Análise do Consumo de Energia em STMs e uma Plataforma de Simulação Multiprocessada com Abstração Híbrida

Este exemplar corresponde à redação da Dissertação apresentada para a Banca Examinadora antes da defesa da Dissertação.

Campinas, 26 de Outubro de 2010.

Prof. Dr. Sandro Rigo (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Silvania Renata de Jesus Ribeiro Cirilo – CRB8 /6592

Moreira, João Batista Corrêa Gomes
M813a Análise do consumo de energia em STMs e uma plataforma de simulação multiprocessada com abstração híbrida/João Batista Corrêa Gomes Moreira-- Campinas, [S.P. : s.n.], 2010.

Orientador : Sandro Rigo
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computadores. 2. Consumo de energia. 3. Memória transacional. 4. Simulação (Computadores). I. Rigo, Sandro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Power consumption analysis of STMs and a hybrid abstraction simulation platform.

Palavras-chave em inglês (Keywords): 1. Computer architecture. 2. Power consumption. 3. Transactional memory. 4. Simulation (Computers).

Área de concentração: Arquitetura de computadores.

Titulação: Mestre em Ciência da Computação.

Banca examinadora: Prof. Dr. Sandro Rigo (IC-UNICAMP)
Prof. Dr. Mário Lúcio Côrtes (IC – UNICAMP)
Profa. Dra. Nahri Balesdent Moreano (UFMS)

Data da defesa: 26/10/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação.

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 26 de outubro de 2010, pela Banca examinadora composta pelos Professores Doutores:

Profª. Drª. Nahri Balesdent Moreano
DCT / UFMS

Prof. Dr. Mario Lúcio Côrtes
IC / UNICAMP

Prof. Dr. Sandro Rigo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Análise do Consumo de Energia em STMs e uma Plataforma de Simulação Multiprocessada com Abstração Híbrida

João Batista Corrêa Gomes Moreira

Outubro de 2010

Banca Examinadora:

- Prof. Dr. Sandro Rigo (Orientador)
- Prof. Dr. Mario Lúcio Côrtes
  Instituto de Computação - UNICAMP
- Prof[a]. Dr[a]. Nahri Balesdent Moreano
  Departamento de Computação e Estatística - UFMS
- Prof. Dr. Paulo César Centoducatte (Suplente)
  Instituto de Computação - UNICAMP
- Prof. Dr. Luiz Cláudio Villar dos Santos (Suplente)
  Departamento de Informática e Estatística - UFSC

Resumo

O surgimento das novas arquiteturas multiprocessadas introduziu novos desafios ao desenvolvimento de software. Dentre estes desafios está a dificuldade de realizar a sincronização adequada entre os fluxos de execução. Para solucionar este problema, novos mecanismos de sincronização com abstrações mais simplificadas tem sido propostos. Seguindo esta corrente, as Memórias Transacionais surgem como uma promissora alternativa aos mecanismos de sincronização tradicionais.

Por se tratar de uma alternativa recentemente proposta, pouco se conhece a respeito dos efeitos no consumo de energia devido ao uso de Memórias Transacionais. Este trabalho apresenta um estudo comparativo entre os consumos de energia observados na execução do benchmark STAMP com usos de um sistema STM (Memória Transacional em Software) e de sincronização baseada em *locks*. Os resultados obtidos demonstram que a STM apresentou um desempenho inferior aos *locks* no que diz respeito ao consumo de energia, apresentando um consumo médio três vezes maior. Também foi avaliada a influência das penalidades decorrentes do uso de *locks* no consumo de energia, mostrando que, em sistemas cujo custo de falha na aquisição de um *lock* supera dez mil ciclos, a aplicação de STMs passa a ser uma abordagem competitiva.

Durante os testes com Memórias Transacionais tornou-se clara a necessidade de ferramentas de simulação que possibilitam projetos de hardware e testes de software de forma mais ágil. Este trabalho descreve a implementação de uma plataforma de simulação para estimar o consumo de energia com abstração híbrida obtida a partir da integração de processadores funcionais que são gerados através da linguagem ArchC com a plataforma MPARM (que possui precisão de ciclos). Esta implementação atingiu ganhos de desempenho médios de até 2.1 vezes, com um máximo de 2.9 vezes. Imprecisões obtidas nas estimativas de consumo de energia puderam ser estatisticamente corrigidas através da aplicação de métodos de regressão linear, apresentando erros médios de 5,85%, sendo o erro mínimo e máximo de 0,87% e 19,6%, respectivamente.

Abstract

The advent of the contemporary multiprocessor architectures has challenged software development. In order to overcome the hurdle of properly ordering the execution and data flows, new synchronization methods with simplified abstraction have been proposed. In this context, Transactional Memories have emerged as an alternative to traditional synchronization methods.

Little is known about the effects on power consumption due to the use of transactional memories since it is a recently proposed alternative. This work compares the power consumption of the STAMP benchmark execution when using a STM system and a lock-based implementation. The results show that the STM implementation presented a worse performance, consuming three times more energy in avarage. In addition, the penalties deriving from the employment of locks in power consumption were assessed, indicating that, in systems where a failure in lock acquisition costs more than ten thousand cycles, the use of STMs becomes a competitive approach.

The experiments with Transactional Memories executed during the first stage of this research indicated that faster simulation tools for hardware design and software testing are needed. Hence, this work describes an implementation of a simulation platform, built using hybrid abstraction level, that is able to estimate power consumption. The platform is the result of integrating functional processors described in the ArchC language with the MPARM platform, which is cycle-based. The implementation displays an average performance speedup of 2.1 and a maximum of 2.9. Inaccuracies due to power consumption estimation could be statistically adjusted by applying corrections based on linear regression. The model carries an average error of 5.85% with a maximum of 19.6% and minimum of 0.86%.

Agradecimentos

Agradeço, inicialmente, ao meu orientador, Professor Doutor Sandro Rigo, pela confiança depositada no início deste trabalho e pelo auxílio prestado nos vários momentos de dificuldade.

Agradeço à minha namorada, Bárbara Lopes Moraes, por todo o amor, paciência e compreensão.

Agradeço à minha família, pelo amor e pelo apoio incondicionais. À minha mãe, Maria do Rosário Gentil Corrêa, por ter me ensinado os valores que hoje me guiam; ao meu pai, João Batista Gomes Moreira, pelo grande exemplo, incentivo e confiança; à minha segunda mãe, Aparecida Luisa da Silva, pelo amor sincero e espontâneo; à minha irmã, Eloína Corrêa Gomes Moreira de Mendonça Telho, a primeira amiga que fiz neste mundo; a Frederico Mendonça Telho, por todos os conselhos e conversas. Agradeço também aos familiares que não puderam estar sempre ao meu lado, Tias Ana Amélia, Conceição e Dulce, Tio Basinho, Tio Gouveia (a quem, sem dúvida, devo mais da metade do meu interesse pelo mundo), Avós Eloína e Lázara, Avôs João e Adherbal, primos e primas.

Aos grandes amigos de Goiânia, por estarem sempre por perto: Gustavo Rezende, Marcelo Melo, Marcos de Castro, Pedro de Castro e Rafael Abdala, além dos amigos do Aizu.

À família que as consequências da vida me deram, Bruno Cardoso Lopes, meu irmão mais velho; Thiago "Bolão" Abdnur, minha terceira mãe; Bruno Dilly e Rafael Antognolli.

Aos amigos de Laboratório, pelas ótimas amizades e pela convivência. A Alexandro Baldassin e a Felipe Vieira Klein, meus mentores espirituais na jornada do mestrado; Daniel Nicácio, pelos 3 pontos; George Barbosa, pelos jantares e por toda ajuda burocrática; Leonardo Ecco e Raoni Fassina, pela parceria no turno da madrugada; Leonardo Piga, por todos os churrascos e jogos do São Paulo; Maxiwell Garcia, pelas aulas de música; Rafael "Charles de Menezes" Auler, pelo exemplo de superação; Rodrigo Faveri, pelo rock'n'roll;

Ao Professor Doutor Wellington Silva, pela inspiração; aos Professores Doutores Paulo Centoducatte, Rodolfo Azevedo, Guido Araújo que, de formas diferentes, contribuíram com esta pesquisa.

Por fim, agradeço ao universo e suas justificativas.

Sumário

Resumo **vii**

Abstract **ix**

Agradecimentos **xi**

1 Introdução **1**
1.1 Memórias Transacionais 2
1.2 Plataformas de simulação 3
1.3 Sobre este trabalho 4

2 Trabalhos Relacionados **7**
2.1 Linguagens para descrição de arquiteturas 7
2.1.1 *SystemC* e TLM 7
2.1.2 ArchC 8
2.2 Plataformas de simulação 10
2.2.1 MPARM 10
2.2.2 ARP 10
2.2.3 Simics e SimWattch 12
2.3 Memórias transacionais 13
2.4 Estimativa de consumo em TMs 14

3 Consumo de energia em STMs **17**
3.1 Comparação de consumo entre STMs e *Locks* 17
3.1.1 Metodologia 17
3.1.2 Ambiente de simulação 18
3.1.3 TL2 19
3.1.4 *Locks* 21
3.1.5 STAMP 22
3.1.6 Resultados 23

3.2 Análise de custo dos *locks* . . . 27
3.3 Caracterização do consumo de STMs . . . 28
3.4 Considerações finais . . . 31

4 Plataforma híbrida para estimativa de consumo de energia 33
4.1 Implementação da plataforma híbrida . . . 33
4.1.1 Modificações aplicadas ao MPARM . . . 34
4.1.2 Integração do modelo ArchC . . . 36
4.1.3 Modificações aplicadas ao modelo ArchC ARMv5 . . . 38
4.1.4 Verificação da plataforma . . . 40
4.2 Avaliação da plataforma . . . 41
4.2.1 Ganho de desempenho . . . 42
4.2.2 Estimativas de energia e consumo . . . 44
4.3 Conclusões e considerações finais . . . 49

5 Conclusão 51
5.1 Trabalhos Futuros . . . 52

Bibliografia 54

Lista de Tabelas

3.1 Características das aplicações do STAMP - Fonte [12] 23
3.2 STMs x *Locks* - Valores sumarizados 26
3.3 STMs x *Locks* - Valores sumarizados (sem *intruder* e *intruder+*) 26
3.4 Taxas de transações abortadas 32

4.1 Diferenças no número de operações de memória 41
4.2 Número de K ciclos simulados por segundo 44
4.3 Coeficientes da Regressão Linear 46
4.4 Estimativas de consumo 48

Lista de Figuras

2.1 Plataforma MPARM - Fonte [9] 11
2.2 Plataforma ARP - Fonte: [4] 12

3.1 Estruturas de dados - TL2 20
3.2 Energia e ganho de desempenho - STAMP versão STM TL2 *lazy* 23
3.3 Energia e ganho de desempenho - STAMP versão STM TL2 *eager* 24
3.4 Energia e ganho de desempenho - STAMP versão com *locks* 24
3.5 *Overhead* de energia e fator de redução de desempenho da versão STM *lazy* em comparação com *locks* 25
3.6 *Overhead* de energia e fator de redução de desempenho da versão STM *eager* em comparação com *locks* 26
3.7 *Overhead* de energia e fator de redução de desempenho em relação à penalidade aplicada aos *locks* - Aplicação *genome* 28
3.8 Análise de consumo do *benchmark* STAMP com a STM TL2 em 1 núcleo . 29
3.9 Análise de consumo do *benchmark* STAMP com a STM TL2 em 2 núcleos 29
3.10 Análise de consumo do *benchmark* STAMP com a STM TL2 em 4 núcleos 30
3.11 Análise de consumo do *benchmark* STAMP com a STM TL2 em 8 núcleos 30

4.1 Plataforma MPARM com Processadores ArchC 36
4.2 Fluxo TLM 38
4.3 Fluxo da cache de decodificação 39
4.4 Desempenho / Ciclos simulados 43
4.5 Erros na medição de energia total (conjunto das 65 aplicações utilizadas na avaliação da plataforma) 46
4.6 Erros na medição de potência total (conjunto das 65 aplicações utilizadas na avaliação da plataforma) 47
4.7 Erros obtidos com conjunto de testes 49

Capítulo 1

Introdução

Durante vários anos a indústria utilizou o aumento na frequência de operação como principal recurso para aumentar o desempenho dos processadores produzidos. Recentemente, este modelo de evolução esbarrou em limitações físicas que impossibilitaram sua continuidade, pois a incapacidade de dissipar o calor gerado obrigou a indústria a buscar novas alternativas para o projeto de chips mais eficientes. A solução encontrada consistiu em multiplicar a quantidade de núcleos de processamento existentes em um único chip, o que originou novas arquiteturas multiprocessadas, também chamadas *multi-cores*.

Sabe-se que o uso das novas arquiteturas multiprocessadas introduziu novas preocupações no desenvolvimento de software, que até então seguiu o modelo de programação sequencial. O desenvolvimento de software paralelo exige a compreensão de novos paradigmas de programação [41] pois, dada a ineficiência dos recursos de paralelização automática, o código precisa ser explicitamente paralelizado pelo programador para que utilize adequadamente os núcleos adicionais. Este processo exige a implementação de um mecanismo de comunicação entre os múltiplos fluxos de execução, uma vez que estes executarão de forma colaborativa.

O uso de memória compartilhada é o modelo de comunicação mais utilizado na implementação de software paralelo. Neste modelo, os diferentes fluxos de execução possuem endereços de memória que são mapeados no mesmo espaço físico, o que torna os valores neles armazenados visíveis em todos os fluxos. Como o acesso à memória compartilhada é realizado de forma semelhante à memória privada, este modelo possibilita que um valor seja modificado por um dos fluxos enquanto é utilizado por outro, o que ocasiona estados inconsistentes de memória. Por este motivo, torna-se necessário aplicar mecanismos que forcem a sincronização dos acessos à memória compartilhada entre os diferentes fluxos, o que evita que uma variável seja modificada enquanto em uso por algum fluxo.

Eventos como preempção e interrupções tornam o comportamento de um computador imprevisível. Os sistemas computacionais são naturalmente assíncronos, e este indeter-

minismo dificulta muito a programação de um software paralelo [24]. O mau uso dos mecanismos de sincronização frequentemente introduz erros de programação ou diminui a eficiência do programa em questão. Dado o indeterminismo do sistema, utilizar tais mecanismos adequadamente não é uma tarefa simples, e se torna ainda mais complexa devido à dificuldade de se depurar erros no código.

Neste cenário, tornam-se evidentes a necessidade de abstrações mais simples para os mecanismos de sincronização e a importância das ferramentas para teste e análise de eficiência do software desenvolvido.

1.1 Memórias Transacionais

A existência de diferentes fluxos de código que executam simultaneamente exige uma cautela especial na sincronização dos acessos aos recursos compartilhados. Um trecho de código que realiza operações no espaço compartilhado de memória é chamado de seção crítica. Realizar a sincronização dos acessos significa garantir a exclusão mútua, isto é, garantir que duas ou mais seções críticas não executem simultaneamente [24].

Os mecanismos mais utilizados para garantir a exclusão mútua são os *locks* e os semáforos. No entanto, estes mecanismos não fornecem uma abstração adequada ao uso [30] pois exigem que os programadores conheçam detalhes mais profundos a respeito do software desenvolvido e do sistema computacional em que este será executado. Esta característica é a causa dos frequentes problemas encontrados em software paralelo.

Com o objetivo de facilitar o desenvolvimento de software paralelo, as Memórias Transacionais (TM[1]) surgiram como uma alternativa aos *locks* e semáforos cuja abstração é facil de se compreender. Este modelo de programação introduz os conceitos de execução especulativa e detecção de conflitos; uma seção crítica, que no contexto de TMs é chamada transação, sempre é executada especulativamente. Por conflito, entende-se que uma transação modificou um valor em memória compartilhada utilizado por outra transação simultaneamente. Quando a transação chega ao fim, se não houver nenhum conflito com as demais transações em execução, as modificações são aplicadas ao sistema. Caso algum conflito seja detectado, as modificações decorrentes da transação são descartadas e a transação é re-executada. Por deixar a detecção de conflitos a cargo do sistema, as TMs fornecem uma abstração muito mais simples, que exige apenas a delimitação das transações por parte dos programadores.

Diferentes propostas e implementações de TMs tem sido desenvolvidas e testadas. Além das STMs, que consistem em bibliotecas de software para suportar transações, existem implementações que utilizam recursos em hardware para diminuir a sobrecarga

[1]Do inglês: Transactional Memories

e atingir um melhor desempenho. TMs cuja implementação é totalmente feita em hardware compõem as chamadas *Hardware Transactional Memories* (HTM). Por fim, algumas TMs combinam recursos de software e hardware em sistemas híbridos chamados *Hybrid Transactional Memories* (HyTM). No que diz respeito a TMs, o escopo deste trabalho restringe-se as STMs.

Grande parte da literatura tem medido a eficiência das TMs apenas com base em ganhos de velocidade decorrentes do seu uso. As principais métricas apresentadas são o *throughput*, que mede a eficiência de uma TM com relação ao número de transações bem sucedidas executadas em um intervalo definido de tempo, e o *speedup*, que mede o quão mais veloz um programa executou em proporção ao seu tempo de execução anterior. Conforme demonstrado no Capítulo 2, poucas pesquisas foram realizadas a respeito do consumo de energia gerado pelas TMs.

1.2 Plataformas de simulação

Enquanto as arquiteturas de hardware tornam-se cada vez mais complexas, a necessidade de novas ferramentas que auxiliem no seu projeto se torna evidente. O uso de plataformas virtuais que permitam a exploração no espaço de projeto mostrou-se um procedimento interessante para acelerar o desenvolvimento de novos componentes de hardware, permitindo exploração e verificação antecipada da arquitetura em questão.

O baixo consumo de energia se tornou uma característica importante no desenvolvimento de componentes de hardware. Além de estender o tempo de operação das baterias em sistemas embarcados, esta característica reduz o calor produzido pelo hardware em geral, diminuindo a necessidade de dissipadores e, consequentemente, o custo de produção. O desenvolvimento de sistemas de baixo consumo é uma tarefa difícil, no entanto, pode ser assistida pelo uso de plataformas virtuais. Através de simulação é possível observar o comportamento dos componentes de hardware conectados, o que permite estimar sua eficiência e consumo.

