Um Modelo de Memória Transacional para Arquiteturas Heterogêneas baseado em Software Cache.

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Felipe Portavales Goldstein e aprovada pela Banca Examinadora.

Campinas, 21 de Junho de 2010.

Rodolfo Jardim de Azevedo (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: Maria Fabiana Bezerra Müller – CRB8 / 6162

Goldstein, Felipe Portavales
G578m Um modelo de memória transacional para arquiteturas heterogêneas baseado em software cache/Felipe Portavales Goldstein-- Campinas, [S.P. : s.n.], 2010.

Orientador : Rodolfo Jardim de Azevedo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Memória transacional. 2.Memória cache. 3.Memória hierárquica (Computação). 4.Multi Core. 5.Arquitetura de computador. I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: A transactional memory model for heterogeneous architectures based in software cache

Palavras-chave em inglês (Keywords): 1.Transactional memories. 2.Cache memory. 3.Hierarchical memory (Computer science) 4.Multi Core. 5.Computer architecture.

Área de concentração: Arquitetura de Computadores

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP)
Prof. Dr. Philippe Olivier Alexandre Navaux (INF - UFRGS)
Prof. Dr. Sandro Rigo (IC-UNICAMP)

Data da defesa: 21/06/2010

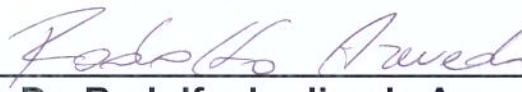
Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 21 de junho de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Philippe Olivier Alexandre Navaux
INF / UFRGS

Prof. Dr. Sandro Rigo
IC / UNICAMP



Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Um Modelo de Memória Transacional para Arquiteturas Heterogêneas baseado em Software Cache.

Felipe Portavales Goldstein

Junho de 2010

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
- Philippe Olivier Alexandre Navaux
  INF - UFRGS
- Sandro Rigo
  IC - Unicamp
- Sergio Takeo Kofuji (suplente)
  LSI - Escola Politécnica - USP
- Ricardo Pannain (suplente)
  IC - Unicamp

Resumo

A adoção de processadores com multiplos núcleos pela indústria, levou à necessidade de novas técnicas para facilitar a programacação de software paralelo. A técnica chamada memórias transacionais é uma das mais promissoras. Esta técnica é capaz de executar tarefas concorrentemente de forma otimista, o que permite um bom desempenho. Outra vantagem é que a sua utilização é muito mais simples comparada com a técnica clássica de exclusão mútua.

Neste trabalho é proposto o primeiro modelo de memória transacional para arquiteturas híbridas, neste caso a arquitetura alvo é o processador *Cell BE*. O processador *Cell BE* é especialmente complexo por causa das dificuldades que a arquitetura deste processador impõe ao programador quando se necessita acessar a memória global compartilhada. O modelo proposto age como uma camada entre o programa e a memória principal, permitindo um acesso transparente aos dados, garantindo coerência e realizando o controle de concorrência de forma automática.

O modelo proposto utiliza Software Cache combinado com a memória transacional para facilitar o acesso à memória externa a partir dos SPEs. Ele foi implementado e testado utilizando 8 aplicativos *benchmark* diferentes, mostrando sua viabilidade para casos de uso reais. Foi feita uma análise detalhada de cada parte da arquitetura proposta com relação ao impacto no desempenho geral do sistema. Este modelo foi capaz de obter um desempenho até duas vezes superior à implementação utilizando um *mutex* global. As vantagens da utilização se concentram principalmente na facilidade de uso, garantias de coerência e por evitar alguns tipos de *bugs* que seriam comuns em uma implementação com *mutex*, como por exemplo *dead-locks*. Este trabalho obteve o prêmio de melhor artigo no SBAC-PAD 2008.

Abstract

The adoption of multi-core processors by the industry has pushed towards the development of new techniques to simplify programming parallel software. The technique called transactional memories is one of the most promising. This technique is able to execute multiple tasks concurrently in an optimistic way to achieve a better performance. Another advantage is that the usage of this technique is simpler than the classic mutual exclusion.

This work proposes the first transactional memory model for hybrid architectures, in this case the target architecture is the *Cell BE* processor. The *Cell BE* is specially complex because of the difficulties when acessing the main shared memory from one of the SPEs. The proposed model acts as a layer between the program running and the main shared memory, allowing transparent access to the data, guaranteeing coherency and automatic concurrency control.

The proposed model uses a Software Cache combined with a transactional memory to facilitate the acess to the main memory from the SPEs. This model was implemented and tested using 8 benchmark applications, showing its feasability in real use cases. A detailed analysis of its internal parts has been made to show the impact of each part in the overal system performance. The model was able to achieve a performance up to two times better than a similar implementation using a global *mutex*. The advantages of this model rely on its usability, coherency guaranty and because it is able to avoid concurrency programming bugs such as dead-lock, which are common in a *mutex* implementation. This work won the best paper award at SBAC-PAD 2008.

Agradecimentos

Começo com um agradecimento especial ao meu orientador, Prof. Rodolfo Jardim Azevedo, que acreditou na minha capacidade de paralelismo[1] em realizar este mestrado concorrentemente com meu atual emprego. Durante minha graduação e pós-graduação eu fui inspirado pelo seu otimismo ao encarar desafios, sejam estes computacionais, da vida acadêmica ou pessoal.

Aos professores Guido Araújo, Sandro Rigo e Paulo Centoducatte, por estarem sempre dispostos à ajudar os alunos do laboratório.

Quero agradecer a todas as amizades que eu fiz nesse período. Aos meus amigos e companheiros de laboratório: Leonardo Piga, Felipe Klein, Bruno Cardoso, Bruno Albertini, Yang Yun Ju, Andre Costa, Samuel Goto, Valdiney Pimenta, George Leandro, Leonardo Garcia e Paulo Vital. Muitos dos quais são colaboradores de alguns trabalhos. Agradeço imensamente ao Alexandro Baldassin que contribuiu com importantes discussões sobre o trabalho realizado.

À minha amiga e companheira de vida, Agnes Leroy, por sua imensa paciência e incansável suporte ao me incentivar a escrever esta dissertação; por seu amor, amizade e é claro, a revisão do texto.

Ao meu grande amigo Lucas Porto e seus pais, por me receberem em sua casa de forma tão acolhedora.

Aos meus pais que me apoiaram em todas as decisões que tomei e que me ajudaram a chegar onde cheguei.

Por fim, quero aproveitar esta oportunidade para agradecer a alguns professores que marcaram a minha experiência acadêmica em todos estes anos na Unicamp. Esses professores, através de alguns pequenos-grandes atos, me mostraram como o ensino é importante. Por seu talento didático: Prof. Rodolfo Jardim Azevedo, Prof. Guido Araújo, Profa. Islene Garcia, Prof. Von Zuben, Prof. Alexandre Anan'in. É certo que existem muitos outros talentosos professores e que por motivos de espaço não é possível listar todos aqui. Espero que a pirâmide do conhecimento iniciada por cada um de vocês continue a crescer através de cada aluno que teve a feliz oportunidade de lhes conhecer.

[1] Apesar de algumas condições de corrida.

Sumário

Resumo **v**

Abstract **vi**

Agradecimentos **vii**

1 Introdução **1**

2 Conceitos Básicos **3**
2.1 Processos e *Threads* . . . 3
2.2 Paralelismo . . . 4
2.2.1 Paralelismo no Nível de Bits . . . 4
2.2.2 Paralelismo no Nível de Instruções . . . 4
2.2.3 Paralelismo no Nível de Tarefas . . . 5
2.2.4 Modelos de programação paralela . . . 7
2.2.5 Linguagens Funcionais . . . 7
2.2.6 *Message Passing Interface (MPI)* . . . 7
2.2.7 Stream Processing / Pipelining . . . 8
2.2.8 Memória Compartilhada . . . 8
2.3 Seção crítica e Exclusão mútua . . . 9
2.4 Memórias Transacionais . . . 11
2.4.1 Atomicidade e *Strong Isolation* . . . 12
2.4.2 Estratégias de Implementação . . . 13
2.4.3 Implementações em Hardware e em Software . . . 16
2.4.4 O processador *Cell BE* . . . 18
2.5 Software Cache . . . 21

3 Trabalhos Relacionados **24**
3.1 Software Cache . . . 24
3.2 OpenMP . . . 25
3.3 MPI . . . 25
3.4 Software Development Kit do *Cell BE* . . . 25
3.5 Memórias Transacionais . . . 27

3.6 Comparação 29

4 Software Cache Transacional **30**
4.1 Programando com Software Cache Transacional 31
4.1.1 Instrumentando o código automaticamente 36
4.2 Infraestrutura da Software Cache Transacional 39
4.2.1 Iniciando a Transação 41
4.2.2 Executando a Transação 42
4.2.3 Efetivando a Transação 44
4.2.4 Abortando a Transação 47

5 Resultados Experimentais **48**
5.1 Depuração e Otimização 48
5.2 *Benchmarks* 50
5.3 Análises 51
5.3.1 Tamanho da Cache 53
5.3.2 Tamanho do bloco de Cache 56
5.3.3 Tamanho do Diretório de Versionamento 58
5.3.4 Tamanho da Transação 60
5.3.5 Número de *Threads* 66
5.4 Isolamento Forte 70

6 Conclusão **73**
6.1 Contribuições 74
6.2 Trabalhos Futuros 74

Bibliografia **76**

Lista de Tabelas

2.1 Granularidade de paralelismo 6

3.1 Comparação dos trabalhos anteriores 29

4.1 Número de linhas de código para o aplicativo *Genome* 38

Lista de Figuras

2.1 Desempenho do ILP (Intel) [29] 5
2.2 Arquiteturas de Memória. 6
2.3 Duas condições de corrida. 12
2.4 Exemplo de Strong Isolation 13
2.5 Arquitetura do processador *Cell BE* 18
2.6 Compartilhamento Falso 23

4.1 Problema 30
4.2 Utilização de Software Cache e de Memória Transacional. 31
4.3 Solução 31
4.4 Código original 32
4.5 Código utilizando DMA no SPE 34
4.6 Código utilizando Software Cache no SPE 35
4.7 Código utilizando Software Cache Transacional no SPE 35
4.8 Código instrumentado usando Software Cache Transacional no SPE 37
4.9 Software Cache Transacional 39
4.10 Infraestrutura 40
4.11 Exemplo onde é necessário salvar a pilha. 41
4.12 Condição de corrida na validação da leitura. 46

5.1 Análise do tamanho da *cache*. 55
5.2 Análise do tamanho do bloco de *cache*. 57
5.3 Análise do tamanho do diretório de versionamento. 59
5.4 Análise do tamanho da transação e dos conjuntos de leitura e escrita. . . . 61
5.5 Análise do tamanho da transação e dos conjuntos de leitura e escrita. . . . 62
5.6 Análise do desempenho vs tamanho da transação. 63
5.7 Análise do desempenho vs tamanho do trabalho total realizado. 64
5.8 Análise do desempenho vs tamanho da transação. 65
5.9 Análise do número de *Threads*. 67
5.10 Sobrecarga devida à memória transacional. 68
5.11 Porcentagem de tempo em transações que abortam ou efetivam. 70
5.12 Impacto do Isolamento Forte no desempenho. 71
5.13 Análise do tamanho do bloco de *cache* (com Isolamento Forte). 72

Capítulo 1

Introdução

A indústria de fabricação de micro-processadores tem uma história de avanços tecnológicos que surgiram pela necessidade imposta pelo programador: aumentar o poder de processamento. O aumento do poder de processamento vinha crescendo exponencialmente a cada nova geração de processadores. Utilizou-se novos métodos de fabricação que diminuiam o tamanho dos transistores e também eram capazes de embutir mais transistores na mesma área de circuito [26]. O tamanho menor do transistor permitia um aumento na frequência de operação do circuito. O aumento da frequência implica em maior geração de calor por unidade de tempo, enquanto o aumento da densidade de transistores implica em maior quantidade de calor que precisa ser dissipada por unidade de área. O aumento do calor alcançou um ponto onde tornou-se inviável a sua dissipação. Este ponto foi chamado de *barreira térmica*. A indústria precisava achar uma solução para aumentar o poder de processamento sem aumentar a frequência de operação do circuito.

A solução encontrada foi o paralelismo: Embutir mais processadores num mesmo circuito integrado aumentando a área do circuito sem aumentar a frequência de operação. Até este ponto na história, o programador foi um observador passivo deste processo e usufruiu dos avanços tecnológicos com pouco esforço. O código legado era capaz de executar mais rápido simplesmente pelo fato de que o processador estava operando em uma frequência mais alta. Porém, agora, com o advento dos múltiplos processadores num só circuito, se o programador quiser tirar proveito do maior poder de processamento que o hardware permite, é necessário que o software sendo programado utilize os diversos processadores disponíveis. Do ponto de vista do programador, o paralelismo representou uma mudança radical no paradigma de programação.

É comum a opinião de que as pessoas não estão acostumadas a lidar com o paralelismo. É difícil uma pessoa ser capaz de, por exemplo, ouvir uma notícia no rádio e ler um livro sem que haja perda de informação vinda de qualquer um dos dois meios. As pessoas estão

acostumadas a dar atenção a uma coisa de cada vez. Isso está de acordo com a opinião de que o raciocínio do ser humano se dá de forma sequencial e esta opinião é compartilhada por muitos. Roger Penrose, em seu clássico livro *The Emperror's New Mind*, argumenta que o ser humano não é capaz de pensar em mais de um assunto ao mesmo tempo. Em um trecho do livro, aqui traduzido livremente, ele diz: "*Se alguém diz estar pensando em duas coisas ao mesmo tempo, isso é mais como uma constante troca de assunto a intervalos regulares, do que realmente pensando neles de forma simultânea, consciente e independente... ...é uma única consciência que pode ser vagamente ciosa de uma certa quantidade de coisas, mas que está concentrada em uma única coisa particular em cada momento.* " [30].

Supondo esta hipótese verdadeira, pode-se argumentar que o ser humano teria mais facilidade em criar um algoritmo ou um programa de computador utilizando lógica sequencial em comparacação com a lógica paralela. Um exemplo de programação paralela é a programação funcional. Este modelo de programação é, por muitos, considerado mais difícil ou complexo de se utilizar do que o modelo sequencial, o que está de acordo com a hipótese. Em outros modelos de programação paralela, a dificuldade principal está na divisão do acesso aos dados que estão sendo processados. Não importa se o motivo de tais dificuldades é a hipótese de que o ser humano está acostumado a raciocinar e lidar com as tarefas de forma sequencial. O fato é que estas dificuldades existem e que avanços para facilitar a programação paralela são bem vindos.

Esta dissertação tem como contribuição um modelo de programação paralela para processadores com múltiplos *cores* heterogêneos que facilita a programação concorrente neste tipo de arquitetura. A arquitetura alvo é o processador *Cell BE*. Para alcançar este objetivo, foi proposto um modelo de memória transacional baseada em Software Cache que tira proveito das características únicas deste tipo de arquitetura. Além disso, este modelo tem duas contribuições na área de memória transacional:

- Acredita-se que este seja o primeiro modelo de memória transacional para arquiteturas híbridas e para o *Cell BE*
- Este modelo propõe uma arquitetura inovadora com a união do conceito de Software Cache com o conceito de memórias transacionais.

No próximo capítulo serão introduzidos alguns conceitos básicos necessários para a compreensão dos capítulos subsequentes. Em seguida é apresentado o capítulo 3, enumerando alguns trabalhos relacionados e suas contribuições. No capítulo 4 é apresentado o modelo de programação paralela chamado Software Cache Transacional, que é o tema e principal contribuição desta dissertação. Nele, são apresentados os algoritmos e como a infraestrutura da arquitetura alvo se encaixa no modelo. O capítulo 5 apresenta os resultados experimentais e mostra como este modelo foi implementado e avaliado. Neste capítulo são utilizandos 8 aplicativos como *benchmarks* e é mostrado como cada aspecto da implementação do modelo afeta o desempenho de cada aplicativo. Por fim o capítulo 6 faz um breve resumo das contribuições e indica como pode ser dada continuidade à esta dissertação em trabalhos futuros.

Capítulo 2

Conceitos Básicos

Neste capítulo serão apresentados alguns conceitos básicos necessários para a compreensão do modelo de memória transacional proposto nesta dissertação, como por exemplo, processos e *threads*, seção crítica, exclusão mútua, Software Cache e memória transacional. O leitor ja familiarizado com estes conceitos pode passar para o próximo capítulo.

2.1 Processos e *Threads*

Em computação, um processo é uma instância de um programa de computador em execução. Um sistema operacional multi-tarefa é capaz de executar mais de um processo ao mesmo tempo. Neste sistema, os processos podem utilizar uma mesma CPU compartilhando o tempo de execução através de interrupções e trocas de contexto de execução, simulando a execução concorrente. Cada processo tem um pequeno período de tempo determinado, que ele pode executar enquanto os outros processos estão esperando em uma fila. Quando existe mais de uma CPU ou *core* disponível, os processos podem executar concorrentemente de forma real. Ao longo do texto, os termos aplicação, programa e processo serão utilizado de forma alternada significando o mesmo conceito.

Threads são blocos de um programa que podem executar de forma paralela e compartilham recursos entre si. *Threads* geralmente realizam um subconjunto de tarefas do processo e, portanto, serão usados os termos tarefa e *thread* de forma alternada no texto fazendo referência ao mesmo conceito. Processos possuem muitos recursos que são independentes. Por exemplo, comumente um processo possui uma área de endereçamento de memória isolada de outros processos enquanto *threads* possuem recursos herdados do processo e compartilham esses recursos, inclusive área de memória.

2.2 Paralelismo

Paralelismo é uma forma de computação onde múltiplos cálculos são realizados simultâneamente. Existem várias maneiras de realizar computação paralela, são elas: Paralelismo no nível de bits, paralelismo no nível de instruções e paralelismo no nível de tarefas.

2.2.1 Paralelismo no Nível de Bits

Nas décadas de 70 e 80, investiu-se no paralelismo no nível de bits. Onde a palavra, menor granularidade onde se realiza processamento, foi aumentada no número de bits. Este tipo de paralelismo aumenta a precisão dos cálculos e permite ganhos em processamento matemático. Atualmente os tamanhos de palavra mais comumente usado são 32 bits e 64 bits. Acima disso os ganhos não são significativos e as aplicações matemáticas que necessitam de uma precisão numérica maior do que 64 bits normalmente utilizam técnicas de aritmética de precisão arbitrária [13].

2.2.2 Paralelismo no Nível de Instruções

O paralelismo no nível de instrução (ILP, do inglês *Instruction Level Paralelism*) tornou-se muito popular na década de 90, principalmente com o lançamento do processador Pentium 4. Este tipo de paralelismo visa obter ganhos executando mais de uma instrução ao mesmo tempo. As técnicas que permitem este tipo de paralelismo são várias, dentre elas, podem ser citadas: *Pipeline*, *Out-of-order execution* e arquiteturas VLIW e superescalares.

Todas as técnicas de ILP aumentam a complexidade do hardware e mostram-se úteis para fazer uso dos transistores extra que a melhoria da tecnologia de fabricação de CIs disponibiliza. Uma vantagem dessas técnicas é que elas são implementadas de maneira a modificar o hardware para melhorar o desempenho computacional com pouca ou nenhuma modificação do software existente. O que facilita a utilização em conjunto com software legado ou programado de forma serial, sem a necessidade de embutir a noção de paralelismo no próprio software. Faz sentido, portanto, que estas técnicas tiveram tanta atenção e pesquisas para aperfeiçoamento.

Estas técnicas, contudo, possuem um limite de aproveitamento do paralelismo. Em um programa real, programado de forma serial, nem todas as instruções podem ser executadas em paralelo, as dependências de dados e de controle impõe um limite rígido ao qual as técnicas de ILP devem obedecer.

A Figura 2.1 mostra o aumento do desempenho dos processadores Intel devido ao aumento do ILP. Como se pode ver a curva do aumento de desempenho tem um pico quando alcança o desempenho relativo de 0.35 e depois volta a cair. Isso indica o limite rígido imposto pela dependência de dados e controle presentes nos programas seriais.

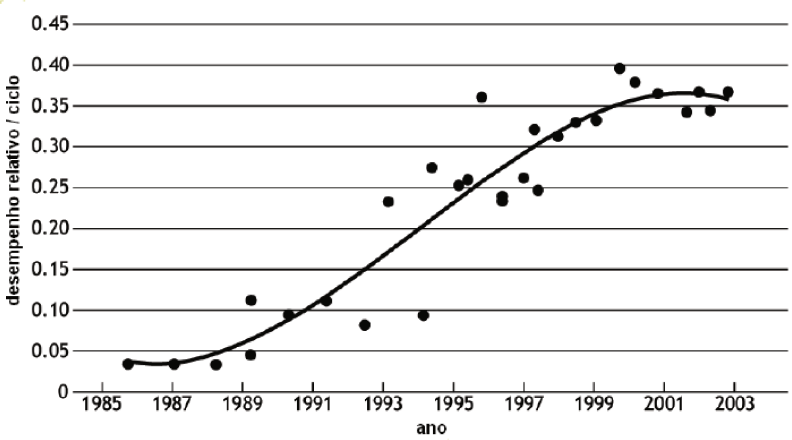


Figura 2.1: Desempenho do ILP (Intel) [29]

2.2.3 Paralelismo no Nível de Tarefas

O limite que o paralelismo no nível de instruções impõe só pode ser superado ao aumentar-se a granularidade. Pode-se chamar de tarefa um conjunto de instruções que realizam computação para gerar um resultado. Define-se, então, que uma aplicação é composta de uma ou mais tarefas. Dependendo do tamanho da computação realizada e do conjunto de instruções que esta sendo observado, pode-se definir uma tarefa em diferentes escalas. Por exemplo um navegador de internet ou um processador de texto são aplicações e podem ser considerados tarefas, assim como cada aba do navegador de internet ou cada arquivo de texto aberto no processador de textos também podem ser considerados tarefas independentes. Ao seguir nesta linha de raciocínio pode-se considerar como uma tarefa até a escala de poucas instruções que a formam, por exemplo, uma função num programa de computador.

Quando se tem pouca ou nenhuma dependência entre tarefas fica claro a vantagem de se usar este tipo de paralelismo e executar as tarefas simultaneamente. Este é o caso do navegador de internet e do processador de texto. Não é difícil de imaginar uma forma de implementar as duas tarefas sem nenhuma dependência. Porém quando se trata de funções dentro de um programa, as dependências de dados e de controle que podem existir entre funções tornam a implementação para parelismo mais complicada.

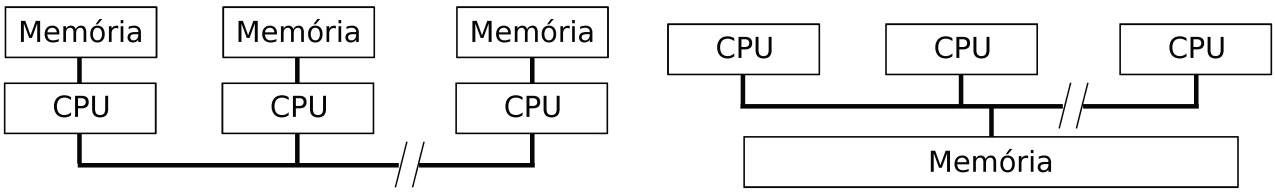
Nesta escala de paralelismo é necessário, portanto, que não apenas o hardware mas também o software esteja implementado de forma a permitir o paralelismo. O desenvolvimento de software que permite este tipo de paralelismo é o que daqui para frente será chamado de programação paralela ou paralelização de software. A paralelização do software pode ser feita de forma completamente manual pelo programador ou pode ser automática ou semi-automática através do compilador. A tabela 2.1 mostra os diferentes níveis de granularidade de paralelismo e o ator responsável pela paralelização [8].

No hardware, uma forma de implementar este tipo de paralelismo é ter mais de uma CPU no mesmo computador. Existem diferentes arquiteturas que implementam essa ca-

Granularidade	Escala no código	Paralelizado por
Muito Fina	Instrução	CPU
Fina	Bloco básico de código / *Loop*	Compilador
Média	Função	Programador
Grossa	Aplicação / Processo separado	Programador

Tabela 2.1: Granularidade de paralelismo

racterística. A principal diferença no desenho destas arquiteturas está no que diz respeito à forma com que cada CPU acessa a unidade de memória principal disponível, podendo esta ser distribuída ou compartilhada. No caso das arquiteturas que utilizam memória distribuída, cada CPU só tem acesso direto à memória local a ela designada, como mostrado na figura 2.2(a). Para uma CPU ter acesso às outras memórias existentes no sistema, é necessário ela se comunique com as outras CPUs através de um barramento ou uma rede de comunicação. No caso das arquiteturas que utilizam memória compartilhada, existe apenas uma unidade lógica de memória e todas as CPUs têm acesso direto a esta memória que é, portanto, compartilhada, como mostrado na Figura 2.2(b).



(a) Memória Distribuída (b) Memória Compartilhada

Figura 2.2: Arquiteturas de Memória.

A vantagem da memória distribuída é que geralmente o programa rodando em cada CPU não precisa se preocupar com controle de concorrência na memória, já que esta é exclusivamente acessada por apenas uma CPU, mas é claro que, depedendo da maneira que o hardware e o software são implementados, os acessos concorrentes podem se dar de uma forma indireta. A desvantagem é que o acesso à memória que está nas outras CPUs é mais lento e muitas vezes é necessário que haja um protocolo de comunicação controlado pelo software. Este tipo de arquitetura é utilizada, por exemplo, em computação em *clusters* onde cada CPU fica fisicamente separada em máquinas independentes e a comunicação se dá pela rede *ethernet*. Ela pode ser vista também em escalas menores, num mesmo circuito integrado, onde cada CPU tem uma pequena memória local que o software pode acessar diretamente. Isso é muito comum em *DSPs* que utilizam as chamadas memórias *scratch pad* e também nas *GPUs* atuais que possuem memória dedicada para cada unidade de processamento. Essas memórias são geralmente de alta velocidade (comumente

SRAM) e são muitas vezes utilizadas no lugar da *cache L1*, permitindo assim que o software controle o comportamento da *cache* o que dá origem ao conceito de *Software Cache* que será explicado na seção 2.5.

A vantagem da memória compartilhada é que cada CPU tem acesso direto à toda a memória incluindo o espaço que está sendo acessado pelas outras CPUs, o que pode permitir um maior desempenho pois não necessita de um protocolo de comunicação. Esta é também a desvantagem desta arquitetura, pois quando duas CPUs acessam a mesma área de memória, é necessário lidar com controle de concorrência para garantir a coerência da execução e dos dados. Este tipo de arquitetura ficou popular nas arquiteturas *multi-cores* para computadores *commodities*, onde se têm múltiplas CPUs idênticas dentro de um mesmo circuito integrado, por isso também chamada de arquitetura homogênea.

