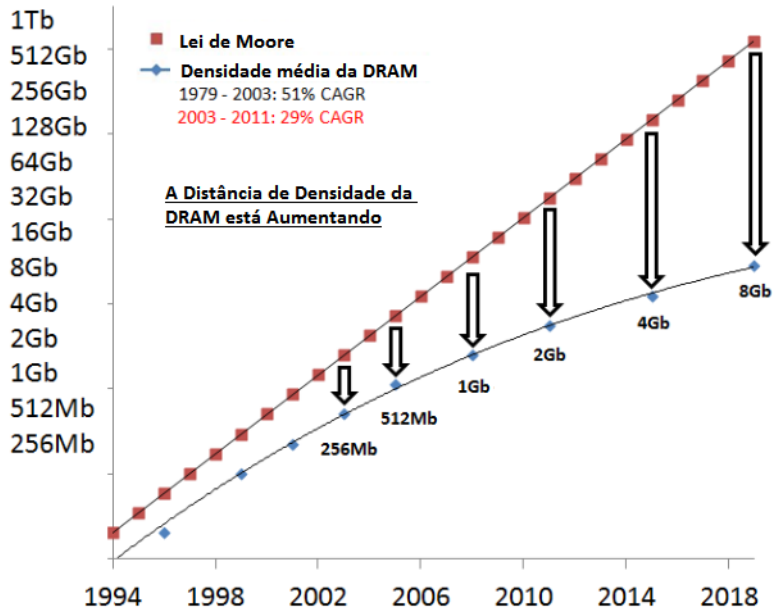


Fonte: HENNESSY; PATTERSON (2012).

A diferença de desempenho entre processadores e memórias também é comprovada através do crescimento da diferença de densidade entre as memórias e os processadores. A figura 1.3 mostra como os processadores cresceram mais em números de transistores, comparados com as memórias DRAMs (usadas como memórias principais) desde 1994 até os dias atuais.

Estudos comprovam que o acesso à memória corresponde a cerca de 50% do consumo de

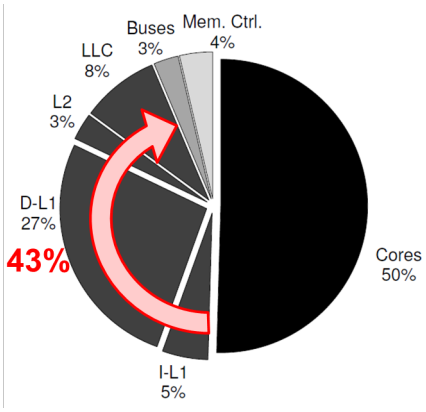
Figura 1.3: Diferença de densidade entre Processador e Memória DRAM.



Fonte: Hewlett-Packard Development Company (Memory-Driven Computing).

energia de um processador em sistemas com um único processador (GORDON-ROSS; VAHID; DUTT, 2009)(VIANA et al., 2008). Em sistemas com múltiplos núcleos processadores esse consumo de energia é bem semelhante. Temos a comprovação que o consumo de energia em acessos à memória de múltiplos processadores em um sistema MPSoC (Sistema com múltiplos processadores em um circuito integrado) fica em torno de 40-50% do consumo total de um MPSoC, como verificamos através da figura 1.4 (ALVES, 2014).

Figura 1.4: Consumo de Energia do Xeon 4 (Usando os benchmark suites SPEC-CPU2006, SPEC-OMP2001 e NAS-NPB).

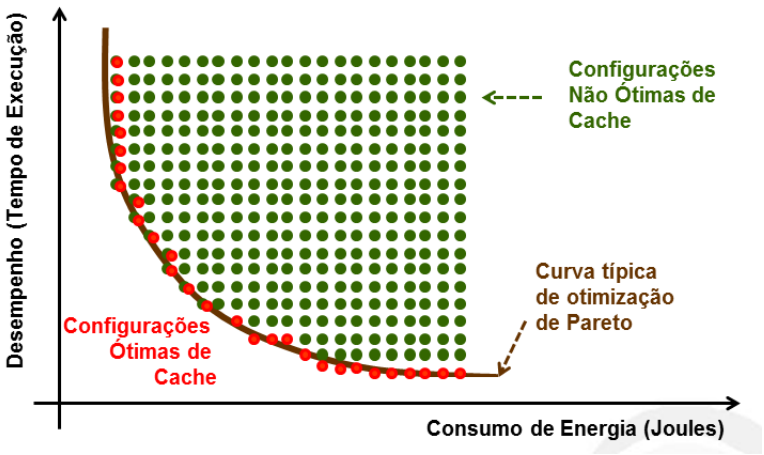


Fonte: ALVES (2014).

A otimização simultânea de métricas de consumo de energia e de desempenho em um sistema com múltiplos núcleos processadores é um grande desafio devido a serem grandezas

que variam de forma divergente. Por exemplo, mudanças nas configurações de cache que melhoram desempenho podem gerar aumento no consumo de energia. Da mesma forma que melhorias no consumo de energia podem degradar o desempenho na execução de aplicações. As diferentes possíveis configurações de hierarquia de cache compõem o que chamamos de espaço de projeto de cache, e podemos ilustrar o efeito dessas diferentes configurações com relação ao desempenho e consumo de energia da cache através da figura 1.5 (MICHANAN; DEWRI; RUTHERFORD, 2014). Cada diferente configuração de cache (executada para uma aplicação) pode gerar diferentes valores de desempenho e consumo de energia (pontos circulares na figura 1.5). Como verificado na figura 1.5, as configurações de cache mais próximas à curva típica de Pareto são as chamadas configurações ótimas de Pareto, e são as configurações de cache otimizadas para os objetivos de maior desempenho e menor consumo de energia.

Figura 1.5: Análise de Pareto para diferentes configurações de cache.



Fonte: MICHANAN; DEWRI; RUTHERFORD (2014).

Os MPSoCs usados em projetos de sistemas embarcados favorecem a otimização de suas hierarquias de cache para melhor desempenho e menor consumo de energia porque esses sistemas executam uma aplicação (ou um conjunto de aplicações) de forma dedicada (sistema projetado para uma finalidade específica). Portanto, os MPSoCs permitem a configuração de número de níveis de cache em sua hierarquia, assim como suas caches individuais em termos de tamanho de bloco, associatividade, tamanho total da cache, política de relocação, etc. Diversos trabalhos têm mostrado que otimização de parâmetros de hierarquia de cache favorecem consideravelmente a redução de consumo de energia e a melhoria de desempenho de sistemas computacionais (HENNESSY; PATTERSON, 2012)(GORDON-ROSS; VAHID; DUTT, 2009)(SILVA-FILHO; CORDEIRO, 2011)(ALSAFRJALANI; GORDON-ROSS, 2016)(NAWINNE et al., 2015).

Encontrar as configurações ótimas de hierarquia de cache, principalmente para sistemas MPSoCs, não é uma tarefa fácil devido a necessidade de se lidar com um espaço de projeto de hierarquia de cache bastante aumentado (comparado ao espaço de projeto dos sistemas SoC), e lidar com coerência de cache (vide descrição sobre o problema de exploração de espaço de

projeto com múltiplos processadores no capítulo 2). As diferentes configurações de hierarquia de cache são conseguidas variando os valores dos parâmetros de cada cache, tais como tamanho total da cache, nível de associatividade e tamanho do bloco. Por exemplo, se uma cache possui 10 diferentes combinações de seus parâmetros, e uma hierarquia de cache possui 2 níveis, a hierarquia de cache terá um espaço de projeto de $10^2 = 100$ diferentes combinações, considerando um sistema SoC. Passando para um MPSoCs de 4 processadores, considerando que não há restrições no uso dos parâmetros, a mesma hierarquia de cache terá um espaço de projeto de $(10^2)^4 = 100.000.000$ diferentes combinações (considerando que todas as configurações obtidas são configurações válidads de cache). As configurações com maior tamanho total da cache são não necessariamente as de maior desempenho e maior consumo de energia. O projetista de um sistema embarcado precisa idealmente explorar todo o espaço de projeto para encontrar as melhores configurações da hierarquia de cache para satisfazer suas restrições de desempenho e consumo de energia, e considerando um gigantesco espaço de projeto, essa tarefa consumiria muito tempo de projeto.

Tuning de cache é como são chamadas as técnicas que exploram o espaço de projeto de cache, dinâmica ou estaticamente, buscando configurações de cache que otimizem objetivos como melhoria de desempenho, de consumo de energia e de redução de área de layout de circuitos integrados, para a execução de uma ou mais aplicações. As técnicas de *tuning* dinâmico de cache são realizadas em tempo de execução e necessitam de *hardware* adicional para realizar o gerenciamento do *tuning*. As técnicas estáticas de *tuning* de cache são realizadas em tempo de projeto, e são as mais aplicadas em projetos de sistemas embarcados devido ao conhecimento prévio das aplicações a serem executadas no sistema (vide descrição sobre *tuning* de cache no capítulo 2).

As técnicas que realizam *tuning* de cache em tempo de projeto foram divididas entre técnicas de configuração com base em modelagem analítica e técnicas de configuração com base em simulação (ZANG; GORDON-ROSS, 2013). As técnicas com base em modelagem analítica possuem uma computação muito complexa para problemas com muitos parâmetros e multi-objetivo, ainda podendo introduzir alguma imprecisão aos resultados, por não possuírem uma resposta linear. As técnicas com base em simulação se dividem entre técnicas *trace-driven* especializadas e técnicas de Aceleração de *Tuning* usando Exploração Eficiente de Espaço de Projeto. As técnicas trace-driven especializadas possuem baixa precisão temporal quando aplicadas a plataformas MPSoCs (BITAR, 1990) (GOLDSCHMIDT; HENNESSY, 1993) (ISSHIKI, 2010). As técnicas baseadas em exploração eficiente de espaço de projeto utilizam heurísticas ou algoritmos de otimização para reduzir o espaço de projeto e tentar convergir para uma configuração de cache ótima para atender os objetivos. O uso de heurísticas pode não evitar a convergência para configurações de cache que levam a um mínimo local. Tentando tornar a tarefa de busca em espaços n-dimencionais mais simples, o uso de algoritmos de otimização, em particular as metaheurísticas, têm aumentado bastante pois eles permitem resolver muitos problemas complexos de busca (KARABOGA; AKAY, 2009) (BOUSSAïD;

LEPAGNOT; SIARRY, 2013) de forma simples e eficiente. As técnicas com base em exploração de espaço de projeto com metaheurísticas possuem uma precisão muito boa de seus resultados quando a função objetivo é realizada através da simulação da plataforma MPSoC executando a aplicação. A simulação da plataforma também permite simular o uso de rede de interconexão através de NoC, simular toda a hierarquia de cache e realizar a comunicação entre tarefas utilizando coerência de cache. O único problema dessa abordagem é que o tempo de simulação fica bem aumentado (comparado com outras que não simulam a execução da plataforma), devido ao tempo de cada simulação da plataforma MPSoC necessários para a tarefa de *tuning* de cache.

As metaheurísticas implementam, de forma simples, estratégias matemáticas e estatísticas que são projetadas para se explorar o espaço de projeto com o objetivo de buscar um ou mais pontos de projeto que satisfaçam um ou mais objetivos. Entre as metaheurísticas podemos destacar alguns algoritmos, como o algoritmo genético (GA), o algoritmo genético de ordenação não dominada (NSGA–II), o algoritmo genético multi-objetivo (MOGA), o de otimização por colônia de formigas (ACO), o de Otimização por Enxame de Partículas (PSO) e o algoritmo de Colônia Artificial de Abelhas (ABC). Outra vantagem do uso de metaheurísticas é que muitas delas são computacionalmente simples, e de fácil implementação e adequação aos problemas de busca reais.

O algoritmo ABC (Colônia Artificial de Abelhas) é um algoritmo relativamente novo para uso em problemas de otimização global. O ABC é um algoritmo com otimização baseada em população de partículas categorizado na classe dos algoritmos baseados em enxame. Ele é capaz de trabalhar com problemas de otimização com ou sem restrições (KARABOGA; BASTURK, 2007a). O ABC tem sido utilizado em uma grande variedade de aplicações, citando alguns exemplos, como predição de estruturas de proteína (KARABOGA; BASTURK, 2007a), treinamento de redes neurais (IRANI; NASIMI, 2011), problemas matemáticos/computacionais (KARABOGA; AKAY, 2009), análise de *clusters* (KARABOGA; BASTURK, 2008) , problemas na área de processamento de imagens (RAHKAR-FARSHI; KESEMEN; BEHJAT-JAMAL, 2014), problemas de engenharia elétrica (NAIDU; MOKHLIS; BAKAR, 2014), problemas de engenharia eletrônica (HARIS; GOPINATHAN; ALI, 2012) e outras como previsão de mercado (KARABOGA; AKAY, 2011).

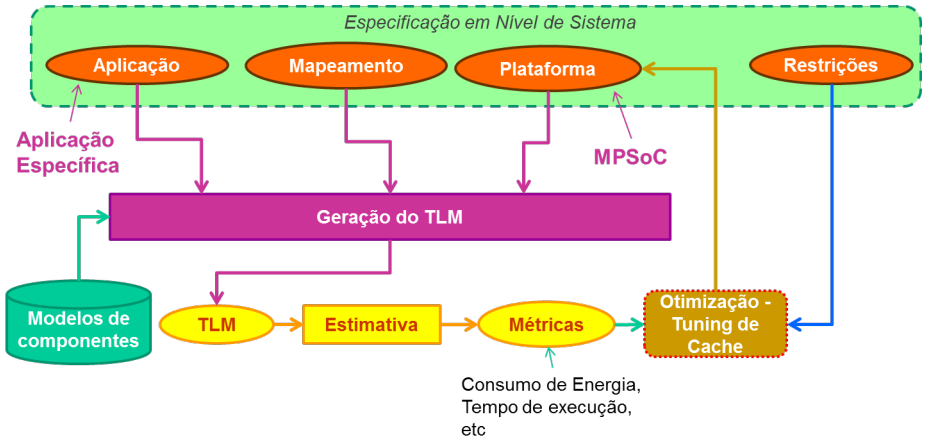
O algoritmo ABC foi selecionado neste trabalho devido a ser uma metaheurística largamente estudada e aplicada (exemplos citados acima) à soluções de problemas reais complexos (KARABOGA et al., 2014) e por possuir um processo de busca completo (KARABOGA; AKAY, 2009), favorecendo simultaneamente a exploração e a explotação do espaço de projeto, e evitando soluções em mínimo local. Outro trabalho de Karaboga também mostra o quanto o algoritmo ABC é superior em desempenho quando comparado a outros algoritmos como o PSO (Otimização por Enxame de Partículas), o DE (Evolução Diferencial), e o EA (Algoritmo Evolucionário) para problemas de engenharia com espaços de projetos com muitas dimensões/parâmetros, que é o caso de caches em plataformas multicore (KARABOGA; BASTURK, 2008).

Como o uso de metaheurísticas com simulação geram tempo de execução elevado

(devido às simulações da plataforma MPSoC), comparado com o uso de técnicas como *trace-driven*, propomos melhorar o tempo de exploração de espaço de projeto de cache introduzindo, ao algoritmo ABC, técnicas de Projeto de Experimentos(DoE) que exploram características específicas do problema para otimizar o tempo médio de execução do algoritmo (vide descrição sobre as técnicas exploradas do DoE no capítulo 2). Portanto, buscaremos neste trabalho realizar otimização de sistemas embarcados propondo uma abordagem de exploração de espaço de projeto de cache com multi-objetivo de otimização de consumo de energia e desempenho aplicáveis a arquiteturas MPSoC de aplicação específica, que executam aplicações cujas tarefas compartilham dados usando a metaheurística algoritmo ABC com tempo de execução otimizado utilizando técnicas de DoE.

Os projetistas de sistemas embarcados necessitam de ferramentas que os auxiliem a identificar um conjunto adequado de configurações de hierarquia de cache que atendam às restrições de projeto. A figura 1.6 mostra um fluxo típico de projeto de um sistema embarcado utilizando técnica de *Tuning de cache* como ferramenta de otimização para atender restrições de projeto atuando sobre a sua plataforma MPSoC. Neste contexto, a ferramenta de otimização fornece configurações de cache para uso na plataforma MPSoC alvo do projeto, e de forma iterativa, vai realizando o tuning de cache até chegar em uma ou mais configurações de cache otimizadas para as restrições do projeto e menor consumo de energia e melhor desempenho.

Figura 1.6: Fluxo de Síntese de Sistema Embarcado dados a plataforma e o mapeamento.



Algumas técnicas foram desenvolvidas para obter resultados de exploração de espaço de projeto de cache em plataformas SoC (GORDON-ROSS; VAHID; DUTT, 2009) (ZANG; GORDON-ROSS, 2012) (SANTOS; SILVA-FILHO, 2014). Outros trabalhos de pesquisa foram propostos para *tuning* de hierarquia de cache reduzindo o seu espaço de projeto em plataformas MPSoC (NAWINNE et al., 2015) (HAQUE et al., 2012)(RAWLINS; GORDON-ROSS, 2013)(WANG; MISHRA; RANKA, 2011)(ADEGBIJA; GORDON-ROSS, 2014). Esses trabalhos do estado da arte estão relacionados com o trabalho proposto e são detalhados no capítulo 3.

1.2 Objetivo do Trabalho

Objetivo Principal

O objetivo principal deste trabalho é desenvolver uma abordagem de exploração do espaço de projeto de hierarquia de caches, utilizando meta heurísticas, que vise encontrar configurações de caches com otimizações de desempenho e consumo de energia, em aplicações de sistemas embarcados, que fazem uso de arquiteturas multi-core, com coerência de cache e múltiplos níveis na hierarquia de cache.

Objetivos Secundários

- Definir/adaptar uma plataforma MPSoC e um conjunto de *benchmarks* para avaliação da abordagem proposta, realizando o *tuning* para mais de um nível de cache;
- Organizar/apresentar os resultados obtidos para a abordagem proposta de exploração de espaço de projeto de cache.

Contribuições

Neste trabalho estamos propondo uma abordagem de exploração do espaço de projeto de cache utilizando metaheurísticas como algoritmo base. Como o uso de metaheurísticas com simulação de plataforma MPSoC demanda muitas simulações, fazendo o tempo de execução da abordagem ser muito alto, nossa contribuição deve se localizar nos seguintes pontos:

- Buscar uma maior eficiência na qualidade da população inicial do algoritmo a ser utilizado, sem retirar a randomicidade no momento da criação dessa população;
- Tornar mais eficiente o mecanismo de busca do algoritmo a ser utilizado fazendo uso de outras técnicas e buscando explorar as características do problema de tuning de cache;

1.3 Estrutura do Documento

Este documento é organizado da seguinte forma: no capítulo 2 são descritos conceitos básicos iniciais para um melhor entendimento do trabalho proposto. Os trabalhos relacionados mais relevantes e atuais são descritos no capítulo 3. No capítulo 4, a abordagem proposta de exploração do espaço de projeto de hierarquia de caches é descrita. Nos capítulos 5 e 6 são detalhados, respectivamente, os resultados obtidos com os experimentos e algumas conclusões sobre a abordagem proposta, bem como sugestões para trabalhos futuros.

2

Conceitos Básicos

Este capítulo tem por objetivo apresentar conceitos básicos relacionados a este trabalho, entre os quais destacam-se: Sistemas de memória cache, equações para estimativa de energia consumidas em sistemas de cache, problemas de exploração de espaço de projeto com múltiplos processadores, introdução a metaheurística e algoritmo ABC (algoritmo de Colônia Artificial de Abelhas), técnicas de DoE (*Design of Experiments*), teste de *Kruskal-Wallis*, indicador de hipervolume, e determinação de tamanho de amostra.

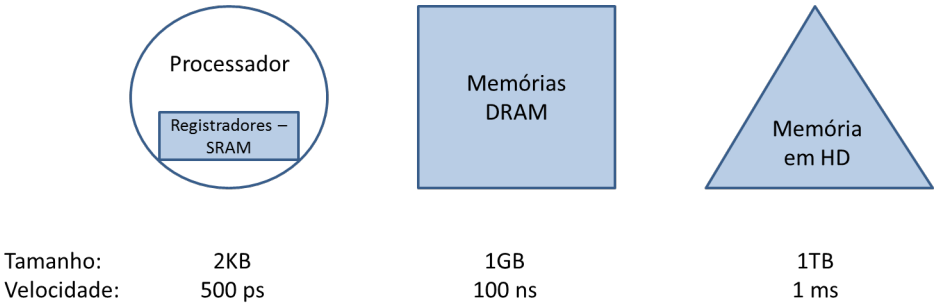
2.1 Sistema de Cache

Os processadores atuais conseguem realizar operações de escrita e leitura de dados e instruções bem mais rapidamente do que as unidades de memória utilizadas em computadores podem responder. A velocidade dos processadores aumentou conforme a lei de Moore da década de 80 até os anos 2000/2004. A velocidade de acesso das memórias não evoluiu na mesma taxa que os processadores. A distância entre a velocidade de execução dos processadores e a velocidade de acesso à memória vem crescendo com o decorrer dos anos, como pôde ser observado através das figuras 1.2 e 1.3.

Existem tecnologias de memória que respondem rapidamente às solicitações de um processador, porém essas memórias são de custo muito elevado para serem utilizadas em grande quantidade. Nas tecnologias de memória verifica-se que quanto maior a capacidade de armazenamento, menor é sua velocidade de acesso e custo. A figura 2.1 mostra a ordem de grandeza de velocidade e tamanho de alguns tipos de memória atuais.

As memórias SRAM (*Static Random Access Memory*) são memórias rápidas que utilizam tecnologia de *latchs* e *flip-flops* para armazenamento de dados. As memórias SRAM são usadas para implementar os registradores internos acessados pelos processadores. A latência dos processadores e das memórias SRAM possuem valores de mesma ordem de grandeza. As memórias DRAM (*Dynamic Random Access Memory*), por sua vez, são memórias que utilizam capacitores para armazenamento de dados. Elas são bem mais lentas que as memórias SRAM, porém possuem menor custo. A memória principal de um computador incluem memórias

Figura 2.1: Velocidade e tamanho de memórias.

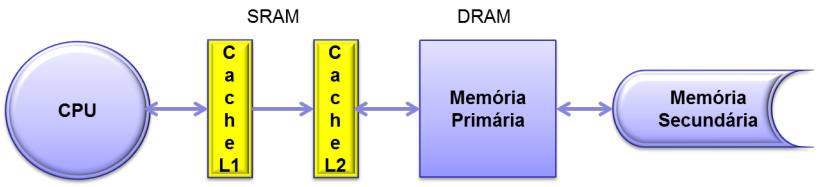


Fonte: STALLINGS (2012).

DRAMs, devido à sua grande quantidade e baixo custo. Porém, ao acessar diretamente uma memória DRAM um processador não obtém resposta imediata, ficando aguardando a resposta da memória. Portanto, foi necessário se utilizar uma técnica que buscasse reduzir/minimizar a distância entre as velocidades de acesso memória/processador utilizando uma pequena quantidade de memória de alta velocidade.

As memórias cache (feitas de memórias SRAM) foram introduzidas entre o processador e as memórias principais DRAMs na intenção de a CPU (Unidade Central de Processamento) acessar a memória cache em alta velocidade como se estivesse acessando a memória principal. O esquema de hierarquia de memória com cache funciona conforme a figura 2.2. O processador quando precisa buscar dados/instrução na memória principal, solicita à memória cache, que retorna rapidamente, caso possua o dado/instrução solicitado. Como a memória cache é bem menor que a memória principal, a memória cache deve ter disponível apenas os dados/instruções que serão usados recentemente pela CPU. Para obter um bom funcionamento desse esquema, a memória cache segue alguns princípios básicos, que são os seguintes:

Figura 2.2: Hierarquia de memórias.



Fonte: STALLINGS (2012).

- A cache é acessada pelo processador para todas as referências à memória;
- Se o conteúdo buscado pelo processador não estiver na cache, um bloco inteiro com conteúdos de endereços conjugados é buscado na memória e salvo na cache antes do conteúdo ser encaminhado ao processador pelo controlador de cache.

- Cada linha da cache inclui os campos de "dados"e de "tag". O campo de "dados"possui o conteúdo de todos os itens do bloco. O campo "tag"possui parte do endereço inicial do bloco referente à sua correspondente linha.

A memória cache funciona buscando explorar o princípio da localidade espacial e o princípio da localidade temporal. O princípio da localidade espacial é relacionado a natureza da execução sequencial de um programa, onde os endereços de instrução de um programa são sequenciais a não ser em casos de desvios. O princípio da localidade temporal, que diz que se uma referência a um dado ocorrer em um determinado momento, existe uma grande probabilidade da ocorrência de uma referência ao mesmo endereço em algum momento próximo.

2.1.1 Funcionamento das memórias Cache

A memória cache, de forma simplificada, funciona conforme ilustrado através da figura 2.3. O processador realiza um endereçamento à memória que é encaminhado à memória cache. A palavra de endereço é dividida nos campos de *Tag* e de *Index*. O campo de *Index* corresponde aos bits menos significativos (LSB) da palavra de endereço, enquanto que o campo de *Tag* corresponde aos bits mais significativos(MSB). Para verificar se a cache contém o dado referente ao endereço, é necessário primeiro acessar a linha da cache, usando os bits de *Index* da palavra de endereço. Depois, na linha selecionada, é verificado se o dado é válido (bit de válido na linha da cache) e se os bits de *Tag* do endereço corresponde exatamente aos bits de *Tag* da linha selecionada da cache. Quando ocorre essa correspondência, é dito que houve um *Hit* (sucesso) ao acessar a cache. Nesse caso, o dado é disponibilizado para o processador. Se não houver correspondência, é dito que houve um *Miss* (falta). Então o controlador da cache (não exibido na figura) realiza um acesso à memória principal para buscar o dado solicitado, atualizando a cache na correspondente linha (relocando o bloco da linha selecionada para atualização, salvando na linha o dado, e atualizando o campo de *Tag* com a Tag do endereço e setando o valor do Válido para 1).

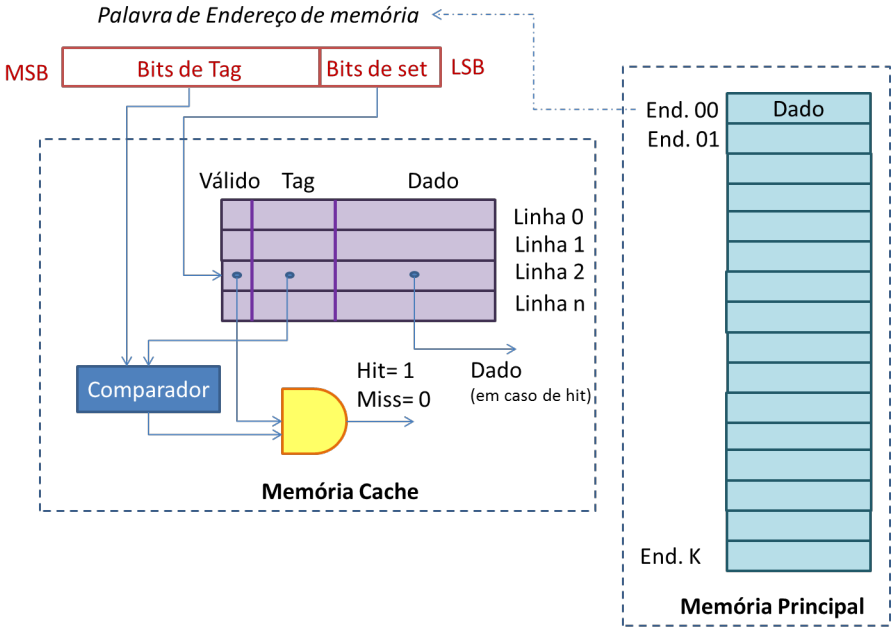
2.1.2 Parâmetros de Configuração de cache

Os principais parâmetros de configuração de uma memória cache são o tamanho total, o nível de associatividade (função de mapeamento), tamanho do bloco em uma linha de cache, e política de escrita.

2.1.2.1 Tamanho Total da Cache

O tamanho total da cache é um parâmetro de configuração de cache cujo valor final ideal não é necessariamente o maior possível. Quanto maior o tamanho total da cache maior será a taxa de acerto da cache (até um certo limite), o que pode elevar bastante o seu desempenho, o seu custo final e o seu consumo de energia. Quanto menor o tamanho total da cache menor

Figura 2.3: Funcionamento da memória cache.



Fonte: STALLINGS (2012).

poderá ser a sua taxa de acerto, seu desempenho e o seu custo final, porém o consumo de energia também poderá ser maior devido às penalidades. Existe um compromisso entre consumo de energia e desempenho na definição do valor do tamanho total da cache.

2.1.2.2 Mapeamento de Cache

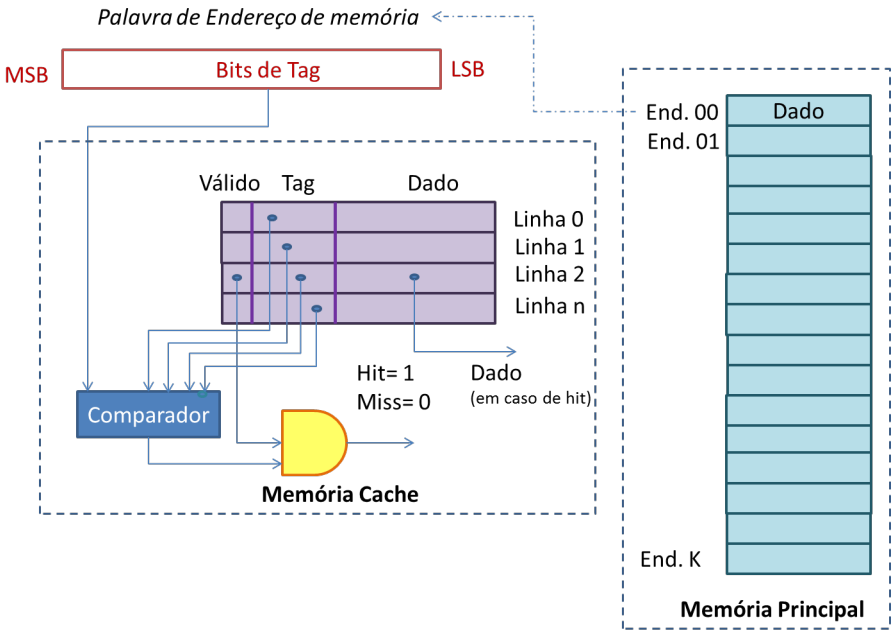
A função de mapeamento de uma cache define quais blocos da memória principal estarão vinculados a cada linha da cache, considerando que o número de blocos da memória é muito maior que o número de linhas da cache. Essa correspondência segue uma entre as seguintes possibilidades:

Mapeamento Direto - Cada bloco da memória principal é mapeado para uma única linha da cache. Cada linha da cache pode receber diferentes blocos da memória principal (comportando apenas um por vez). Dado que uma linha A de cache recebeu um bloco X da memória principal, se outro bloco Y da memória principal (mapeado para a mesma linha A de cache) for acessado, a linha A da cache receberá o valor do bloco Y da memória principal, sobrescrevendo o valor bloco X da memória principal previamente salvo na cache. A figura 2.3 ilustra uma cache com mapeamento direto.

Mapeamento Associativo - Cada bloco da memória principal é mapeado para qualquer linha da cache. Dado que uma linha A de cache recebeu um bloco X da memória principal, se outro bloco Y da memória principal for acessado, ele poderá ser salvo em uma outra linha de cache disponível (se houver linha de cache não usada), ou uma linha de cache poderá ser escolhida (de acordo com um algoritmo de relocação) para receber o bloco Y da memória

principal. A implementação de uma cache totalmente associativa é muito custosa, pois requer muito hardware paralelo nos circuitos de comparação. A figura 2.4 ilustra uma cache com mapeamento direto.

Figura 2.4: Mapeamento Associativo (totalmente Associativo).



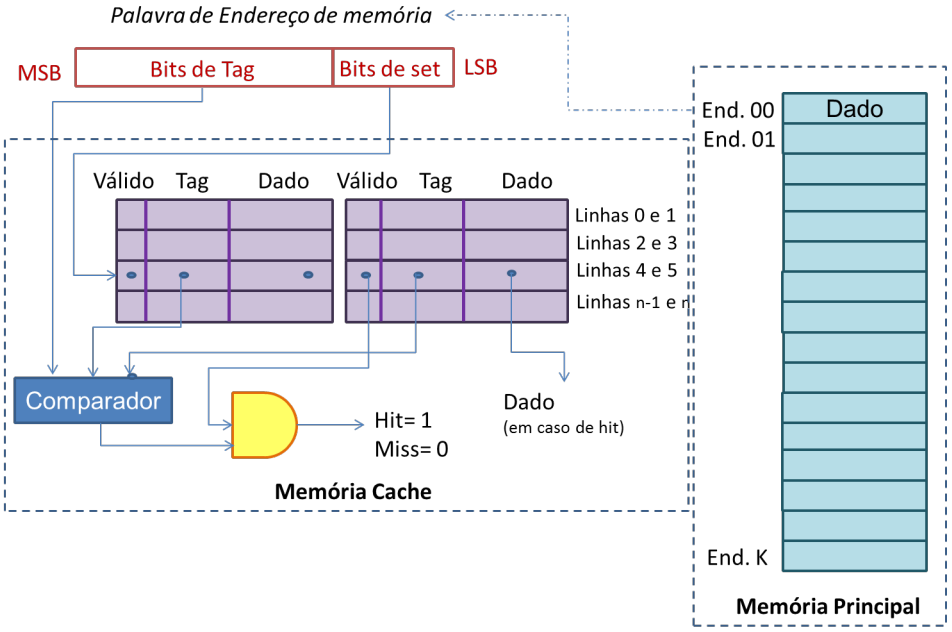
Fonte: STALLINGS (2012).

Mapeamento Associativo por Conjunto - No mapeamento associativo por conjunto cada linha da cache possui tipicamente um conjunto de alocações cujo valor é uma potência de 2. Dessa forma quando uma cache é 2-*way*, cada linha possui duas alocações possíveis, portanto pode comportar dois blocos de memória. Quando a cache é 4-*way*, cada linha possui quatro alocações possíveis. Cada linha da cache pode receber diferentes blocos da memória principal, comportando apenas n-*ways* blocos por vez. Considerando uma cache 2-*way*, dado que uma linha *A* da cache recebeu um bloco *X* da memória principal, se outro bloco *Y* da memória principal for acessado, ele poderá ser salvo na segunda alocação disponível (se a segunda alocação estiver vazia), ou uma das duas alocações poderá ser escolhida (de acordo com um algoritmo de relocação) para receber o bloco Y da memória principal. A figura 2.5 ilustra uma cache com mapeamento 2-*way*.

2.1.2.3 Tamanho do Bloco da Cache

Quanto maior for o tamanho do bloco, mais dados, provavelmente úteis, serão carregados na cache na ocorrência de uma falta (devido ao princípio da localidade espacial) favorecendo a uma maior taxa de acertos (taxa de hit). Porém, blocos grandes reduzem o número de blocos que cabem em uma cache, e um pequeno número de blocos resulta em dados sendo sobrescritos.

Figura 2.5: Mapeamento Associativo por conjunto (2-*way* associativo).



Fonte: STALLINGS (2012).

2.1.2.4 Política de Escrita

Em um sistema de memória de um computador é importante que um acesso à memória principal retorne um dado válido. Se esse dado foi atualizado somente na cache, e ocorrer um acesso à memória (por outro dispositivo) para esse mesmo bloco, esse dispositivo não poderá receber um conteúdo desatualizado. Essa atualização da memória principal, após uma escrita em cache é chamada de política de escrita. Diferentes políticas de escrita relacionam-se ao momento em que a memória principal será atualizada. Considerando o momento de atualização do dado em memória, existem dois tipos de política de escrita. Na política de escrita *Write Through* todas as operações de escrita realizadas em cache são também simultaneamente realizadas na memória principal. Dessa forma os dados da memória principal estarão sempre coerentes com os valores da cache. A desvantagem dessa política é que a memória principal sofrerá uma possível sobrecarga de acessos, e haverá penalidade devido a essa necessidade de escrita em memória. Em um acesso via barramento, essa sobrecarga poderá gerar um gargalo no acesso à memória. A segunda política de escrita é o *Write Back*. No *Write Back*, quando um bloco é atualizado em cache, um bit adicional chamado *Update*, associado ao bloco, é setado. Quando esse bloco precisar ser substituído ele será primeiro atualizado na memória para depois ser sobrescrito. O uso dessa política reduz a quantidade de acessos à memória quando não existe conflito, isso é nas escritas em cache em que ocorre *write hit* (escrita em bloco já previamente alocado em cache).

2.1.3 Consumo de Energia

A potência instantânea $P(t)$ provida a um elemento de circuito qualquer é o produto da corrente que atravessa o circuito pela tensão entre seus terminais.

$$P(t) = I(t) \times V(t) \tag{2.1}$$

A energia consumida (ou provida) em um intervalo de tempo T é a integral da potência instantânea.

$$E = \int_0^T P(t)dt \tag{2.2}$$

A energia média para o intervalo de 0 a T é dada por:

$$E_{avg} = \frac{E}{T} = \frac{1}{T}\int_0^T P(t)dt \tag{2.3}$$

A potência é dada em *Watts* (W). Energia em circuitos é expressa em *Joules* (J), onde $1W = 1J/s$.

A dissipação de energia em circuitos CMOS (*Complementary Metal Oxide Semiconductor*) se origina de dois componentes, a **dissipação dinâmica** e a **dissipação estática**. A dissipação dinâmica, ou energia dinâmica, ocorre devido ao consumo dinâmico de energia dos circuitos, que se relaciona com a dissipação pela carga e descarga de capacitâncias de cargas (como chaveamento de portas lógicas), e com correntes de "curto-circuito"enquanto os *stacks* pMOS e nMOS estão parcialmente em condução. A dissipação estática ocorre devido às correntes de fuga de limiar com os transistores polarizados inversamente, devido às fugas nas portas lógicas (através dos dielétricos das portas), devido às fugas de junção das disfunções fonte/dreno, e devido às correntes de contenção em circuitos de rádio frequência (vide equação 2.5).

$$E_{dinamica} = E_{chaveamento} + E_{curto\_circuito} \tag{2.4}$$

$$E_{estatica} = E_{fuga\_limiar} + E_{fuga\_portas} + E_{fuga\_juncao} + E_{fuga\_contencao} \tag{2.5}$$

$$E_{Total} = E_{dinamica} + E_{estatica} \tag{2.6}$$

Consumo de energia também pode ser classificado nos modos *ativo*, *standby* e *sleep*. A energia em modo ativo é consumida enquanto o circuito integrado está realizando trabalho útil. Normalmente relacionada com a energia de chaveamento $E_{chaveamento}$. Energia em modo *standby* é a energia consumida quando o circuito integrado está em modo *idle* (disponível para trabalho). Se os *clocks* foram interrompidos e circuitos de rádio frequência estão inativos a energia consumida é por fugas. No modo sleep, as fontes de alimentações em circuitos fora de uso estarão desligadas para evitar as correntes de fuga. Nesse modo o consumo de energia é reduzido fortemente.

A proporção do consumo de energia dinâmica e estática também se relaciona com a tecnologia de fabricação do circuito integrado em análise. Para circuitos de baixa frequência o

consumo de energia estática é muito baixo. Isso ocorre em circuitos de tecnologia, por exemplo, de 100 microns. Nos circuitos com tecnologia de 0,02 microns o percentual de consumo de energia estática se eleva bastante não podendo ser mais desprezada nos cálculos de consumo de energia.

Considerando que a cache normalmente ocupa uma área significativa de um circuito integrado de um processador, reduzir o tamanho da cache ou colocar uma parte dela em modo sleep, potencialmente proporciona uma excelente forma de redução de energia estática. Alternativamente, visto que as atividades de chaveamento durante os acessos à cache (nas leituras e escritas) e acessos à memória principal (trazendo dados para a cache) ditam o consumo de energia dinâmica, a redução da taxa de faltas da cache (*cache misses*) proporciona uma grande economia de energia dinâmica, pela redução da penalidade. Adicionalmente, ao reduzir a taxa de faltas da cache reduzimos também o número total de ciclos de execução de uma aplicação (melhorando o desempenho da aplicação) que também reduz o consumo de energia estática. Portanto, as técnicas que trabalham tanto com ajustes de parâmetros de cache quanto com a redução da taxa de faltas em cache são de grande importância para redução de energia estática e dinâmica em sistemas de cache.

2.1.4 Tuning de Cache

Cache tuning é o processo de determinação da melhor configuração de cache (quanto à sua organização e valores específicos de parâmetros de cache) no espaço de projeto (coleção de todas as possíveis configurações de cache) para uma aplicação em particular (*tuning* de cache baseado em aplicação), para um conjunto de aplicações, ou em fases de uma aplicação (*tuning* de cache baseado em fases), considerando um ou mais objetivos (como redução de consumo de energia, melhoria de desempenho, etc) (ZANG; GORDON-ROSS, 2013).

Os principais parâmetros de cache que determinam o consumo de energia e desempenho da cache são tamanho total da cache, tamanho do bloco e grau de associatividade (ZHANG; VAHID; LYSECKY, 2004). Se o tamanho da cache é muito grande, o seu consumo de energia poderá ser maior que o necessário. Se o tamanho da cache é muito pequeno, haverá um maior consumo de energia devido às faltas (*cache misses*) que ocorrerão, devido às operações de relocação (*replacement*) em memória ou em outros *cores* (núcleos de uma CPU). Outro importante parâmetro é o tamanho do bloco da cache. Devido ao princípio da localidade espacial, a energia pode ser desperdiçada tanto durante os ciclos de *stall* do processador utilizando um tamanho de bloco pequeno, quanto trazendo para cache informações que nunca serão usadas, no caso de usar um tamanho de bloco muito grande. Finalmente, o parâmetro de associatividade, que explora o princípio da localidade temporal, também tem grande influência no consumo de energia. Um nível de associatividade muito alto pode gerar desperdício de espaço por guardar dados que não serão mais usados. Usando um nível de associatividade muito baixo, poderá haver um aumento no número de faltas por conflito e penalidades por *raplacement*.