Plataformas virtuais podem utilizar diferentes níveis de abstração. Um simulador com precisão de ciclo consiste em uma implementação em baixo nível de um componente de hardware, considerando seus estados em cada ciclo simulado. Se esta abstração é utilizada para descrever um processador, todo o comportamento dos componentes da micro arquitetura, como *pipeline* e coprocessadores, devem ser detalhadamente simulados. Utilizar simuladores com precisão de ciclo garante simulações precisas e fornece resultados bastante confiáveis.

A principal desvantagem de executar simulações com precisão de ciclos é a complexidade envolvida. Para atingir alta precisão, esta abstração simula todos os detalhes do hardware, o que aumenta significativamente a complexidade computacional de execução.

Esta característica impõe limitações de desempenho que se tornam críticas durante a execução de grandes quantidades de computação. Utilizar esta abstração na execução de um grande conjunto de simulações certamente se torna problemático e, possivelmente, inviável.

Uma forma de aumentar a velocidade de uma simulação é utilizar níveis de abstração mais altos. Simuladores funcionais são muito mais simples do que os modelos com precisão de ciclo. Estes simuladores não fornecem o número de ciclos simulados com precisão, o que permite desconsiderar vários detalhes do componente durante a simulação. Por serem mais simples, estes simuladores são mais velozes e reduzem o tempo total de simulação. Apesar de aumentar o desempenho das plataformas virtuais, utilizar esta abstração reduz a precisão dos resultados obtidos.

1.3 Sobre este trabalho

Durante o desenvolvimento deste trabalho, contribuições foram realizadas em duas diferentes frentes. Em um primeiro momento, o consumo de energia decorrente do uso de STM foi avaliado. Este estudo foi realizado através da comparação do consumo gerado pela execução simulada do *benchmark* STAMP compilado com a STM TL2 [15] e com um mecanismo de sincronização baseado em *locks*. Durante a execução dos testes de medição de consumo, observou-se limitada eficiência da plataforma de simulação utilizada em decorrência do nível de abstração simulado. Com base nestas limitações, o trabalho seguiu com a implementação de uma plataforma de simulação com abstração híbrida. Nesta implementação, os modelos de processador foram substituídos por modelos com abstração funcional mais simplificada, o que diminuiu a complexidade computacional da plataforma e aumentou o seu desempenho.

No que diz respeito ao consumo de energia em STMs, as contribuições deste trabalho são: (i) a caracterização das principais causas do aumento de consumo de energia gerado pela STM; (ii) uma comparação do consumo de energia observado entre a STM adotada e uma versão do *benchmark* baseada em *locks*; (iii) uma avaliação profunda do mecanismo de *locks* demonstrando que, apesar de normalmente apresentarem maior eficiência que um sistema STM, estes mecanismos são afetados pelos altos custos associados a implementação do sistema operacional. Todos estes aspectos contribuem para uma melhor compreensão do consumo de energia em STMs.

Os estudos envolvendo consumo de energia em STMs [28] foram publicados no *International Symposium of Low Power Eletronics Design* (ISLPED).

Em relação a implementação de plataformas virtuais, este trabalho apresenta as seguintes contribuições: (i) a introdução de um novo recurso de simulação na plataforma MPARM; (ii) uma detalhada descrição da implementação de uma plataforma de simulação

com abstração híbrida, demonstrando como vários problemas foram corrigidos; (iii) uma proposta de correção estatística da imprecisão introduzida na plataforma pela elevação do nível de abstração.

No dado momento, um artigo a respeito dos estudos com plataformas virtuais encontra-se em elaboração e em breve será submetido a uma conferência da área.

Esta dissertação está organizada da seguinte forma: O Capítulo 1 consiste nesta introdução. O Capítulo 2 apresenta trabalhos relacionados aos diferentes tópicos abordados por esta pesquisa. O Capítulo 3 relata os estudos a respeito da sobrecarga de energia decorrente do uso de STMs. O Capítulo 4 apresenta a plataforma de simulação híbrida desenvolvida. O Capítulo 5 demonstra as conclusões obtidas ao fim deste trabalho.

Capítulo 2

Trabalhos Relacionados

Neste capítulo serão apresentados trabalhos relacionados ao desenvolvimento de plataformas de simulação voltadas para estimativa de consumo de energia. Durante a descrição dos trabalhos mencionados, vários conceitos básicos importantes para a compreensão desta dissertação serão apresentados. Grande parte destas pesquisas possui influência direta no resultado final deste projeto, tendo sido utilizadas durante as fases de implementação ou teste da plataforma de simulação a ser introduzida no Capítulo 4.

Também serão introduzidos trabalhos relacionados aos sistemas de memória transacional e ao consumo de energia decorrente do uso destes sistemas. Estes trabalhos são importantes uma vez que parte desta pesquisa consistiu na comparação do consumo de energia em aplicações que utilizam memória transacional e em aplicações que utilizam *locks*. Os estudos referentes a este tema serão apresentados no Capítulo 3.

2.1 Linguagens para descrição de arquiteturas

2.1.1 *SystemC* e TLM

SystemC [10] é uma biblioteca C++ contendo classes e macros que fornecem recursos de programação necessários ao processo de prototipagem de sistemas de hardware. Além de incluir tipos de dados capazes de representar todos os níveis lógicos, esta biblioteca também funciona como um ambiente de simulação baseado em eventos, permitindo a simulação paralela de diferentes componentes de hardware. O uso de *SystemC* na prototipagem de alto-nível de sistemas permite a exploração antecipada do espaço de projeto de hardware, o que permite testes em estágios iniciais do desenvolvimento e diminui o tempo total de implementação. A biblioteca *SystemC* está disponível para download no site da *Open SystemC Initiative* (OSCI) [25].

Transactional Level Modeling (TLM) [21] é uma metodologia de alto nível para o

projeto de sistemas complexos, que necessitam de uma abstração mais elevada e que possam tirar vantagem da exploração e teste antecipados do projeto. A metodologia TLM permite uma prototipagem de hardware mais simples, que independe de detalhes lógicos dos componentes. Além disso, seu uso em modelos de simulação proporciona altíssimos ganhos de desempenho quando comparado ao uso de uma abstração mais detalhada, como *Register Transfer Level* (RTL). Desde sua versão 2.0, a biblioteca *SystemC* suporta o uso de TLM como metodologia de projeto [38].

Os componentes de hardware integrantes de uma plataforma desenvolvida de acordo com a metodologia TLM são implementados na forma de módulos. A comunicação realizada entre estes módulos é implementada através de transações, que são executadas na forma de chamadas a funções e métodos que realizam trocas de pacotes de dados. Estas funções e métodos, por sua vez, implementam um protocolo de comunicação que funciona como uma interface, tornando os pacotes trocados compatíveis com os formatos esperados pelos módulos que se comunicam. Os pacotes trocados normalmente são compostos por uma operação a ser executada (leitura ou escrita), o endereço onde a operação será executada, os dados, uma máscara de bits indicando quais porções dos dados são válidas e uma resposta que indica o sucesso ou falha da transação.

2.1.2 ArchC

ArchC [5, 37] é uma linguagem para descrição de arquiteturas (ADL[1]) baseada na biblioteca *SystemC*, capaz de gerar simuladores de conjunto de instruções (ISSs[2]) rápidos e funcionais. Esta linguagem, que foi desenvolvida na Unicamp[3], possui seu código fonte aberto, disponível sob a licença GPL [20]. Um grande conjunto de modelos de ISSs desenvolvidos com ArchC está disponível para download e uso no site da linguagem [42].

Modelos desenvolvidos com ArchC são descritos em duas partes. A primeira, chamada *AC_ARCH*, descreve a arquitetura do modelo através de declarações dos seus componentes como memória, banco de registradores e *pipeline*. A segunda, chamada *AC_ISA*, descreve o conjunto de instruções do modelo, apresentando o formato e codificação de cada uma. As Listagens 2.1 e 2.2 demonstram estas duas partes da descrição da arquitetura ARMv5. Na Listagem 2.1 estão declarados uma memória interna de 256 megabytes, um banco de registradores com 31 unidades e o tamanho da palavra de dados, contendo 32 bits. Na Listagem 2.2 um trecho da descrição do conjunto de instruções é apresentado, demonstrando a declaração da instrução de *branch* "b". Como pode ser observada, a gramática utilizada na escrita destes dois arquivos é própria da linguagem.

[1]Do inglês: Architecture Description Language

[2]Do inglês: Instruction Set Simulators

[3]Laboratório de Sistemas de Computação, Instituto de Computação, Universidade Estadual de Campinas

Listing 2.1: ARMv5 AC_ARCH

```
AC_ARCH(armv5e){
  ac_mem        MEM:256M;
  ac_regbank  RB:31;
  ac_wordsize 32;
  ARCH_CTOR(armv5e) {
    ac_isa("armv5e_isa.ac");
    set_endian("little");
  };
};
```

Listing 2.2: ARMv5 AC_ISA

```
AC_ISA(armv5e) {
  ac_format Type_BBL = "%cond:4 %op:3 %h:1 %offset:24";
  (...)
  ac_instr<Type_BBL> b;
  (...)
  ISA_CTOR(armv5e) {
    b.set_asm("b%[cond] %exp(bimm)", cond, offset, h=0);
    b.set_asm("bl%[cond] %exp(bimm)", cond, offset, h=1);
    b.set_decoder(op=0x05);
  }
}
```

A linguagem ArchC suporta nativamente a declaração de interfaces de comunicação TLM em seus modelos, o que possibilita a comunicação com módulos externos e facilita sua integração em plataformas de simulação. Este suporte é implementado através do uso do modelo de interface *transport* fornecido pelo *SystemC*. Modificar um modelo ArchC existente para utilizar a interface TLM exige apenas uma pequena modificação no arquivo *AC_ARCH* e a implementação das funções que são responsáveis pelo transporte do pacote de dados. Este processo, que não exige nenhuma outra modificação adicional, será descrito no Capítulo 4.

O modelo ArchC ARMv5 é um simulador funcional. Nesta implementação, a estrutura do pipeline presente no processador é ignorada, o que simplifica bastante a execução do simulador. Esta característica implica em altos ganhos de desempenho quando o modelo é comparado a outros simuladores com precisão de ciclo. Mais detalhes a respeito deste modelo específico também serão apresentados no Capítulo 4.

Em [33], um modelo de estimativa de consumo de energia em nível de instrução utilizando ArchC é proposto. Neste trabalho, um processador SPARCv8 gerado com ArchC é modificado para gerar estatísticas a respeito das instruções executadas. Estas estatísticas, em conjunto com informações de custo de cada instrução, são utilizadas como entrada pela ferramenta *acpower*, que então gera um arquivo contendo as estimativas de consumo para

o programa simulado. Este mecanismo utiliza médias de consumo para cada instrução, compondo uma abordagem de estimativa de mais alto nível em relação as abordagens apresentadas em [9, 13].

2.2 Plataformas de simulação

2.2.1 MPARM

O MPARM [9] é uma plataforma de sistema completa[4] desenvolvida para realizar a simulação de *Multi-processor Systems-on-Chip* (MPSoCs). Esta plataforma é escrita na linguagem C++ e utiliza a biblioteca *SystemC* [10] como mecanismo de simulação. O MPARM tem sido bastante utilizado para análise de consumo de energia em MPSoCs [31] e sistemas de memória transacional em hardware e software [8, 18, 19, 27].

O ISS originalmente distribuído em conjunto com a plataforma MPARM é o SWARM [14], que consiste em um modelo com precisão de ciclos, implementado em C++, do processador ARM7. O MPARM ainda inclui implementações do modelo de barramento AMBA, de um sistema hierárquico de memórias e um sistema de sincronização para múltiplos processadores. Modelos de consumo de energia para todos os módulos que acompanham a plataforma também estão incluídos. Por possuir precisão de ciclo, as simulações realizadas neste ambiente são bastante precisas, porém mais lentas que simulações executadas em modelos de mais alta abstração. A plataforma MPARM pode ser visualizada com detalhes na Figura 2.1.

Por ser construída em cima da biblioteca *SystemC*, esta plataforma possui relativa compatibilidade com os processadores descritos com a linguagem ArchC. Uma vez que ambos utilizam o mesmo mecanismo de sincronização fornecido pelo *SystemC*, a integração dos processadores na plataforma exige apenas a implementação de estruturas para comunicação.

2.2.2 ARP

Processadores modelados com a linguagem ArchC são utilizados em plataformas de simulação desde seu lançamento. Este uso foi recentemente sistematizado através da ferramenta ARP[5] [1, 4], que também utiliza a biblioteca *SystemC* na sincronização de seus processadores, barramento e memórias. Esta plataforma consiste basicamente em uma estrutura de diretórios, onde estão armazenados os componentes do sistema, e um conjunto de *scripts* que auxiliam na construção e execução das plataformas.

[4] Do inglês: *Full System Platform*
[5] ArchC Reference Platform

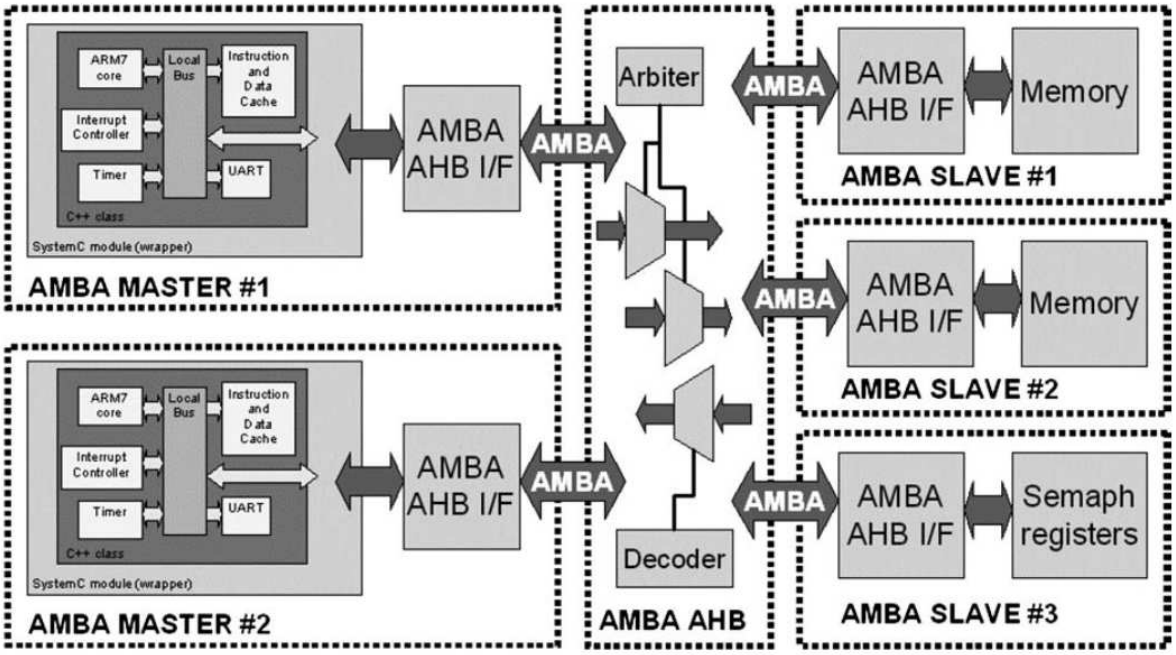


Figura 2.1: Plataforma MPARM - Fonte [9]

As plataformas construídas com o ARP são descritas em um arquivo chamado `defs.arp`, que contém informações a respeito de quais componentes utilizar. Dentre estes componentes estão incluídos processadores, *wrapper* e barramento. O software a ser executado quando a plataforma for inicializada também é especificado neste arquivo. Conforme pode ser observado na Figura 2.2, a comunicação entre o processador e o barramento é intermediada por um *wrapper*. Este *wrapper* funciona realizando a tradução de pacotes TLM entre os dois componentes, isto é, sempre que o processador precisa realizar uma leitura na memória, ele invoca a função *transport* que realiza a tradução dos dados para um formato que pode ser compreendido pelo barramento. Por ser um canal obrigatório para o acesso à memória, o *wrapper* também é um mecanismo bastante eficiente para realizar mapeamentos de endereços de memória, distinguindo entre regiões compartilhadas e privadas de acordo com o processador.

O conceito do *wrapper* presente na plataforma ARP mostrou-se muito importante para a conclusão deste trabalho. Uma metodologia muito semelhante foi aplicada na integração dos processadores ArchC com a plataforma MPARM, etapa a ser descrita no Capítulo 4.

Em [29], a plataforma ARP foi utilizada para na implementação do sistema de memória transacional em hardware proposto em [23]. Nesta implementação, os processadores PowerPC e MIPS utilizados foram modificados para suportar novas instruções transacionais. Memórias cache com um mecanismo de controle de versão e detecção de conflitos

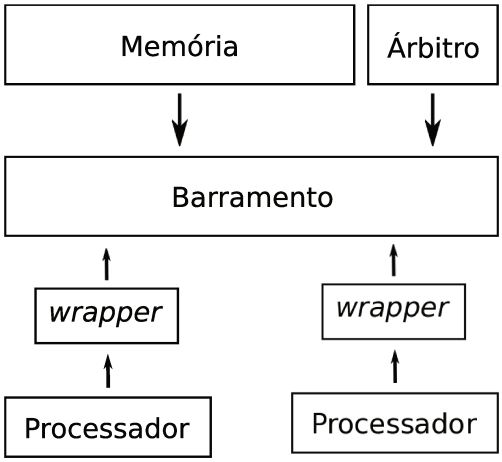


Figura 2.2: Plataforma ARP - Fonte: [4]

também foram utilizadas. Conforme descrito, o trabalho de modificação dos processadores foi bastante facilitado pelo uso de uma ADL de alto nível que, no caso, é a linguagem ArchC. Alguns conceitos a respeito de memórias transacionais serão descritos adiante, na Seção 2.3.