2.2.4 Modelos de programação paralela

Para satisfazer as características específicas de cada tipo de aplicação e tirar proveito da arquitetura utilizada, foram criados alguns modelos de programação paralela que tentam abstrair a arquitetura utilizada e definem o desenho básico do software que esta sendo desenvolvido. A seguir, serão descritos alguns desses modelos.

2.2.5 Linguagens Funcionais

Neste modelo o programa é implementado como uma avaliação de funções matemáticas que evita estados ou dados mutáveis. A função recebe valores de entrada, realiza cálculos e retorna uma saída. Esta estratégia explicita a dependência de dados entre funções e permite que a decisão de quais funções podem executar em paralelo seja tomada em tempo de compilação ou mesmo de execução. Essa maneira de programar é, muitas vezes, considerada mais complexa que a maneira sequencial e estruturada. As linguagens funcionais são pouco utilizadas comercialmente e algumas ficaram restritas ao meio acadêmico. Alguns exemplos de linguages funcionais são: *Erlang*, *lisp*, *ml* e etc.

2.2.6 *Message Passing Interface (MPI)*

Este é um modelo desenhado para aquitetura de memória distribuída. Neste modelo, define-se um nodo de trabalho sendo uma CPU com sua memória privada. O programador é responsável por paralelizar o software explicitamente dividindo as tarefas em nodos e realizando a comunicação entre nodos conforme o necessário. Cada nodo realiza um trabalho independente do outro e a comunicação entre nodos se dá através da troca de mensagens em uma rede de comunicação. As trocas de mensagens geralmente tem um

custo muito mais alto do que um acesso à memória, pois os nodos podem estar fisicamente distantes entre si e o próprio protocolo de troca de mensagens acarreta um custo.

Este modelo é muito utilizado em computação em larga escala, onde se têm muitas máquinas cada uma com sua CPU e memória privada, conectadas por uma rede *ethernet* onde são trocadas mensagens para comunicação.

Outros modelos semelhantes podem ser implementados utilizando o MPI como, por exemplo, as chamadas de procedimento remoto. Neste modelo uma função do software é executada por uma outra CPU como um serviço de forma assíncrona. Os parâmetros da função e os valores de retorno são transmitidos por mensagens.

2.2.7 Stream Processing / Pipelining

Este modelo de programação tenta extrair o paralelismo empregando a técnica de *pipelining* na divisão das tarefas da aplicação. Algumas aplicações podem ser vistas como uma sequência de pequenas tarefas que são executadas em cadeia e cada pequena tarefa tem acesso somente ao fluxo de dados sequencialmente. Cada tarefa executa um trabalho em cima de uma pequena parte dos dados e passa estes dados para a tarefa seguinte. Esta propriedade permite que os dados sejam processados de forma concorrente. As aplicações multimídia e de processamento de sinais são um exemplo típico onde esta técinca pode ser empregada. Porém nem toda aplicação pode ser divida desta maneira. Um exemplo é o caso onde se necessita acessar dados globais ou em uma ordem previamente não definida.

2.2.8 Memória Compartilhada

Neste modelo todos os processadores têm acesso direto à uma memória que é compartilhada por todos. Cada tarefa está executando em paralelo com as outras que podem acessar concorrentemente os dados. Este modelo de programação é um dos mais empregados pois é um modelo flexível e que melhor se adapta a diferentes tipos de aplicação. Contudo, acessar os dados concorrentemente traz uma série de problemas que precisam ser resolvidos. Como será discutido, alguns desses problemas são sincronização e condição de corrida.

Todos os modelos apresentados aqui tem vantagens e desvantagens em sua utilização. Porém, todos colocam nas mãos do programador a decisão da divisão das tarefas e principalmente a divisão do acesso aos dados. Este último ponto é um dos principais tópicos desta dissertação. O objetivo é tirar das mãos do programador a divisão do acesso aos dados. Permitir o acesso concorrente com garantias de corretude.

2.3 Seção crítica e Exclusão mútua

A seção crítica em um programa de computador é um bloco de instruções que acessa um recurso compartilhado e necessita garantir que a execução completa daquela região será feita sem a intervenção de outro processo ou tarefa no mesmo recurso compartilhado. Comumente o recurso compartilhado é a memória principal e deseja-se garantir que nunca mais de uma tarefa estará acessando aquela região de memória ao mesmo tempo durante a execução daquele bloco de instruções. O bloco de instruções pode ter o tamanho de uma instrução apenas ou pode ser até mesmo uma função ou uma tarefa inteira.

Para fazer uma analogia, pode-se citar uma impressora que é capaz de imprimir apenas uma página por vez, mas é compartilhada por mais de uma pessoa e pode receber mais de um documento para impressão ao mesmo tempo. Neste caso, a seção crítica será a sequência de instruções que realiza a impressão de um documento inteiro e isso é o mesmo que garantir que apenas um documento esteja acessando a impressora em um dado momento que leva ao segundo conceito: Exclusão Mútua.

O método tradicional para se fazer sincronização e controle de concorrência entre tarefas em programação paralela é utilizando-se um mecanismo de exclusão mútua. O conceito de exclusão mútua, como o próprio nome já diz, trata-se de impedir que mais de um processo ou tarefa atue sobre um recurso compartilhado num dado período de tempo ou bloco de instruções, ou seja, durante a seção crítica.

Na literatura, a técnica de exclusão mútua é chamada de *mutex* do inglês *mutual exclusion* ou mesmo *lock* que significa trava, esta nomenclatura também será usada neste texto.

A implementação do *mutex* é normalmente feita utilizando instruções especiais do processador para ler e alterar o valor de uma variável compartilhada em memória de forma atômica, ou seja, garantidamente sem a intervenção de outra *thread*. Cada arquitetura implementa uma instrução similar e as diferenças entre as arquiteturas e a forma como esta instrução é implementada não serão discutidas aqui.

Quando uma *thread* entra na seção crítica, ela adquire o *mutex*, executa a seção crítica e libera o *mutex*. Caso o *mutex* já tenha sido adquirido por outra *thread*, a *thread* atual espera até que seja liberado. Para isso, o programador deve, explicitamente, indicar onde está a seção crítica do seu programa colocando uma chamada para adquirir o mutex e uma para liberar.

Embora seja uma idéia simples, a utilização da exclusão mútua é bastante complexa pois deixa nas mãos do programador toda a responsabilidade no tratamento da sincronização e controle de concorrência e não permite composição [39].

A utilização incorreta do *mutex* pode gerar erros (*bugs*) difíceis de encontrar e reproduzir, dentre eles pode-se citar:

- *Race Conditions*: Se o programador esquecer de adquirir o *mutex* quando entra em uma seção crítica, não se poderá garantir a exclusão mútua daquela seção.

- *Dead-lock*: Se o programador não liberar o *mutex* na saída de uma seção crítica,

não será mais possível nenhuma *thread* entrar naquela seção novamente. Cenários mais complexos podem ocorrer quando se têm mais de uma variável compartilhada entre as *threads*. Uma *thred A* que adquiriu primeiramente o *mutex A* fica esperando o *mutex B* que foi préviamente adquirido pela *thread B* enquanto a própria *thread B* também está esperando no *mutex A* que *thread A* já havia adquirido. As duas *threads* nunca vão sair deste cenário, é como trancar uma gaveta com a chave dentro, daí vem a expressão *dead-lock*.

- *Live-lock, Starvation*: Dependendo do balanceamento de carga e da granularidade das regiões críticas, pode acontecer *live-locks* levando uma *thread* a nunca executar o trabalho designado, uma situação que também é difícil de detectar e corrigir.

Desempenho

A granularidade das regiões críticas em um programa e a utilização de *mutex* nessas regiões influencia diretamente no desempenho do programa. Por exemplo, quando se tem duas regiões críticas que acessam apenas áreas de memória independentes, essas duas regiões críticas podem ser executadas concorrentemente com segurança pois não estão acessando exatamente o mesmo recurso compartilhado. Obviamente que uma dada seção crítica não pode ser executada concorrentemente consigo mesma (em duas *threads* por exemplo) e portanto necessita-se de um *mutex* para cada uma. Neste caso, utilizar um *mutex* independente para cada seção crítica é utilizá-lo na menor granularidade possível e isso é a maneira mais eficiente de se ganhar desempenho. Se fosse utilizado um único *mutex* global para as duas seções críticas o programa continuaria correto, porém o desempenho seria menor devido à serialização desnecessária que aconteceria.

A utilização perfeita de *mutex* seria diminuir as seções críticas até o menor tamanho possível ainda garantindo a coerência da execução e o acesso aos recursos compartilhados e utilizar um *mutex* para cada intersecção no conjunto de endereços acessados por cada seção crítica. O problema é que o conjunto acessado pode depender da própria execução do programa e identificar os limites das seções críticas é uma tarefa complexa. Para o programador, fazer isso manualmente garantindo a inexistencia dos *bugs* acima citados, é extremamente difícil. Além disso, existe a tentativa de realizar essa tarefa de forma automática, mas ainda não se encontrou uma solução definitiva [31].

Portanto, quando se utiliza *mutex*, o programador se vê na necessidade de ter de escolher 2 entre 3 características: Corretude, Simplicidade e Eficiência.

Os cuidados que programador precisa ter e a complexidade em se detectar os erros que possam existir em uma implementação utilizando *mutex*, faz com que o software seja mais suscetível a erros, tornando este método pouco viável para se desenvolver programas eficientes e de fácil manutenção.

2.4 Memórias Transacionais

Uma alternativa promissora para resolver o problema de sincronização e controle de concorrência é conhecida como Memória Transacional (TM, do inglês *Transactional Memory*).

Similarmente ao *mutex*, neste modelo de programação, o programador especifica partes do código que são as seções críticas e devem ser executadas de maneira atômica, estas partes são as chamadas transações. O sistema de TM é responsável por fazer a abstração dos acessos à área de memória compartilhada e executar cada transação de maneira atômica e de uma forma isolada [14].

Diferentemente do *mutex*, o sistema de TM tenta garantir um bom desempenho através de uma estratégia otimista e não bloqueante na manipulação dos dados compartilhados e também, ao utilizar uma granularidade mais fina, é possível obter uma menor contenção no acesso aos dados compartilhados. Essas duas características são cruciais para evitar *bugs* pela utilização errada, como os *dead-locks* existentes no *mutex*. Além disso, também permitem que o programador especifique seções críticas maiores possivelmente com pouca ou nenhuma perda de desempenho, pois o sistema de TM cuidará de manipular os dados da forma mais granular possível. Essa é a grande vantagem desse sistema: a sua utilização é muito mais fácil. O objetivo do sistema de TM é facilitar o desenvolvimento do software abstraindo a complexidade inerente à programação paralela obtendo Corretude, Simplicidade e Eficiência.

O conceito de transação se baseia no mesmo conceito encontrado nos sistemas de banco de dados. Uma transação é uma operação ou conjunto de operações realizadas de forma atômica em uma base de dados, ou em qualquer outro sistema computacional, que deve ser executada completamente em caso de sucesso, ou ser abortada completamente em caso de erro. No caso das memórias transacionais, esta base de dados é a própria memória RAM.

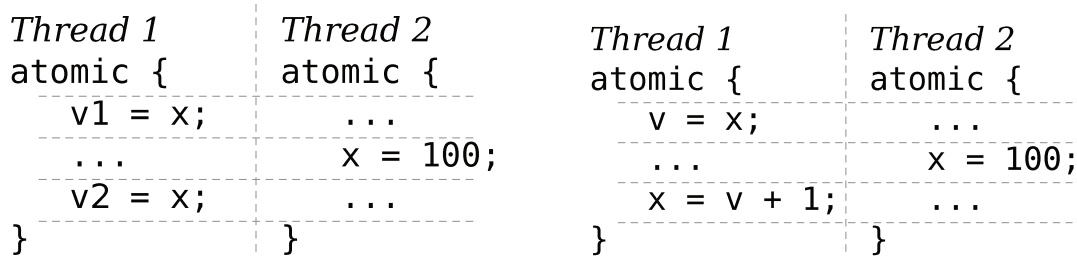
Uma transação pode também ser vista como um conjunto de instruções executadas seqüencialmente por uma *thread* satisfazendo as seguintes propriedades [33]:

- Atomicidade (tudo ou nada) - ou todas as instruções são efetivadas, ou então nenhuma o é. No primeiro caso, diz-se que a transação sofreu efetivação (*commit*) e o resultado da computação é permanente. No segundo caso, diz-se que a transação sofreu cancelamento (*abort*) e qualquer valor parcialmente computado é invalidado;

- Seriabilidade (isolamento) - transações concorrentes não vêem resultados intermediários produzidos por outras transações. Ou seja, o resultado da execução de diversas transações concorrentes equivale ao resultado da execução dessas mesmas transações em alguma ordem serial.

Um exemplo que ilustra a necessidade da atomicidade e seriabilidade é mostrado na Figura 2.3(a) e um na 2.3(b).

O código mostrado na Figura 2.3(a) ilustra uma situação onde uma *thread* lê o valor de uma variável `x` em dois momentos distintos e ela espera que este valor continue sendo



(a) Leituras não consistentes (read-after-write) (b) Escritas perdidas (write-after-write)

Figura 2.3: Duas condições de corrida.

o mesmo. Concorrentemente, uma outra *thread* altera o valor de `x` levando a um conflito do tipo *read-after-write* no qual as duas leituras não estão consistentes. Nesta situação, se as duas *threads* estiverem dentro de uma seção crítica marcada como transacional, o valor de `x` lido pela primeira *thread* permanecerá o mesmo, o que equivale a executar as duas *threads* serialmente.

A Figura 2.3(b) ilustra uma outra situação onde duas *threads* querem alterar o valor de `x`. A *thread 1* supõe que o valor de `x` lido e copiado para a variável local `v` permanecerá o mesmo quando ela alterar o valor de `x`. A *thread 2* supõe que o valor escrito por ela em `x` permanecerá o mesmo até o fim da execução da seção crítica, caso isso não seja garantido, pode acontecer um conflito do tipo *write-after-write* no qual uma escrita na variável é perdida. O isolamento provido pelo sistema de memória transacional vai garantir que esses casos sejam executados de forma atômica como esperado.

2.4.1 Atomicidade e *Strong Isolation*

As duas situações ilustradas supõem que o programador marcou todo o código que altera a memória compartilhada como sendo atômico ou transacional, definindo assim as regiões críticas. Porém, o que aconteceria, por exemplo, se o código da *thread 2* na Figura 2.3(b) não estivesse inserido em um bloco transacional? Nesta situação, o programador está informando ao sistema que o bloco da *thread 1* necessita ser executado de forma atômica. Porém o código da *thread 2* não precisa ser atômico e a única instrução que altera uma variável compartilhada (`x`) não tem uma relação de atomicidade com o resto do código da *thread 2*. Então, esta instrução que altera a variável `x` pode afetar a execução da *thread 1*. Nesta situação, o sistema de TM pode ou não garantir que a atomicidade da *thread 1* continuará válida. Dependendo do modelo da TM e da sua implementação, esta garantia pode ou não ser satisfeita [21].

Quando um código não transacional interfere em um código transacional e não existe uma garantia de atomicidade nesta ação, diz-se que o sistema de TM têm apenas garantias de Isolamento Fraco, ou *Weak Isolation*. Quando esta garantia é satisfeita mesmo que

uma das *threads* não esteja executando dentro de um bloco transacional, diz-se que o sistema de TM possui garantias de Isolamento Forte, ou *Strong Isolation* [23, 40].

Resumidamente pode-se dizer:

- **Weak Isolation**: Garante atomicidade somente entre blocos transacionais.
- **Strong Isolation**: Garante atomicidade entre blocos transacionais e não transacionais.

O Isolamento Forte é equivalente a realizar uma transação para cada instrução fora de uma região crítica (bloco transacional). Um exemplo onde Strong Isolation é necessário está ilustrado na Figura 2.4. Neste exemplo, a *thread 2* executa um código não transacional. Neste código não há a necessidade de que várias instruções sejam executadas de forma atômica e, portanto, não há região crítica. Porém, ele altera a variável compartilhada `x` que a *thread 1* utiliza. O código da *thread 1* possui 2 regiões críticas que necessitam ser executados cada uma de forma atômica. Caso haja garantias de Strong Isolation, as duas regiões críticas do código da *thread 1* serão executadas como se fossem uma única instrução na visão da *thread 2*. Dessa forma, cada região crítica será executada como se fosse imediatamente depois de uma instrução da *thread 2* e imediatamente antes de uma outra instrução da *thread 2*, assim como está apontado pelas duas setas cinzas ilustradas na Figura 2.4, o que equivale a uma possível execução sequencial.

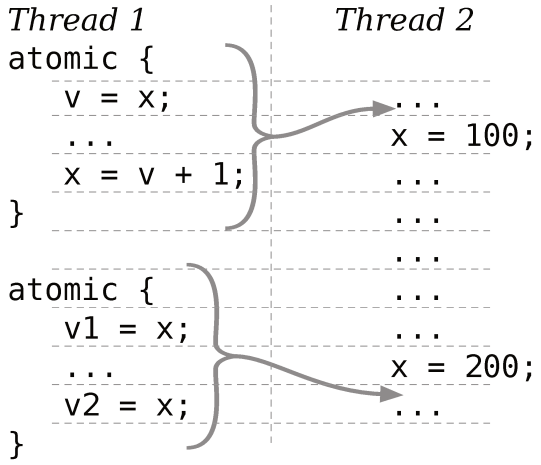


Figura 2.4: Exemplo de Strong Isolation

2.4.2 Estratégias de Implementação

Para se conseguir garantir atomicidade, seriabilidade, eficiência e facilidade para o programador, diversos modelos de TM foram criados e diferem na maneira de implementação. Estes modelos se baseiam em alguns dos conceitos que se seguem [7].

Armazenando o estado

- **Undo-Log:** Utilizando este conceito, o sistema de TM, ao executar um bloco de código transacional, faz todas as alterações necessárias atuando diretamente nos dados finais, ou seja, na memória RAM principal, de uma maneira otimista. Desta maneira, se a transação é efetivada, nenhum trabalho a mais precisa ser feito, pois todas as alterações que eram para serem realizadas na memória principal já o foram. Porém, se uma transação precisa ser abortada, todas as alterações terão que ser desfeitas. Para isso é necessário um estado das ações executadas na memória, para que cada ação possa ser desfeita posteriormente. Este estado é o chamado *Undo-log*. O interessante deste sistema é que utilizando essa estratégia mais otimista, o custo de manter o estado é distribuído durante toda a execução da transação. Efetivar a transação é uma tarefa extremamente rápida pois não necessita alterar a memória principal. O custo maior é quando se necessita abortar a transação e desfazer todas ações feitas na memória [32].

- **Commit-Log:** Ao executar um bloco de código transacional utilizando este conceito, o sistema de TM vai atuar em um espaço de memória especial separado, chamado de *read-set* (ou conjunto de leituras) e *write-set* (ou cojunto de escrita). O conjunto de leituras é o conjunto que armazena referências a todos os dados acessados (lidos) e o conjunto de escritas é o conjunto que armazena o conteúdo e a referência aos dados escritos pela transação corrente. Ao gravar em um espaço de memória separado, a transação evita que haja a necessidade de desfazer as ações quando uma transação precisa abortar. Porém quando a transação é efetivada, todos os valores gravados no *write-set* devem ser copiados para os seus respectivos lugares na memória principal. Este método posterga o custo da transação até a fase de *commit*. Isto parece uma estratégia menos otimista. No entanto, este método é muito mais fácil de implementar que o método do *Undo-Log* e, na prática, ele se mostra bastante eficiente [32].

Efetivando a transação

Quando o sistema de TM precisa alterar um dado na memória (seja durante a fase de *commit* no método *Commit-Log* seja durante a execução da transação no método *Undo-Log*), pode-se escolher entre uma abordagem que utiliza um *mutex* ou um algoritmo sem *mutex*.

- **Lock-Based:** Nesta abordagem, um *mutex* é utilizado ao se alterar um dado na memória. Existe um *mutex* designado para cada região de memória na granularidade em que o sistema de TM trabalha (geralmente uma palavra ou um objeto). Esta abordagem corre o risco de gerar contenção devido ao *mutex*, porém sabe-se que na prática isto não acontece freqüentemente e ela funciona muito bem. A grande

vantagem é que a utilização do *mutex* facilita muito a implementação do sistema de TM [6, 27].

- **Lock-Free:** Esta abordagem utiliza um algoritmo para alterar os dados na memória que garante não haver condições de corrida mesmo sem utilizar *mutex*. O grande problema é que esta abordagem tem uma implementação complicada e, na prática, o custo acrescentado pelo algoritmo não compensa a eliminação do *mutex*.

Abortando a transação

Quando um conflito é detectado, a transação é abortada e o bloco transacional é re-executado. A detecção do conflito pode se dar em duas formas:

- **Adiantado:** A detecção de conflito é feita em cada leitura da área de memória compartilhada durante a execução do bloco transacional. Caso detectado, a transação é imediatamente abortada. Isso adiciona um custo a cada leitura da memória compartilhada e, portanto é uma estratégia pessimista. Porém, ela garante que o bloco transacional nunca irá executar num estado de memória inconsistente. Isso é equivalente a dizer que o conjunto de leituras da transação sempre será um instantâneo coerente da memória [32].
- **Atrasado:** A detecção de conflito é postergada até o último momento durante a tentativa de efetivação da transação. Esta estratégia é otimista e visa diminuir os custos de leitura da área compartilhada para o caso em que o conflito não existe. Porém, ao deixar para detecar o conflito no fim da transação, a execução do bloco transacional pode entrar em um estado inconsistente da memória quando outra *thread* altera a memória compartilhada pela *thread* atual. Este estado inconsistente não será detectado até se alcançar o fim da transação. Caso a memória compartilhada implique em decisões de controle do programa, a *thread* pode nunca alcançar o fim da transação. Um exemplo claro é quando a decisão de iteração do laço é uma variável compartilhada, o que pode acarretar num laço infinito ou mesmo em um erro de segmentação ao tentar executar uma área ilegal. Os modelos de TM que utilizam esta estratégia utilizam rotinas auxiliares para capturar esses eventos ilegais e contornar os problemas, porém acredita-se que o não-determinismo acrescentado não vale a pena [32].

Granularidade

- **Baseado em palavras:** Este tipo de TM atua na granularidade de uma palavra de memória. Para isso, o *Undo-Log* guarda todas as palavras alteradas no seu estado armazenado para desfazer depois, enquanto o *Commit-Log* guarda cada palavra lida e escrita respectivamente no seu *read-set* e *write-set*. A vantagem deste tipo de

TM é que a pequena granularidade permite uma boa eficiência para evitar abortos desnecessários, pois os dados alterados são no menor nível de granularidade possível. Um problema enfrentado por este tipo de TM é que ao se utilizar o método *Lock-Based*, pode-se gastar até o dobro de memória RAM para guardar os *mutex* de cada palavra de memória. Porém, existem métodos para contornar este problema como o *bloom-filter* [25], que basicamente é uma estrutura de compressão com perda de dados. Esta estrutura de dados é capaz de armazenar a presença ou não de um elemento num conjunto. Caso a resposta seja não, com certeza o elemento não está, mas caso a resposta seja sim, o elemento pode ou não estar. A estrutura utiliza um *hash* para diminuir o tamanho de memória utilizado, podendo haver colisões. Por isso, é possível haver falso conflito.

- **Baseado em objetos:** Em uma linguagem orientada a objetos têm-se a opção de trabalhar na granularidade de um objeto ao invés da granularidade fina de uma palavra de memória. Neste cenário, o método *Lock-Based* utiliza um *mutex* para cada objeto na memória. O problema com este método é que *threads* diferentes podem alterar variáveis membro diferentes de um mesmo objeto mas, com a granularidade sendo o objeto, as *threads* terão que abortar mesmo sem a necessidade.

2.4.3 Implementações em Hardware e em Software

Utilizando os conceitos apresentados, surgiram propostas de modelos de TM para serem implementados em três diferentes maneiras:

- **Software Transacional Memory [35] (STM):** Neste modelo, todo o sistema de memória transacional é implementado em software na forma de uma biblioteca. O programador precisa usar as funções fornecidas pela biblioteca para fazer leituras e escritas de dados na memória ou mesmo o próprio compilador substitui as leituras e escritas pelas chamadas de função da TM de forma automática. As principais características dos modelos de STM propostos são: Eles lidam bem com quaisquer tamanho de transação. A quantidade dos dados lidos ou escritos na memória durante uma transação não é limitada. Eles geralmente têm baixo desempenho comparados com os outros dois modelos propostos (HTM e HybridTM). No STM, o custo para se garantir *Stong Isolation* em software é tão alto que torna o modelo inviável na prática.

- **Hardware Transacional Memory [4] (HTM):** Neste modelo, todo o sistema de memória transacional é implementado em hardware. Geralmente, o sistema é implementado como uma camada de abstração entre a *cache* e a memória RAM. Esta camada cuida de realizar as transações da forma correta. Neste caso, a *cache* é compartilhada entre as diferentes CPUs e o protocolo de coerência de *cache* é

aproveitado para realizar as ações que garantem o correto funcionamento do sistema de transação. Como este sistema é totalmente implementado em hardware, o estado armazenado do *Undo-Log* ou o *read and write-set* do *Commit-Log* precisam ser mantidos no hardware. Esses estados são geralmente armazenados como uma extensão da *cache*, utilizando área do circuito. Com isso, o tamanho dessa estrutura é bastante limitado fazendo com que o tamanho das transações seja também limitado. Essa limitação faz com que este sistema não seja escalável e pode tornar inviável dependendo do tipo de aplicação. Porém, HTM apresenta algumas vantagens interessantes, a mais importante delas é o bom desempenho, pois todas as operações transacionais são realizadas em hardware. Além disso, este sistema consegue garantir *Strong Isolation* sem um custo maior como é o caso do STM. O *Strong Isolation* é possível pois a camada de abstração entre a *cache* e a RAM, e o protocolo de coerência de *cache* permitem que as garantias de atomicidade sejam respeitadas, mesmo quando uma das *threads* sendo executadas não esteja dentro de um bloco transacional.

- **Hybrid Transactional Memory [19] (HybridTM):** Este modelo foi desenhado com o propósito de herdar as melhores características do HTM e do STM. Neste intuito ele tenta obter a bom desempenho do HTM, garantir *Strong Isolation* a baixo custo e também permitir um tamanho de transação não limitado como o STM. Para isso ele é implementado de forma parecida com o HTM mas, quando o estado armazenado da transação ultrapassa o limite que cabe na estrutura interna do hardware, este estado é passado para um local especial da memória RAM e, assim, como o STM faz, mantém o estado na RAM para garantir que uma transação de qualquer tamanho possa ser realizada. Além disso, este modelo utiliza o software para implementar algumas operações do sistema de TM que não impactem tanto o desempenho e que simplifique a implementação do hardware. Este modelo de memórias transacionais híbrido é o modelo que está se mostrando ser o mais promissor para uma utilização prática e comercial.