A literatura propõe a classificação dos mecanismos de *cache tuning* em dois grupos (ZANG; GORDON-ROSS, 2013): *Técnicas estáticas de tuning* e *Técnicas dinâmicas de tuning*. Os dois grupos são detalhados nas subseções seguintes.

2.1.4.1 Tuning Estático de Cache

O *tuning estático de cache* é realizado em tempo de projeto buscando explorar o espaço de projeto através de simulações da aplicação em um modelo de simulação de um computador. Nesse tipo de *tuning* de cache não existe *overhead* do tempo de execução porque os ajustes são realizados em tempo de projeto. O *Tuning Estático de Cache* é mais amplamente utilizado para sistemas embarcados de aplicações específicas porque neles são executadas uma ou mais aplicações específicas de forma contínua e única. Como sempre são executadas as mesmas aplicações, é possível se buscar em tempo de projeto a melhor configuração para a execução das aplicações. Existem diversas técnicas de *tuning estático de cache*, incluindo técnicas que simulam o comportamento da cache usando um simulador e técnicas que formulam taxas de falta de cache com modelos matemáticos (modelagem analítica).

Tuning de cache baseado em simulação usa software para modelar as operações de cache e estimar as faltas de cache (*cache misses*) ou outras métricas utilizando como entrada uma dada aplicação. O simulador permite que os parâmetros de cache sejam variados para diferentes configurações de cache do espaço de projeto simulando facilmente sem necessidade de protótipos de hardware físicos. Segue abaixo alguns tipos de *cache tuning* baseados em simulação (ZANG; GORDON-ROSS, 2013):

- Simulação baseada em *Traces* (*Trace-driven*);
- Aceleração de *Tuning de cache* usando exploração eficiente do espaço de projeto

Tuning de cache baseado em modelagem analítica calcula diretamente as faltas de cache para cada configuração usando modelos matemáticos. Uma vez que as taxas de falha de cache são calculadas quase instantaneamente usando fórmulas computacionais, o tempo de ajuste de cache é significativamente reduzido. Modelos analíticos requerem informações estatísticas detalhadas e/ou informações sobre eventos de aplicações críticas, que podem ser coletados usando *profilers*. Alguns tipos de *tuning* de cache baseados em modelagem analítica (ZANG; GORDON-ROSS, 2013), incluem Modelagem analítica baseada em estruturas de aplicação e Modelagem analítica baseada em traces de acesso à memória.

2.1.4.2 Tuning Dinâmico de Cache

O *Tuning dinâmico de cache* é realizado em tempo de execução da aplicação buscando utilizar dados da execução que conduzam à uma melhor configuração de cache. Neste caso é observado um *overhead* de tempo por uso de configurações de cache não ótimas (quanto aos objetivos) durante o processo de *tuning*, pois a seleção de configurações é realizada em tempo

de execução. Essas configurações não ótimas são utilizadas pelo algoritmo de tuning, para se chegar às melhores configurações de cache (configurações ótimas) quanto aos objetivos. O *Tuning Dinâmico de Cache* exige que a plataforma incorpore uma cache ajustável em tempo de execução. Para realizar dinamicamente o *tuning* da cache, é necessário que um gerenciador de *tuning* esteja incorporado à plataforma como um dispositivo adicional que interage com os controladores de cache. Chamamos de Ciclos de *Overhead de Tuning* (*tuning stall cycles*), o número de vezes que esse sistema demanda a cache selecionando configurações de cache não ótimas para a execução das aplicações correntes. Algumas técnicas de *tuning* de cache para processadores single-core podem ser utilizadas para *tuning* de cache em processadores multi-core, como desativação de Way/Set e concatenação de ways. Porém, no ajuste dos parâmetros de cache, a interdependência entre cores deve ser considerada, por exemplo, como consistência de dados compartilhados e contenção de recursos. Adicionalmente, várias organizações de cache em processadores multi-core incluem alguns desafios, que aumentam a complexidade da configuração da cache. Seguem abaixo alguns tipos de técnicas para *tuning* dinâmicos de cache (ZANG; GORDON-ROSS, 2013):

- Aplicáveis a Arquiteturas *Single-Core*;
  - Gerenciamento de *Way*;
  - Particionamento de bloco;
- Aplicáveis a Arquiteturas *Multi-Core*;
  - Particionamento de LLC (Cache de último nível);
    - Particionamento de *Way*;
    - Particionamento de *Set*;
  - Exploração de espaço de projeto;
- Detecção de mudança de fase.

Na próxima seção discutiremos mais detalhadamente sobre o problema de exploração de espaço de projeto de caches em plataformas multiprocessadores.

2.2 O problema de exploração de espaço de projeto com múltiplos processadores conectados por NoC

A exploração do espaço de projeto de memórias cache em sistema com múltiplos processadores é um problema bem mais complexo que em sistemas com um único processador. Destacamos as quatro principais características que aumentam a complexidade: o aumento

significativo do espaço de projeto, a coerência dos dados compartilhados em cache, o compartilhamento das caches entre os processadores, e o uso de rede NoC para comunicação entre os processadores.

2.2.1 Aumento do Espaço de Projeto de Cache

Considere uma plataforma com múltiplos núcleos em que cada núcleo processador possui sua hierarquia de cache. Se cada um dos N núcleos processadores possui um espaço de projeto de cache de X configurações, o espaço de projeto total da cache passa a ser X^N. Por exemplo, no caso de uma plataforma com 8 núcleos processadores com 10 diferentes configurações para a suas hierarquias de cache para cada núcleo processador, o espaço total de configurações de cache será $10^8 = 100.000.000$ configurações. A distância entre o espaço de projeto da cache local em cada núcleo processador (10 configurações) e o total do espaço de projeto de cache completo do MPSoC (100.000.000) é muito grande. Portanto, o aumento do espaço de projeto ao se considerar uma plataforma com múltiplos núcleos pode gerar um problema de escalabilidade dependendo da técnica a ser utilizada para *tuning* da hierarquia de cache.

2.2.2 Coerência de Cache

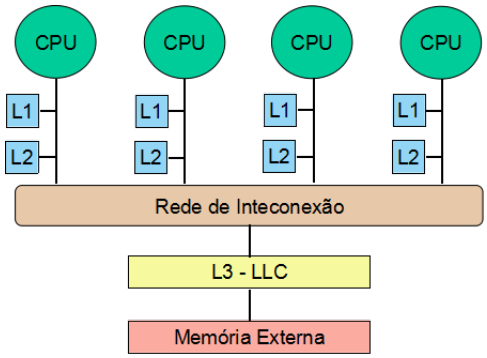
Em um sistema com diversos núcleos processadores, quando temos um dado compartilhado entre dois processadores (o dado estando no estado *shared* na cache dos dois processadores), se um processador atualiza esse dado em sua cache, esse mesmo dado na cache privada do segundo processador será invalidado para manter a coerência das caches. Nesse caso o ajuste da configuração de uma cache deve levar em conta os acessos do processador local e os acessos realizados por outros processadores com quem compartilham dados. Portanto, para buscar um conjunto de configurações de cache que melhor atenda a um ou mais objetivos em um sistema com múltiplos processadores, é imperativo que a coerência das caches seja considerada na técnica de *tuning*.

2.2.3 Caches de Último Nível compartilhadas

Outro problema muito comum em *tuning* de cache de sistemas com múltiplos processadores é a existência de caches compartilhadas entre os diversos cores. Na maioria dos casos essas caches são de comum acesso a todos os núcleos processadores e são chamadas de caches de último nível (LLC). Na figura 2.6, a cache de último nível é a cache L3.

Essas caches LLC são preenchidas com dados das aplicações executadas por todos os núcleos processadores. Quando existe um núcleo que demanda o uso de uma quantidade muito grande de dados, a cache LLC fica repleta de dados desse núcleo, pois na medida que o referido núcleo demanda dados, eles são trazidos para a LLC substituindo dados usados pelos demais núcleos. Fixar um tamanho máximo, por exemplo, igual para cada um dos cores pode não ser

Figura 2.6: L3 - Cache de último nível (LLC).



eficiente, portanto, existem técnicas de *tuning* de cache de último nível que particionam a cache LLC entre os núcleos proporcionalmente à demanda de dados de cada um deles. Tais técnicas são aplicadas apenas a sistemas com múltiplos processadores e geram como resultado um fator de particionamento de cache de último nível (em número de *ways* ou de *sets*) para cada um dos processadores.

2.2.4 Uso de Interconexão por NoC

Outro desafio em *tuning* de cache está relacionado à interconexão entre os processadores. Em um sistema multi-core, quando a quantidade de processadores é pequena o uso de uma interconexão baseada em barramento não representa maiores problemas no desempenho do sistema. Porém, na medida que o número de processadores aumenta, a serialização do acesso à memória e da implementação da coerência de cache (intrínseca ao uso de barramento) implica numa degradação do desempenho para todos os processadores. A solução encontrada para minimizar esse problema é o uso de interconexão em rede, chamadas de NoC.

As NoCs permitem que a comunicação dos processadores com a memória e entre eles não seja serializada. Essa comunicação passa a ser realizada através do envio de mensagens simultâneas de transação na rede NoC. A retirada da serialização da interconexão remove o gargalo de comunicação, porém insere alguns problemas adicionais, entre os quais destaca-se a imprevisibilidade temporal de execução das aplicações nos processadores. Com relação ao referido problema, verificamos que se um primeiro processador possui um dado em sua cache, que foi previamente buscado por um segundo processador, se o primeiro processador acessar esse dado em um momento posterior, ele deveria receber um *cache miss*, porém se a mensagem do segundo processador ainda não chegou, ele poderá acessar esse dado com sucesso (cache hit). Isso não ocorreria se houvesse serialização, pois o dado teria sido invalidado a tempo. Essa imprevisibilidade temporal de execução inviabiliza o uso de técnicas baseadas em trace (trace-driven) em MPSoCs devido à imprecisão temporal introduzida (BITAR, 1990) (GOLDSCHMIDT; HENNESSY, 1993) (ISSHIKI, 2010). Essa imprecisão temporal pode ser agravada com o uso de uma arquitetura MPSoC baseada em NoC.

Considerando a escolha de trabalharmos com exploração de espaço de projeto em

MPSoC, buscamos uma técnica que evitasse os problemas citados acima. Portanto, optamos por utilizar na exploração de espaço de projeto de cache uma técnica meta-heurística baseada em inteligência de enxame (algoritmo ABC), que será descrita na próxima seção.

2.3 Introdução a Metaheurística e Algoritmo ABC (Colônia Artificial de Abelhas)

Uma metaheurística é um conjunto de conceitos usados para definir métodos de heurística que podem ser aplicados a uma grande variedade de diferentes problemas. Em outras palavras, uma metaheurística pode ser vista como um framework algorítmico geral que pode ser aplicado a diferentes problemas de otimização com relativamente poucas modificações para adaptá-lo ao problema específico. Também pode ser vista formalmente como um processo de geração iterativo que guia uma heurística subordinada pela combinação inteligente de diferentes conceitos para explorar (*exploration* e *exploitation*) o espaço de busca, onde estratégias de aprendizado são usadas para estruturar informações buscando encontrar eficientemente soluções próximas dos ótimos resultados quanto aos objetivos de otimização (OSMAN; LAPORTE, 1996)(EL-GHAZALI TALBI FAROUK YALAOUI, 2016). O termo *Exploration* é o processo de visitar regiões ainda não exploradas no espaço de busca, enquanto que o termo *Explotation* é o processo de visitar essas regiões de um espaço de busca dentro de uma vizinhança de pontos previamente visitados (CREPINSEK; LIU; MERNIK, 2013).

Exploration is the process of visiting entirely new regions of a search space, whilst exploitation is the process of visiting those regions of a search space within the neighborhood of previously visited points.

Problemas complexos são difíceis de resolver utilizando técnicas de otimização clássicas. Técnicas de metaheurística trazem métodos computacionais que buscam iterativamente melhorar uma solução candidata tomando como base alguma medida de qualidade. As técnicas metaheurísticas não usam como base nenhuma premissa do problema e podem explorar espaços de projeto de diferentes tamanhos. Algoritmos de Metaheurística são capazes de resolver problemas complexos de otimização onde outros métodos falham ao tentarem eficiência e efetividade. Uma metaheurística inicia seu processo de busca obtendo uma solução inicial (ou um conjunto inicial de soluções) e iterativamente realiza uma busca para melhorá-la guiada por certos princípios.

A literatura já propôs diversos métodos como metaheurística, entre os quais destacamos:

- Colônia de Formigas (*Ant Colony Optimization*);
- Procedimento Aleatório Adaptativo Guloso (*Greedy Ramdomized Adptative Procedure* - GRASP);
- Algoritmos Genéticos (*Genetic Algorithms*).

- Enxame de Partículas (*Particle Swarm*).
- Colônia Artificial de Abelhas (*Artificial Bee Colony*).
- *Simulated Annealing*.
- Busca Tabu (*Taboo Search*).

Na sequência apresentaremos o algoritmo ABC (Colônia Artificial de Abelhas).

2.3.1 Algoritmo ABC (Colônia Artificial de Abelhas)

As técnicas de otimização baseadas em Inteligência de Enxame derivam do comportamento coletivo de grupo de organismos. Sua convivência em grupo permite que esses organismos resolvam problemas que são difíceis ou impossíveis de solução por um único indivíduo. Logo, Inteligência de Enxame pode ser visto como um mecanismo que indivíduos podem usar para superar suas limitações cognitivas. A inteligência de enxame pode ser definida como a habilidade para gerenciar sistemas complexos com uma coleção de indivíduos em interação através de uma comunicação mínima entre vizinhos locais, para produzir um comportamento global inteligente. Esses indivíduos normalmente não seguem comandos de um líder, ou algum planejamento maior. Por essas características, a inteligência de enxame tem sido utilizada em diversas finalidades em áreas diversas da engenharia. O termo Engenharia de Enxame também vêm sido utilizado em aplicações com diversos pequenos robôs de comandos e comunicação simples (KAZADI, 2011).

Algoritmos de otimização baseados em inteligência de enxame (SOA) imitam modelos da natureza para chegar a soluções próximas do ótimo. A principal diferença entre SOAs e outros algoritmos de busca direta, como *Hill Climbing* e *Random Walk*, é que os SOAs utilizam uma população de possíveis soluções para cada iteração ao invés de apenas uma solução como os outros algoritmos. Esta característica enquadra tais algoritmos como sendo baseados em população. Se um problema de otimização é de objetivo único, é esperado que a população convirja para encontrar apena uma solução ótima. Se um problema tem múltiplos objetivos, o SOA obtém um conjunto de soluções a partir de sua população final. Alguns algoritmos SOA são: Otimização por colônia de formigas(ACO), Otimização por enxame de partículas (PSO), Algoritmos genéticos (GA), Busca por difusão estocástica (SDS) e Colônia Artificial de Abelhas (ABC).

Em 2005, Karamboga apresentou o algoritmo *Artificial Bee Colony* (ABC)(KARABOGA, 2005)(KARABOGA; BASTURK, 2007b). O algoritmo ABC é uma meta-heurística baseada em população, enxame de abelhas, que modela o comportamento de abelhas forrageiras. O algoritmo foi inicialmente proposto para otimização de problemas de funções numéricas, mas ele têm sido amplamente utilizado em problemas multi-dimensionais complexos em áreas como análise de imagens, engenharia, aprendizado de máquinas, bio-informática e negócios. O algoritmo ABC usa a dança *waggle* (dança indicativa de localização e qualidade de suas fontes de alimento)

como mecanismo inteligente de comunicação entre as abelhas buscando obter informação sobre fontes de alimento de qualidade. Para desenvolver o modelo matemático do algoritmo, os autores utilizaram os seguintes elementos: Abelhas *Employed*, abelhas *Onlookers*, abelhas *Scout*, fontes de alimento e *limite*. Cada fonte de alimento possui D dimensões correspondendo às dimensões do problema. Para o problema de exploração de espaço de projeto de cache, as dimensões são os parâmetros de configuração de cache de cada um dos cores. As abelhas *employed* exploram algumas fontes de alimento e quando retornam à colmeia realizam a dança *waggle*, para as abelhas *onlookers*, exibindo a qualidade e a localização de suas fontes de alimento. Com base na informação da dança *waggle*, as abelhas *onlookers* escolhem as melhores fontes de alimento para explorar. O *limite* é uma variável usada para controlar se uma fonte de alimento deve ser abandonada depois de diversas buscas sem sucesso na exploração por melhores fontes de alimento em sua vizinhança. O pseudo-código do algoritmo ABC é mostrado no algoritmo 1.

Algoritmo 1 Otimizando uma hierarquia de cache de N-níveis

```
1: Fase de iniciação
2: repeat
3:     Fase das abelhas Employed
4:     Fase das abelhas Onlookers
5:     Fase das abelhas Scout
6:     Salva a melhor solução obtida
7: until Número máximo de ciclos
```

Seguindo o pseudo-código, o algoritmo inicializa seu processo criando randomicamente a população de abelhas para explorar o espaço de busca e as fontes de alimento com suas respectivas adequações (*fitness*). Na fase de abelhas *employed*, estas buscam por melhores fontes de alimento na vizinhança de suas respectivas fontes de alimento. Cada abelha *employed* produz uma fonte de alimento candidata v_{id} usando dados de sua atual fonte de alimento conforme a seguinte expressão:

$$v_{id} = X_{id} + \phi_{id}(X_{id} - X_{kd}), \tag{2.7}$$

onde i é a fonte de alimento corrente em $\{1,2,\cdots,SN\}$, k é uma fonte de alimento escolhida randomicamente em $\{1,2,\cdots,SN\} - \{i\}$, X é o valor do parâmetro e d é um parâmetro randomicamente escolhido em $\{1,2,\cdots,D\}$(SN é o número total de fontes de alimento e D é o número total de parâmetros de configuração). O ϕ_{ij} é um número aleatório entre (-1, 1). Logo, X_{id} é o valor do parâmetro d da fonte de alimento i, e X_{kd} é o valor do parâmetro d da fonte de alimento k. Se a adequação (*fitness*) da nova fonte de alimento for melhor que o da atual, a nova fonte de alimento substitui a atual.

Depois, as abelhas *onlookers* buscam por fontes de alimentos na vizinhança (como as abelha *employed* fazem) usando apenas as atuais fontes de alimento de melhor qualidade dependendo do valor de probabilidade associado com essas fontes de alimento. O valor de probabilidade é calculado utilizando a seguinte expressão:

$$p_i = \frac{fit_i}{\sum_{n=1}^{SN} fit_n}, \tag{2.8}$$

onde fit_i é o valor de adequação da fonte de alimento i. Depois, as abelhas *scout* verificam se existem fontes de alimento excedendo o *limite* para as substituir por novas fontes randômicas de alimento(X_i). Essa operação pode ser calculada da seguinte forma para cada parâmetro d de i:

$$X_i^d = X_{min}^d + rand[0,1](X_{max}^d - X_{min}^d), \tag{2.9}$$

onde i é a fonte de alimento corrente a ser substituída, $rand[0,1]$ é um índice escolhido randomicamente e X_{max}^d e X_{min}^d são respectivamente os limites máximo e mínimo do parâmetro d.

Finalmente, a melhor solução obtida é comparada com as novas fontes de alimento. A melhor entre todas é salva como melhor solução. Os passos 3-6 são repetidos até que o número máximo de ciclos seja atingido.

Com a exploração de espaço de projeto de cache, com base no algoritmo ABC, estamos tratando o problema de explorar um grande espaço de projeto buscando otimizar a busca pela melhor solução de forma inteligente, de forma a conseguir: evitar as soluções de mínimos locais (o algoritmo implicitamente evita convergir para soluções em mínimo local), considerar a coerência de cache por realizar diretamente a simulação sobre a arquitetura proposta, e evitar a imprecisão temporal por não usar técnica de simulação baseada em *trace* (*trace-driven*).

2.4 DoE - Projeto de Experimentos

Na área de projetos de experimentos (DoE) existem diversas técnicas para se trabalhar com melhorias de resultados de experimentos. Entre tantas, exemplificamos com as técnicas de Projetos Fatoriais, Projetos Fatoriais Fracionários, Modelos de Regressão, Métodos de Superfície de Resposta (RSM), Experimentos com Fatores Randômicos, entre outros. Neste trabalho utilizamos técnicas de Projetos Fatoriais Fracionários, devido ao usarmos uma fração do espaço de projeto. A técnica de Projetos Fatoriais Fracionários utiliza o conceito de população balanceada em seus experimentos. Portanto segue-se a definição do uso de população balanceada e Análise Estatística do Modelo de Efeitos, para Projetos Fatoriais Fracionários.

2.4.1 DoE - Uso de População Balanceada

O uso de uma população balanceada é fundamental para aplicação das técnicas de análise estatística do modelo dos efeitos. O balanceamento da população garante que os efeitos que cada fator (parâmetro de cache) exerce sobre os demais fatores seja equilibrado para o conjunto de configurações de cache pertencentes à população balanceada (JAIN, 1991). Uma população balanceada de um experimento também é chamada de ortogonal.

Para entender melhor o conceito de população balanceada, é necessário se entender alguns termos utilizados. Uma carga balanceada é formada por diversas linhas, que correspondem ao valor de cada fator (configuração) e do valor do correspondente objetivo (exemplo: Consumo de energia). Cada coluna corresponde aos valores de um determinado fator (parâmetro de cache) para cada uma das linhas. Como o projeto com o qual estamos trabalhando é 3^{k-p}, teremos para cada fator exatamente 3 possíveis níveis (valores), representados nesse esquema como (-1), (0) e (+1). O valor k representa o número de parâmetros utilizados no experimento, e p é o fator de redução do espaço de projeto. Para $p = 1$ é utilizado a metade do espaço de projeto total de cache como população do experimento. Para $p = 2$ será utilizado um quarto do espaço de projeto total de cache, e sim por diante. O experimento fatorial fracionário será utilizado, pois o algoritmo usado no trabalho proposto utiliza em sua fase de inicialização uma população inicial de configurações de cache simuladas (que é uma fração do espaço de projeto total), e utilizaremos uma população inicial balanceada com base na técnica de fatorial fracionário 3^{k-p}.

O balanceamento de uma população é conseguido quando são satisfeitas as seguintes condições:

- A soma de cada coluna é zero;
- A soma do produto de quaisquer duas colunas é zero;

A figura 2.7 mostra um exemplo de uma população balanceada que segue as condições supracitadas. O espaço de projeto total corresponde a $3^4 = 81$ diferentes configurações de cache. A população balanceada utilizada no exemplo contém $3^{k-p} = 3^{4-2} = 9$ diferentes configurações de cache (balanceadas entre si). É possível verificar que a soma dos valores de cada coluna corresponde a zero, bem como a soma do produto de todas as combinação de colunas (duas a duas) também correspondem a zero. Qualquer espaço de projeto total (no exemplo $3^4 = 81$) com todas as configurações de cache (no exemplo para 4 parâmetros) também corresponde a uma população balanceada, e atende as condições descritas acima.

Figura 2.7: Exemplo de população balanceada.

	P_0	P_1	P_2	P_3	ENERGIA
1	1	0	-1	-1	57
2	-1	1	0	-1	75
3	0	-1	1	-1	48
4	0	1	-1	0	68
5	1	-1	0	0	71
6	-1	0	1	0	31
7	-1	-1	-1	1	32
8	0	0	0	1	38
9	1	1	1	1	54
Soma	0	0	0	0	

P_0*P_1	P_0*P_2	P_0*P_3	P_1*P_2	P_1*P_3	P_2*P_3
0	-1	-1	0	0	1
-1	0	1	0	-1	0
0	0	0	-1	1	-1
0	0	0	-1	0	0
-1	0	0	0	0	0
0	-1	0	0	0	0
1	1	-1	1	-1	-1
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0

2.4.2 DoE - Análise Estatística do Modelo de Efeitos

Em DoE podemos avaliar o efeito/impacto que cada fator/parâmetro possui no resultado geral/final de um experimento (MONTGOMERY, 2013). Exemplificaremos como DoE realiza essa análise com um exemplo de um experimento com dois fatores.

Considere que existem a níveis de um parâmetro A e b níveis de um parâmetro B, e que eles serão analisados em um projeto fatorial, isto é, o projeto possui todas as $a \times b$ combinações dos fatores. Em geral existem n réplicas de cada experimento. Temos que $y_{i..}$ denota o total de todas as observações dos i´ésimos níveis do parâmetro A, e que $y_{.j.}$ denota o total de todas as observações dos j´ésimos níveis do parâmetro B, e que $y_{ij.}$ denota o total de todas as observações nas ij´ésimas células, e que $y_{...}$ denota o total maior de todas as observações. Definiremos $\overline{y}_{i..}$, $\overline{y}_{.j.}$, $\overline{y}_{ij.}$, e $\overline{y}_{...}$, como os valores médios dos respectivos itens listados anteriormente. Expressamos matematicamente,

$$y_{i..} = \sum_{j=1}^{b}\sum_{k=1}^{n} y_{ijk} \qquad \overline{y}_{i..} = \frac{y_{i..}}{bn} \qquad i = 1,2,3,...,a \tag{2.10}$$

$$y_{.j.} = \sum_{i=1}^{a}\sum_{k=1}^{n} y_{ijk} \qquad \overline{y}_{.j.} = \frac{y_{.j.}}{an} \qquad j = 1,2,3,...,b \tag{2.11}$$

$$y_{ij.} = \sum_{k=1}^{n} y_{ijk} \qquad \overline{y}_{ij.} = \frac{y_{ij.}}{n} \qquad i = 1,2,...,a \qquad j = 1,2,...,b \tag{2.12}$$

$$y_{...} = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n} y_{ijk} \qquad \overline{y}_{...} = \frac{y_{...}}{abn} \tag{2.13}$$

A soma total dos quadrados pode ser escrita como

$$\sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(y_{ijk}-\overline{y}_{...})^2 = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}[(\overline{y}_{i..}-\overline{y}_{...})+(\overline{y}_{.j.}-\overline{y}_{...})+(\overline{y}_{ij.}-\overline{y}_{i..}-\overline{y}_{.j.}+\overline{y}_{...})+(\overline{y}_{ijk}-\overline{y}_{ij.})]^2 \tag{2.14}$$

$$\sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(y_{ijk}-\overline{y}_{...})^2 = bn\sum_{i=1}^{a}(\overline{y}_{i..}-\overline{y}_{...})^2 + an\sum_{j=1}^{b}(\overline{y}_{.j.}-\overline{y}_{...})^2 + n\sum_{i=1}^{a}\sum_{j=1}^{b}(\overline{y}_{ij.}-\overline{y}_{i..}-\overline{y}_{.j.}+\overline{y}_{...})^2 + \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(\overline{y}_{ijk}-\overline{y}_{ij.})^2 \tag{2.15}$$

porque os seis produtos cruzados do lado direito são zero (na equação 2.14).

Podemos verificar que a soma total dos quadrados (na equação 2.15) foi particionada na soma dos quadrados devido ao fator A (SS_A), na soma dos quadrados devido ao fator B (SS_B), na soma dos quadrados devido à iteração entre A e B (SS_{AB}), e na soma dos quadrados devido ao erro (SS_E). Esta é a equação fundamental ANOVA (*ANalysis Of VAriance*) para fatorial de dois fatores.

Podemos reescrever a equação 2.15 como

$$SS_T = SS_A + SS_B + SS_{AB} + SS_E. \tag{2.16}$$

Após simplificação, podemos novamente reescrever os elementos das somas dos quadrados em termos de linha, coluna e células totais da seguinte forma:

$$SS_T = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n} y_{ijk}^2 - \frac{y_{...}^2}{abn} \tag{2.17}$$

$$SS_A = \frac{1}{bn}\sum_{i=1}^{a} y_{i..}^2 - \frac{y_{...}^2}{abn} \tag{2.18}$$

$$SS_B = \frac{1}{an}\sum_{j=1}^{b} y_{.j.}^2 - \frac{y_{...}^2}{abn} \tag{2.19}$$

$$SS_{AB} = \frac{1}{n}\sum_{i=1}^{a}\sum_{j=1}^{b} y_{ij.}^2 - \frac{y_{...}^2}{abn} - SS_A - SS_B \tag{2.20}$$

$$SS_E = SS_T - SS_A - SS_B - SS_{AB} \tag{2.21}$$

Como verificamos em um experimento com dois fatores, os valores de SS_A e SS_B representam quantitativamente a importância/impacto dos fatores A e B para o resultado final de um experimento de dois fatores. $y_{i..}^2$ denota o quadrado de todas as observações de um nível i do fator A. Quando calculamos $y_{i..}^2$ para cada um dos níveis, obtemos quantitativamente a importância/impacto do respectivo nível dentro do fator A. Da mesma forma conseguimos obter a importância/impacto de um nível para o fator B usando $y_{.j.}^2$.

Quando trabalhamos com uma quantidade de 3 fatores, extrapolamos os valores das equações de SS_A e SS_B para encontrar SS_C, obtendo os seguintes resultados:

$$SS_A = \frac{1}{bcn}\sum_{i=1}^{a} y_{i...}^2 - \frac{y_{....}^2}{abcn} \tag{2.22}$$

$$SS_B = \frac{1}{acn}\sum_{j=1}^{b} y_{.j..}^2 - \frac{y_{....}^2}{abcn} \tag{2.23}$$

$$SS_C = \frac{1}{abn}\sum_{k=1}^{c} y_{..k.}^2 - \frac{y_{....}^2}{abcn} \tag{2.24}$$

No caso de 3 fatores, temos outras somas dos quadrados dos efeitos combinados. Nesse caso os combinados são SS_{AB}, SS_{BC}, SS_{CA}, e SS_{ABC}.

As equações fundamentais da ANOVA (*ANalysis Of VAriance*) de n-fatores podem ser utilizadas tanto para experimentos 2^k (projeto fatorial com dois níveis) como para outros experimentos como 2^{k-r} (projeto fatorial fracionário com dois níveis), 3^k (projeto fatorial com

três níveis), e para 3^{k-r} (projeto fatorial fracionário com três níveis)(MONTGOMERY, 2013). Neste trabalho aplicamos as equações fundamentais da ANOVA para experimentos de projeto fatorial fracionário com 3 níveis e K parâmetros (3^{k-r}), sem réplicas para os experimentos $(n = 1)$, logo as equações de SSA, SSB, e SSC ficarão da seguinte forma:

$$SS_A = \frac{1}{bc}\sum_{i=1}^{a} y_{i..}^2 - \frac{y_{...}^2}{abc} \tag{2.25}$$

$$SS_B = \frac{1}{ac}\sum_{j=1}^{b} y_{.j.}^2 - \frac{y_{...}^2}{abc} \tag{2.26}$$

$$SS_C = \frac{1}{ab}\sum_{k=1}^{c} y_{..k}^2 - \frac{y_{...}^2}{abc} \tag{2.27}$$

Analisando um exemplo prático, consideremos um sistema que é alimentado com os parâmetros de entrada A, B e C (Por exemplo, tamanho total da cache, nível de associatividade, e tamanho do bloco da cache), e que gera como saída a métrica Desempenho (Ciclos de CPU) como resultado. Para cada parâmetro existem os possíveis níveis/valores (0,1 e 2). Temos então a tabela da figura 2.8 como os experimentos realizados para todos os possíveis valores das entradas A, B e C, gerando como resultado os valores de Desempenho do núcleo processador, em número de ciclos de CPU.

Figura 2.8: Exemplo de DoE com parâmetros A, B e C.

A	B	C	Desempenho (Ciclos de CPU)
0	0	0	10
0	0	1	20
0	0	2	30
0	1	0	40
0	1	1	50
0	1	2	60
0	2	0	70
0	2	1	80
0	2	2	90
1	0	0	100
1	0	1	110
1	0	2	120
1	1	0	130
1	1	1	140
1	1	2	150
1	2	0	160
1	2	1	170
1	2	2	180
2	0	0	190
2	0	1	200
2	0	2	210
2	1	0	220
2	1	1	230
2	1	2	240
2	2	0	250
2	2	1	260
2	2	2	270

Os valores de SS_A, SS_B e SS_C são calculados utilizando as equações 2.25, 2.26 e 2.27, respectivamente. Os valores de a, b e c são iguais a 3, pois os três parâmetros possuem 3 níveis

(0,1 e 2). Os valores calculados para $y_{i..}$, $y_{.j.}$, $y_{..k}$ e $y_{...}$ e seus valores quadrados, são detalhados através da tabela da figura 2.9.

Logo, teremos o cálculo de SS_A, SS_B e SS_C da seguinte forma:

$$SS_A = \frac{1}{3\times 3}\sum_{i=1}^{3} y_{i..}^2 - \frac{y_{...}^2}{3\times 3\times 3} = \frac{1}{9}\times(6075000) - \frac{14288400}{27} = 145800,$$

$$SS_B = \frac{1}{3\times 3}\sum_{j=1}^{3} y_{.j.}^2 - \frac{y_{...}^2}{3\times 3\times 3} = \frac{1}{9}\times(4908600) - \frac{14288400}{27} = 16200,$$

$$SS_A = \frac{1}{3\times 3}\sum_{k=1}^{3} y_{..k}^2 - \frac{y_{...}^2}{3\times 3\times 3} = \frac{1}{9}\times(4779000) - \frac{14288400}{27} = 1800,$$

O parâmetro *A* é o de maior impacto para o resultado final do Desempenho. Podemos confirmar isso ao verificar que os seus níveis 0 e 2 indicam, respectivamente, os menores e maiores valores na coluna de Desempenho (ciclos de CPU). Isto é, variando o nível parâmetro *A* para cima ou para baixo verifica-se que o valor de Desempenho também acompanha variação semelhante. O parâmetro *C* é o de menor impacto. Podemos confirmar isso verificando que para qualquer um de seus níveis (0, 1 ou 2), encontramos na coluna de Desempenho (ciclos de CPU) tanto valores altos quanto baixos. Isto é, variando o nível parâmetro *C* para cima ou para baixo verificamos que o valor de Desempenho não acompanha de forma semelhante. O parâmetro *B* possui um impacto intermediário entre os parâmetros *A* e *C*.

2.5 Teste de Kruskal-Wallis

O teste de *Kruskal-Wallis* é extremamente útil para decidir se k amostras ($k > 2$) independentes provêm de populações com médias iguais. Esse teste de análise de variância foi desenvolvido por Kruskal e Wallis em 1952 (KRUSKAL; WALLIS, 1952)(MONTGOMERY, 2013). Utilizaremos esse teste para verificar se os possíveis valores a serem usados para configurar os parâmetros do algoritmo proposto neste trabalho, possuem baixas variações, para ajudar na decisão de que valor utilizar nas simulações. O teste foi criado para verificar a hipótese nula de que diferentes valores de um parâmetro geram respostas idênticas contra a hipótese alternativa de que eles geram respostas diferentes entre si. O teste de *Kruskal-Wallis* é uma alternativa não paramétrica à análise de variância usual.

Para realizar o teste de *Kruskal-Wallis* é necessário seguir os seguintes passos:

- Dispor, em ordem crescente, as observações de todos os k grupos, atribuindo-lhes postos de 1 a n. Caso haja empates, atribuir o posto médio;
- Determinar o valor da soma dos postos para cada um dos k grupos: R_i, onde $i = 1, 2, ..., k$;

Figura 2.9: Exemplo de DoE com parâmetros A, B e C incluindo cálculo de $y_{i..}$, $y_{.j.}$, $y_{..k}$ e $y_{...}$ e seus valores quadrados.

A	Desempenho para A= 0	Desempenho para A= 1	Desempenho para A= 2	B	Desempenho para B= 0	Desempenho para B= 1	Desempenho para B= 2	C	Desempenho para C= 0	Desempenho para C= 1	Desempenho para C= 2	Desempenho (Ciclos de CPU)
0	10			0	10			0	10			10
0	20			0	20			1		20		20
0	30			0	30			2			30	30
0	40			1		40		0	40			40
0	50			1		50		1		50		50
0	60			1		60		2			60	60
0	70			2			70	0	70			70
0	80			2			80	1		80		80
0	90			2			90	2			90	90
1		100		0	100			0	100			100
1		110		0	110			1		110		110
1		120		0	120			2			120	120
1		130		1		130		0	130			130
1		140		1		140		1		140		140
1		150		1		150		2			150	150
1		160		2			160	0	160			160
1		170		2			170	1		170		170
1		180		2			180	2			180	180
2			190	0	190			0	190			190
2			200	0	200			1		200		200
2			210	0	210			2			210	210
2			220	1		220		0	220			220
2			230	1		230		1		230		230
2			240	1		240		2			240	240
2			250	2			250	0	250			250
2			260	2			260	1		260		260
2			270	2			270	2			270	270
	450	1260	2070		990	1260	1530		1170	1260	1350	3780
	$Y_{0..}$	$Y_{1..}$	$Y_{2..}$		$Y_{.0.}$	$Y_{.1.}$	$Y_{.2.}$		$Y_{..0}$	$Y_{..1}$	$Y_{..2}$	$Y...$
	202500	1587600	4284900		980100	1587600	2340900		1368900	1587600	1822500	14288400
	$(Y_{0..})^2$	$(Y_{1..})^2$	$(Y_{2..})^2$		$(Y_{.0.})^2$	$(Y_{.1.})^2$	$(Y_{.2.})^2$		$(Y_{..0})^2$	$(Y_{..1})^2$	$(Y_{..2})^2$	$(Y...)^2$
	6075000 $\sum(Y_{i..})^2$				4908600 $\sum(Y_{.j.})^2$				4779000 $\sum(Y_{..k})^2$			

- Escolher uma variável Qui-quadrada com $v = k - 1$ (cada amostra deve conter pelo menos 5 observações);

- Realizar o teste utilizando a fórmula:

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{K} \frac{{R_i}^2}{N_i} - 3(N+1) \tag{2.28}$$

Os dados da tabela 2.1 são resultados do tempo de execução (em segundos) de uma aplicação genérica em um servidor com tamanhos de cache de 1GB, 2 GB e 4GB, respectivamente.

Tabela 2.1: Exemplo Kruskal-Wallis

1GB	**2GB**	**4GB**
12,5	12,3	10,3
13,4	11,7	14,2
11,2	12,4	15,0
13,7	15,3	13,6
15,0	18,5	12,5
12,8	13,4	15,1
15,9	17,2	12,9
12,4	11,5	12,4
14,2	13,9	13,7
11,9	13,3	12,2

- Estabelecemos as hipóteses:

$$\begin{cases} H_0 : \tau_1 = \tau_2 = ... = \tau_k \\ H_1 : \tau_1, \tau_2, ..., \tau_n \text{ são todos iguais} \end{cases}$$

- A partir dos dados, temos a tabela 2.2, relacionando os postos de cada elemento, os tamanhos amostrais de cada grupo e os valores Ri para cada grupo.

Tabela 2.2: Exemplo Kruskal-Wallis com os postos de cada elemento.

j	**1GB**	**2GB**	**4GB**
1	11,5	7	1
2	16,5	4	22,5
3	2	9	24,5
4	19,5	27	18
5	24,5	30	11,5
6	13	16,5	26
7	28	29	14
8	9	3	9
9	22,5	21	19,5
10	5	15	6
R_i	151,5	161,5	152
N	30	30	30
n_i	10	10	10

- Cálculo da estatística H.

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{K} \frac{{R_i}^2}{N_i} - 3(N+1),$$

$$H = \frac{12}{30(30+1)} \sum_{i=1}^{3} \frac{{R_i}^2}{N_i} - 3(30+1),$$

$$H = \frac{12}{930}[\frac{(151,5)^2}{10} + \frac{(161,5)^2}{10} + \frac{(152)^2}{10}] - 3(30+1),$$

$$H = \frac{12}{930}[2295,225 + 2608,225 + 2310,4] - 93,$$

$$H = 93,081 - 93, \Rightarrow H = 0,081$$

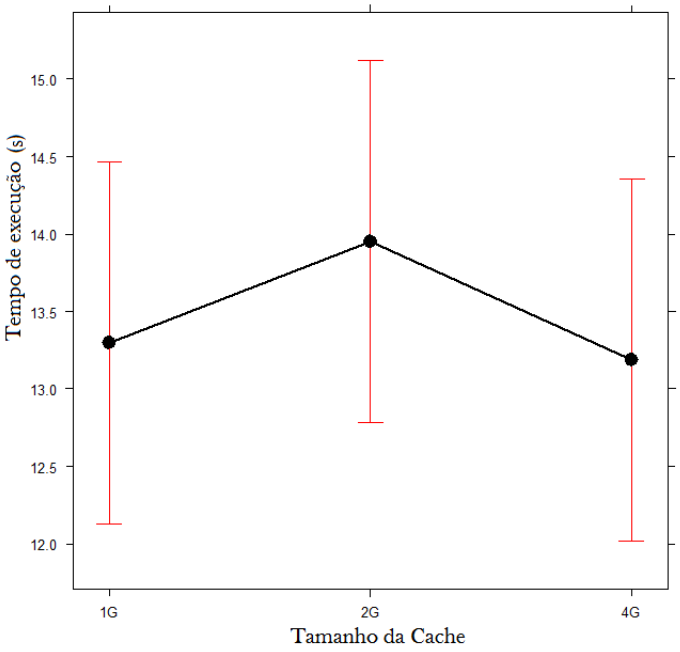
- Cálculo dos valores críticos.

Fixando o nível de significância $\alpha = 0,05$ e sabendo que $k = 3$, temos que o valor crítico corresponde ao ponto $Q_{0,95} = 7,815$ (consulta em tabela de distribuição do qui-quadrado - *Percentage Points of the* χ^2 *Distribution* - (MONTGOMERY, 2013)).

- Critério de rejeição.

Como $H_{obs} = 0,081 < Q_{0,95} = 7,815$, aceitamos a hipótese nula. Neste caso, podemos concluir que não é significativa a diferença de uso de qualquer dos três valores de cache (1Gb, 2Gb ou 4Gb) para uso nos servidores para a aplicação utilizada no teste. A figura 2.10 mostra os valores médios, máximos e mínimos dos tempos de execução utilizando cada um dos tamanhos de cache.

Figura 2.10: Valores médios, máximos e mínimos de tempo de execução para os tamanhos de cache de 1Gb, 2Gb e 4Gb.