2.2.3 Simics e SimWattch

O Simics [34] é uma plataforma de sistema completa e comercial cujo objetivo é simular cargas realistas de trabalho. Esta plataforma possui suporte a várias famílias de processadores e é capaz de emular diferentes sistemas operacionais (SOs).

SimWattch é uma plataforma de simulação baseada no simulador de sistemas Simics e no Wattch [11], uma extensão de modelagem de consumo presente na ferramenta SimpleScalar [3]. Esta ferramenta utiliza o Simics para realizar a simulação do sistema, gerando *traces* das instruções executadas, que são traduzidas para o conjunto nativo do SimpleScalar. Uma vez traduzidas, estas instruções são utilizadas pelo Wattch na caracterização do consumo. Devido ao suporte fornecido pelo Simics, esta plataforma é capaz de levar em consideração os efeitos do SO no consumo de energia, relevando, por exemplo, mudanças no estado da microarquitetura decorrentes de mudanças de contexto entre chamadas de sistema e a execução da aplicação. Apesar de o simulador Simics ser bastante completo, o suporte à estimativa de consumo do SimWattch limita-se à microarquitetura. A plataforma não possui modelos de consumo para componentes periféricos.

2.3 Memórias transacionais

Com o aumento do uso de processadores *multicore*, tem se evidenciado cada vez mais as dificuldades na programação de um software paralelo. Os modelos tradicionais de desenvolvimento deste tipo de software deixam a cargo do programador todo o trabalho de sincronização dos fluxos de execução do código (*threads*). Mecanismos contra intuitivos, como *locks* e semáforos, frequentemente são utilizados de forma incorreta, resultado em uma sincronização ineficiente que pode diminuir o desempenho do software e inserir erros difíceis de depurar.

A busca por uma abstração mais simples para a programação paralela elevou o interesse pelos sistemas de memória transacional (TM[6]). A idéia de utilizar abstrações semelhantes às transações de banco de dados para garantir consistência entre dados compartilhados por vários processos surgiu em 1977, em [32].

Ravi Rajwar e James R. Goodwin propuseram em [36] um sistema onde os *locks* de um software seriam reconhecidos e removidos pelo hardware, eliminando possíveis serializações. Este sistema executaria as transações de forma especulativa, isto é, todas as modificações geradas pela transação seriam escritas em um *buffer* local até que esta chegue ao fim. Quando a transação encerra, caso nenhum conflito seja detectado, as operações realizadas no *buffer* são aplicadas nos verdadeiros endereços de memória do sistema, tornando as modificações visíveis para os demais processos que compartilham o hardware. Executar as transações de maneira especulativa é uma estratégia utilizada em vários sistemas de memória transacional até hoje, como pode ser observado em [15,17,40].

A primeira proposta de TM suportada por hardware surgiu em [23]. Nesta implementação, a fim de fornecer as operação básicas necessárias para se realizar uma transação, 6 novas instruções foram adicionadas ao conjunto original do processador. Uma cache transacional também foi incluída para cada processador com o objetivo de funcionar como *buffer* para os valores especulativos. Neste sistema, a detecção de conflitos é realizada através de uma versão estendida do protocolo de coerência de cache proposto em [22].

Por conflito, entende-se que duas transações diferentes realizaram operações simultâneas no mesmo endereço de memória e, pelo menos uma dessas operações, foi uma operação de escrita. Além disso, diz-se que a contenção do sistema está muito alta quando uma transação é executada várias vezes até que consiga ser concluída sem nenhum conflito e, então, aplicada ao sistema.

Em [15], uma implementação de TM em software (STM) chamada TL2 é descrita. Este sistema baseia-se na idéia de manter um relógio global de versões das variáveis compartilhadas. Sempre que uma leitura ou escrita é realizada, um algoritmo de verificação de versão é seguido, garantindo a consistência dos dados acessados. O relógio global é

[6]Do inglês: Transactional Memories

incrementado ao final de cada transação, indicando às demais transações em execução que a versão dos dados lidos por elas está defasada. As transações que só executam leituras são bastante facilitadas neste sistema, uma vez que só é necessário verificar se o relógio global não foi acessado entre o início e o fim da transação.

2.4 Estimativa de consumo em TMs

Em [9] a plataforma MPARM foi utilizada como simulador para análise de consumo de energia em MPSoCs. Neste trabalho, modelos de consumo de energia foram incorporados a cada um dos componentes de hardware, possibilitando que a plataforma observe os valores de consumo de energia em cada um dos ciclos simulados.

Um estudo a respeito dos efeitos no consumo de energia gerados pelo uso de uma TM em hardware é apresentado em [35]. Os experimentos realizados neste trabalho utilizaram o *benchmark* SPLASH-2 [45] e a plataforma de simulação Simics. A TM utilizada é baseada no sistema introduzido em [23] porém inclui um mecanismo semelhante ao apresentado em [46], que serializa a execução das transações quando momentos de alta contenção são detectados. Os resultados apresentados indicam que, uma vez que não é mais necessário realizar aquisições de *locks*, o uso de TMs reduz significativamente a quantidade de acessos à memória compartilhada e, em sistemas de baixa contenção, implica em um menor consumo de energia.

A plataforma MPARM foi utilizada para investigar o consumo de energia de uma HTM em [18]. A implementação, que é semelhante a implementação apresentada em [23], apresentou bons resultados, reduzindo em até 71% o produto da energia pelo atraso[7] (EDP) de aplicações simuladas. Os resultados apresentados, no entanto, limitaram-se a um pequeno *microbenchmark* desenvolvido para testar a plataforma, sendo pouco abrangentes para serem considerados conclusivos.

Em [19], o MPARM foi utilizado para avaliar o desempenho de uma HTM voltada para sistemas embarcados. A HTM implementada utiliza uma memória *scratchpad* para salvar o estado dos registradores da CPU antes da transação iniciar, utilizando-os para restaurar o estado inicial caso a transação seja abortada. Caso uma transação exceda a capacidade da memória cache transacional, as demais CPUs são paralisadas e a transação termina sua execução utilizando o resto da hierarquia de memória. Por fim, com o objetivo de economizar energia, a HTM é capaz de desligar a cache transacional quando esta não está sendo utilizada. Os resultados demonstraram ganhos em consumo de energia de até 17% sobre a implementação original da HTM.

Em [6] é apresentada uma metodologia para medição de consumo de STMs que isola

[7] Do inglês: *Energy Delay Product*

os valores mensurados no escopo da API da STM dos valores medidos no restante da aplicação. Através desta metodologia é possível obter o consumo introduzido pela STM de forma classificada, o que possibilita o estudo mais apurado a respeito dos efeitos gerados pelas STMs no consumo de energia e das formas de minimizar este consumo.

O consumo de energia da STM TL2 em conjunto com o *benchmark* STAMP foi caracterizado em [27]. Observou-se neste trabalho que as STMs normalmente introduzem um grande impacto negativo no consumo de energia de uma aplicação, porém, assim como em [35], podem ser interessantes em casos de baixa contenção. Neste trabalho também são apresentadas versões da STM TL2 em que objetos da memória principal são movidos para memórias *scratchpad* presentes na plataforma, demonstrando bons ganhos de desempenho. Por fim, demonstra-se que, no que diz respeito ao consumo de energia, STMs com algoritmos de detecção de conflito do tipo *lazy* e do tipo *eager* possuem um comportamento semelhante.

Em [8] é apresentada uma técnica para reduzir o consumo de energia em STMs baseada em *dynamic voltage and frequency scaling* (DVFS) [26]. Antes de ser re-executada, uma transação abortada é mantida em *backoff mode*, isto é, não é instantaneamente reiniciada, com o objetivo de evitar conflitos consecutivos entre as mesmas transações. Na técnica apresentada neste trabalho, a frequência e a voltagem do processador são reduzidos enquanto uma transação está em *backoff mode*, diminuindo o consumo gerado pelas STMs. Esta técnica apresentou bons ganhos em sistemas com alta contenção, reduzindo o EDP em 45% em média.

Capítulo 3

Consumo de energia em STMs

Neste capítulo serão introduzidos estudos realizados a respeito do consumo de energia em STMs. Inicialmente serão apresentados conceitos para compreensão dos estudos realizados, incluindo a metodologia, ambiente de simulação, mecanismos de sincronização e *benchmark* utilizados, seguidos pelos resultados obtidos. Em seguida serão apresentados breves estudos a respeito do custo de se utilizar *locks* e a caracterização do consumo de energia em STMs. Por fim, serão apresentadas considerações finais a respeito da co-autoria e publicação deste trabalho.

3.1 Comparação de consumo entre STMs e *Locks*

Com o objetivo de compreender os efeitos das STMs em relação ao consumo de energia, foram realizados experimentos envolvendo a simulação de um *benchmark* em duas versões, uma utilizando STM e outra utilizando *locks*. Os resultados obtidos foram comparados com o objetivo de possibilitar a compreensão da eficiência destes mecanismos no que diz respeito ao consumo de energia.

3.1.1 Metodologia

Com o objetivo de caracterizar o consumo de energia, utilizou-se um procedimento semelhante ao apresentado em [8]. Este procedimento permite isolar o consumo de energia gerado pelo mecanismo de sincronização do consumo de energia gerado pela aplicação, o que possibilita identificar problemas e pontos críticos nestas implementações e favorece o desenvolvimento de sistemas mais eficientes.

Como o uso de STMs mostrou-se bastante promissor para o desenvolvimento de software paralelo, diversas implementações foram desenvolvidas e estão disponíveis para uso [30]. Apesar das diferenças, a grande maioria destas STMs baseia-se nas mesmas pri-

mitivas de sincronização, *TxStart*, *TxCommit*, *TxLoad* e *TxStore*, o que as torna compatíveis com este procedimento de medição. Entre os valores energéticos medidos classificou-se os custos gerados por cada uma das primitivas, além do custo relacionado à re-execução de transações abortadas e ao gerenciador de contenção.

O procedimento de caracterização do consumo consiste basicamente em distinguir os trechos de código em execução e acumular os valores de consumo medidos em instâncias diferentes. Desta forma, todo o consumo de energia medido no código da aplicação é acumulado como consumo da aplicação. O consumo medido dentro da API da STM é acumulado como consumo de energia da STM, classificado por primitiva. Por fim, sempre que uma transação é abortada, a energia da aplicação e da STM medidas desde o seu início são armazenadas como energia de *rollback*[1], exceto pela energia referente à execução do gerenciador de contenção, que é armazenada como energia de *backoff*[2]. Diferenciar o consumo gerado pela execução de *rollbacks* possibilita identificar com precisão os *overheads* introduzidos pela STM em aplicações com alta contenção.

Para realizar os experimentos, utilizou-se a plataforma MPARM na simulação das 8 diferentes aplicações do *benchmark* STAMP. Cada uma das aplicações foi executada em sua versão STM e em sua versão com *locks*. Foram simuladas plataformas com 1, 2, 4 e 8 núcleos. Algumas das aplicações foram executadas em mais de uma configuração.

3.1.2 Ambiente de simulação

A plataforma MPARM, utilizada nos testes executados, consiste em uma plataforma de simulação com precisão de ciclos que possui modelos validados de consumo de energia. O ISS utilizado nos testes correspondem a processadores ARMv7 com caches de 4 vias divididas em cache de instrução e dados, com 8KB e 4KB de espaço, respectivamente. A plataforma fornece ainda acesso a 12MB de memória privada para cada processador e 16MB de memória compartilhada. Nenhum mecanismo de coerência de cache foi utilizado, em vez disso, acessos a memória compartilhada não foram armazenados na cache. A plataforma ainda fornece um dispositivo de semáforo em hardware e uma API de suporte à simulação. Os módulos de memória possuem uma arquitetura baseada no modelo SRAM, que possui menor latência e é mais eficiente em relação ao consumo que o modelo DRAM. Todos os módulos estão conectados por um barramento AMBA AHB [2].

Para executar os testes, foram necessárias duas primitivas de sincronização, *lock* e *compare-and-swap* (CAS). Como o processador ARM não possui as instruções *test-and-set* e *compare-and-swap* no seu conjunto de instruções, estas operações foram implementadas em software, utilizando as chamadas *wait* e *signal*, fornecidas pela API de suporte a

[1] Termo inglês utilizado para denominar uma re-execução de uma transação

[2] Tempo que uma transação aguarda, após ser abortada, para ser novamente executada

simulação da plataforma.

Os códigos das chamadas *wait* e *signal*, que podem ser vistos nas Listagens 3.1 e 3.2, são importantes para se compreender as implementações dos mecanismos de sincronização nas próximas seções. Na Listagem 3.1 a variável *lock[ID]* referencia o endereço do semáforo implementado em hardware, que atualiza seu valor automaticamente quando lido pela primeira vez.

Listing 3.1: WAIT(int ID)

```
int WAIT(int ID)
{
  while (lock[ID]) {}
  return(0);
}
```

Listing 3.2: SIGNAL(int ID)

```
void SIGNAL(int ID)
{
  lock[ID] = 0;
}
```

A arquitetura da plataforma MPARM será descrita com mais detalhes no Capítulo 4.

3.1.3 TL2

A STM *Transactional Locking 2* (TL2) [15] consiste em uma TM baseada na implementação do algoritmo *Transactional Locking* [16]. Nesta STM, cada aplicação possui um relógio global que é utilizado durante a execução das transações para verificar a validade dos conjuntos de leitura e escrita. Este relógio global funciona como um mecanismo de versão para as variáveis compartilhadas.

O conjunto de escrita de uma transação pode ser entendido como o conjunto de variáveis que estão sendo modificadas pela transação. O conjunto de leitura, por sua vez, é o conjunto de todas as variáveis cujo valor é lido e utilizado, mas não modificado, pela transação.

Para cada transação em execução, A TL2 mantém uma estrutura de dados chamada "descritor de transação", que armazena informações como o valor do relógio global no momento em que a transação iniciou e os endereços dos elementos dos conjuntos de leitura e escrita. Cada um destes endereços pode ser mapeado para outra estrutura de dados chamada tabela de registros de posse (ORT[3]), que é compartilhada pelas transações. Cada elemento da ORT é composto por dois campos, um bit de *lock* e um valor. Caso o

[3]Do inglês: *Ownership Record Table*

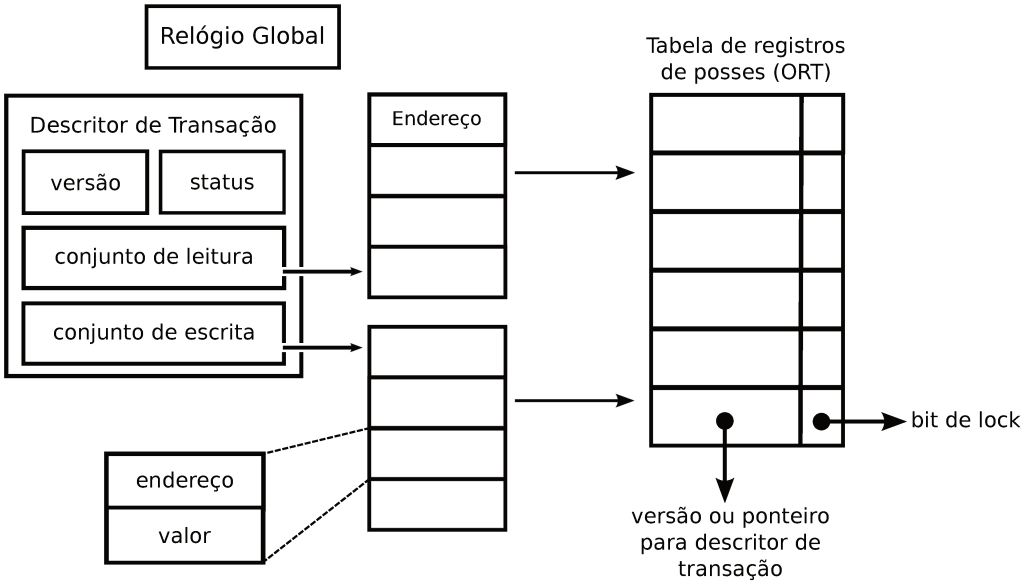


Figura 3.1: Estruturas de dados - TL2

bit de *lock* seja 0, o valor da entrada é a versão atual dos endereços de memória mapeados para aquela entrada. Caso o bit de *lock* seja 1, o valor da entrada é um ponteiro para a transação que possui o *lock* dos endereços de memória desta entrada, detendo acesso exclusivo. A Figura 3.1 apresenta as estruturas de dados e as respectivas referências utilizadas pela TL2.

As transações da TL2 baseiam-se em quatro primitivas principais, *TxStart*, *TxStore*, *TxLoad* e *TxCommit*:

- **TxStart:** Marca o início de uma transação. Durante sua execução o valor do relógio de versão global é armazenado localmente no descritor da transação.

- **TxStore:** É utilizada quando a transação supostamente escreve um valor na memória. Em vez de modificar a memória diretamente, o valor e o endereço a serem modificados são inseridos no conjunto de escrita da transação, para que possam ser aplicados no sistema caso a transação venha ser efetivada.

- **TxLoad:** Utilizada sempre que uma transação realiza a leitura de um valor da memória. Primeiramente busca o valor a ser lido no conjunto de escrita e o retorna, caso encontrado. Se não for encontrado, a entrada referente ao endereço é lida da ORT e o valor é lido da memória. Caso a entrada da ORT esteja bloqueada ou possua uma versão maior que o relógio de versão global lido durante *TxStart*, a transação é abortada. Caso contrário, o valor lido da memória é retornado.

- **TxCommit:** Marca o fim de uma transação. É utilizada para verificar conflitos e efetivar as modificações realizadas pela transação. Inicialmente tenta obter todos os *locks* para os elementos do conjunto de escrita e, caso não seja possível, aborta a transação. Em seguida, incrementa do valor do relógio global de versão e valida todos os elementos do conjunto de leitura. Para realizar a validação, verifica-se se eles estão bloqueados por outra transação e se sua versão não é maior do que o relógio de versão global lido durante *TxStart*. Se o conjunto de leitura for inválido, a transação é abortada. Finalmente a memória é atualizada e os elementos da ORT são desbloqueados e atualizados.

