Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Alexandro José Baldassin e aprovada pela Banca Examinadora.

Campinas, 18 de Dezembro de 2009.

Prof. Dr. Paulo Cesar Centoducatte
(Orientador)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Crisllene Queiroz Custódio – CRB8 / 7966

Baldassin, Alexandro José
B19e Explorando memória transacional em software nos contextos de arquiteturas assimétricas, jogos computacionais e consumo de energia / Alexandro José Baldassin -- Campinas, [S.P. : s.n.], 2009.

Orientador : Paulo Cesar Centoducatte
Tese (Doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Memória transacional. 2. Programação paralela (Computação). 3. Arquitetura de computador. 4. Estimativa de potência. I. Centoducatte, Paulo Cesar. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Exploiting software transactional memory in the context of asymmetric architectures, computational games and energy consumption.

Palavras-chave em inglês (Keywords): 1. Transactional memory. 2. Parallel programming (Computer science). 3. Computer architecture. 4. Power estimation.

Área de concentração: Sistemas de Computação

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Paulo Cesar Centoducatte (IC-Unicamp)
Prof. Dr. Wagner Meira Júnior (UFMG)
Prof. Dr. Manoel Eusebio de Lima (UFPE)
Profa. Dra. Islene Calciolari Garcia (IC-Unicamp)
Prof. Dr. Mario Lúcio Côrtes (IC-Unicamp)

Data da defesa: 18/12/2009

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 18 de dezembro de 2009, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Wagner Meira Junior
DCC / UFMG

Prof. Dr. Manoel Eusébio de Lima
CIN / UFPE

Profª. Drª. Islene Calciolari Garcia
IC / UNICAMP

Prof. Dr. Mario Lúcio Côrtes
IC / UNICAMP

Prof. Dr. Paulo Cesar Centoducatte
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia

Alexandro José Baldassin[1]

18 de Dezembro de 2009

Banca Examinadora:

- Prof. Dr. Paulo Cesar Centoducatte (Orientador)
- Prof. Dr. Wagner Meira Jr.
  Departamento de Ciência da Computação – UFMG
- Prof. Dr. Manoel Eusebio de Lima
  Centro de Informática – UFPE
- Profª. Drª. Islene Calciolari Garcia
  Instituto de Computação – UNICAMP
- Prof. Dr. Mario Lúcio Côrtes
  Instituto de Computação – UNICAMP
- Prof. Dr. Alberto Ferreira de Souza (Suplente)
  Departamento de Informática – UFES
- Prof. Dr. Ricardo Pannain (Suplente)
  Instituto de Computação – UNICAMP

[1]Suporte financeiro pelo CNPq sob processo 141238/2005-4.

Resumo

A adoção dos microprocessadores com múltiplos núcleos de execução pela indústria semicondutora tem criado uma crescente necessidade por novas linguagens, metodologias e ferramentas que tornem o desenvolvimento de sistemas concorrentes mais rápido, eficiente e acessível aos programadores de todos os níveis. Uma das principais dificuldades em programação concorrente com memória compartilhada é garantir a correta sincronização do código, evitando assim condições de corrida que podem levar o sistema a um estado inconsistente.

A sincronização tem sido tradicionalmente realizada através de métodos baseados em *travas*, reconhecidos amplamente por serem de difícil uso e pelas anomalias causadas. Um novo mecanismo, conhecido como *memória transacional* (TM), tem sido alvo de muita pesquisa recentemente e promete simplificar o processo de sincronização, além de possibilitar maior oportunidade para extração de paralelismo e consequente desempenho.

O cerne desta tese é formado por três trabalhos desenvolvidos no contexto dos sistemas de memória transacional em software (STM). Primeiramente, apresentamos uma implementação de STM para processadores assimétricos, usando a arquitetura Cell/B.E. como foco. Como principal resultado, constatamos que o uso de sistemas transacionais em arquiteturas assimétricas também é promissor, principalmente pelo fator escalabilidade. No segundo trabalho, adotamos uma abordagem diferente e sugerimos um sistema de STM especialmente voltado para o domínio de jogos computacionais. O principal motivo que nos levou nesta direção é o baixo desempenho das implementações atuais de STM. Um estudo de caso conduzido a partir de um jogo complexo mostra a eficácia do sistema proposto. Finalmente, apresentamos pela primeira vez uma caracterização do consumo de energia de um sistema de STM considerado estado da arte. Além da caracterização, também propomos uma técnica para redução do consumo em casos de alta contenção. Resultados obtidos a partir dessa técnica revelam ganhos de até 87% no consumo de energia.

Abstract

The shift towards multicore processors taken by the semiconductor industry has initiated an era in which new languages, methodologies and tools are of paramount importance to the development of efficient concurrent systems that can be built in a timely way by all kinds of programmers. One of the main obstacles faced by programmers when dealing with shared memory programming concerns the use of synchronization mechanisms so as to avoid race conditions that could possibly lead the system to an inconsistent state.

Synchronization has been traditionally achieved by means of *locks* (or variations thereof), widely known by their anomalies and *hard-to-get-it-right* facets. A new mechanism, known as *transactional memory* (TM), has recently been the focus of a lot of research and shows potential to simplify code synchronization as well as delivering more parallelism and, therefore, better performance.

This thesis presents three works focused on different aspects of software transactional memory (STM) systems. Firstly, we show an STM implementation for asymmetric processors, focusing on the architecture of Cell/B.E.. As an important result, we find out that memory transactions are indeed promising for asymmetric architectures, specially due to their scalability. Secondly, we take a different approach to STM implementation by devising a system specially targeted at computer games. The decision was guided by poor performance figures usually seen on current STM implementations. We also conduct a case study using a complex game that effectively shows the system's efficiency. Finally, we present the energy consumption characterization of a state-of-the-art STM for the first time. Based on the observed characterization, we also propose a technique aimed at reducing energy consumption in highly contended scenarios. Our results show that the technique is indeed effective in such cases, improving the energy consumption by up to 87%.

Agradecimentos

Confesso ter postergado esta seção para o último instante no terrível engano que seria mais fácil escrevê-la. Só então tomei ciência de que sumarizar em uma página todos os que devo agradecer durante cinco anos de trabalho é uma tarefa impossível. Tenho certeza que todos os que participaram da minha vida ao longo deste trabalho saberão reconhecer minha gratidão, mesmo que seus nomes não estejam explicitamente presentes neste pequeno pedaço de papel.

Começo com um agradecimento especial ao meu orientador, Prof. Paulo Centoducatte, que tornou tudo isso possível. Agradeço sinceramente por ter me incentivado a trilhar este rumo há cinco anos atrás.

Aos professores Rodolfo Azevedo, Sandro Rigo e Guido Araújo, por sempre estarem à disposição e proporcionarem discussões sadias e construtivas.

Aos colegas Bruno Cedraz Brandão, Auguto Devegili (brother, where art thou?), Felipe Klein e Patrick Brito pelo convívio do dia-a-dia, seja ele em pensão, *kitnet* ou apartamento.

Aos colegas de laboratório, Bruno Albertini, Daniel Nicácio, Felipe Goldstein, Felipe Klein (olha ele de novo), Leonardo Piga e Yang Yun Ju por propiciarem um excelente e alegre ambiente de trabalho, e também pelas colaborações em alguns projetos.

Aos colegas Alexandre Rademaker, Daniel Lucrédio, Leonardo Oliveira e Matko Botincan pela curta mas frutífera convivência durante nosso estágio na Microsoft Research.

Sou eternamente grato ao Dr. James Larus pela oportunidade de participar de seu grupo de pesquisa na Microsoft Research.

Aos pesquisadores Sebastian Burckhardt, Tom Ball, Patrice Godefroid, Trishul Chilimbi, David Detlefs e Tim Harris pelas discussões e colaborações.

Ao CNPq pelo indispensável apoio financeiro durante quatro anos.

À Microsoft Research pelo apoio financeiro parcial.

Por fim, agradeço minha família por me aturar durante todo este tempo e pelo apoio e confiança no meu trabalho.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Acrônimos

2PL	Two-Phase Locking
API	Application Programming Interface
BEI	Broadband Engine Interface
CAS	Compare and Swap
Cell/B.E.	Cell Broadband Engine
CMP	Chip Multi-Processor
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSTM	Dynamic Software Transactional Memory
DVFS	Dynamic Voltage and Frequency Scaling
EA	Effective Address
EDP	Energy-Delay Product
EIB	Element Interconnect Bus
FPS	Frames Per Second
GNU	GNU's Not Unix
HTM	Hardware Transactional Memory
ILP	Instruction-Level Parallelism
LS	Local Storage
LSA	Local Storage Address
MFC	Memory Flow Controller
MIC	Memory Interface Controller
MVC	Model-View-Controller
ORT	Ownership Record Table
OSTM	Object-based Software Transactional Memory
PPE	PowerPC Processor Element
PPU	PowerPC Processor Unit
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SMC	Software-Managed Cache

SMT	Simultaneous MultiThreading
SPE	Synergistic Processor Element
SPU	Synergistic Processor Unit
SRAM	Static Random Access Memory
STAMP	Stanford Transacional Applications for Multi-Processing
STM	Software Transactional Memory
TCM	Transactional Commit Manager
TL2	Transactional Locking II
TLP	Thread-Level Parallelism
TLS	Thread-Level Speculation
TM	Transactional Memory

Lista de Algoritmos

Capítulo 1

Introdução

"If a problem is not completely understood it is probably best to provide no solution at all."

Um dos princípios de projeto do sistema de janelas X

Por cerca de pelo menos três décadas a indústria de semicondutores garantiu a cada nova geração de microprocessadores sucessivos ganhos em desempenho. Parte devido aos avanços microarquiteturais, e parte em decorrência da tecnologia de fabricação que permitiu frequências de operação mais altas, esse progressivo desempenho se refletiu diretamente no software: novo processador era sinônimo de execução mais rápida do software.

O panorama começou a se alterar no início do século 21, quando limitações na extração de paralelismo no nível de instrução e, principalmente, a elevada dissipação de calor fizeram com que a indústria repensasse seu futuro [95]. A lei de Moore ainda continua válida, ou seja, o número de transistores ainda dobra a cada dois anos, mas as técnicas exploradas nas décadas que se passaram não podem mais ser empregadas com a mesma eficiência. Como consequência, a indústria optou por replicar múltiplos núcleos de execução em um único *chip*, dando origem a era dos processadores *multicore*, *manycore* ou ainda CMP (*Chip Multi-Processor*) [96].

Essa mudança de paradigma tem causado um profundo impacto na indústria de software, já que a grande maioria das aplicações foram desenvolvidas usando o modelo de programação sequencial. Para essas aplicações, os processadores com múltiplos núcleos representam o fim do aumento implícito de desempenho. Extração automática de paralelismo usando compiladores paralelizadores e técnicas como especulação no nível de *threads* (*Thread-Level Speculation* – TLS) não apresentam ganhos significativos para aplicações de propósito geral [99]. Desta forma, para que todo o potencial das arquiteturas multiprocessadas possa ser explorado, o software deve ser explicitamente paralelizado. O grande desafio está portanto em prover formas eficientes e simples para desenvolvimento

de software usando programação concorrente. No entanto, o suporte atual para paralelismo em linguagens e ferramentas de desenvolvimento como depuradores está longe de ser considerado ideal.

O modelo de programação paralela mais difundido se baseia em memória compartilhada. Neste modelo, os acessos concorrentes ao estado compartilhado devem ser sincronizados de modo a evitar condições de corrida e o não-determinismo. As primitivas de sincronização atuais são baseadas em travas e variáveis de condição, amplamente conhecidas por apresentarem uma série de complicações tais como: dificuldade de composição, anomalias no sistema (por exemplo, inversão de prioridade e *deadlock*) e baixo desempenho se não empregadas corretamente [118]. Novas abstrações que facilitem a programação paralela são portanto essenciais.

A busca por métodos mais simples e eficientes para codificação do paralelismo fez ressurgir o interesse por parte de pesquisadores em programação concorrente. Em especial, a ideia de usar transações como mecanismo de sincronização em linguagens convencionais ganhou nova e revigorada força. Esse conceito, largamente conhecido como *memória transacional* (*Transactional Memory* – TM) [71, 49, 70], permite ao programador especificar a sequência de instruções que deve ser executada de forma atômica e isolada, deixando para o sistema de execução a obrigação de implementar eficientemente a sincronização de baixo nível. As principais vantagens do modelo transacional estão no aumento do nível de abstração, no potencial ganho de desempenho e escalabilidade do código, e na praticidade do uso de técnicas já conhecidas em engenharia de software, como a composição de código, no contexto de programação concorrente.