2.6 Indicador de Hipervolume

O Hipervolume é um dos indicadores mais populares usados para algoritmos otimizadores multi-objetivo. Também conhecido como Métrica-S ou Medida Lebesgue, o hipervolume é o espaço n-dimensional que está contido em um conjunto solução, isto é o volume n-dimensional de um conjunto de soluções para algum ponto de referência (KNOWLES; CORNE, 2002). Devido às propriedades do indicador, um conjunto com um hipervolume maior apresenta um melhor compromisso entre os objetivos que um conjunto com um hipervolume menor. O indicador de hipervolume sintetiza em um valor único a contribuição de cada elemento de um conjunto de Pareto, bem como indica uma distância para o conjunto do Pareto-ótimo.

A figura 2.11 mostra um exemplo, em que destacamos o Pareto Ótimo de um experimento, bem como os conjuntos de Pareto (conjuntos 1 e 2) obtidos através de um algoritmo que busca encontrar configurações de cache próximas (o mais que possível) do referido Pareto Ótimo. O conjunto Pareto 1 é a solução mais próxima ao Pareto Ótimo comparado ao conjunto Pareto 2.

A figura 2.12 mostra a área delimitada entre os elementos do conjunto 1 e o ponto de referência. Também é exibida a área delimitada entre os elementos do conjunto 2 e o mesmo ponto de referência. A área de cada um corresponde ao hipervolume de cada um dos conjuntos. Neste caso a unidade do hipervolume será em *Ciclos x nJ*. Calculando, o hipervolume do conjunto 1 possui o valor de 53200 *Ciclos x nJ* e o conjunto 2 possui o valor de 50160 *Ciclos x nJ*, logo a solução do conjunto 1 é melhor que a solução do conjunto 2.

A figura 2.13 mostra a diferença de hipervolume do conjunto 1, do conjunto 2 e do Pareto

Figura 2.11: Resultados Conjunto Pareto 1, Conjunto Pareto 2 e o Pareto Ótimo.

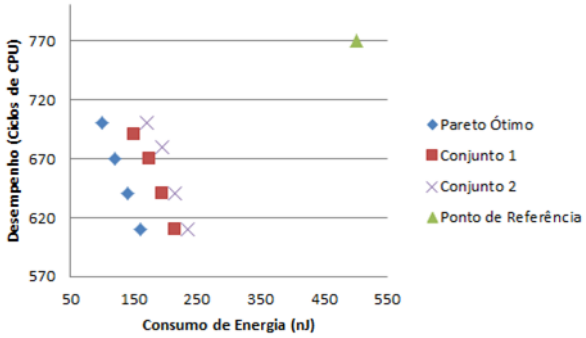
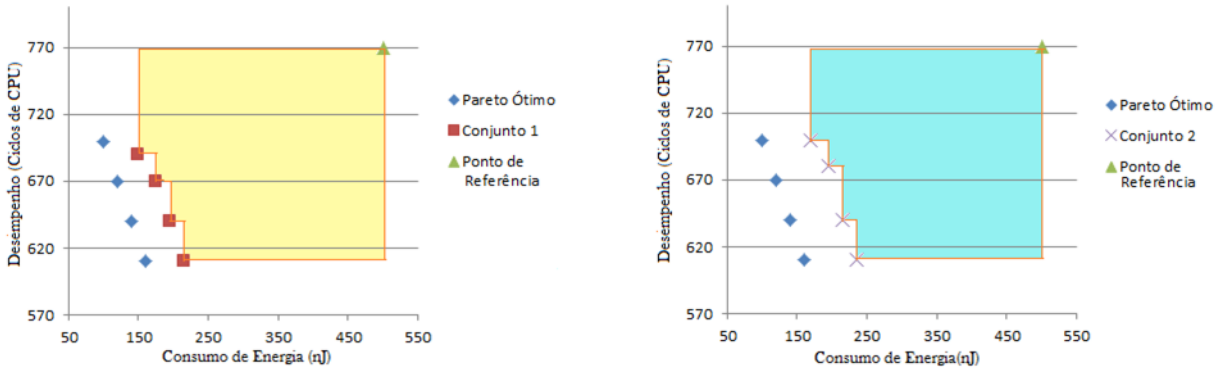
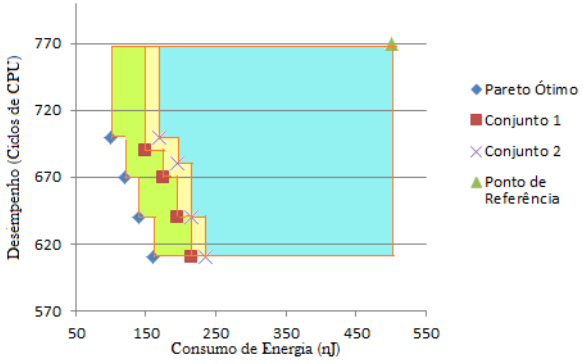


Figura 2.12: Hipervolume do Conjunto Pareto 1 e do Conjunto Pareto 2.



òtimo, que o hipervolume máximo para

Figura 2.13: Diferença do Hipervolume do Conjunto Pareto 1, do Conjunto Pareto 2 e do Pareto Ótimo.



2.7 Determinação de Tamanho de Amostra

Uma das questões mais importantes numa análise estatística é determinar qual o melhor tamanho de amostras que devemos ter, pois amostras muito grandes são desnecessárias e demandam mais tempo de manipulação e estudo, e amostras pequenas são menos precisas e

pouco confiáveis. Quando utilizamos dados amostrais para estimar uma média populacional μ a margem de erro (E) é a diferença máxima provável (com probabilidade $1-\alpha$) entre a média amostral observada $\bar{x}$ e a verdadeira média da população (μ), conforme a equação 2.29, conhecidos os valores do intervalo de confiança ($Z_{\alpha/2}$), do desvio padrão σ e do tamanho da população n (MONTGOMERY, 2014)(RYAN, 2013).

$$E = Z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}} \tag{2.29}$$

Podemos usar essa equação para determinar o tamanho de mínimo de uma amostra populacional, dados os valores do intervalo de confiança ($Z_{\alpha/2}$), o desvio padrão σ e a margem de erro E. Logo o tamanho de uma amostra pode ser calculado usando a equação 2.30.

$$n = \left(\frac{Z_{\alpha/2} \times \sigma}{E} \right)^2 \tag{2.30}$$

Os valores de intervalo de confiança e seus respectivos valores de $Z_{\alpha/2}$ são encontrados na tabela 2.3.

Tabela 2.3: Valores de Z para principais intervalos de confiança

Intervalo de Confiança	**Valor Crítico de Z**
90%	1,645
95%	1,967
99%	2,576

Exemplificando, para uma amostra populacional de 10 elementos, conforme a tabela 2.4, verificamos que o valor da média ($\bar{x}$) será 237,9 e o valor do desvio padrão será $\sigma = 20,776$. Utilizando a equação 2.30, para o intervalo de confiança de 90% e um erro de 10%, teremos o seguinte valor de tamanho de uma amostra:

Tabela 2.4: Amostra Populacional exemplo.

	População
	235
	243
	201
	210
	242
	221
	280
	245
	260
	242
$\bar{x}$	237,9
σ	20,776

$$n = \left(\frac{Z_{\alpha/2} \times \sigma}{E}\right)^2 = \left(\frac{1,645 \times 20,776}{237,9 \times 0,1}\right)^2 = 2,06 => 3 \; amostras$$

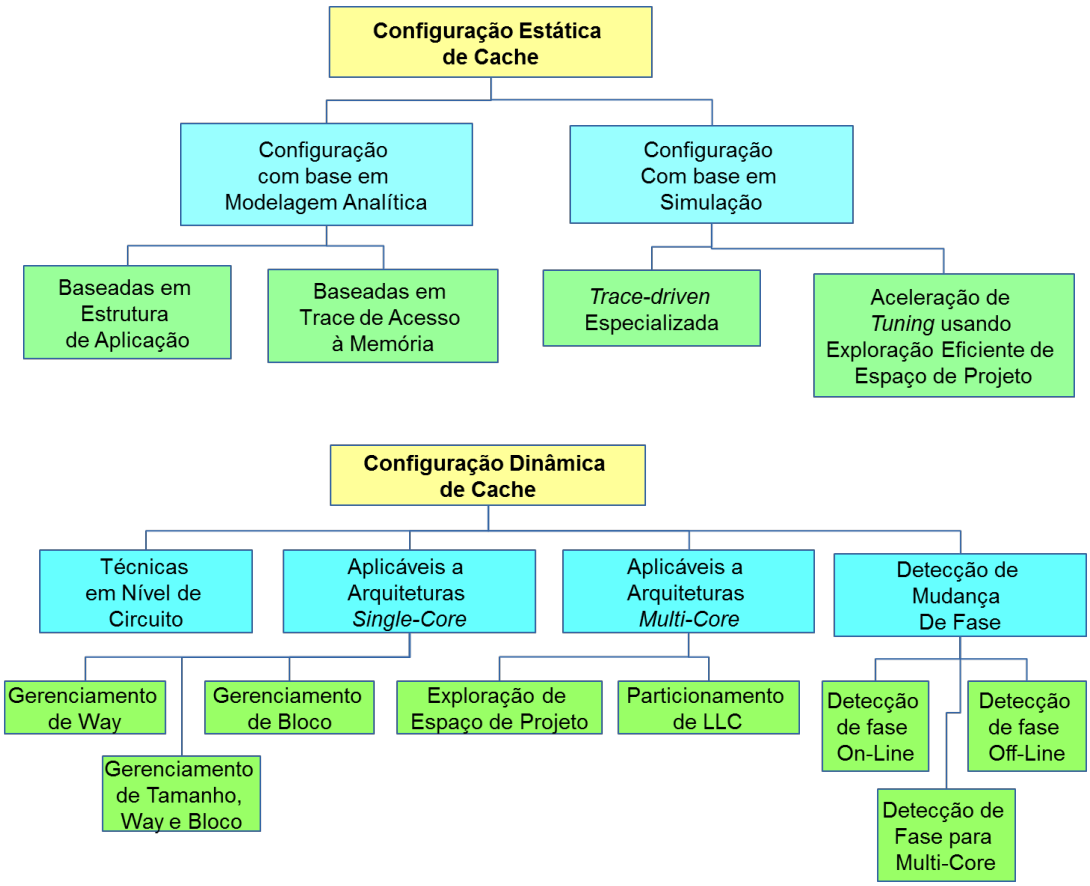
Logo, para o exemplo acima, bastariam três amostras para que estatisticamente a quantidade de amostras fosse representativa e confiável.

3

Trabalhos Relacionados

Considerando a tarefa de busca de uma ou mais configurações de cache, com os objetivos de redução de consumo de energia e/ou melhoria de desempenho, diversos trabalhos foram propostos utilizando diferentes arquiteturas e técnicas de exploração. Visto que a amplitude de trabalhos é bem extensa, buscamos descrever alguns trabalhos relevantes e atuais apresentando-os conforme classificação proposta por Zang e Gordon-Ross (ZANG; GORDON-ROSS, 2013). A figura 3.1 ilustra essa classificação.

Figura 3.1: Classificação das Técnicas de Cache Tuning (ZANG; GORDON-ROSS, 2013).



O primeiro grupo, *Configuração Estática de Cache*, inclui as técnicas que realizam

reconfiguração de cache em tempo de projeto. Neste caso, uma aplicação ou um sistema é avaliado obtendo informações para reconfigurar a cache com um ou mais objetivos. No segundo grupo, *Configuração Dinâmica de Cache*, verificamos técnicas que incluem um reconfigurador de cache como um sub-sistema dentro da arquitetura que monitora e reconfigura a hierarquia de cache em tempo de execução, conforme o comportamento de aplicativos (ou partes dele) em execução, conforme objetivos pretendidos, tais como redução de consumo de energia, tempo de execução, área, etc.

As técnicas que mais se aproximam da abordagem proposta neste trabalho são as técnicas do grupo de *Configuração Estática de Cache*, pois são de configuração de cache em tempo de projeto. Não apresentaremos técnicas para o grupo *Configuração Dinâmica de Cache*.

As seções e subseções seguintes buscam detalhar algumas técnicas de reconfiguração de cache importantes e atuais utilizando a classificação supracitada.

Abaixo das abordagens em *Configuração Estática de Cache*, as técnicas são divididas entre as de *Configuração com Base em Simulação* e as de *Configuração com Base em Modelagem Analítica*. As técnicas de *Configuração com Base em Simulação* utiliza software para modelar as operações da cache e estima taxas de falta de cache e outras métricas simulando a execução de uma aplicação. Já as técnicas de *Configuração com Base em Modelagem Analítica* calculam diretamente as taxas de falta para cada configuração de cache usando modelos matemáticos. Como as taxas de falta de cache são calculadas em alta velocidade de processamento, o tempo de execução e precisão dos resultados dessas técnicas são bastante reduzidos.

Não trouxemos do estado da arte nenhuma técnica de *Configuração com Base em Modelagem Analítica* devido a serem técnicas bem distantes do trabalho proposto, e por isso direcionamos nosso foco para as técnicas de *Configuração com Base em Simulação*, que são apresentadas na próxima seção.

3.1 Configuração com Base em Simulação

Conforme a classificação (ZANG; GORDON-ROSS, 2013), abaixo das abordagens em *Configuração com Base em Simulação*, as técnicas foram divididas entre as de *Trace-Driven Especializada* e as de *Aceleração de Tuning usando Exploração Eficiente de Espaço de Projeto*. Nas técnicas de *Trace-Driven Especializada* o comportamento da cache é simulado usando uma sequência, ordenada no tempo, de acessos à memória conhecida como *Access Trace*. O *Access Trace* é coletado por um simulador, e contém os acessos à endereços de memória realizados por um aplicativo, e é utilizado para simular o módulo de cache sob a execução do referido aplicativo. Nas técnicas de *Aceleração de Tuning usando Exploração Eficiente de Espaço de Projeto*, o espaço de projeto total não é simulado. Essas técnicas utilizam um subconjunto do espaço de projeto total buscando reduzir o tempo total de reconfiguração da cache. Embora consumam, ainda assim, um bom tempo de execução para grande espaços de projeto de cache, essas técnicas geralmente produzem resultados bem próximos do ótimo.

Os trabalhos de Zang e Gordon-Ross (ZANG; GORDON-ROSS, 2013) [aplicado para arquitetura SoC], o de Nawinne (NAWINNE et al., 2015) e o de Haque (HAQUE et al., 2012) [aplicados para arquitetura MPSoC] são descritos nas subseções 3.1.1, 3.1.2 e 3.1.3 como técnicas *Trace-Driven Especializada*. O trabalho de Santos e Silva-Filho (SANTOS; SILVA-FILHO, 2014) é descrito na subseção 3.1.4 como uma técnica de *Aceleração de Tuning usando Exploração Eficiente de Espaço de Projeto* [aplicado para arquitetura SoC].

3.1.1 U-SPaCS - Uma Metodologia de Simulação de Cache em passo Único para Caches de Dois-Níveis Unificadas

Zang e Gordon-Ross propuseram, neste trabalho, o U-SPaCS (ZANG; GORDON-ROSS, 2012), uma metodologia de simulação de cache em passo-único para configuração de cache em tempo de projeto de hierarquias de cache de dois-níveis, com uma cache de segundo nível unificada. Para conseguir um tempo de simulação rápido, o U-SPaCS utiliza um *stack* de instrução e outro *stack* de dados, que permite a simulação de todas as configurações de cache para as caches de dados e de instrução do nível 1, e a cache unificada de nível 2, de forma simultânea em apenas um passo do *trace* de acesso de dados e instrução da aplicação.

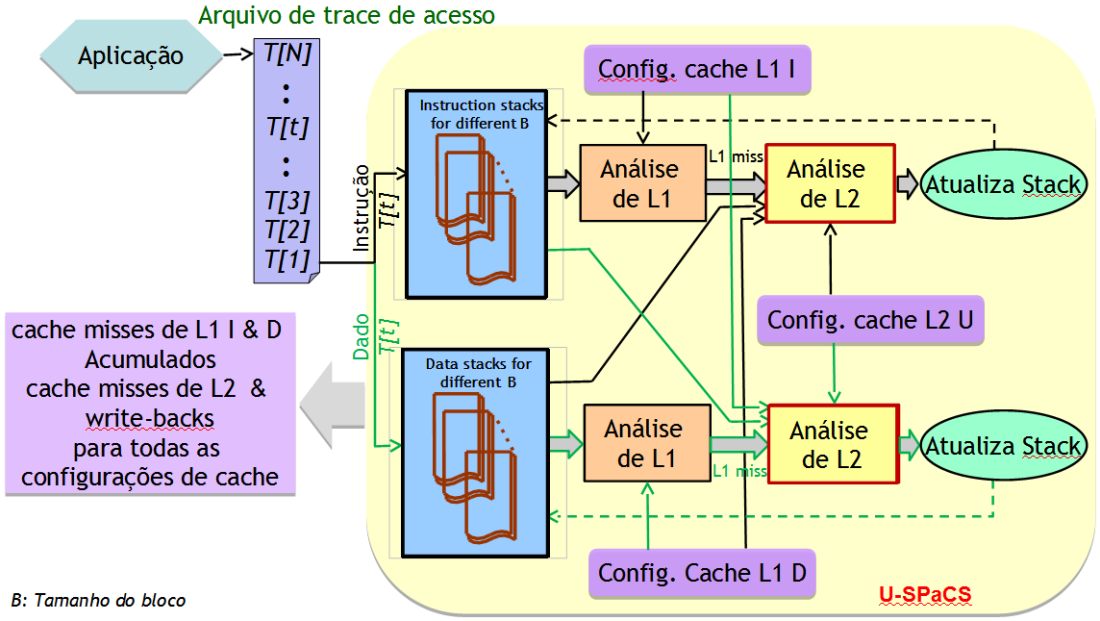
A arquitetura alvo do U-SPaCS é uma cache de dois níveis unificada e exclusiva, consistindo de caches L1 configuráveis, de instrução e dados, e uma cache L2 unificada configurável. Cada uma das caches são consideradas com seu tamanho total, tamanho de bloco e associatividade configuráveis de forma independente. As caches L1 usam a política de relocação LRU (*Least Recently Used*) e a cache L2 usa a política de relocação FIFO (*first-in-first-out*). Dessa forma as três caches podem ser logicamente analisadas como uma única cache combinada, tendo o espaço de projeto total sendo composto por todas as possibilidades de combinação dos valores dos parâmetros de configuração da três caches citadas (desde que o tamanho de bloco das três seja o mesmo).

Para cada configuração da hierarquia de cache, o U-SPaCS obtém o número de *misses* e o número de *write-backs* da cache L2. Esses valores são usados como entrada nos modelos de desempenho/consumo de energia, para determinar a configuração de hierarquia de cache otimizada. O U-SPaCS processa sequencialmente cada endereço do *trace* e avalia cada endereço de *trace* processado comparando-o com endereços acessados previamente para determinar se o endereço processado será um *miss/hit* de cache para cada configuração da hierarquia de cache. Para classificar o endereço processado T como *hit* ou *miss* para uma configuração de hierarquia de cache, o número de blocos previamente acessados que são mapeados para o mesmo *set* de cache que o bloco de cache que contém T, caracterizados como conflitos, devem ser localizados. Se o número de conflitos for grande o suficiente para propagar (*evict*) os acessos anteriores dos blocos de T, o acesso corrente a T resulta em um *miss* de cache. Caso contrário é um *hit*.

O U-SPaCS é um simulador de cache *trace-driven* que mantém estruturas de *stack* separadas para endereços de dados e de instrução. Para cada tamanho de bloco B, U-SPaCS

processa o endereço de *trace* e grava a sequência ordenada de endereços de bloco único de instrução/dados que são mapeados para o mesmo *set* de cache para o número mínimo de *sets* em um *stack* de instrução/dados. A figura 3.2 esboça uma visão geral da operação do U-SPaCS.

Figura 3.2: Visão Geral da Operação do USPACS.



Fonte: ZANG; GORDON-ROSS (2012).

Uma única execução da aplicação (usando um simulador de conjunto de instruções) gera o *trace* de acesso dos endereços de dados e de instrução, incluindo uma informação adicional de leitura/escrita de cada endereço de dados.

O U-SPaCS processa sequencialmente os endereços do *trace*. Dado um particular tamanho de bloco B, o endereço de bloco A de um endereço de instrução T será $A = T >> log_2 B$, onde '>>' é um operador de deslocamento bit-a-bit para a direita. Quando processando o endereço de instrução T, o U-SPaCS primeiro escaneia o *stack* de instruções K_{i_inst} (que possui o *index set i_inst*) buscando o endereço de bloco A de T, para verificar se A já foi buscado previamente. Se A não for localizado em K_{i_inst}, A estará sendo buscado pela primeira vez, e o acesso a T resulta em um *miss* compulsório de cache para todas as configurações de L1 (instrução e dados) e de L2, e o U-SPaCS processará o próximo endereço T do arquivo de *trace*. Se A for localizado em K_{i_inst} em uma locação h ($K_{i\_inst}[h] = A$), A foi previamente buscado e é dado início a "Análise de L1".

A "Análise de L1" avalia os conflitos da cache L1 em K_{i_inst}, entre $K_{i\_inst}[1]$ e $K_{i\_inst}[h]$ para todos os possíveis *sets* de L1 (instruções) para determinar se T gera um *miss/hit* de cache para cada configuração de L1 (instruções)(vide figura 3.2). Para cada configuração de L1 (instruções) que resulta em *miss* de cache, é realizada uma combinação com todas as possíveis configurações da cache L1 de dados e de L2 para formar a combinação de todas as configurações de cache da hierarquia de cache, que será analisada para *hits/miss* de L2 durante a "Análise de

L2". De forma semelhante é realizado para o K_{i_dados}.

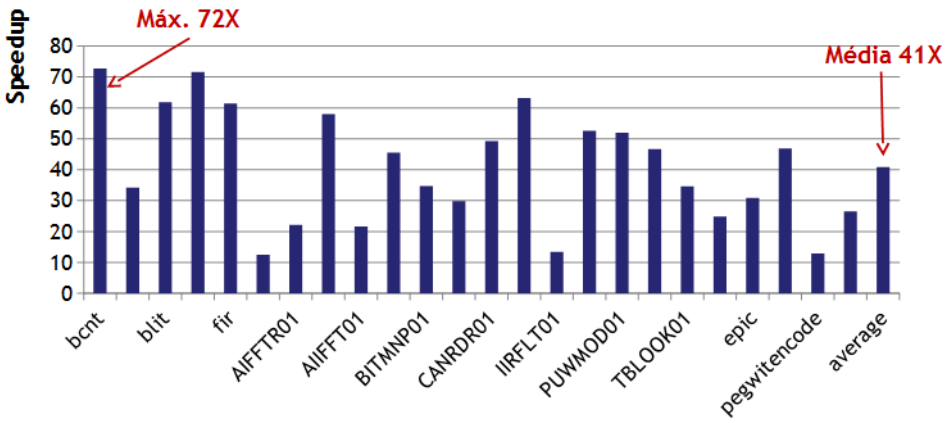
Na "Análise de L2" é realizada uma avaliação de conflito usando K_{i_inst} e K_{i_dados} para todos os possíveis *sets* de L2.(vide figura 3.2). Depois, para cada configuração da hierarquia de cache (L1(instrução e dados) e L2), a "Análise de L2" determina os conflitos da cache L2 para A, que foram propagados (*evicted*) das caches L1 de instrução e de dados depois da propagação de $K_{i\_inst[h]}$ da cache L1 de instrução. O número de conflitos da cache L2 definirá se o acesso ao endereço T gera um *miss/hit* de cache L2.

Depois de avaliar toda a hierarquia de cache para T, o processo "Atualiza Stack" modifica o K_{i_inst} para refletir os acessos ao endereço T. Se A foi previamente acessado, o processo "Atualiza Stack" remove $K_{i\_inst[h]}$ de K_{i_inst} e põe A no topo de K_{i_inst}. Se A não foi previamente acessado, o processo "Atualiza Stack" põe A diretamente no topo de K_{i_inst}.

Depois de processar todos os endereços T do arquivo de *trace*, o U-SPaCS gera o número de *misses* da cache L1 de instruções, da cache L1 de dados, e da cache L2, e o número de *write-backs* para todas as configurações da hierarquia de cache. Esses valores são utilizados para calcular o consumo de energia para todas as configurações da hierarquia de cache, e seleção da configuração de menor consumo de energia e menor tempo de execução.

Como resultado, podemos resumir que o U-SPaCS obteve uma média de 41x de *speedup* de tempo de simulação quando comparado ao simulador Dinero's (EDLER; HILL, 2003). Essa melhoria pode ser verificada através da figura 3.3 que mostra o *speedup* médio e para diversas aplicações.

Figura 3.3: *Speedup* de tempo de simulação do U-SPaCS comparado ao Dinero.



Fonte: ZANG; GORDON-ROSS (2012).

Analisando o U-SPaCS, podemos destacar como pontos positivos o fato da abordagem ser uma abordagem de solução rápida (mesmo explorando todo o espaço de projeto), e por ser aplicável a caches de nível 2 unificadas. Contudo, a abordagem possui como desvantagem os seguintes pontos:

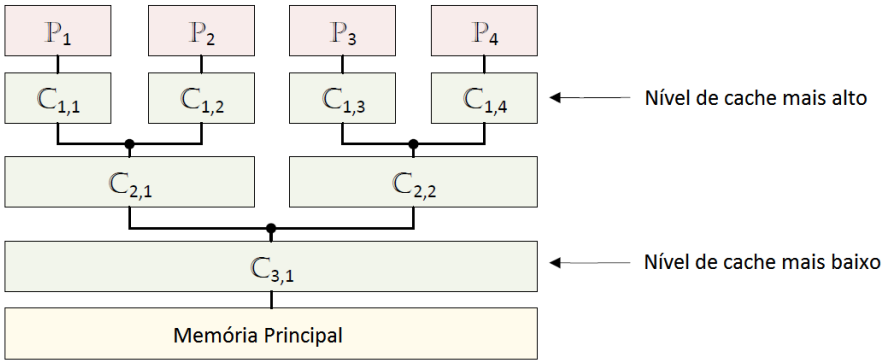
- A abordagem é aplicável apenas para arquiteturas SoC.

- Abordagem amarrada à arquitetura de cache L1/L2 proposta.

3.1.2 Nawinne2015 - Explorando Hierarquias de Cache Multinível em MPSoCs de Aplicação Específica

Nesse trabalho o Nawinne (NAWINNE et al., 2015) apresentou um algoritmo que explora iterativamente a hierarquia de cache de um MPSoC de multi-níveis para encontrar uma configuração de cache de forma a otimizar a velocidade de acesso à memória. Nawinne utilizou um hardware de simulação para calcular rapidamente a quantidade de acessos diretos (*cache hits*) e para extração do *trace* em tempo real, o que permitiu realizar múltiplas iterações sobre a hierarquia de cache do sistema.

Figura 3.4: Arquitetura MPSoC.



Fonte: NAWINNE et al. (2015).

O objetivo principal do trabalho de Nawinne é otimizar o tempo médio de resposta da cache considerando tanto os *misses* de cache quanto os *hits* de cache, em cada nível de cache (vide arquitetura usada no trabalho na figura 3.4). O trabalho assume os acessos a bloco de memória ordenados (uso de barramento) e que as caches não implementam técnica de *pre-fetching*. Essa suposição permite simulação determinística dos *misses* e *hits* de cache.

O termo $T_{i,j,k}$ é definido como o tempo médio de acesso da cache para a configuração $K_{i,j,k}$. $T_{i,j,k}$ para a cache é a média combinada do tempo gasto com *hits* de cache e o tempo gasto com *misses* de cache. O valor de $T_{i,j,k}$ é descrito conforme a equação abaixo:

$$T_{i,j,k} = HL_{i,j,k} + [1 - HR_{i,j,k}] * ML_{i,j,k}, \tag{3.1}$$

onde i é o nível de cache, j é o número da cache em um nível, k é o número da configuração de cache em i e j, $T_{i,j,k}$ é o tempo médio de acesso a hierarquia de cache, $HL_{i,j,k}$ e $ML_{i,j,k}$ são respectivamente a latência de *hit* e a latência de *miss* de cache , e $HR_{i,j,k}$ é a taxa de *hit*. A latência de *miss* é dada por $ML_{i,j,k} = T_{i+1,j',k'} + UL_{i,j,k}$, onde $UL_{i,j,k}$ é o tempo para atualizar um dado/instrução na cache depois da ocorrência de um *miss* de cache, e $T_{i+1,j',k'}$ é o tempo

de médio de resposta da cache no nível $i+1$. Portanto o objetivo principal, que é minimizar o tempo médio de acesso à memória, será dado por T_{TOT} conforme a equação abaixo:

$$T_{TOT} = \sum_{j=1}^{M_1} T_{1,j,k}. \tag{3.2}$$

O algoritmo do sistema de exploração de espaço de projeto de cache proposto segue o esquema do algoritmo 2.

Algoritmo 2 Otimizando uma hierarquia de cache de N-níveis

1: // *Pre-pass*: Simulação para L_1 precisa ser realizada apenas uma vez
2: $\forall\ j := 1$ a M_1 e $\forall\ k := 1$ a $D_{i,j}$: calcular $HR_{1,j,k}$
3: **for** $j := 1$ a M_1 **do**
4: **for** $k := 1$ a $D_{i,j}$ **do** ▷ The g.c.d. of a and b
5: Calcular $ML_{1,j,k}$ para $K_{1,j,k}$, usando o tempo de acesso da DRAM para $T_{2,j',k'}$
6: Avaliar $T_{1,j,k}$
7: Selecionar $K_{1,j,k_{min}} \mid T_{1,j,k_{min}} = \min(T_{1,j,*})$
8: Incluir $K_{1,j,k_{min}}$ no MPSoC para $C_{1,j}$
9: numero de iteração $r = 1$
10: **repeat**
11: **for** $i := 2$ a N **do** ▷ *Forward Pass (FP)*
12: Substituir $C_{i,j}$ pelos simuladores de cache em hardware $\forall\ j := 1$ para M_1 no MPSoC
13: Re-sintetizar o MPSoC
14: $\forall\ j := 1$ a M_i e $\forall\ k := 1$ a $D_{i,j}$: calcular $HR_{i,j,k}$
15: **for** $j := 1$ a M_i **do**
16: **for** $k := 1$ a $D_{i,j}$ **do**
17: **if** $i = N\ OU\ r = 1$ **then**
18: Calcular $ML_{i,j,k}$ substituindo o tempo de acesso da DRAM $T_{i+1,j',k'}$
19: **else**
20: Calcular $ML_{i,j,k}$ usando $T_{i+1,j',k'}$ do próximo nível de cache
21: Avaliar $T_{i,j,k}$
22: Selecionar $K_{i,j,k_{min}} \mid T_{i,j,k_{min}} = \min(T_{i,j,*})$
23: Incluir $K_{i,j,k_{min}}$ no MPSoC para $C_{i,j}$
24: **for** $i := N-1$ a 1 **do** ▷ *Backward Pass (BP)*
25: **for** $j := 1$ a M_i **do**
26: **for** $k := 1$ a $D_{i,j}$ **do**
27: Calcular $ML_{i,j,k}$ usando $T_{i+1,j',k'}$ do próximo nível de cache
28: Avaliar $T_{i,j,k}$
29: Selecionar $K_{i,j,k_{min}} \mid T_{i,j,k_{min}} = \min(T_{i,j,*})$
30: Incluir $K_{i,j,k_{min}}$ no MPSoC para $C_{i,j}$
31: $r := r+1$
32: **until** *nenhuma mudança tenha ocorrido em qualquer configuração* $K_{i,j,k_{min}}$ *selecionada no Backward Pass (BP)*

A simulação *trace-driven* dos espaços de projeto da cache L1 é realizada apenas uma vez (*pre-pass*) no início do algoritmo (linhas 2-8). Os *hits* de cache são calculados simultaneamente para todas as configurações (linhas 5-6). O valor de $T_{2,j',k'}$ é obtido usado apenas o tempo de

acesso da memória DRAM, visto que os níveis de cache são criados na medida que o algoritmo é executado. As configurações com os tempos de acesso mínimos são selecionadas e inclusas na hierarquia de cache (linhas 7-8).

No *forward pass (FP)*, o algoritmo passa pelos níveis de cache iniciando pelo nível 2. As linhas 12-14 mostram que uma mudança de configuração em qualquer nível de cache acima muda os *traces* de acesso recebidos pelos níveis mais baixos sendo necessário se recalcular os *hits* para cada configuração em cada espaço de projeto da cache. Os componentes do simulador de hardware são conectados ao MPSoC (na linha 12) e o sistema é novamente sintetizado(na linha 13). Nas linhas 15-21 os tempos médios de acesso $T_{i,j,k}$ são re-avaliados para todas as configurações. O valor de $T_{i+1,j',k'}$ é utilizado como o dos parâmetros da DRAM apenas se o nível de cache corrente for o último ou se estiverem na primeira iteração, caso contrário o valor de $T_{i+1,j',k'}$ é obtido do nível de cache abaixo. As configurações com os tempos de acesso mínimo são então selecionadas e incluídas no MPSoC (linhas 22-23) substituindo a última configuração definida.

O *backward pass (BP)* é iniciado pelo penúltimo nível de cache e segue até o primeiro nível de cache. Qualquer mudança de configuração em um nível abaixo afetará a penalidade de *miss* do nível de cache sendo trabalhado no *backward pass*. Portanto, $ML_{i,j,k}$ precisa ser atualizado (linha 27). Consequentemente, precisa-se atualizar os tempos de acesso de cada configuração (linha 28). Finalmente, as configurações de cache com menor tempo de acesso para todas as caches visitadas são selecionadas e inclusas no sistema (linhas 29-30), atualizando as configurações da hierarquia de cache.

A execução de um *forward pass* seguido de um *backward pass* formam uma iteração do algoritmo. Se a execução de um *backward pass* não resultar em alterações de configuração de cache, o algoritmo é finalizado e as configurações da hierarquia de cache são retornadas e o algoritmo finaliza sua execução.

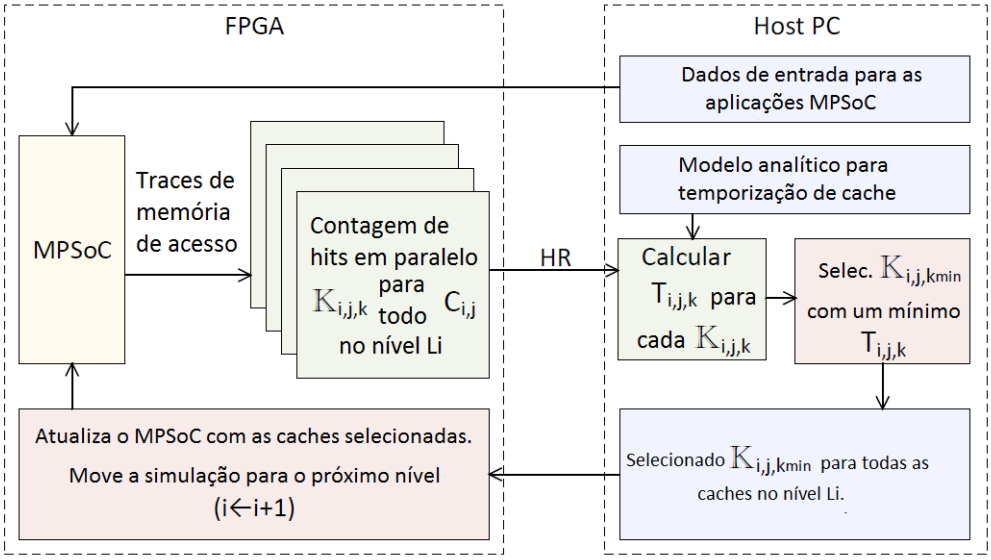
O hardware é incorporado ao sistema no *forward pass*. Figura 3.5 mostra como o dispositivo FPGA é conectado ao sistema.

Um exemplo de como os simuladores de hardware (módulos hSim) são conectados ao MPSoC no *forward pass* é mostrado na figura 3.6.

A figura 3.7 ilustra os detalhes de interface do módulo hSim, que consiste de três portas. A primeira porta é conectada ao nível de cache imediatamente anterior. A segunda porta se conecta ao nível de cache imediatamente superior; e a terceira porta é usada para sinais de controle.

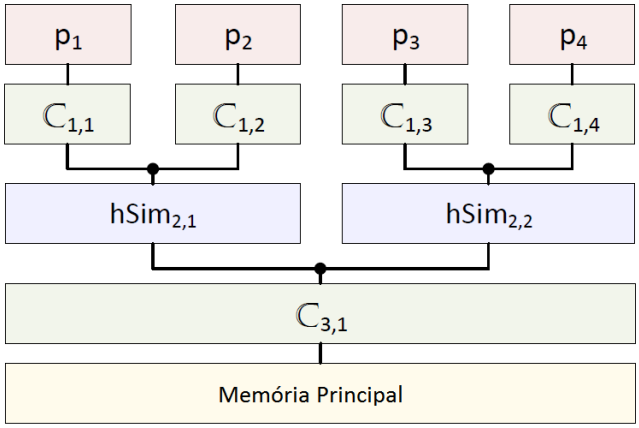
Para obtenção de resultados, o algoritmo foi aplicado a dois diferentes sistemas MPSoC com quatro processadores, o sistema A e o sistema B, cada um executando um grupo de quatro aplicações diferentes (não existe comunicação entre as aplicações). No sistema A o MPSoC contém uma hierarquia de cache de três níveis, com quatro cache L1 privadas, duas caches L2 compartilhadas, e uma cache L3 compartilhada. No sistema B o MPSoC contém uma hierarquia de cache de dois níveis, com quatro caches L1 privadas, e uma cache L2 compartilhada. A figura

Figura 3.5: Sistema de Exploração de Espaço de Projeto.



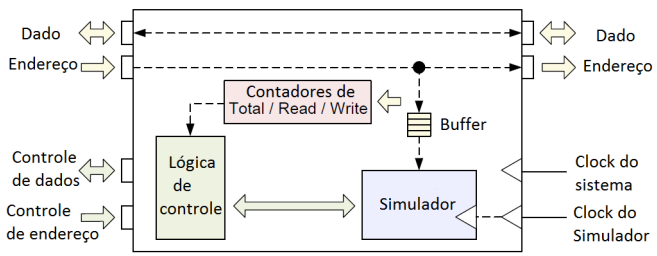
Fonte: NAWINNE et al. (2015).

Figura 3.6: Exemplo de Uso dos Simuladores de Hardware(hSim) no Nivel 2 da Hierarquia de Cache.



Fonte: NAWINNE et al. (2015).

Figura 3.7: Interface e Estrutura do Módulo de *Hardware* - hSim.



Fonte: NAWINNE et al. (2015).

Tabela 3.1: Porção explorada do espaço de projeto comparado com trabalho prévio de Nawinne (NAWINNE et al., 2014)

Sistema	Espaço de projeto total	Explorado em (NAWINNE et al., 2014)	Exploradas neste trabalho	Acréscimo de configurações exploradas
Sistema A	$1,04 \times 10^{13}$	524	2256	$4,31\times$
Sistema B	$5,4 \times 10^{9}$	488	1952	$4\times$

3.8 mostra os resultados da execução do algoritmo para o sistema A. Durante a execução do algoritmo em suas iterações, verificamos na figura 3.8-a a evolução do tamanho de cache, na figura 3.8-b a evolução do tempo médio de acesso, e na figura 3.8-c o tempo total de acesso a dados (T_{TOT}).

De forma similar, a figura 3.9 apresenta a convergência do algoritmo utilizando o sistema B. Durante a execução do algoritmo em suas iterações, verificamos na figura 3.9-a a evolução do tamanho de cache, na figura 3.9-b a evolução do tempo médio de acesso, e na figura 3.9-c o tempo total de acesso a dados (T_{TOT}).

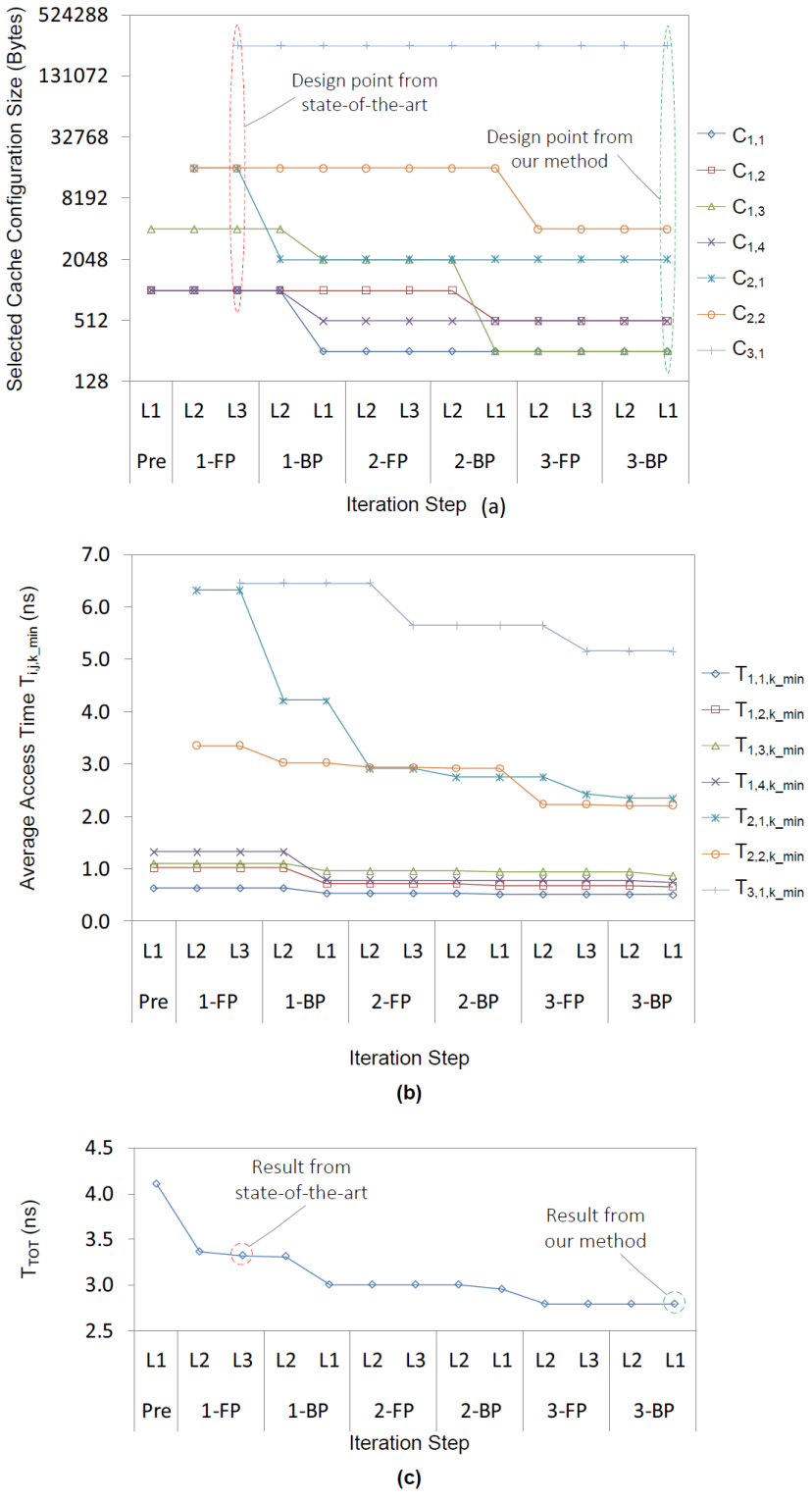
A tabela 3.1 reporta a quantidade de configurações de cache explorados pelo algoritmo, comparado com o trabalho do Nawinne publicado previamente (NAWINNE et al., 2014).