Pode se dizer que o algoritmo TL2, conforme apresentado, é do tipo *lazy* no que diz respeito à atualização dos dados. Isso significa que os valores da memória modificados são armazenados em um *buffer* e só são aplicados no fim da transação. A versão alternativa é chamada *eager*. Nesta versão, as modificações realizadas pela transação são aplicadas diretamente no endereço de memória destino e um *log* destas operações é mantido. Caso a transação seja abortada, este *log* é utilizado para restaurar o estado válido anterior à sua execução.

A STM TL2 utiliza ainda um mecanismo de *backoff* que mantém uma transação paralisada por um tempo após esta ter sido abortada 3 vezes. Se a transação abortar novamente, o tempo de *backoff* é incrementado de forma linear ou exponencial. O objetivo deste mecanismo é evitar que uma transação continue abortando consecutivamente.

Descrições mais profundas a respeito do algoritmo TL2 são apresentadas em [7, 15].

3.1.4 *Locks*

Os *locks* utilizados no experimento consistem em uma implementação típica de um *spin-lock*[4] global, em que uma única variável booleana sincroniza todo o acesso à memória compartilhada. O código original do *benchmark* foi modificado e as chamadas a API da STM responsáveis por inicializar e concluir uma transação (*TxStart* e *TxCommit*) foram substituídas por chamadas para adquirir e liberar o *lock*.

Em um cenário real de computação, em que a execução de um programa seria assistida por um sistema operacional, enquanto a aplicação aguarda a aquisição do *lock*, outro processo provavelmente seria escalonado para execução. Dada esta característica, a implementação utilizada foi modificada para descartar os valores de consumo medidos enquanto o *lock* está em *loop*. Esta implementação pode ser observada na Listagem 3.3, em que as chamadas *stop_metric()* e *start_metric()* são fornecidas pela API para ativar e desativar a medição de consumo de energia.

[4]Tipo de *lock* que coloca a *thread* em *loop* até que esteja disponível

Listing 3.3: WAIT(int ID)

```
int WAIT(int ID)
{
  if (is_measuring()) {
    stop_metric();
    while (lock[ID]) {}
    start_metric();
  } else {
    while (lock[ID]) {}
  }
}
```

3.1.5 STAMP

Em [12] o *benchmark* STAMP (*Stanford Transactional Applications for Multi-Processing*) é descrito. Este *benchmark* consiste em um conjunto de aplicações desenvolvidas para uso específico em pesquisas a respeito de TMs. As aplicações que compõem o *benchmark* são provenientes de diferentes domínios de pesquisa, possuem uma larga possibilidade de configuração e são paralelizáveis em escalas diferentes. Estas aplicações formam uma coleção de problemas complexos próximos de problemas reais encontrados em ambientes de produção.

As aplicações presentes no STAMP são:

- **Bayes**: Algoritmo de aprendizado baseado em redes bayesianas.
- **Genome**: Compara de segmentos de DNA para reconstruir um genoma original.
- **Intruder**: Detecta intrusão em redes com base na comparação de pacotes com conjunto de assinaturas de ataques.
- **Kmeans**: Algoritmo de classificação utilizado para agrupar objetos de um espaço N-dimensional em K grupos.
- **Labyrinth**: Variação do algoritmo de Lee [43], que busca atravessar um labirinto tridimensional.
- **SSCA2**[5]: Constrói uma estrutura de grafos utilizando vetores e vetores auxiliares.
- **Vacation**: Simula um sistema de reserva de viagens em que ocorrem reservas, cancelamentos e atualizações.

[5]Originalmente composto por 4 kernels, apenas 1 é utilizada no STAMP.

Aplicação	Tamanho Tx	Conjunto R/W	Tempo Tx	Contenção
Bayes	Longa	Grande	Alto	Alta
Genome	Média	Médio	Alto	Baixa
Intruder	Curta	Médio	Médio	Alta
Kmeans	Curta	Pequeno	Baixo	Baixa
Labyrinth	Longa	Grande	Alto	Alta
SSCA2	Curta	Pequeno	Baixo	Baixa
Vacation	Média	Médio	Alto	Baixo/Média
Yada	Longa	Grande	Alto	Média

Tabela 3.1: Características das aplicações do STAMP - Fonte [12]

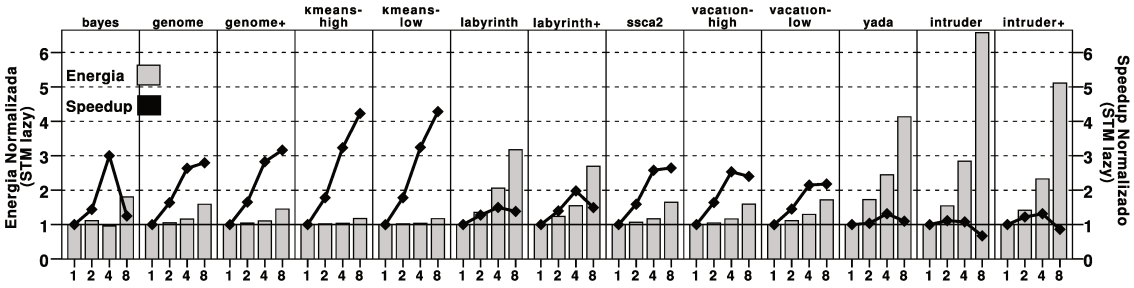


Figura 3.2: Energia e ganho de desempenho - STAMP versão STM TL2 *lazy*

- **Yada**: Algoritmo de Ruppert [39] para realizar o refinamento em uma malha de triângulos representada por um grafo.

As características das aplicações do STAMP, no que influencia a execução de aplicações utilizando TMs, são especificadas na Tabela 3.1. Nesta tabela a palavra transação foi abreviada para Tx, e R/W[6] refere-se a Leitura/Escrita.

3.1.6 Resultados

As Figuras 3.2, 3.3 e 3.4 apresentam o resultados obtidos durante a simulação das aplicações do *benchmark* STAMP utilizando STM e *locks*, respectivamente, como mecanismos de sincronização. Os resultados apresentam os valores referentes ao consumo de energia e ao ganho de desempenho[7], ambos normalizados em relação à simulação com 1 processador. A nomenclatura e configurações das aplicações apresentadas segue a definição original apresentada em [12]. Todos os resultados analisados utilizaram um tempo de *backoff* que, a cada transação abortada, é incrementado de acordo com uma função linear.

[6]Do inglês: Read/Write
[7]*Speedup*

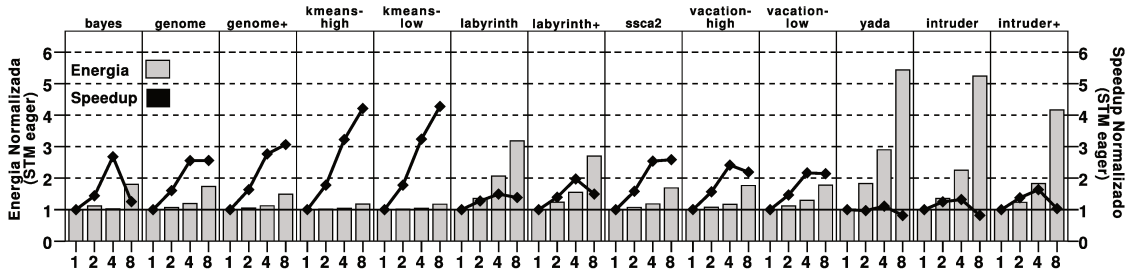


Figura 3.3: Energia e ganho de desempenho - STAMP versão STM TL2 *eager*

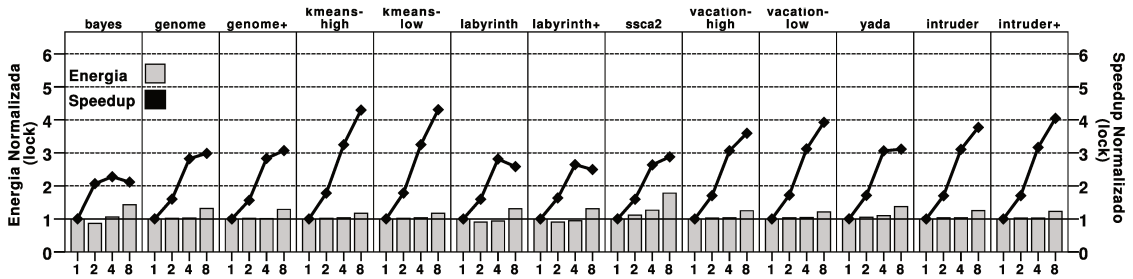


Figura 3.4: Energia e ganho de desempenho - STAMP versão com *locks*

Na Figura 3.2 nota-se que o consumo de energia normalmente aumenta conforme o número de processadores, exceto pela aplicação *bayes* com 4 núcleos, cujo desempenho acaba compensando a inclusão dos novos núcleos. O aumento do desempenho, por sua vez, não acontece seguindo o mesmo padrão. Nas aplicações *genome*, *genome+* e *kmeans* é possível observar que o desempenho aumenta à medida que mais núcleos são utilizados, porém, nas demais aplicações, este comportamento não é observado.

Conforme pode ser observado nas Figuras 3.2 e 3.3, pequenas são as diferenças decorrentes da troca do mecanismo de controle de versão utilizado pela STM. As diferenças mais notáveis estão nas aplicações *intruder* e *intruder+*, que apresentaram maior consumo e pior desempenho na sua versão *lazy*, e *yada*, que apresentou o mesmo comportamento em sua versão *eager*.

Na Figura 3.4 é possível observar um comportamento bastante distinto do apresentado nas Figuras 3.2 e 3.3. No que diz respeito ao consumo de energia, é possível observar que os valores tendem a se manter constantes para execuções com 1 a 4 núcleos, chegando até mesmo a diminuir em alguns casos. O desempenho, por sua vez, apresenta aumentos significativos à medida que o número de processadores é aumentado de 1 a 4. Nestas configurações, o consumo decorrente da inclusão de novos processadores é compensado pelo ganho de desempenho e, por isso, mantém-se a constância nos valores medidos. Quando o número de processadores é aumentado de 4 para 8, as aplicações *bayes*, *labyrinth* e *labyrinth+* apresentaram uma queda no seu desempenho, enquanto as demais demons-

traram ganho, ainda que menos significativo do que os anteriormente observados. Este comportamento acontece em decorrência da escalabilidade das aplicações na plataforma que, independente da quantidade de núcleos instanciados, possui apenas um barramento que fica sobrecarregado na configuração com 8 processadores.

Os resultados acima apresentados levam a entender que a inclusão de novos processadores na execução normalmente mantém ou aumenta o consumo de energia da aplicação. No entanto, é notável a ausência de um padrão que permita relacionar o consumo de energia ao ganho de desempenho, uma vez que em alguns exemplos citados o aumento do consumo não foi acompanhado pela redução do tempo de execução.

Nas Figuras 3.5 e 3.6 são apresentados os *overheads* de energia e o fator de redução de desempenho encontrados devido ao uso da STM em suas configurações *lazy* e *eager*. Estes resultados estão normalizados em relação ao valor observado na respectiva execução utilizando *locks*.

Conforme pode ser observado na Figura 3.5, a maior parte das aplicações é mais eficiente em sua versão com *locks*. As aplicações *kmeans* e *bayes* apresentaram valores de consumo de energia e fator de redução de desempenho semelhantes aos valores observados na sua versão com *locks*. A aplicação *kmeans* possui baixa contenção e utiliza pesadas operações de ponto flutuante emuladas em software, o que amortiza o *overhead* introduzido pela STM em relação ao custo total de execução. O baixo consumo da aplicação *bayes* é explicado pelas suas baixas taxas de transações abortadas e o pequeno *overhead* de primitivas transacionais.

Na Figura 3.5 ainda é possível notar que a aplicação *intruder* apresenta péssima eficiência em sua versão STM. O consumo de energia nesta versão chega superar em 22x o consumo de energia da versão com *locks*. Esta característica pode ser explicada pela altíssima taxa de transações abortadas desta aplicação.

Assim como discutido anteriormente, a Figura 3.6 ilustra um comportamento bastante semelhante ao apresentado na Figura 3.5. As diferenças mais notáveis estão na aplicação

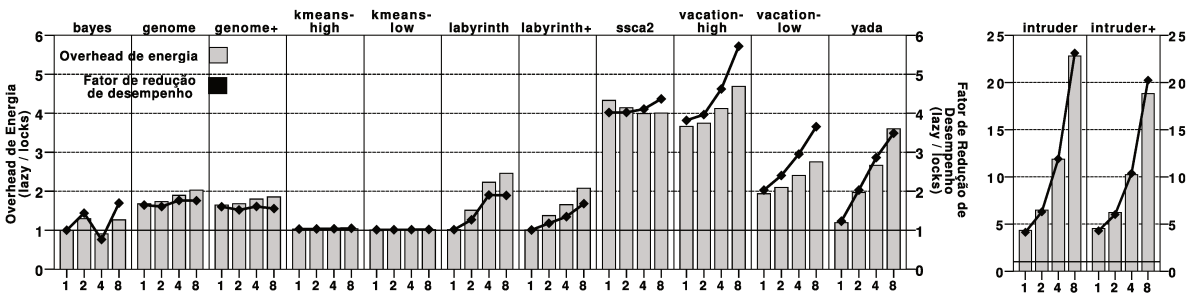


Figura 3.5: *Overhead* de energia e fator de redução de desempenho da versão STM *lazy* em comparação com *locks*

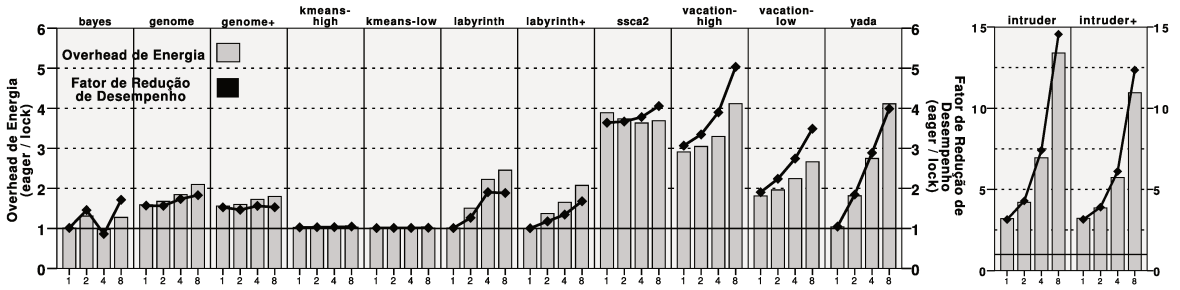


Figura 3.6: *Overhead* de energia e fator de redução de desempenho da versão STM *eager* em comparação com *locks*

Versão	**Atributo**	**Mínimo**	**Máximo**	**Médio**	**Desvio padrão**
STM *lazy*	Consumo	0.91	22.8	3.4	4.16
	Redução de desempenho	0.76	23.1	3.44	4.31
STM *eager*	Consumo	0.98	13.4	2.65	2.33
	Redução de desempenho	0.86	14.55	2.74	2.6

Tabela 3.2: STMs x *Locks* - Valores sumarizados

yada, que apresenta maior *overhead* de energia e maior fator de redução de desempenho em sua execução com 8 núcleos (mas não nas execuções com uma quantidade menor de núcleos), e nas aplicações *intruder* e *intruder+*, que são apresentadas em uma escala bem menor na versão *eager*.

Os valores de *overhead* de consumo e fator de redução de desempenho utilizados na elaboração dos gráficos acima foram sumarizados e são apresentados na Tabela 3.2. Os valores apresentados nesta tabela estão normalizados em relação à versão com *locks*.

As aplicações *intruder* e *intruder+* apresentaram valores muito altos de *overhead* de consumo e fator de redução de desempenho se comparadas as demais aplicações. Estes valores influenciaram de forma significativa o cálculo dos valores apresentados na Tabela 3.2. A Tabela 3.3 foi elaborada sem levar em consideração os valores das aplicações citadas, fornecendo resultados mais focados no que foi observado na grande maioria das aplicações.

Versão	**Atributo**	**Mínimo**	**Máximo**	**Médio**	**Desvio padrão**
STM *lazy*	Consumo	0.91	4.69	2.08	1.12
	Redução de desempenho	0.76	5.72	2.1	1.26
STM *eager*	Consumo	0.98	4.12	1.96	0.97
	Redução de desempenho	0.86	5.04	1.99	1.1

Tabela 3.3: STMs x *Locks* - Valores sumarizados (sem *intruder* e *intruder+*)

3.2 Análise de custo dos *locks*

Conforme explicado na Seção 3.1.4, no experimento realizado não foram considerados valores de energia consumidos no tempo em que a aplicação permanece em *loop* tentando adquirir um *lock*. Com o objetivo de compreender melhor o custo envolvido em se utilizar *locks*, alguns experimentos foram realizados.

Os experimentos executados consistiram em aplicar penalidades em ciclos sempre que uma tentativa de adquirir um *lock* não fosse realizada com sucesso. Esta penalidade representa a quantidade de ciclos necessários para que o sistema operacional escalone outra aplicação para execução. Um experimento conduzido em uma máquina Linux/x86 demonstrou que esta penalidade é, em média, de 10K ciclos. Como uma aquisição bem sucedida de um *lock* não representa um grande *overhead*, nenhuma penalidade foi aplicada para este caso. A aplicação *genome* foi então simulada variando-se a penalidade por *locks* não adquiridos entre 1K e 10K ciclos.

A Figura 3.7 apresenta os resultados obtidos com o experimento, denotando o valor da penalidade aplicada. O painel identificado pela palavra "base" demonstra os valores obtidos na execução da versão com *locks* sem nenhuma penalidade aplicada. Os demais painéis são identificados pela respectiva penalidade em ciclos. Conforme pode ser observado, existe um ponto de balanço a partir do qual a versão STM passa a ser comparável à versão com *locks*. A partir de 6K ciclos, o consumo de energia da versão STM em 8 núcleos é mais eficiente que a versão com *locks*. A partir de 8K ciclos, o consumo de energia da versão STM em 4 núcleos também supera a versão com *locks* em eficiência. Quando uma penalidade de 10K ciclos é aplicada, a versão STM em 8 núcleos apresenta mais eficiência que a versão com *locks* no que diz respeito ao tempo de execução.