Apesar de promissora, TM ainda é essencialmente tema de pesquisa. Antes de ser considerada como parte efetiva da solução para o problema da concorrência, muitos desafios precisam ser vencidos em várias frentes. Primeiramente, não há um consenso sobre a semântica de execução e construções de linguagem a serem introduzidas. Segundo, as implementações atuais ou são complexas, quando feitas diretamente em hardware (*Hardware Transactional Memory* – HTM), ou não possuem desempenho aceitável, no caso de implementações puras em software (*Software Transactional Memory* – STM) . Uma saída explorada é mesclar ambos sistemas (hardware e software) em uma abordagem híbrida. Finalmente, a experiência com o desenvolvimento de aplicações transacionais ainda é insuficiente para se averiguar de maneira objetiva o ganho concreto conseguido com o modelo transacional. Só recentemente programas complexos e que resolvem problemas reais começam a aparecer, fornecendo uma caracterização mais precisa das implementações de TM existentes.

Os desafios proporcionados por TM têm incentivado ainda mais a pesquisa na área. A Figura 1.1 atesta essa tendência mostrando o número de publicações nas principais conferências durante os últimos anos. Além da academia, empresas como Microsoft, Intel

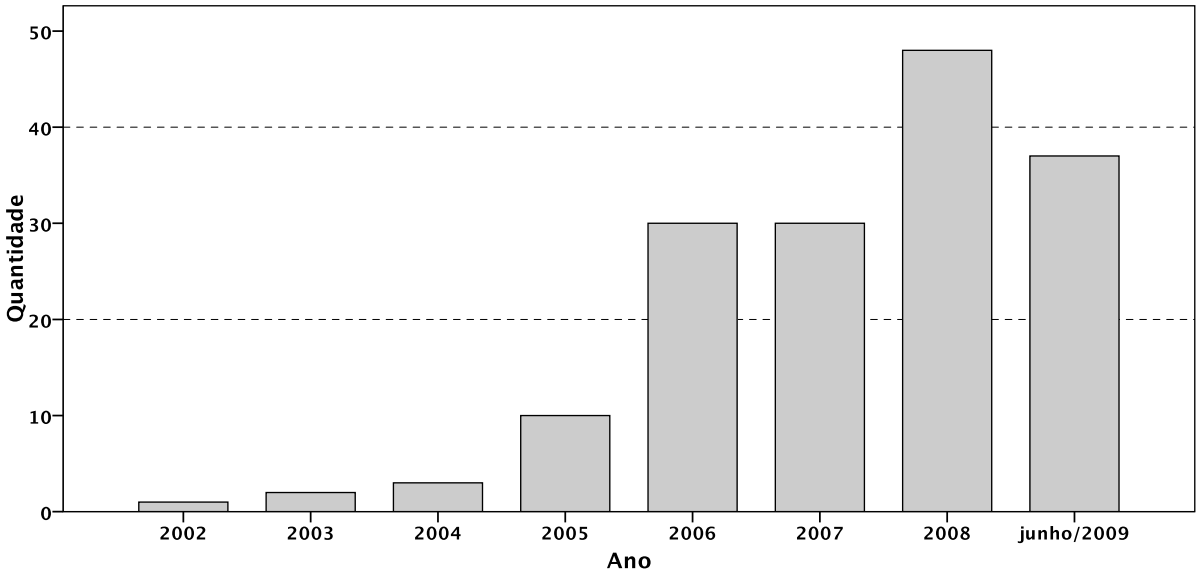

Figura 1.1: Número de publicações sobre TM nos últimos anos

e Sun têm mostrado grande interesse e possuem algum projeto envolvendo TM. O caso mais notório é o da Sun, que programou para 2009 o lançamento de um processador chamado Rock [19], com suporte parcial a transações.

1.1 Foco do trabalho e contribuições

Os tópicos desenvolvidos nesta tese têm como foco principal os sistemas de memória transacional em software (STM). Neste contexto, apresentamos contribuições que exploram a implementação de STM em arquiteturas assimétricas, introduzem técnicas eficientes para o domínio de jogos computacionais e, ainda, mostram a caracterização do consumo de energia para um algoritmo considerado estado da arte de STM.

Mais especificamente, apresentamos os seguintes temas e contribuições:

- **Memória transacional e arquiteturas assimétricas**
  Um sistema transacional para multiprocessadores assimétricos é introduzido, usando como base a arquitetura Cell Broadband Engine [60]. Arquiteturas assimétricas são notórias pela dificuldade que impõem no ciclo de desenvolvimento do software aplicativo e de sistema. Em particular, a adoção do modelo de programação com memória compartilhada necessita que o programador gerencie manualmente os aspectos de transferência de dados entre memória local e memória do sistema. O sistema proposto, denominado CellSTM, estende o conceito de cache gerenciada por software com o modelo transacional e garante uma visão consistente do estado compartilhado

entre os elementos do sistema. Os resultados dos experimentos, conduzidos através de uma série de microaplicações e uma aplicação para sequenciamento de genes chamada Genoma, mostram a escalabilidade do sistema transacional proposto em uma plataforma Cell usando o console Playstation 3. Em particular, um ganho de um pouco mais de 2x foi observado para Genoma quando quatro ou mais elementos de processamento são empregados.

- **Memória transacional e jogos computacionais**
  Este trabalho introduz um modelo transacional voltado para o domínio de jogos computacionais e apresenta uma implementação eficiente do sistema de execução para o modelo proposto. As implementações atuais em software de memória transacional são criticadas pelo baixo desempenho, principalmente quando usadas para codificar aplicações mais complexas. A grande diferença para os sistemas tradicionais é que as transações nunca são canceladas pelo nosso sistema. Quando um conflito é detectado, o sistema de execução transfere a resolução do conflito para o código predeterminado pelo programador. Um estudo de caso usando um jogo complexo mostra a eficácia do sistema proposto, com o qual conseguimos um ganho em desempenho de aproximadamente 2x em uma máquina com quatro núcleos de processamento, apesar de um consumo maior em memória utilizada (cerca de 1,5x).

- **Memória transacional e consumo de energia**
  Os sistemas de memória transacional em software são tradicionalmente avaliados e caracterizados com base somente em desempenho. Este trabalho é o primeiro a mostrar uma caracterização do consumo de energia de uma STM considerada estado da arte. O processo de caracterização foi conduzido através de uma ampla faixa de aplicações, cobrindo diferentes parâmetros transacionais como tamanho de transação, diferentes níveis de contenção, tempo gasto em transação e ainda diversos tamanhos para os conjuntos de escrita e leitura. Com base nos resultados providos pela caracterização, ainda propomos uma técnica para redução do consumo em casos de alta contenção. O emprego da técnica revela um ganho máximo em consumo de energia de até 87% e, no caso médio, de 45%.

Além das contribuições já citadas, e não diretamente discutidos neste texto, o autor também contribuiu com aplicativos e detecção de *bugs* em uma fase inicial do sistema de memória transacional desenvolvido pela Microsoft (atualmente chamado STM.NET). Durante o tempo em que estagiou na sede da empresa em Redmond (EUA), o autor também contribuiu com a transcrição para `C#` da aplicação Genoma para o trabalho de Tim Harris em sistemas de memória transacional com atomicidade forte [2].

1.1.1 Publicações

Esta tese está organizada com base nos seguintes resultados publicados em oficinas, conferências e revistas:

- *On the Energy-Efficiency of Software Transactional Memory.*
  Felipe Klein, Alexandro Baldassin, Paulo Centoducatte, Guido Araujo e Rodolfo Azevedo.
  Proceedings of the 22nd Symposium on Integrated Circuits and Systems Design (SBCCI'09), pg 1–6, Setembro de 2009.

- *Characterizing the Energy Consumption of Software Transactional Memory.*
  Alexandro Baldassin, Felipe Klein, Rodolfo Azevedo, Guido Araujo e Paulo Centoducatte.
  IEEE Computer Architecture Letters (CAL), 8(2):56–59, Julho-Dezembro de 2009.

- *Lightweight Software Transactions for Games.*
  Alexandro Baldassin e Sebastian Burckhardt.
  First USENIX Workshop on Hot Topics in Parallelism (HotPar'09), Março de 2009.

- *A Software Transactional Memory System for an Asymmetric Processor Architecture.*
  Felipe Goldstein, Alexandro Baldassin, Paulo Centoducatte, Rodolfo Azevedo e Leonardo Garcia.
  Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'08), pg 175–182, Outubro de 2008.

O artigo *"A Software Transactional Memory System for an Asymmetric Processor Architecture"* foi agraciado com o prêmio Julio Salek Aude de melhor artigo entre todos os apresentados no SBAC-PAD'08.

O autor também participou da escrita do texto usado em um mini-curso, intitulado *"Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente"*, ministrado pelo Prof. Sandro Rigo no oitavo Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD'07).

1.1.2 Outras publicações

Durante o programa de doutorado os seguintes artigos também foram publicados, mas não são discutidos nesta tese:

- *An Open-Source Binary Utility Generator.*
  Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo, Daniel Casarotto, Luiz Santos, Max Schultz e Olinto Furtado.
  ACM Transactions on Design Automation of Electronic Systems (TODAES), 13(2):1–17, 2008.

- *Automatic Retargeting of Binary Utilities for Embedded Code Generation.*
  Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo, Daniel Casarotto, Luiz Santos, Max Schultz e Olinto Furtado.
  Proceedings of the IEEE Annual Symposium on VLSI (ISVLSI'07), pg 253–258, Maio de 2007.

- *A Flexible Platform Framework for Rapid Transactional Memory Systems Prototyping and Evaluation.*
  Fernando Kronbauer, Alexandro Baldassin, Bruno Albertini, Paulo Centoducatte, Sandro Rigo, Guido Araujo e Rodolfo Azevedo.
  Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP'07), pg 123–129, Maio de 2007.

1.2 Notas gerais

Os termos *concorrência* e *paralelismo* são usados de forma intercambiável neste texto. Da mesma forma são usados os termos *energia* e *potência*.

1.3 Organização do texto

Este texto está organizado em sete capítulos. O Capítulo 2 justifica o ressurgimento e importância da programação concorrente no cenário atual, destacando as tendências de hardware e a precariedade das ferramentas e modelos de programação existentes para desenvolvimento de software paralelo.

O Capítulo 3 introduz o conceito de transações em memória, as principais vantagens e desvantagens e resume o estado da arte da pesquisa. Esse capítulo serve como base para as discussões apresentadas nos que o seguem.

Os próximos três capítulos descrevem os principais trabalhos que formam o cerne desta tese. Esses capítulos possuem a seguinte estrutura em comum: todos são iniciados com uma breve descrição do conteúdo, seguido da motivação. Ao final, uma seção chamada *Epílogo* resume o trabalho e apresenta notas importantes, como por exemplo as contribuições específicas do autor em trabalhos cooperativos. O Capítulo 4 apresenta o sistema

CellSTM para arquiteturas assimétricas, enquanto o Capítulo 5 introduz a STM para o domínio de jogos computacionais e o Capítulo 6 mostra o processo de caracterização energética de uma STM.

Por fim, o Capítulo 7 sumariza as principais contribuições e apresenta o que entendemos como possíveis trabalhos futuros.

Capítulo 2

Paralelismo: Uma Breve Visão Contemporânea

> *"Parallel programming has proven to be a really hard concept. Just because you need a solution doesn't mean you're going to find it."*
>
> David Patterson, 2006

O objetivo deste capítulo é expor as tendências que motivaram o surgimento dos sistemas de memória transacional e sua importância. Visando uma apresentação mais clara, os aspectos de hardware e software são apresentados separadamente.

2.1 Hardware

Em 1965, Gordon Moore especulou que o número de transistores utilizados em circuitos integrados dobraria anualmente. Em 1975 ele alterou sua projeção, estipulando que esse número agora dobraria em intervalos de dois anos [85]. Esta previsão, amplamente conhecida como *lei de Moore*, tem-se mantido até os dias atuais e se tornou a força motora que alavancou o avanço da indústria de semicondutores. A história dos números que mostra a evolução desta indústria é repleta de *exponenciais*. Enquanto o tamanho e o custo relativo do transistor têm decrescido exponencialmente, o desempenho dos microprocessadores tem aumentado da mesma forma. A Figura 2.1 mostra esse aumento para os microprocessadores Intel com um único núcleo de processamento.

Há duas razões principais para o grande aumento do desempenho dos microprocessadores. Primeiro, com o progresso do processo de fabricação os transistores ficaram mais rápidos. Segundo, com o crescente número de transistores, os projetistas conseguiram

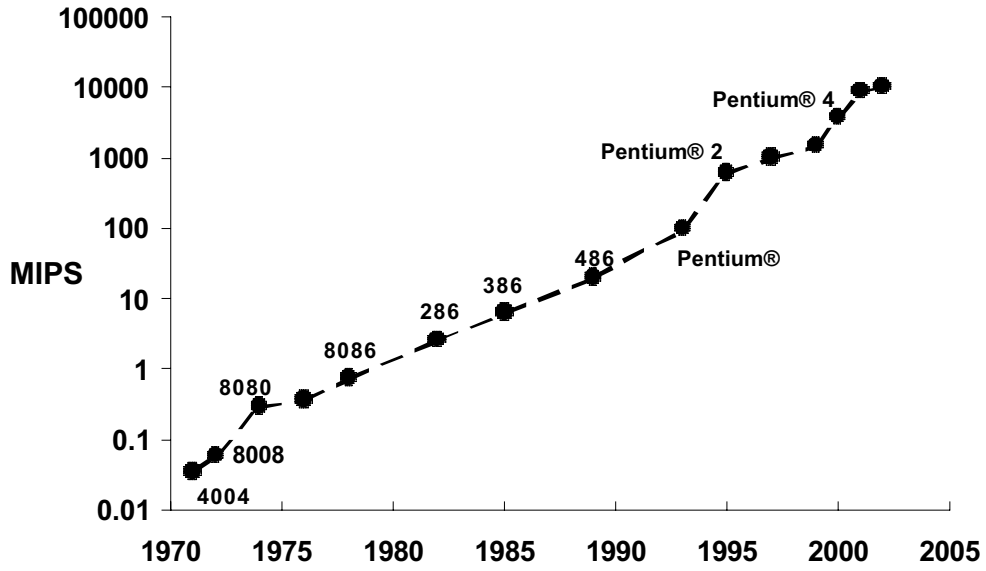

Figura 2.1: Desempenho dos microprocessadores Intel (fonte: [86])

avanços na microarquitetura através da adição de novas funcionalidades. É fato que 32% do aumento total de desempenho na última década é decorrente de inovações na microarquitetura [40].