Para o algoritmo também foram submetidos experimentos para investigar se existe convergência para a mesma configuração se a exploração iniciar tanto do topo da hierarquia de cache quanto iniciando da base da hierarquia de cache. Os testes comprovaram que existe convergência para a mesma configuração de cache.

Analisando essa nova abordagem de Nawinne de exploração de espaço de projeto de cache, podemos destacar como pontos positivos o fato de ela ser uma abordagem de solução rápida, de explorar um pequeno percentual do espaço de projeto, e por ser escalável para hierarquias de cache com diversos níveis. Contudo, a abordagem possui como desvantagem os seguintes pontos:

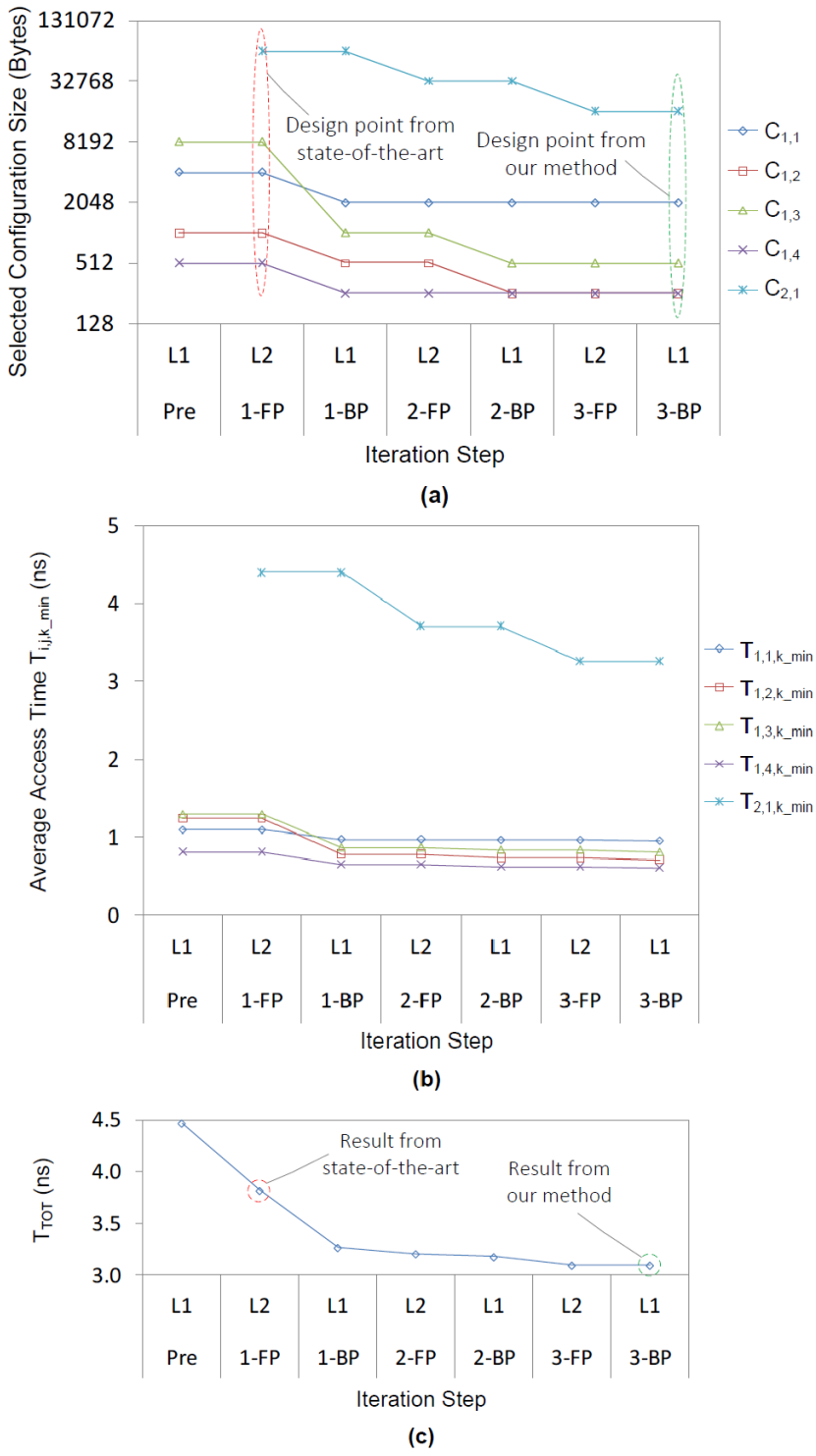
- Não é uma abordagem multi-objetivo pela natureza do algoritmo;
- O trabalho considera o acesso dos processadores ordenado, e para tanto a arquitetura usa um barramento para interligar os processadores. A técnica não se aplica a uma

Figura 3.8: Resultados do experimentos sobre o sistema A: (a) selecionada a configuração de tamanho para as caches $C_{i,j}$; (b) resultados de $T_{i,j,k_{min}}$ para as caches $C_{i,j}$ como verificado no algoritmo; T_{TOT}, configuração obtida em cada passo da iteração.



Fonte: NAWINNE et al. (2015).

Figura 3.9: Resultados do experimentos sobre o sistema B: (a) selecionada a configuração de tamanho para as caches $C_{i,j}$; (b) resultados de $T_{i,j,k_{min}}$ para as caches $C_{i,j}$ como verificado no algoritmo; T_{TOT}, configuração obtida em cada passo da iteração.



Fonte: NAWINNE et al. (2015).

arquitetura com múltiplos processadores interligados por uma NoC.

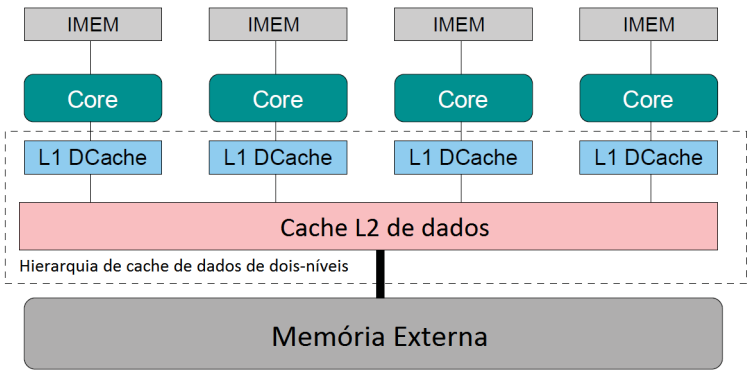
- Não ficou comprovado que a técnica consegue evitar mínimos locais;
- Para atingir o objetivo de redução do tempo de exploração a técnica necessita de Hardware adicional para medição de tempos de acesso a serem usados em simulação das cache através de abordagem *trace-driven*;
- A abordagem não utiliza coerência de cache para as hierarquias de cache utilizadas;
- A abordagem não considera coerência de cache em qualquer nível da hierarquia de cache.

3.1.3 DIMSim - Uma Abordagem Rápida de Simulação de Cache de Dois Níveis para MPSoC baseado em Deadline

No trabalho de Haque (HAQUE et al., 2012) foi proposta uma abordagem de simulação de cache *botton-up* para obter estimativas que satisfaçam restrições de *deadline* para um sistema contendo hierarquias de cache de dados inclusiva de dois-níveis em MPSoCs baseados em *deadline*. Esse trabalho foi chamado de DIMSim.

Com o objetivo de redução do tempo de projeto, Haque escolheu uma estratégia de simulação rápida de cache, simulação *trace-driven* de passo único, para estimar as informações das caches. A abordagem considera coerência de cache e escalabilidade na exploração de espaço de projeto de cache. A função objetivo da abordagem é a redução de tempo de execução do aplicativo. Os parâmetros utilizados para exploração do espaço de projeto são o tamanho total da cache, o nível de associatividade e o tamanho do bloco da cache. A figura 3.10 mostra a arquitetura MPSoC utilizada para a abordagem DIMSim. A arquitetura mostra que a interconexão entre os processadores é feita através de barramento.

Figura 3.10: Hierarquia de cache de dois-níveis em arquitetura MPSoC.



Fonte: HAQUE et al. (2012).

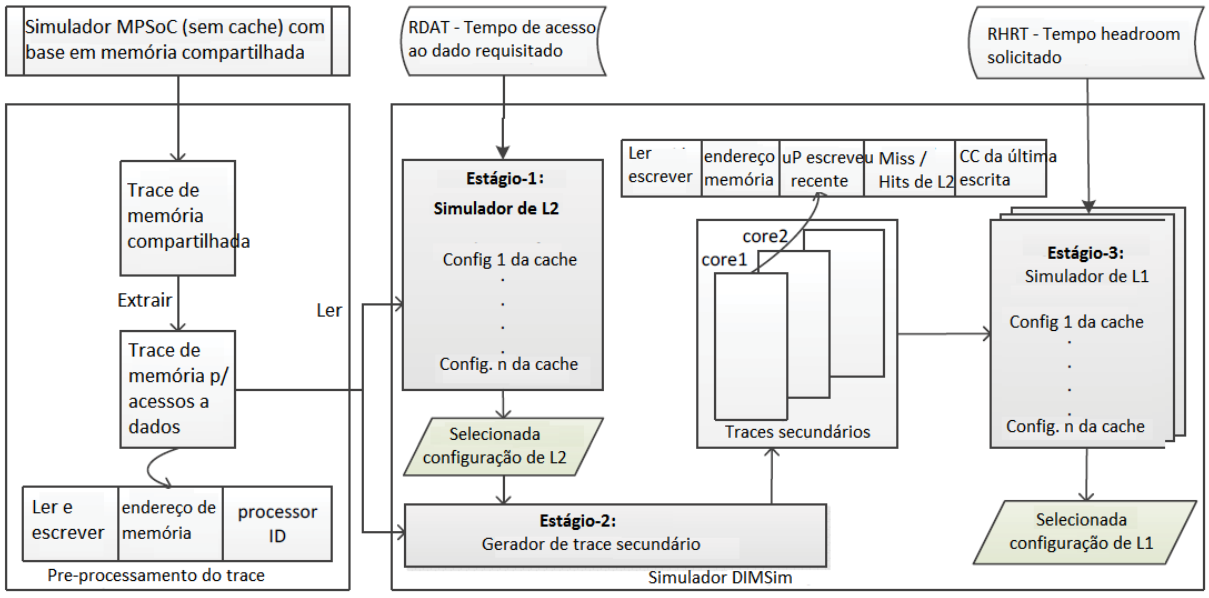
O DIMSim utiliza como entrada o *deadline* (tempo de execução requisitado - RET) da aplicação em uso. A seguinte equação representa o *deadline* RET:

$$RET \geq AET = TTDM + TTNM + TTOH, \tag{3.3}$$

onde RET é o tempo de execução requisitado, AET é o tempo de execução atual, $TTDM$ é o tempo total gasto com operações com memória de dados, $TTNM$ é tempo total gasto em operações que não são como memória de dados, $TTOH$ é o tempo total gasto em *overheads* (como o tempo gasto com os *overheads* do sistema operacional, que são específicas do sistema e não são cobertos pelo trace da aplicação). O modelo de temporização do DIMSim é formulado de forma que ao introduzir uma hierarquia de cache (inicia somente com a memória) ou modificar a sua configuração, apenas $TTDM$ será afetado ($TTNM$ e $TTOH$ não mudam).

O RET é provido pelo usuário (considerando as restrições de latência e *throughput* da aplicação). O Usuário também fornecerá o RHRT (tempo *headroom* requisitado). O DIMSim busca satisfazer tanto o $TTDM$ (através de sua estimativa de L2) quanto o $TTOH$ (através de estimativa de L1). A figura 3.11 mostra o fluxo do DIMSim.

Figura 3.11: Fluxo de simulação do DIMSim.



Fonte: HAQUE et al. (2012).

O pré-processamento do trace é realizado antes da execução da simulação DIMSim, para criar o *trace* de memória compartilhada e adicionar a ele *tags*. A simulação DIMSim é dividida em três estágios: (1) Simulação de L2, (2) Geração de trace secundário; e (3) Simulação de L1. O pré-processamento é realizado para preparar o trace para o simulador. Enquanto o MPSoC (sem caches) está executando uma aplicação com tarefas que se comunicam ou múltiplas aplicações, os acessos à memória são observados e registrados no controlador de memória (a ordem de acesso registrada é a correta). Os acessos aos dados são extraídos e anotados (leitura/escrita,

endereço de memória, *processor ID*) conforme descrito na figura 3.11. Segue a descrição dos três estágios do simulador:

3.1.3.1 Estágio-1: Configuração de cache L2 de dados que satisfaça o RDAT

Como exibido na figura 3.11, no estágio-1, o DIMSim usa o trace de acesso a dados completo e o tempo RDAT (= $RET - TTNM$) e estima a configuração de cache L2 compartilhada. No estágio-1, o DIMSim assume que o sistema possui apenas uma memória compartilhada e que não existe cache L1. Como o tempo $TTDM_{L2}$ (apenas a cache L2 está presente) da aplicação aumenta com o número de faltas de cache (*cache misses*), o número aceitável de faltas de cache em L2, chamado *ML*, que satisfaz o *RDAT* é calculado usando a seguinte equação:

$$RDAT \geq TTDM_{L2} = ML \times (T_M + T_{L2}) + (A - ML) \times T_{L2}. \quad (3.4)$$

Como descrito pela equação, quando um dado é requisitado pelo processador, a cache L2 é verificada primeiro e se o dado não for encontrado, a memória é verificada. Cada falta de cache em L2 significará T_M segundos adicionais. O dado que é encontrado em L2 é servido por L2 em T_{L2} segundos. Considerando que o número total de acessos é A (por todos os processadores), o tempo total de sucessos (número de *hits*) é $(A - ML) * T_{L2}$ segundos.

Lendo o arquivo de trace apenas uma vez, o DIMSim simula todas as configurações de cache disponíveis para L2. O resultado dessa simulação será um conjunto de configurações que satisfazem o *ML*. Entre elas, a configuração de menor tamanho é escolhida como a cache compartilhada L2 adequada para minimizar a área e minimizar o consumo de energia. A simulação de L2 é implementada de forma a evitar a simulação de configurações de cache que não estejam em conformidade com *ML*.

3.1.3.2 Estágio-2: Trace secundário para o simulador da cache L1

Quando uma cache L2 é compartilhada entre todos os processadores, a coerência de cache é o primeiro desafio que deve ser tratado pelo simulador para encontrar as caches L1 privadas. A proposta do DIMSim é usar dois simuladores (de L2 e L1), de forma independente e comunicante, passando as informações do simulador de L2 para o simulador de L1. A interação entre os simuladores é realizada através da criação do arquivo de trace secundário que é elaborado com base no trace original e com informações da cache L2 selecionada no estágio-1. Como verificado na figura 3.11, para cada endereço de bloco de memória, um arquivo de trace secundário inclui: (1) código de operação (*opcode*) do acesso ao dado,(2) o endereço de memória solicitado, (3) o identificador do processador que escreveu nesse bloco de memória mais recentemente, (4) se o bloco gerou um *miss* ou um *hit* na cache L2 selecionada, e (5) ciclo de *clock* quando bloco de memória foi escrito na memória cache L2. Toda a informação extra (comparada ao trace original) gravada no endereço do bloco de memória ajuda na implementação do protocolo de coerência de cache do MPSoC.

3.1.3.3 Estágio-3: Configurações de cache L1 para a cache L2 compartilhada

Como visto na figura 3.11, uma vez que o arquivo de trace secundário foi gerado, ele é dividido em arquivos de trace menores agrupando os acessos de cada processador em um arquivo separado. Para encontrar a cache L1 privada adequada para um processador em particular, seu respectivo arquivo de trace é utilizado. Dividindo o arquivo de trace secundário em arquivos de trace menores permite que o processo de simulação seja paralelizado e portanto a simulação de L1 pode ser realizada mais rapidamente se desejado. O DIMSim faz uso do fato de que adicionando caches L1 ao sistema acelerará o sistema para satisfazer o segundo requisito de tempo, $RHRT$, submetido como outra entrada (independente do RET). O requisito $RHRT$ é distribuído entre os processadores proporcionalmente ao número de acessos à memória de dados por cada processador. As configurações de cache L1 adequadas são selecionadas para satisfazer o requisito $RHRT$ de cada processador. É importante notar que o DIMSim ignora o fato que processadores individuais podem acessar suas caches L1 privadas simultaneamente como o trace de entrada não tem essa informação. Portanto, o DIMSim deve potencialmente satisfazer o requisito $RHRT$ com uma margem maior.

Ao ler o arquivo de trace secundário uma vez, o DIMSim simula todas as configurações de cache disponíveis para encontrar a cache L1 privada adequada para cada processador. Para cada configuração de cache L1 simulada em cada processador, o número total de requisições de acesso à memória que não pôde ser servida pela configuração de cache particular (faltas na cache L1) é registrada em tempo de execução. Para calcular o número total de faltas de cache incorporando coerência de cache, o DIMSim registra o último tempo de atualização de conteúdo em cada linha de cache para as configurações de cache simuladas. Qualquer endereço de memória que é indicado como uma falta em L2 no arquivo de trace secundário, causará também uma falta na memória cache privada L1 durante a simulação devido a propriedade inclusiva. Quando um conteúdo de endereço de memória requisitado é encontrado na memória compartilhada L2 bem como na cache privada L1, o último tempo de atualização escrito na linha de cache privada, que está mantendo o conteúdo, e o tempo de atualização L2 escrito no arquivo de trace secundário são comparados. Se eles forem iguais, um hit de cache é declarado; caso contrário, uma falta de cache é registrada. Faltas de cache são também registradas se o conteúdo do endereço de memória não estiver presente na configuração de cache privada. Em uma falta de cache, o conteúdo solicitado é colocado em uma linha de cache na configuração de cache simulada, juntamente com seu último tempo de atualização na memória cache L2 compartilhada. Dessa forma, a abordagem DIMSim implementa protocolo de coerência sem consumos adicionais de armazenamento e de tempo.

O algoritmo 3 lê cada entrada dos traces e tenta avaliar se ocorre um *cache miss* ou *cache hit* em cada configuração de cache simulada. O algoritmo recebe o endereço requisitado (RA) do arquivo de trace secundário (para cada cache L1 privada haverá um arquivo de trace secundário). Cada linha no arquivo de trace secundário é marcada a informação de *miss* ou *hit*

como mostrado na figura 3.11. Devido à propriedade inclusiva das cahes L1 e L2, um *miss* em L2 causará um *miss* em L1. Como mostrado no algoritmo 3, a função *HandleCacheMiss* é chamada quando RA não está disponível na cache L2 (o *hit* em L2 para esse endereço no arquivo de trace secundário indica que o dado está disponível, da mesma forma um *miss* em L2 para esse dado indica que o endereço está indisponível). Se o *RA* está disponível em L2 (else da linha 3), buscar na estrutura de dados L1 (isto é, na *CLT*) se o *RA* estiver disponível. A função *HandleCacheMiss* é chamada se *RA* não estiver disponível na *CLT*, o que indica um *miss* nesta cache L1 particular. Um *hit* em L1 [isto é, *RA* está disponível na *CLT* (Na linha 7)] exige que a informação seja registrada na árvore de simulação. Iniciando a partir da raiz da árvore de simulação, todos os níveis na árvore são explorados, até atingir o nível do maior tamanho de linha. É importante notar que os *cache misses* podem crescer devido a leitura ou escrita de dado. Para cada nível (isto é, tamanho de linha), explorar todas as associatividades, iniciando da menor (esta iteração é considerada como a análise de uma configuração de cache). Se *RA* é encontrado na configuração de cache selecionada na árvore, verificar se o tempo de escrita corresponde com o tempo de escrita em L2. Uma correspondência sugere um *cache hit* de dados em L1, portanto será atualizado como um *hit* para essa configuração na árvore de simulação. Uma não correspondência dos tempos denota um *cache miss* em L1. Portanto a configuração de cache na árvore de simulação é atualizada, incluindo o tempo de escrita e um registro no *CLT* para indicar inserção de *tag*. Se a configuração de cache não inclui o *RA*, registrar um *cache miss* na configuração e atualizar a árvore de simulação, enquanto anexa o tempo de escrita mais recente para o *RA* e uma atualização na *CLT* para indicar inserção de *tag*. O loop mais interno é realizado para cada associatividade, e o *loop* mais externo é iterado para cada tamanho do bloco.

A abordagem de Haque (DIMSim), em sua execução no ambiente utilizado em seus experimentos, tomou cerca de uma hora (pior caso) para estimar a cache compartilhada L2, e as caches L1 privadas para uma hierarquia de dois níveis de cache de dados (Figura 3.10). Para as configurações de cache selecionadas a abordagem obteve uma média de 12,6% de desvio do tempo de acesso ao dado requisitado (RDAT) fornecidos ao algoritmo como *deadline* das aplicações.

Analisando a abordagem de Haque de exploração de espaço de projeto de cache, podemos destacar como pontos positivos o fato de ela ser uma abordagem de solução rápida (quantidade reduzida de simulações), de considerar na simulação a coerência das caches, não utiliza hardware adicional nas simulações e por ser escalável para hierarquias de dois-níveis de cache em diversos processadores. Contudo, a abordagem possui como desvantagem os seguintes pontos:

- Não é uma abordagem multi-objetivo pela natureza do algoritmo;
- O trabalho de Haque, mesmo considerando coerência das cache, não considera que o acesso à memória ou ao último nível de cache seja realizado de forma não ordenada. A técnica não se aplica a uma arquitetura com múltiplos processadores interligados por uma NoC.

Algoritmo 3 AddressEvaluation(RequestedAddress(RA))

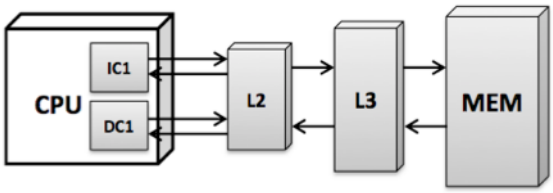
```
1: if RA não estava disponível na cache L2 then
2:     HandleCacheMiss(RA)
3: else
4:     Busca RA na look-up table (CLT);
5:     if RA não estava disponível na CLT then
6:         HandleCacheMiss(RA)
7:     else
8:         seleciona a raiz da árvore L = 0 (para o tamanho do bloco da cache S = 2^L);
9:         Tanto para leitura quanto escrita de dados;
10:        while 2^L não for maior que o maior tamanho do bloco da cache do
11:            A= a menor associatividade;
12:            while A não for maior que o maior associatividade da cache do
13:                if RA está disponível na configuração de cache selecionada na árvore then
14:                    if O tempo de escrita de RA ≠ tempo de escrita em L2 then
15:                        Registra um cache miss na configuração de cache e atualiza a árvore;
16:                        Anexa o tempo de escrita com RA na árvore;
17:                        Atualiza os records do CLT para indicar inserção de tag;
18:                    else
19:                        Registra cache hit para a configuração de cache selecionada;
20:                else
21:                    Registra um cache miss na configuração de cache e atualiza a árvore;
22:                    Anexa o tempo de escrita mais recente com RA na árvore;
23:                    Atualiza os records do CLT para indicar inserção de tag;
24:                A= a próxima maior associatividade;
25:            L = L + 1 (segue para o próximo nível);
```

- Não ficou comprovado que a técnica consegue evitar mínimos locais;
- Aplicado apenas a aplicações que tenham *deadline* predefinido;
- Existe uma imprecisão implícita dos resultados de tempo na simulação;
- Técnica totalmente atrelada ao número de níveis da hierarquia de cache.

3.1.4 ABCMOP - Algoritmo de Colônia de Abelhas aplicado a Exploração de Arquitetura de Memória para Redução de Energia

O ABCMOP (SANTOS; SILVA-FILHO, 2014) é um algoritmo de busca baseado no algoritmo ABC (KARABOGA, 2005) que foi adaptado para problemas multi-objetivo de cache com a finalidade de otimizar consumo de energia e o número de ciclos para execução de uma aplicação. Neste trabalho, três níveis de hierarquia de cache foram analisadas, e em cada nível os parâmetro de cache configuráveis utilizados foram tamanho de cache, associatividade e tamanho do bloco. A figura 3.12 ilustra a arquitetura utilizada.

Figura 3.12: Arquitetura SoC com Hierarquia de memória.



Fonte: SANTOS; SILVA-FILHO (2014).

Os modelos de energia e de ciclos de CPU usados para calcular foi baseado no método proposto por Silva-Filho (SILVA-FILHO et al., 2006), as ferramentas utilizadas para execução do experimento foram o SimpleScalar (BURGER; AUSTIN, 1997), o CACTI (MURALIMANOHAR; BALASUBRAMONIAN; JOUPPI, 2008) e o CAeTO (ferramenta desenvolvida por um grupo de pesquisa na Universidade Federal de Pernambuco)(SILVA-FILHO et al., 2007).

O pseudo-código do ABCMOP é exibido no algoritmo 4.

Na fase inicial é criada a população inicial *P*. Cada fonte de alimento da população *P* é representada por um vetor de 12 inteiros correspondendo aos 3 parâmetros de cada cache (IC1, DC1, L2 e L3). Depois o algoritmo calcula o *fitness* de cada fonte de alimento utilizando os valores de consumo de energia e o número de ciclos de CPU, da simulação da aplicação para a configuração de cache de cada elemento de *P*.

Os passos 5-27 representam o *loop* principal do algoritmo que finaliza com o critério de parada do algoritmo. O critério de parada usado pelo ABCMOP é o Indicador de Cobertura (TAN; LEE; KHOR, 2001). O Indicador de Cobertura avalia se houve uma melhoria da população anterior para a população corrente. Se o Indicador de Cobertura é zero (não houve melhorias),

Algoritmo 4 Algoritmo ABCMOP

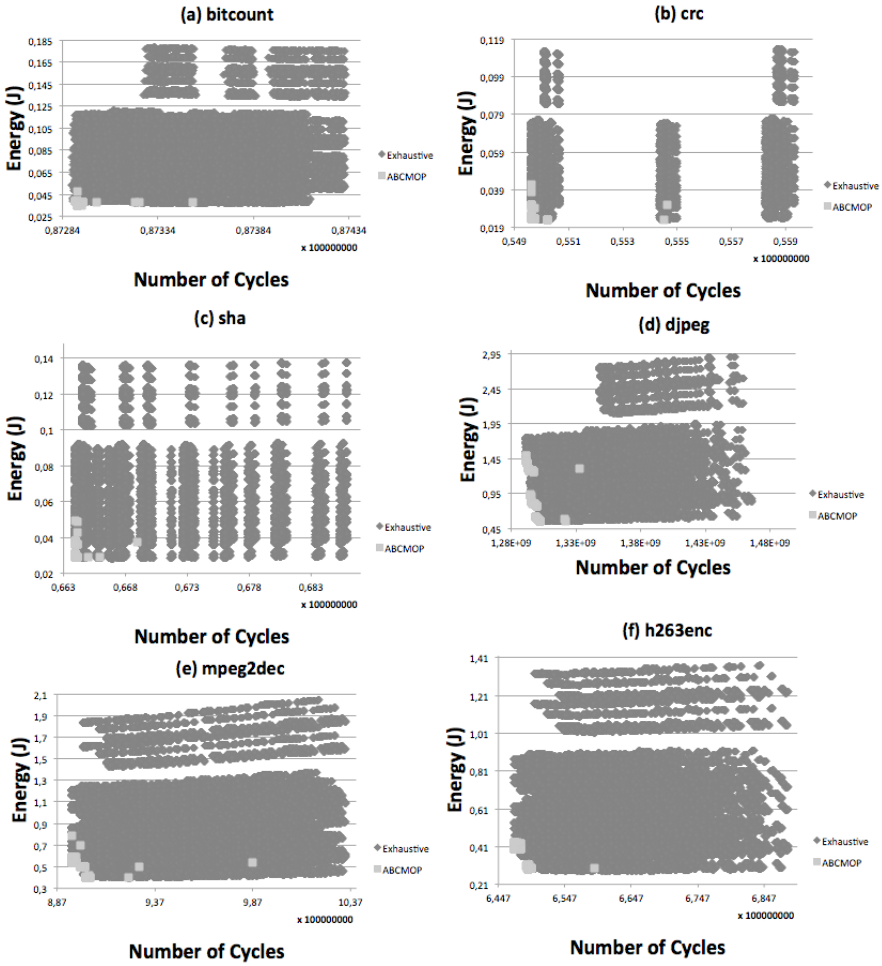
```
1: Fase de Inicialização:
2:     Gera uma população P com indivíduos selecionados randomicamente.
3:     Calcula os objetivos para cada indivíduo na população P.
4: Fim
5: repeat
6:     Fase das Abelhas Employed
7:         for Cada abelha Employed i do
8:             Produz uma nova fonte de alimento i'
9:             Obtém a a função objetivo para a nova fonte de alimento i'
10:            if i'<i conforme critério de dominância then
11:                i' é colocada na população P'
12:    Fim
13:    Fase das Abelhas OnLookers
14:        for Cada abelha onLooker j do
15:            if Número randômico [0.5,1] < p_i, onde p_i é sua probabilidade de recrutamento then
16:                produz uma nova fonte de alimento i'
17:                Calcula o objetivo da nova fonte de alimento i'
18:                if i'<i conforme critério de dominância then
19:                    i' é colocada na população P'
20:    Fim
21:    Fase das Abelhas Scout
22:        Gera uma nova solução randomica w
23:        w é colocado na população P'
24:        Ordena P e P' de acordo com critério de dominância
25:        de forma que as melhores soluções fiquem em PUP'.
26:    Fim
27: until Critério de parada ser atingido
```

então o algoritmo termina sua execução. A última população é considerada a solução do algoritmo. Durante a fase das abelhas *employed*, cada uma dessas abelhas produzirá uma nova fonte de alimento i' através de um processo de mutação. A probabilidade D de mutação de cada parâmetro é definida. Depois de criada a nova fonte de alimento i', o algoritmo calcula o seu *fitness*. Para avaliar se i' é melhor que i, o ABCMOP usa critério de dominância. Caso i' seja melhor que i, o i' é colocado em P'. Na fase das abelhas *onLookers*, essas abelhas realizam um trabalho semelhante ao das abelhas *employed*, exceto que as melhores fontes de alimento j é que serão selecionadas, de acordo com a sua probabilidade de recrutamento. Na fase das abelhas *scout* uma solução randômica com base em procedimento de inserção é gerada. Esse indivíduo é colocado na população P'. Não é utilizado aqui o parâmetro *Limit* (original do algoritmo ABC) nessa fase, que trabalha para evitar resultados em mínimo local. No final, todas as fontes de alimento de P e de P' são ordenadas conforme critério de dominância, e as N melhores são selecionadas no conjunto $P \cup P'$.

Para obtenção de resultados, simulações foram realizadas usando seis aplicações. Os resultados obtidos foram comparados com o espaço de solução completo (simulação exaustiva) que utiliza todas as 28.800 possíveis configurações de cache. Gráficos comparativos são mostrados

na figura 3.13.

Figura 3.13: ABCMOP versus Simulação Exaustiva para as seguintes aplicações do MIBench: (a)-Bitcount, (b)-CRC, (c)-Sha, (d)-Djpeg, (e)-Mpeg2dec, (f)-H263enc.



Fonte: SANTOS; SILVA-FILHO (2014).

A figura 3.14 mostra os resultados do ABCMOP em comparação com as execuções exaustivas.

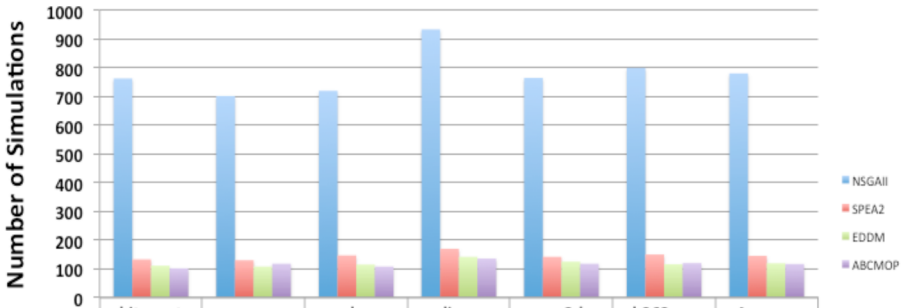
O ABCMOP também foi comparado com três outros diferentes algoritmos - o SPEA2, o NSGAII e o *Diferential Evolution*. A figura 3.15 mostra o resultado da comparação relativo ao número de simulações necessárias para executar os algoritmos para cada aplicação.

Figura 3.14: Tempo de Simulação ABCMOP versus Tempo de Simulação Exaustiva.

Aplicação	% de Exploração	Tempo ABCMOP	Tempo Exaustivo
Bitcount	0,424%	0h 22m	03d 16h
CRC	0,479%	0h 23m	03d 08h
SHA	0,549%	0h 32m	04d 00h
DJPEG	0,611%	2h 44m	18d 16h
MPEG3DEC	0,549%	2h 01m	15d 08h
H263ENC	1,80%	5h 46m	13d 08h

Fonte: SANTOS; SILVA-FILHO (2014).

Figura 3.15: Número de Simulações necessárias para cada Algoritmo e aplicação.



	bitcount	crc	sha	djpeg	mpeg2dec	h263enc	Average
NSGAII	763	702	720	934	765	800	780,67
SPEA2	133	130	147	170	142	150	145,33
EDDM	111	108	115	142	125	116	119,5
ABCMOP	102	118	108	136	118	120	117

Fonte: SANTOS; SILVA-FILHO (2014).

Analisando o ABCMOP, podemos destacar como pontos positivos o fato de ela ser uma abordagem de solução simples, de explorar um percentual relativamente pequeno do espaço de projeto, e por ser aplicável a hierarquia de cache completa. Contudo, a abordagem possui como pontos de melhoria as seguintes características:

- A abordagem não foi aplicada/analizada em arquiteturas MPSoC com múltiplos processadores.
- Não ficou comprovado que a abordagem evita mínimos locais em sua execução, devido às mudanças realizadas na sua fase das abelhas *scout*.
- A abordagem utiliza o algoritmo ABC modificando-o apenas para multi-objetivo sem realizar melhorias para redução de tempo de execução e/ou redução de execuções da plataforma. Essas mudanças são fundamentais para sua aplicação a plataformas com múltiplos processadores.

3.2 Análise Comparativa do Estado da Arte

Segue abaixo tabela comparativa (tabela 3.2) entre as abordagens de exploração de espaço de projeto de cache em plataformas MPSoC utilizadas como estado da arte deste trabalho:

Tabela 3.2: Análise Comparativa do Estado da Arte.

Técnica	**Objetivo de Melhoria**	**Evita mínimos locais**	**Arquitetura**	**MPSoC com Coerência de cache**	**Aplicada aos níveis de cache**	**Aplicável a MPSoCs com NoC**	**Escalabilidade do tempo de execução**	**Necessidade de Uso de memória**
U-SPaCS	Desempenho e Energia	Evita	SoC	Não Aplicável	Apenas aos níveis L1 e L2	Aplicável a SoCs	Possui	Baixo Uso
NAWINNE et al. (2015)	Desempenho	Não comprovado	MPSoC	Não considera	Múltiplos níveis	Não	Possui	Baixo Uso
DIMSim	Desempenho	Não comprovado	MPSoC	Considera	Apenas aos níveis L1 e L2	Não	Possui	Baixo Uso
ABCMOP	Desempenho e Energia	Não comprovado	SoC	Não Aplicável	Múltiplos níveis	Aplicável a SoCs	Tempo de execução não otimizado	Baixo Uso

Analisando a tabela 3.2 podemos verificar que as técnicas que trabalham com multi-objetivo, neste caso desempenho e consumo de energia, realizam uma tarefa bem mais complexa que as técnicas que trabalham com objetivo único. Também vimos que os projetos de sistemas embarcado buscam otimizar múltiplos-objetivos cujos principais são minimizar desempenho de aplicações específicas, consumo de energia e preço. São poucas as técnicas aplicadas a sistemas MPSoC em tempo de projeto que trabalham com multi-objetivo no estado da arte.

Evitar mínimos locais significa que o algoritmo inclui alguma técnica que avalia se os valores obtidos de configurações de cache visando minimização dos objetivos, não foi conduzido na direção de valores em mínimo local. Quando não se evita mínimos locais, a solução final pode não ficar próxima às soluções ótimas. Apenas o U-SPaCS comprovadamente evita mínimos locais.

Independente da classificação de Zang e Gordon-Ross (ZANG; GORDON-ROSS, 2013), todas as técnicas de reconfiguração de cache são aplicadas a arquiteturas com um processador ou com múltiplos processadores. As técnicas que são aplicadas a arquiteturas SoC, nem sempre são aplicáveis à arquiteturas MPSoC, pois as arquiteturas MPSoC exigem tratamentos adicionais para resolver problemas como os descritos na seção 2.2. Apesar disso, alguns trabalhos de recon-

figuração de cache para arquiteturas MPSoC, são abordadas de forma a tratarem a arquitetura como sendo múltiplos SoCs sem considerar coerência de cache, como o trabalho de Nawinne (NAWINNE et al., 2015).

Vimos que os MPSoCs favorecem a otimização de suas hierarquias de cache para melhor desempenho e menor consumo de energia porque esses sistemas executam uma aplicação ou um conjunto de aplicações de forma repetida. Portanto, é de relevante importância que os projetos de sistemas embarcados com base em plataformas MPSoC façam uso de técnicas *tuning* de cache que sejam aplicáveis a toda hierarquia de cache para melhor resolver restrições de projeto como melhoria de desempenho e redução de consumo de energia. Entre os trabalhos apresentados, apenas as técnicas ABCMOP e o Nawinne2015 são aplicáveis a toda hierarquia de cache de uma plataforma.

Observamos, também, que algumas técnicas aplicáveis a arquiteturas MPSoC são aplicáveis especificamente para às baseadas em barramento, como os trabalhos de Nawinne (NAWINNE et al., 2015) e o DIMSim (HAQUE et al., 2012). Porém, o estado da arte ainda apresenta poucos trabalhos que são aplicáveis a arquiteturas MPSoC baseadas em NoC, como o trabalho de Rawlins e Gordon-Ross (RAWLINS; GORDON-ROSS, 2013).

Por fim, verificamos que os trabalhos de Nawinne (NAWINNE et al., 2015) e o DIMSim (HAQUE et al., 2012) comprovam ser abordagens que possuem escalabilidade quanto ao seu tempo de execução. Isso quer dizer que essas técnicas podem ser executadas, sem grande impacto de tempo de execução, para espaços de projeto de cache muito amplos. Nos dois casos isso ocorre pelo fato de usarem técnicas de exploração *trace-driven*. O mesmo pode não ocorrer para a abordagem ABCMOP, devido ao fato de trabalhar com simulação da arquitetura e de não possuir otimização de tempo de execução para a metaheurística utilizada.

Todos os trabalhos analisados possuem baixo uso de memória para suas execuções. A única observação a ser feita é para a abordagem de Haque (HAQUE et al., 2012) que faz uso de hardware FPGA (*Field-Programmable Gate Array*) que consome recursos de memória internamente e que foi descrito no trabalho.

A abordagem de exploração de espaço de projeto de cache proposta neste trabalho, busca trabalhar com multi-objetivo (consumo de energia e desempenho), utilizando uma meta-heurística que evita resultados de busca em mínimos locais, aplicável a plataformas MPSoC baseadas em NoC que trabalham com múltiplos níveis de cache, incluindo coerência de cache. Adicionalmente, a meta-heurística selecionada (algoritmo ABC) foi alterada para melhoria de sua busca global, visando reduzir a quantidade de simulações da plataforma MPSoC e buscar garantir escalabilidade de seu tempo de execução.

4

Abordagem Proposta

4.1 Introdução

Em projetos de sistemas embarcados, a tarefa de otimização de métricas envolve múltiplas possibilidades de atuação em elementos diversos de uma plataforma. Analisamos e definimos a memória cache como circuito foco dessa otimização na plataforma. Portanto, buscaremos neste trabalho realizar otimizações de sistemas embarcados propondo uma abordagem de exploração do espaço de projeto de cache multi-nível com otimização multi-objetivo incluindo consumo de energia e desempenho, os quais serão aplicados a sistemas embarcados em plataformas multi-core que compartilham dados entre os núcleos processadores. A abordagem proposta se baseia no algoritmo ABC como mecanismo de busca otimizando o seu tempo de execução através da utilização de técnicas de DoE.

Na seção 4.2 é apresentado o algoritmo ABC para otimização multi-objetivo que será usado como base para construção do algoritmo proposto AbcDE (*Artificial bee colony cache Design space Explorer*). A seção 4.3 apresenta o algoritmo AbcDE detalhando as técnicas propostas para melhoria de eficiência do algoritmo. O modelo de energia utilizado na plataforma para gerar o consumo de energia demandado por cada cache é descrito na seção 4.4.