Além das características já elucidadas, vale mencionar que o sistema de STM tem ainda algumas vantagens sobre os outros dois:

- Não exige re-fabricação da CPU
- Fácil de implementar, testar e depurar, em comparação com as versões que utilizam hardware.
- Permite portabilidade

Estas características fazem do modelo STM um bom candidato para se realizar pesquisas com memórias transacionais pois permitem modificações e realizar experimentos de forma mais fácil.

2.4.4 O processador *Cell BE*

Apesar de o propósito inicial do *Cell BE* ser um processador voltado para alto desempenho de processamento matemático, jogos e aplicações multimídia [16], a sua arquitetura diferenciada e um tanto flexível permitiu o seu uso em diversos outros tipos de aplicações.

O processador Cell Broadband Engine é composto por um núcleo chamado PPE (*Power Processing Element*) e oito SPEs (*Synergistic Processing Elements*) conectados através do barramento chamado *Element Interconnect Bus* (EIB) [15].

Pode-se dizer que a arquitetura do *Cell BE* é heterogênea em dois sentidos. O primeiro, os núcleos de processamento (o PPE e o SPE) possuem conjunto de instruções diferentes. O segundo é a arquitetura de memória que mistura o conceito de memória compartilhada com memória distribuída. As memórias locais dos SPEs são uma forma de memória distribuída, porém os SPEs também podem acessar a memória principal compartilhada através de transferências DMAs. Essas trasnferências, como em outras arquiteturas, podem se dar em paralelo e, com isso, os problemas de concorrência continuam existindo.

A sua arquitetura está ilustrada na figura 2.5.

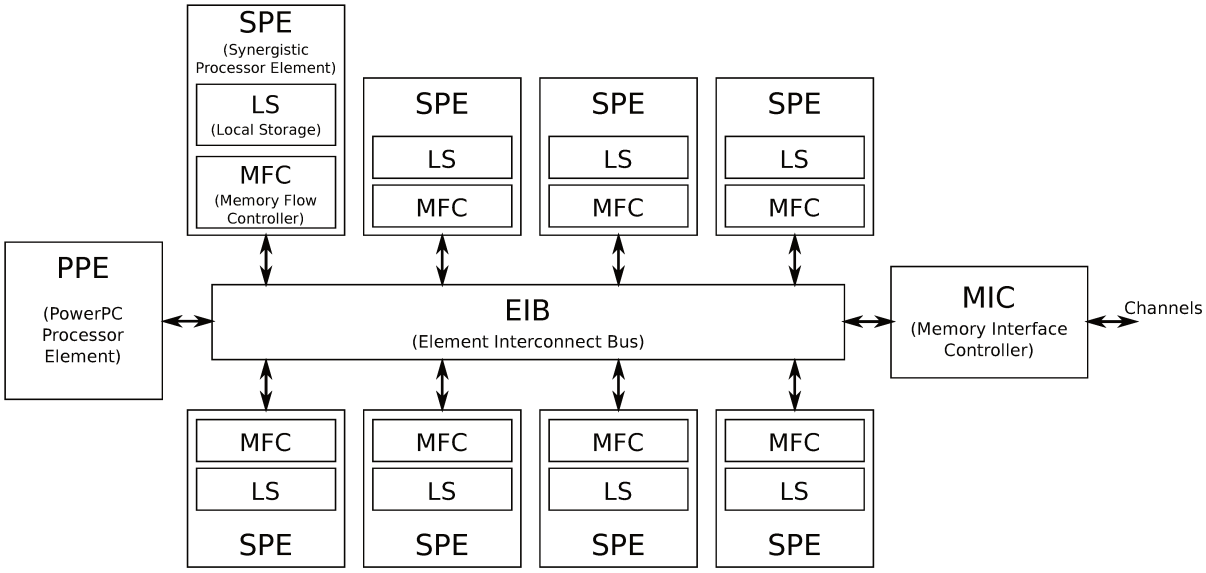


Figura 2.5: Arquitetura do processador *Cell BE*

- **PowerPC Processor Element (PPE)**: O PPE é o processador principal. Ele contém uma arquitetura *PowerPC* de 64-bits com acesso direto à hierarquia de memória RAM tradicional (*cache* L1 e L2, barramento até a memória RAM, etc). Este processador roda o sistema operacional e gerencia os recursos, por exemplo, fazendo troca de contexto das tarefas, alocação de threads nos SPEs, etc.
- **Synergistic Processor Elements (SPEs)**: Os SPEs são processadores *SIMD* otimizados para operações matemáticas e vetoriais. Eles são compostos por um

núcleo RISC com um conjunto de instruções (*instruction set*) específico e diferente do PPE, que não é baseado no PowerPC; e uma unidade de memória local (também chamada de LS ou *Local Store* de 256 Kbytes de rápido acesso (tão rápida quanto a *cache* L1) que é onde ficam armazenados os dados e o código executado no núcleo SPE. Os SPEs dependem de transferências DMA para mover dados da memória principal (externa) até a memória local e vice-versa. O *Memory Flow Controller (MFC)* é o mecanismo que controla essas transferências. As transferências DMA acessam a memória principal através do espaço de endereçamento efetivo do PPE. Como no PPE, a tradução de endereços é governada pela segmentação e tabela de páginas.

- **Element Interconnect Bus (EIB)**: O PPE e os SPEs comunicam-se coerentemente com cada um e a memória principal através do EIB. O EIB é uma estrutura em forma de 4 anéis cocêntricos, dois transportam dados no sentido horário e dois no anti-horário. Este anel funciona de uma forma similar a uma rede *ethernet* dentro do próprio *chip*. Os dados podem ser enviados de um SPE para o outro ou mesmo do PPE. Esta estrutura tem uma largura de banda para transferência de até 96 bytes por ciclo de *clock* e suporta até 100 comandos DMA enfileirados para acesso à memória principal e os SPEs.
- **Memory Interface Controller (MIC)**: É o controlador da memória principal (externa).

Do ponto de vista do desenvolvimento de software, a principal diferença entre o PPE e o SPE é a maneira como eles acessam a memória principal. O programa rodando no PPE pode acessar livremente o espaço de endereçamento completo da memória principal através de instruções de *LOAD* e *STORE*. Enquanto os SPEs acessam a memória principal apenas através de transferências assíncronas por *DMA*. Todas as instruções *LOAD* e *STORE* executadas dentro de um SPE acessam somente a sua memória local. Essa estrutura é uma mudança radical comparada com a arquitetura e os modelos de programação convencionais. Ao forçar o uso de DMA e utilizar essa hieraquia diferenciada, essa arquitetura explicitamente paraleliza a computação e as transferências de dados. Porém, agora, o programador é o responsável por realizar manualmente as transferências necessárias e com isso a programação nesta arquitetura se torna mais complicada.

Transferências DMA

As transferências DMA podem ser iniciadas pelo PPE ou pelo SPE e podem ser entre duas memórias locais ou entre a memória principal e a memória local do SPE. Em todos os casos as transferências podem se dar num sentido ou no outro. O mecanismo de controle das transferências DMA é chamado *Memory Flow Control (MFC)*.

Cada SPE tem seu próprio MFC que possui uma fila de requisições DMA. O software executando no SPE pode emitir uma requisição DMA que é não bloqueante e após

enfileirar a requisição no MFC, o controle volta para o software e a execução continua. Para cada requisição ou grupo de requisições DMA enfileirados, o software atribui um número de 5 bits de identificação, chamado de TAG. Esta TAG serve para identificar as requisições DMA. Existe um registrador no MFC, para cada TAG, indicando o status se todas as requisições daquela TAG foram completadas. O SPE possui uma instrução para leitura deste registrador de forma não bloqueante e uma instrução bloqueante que espera até que todas as requisições daquela TAG sejam completadas.

Todas as transferências DMA devem ser feitas entre endereços de origem e destino naturalmente alinhados, ou seja, o dois endereços devem ser divisíveis pelo tamanho da transferência e ter o mesmo deslocamento dentro de um bloco de 16 bytes (a operação módulo 16 nos dois endereços deve gerar o mesmo resultado). As transferências podem ser de 1, 2, 4, 8 bytes ou múltiplos de 16 bytes, com tamanho máximo de 16 KB. Por causa das características do EIB, as transferências DMA obtém o melhor desempenho quando tanto o endereço na memória local quanto na memória principal são alinhados em 128 bytes e o tamanho da transferência é múltiplo de 128 bytes.

Além do DMA, existe ainda um outro mecanismo que permite a comunicação entre os SPEs e o PPE. Os *mailboxes* são filas de troca de mensagens de 32 bits entre SPEs e PPE. Ao fazer uma requisição de escrita, a requisição é enfileirada na *mailbox* da CPU de destino (seja ela PPE ou SPE). A CPU de destino pode optar por ler as mensagens da *mailbox* de forma bloqueante ou através de interrupções. Esse tipo de comunicação é restrita (por carregar poucos dados, apenas 32 bits), mas serve principalmente como uma forma de sinalização de eventos ou por exemplo para implementar mecanismos de chamadas de procedimento remoto.

Mecanismos de sincronização

Como em outras arquiteturas, o *Cell BE* possui instruções para auxiliar a implementação de *mutexes*. No caso dos SPEs, são dois comandos DMA especiais, um para leitura da variável compartilhada e um para escrita (e suas variantes com condicionais). Os dois comandos são atômicos. O comando de leitura faz a reserva do bloco de *cache* e o comando de escrita retorna um valor dizendo se a escrita foi realizada com sucesso desde que o comando de leitura foi realizado (se o bloco de *cache* continua reservada para a SPE corrente). O programador precisa re-executar o bloco atômico caso a escrita não tenha sido realizada com sucesso. Como se ve, esta estratégia é semelhante aos algoritmos que não utilizam *mutex* e fazem espera ocupada na CPU para garantir que a operação atômica seja realizada com sucesso. Esta estratégia tende a ser muito custosa, pois quando o SPE faz espera ocupada verificando se houve sucesso na escrita na variável compartilhada, ele realiza transferências DMA e pode gerar contenção no barramento EIB fazendo com que outras trasnferências DMA de outros SPEs sejam atrasadas.

Uma biblioteca de *mutex* convencional pode ser implementada utilizando essas instruções, porém os problemas de contenção existentes no próprio modelo de *mutex* são

ainda mais sensíveis com a estratégia de espera ocupada utilizada no SPE. Isso torna bastante difícil o uso de *mutex* para programação paralela no SPE utilizando o conceito de memória compartilhada, o que explica o grande sucesso dos modelos de programação que utilizam memória distribuída enquanto que os modelos de memória compartilhada são pouco utilizados atualmente no *Cell BE*.

Uma outra diferença importante entre os processadores convencionais e o *Cell BE* é o fato de que somente um programa é executado por vez em um SPE, ou seja, não existe troca de contexto ou preempção de forma rápida em um núcleo SPE. A preempção é suportada, mas não da mesma forma que é feita em um processador comum (como a arquitetura x86 por exemplo), e é bastante custosa. Apesar disso parecer um pouco estranho à primeira vista, este tipo de limitação foi uma escolha de projeto para facilitar diversos aspectos da implementação da arquitetura.

Como se vê, programador no *Cell BE* precisa lidar com um nível de detalhamento muito grande no controle de acesso aos dados. A questão da facilidade de programação nesta arquitetura é um ponto muito importante que será abordado nesta dissertação. O projeto desta arquitetura deixa todo o trabalho no que diz respeito ao acesso a dados e paralelização, nas mãos do programador, o que pode parecer um passo atrás no sentido de facilitar o desenvolvimento de software paralelo, mas como será visto, esta arquitetura complexa pode ser explorada para, no fim, facilitar o desenvolvimento de software paralelo.

Essas dificuldades impostas pela arquitetura estão sendo aos poucos abstraídas pelo compilador e por bibliotecas que visam auxiliar o programador em cada cenário, porém, o estado da arte atual ainda está muito longe de chegar numa solução ideal. Algumas das soluções de programação para o *Cell BE* serão apresentadas na seção 2.4.4

2.5 Software Cache

Este conceito se baseia em implementar em software um mecanismo semelhante à *cache* existente em hardware. A motivação se dá principalmente pelo fato de que a memória local tem acesso muito mais rápido do que a memória externa e além disso, como foi explicado, o *Cell BE* impõe restrições que dificultam a implementação do software acessando diretamente a memória externa.

Para isso pode-se utilizar uma biblioteca que implementa os mecanismos da *cache* na memória local: um vetor com diretório de meta-dados e um vetor com as linhas de *cache*. Todas as referências à memória externa no software são então substituídas por chamadas na biblioteca de Software Cache que monitora o acesso à memória principal. Em geral, o tratamento das referências à memória externa utilizando Software Cache pode ser resumido nos seguintes passos:

1. Verificar se o bloco de *cache* referenciado está presente na memória local.

2. Caso não esteja presente, procurar um bloco de *cache* não utilizado.

3. Caso não encontre, procurar um bloco de *cache* que possa ser retirado da memória local.

4. Se o bloco de *cache* estiver modificado (também chamada de sujo), realizar a escrita do bloco na memória principal. Esta operação é chamada de *writeback*.

5. Realizar a transferência do bloco de *cache* desejado para a memória local. Esta operação é chamada de *fetch*.

6. Retornar o endereço local onde o bloco de *cache* foi gravado.

Substituir as referências à memória principal pelas chamadas da biblioteca de Software Cache é uma tarefa que pode ser facilmente implementada utilizando um compilador. O próprio GCC para o processador *Cell BE* tem essa capacidade [9, 15]. O programador pode utilizar o atributo `__ea` ao declarar ponteiros para fazer referências à memória principal a partir do software executando no SPE. O compilador automaticamente utiliza uma biblioteca padrão de Software Cache e substitui todas as referências a endereços da memória principal por chamadas à esta biblioteca.

A biblioteca que o GCC utiliza para isso, possui características muito semelhantes à que existe no SDK do *Cell BE* fornecida pela IBM, são elas:

- Associatividade restrita à `1-via` (mapeada diretamente) ou à `4-vias` (utiliza instruções SIMD para fazer comparação dos 4 endereços simultaneamente).

- Tamanho do bloco de *cache* é preferencialmente de 128 bytes para melhor utilização de DMA.

- A escrita na memória principal de blocos de *cache* sujos se dá de forma automática quando não há mais espaço na *cache*, porém a operação de descarga completa da *cache* para a memória principal deve ser feita pelo programador através de uma chamada de função. Esta operação é geralmente usada para atualização dos dados na memória principal quando necessita fazer sincronização entre os SPEs.

- O mais importante: Não existe coerência neste modelo de Software Cache. O bloco de *cache* alterado em um SPE não é atualizado nas *caches* dos outros SPEs que também estão lendo ou escrevendo neste bloco de *cache*.

Coerência em Software Cache

Por não garantir coerência de *cache*, este modelo pode trazer diversos problemas. Um dos principais problemas acontece quando duas referências em *threads* diferentes compartilham o mesmo bloco de *cache* com deslocamento diferente e não alteram a mesma posição. Este tipo de situação é descrita como compartilhamento falso. Isso acontece, por exemplo, quando dois inteiros de 4 bytes estão sendo acessados por duas *threads*. Um faz referência

no começo do bloco de *cache*, com deslocamento 0 e o outro no fim do bloco de *cache* com deslocamento 124, como ilustrado na figura 2.6. Para o programador, as duas variáveis são independentes e não compartilhadas e, portanto, deveria ser seguro alterar o valor de cada uma concorrentemente. Porém sem a garantia de coerência, tudo que acontece entre a leitura e a escrita do bloco de *cache* na memória principal não é visto pelos outros SPEs e, portanto, pode haver uma condição de corrida com escrita perdida (*write-after-write*).

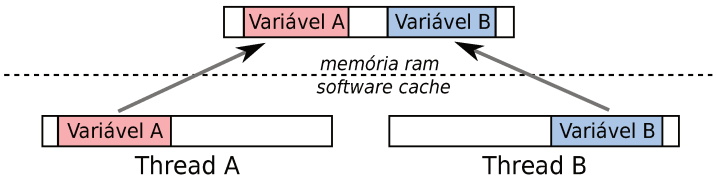


Figura 2.6: Compartilhamento Falso

Mesmo que o programador esteja ciente deste problema, ele precisa tomar o cuidado para que, em tempo de execução, todas as referências concorrentes à memória principal usando Software Cache sejam feitas utilizando blocos de *cache* independentes. Por exemplo, cada alocação de memória teria que ser alinhada em 128 bytes e com tamanho múltiplo de 128 bytes, o que pode desperdiçar espaço.

Os mecanismos tradicionais para garantir coerência na *cache* em hardware não podem ser aplicados em Software Cache, visto que seriam extremamente dispendiosos [37]. Existe, porém, uma solução pouco elegante que evita esse problema no caso das implementações de Software Cache para o *Cell BE*. Para cada bloco de *cache*, mantêm-se um vetor de 128 bits como uma máscara indicando quais dos 128 bytes da *cache* foram alterados. Quando é necessário realizar uma transferência da *cache* para a memória principal, somente os bytes alterados são copiados. Porém, para que a operação de máscara seja realizada não se pode fazer o DMA simplesmente para a memória principal. Deve-se, antes, copiar o novo bloco de *cache* que está atualmente na memória principal para um registrador temporário no SPE e então aplicar a máscara copiando os bytes alterados do bloco de *cache* local para o registrador temporário. Em seguida deve-se copiar de volta o bloco de *cache* para a memória principal. Porém, se algum outro SPE alterou novamente o bloco de *cache* entre o momento da leitura e da escrita, ocorrerá uma condição de corrida, então deve-se utilizar as instruções DMA especiais descritas na seção 2.4.4. Não existe maneira de o SPE saber se houve compartilhamento falso do bloco de *cache* antes dele realizar a cópia da mesma para o registrador temporário. Portanto, essa solução exige que todas as tranferências de escrita do bloco de *cache* para a memória principal sejam realizadas desta forma, gastando 2 vezes mais transferências DMA, sobrecarregando o barramento e aumentando a contenção.

Como se vê, este é um problema grave deste modelo de Software Cache e existe, portanto, a necessidade de um mecanismo eficiente para garantia de coerência. O trabalho apresentado nesta dissertação propõe e implementa um mecanismo que garante coerência de forma eficiente e elegante.

Capítulo 3

Trabalhos Relacionados

Como visto, a programação paralela no modelo de memória compartilhada impõe dificuldades ao programador no controle de concorrência e sincronização. Junto a isso, a arquitetura do *Cell BE* traz novas dificuldades no acesso à memória principal compartilhada e os mecanismos para sincronização existentes são ainda mais ineficientes para esse modelo de programação. Diversos trabalhos surgiram para facilitar a sincronização e controle de concorrência utilizando Memórias Transacionais, enquanto outros focaram no *Cell BE* para tentar facilitar a programação paralela nesta arquitetura. Como será mostrado, porém, nenhum dos trabalhos até a presente data foi capaz de trazer uma solução elegante para os dois problemas.

3.1 Software Cache

Em [2], é apresentada uma implementação de Software Cache que visa sobrepor computação com comunicação. A principal contribuição deste trabalho concentra-se no compilador. Após a análise das dependências de dados, o código é instrumentado para que no momento em que já é possível saber o endereço acessado, a requisição DMA é feita e o código continua executando até que o acesso ao bloco de *cache* seja feito. Neste momento é realizada uma chamada bloqueante para esperar até que o dado esteja disponível após a transferência DMA. O mesmo é feito para as transferências dos blocos de *cache* sujos para a memória principal. O compilador procura o ponto onde é feita a última referência a este bloco de *cache* e tenta fazer a transferência antes que seja necessário bloquear a execução do código. Este modelo de Software Cache, porém, é semelhante aos outros e não garante coerência. Ele, portanto, possui as mesmas deficiências já explicadas.

3.2 OpenMP

OpenMP [5] é uma plataforma de desenvolvimento de software paralelo utilizando o modelo de memória compartilhada. A plataforma visa facilitar o desenvolvimento provendo uma interface simples e abstraindo certos aspectos da divisão do código em múltiplas tarefas paralelas. O objetivo desta plataforma não é fazer o controle do acesso concorrente à memória compartilhada, mas sim facilitar a criação de *threads* diminuindo a quantidade de código necessário e os custos inerentes à inicialização e manipulação das *threads*, incluindo distribuição da carga de trabalho e sincronização.

Em [28], os autores mostram como o *OpenMP* pode ser implementado no *Cell BE* considerando as restrições da arquitetura e respeitando a especificação do *OpenMP*, que define o modelo de memória compartilhada com consistência relaxada: Cada *thread* tem sua própria visão temporária da memória. Um valor lido ou escrito em uma variável pode permanecer na visão temporária da *thread* até que o compartilhamento seja forçado por uma operação de descarga do *OpenMP*. Essa funcionalidade é implementada utilizando a memória local do SPE, mas a consistência relaxada não garante coerência entre duas *threads* e, portanto, o controle de concorrência fica a cargo do programador. Para isso, é apresentado também uma implementação de *mutex* e barreiras utilizando os comandos especiais de transferência DMA apresentados na seção 2.4.4.

3.3 MPI

Em [18] os autores implementam o modelo MPI versão 1 onde cada SPE é considerado um nodo MPI e utilizando a memória local da maneira convencional no modelo de memória distribuída e utilizando a memória principal para realizar as trocas de mensagens entre os nodos. A paralelização e distribuição de carga é feita explicitamente pelo programador, da forma convencional que o modelo de programação MPI requer.

3.4 Software Development Kit do *Cell BE*

Em qualquer processador, as otimizações de código geralmente exploram as características intrínsecas do hardware. Algumas dessas otimizações podem ser feitas pelo compilador enquanto outras ficam a cargo do programador. No caso do *Cell BE* as características únicas de sua arquitetura deixam em grande parte a responsabilidade pela otimização nas mãos do programador. Pensando nisso, foram criados alguns modelos de programação que se encaixam muito bem na arquitetura do *Cell BE* e cada modelo é específico para uma classe de aplicações. O SDK do *Cell BE* provê algumas bibliotecas que abstraem alguns aspectos da arquitetura dependendo do modelo de programação utilizado. Os modelos

descritos no manual de programação do SDK do *Cell BE* que mais são relacionados com o trabalho aqui proposto são [15]:

Modelo de descarga de função ou chamada de procedimento remoto: Neste modelo, os SPEs são usados como aceleradores para funções de desempenho crítico. Esta é uma maneira rápida para transformar uma aplicação existente em uma aplicação que usa efetivamente os SPEs. Este modelo também é chamado de Chamada de Procedimento Remoto. O programa principal roda no PPE e faz chamadas às funções que estão alocadas nos SPEs. As chamadas de função podem ser assíncronas (podendo executar em paralelo) ou síncronas e o software continua serial. Os parâmetros da função e valores de retorno são trasmitidos através de chamadas DMA que podem ser abstraídas do programador pelo compilador ou por uma biblioteca que forneça o cabeçalho padrão para as funções que rodam no SPE.

Modelo de aceleração de cálculos: Este modelo é descrito pelo manual do SDK como um modelo centrado no SPE. A maior parte da execução da aplicação fica a cargo do SPE provendo um uso mais integrado dos SPEs. Ele é geralmente usado para acelerar aplicações que utilizam intensivamente cálculos matemáticos. O trabalho deve ser particionado manualmente pelo programador (ou automaticamente por compiladores). Os SPEs devem escalonar eficientemente as transferências DMA necessárias para acesso aos dados. Este modelo pode usar memória compartilhada mas ele não prevê mecanismos para ajudar no acesso aos dados compartilhados e não impede condições de corrida. Pode-se também usar o modelo de *Message Passing Interface (MPI)*.

Modelo de fluxo de dados: Neste modelo os SPEs trabalham em um modo de *pipeline*, seja serial ou paralelo, realizando computação conforme os dados fluem. O PPE deve agir apenas como controlador e os SPEs agem como processadores de fluxo de dados. Geralmente desenha-se um *pipeline* onde os dados são transmitidos por DMA para um SPE inicial que realiza uma parte da computação. O dado é então copiado para o SPE adjacente (dentro do *chip*, sem usar a memória principal) e assim por diante até o último SPE que copia de volta o resultado final para a memória principal. As transferências de dados entre os SPEs são feitas através de DMA de memória local para memória local nunca precisando sair do *chip* para a memória principal, por isso essas trasnferêcias são muito mais rápidas. Se a aplicação puder ser particionada neste modelo de forma que o *pipeline* seja usado eficientemente (sem gerar bolhas, etc), este modelo é um perfeito candidato. Porém, a dificuldade de se desenhar a aplicação em forma de *pipeline* faz com que este modelo seja utilizado somente para aplicações muito específicas. Por exemplo aplicações multimídias ou processamento de sinais onde o fluxo de dados na aplicação é evidente.

Modelo de memória compartilhada: Por fim, o manual do SDK descreve o modelo de memória compartilhada tradicional onde se deve usar *mutexes* manualmente para

fazer o controle de concorrência. No caso do *Cell BE*, como já foi descrito, os SPEs devem utilizar as transferências DMA para acesso à memória compartilhada e os *mutexes* são implementados com as instruções DMA especiais. O SDK não provê nenhum método para ajudar no controle de concorrência. Para acesso à memória principal pode-se utilizar uma biblioteca de Software Cache presente no SDK, mas como já descrito, não existe garantia de coêrencia nesta implementação.

O SDK do *Cell BE*, portanto, traz para esta arquitetura os mesmos modelos já existentes nas arquiteturas clássicas de multi-processadores sem adicionar nenhuma novidade.

3.5 Memórias Transacionais

Dice et al. em [6] argumentam que contribuíram com o primeiro modelo de memória transacional em software capaz de garantir de forma eficiente que a transação em nenhum momento execute em um estado de memória inconsistente. Ou seja, utilizando o modelo de validação adiantado como discutido na seção 2.4.2.

Por causa disso e pelo seu bom desempenho, este modelo chamado *Transacional Locking II (TL2)* ficou famoso no meio acadêmico. O segredo do bom desempenho deste algoritmo está na utilização de um relógio global de versão similar ao relógio de Lamport [20] para realizar a validação das leituras e da efetivação das transações. Neste trabalho, os autores mostram que este relógio pode ser construído de maneira que para qualquer tipo de transação (com exceção das muito pequenas), os efeitos de contenção são mínimos.

O algoritmo usado pela TL2 pode ser descrito através dos conceitos introduzidos na seção 2.4 tais como:

- Utiliza *Commit-Log*
- Baseado em *mutex* (*Lock-Based*)
- Com detecção de conflitos Adiantado
- Baseado em palavras
- Completamente implementado em Software (STM)

O algoritmo TL2 pode ser brevemente descrita nos seguintes passos:

1. **Ler o relógio global de versão:** O relógio global de versão é mantido pelo sistema para realizar a consistência do versionamento das palavras na memória. Cada palavra na memória é associada a um *mutex* e um número de versão (que é chamado de diretório de versionamento). Ao iniciar uma transação, o relógio global de versão é copiado para uma variável local da *thread*, esta variável é chamada de `versão da transação`.