No começo do século 21, novos fatores arquiteturais e tecnológicos limitaram o ritmo de crescimento do desempenho dos microprocessadores. No lado arquitetural, a exploração do paralelismo no nível de instruções (*Instruction-Level Paralellism* – ILP) parece ter chegado ao seu limite. Pesquisas mostraram que sequências de instruções em aplicações típicas exibem paralelismo reduzido, na ordem de quatro instruções [121]. Logo, microprocessadores superescalares ganham pouco desempenho ao serem capazes de disparar mais do que quatro instruções por ciclo. Da mesma forma, a construção de *pipelines* muito profundos (20 estágios ou mais) fica prejudicada por paradas (*stall*) e esvaziamentos (*flush*) mais frequentes. Além disso, a complexidade e o custo de projeto dos microprocessadores que exploram agressivamente o ILP cresceu. Por exemplo, a complexidade da lógica adicional necessária para encontrar instruções paralelas dinamicamente é, de forma aproximada, proporcional ao quadrado do número de instruções que podem ser disparadas simultaneamente [95]. Esses fatores fizeram com que a contribuição do ILP para o aumento do desempenho decrescesse no começo desta década, como ilustrado pela Figura 2.2.

No lado tecnológico, as limitações são ainda mais contundentes. Até o final da década passada, a acentuada diminuição na dimensão dos transistores implicava frequências de operação mais altas e um número maior de dispositivos que puderam ser empregados na implementação de novas funcionalidades. Embora essa tendência do *mais rápido e em*

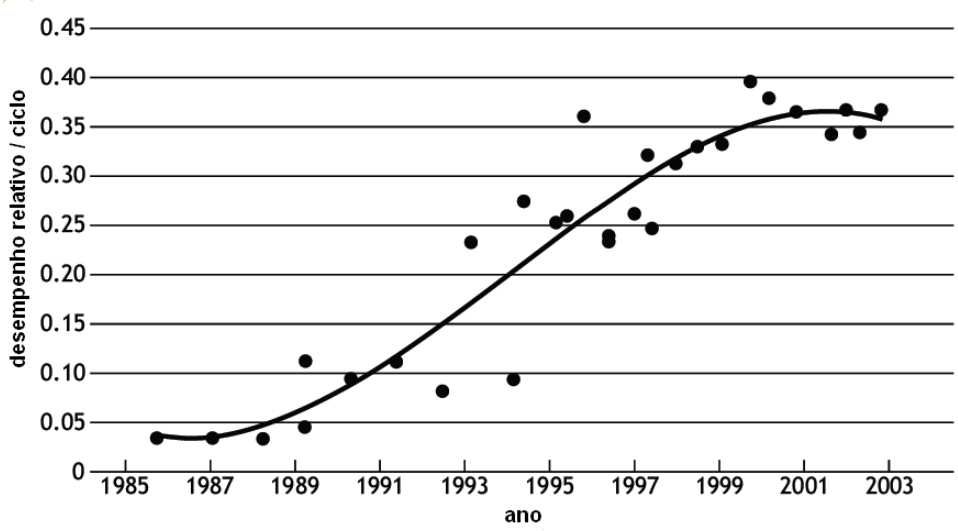

Figura 2.2: Desempenho do ILP (Intel) (fonte: [95])

maior quantidade continue ainda hoje, uma série de efeitos imperceptíveis em gerações passadas começa a aparecer. Pesquisas mostraram que o número de dispositivos que podem ser atingidos em um ciclo de relógio diminui a cada nova geração, já que os fios estão se tornando mais lentos em relação à velocidade de operação dos transistores [5].

Um outro fator limitante, caso a tendência da década passada continue, é a quantidade de potência dissipada pelo processadores. Estimativas indicam que, ao final da década atual, a densidade de potência ficará próxima à da superfície do sol [24]. A potência consumida resulta basicamente do chaveamento dos transistores (potência dinâmica) ou do não chaveamento (potência estática). A maior causa da potência estática é a corrente de fuga, que persiste mesmo com os transistores em operação. Espera-se que, a partir de tecnologias de 60-65nm, as potências estática e dinâmica contribuam igualmente para o consumo total da potência em um microprocessador [92].

Os limites tecnológicos e arquiteturais têm alterado o rumo da indústria de microprocessadores. Só em 2004, a Intel cancelou dois projetos baseados em uniprocessadores devido ao fator potência [37]. Ao invés de dedicar processamento a programas sequenciais, aumentando o consumo de potência e com baixo retorno em desempenho, a nova geração de microprocessadores agora explora o paralelismo no nível de *threads* (*Thread-Level Parallelism* – TLP). A primeira geração desses microprocessadores implementa um mecanismo que permite a execução de múltiplas *threads* simultaneamente (*Simultaneous MultiThreading* – SMT) [120]. Nessa abordagem, um único processador dispara instruções de *threads* diferentes no mesmo ciclo de relógio. Para o software, a impressão é que existem múltiplos processadores, embora estes sejam apenas virtuais. A grande vantagem do SMT é que as unidades funcionais podem ser melhor utilizadas, já que agora um conjunto

maior de instruções (relativamente independentes) pode ser disparado. Além disso, SMT pode esconder latências decorrentes da espera por operações de memória. Como os recursos físicos são compartilhados, a área total requerida não aumenta consideravelmente. Relatos mostram que, em geral, a implementação de SMT aumenta o *throughput* do sistema em pelo menos 30%, com um aumento em torno de 10% da área e um acréscimo no consumo de potência abaixo de 10% [40].

A abordagem atual para fornecer TLP é integrar em um mesmo circuito integrado dois ou mais núcleos de processamento, dando origem aos *processadores multicore* (CMP). Como mais trabalho pode ser feito por ciclo de relógio nessa tecnologia, a frequência de operação pode ser reduzida, diminuindo o consumo total de energia. Além disso, como os núcleos de processamento estão integrados em um mesmo *chip*, o compartilhamento da rede de interconexão pode resultar em redução extra do consumo de potência. A tecnologia baseada em CMP também diminui a complexidade de projeto, já que novas gerações requerem simples replicações de núcleos de processamento e modificações na lógica que interliga os componentes. Com o aumento do número de processadores, o fator limitante passa a ser a velocidade e o tamanho da rede de interconexão.

Com o mercado de microprocessadores investindo suas forças em TLP, a responsabilidade pelo aumento do desempenho agora é principalmente do software. Ou seja, o software deve ser devidamente codificado de forma a explorar adequadamente os processadores com múltiplos núcleos. Essa mudança de paradigma de programação tem atraído a atenção tanto do meio acadêmico quanto da indústria, já que os modelos e ferramentas atuais para programação concorrente são considerados precários [118].

2.2 Software concorrente e memória compartilhada

Programação concorrente é tão antiga quanto a sequencial, mas é inerentemente mais difícil. O modelo de programação concorrente discutido nesta tese é o baseado em memória compartilhada (em oposição ao modelo que emprega passagem de mensagens), já que este é o empregado pelas transações em memória. Neste modelo, a unidade básica de execução é conhecida por *thread*. Um processo pode ser composto por uma ou mais *threads*, sendo que cada *thread* possui contexto de execução próprio, embora compartilhe o espaço de endereçamento do processo. *Threads* são assíncronas, ou seja, possuem diferentes velocidades de processamento e podem, a qualquer instante, serem interrompidas por tempo indeterminado.

Em geral, uma aplicação concorrente inicialmente é composta por uma única *thread*. Eventualmente, e de acordo com a lógica da aplicação, novas *threads* são disparadas. Contanto que o problema sendo resolvido não requeira que diferentes *threads* acessem dados em comum, a computação pode prosseguir de forma elementar. Problemas desse tipo

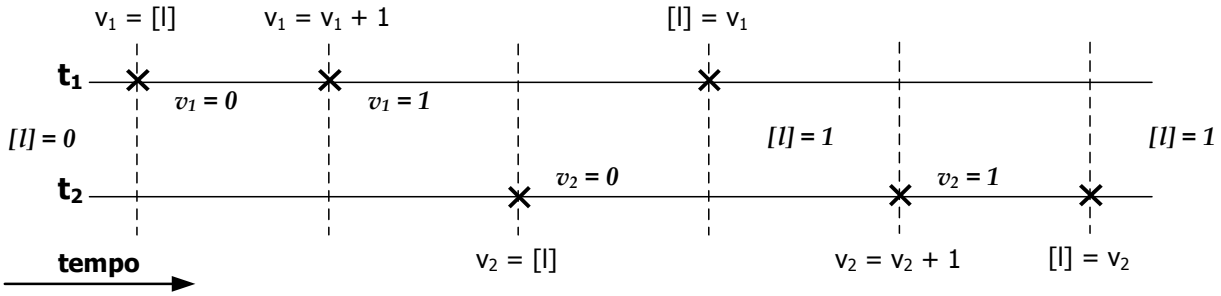

Figura 2.3: Sequência de operações levando a um estado inconsistente

são conhecidos na literatura como embaraçosamente paralelos (*embarrassingly parallel*) e ocorrem com mais frequência em aplicações científicas e gráficas. O grande desafio em programação paralela realmente ocorre quando diferentes *threads* necessitam compartilhar uma mesma região de memória. Neste caso, para impedir que acessos concorrentes gerem inconsistências, são necessários mecanismos de sincronização. Usar corretamente tais mecanismos de forma eficiente tem se mostrado um grande desafio e é assunto da próxima seção.

2.2.1 Sincronização

Acessos concomitantes à mesma região de memória por parte de duas ou mais *threads* podem potencialmente gerar inconsistência se não realizados de maneira correta. Considere o seguinte exemplo: duas *threads* t_1 e t_2 precisam incrementar em um o valor contido em um endereço de memória l, inicialmente zero. Esta tarefa requer pelo menos três operações básicas: (i) leitura do valor em l para uma variável local v; (ii) incremento de v; e (iii) escrita de v novamente em l. Na ausência de qualquer sincronização, é fácil gerar uma ordem de execução na qual o valor final em l após a execução das duas *threads* seja inconsistente, como ilustrado pela Figura 2.3. Note que o valor final correto para l é dois, mas a sequência gerou o valor um.

O problema se deve ao fato de o acesso ao dado compartilhado não ser controlado, gerando uma condição de corrida (*race condition*). O objetivo da sincronização é restringir a ordem em que as operações de diferentes *threads* são executadas, desta forma limitando o número de intercalações possíveis. A sincronização aparece em duas situações: (a) *contenção*, na qual a ação de uma *thread* pode restringir ações das demais; e (b) *cooperação*, na qual a ação de uma *thread* pode habilitar ações das demais. Os exemplos clássicos que ilustram esses dois cenários são, respectivamente, o problema da *exclusão mútua* e o do *produtor-consumidor*, mencionados no trabalho seminal de Dijkstra [29].

De maneira informal, o problema da exclusão mútua consiste em garantir que, na existência de múltiplas *threads*, somente uma obtenha acesso exclusivo a uma determi-

nada região de código, denominada *seção crítica*. Desta forma, *threads* competem pela permissão de acesso a tal seção. Já no problema do produtor-consumidor com *buffer* limitado, *threads* produtoras inserem itens em uma fila compartilhada, enquanto *threads* consumidoras retiram itens da mesma fila. Problemas ocorrem quando uma *thread* produtora tenta inserir um item em uma fila cheia, ou quando uma *thread* consumidora tenta retirar um item de uma fila vazia. A solução envolve coordenação das ações por parte das *threads* envolvidas, geralmente a partir de um protocolo de espera e sinalização.