4.2 Algoritmo ABC modificado para Otimização MultiObjetivo

O algoritmo ABC tem sido largamente estudado e aplicado à soluções de problemas reais (KARABOGA et al., 2014) e possui um processo de busca completo, que busca eficiência no processo de exploração e explotação do espaço de projeto (KARABOGA; AKAY, 2009). Diversos trabalhos no estado da arte têm apresentado excelentes resultados onde algoritmos que utilizam a colônia artificial de abelhas mostram desempenho superior aos baseados em abordagem genética (HEDAYATZADEH et al., 2010)(XINYI; ZUNCHAO; LIQIANG, 2012)(ZOU et al., 2011). Os bons resultados do algoritmo ABC estão fortemente relacionados com seu processo de

busca (KARABOGA; AKAY, 2009) que inclui: (1) um processo probabilístico global (aplicado pelas abelhas *onlookers*), (2) um processo probabilístico local (aplicado pelas abelhas *employeds* e *onlookers*), (3) um processo de seleção guloso local(aplicado pelas abelhas *employeds* e *onlookers*), e (4) um processo de seleção randômico (realizado pelas abelhas *scout*).

Devido à técnica proposta de exploração de espaço de projeto considerar dois diferentes objetivos, que são otimizar desempenho e consumo de energia, foi necessário adaptar o algoritmo original ABC para tratar com otimização multi-objetivo. Um conjunto adicional de fonte de alimento foi introduzido, chamado de *bestFoods* para substituir o *best solution* (fonte de alimento única) do algoritmo ABC original. A análise de adequação (*fitness*) das fontes de alimento foi substituída por uma análise de Dominância de Pareto da fontes de alimento.

Essa versão do algoritmo ABC com otimização multi-objetivo foi usada como base para a criação do algoritmo AbcDE proposto neste trabalho. As características aqui descritas não caracterizam uma contribuição para o trabalho final proposto. Abordagens anteriores do algoritmo ABC já introduziram otimização multi-objetivo (SANTOS; SILVA-FILHO, 2014)(NAIDU; MOKHLIS; BAKAR, 2014) e alguns dos conceitos e técnicas aqui descritos. Nas sub-seções seguintes descreveremos as atualizações por nós propostas e realizadas para o algoritmo ABC trabalhar como multi-objetivo.

4.2.1 Análise de Dominância de Pareto

A análise de adequação (*fitness*) das fontes de alimento foi substituída por uma análise de Dominância de Pareto das fontes de alimento. Os conceitos relacionados com Dominância de Pareto são bem adequados para uso com funções multi-objetivos concorrentes cuja relativa importância dos objetivos é desconhecida. Emprega-se o conceito de dominância de Pareto para comparar duas soluções factíveis do problema. Uma solução $x^{(1)}$ é dita dominar uma solução $x^{(2)}$ se ambas as condições a seguir forem satisfeitas:

- A solução $x^{(1)}$ não é pior que a solução $x^{(2)}$ em nenhum dos objetivos, ou seja, $f_m(x^{(1)}) \leq f_m(x^{(2)})$ para todo $m \in \{1,...,M\}$.
- A solução $x^{(1)}$ é estritamente melhor que a solução $x^{(2)}$ em pelo menos um objetivo, ou seja, $f_m(x^{(1)}) < f_m(x^{(2)})$ para algum $m \in \{1,...,M\}$.

Os valores $x^{(1)}$ e $x^{(2)}$ são diferentes configurações de cache cujas execuções em plataforma geram funções objetivo f_m, sendo m objetivos, tais como, desempenho e consumo de energia.

4.2.2 Cálculo da função de adequação (Fitness)

No algoritmo ABC original, as abelhas *onlooker* usam uma função de probabilidade para selecionar as melhores fontes de alimento a serem exploradas. Essa função de probabilidade é

obtida utilizando os valores da adequação(*fitness*) obtidos das funções objetivo de cada fonte de alimento. Com o objetivo de manter essa característica, a abordagem proposta utiliza uma técnica chamada de Soma Ponderada (*weighted Sum*)(NAIDU; MOKHLIS; BAKAR, 2014), na qual a distribuição de probabilidade multi-objetivo é baseada nos resultados de cada fonte quanto ao consumo de energia e desempenho simultaneamente. Então, a adequação(*fitness*) de cada fonte de alimento passou a ser calculada da seguinte forma:

$$fitness_i = \frac{1}{((Cycles_i \times W_{Cycle}) + (Energy_i \times W_{Energy}))}, \tag{4.1}$$

onde i é a fonte de alimento corrente em $\{1,2,\cdots,NP\}$, $W_{cycle} + W_{energy} = 1$, e $Cycles_i$ e $Energy_i$ são respectivamente os valores medidos de desempenho e consumo de energia para a fonte de alimento i. Os pesos W_{cycle} e W_{energy} foram definidos como 50% para consumo de energia e 50% para desempenho (ciclos da CPU), pois estamos considerando que ambos possuem a mesma importância para o resultado final da execução do algoritmo. Os valores de desempenho e de consumo de energia, antes de aplicados à equação 4.1, são normalizados (FISHER et al., 2003) para os valores mínimo e máximos de 0 a 10000, utilizando as equações 4.2 e 4.3, onde *Energia_Normal* e *Desempenho_Normal* são os valores de consumo de energia e desempenho normalizados. Os valores *Energia* e *Ciclos* são os valores de consumo de energia e desempenho não normalizados (obtidos na simulação). Os valores *Min_Energia* e *Min_Ciclos* são valores minímos veficados para consumo de energia e para desempenho. Os valores *Max_Energia* e *Max_Ciclos* são valores máximos veficados para consumo de energia e para desempenho. Os valores *Max_Normal* e *Min_Normal* são valores máximo e minímo usados para a normalização (usados como 0 e 10.000).

$$\begin{aligned} Energia\_Normal = ((Energia - Min\_Energia) * (Max\_Normal - Min\_Normal) \\ /(Max\_Energia - Min\_Energia)) + Min\_Normal, \end{aligned} \tag{4.2}$$

$$\begin{aligned} Desempenho\_Normal = ((Ciclos - Min\_Ciclos) * (Max\_Normal - Min\_Normal) \\ /(Max\_Ciclos - Min\_Ciclos)) + Min\_Normal, \end{aligned} \tag{4.3}$$

Como o problema busca minimizar os valores de consumo de energia e redução de ciclos de CPU, o valor de *fitness* é o inverso da função objetivo. Logo a função objetivo f será dada por:

$$f_i = ((Cycles_i * W_{Cycle}) + (Energy_i * W_{Energy})), \tag{4.4}$$

As equações 4.1 e 4.4 não são usadas para decisões multi-objetivo. Aqui elas são usadas apenas para obter a probabilidade de uma fonte de alimento ser selecionada, como descrito na equação 2.8. Para decisão se uma fonte de alimento i é mais adequada aos objetivos que outra

fonte de alimento *i*, o ABC multi-objetivo usa análise de dominância de Pareto (seção 4.2.1).

4.2.3 Critério de Parada Automático

O critério de parada do algoritmo ABC original é baseado em um número máximo de iterações. É bem difícil minimizar o tempo de execução do algoritmo mantendo um resultado de qualidade na exploração do espaço de projeto de cache, utilizando como critério de parada do algoritmo um número máximo de iterações. Isso ocorre porque, por um lado, se deseja que o número de iterações deva ser o menor possível buscando se otimizar o tempo de execução do algoritmo. Por outro lado, também se deseja que o número de iterações seja grande o suficiente para que o algoritmo explore bem o espaço de projeto e possa progressivamente convergir para um conjunto de configurações ótimas que atendam simultaneamente os dois objetivos. No algoritmo proposto baseado na técnica ABC multi-objetivo estamos utilizando um critério de parada baseado no Indicador de Taxa de Dominação Mútua (MDRI) proposta por Marti (MARTI et al., 2009). O MDRI baseia-se no conjunto de soluções não-dominadas de duas iterações sucessivas P_t^* e P_{t-1}^*. O indicador de progresso $I_{mdr}(t) \in [-1,1]$ compara quantos elementos não-dominados de uma iteração t domina os elementos não-dominados de uma iteração anterior $t-1$, e vice-versa. Para simplificar a descrição, considere que $\Delta(A,B)$ retorna o conjunto de elementos de A que são dominados por ao menos um elemento de B, e que $|C|$ é o número de elementos de C.

O indicador é calculado utilizando a seguinte equação:

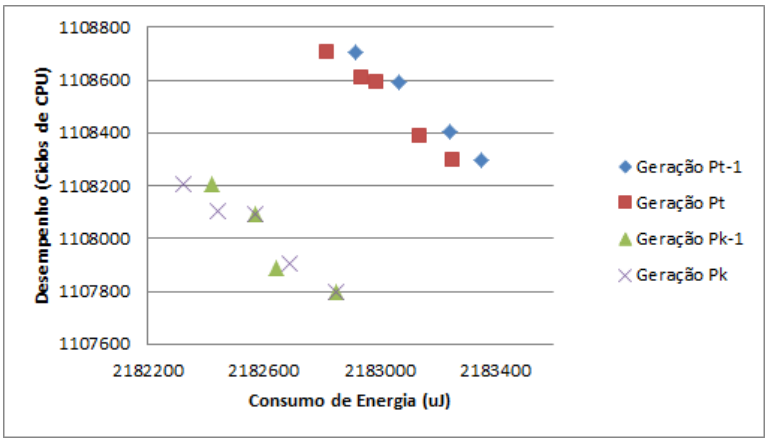
$$I_{mdr}(P_t^*, P_{t-1}^*) = \frac{|\Delta(P_{t-1}^*, P_t^*)|}{|P_{t-1}^*|} - \frac{|\Delta(P_t^*, P_{t-1}^*)|}{|P_t^*|}, \tag{4.5}$$

Em qualquer iteração, fontes de alimento não-dominadas são as fontes de alimento chamadas de *bestFoods*. Ao final de cada iteração será calculado o valor de I_{mdr} para se decidir se o algoritmo deve iniciar uma nova iteração ou parar sua execução. O indicador I_{mdr} gera diferentes tipos de informação. Se $I_{mdr} = 1$ significa que o resultado da iteração corrente t é amplamente melhor que o da iteração precedente $t-1$. Para $I_{mdr} = 0$ temos que a iteração corrente t não teve qualquer progresso. O pior caso, $I_{mdr} = -1$, indica que a iteração corrente tem o resultado pior que o da iteração anterior. Portanto, será feito monitoramento de fim de iteração acompanhando a evolução do I_{mdr} inicialmente próximo a 1 e decrescendo até o valor zero, quando o algoritmo inicia sua fase sem evoluir as melhorias. Com objetivo de melhor qualidade dos resultados, será utilizado como critério de parada o fato de três iterações sucessivas resultarem o $I_{mdr} = 0$, pois verificamos experimentalmente que após uma obtenção do $I_{mdr} = 0$, é possível (deixando o algoritmo continuar) que se consiga obter outros $1 > I_{mdr} > 0$. O mesmo ocorre após dois sucessivos $I_{mdr} = 0$. Porém, depois de três sucessivos $I_{mdr} = 0$, os valores de I_{mdr} seguintes serão sempre zero.

A figura 4.1 mostra dois exemplos de iterações sucessivas em que exemplificaremos o

uso do I_{mdr}. Primeiro analisaremos as iterações P_{t-1} e P_t. A iteração P_t (com 5 configurações de cache) é posterior à iteração P_{t-1} (com 4 configurações de cache). Temos que quatro configurações da iteração P_t dominam ao menos uma configuração da iteração P_{t-1}, e que nenhuma configuração da iteração P_{t-1} domina alguma configuração de P_t. Logo o Indicador de Taxa de Dominação Mútua é calculado da seguinte forma:

Figura 4.1: Exemplo de cálculo do Indicador de Taxa de Dominação Mútua.



$$I_{mdr}(P_t, P_{t-1}) = \frac{|\Delta(P_{t-1}, P_t)|}{|P_{t-1}|} - \frac{|\Delta(P_t, P_{t-1})|}{|P_t|} = \frac{4}{4} - \frac{0}{5} = 1$$

Verificamos que o $I_{mdr}(P_t, P_{t-1}) = 1$ indica que as fontes de alimento da iteração P_t são amplamente melhores que as da iteração precedente P_{t-1}. Analisando agora as fontes de alimento das iterações P_{k-1} e P_k, temos que a iteração P_k (com 5 configurações de cache) é posterior à iteração P_{k-1} (com 4 configurações de cache). Temos que apenas uma configuração da iteração P_k que domina ao menos uma configuração da iteração P_{k-1}, e que uma configuração da iteração P_{k-1} domina alguma configuração de P_k. Logo o Indicador de Taxa de Dominação Mútua é calculado da seguinte forma:

$$I_{mdr}(P_k, P_{k-1}) = \frac{|\Delta(P_{k-1}, P_k)|}{|P_{k-1}|} - \frac{|\Delta(P_k, P_{k-1})|}{|P_k|} = \frac{1}{4} - \frac{1}{5} = 0,05$$

Verificamos que o $I_{mdr}(P_k, P_{k-1}) = 0,05$ indica que a iteração P_k praticamente não teve progresso sobre a iteração precedente P_{k-1}. Ao executar o algoritmo ABC utilizando o Indicador de Taxa de Dominação Mútua como critério de parada automática, verifica-se que no final da primeira iteração o I_{mdr} é bem próximo de 1, e esse valor vai decrescendo (em média) com o final de cada nova iteração, até que o seu valor chega, de forma repetida, ao valor $I_{mdr} = 0$. Logo, o valor $I_{mdr} = 0,05$, nos dá a ideia de que o algoritmo está próximo de chegar ao Pareto ótimo.

4.3 O Algoritmo proposto AbcDE

Nessa seção descreveremos a abordagem de exploração de espaço de projeto de cache proposta neste trabalho e que denominamos de algoritmo AbcDE. O algoritmo proposto AbcDE é um algoritmo ABC multi-objetivo com melhorias, que é baseado no algoritmo ABC modificado descrito na seção anterior. As melhorias propostas são listadas a seguir:

- Otimização da busca global usando DoE;
    - Uso de população balanceada;
    - Uso de análise estatística do modelo de efeitos;
    - Cálculo do impacto dos níveis de cada parâmetro;
    - Seleção de fonte de alimento com base em impacto;

O pseudo-código do algoritmo 5 mostra o funcionamento do algoritmo AbcDE, onde P é a população a ser considerada pelo algoritmo, $BestFoods$ é o conjunto das melhores fontes de alimento (melhores configurações de cache) evoluídas pelo algoritmo, P_i é probabilidade de que uma fonte de alimento i possui para ser selecionada na fase das abelhas *OnLookers*, $Trials_i$ é o número de tentativas sem sucesso de busca de uma melhor fonte de alimento na vizinhança realizadas para uma fonte de alimento i, e $limit$ é o número máximo de tentativas sem sucesso que poderão ser realizadas para substituir uma fonte de alimento i por outra de sua vizinhança.

Na fase de inicialização alguns elementos são iniciados e são calculados os impactos de cada um dos parâmetros das caches utilizando a técnica de *Design of Experiments* (DoE) (passos 1-5). O cálculo do impacto dos parâmetros por uso de DoE é detalhado na subseção 4.3.2. São inicializados alguns elementos como as populações de abelhas, as fontes de alimento P (população balanceada), as fontes de alimento *bestFoods* (evoluídas pelo algoritmo como solução final) e as fontes de alimento *simulatedFoods* (usadas para diversos fins como detalhado nas subseções 4.3.2 e 5.2.1).

Os ciclos de execução do algoritmo AbcDE (passos 6-40) são realizados através de iterações sucessivas, até que o critério de parada (passo 40) seja alcançado e o algoritmo é finalizado. No início de cada iteração é realizado o cálculo do impacto de cada um dos diferentes níveis de cada parâmetro. Esse cálculo é detalhado na subseção 4.3.2. Após o cálculo do impacto dos níveis, seguem-se as fases das abelhas *Employed*, da abelhas *OnLookers* e das abelhas *Scout*.

Na fase das abelhas *employed* (passos 8-17), cada abelha *employed* busca na vizinhança de sua fonte de alimento i, uma fonte de alimento i' para ser analisada. A nova fonte de alimento i' é obtida através da equação 2.7 (passo 10). No passo 11, é obtida a função objetivo da nova fonte de alimento i'. Se a nova fonte de alimento i' domina a atual fonte de alimento i, a nova fonte de alimento i' substituirá a fonte de alimento atual i (passo 13). O conjunto *bestFoods* é atualizado se a nova fonte de alimento dominar alguma de suas fontes de alimento. Nesse caso

Algoritmo 5 Algoritmo AbcDE

```
1: Fase de Inicialização:
2:     Carrega as fontes de alimento do DoE (população balanceada)
3:     DoE - Calcula o impacto de cada um dos parâmetros
4:     Cria P e em BestFoods a partir das fonte de alimento do DoE
5: Fim
6: repeat
7:     Calcula o impacto de cada um dos níveis dos parâmetros
8:     Fase das Abelhas Employed
9:         for Cada fonte de alimento i em P do
10:            Seleciona uma nova fonte de alimento i' utilizando a equação 2.7
11:            Obtém a função objetivo para a nova fonte de alimento i'
12:            if i' dominar i then
13:                i' é colocada na população P substituindo i
14:                Ordenar BestFoods por dominância
15:                if i' dominar alguma fonte de alimento em BestFoods then
16:                    Substituir a última fonte de alimento em BestFoods por i'
17:    Fim
18:    Calcula a probabilidade P_i de cada elemento i em P ser selecionado
19:    Fase das Abelhas OnLookers
20:        for Cada fonte de alimento i em P do
21:            r ← random() (É atribuído a r um valor randômico)
22:            if P_i >= r then
23:                Seleciona uma nova fonte de alimento i' com base em impacto de parâmetro e de nível
24:                Obtém a a função objetivo para a nova fonte de alimento i'
25:                if i' dominar i then
26:                    i' é colocada na população P substituindo i
27:                    if i' dominar alguma fonte de alimento em BestFoods then
28:                        Substituir a última fonte de alimento em BestFoods por i'
29:    Fim
30:    Fase das Abelhas Scout
31:        for Cada fonte de alimento i em P do
32:            if Trials_i >= Limit then
33:                Seleciona randomicamente uma nova fonte de alimento i'
34:                Substitui i por i'
35:                Trials_i' = 0
36:                Ordenar BestFoods por dominância
37:                if i' dominar alguma fonte de alimento em BestFoods then
38:                    Substituir a última fonte de alimento em BestFoods por i'
39:    Fim
40: until Critério de parada ser atingido
```

o *bestFoods* é ordenado por dominância e sua última fonte de alimento é substituída pela nova fonte de alimento i' (passos 15-16).

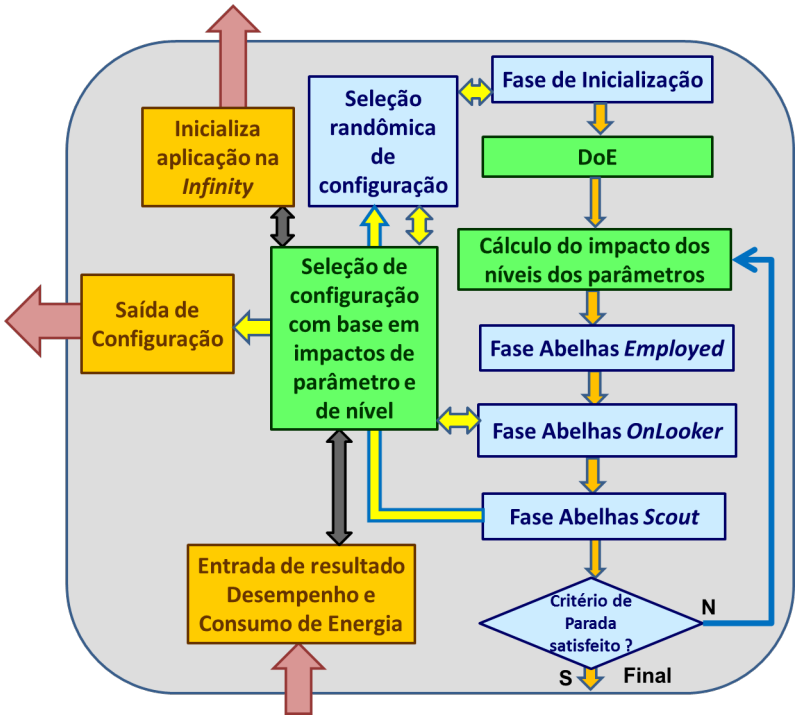
A fase da abelhas *onlookers*, procede de forma semelhante à fase das abelhas *employed* (passos 19-29), porém as abelhas *onlookers* são selecionadas conforme o valor de probabilidade P_i de seleção de suas fontes de alimento, e a fonte de alimento candidata i' é obtida randomicamente com base no impacto de cada parâmetro e no impacto dos níveis de cada parâmetro (passos

21-23). Essa obtenção da fonte de alimento *i* é detalhada na subseção 4.3.2.

Depois, as abelhas *scout* verificam se existe alguma fonte de alimento em *P* excedendo o limite de exploração sem sucesso. Caso o limite de exploração de uma fonte de alimento *i* exceda o valor da variável *limit* (descrita anteriormente para o algoritmo 5) ela será substituída por uma nova fonte de alimento gerada randomicamente considerando todo o espaço de exploração. Os passos 6-40 são repetidos até que o critério de parada seja atingido.

O diagrama de blocos do AbcDE, já considerando as suas melhorias, é ilustrado conforme a figura 4.2.

Figura 4.2: Diagrama em blocos do AbcDE.



Os blocos "*Inicializa aplicação na Infinity*", "*Saída de Configuração*" e "*Entrada de Resultado de Desempenho e Consumo de Energia*" são blocos que fazem apenas comunicação com a plataforma MPSoC, respectivamente iniciando a simulação de uma aplicação na plataforma, preparando a plataforma com a configuração de cache desejada pelo algoritmo, e buscando (no fim da simulação) os resultados de consumo de energia e desempenho obtidos para a referida configuração de cache. Eles não são blocos de melhorias introduzidas. Os blocos "*DoE*", "*Cálculo do impacto dos níveis dos parâmetros*", e o "*Seleção de Configuração com base em Impactos de parâmetro e de nível*" foram os blocos que foram introduzidos ao AbcDE para melhoria de sua eficiência na busca de melhores soluções na vizinhança e para que o algoritmo seja iniciado com uma população balanceada e bem distribuída no espaço de projeto. Os blocos restantes são blocos originais do algoritmo ABC multi-objetivo.

Nas subseções seguintes descreveremos como foram definidos os valores dos parâmetros do algoritmo ABC a serem utilizados na execução do algoritmo AbcDE, as mudanças introduzidas para criação do algoritmo AbcDE, e os modelos de energia utilizados para obtenção dos

valores de consumo de energia das caches na plataforma MPSoC utilizada nos experimentos.

4.3.1 Definição dos parâmetros do Algoritmo AbcDE

Os parâmetros de configuração do algoritmo ABC são: a- População de abelhas e Número de fontes de alimento, b- Limite de tentativas (*Limit*), c- Número máximo de iterações, e c- Total de execuções do algoritmo para cada aplicação. No caso dos algoritmos ABC multi-objetivo e AbcDE, é utilizado um critério de parada automático que torna sem efeito a definição de um valor para o Número máximo de iterações. Mostraremos a seguir como foram definidos os valores dos demais parâmetros para uso no algoritmo proposto AbcDE.

4.3.1.1 Definição do Número de Fontes de Alimento e População de abelhas

O algoritmo ABC original propõe que o número de abelhas seja o dobro do número de fontes de alimento utilizados no algoritmo. Logo, definindo um deles, o outro já estará definido. Utilizamos o teste de *Kruskal-Wallis* (vide Seção 2.5) para verificar se, para o algoritmo AbcDE existe diferença relevante na escolha entre três diferentes valores de número de fontes de alimento. Considerando o espaço de projeto, que é de 3^{13} diferentes configurações de cache (13 parâmetros de cache com 3 possíveis valores), e que a população inicial deve ser balanceada e com quantidade de 3^{k-p} (vide Seção 2.4.1), propomos testar três diferentes quantidades de fontes de alimento: 81, 243, e 729. Realizamos o teste de *Kruskal-Wallis* para as aplicações Matrix, Radix e FFT (do benchmark Splash2), simulando cada uma com os três diferentes números de fontes de alimento, e utilizando como base de análise a métrica de número de simulações da plataforma.

Fixando o nível de significância $\alpha = 0,05$ e sabendo que $k = 3$, temos que o valor crítico correspondente ao ponto Q é igual a $Q_{0,95} = 7,815$. Para a amostra da tabela 4.1 obtivemos um H= 25,814 > $Q_{0,95} = 7,815$, portanto aceitamos a hipótese não nula neste estudo. Verificamos que o teste de Kruskal-Wallis apontou que os diferentes valores de fontes de alimento possuem diferenças críticas nos resultados para a execução aplicação Matrix. Portanto, analisando os 10 resultados ordenados das simulações da aplicação Matrix, vemos que todos os resultados foram melhores em número de execuções da plataforma para a quantidade de 81 fontes de alimento.

Vemos através da tabela 4.2 que o teste de Kruskal-Wallis apontou para que os diferentes valores de fontes de alimento também possuem diferenças críticas nos resultados para a execução aplicação Radix, pois obtivemos um H= 25,811 > $Q_{0,95} = 7,8154$. Em todas as simulações o número de fontes de alimento 81 foi melhor.

A tabela 4.3 também nos mostra que o teste de Kruskal-Wallis apontou que os diferentes valores de fontes de alimento geram diferenças críticas nos resultados para a execução aplicação FFT, pois obtivemos um H= 25,56 > $Q_{0,95} = 7,815$. No FFT, todas as fontes de alimento 81 foram melhores em termos da métrica número de simulações.

Portanto concluímos que os diferentes tamanhos geram diferenças estatísticas relevantes

Tabela 4.1: Teste de Kruskal-Wallis para números de fontes de alimento usando a aplicação Matrix.

	Matrix			
	81	**243**	**729**	**Vencedor**
1	1352	4541	16644	81
2	1507	4595	16734	81
3	1527	5036	17981	81
4	1663	5542	18185	81
5	1674	5964	19787	81
6	1679	7554	21212	81
7	2156	8089	22610	81
8	2311	8351	22778	81
9	2546	8441	26886	81
10	2648	8484	26949	81
Média	1906	6660	20977	
Desvio Padrão	394	1489	3108	
H	25,814			
$Q_{0,95}$	7,815			
H_0	H >$Q_{0,95}$. Os diferentes valores possuem diferenças críticas.			

Tabela 4.2: Teste de Kruskal-Wallis para números de fontes de alimento usando a aplicação Radix.

	Radix			
	81	**243**	**729**	**Vencedor**
1	710	5095	24230	81
2	1722	5177	28849	81
3	1730	5611	28866	81
4	1746	5638	34505	81
5	1762	6085	43236	81
6	1895	7514	48763	81
7	2049	7547	53675	81
8	2127	7574	56604	81
9	2195	8466	57479	81
10	3185	9478	57874	81
Média	1912	6819	43408	
Desvio Padrão	570	1331	11781	
H	25,811			
$Q_{0,95}$	7,815			
H_0	H >$Q_{0,95}$. Os diferentes valores possuem diferenças críticas.			

Tabela 4.3: Teste de Kruskal-Wallis para números de fontes de alimento usando a aplicação FFT.

	FFT			
	81	**243**	**729**	**Vencedor**
1	2088	3612	48465	81
2	2799	6431	49802	81
3	2945	6873	50006	81
4	3007	7397	51608	81
5	3053	7571	51608	81
6	3066	7891	53033	81
7	3071	8305	54431	81
8	3141	8314	54599	81
9	3573	8819	58707	81
10	4039	9237	78770	81
Média	3084	7445	55103	
Desvio Padrão	461	1448	8293	
H	25,56			
$Q_{0,95}$	7,815			
H_0	H > $Q_{0,95}$. Os diferentes valores possuem diferenças críticas.			

e que a opção que gera o melhor resultado para a métrica de número de execuções da plataforma é a opção de 81 fontes de alimento. Usaremos nas execuções do algoritmo AbcDE a quantidade de 81 fontes de alimento e 162 abelhas (2x fontes de alimento).

4.3.1.2 Definição do Limite

Nesta seção descrevemos como chegamos no valor do parâmetro *Limit* da configuração do algoritmo AbcDE. Mais uma vez utilizamos o teste de *Kruskal-Wallis* para verificar se existe diferença relevante na escolha entre três diferentes valores para o parâmetro *Limit*. Propomos testar três diferentes valores para o referido parâmetro: 13, 15, e 17. Esses três valores foram escolhidos, pois utilizando experimentalmente valores menores que 9 e maiores que 20, o algoritmo demorou bastante na sua convergência em direção ao Pareto Ótimo. Realizamos o teste de *Kruskal-Wallis* para as aplicações Matrix, Radix e FFT, simulando cada uma com os três diferentes valores do parâmetro *Limit*, e utilizando como base de análise a métrica de número de simulações da plataforma.

Para a aplicação Matrix, o teste de *Kruskal-Wallis* nos mostra que não existe diferença crítica no uso dos valores de limite 13, 15 ou 17. Neste caso, o valor de limite 15 obteve os melhores resultados, apesar de que o limite 13 foi vencedor em 4 das 10 das execuções ordenadas. A tabela 4.4 mostra esses resultados.

Para a aplicação Radix, o teste de *Kruskal-Wallis* deixa claro que existe diferença crítica no uso dos valores de limite 13, 15 ou 17. Neste caso, o valor de limite melhor em todos os casos

Tabela 4.4: Teste de Kruskal-Wallis para o parâmetro Limite o usando a aplicação Matrix.

	Matrix			
	13	**15**	**17**	**Vencedor**
1	3439	2814	2821	15
2	3880	2946	3254	15
3	3881	2975	3787	15
4	4436	3803	3806	15
5	4545	3811	5052	15
6	5347	5927	5522	13
7	5500	6288	6041	13
8	6278	6773	6473	13
9	7502	7012	6981	17
10	7579	7796	7681	13
Média	5238	5014	5142	
Desvio Padrão	1212	1717	1475	
H	0,217			
$Q_{0,95}$	7,815			
H_0	H < $Q_{0,95}$. Os diferentes valores não possuem diferenças críticas.			

foi o limite 13. A tabela 4.5 mostra esses resultados.

Para a aplicação FFT, o teste de *Kruskal-Wallis* nos mostra mais uma vez que não existe diferença crítica no uso dos valores de limite 13, 15 ou 17. Neste caso, o valor de limite 13 obteve melhores resultados em 8 das 10 amostras. A tabela 4.6 mostra esses resultados.

Dessa forma, consideramos que o uso do parâmetro limite com o valor 13 responde melhor que os demais valores na maioria das aplicações em que realizamos os testes.

4.3.1.3 Definição do Total de Execuções do Algoritmo para cada aplicação

Para definir o valor do parâmetro "Total de execuções do algoritmo" para cada aplicação utilizamos a técnica de determinação do tamanho da amostra (vide seção 2.7). Definimos um intervalo de confiança de 90% e um erro de 10% relativo ao valor médio. Para cada aplicação realizamos uma certa quantidade de execuções (30) do algoritmo AbcDE e listamos os resultados em número de simulações da plataforma. O número de execuções do algoritmo precisava ser igual ou maior ao tamanho da amostra obtida. Na tabela 4.3 verificamos que o maior tamanho da amostra obtida foi 29,80, por isso não foi necessário execuções adicionais do algoritmo AbcDE para nenhuma das aplicações. Os resultados estão listados na figura 4.3.

Portanto o total de execuções do algoritmo para cada aplicação ficou definido conforme a tabela 4.7.

Tabela 4.5: Teste de Kruskal-Wallis para o parâmetro Limite o usando a aplicação Radix.

	Radix			
	13	**15**	**17**	**Vencedor**
1	2160	3395	4603	13
2	2808	3745	4645	13
3	3084	4107	5086	13
4	3128	4327	5499	13
5	3661	4803	5908	13
6	3703	4917	6087	13
7	3995	5506	6991	13
8	4151	5985	7782	13
9	4832	6661	8446	13
10	5698	7805	9886	13
Média	3722	5125	6493	
Desvio Padrão	906	1216	1545	
H	13,133			
$Q_{0,95}$	7,815			
H_0	H > $Q_{0,95}$. Os diferentes valores possuem diferenças críticas.			

Tabela 4.6: Teste de Kruskal-Wallis para o parâmetro Limite o usando a aplicação FFT.

	FFT			
	13	**15**	**17**	**Vencedor**
1	3296	3371	3411	13
2	3324	3512	3686	13
3	4104	4129	4131	13
4	4163	4177	4164	13
5	6251	5464	4663	17
6	6347	5568	4766	17
7	6354	6738	7095	13
8	6477	6853	7215	13
9	7291	7539	7765	13
10	7372	7747	8096	13
Média	5498	5510	5499	
Desvio Padrão	1407	1434	1574	
H	0,217			
$Q_{0,95}$	7,815			
H_0	H < $Q_{0,95}$. Os diferentes valores não possuem diferenças críticas.			

Figura 4.3: Definição do total de execuções do Algoritmo AbcDE através da análise do Tamanho da Amostra, utilizando a métrica de número de simulações da plataforma.

	Matrix	Radix	FFT	DiJkstra	Sha	Stringsearch	Basicmath
1	2.512	1.810	3.086	2.635	1.658	2.047	3.278
2	2.011	2.195	2.975	1.666	1.878	1.865	2.005
3	1.358	1.730	3.172	2.864	3.178	2.918	1.672
4	1.677	1.762	2.988	2.295	1.970	2.045	1.685
5	1.532	1.895	3.077	2.317	2.369	2.877	1.602
6	1.345	1.614	2.971	1.808	1.689	1.688	2.811
7	1.540	1.929	3.074	2.762	1.593	1.202	1.896
8	1.500	1.856	2.929	1.978	2.877	1.516	1.695
9	1.370	2.042	3.010	2.468	2.425	1.590	1.849
10	1.513	1.785	3.255	3.123	2.163	1.569	1.683
11	2.648	1.746	2.945	2.646	1.901	1.727	2.163
12	1.679	2.049	2.799	1.678	2.040	1.352	1.353
13	1.663	1.722	3.053	2.877	1.619	1.809	1.826
14	2.546	3.185	3.141	2.306	1.513	2.141	2.656
15	1.352	2.127	3.066	2.324	4.208	2.070	1.208
16	1.507	710	4.093	1.821	1.846	1.666	2.147
17	2.311	2.195	3.007	2.773	3.293	2.107	1.369
18	1.674	1.730	3.071	1.983	2.699	1.986	1.793
19	2.156	1.762	2.088	2.481	2.308	2.969	1.718
20	1.527	1.895	3.573	3.134	2.948	2.423	1.586
21	1.508	1.614	3.072	2.623	2.581	2.415	2.827
22	1.203	929	1.776	1.653	1.668	2.150	2.623
23	1.509	1.856	2.945	2.850	1.970	3.031	3.311
24	1.369	2.042	2.999	2.280	2.583	2.736	1.989
25	1.386	2.185	3.253	2.304	1.855	1.340	1.688
26	1.534	1.901	3.341	1.794	2.736	1.943	1863
27	1.334	1.647	2.866	2.750	1.770	2.608	1728
28	1.531	1.765	5.093	1.961	1.985	1.344	1833
29	1.498	2.395	3.071	2.454	1.863	1.958	1699
30	1.681	1.914	3.471	3.111	2.077	2.608	1175
Média	**1666**	**1866**	**3109**	**2391**	**2242**	**2.057**	**1958**
Desvio Padrão	380,72	408,81	532,55	449,21	507,08	518,55	543,34
Tamanho da Amostra	20,21	18,57	11,35	13,66	19,79	24,60	29,80

Tabela 4.7: Total de execuções do algoritmo para cada aplicação.

Aplicação	Número de Execuções do Algoritmo
Matrix	21
FFT	12
Radix	19
Djisktra	14
Sha	20
StringSearch	25
Basicmath	30

4.3.2 Otimização da Busca Global usando DoE

Conforme descrito na seção 1.1, o AbcDE inclui o uso da técnica de DoE para encontrar as melhores configurações de cache para otimização de consumo de energia e desempenho. Esta característica resultou em uma melhora do tempo médio de execução do algoritmo, como será visto posteriormente.

As técnicas baseadas no algoritmo ABC com otimização multi-objetivo têm como vantagens a precisão dos resultados simulando diretamente a execução da plataforma alvo, a característica de evitar resultados em mínimo local e a simplicidade de sua implementação e uso; no entanto, estas técnicas possuem como desvantagem um maior tempo de execução em comparação com outras abordagens que usam técnicas de *trace-driven*, pois as de trace-driven obtém as estatísticas de cache sem simular a plataforma e sim através de análise do trace de acesso à memória, que é bem mais rápido. A redução do tempo de execução do algoritmo foi conseguida através da introdução de mecanismos para melhorar a busca global do algoritmo que consideram características específicas do problema de exploração de cache. O problema de cache inclui duas importantes características para uso do algoritmo: 1- valores discretos dos parâmetros de cache e 2- impacto dos parâmetros com valores bastante diferentes (alguns parâmetros de cache com alto e outros com baixo impacto quanto a os valores de desempenho de consumo de energia que eles geram).

É possível usar uma população balanceada e calcular o impacto dos parâmetros já que os valores dos parâmetros são discretos. A técnica de DoE foi aplicada neste caso. A distância dos valores dos impactos dos diversos parâmetros pode ser usada para se trabalhar as novas fontes de alimento priorizando os parâmetros de mais alto impacto. O uso de uma população balanceada é importante para se obter, na fase inicial do algoritmo, configurações de cache bem distribuídas no espaço de projeto com uso de todos os níveis de cada parâmetro de forma uniforme. Mais adiante descreveremos um exemplo de população balanceada. O uso de uma população balanceada também é importante para realizar a análise de impacto dos parâmetros da cache.

Para se realizar a análise de impacto de cada um dos parâmetros da cache em uma plataforma MPSoC utilizamos a técnica de Design of Experiments (DoE) chamada de soma dos

quadrados dos principais efeitos (*sum of squares -SS*). Nas próximas subseções descreveremos como fizemos uso de população balanceada usando DoE, como realizamos a análise do modelo de efeitos, e como realizamos a seleção das fontes de alimento com uso dos impactos dos parâmetros de cache e de seus respectivos níveis.

4.3.2.1 Uso de População Balanceada

Como verificado na seção 4.3.1.1, foram escolhidas 81 fontes de alimento. Neste trabalho estaremos utilizando uma carga balanceada para o projeto de 3^{k-p} fatorial fracionário, onde cada parâmetro (dimensão) do problema possui 3 níveis (possíveis valores para um parâmetro), $k = 13$ parâmetros para 4 processadores, e p é o fator de redução do espaço de projeto a ser analisado. Para $p = 1$ o espaço de projeto simulado será a metade do total. Para $p = 2$ o espaço de projeto será um quarto do total, e assim por diante. Para quatro processadores (3^{13}), utilizaremos o fator de redução $p = 9$ ($3^{13-9} = 3^4$) reseltando em uma população balanceada de 81 diferentes configurações balanceadas.

As populações iniciais no algoritmo ABC original são geradas randomicamente. Para o algoritmo AbcDE, como usaremos uma população balanceada, conseguiremos realizar a geração randômica da população balanceada da seguinte forma: primeiro definimos uma matriz $13x81$ (tabela 4.8) que contém 13 colunas (13 parâmetros) e 81 diferentes configurações. Como o balanceamento existe para quaisquer da 13 colunas, iniciamos em cada execução do algoritmo AbcDE (na fase de inicialização) uma ordenação aleatória dessas colunas. Dessa forma obtemos mais de 6 bilhões (13!) possíveis configurações balanceadas a serem selecionadas randomicamente para cada execução do algoritmo.