Na Figura 3.7 ainda é possível observar que a diferença entre os valores expressos em dois painéis adjacentes tende a diminuir a medida que a penalidade cresce, indicando que parte dos ciclos de penalidade aplicados foram utilizados por outros processadores para concluir seções críticas concorrentes. Estas diferenças diminuíram de 21% e 9.75% para os valores de consumo de energia e fator de redução de desempenho no par <**Base, 1K**> para 3.22% e 2.8% no par <**9K, 10K**>. Ainda é possível observar que, à medida que o número de núcleos da execução aumenta, maior é a diferença entre os valores observados no par <**Base, 10K**>. Esta diferença, na execução com 8 núcleos, é de 153% para o *overhead* de energia e 80% para o fator de redução de desempenho.

Um experimento similar foi executado com a aplicação *intruder*, porém nenhum ponto de balanço pode ser encontrado, dado o alto *overhead* decorrente desta aplicação em sua versão STM.

Os resultados apresentados indicam que STMs podem ser interessantes em aplicações com baixas taxas de contenção, principalmente se altas penalidades estiverem inerentes

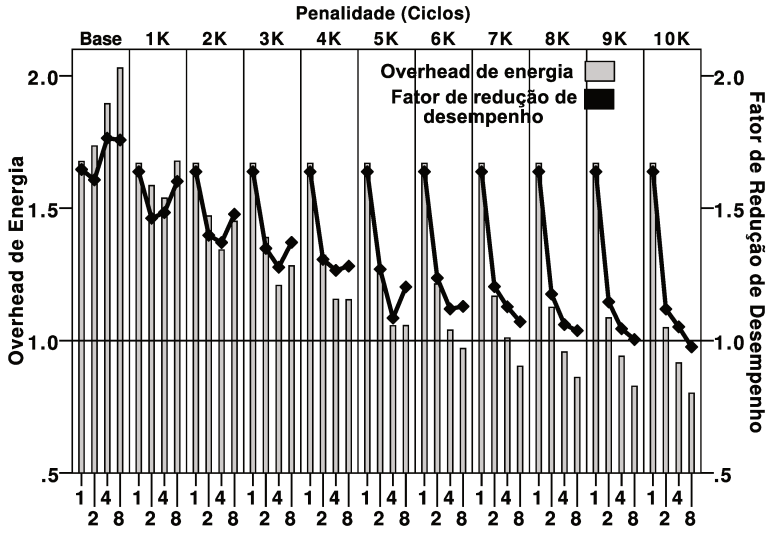


Figura 3.7: *Overhead* de energia e fator de redução de desempenho em relação à penalidade aplicada aos *locks* - Aplicação *genome*

.

ao uso *locks* no processo de sincronização.

3.3 Caracterização do consumo de STMs

O uso de STMs introduz a necessidade de operações primitivas para suportar a execução especulativa das transações. Estas operações, que são um *overhead* em relação ao software original, são responsáveis por inicializar, abortar e efetivar as transações, assim como detectar possíveis conflitos. Para analisar este *overhead*, o consumo de energia das aplicações foi separado de acordo com as primitivas responsáveis, o que possibilitou compreender a influência da STM. Os valores são apresentados nas Figuras 3.8, 3.9, 3.10 e 3.11 e estão normalizados em relação ao consumo observado na execução do código sequencial.

A partir da Figura 3.8, que apresenta o consumo das primitivas em uma execução com 1 núcleo, é possível observar que as aplicações *bayes*, *labyrinth*, *labyrinth+* e *yada* não apresentam *overhead* significativo em sua versão STM. Isso se dá devido as longas transações e ao baixo custo das primitivas transacionais. A aplicação *kmeans*, que também aparenta não possuir um grande *overhead*, amortiza os custos da STM devido ao uso de pesadas operações de ponto flutuante que são emuladas em software. Nas aplicações *ssca2*, *vacation*, *intruder* e *intruder+* o *overhead* mostra-se bastante acentuado devido a otimizações de compilação que não puderam ser aplicadas ao código paralelo.

Outra observação que pode ser feita a partir da Figura 3.8 é o fato de que a primitiva

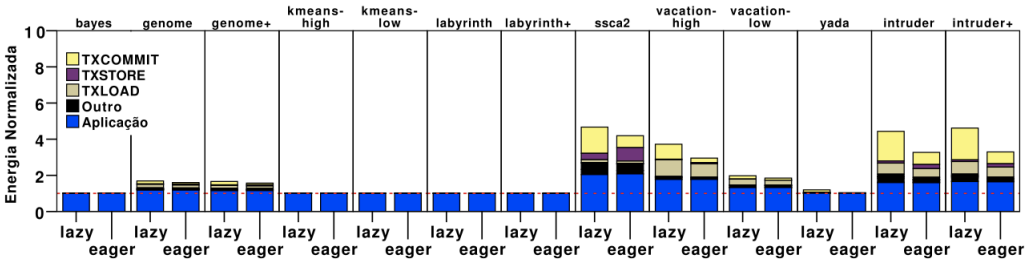


Figura 3.8: Análise de consumo do *benchmark* STAMP com a STM TL2 em 1 núcleo

.

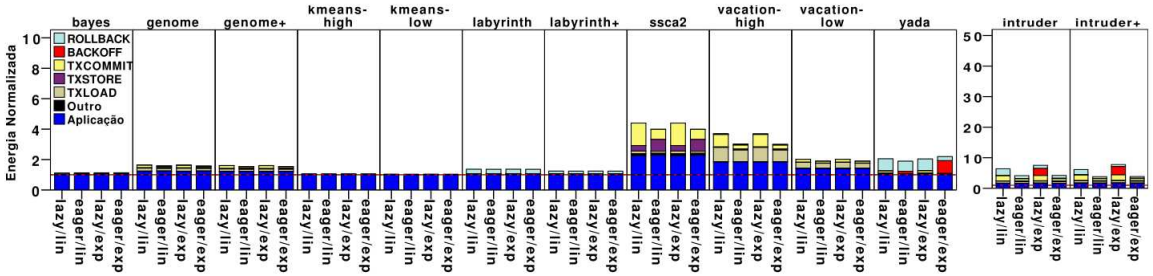


Figura 3.9: Análise de consumo do *benchmark* STAMP com a STM TL2 em 2 núcleos

.

TxStore custa menos na versão *lazy* da STM e a primitiva *TxCommit* custa menos na versão *eager*. Esta característica é explicada uma vez que a aquisição dos *locks* é realizada durante o *TxStore* na versão *eager*, enquanto na versão *lazy*, acontece durante o *TxCommit*. O custo da operação *TxLoad* também é menor na versão *eager*, uma vez que não é necessário consultar o conjunto de escrita sempre que um valor é lido.

As Figuras 3.9, 3.10 e 3.11 apresentam o consumo de energia das aplicações em suas versões paralelas, demonstrando também o custo dos *rollbacks* e *backoffs*. Nestes resultados é possível observar que o *overhead* na aplicação *kmeans* continua pequeno em decorrência das operações de ponto flutuante. As aplicações *bayes*, *genome*, *genome+*, *ssca2* e *vacation-low* apresentaram baixos *overheads* de *rollback* e *backoff* devido as suas baixas taxas de transações abortadas. Apesar de possuírem taxas de transações abortadas um pouco superiores, as aplicações *labyrinth* e *labyrinth+* não apresentam *overhead* de *backoff* significativo. Como o mecanismo de *backoff* só é aplicado após 3 tentativas mal-sucedidas de execução de uma transação, seu acionamento acaba sendo evitado pelos longos tempos de *rollback* decorrentes das longas transações destas aplicações. As demais aplicações apresentam custos de *rollback* e *backoff* altos e em diferentes proporções.

A Tabela 3.4 apresenta as taxas de transações abortadas das aplicações do *benchmark*

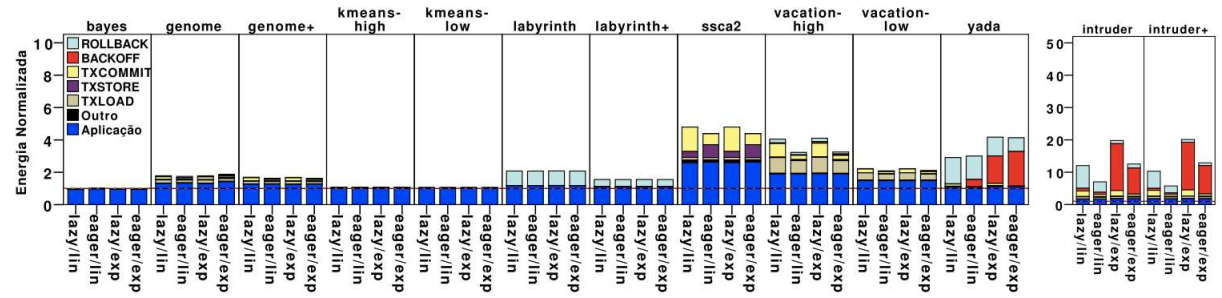


Figura 3.10: Análise de consumo do *benchmark* STAMP com a STM TL2 em 4 núcleos

.

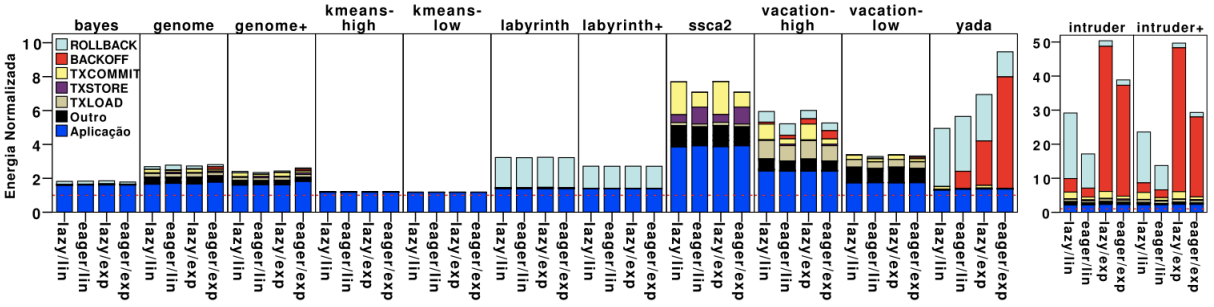


Figura 3.11: Análise de consumo do *benchmark* STAMP com a STM TL2 em 8 núcleos

.

STAMP. Conforme pode ser observado, os resultados obtidos nas simulações mostram que as aplicações *bayes* e *yada* apresentam taxas de contenção que não condizem com a classificação original (Tabela 3.1, retirada de [12]). Estas divergências são decorrentes das diferenças do ambiente de simulação utilizado nos dois trabalhos. Além das diferentes arquiteturas dos ambientes de simulação (*x86* e *ARM*), a plataforma MPARM utilizada nos testes deste trabalho impõe severas limitações de espaço em memória. Estas limitações exigiram que a aplicação *bayes* fosse adaptada para utilizar menos memória, com uma configuração de entradas ainda menor do que a versão mais leve sugerida pelo artigo.

3.4 Considerações finais

Parte dos resultados apresentados neste capítulo foi publicada [28] em co-autoria com Felipe Klein, Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo e Rodolfo Azevedo. A principal colaboração deste autor consistiu na realização dos experimentos e parte das análises que envolveram a versão com *locks* das aplicações. Além disto, a análise dos dados referentes as aplicações na versão STM *eager* não foi incluída na publicação.

Aplicação	**Número de núcleos**	**% de abortos *eager***	**% de abortos *lazy***
bayes	2	1.0	1.0
bayes	4	2.8	0.7
bayes	8	7.6	6.6
genome	2	2.1	0.6
genome	4	3.7	1.6
genome	8	9.6	3.8
genome+	2	0.5	0.3
genome+	4	1.3	0.8
genome+	8	3.2	2.0
intruder	2	31.3	25.4
intruder	4	53.3	50.4
intruder	8	68.8	67.7
intruder+	2	21.8	22.0
intruder+	4	42.3	41.6
intruder+	8	59.9	55.6
kmeans-high	2	1.0	0.9
kmeans-high	4	4.7	3.7
kmeans-high	8	15.5	10.5
kmeans-low	2	1.2	0.8
kmeans-low	4	3.1	1.6
kmeans-low	8	12.6	7.0
labyrinth	2	7.1	6.7
labyrinth	4	19.7	18.7
labyrinth	8	31.8	30.0
labyrinth+	2	6.4	6.4
labyrinth+	4	13.9	13.4
labyrinth+	8	33.0	29.4
ssca2	2	0	0
ssca2	4	0.1	0.1
ssca2	8	0.2	0.2
vacation-high	2	4.7	3.8
vacation-high	4	10.7	9.2
vacation-high	8	29.4	18.5
vacation-low	2	0.9	0.3
vacation-low	4	3.9	1.7
vacation-low	8	7.9	1.8
yada	2	85.9	41.8
yada	4	93.0	55.7
yada	8	95.6	52.1

Tabela 3.4: Taxas de transações abortadas

Capítulo 4

Plataforma híbrida para estimativa de consumo de energia

Os estudos apresentados no Capítulo 3 foram realizados com auxílio da plataforma de simulação MPARM, que possui precisão de ciclos e, por isso, é capaz de gerar resultados confiáveis em relação a um ambiente real de execução. No entanto, a complexidade inerente dos modelos com precisão de ciclos introduz pesados *overheads* de computação ao processo de simulação e, consequentemente, aumenta drasticamente o tempo total de execução. Esta característica, que se mostrou uma limitação bastante relevante nos estudos realizados no Capítulo 3, motivou o desenvolvimento de uma plataforma de simulação com abstração híbrida, composta por processadores funcionais escritos na linguagem ArchC integrados aos demais componentes da infra-estrutura do MPARM.

Neste capítulo será descrita a integração dos processadores ArchC na plataforma MPARM. Inicialmente serão apresentados os detalhes a respeito do funcionamento da plataforma, demonstrando a implementação da interface de comunicação e as técnicas utilizadas para diminuir a perda de precisão. Em seguida serão apresentados resultados experimentais que demonstram os ganhos de desempenho obtidos com a nova plataforma e os métodos estatísticos utilizados na correção destes resultados. Por fim, serão apresentadas as considerações finais e conclusões a respeito deste trabalho.

4.1 Implementação da plataforma híbrida

Durante as experiências realizadas no Capítulo 3 observou-se que o custo imposto pela precisão de ciclo da plataforma MPARM pode facilmente se transformar em um problema relevante ao desenvolvimento de pesquisas que exigem uma quantidade extensiva de simulações. Um *benchmark* normalmente é composto por um conjunto de diferentes aplicações que, por sua vez, podem ser executadas com diferentes configurações. Se considerarmos

que o desenvolvimento de um novo componente irá exigir subsequentes modificações em seu projeto e a execução de novos testes, a necessidade de esperar horas (ou até mesmo dias) por uma única simulação se transforma em um problema crítico. Dado este cenário, diminuir o tempo de simulação de uma plataforma torna-se crucial na avaliação do projeto de um componente.

Para ilustrar o cenário exposto, consideremos a versão baseada em *locks* da aplicação *genome* presente no *benchmark* STAMP, descrito no Capítulo 3. A execução desta aplicação na plataforma MPARM em uma configuração com um único processador levou 3:16 minutos. A medida que o número de processadores foi aumentado para 2, 4 e 8, o tempo total de simulação aumentou para 5:20, 8:47 e 21:21 minutos. Uma simulação com 16 núcleos, que não foi incluída no nosso conjunto de testes devido as limitações de tempo, teria levado em torno de 40 minutos.

Como a aplicação *genome* é uma das mais velozes do *benchmark* STAMP, é fácil concluir que o tempo de simulação de grandes aplicações é proibitivo. A execução da aplicação *yada* (outra aplicação do STAMP) com 8 núcleos levou aproximadamente 31 horas. Durante a exploração de uma nova arquitetura, em que uma mesma simulação será executava várias vezes, um tempo de simulação tão alto torna impraticável a realização de testes exaustivos, o que limita o tamanho do conjunto de simulações e prejudica uma avaliação mais apurada.

Considerando estas limitações, a necessidade de alternativas de simulação mais velozes, capazes de fornecer resultados confiáveis, torna-se clara.

4.1.1 Modificações aplicadas ao MPARM

Na busca por maior eficiência, decidiu-se substituir os simuladores de processador com precisão de ciclo presentes na plataforma MPARM por simuladores funcionais mais velozes. Originalmente o MPARM é distribuído com uma implementação do processador ARM chamada *SWARM simulator*. Esta implementação foi substituída por processadores gerados com o conjunto de ferramentas ArchC, tarefa que exigiu modificações na plataforma e no modelo ArchC utilizado. Este trabalho resultou em um simulador híbrido cujos módulos, exceto o processador, possuem precisão de ciclo. Conforme será visto, as estimativas de consumo obtidas através da nova plataforma não foram seriamente comprometidas e algumas técnicas estatísticas puderam ser aplicadas para aumentar sua precisão.

Na plataforma MPARM original, cada instância de processador consiste em uma implementação C/C++ de um simulador de conjunto de instruções (ISS[1]) encapsulado em uma camada *SystemC*. Esta camada funciona como uma interface entre o processador e o restante da plataforma, lidando com a comunicação entre eles. Esta camada também é

[1]Do inglês: Instruction Set Simulators

responsável por fornecer o controle para que um processador execute um novo ciclo de simulação, o que a torna um mecanismo eficiente para realizar a sincronização dos módulos da plataforma.

A linguagem de descrição de arquiteturas ArchC [5,37] é uma ferramenta *open-source*, baseada em *SystemC*, capaz de gerar ISSs rápidos e funcionais. Vários modelos de ISS diferentes já foram desenvolvidos, testados e estão disponíveis para *download* e uso no *site* da linguagem [42]. Adaptar um ISS desenvolvido com ArchC para utilizar portas TLM [21, 38] na comunicação com módulos externos é uma tarefa relativamente fácil, uma vez que este recurso é nativamente suportado pela linguagem.