Um aspecto importante da sincronização refere-se ao desempenho. Como a ordem de execução é restringida, a sincronização potencialmente introduz gargalos seriais na aplicação. A forma mais comum de analisar tal impacto é utilizando a famosa lei de Amdahl [9], dada pela equação 2.1. A equação descreve o *speedup* máximo $S(p)$ alcançado com p processadores em um programa cuja fração de tempo serial é dada por f_s.

$$S(p) = \frac{1}{f_s + \frac{1-f_s}{p}} \tag{2.1}$$

Para um número infinitamente grande de processadores, a lei de Amdahl é descrita pela equação 2.2. Considerando que 50% do tempo de uma aplicação seja de processamento estritamente serial, então, de acordo com a equação 2.2, o *speedup* máximo será 2, independente do número de processadores empregados.

$$\lim_{p\to\infty} S(p) = \frac{1}{f_s} \tag{2.2}$$

A equação 2.2 evidencia a necessidade de se reduzir ao máximo a fração serial do código. Em outras palavras, a granularidade da sincronização é um fator crítico no desenvolvimento de algoritmos concorrentes eficientes.

É pertinente analisar os mecanismos de sincronização de acordo com a garantia de progresso da computação que proveem. Nos mecanismos *não-bloqueantes*, o atraso de uma *thread* não causa atrasos em outras. Já em mecanismos *bloqueantes*, o atraso de qualquer uma das *threads* pode provocar atrasos em outras.

2.2.2 Mecanismos bloqueantes

As primitivas de sincronização bloqueantes são essencialmente baseadas em *travas* (*locks*) e *variáveis de condição* (*condition variables*). O desenvolvimento do conceito de trava aparece em textos da década de 60 [25], enquanto que variáveis de condição aparecem um pouco depois ligadas ao conceito de *monitor* [57]. Outra primitiva famosa é o *semáforo* [29], uma generalização da trava. É importante notar que, em termos de sincronização, essas primitivas constituem o *status quo*, mesmo tendo cerca de 40 anos.

O uso de travas para sincronização é feito da seguinte maneira: (a) uma trava tr é definida para restringir o acesso a uma determinada faixa de memória compartilhada l; (b) antes de acessar l, uma *thread* T requisita posse exclusiva de tr através de uma operação `Lock` (tr) – é garantido a somente uma *thread* êxito na requisição, forçando que as demais aguardem eventual disponibilidade de tr; e (c) após acesso a l, T libera a trava tr através de uma operação `Unlock` (tr). Note que travas resolvem trivialmente o problema da exclusão mútua. É claro que a questão ainda está em como implementar corretamente a operação `Lock` [1]. As implementações atuais fazem uso de instruções capazes de ler, modificar e escrever em uma posição de memória atomicamente.

Pela descrição anterior de travas é possível observar alguns pontos que tornam complicado seu uso, principalmente ao programador inexperiente. Mais especificamente, as seguintes dificuldades ocorrem:

Associação entre trava e dado – não é clara a natureza da relação entre as travas e os dados que devem proteger. É necessário dizer tanto *o que* deve ser sincronizado quanto *como* essa sincronização é feita. O programador deve adotar uma convenção e garantir que essa seja seguida, dificultando a manutenção do código.

Granularidade e desempenho – associar uma única trava com todas as operações em um objeto é uma maneira fácil de garantir corretude mas pode acabar serializando todos os acessos ao objeto. Por exemplo, uma única trava protegendo as operações de inserir e consultar elementos em um lista não permite que diferentes *threads* acessem o objeto concorrentemente, muito embora não seja causada inconsistência na maioria das vezes. Conforme visto, pela lei de Amdahl o desempenho é inversamente proporcional à fração de código serial. Uma solução é usar uma granularidade de travamento mais fina, associando diferentes partes do objeto com diferentes travas (uma trava para cada nodo da lista, por exemplo).

Instabilidade – o travamento fino aumenta a concorrência ao custo da complexidade de codificação. Como várias travas podem ser usadas ao mesmo tempo, a ordem com que elas são adquiridas é extremamente importante. *Threads* diferentes adquirindo duas ou mais travas em ordem inversa podem causar uma dependência de espera circular, um cenário conhecido como *deadlock*. Outras instabilidades incluem: (i) *inversão de prioridade*, em que uma *thread* de baixa prioridade toma posse da trava, sofre preempção pelo sistema operacional e impede que *threads* de alta prioridade prossigam; e (ii) *convoying*, uma generalização do caso de inversão de prioridade - o bloqueio de uma *thread* com posse de trava causa enfileiramento e bloqueio de outras que também dependem da mesma trava.

[1] O livro de Herlihy e Shavit [56] discute diversas soluções e serve como boa referência.

Composição – construir novas operações a partir de outras preexistentes é extremamente difícil com travas. Considere o caso de uma operação que deva mover um item entre duas filas. Em princípio essa operação poderia ser composta através dos métodos de remoção e inserção, sendo requerido atomicidade (ou seja, não pode ser percebida a ausência do item em uma fila antes que o mesmo seja inserido na outra). No entanto, o uso de travas torna complicada tal composição sem que o encapsulamento do objeto seja quebrado ou ainda que novas travas sejam introduzidas.

Alguns casos reais ressaltam a dificuldade de desenvolvimento de sistemas concorrentes. Entre 1985 e 1987, uma condição de corrida existente no software para o equipamento de radiação médico Therac-25 causou pelo menos seis mortes [72]. Em 1997, falhas sucessivas de operação do dispositivo da missão Pathfinder em Marte foram atribuídas a um problema de inversão de prioridade no software [125]. Problemas decorrentes de condição de corrida contribuíram para um dos maiores blecautes da história norte-americana em agosto de 2003, afetando cerca de 50 milhões de pessoas [98].

2.2.3 Mecanismos não-bloqueantes

Os mecanismos não-bloqueantes visam resolver os principais problemas existentes com os bloqueantes, vistos anteriormente. Como um atraso arbitrário provocado por uma *thread* não necessariamente impede que as demais façam progresso, problemas como *deadlock* e inversão de prioridade não ocorrem. Note também que uma implementação não-bloqueante exclui qualquer uso de trava, já que essa pode induzir estado de espera.

A literatura atual [2] classifica os algoritmos não-bloqueantes em três níveis, de acordo com a garantia de progresso de suas operações. Do mais forte para o mais relaxado, tem-se:

1. livre de espera (*wait-free*) – todas as operações terminam após um número finito de passos. O sistema como um todo faz progresso.

2. livre de trava (*lock-free*) – alguma operação termina após um número finito de passos. Há pelo menos uma *thread* fazendo progresso.

3. livre de obstrução (*obstruction-free*) – alguma operação sempre termina em um número finito de passos quando livre da interferência de outras operações. Uma *thread*, quando executada de forma isolada, sempre faz progresso.

[2] A nomenclatura ainda não é totalmente homogênea (e às vezes confusa). Para fins desta tese, a apresentada em [56] é seguida.

Dessa forma, a sincronização livre de espera garante progresso incondicional. Já a livre de trava admite que determinada *thread* não faça progresso, cenário denominado *starvation*. Por fim, como forma mais fraca, a livre de obstrução não garante progresso em condições em que haja *threads* concorrentes e conflitantes. Nesse esquema é possível que *threads* invalidem uma o trabalho da outra sem haver progresso algum, situação conhecida como *livelock*. Uma técnica baseada em *backoff* [4] pode ser usada nesse caso. É importante observar que tanto *starvation* quanto *livelock* também podem ocorrer com mecanismos bloqueantes.

As primitivas de sincronização usadas na abordagem não-bloqueante geralmente são as próprias instruções atômicas disponibilizadas pelo hardware, sendo `CAS` (*compare-and-swap*) a mais comum. Essa operação aceita como parâmetros uma posição de memória l, um valor esperado v_e e um valor novo v_n. A seguinte ação é então realizada de forma atômica: se o conteúdo de l for igual a v_e, então v_n é escrito em l; caso contrário l é deixado intacto. Em ambos os casos o conteúdo inicial de l é retornado pela operação.

O grande problema com a sincronização não-bloqueante é a extrema dificuldade em projetar, implementar e verificar a corretude dos algoritmos. Ainda hoje, a construção de algoritmos não-bloqueantes eficientes para determinadas estruturas de dados é resultado publicável em conferências e revistas prestigiosas. Isso não quer dizer que os algoritmos não sejam práticos. Por exemplo, o pacote para concorrência atual de `Java` (versão 6) disponibiliza uma classe para objetos do tipo fila (`ConcurrentLinkedQueue`) cuja implementação é baseada no algoritmo livre de trava de Michael e Scott [82]. Quanto ao desempenho, a abordagem não-bloqueante oferece, sob condições típicas, maior escalabilidade porque reduz potencialmente a fração de trecho serial da aplicação.

Capítulo 3

Transações em Memória Compartilhada

> *"The transaction concept has been very convenient in the database area and may be applicable to some parts of programming beyond conventional transaction processing."*
>
> Jim Gray, 1981

Pelo exposto no capítulo anterior fica evidente a necessidade de novas abstrações que facilitem o projeto, desenvolvimento e verificação de sistemas concorrentes. Ao mesmo tempo, e de forma conflitante, é exigido que tais sistemas tenham bom desempenho e sejam escaláveis. Uma abordagem recentemente proposta, denominada *memória transacional* (TM), surgiu como alternativa aos mecanismos de sincronização existentes, principalmente àqueles baseados em trava. TM usa como principal abstração o conceito de *transação* para estruturar e facilitar a escrita de programas concorrentes em memória compartilhada.

A ideia inicial de transação nasceu e se desenvolveu no contexto de sistemas de banco de dados [43]. Nesse domínio, uma transação é vista como um grupo de operações satisfazendo as seguintes propriedades (conhecidas como ACID):

- Atomicidade – ou todas as operações são executadas, ou então nenhuma o é. A presença de alguma falha durante a execução da transação faz com que os resultados das operações produzidos até aquele ponto sejam descartados [1];

- Consistência – uma transação leva o sistema de um estado consistente (pré-transação) a outro estado consistente (pós-transação);

[1] Esse comportamento também é conhecido como *failure atomicity*.

- Isolamento – os resultados parciais decorrentes do processamento de uma transação não são vistos por outras transações;
- Durabilidade – as mudanças efetivadas pela transação são permanentes e resistentes a uma eventual falha no sistema.

O conceito de transação no contexto de TM é basicamente o mesmo, exceto pela última propriedade: como as transações operam em memória volátil, faz pouco sentido definir durabilidade. Uma observação sobre o termo *atômico* faz-se necessária. A comunidade de TM costuma usar essa palavra no sentido de isolamento, para designar a não interferência entre duas ou mais transações. Este texto adota a mesma postura, fazendo a distinção somente nos casos em que o contexto não deixar claro o significado pretendido.

Transações em memória aumentam o nível de abstração para o controle de concorrência em programas concorrentes, oferecendo as seguintes vantagens:

Facilidade de programação – o programador não precisa se preocupar em *como* garantir a sincronização, e sim especificar *o que* deve ser executado atomicamente. Note que no caso das travas é necessário fazer a associação entre uma trava e o respectivo dado protegido. A abordagem transacional é muito mais declarativa nesse sentido, deixando o *como* para o sistema de execução.

Potencial ganho em desempenho – o fato da sincronização estar a cargo do sistema de execução permite explorar concorrência de forma mais agressiva. No caso das travas, o programador precisa adotar uma abordagem conservadora (ou pessimista) porque estaticamente não é possível prever todas as possíveis intercalações e regiões de memória acessadas. Já com transações, o sistema de execução pode permitir que um mesmo método seja executado concorrentemente (de forma otimista) e verificar se há conflitos entre os acessos. Isso permite total paralelismo nas situações em que os acessos são a dados disjuntos.

Composição – o maior nível de abstração fornecido pelas transações permite naturalmente a composição de código. Para criar uma nova operação com base em outras preexistentes basta englobá-las em uma nova transação. Isso é possível porque uma nova transação pode ser iniciada sob o contexto de outra, característica conhecida como *aninhamento*.

O termo memória transacional foi cunhado em 1993 por Herlihy e Moss para designar: *"uma nova arquitetura para microprocessadores que objetiva tornar a sincronização livre de trava tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua"* [55]. Novas instruções são introduzidas possibilitando ao programador especificar

operações de leitura, modificação e escrita em diversas regiões de memória de forma atômica. Uma abordagem bem semelhante foi publicada por Stone et al. no mesmo ano, diferenciando-se essencialmente nas mudanças propostas para o hardware [117].

No entanto, a iniciativa de usar transações como forma de estruturação de programas concorrentes foi feita bem antes por Lomet, em 1977 [75]. É interessante observar que muitas das propostas sugeridas nesse artigo são encontradas em sistemas de TM atuais. Todavia, o conceito não se popularizou na época principalmente por não se conhecer uma implementação eficiente. Lomet sugeriu uma implementação usando controle de concorrência pessimista, com base no protocolo de travamento em duas fases (*two-phase locking* – 2PL), comum em banco de dados [33]. Além disso, os mecanismos explícitos de sincronização baseados em trava eram bastante pesquisados naquele período, contribuindo para que o trabalho de Lomet não ganhasse maior importância.