4.3.2.2 Uso da análise estatística do modelo de efeitos

Utilizando a análise estatística do modelo de efeitos do *design of experiments* conseguimos definir o impacto de cada um dos parâmetros de cache envolvidos nos experimentos. Explicaremos o funcionamento utilizando um exemplo para cache. Considerando um sistema de cache com apenas dois parâmetros A e B, o impacto do parâmetro A é calculado da seguinte forma (MONTGOMERY, 2013):

$$SS_A = \frac{1}{b}\sum_{i=1}^{a} y_{i.}^2 - \frac{y_{..}^2}{ab}, \tag{4.6}$$

onde $y_{i.} = \sum_{j=1}^{b} y_{ij}$, os efeitos A e B possuem a e b níveis (respectivamente), $i = 1,2,...,a$, $j = 1,2,...,b$ e $y_{..} = \sum_{i=1}^{a}\sum_{j=1}^{b} y_{ij}$. O Y_{ij} corresponde ao valor do *fitness* de uma configuração de cache da população balanceada. Um exemplo desse cálculo pode ser encontrado na subseção 2.4. De forma semelhante também calculamos o efeito do parâmetro B como:

$$SS_B = \frac{1}{a}\sum_{j=1}^{b} y_{.j}^2 - \frac{y_{..}^2}{ab}, \tag{4.7}$$

Tabela 4.8: Matriz utilizada para geração de população balanceada de 81 fontes de alimento.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
1	0	0	0	1	0	1	0	1	0	-1	-1	-1	-1
2	-1	0	0	-1	1	1	0	1	0	0	-1	-1	-1
3	1	0	0	0	-1	1	0	1	0	1	-1	-1	-1
4	0	-1	0	-1	1	-1	1	1	0	-1	0	-1	-1
5	-1	-1	0	0	-1	-1	1	1	0	0	0	-1	-1
6	1	-1	0	1	0	-1	1	1	0	1	0	-1	-1
7	0	1	0	0	-1	0	-1	1	0	-1	1	-1	-1
8	-1	1	0	1	0	0	-1	1	0	0	1	-1	-1
9	1	1	0	-1	1	0	-1	1	0	1	1	-1	-1
10	0	0	-1	1	0	-1	1	-1	1	-1	-1	0	-1
11	-1	0	-1	-1	1	-1	1	-1	1	0	-1	0	-1
12	1	0	-1	0	-1	-1	1	-1	1	1	-1	0	-1
13	0	-1	-1	-1	1	0	-1	-1	1	-1	0	0	-1
14	-1	-1	-1	0	-1	0	-1	-1	1	0	0	0	-1
15	1	-1	-1	1	0	0	-1	-1	1	1	0	0	-1
16	0	1	-1	0	-1	1	0	-1	1	-1	1	0	-1
17	-1	1	-1	1	0	1	0	-1	1	0	1	0	-1
18	1	1	-1	-1	1	1	0	-1	1	1	1	0	-1
19	0	0	1	1	0	0	-1	0	-1	-1	-1	1	-1
20	-1	0	1	-1	1	0	-1	0	-1	0	-1	1	-1
21	1	0	1	0	-1	0	-1	0	-1	1	-1	1	-1
22	0	-1	1	-1	1	1	0	0	-1	-1	0	1	-1
23	-1	-1	1	0	-1	1	0	0	-1	0	0	1	-1
24	1	-1	1	1	0	1	0	0	-1	1	0	1	-1
25	0	1	1	0	-1	-1	1	0	-1	-1	1	1	-1
26	-1	1	1	1	0	-1	1	0	-1	0	1	1	-1
27	1	1	1	-1	1	-1	1	0	-1	1	1	1	-1
28	0	0	1	1	0	1	0	-1	1	-1	-1	-1	0
29	-1	0	1	-1	1	1	0	-1	1	0	-1	-1	0
30	1	0	1	0	-1	1	0	-1	1	1	-1	-1	0
31	0	-1	1	-1	1	-1	1	-1	1	-1	0	-1	0
32	-1	-1	1	0	-1	-1	1	-1	1	0	0	-1	0
33	1	-1	1	1	0	-1	1	-1	1	1	0	-1	0
34	0	1	1	0	-1	0	-1	-1	1	-1	1	-1	0
35	-1	1	1	1	0	0	-1	-1	1	0	1	-1	0
36	1	1	1	-1	1	0	-1	-1	1	1	1	-1	0
37	0	0	0	1	0	-1	1	0	-1	-1	-1	0	0
38	-1	0	0	-1	1	-1	1	0	-1	0	-1	0	0
39	1	0	0	0	-1	-1	1	0	-1	1	-1	0	0
40	0	-1	0	-1	1	0	-1	0	-1	-1	0	0	0
41	-1	-1	0	0	-1	0	-1	0	-1	0	0	0	0
42	1	-1	0	1	0	0	-1	0	-1	1	0	0	0
43	0	1	0	0	-1	1	0	0	-1	-1	1	0	0
44	-1	1	0	1	0	1	0	0	-1	0	1	0	0
45	1	1	0	-1	1	1	0	0	-1	1	1	0	0
46	0	0	-1	1	0	0	-1	1	0	-1	-1	1	0
47	-1	0	-1	-1	1	0	-1	1	0	0	-1	1	0
48	1	0	-1	0	-1	0	-1	1	0	1	-1	1	0
49	0	-1	-1	-1	1	1	0	1	0	-1	0	1	0
50	-1	-1	-1	0	-1	1	0	1	0	0	0	1	0
51	1	-1	-1	1	0	1	0	1	0	1	0	1	0
52	0	1	-1	0	-1	-1	1	1	0	-1	1	1	0
53	-1	1	-1	1	0	-1	1	1	0	0	1	1	0
54	1	1	-1	-1	1	-1	1	1	0	1	1	1	0
55	0	0	-1	1	0	1	0	0	-1	-1	-1	-1	1
56	-1	0	-1	-1	1	1	0	0	-1	0	-1	-1	1
57	1	0	-1	0	-1	1	0	0	-1	1	-1	-1	1
58	0	-1	-1	-1	1	-1	1	0	-1	-1	0	-1	1
59	-1	-1	-1	0	-1	-1	1	0	-1	0	0	-1	1
60	1	-1	-1	1	0	-1	1	0	-1	1	0	-1	1
61	0	1	-1	0	-1	0	-1	0	-1	-1	1	-1	1
62	-1	1	-1	1	0	0	-1	0	-1	0	1	-1	1
63	1	1	-1	-1	1	0	-1	0	-1	1	1	-1	1
64	0	0	1	1	0	-1	1	1	0	-1	-1	0	1
65	-1	0	1	-1	1	-1	1	1	0	0	-1	0	1
66	1	0	1	0	-1	-1	1	1	0	1	-1	0	1
67	0	-1	1	-1	1	0	-1	1	0	-1	0	0	1
68	-1	-1	1	0	-1	0	-1	1	0	0	0	0	1
69	1	-1	1	1	0	0	-1	1	0	1	0	0	1
70	0	1	1	0	-1	1	0	1	0	-1	1	0	1
71	-1	1	1	1	0	1	0	1	0	0	1	0	1
72	1	1	1	-1	1	1	0	1	0	1	1	0	1
73	0	0	0	1	0	0	-1	-1	1	-1	-1	1	1
74	-1	0	0	-1	1	0	-1	-1	1	0	-1	1	1
75	1	0	0	0	-1	0	-1	-1	1	1	-1	1	1
76	0	-1	0	-1	1	1	0	-1	1	-1	0	1	1
77	-1	-1	0	0	-1	1	0	-1	1	0	0	1	1
78	1	-1	0	1	0	1	0	-1	1	1	0	1	1
79	0	1	0	0	-1	-1	1	-1	1	-1	1	1	1
80	-1	1	0	1	0	-1	1	-1	1	0	1	1	1
81	1	1	0	-1	1	-1	1	-1	1	1	1	1	1

O cálculo do impacto dos parâmetros é feita no bloco "*DoE*" (figura 4.2). No exemplo consideramos que a hierarquia de cache possui 9 parâmetros ($\{a,b,c,...,i\}$), portanto as equações a serem utilizadas são descritas abaixo:

$$SS_A = (\frac{1}{bcdefghi}\sum_{j=1}^{a} y_{j.....}^2 - \frac{y_{.....}^2}{abcdefghi}), \tag{4.8}$$

$$SS_B = (\frac{1}{acdefghi}\sum_{k=1}^{b} y_{.k.....}^2 - \frac{y_{.....}^2}{abcdefghi}), \tag{4.9}$$

...

$$SS_I = (\frac{1}{abcdefgh}\sum_{r=1}^{i} y_{.....r}^2 - \frac{y_{.....}^2}{abcdefghi}), \tag{4.10}$$

onde $y_{j.....} = \sum_{k=1}^{b}\sum_{l=1}^{c}\sum_{m=1}^{d}\sum_{n=1}^{e}\sum_{o=1}^{f}\sum_{p=1}^{g}\sum_{q=1}^{h}\sum_{r=1}^{i}\sum_{s=1}^{n} y_{jklmnopqrs}$,
$y_{.k.....} = \sum_{j=1}^{a}\sum_{l=1}^{c}\sum_{m=1}^{d}\sum_{n=1}^{e}\sum_{o=1}^{f}\sum_{p=1}^{g}\sum_{q=1}^{h}\sum_{r=1}^{i} y_{jklmnopqrs}$ os efeitos A, B até I possuem respectivamente a, b até i níveis (respectivamente), $j = 1,2,...,a$, $k = 1,2,...,b$ e $r = 1,2,...,i$, e $y_{.....} = \sum_{j=1}^{a}\sum_{k=1}^{b}\sum_{l=1}^{c}\sum_{m=1}^{d}\sum_{n=1}^{e}\sum_{o=1}^{f}\sum_{p=1}^{g}\sum_{q=1}^{h}\sum_{r=1}^{i} y_{jklmnopqrs}$.

Os resultados do impacto realizados no bloco "*DoE*" são obtidos na fase de inicialização do algoritmo, e serão utilizados em todos os ciclos na execução do algoritmo AbcDE.

Dada uma configuração de hierarquia de cache, quando um parâmetro de alto impacto foi selecionado e mudamos seu valor, verificamos um forte impacto no resultado dos objetivos, fazendo aumentar ou diminuir o seu valor dependendo do valor selecionado para o parâmetro. Verificamos isso anteriormente através do exemplo na subseção 2.4. Portanto, é importante se estudar o impacto dos níveis de cada parâmetro para se selecionar o valor dos parâmetros de alto impacto de forma a otimizar os objetivos.

4.3.2.3 Cálculo do impacto dos níveis de cada parâmetro de cache

O cálculo do grau de impacto dos níveis de cada parâmetro é realizado no bloco "*Cálculo do impacto dos níveis dos parâmetros*" (figura 4.2). Quando iniciamos a população dos *SimulatedFoods* com uma população balanceada do DoE, temos algum resultado semelhante ao mostrado na figura 4.4. Nessa figura verificamos que os itens do *SimulatedFoods* estão ordenados por dominância. Os valores de menor energia e menor quantidade de ciclos de CPU estão no início da lista. O *fitness* de cada linha foi calculado através da equação 4.1. As colunas de P_0 a P_8 representam os parâmetros de cache de uma plataforma com 4 processadores. Podemos verificar que o parâmetro P_8 possui um forte impacto, pois os seus valores relativos aos níveis 3 e 4 definem claramente os menores e maiores valores, respectivamente, de *fitness* (vide figura 4.4 nas 9 primeiras e 9 últimas linhas). Já para o parâmetro P_1, a figura 4.4 não mostra o impacto dos níveis com a mesma clareza. O parâmetro P_1 tem seus níveis (0, 1 e 2) bem distribuídos quanto a valores maiores e menores de *fitness*, e não fica claro que seu nível 2 é o nível de maior

impacto para altos valores de *fitness*, e o de menor impacto é o seu nível 0.

Figura 4.4: População Balanceada de configurações de cache (*SimulatedFoods*).

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	Energia(µJ)	Desempenho (Ciclos CPU)	Fitness
12	2	14	0	12	2	13	1	3	2181922	1108706	0,00000060779
12	2	12	1	14	1	14	0	3	2181924	1108706	0,00000060779
14	1	13	2	12	2	12	1	3	2181927	1108706	0,00000060779
12	2	13	2	13	0	12	2	3	2182527	1108706	0,00000060767
14	1	12	1	13	0	14	2	3	2182534	1108706	0,00000060767
14	1	14	0	14	1	13	0	3	2184139	1108814	0,00000060736
13	0	12	1	12	2	14	1	3	2203892	1113890	0,00000060281
13	0	14	0	13	0	13	2	3	2204498	1113890	0,00000060270
13	0	13	2	14	1	12	0	3	2214800	1113578	0,00000060089
13	1	14	2	14	2	14	2	2	2862046	1176578	0,00000049522
14	2	13	1	13	1	13	1	2	2862046	1176578	0,00000049522
14	2	14	2	12	0	14	0	2	2866091	1176110	0,00000049478
13	1	13	1	12	0	13	0	2	2868444	1176214	0,00000049448
13	1	12	0	13	1	12	1	2	2869799	1175894	0,00000049435
14	2	12	0	14	2	12	2	2	2869804	1175894	0,00000049435
12	0	13	1	14	2	13	2	2	3042150	1221088	0,00000046913
12	0	14	2	13	1	14	1	2	3042152	1221088	0,00000046913
12	0	12	0	12	0	12	0	2	3070335	1219228	0,00000046625
12	1	13	0	12	1	14	2	4	8738380	1612812	0,00000019321
13	2	14	1	12	1	12	2	4	8738384	1612812	0,00000019321
12	1	14	1	14	0	12	1	4	8738385	1612812	0,00000019321
13	2	13	0	13	2	14	0	4	8738386	1612812	0,00000019321
13	2	12	2	14	0	13	1	4	8738386	1612812	0,00000019321
14	0	12	2	12	1	13	2	4	8738389	1612812	0,00000019321
14	0	13	0	14	0	14	1	4	8738397	1612812	0,00000019321
12	1	12	2	13	2	13	0	4	8748112	1613274	0,00000019302
14	0	14	1	13	2	12	0	4	8787106	1615225	0,00000019226

A figura 4.5 mostra o conjunto *SimulatedFoods* da figura 4.4 depois de evoluída duas iterações do algoritmo ABC. Verificamos que o parâmetro P_8 ficou com o seu nível de valor 3 bem mais claramente dominando os valores de maior *fitness* que os demais níveis (2 e 4). O parâmetro P_1 mudou sua visibilidade, sendo possível verificar que no momento o nível de valor 1 é o de maior impacto para altos valores de *fitness*. Essa definição do impacto dos níveis se torna cada vez mais visível na medida que mais configurações são simuladas.

O valor de impacto de um nível será maior quanto maior for seu respectivo valor de *fitness* e quanto mais itens de *fitness* existir para o nível. Portanto, neste trabalho estamos calculando o valor do impacto de um nível pelo somatório dos valores de *fitness* que contém o referido nível em uma amostra. A equação fica assim definida para cálculo do impacto dos níveis de um parâmetro P_1:

$$Impact(P_1, i) = \sum_{j=1}^{n} fitness_{ij}, \tag{4.11}$$

Figura 4.5: População Desbalanceada de configurações de cache (*SimulatedFoods*).

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	Energia(µ J)	Desempenho (Ciclos CPU)	Fitness
12	1	13	1	14	2	13	2	3	2181920	1108706	0,00000060779
12	2	14	0	12	2	12	1	3	2181921	1108706	0,00000060779
12	2	14	0	12	2	13	1	3	2181922	1108706	0,00000060779
14	2	13	1	13	1	13	1	3	2181923	1108706	0,00000060779
13	1	14	2	14	2	14	2	3	2181923	1108706	0,00000060779
12	1	12	1	14	1	14	0	3	2181923	1108706	0,00000060779
12	2	12	1	14	1	14	0	3	2181924	1108706	0,00000060779
14	0	13	2	12	2	12	1	3	2181926	1108706	0,00000060779
14	1	13	2	12	2	12	1	3	2181927	1108706	0,00000060779
14	0	13	2	14	1	14	0	3	2181931	1108706	0,00000060779
12	1	12	1	13	0	14	2	3	2182526	1108706	0,00000060768
12	1	13	2	13	0	12	2	3	2182526	1108706	0,00000060768
12	1	12	1	13	0	14	0	3	2182527	1108706	0,00000060767
12	1	13	2	13	0	14	2	3	2182527	1108706	0,00000060767
12	2	13	2	13	0	12	2	3	2182527	1108706	0,00000060767
13	1	14	0	13	0	13	2	3	2182528	1108706	0,00000060767
14	1	12	1	13	0	14	2	3	2182534	1108706	0,00000060767
14	2	14	2	12	0	14	0	3	2184024	1108482	0,00000060744
14	0	14	2	12	0	14	0	3	2184028	1108482	0,00000060744
14	1	14	0	14	1	13	0	3	2184139	1108814	0,00000060736
13	1	12	0	13	1	12	1	3	2186880	1108282	0,00000060695
14	2	12	0	14	2	12	2	3	2186885	1108282	0,00000060695
14	0	13	2	12	2	12	0	3	2192894	1108394	0,00000060582
14	0	13	2	14	1	12	0	3	2192896	1108394	0,00000060582
13	0	12	1	12	2	14	1	3	2203892	1113890	0,00000060281
13	0	14	0	13	0	13	2	3	2204498	1113890	0,00000060270
13	0	13	2	14	1	12	0	3	2214800	1113578	0,00000060089
12	0	13	1	14	2	13	2	3	2369508	1160302	0,00000056660
12	2	14	0	12	2	12	1	2	2862045	1176578	0,00000049522
14	2	13	1	13	1	13	1	2	2862046	1176578	0,00000049522
13	1	14	2	14	2	14	2	2	2862046	1176578	0,00000049522
13	1	14	0	13	0	13	2	2	2862931	1176578	0,00000049511
14	1	14	0	14	1	13	0	2	2864414	1176684	0,00000049491
14	2	14	2	12	0	14	0	2	2866091	1176110	0,00000049478
13	1	13	1	12	0	13	0	2	2868444	1176214	0,00000049448
14	1	13	1	12	0	13	0	2	2868452	1176214	0,00000049448
13	1	12	0	13	1	12	1	2	2869799	1175894	0,00000049435
14	2	12	0	14	2	12	2	2	2869804	1175894	0,00000049435
14	1	12	0	14	2	12	2	2	2869807	1175894	0,00000049435
13	0	12	1	12	2	14	1	2	2880245	1179778	0,00000049261
12	0	13	1	14	2	13	2	2	3042150	1221088	0,00000046913
12	0	14	2	13	1	14	1	2	3042152	1221088	0,00000046913
12	0	12	0	12	0	12	0	2	3070335	1219228	0,00000046625
12	1	13	0	12	1	14	2	4	8738380	1612812	0,00000019321
13	2	14	1	12	1	12	2	4	8738384	1612812	0,00000019321
12	1	14	1	14	0	12	1	4	8738385	1612812	0,00000019321
13	2	13	0	13	2	14	0	4	8738386	1612812	0,00000019321
13	2	12	2	14	0	13	1	4	8738386	1612812	0,00000019321
14	0	12	2	12	1	13	2	4	8738389	1612812	0,00000019321
14	0	13	0	14	0	14	1	4	8738397	1612812	0,00000019321
12	1	12	2	13	2	13	0	4	8748112	1613274	0,00000019302
14	0	14	1	13	2	12	0	4	8787106	1615225	0,00000019226
13	0	12	1	12	2	14	1	4	8856599	1626700	0,00000019078

onde o parâmetro P_1 possui a diferentes níveis $i = \{1,2,...,a\}$, o conjunto dos $simulatedFoods$ possui n itens de configuração simulados, e o $fitness_{ij}$ é o valor do *fitness* quando o parâmetro A possui o valor i (de outra forma o $fitness_{ij} = 0$). No caso da figura 4.4, para o parâmetro P_1, o impacto dos níveis de valor 0, 1 e 2 serão calculados da seguinte forma:

$$Impact(P_1;0) = \sum_{s=1}^{27} fitness_{0j} = 0,0000037896026792$$
$$Impact(P_1;1) = \sum_{s=1}^{27} fitness_{1j} = 0,0000038863207341$$
$$Impact(P_1;2) = \sum_{s=1}^{27} fitness_{2j} = 0,0000038872415502$$

O valor mais preciso do impacto dos níveis de cada parâmetro somente seria obtido quando o conjunto dos elementos dos *SimulatedFoods* contivesse a simulação de todas as configurações do espaço de projeto. A precisão do cálculo do impacto dos níveis aumenta na medida em que o conjunto dos elementos dos *SimulatedFoods* cresce. No algoritmo AbcDE proposto essa precisão cresce a cada nova iteração. Não foi definido um tamanho limite para a memória que armazena as configurações simuladas (*simulatedFoods*), pois ele depende do tamanho do espaço de projeto da hierarquia de cache e da aplicação em execução na plataforma. No algoritmo AbcDE o tamanho dessa memória é escalável.

4.3.2.4 Seleção de fonte de alimento com base em impacto

No algoritmo ABC multi-objetivo uma nova fonte de alimento candidata v_{ij} obtida na vizinhança de uma fonte de alimento i é obtida utilizando a equação 2.7, respeitando os seus valores máximo e mínimo definidos para o parâmetro j. O parâmetro j a ser modificado é selecionado randomicamente, e seu valor de nível é calculado também randomicamente. Em outras palavras, no ABC multi-objetivo, uma nova fonte de alimento candidata v_{ij} é obtida na vizinhança de uma fonte de alimento i utilizando um clone da fonte de alimento i selecionando randomicamente um dos seus parâmetros j para alterar seu valor utilizando a equação 2.7.

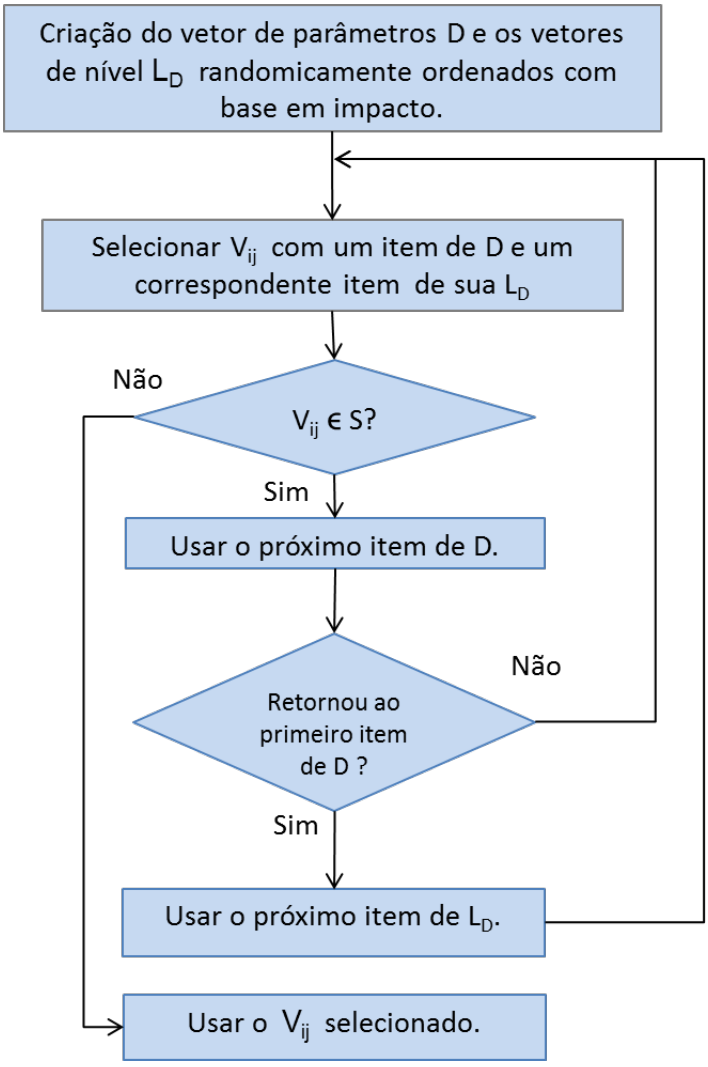
Para o algoritmo AbcDE, estamos propondo melhorar sua busca global, utilizando as informações globais de impacto de parâmetro de cache e impacto de nível de parâmetro. Como a busca global é realizada apenas na fase das abelhas *onlooker*, apenas essa fase será alterada quanto à sua busca. Outros trabalhos já propuseram melhoria de busca global do algoritmo ABC (YU; ZHANG; CHEN, 2013)(YURTKURAN; EMEL, 2015), alterando apenas a fase das abelhas *onlooker*. Nestes trabalhos a equação do $v_{id} = X_{id} + \phi_{id}(X_{id} - X_{kd})$ foi alterada considerando a melhor fonte de alimento global no lugar de uma fonte de alimento $k \neq i$ sorteada em P. No algoritmo AbcDE proposto, a ideia é substituir na fase das abelhas *onlooker* o uso da equação do v_{ij} pelo uso do impacto de parâmetro de cache e impacto de nível de parâmetro, mantendo a randomicidade na busca.

Em cada iteração obtida pela execução do algoritmo, um vetor de parâmetros D é gerado e ordenado randomicamente com base no impacto desses parâmetros, utilizando a técnica da roleta (*rollete wheel*)(BAKER, 1987). Exemplificando, se a ordem do impacto dos

parâmetros (considerando 9 parâmetros) é [5][3][8][7][1][4][2][0][6], sendo [5] o parâmetro de maior impacto e [6] o de menor impacto, um possível resultado de ordenação randômica do vetor D com base no impacto seria [5][3][7][8][2][4][1][0][6]. Da mesma forma, em cada iteração um conjunto de vetores P_D (um vetor para cada parâmetro) é gerado e os vetores ordenados randomicamente com base no impacto dos níveis de cada parâmetro. Mais uma vez exemplificando, se a ordem de impacto dos níveis de um parâmetro P_0 (considerando 3 níveis) é [0][2][1], sendo [0] o nível de maior impacto e [1] o de menor impacto, um possível resultado de ordenação randômica do vetor L_1 com base no impacto seria [0][1][2]. O mesmo ocorre para os demais vetores $L_1, L_2, ..., L_8$.

Um vetor de parâmetros D e um vetor de níveis L_D, para cada parâmetro, são usados na busca na vizinhança porque, nesta fase, é possível selecionar uma configuração que já esteja presente no próprio conjunto S das fontes de alimento em evolução. Portanto, uma nova busca seria necessária e a ordenação randômica já define qual será o próximo parâmetro e nível a serem buscados. O algoritmo pode ser melhor entendido através da figura 4.6.

Figura 4.6: Fluxograma de seleção de parâmetro e nível para obter a fonte de alimento candidata v_{id}.



O fluxograma da figura 4.6 mostra como o algoritmo AbcDE proposto realiza a seleção de parâmetro e de nível para obter a fonte de alimento candidata v_{id}. Na sequência, inicialmente é usado o primeiro item do vetor de parâmetro D e o primeiro elemento de seus correspondentes vetores L_D (os randomicamente de maior impacto), para obter v_{id}. Se a v_{id} obtida já está inclusa em P (população evoluída pelo algoritmo), é selecionado o próximo item do vetor D mantendo o primeiro elemento de seu correspondente vetor L_D e é obtida uma nova fonte de alimento v_{id}. Se todos os itens do vetor D foram selecionados (sempre definindo um v_{id} existente em P), a sequência aponta novamente para o primeiro item do vetor D usando agora o segundo item do correspondente vetor L_D. Essa seleção continua até se obter um parâmetro e um nível para gerar um v_{id} não incluso em P. Essa heurística define que cada fonte de alimento em P, deve prioritariamente e progressivamente convergir para configurações de cache de menor consumo de energia e melhor desempenho. O cálculo de uma nova fonte de alimento na vizinhança é realizado pelo bloco "*Seleção de Configuração com base em Impactos de Parâmetro e de Nível*" (figura 4.2).

4.4 Modelo de Energia

O modelo de energia utilizado neste trabalho foi proposto por Rawlins e Gordon-Ross e sua descrição completa pode ser verificada em (RAWLINS; GORDON-ROSS, 2013). O referido modelo de energia pode ser verificado na figura 4.7. O modelo mostra que a *Energia Total* consumida pelas caches é o somatório da energia consumida pela cache de cada núcleo processador individualmente. Nesse modelo de energia verifica-se que foi incluído como energia consumida por cada cache tanto os valores de energia referentes aos acessos realizados à cache para leitura/escrita tanto em *hits* como em *misses (Energia Estática* e *Energia Dinâmica*), como os acessos à memória (*Energia de Preenchimento* e *Energia de WriteBack*) devidos aos *misses* em L1 e aos writebacks, e a energia que a CPU gasta enquanto espera a chegada de um *entry* (dado de escrita ou leitura) devido a um *miss* em L1 (*Energia da CPU esperando*).

O modelo de energia proposto por Rowlins foi concebido e descrito para apenas um nível de cache. Como este trabalho é proposto para caches multi-nível, descrevemos a seguir seu uso considerando apenas um nível de cache e em múltiplos níveis de cache.

4.4.1 Modelo de energia para cache de apenas um nível (L1)

Na figura 4.8, para calcular os valores de *Energia Estática* e *Energia Dinâmica*, foi necessário usar a ferramenta CACTI6.5 [que combina as melhorias do CACTI5.0 e CACTI6.0 (MURALIMANOHAR; BALASUBRAMONIAN; JOUPPI, 2008)] para obter os valores de *Energia_Acesso_L1* e de *Energia_Estática_L1*. Usando o CACTI6.5 foi selecionada a tecnologia 0,032μmetros, o modelo de cache UCA e os valores de cada configuração de cache. Para calcular a *Energia de Preenchimento* e a *Energia de WriteBack*, foi utilizado o simulador *DDR4L*

Figura 4.7: Modelo de energia para sistemas *multicore*.

Energia total= ∑ (energia consumida em cada core)
Energia consumida por cada core:
Energia= Energia Dinâmica + Energia Estática + Energia de preenchimento + Energia de Writeback + Energia da CPU esperando

Energia Dinâmica= Acessos _L1 x Energia _Acesso_L1
Energia Estática= ((misses_L1 x Ciclos_Latência_misses) + (hits_L1 x Ciclos_Latência_hits) + (writebacks_L1 x Ciclos_Latência_writebacks)) x Energia_Estática_L1
Energia de Preenchimento= misses_L1 x (tamanho_Linha/tamanho_Palavra) x Energia_leitura_Memória_porPalavra
Energia de Writeback= writebacks_L1 x (tamanho_Linha/tamanho_Palavra) x Energia_escrita_Memória_porPalavra
Energia da CPU esperando= ((misses_L1 xCiclos_Latência_misses) + (writebacks_L1 x Ciclos_Latência_writebacks)) x Energia_CPU_Disponível

Fonte: RAWLINS; GORDON-ROSS (2013).

Power Calculator da Micron Technology para obter suas características de alto desempenho e consumo de potência, de onde extraímos os valores de *Energia_Leitura_Memória_porPalavra* e *Energia_Escrita_Memória_porPalavra*. Finalmente, para calcular o valor de *Energia da CPU Esperando* definimos o valor de *Energia_CPU_Disponível* como 25% da energia ativa do SPARC-V8 (RAWLINS; GORDON-ROSS, 2013), obtida do manual do Sparc-V8 "Rad-Hat 32 bits ATMEL AT697E SparcV8 manual". A plataforma *Infinity* foi atualizada inserindo um módulo adicional para realizar o cálculo da energia, que realiza todos os cálculos definidos no modelo de energia usado. A plataforma *Infinity* já realizava a contabilização do número de *misses* e *hits*, e os ciclos de latência de *misses* e *hits*. No cálculo do número de ciclos de latência de *misses*, a plataforma *Infinity* também contabiliza os ciclos em que os dados são solicitados para um outro processador e trafegam entre os roteadores da NoC até chegar no seu processador destino. Foi necessário incluir a contabilização do número de *writebacks* que não havia disponível na plataforma *Infinity*. O cálculo de desempenho de cada cache L1 também é realizado no *Módulo Medidor de Energia e Desempenho*, e o valor do desempenho foi definido como o número de ciclos que a CPU espera o *entry* solicitado, tanto nos ciclos de leitura quanto os de escrita, considerando tanto as situações de *hits* quanto as situações de *misses*.

4.4.2 Modelo de energia para cache multinível

Os valores de *Energia Estática*, *Energia Dinâmica*, *Energia de Preenchimento*, *Energia de WriteBack*, e *Energia da CPU Esperando* são calculadas como descrito na seção 4.4.1. Para calcular energia em cache multinível foram realizadas as seguinte mudanças:

Figura 4.8: Modelo de energia para cache de apenas um nível (L1).

$$\texttt{Energia Total} = \sum_{p=0}^{i} Energia_i \qquad \Rightarrow \qquad p = processadores = \{0,1,...,i\}$$

$$Energia_i = Energia\_Dinamica_i + Energia\_Estatica_i + Energia\_Preenchimento_i + Energia\_WriteBack_i + Energia\_CPU\_Esperando_i$$

$$Energia\_Dinamica_i = Acessos\_L1_i \times Energia\_Acesso\_L1_i$$

$$Energia\_Estatica_i = ((Misses\_L1_i \times Ciclos\_Latencia\_Misses\_L1_i) + Hits\_L1_i \times Ciclos\_Latencia\_Hits\_L1_i) + Writebacks\_L1_i \times Ciclos\_Latencia\_Writebacks\_L1_i)) \times Energia\_Estatica\_L1_i$$

$$Energia\_Preenchimento_i = Misses\_L1_i \times (Tamanho\_Linha_i / Tamanho\_Palavra_i) \times Energia\_Leitura\_Memoria\_porPalavra$$

$$Energia\_WriteBack_i = WriteBacks\_L1_i \times (Tamanho\_Linha_i / Tamanho\_Palavra_i) \times Energia\_Escrita\_Memoria\_porPalavra$$

$$Energia\_CPU\_Esperando_i = ((Misses\_L1_i \times Ciclos\_Latencia\_Misses\_L1_i) + Writebacks\_L1_i \times Ciclos\_Latencia\_Writebacks\_L1_i)) \times Energia\_CPU\_Disponivel_i$$

Fonte: RAWLINS; GORDON-ROSS (2013).

- A *Energia da CPU Esperando* é calculada apenas para a cache L1;
- A *Energia de Preenchimento* é calculada apenas para o último nível de cache;
- As demais energias são calculadas para todos níveis de cache;

Considerando uma cache com apenas dois níveis (L1 e L2), podemos realizar uma medição de energia das caches conforme descrito na figura 4.9.

Portanto, o algoritmo propostro AbcDE, foi criado com base no algoritmo ABC modificado para otimização multi-objetivo, tendo seus parâmetros de número de fontes de alimento, limite e número total de execuções ajustados para execução de aplicações diversas. Foi descrito como o algoritmo AbcDE introduziu o uso de população balanceada e estatísticas de modelo de efeito (técnicas do DoE), para melhorar a sua busca global, buscando assim ser mais eficiente quanto ao número de execuções da plataforma MPSoC a ser utilizada pelo algoritmo. Na próxima seção apresentamos os resultados obtidos com os experimentos para comprovar as melhorias obtidas com a aplicação das técnicas de DoE.

Figura 4.9: Modelo de energia para cache de dois níveis.

$$\texttt{Energia Total} = \sum_{p=0}^{i}(EnergiaL1_i + EnergiaL2_i) \quad \Rightarrow \quad p = procs = \{0,1,...,i\}$$

$$EnergiaL1_i = Energia\_Dinamica_i + Energia\_Estatica_i + Energia\_WriteBack_i + Energia\_CPU\_Esperando_i$$

$$EnergiaL2_i = Energia\_Dinamica_i + Energia\_Estatica_i + Energia\_Preenchimento_i + Energia\_WriteBack_i$$

$$Energia\_Dinamica_i = Acessos\_Ln_i \times Energia\_Acesso\_Ln_i$$

$$Energia\_Estatica_i = ((Misses\_Ln_i \times Ciclos\_Latencia\_Misses\_Ln_i) + Hits\_Ln_i \times Ciclos\_Latencia\_Hits\_Ln_i) + Writebacks\_Ln_i \times Ciclos\_Latencia\_Writebacks\_Ln_i)) \times Energia\_Estatica\_L1_i$$

$$Energia\_Preenchimento_i = Misses\_Ln_i \times (Tamanho\_Linha_i / Tamanho\_Palavra_i) \times Energia\_Leitura\_Memoria\_porPalavra$$

$$Energia\_WriteBack_i = WriteBacks\_Ln_i \times (Tamanho\_Linha_i / Tamanho\_Palavra_i) \times Energia\_Escrita\_Memoria\_porPalavra$$

$$Energia\_CPU\_Esperando_i = ((Misses\_Ln_i \times Ciclos\_Latencia\_Misses\_Ln_i) + Writebacks\_Ln_i \times Ciclos\_Latencia\_Writebacks\_Ln_i)) \times Energia\_CPU\_Disponivel_i$$

5

Resultados

Este capítulo apresenta os resultados das simulações realizadas usando o algoritmo AbcDE na exploração do espaço de projeto de caches para otimização de consumo de energia e desempenho utilizando a plataforma *Infinity* como plataforma MPSoC de aplicação específica. Preliminarmente aplicamos o algoritmo AbcDE para a plataforma Infinity explorando espaço de projeto de um único nível de cache L1. Esse trabalho inicial foi importante para validação do uso das técnicas de DoE no AbcDE e apresentamos seus resultados na seção 5.4 comparando seus resultados com o algoritmo ABC multi-objetivo descrito na seção 4.2. Esse trabalho foi publicado em (SANTOS; BARROS; AZIZ, 2016). Na seção 5.5 apresentamos os resultados do uso do Algoritmo AbcDE aplicado às caches em múltiplos níveis, utilizando também a plataforma Infinity com caches L1 e L2, e comparando seus resultados com os dos algoritmos MOPSO e algoritmo ABCMOP.

As aplicações selecionadas para simulação na plataforma *Infinity* foram FFT, Radix (*benchmark* Splash-2), Dijkstra (*benchmark* ParMiBench) e Matrix (multiplicação de matrizes usando método Fox (PACHECO, 2011)), aplicadas tanto para a plataforma em nível único de cache quanto nas simulações em múltiplos níveis de cache. As aplicações SHA, StringSearch e BasicMath (*benchmark* ParMiBench) também foram utilizadas com a plataforma com cache em múltiplos níveis. Todas as aplicações foram atualizadas para execução distribuída entre os diversos núcleos da arquitetura, com a comunicação através de variáveis compartilhadas. Isso se fez necessário para que fosse explorado na plataforma MPSoC a comunicação entre núcleos, bem como a utilização das invalidações devido a coerência de cache no nível L1 da plataforma. A plataforma *Infinity* foi compilada como MPSoC de quatro núcleos.

Apresentaremos nas seções seguintes o ambiente de simulação utilizado, a descrição da plataforma MPSoC Infinity utilizada para simulação, a descrição das alterações realizadas às aplicações para uso na plataforma Infinity, o reuso dos resultados das execuções das aplicações para diferentes configurações de caches, e os resultados dos experimentos para o uso do Algoritmo AbcDE aplicado a caches em nível único (L1) e em múltiplos níveis de cache (L1 e L2).

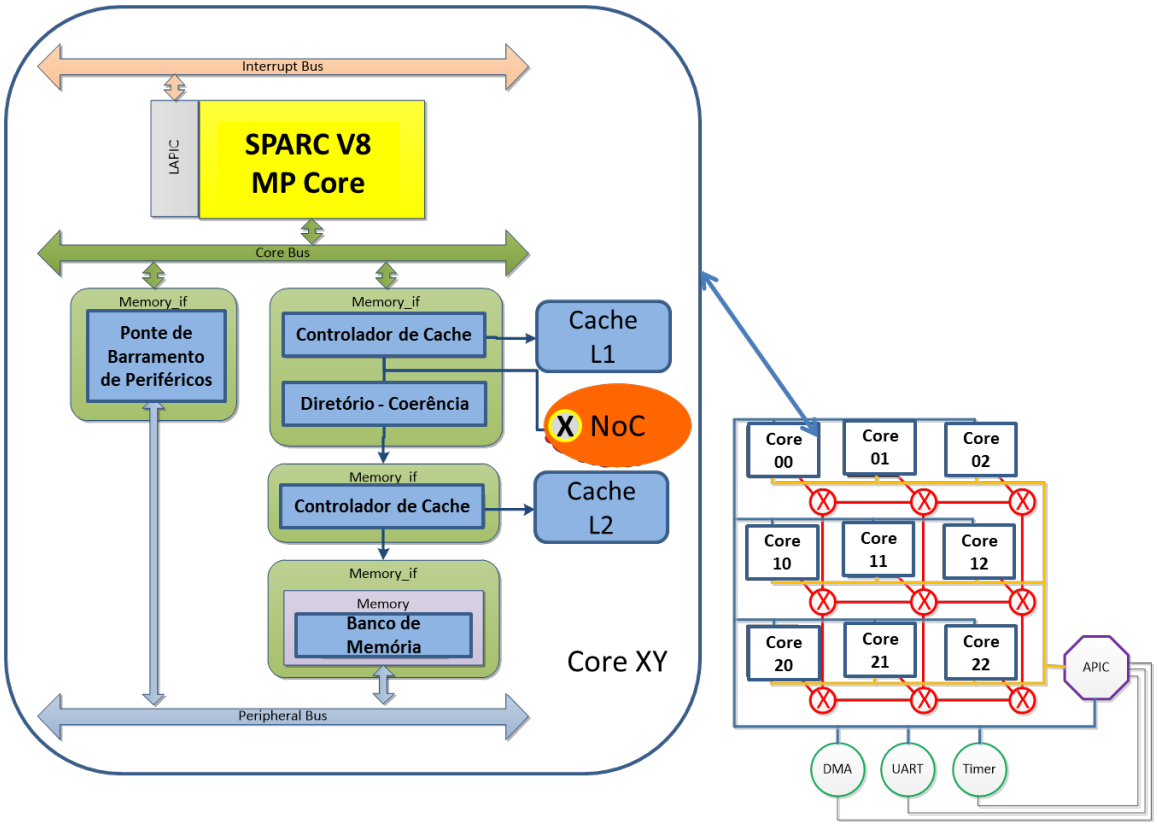
5.1 Ambiente de Simulação

O ambiente de simulação inclui 8 máquinas virtuais Linux Ubuntu usados para executar as simulações, a plataforma MPSoC *Infinity* de aplicação específica, e o espaço de projeto de cache L1 da plataforma *Infinity*. As máquinas virtuais foram configuradas para uso de 4GB de memória DRAM, com o SO Linux Ubuntu, versão 10.10, e 1 processador.

5.2 Plataforma de Simulação

Em um nível de cache existem k possibilidades de configurações para seus parâmetros: tamanho total, tamanho de bloco e nível de associatividade. Para n níveis de cache as possibilidades de configurações de cache aumentam para $n \times k$. Em uma arquitetura MPSoC, em que c é o número de núcleos, a quantidade de configurações de cache é aumentada para $(n \times k)^c$.

Figura 5.1: Plataforma MPSoC *Infinity*



Fonte: (FARIAS et al., 2013)(AZIZ et al., 2014).

Como mencionado, nos experimentos utilizamos uma plataforma MPSoC baseada em NoC chamada *Infinity* (figura 5.1). A plataforma *Infinity* foi implementada em SystemC TLM e em cada *core* na NoC, contém um elemento de processamento incluindo um processador Sparc-V8, dois níveis de cache, gerenciadores de cache com coerência baseada em diretório,

banco de memória e endereços de periféricos (FARIAS et al., 2013)(AZIZ et al., 2014). Para uso neste trabalho, foi realizada uma alteração na plataforma *Infinity* para que fosse possível executar diferentes configurações apenas atualizando um arquivo de configuração entre subsequentes execuções. O parâmetro tamanho do bloco precisou ser definido como o mesmo valor para todos os *cores* devido a limitações tecnológicas na implementação da coerência de cache da plataforma *Infinity*. Todas as aplicações utilizadas para execução na plataforma foram atualizadas para execução em todos os *cores* simultaneamente, em esquema *multi-thread*, buscando assim explorar paralelismo de execução e comunicação entre os *cores*.