O processador utilizado para substituir o *SWARM simulator* consiste em um modelo funcional do simulador ARMv5 gerado com a linguagem ArchC. O *pipeline* presente nesta arquitetura não foi modelado nesta implementação, o que a torna simples e, consequentemente, mais veloz. Devido à perda de precisão decorrente do uso de uma abstração de mais alto nível, algumas modificações precisaram ser aplicadas ao modelo e serão descritas a seguir.

No modelo original do processador ARMv5 gerado com o ArchC substituiu-se a memória interna por duas portas TLM, por onde todas as operações são emitidas ao barramento. O mecanismo de inicialização da memória de código foi removido, uma vez que já existe um recurso semelhante na plataforma MPARM. Modificações estruturais foram feitas no código da plataforma para instanciar os novos tipos de processador. Funções para estimar o consumo de energia dos novos processadores e gerar relatórios foram incluídas na plataforma. Finalmente, algumas modificações foram realizadas na estrutura de dados utilizada na comunicação entre o processador e o barramento AMBA, tornando-a compatível com a versão 2.1 da biblioteca *SystemC*. Esta modificação foi importante, uma vez que seria impossível compilar os modelos ArchC com suporte a TLM utilizando a versão 2.0.1, originalmente suportada pelo MPARM.

Toda a implementação exigiu a programação de aproximadamente 500 linhas de código para escrever as interfaces TLM e a camada *SystemC* que encapsula os novos processadores. Aproximadamente 200 linhas de código foram escritas na plataforma original para suportar os processadores ArchC. O código escrito pode ser facilmente reutilizado para integrar diferentes processadores modelados com ArchC na plataforma MPARM, permitindo a exploração de diferentes arquiteturas ainda não suportadas pelo MPARM. Este novo recurso expande o contexto de aplicações da plataforma, tornando-a muito mais abrangente.

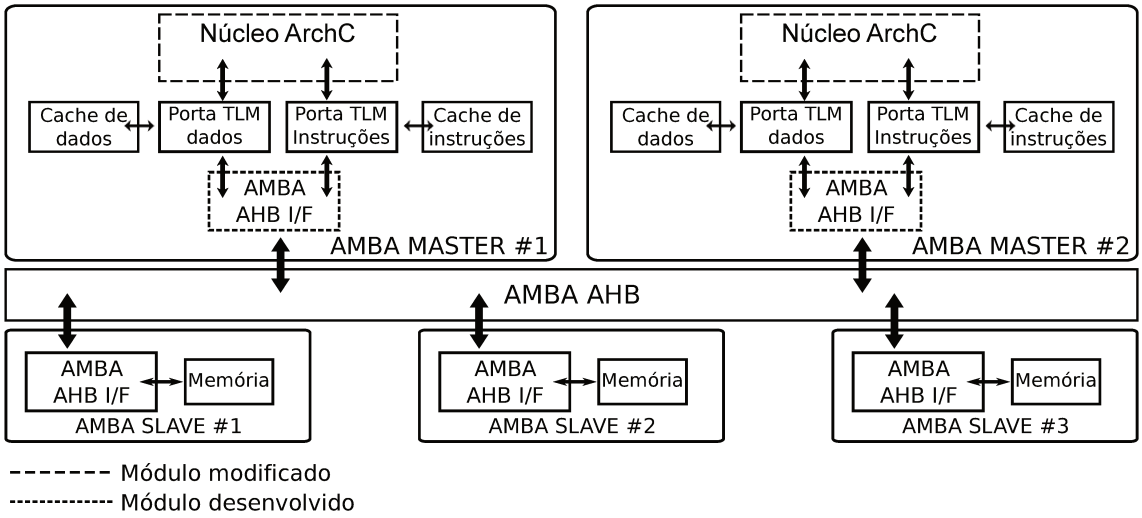


Figura 4.1: Plataforma MPARM com Processadores ArchC

4.1.2 Integração do modelo ArchC

A linguagem ArchC suporta nativamente o uso de portas TLM em seus modelos. Este recurso permite que os processadores gerados se comuniquem facilmente com módulos externos, como memórias e barramentos. Quando este recurso é utilizado, dados que descrevem a operação de memória são encapsulados em um pacote e submetidos para a interface TLM. Esta interface, que fornece funções e modelos necessários para realizar a conexão entre o processador e a plataforma, deve ser implementada pelo usuário, uma vez que grande versatilidade é necessária para atender os requisitos particulares de cada plataforma em que os modelos são integrados.

Por ser um canal centralizado de comunicação entre o processador e o barramento, a interface TLM é o ponto ideal para a implementação dos mecanismos de hierarquia de memória. Como todas as operações de leitura e escrita são encaminhadas para esta interface, conectar memórias cache ao sistema consistiu em apenas implementar os módulos em questão e invocar suas funções entre as operações. Além de simplificar a estrutura da plataforma, esta decisão permitiu que a hierarquia de memória fosse implementada sem modificar o modelo dos processadores.

Para suportar memórias cache de dados e instrução exclusivas, duas portas TLM foram criadas no modelo do processador. Uma porta é exclusivamente utilizada para leitura de instruções e a outra é utilizada para os demais acessos a memória. Cada porta TLM é conectada a sua própria memória cache, que é consequentemente transformada em uma cache de instruções ou dados. Um diagrama que detalha a implementação da interface TLM e a integração dos processadores na plataforma pode ser visto na Figura 4.1.

A plataforma MPARM não utiliza TLM em sua comunicação. Por esta razão, uma

vez que chegam na interface TLM, os pacotes são traduzidos para a estrutura de dados de sinalização[2] do MPARM e uma solicitação de operação de memória é enviada ao mestre do barramento. Estas operações de memória bloqueiam o processador até que seja recebido o sinal que informa o fim da transação.

Alguns endereços de memória são utilizados pela plataforma para criar um canal de comunicação entre a aplicação simulada e o próprio simulador. Este recurso é útil para permitir o uso de uma API de suporte à simulação, que fornece chamadas para funcionalidades como habilitar e desabilitar a medição de consumo ou imprimir mensagens na tela.

O barramento AMBA original não foi modificado e sua precisão de ciclos original foi mantida. Por este motivo, quando operações de memória são realizadas, o processador funcional permanece bloqueado pelo mesmo número de ciclos que o processador original ficaria. Como o processador permanece bloqueado até receber o sinal de fim da transação, a interface TLM força alguma sincronização entre o processador e os demais módulos da plataforma.

Para fornecer uma estimativa de consumo de energia correta, a interface TLM também foi utilizada para encapsular chamadas para a API de medição de consumo. O *SWARM simulator*, originalmente embutido no MPARM, é modelado como um autômato com 8 estados. Cada um destes estados descreve um modo de operação do processador e suas transações são definidas por eventos internos, como operações nas memórias cache e principal. Em cada ciclo simulado, o estado do processador é contabilizado pelas chamadas da API de medição de consumo. No final da simulação, o número de ciclos em cada estado é utilizado na estimativa de consumo.

Como o processador ArchC é funcional, o mecanismo de estimativa de consumo não pode ser diretamente reutilizado e precisou ser reorganizado na nova implementação. Uma opção seria utilizar fluxos de estado ou valores de consumo estimados fixos para cada instrução simulada, como feito em [33], mas esta abordagem implicaria em grande perda de precisão devido a rigidez implícita no modelo. A solução alternativa proposta consistiu em realizar chamadas à API de medição de acordo com o fluxo de eventos da interface TLM. Como as operações na cache e na memória principal são realizadas por chamadas feitas dentro do escopo da interface TLM, esta abordagem permitiu determinar dinamicamente uma sequência de estados do processador para cada instrução simulada. Utilizar esta abordagem melhorou bastante a precisão da estimativa de consumo da plataforma híbrida.

Um esquema ilustrando o fluxo acima descrito, em que a organização das chamadas à API de medição na interface TLM são demonstradas, pode ser visto na Figura 4.2.

[2]Core Signal

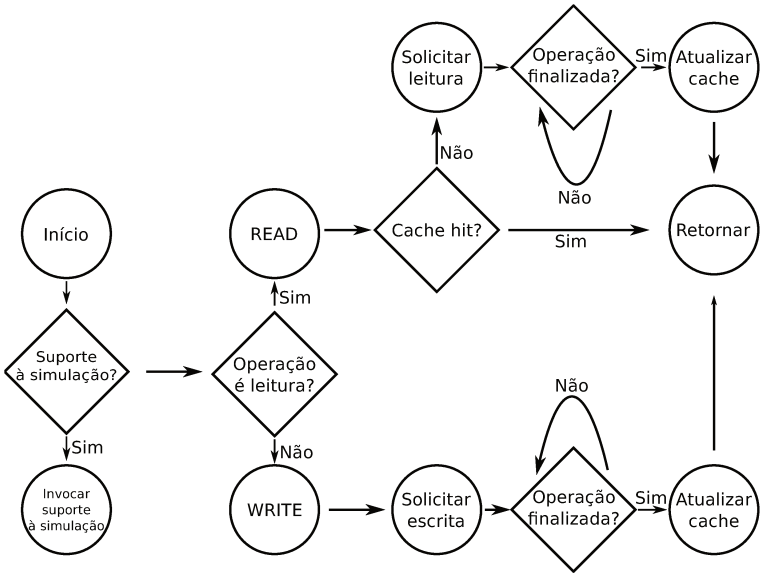


Figura 4.2: Fluxo TLM

4.1.3 Modificações aplicadas ao modelo ArchC ARMv5

Com o objetivo de aumentar a velocidade de simulação, os modelos ArchC utilizam uma cache para instruções decodificadas. Esta cache armazena os resultados da decodificação de instrução em uma estrutura de dados que utiliza o respectivo endereço de memória como índice. Quando esta instrução é novamente necessária, ela pode ser obtida já decodificada a partir desta cache. Apesar de ser essencial para atingir alta eficiência, este mecanismo evita a execução das operações de memória para busca de uma instrução previamente decodificada, o que resulta em contagens erradas de acessos a memória principal e cache. Para solucionar este problema, o mecanismo foi modificado para realizar um acesso figurativo[3] à memória no endereço correspondente sempre que a instrução em questão já estiver na cache. Esta solução força o modelo a realizar os mesmos acessos a memória originais, mas ainda evita que a mesma instrução seja decodificada mais de uma vez. Este problema pode ser melhor compreendido na Figura 4.3.

Em uma primeira comparação entre o modelo com precisão de ciclos e o modelo funcional foi constatada uma grande diferença no número de operações de leitura de memória. Esta diferença é um efeito gerado pela inexistência de um *pipeline* no simulador gerado com ArchC. No modelo com precisão de ciclos, a execução de uma instrução de *branch* pode resultar no descarte das instruções que preenchem o primeiro e o segundo estágio do seu *pipeline*. Como no modelo funcional não existe um *pipeline*, uma instrução de *branch* é obtida da memória e executada atomicamente, o que impede a busca das instruções

[3]Tradução livre do Inglês para o termo *dummy*

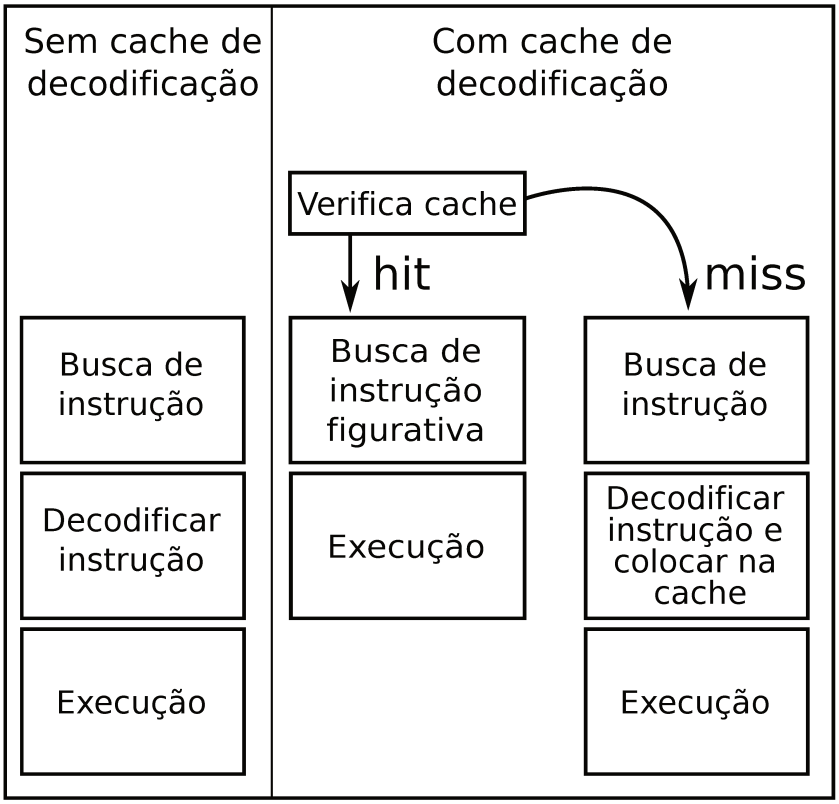


Figura 4.3: Fluxo da cache de decodificação

seguintes que, eventualmente, seriam descartadas.

Para contornar a diferença decorrente do uso de uma abstração mais simplificada e obter resultados mais precisos, um mecanismo de detecção de *branches* foi implementado dentro do processador ArchC. Sempre que a execução de um *branch* modifica o fluxo de execução do programa, duas leituras de memória figurativas são geradas nos endereços das instruções que, em um simulador com precisão de ciclos, estariam no *pipeline* e seriam descartadas. Este mecanismo aumentou a similaridade das simulações com diferentes abstrações e permitiu medições de consumo de energia mais precisas.

Utilizar o recurso apresentado na Figura 4.3 e o mecanismo de detecção de *branches* introduz complexidade ao simulador e diminui seu desempenho. Durante a realização de testes para os quais a quantidade de acessos a memória não importa, este mecanismo pode ser removido para aumentar a velocidade das simulações. Todos os testes realizados na nova plataforma e apresentados neste capítulo utilizaram estes recursos.

4.1.4 Verificação da plataforma

A nova implementação desenvolvida foi testada com o uso do *benchmark* STAMP, descrito na Seção 3.1.5. Este *benchmark* foi portado para execução na plataforma MPARM durante os estudos apresentados no Capítulo 3. Além disso, outras características do STAMP mostraram-se adequadas aos requisitos de teste da plataforma, pois sua concorrência inerente contribuiu para a verificação das interfaces TLM, acesso à memória e sincronização. Além disso, a variedade de algoritmos do STAMP possibilitou o teste da plataforma em diferentes condições.

Nos testes realizados, utilizou-se uma versão do STAMP sequencial e outra paralela, sincronizada com um mecanismo de *spin-locks*[4]. Durante o processo, as 8 aplicações disponíveis no STAMP foram executadas com configurações de parâmetros sugeridas pelo artigo original em que *benchmark* foi publicado [12]. As aplicações *genome, intruder, kmeans, labyrinth* e *vacation* foram executadas em 2 configurações diferentes, totalizando um conjunto de simulações com 13 aplicações.

A verificação da plataforma foi realizada em três etapas. Em um primeiro momento, as aplicações foram executadas e suas saídas foram comparadas com as saídas geradas pela plataforma original. Inicialmente os testes foram realizados com aplicações pertencentes a *micro-benchmarks*, mas foram posteriormente concluídas com a execução de todas as aplicações do STAMP. Nesta etapa, todos os testes foram executados com um único processador instanciado na plataforma.

A segunda etapa consistiu na comparação de *traces*[5] de memória gerados pelas duas plataformas. Este teste permitiu verificar se as operações de memória e traduções de endereços foram executadas corretamente. Depois de comparar os *traces*, relatórios de saída gerados pelas plataformas também foram analisados, mostrando que as duas plataformas realizaram a mesma quantidade de operações de memória quando executaram o mesmo *benchmark*. Mais uma vez, apenas um processador foi instanciado na plataforma.

A última etapa da verificação consistiu em realizar novamente a primeira e a segunda etapas com múltiplos processadores instanciados (2, 4 e 8). Uma vez que as saídas esperadas foram geradas pelas aplicações, os *traces* de memória foram comparados. Como as diferentes abstrações implicam em diferentes contenções no barramento, algumas diferenças no fluxo de execução das aplicações e nos *traces* foram observadas.

Os relatórios de saída gerados pelas plataformas foram comparados e os números de operações de memória realizados por cada uma apresentou diferenças de 4,32%, em média. Neste cálculo foram desconsideradas leituras e escritas na área de memória destinada à comunicação entre a aplicação simulada e o simulador. Nos valores encontrados na

[4]Tipo de *lock* em que tentativas de aquisição são realizadas consecutivamente até que uma seja bem sucedida

[5]Lista de operações ou ações

plataforma híbrida, foi notável a melhoria no número de leituras de memória na área privada após a aplicação do mecanismo de detecção de *branches*, mencionado na Seção 4.1.3.

A porcentagem de diferença por operação e área de memória lida entre 4 aplicações pode ser vista na Tabela 4.1. Valores apresentados com o sinal + denotam que a simulação realizada com o processador ArchC executou um número maior de operações de memória. Valores descrevendo simulações com mais de um processador referem-se ao número de operações executados pelo primeiro núcleo instanciado.