Foram necessários praticamente 20 anos para que abordagens explorando transações no contexto de linguagens voltassem a aparecer. Os trabalhos de Knight (1986) [64] e de Tinker e Katz (1988) [119] usam transações de forma implícita para paralelizar programas escritos em linguagens funcionais (Lisp e Scheme, respectivamente). O termo memória transacional em software (STM) foi cunhado um pouco mais tarde, em 1995, por Shavit e Touitou [111]. A principal limitação dessa primeira abordagem é que as posições de memória acessadas por uma transação devem ser conhecidas *a priori*. A próxima seção apresenta o estado da arte em TM, introduzindo os principais trabalhos desenvolvidos na primeira década do século 21 e relevantes no contexto dos trabalhos apresentados nesta tese.

3.1 Estado da arte

A pesquisa em TM é tema atual, com presença constante nas principais conferências voltadas ao paralelismo. Grosseiramente, os trabalhos nessa área podem ser divididos em: (1) linguagem e semântica; (2) implementação (hardware, software e híbrido); e (3) caracterização. A Figura 3.1 mostra o panorama de publicações em cada subárea nos últimos 5 anos. Foram considerados nesse levantamento somente os trabalhos mais relevantes e os publicados nos principais eventos, tais como: *Symposium on Principles and Practice of Parallel Programming* (PPoPP), *Symposium on Parallel Algorithms and Architectures* (SPAA), *Symposium on Distributed Computing* (DISC), *Symposium on Principles of Distributed Computing* (PODC), *Conference on Programming Language Design and Implementation* (PLDI), *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), *Conference on Parallel Architectures and Compilation Techniques* (PACT), *Symposium on High-Performance Computer Architecture* (HPCA) e *International Symposium on Computer Architecture* (ISCA).

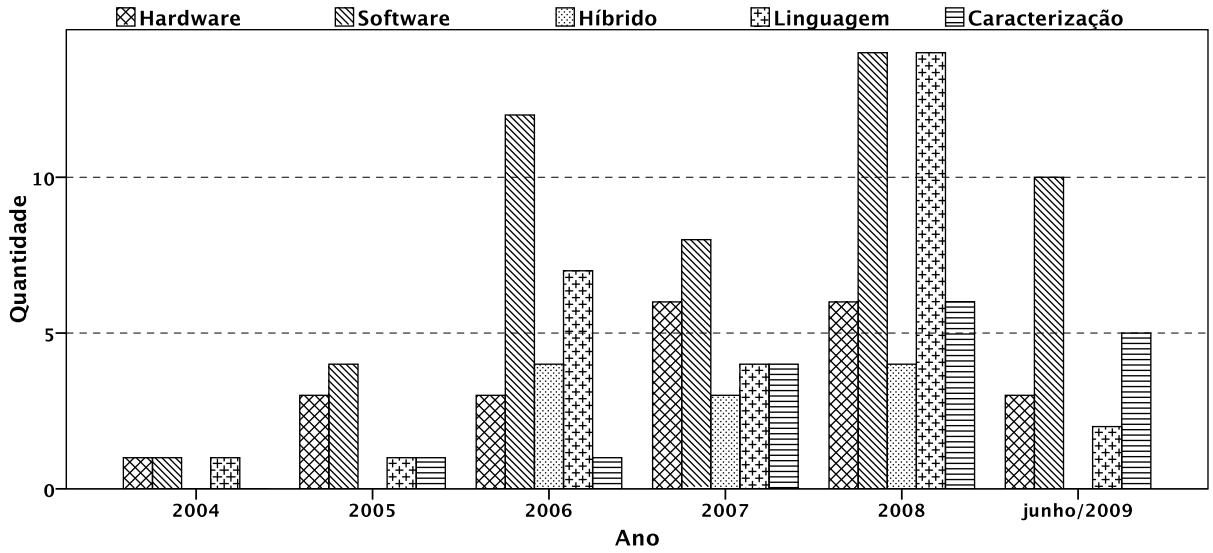

Figura 3.1: Número de publicações nas principais subáreas de TM por ano

Duas principais observações podem ser feitas a partir da Figura 3.1. Primeiro, a pesquisa na área como um todo vem crescendo muito, particularmente no último ano. Segundo, a integração com linguagens (envolvendo a parte semântica) e a caracterização (envolvendo a criação de aplicações transacionais) têm recebido maior importância, o que demonstra uma maturação da área. É importante notar que a divisão em subáreas é sutil. Muitas vezes um mesmo trabalho pode estar inserido em mais de um contexto. Nas próximas seções cada uma das subáreas é vista em maior detalhe.

3.2 Linguagem e semântica

Aspectos referentes à linguagem e semântica são importantes porque indicam o grau de expressividade da linguagem e definem univocamente o significado de cada uma de suas construções. No contexto de transações, as seguintes três construções são atualmente aceitas: `atomic`, `retry` e `orElse`. A palavra `atomic`, comumente usada para denotar transações, foi introduzida por Harris e Fraser no seminal artigo de 2003 [50]. As duas restantes foram posteriormente introduzidas por Harris *et al.* [51] visando especialmente a coordenação de transações e composição de código.

A Figura 3.2 ilustra o uso destas construções transacionais através de um exemplo envolvendo uma fila com tamanho limitado. Para facilitar a leitura, uma sintaxe baseada em linguagens orientadas a objeto (estilo `Java` e `C#`) é empregada. A Figura 3.2a mostra a declaração parcial de uma classe para a fila de tamanho limitado (`Queue`), com a operação

```
1 class Queue {
2   int head, tail;   //
3   int[] items;      // compartilhados
4
5   boolean enq(int value) {
6     atomic {
7       if (tail-head == items.length)
8         return false;
9
10      items[tail%items.length] = value;
11      tail++;
12      return true;
13    }
14  }
15 }
```

(a) Bloco atômico

```
1 void enq(int value) {
2   atomic {
3     if (tail-head == items.length)
4       retry;
5
6     items[tail%items.length] = value;
7     tail++;
8   }
9 }
```

(b) Uso do `retry`

```
1 ...
2
3 atomic {
4   queue1.enq(x);
5 } orElse {
6   queue2.enq(x);
7 }
8
9 ...
```

(c) Uso do `orElse`

```
1 void move(queue1, queue2) {
2
3   atomic {
4     int x = queue1.deq();
5     queue2.enq(x);
6   }
7
8 }
```

(d) Composição

Figura 3.2: Construções transacionais atuais

`enq` que insere um elemento no final da fila (linhas 5–14). Também é comum a definição da operação `deq` (não mostrado na figura) que retira o elemento do início da fila (assumindo que a fila não esteja vazia). Um objeto do tipo `Queue` pode ser usado concorrentemente, portanto os métodos que acessam as variáveis compartilhadas (linhas 2 e 3) devem ser sincronizados. A palavra `atomic` na linha 6 inicia um bloco atômico que se estende até a linha 13. As instruções contidas nesse bloco compreendem, desta forma, uma transação. Uma propriedade conhecida como *serializabilidade* [97] é geralmente usada como critério de corretude para execução das transações. Essa propriedade diz que o efeito resultante ocasionado pela execução de um grupo de transações é o mesmo obtido caso essas mesmas transações fossem executadas isoladamente, em alguma ordem, mesmo que na verdade tenham sido executadas concorrentemente. Ou seja, o programador pode assumir que os acessos ao estado compartilhado (leituras e escritas) efetuados dentro de uma transação são sempre consistentes.

O código para o método `enq` apresentado na Figura 3.2a retorna falso caso a fila esteja cheia. Um comportamento alternativo, e muitas vezes mais apropriado, é esperar até que haja espaço na fila para que o item seja inserido. Esse comportamento é mostrado na

Figura 3.2b através do `retry`. A construção `retry` causa o cancelamento da transação, desfazendo todas as ações intermediárias. Quando algum dado compartilhado recém acessado pela transação é modificado, ela é executada novamente. No exemplo, a transação compreendida pelas linhas 2–8 será reexecutada quando as variáveis `tail` ou `head` forem alteradas por outra transação (mais especificamente, quando uma outra transação retirar um elemento via `deq` e alterar `head`). Nota-se, portanto, que a função de `retry` é prover um mecanismo de coordenação entre transações, semelhante à função das variáveis de condição no caso da sincronização bloqueante.

Imagine agora que há a necessidade, em algum outro programa, de inserir um elemento em uma de duas filas quaisquer. Caso uma delas esteja cheia, o comportamento mais adequado é inserir o elemento na outra fila ao invés de esperar pela primeira. O código mostrado na Figura 3.2c ilustra como a construção `orElse` resolve esse problema. A chamada feita na linha 4 bloquearia caso a respectiva fila estivesse cheia. Quando `orElse` é usado, o `retry` invocado internamente por `queue1` cancela a subtransação e passa o controle ao bloco `orElse` seguinte que, por sua vez, causa a execução da subtransação presente na linha 6. Se essa fila também estiver cheia, `retry` será novamente executado, dessa vez causando o cancelamento e reexecução de toda a transação (linhas 3–7). Esse exemplo mostra a utilidade de `orElse` em situações envolvendo seleção de alternativas.

Por fim, a Figura 3.2d mostra um exemplo de composição de transações. Um novo método (`move`) é definido para mover um elemento de uma fila (`queue1`) para outra (`queue2`). Essa operação é facilmente descrita pela transação definida nas linhas 3–6. Pelo fato da operação ser atômica, é garantido que nenhuma outra transação verá o estado intermediário no qual o elemento retirado de uma fila (linha 4) ainda não tenha sido inserido na outra (linha 5).

3.2.1 Desafios

Há muitos desafios relacionados à semântica transacional. Uma das questões mais antigas, inicialmente levantada por Blundell *et al.* [16], é relacionada à interação entre código transacional e código não transacional. No modelo com *atomicidade forte* (*strong atomicity*) o acesso a um dado compartilhado fora de uma transação é consistente com os acessos efetuados ao mesmo dado dentro da transação. Em contrapartida, no modelo com *atomicidade fraca* (*weak atomicity*) o acesso a um mesmo dado compartilhado, dentro e fora de uma transação, causa uma condição de corrida e portanto o resultado é não-determinístico. As implementações em hardware geralmente garantem atomicidade forte, enquanto as implementações em software só garantem atomicidade fraca por motivos de desempenho.

Uma caracterização mais formal de TM foi introduzida por Scott *et al.* [108], seguida

pelos trabalhos de Abadi *et al.* [1] e de Moore e Grossman *et al.* [88]. Uma visão mais pragmática é dada por McDonald *et al.* [79], descrevendo a relação entre hardware e software em um sistema transacional. Recentemente, Guerraoui e Kapalka *et al.* [45] sugerem *opacidade* como critério de corretude. Eles definem opacidade como uma extensão de serializabilidade, impedindo até que transações não efetivadas acessem estados inconsistentes. Em trabalho posterior, eles ainda apresentam formalmente propriedades para implementações não-bloqueantes [44] e bloqueantes [46]. Outros trabalhos investigam ainda a semântica e viabilidade de transações aninhadas [8, 91, 6, 7].

Como pode ser visto, a semântica e integração de transações em linguagens convencionais têm ganhado bastante destaque nos últimos anos. No entanto, há muito ainda o que ser feito. Como exemplo, considere o uso de operações de entrada e saída dentro de transações. Qual é o comportamento esperado? Muitas dessas operações são irreversíveis e portanto a abordagem atual de simplesmente refazer a transação não pode ser aplicada.

3.3 Implementação

A implementação de um sistema com memória transacional deve essencialmente garantir atomicidade e isolamento das transações. As implementações são construídas com base em dois conceitos chaves: *versionamento de dados* e *detecção/resolução de conflitos*.

Versionamento de dados

O versionamento lida com o gerenciamento das diferentes versões dos dados acessados por uma transação. Geralmente duas versões são mantidas para cada dado: a *original* e a *especulativa*. A versão original corresponde ao valor do dado antes do início da transação, enquanto a versão especulativa representa o valor intermediário sendo trabalhado pela transação. Caso a transação seja efetivada, os valores especulativos tornam-se os valores correntes, e os valores originais são descartados. Do contrário, os especulativos são descartados e os originais mantidos.

Existem duas formas de versionamento: (i) *versionamento direto* (*direct update* ou *eager versioning*), no qual os valores especulativos são armazenados diretamente na memória e os originais salvos em um *undo log* interno; e (ii) *versionamento diferido* (*deferred update* ou *lazy versioning*), no qual os valores especulativos são mantidos internamente em um *buffer* de escrita e a memória não é alterada diretamente.