2. **Executar a transação:** Para cada transação, um conjunto de leituras e um de escritas é mantido para que, no fim da transação, se efetivada, os dados do conjunto de escritas sejam copiados para o destino final.

   Cada palavra de memória acessada pela transação é primeiro verificada se ela já está no conjunto de escritas. Isso impede que os dados já escritos por essa transação deixem de ser vistos por ela mesma (semelhante à condição de corrida do tipo *read-after-write*).

   Após isso, caso a palavra não seja encontrada no conjunto de escritas, o valor da variável é lido da memória compartilhada. Porém, é necessário que esta palavra lida seja primeiro validada. Se for inválida, a transação é abortada.

3. **Adquirir mutexes:** Quando a transação entra em fase de efetivação, ela precisa adquirir todos os *mutexes* para cada palavra no seu conjunto de escritas. A aquisição é feita usando a técnica de *Two-Phase-Locking*, de forma ordenada no endereço da palavra, para evitar *dead-locks*.

4. **Incrementar o relógio global de versão:** Em seguida deve-se incrementar o relógio global de versão na memória principal. Este novo número é a `versão de efetivação` da transação.

5. **Validar:** Deve-se validar novamente (por completo) o conjunto de leituras. Se não for válido a transação é abortada.

6. **Efetivar a transação:** Agora a transação pode, garantidamente, escrever os dados do conjunto de escritas nos locais corretos. Para cada palavra do conjunto de escritas, é necessário ainda escrever a `versão de efetivação` no diretório de versionamento correspondente a cada endereço de palavra. Por fim libera-se todos os *mutexes* adquiridos.

Tanto durante a execução da transação quanto na fase de efetivação, as leituras são validadas da seguinte forma: O *mutex* correspondente àquela palavra não pode estar adquirido por outra *thread* e a versão daquela palavra de memória (contida no diretório de versionamento) deve ser menor ou igual à `versão da transação` (obtida no início da transação). Para mais detalhes sobre o algoritmo *TL2* veja [6].

O modelo de STM chamado *tinySTM* descrito em [11] possui características semelhantes ao TL2 com apenas alguns aspectos sutis de diferença. A *tinySTM* adquire os *mutexes* à medida que encontra uma operação de escrita. Isso pode permitir um desempenho um pouco maior pois, ao adquirir o *mutex* com antecedência, as outras *threads* que lerem esta palavra vão ser obrigadas a abortar mais cedo salvando trabalho inútil que seria abortado ao fim da transação.

Ao utilizar memórias transacionais, o programador define seção crítica no código através de alguma anotação especial. Todos os acessos à memória dentro da seção crítica devem ser feitos utilizando a memória transacional. Para ajudar o programador na tarefa

de transformar cada acesso à memória numa chamada à biblioteca de memória transacional surgiram alguns trabalhos que visam alterar o compilador para realizar esta tarefa de forma automática. Em [10] os autores descrevem como o compilador LLVM foi alterado para instrumentar o código transacional desta maneira.

3.6 Comparação

Ao utilizar uma das soluções fornecidas pelos trabalhos anteriores, o programador do *Cell BE* se vê na necessidade de escolher um modelo que provê algumas mas não todas as seguintes características:

- Provê mecanismos de sincronização e controle de concorrência?
- Provê garantia de coerência no acesso aos dados?
- Facilita o acesso à memória principal a partir do SPE?
- Provê mecanismos para facilitar a programação na arquitetura do *Cell BE*?

A tabela 3.1 mostra uma comparação dos trabalhos apresentados (colunas) em relação à estas características (linhas).

	[2]	[28]	[18]	[15]	[6]	[10]
Sincronização e controle de concorrência					•	•
Garantia de coerência					•	•
Acesso à memória principal	•			•		
Facilita programação na arquitetura do *Cell BE*	•	•	•	•		

Tabela 3.1: Comparação dos trabalhos anteriores

Como se vê, até o momento nenhum modelo é capaz de fornecer todas estas características em conjunto. Esta é a proposta desta dissertação.

Capítulo 4

Software Cache Transacional

Como foi mostrado, as soluções que abstraem a sincronização e controle de concorrência não são capazes de lidar com a arquitetura do *Cell BE* ou acesso à memória principal. A figura 4.1 ilustra as duas soluções que resolvem cada problema separadamente, são elas:

Software Cache: Resolve o problema do acesso à memória principal / DMA. Porém adiciona o problema de coerência dos dados na *cache*.

Memória Transacional: Resolve o problema de controle de concorrência.

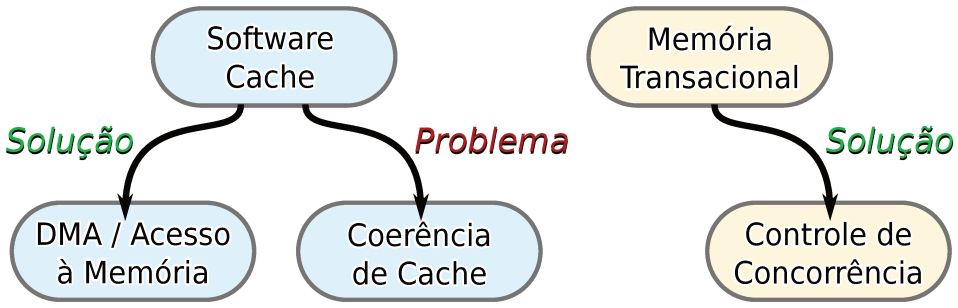


Figura 4.1: Problema

Soma-se a isso então, o fato de que a utilização da Software Cache em um código convencional é muito similar à utilização da Memória Transacional em Software em um código paralelo. Nos dois casos é necessário substituir todas as leituras e escritas da memória principal por uma chamada à biblioteca. Frequentemente isto é feito utilizando-se macros como nos exemplos mostrados nas Figuras 4.2(a), usando Software Cache, e 4.2(b), usando Memória Transacional. Como se vê os dois códigos são similares e portanto, do ponto de vista do programador, existe uma motivação para unir estes dois conceitos.

```
1 int local_value = LOAD(value_ptr);
2 local_value++;
3 STORE(value_ptr, local_value);
```

(a) Software Cache

```
1 TRANSACTION_BEGIN();
2 int local_value = LOAD(value_ptr);
3 local_value++;
4 STORE(value_ptr, local_value);
5 TRANSACTION_END();
```

(b) Memória Transacional

Figura 4.2: Utilização de Software Cache e de Memória Transacional.

Unindo esses dois conceitos, a Software Cache facilita o acesso à memória principal enquanto a Memória Transacional resolve o controle de concorrência e a coerência de dados na *cache* como um único problema, como ilustrado na figura 4.3. Isso facilita o acesso aos dados na memória principal a partir do SPE e permite que o programador acesse elegantemente dados compartilhados sem a necessidade do uso de *mutex*.

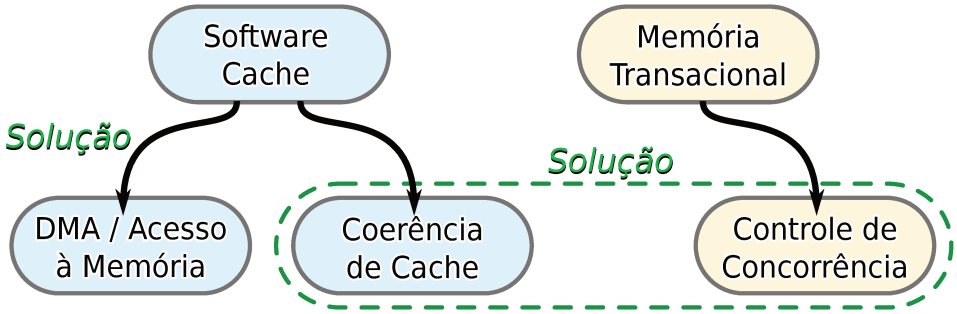


Figura 4.3: Solução

Neste trabalho será descrito um modelo de memória transacional baseado em Software Cache, aqui chamado de *Software Cache Transacional*. Este modelo é baseado no algoritmo do TL2 [6] adaptado para a arquitetura do *Cell BE* em conjunto com uma implementação de Software Cache. Esta Software Cache baseia-se na implementação de Software Cache originalmente presente no IBM SDK [15] e em pequena parte na implementação de Software Cache presente no compilador GCC para o *Cell BE*.

4.1 Programando com Software Cache Transacional

Do ponto de vista do programador, o modelo de Software Cache Transacional não adiciona nenhuma complexidade além da existente na Software Cache convencional. Como foi explicado antes, o modelo de Software Cache requer que cada acesso à memória principal seja instrumentado por uma chamada de função da biblioteca que esta sendo utilizada, e isso também acontece com a maioria das implementações de Memória Transacional em

```
1  bool_t list_remove (list_t* listPtr, void* dataPtr) {
2      list_node_t* prevPtr = list_findPrevious(listPtr, dataPtr);
3      list_node_t* nodePtr = prevPtr->nextPtr;
4      if ((nodePtr != NULL) &&
5          ( compare(nodePtr->dataPtr, dataPtr) == 0 )) {
6          prevPtr->nextPtr = nodePtr->nextPtr;
7          nodePtr->nextPtr = NULL;
8          freeNode(nodePtr);
9          listPtr->size--;
10         return TRUE;
11     }
12     return FALSE;
13 }
14 int main() {
15     ...
16     list_insert(listPtr, value_X);
17     list_find(listPtr, value_X);     // Deve sempre encontrar X
18     list_remove(listPtr, value_X);
19     ...
20 }
```

Figura 4.4: Código original

Software. O primeiro modelo aqui proposto, utiliza macros da linguagem C da seguinte forma:

- `LOAD(endereco_efetivo)`: Retorna o valor da palavra na memória principal apontada por: `endereco_efetivo`;
- `STORE(endereco_efetivo, valor)`: Grava o `valor` na memória principal apontada por: `endereco_efetivo`;

As *threads* executando nos SPEs utilizam essas macros para acesso aos dados. O controle de concorrência nos dados compartilhados, a coerência de dados entre as *caches*, as operações DMA para acesso à memória principal, tudo isso fica escondido do programador atrás dessas duas macros que, por fim, realizam chamadas de função na biblioteca de Software Cache Transacional.

Do ponto de vista do programador, portanto, o uso do modelo de Software Cache Transacional é similar à Software Cache convencional. Para ilustrar isso, é apresentado aqui um exemplo prático de como converter o código fonte de uma fução de remoção em lista ligada ordenada para o modelo de Software Cache Transacional. Inicialmente tem-se o código original da função de remoção em lista ligada e sua utilização em um bloco de código aprensentado na figura 4.4. Este código foi escrito em C de forma padrão para executar sequencialmente em um processador com acesso direto à memória.

O objetivo é transformar este código para que ele execute no SPE. Primeiramente, considera-se que o código vai executar em apenas um SPE, não havendo, portanto, pa-

ralelismo. As linhas 3, 5, 6, 7 e 9 no código da Figura 4.4 fazem acessos à memória e, portanto, precisam ser convertidos. Caso o programador opte por utilizar DMA sem Software Cache e Memória Transacional, o código após a transformação necessária ficará semelhante ao apresentado na figura 4.5. Perceba que, neste caso, é necessário a introdução de quatro variáveis locais novas para guardar temporariamente os valores que foram lidos da memória principal. Além disso o código fica consideravelmente mais complexo. Deve-se enfatizar também que todas as outras sub-rotinas chamadas pela função `list_remove` também devem ser transformadas para acessar a memória principal da forma correta.

Caso o programador opte por utilizar Software Cache, porém, o código fica um pouco mais simples. Neste caso ele deve utilizar as macros `LOAD` e `STORE`, apresentadas anteriormente, para acessar a memória principal. Por exemplo, a primeira linha de código marcada em negrito na figura 4.4 é uma cópia da variável `prevPtr->nextPtr` para uma variável local. Esta variável está localizada na memória principal no endereço `&(prevPtr->nextPtr)` e portanto é necessário utilizar a macro `LOAD` para fazer a leitura deste valor. A figura 4.6 mostra como o código vai ficar após ser transformado para utilizar Software Cache desta maneira. Todo o código marcado em negrito sofreu uma modificação parecida com a exemplificada.

Até então foi mostrado como o código é transformado para executar sequencialmente em um único SPE. Caso o programador necessite executar este código em paralelo com outros SPEs, será necessário utilizar o modelo de Software Cache Transacional. Para isso é necessário encontrar quais partes do código deveriam executar de forma atômica, ou seja, encontrar a seção crítica no código original na Figura 4.4. As linhas 16 à 18 englobam a parte do código que é a seção crítica. Para utilizar o modelo de Software Cache Transacional, tudo que o programador precisa fazer para que este código seja executado atômicamente é adicionar duas macros `TM_BEGIN` e `TM_END` no começo e no fim da seção crítica como mostrado na Figura 4.7. A função `list_remove` permanece exatamente igual à transformação feita para Software Cache e agora o código já pode executar em paralelo.

```
1  bool_t list_remove (list_t* listPtr, void* dataPtr) {
2      list_node_t* prevPtr = list_findPrevious(listPtr, dataPtr);
3
4      list_node_t* nodePtr;
5      mfc_get(&nodePtr, &(prevPtr->nextPtr), sizeof(list_node_t*), 1, 0, 0);
6      mfc_write_tag_mask(TAG);
7      mfc_read_tag_status(TAG);
8
9      if (nodePtr != NULL) {
10
11       void* local_dataPtr;
12       mfc_get(&local_dataPtr, &(nodePtr->dataPtr), sizeof(void*), 1, 0, 0);
13       mfc_write_tag_mask(TAG);
14       mfc_read_tag_status(TAG);
15
16       if ( compare(local_dataPtr, dataPtr) == 0 )) {
17
18         void* local_nodePtr_nextPtr;
19         mfc_get(&local_nodePtr_nextPtr, &(nodePtr->nextPtr),
20                 sizeof(list_node_t*), 1, 0, 0);
21         mfc_write_tag_mask(TAG);
22         mfc_read_tag_status(TAG);
23
24         mfc_put(&local_nodePtr_nextPtr, &(prevPtr->nextPtr),
25                 sizeof(list_node_t*), 1, 0, 0);
26         mfc_write_tag_mask(TAG);
27         mfc_read_tag_status(TAG);
28
29         list_node_t* nulo = NULL;
30         mfc_put(&nulo, &(nodePtr->nextPtr), sizeof(list_node_t*), 1, 0, 0);
31         mfc_write_tag_mask(TAG);
32         mfc_read_tag_status(TAG);
33
34         freeNode(nodePtr);
35
36         int local_size;
37         mfc_get(&local_size, &(listPtr->size), sizeof(int), 1, 0, 0);
38         mfc_write_tag_mask(TAG);
39         mfc_read_tag_status(TAG);
40
41         local_size--;
42
43         mfc_put(&local_size, &(listPtr->size), sizeof(int), 1, 0, 0);
44         mfc_write_tag_mask(TAG);
45         mfc_read_tag_status(TAG);
46
47         return TRUE;
48       }
49     }
50     return FALSE;
51 }
```

Figura 4.5: Código utilizando DMA no SPE

```
1  bool_t list_remove (list_t* listPtr, void* dataPtr) {
2      list_node_t* prevPtr = findPrevious(listPtr, dataPtr);
3      list_node_t* nodePtr = LOAD( &(prevPtr->nextPtr) );
4      void* node_dataPtr;
5      if (nodePtr != NULL) {
6          node_dataPtr = LOAD( &(nodePtr->dataPtr) );
7      }
8      if ((nodePtr != NULL) &&
9          ( compare(node_dataPtr, dataPtr) == 0) ) {
10         STORE( &(prevPtr->nextPtr), LOAD(&(nodePtr->nextPtr)));
11         STORE( &(nodePtr->nextPtr), NULL );
12         freeNode(nodePtr);
13         long size = LOAD( &(listPtr->size) );
14         size--;
15         STORE( &(listPtr->size), size );
16         return TRUE;
17     }
18     return FALSE;
19 }
```

Figura 4.6: Código utilizando Software Cache no SPE

```
1  int main() {
2      ...
3      TM_BEGIN();
4        list_insert(listPtr, value_X);
5        list_find(listPtr, value_X);    // Must Always Find X
6        list_remove(listPtr, value_X);
7      TM_END();
8      ...
9  }
```

Figura 4.7: Código utilizando Software Cache Transacional no SPE

4.1.1 Instrumentando o código automaticamente

Até agora foi apresentado um exemplo de como transformar um código para Software Cache Transacional instrumentando o código manualmente. Porém como já foi dito anteriormente, a instrumentação do código para Software Cache ou para Memória Transacional pode ser feita de forma automática pelo compilador.

O próprio compilador GCC para *Cell BE* possui um modelo de Software Cache simples que pode ser utilizado pelo programador através do atributo `__ea` ao declarar uma variável ponteiro que aponta para a memória externa [15]. A variável com este atributo é declarada da seguinte maneira:

```
__ea int * ponteiro_de_inteiro = [ENDERECO NA MEMORIA PRINCIPAL];
```

O ponteiro declarado com este atributo pode ser lido e escrito diretamente através das construções padrões da linguagem C, por exemplo: `*ponteiro`. O GCC instrumenta automaticamente o código de maneira que os acessos desta variável a partir do SPE à memória principal sejam feitos através da Software Cache. Junto com o código instrumentado que foi compilado, o GCC inclui o código objeto da biblioteca de Software Cache interna.

O trabalho apresentado aqui tira proveito desta característica do GCC para instrumentar automaticamente o código com Software Cache Transacional. Utilizando Software Cache Transacional, o programador pode declarar variáveis com o atributo `__ea` para apontar para endereços da memória externa da mesma maneira que a Software Cache comum que o GCC provê. Ao instrumentar o código, o GCC substitui as leituras de ponteiros declarados com `__ea` por chamadas de função da biblioteca de Software Cache interna. As assinaturas das principais funções da biblioteca de Software Cache interna do GCC são:

- `void *__cache_fetch (__ea void *effective_address)` :
  Recebe um endereço na memória principal, faz a leitura da memória principal e copia para o bloco de *cache* correspondente na memória local. Retorna o endereço do bloco de *cache* na memória local. O endereço na memória principal e o endereço de retorno não precisam estar na borda de um bloco de *cache*, o deslocamento necessário é manipulado internamente pela biblioteca. O endereço retornado não pode ser usado para escrita, pois o bloco de *cache* não foi marcado como sujo, mas o compilador garante isso ao usar esta função somente em leituras.

- `void *__cache_fetch_dirty (__ea void *effective_address, int n_bytes_dirty)` :
  Realiza o mesmo trabalho que a função `__cache_fetch()`, porém marca os `n_bytes_dirty` como sujos no bloco de *cache*. Isso permite que o endereço retornado seja usado para escrita. A escrita é feita no bloco de *cache* presente na memória local e em um dado momento a biblioteca cuidará de copiar para a memória externa.

- `void __cache_flush (void)` :
  Realiza a descarga completa da memória *cache*, copiando todos os blocos de *cache* sujas para os respectivos lugares na memória principal.

As chamadas de função para leitura de uma variável utilizando Software Cache comum são semanticamente idênticas às chamada de Software Cache Transacional. Com isso pode-se seguramente substituir as chamadas de Software Cache comum pelas Transacionais (o inverso não é verdade pois a Software Cache comum não garante coerência nem controle de concorrência). Portanto, foi fácil tornar o GCC um compilador capaz de instrumentar código com Software Cache Transacional. Para isso, bastou que, no momento de ligação do código objeto não se incluísse a biblioteca de Software Cache do GCC e incluísse a biblioteca de Software Cache Transacional utilizando a mesma assinatura de função para leitura e escrita de variáveis.

Utilizar a instrumentação automática deixa o código muito mais limpo, menor e mais fácil de erros serem detectados pelo compilador. Um dos problemas de se utilizar as macros `LOAD` e `STORE` explicitamente é que a checagem de tipo feita na hora de compilação não pode ser feita nas atribuições de variáveis que utilizam estas macros. Isso acontece pois, internamente, a biblioteca de Software Cache não possui um tipo específico para os dados que estão sendo guardados e todas as atribuições são feitas como ponteiros `void`. O mesmo acontece ao se utiliazr DMA explicitamente. Com isso, erros simples de atribuição de variáveis de forma incorreta passam desapercebidos. Ao declarar o ponteiro com o atributo `__ea` o GCC é capaz de fazer checagem de tipos e ainda realizar algumas das otimizações que são possíveis em ponteiros.

```
1  bool_t list_remove (__ea list_t* listPtr, __ea void* dataPtr) {
2      __ea list_node_t* prevPtr = list_findPrevious(listPtr, dataPtr);
3      __ea list_node_t* nodePtr = prevPtr->nextPtr;
4      if ((nodePtr != NULL) &&
5          ( compare(nodePtr->dataPtr, dataPtr) == 0 )) {
6          prevPtr->nextPtr = nodePtr->nextPtr;
7          nodePtr->nextPtr = NULL;
8          freeNode(nodePtr);
9          listPtr->size--;
10         return TRUE;
11     }
12     return FALSE;
13 }
14 int main() {
15     ...
16     TM_BEGIN();
17     list_insert(listPtr, value_X);
18     list_find(listPtr, value_X);     // Deve sempre encontrar X
19     list_remove(listPtr, value_X);
20     TM_END();
21     ...
22 }
```

Figura 4.8: Código instrumentado usando Software Cache Transacional no SPE

Por fim, o código original da Figura 4.4 utilizando Software Cache Transacional com instrumentação automática está apresentado na Figura 4.8. Veja como o código fica muito parecido com o original. Apenas as variáveis que apontam para a memória externa necessitam serem declaradas com o atributo `__ea`. Todas as leituras e escritas nessas variáveis são manipuladas pelo compilador para utilizar a biblioteca, inclusive nos acessos à variáveis membros de uma estrutura que está na memória principal, por exemplo: `listPtr->size--;`. Para receber o ponteiro de uma estrutura na memória principal, o programador também utiliza a sintaxe usual da linguagem C, por exemplo `&(listPtr->size)` que retorna o endereço efetivo da variável `size` da estrutura apontada por `listPtr` que está na memória principal. Este é o modelo final que foi utilizado para implementação dos aplicativos utilizados como *benchmark* para medir o desempenho da biblioteca de Software Cache Transacional mostrados no capítulo 5.

Um destes aplicativos se chama *Genome*. No caso do *Genome*, foram feitas duas versões, na primeira o código original foi instrumentado manualmente para utilizar as macros `LOAD` e `STORE`, e na segunda foi utilizado o atributo `__ea` com instrumentação automática. Para comparação, o número de linhas de código, incluindo cabeçalhos, excluindo linhas em branco, comentários e testes de unidade é apresentado na tabela 4.1. Nesta tabela o código original não utiliza Software Cache nem Memória Transacional. A instrumentação manual e a instrumentação automática utilizam a Software Cache Transacional e, por comparação, é apresentada uma estimativa de quantas linhas de código seriam necessárias para se ter o mesmo código utilizando DMA explicitamente sem Software Cache nem Memória Transacional.

	Número de linhas	Porcentagem
Código Original	2103	100%
Instrumentação Automática	2153	102%
Instrumentação Manual	2716	129%
DMA explícito[1]	$\geq$ 3196	$\geq$ 152%

Tabela 4.1: Número de linhas de código para o aplicativo *Genome*

[1]Estimativa considerando que cada linha de código contendo uma leitura de variável na memória externa precisa de pelomenos 3 linhas de código para criar a variável local, fazer o DMA e fazer a atribuição necessária.

4.2 Infraestrutura da Software Cache Transacional

No modelo de Software Cache Transacional proposto, a Memória Transacional atua como uma camada em cima da Software Cache convencional intermediando os acessos à memória principal como ilustrado na figura 4.9.

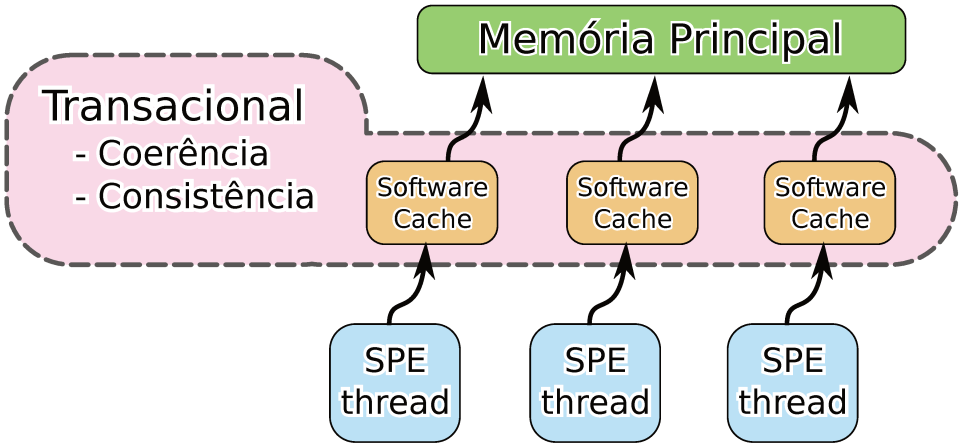


Figura 4.9: Software Cache Transacional

Para isso, além das operações mostradas na seção 2.5 foram adicionadas duas operações novas à Software Cache convencional: O *fetch* transacional e o *writeback* transacional. Quando uma operação de *fetch* é executada dentro da transação, a versão transacional desta operação é executada. O mesmo acontece para a operação de *writeback*.

Uma visão global da infraestrutura está representada na figura 4.10. Os principais componentes que a integram são:

Unidade SPU: Contém o conjunto de leituras, o conjunto de escritas e a estrutura de Software Cache na memória local do SPE. Existe apenas uma *thread* executando em cada processador SPE.

Unidade PPU: Contém a *thread* PPE responsável por realizar o procedimento de efetivação das transações.

Memória Principal: Contém a memória compartilhada endereçável pelo programa paralelo sendo executado e o diretório de versionamento da memória transacional. Este diretório mantém um número de versão e um *mutex* para cada bloco de *cache* da memória compartilhada endereçável.