Operação	**Memória**	**genome 1 núcleo**	**genome 8 núcleos**	**Intruder 1 núcleo**	**Intruder 8 núcleos**
Leitura	Privada	0.33%	9.16%	0.05%	14.04%
Escrita	Privada	0%	0.2%	0%	+1.44%
Leitura	Compartilhada	0%	0.97%	0%	2.89%
Escrita	Compartilhada	0%	1.08%	0%	+ 1.54%
Leitura	Semáforo	0%	3.12%	0%	2.47%
Escrita	Semáforo	0%	0.49%	0%	+ 1.06%

Tabela 4.1: Diferenças no número de operações de memória

Conforme demonstrado na Tabela 4.1, o número de leituras na área privada de memória é significativamente diferente para as simulações com 8 núcleos. Esta característica é um efeito das diferenças no acesso à memória introduzidas pela mudança de abstração, que impõe diferentes contenções no barramento. Com um número maior de processadores, a contenção no barramento torna-se mais sensível à mudança de abstração, o que gera mudanças no fluxo de execução das aplicações. Como as instruções da aplicação simulada são armazenadas na memória privada, o número de leituras nesta área de memória é bastante influenciado por estas mudanças, o que explica a diferença observada em simulações com maior número de processadores. Conforme pode ser observado, apenas uma pequena porcentagem de erros foi detectada quando as aplicações foram executadas com 1 núcleo, o que evidencia os efeitos da contenção e do fluxo de execução nas simulações com 8 processadores.

Após finalizar o processo de verificação, uma série de experimentos foram realizados para avaliar a eficiência e a medição de consumo de energia da nova implementação. Estes experimentos são discutidos na Seção 4.2.

4.2 Avaliação da plataforma

O desempenho da plataforma híbrida foi avaliado em duas etapas diferentes. Primeiro foi avaliado o ganho de desempenho obtido com a nova implementação, comparando o tempo

de execução das mesmas 13 simulações de aplicações do *benchmark* STAMP utilizadas na verificação da nova plataforma. As plataformas foram comparadas em simulações com 1, 2, 4 e 8 processadores instanciados. Além das versões paralelas, uma versão sequencial das aplicações também foi simulada, totalizando 65 simulações.

A segunda etapa de avaliação consistiu em mensurar o erro na estimativa de consumo de energia introduzido pelo uso de uma abstração mais simplificada. Os resultados gerados pelas 65 simulações foram comparados e, a partir deles, modelos estatísticos foram calculados para aproximar os resultados. Após seu cálculo, o modelo estatístico foi aplicado a um conjunto de testes para verificar sua eficiência.

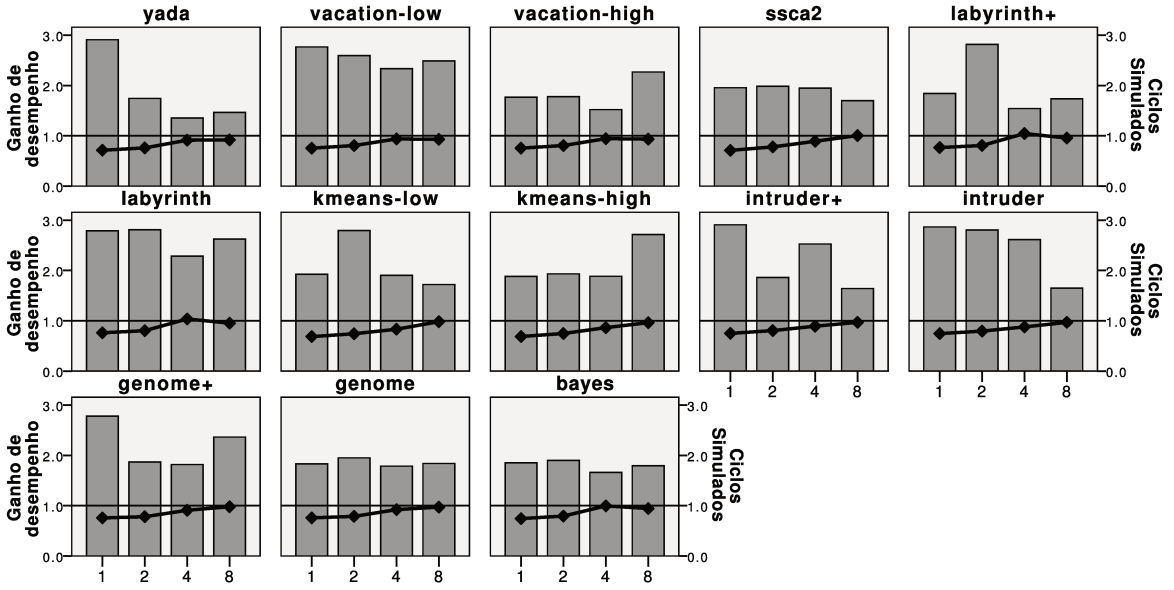
Todas as simulações foram executadas em máquinas com processadores de 2,4GHz e 4GB de memória RAM. As máquinas executavam o sistema operacional Ubuntu Linux, com o Kernel 2.6.9. Devido a restrições impostas pelo código fonte da plataforma, todo o código foi compilado com o GCC 3.4 e a opção "$-O3$" foi aplicada para especificar as otimizações de compilação.

4.2.1 Ganho de desempenho

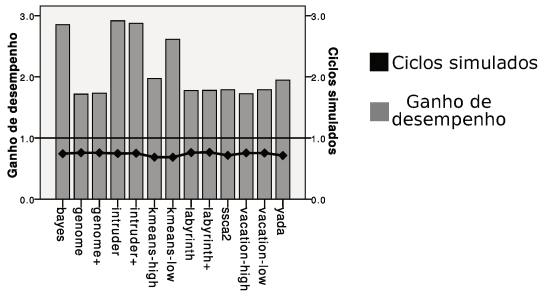
Após finalizar as 65 simulações com a plataforma híbrida, o resultado de cada uma foi comparado com a simulação similar executada na plataforma MPARM original. O ganho de desempenho obtido com a plataforma híbrida pode ser visto nas Figuras 4.4(a) e 4.4(b), cuja representação é feita pelas barras de cor cinza. A identificação de cada simulação segue a nomenclatura apresentada em [12]. Como pode ser visto, a implementação híbrida atingiu um ganho de desempenho de pelo menos 1.8x para cada simulação. O ganho de desempenho médio obtido foi de 2.1x e o máximo foi de 2.9x.

A ausência de um *pipeline* reduziu o número total de ciclos executados para cada simulação. O número de ciclos, normalizado em relação aos valores obtidos com a plataforma original, pode ser identificado por uma linha preta nas Figuras 4.4(a) e 4.4(b). Conforme demonstrado, 70% dos ciclos originais foram executados na simulação sequencial. Este valor também permanece nas simulações da versão do STAMP baseada em *spin-locks* com 1 processador, mas aumenta quando mais processadores são instanciados na plataforma. Isto acontece devido a uma maior contenção do barramento decorrente da inclusão dos novos processadores. Como o barramento não foi modificado e possui precisão de ciclos, operações de memória bloqueiam as duas implementações pelo mesmo tempo, o que faz o número de ciclos da simulação com processadores funcionais aumentar em direção ao número de ciclos obtido com a plataforma original.

Simular menos ciclos não foi a única razão para obter o desempenho apresentado. A abstração mais alta dos processadores ArchC os torna mais simples e, consequentemente, mais velozes que o *SWARM Simulator*. Em média, utilizar os processadores ArchC au-



(a) Implementação *lock-based* por quantidade de núcleos



(b) Implementação Sequencial

Figura 4.4: Desempenho / Ciclos simulados

mentou o número de ciclos simulados por segundo em 1.9x, atingindo um máximo de 2.4x. Uma comparação de ciclos simulados por segundo pode ser vista na Tabela 4.2. Nesta tabela, colunas identificados pelas letras "AC" referem-se a simulações realizadas com o processador ArchC e colunas identificadas com as letras "SW" referem-se a simulações realizadas com o *SWARM Simulator*.

Executar as 65 simulações em série levaria 187 horas e 30 minutos no MPARM original. Este mesmo conjunto de simulações foi realizado em 98 horas e 19 minutos na implementação híbrida do MPARM.

Simulações que utilizam uma configuração mais complexa, como plataformas com 16 processadores instanciados, são uma tarefa que levaria muito tempo para ser concluída

Aplicação	Sequencial		1 núcleo		2 núcleos		4 núcleos		8 núcleos	
	AC	SW	AC	SW	AC	SW	AC	SW	AC	SW
kmeans-low	270	150	276	209	406	196	546	345	870	515
kmeans-high	275	203	279	216	407	282	587	360	938	359
yada	297	214	295	142	409	308	465	375	727	540
bayes	318	150	312	227	487	322	657	396	991	584
intruder	351	161	341	159	480	216	603	262	889	556
intruder+	345	160	350	160	488	325	590	261	894	562
labyrinth	319	236	317	149	461	204	575	243	892	356
labyrinth+	318	233	316	223	465	204	575	357	880	530
vacation-low	352	260	341	163	468	223	588	268	906	391
vacation-high	340	261	347	260	479	333	613	427	880	415
ssca2	326	254	339	244	502	324	710	408	1063	623
genome	324	248	322	232	519	337	711	431	1083	605
genome+	334	255	330	157	520	355	709	428	1075	464

Tabela 4.2: Número de K ciclos simulados por segundo

com a plataforma MPARM original. Adicionar todas as aplicações do STAMP executando em 16 processadores ao conjunto de simulações descrito certamente aumentaria significativamente o tempo total de simulação. Nos estudos apresentados no Capítulo 3, esta configuração foi removida do conjunto de simulações devido a limitações de eficiência. Pelo mesmo motivo, algumas configurações do STAMP que exigiriam maior poder computacional não foram testadas.

Simulações com as 13 diferentes aplicações em uma configuração com 16 processadores instanciados foram realizadas na plataforma híbrida para avaliar o seu desempenho em execuções de trabalhos mais complexos e inviáveis de se realizar com precisão de ciclo. Este conjunto de 13 simulações, se executado em série, levaria 88 horas e 17 minutos para ser finalizado na plataforma híbrida. A execução em série destas simulações em conjunto com as 65 simulações inicialmente utilizadas levaria 186 horas e 36 minutos. Como pode ser observado, mesmo sem realizar as simulações com 16 processadores, a plataforma com precisão de ciclos ainda é 54 minutos mais lenta que a implementação híbrida.

4.2.2 Estimativas de energia e consumo

A estimativa de consumo de energia no MPARM é calculada com base na contagem de estados do processador, conforme mencionado na Seção 4.1.2. A cada ciclo, o estado em que o processador se encontra é acumulado em um contador e, ao final da simulação, o valor deste contador é aplicado ao modelo de energia incluso na plataforma. Os valores das estimativas de energia obtidos durante a simulação são apresentados no relatório final gerado pela plataforma, na forma de energia (pJ) e potência (mW).

Conforme esperado, o uso de uma abstração menos detalhada introduziu imprecisão nos cálculos de consumo de energia. Um gráfico demonstrando a perda na precisão da estimativa de energia pode ser visto na Figura 4.5. Neste gráfico, os pontos representam os resultados obtidos com a plataforma híbrida e a linha diagonal representa o valor obtido com a plataforma original, utilizado como referência de corretude para a medição.

O erro médio encontrado nas comparações apresentadas na Figura 4.5 é de 6.25%, com um máximo de 27.3% e um mínimo de 0.7%. As aplicações *bayes, labyrinth, labyrinth+* e *yada* apresentaram um erro maior do que a média encontrada quando executadas com 4 processadores. Estas aplicações possuem longas seções críticas e, por isso, não apresentaram boa escalabilidade nesta configuração. Por simular a mesma quantidade de instruções em um número menor de ciclos, a abstração funcional realiza maior quantidade de operações de memória em um mesmo intervalo de tempo. Esta característica, combinada às sucessivas tentativas de aquisição de *locks* para execução das seções críticas, aumenta a carga no barramento fazendo-o atingir seu estado de contenção máxima, em que todos os processadores aguardam a conclusão de operações simultaneamente. Como a plataforma com precisão de ciclos não introduz sobrecarga no barramento, estas simulações não apresentaram o problema de contenção quando executadas nesta abstração, o que deu origem a divergência dos resultados. Esta análise é evidenciada pela quantidade de ciclos simulados que, na plataforma híbrida, atingiram ou superaram os valores observados na simulação com precisão de ciclos. Quando estas aplicações foram simuladas com 8 processadores, ambas plataformas atingiram seu estado de contenção máxima e, por isso, a comparação dos resultados não apresentou erros acima da média.

A estimativa de potência é obtida através do valor de energia dividido pela quantidade de ciclos simulados. Como a mudança de abstração introduz imprecisão não só na contagem de estados do processador, mas também na quantidade de ciclos simulados, as taxas de erro são propagadas e se tornam mais visíveis no cálculo de potência. Os resultados gerados diretamente pela plataforma híbrida apresentaram um erro médio de 21.2%, com um máximo de 27.3% e um mínimo de 17.0%.

Para melhorar os resultados de potência obtidos, um modelo baseado em análise de regressão foi construído com o uso de método dos mínimos quadrados. Para construir o modelo, os valores medidos na plataforma original foram utilizados como valores ideais. Com o modelo obtido foi possível reduzir o erro médio para 14.45%, com um máximo de 21.34% e um mínimo de 8.7%. Por cobrir todas as simulações, este é identificado na Tabela 4.3 como modelo genérico.

Conforme demonstrado na Figura 4.4(a) e explicado na Sessão 4.2.1, a proporção de ciclos simulados não é a mesma para simulações com um número diferente de processadores instanciados. Esta variação no número de ciclos tem reflexos nas estimativas de consumo geradas. Dada esta característica, decidiu-se calcular novos modelos específicos

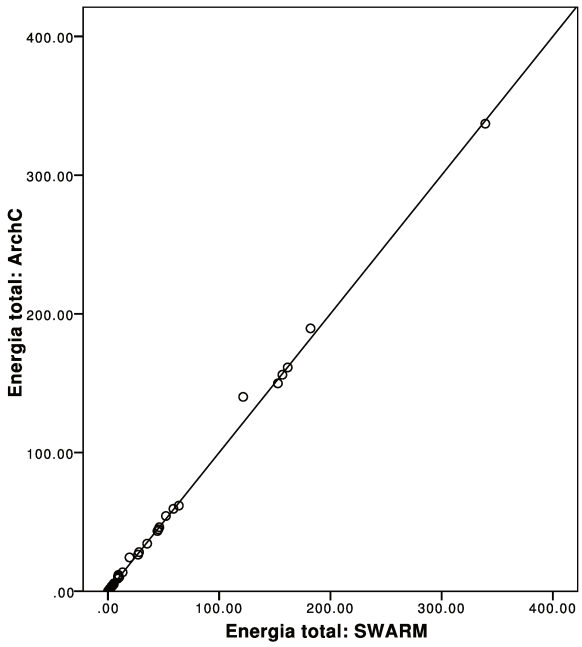


Figura 4.5: Erros na medição de energia total (conjunto das 65 aplicações utilizadas na avaliação da plataforma)

núcleos	**modelo**
modelo genérico	$0.9 \times x - 3$
1	$0.71 \times x + 0.57$
2	$0.8 \times x$
4	$0.75 \times x + 7.5$
8	$0.8 \times x + 24$

Tabela 4.3: Coeficientes da Regressão Linear

para cada número de processadores utilizados na simulação. Os novos coeficientes calculados também estão apresentados na Tabela 4.3. Nesta tabela, a variável x representa o valor diretamente estimado pela simulação funcional.

Ao aplicar os novos modelos específicos para o número de processadores, a margem de erro foi reduzida para uma média de 3.25%, com um máximo de 7.9% e um mínimo de 0.02%. Um gráfico demonstrando o erro dos resultados finais pode ser visualizado na Figura 4.6. Neste gráfico, os pontos representam os resultados após aplicar os modelos de aproximação e a linha diagonal representa o valor obtido na plataforma original.

É importante observar que a precisão de um modelo de regressão linear depende da quantidade de dados utilizados na sua formulação. Um conjunto de dados mais amplo fornece maior abrangência ao modelo, tornando-o mais confiável e versátil. Para avaliar os

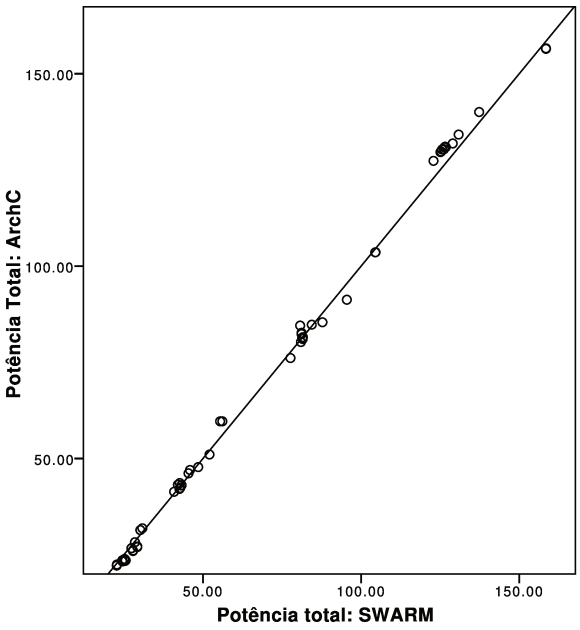


Figura 4.6: Erros na medição de potência total (conjunto das 65 aplicações utilizadas na avaliação da plataforma)

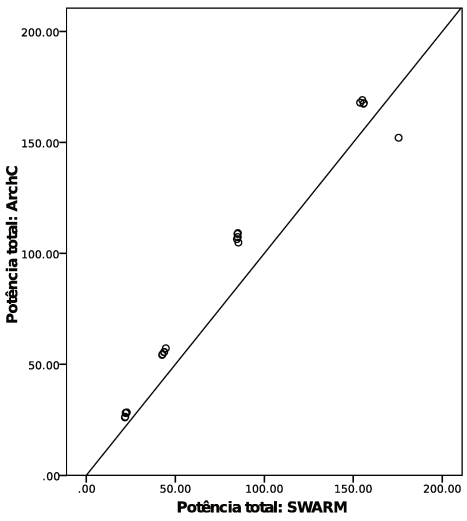
modelos formulados, um pequeno conjunto de teste, composto por aplicações do STAMP executadas com entradas de dados diferentes, foi elaborado. Aos resultados obtidos foram aplicados o modelo genérico e os modelos específicos previamente apresentados. Os resultados estão apresentados na Tabela 4.4, cujos campos marcados com a letra G identificam valores modificados pelo modelo genérico e campos marcados com a letra S identificam valores modificados pelo modelo específico. Nesta tabela, os erros estão calculados com relação ao valor original, obtido com o processador SWARM.