A Figura 3.3 ilustra um exemplo com os versionamentos (a) direto e (b) diferido. Inicialmente, o conteúdo da variável X na memória corresponde ao valor 100 e os respectivos *undo log* e *buffer* estão vazios ①. Quando uma transação atribui o valor 77 à variável X, o versionamento direto altera imediatamente o conteúdo da memória e salva o valor antigo localmente ②, enquanto o versionamento diferido somente necessita salvar o novo valor em

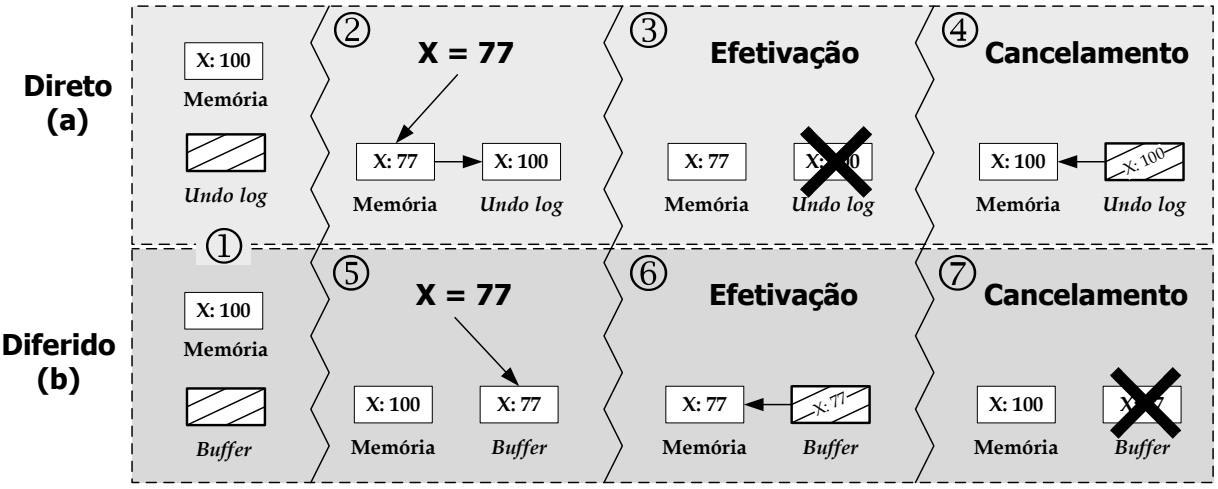

Figura 3.3: Exemplo ilustrando versionamento direto (a) e diferido (b)

memória local ⑤. No momento da efetivação, a única ação necessária no versionamento direto é invalidar o *undo log* ③. Já o versionamento diferido precisa primeiramente mover o valor armazenado localmente para a memória do sistema ⑥. Uma situação inversa acontece caso a transação necessite ser cancelada. Nesse caso, o versionamento direto precisa restaurar as mudanças efetuadas na memória, movendo para esta o valor antigo presente no *undo log* local ④. Com o versionamento diferido só é necessário descartar os valores especulativos ⑦.

Como pode ser percebido pelo exemplo da Figura 3.3, o versionamento direto torna mais rápida a efetivação da transação, porque os dados já estão na memória. No entanto, caso a transação seja cancelada, é necessário restaurar os valores na memória usando o *undo log*. O comportamento do versionamento diferido é o oposto, ou seja, o cancelamento é rápido (basta descartar os dados especulativos) mas a efetivação requer que os dados especulativos sejam transferidos para a memória.

Detecção de conflitos

Para detecção de conflitos uma transação geralmente mantém um conjunto de leitura com os endereços dos elementos lidos, e um conjunto de escrita com os endereços dos elementos que foram alterados. Há um conflito entre duas ou mais transações quando a intersecção entre o conjunto de leitura e o conjunto de escrita de transações diferentes é não vazia. Em outras palavras, quando duas ou mais transações acessam o mesmo elemento e um dos acessos é de escrita.

Assim como no versionamento, existem duas opções para detecção de conflitos: (i) *pessimista* (*pessimistic* ou *eager conflict detection*), no qual o conflito é detectado assim que se manifesta; e (ii) *otimista* (*optimistic* ou *lazy conflict detection*), no qual o conflito

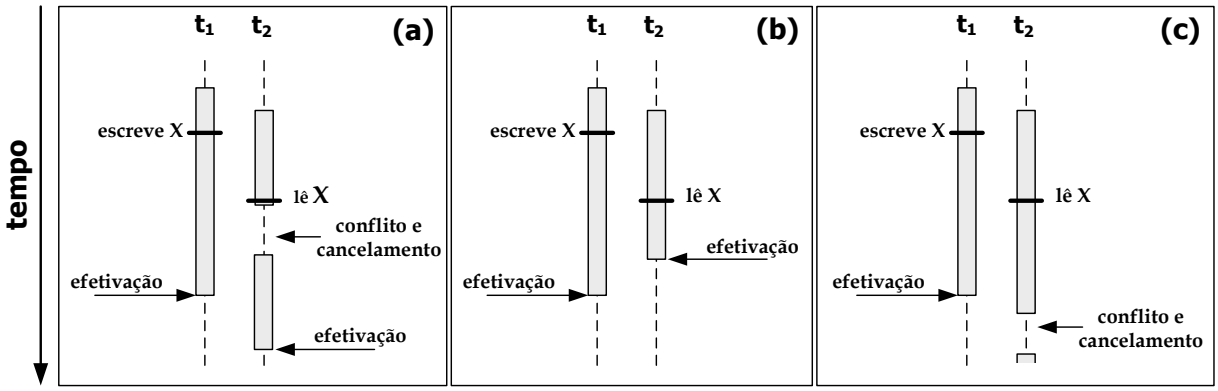

Figura 3.4: Possíveis cenários para detecção de conflito: (a) pessimista e (b)(c) otimista

é detectado somente no momento em que a transação é efetivada.

A abordagem pessimista pode evitar que trabalho inútil seja realizado ao cancelar uma transação antecipadamente. Por outro lado, a abordagem otimista permite que transações conflitantes prossigam na esperança do conflito não se efetivar de fato. Para ilustrar essas situações, considere os cenários mostrados na Figura 3.4. Duas transações, t_1 e t_2, são executadas concorrentemente. Uma posição de memória compartilhada, denotada por X, é escrita por t_1 e somente lida por t_2. O cenário descrito em (a) usa o esquema pessimista: t_2 lê uma posição de memória já alterada por t_1 e deve, desta forma, ser cancelada. Esse cancelamento poderia ser evitado caso t_2 fosse efetivada antes de t_1 e a detecção fosse feita de forma otimista. O cenário (b) ilustra exatamente este caso. Como os conflitos são conhecidos somente no momento da efetivação, a escrita de X por t_1 ainda não é vista por t_2 no momento da sua efetivação. No entanto, se t_2 for efetivada depois de t_1, como ilustrado em (c), então todo o trabalho realizado é perdido. Nesse caso a detecção pessimista do caso (a) permitiria uma execução mais eficiente.

A resolução de conflitos deve ser feita de tal forma a garantir progresso do sistema, ou seja, deve evitar *starvation* e *livelock*. O componente do sistema transacional responsável por garantir progresso é conhecido como *gerenciador de contenção*.

Não existe um consenso na comunidade de TM sobre qual estratégia para versionamento e detecção/resolução de conflitos é a mais efetiva, já que uma estratégia pode funcionar bem para algumas aplicações mas não para outras. Além do versionamento e gerenciamento de conflitos, há outros fatores que também influenciam a implementação, tais como: granularidade de acesso (palavra, linha de cache, objeto), atomicidade (forte ou fraca) e aninhamento de transações. Em um nível mais geral, as implementações são comumente divididas em hardware, software e híbridas. As próximas seções descrevem cada uma delas, com destaque para implementações em software.

3.3.1 Hardware

As implementações de TM em hardware (HTM) usualmente fazem o versionamento e controle de conflito através de alterações na cache e no protocolo de coerência. A cache de dados é estendida com mais estados de forma a identificar se uma determinada linha foi lida ou escrita transacionalmente. Como consequência, geralmente o protocolo de coerência de cache deve ser adaptado para suportar os novos estados. É importante observar que HTM tem como características naturais o isolamento forte e a granularidade de linha de cache.

As primeiras implementações desta década procuravam suprir as principais deficiências da proposta seminal de Herlihy e Moss de 1993 [55], a qual restringia o número de posições acessadas pela transação (espaço) e o tempo em que a transação poderia estar ativa (por exemplo, não poderia haver troca de contexto). As principais são: TCC (*Transactional Coherence and Consistency*) [48], UTM (*Unbounded Transactional Memory*) [10], VTM (*Virtual Transactional Memory*) [100], LogTM (*Log-based Transactional Memory*) [87], PTM (*Page-based Transactional Memory*) [21], OneTM [15] e MetaTM [101]. Essas propostas têm suporte para virtualização de espaço e tempo, diferenciando-se entre si pelo hardware extra exigido e estratégia adotada para versionamento e controle de conflitos.

Abordagens posteriores surgiram com suporte para novas funcionalidades em hardware, como o aninhamento de transações [89]. Pesquisadores de HTM ainda procuram reduzir a complexidade de hardware necessário e, ao mesmo tempo, aumentar o desempenho. As propostas mais recentes que merecem destaques são: LogTM-SE [127], TokenTM [17] e DATM (*Dependence-Aware Transactional Memory*) [102]. No entanto, as implementações em hardware ainda são consideradas complexas, tornando inviável a adoção e implementação efetiva de alguma delas em um processador comercial. Além disso, a semântica transacional ainda não é considerada madura o suficiente para admitir que estratégias e técnicas específicas sejam implementadas diretamente em hardware. A grande vantagem de HTM, sem dúvidas, é o desempenho. O fato de proverem isolamento forte também pode ser considerado uma vantagem, já que esse comportamento tem sido considerado ultimamente o mais adequado.

3.3.2 Software

As abordagens que implementam os mecanismos transacionais diretamente em software (STM) são atualmente alvo de muita pesquisa. O motivo principal deve-se à flexibilidade na implementação de novos algoritmos, principalmente em face da diversidade semântica existente. Além disso, as abordagens em software também podem ser executadas diretamente nos processadores atuais, enquanto as técnicas em hardware devem ser testadas em simuladores.

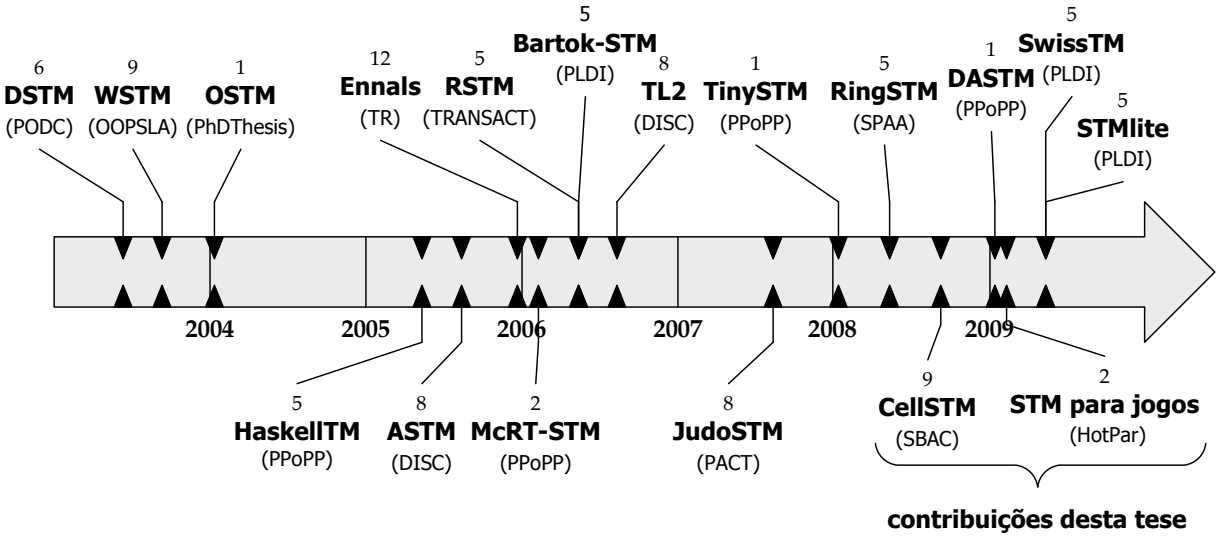

Figura 3.5: Linha do tempo com os principais sistemas de STM

A Figura 3.5 mostra os principais sistemas de STM existentes na literatura, o ano e mês em que foram publicados, e a respectiva conferência. Duas das contribuições desta tese estão destacadas na figura e serão apresentadas nos Capítulos 4 e 5, respectivamente. A figura ilustra o grande volume de abordagens existentes e serve como referência para a discussão a seguir.

Os sistemas em software usam barreiras de leitura e escrita para interceptar os acessos aos dados compartilhados e manter os conjuntos de leitura e escrita. Existem dois tipos principais de implementações: não-bloqueantes e bloqueantes. Além disso, os diversos sistemas de STM se diferenciam principalmente pela granularidade de acesso permitido (palavra ou objeto), e pela estratégia escolhida para versionamento, detecção e resolução de conflitos. Ao contrário de HTM, a maioria das implementações em software garantem apenas atomicidade fraca, principalmente por motivos de desempenho: garantir atomicidade forte exigiria incluir barreiras também em código não transacional.

As primeiras implementações de STM são não-bloqueantes. Entre as mais famosas estão: DSTM (*Dynamic STM*) [54], WSTM (*Word-based STM*) [50], OSTM (*Object-based STM*) [38], HaskellTM [51], ASTM (*Adaptive STM*) [76] e RSTM (*Rochester STM*) [77]. Em especial, DSTM, ASTM e RSTM são livres de obstrução, enquanto WSTM, OSTM e HaskellTM são livres de trava. Nas implementações livres de obstrução, o gerenciador de contenção tem um papel vital, pois determina o progresso do sistema. A maioria desses sistemas trabalham com granularidade de objeto, exceto WSTM e HaskellTM que usam granularidade de palavra.