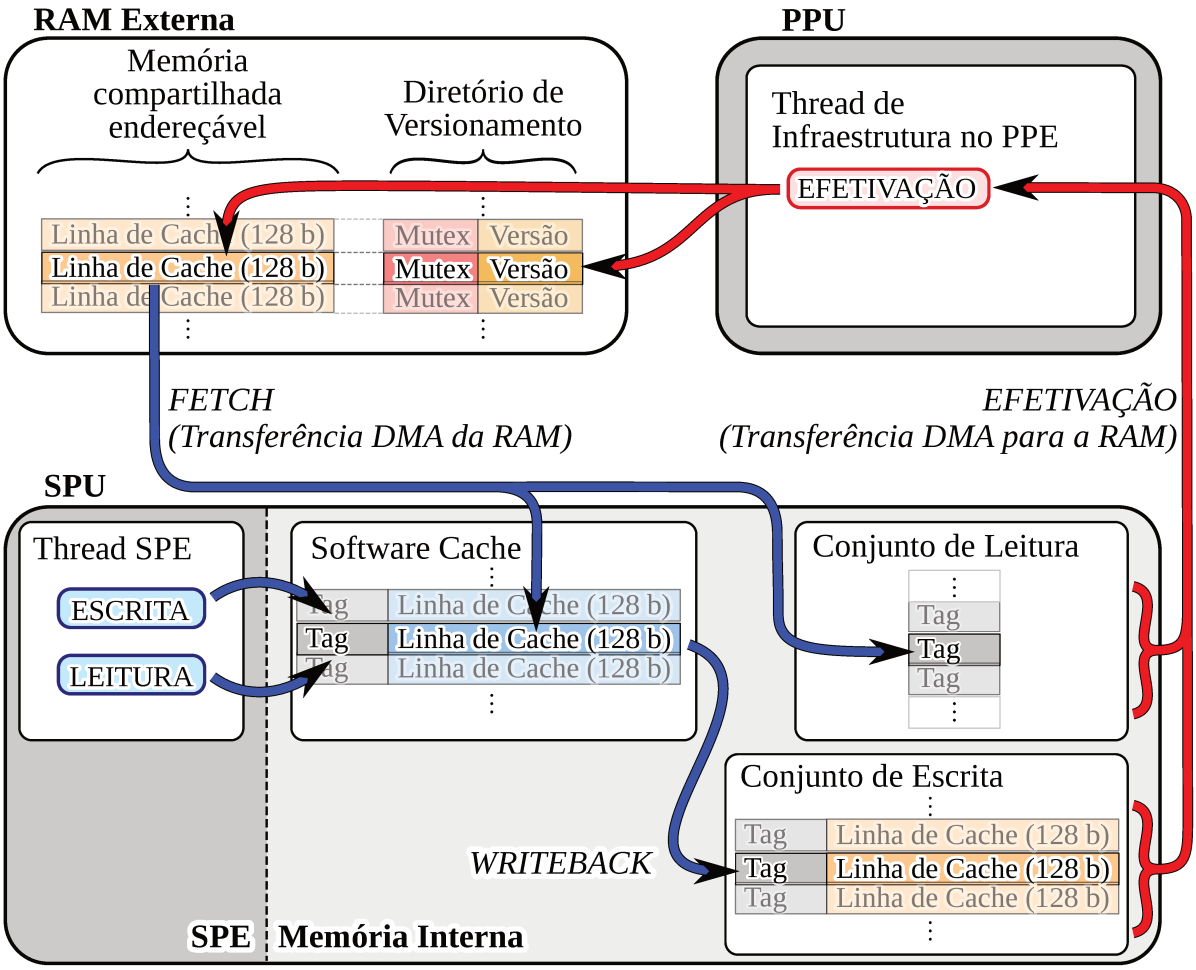


Figura 4.10: Infraestrutura

O conjunto de escritas e de leituras, junto com uma estrutura chamada descritor da transação são guardados na memória local do SPE, assim como a própria Software Cache. O descritor da transação contém a `versão da transação` usada para validar as leituras, um ponteiro para o diretório de versionamento, um ponteiro para a estrutura de efetivação da transação, além de algumas outras variáveis como o ID do SPE, o número de *threads* entre outras. O diretório de versionamento e a estrutura de efetivação são mantidos na memória principal. A linha azul na Figura 4.10 ilustra as operações de *fetch* transacional e *writeback* transacional ocorrendo após uma escrita e uma leitura de blocos de *cache*, enquanto a linha vermelha ilustra a operação de efetivação. Cada parte dessa estrutura, as operações e os algoritmos serão detalhados a seguir.

4.2.1 Iniciando a Transação

Quando uma transação é iniciada, é necessário salvar o contexto de execução para que seja possível reiniciar a transação restaurando o contexto quando uma transação é abortada. Para salvar e restaurar o contexto de execução foram utilizadas as funções `setjmp()` e `longjmp()` da *libC*. O que a função `setjmp()` faz é salvar o conteúdo dos registradores da máquina em uma estrutura do tipo `jmp_buf`. Desta maneira, a função `longjmp()` pode restaura-los e fazer com que o contexto de execução volte ao estado em que o `setjmp()` foi chamado.

Porém, para que uma transação possa ser reiniciada da maneira correta ao ser abortada, é necessário restaurar não somente os registradores, mas também os valores das variáveis locais do momento em que a transação foi iniciada. Ou seja, além de salvar os registradores, é necessário salvar também a pilha de execução. Essa necessidade é evidenciada na Figura 4.11 onde é mostrada uma transação que engloba um laço. Por simplicidade, suponha que a variável de indução do laço é uma variável local que está na pilha e não foi promovida para um registardor pelo compilador. Quando a transação é abortada, caso esta variável não seja restaurada, a transação vai reiniciar e o laço vai usar o valor corrente da variável de indução. Se, por exemplo, o laço fez 3 iterações e a transação abortou, ao reiniciar a transação o laço vai começar com a variável de indução igual a 3, porém, como a transação abortou, as 3 primeiras iterações do laço foram em vão e deveriam ser re-executadas. Este exemplo foi uma situação simplista porém, como se pode imaginar, existem diversas situações onde é necessário restaurar as variáveis locais, isto acontece comumente quando a transação está no meio de um bloco de código e não engloba uma função inteira. Para corrigir este problema foi implementada uma fução simples que analisa a pilha de execução e salva os valores da pilha numa estrutura separada. Para restaurar, é feita uma cópia dos valores de volta para a pilha. Quanto às variáveis que não são locais, ou seja, as globais e compartilhadas, elas são versionadas pelo sistema de memória transacional e, portanto, não existe necessidade para salva-las e restaura-las.

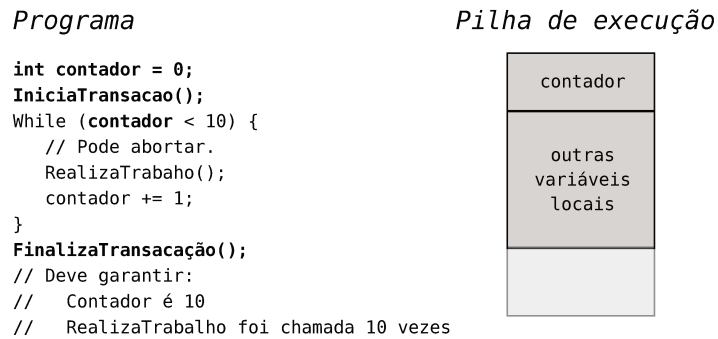


Figura 4.11: Exemplo onde é necessário salvar a pilha.

Além de salvar a pilha e o contexto, o relógio global de versionamento é copiado da memória principal para a variável `versão da transação` no descritor de transação

Algoritmo 4.1 Inicia Transacao

```
1: conjunto_leituras: Conjunto de leituras.
2: conjunto_escritas: Conjunto de escritas.
3: versao_transacao: Versão da transação corrente.
4: transacao_iniciada: Variável booleana indicando se esta dentro de uma transação.
5: endereco_relogio_global: Endereço do relógio global de versão.

6: SalvarPilha() {Variáveis locais.}
7: SalvarContexto() {Registradores.}
8: DescargaCache()
9: transacao_iniciada ← Verdadeiro
10: versao_transacao ← TransferenciaDMA_De(endereco_relogio_global)
11: conjunto_leituras ← ∅
12: conjunto_escritas ← ∅
```

armazenado na memória local, e os conjuntos de leitura e escrita são inicializados como vazio. O algoritmo 4.1 mostra o procedimento necessário para iniciar uma transação.

4.2.2 Executando a Transação

Dentro da transação, a cada leitura de um valor na memória principal, uma operação de *fetch* transacional é realizada. Esta operação insere o endereço do bloco de *cache* lido no conjunto de leituras (se já não estiver presente). Caso o endereço que está sendo lido já esteja presente na *cache*, o valor é retornado e a operação está terminada. Caso contrário, é então selecionado um local no vetor de linhas de *cache* conforme a política de substituição e a associatividade da *cache*. A política de substituição utilizada é sempre a *Round-Robin* e a associatividade é de `4-vias`. Então, verifica-se se o conjunto de escritas já contém o endereço sendo lido (ou seja, se o endereço já foi previamente alterado pela transação corrente). Em caso afirmativo o valor é copiado do conjunto de escritas para o local selecionado no vetor de linhas de *cache*, isso é necessário para garantir que valores previamente escritos não serão perdidos durante a transação corrente. Em caso negativo, é necessário realizar uma transferência DMA para copiar o valor da memória principal para o bloco de *cache* na memória local. Junto com a transferência DMA, é copiado também o número da versão atual no diretório de versionamento correspondente àquele bloco de *cache*. Este número de versão é comparado com a `versão da transação` corrente para validação. Caso a validação da leitura falhe, a transação é abortada. O algoritmo 4.2 mostra a validação da leitura. E o algoritmo 4.3 mostra as operações necessárias para realizar o *Fetch* Transacional.

A operação de *writeback* transacional está representada no algoritmo 4.4 e diferentemente do *writeback* convencional, não transfere o bloco de *cache* modificada para o seu endereço final na memória principal. Em vez disso, o endereço e o valor do bloco de *cache* são adicionados no conjunto de escritas. Esta operação é realizada quando um bloco de

cache modificado precisa ser substituído durante uma transação.

Algoritmo 4.2 Valida Leitura

```
1: endereco_leitura: Endereço do dado sendo acessado.
2: cache: Linhas/blocos de cache na memória local.
3: versao_transacao: Versão da transação corrente.

   {O endereço da versão no diretório de versionamento aponta para memória principal.}
4: endereco_versao ← DiretorioDeVersionamento(endereco_leitura)
5: versao_leitura ← TransferenciaDMA_De(endereco_versao)
   {O mutex é implementado como 1 bit do número de versão.}
6: se BitMutexEstaLigado(versao_leitura) ou versao_leitura > versao_transacao então
7:    AbortaTransação()
8: senão
9:    retorna Sucesso
10: fim se
```

Algoritmo 4.3 *Fetch* Transacional

```
1: endereco_leitura: Endereço do bloco de cache sendo acessado.
2: conjunto_leituras: Conjunto de leituras.
3: conjunto_escritas: Conjunto de escritas.
4: versao_transacao: Versão da transação corrente.
5: cache: Linhas/blocos de cache na memória local.
6: transacao_iniciada: Variável booleana indicando se esta dentro de uma transação.

7: se cache contém endereco_leitura então
8:    retorna cache[endereco_leitura]
9: senão se transacao_iniciada então
10:   conjunto_leituras[] ← endereco_leitura
11:   se conjunto_escritas contém endereco_leitura então
12:      retorna conjunto_escritas[endereco_leitura]
13:   senão
14:      ValidaLeitura(endereco_leitura, versao_transacao)
15:      cache[endereco_leitura] ← TransferenciaDMA_De(endereco_leitura)
16:      ValidaLeitura(endereco_leitura, versao_transacao)
17:   fim se
18: senão
19:   FetchConvencional(endereco_leitura)
20: fim se
```

Note que as operações mencionadas até então são realizadas por um SPE sozinho. Para tirar vantagem da arquitetura do *Cell BE*, algumas operações que SPE necessita fazer são realizadas com a ajuda do PPE, por exemplo, alocar memória na memória principal entre outras. Essas operações que necessitam da ajuda do PPE são realizadas como uma chamada de procedimento remoto (*RPC*) e foram implementadas utilizando *mailboxes*. A implementação desse tipo de *RPC* foi feita a partir do zero para se encaixar na infraestrutura de Software Cache Transacional deste trabalho, porém seu funcionamento é

Algoritmo 4.4 *Writeback* Transacional

```
1: conjunto_escritas: Conjunto de escritas
2: endereco: Endereço do bloco de cache
3: valor: Valor do bloco de cache
4: transacao_iniciada: Variável booleana indicando se esta dentro de uma transação.

5: se transacao_iniciada então
6:     conjuntoEscrita[endereco] ← valor
7: senão
8:     WritebackConvencional(endereco, valor)
9: fim se
```

similar ao mecanismo chamado `spe_callback_handler_register` presente no SDK da IBM, a diferença é que este último utiliza DMA para transferência dos argumentos da chamada *RPC*.

4.2.3 Efetivando a Transação

A tarefa mais importante pela qual o PPE é responsável é o procedimento de efetivação da transação. Quando um SPE está pronto para realizar a efetivação, ele transfere seu conjunto de leituras e de escritas para a estrutura de efetivação da transação localizada na memória principal. Esta estrutura é previamente alocada pelo PPE antes do início da aplicação e cada SPE possui um local separadamente designado, portanto a etapa de transferência dos conjuntos de leitura e escrita podem ser feitos em paralelo pelos SPEs. Após esta transferência, o SPE faz uma *RPC* avisando o PPE que a estrutura de efetivação foi preenchida e que o PPE pode realizar o procedimento de efetivação. Junto com o conjunto de leituras e de escritas, o SPE também envia ao PPE a `versão da transação` corrente que é utilizada para validação da transação. A partir de então, a operação de efetivação é realizada pelo PPE seguindo passos similares ao algoritmo do TL2 citado na seção 2.4: (1) Adquirir *mutexes* do conjunto de escrita; (2) Incrementar o relógio global de versão; (3) Validar o conjunto de leituras; (4) Efetivar a transação e liberar os *mutexes*. Ao fim da operação, o PPE envia uma resposta para o SPE avisando se a transação foi efetivada ou abortada. Se a transação necessita ser abortada, o SPE realiza o procedimento necessário reiniciando a transação, caso contrário ele continua a execução da aplicação normalmente. A parte da efetivação que é realizada pelo SPE pode ser vista no algoritmo 4.5 e o algoritmo 4.6 mostra a parte realizada pelo PPE.

Algoritmo 4.5 Efetivação da Transação (SPE)

```
1: conjunto_leituras: Conjunto de leituras.
2: conjunto_escritas: Conjunto de escritas.
3: versao_transacao: Versão da transação corrente.
4: estrutura_de_efetivacao: Estrutura de efetivação da transação localizada na memória principal.

5: DescargaCache() {Realiza Writebacks necessários.}
6: TransferenciaDMA_Para(estrutura_de_efetivacao.conjunto_leituras) ← conjunto_leituras
7: TransferenciaDMA_Para(estrutura_de_efetivacao.conjunto_escritas) ← conjunto_escritas
8: TransferenciaDMA_Para(estrutura_de_efetivacao.versao_transacao) ← versao_transacao
9: deve_abortar ← EnviaRPC_PPE('efetivacao') {RPC é síncrona/bloqueante.}
10: se deve_abortar então
11:     AbortaTransação()
12: senão
13:     retorna Sucesso
14: fim se
```

Uma discussão que vale a pena mencionar aqui é o motivo de se ter designado o PPE para realizar a maior parte da fase de efetivação. Da maneira que foi implementado, cada SPE possui uma *thread* especial correspondente executando no PPE. Esta *thread* especial é responsável pelo procedimento de efetivação e, portanto, mais de uma *thread* de efetivação pode estar executando concorrentemente no PPE. Como o PPE possui capacidade de SMT (*Simultaneous Multi-Threading*), é possível ganhar um pequeno paralelismo quando duas operações de efetivação são realizadas em paralelo e não são conflitantes. Porém, o PPE é apenas um processador com SMT e, portanto, tem uma capacidade de paralelismo menor do que a soma de todos os SPEs. Esta escolha foi feita por vários motivos. O principal motivo é que o algoritmo de efetivação utilizado necessita adquirir mutexes na memória principal e, como ja foi mencionado, esta é uma operação extremamente ineficiente se for realizada pelo SPE. O segundo motivo, a cópia final dos dados do conjunto de escritas no momento da efetivação é geralmente feita em locais esparços da memória principal, a não ser que a transação sempre escreva em endereços ajacentes. Realizar múltiplas transferências DMA para endereços esparços é menos eficiente do que realizar apenas uma transferência DMA com todo o conjunto de escritas para a estrutura de efetivação. Além disso, validar o conjunto de leituras no SPE também seria ineficiente pois teriam que ser realizados múltiplos DMAs para ler a versão de cada endereço do conjunto de leitura.

Um outro ponto importante a ser discutido é em relação à execução concorrente de uma transação que está lendo um dado na memória compartilhada (algoritmo 4.3) com outra transação que está na fase de efetivação (algoritmo 4.6). No algoritmo 4.3 podem ser vistas as duas validações da leitura, uma na linha 14 e uma na linha 16. As duas validações são estritamente necessárias nesta ordem: antes e depois da transferência DMA do valor a ser lido, caso contrário, uma condição de corrida pode acontecer conforme mostrado na figura 4.12. Nesta figura, o algoritmo de leitura erroneamente não faz a primeira validação. A *thread 1* está fazendo a efetivação e, após incrementar o relógio global de versão, a *thread*

Algoritmo 4.6 Efetivação da Transação (PPE)

1: *mutexes*: Mutexes do conjunto de escritas da transação a ser efetivada. Este conjunto possui multiplicidade 1.
2: *relogio_global*: Relógio global de versão.
3: *versao_transacao*: Versão da transação corrente.
4: *conjunto_leituras*: Conjunto de leituras.
5: *conjunto_escritas*: Conjunto de escritas.

{Adquirir os mutexes ordenados pelo endereço para evitar dead-lock.}
6: *mutexes* ← Ordena(*mutexes*)
7: **para cada** *mutex* **em** *mutexes* **faça**
8: Adquire(*mutex*)
9: **fim para**

{A versao de efetivacao é o relogio global incrementado.}
10: $versao_efetivacao_transacao$ ← IncrementaAtomicamente(*relogio_global*)

{Validar o conjunto de leituras.}
11: **para cada** *endereco_leitura* **em** *conjunto_leituras* **faça**
12: *versao_leitura* ← DiretorioDeVersionamento(*endereco_leitura*)
13: **se** (*versao_leitura* > *versao_transacao*) **ou**
(BitMutexEstaLigado(*endereco_leitura*) **e** *mutexes* **não contém** *endereco_leitura*) **então**
14: **para cada** *mutex* **em** *mutexes* **faça**
15: Libera(*mutex*)
16: **fim para**
17: EnviaRespostaRPC_SPE() ← Fracasso {Abortar a transação}
18: **fim se**
19: **fim para**

{Efetivação.}
20: **para cada** *endereco*, *valor* **em** *conjunto_escritas* **faça**
21: *memoria*[*endereco*] ← *valor*
22: DiretorioDeVersionamento(*endereco*) ← $versao_efetivacao_transacao$
23: Libera(MutexEm(*endereco*))
24: **fim para**
25: EnviaRespostaRPC_SPE() ← Sucesso

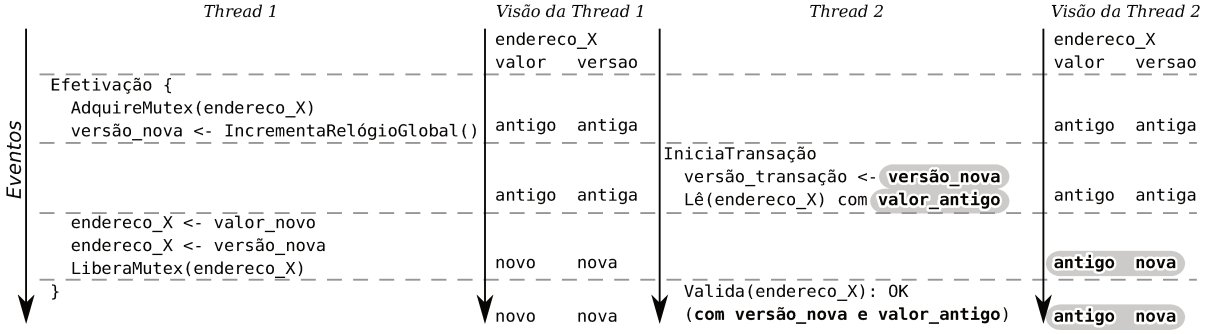


Figura 4.12: Condição de corrida na validação da leitura.

2 inicia uma transação lendo este relógio com o valor `versão_nova`. Logo em seguida, a mesma *thread* lê o valor do `endereco_X` sem realizar a primeira validação. Neste momento a *thread 2* possui a versão da transação com o novo valor porém o `endereco_X` lido possui o valor antigo. Agora, ao validar a leitura, a *thread 2* acha que foi sucesso, porém o `endereco_X` foi validado com a versão nova mas está sendo utilizado o valor antigo, o que é uma inconsistência. Este exemplo simples não gera efeitos colaterais, porém se existem duas variáveis dependentes sendo lidas, a transação pode terminar com o valor antigo de uma e o valor novo da outra e não abortar, mostrando assim a necessidade das duas validações.

4.2.4 Abortando a Transação

Existem dois momentos em que a transação pode ser abortada: após a validação de uma leitura ou após a tentativa de efetivação. Nas duas situações o procedimento para abortar são os mesmos e o SPE realiza este procedimento sem interferência do PPE. Como foi visto, a pilha contendo as variáveis locais foi salva no início da transação, assim como o contexto com os registradores. Para abortar, basta restaurar a pilha realizando a cópia dos valores e fazer uma chamada à função `longjmp()` para restaurar o contexto. Quando o contexto é restaurado, a *thread* volta para o estado em que estava quando chamou a função `setjmp()` na linha 7 do algoritmo 4.1. A partir de então a transação foi re-iniciada e pode re-executar normalmente.

Até então foram mostrados os algoritmos e a infraestrutura do modelo de Software Cache Transacional. A seguir, no próximo capítulo, será descrito como esse modelo foi implementado no *Cell BE*, alguns *benchmarks* que foram portados e medidas de desempenho.

Capítulo 5

Resultados Experimentais

A máquina utilizada para o desenvolvimento e execução dos experimentos foi um *PlayStation 3*[1] que possui um processador *Cell BE* rodando a 3.2 GHz com um *Power Processor Unit* (PPE) e seis *Synergistic Processor Units* (SPU)[2].

5.1 Depuração e Otimização

A biblioteca de Software Cache Transacional foi completamente implementada em C e possui 1808 linhas de código (excluindo comentários e linhas em branco). Durante sua fase de desenvolvimento, foi criado um pequeno aplicativo que auxiliou testar e encontrar bugs na memória transacional. Este aplicativo consiste de 3 *threads*.

Uma *thread* altera duas variáveis compartilhadas dentro de uma transação de maneira que as duas variáveis tenham valores relacionados e portanto devem ser coerentes. Por exemplo fazer a soma de dois valores e guardar numa variável e a multiplicação na outra. Uma segunda *thread* faz a mesma coisa mas com valores diferentes. A terceira *thread* apenas lê os valores das variáveis e verifica se os valores estão num estado coerente, seja o estado causado pela primeira ou segunda *thread*.

Este aplicativo foi muito importante para ajudar a encontrar bugs de condição de corrida pois, como era muito simples, ficava mais fácil de analisar a pilha de execução e memória acessada quando um erro ocorria. Porém, para analisar a pilha de execução, por exemplo, não se pôde utilizar depuradores convencionais.

[1] *PlayStation* é uma marca registrada de *Sony Computer Entertainment Inc.*

[2] Do total de 8 SPUs, uma vem travada para aumentar o rendimento da fabricação e a outra é inacessível, reservada para decriptação DRM.

Os depuradores convecionais, como o *GDB*, não possuem mecanismos para auxiliar a depuração de programas que alteram diretamente a pilha ou fazem acessos arbitrários à memória RAM. No caso da memória transacional, a pilha é salva e restaurada numa transação e o uso do `longjmp` altera o ponteiro da pilha. Tudo isso confunde o depurador, sem mencionar as complicações que trazem a própria Software Cache.

De fato, para auxiliar a depuração, foi também necessária a criação de uma estrutura auxiliar que mantinha a lista em ordem temporal de todas as operações realizadas pela Software Cache e pela Memória Transacional. Quando uma operação ilegal acontecia, o programa fazia o despejo da lista de operações realizadas, da pilha de execução e de toda memória alocada pelo programa, com isso era possível reproduzir os passos executados pelo programa e a biblioteca de Software Cache Transacional e encontrar o erro.

Após a fase de implementação e depuração, foram feitas algumas pequenas otimizações no código. A principal delas que vale a pena mencionar é o fato de que o algoritmo de Software Cache Transacional, em nenhum momento, exige que o conjunto de leituras possua multiplicidade 1, ou seja, podem haver elementos repetidos. Num primeiro momento pensou-se que manter o conjunto de leituras menor sem elementos repetidos seria mais eficiente pois economiza espaço e a cópia para a memória externa no momento da efetivação seria menor.

O conjunto de leitura, assim como o de escrita, é mantido em um vetor. Para garantir que não existe elementos repetidos é feita uma busca linear neste vetor. Como a operação de leitura de memória é realizada muitas vezes na execução de qualquer aplicação, qualquer modificação no número de instruções executadas por esta operação influencia muito o tempo de execução da aplicação. Portanto, fazer uma busca linear no vetor não é uma boa opção. Para evitar a busca linear, foi implementado um *bloom-filter* de 128 bits, com duas funções *hash* usando instruções intrínsecas do SPE para melhor desempenho. O *bloom-filter* ajudou a diminuir o número de buscas lineares a serem feitas no vetor, porém um *bloom-filter* de 128 bits satura muito rápido para este tipo de uso e o desempenho geral da aplicação não melhorou muito. Aumentar o número de bits do *bloom-filter* significa também aumentar o número de instruções executadas e, no fim, acabou não valendo a pena. A solução final foi permitir que o conjunto de leituras tenha elementos repetidos que obteve um desempenho até 2 vezes maior do que a versão que utiliza busca linear. O leitor deve estar se perguntando porque não foi utilizada uma outra estrutura como uma árvore ou uma tabela *hash* no lugar de um vetor simples. A resposta é bem simples, a árvore ocuparia pelo menos 3 vezes mais espaço (para guardar os ponteiros) na pequena memória local do SPE. A tabela *hash* teria colisões, usar uma lista ligada para tratar colisões teria o mesmo problema de espaço que a árvore e usar o próximo espaço livre da tabela aumentaria o número de colisões e, no fim, a busca seria linear quase sempre. Lembre-se que o espaço disponível para guardar estes dois conjuntos é muito pequeno.

Além de tudo isso, um segundo tipo de otimização ainda pode ser feita. O SPE possui uma série de instruções especiais para trabalhar com múltiplos dados numa única instrução, podendo realizar cálculos em 4 palavras em uma única instrução. Na biblioteca de Software Cache Transacional existem algumas operações que poderiam tirar proveito

dessas instruções. A diferença de desempenho, porém, que esse tipo de otimização traria, não será relevante para a discussão presente neste capítulo pois todas as comparações foram feitas utilizando a mesma base de código sem essas otimizações.