Conforme pode ser observado na Tabela 4.4, após aplicar o modelo genérico nos resultados obtidos, o erro médio foi reduzido de 22.21% para 18.74%. Quando o modelo específico foi aplicado, o erro médio reduziu para 5.85%. Apesar de apresentar uma eficiência inferior ao modelo específico para os demais casos, o modelo genérico mostrou-se mais preciso quando aplicado a resultados simulados com 8 processadores. Gráficos que demonstram os erros obtidos com os modelos podem ser vistos nas Figuras 4.7(a), 4.7(b) e 4.7(c).

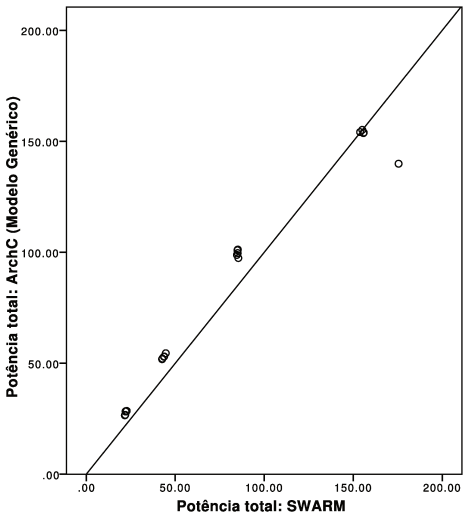
Como o conjunto de simulações utilizado na regressão linear possui apenas 65 configurações, os modelos foram prejudicados pela falta de amplitude dos dados gerados. Esta característica é ainda mais crítica nos modelos específicos, que exigiram a divisão das 65 configurações em subconjuntos de acordo com o número de processadores da simulação.

			Sequencial	1 Núcleo	2 Núcleos	4 Núcleos	8 Núcleos
Aplicações	Bayes	Valor SWARM	22.73	22.70	44.67	85.43	175.46
		Valor ARCHC	28.37	28.31	57.25	104.90	152.12
		Valor G	28.53	28.48	54.52	97.41	139.91
		Valor S	20.71	20.67	45.80	86.18	145.70
		Erro inicial	24.8	24.7	28.2	22.8	13.3
		Erro G	25.53	25.46	22.06	14.02	20.26
		Erro S	8.88	8.94	2.53	0.87	19.96
	Intruder	Valor SWARM	22.09	22.14	43.65	84.63	153.85
		Valor ARCHC	28.11	28.23	55.51	106.46	167.94
		Valor G	28.30	28.41	52.96	98.81	154.15
		Valor S	20.53	20.61	44.41	87.34	158.35
		Erro inicial	27.3	27.5	27.2	25.8	9.2
		Erro G	28.11	28.31	21.33	16.76	0.19
		Erro S	7.07	6.90	1.74	3.21	2.93
	Intruder+	Valor SWARM	22.13	22.19	43.76	85.07	155.07
		Valor ARCHC	28.04	28.19	55.63	107.37	169.08
		Valor G	28.24	28.37	53.07	99.63	155.17
		Valor S	20.48	20.58	44.50	88.03	159.26
		Erro inicial	26.7	27.0	27.1	26.2	9.0
		Erro G	27.59	27.85	21.27	17.12	0.07
		Erro S	7.46	7.23	1.70	3.48	2.70
	Labyrinth	Valor SWARM	21.68	21.68	42.61	84.92	155.71
		Valor ARCHC	26.14	26.14	54.27	108.78	167.44
		Valor G	26.53	26.53	51.84	100.90	153.70
		Valor S	19.13	19.13	43.42	89.09	157.95
		Erro inicial	20.06	20.06	27.4	28.1	7.5
		Erro G	22.35	22.35	21.67	18.82	1.29
		Erro S	11.76	11.76	1.89	4.90	1.44
	Labyrinth+	Valor SWARM	21.90	21.90	42.79	85.15	155.91
		Valor ARCHC	26.40	26.40	54.55	109.14	167.72
		Valor G	26.76	26.76	52.10	101.23	153.95
		Valor S	19.31	19.31	43.64	89.36	158.18
		Erro inicial	20.5	20.5	27.5	28.2	7.6
		Erro G	22.19	22.19	21.75	18.88	1.26
		Erro S	11.81	11.81	1.99	4.94	1.45

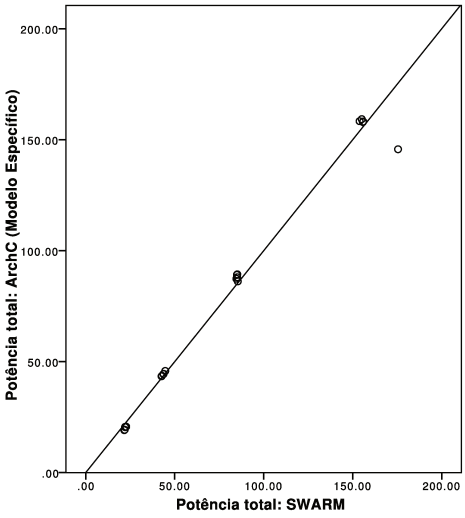
Tabela 4.4: Estimativas de consumo



(a) Dados originais



(b) Modelo genérico



(c) Modelo específico

Figura 4.7: Erros obtidos com conjunto de testes

4.3 Conclusões e considerações finais

Conforme demonstrado, a plataforma híbrida desenvolvida apresentou bons ganhos de desempenho e moderada perda de precisão nos valores medidos em relação a plataforma com precisão de ciclos. Estes resultados demonstram que o uso de abstrações mais simplificadas ou até mesmo o uso de múltiplas abstrações na mesma plataforma de simulação pode ser interessante para aumentar a agilidade do desenvolvimento de novos componentes de

hardware.

Na plataforma desenvolvida, observou-se melhores resultados em simulações realizadas com configurações mais complexas, no caso, com 8 ou mais processadores. Nestes casos, que normalmente consistem nas simulações mais longas, as estimativas de energia mostraram-se pouco prejudicadas e os desempenhos obtidos foram os mais altos, fato que torna ainda mais justificável a aplicação de tais recursos.

Nesta plataforma, vários mecanismos foram utilizados para figurar comportamentos que exigiriam precisão de ciclos para serem simulados. Estes mecanismos melhoraram significativamente os resultados obtidos na plataforma de simulação híbrida e as idéias por trás de suas implementações podem ser utilizadas durante a integração dos simuladores ArchC (ou outros simuladores funcionais) em outras plataformas de simulação com precisão de ciclos.

Através deste trabalho, abre-se a possibilidade de integrar simuladores ArchC de outras arquiteturas à plataforma MPARM. Dadas a variedade de modelos existentes e a facilidade para se desenvolver novas arquiteturas com a linguagem ArchC, a amplitude de aplicação da plataforma é significativamente aumentada, tornando-a mais versátil, genérica e não mais limitada ao escopo de sistemas embarcados.

O uso de técnicas de regressão linear pode auxiliar na correção dos valores prejudicados pela mudança de abstração da simulação. Os dados apresentados demonstram que os valores obtidos nas simulações tendem a se comportar de maneira similar quando realizados com a mesma quantidade de processadores, o que possibilita a formulação de modelos de aproximação específicos para cada grupo.

Para melhorar os modelos formulados seria ideal utilizar um conjunto de dados mais amplo. Como a obtenção de tais conjuntos depende da realização de simulações com diferentes configurações, maior complexidade computacional e em ambas as plataformas, esta necessidade esbarrou em limitações de tempo e, por isso, não pode ser devidamente cumprida. A obtenção de modelos mais precisos, baseados em um maior conjunto de dados, fica como uma sugestão de trabalhos futuros.

Capítulo 5

Conclusão

Neste trabalho foram apresentadas contribuições que buscam auxiliar desenvolvedores de software paralelo em duas diferentes frentes. Inicialmente foi apresentado um estudo a respeito do consumo de energia gerado pelo uso de STMs, fornecendo parâmetros para que programadores possam escolher adequadamente a estrutura de sincronização mais eficiente para seus requisitos. Durante este estudo, observou-se uma crítica limitação em relação ao desempenho das plataformas de simulação, o que motivou o desenvolvimento de uma plataforma mais veloz que utiliza abstração híbrida.

No Capítulo 3 apresentou-se um estudo caracterizando o consumo de energia introduzido pelo uso de uma STM nas aplicações do *benchmark* STAMP. O consumo gerado por esta versão das aplicações foi comparado com o consumo gerado pelas mesmas aplicações em uma versão baseada em *locks*, o que possibilitou a visualização da sobrecarga decorrente do uso da STM.

De maneira geral, a STM testada apresentou um desempenho inferior aos *locks* utilizados, atingindo uma média 3 vezes pior. Conforme demonstrado, os piores desempenhos foram observados nos casos cuja execução apresentou alta contenção, como nas aplicações *intruder* e *yada*. Observaram-se nestes casos altos valores de consumo de energia gastos em operações de *Rollback* e *Backoff* em função da alta contenção, o que justifica o aumento dos valores totais de consumo para estes casos.

Uma avaliação a respeito do mecanismo de *lock* foi apresentada, demonstrando que, apesar de normalmente serem mais eficientes que STMs, estas estruturas são diretamente afetadas por sua implementação e pelo sistema em que são executadas. Conforme apresentado, penalidades geradas pelo insucesso na aquisição de um *lock* relacionam-se com a execução de uma aplicação e, quando muito altas, influenciam negativamente a eficiência e o consumo de energia.

No Capítulo 4 descreveu-se a implementação de um novo recurso na plataforma MPARM, que possibilitou a execução de simulações com abstração híbrida. Diferentes técnicas uti-

lizadas para diminuir o impacto decorrente do uso de uma abstração mais simples foram descritas. Conforme demonstrado, a mudança na abstração dos modelos do processador aumentou o desempenho da plataforma em até 2.9 vezes, apesar de ter introduzido imprecisão aos resultados.

Os resultados obtidos com a simulação híbrida puderam ser estatisticamente aproximados com o uso de técnicas de regressão linear, o que diminuiu a proporção dos erros introduzidos. Observou-se que a similaridade das simulações executadas na plataforma híbrida em relação a plataforma original varia de acordo com a quantidade de núcleos instanciados e, por isso, o uso de um modelo estatístico específico para cada quantidade de núcleos mostrou-se mais eficiente. A execução de um conjunto de testes e subsequente aplicação do modelo específico sobre os resultados apresentaram um erro médio de 5.85%, demonstrando que a metodologia é confiável e se mostra como uma interessante alternativa para ambientes de desenvolvimento que exijam a execução de simulações mais rápidas.

Por fim, pode-se observar que a integração de processadores modelados com a linguagem ArchC na plataforma MPARM é uma tarefa possível, o que possibilita a integração de modelos referentes a novas arquiteturas nesta plataforma de simulação.

5.1 Trabalhos Futuros

A partir dos estudos apresentados neste trabalho, é possível imaginar uma grande quantidade de possíveis trabalhos futuros. Estes trabalhos encontram-se descritos nesta última seção.

Uma análise mais ampla, envolvendo diferentes implementações de STM, pode ser realizada na busca pelos parâmetros que fornecem maior eficiência energética para estes sistemas. Com base nestes parâmetros, um novo algoritmo STM pode ser desenvolvido para fornecer baixo consumo de energia e uma boa abstração de desenvolvimento paralelo.

A caracterização do consumo de energia em STMs pode ser aprimorada através de medições em ambientes reais, não simulados, com a ajuda de dispositivos de medição. Esta análise permitirá a análise do consumo em ambientes comuns, como máquinas *x86* executando um sistema operacional. Este procedimento, além de fornecer resultados puros, permitirá compreender melhor os efeitos de um sistema operacional nos resultados obtidos.

Modelos estatísticos mais apurados podem ser obtidos através do uso de um conjunto de treinamento maior no processo de regressão linear. Para tal, será necessário realizar um número maior de simulações com ambas as abstrações na plataforma MPARM. Além disso, modelos para configurações com um maior número de núcleos instanciados também podem ser desenvolvidos.

A validação da plataforma pode ser reforçada pela execução de outros *benchmarks*

paralelos, como o SPLASH-2 [44]. Os resultados obtidos neste processo também podem ser utilizados para compor os conjuntos de treinamento e apurar os modelos estatísticos.

Novos modelos de consumo que não se baseiam no estado do processador podem ser desenvolvidos e incluídos na plataforma. O desenvolvimento destes modelos pode revelar formas mais precisas do que as apresentadas para realizar a estimativa de consumo em simulações com precisão híbrida.

Por fim, outros processadores gerados com a linguagem ArchC podem ser integrados à plataforma MPARM, possibilitando a execução de simulações em diferentes arquiteturas. O suporte nativo à integração com o MPARM pode ser desenvolvido e incluído nas novas versões do ArchC, o que possibilitaria a conexão dos processadores gerados à plataforma sem a necessidade de modificações nos modelos ou no código do MPARM.

Referências Bibliográficas

[1] Bruno Carvalho Albertini. Um framework de desenvolvimento de plataformas e um mecanismo de depuração baseado em reflexão computacional. Master's thesis, Unicamp, 2007.

[2] ARM. Amba specification (rev 2.0), May 1999. `www.cl.cam.ac.uk/ mwd24/phd/swarm.html`.

[3] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[4] Rodolfo Azevedo, Bruno Albertini, and Sandro Rigo. ARP: Um gerenciador de pacotes para sistemas embarcados com processadores modelados em ArchC. In *Workshop de Sistemas Embarcados, SBRC*, may 2010.

[5] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *Int. J. Parallel Program.*, 33(5):453–484, 2005.

[6] A. Baldassin, F. Klein, P. Centoducatte, G. Araujo, and R. Azevedo. A first study on characterizing the energy consumption of software transactional memory. Technical report, Unicamp, 2009.

[7] Alexandro Baldassin. *Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia*. PhD thesis, Unicamp, 2009.

[8] Alexandro Baldassin, Felipe Klein, Guido Araujo, Rodolfo Azevedo, and Paulo Centoducatte. Characterizing the energy consumption of software transactional memory. *Computer Architecture Letters*, 8(2):56–59, Feb. 2009.

[9] Luca Benini, Davide Bertozzi, Alessandro Bogliolo, Francesco Menichelli, and Mauro Olivieri. MPARM: Exploring the multi-processor SoC design space with SystemC. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[10] David C. Black and Jack Donovan. *SystemC: from the ground up.* Springer, 2004.

[11] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20:26–44, 2000.

[12] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[13] Jianwei Chen, M. Dubois, and P. Stenstrom. Integrating complete-system and user-level performance/power simulators: the simwattch approach. In *ISPASS '03: Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 1–10, 2003.

[14] M. Dales. Swarm 0.44 documentation, February 2003. `www.cl.cam.ac.uk/ mwd24/phd/swarm.html`.

[15] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[16] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006. Electronic, no. page numbers.

[17] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[18] Cesare Ferri, Tali Moreshet, R. Iris Bahar, Luca Benini, and Maurice Herlihy. A hardware/software framework for supporting transactional memory in a MPSoC environment. *SIGARCH Comput. Archit. News*, 35(1):47–54, 2007.

[19] Cesare Ferri, Amber Viescas, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy efficient synchronization techniques for embedded architectures. In *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 435–440, New York, NY, USA, 2008. ACM.

[20] Free Software Foundation. Gnu general public license v.2, June 1991. `http://www.gnu.org/licenses/gpl-2.0.txt`.

[21] Frank Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Springer, 2006.

[22] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '98: 25 years of the International Symposia on Computer Architecture (selected papers)*, pages 255–262, New York, NY, USA, 1998. ACM.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[24] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[25] Open SystemC Initiative. Open systemc initiative. `http://www.systemc.org/home/`, July 2010.

[26] Stefanos Kaxiras. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 2008.

[27] F. Klein, A. Baldassin, G. Araujo, P. Centoducatte, and R. Azevedo. On the energy-efficiency of software transactional memory. In *SBCCI '09: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design*, pages 1–6, 2009.

[28] F. Klein, A. Baldassin, J. Moreira, P. Centoducatte, S. Rigo, and R. Azevedo. STM versus lock-based systems: An energy consumption perspective. *International Symposium on Low Power Electronics and Design*, 2010.

[29] Fernando Kronbauer, Alexandro Baldassin, Bruno Albertini, Paulo Centoducatte, Sandro Rigo, Guido Araujo, and Rodolfo Azevedo. A flexible platform framework for rapid transactional memory systems prototyping and evaluation. *Rapid System Prototyping, IEEE International Workshop on*, 0:123–129, 2007.

[30] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[31] Mirko Loghi, Massimo Poncino, and Luca Benini. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 410–406, 2004.

[32] David B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Language Design for Reliable Software*, pages 128–137, 1977.

[33] Josue Tzan Hsin Ma. Estimativa de consumo de energia em nível de instrução para processadores modelados em ArchC. Master's thesis, Unicamp, 2008.

[34] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, 2002.

[35] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy-aware microprocessor synchronization: Transactional memory vs. locks. In *4th Workshop on Memory Performance Issues (held in conjunction with the 12th International Symposium on High-Performance Computer Architecture)*, 2006.

[36] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[37] Sandro Rigo, Guido Araujo, M Bartholomeu, and Rodolfo Azevedo. ArchC: a SystemC-based architecture description language. *Computer Architecture and High Performance Computing*, pages 66–73, October 2004.

[38] A. Rose, S. Swan, J. Pierce, and J.M. Fernandez. Transaction level modeling in SystemC. Technical report, OSCI, 2005.

[39] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[40] Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay triangulation with transactions and barriers. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 107–113, 2007.

[41] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[42] ArchC Team. The archc website. `http://www.archc.org`, July 2010.

[43] Ian Watson, Chris Kirkham, and Mikel Luján. A study of a transactional parallel routing algorithm. In *In PACT ?07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[44] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the*

22nd annual international symposium on Computer architecture, pages 24–36, New York, NY, USA, 1995. ACM.

[45] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations, 1995.

[46] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178, New York, NY, USA, 2008. ACM.