A Figura 3.6 ilustra a estrutura de uma STM livre de obstrução, com base na implementação da DSTM para a linguagem `Java`. Nessa infraestrutura, um objeto do tipo

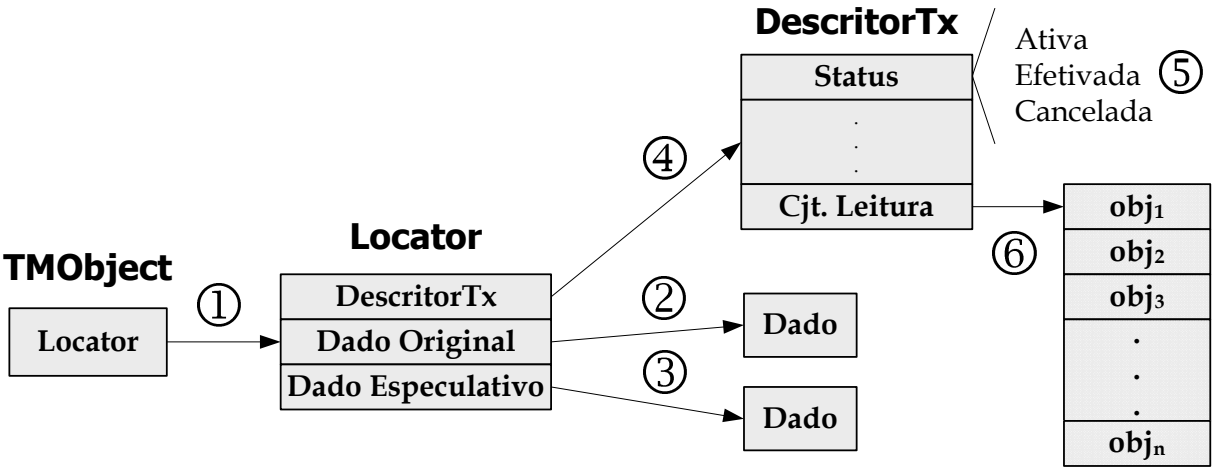

Figura 3.6: Estrutura de uma STM livre de obstrução (DSTM)

`TMObject` serve como contêiner para objetos regulares que possam ser compartilhados. O contêiner `TMObject` adiciona dois níveis de indireção para o acesso aos dados do objeto regular: uma referência para uma estrutura do tipo `Locator` ①, que por sua vez contém referências para as duas possíveis versões do objeto: a original ② e a especulativa ③. A estrutura `Locator` também armazena uma referência para o descritor da transação (`DescritorTx`) ④ que contém, entre outros, os campos com o *status* atual da transação (*ativa*, *efetivada* ou *cancelada*) ⑤ e o conjunto de objetos lidos ⑥.

Antes de alterar um objeto, uma transação deve solicitar sua abertura para escrita através de uma chamada explícita disponibilizada pela DSTM. Isso permite o versionamento e detecção de conflitos, que ocorre conforme a seguinte lógica. Primeiramente, o descritor da transação associado ao objeto é acessado e o *status* da transação é verificado. Se o *status* for *ativa*, então há um conflito e o gerenciador de contenção é invocado para resolver o impasse, geralmente abortando uma das transações. Caso contrário, um novo objeto do tipo `Locator` é criado, o campo `DescritorTx` é inicializado com a referência para o descritor da transação corrente, e o objeto regular é clonado de acordo com o *status* da última transação que acessou o objeto. Se o *status* for *efetivada*, o dado especulativo é clonado. Senão, o clone é criado com base na versão original. Uma operação `CAS` é então efetuada para alterar atomicamente o campo `Locator` de `TMObject` para o novo objeto `Locator` recém-criado. Por fim, a operação de abertura retorna a referência para o novo objeto.

Um objeto também pode ser aberto só para leitura. Nesse caso, se não houver conflitos, não há a necessidade da clonagem, bastando que o objeto seja inserido no conjunto de leitura da transação. Note que não é possível descobrir, dado um objeto, as transações que o estão usando para leitura. Implementações com essa característica possuem *leitores*

invisíveis (*invisible readers*) e requerem validação de todo o conjunto de leitura na abertura de qualquer objeto, com a finalidade de evitar inconsistências. A operação de efetivação é a mais simples: depois de validar o conjunto de leitura, uma operação `CAS` é utilizada para mudar o *status* da transação de *ativa* para *efetivada*.

As implementações bloqueantes começaram a aparecer a partir de 2006. Ennals [32] mostrou que implementações livres de bloqueio não eram necessárias e, principalmente, que não forneciam bom desempenho. O principal argumento de Ennals é que as indireções necessárias para acessar os objetos aumentavam o número de faltas na cache de dados, degradando o desempenho. Ennals apresentou uma implementação usando travas com as seguintes características: (i) escritas usam um protocolo de travamento em duas fases [33]; e (ii) leituras são feitas otimisticamente, usando uma técnica já conhecida em banco de dados [69]. Os resultados de Ennals mostraram um ganho em desempenho de até 3 vezes em relação a DSTM e OSTM.

A partir do importante trabalho de Ennals a maioria das implementações propostas passaram a ser bloqueantes. Em especial, foram apresentadas: McRT-STM (*Multi-core RunTime STM*) pela Intel [106], Bartok-STM pela Microsoft [52], TL2 (*Transactional Locking II*) pela Sun [27], TinySTM pelas universidades de Neuchâtel e de Dresden [34] e SwissTM pela EPFL [30]. Em geral, todas partem da implementação básica de Ennals e adicionam novas características com o propósito de aumentar a expressividade das construções transacionais e melhorar o desempenho. Por serem implementadas com travas, estas abordagens estão expostas a anomalias como o *deadlock*. Portanto, o algoritmo deve ser cuidadosamente projetado e considerar esses cenários.

Atualmente, a TL2 é amplamente considerada o estado da arte de STM bloqueantes [116, 80]. TL2 usa um relógio global para garantir consistência das leituras e escritas e evitar transações zumbis. Uma transação zumbi é aquela que já está fadada a abortar mas ainda continua a executar com seu conjunto de leitura inconsistente, podendo encontrar cenários como laços infinitos ou ainda efetuar acessos ilegais à memória. As implementações McRT-STM e Bartok-STM, por exemplo, admitem transações zumbis. Elas resolvem os problemas de inconsistência inserindo código extra de validação em laços e, como são implementadas em ambientes gerenciáveis, interceptam as exceções causadas por acessos ilegais. Uma discussão mais detalhada da TL2 é postergada até o Capítulo 4.

Outras implementações mais recentes divergem em algum grau do modelo proposto por Ennals, mais notadamente: JudoSTM [94], RingSTM [116], DASTM (*Dependence-Aware STM*) [103] e STMlite [80]. Resumidamente, JudoSTM usa a técnica de reescrita binária para instrumentar código transacional em tempo de execução e permitir que código legado seja usado transacionalmente. RingSTM usa uma estrutura centralizada de anel onde o conjunto de escrita de uma transação é representado por uma assinatura. O anel controla a ordem de efetivação (serialização) e serve para detectar conflitos entre

```
1 int x; // shared
2
3 atomic {
4
5   x++;
6
7 }
```

(a) Versão original

```
1  int x; // shared
2  jmp_buf checkpoint;
3
4  setjmp(checkpoint);
5
6  TxStart(&checkpoint);
7
8    int temp;
9    temp = TxLoad((intptr_t *)(void *)&x);
10   temp++;
11   TxStore((intptr_t *)(void *)&x, (intptr_t)temp);
12
13 TxCommit();
```

(b) Versão instrumentada

Figura 3.7: Inserção de barreiras transacionais em bloco atômico

transações. Já a abordagem adotada por DASTM permite que conflitos do tipo escrita após leitura (*read-after-write*) não causem o cancelamento imediato da transação, mas sim que o valor modificado seja repassado para a outra transação. Por fim, STMlite usa uma *thread* dedicada, chamada TCM (*Transactional Commit Manager*), para validar e efetivar transações, de maneira similar a RingSTM. Transações transmitem somente as assinaturas de seus conjuntos de leitura e escrita, que são então processadas pela TCM.

Apesar de grandes avanços, as implementações em software ainda sofrem com baixo desempenho. Como dito, todo o acesso a um dado compartilhado deve ser instrumentado, causando uma significativa perda de desempenho. Considere o exemplo de instrumentação ilustrado pela Figura 3.7. A Figura 3.7a representa o código escrito pelo programador usando a construção `atomic`. A operação consiste em simplesmente incrementar o valor de uma variável compartilhada. Um compilador transformaria esse código em algo parecido com o da Figura 3.7b, assumindo uma STM no estilo da TL2 e uma linguagem estruturada como `C`. A Figura 3.8 apresenta a quantificação do custo extra introduzido pelas barreiras para um subconjunto de aplicações do pacote STAMP (*Stanford Transactional Applications for Multi-Processing*) [83] usando a TL2. A figura mostra o tempo de execução, normalizado em relação à execução sequencial (sem barreiras), gasto pelo código transacional usando uma única *thread*. Como pode ser visto, o tempo de execução pode chegar a até 4 vezes o da versão sequencial (`intruder`). O custo total de cada barreira depende do sistema de STM usado e da aplicação. No caso do exemplo, o tempo gasto com efetivações é predominante, seguido pelas barreiras de leitura e escrita.

Existe muita pesquisa com foco em reduzir o custo causado pelas barreiras transacionais em STM [52, 3, 28, 122, 78, 115]. Outras pesquisas focam em prover atomicidade forte em STM [112, 81, 107, 2] e implementar suporte para transações aninhadas [93, 6].

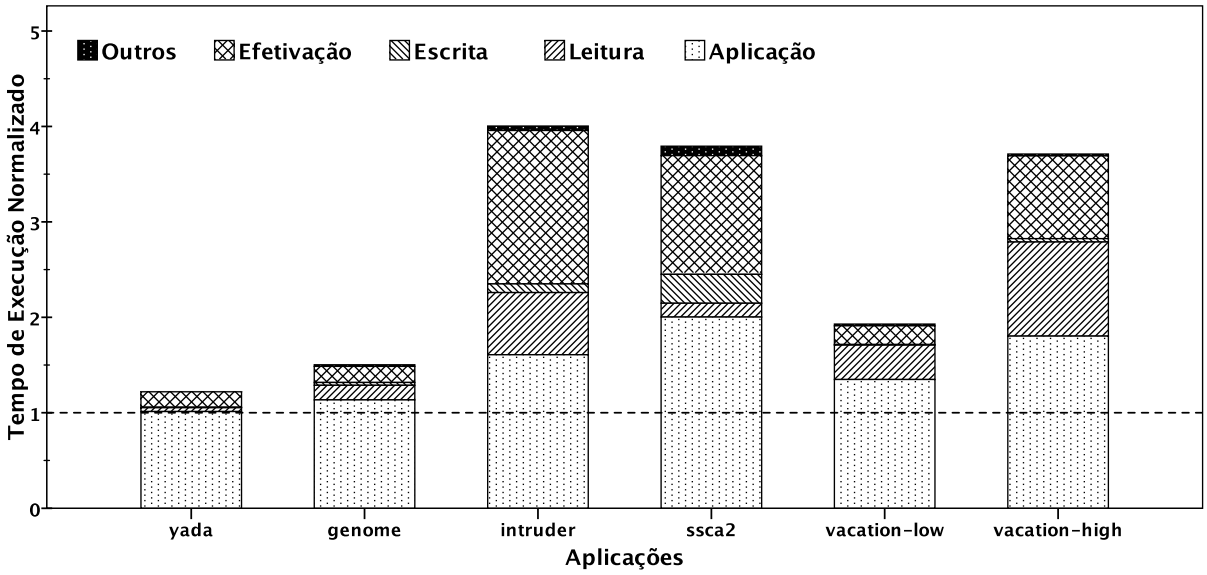

Figura 3.8: Custo introduzido pelas barreiras transacionais (TL2)

3.3.3 Híbrida

Os sistemas de HTM geralmente usam hardware complexo, inviabilizando uma implementação real. Por outro lado, os sistemas de STM causam considerável degradação de desempenho devido à instrumentação introduzida no código. A solução natural foi então mesclar as soluções em hardware com as em software, originando as abordagens híbridas.

As primeiras propostas usavam um hardware limitado e simples como modo padrão para execução das transações [23, 68]. As transações que não tivessem mais como executar em modo de hardware (por exemplo, devido ao fato da cache de dados transbordar) passavam então a serem executadas totalmente em software. O grande desafio nessa abordagem é quando existem transações concorrentes executando tanto em modo de hardware quanto em modo de software. Contornar esse problema geralmente requer soluções que tornam as transações em hardware mais lentas do que o desejado.