5.2 *Benchmarks*

Para testar a implementação da Software Cache Transacional e medir o seu desempenho foram utilizados dois conjuntos de *benchmarks*: o *Intset* e o *STAMP*. Considerando a falta de *benchmarks* representativos de casos de uso reais para memórias transacionais, em 2008, um grupo de pesquisadores de Stanford projetou o *benchmark STAMP*. Este *benchmark* consiste em oito aplicativos programados em C que representam casos de uso reais de diferentes domínios como bioinformática, segurança de redes e mineração de dados. Foram escolhidos 5 desses aplicativos para representar o corpo de teste para a implementação da Software Cache Transacional. Estes 5 aplicativos são representativos de casos de uso reais e suficientes para testar o sistema no escopo do trabalho aqui proposto. A descrição, a seguir, de cada aplicativo, foi baseada na descrição dada no artigo original do *STAMP* [22].

1) Intset: Este *benchmark* consiste em um conjunto de números inteiros onde se realizam operações de inserção, remoção e busca. Ele é, na verdade, dividido em três, pois o conjunto pode ser implementado usando uma lista ligada: aqui chamado de *Intset_LL*, uma tabela *hash* (*Intset_HT*) ou uma árvore vermelho-preta (*Intset_RB*). Cada operação no conjunto é realizada dentro de uma transação separadamente e nenhum outro trabalho é realizado além das operações citadas. Como pode-se imaginar, as transações realizadas por esses três *benchmarks* são pequenas e não representam um caso de uso real e, por isso, eles são também chamados de *micro-benchmarks*. Eles são muito úteis para testar o sistema de memória transacional pois têm alta conteção, ou seja, todas as *threads* estão dentro de uma transação a maior parte do tempo e, dependendo do tamanho do conjunto pode haver um grande número de conflitos. Porém, este tipo de *benchmark* não é muito útil para medir o desempenho (mesmo que relativo) do sistema de memória transacional pois, como será mostrado, o tempo de execução e o número de transações abortadas é muito variável, e o desvio padrão das medições é muito alto.

2) Genome: Este aplicativo realiza sequenciamento de DNA que consiste em reconstruir a sequência original de nucleotídeos do DNA a partir de um grande número de segmentos de nucleotídeos. Este aplicativo tem duas fases para realizar esta tarefa. Como existe uma quantidade relativamente grande de segmentos, frequentemente podem haver segmentos duplicados. A primeira fase do algoritmo utiliza um *hash* para criar um conjunto de segmentos únicos. Na segunda fase do algoritmo, cada *thread* tenta remover um segmento do conjunto de segmentos global ainda não processados e adicionar este segmento à sua partição de segmentos encadeados. O algoritmo Rabin-Karp para *string matching* é usado para acelerar o processo de encadeamento. Transações são usadas

nas duas fases da aplicação. Adições ao conjunto de segmentos únicos são protegidas por transações para permitir acesso concorrente e acessos ao conjunto global de segmentos não encadeados são também protegidos por transações, pois as *threads* podem tentar remover o mesmo segmento.

3) Intruder: Este é um sistema de detecção de intrusos baseado em assinatura. Este aplicativo lê pacotes TCP de uma rede e compara com um conjunto de assinaturas de intrusos conhecidos. Pacotes TCP são processados em paralelo e passam por três fases: captura, reconstrução e detecção. A estrutura de dados principal na fase de captura é uma fila *FIFO* simples. A fase de reconstrução utiliza um dicionário (implementado por uma árvore auto-balanceada) que contem listas de pacotes que pertencem à mesma seção. A fase de captura e de reconstrução são, cada uma, protegidas por uma transação.

4) Kmeans: O algoritmo *K-means* agrupa objetos em um espaço N-dimensional em K grupos. Este algoritmo é comumente usado para particionar itens de dados em subconjuntos relacionados. Uma transação protege as alterações do centro do grupo em cada *thread* que ocorre em cada iteração.

5) Labyrinth: Este aplicativo implementa um algoritmo para encontrar os caminhos que levam à saída em um labirinto tridimensional. A principal estrutura de dados é uma grade tridimensional que representa o labirinto. Cada *thread* pega um ponto de início e fim que precisam ser conectados por uma rota dentro do labirinto. O cálculo de cada rota e sua adição à grade global do labirinto são protegidos por uma transação. Um conflito ocorre quando duas *threads* pegam caminhos que se sobrepõe.

6) Vacation: Este aplicativo implementa um sistema de transações on-line que emula um servidor de reservas de viagens (com compra de passagens, reserva de hotéis, etc...). O sistema é implementado com um conjunto de árvores que mantém os dados dos consumidores e suas reservas para vários itens da viagem. Durante a execução, as *threads* cliente realizam um número de seções que interagem com o banco de dados do sistema de viagens. Em particular, existem três tipos de seções: reservas, cancelamentos e alterações. Cada uma das seções clientes é protegida por uma transação para garantir a coerência dos dados no banco de dados.

5.3 Análises

Cada aplitativo foi portado para o modelo de Software Cache Transacional descrito nesta dissertação utilizando o atributo `__ea` com auxílio do compilador. Inicialmente o modelo de Software Cache Transacional foi implementado sem a garantia de Isolamento Forte. A implementação de Isolamento Forte e a análise do seu impacto no desempenho está descrita na seção 5.4.

Para comparação de desempenho nas análises a seguir, foi implementado um modelo equivalente com exatamente a mesma Software Cache, porém, em vez de Memória Transacional, foi utilizado um *mutex* global. No início e fim de uma transação o *mutex* global

é adquirido e liberado. Como os dois modelos são equivalentes, foi possível utilizar extamente o mesmo código das aplicações portadas. O modelo utilizando *mutex* servirá como base para o desempenho mínimo desejado de cada aplicação. A intuição é simples, se o modelo de execução otimista da memória transacional não é capaz de superar a serialização imposta pelo *mutex* então, para aquela aplicação, é melhor utilizar o *mutex*, ou mesmo executar com apenas uma *thread*. Esta é uma comparação justa, porém deve-se levar em conta que a memória transacional em software agrega uma sobrecarga de execução muito grande. Dependendo do tamanho da aplicação sendo executada, a sobrecarga do sistema de memória transacional é maior do que o trabalho útil realizado pela aplicação e no fim esta carga não compensa. De fato, na literatura de memória transacional, a maioria das implementações em software foram feitas visando demonstrar uma prova de conceito e os próprios autores assumem que a sobrecarga é tão grande que inviabiliza a utilização na prática para a maioria das aplicações. Como já foi mencionado, os modelos em Hardware e o Híbrido são os modelos que conseguem obter um melhor desempenho comparados com as implementações usando *mutex*.

Para começar, será mostrada a análise das estruturas internas da Software Cache Transacional. Como o tamanho da *cache*, o bloco de *cache* e o diretório de versionamento influenciam no desempenho de cada uma das aplicações. Em seguida, será mostrada uma análise do compartamento de cada aplicação em relação ao tamanho das transações, tamanho do conjunto de leitura e escrita e como a quantidade de trabalho realizado pela aplicação influencia nestas métricas. Por fim, será feita uma análise comparativa do desempenho de cada aplicação com a versão usando *mutex* e como o desempenho é afetado pelo número de *threads* executando em paralelo. Para fazer a medição do tempo médio de cada transação com precisão, foi utilizado o registrador especial do SPE chamado *Decrementer Register*. Este registrador é constantemente decrementado na taxa de 79.8Mhz no caso da máquina *PS3*. A precisão das medições é, portanto, na casa da unidade de microsegundo [15].

Na maioria dos experimentos descritos a seguir são utilizados dois tipos de entrada para cada aplicativo. Uma entrada que exige a realização de maior quantidade de trabalho, aqui chamada de *Entrada Grande* e uma entrada que realiza menor quantidade de trabalho, aqui chamada de *Entrada Pequena*. As entradas são definidas explicitamente para cada aplicativo, são elas:

Entrada Grande:
kmeans: Conjunto de 1000 linhas, 128 dimensões e 16 centros.
genome: -g 4000 -s 16 -n 500000
intruder: -n 1000 -a 50 -l 1024
intset: -o 10000 -i 5000 -r 10^{17} -u 10
labyrinth: Labirinto de 64x64x1 com 30 rotas.
vacation: -n 4 -r 524288 -t 4096 -u 90 -q 60

Entrada Pequena:
kmeans: Conjunto de 500 linhas, 64 dimensões e 16 centros.
genome: -g 500 -s 16 -n 2000
intruder: -n 100 -a 10 -l 512
intset: -o 10000 -i 500 -r 10^{17} -u 10
labyrinth: Labirinto de 32x32x2 com 6 rotas.
vacation: -n 4 -r 16384 -t 4096 -u 90 -q 60

A definição exata de cada parâmetro de cada aplicativo está descrita no artigo do *STAMP* [22] e o código fonte deste *benchmark* disponível no web-site *http://www.stamp.stanford.edu*. Esses dois conjuntos de entradas serão referenciados em cada análise e, conforme necessário, um novo conjunto de entradas será definido para cada caso.

A menos que seja especificada uma configuração diferente, todos os experimentos mostrados aqui utilizam a seguinte configuração padrão: A *cache* de 4-vias, usando a política de substituição *round-robin*, execução de 6 *threads*, o tamanho do bloco de *cache* usado é 2^4 bytes, o tamanho da *cache* utilizado é 2^4 linhas de 4 vias e o tamanho do diretório de versionamento é 2^{20} entradas. O conjunto de entradas utilizado por padrão é o *Entrada Grande*, a menos que seja especificado. Da maneira como foi implementada a estrutura interna da Software Cache Transacional permite flexibilidade em alguns parâmetros. Os parâmetros analisados a seguir são: Tamanho da *cache*, Tamanho do bloco de *cache* e Tamanho do diretório de versionamento.

5.3.1 Tamanho da Cache

O tamanho da *cache* é definido como uma potência de 2 que dita a quantidade de linhas de 4 vias que a *cache* contem. O tamanho é, oviamente, discretizado em múltiplos de 4 pois está se usando a configuração de 4-vias. Como foi explicado, a *cache* é instalada dentro da memória local de cada SPE junto com outras estruturas necessárias como o conjunto de leituras e escritas. Esta memória local do SPE é muito pequena, limitando o tamanho máximo da *cache*.

A Figura 5.1 mostra como o desempenho de cada aplicativo é influenciado pelo tamanho da *cache*. O eixo y esquerdo do gráfico mede o número de transações efetivadas por segundo, que é um indicador do desempenho da aplicação. O eixo y do lado direito do gráfico mede a quantidade total de transações abortadas. Os pontos circulares azuis são o número de transações por segundo enquanto os pontos triangulares vermelhos, o número total de transações abortadas. Cada ponto no gráfico é a média de 10 execuções e junto de cada ponto no gráfico é mostrada uma barra de erro que é duas vezes o desvio padrão da distribuição das execuções, indicando um intervalo de confiança de 95%.

O principal fato a ser notado é que podemos dividir as aplicações em dois conjuntos com relação à forma com que o desempenho é afetado pelo tamanho da *cache*. O primeiro conjunto corresponde às aplicações *Kmeans*, *Genome*, *Intruder* e *Intset_RB*. Este conjunto

obtém um crescimento do desempenho gradual com o crescimento da *cache* até chegar num ponto máximo e em seguida o desempenho cai. O segundo conjunto corresponde ao *Labyrinth* e ao *Vacation*. Este conjunto obtém um desempenho proporcional ao aumento do tamanho da *cache*. No caso do *Kmeans* o ponto máximo de desempenho alcançado corresponde à aproximadamente 3400 transações por segundo quando a *cache* tem 2^7 linhas de 4 vias. Este ponto, para o *Genome*, *Intruder* e *Intset_RB* é alcançado com 2^3 linhas de 4 vias, cada um alcançando o desempenho de respectivamente: 2800, 27400 e 34200 transações por segundo. No segundo grupo, o *Vacation* obtêm um desempenho de 900 transações por segundo com 2^8 linhas de 4 vias e o *Labyrinth*, 200 transações por segundo com 2^{10} linhas de 4 vias.

O desempenho da aplicação é afetado de duas formas, o primeiro, a aplicação pode executar mais lentamente de forma não relacionada com o número de transações abortadas, o segundo, quanto maior o número de transações abortadas, menor é o número de transações efetivadas por segundo e mais tempo a aplicação vai levar para realizar o trabalho total necessário.

Se o número de transações abortadas fosse o fator decisivo, então as duas métricas seriam anti-simétricas seguindo a intuição de que quanto mais transações abortadas menor a quantidade de transações por segundo. Porém como se pode observar nos gráficos, isto não acontece por exemplo no *Labyrinth* onde o desempenho cresce junto com o número de transações abortadas ou o *Genome* onde o desempenho cai junto com a diminuição do número de transações abortadas.

O que deve ser levado em conta neste experimento é que o parâmetro que está sendo alterado é o tamanho da *cache*. O fator determinante de mudança é a carga extra de trabalho que existe quando aumentamos o tamanho da *cache*. Lembrando que a *cache* é implementada em software, a carga extra de trabalho executada em certas operações é proporcional ao seu tamanho. Por exemplo a operação de descarga da *cache*.

O algoritmo que cada aplicação executa e a forma com a qual é implementado é influenciado de diferentes formas pela *cache*. O aumento no número de transações abortadas em algumas aplicações, por exemplo, pode ser consequência do pequeno aumento do tempo que cada transação leva para ser executada (devido ao aumento da carga em algumas operações na *cache*) aumentando as chances de conflito. Outras aplicações que têm menos contenção podem se beneficiar do aumento da *cache* com o ganho na proximidade dos dados armazenados na *cache* maior. Diversos fatores influenciam, levando cada aplicação a ter um ponto ótimo do tamanho da *cache*.

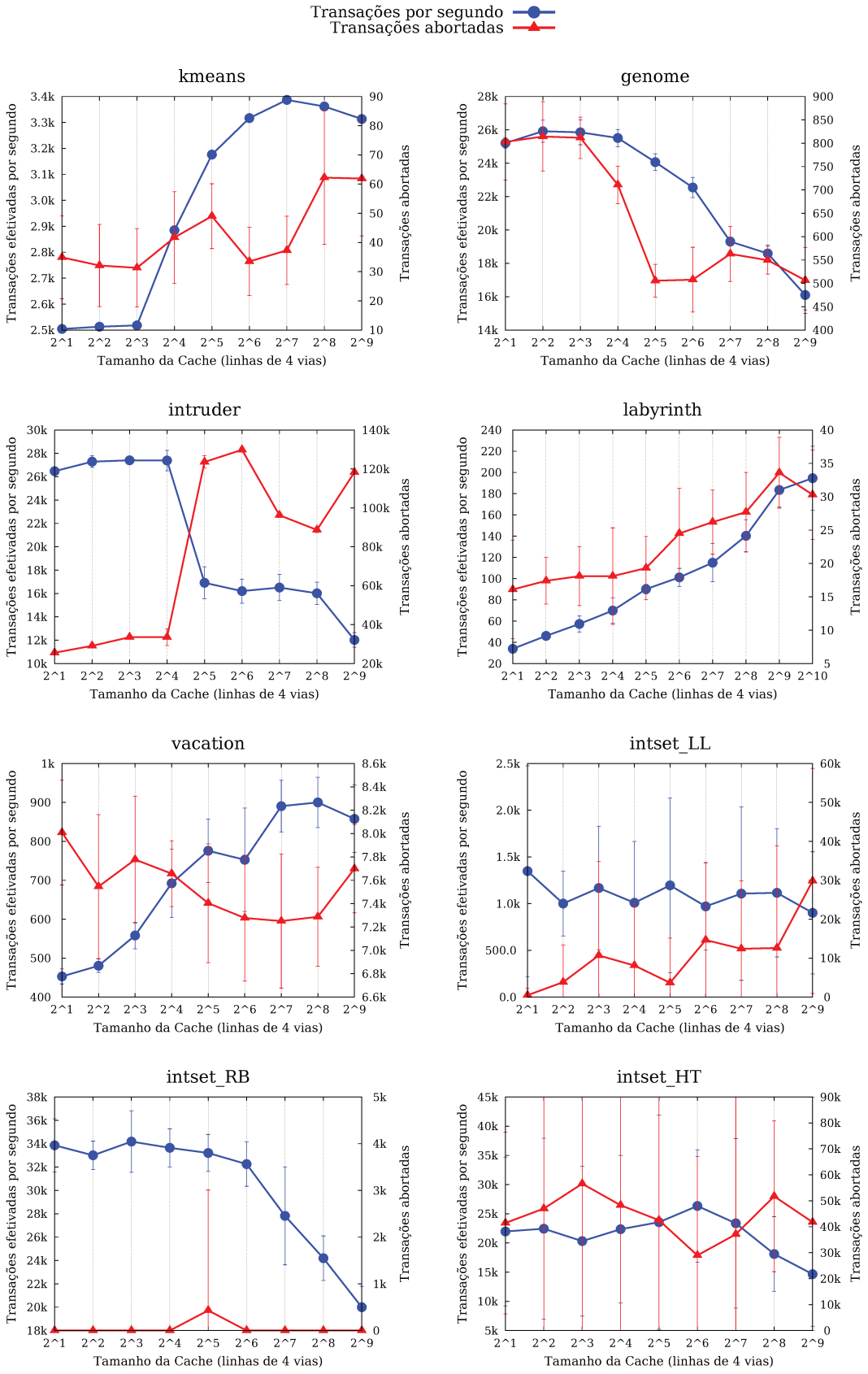


Figura 5.1: Análise do tamanho da *cache*.

5.3.2 Tamanho do bloco de Cache

Ao aumentar o tamanho do bloco de *cache*, mais elementos da memória externa podem ser colocados em cada bloco. Isso beneficia leituras de dados adjacentes e, potencialmente, diminui a quatidade de *miss* compulsório. Portanto, intuitivamente, aumentar o tamanho do bloco de *cache* deveria aumentar o desempenho. Porém, outros fatores influenciam o desempenho. O principal deles é o chamado falso compartilhamento onde duas transações acessam dados diferentes no mesmo bloco de *cache*. Caso uma das transações altere o dado, a outra terá que ser abortada mesmo que o dado alterado não seja extamente o mesmo (no mesmo endereço).

A Figura 5.2 mostra os gráficos de cada aplicação para os vários tamanhos de blocos de *cache*. O tamanho em bytes do bloco de *cache* é definido como uma potência de 2. Veja que agora, diferentemente da análise anterior, a maioria das aplicações segue a intuição de que quanto mais transações abortadas menor a quantidade de transações por segundo e essas duas métricas são anti-simétricas nos gráficos. Este é o fator determinante que faz com que as aplicações *Genome*, *Intset_RB* e *Intset_HT* obtenham um menor desempenho com o aumento do bloco de *cache*. O número de transações abortadas no *Genome* passa de 700, usando o tamanho do bloco de *cache* com 2^4 bytes, para quase 50 mil com 2^8 bytes no bloco de *cache*. A queda de desempenho nesta aplicação chega a mais de duas vezes, caindo de 25400 transações por segundo para aproximadamente 11500. O *Intset_RB* sofre uma queda de desempenho menos acentuada, caindo de 33200 para 28600 aproximadamente, enquanto o número de transações abortadas sobre de zero para 17100. O *Intset_HT*, como o *Genome*, sofre uma queda de duas vezes no desempenho, de 21700 transações por segundo para 9800. O número de transações abortadas dobra, indo de 49600 para 101900.

A aplicação *Vacation* e *Kmeans* conseguem tirar proveito do aumento do bloco de *cache* pois, provavelmente, a maneira como estas aplicações foram implementadas evita o falso compartilhamento do bloco de *cache* até certo ponto. Isso seria possível, por exemplo, se o tamanho da estrutura de dados básica sendo acesada é maior ou igual ao tamanho do bloco de *cache*. O *Vacation* alcança um desempenho máximo de 1095 transações por segundo enquanto o *Kmeans* alcança 4900 utilizando o tamanho máximo do bloco de *cache*.

As aplicações *Intruder* e *Labyrinth* sofreram de um problema ao aumentar o tamanho do bloco de *cache*. Nessas duas aplicações a execução com tamanho do bloco maior que 2^4 geravam resultados incompatíveis com a saída esperada ou geravam erros de execução. Isso se deve ao compartilhamento do bloco de *cache* entre uma *thread* executando uma transação e outra *thread* fora da transação. Até agora está se utilizando uma implementação que não garante Isolamento Forte e, portanto, esta situação gera uma inconsistência que por consequência gera o resultado errado. Para resolver este problema a garantia de Isolamento Forte foi implementada e será discutida na seção 5.4.

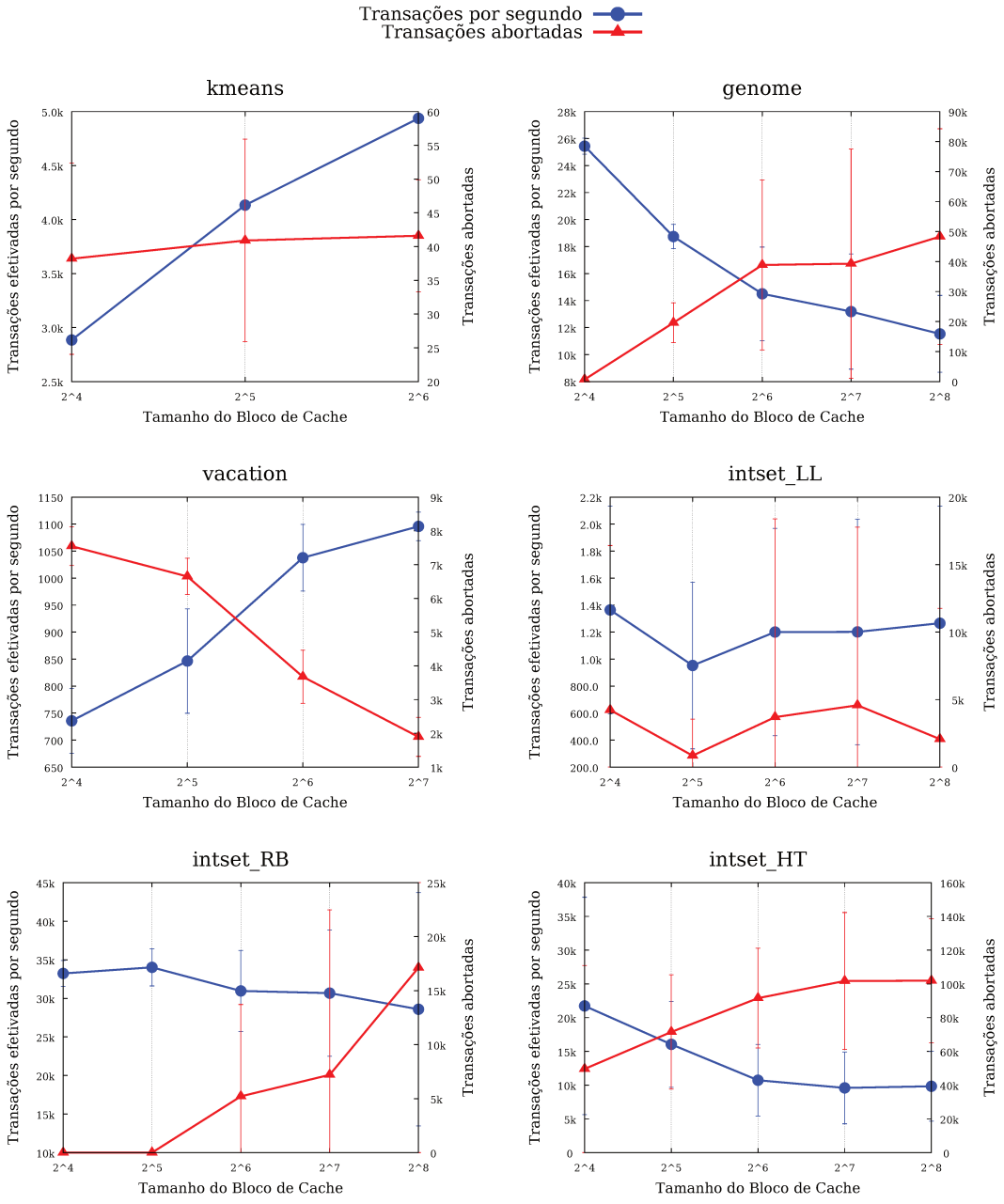


Figura 5.2: Análise do tamanho do bloco de *cache*.

5.3.3 Tamanho do Diretório de Versionamento

O diretório de versionamento foi implementado como uma tabela *hash* onde pode haver colisões e, portanto, mais de um endereço de memória compartilha o mesmo número de versão. Quanto menor é o tamanho do diretório, mais chances de colisão existem. Também é maior a chance de um número de versão ser incrementado por causa de um endereço de memória por uma transação que efetiva enquanto outra transação lê outro endereço de memória que colide no diretório de versionamento com o primeiro endereço efetivado. Esta segunda transação não precisaria abortar pois o conflito existente é falso, decorrente da colisão no diretório de versionamento. Este tipo de falso conflito pode ser evitado aumentando o tamanho do diretório mas o próprio diretório consome memória RAM. Portanto é interessante observar qual é o tamanho mínimo que gera poucos conflitos.

No caso desta análise, é de se esperar que o fator determinante seja o número de transações abortadas causadas pelos falsos conflitos decorrentes do tamanho do diretório de versionamento. Isto é exatamente o que acontece. Na Figura 5.3 pode se observar que, para maioria das aplicações, quanto menor o tamanho do diretório de versionamento, maior o número de conflitos e menor o desempenho.

Para as aplicações *Vacation*, *Intset_LL*, *Intset_RB* e *Intset_HT* o desvio padrão das medições é tão grande que fica difícil dizer qualquer coisa sobre elas. Porém, as aplicações *Kmeans*, *Genome*, *Intruder* e *Labirinth* mostram claramente a tendendência da diminuição do número de transações abortadas com o aumento do tamanho do diretório de versionamento. O número de transações abortadas diminui até um ponto onde estabiliza, este é o ponto que se quer utilizar. No caso do *Genome* e do *Intruder* este ponto é o 2^{18} e do *Kmeans* e *Labyrinth* este ponto é o 2^{16}. Para garantir que o diretório de versionamento será grande o suficiente, será usado o tamanho 2^{20} para todos os experimentos daqui para frente.