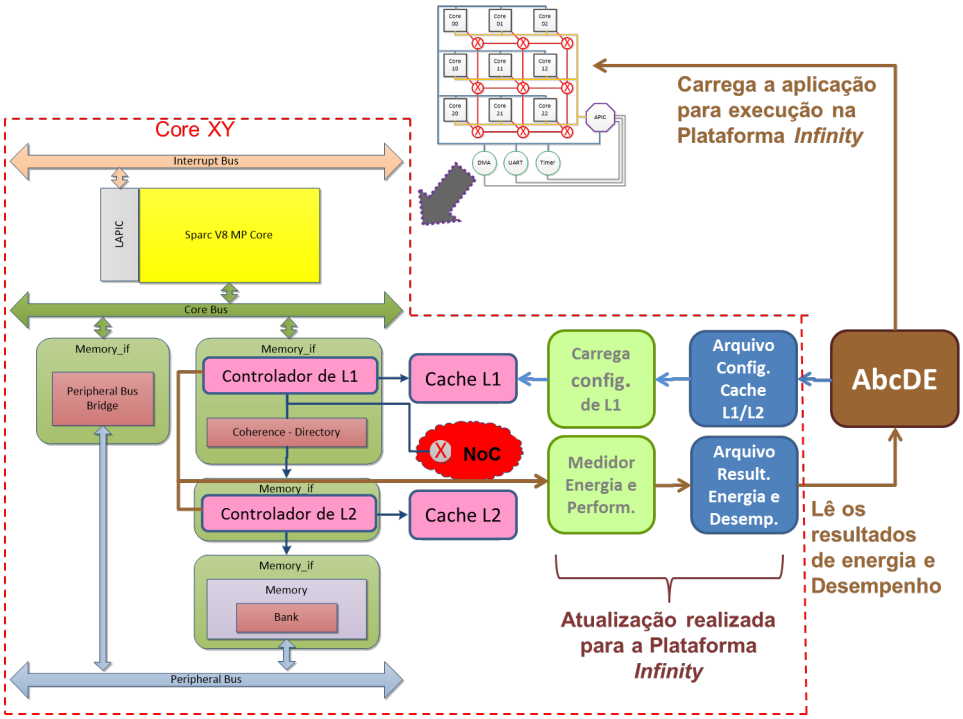
Nos experimentos com cache em nível único (seção 5.4), foram usados nos experimentos a reconfiguração apenas da cache L1, em que foram utilizados três diferentes valores de tamanho de cache (2k, 4K e 8K bytes), três valores de tamanho de bloco (4, 8 e 16 palavras/bloco) e três valores para níveis de associatividade (mapeamento direto, conjunto de 2 e conjunto de 4). Esses parâmetros geram 27 configurações diferentes de cache para cada *core*. Considerando a plataforma com 4 núcleos, o espaço de projeto de cache deveria atingir o valor de $(27)^4$, diferentes configurações, mas ela fornece um valor total de 19683 configurações válidas (todos os cores devem usar o mesmo tamanho de bloco). O total de configurações passa a ser calculado como $3^{(2\times c)+1}$, onde c é o número de processadores do MPSoC.

Nos experimentos com cache em multinível (seção 5.5), foram usados nos experimentos a reconfiguração das caches L1 e L2. Além dos valores de configuração da cache L1 citados acima, também foram utilizados três diferentes valores de tamanho de cache L2 (16k, 32K e 64K bytes). Para a cache L2, foram fixados os valores de tamanho de bloco e nível de associatividade em respectivamente 32 palavras/bloco e associatividade por conjunto de 8.

Os valores de tamanho de cache, níveis de associatividade, e tamanho do bloco, utilizados nos experimentos das seções 5.4 e 5.5, foram selecionados de forma a gerarem uma maior distribuição de diferentes valores de consumo de energia e número de ciclos de CPU testados simulando a plataforma *Infinity* com as aplicações Matrix (multiplicação de matrizes) e FFT.

A figura 5.2 mostra o diagrama do sistema de exploração de espaço de projeto proposto (algoritmo AbcDE) em interação com a plataforma *Infinity*. Para o algoritmo AbcDE obter o conjunto de configurações otimizadas, é necessário se executar diversas simulações da aplicação alvo na plataforma *Infinity*. Cada execução deverá ser realizada para uma configuração de cache (de todos os *cores*) específica conforme o definido pelo AbcDE. Depois de cada execução, o algoritmo AbcDE busca os resultados de consumo de energia e de desempenho da cache L1/L2 para a referida execução. O algoritmo AbcDE disponibiliza a configuração das caches L1 e L2 para a plataforma *Infinity* escrevendo em um arquivo de configuração. Ao iniciar a execução da aplicação, a plataforma Infinity lê o referido arquivo para iniciar os seus objetos utilizando como entrada os parâmetros descritos no arquivo de configuração das caches. Depois de levantar toda a plataforma Infinity reconfigurada, a aplicação é executada e os gerenciadores de cache L1 e L2 contabilizam os dados de estatística de cache para serem disponibilizados para o módulo medidor de energia e desempenho. O referido módulo utiliza o modelo de energia

Figura 5.2: Diagrama do sistema em exploração de espaço de projeto AbcDE em interação com a plataforma Infinity.



(descrito na Seção 4.4) da cache para contabilizar a energia consumida pelas caches de cada *core* durante a execução da aplicação, bem como o número de ciclos utilizados pelas CPUs na mesma execução (desempenho). Essas informações são gravadas pelo medidor de energia em um arquivo de resultados de energia e desempenho. Quando a plataforma Infinity finaliza a execução, o algoritmo AbcDE lê o arquivo de resultados de energia e desempenho, e utiliza esses valores para tomada de decisão em sua execução.

5.2.1 Reuso das configurações previamente simuladas

O algoritmo ABC original ou o ABC em multi-objetivo, depois de selecionarem uma nova fonte de alimento candidata v_{id}, precisam simular o modelo da plataforma MPSoC com a configuração da nova fonte de alimento candidata v_{id} (nova configuração das caches) para obter os valores de consumo de energia e o número de ciclos de execução da CPU. A simulação da plataforma poderia ser evitada se a referida configuração de v_{id} já tivesse sido executada e o resultado guardado em memória. Como no algoritmo AbcDE existe a necessidade de uso do novo conjunto de fontes de alimento *SimulatedFoods* para cálculo do impacto dos diferentes níveis de cada parâmetro, e esse conjunto vai aumentando durante e após cada iteração do algoritmo, temos uma grande memória de dados valiosa e cara que precisa ser melhor explorada. A ideia é que esse conjunto de fontes de alimento *SimulatedFoods* seja consultado antes de se simular a plataforma com a configuração de v_{id} para reusar o resultado da referida simulação caso ela já tenha sido previamente simulada e esteja no referido conjunto. O reuso das configurações

de cache simuladas é bem útil e correto no ambiente de simulação que usamos, pois re-simular a plataforma para a mesma configuração de cache e aplicação(ões) gera exatamente o mesmo resultado que a simulação já previamente realizada. No início da execução do algoritmo o reuso das configurações executadas aparece de forma discreta e ele vai aumentando na medida que o conjunto *bestFoods* vai convergindo para as configurações de menor consumo de energia e ciclos de CPU, isto é, as fontes de alimento i vão ficando cada vez mais próximas uma das outras, de forma que uma nova fonte de alimento i' pode coindidir com alguma i de S, ou outra configuração próxima já previamene simulada.

5.3 Aplicações Selecionadas para Simulação na Plataforma MPSoC

As aplicações simuladas na plataforma *Infinity* precisam ser adaptadas para que a alocação de tarefas entre os diferentes processadores funcione corretamente, bem como haja comunicação entre as mesmas através de compartilhamento de dados. Selecionamos as seguintes aplicações de benchmarks para uso na plataforma MPSoC *Infinity*, todas com execução concorrente com tarefas que se comunicam através de variáveis compartilhadas:

- Multiplicação de Matrizes;
- Radix (benchmark Splash2);
- FFT (benchmark Splash2);
- Dijkstra (benchmark ParMiBench);
- Sha (benchmark ParMiBench);
- StringSearch (benchmark ParMiBench);
- BasicMath (benchmark ParMiBench);

5.3.1 Aplicação Multiplicação de Matrizes - Matrix

Esta aplicação foi desenvolvida pelo grupo de pesquisas que trabalhou com a plataforma *Infinity* no CIN-UFPE. É uma aplicação que realiza a multiplicação de duas matrizes A e B utilizando método Fox (PACHECO, 2011). Executando essa aplicação na plataforma *Infinity*, cada processador realiza uma parte das multiplicações necessárias. É utilizada uma cláusula $case = 0$ para o $switch(PROCESSOR\_ID)$, para que o processador 0 crie as matrizes A e B, e dê início à multiplicação também incluindo os demais processadores na divisão de tarefas.

As variáveis compartilhadas são cada item das matrizes A e B e a variável global *NLocal* que corresponde ao tamanho da matriz local. Existe concorrência também na impressão de

resultados parciais, utilizando a biblioteca *halPrintf*, que simula na plataforma *Infinity* o *printf* da linguagem C.

5.3.2 Aplicação Radix

A aplicação Radix realiza a tarefa de ordenação de 8.388.608 números inteiros. O processador 0 faz a inicialização e fechamento da aplicação. Os demais processadores farão a tarefa de ordenação de forma concorrente. Existe um objeto global (struct) chamado *global_memory* que é compartilhado, em execução, por todos os processadores simultaneamente (exceto o processador 0), e para acessá-lo é necessário solicitar o "lock global", depois liberando-o, para obter acesso para leitura e escrita. Os atributos do objeto *global_memory* são dados compartilhados entre processadores. Existe concorrência também na impressão de resultados parciais, utilizando a biblioteca *halPrintf*, que simula na plataforma o *printf* da linguagem C.

5.3.3 Aplicação FFT

A aplicação FFT (*Fast Fourier Transform*) realiza uma transformada rápida de Fourier e sua transformada inversa sobre um array de dados. O processador 0 inicializa e finaliza a aplicação, enquanto que os demais processador realizam a transformadas rápidas de Fourier de forma concorrente. Da mesma forma que na aplicação Radix, no FFT existe um objeto global (struct) chamado *global_memory* que é compartilhado, em execução, por todos os processadores de forma concorrente (exceto o processador 0), e para acessá-lo é necessário solicitar o "lock global", depois liberando-o, para obter acesso para leitura e escrita. Existe concorrência também na impressão de resultados parciais, utilizando a biblioteca *halprintf*, que simula na plataforma o *printf* da linguagem C.

5.3.4 Aplicação Dijkstra

A aplicação Dijkstra constrói um gráfico em uma representação matricial e calcula o caminho mais curto entre cada par de nodos de um caminho completo. É um benchmark importante para uso em arquiteturas com comunicação em rede. Em nossa implementação, o processador 0 inicializa todas as *threads* que serão executadas pelos demais processadores, e inicializa/finaliza a aplicação. Os dados compartilhados são referentes ao objeto global *rgnNodes*, e as variáveis globais *chStart* e *chEnd*. Mais uma vez, existe concorrência a dados para realizar impressão parcial dos resultados, como nas aplicações anteriores.

5.3.5 Aplicação Sha

O aplicativo Sha é um algoritmo de *hash* de segurança que produz uma mensagem de 160-bits para uma dada entrada. É frequentemente usado em troca de mensagens. Nessa versão da aplicação Sha, o processador 0 inicializa todas as *threads* que serão executadas pelos demais

processadores, e inicializa/finaliza a aplicação. Os dados compartilhados são referentes ao objeto global *paramsArr*. Também existe concorrência a dados para realizar impressão parcial dos resultados.

5.3.6 Aplicação StringSearch

A aplicação StringSearch realiza uma busca para dadas palavras em frases usando um algoritmo de comparação que ignora maiúsculas e minúsculas. Mais uma vez, o processador 0 inicializa todas as *threads* que serão executadas pelos demais processadores, e inicializa/finaliza a aplicação. Os dados compartilhados são referentes ao objeto global $paramsArr[]$, e às variáveis globais *here* e $inFileName$. Também existe concorrência a dados para realizar impressão parcial dos resultados.

5.3.7 Aplicação BasicMath

A aplicação BasicMath executa cálculos matemáticos simples que muitas vezes não têm suporte de hardware dedicado em processadores embarcados, como por exemplo, resolver funções cúbicas, conversão de graus para radiando, etc. Os dados de entrada são um conjunto fixo de constantes. O processador 0 inicializa todas as *threads* que serão executadas pelos demais processadores, e inicializa/finaliza a aplicação. Os dados compartilhados são referentes ao objeto global ${}_{a}dlimParams$. Também existe concorrência a dados para realizar impressão parcial dos resultados.

5.4 Resultados do Algoritmo AbcDE aplicado a Cache de Único Nível - L1

Nessa seção apresentamos os resultados experimentais para o Algoritmo AbcDE comparando com os resultados obtidos pelo algoritmo ABC multi-objetivo (descrito na seção 4.2) para o espaço de projeto de cache L1 da plataforma MPSoC *Infinity* (com quatro processadores),observando a otimização no consumo de energia e desempenho. As aplicações multi-thread selecionadas foram FFT, Radix, Djikstra e Multiplicação de matrizes (Matrix), descritas na seção 5.3.

Os algoritmos ABC multi-objetivo e o AbcDE foram configurados usando a população de colônia de abelhas de 32 abelhas (50% como abelhas *employed* e 50% como abelhas *onlooker* e *scout*); o número de fontes de alimento é igual 16, e o parâmetro *Limit* definido como 9 tentativas na vizinhança. Os algoritmos foram executados 30 vezes para cada aplicação para obtenção de valores médios. Esses valores foram definidos com base em testes preliminares realizados para encontrar os melhores resultados em um menor tempo de execução.

As simulações para o algoritmo ABC em multi-objetivo e para o algoritmo ABcDE foram realizadas e seus resultados comparados. O espaço de projeto (execução exaustiva) inclui os resultados de simulação para todas as 19683 diferentes configurações.

A figura 5.3 mostra a simulação exaustiva e os resultados dos algoritmos AbcDE e o ABC em Multi-Objetivo em termos de desempenho (ciclos de CPU) e consumo de energia ($\mu Joules$) para as aplicações multi-thread Matrix, Radix, FFT e Djikstra. Para cada aplicação é mostrada uma imagem em zoom com as configurações de cache resultante da execução dos algoritmos com algumas configurações de cache do exaustivo, e uma menor incluindo o exaustivo completo.

Foram feitas análises quanto à precisão e à eficiência dos dois algoritmos. As métricas de tempo de execução do algoritmo e do número de simulações da plataforma são apresentadas como medidas de eficiência. O critério de precisão corresponde a quão próximo os algoritmos chegam, em termos de ADRS (Distância Média do Conjunto de Referência) (MARIANI et al., 2012), do conjunto exato de Pareto. Quanto menor o valor do ADRS melhor será a qualidade do resultado do algoritmo.

A figura 5.4 mostra os resultados das execuções dos algoritmos em termos de tempo médio de execução, número de simulações da plataforma e o ADRS. Foi verificado que o algoritmo AbcDE apresentou um melhor tempo médio de execução do algoritmo para todas as aplicações analisadas, obtendo uma melhoria média de 42,3% sobre o ABC em Multi-Objetivo. O algoritmo AbcDE necessitou de menor quantidade de execuções da plataforma MPSoC, para todas as aplicações obtendo uma melhoria média dee 40,4% sobre o ABC em Multi-Objetivo. A figura 5.4-c mostra a precisão em termos de ADRS. O algoritmo AbcDE obteve um melhor/menor ADRS para todas as aplicações exceto para a aplicação Djikstra, resultando em um valor médio geral de 0,57%.

Esse experimento com apenas um nível de cache é bem importante para verificação da precisão do algoritmo, pois com apenas um nível de cache é bem mais fácil se obter um experimento exaustivo de todo o espaço de projeto. Tendo o resultado do exaustivo podemos comparar o resultado obtido pelo algoritmo AbcDE com o conjunto do *Pareto* (figura 5.3). A métrica ADRS só pode ser obtida conhecendo-se o conjunto do *Pareto*, pois ela representa a distância percentual que o algoritmo possui, no resultado de sua execução, com relação ao conjunto do *Pareto*.

5.5 Resultados do Algoritmo AbcDE aplicado a Cache Multinível

Nessa seção realizamos experimentos e analisamos o desempenho do algoritmo AbcDE aplicado à uma plataforma MPSoC com dois níveis de cache (L1 e L2), e comparando esses resultados com simulações semelhantes utilizando outros algoritmos multi-objetivo do estado da arte. Selecionamos dois algoritmos para essa análise comparativa que são o MOPSO (COELLO;

Figura 5.3: Consumo de energia versus Desempenho - ABC Multi-Objetivo, Algoritmo AbcDE e Exaustivo para as aplicações Matrix, Radix, FFT e Djikstra.

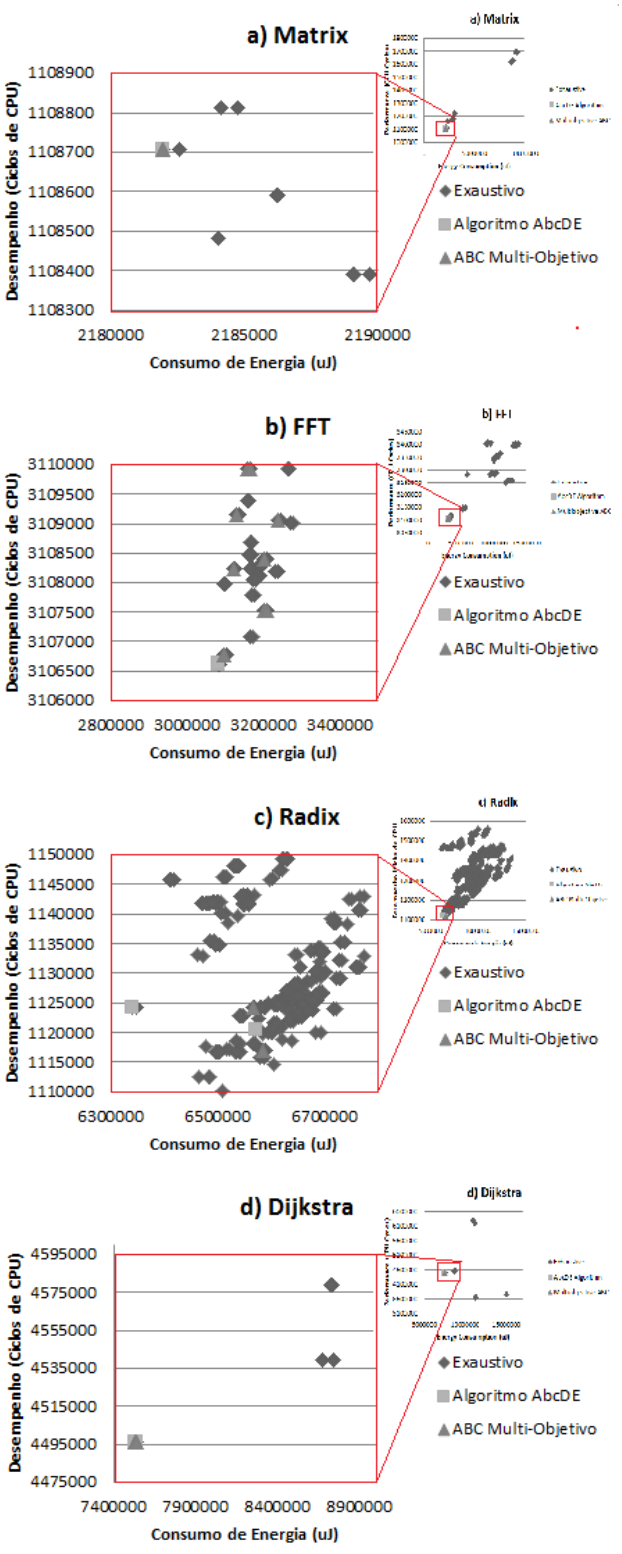
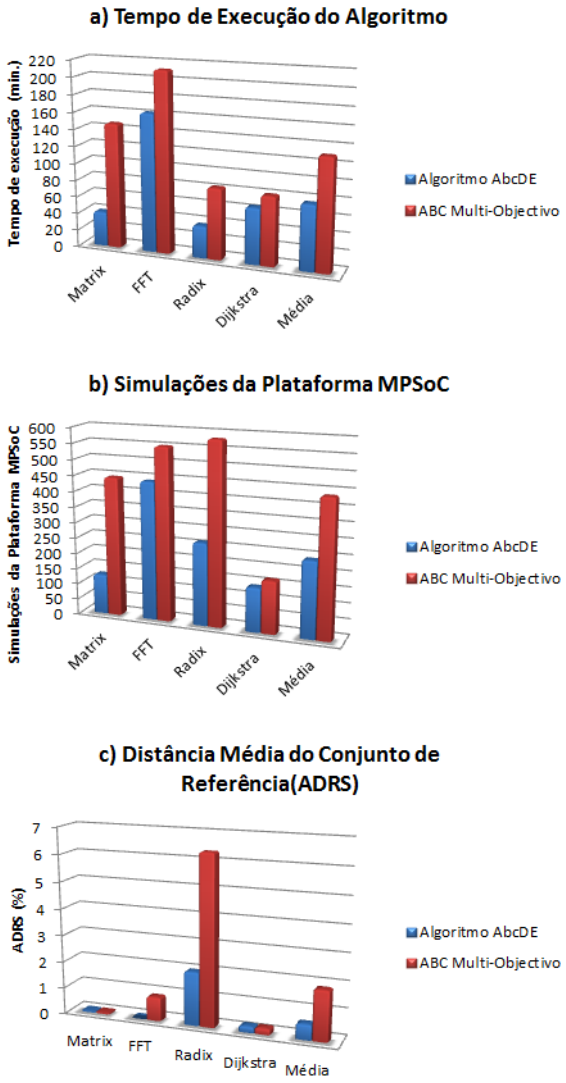


Figura 5.4: Resultados para tempo médio de execução, Simulações da Plataforma, e ADRS.



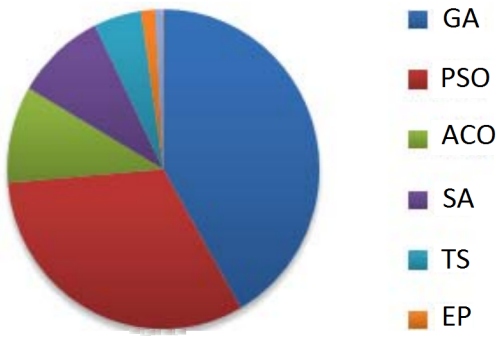
LECHUGA, 2002) e o ABCMOP (SANTOS; SILVA-FILHO, 2014).

Como a abordagem do algoritmo AbcDE propõe realizar um *tuning* estático de cache considerando cache multinível, é importante apresentar resultados não só aplicados a um nível de cache como também a múltiplos níveis de cache. Para isso é necessário o uso do modelo de energia de múltiplos níveis (seção 4.4.2) e de tratamento para o caso de parâmetros adicionais de cache. Com a inclusão de múltiplos níveis o espaço de projeto cresce substancialmente, pois quando utilizamos apenas um nível de cache L1 com 4 processadores (seção 5.4), o algoritmo AbcDE utilizou nove parâmetros de cache L1 (quatro para tamanho total de cache, quatro para associatividade, e um para tamanho de bloco) e isso gerou um espaço de projeto de 19.683(3^9) diferentes configurações de cache a serem exploradas. Com a introdução de mais um nível de cache (L2) o número de parâmetros de cache com 4 processadores dobra a 18 parâmetros, e o espaço de projeto passaria para 387.420.489 (3^{18}) diferentes configurações de cache a serem exploradas pelo referido algoritmo. Nesse caso, o tempo de execução do algoritmo fica demasiadamente aumentado para realizar os experimentos para múltiplas aplicações para a realização de análise comparativa com outros algoritmos. Portanto, nos experimentos com múltiplos níveis de cache utilizamos dois níveis de cache (L1 e L2), utilizando apenas de L2 o parâmetro de tamanho de cache (parâmetro de maior impacto dos três), totalizando assim 13 parâmetros (4 parâmetros de tamanho de cache L2 adicionais), e obtendo um espaço de projeto significativo de 1.594.323 (3^{13}) diferentes configurações de cache. Dessa forma a execução de diversos experimentos e comparações com outros algoritmos, utilizando múltiplas aplicações, se tornou possível. Mesmo tendo um espaço de projeto menor, o uso de treze parâmetros (1.594.323 configurações de cache) já inviabiliza a realização de uma simulação exaustiva para se saber efetivamente quais as melhores configurações de cache entre todas.

A metaheurística MOPSO (*Multi-Objective Particle Swarm Optimization*) foi selecionada para avaliação do Algoritmo AbcDE por possuir diversas similaridades com a meta-heurística ABC multiobjetivo. Ambas são meta-heurísticas de otimização multiobjetivo baseadas em população que usam enxame de partículas para evoluir seus objetivos de otimização. Portanto alguns parâmetros importantes de um poderá ser utilizado para configurar o outro gerando semelhanças de configurações para execução. Esses parâmetros são o número de partículas (ou número de abelhas) e o número de iterações. O MOPSO é uma meta-heurística bastante explorada em publicações científicas, como mostra pesquisa da figura 5.5 em que o PSO só não é mais referenciada que o Algoritmo Genético (GA), que já existe há mais tempo, desde 1975. O MOPSO vêm sendo utilizado para solução de problemas numa grande variedade de áreas tecnológicas, conforme pode ser verificado na figura 5.6.

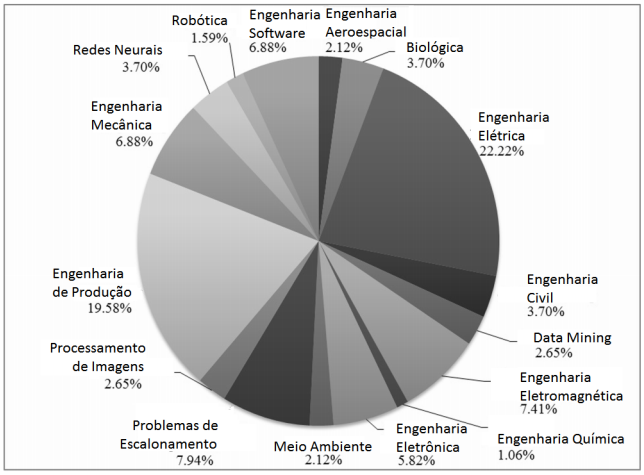
O MOPSO foi proposto por Coello (COELLO; LECHUGA, 2002)(MOSTAGHIM; BRANKE; SCHMECK, 2007) e é basicamente o algoritmo PSO (KENNEDY; EBERHART, 1995) operacionalizado para trabalhar com otimização multi-objetivo com base em análise de dominância de Pareto. O MOPSO se baseia na ideia de cada partícula atualizar a sua experiência de voo em um arquivo após cada ciclo de voo. Essas atualizações em arquivo consideram um

Figura 5.5: Distribuição de publicações de algoritmos de metaheurística.



Fonte: ESLAMI et al. (2012).

Figura 5.6: Áreas de aplicação publicadas com uso do MOPSO.



Fonte: LALWANI et al. (2013).

sistema com base geográfica definido em termos de valores da função objetivo de cada partícula. O espaço de projeto é dividido em hipercubos que recebem um valor de *fitness* com base no número de partículas nele contido. Com objetivo de selecionar um líder para cada partícula do enxame, é feita uma seleção usando a técnica da roleta e os valores de fitness são aplicados uma vez, para selecionar o hipercubo a partir do qual o líder será buscado. Depois o líder é selecionado randomicamente desse hipercubo. Para a implementação do algoritmo MOPSO usado nesse trabalho, foi utilizado o código C original do autor, adaptado para o nosso problema de cache multinível. O número de partículas utilizado na execução do MOPSO é igual ao número de abelhas utilizado no algoritmo AbcDE, que foi de $2 \times 81 = 162$.

Já a meta-heurística ABCMOP (*Artficial Bee Colony for Multi-Objective Problems*) foi selecionada porque, assim como o algoritmo AbcDE, também foi baseada no algoritmo ABC, é multi-objetivo e também aplicada a *tuning* de cache. Portanto, ela possui muitas similaridades com o algoritmo proposto ABcDE e sua aplicação, e poderá utilizar os mesmos parâmetros selecionados para o técnica AbcDE nas simulações comparativas. As principais diferenças

do ABCMOP para o AbcDE (além de não usar DoE) são que o ABCMOP usa o *indicador de cobertura* (TAN; LEE; KHOR, 2001) como base do seu critério de parada automática, e a distribuição da população de abelhas é 60% para abelhas *Employed*, 30% para abelhas *Onlookers* e 10% para abelhas *Scout*. Adicionalmente, a fase das abelhas *Scout* não trabalha com análise de tentativas sem sucesso comparando com o parâmetro *limit*. Para a implementação do algoritmo ABCMOP usado nesse trabalho, utilizamos o código C original do algoritmo ABC (KARABOGA, 2005), incluindo todas as características do ABCMOP (SANTOS; SILVA-FILHO, 2014).

As figuras 5.7 e 5.8 mostram os resultados dos algoritmos AbcDE, MOPSO e o ABC-MOP em termos de desempenho (ciclos de CPU) e consumo de energia ($\mu Joules$) para as aplicações multi-thread Matrix, Radix, FFT e Djikstra, e para Sha, StringSearch e Basicmath, respectivamente.

Os valores de consumo de energia e desempenho (ciclos de CPU) usados nas figuras 5.7 e 5.8 para cada algoritmo em cada aplicação foram selecionados de uma de suas execuções cujos valores médios das métricas "número de simulações da plataforma"e "hipervolume"correspondiam a mediana entre os valores médios dessas mesmas métricas em todas as suas execuções.

Por exemplo, no algoritmo AbcDE simulando a execução da aplicação Matrix, suas execuções geraram os seguintes valores médios: Número de simulação da plataforma(NSP)= **1689** e hipervolume(HYP)= *369986701*. Considerando as duas execuções com valores mais próximos dessas duas médias (exemplo: execução 1:NSP= **1682** e HYP=369986865; e execução 2:NSP= 1682 e HYP=*369986863*), buscamos a execução mediana, que se situa mais ao centro entre essas duas execuções selecionadas, como a execução mais representativa do algoritmo para a referida aplicação (selecionada a seguinte execução: NSP= **1538** e HYP=*369986864*, onde $1682 < \mathbf{1538} < 1682$ e $369986863 < 369986864 < 369986865$). Ela pode ser designada como a execução mais próxima de uma média de execuções tomando como base de cálculo duas métricas, a de hipervolume(precisão) e a de número de execuções da plataforma(eficiência).

A tabela 5.1 mostra os dados das execuções mais próximas da média para todos os algoritmos e aplicações simuladas.

Foram feitas análises quanto à precisão e à eficiência dos algoritmos. A eficiência foi medida através da métrica de número de simulações da plataforma. O critério de precisão corresponde a quão próximo os algoritmos chegam do conjunto de Pareto do espaço de projeto, em termos de hiper-volume. Quanto maior o valor do hiper-volume, mais preciso será o resultado do algoritmo, mais se aproxima do conjunto de Pareto do espaço de projeto.

A figura 5.9-a mostra os resultados das execuções dos algoritmos em termos de número de simulações da plataforma para as aplicações Matrix, Radix, FFT e Djikstra. A figura 5.9-b mostra os mesmos resultados para as aplicações Sha, Stringsearch e Basicmath. Quanto menor for a quantidade de execuções da plataforma, melhor será o resultado para essa métrica. O algoritmo AbcDE apresentou melhores valores de número de execuções da plataforma MPSoC que os outros dois algoritmos, ABCMOP e MOPSO. Foi verificado que o algoritmo AbcDE

Figura 5.7: Consumo de energia versus Desempenho - Algoritmo AbcDE, MOPSO e ABCMOP para as aplicações Matrix, Radix, FFT e Djikstra.

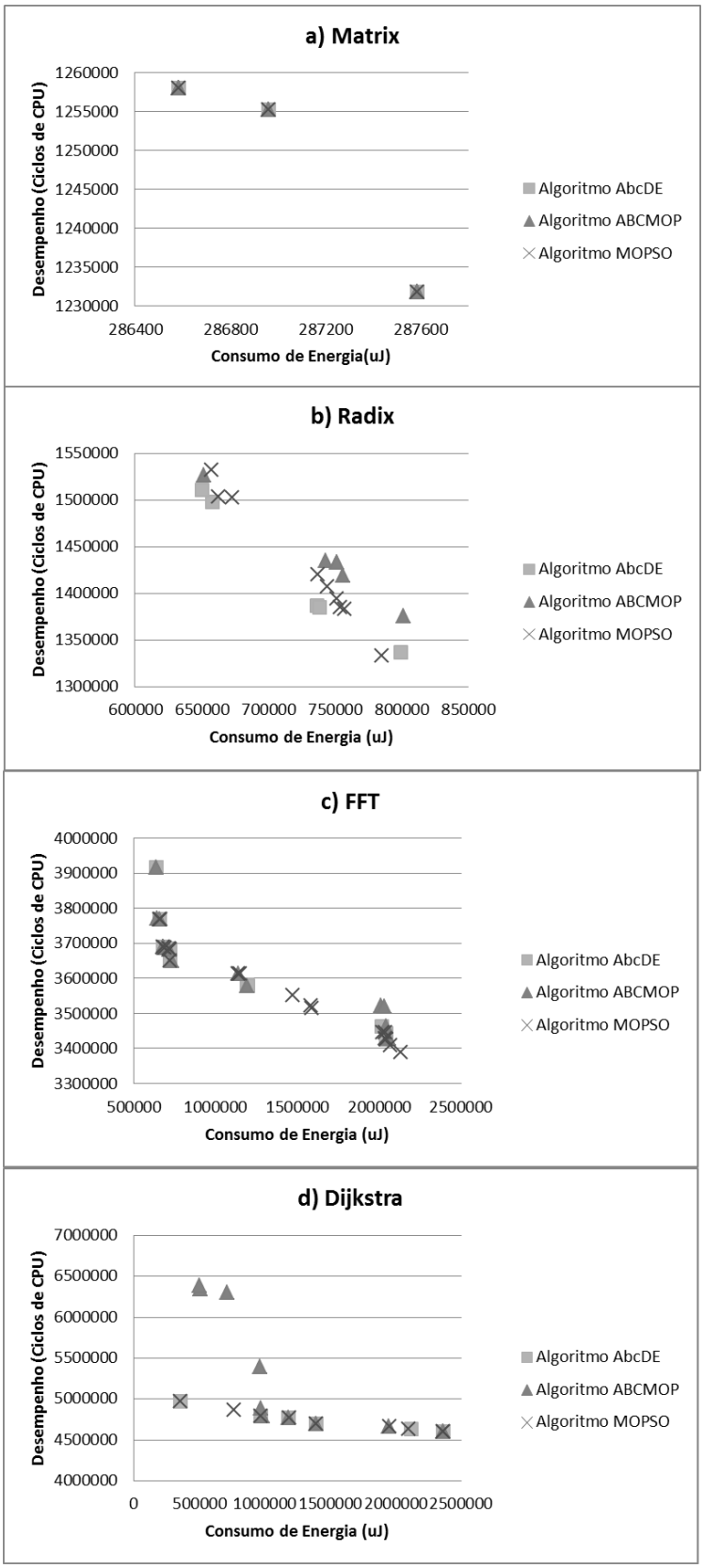


Figura 5.8: Consumo de energia versus Desempenho - Algoritmo AbcDE, MOPSO e ABCMOP para as aplicações Sha, StringSearch e Basicmath.

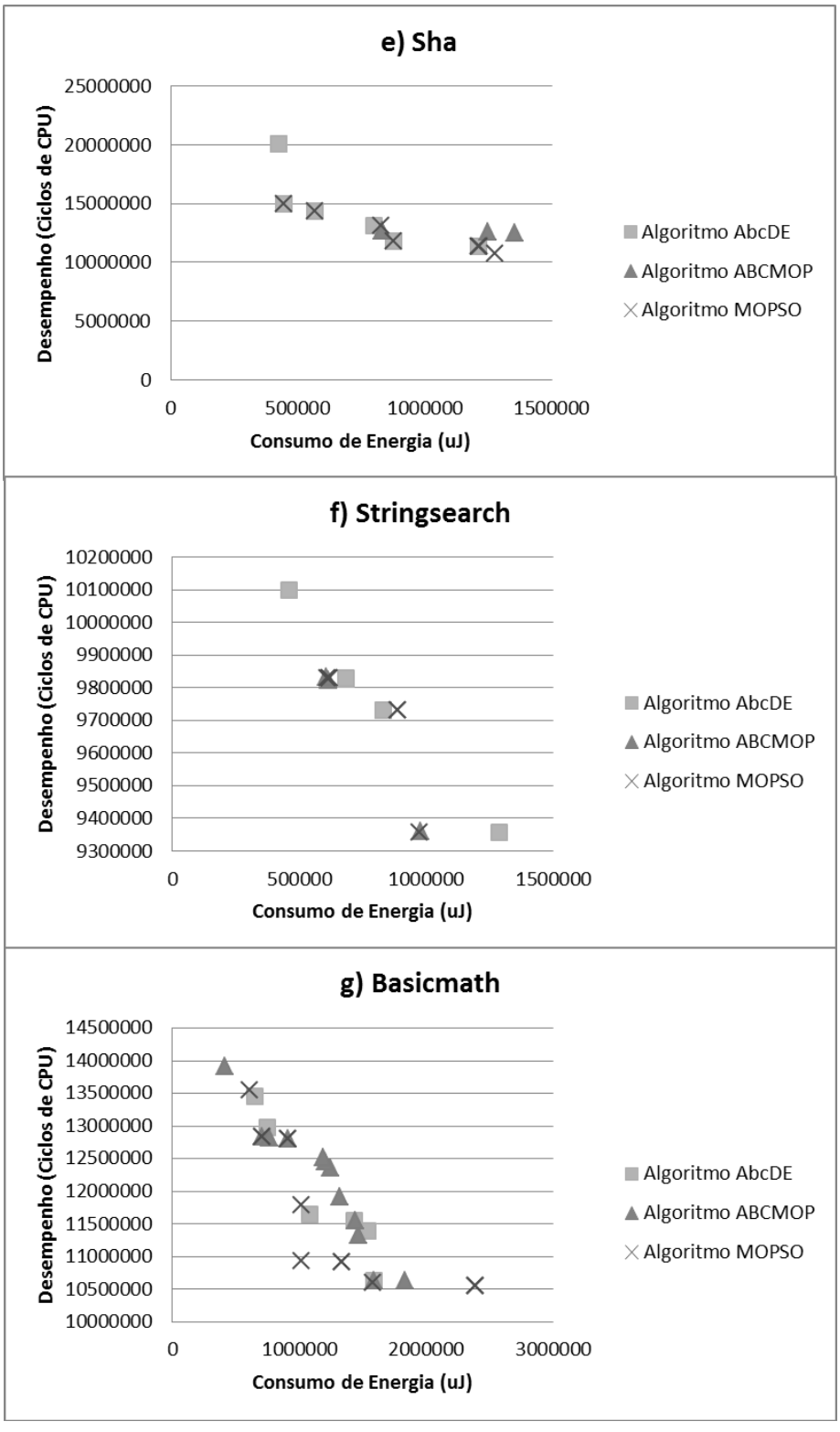


Tabela 5.1: Dados do Pareto set exibidos nas figuras 5.7 e 5.8

ALGORITMO ABCDE													
MATRIX		RADIX		FFT		DJIKSTRA		SHA		STRINGSEARCH		BASICMATH	
Energia	Ciclos CPU	Energia	Ciclos CPU	Energia	Ciclos CPU	Energia	Ciclos CPU	Energia	Ciclos CPU	Energia	Ciclos CPU	Energia	Ciclos CPU
287587	1231723	736671	1386663	676504	3688213	350749	4970994	1210976	11340408	686367	9825228	1589282	10636858
287587	1231723	738211	1385403	676189	3689525	969670	4792234	1210976	11340408	685684	9828278	1082341	11651279
287587	1231723	799554	1336615	712946	3684521	1182220	4773687	1210976	11340408	829928	9730284	1539981	11386416
287587	1231723	658394	1497957	719500	3682367	1182220	4773687	1209208	11368230	831054	9729726	1434416	11554577
287587	1231723	650206	1511517	720735	3653543	1393510	4697379	877729	11834099	461454	10097553	750645	12970868
287587	1231723			658401	3769425	2117955	4631793	799025	13116121	1287703	9358173	654364	13447791
287587	1231723			635126	3915135	2362247	4596865	566050	14339308				
287587	1231723			637130	3915038			446160	14937161				
287587	1231723			1189259	3579658			425049	20073475				
286960	1255255			1195829	3579468			423097	20092626				
286960	1255255			2041440	3425215								
286960	1255255			2011473	3462085								
286582	1257987			2040466	3443125								
286582	1257987												
286582	1257987												
286582	1257987												
286582	1257987												
286582	1257987												
286582	1257987												
286582	1257987												

apresentou uma quantidade média de execuções da plataforma MPSoC em cerca de 37,14% menor que o algoritmo ABCMOP, e em cerca de 37,10 % menor que o algoritmo MOPSO (considerando todas as aplicações dos experimentos).

Outro resultado considerando a eficiência do algoritmo é o percentual de exploração do espaço de projeto. Considerando o número de simulações da plataforma necessárias para a execução do algoritmo e definição de uma configuração ótima para a hierarquia de cache, realizamos experimentos para as aplicações Matrix, Radix, FFT, Dijkstra, Sha, stringsearch e Basicmath. Considerando um número médio de execuções da plataforma para as aplicações chegamos no valor de 2148,4 simulações, e o espaço de projeto total de 1.594.323 diferentes configurações da hierarquia de cache, obtivemos o valor médio de 0,13% para o percentual de exploração do espaço de projeto total.

A figura 5.10 mostra o quanto o algoritmo AbcDE, utilizando reuso de simulações já realizadas, consegue reduzir ainda mais o número de simulações da plataforma. Os números mostram que foi possível reduzir o número médio de execuções da plataforma MPSoC em cerca de 26,61%. Isso representa mais uma melhoria do algoritmo AbcDE sobre os demais. Considerando reuso, o algoritmo AbcDE apresentou uma quantidade média de execuções da plataforma MPSoC em cerca de 54,12% menor que o algoritmo ABCMOP, e em cerca de 54,09 % menor que o algoritmo MOPSO (considerando todas as aplicações dos experimentos).

Ainda considerando a eficiência dos algoritmos para cada aplicação, a figura 5.11-a apresenta o tempo de execução (em horas) dos algoritmos AbcDE, ABCMOP e o MOSPO para as aplicações Matrix, Radix, FFT e Dijkstra. A figura 5.11-b mostra, em horas, o tempo de execução dos mesmos algoritmos para as aplicações Sha, StringSearch e Basicmath.

A figura 5.12 mostra a precisão dos algoritmos em termos da métrica Hiper-volume. A figura 5.12-a mostra os resultados das execuções dos algoritmos em termos de hiper-volume para as aplicações Matrix, Radix, FFT e Djikstra. A figura 5.12-b mostra os mesmos resultados para as aplicações Sha, Stringsearch e Basicmath. Mesmo obtendo uma melhoria significativa em termos eficiência, comparado aos algoritmos ABCMOP e MOPSO, o algoritmo AbcDE não

Figura 5.9: Resultados para número médio de simulações da Plataforma para aplicações Matrix, Radix, FFT e Djikstra (a) e Sha, Stringsearch e Basicmath (b).

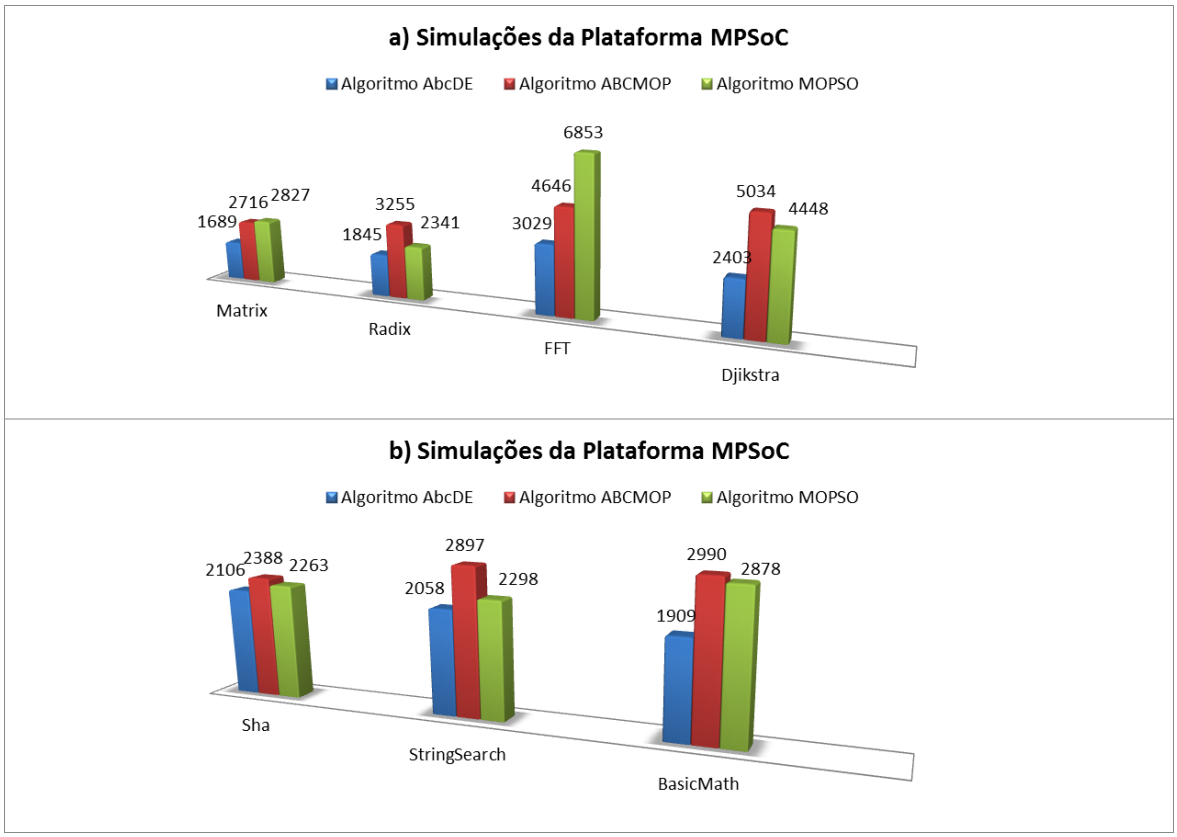


Figura 5.10: Resultado de redução de simulações da plataforma devido ao reuso de simulações.

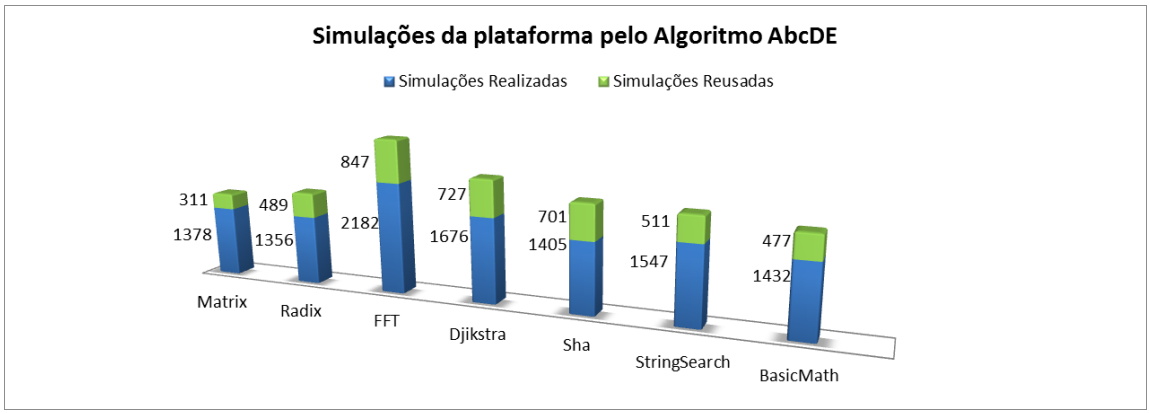
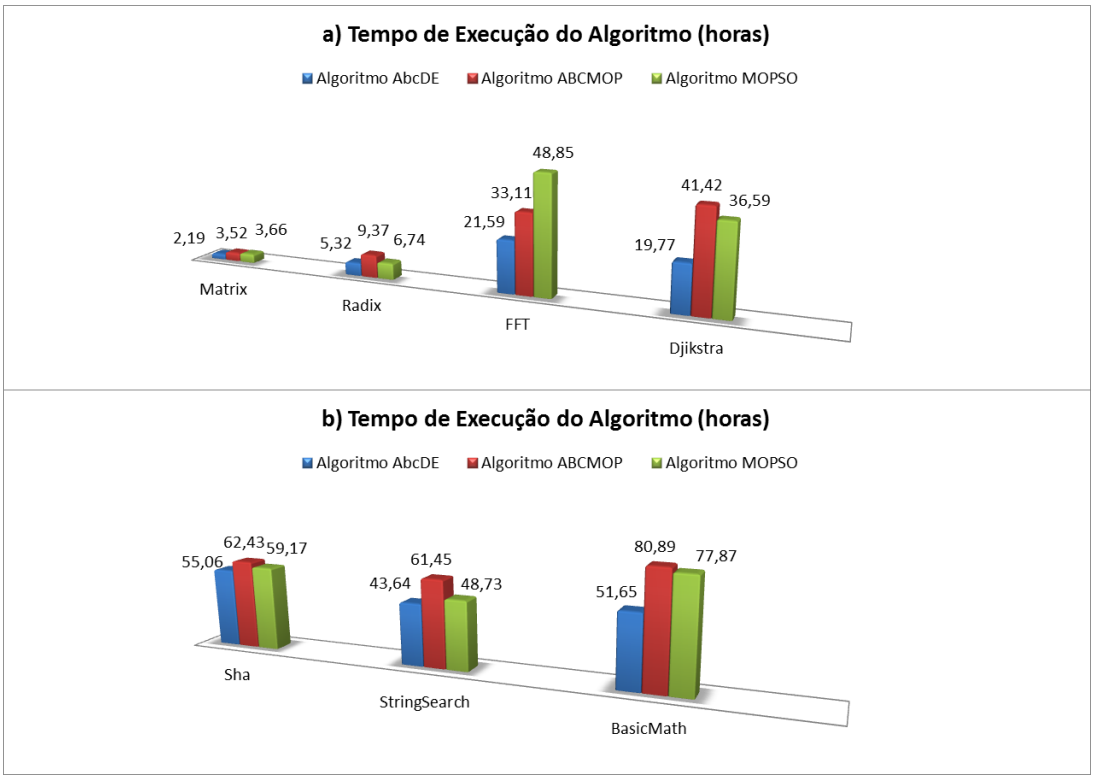


Figura 5.11: Tempo de execução do algoritmos AbcDE, ABCMOP e MOPSO a- para as aplicações Matrix, Radix, FFT e Dijkstra, e b- para as aplicações Sha, Stringsearch e BasicMath.



degradou sua precisão. Considerando hiper-volume, quanto maior o seu valor melhor será o seu resultado. Os números mostram que o algoritmo AbcDE, em termos de hiper-volume, foi em média inferior ao algoritmo ABCMOP em apenas 0,91%, e foi em média superior ao algoritmo MOPSO em apenas 0,66%.

A tabela 5.2 mostra os valores de número de simulações da plataforma e hiper-volume, com seus valores médios e desvio padrão, para as execuções dos algoritmos AbcDE, ABCMOP e MOPSO, simulando a execução das aplicações Matrix, Radix, FFT, Djikstra, Sha, StringSerach e BasicMath na plataforma MPSoC utilizada nos experimentos.

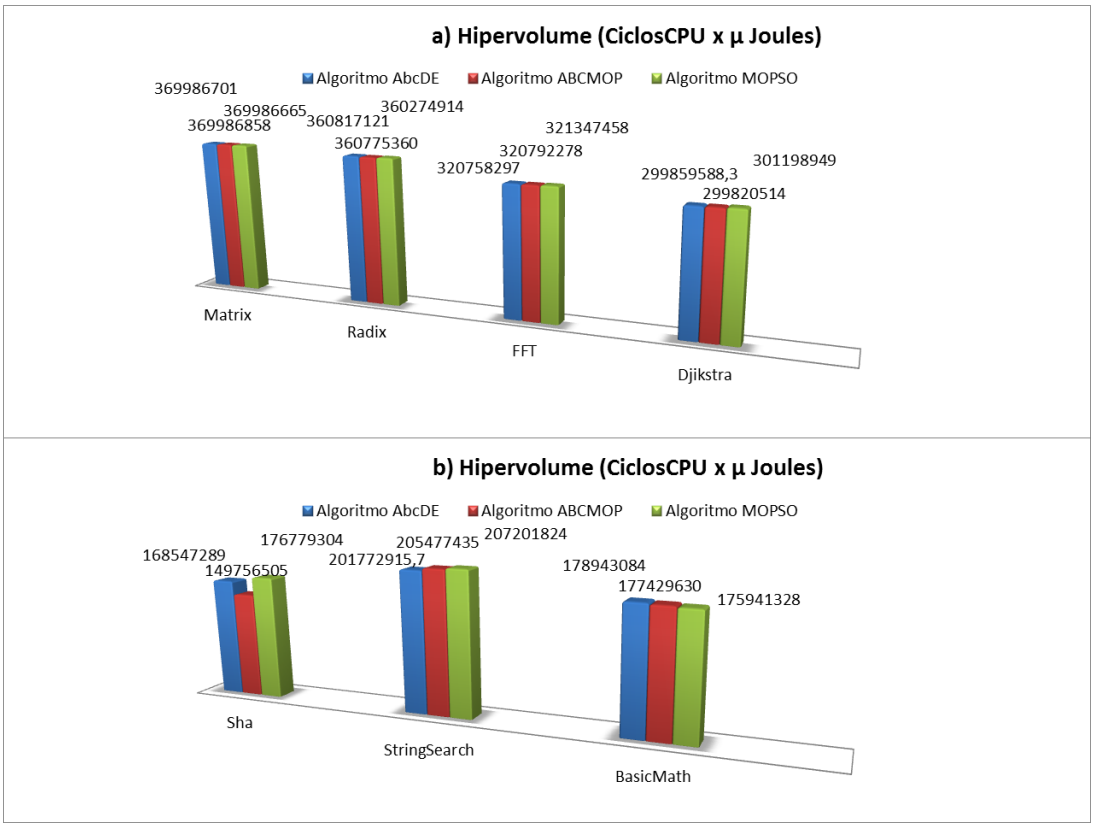
Devido ao aumento do uso de recursos durante a sua execução, em comparação com o algoritmo ABC original, realizamos a análise de uso de recurso de memória para o algoritmo AbcDE e comparamos com o uso de memória dos algoritmos ABCMOP e o MOPSO. Para realizar essa medição utilizamos a ferramenta mallocCount, que é uma ferramenta de análise de uso de memória em tempo de execução. O projeto malloc count foi desenvolvido por Timo Bingamnn como projeto open-source no Karlsruhe Institute of Technology. (BINGAMNN, 2013).

Através da figura 5.13, que é um gráfico de uso de memória no tempo, podemos verificar que o algoritmo AbcDE faz um uso de memória bem superior aos algoritmos ABCMOP e o MOPSO. No início da sua execução, o algoritmo AbcDE faz uso intenso de memória na sua de inicialização, onde fazemos uso das técnicas de DoE para obtenção da população inicial

Tabela 5.2: Dados de valores médios e desvio padrão das execuções dos algoritmos AbcDE. ABCMOP e MOPSO para as simulações das aplicações Matrix, Radix, FFT, Dijkstra, Sha, StringSearch e BasicMath na plataforma MPSoC.

	Algoritmo AbcDE						
	Matrix	**Radix**	**FFT**	**Djikstra**	**Sha**	**StringSearch**	**BasicMath**
Média de simulações da plataforma	1689	1845	3029	2403	2106	2058	1909
Desvio Padrão de simulações da plataforma	280,55	673,74	1057,38	468,62	449,89	547,38	502,58
Média do hipervolume	369986701	360817121	320758297	299859588	168547289	201772916	178943084
Desvio Padrão do hipervolume	532,43	643528,78	265130,38	767223,48	4912071,40	9141031,42	1868933,71
	Algoritmo ABCMOP						
	Matrix	**Radix**	**FFT**	**Djikstra**	**Sha**	**StringSearch**	**BasicMath**
Média de simulações da plataforma	2716	3255	4646	5034	2388	2897	2990
Desvio Padrão de simulações da plataforma	821,85	726,57	800,72	1164,70	1198,28	842,54	610,56
Média do hipervolume	369986858	360274914	320792278	299820514	149756505	205477435	177429630
Desvio Padrão do hipervolume	26,0826	292243,06	271025,65	835817,79	1930589,90	2140012,27	4506394,30
	Algoritmo MOPSO						
	Matrix	**Radix**	**FFT**	**Djikstra**	**Sha**	**StringSearch**	**BasicMath**
Média de simulações da plataforma	2827	2341	6853	4448	2263	2298	2878
Desvio Padrão de simulações da plataforma	961,79	820,42	3062,86	2614,08	908,43	871,19	1117,92
Média do hipervolume	369986665	360775360	321347458	301198949	176779304	207201824	175941328
Desvio Padrão do hipervolume	0,611650	360855,37	76149,17	1438032,69	4622526,54	634201,10	5959518,55

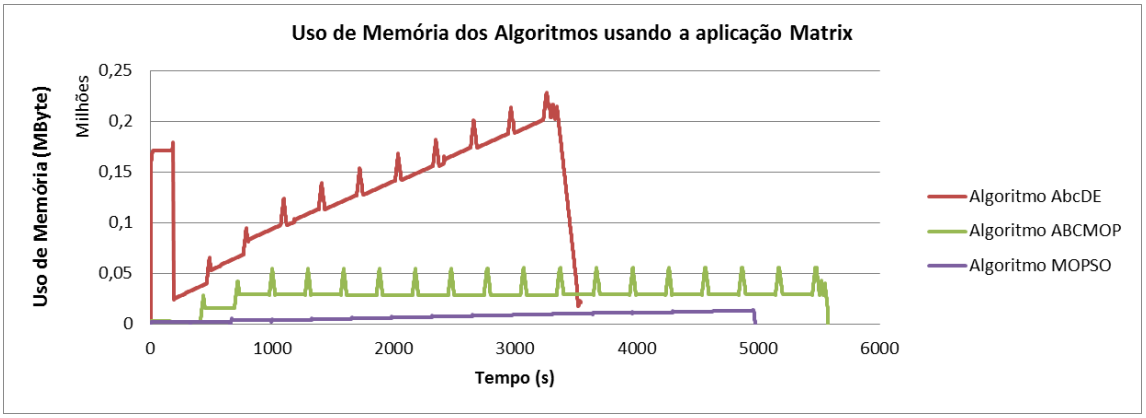
Figura 5.12: Resultados de hiper-volume médio para execução das aplicações Matrix, Radix, FFT e Djikstra (a) e Sha, Stringsearch e Basicmath (b).



balanceada e para o cálculo do impacto dos parâmetros de cache.

Depois dessa fase, nos ciclos de iteração do algoritmo, o uso de memória vai sendo gradativamente aumentado devido ao incremento do objeto *SimulatedFoods* (seção 4.3) após cada simulação da plataforma. Para o algoritmo ABCMOP verifica-se um pequeno incremento apenas nas três primeiras iterações (até 1000 segundos) onde se guarda dos dados de três iterações para uso no critério de parada automático. O algoritmo MOPSO fez um uso bem reduzido de memória comparado aos demais algoritmos.

Figura 5.13: Uso de memória na execução dos algoritmos AbcDE, ABCMOP e o MOPSO utilizando a aplicação Matrix.



5.6 Análise Comparativa com o Estado da Arte

Nessa seção realizamos uma análise qualitativa da abordagem proposta (AbcDE) comparando com as abordagens de configuração estática do estado da arte. O resumo é ilustrado no seguinte quadro comparativo (tabela 5.3):

Tabela 5.3: Análise Comparativa do Estado da Arte com o ABcDE.

Técnica	**Objetivo de Melhoria**	**Evita mínimos locais**	**Arquitetura**	**MPSoC com Coerência de cache**	**Aplicada aos níveis de cache**	**Aplicável a MPSoCs com NoC**	**Escalabilidade no tempo de execução**	**Necessidade de uso de memória**
U-SPaCS	Desempenho e Consumo de Energia	Evita	SoC	Não Aplicável	Apenas aos níveis L1 e L2	Aplicável a SoCs	Possui	Baixo uso de memória
NAWINNE et al. (2015)	Desempenho	Não comprovado	MPSoC	Não considera	Múltiplos níveis	Não	Possui	Baixo uso de memória
DIMSim	Desempenho	Não comprovado	MPSoC	Considera	Apenas aos níveis L1 e L2	Não	Possui	Baixo uso de memória
ABCMOP	Desempenho e Consumo de Energia	Não comprovado	SoC	Não Aplicável	Múltiplos níveis	Aplicável a SoCs	Tempo de execução não otimizado	Baixo uso de memória
ABcDE	Desempenho e Consumo de Energia	Evita	MPSoC	Considera	Múltiplos níveis	Aplicável	Tempo de execução otimizado	Elevado uso de memória

Analisando a tabela 5.3 podemos verificar que apenas uma das técnicas aplicáveis a múltiplos processadores realizaram *tuning* de cache considerando múltiplos objetivos, e foi o algoritmo ABcDE proposto. O ABcDE realiza sua abordagem multi-objetivo considerando simultaneamente os valores medidos de energia e desempenho através de análise de dominância.

Dentre as técnicas analisadas, as únicas abordagens que comprovadamente evitam mínimos locais em seus resultados são o ABCMOP, o ABcDE e o U-SPaCS. O ABCMOP e o AbcDE evitam mínimos locais, pois foram realizados com base em algoritmo ABC que evita mínimos locais por construção.

Apesar de ter sido realizado com base no algoritmo ABC, o ABCMOP foi construído para ser aplicado em ambiente SoC e não introduziu melhorias à eficiência do algoritmo para redução do seu tempo médio de execução (que é o principal problema quando aplicando o

algoritmo evolucionário a arquiteturas MPSoC).

Alguns trabalhos de reconfiguração de cache para arquiteturas MPSoC são abordados de forma a tratarem a arquitetura alvo como sendo múltiplos SoCs sem considerar coerência de cache, como o trabalho de Nawinne (NAWINNE et al., 2015). O ABcDE é aplicável a plataformas com múltiplos processadores baseados em barramento e NoC, e considera coerência de cache.

As técnicas ABCMOP, AbcDE e a Nawinne2015 foram concebidas para uso com múltiplos níveis de hierarquia de cache. As demais técnicas são atreladas à níveis de cache predefinidos.

Algumas técnicas aplicáveis a plataformas com multiprocessadores são aplicáveis especificamente para às técnicas baseadas em barramento, como os trabalhos de Nawinne (NAWINNE et al., 2015) e o DIMSim (HAQUE et al., 2012). Poucos trabalhos são aplicáveis à plataformas multi-processadores baseadas em NoC. Apenas o algoritmo AbcDE é também aplicável a este tipo de plataforma.

Também verificamos que os trabalhos de Nawinne (NAWINNE et al., 2015) e o DIMSim (HAQUE et al., 2012) comprovam ser abordagens que possuem escalabilidade quanto ao seu tempo de execução. Isso quer dizer que essas técnicas podem ser executadas, sem grande impacto de tempo de execução, para espaços de projeto de cache muito amplos. O mesmo não pode ser comprovado para abordagem ABCMOP, devido ao fato de trabalhar com simulação da arquitetura e de não possuir otimização de tempo de execução para a meta-heurística utilizada. A abordagem proposta AbcDE teve seu tempo de execução otimizado através da redução de simulações da plataforma, melhorando consideravelmente a sua escalabilidade quanto a tempo de execução do algoritmo.

A abordagem proposta AbcDE utilizou técnicas de DoE para realizar sua melhoria de tempo de execução, o que implica em um aumento da utilização de memória para sua implementação. Isso tornou a técnica aplicável apenas em ambientes de processamento com elevada disponibilidade de memória RAM. As demais técnicas não necessitam de tal grandeza de uso de memória e podem ser aplicadas em ambiente computacionais com pouca memória.

6

Conclusão

Nesse trabalho foi apresentada a técnica AbcDE, que é uma abordagem para exploração de espaço de projeto de cache L1 para plataformas MPSoCs que usa o algoritmo ABCs (Colônia Artificial de Abelhas), em modo multi-objetivo (melhoria de desempenho e consumo de energia simultaneamente) e que faz uso de técnicas do DoE para melhoria da eficiência da busca do algoritmo ABC utilizado. De DoE foram utilizadas as técnicas de população balançeada e a de análise estatística do modelo de efeitos. Com o uso da população balançeada buscamos iniciar o algoritmo com uma população de fontes de alimento mais representativa e bem distribuida no espaço de projeto. A técnica de análise estatística do modelo de efeitos foi utilizada para se calcular o graus de importância de cada um dos parâmetros da hierarquia de cache, visando melhorar a eficiência da busca global do algoritmo utilizado.

Nos experimentos com o algoritmo AbcDE foram utilizados os *benchmarks* Splash2 (fft, radix e multiplicação de matrizes) e ParMibench (Dijkstra) utilizando cache em nível único (L1). Os resultados do algoritmo AbcDE foram comparados com a execução do algoritmo ABC multi-objetivo. Obtivemos como resultado diversas configurações de cache L1 no conjunto *Pareto front*, reduzindo o tempo de execução em 42,3% em média para uma plataforma MPSoC de quatro processadores. Esses resultados foram reunidos e publicados em um artigo completo do ASAP 2016 (*The 27th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors*, de 6 a 8 de julho de 2016, em Londres-Inglaterra) (SANTOS; BARROS; AZIZ, 2016).

Também avaliamos a abordagem AbcDE executando as aplicações dos *benchmarks* previamente citados em conjunto com as aplicações do benchmark ParMibench (Sha, Stringsearch e Basicmath) para hierarquia de cache em multinível (L1 e L2). Foram obtidas configurações de cache dentro do *Pareto Front* apresentando uma quantidade média de execuções da plataforma MPSoC em cerca de 37,14% menor que o algoritmo ABCMOP, e em cerca de 37,10 % menor que o algoritmo MOPSO (considerando todas as aplicações dos experimentos). Mesmo obtendo uma melhoria significativa em termos eficiência, comparado aos algoritmos ABCMOP e MOPSO, o algoritmo AbcDE não degradou sua precisão. O algoritmo AbcDE, em termos de hiper-volume, foi em média inferior ao algoritmo ABCMOP em apenas 0,91%, e em média

superior ao algoritmo MOPSO em apenas 0,66%.

O algoritmo AbcDE possibilitou encontrar configurações de cache dentro do *Pareto Front* explorando, em média, cerca de 0,13% do espaço total de projeto em múltiplos níveis de cache. Portanto, o algoritmo AbcDE pode ser aplicado com sucesso para exploração de espaço de projeto de caches multi-nível em plataformas MPSoC para otimização de desempenho e consumo de energia. Adicionalmente, o algoritmo AbcDE permite a exploração de espaço de projeto de cache em plataformas MPSoC baseada em NoC e obteve resultados satisfatórios (considerando os algoritmos utilizados e análise comparativa) para otimizar consumo de energia e desempenho de cache.

Considerando novos espaços de pesquisas na área deste trabalho, podemos citar os seguintes trabalhos futuros:

- Realizar os mesmos experimentos já propostos, porém considerando incluir outras aplicações dos benchmarks utilizados, e/ou utilizando outros benchmarks para execução concorrente de tarefas;

- Realizar estudo sobre a possibilidade de uso da técnica de DoE com análise de efeito de múltiplos parâmetros em conjunto. Esse experimento não poderia ter sido feito neste trabalho sem uma criteriosa análise do efeito de mudança simultânea em mais de um parâmetro por vez durante as buscas locais e globais no algoritmo ABC, pois o mesmo foi concebido considerando o conceito de vizinhança sendo realizado alterando apenas o valor de um parâmetro por vez. Alterar mais de um parâmetro por vez implicaria em uma mudança que alteraria fortemente as buscas locais e globais do algoritmo original, podendo gerar distorções muito impactantes na exploração e explotação originais do algoritmo ABC. Usar análise de efeitos do DoE para apenas um parâmetro proporcionou excelentes resultados, conforme observados nesse trabalho. Considerar a análise de efeitos simultâneos de dois parâmetros geraria também uma sobre carga computacional significativa que precisaria também ser avaliada a viabilidade;

- O algoritmo AbcDE também pode ser aplicado na análise multi-objetiva não somente de parâmetros de cache em uma plataforma MPSoC. É possível se fazer essa análise multi-objetiva considerando os parâmetros de outros circuitos da plataforma como processadores (ex. número de processadores, tamanho das *issue* do processador) e NoC (ex. tipos de algoritmo de roteamento). Esse tipo de trabalho tem sido explorado com outras abordagens no estado da arte (MARIANI et al., 2012);

- O algoritmo AbcDE faz uso de grande quantidade de memória no tempo. Realizar um estudo de como o algoritmo poderia funcionar com uso de pouca memória é bem importante para que a disponibilidade de memória não limite a aplicação do algoritmo em ambientes de pouca memória primária;

- Utilizar as contribuições do uso das técnicas de DoE para melhoria da busca global, aplicando-as para outros algoritmos de partícula de enxame e/ou para outras metaheurísticas ou algoritmos evolucionários, que fazem inicialização de parâmetros. No final comparar os resultados com o trabalho corrente.

Referências

ADEGBIJA, T.; GORDON-ROSS, A. Energy-efficient phase-based cache tuning for multimedia applications in embedded systems. In: CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE (CCNC), 2014 IEEE 11TH. **Anais...** [S.l.: s.n.], 2014. p.89–94.

ALSAFRJALANI, M. H.; GORDON-ROSS, A. Quality of service-aware, scalable cache tuning algorithm in consumer-based embedded devices. In: INTERNATIONAL GREAT LAKES SYMPOSIUM ON VLSI (GLSVLSI), 2016. **Anais...** [S.l.: s.n.], 2016. p.357–360.

ALVES, M. A. Z. **Increasing Energy Efficience of Processor Caches Via Line Usage Predictors**. 2014. Tese (Doutorado em Ciência da Computação) — .

AZIZ, A. et al. Balanced prefetching aggressiveness controller for NoC-based multiprocessor. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 2014 27TH SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2014. p.1–7.

BAKER, J. E. Reducing Bias and Inefficiency in the Selection Algorithm. In: SECOND INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS ON GENETIC ALGORITHMS AND THEIR APPLICATION, Hillsdale, NJ, USA. **Proceedings...** L. Erlbaum Associates Inc., 1987. p.14–21.

BINGAMNN, T. **Malloc-Count - Tools for Runtime Memory Usage Analysis and Profiling**. URL: https://panthema.net/2013/malloc-count/.

BITAR, P. A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors. In: DUBOIS, M.; THAKKAR, S. (Ed.). **Cache and Interconnect Architectures in Multiprocessors**. [S.l.]: Springer US, 1990. p.37–52.

BOUSSAïD, I.; LEPAGNOT, J.; SIARRY, P. A survey on optimization metaheuristics. **Information Sciences**, [S.l.], v.237, n.0, p.82 – 117, 2013. Prediction, Control and Diagnosis using Advanced Neural Computations.

BURGER, D.; AUSTIN, T. M. The SimpleScalar Tool Set, Version 2.0. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.25, n.3, p.13–25, June 1997.

COELLO, C. A. C.; LECHUGA, M. S. MOPSO: a proposal for multiple objective particle swarm optimization. In: EVOLUTIONARY COMPUTATION, 2002. CEC '02. PROCEEDINGS OF THE 2002 CONGRESS ON. **Anais...** [S.l.: s.n.], 2002. v.2, p.1051–1056.

CREPINSEK, M.; LIU, S.-H.; MERNIK, M. Exploration and Exploitation in Evolutionary Algorithms: a survey. **ACM Comput. Surv.**, New York, NY, USA, v.45, n.3, p.35:1–35:33, 2013.

EDLER, J.; HILL, M. **Dinero IV Trace-Driven Uniprocessor Cache Simulator**. URL: http://pages.cs.wisc.edu/ markhill/DineroIV/.

EL-GHAZALI TALBI FAROUK YALAOUI, L. A. e. **Metaheuristics for Production Systems**. 1.ed. [S.l.]: Springer International Publishing, 2016. (Operations Research/Computer Science Interfaces Series 60).

ESLAMI, M. et al. A Survey of the State of the Art in Particle Swarm Optimization. **Research journal of Applied Sciences, Engineering and Technology**, [S.l.], v.4, n.9, p.1181–1197, 2012.

FARIAS, M. et al. An Ant Colony metaheuristic for energy aware application mapping on NoCs. In: ELECTRONICS, CIRCUITS, AND SYSTEMS (ICECS), 2013 IEEE 20TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2013. p.365–368.

FISHER, R. et al. **Hypermedia Image Processing Reference**. URL: http://homepages.inf.ed.ac.uk/rbf/HIPR2/stretch.htm.

GOLDSCHMIDT, S. R.; HENNESSY, J. L. The Accuracy of Trace-driven Simulations of Multiprocessors. **SIGMETRICS Perform. Eval. Rev.**, New York, NY, USA, v.21, n.1, p.146–157, June 1993.

GORDON-ROSS, A.; VAHID, F.; DUTT, N. Fast Configurable-Cache Tuning With a Unified Second-Level Cache. **Very Large Scale Integration (VLSI) Systems, IEEE Transactions on**, [S.l.], v.17, n.1, p.80–91, Jan 2009.

HAQUE, M. S. et al. DIMSim: a rapid two-level cache simulation approach for deadline-based mpsocs. In: EIGHTH IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, New York, NY, USA. **Proceedings...** ACM, 2012. p.151–160. (CODES+ISSS '12).

HARIS, P. A.; GOPINATHAN, E.; ALI, C. K. Artificial Bee Colony and Tabu Search Enhanced TTCM Assisted MMSE Multi-User Detectors for Rank Deficient SDMA-OFDM System. **Wirel. Pers. Commun.**, [S.l.], v.65, n.2, p.425–442, July 2012.

HEDAYATZADEH, R. et al. A multi-objective Artificial Bee Colony for optimizing multi-objective problems. In: ADVANCED COMPUTER THEORY AND ENGINEERING (ICACTE), 2010 3RD INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2010. v.5, p.V5–277–V5–281.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Fifth Edition**: a quantitative approach. Fifth Edition.ed. [S.l.]: Elsevier Inc., 2012.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Organization and Design - the Hardware/Software Interface ARM Edition**. First Edition.ed. [S.l.]: Elsevier Inc., 2017.

IRANI, R.; NASIMI, R. Application of artificial bee colony-based neural network in bottom hole pressure prediction in underbalanced drilling. **Journal of Petroleum Science and Engineering**, [S.l.], v.78, n.1, p.6–12, 2011.

ISSHIKI, T. Trace-Driven Workload Simulation for MPSoC Software Performance Estimation. In: LEUPERS, R.; TEMAM, O. (Ed.). **Processor and System-on-Chip Simulation**. [S.l.]: Springer US, 2010. p.325–340.

JAIN, R. **The Art of Computer Systems Performance Analysis**: techniques for experimental design, measurement, simulation, and modeling. [S.l.]: Wiley, 1991. (Wiley professional computing).

KARABOGA, D. **An Idea Based on Honey Bee Swarm for Numerical Optimization**. [S.l.]: Erciyes University, Engineering Faculty, Computer Engineering Department, 2005. White Paper.

KARABOGA, D.; AKAY, B. A comparative study of Artificial Bee Colony algorithm. **Applied Mathematics and Computation**, [S.l.], v.214, n.1, p.108 – 132, 2009.

KARABOGA, D.; AKAY, B. A modified Artificial Bee Colony (ABC) algorithm for constrained optimization problems. **Applied Soft Computing**, [S.l.], v.11, n.3, p.3021 – 3031, 2011.

KARABOGA, D.; BASTURK, B. Artificial Bee Colony ABC Optimization Algorithm for Solving Constrained Optimization Problems. , [S.l.], p.789–798, 2007.

KARABOGA, D.; BASTURK, B. A Powerful and Efficient Algorithm for Numerical Function Optimization: artificial bee colony (abc) algorithm. **J. of Global Optimization**, Hingham, MA, USA, v.39, n.3, p.459–471, nov 2007.

KARABOGA, D.; BASTURK, B. On the Performance of Artificial Bee Colony (ABC) Algorithm. **Appl. Soft Comput.**, [S.l.], v.8, n.1, p.687–697, 2008.

KARABOGA, D. et al. A comprehensive survey: artificial bee colony (abc) algorithm and applications. **Artificial Intelligence Review**, [S.l.], v.42, n.1, p.21–57, 2014.

KAZADI, S. T. **Model Independent Economics Based on Swarm Engineering**. [S.l.]: Nova Science Publishers, 2011. 233-247p. v.18.

KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: NEURAL NETWORKS, 1995. PROCEEDINGS., IEEE INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 1995. v.4, p.1942–1948.

KNOWLES, J.; CORNE, D. On metrics for comparing nondominated sets. In: EVOLUTIONARY COMPUTATION, 2002. CEC '02. PROCEEDINGS OF THE 2002 CONGRESS ON. **Anais...** [S.l.: s.n.], 2002. v.1, p.711–716.

KRUSKAL, W. H.; WALLIS, W. A. Use of Ranks on One Criterion Variance Analysis. **Journal of the American Statistical Association**, [S.l.], v.47, p.583–621, 1952.

LALWANI, S. et al. A Comprehensive Survey: applications of multi-objective particle swarm optimization (mopso) algorithm. **Transactions on Combinatorics**, [S.l.], v.2, n.1, p.39–101, 2013.

MARIANI, G. et al. OSCAR: an optimization methodology exploiting spatial correlation in multicore design spaces. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.31, n.5, p.740–753, May 2012.

MARTI, L. et al. An approach to stopping criteria for multi-objective optimization evolutionary algorithms: the mgbm criterion. In: EVOLUTIONARY COMPUTATION, 2009. CEC '09. IEEE CONGRESS ON. **Anais...** [S.l.: s.n.], 2009. p.1263–1270.

MICHANAN, J.; DEWRI, R.; RUTHERFORD, M. J. Understanding the power-performance tradeoff through Pareto analysis of live performance data. In: GREEN COMPUTING CONFERENCE (IGCC), 2014 INTERNATIONAL. **Anais...** [S.l.: s.n.], 2014. p.1–8.

MONTGOMERY, D. C. **Design and Analysis of Experiments**. [S.l.]: John Wiley & Sons, 2013.

MONTGOMERY, D. C. **Applied Statistics and Probability for Engineers**. [S.l.]: Wiley, 2014.

MOSTAGHIM, S.; BRANKE, J.; SCHMECK, H. Multi-objective Particle Swarm Optimization on Computer Grids. In: ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 9., New York, NY, USA. **Proceedings...** ACM, 2007. p.869–875. (GECCO '07).

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. Architecting Efficient Interconnects for Large Caches with CACTI 6.0. **Micro, IEEE**, [S.l.], v.28, n.1, p.69–79, Jan 2008.

NAIDU, K.; MOKHLIS, H.; BAKAR, A. Multiobjective optimization using weighted sum Artificial Bee Colony algorithm for Load Frequency Control. **International Journal of Electrical Power and Energy Systems**, [S.l.], v.55, n.0, p.657 – 667, 2014.

NAWINNE, I. et al. Hardware-based fast exploration of cache hierarchies in application specific MPSoCs. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION (DATE), 2014. **Anais...** [S.l.: s.n.], 2014. p.1–6.

NAWINNE, I. et al. Exploring Multi-Level Cache Hierarchies in Application Specific MPSOCs. **Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on**, [S.l.], v.PP, n.99, p.1–1, 2015.

OSMAN, I. H.; LAPORTE, G. Metaheuristics: a bibliography. **Annals of Operations Research**, [S.l.], v.63, n.5, p.511–623, 1996.

PACHECO, P. **An Introduction to Parallel Programming**. 1st.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

RAHKAR-FARSHI, T.; KESEMEN, O.; BEHJAT-JAMAL, S. Multi hyperbole detection on images using modified artificial bee colony (ABC) for multimodal function optimization. In: SIGNAL PROCESSING AND COMMUNICATIONS APPLICATIONS CONFERENCE (SIU), 2014. **Anais...** [S.l.: s.n.], 2014. p.894–898.

RAWLINS, M.; GORDON-ROSS, A. A Cache Tuning Heuristic for Multicore Architectures. **Computers, IEEE Transactions on**, [S.l.], v.62, n.8, p.1570–1583, Aug 2013.

RYAN, T. P. **Sample size determination and power**. [S.l.: s.n.], 2013. (Wiley series in probability and statistics).

SANTOS, C.; SILVA-FILHO, A. Bee colony algorithm applied to memory architecture exploration intended for energy reduction. In: INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 2014 27TH SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2014. p.1–7.

SANTOS, M. V.; BARROS, E.; AZIZ, A. A MPSoC Cache Design Space Exploration Approach Based on ABC Algorithm to Optimize Energy Consumption and Performance. In: APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS (ASAP), 2016 27TH IEEE COMPUTER SOCIETY. **Anais...** [S.l.: s.n.], 2016.

SILVA-FILHO, A.; CORDEIRO, F. DoE applied to two-level memory hierarchies for energy consumption reduction. In: SYSTEMS, MAN, AND CYBERNETICS (SMC), 2011 IEEE INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2011. p.3084–3089.

SILVA-FILHO, A. et al. Heuristic for Two-Level Cache Hierarchy Exploration Considering Energy Consumption and Performance. In: VOUNCKX, J.; AZEMARD, N.; MAURINE, P. (Ed.). **Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation**. [S.l.]: Springer Berlin Heidelberg, 2006. p.75–83. (Lecture Notes in Computer Science, v.4148).

SILVA-FILHO, A. et al. An Intelligent Mechanism to Explore a Two-Level Cache Hierarchy Considering Energy Consumption and Time Performance. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2007. SBAC-PAD 2007. 19TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2007. p.177–184.

STALLINGS, W. **Computer Organization and Architecture - Designing for Performance**. 9th.ed. San Francisco, CA, USA: Pearson Education, Inc., 2012.

TAN, K.; LEE, T.; KHOR, E. Evolutionary algorithms with dynamic population size and local exploration for multiobjective optimization. **Evolutionary Computation, IEEE Transactions on**, [S.l.], v.5, n.6, p.565–588, Dec 2001.

VIANA, P. et al. A Table - based Method for Single-Pass Cache Optimization. In: ACM GREAT LAKES SYMPOSIUM ON VLSI, 18. **Proceedings...** [S.l.: s.n.], 2008. (GLSVLSI'08).

WANG, W.; MISHRA, P.; RANKA, S. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In: DESIGN AUTOMATION CONFERENCE (DAC), 2011 48TH ACM/EDAC/IEEE. **Anais...** [S.l.: s.n.], 2011. p.948–953.

XINYI, L.; ZUNCHAO, L.; LIQIANG, L. An Artificial Bee Colony Algorithm for Multi-objective Optimization. In: INTELLIGENT SYSTEM DESIGN AND ENGINEERING APPLICATION (ISDEA), 2012 SECOND INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2012. p.153–156.

YU, W.-j.; ZHANG, J.; CHEN, W.-n. Adaptive Artificial Bee Colony Optimization. In: ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 15., New York, NY, USA. **Proceedings...** ACM, 2013. p.153–158. (GECCO '13).

YURTKURAN, A.; EMEL, E. An adaptive artificial bee colony algorithm for global optimization. **Applied Mathematics and Computation**, [S.l.], v.271, p.1004 – 1023, 2015.

ZANG, W.; GORDON-ROSS, A. A single-pass cache simulation methodology for two-level unified caches. In: PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE (ISPASS), 2012 IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2012. p.168–177.

ZANG, W.; GORDON-ROSS, A. A Survey on Cache Tuning from a Power/Energy Perspective. **ACM Computing Survey**, New York, NY, USA, v.45, n.3, p.32:1–32:49, July 2013.

ZHANG, C.; VAHID, F.; LYSECKY, R. A Self-tuning Cache Architecture for Embedded Systems. **ACM Trans. Embed. Comput. Syst.**, New York, NY, USA, v.3, n.2, p.407–425, May 2004.

ZOU, W. et al. Solving Multiobjective Optimization Problems Using Artificial Bee Colony Algorithm. **Discrete Dynamics in Nature and Society**, [S.l.], p.37, August 2011.