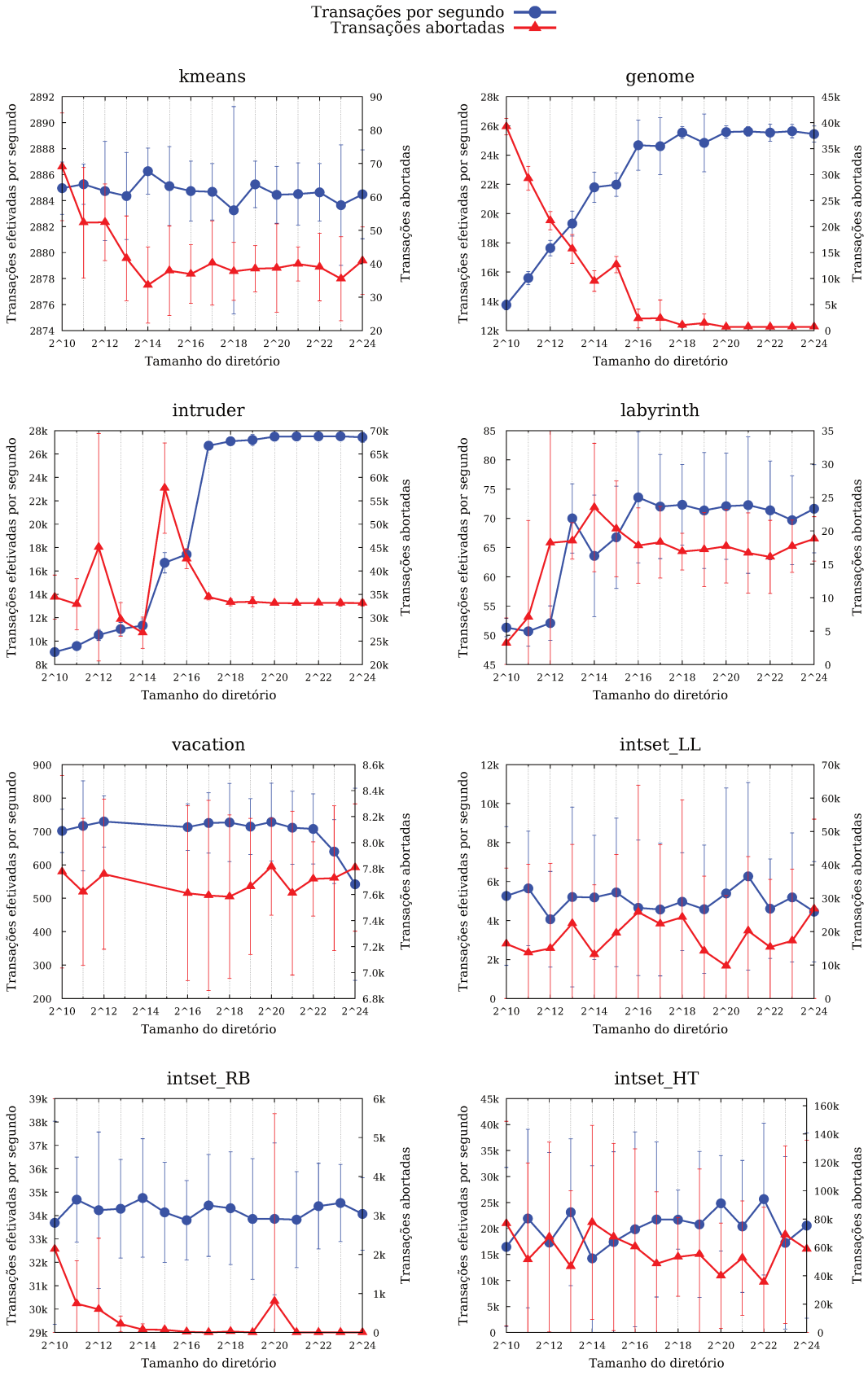


Figura 5.3: Análise do tamanho do diretório de versionamento.

5.3.4 Tamanho da Transação

Para se entender melhor a influência dos diversos parâmetros do sistema de memória transacional e da Software Cache no desempenho de cada aplicação, deve-se observar como cada aplicação interage com esses sistemas. Isso pode ser observado através de algumas métricas como o tempo total gasto em transações, o tamanho médio da transação, e o tamanho dos conjuntos de leitura e escrita. Essas três métricas foram medidas aqui utilizando a configuração padrão já mencionada.

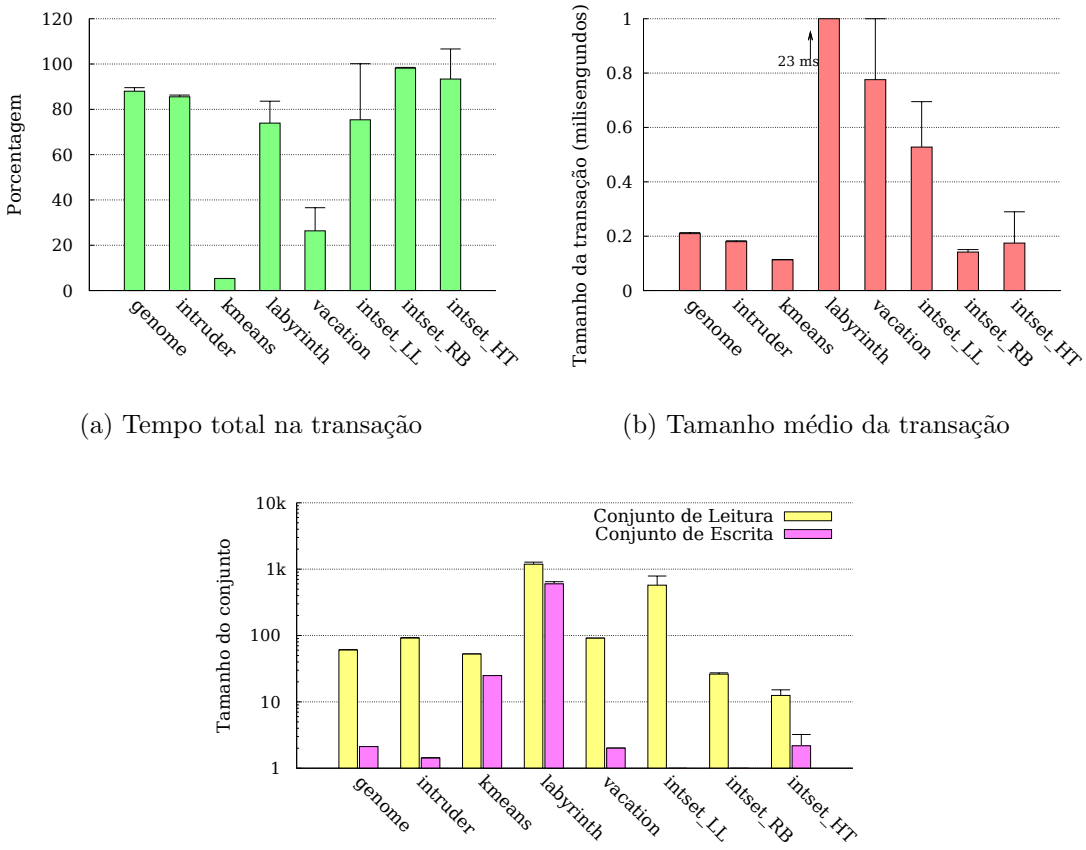
Na Figura 5.4(a) é mostrado o gráfico da porcentagem do tempo total que a aplicação gasta executando dentro de transações, na Figura 5.4(b) é mostrado o gráfico do tamanho médio das transações de cada aplicação e por fim na Figura 5.4(c) é mostrado o tamanho dos conjuntos de leitura e escrita.

O *Genome* e *Intruder* têm comportamentos parecidos com relação à porcentagem do tempo de execução em transações (por volta de 80%) e o tamanho médio da transação que é bastante pequeno flutuando na ordem de 0.2 milissegundos. O tamanho do conjunto de leitura e escrita também são parecidos para essas duas aplicações. Observando o código das duas aplicações, pode-se notar semelhanças que se fazem refletir nestes gráficos: As duas aplicações realizam muitas operações de encadeamento de strings e suas entradas são como um fluxo de dados que pode ser dividido em pedaços menores. O *Kmeans*, por ser um tipo de aplicação diferente, utiliza menos tempo de execução em transações (por volta de 5%) e tem um tamanho médio de transação também menor: por volta de 0.1 milissegundo.

O *Labyrinth* tem um problema sério na forma como foi implementado: em cada transação ele faz a cópia completa da grade que representa o labirinto. Isso faz com que a transação seja enorme devido ao gasto do tempo copiando e o seu conjunto de leituras e escritas é maior ou igual ao tamanho do labirinto. Essa é uma estratégia ineficiente para se utilizar em conjunto com a memória transacional pois a sobrecarga gerada por essa cópia dentro da transação é muito grande. Como se pode observar, o tamanho médio da transação para esta aplicação é muito maior do que qualquer outra, alcançando 23 milissegundos.

No caso do *Vacation*, o tempo gasto por transações é menos de 30% do tempo total da aplicação. Isso se deve à maneira como a aplicação foi implementada e o tipo de entrada utilizada.

O *Intset_LL* utiliza uma lista ligada e ao fazer uma busca, ele precisa percorrer a lista de elementos linearmente. Portanto, o tamanho do conjunto de leituras é maior e, consequentemente, o tempo médio da transação também. Os outros dois *micro-benchmarks* têm um comportamento diferente do *intset_LL* pois utilizam duas estruturas mais eficientes, uma árvore auto-balanceada e um *hash*. A busca nessas duas estruturas é geralmente pequena e o tamanho da transação também.



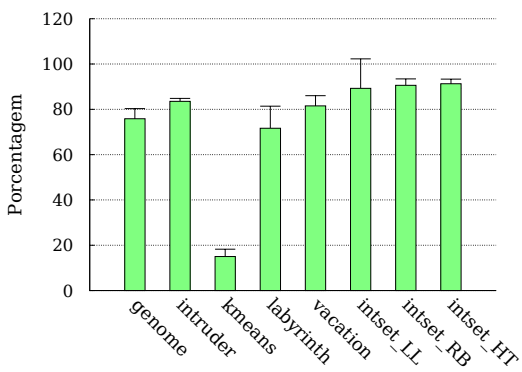
(a) Tempo total na transação

(b) Tamanho médio da transação

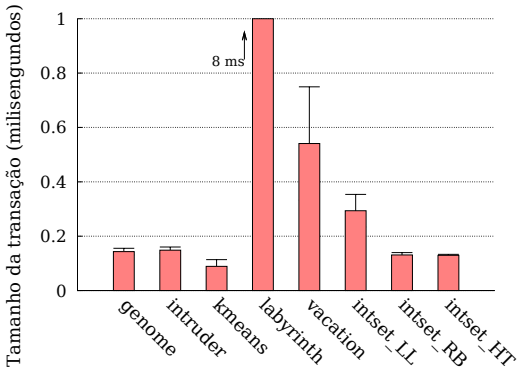
(c) Conjunto de leitura e escrita

Figura 5.4: Análise do tamanho da transação e dos conjuntos de leitura e escrita.

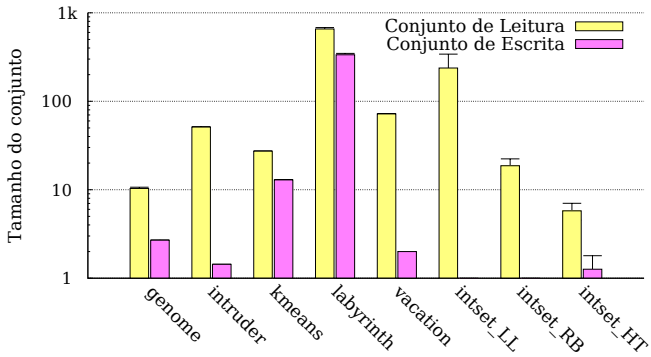
Até agora foi mostrado o comportamento das aplicações para entradas grandes. A Figura 5.5 mostra as mesmas métricas utilizando o conjunto *Entrada Pequena*. Neste caso, o compartamento das aplicações é relativamente parecido, com exceção de que o tamanho médio das transações cai e o tamanho dos conjuntos de leitura e escrita caem aproximadamente uma ordem de grandeza. Um ponto a se destacar é o aplicativo *Vacation* que, neste caso, obtém maior tempo percentual executando dentro de transações. É difícil cogitar o motivo de tal mudança sem a análise completa do algoritmo usado nesta aplicação. Provavelmente a entrada maior exercita mais a execução fora das transações.



(a) Tempo total na transação



(b) Tamanho médio da transação



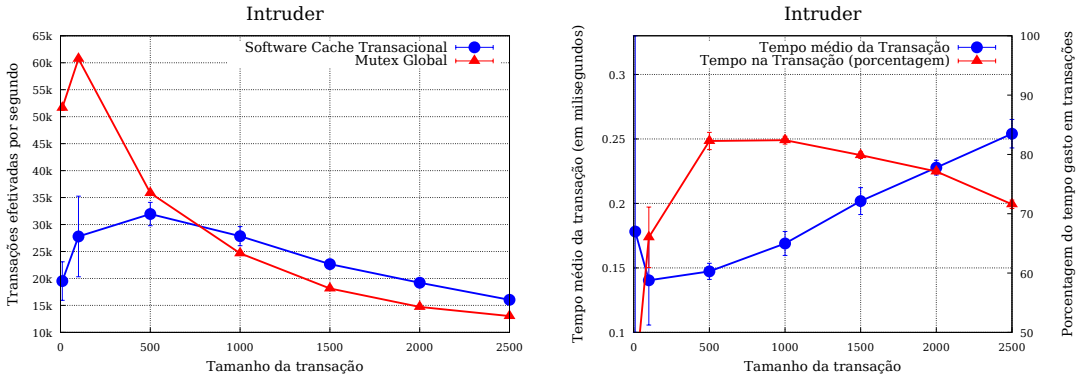
(c) Conjunto de leitura e escrita

Figura 5.5: Análise do tamanho da transação e dos conjuntos de leitura e escrita.

Estes gráficos mostram como cada aplicação difere no quesito porcentagem de tempo gasto na transação, tamanho médio da transação e tamanho dos conjuntos de leitura e escrita para dois tamanhos de entradas diferentes. Porém, uma coisa que não está sendo mostrada é a quantidade de trabalho útil total que está sendo realizado para cada tamanho de entrada e como isso interfere eficiência do sistema de memória transacional.

Quando se está medindo o desempenho de uma aplicação comparando o sistema de memória transacional com uma versão utilizando *mutex* é necessário dar atenção para três fatores:

- A natureza da aplicação
- Quantidade de trabalho realizado pela transação
- Quantidade total de trabalho realizado



(a) (b)

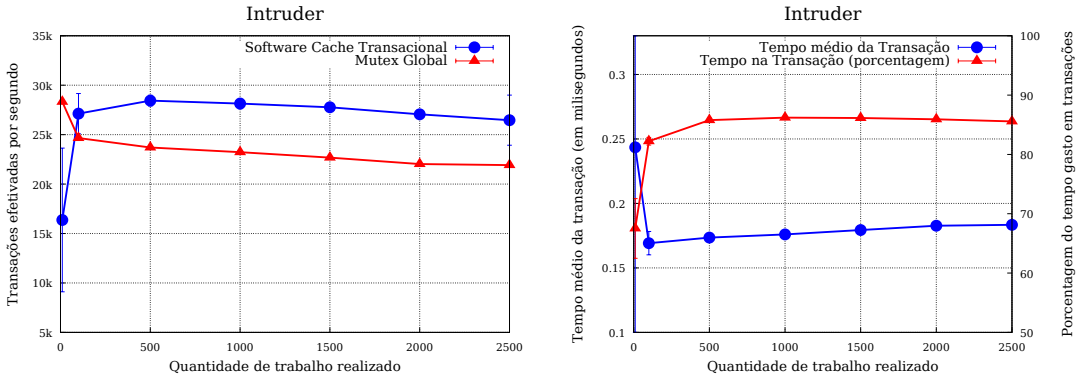
Figura 5.6: Análise do desempenho vs tamanho da transação.

A natureza da aplicação vai ditar se ela é apropriada para execução em paralelo. É fato que algumas aplicações simplesmente não são apropriadas para paralelismo quando a quantidade de contenção é tão grande que se torna mais eficiente executar sequencialmente. Além disso, a natureza da aplicação pode impedir que se faça uma medição apropriada do seu desempenho, tornando difícil a decisão se é melhor ou pior utilizar memória transacional. Este é o caso dos *micro-benchmarks Intset* mostrados aqui. O desvio padrão das medições é tão alto que se torna difícil tirar alguma conclusão útil. Isso é aceitável quando tais *benchmarks* não representam um caso de uso real.

O segundo e terceiro ponto tem um fator em comum. A porcentagem do tempo gasto em transações e o tamanho médio da transação são duas das métricas mais importantes para se tentar estimar quando vale a pena utilizar memória transacional em vez de uma abordagem com *mutex* global. Como foi dito, a memória transacional tem uma sobrecarga de execução fixa por cada transação, se o tamanho da transação é muito pequeno, a sobrecarga de se iniciar e terminar uma transação pode ser maior do que o próprio trabalho realizado dentro da transação. Analogamente, se a quantidade total de trabalho realizado é pequena, a sobrecarga de se executar as operações em paralelo pode ser maior do que realizar o trabalho completamente de forma serial. Ou seja, se existe pouco trabalho a ser feito, não há necessidade de dividir o trabalho.

Para analisar estes dois pontos, foram feitos dois experimentos: um aumentando o tamanho da transação gradualmente e o outro aumentando a quantidade total de trabalho realizado. A aplicação *Intruder* foi a escolhida para este experimento pois os seus parâmetros de entrada permitem controlar facilmente o tamanho da transação (aumentando o tamanho de cada pacote TCP analisado) e a quantidade total de trabalho realizado (aumentando o número de pacotes). Para comparação, foi executada a memória transacional com 6 *threads* e a versão com *mutex* global usando uma *thread*.

A Figura 5.6(a) mostra o gráfico de como o desempenho muda em relação ao tamanho da transação, neste caso, em relação ao parâmetro de entrada que é o tamanho dos



(a) (b)

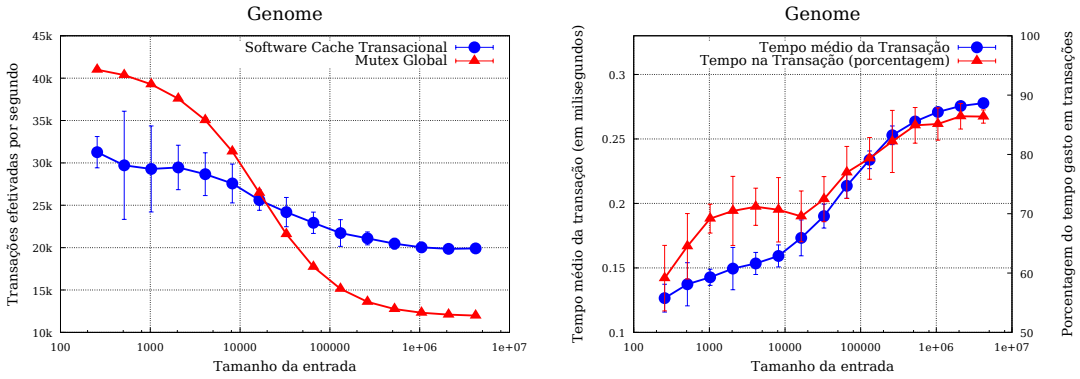
Figura 5.7: Análise do desempenho vs tamanho do trabalho total realizado.

pacotes TCP. Para comparação, a versão com *mutex* utiliza apenas 1 *thread* pois, como será mostrado na análise da seção 5.3.5, a contenção no *mutex* é tão grande que a utilização de 1 *thread* neste caso tem um desempenho melhor do que com mais *threads*. Os outros parâmetros são idênticos à configuração padrão. A versão com *mutex* está representada pelos pontos triangulares vermelhos no gráfico da Figura 5.6(a). A versão com a Software Cache Transacional utiliza 6 *threads* e está representada pelos pontos circulares azuis no gráfico da Figura 5.6(a).

Como pode-se ver, o número de transações efetivadas por segundo cai mais vagarosamente na versão que usa a memória transacional do que na versão com *mutex* global. O ponto de corte, onde vale a pena usar memória transacional, para esta aplicação está entre o tamanho 500 e 1000 bytes para o pacote TCP. A figura 5.6(b) mostra como o tempo médio gasto em cada transação muda com tamanho da transação. Como era de se esperar, neste caso a mudança é linear. Uma curiosidade que não parece intuitiva é que o tempo total gasto nas transações diminui conforme a o tamanho das transações aumenta. Novamente, isto se deve à maneira como o algoritmo dessa aplicação foi implementado e para se ter uma resposta mais precisa do motivo deste comportamento seria necessário analisar a fundo a implementação.

As figuras 5.7(a) e 5.7(b) são semelhantes às analisadas anteriomente. Porém, aqui o que se está mudando é o tamanho do trabalho total realizado. Neste caso é a quantidade total de pacotes TCP processados pela aplicação. O tamanho do pacote utilizado é 1000 bytes, que é o ponto de corte onde vale a pena usar memória transacional segundo a figura 5.6(a) mencionado anteriormente.

Com exceção do primeiro ponto, nesses dois gráficos, o desempenho, o tempo médio da transação e o percentual do tempo gasto na transação permanecem relativamente estáveis conforme a quantidade de trabalho aumenta. Isso era esperado visto que como o tamanho da transação não aumenta, o tempo médio da transação deveria permacer o mesmo assim como todo o resto. Porém, o primeiro ponto do gráfico tem um fator diferente. Este



(a) (b)

Figura 5.8: Análise do desempenho vs tamanho da transação.

ponto corresponde ao tamanho 10, ou seja, 10 pacotes TCP. Com 10 pacotes TCP para processar, a aplicação *Intruder* realiza tão pouco trabalho útil que a sobrecarga imposta pela paralelização e a memória transacional é maior do que o trabalho realizado e portanto é mais eficiente utilizar a execução serial. Por isso neste primeiro ponto o desempenho da versão usando um *mutex* global e uma *thread* ganha.

Para adicionar mais dados a esta análise, o aplicativo *Genome* foi analisado também da mesma forma. Porém, para este aplicativo, não existe uma maneira fácil de controlar o tamanho da transação e a quantidade de trabalho separadamente. O parâmetro da entrada do *Genoma* que foi alterado é a quantidade inicial de segmentos disponível para fazer a reconstrução da cadeia de nucleotídeos original. Desta maneira, foram alterados tanto o tamanho da transação quanto a quantidade total de trabalho.

A Figura 5.8(a) mostra como o desempenho cai conforme o aumento do tamanho da entrada para a aplicação *Genome*. Isso é esperado pois, como foi dito, o aumento do tamanho da entrada afeta o tamanho da transação também. Nos dois casos, da memória transacional e do *mutex* global, o desempenho cai de forma logarítmica. Porém, como se pode ver, o desempenho da versão com memória transacional cai bem mais lentamente e obedece uma curva que difere assintoticamente da curva da versão com *mutex*. O ponto de corte onde as duas linhas se cruzam e onde o desempenho da memória transacional supera o *mutex* está no tamanho de entrada 16384. Uma observação: apesar da linha da memória transacional no gráfico de desempenho parecer uma reta, deve-se notar que o eixo X está em escala logarítmica, logo a reta é na verdade uma curva logarítmica. Analogamente, a figura 5.8(b) mostra o tempo médio e a porcentagem do tempo gasto em transações para a aplicação *Genome*. O tempo médio por transação cresce proporcionalmente com o tamanho da entrada como era de se esperar e a curva obedece uma função logarítmica. Porém, a curva do percentual de tempo gasto em transações segue um comportamento curioso. O percentual do tempo gasto em transações cresce rapidamente no início e em seguida tende a cair novamente quando volta a crescer bruscamente e segue quase

exatamente o crescimento da curva do tempo médio da transação. O interessante é que o ponto onde essa mudança brusca acontece é extamente o mesmo ponto de corte do desempenho, onde passa a valer a pena a utilização de memória transacional: no tamanho de entrada 16384.

O leitor deve estar se perguntando porque, para comparação, foi utilizada uma versão com *mutex* global se estava executando com apenas uma *thread*. A comparação foi feita desta maneira para fins de coerência com as análises feitas na próxima seção onde se observará como o desempenho escala com o número de *threads*. Quando executando com mais de uma *thread*, a versão com *mutex* pode gerar muita contenção devido ao uso intenso do barramento e soma-se a isso a serialização durante região crítica. Tudo isso faz com que o desempenho com mais de uma *thread* muitas vezes seja pior do que com apenas uma. Por isso foi escolhido fazer estas comparações da versão com *mutex* com uma *thread* versus a versão com memória transacional com 6 *threads*. Uma análise mais profunda do desempenho com várias *threads* será feita na próxima seção .

5.3.5 Número de *Threads*

Até agora foi discutido como os fatores da Software Cache Transacional e os fatores da própria aplicação interferem no desempenho. Aqui será discutido como o desempenho escala com o número de *threads*. A figura 5.9 mostra o desempenho que cada aplicação consegue obter com o aumento no número de *threads*. Os pontos triangulares vermelhos são a versão utilizando *mutex* e os pontos circulares azuis utilizam a Software Cache Trasacional. O número máximo de *threads* usado foi seis pois a máquina *PS3* utilizada só tem 6 SPEs. Nos dois casos foi utilizado a configuração padrão dos parâmetros, alterando apenas o número de *threads*.

Os aplicativos *Genome* e *Intruder* escalam muito bem, conseguindo tirar proveito do paralelismo no sistema de memória transacional superando o *mutex*. O *Intruder* obedece uma curva de crescimento rápida mas tem uma queda no último ponto do gráfico com 6 *threads*. Isso decorre do fato de que com o maior número de *threads* maior é a chance de colisão e necessidade de abortar transações. A partir de um ponto, o número de colisões não compensa o ganho de velocidade de execução de transações (considerando que nem todas são efetivadas por causa das colisões). Portanto, existe um ponto ótimo no número de *threads* a ser executada para cada aplicação.

O caso do *Kmeans* é curioso, ele escala muito bem, aumentando o desempenho proporcionalmente ao número de *threads*. É possível cogitar que a forma que ele foi implementado é semelhante a um esquema com granularidade muito fina das seções críticas. Isso é benéfico tanto para a versão com *mutex* quanto para a versão com transações. Porém vale ressaltar, como já foi discutido, que a implementação usando granularidade fina das seções críticas nem sempre é uma tarefa fácil para o programador. Deve-se notar também que a versão com *mutex* ficou ligeiramente mais veloz do que a com transações. Isso se deve à carga extra de execução nas quais as transações incorrem.

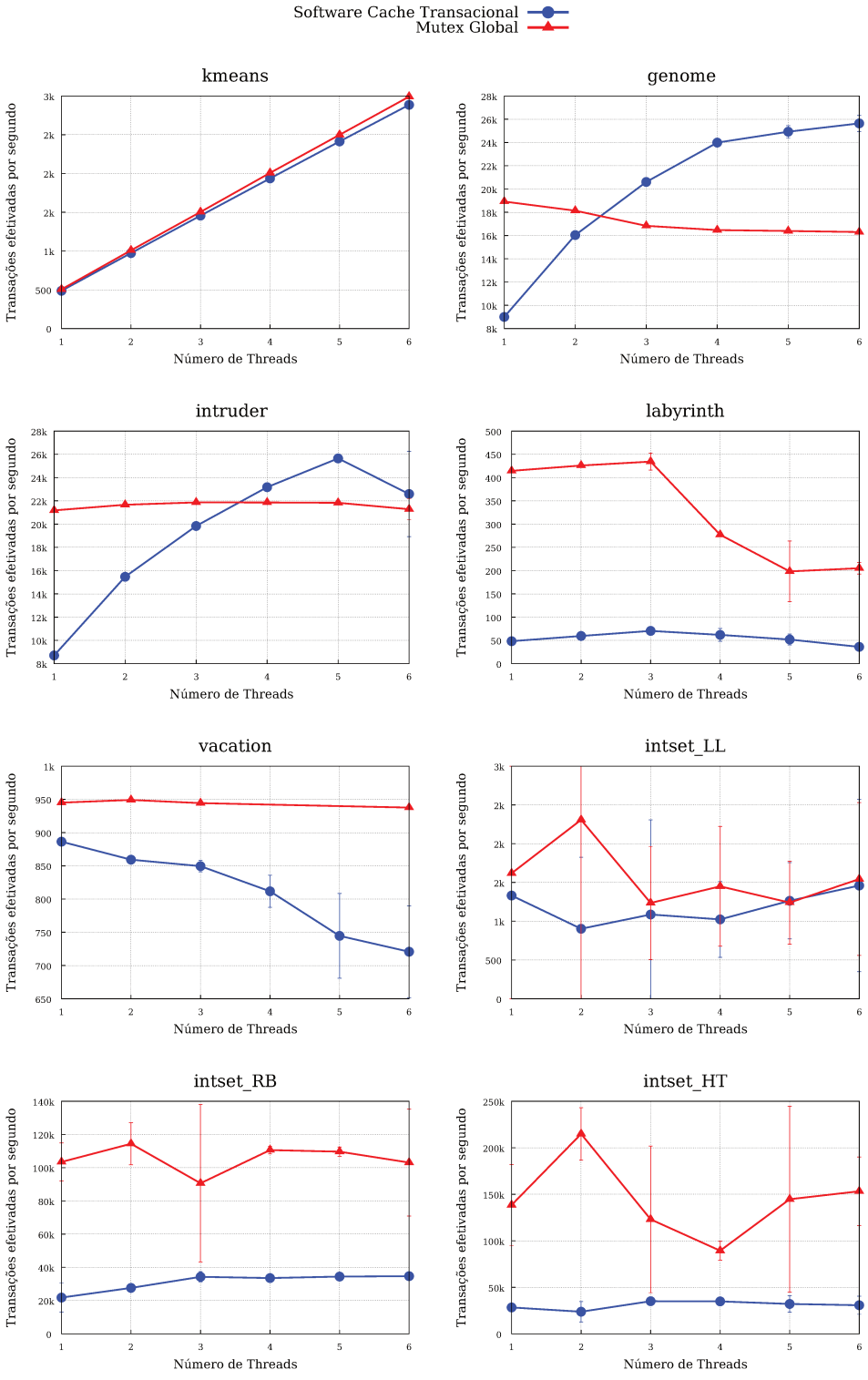


Figura 5.9: Análise do número de *Threads*.

As aplicações *Vacation*, *Labyrinth* e *Intset* não escalam bem usando a Software Cache Transacional e não conseguem superar a versão com *mutex*. Cada uma dessas aplicações têm pequenos problemas. O *Labyrinth* foi implementado de maneira que ele faz a cópia completa da grade do labirinto em cada transação. Esta é uma operação dispendiosa e implica que a maioria das transações vão abortar devido à leitura completa da grade onde outras transações podem gravar causando conflito. O *Vacation* tem uma contenção muito grande. A quantidade de transações abortadas é muito grande em relação à quantidade total de transações. O *Intset* executa muito pouco trabalho real dentro de uma transação (inserção de um elemento no conjunto).

Até agora falou-se muito no impacto da sobregarga de trabalho adicionada pelo próprio sistema da Software Cache Transacional mas não foi estimado a proporção deste impacto e nem em que parte do sistema a sobrecarga é maior. As diferenças principais entre utilizar o *mutex* ou a memória transacional estão nas operações:

- Inciar a transação.
- Finalizar a transação - que significa efetivar ou abortar a transação.
- Realizar uma leitura transacional - que corresponde ao *Fetch* transacional.
- Realizar uma escrita transacional - que corresponde ao *Writeback* transacional.

A sobrecarga devida ao sistema de memória transacional deve estar presente em uma ou mais dessas quatro operações. Portanto, foi feito uma outra análise, agora medindo o tempo gasto em cada uma dessas operações. Estas medições também utilizaram o registrador *Decrementer Register* do SPE para maior precisão. A Figura 5.10 mostra o gráfico com a porcentagem do tempo total de execução para cada uma dessas operações e também o trabalho realizado pela aplicação propriamente dita.

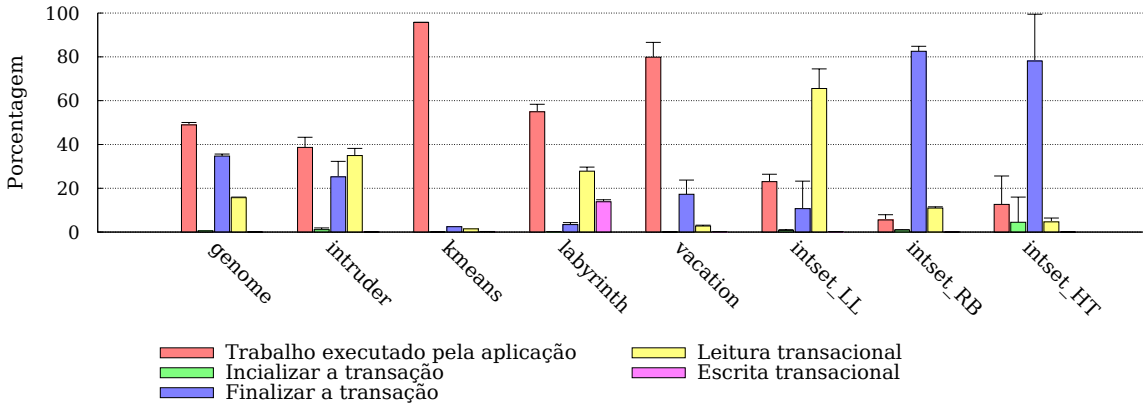


Figura 5.10: Sobrecarga devida à memória transacional.

Os aplicativos *Genome* e *Intruder* têm um comportamento parecido como era de se esperar pois, como já foi dito, os algoritmos das duas aplicações realizam operações pa-

recidas. Pode-se notar que uma boa parte do tempo de execução das aplicações é gasto realizando leituras transacionais ou finalizando a transação.

O aplicativo *Kmeans* passa quase a totalidade do seu tempo executando código da própria aplicação. Isso explica o fato de o desempenho escalar tão bem com o número de *threads*.

Os aplicativos *Intset_RB* e *Intset_HT* sofrem muito pela sobrecarga causada pela finalização da transação. Nestes dois casos, o tempo realizando trabalho da aplicação efetivamente é em torno de apenas 10% contra 80% do tempo gasto na finalização da transação (provavelmente realizando a efetivação). Isso é consequência do fato de que estas aplicações realizam muito pouca tarefa dentro da transação. O *Intset_LL* gasta a maior parte do tempo na leitura transacional, como já foi dito, esta aplicação faz muitas leituras para cada busca de elemento na lista ligada.

O *Labyrinth* gasta tempo tanto na leitura quanto na escrita transacional. Isso é uma consequência da cópia do labirinto que ele realiza na transação. Apesar disso, o trabalho realizado pela aplicação representa uma boa porcentagem do tempo total. Deve-se notar, porém, que esta métrica inclui tanto o trabalho pelas transações que foram efetivadas quanto pelas que foram abortadas. Neste caso, a hipótese é de que a maioria das transações são abortadas devido à contenção causada pela cópia do labirinto.

O *Vacation* tem um comportamento parecido, boa parte da execução se dá por trabalho realizado pela aplicação em si e acredita-se que a maior parte das transações é abortada. Uma grande parte do tempo da aplicação também é gasta por execução fora da transação. Como foi visto, a porcentagem do tempo gasto em transações para essa aplicação é pequena.

Até agora foi possível explicar de forma satisfatória o comportamento de quase todos os aplicativos com exceção do *Vacation* e do *Labyrinth*. Para analisar melhor estes dois e confirmar a hipótese de que uma grande parte das transações são abortadas, foi feita a medição do tempo separadamente nas transações que efetivaram e as transações que abortaram. O percentual do tempo que foi gasto em transações que efetivaram e abortaram para cada aplicação está mostrado no gráfico da Figura 5.11.

Essa é uma métrica mais interessante do que a métrica tradicionalmente usada na literatura que é simplesmente o número de transações que abortam ou efetivam. A proporção do tempo gasto em transações que abortam permite mostrar quanto tempo é desperdiçado pela aplicação realizando trabalho que não será efetivado. Agora, pode-se confirmar a hipótese, como se pode ver na Figura 5.11, as aplicações *Vacation* e *Labyrinth* gastam uma boa porcentagem do tempo total em transações que abortam.

Foi mostrado, até então, que duas causas principais da sobrecarga imposta pelo sistema de memória transacional são a operação de leitura transacional e a finalização da transação. Para melhorar o desempenho geral do sistema, deve-se portanto, melhorar o tempo de execução dessas duas operações.

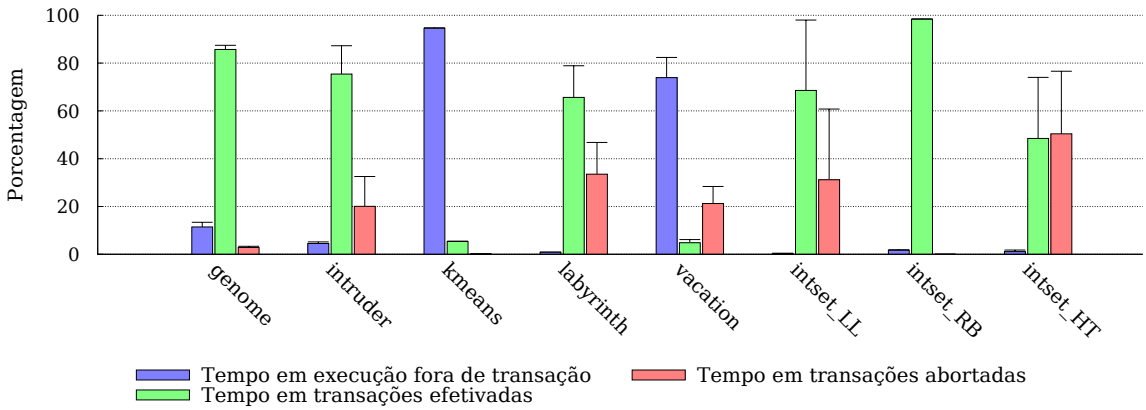


Figura 5.11: Porcentagem de tempo em transações que abortam ou efetivam.

5.4 Isolamento Forte

Como foi visto, a garantia de Isolamento Forte é um requerimento previsto pelo modelo de Software Cache Transacional. Existem diversas maneiras de se implementar o Isolamento Forte com diferentes impactos no desempenho e possivelmente diferentes semânticas. Uma discussão sobre as sutilezas da implementação e semântica do Isolamento Forte pode ser encontrada em [1, 3, 24, 34, 36].

A implementação feita aqui foi a mais ingênua que consiste em simplesmente transformar todas as escritas executadas fora de transação em uma transação pequena cujo único trabalho executado é a escrita na memória. Esta implementação não é a mais eficiente, visto que a carga extra de execução imposta pela transação é muito superior ao trabalho executado por apenas uma escrita em memória. Como já foi visto, não é recomendado ter transações que executam tão pouco trabalho.

Para mostrar como o Isolamento Forte impacta no desempenho das aplicações, foi gerado o gráfico, representado na Figura 5.12. Neste gráfico é mostrado, com uma barra vermelha, o tempo de execução com Isolamento Forte comparado (em porcentagem) com o tempo de execução sem Isolamento Forte. Estas execuções utilizaram a configuração padrão da *cache* e a entrada padrão. O fato de transformar todas as escritas não transacionais em uma transação por si só, faz com que o número total de transações da aplicação aumente. Para observar isto, o gráfico também faz a comparação entre a execução usando garantia de Isolamento Forte com a sem Isolamento Forte no quesito número total de transações executada pela aplicação.

No caso do *Genome Intruder* e *Kmeans* o número total de transações executadas tem um aumento de 50% quando se utiliza Isolamento Forte, comparado com a versão sem Isolamento Forte. As aplicações do *benchmark Intset* têm um aumento de aproximadamente 125%. Já as aplicações *Labyrinth* e *Vacation*, o número de transações totais executadas tem um aumento explosivo e alcança mais de 26 mil % para o *Labyrinth* e 77 mil % para o *Vacation*. Esse aumento explosivo provavelmente se deve à maneira que o algoritmo

dessas aplicações foi implementado, realizando muitas escritas fora da transação.

O aumento no número de transações tem um impacto no desempenho. Este impacto é mostrado aqui como um aumento no tempo de execução que significa uma diminuição no desempenho. No caso do *Intruder*, *Kmeans*, *Labyrint* e *Intset_LL*, o aumento do tempo de execução é de aproximadamente 25%, enquanto as aplicações *Genome*, *Intset_HT* e *Intset_RB*, esse aumento é de aproximadamente 50%. O *Vacation* teve um aumento de mais de 1400% no tempo de execução, provavelmente devido ao aumento de contenção e transações abortando.

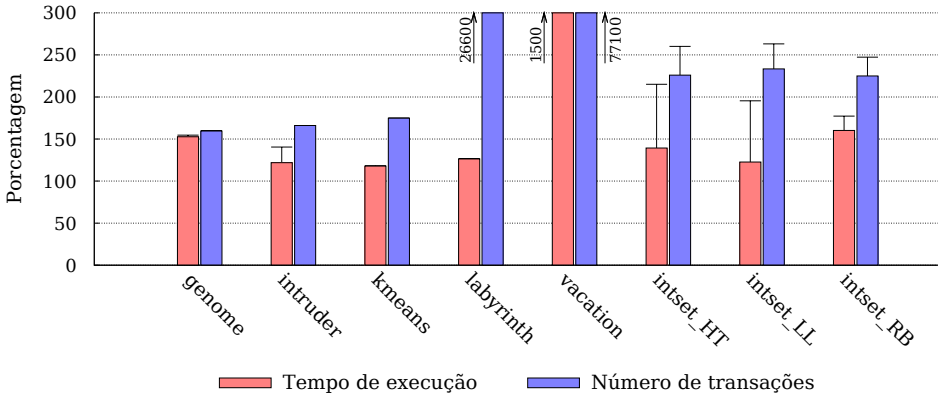


Figura 5.12: Impacto do Isolamento Forte no desempenho.

Agora, com Isolamento Forte implementado, é possível executar as aplicações *Intruder* e *Labyrinth* com tamanhos maiores para o bloco de *cache* como feito na análise da seção 5.3.2. A Figura 5.13 mostra essas duas aplicações com a entrada padrão, a configuração de *cache* padrão e o tamanho do bloco de *cache* variando de 2^4 a 2^8 bytes. O desempenho do *Intruder* cai com o aumento do tamanho do bloco de *cache*, variando de aproximadamente 38 mil transações por segundo no tamanho 2^4 para 14 mil no tamanho 2^8. O *Labyrinth* obtem um crescimento de desempenho crescente como tamanho do bloco de *cache*, passando de aproximadamente 10 mil transações por segundo no tamanho 2^4 para 36 mil no tamanho 2^8.

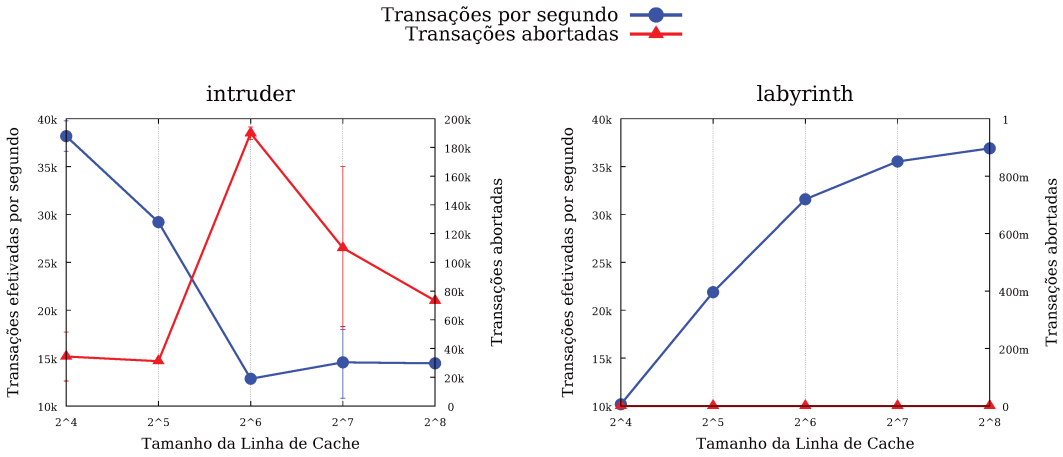


Figura 5.13: Análise do tamanho do bloco de *cache* (com Isolamento Forte).

Capítulo 6

Conclusão

Nesta dissertação foi mostrado que os modelos de programação existentes atualmente para o processador *Cell BE* não lidam bem com o modelo de paralelização que utiliza memória compartilhada. Foram apresentadas os diversos problemas que o programador precisa lidar quando necessita utitilizar memória compartilhada nesta arquitetura. Para resolver os problemas e facilitar a programação, foi proposto um modelo de memória transacional baseado em Software Cache, aqui chamado de Software Cache Transacional. O modelo proposto tira proveito das características únicas do processador *Cell BE*, por ser uma arquitetura híbrida e utilizar uma hierarquia de memórias não convencional.

Para provar a viabilidade deste modelo, foi feita uma implementação usando a linguagem C. O compilador GCC para o *Cell BE* foi alterado de maneira que, ao realizar instrumentação automatica do código, ele utilize a biblioteca de Software Cache Transacional implementada aqui. Foram portados 8 *benchmarks*, que representam casos de uso reais, para medir o desempenho e analisar diversas configurações da Software Cache e da memória transacional. Ao portar estes *benchmarks* para o *Cell BE* usando o modelo proposto, foi feita também uma análise do número de linhas de código total do aplicativo ao usar o modelo de Software Cache Transacional comparado com a utilização de DMAs explicitamente sem memória transacional e software cache.

Foi mostrado que o desempenho varia com a configuração da *cache*, tamanho do diretório de versionamento e da entrada utilizada. Mostrou-se como o desempenho e a eficiência do sistema varia com o tamanho da entrada. Para cada aplicação é mostrado o ponto ótimo da configuração da cache e da memória transacional. O impacto da carga extra de execução da memória transacional não é trivial e foi analisado como isto afeta o desempenho de cada aplicação.

6.1 Contribuições

Este modelo é um avanço na área de memória transacional por dois motivos:

- É o primeiro modelo de memória transacional para arquiteturas híbridas e para o *Cell BE*
- Este modelo propõe uma arquitetura inovadora com a união do conceito de Software Cache com o conceito de memórias transacionais.

Além disso, este trabalho também traz uma contribuição para o desenvolvimento de software no processador *Cell BE*. O modelo proposto é consideravelmente mais simples de se utilizar do que outros modelos de programação existentes para o *Cell BE*.

O trabalho proposto se mostrou um sucesso com suas contribuições resultando na publicação de um artigo em um importante congresso da área (SBAC-PAD 2008). O artigo publicado [12] foi o ganhador do prêmio Julio Salek Aude, melhor artigo da conferência.

6.2 Trabalhos Futuros

Para diminuir este impacto e melhorar o desempenho da Software Cache Trasacional são propostas aqui algumas melhorias na implementação do modelo. As propostas para alcançar essas melhorias ficam como indicações para trabalhos futuros, são elas:

- Transformar um dos SPEs em um servidor dedicado do diretório de versionamento. Um dos maiores custos de se realizar a leitura transacional são as duas validações realizadas no algoritmo 4.3. Como pode-se ver no algoritmo 4.2, a validação necessita fazer uma transação DMA para buscar a versão do endereço a ser lido. O sistema do *Cell BE* permite realizar DMAs entre as memórias locais de dois SPEs em vez da memória principal. Esse tipo de DMA é muito mais rápido do que acessa a memória princial [17]. Com isso, um SPE utilizando sua memória interna para servir o diretório de versionamento pode acelerar esta operação.
- Começar a executar especulativamente uma segunda transação enquanto o PPE está analisando se pode efetivar a primeira. Na fase de efetivação da transação, o SPE pode fazer o DMA do conjunto de leitura e escrita de forma assíncrona e sem esperar a resposta do PPE, ele pode começar a realizar a execução especulativa de uma nova transação. Para isso, o SPE precisa ter uma dupla estrutura para guardar a pilha e o contexto de execução das duas transações. Durante a execução da segunda transação o PPE vai terminar o processo de efetivação da primeira e enviar uma resposta para o SPE usando uma interrupção. Ao ser interrompido, o SPE precisa analisar se a primeira transação foi efetiva ou se necessita abortar. Caso necessite abortar ele aborta as duas transações (primeira e atual) e reinicia a execução da primeira.

Essas duas propostas podem, potencialmente, aumentar o desempenho de aplicações similares ao *Genome*, *Intruder* e *Intset*. Além dessas propostas que visam diminuir a carga extra de execução da memória transacional, outras propostas para trabalhos futuros são sugeridas:

- Como foi explicado, a garantia de Isolamento Forte foi implementada de uma maneira ingênua e pode ser melhorada, focando o desempenho. Por exemplo, em vez de executar uma transação completa a cada escrita, poderia enviar a linha de *cache* suja para o PPE. Este, então, tenta realizar a escrita adquirindo o *mutex* neste bloco de *cache* e validando a leitura deste bloco como numa transação. Porém, caso a validação seja negativa, o próprio PPE pode re-ler o bloco e tentar novamente realizar a escrita sem contactar o SPE, diminuindo a quantidade de comunicação e tempo de espera. Para que isso seja possível, o PPE precisa saber também quais bytes do bloco de *cache* foram alterados (sujos) pelo SPE. Isso pode ser implementado eficientemente por um vetor de bits.

- Mesmo que seja implementada da maneira mais otimizada possível, sempre haverão casos onde a memória transacional tem um desempenho menor do que a execução serial ou a utilização do *mutex* global. Um exemplo desse caso é quando a entrada é muito pequena. Neste caso, poderia haver um sistema capaz de decidir quando utilizar memória transacional, *mutex* ou mesmo a execução serial. Esta decisão poderia ser tomada em tempo de execução e alternar de um modo para o outro conforme a necessidade, garantindo assim um bom desempenho em diversos casos. Um abordagem híbrida como essa, misturando *mutex* com memória transacional pode ser explorada como descrito no artigo [38].

Como visto, esta implementação do modelo proposto ainda tem muito a melhorar para alcançar um desempenho mais competitivo. Contudo, o que importa é a facilidade que este modelo traz para o programador quando comparado com outros modelos existentes para a arquitetura do *Cell BE*.

Referências Bibliográficas

[1] Martin Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pages 185–196, February 2009. ACM Portal.

[2] Jairo Balart, Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Zehra Sura, Tong Chen, Tao Zhang, Kevin O'Brien, and Kathryn O'Brien. A novel asynchronous software cache implementation for the Cell-BE processor. In *20th International Workshop on Languages and Compilers for Parallel Computing*, pages 125–140, October 2008.

[3] Colin Blundell, E. Lewis, and Milo Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, July-December 2006. ACM Portal/IEEE Xplore.

[4] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 81–91, June 2007.

[5] Greg Bronevetsky and Bronis R. de Supinski. Complete formal specification of the openmp memory model. *Int. J. Parallel Program.*, 35(4):335–392, 2007.

[6] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *20th International Symposium on Distributed Computing*, pages 194–208, September 2006.

[7] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Dividing Transactional Memories by Zero. Transact, Salt Lake City, Utah, USA, 23.02.2008, 2008.

[8] Luis Moura e Silva and Rajkumar Buyya. *Parallel Programming Models and Paradigms*, volume 2. Prentice Hall, 1999.

[9] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura,

Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the CELL processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, September 2005.

[10] Pascal Felber, Christof Fetzer, Ulrich Muller, Torvald Riegel, and Martin Susskraut. Transactifying applications using an open compiler framework. In *Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[11] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pages 237–246, February 2008.

[12] F. Goldstein, A. Baldassin, P. Centoducatte, R. Azevedo, and L. A. G. Garcia. A software transactional memory system for an asymmetric processor architecture. In *Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing*, pages 175–182, October/November 2008.

[13] Torbjorn Granlund and the GMP development team. *GNU MP - The GNU Multiple Precision Arithmetic Library.* 5.0.1 edition, 2010.

[14] T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, May/June 2007.

[15] IBM. Software development kit for multicore acceleration version 3.1, programmer's guide, 2008.

[16] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, July 2005.

[17] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, pages 10–23, 2006.

[18] Arun Kumar, Ganapathy Senthilkumar, Murali Krishna, Naresh Jayam, Pallav K. Baruah, Raghunath Sharma, Ashok Srinivasan, and Shakti Kapoor. A buffered-mode mpi implementation for the cell betm processor. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I*, pages 603–610, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, March 2006.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

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

[22] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[23] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[24] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 69–80, San Diego, California, USA, June 2007. ACM Press. ACM Portal.

[25] Michael Mitzenmacher. Compressed bloom filters. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 144–150. IEEE/ACM, 2001.

[26] Gordon E. Moore. Cramming more components onto integrated circuits. In *Readings in computer architecture*, pages 56–59, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[27] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. IEEE Computer Society, 2006.

[28] K. O'Brien, K. O'Brie, Z. Sura, T. Chen, and Tao Zhang. Support OpenMP on Cell. In *International Workshop on OpenMP (2007)*, 2007.

[29] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–34, September 2005.

[30] Roger Penrose. *The Emperor's New Mind*. Oxford University Press, 1989.

[31] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[32] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM Press, 2006.

[33] Paulo Centoducatte Sandro Rigo and Alexandro Baldassin. Memorias transacionais: Uma nova alternativa para programacao concorrente. In *VIII Workshop on High Performance Computing Systems (WSCAD'07)*, 2007.

[34] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 181–194, October 2008.

[35] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[36] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, San Diego, California, USA, June 2007. ACM Press. ACM Portal.

[37] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14:473–530, 1982.

[38] Michael F. Spear, Arrvindh Shriraman, Luke Daless, and Michael L. Scott. Transactional mutex locks. In *SIGPLAN Workshop on Transactional Computing*.

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

[40] Craig Zilles and David H. Flint. Challenges to providing performance isolation in transactional memories. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.