Posteriormente, as propostas se basearam em aceleração do software através de suporte em hardware [105, 114, 84, 113]. A principal característica dessas propostas é que as transações são sempre executadas em modo de software, mas usam o hardware para acelerar tarefas críticas. Por exemplo, a validação é reconhecidamente um gargalo na maioria das STMs. Uma alternativa seria adicionar hardware para acelerar essa tarefa, como feito por Saha *et al.* [105]. A grande vantagem é que o hardware adicionado não é específico para TM, podendo ser potencialmente usado em outras tarefas.

Os sistemas híbridos fazem relativo sucesso. Por exemplo, a Sun anunciou para 2009 o lançamento do processador Rock com suporte transacional [19]. A estratégia adotada

pela Sun na confecção do processador foi usar uma abordagem híbrida [26].

3.4 Caracterização

Caracterizar o comportamento de implementações de TM é importante porque revela aspectos ineficientes, sugerindo modificações no projeto e melhora do sistema. Há um grande problema porém, principalmente para as primeiras implementações: como a semântica ainda não é bem estabelecida, confeccionar aplicações transacionais complexas e reais é uma tarefa difícil e geralmente não portável.

As primeiras propostas usavam os chamados *micro-benchmarks*: aplicações pequenas que exercitam operações como inserção, remoção e pesquisa em estruturas de dados como lista ligada, *hashtable* e árvores. Também era comum o uso do pacote SPLASH-2 [126] nas abordagens de HTM. A versão transacional era gerada pela substituição das seções críticas por transações. No entanto, as aplicações do SPLASH-2 já eram codificadas de forma a minimizar o custo da sincronização e estavam longe de representar o cenário típico de aplicações transacionais. O primeiro estudo mais detalhado de caracterização foi conduzido por Chung *et al.* [22], considerando 35 aplicações de diversos domínios, como computação científica e comercial. A versão transacional de cada aplicação era gerada automaticamente como no SPLASH-2, ou seja, as operações de requisição e liberação de uma trava eram substituídas por uma transação. Novamente, não ficou claro se as aplicações apresentadas faziam uso adequado de transações. O mais desejável seriam aplicações construídas já com o modelo transacional em mente.

Felizmente, as pesquisas relacionadas a caracterização e confecção de aplicações transacionais tem crescido bastante. Apesar da semântica de TM ainda não estar totalmente definida, o número de aplicações disponíveis aumentou consideravelmente, tanto em quantidade como em qualidade. Os pacotes mais conhecidos são o STAMP [83], conjunto com 8 aplicações, e o STMBench7 [47], provendo customização e suporte para operações em estruturas de dados complexas. Estudos individuais analisaram e implementaram versões transacionais para a triangularização de Delaunay [109], algoritmo de roteamento de Lee [123, 11], computação de árvore de espalhamento mínima em grafos esparsos [61] e um servidor de jogos multi-jogador [129, 39].

Os trabalhos de caracterização têm focado exclusivamente em desempenho. O Capítulo 6 desta tese apresenta um trabalho pioneiro de caracterização de energia para a TL2, usando o pacote STAMP.

Capítulo 4

CellSTM: Uma Implementação de STM para a Arquitetura Cell/B.E.

"Heterogeneous (or asymmetric) chip multiprocessors present unique opportunities for improving system throughput, reducing processor power, and mitigating Amdahl's law."

Kumar, Tullsen, Jouppi e Ranganathan, 2005

Este capítulo explora a implementação de um sistema de memória transacional em software para arquiteturas assimétricas, mais especificamente, para a arquitetura Cell/B.E.. A Seção 4.1 ressalta a importância do trabalho, enquanto a Seção 4.2 apresenta os conceitos necessários ao restante do capítulo, sendo composta pela descrição da arquitetura Cell/B.E., cache gerenciada por software e o sistema TL2. O sistema CellSTM é apresentado na Seção 4.3, seguido dos principais resultados experimentais na Seção 4.4 e das considerações finais na Seção 4.5.

4.1 Motivação

As arquiteturas CMP podem ser, de maneira mais geral, classificadas de acordo com a variedade dos núcleos replicados. Em uma arquitetura simétrica (ou homogênea) todos os núcleos são idênticos, enquanto uma arquitetura assimétrica (ou heterogênea) pode empregar diferentes núcleos. Embora as arquiteturas homogêneas sejam maioria hoje, as heterogêneas têm ganhado crescente destaque. Isto porque apresentam oportunidades únicas para maior *throughput* com menor consumo de energia, já que permitem um mapeamento mais eficiente dos recursos arquiteturais para uma faixa de aplicações mais

ampla [67]. Considere, como exemplo, uma arquitetura homogênea com quatro núcleos idênticos de propósito geral. A troca de um desses núcleos por um número maior de processadores mais simples e especializados, poderia aumentar o desempenho de uma série de aplicações e ainda diminuir o gasto total com energia.

Um dos principais desafios à disseminação das arquiteturas assimétricas é o desenvolvimento do software aplicativo e de sistema. Para tirar o máximo desempenho, muitas vezes o modelo de programação exige do programador conhecimentos específicos da arquitetura. Visando prover um modelo mais abstrato de programação, este capítulo propõe, pela primeira vez, um sistema de STM para arquiteturas assimétricas. A abordagem apresentada aqui assume que tais arquiteturas sejam compostas por pelo menos um processador de propósito geral, e um ou mais núcleos específicos (chamados de *aceleradores*). Esses aceleradores geralmente possuem memória local segregada da memória global do sistema, a qual é acessada através de operações especiais como as de acesso direto à memória (*Direct Memory Access* – DMA).

Embora a abordagem aqui proposta possa ser usada em qualquer arquitetura assimétrica com as propriedades mencionadas, o texto se concentra no processador Cell Broadband Engine (Cell/B.E.) [60] por questões pragmáticas. O sistema de STM apresentado neste capítulo, chamado de CellSTM, estende o conceito de cache gerenciada por software e fornece, de forma transparente, uma visão consistente da memória compartilhada aos aceleradores.

4.2 Contextualização

4.2.1 Cell/B.E.

O processador Cell/B.E. é o resultado da colaboração entre Sony, Toshiba e IBM. A arquitetura, ilustrada através da Figura 4.1, é composta por nove elementos de processamento (incluindo um processador de propósito geral e oito aceleradores) ①, interfaces externas para memória principal (*Memory Interface Controller* – MIC) ② e entrada/saída (*Broadband Engine Interface* – BEI) ③, conectados entre si por um barramento coerente de alto desempenho (*Element Interconnect Bus* – EIB) ④.

Os nove elementos de execução são divididos em duas categorias principais: um processador PowerPC (*PowerPC Processor Element* – PPE) e oito processadores sinergísticos (*Synergistic Processor Element* – SPE). O PPE é composto por um núcleo de execução de 64 bits baseado na arquitetura PowerPC (com suporte para duas *threads* simultâneas) e a respectiva cache L1, geralmente chamados de PPU (*PowerPC Processor Unit*), mais uma cache externa L2. A função primária do PPE é executar código de controle, como o sistema operacional e gerenciamento dos processadores sinergísticos. Os SPEs são pro-

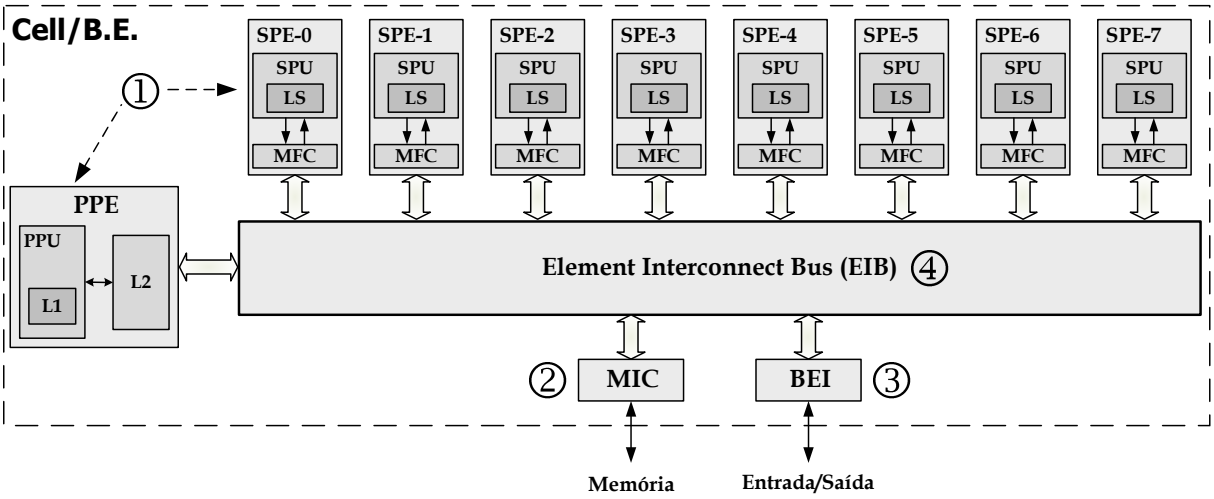

Figura 4.1: Arquitetura simplificada do Cell/B.E.

cessadores SIMD (*Single Instruction Multiple Data*) especializados para aplicações de computação intensiva. Possuem um núcleo de execução simples (sem execução fora de ordem e predição de saltos), um conjunto de 128 registradores de 128 bits e uma memória local (*Local Storage* – LS) para instruções e dados de 256KB, conhecidos conjuntamente por SPU (*Synergistic Processor Unit*). A comunicação com a memória principal e outros processadores é feita através do controlador de memória MFC (*Memory Flow Controller*).

Além do tipo de aplicação a que se destinam, o PPE e os SPEs possuem ainda duas diferenças capitais. Primeiramente, os respectivos conjuntos de instruções não são compatíveis. Ou seja, o fluxo de desenvolvimento necessita de versões diferentes para compiladores e ferramentas binárias. Segundo, enquanto o PPE acessa memória compartilhada através de instruções de leitura e escrita, os SPEs necessitam executar operações de DMA através de seus respectivos MFCs, uma vez que as operações de leitura e escrita emitidas por um SPE afetam apenas sua memória local (LS). Além de DMA, os MFCs também permitem a comunicação direta entre os processadores (PPE e SPEs) através dos mecanismos de *sinal* e *caixa de mensagem* (*mailbox*).

Como uma operação de DMA tem um custo bem maior do que simples leituras e escritas, é importante que seja usada de forma eficiente. Idealmente, um SPE emitiria uma operação de DMA de forma assíncrona e, enquanto a operação é processada, continuaria com algum processamento efetivo, desta forma sobrepondo comunicação com computação. As operações de DMA geralmente tomam centenas de ciclos. Este tempo é, em geral, proporcional ao tempo de penalidade por falta na L2 do PPE [31].

4.2.2 Cache gerenciada por software

Devido ao número de núcleos disponíveis, à numerosa quantidade de registradores por SPE e à necessidade de sobrepor processamento e comunicação nas operações de DMA, vários modelos de programação são suportados para o processador Cell/B.E. (a referência [60] sumariza alguns). Dentre eles, o modelo de programação com memória compartilhada é atrativo dado seu emprego na maioria das arquiteturas CMP homogêneas. No entanto, tal modelo geralmente apresenta elevado nível de dificuldade no Cell/B.E., em razão da diferenciação entre memória global (usada pelo PPE) e memória local (usada pelos SPEs). Para diferenciar os endereços globais dos locais, os termos *endereço efetivo* (*Effective Address* – EA) e *endereço local* (*Local Storage Address* – LSA) são usados, respectivamente.

Para que seja possível adotar o modelo de programação com memória compartilhada no Cell/B.E., um SPE necessita usar o mecanismo de DMA provido pelo seu respectivo MFC para acessar a memória global. Assumindo que o EA seja conhecido pelo SPE (e que o mesmo espaço de endereçamento seja usado por todos os elementos), uma operação de leitura compartilhada requer: (i) uma operação de DMA para trazer o valor contido no EA para a memória local; (ii) uma operação de leitura para mover o valor da memória local para um registrador de destino. Da mesma forma, uma operação de escrita compartilhada primeiro requer que o dado seja movido de um registrador para a memória local e, em seguida, que este seja movido para o EA na memória global através de outra operação de DMA. O PPE, por sua vez, acessa a memória através de instruções convencionais de leitura e escrita.

Note, portanto, que o programa a ser executado no SPE precisa manusear operações de DMA para cada acesso à memória compartilhada. Essa característica, aliado ao tamanho reduzido da LS, torna a programação dos SPEs complexa e fadada a erros. Modelos mais abstratos que livrem o programador dos detalhes de baixo nível são altamente necessários. Uma alternativa é usar uma técnica conhecida por *cache gerenciada por software* (*Software-Managed Cache* – SMC). A ideia principal consiste em usar o LS como uma cache para a SPE. O programador acessa a memória compartilhada através da API (*Application Programming Interface*) provida pela SMC, que consiste de operações como leitura, escrita e descarga (*flush*) da cache. Todo o aspecto de comunicação via operações de DMA é encapsulado e isolado do programador pela implementação da SMC. Além de facilitar a programação, uma SMC pode também melhorar o desempenho da aplicação consideravelmente nos casos em que os acessos aos dados mostrem localidade temporal ou espacial.

É possível encontrar na literatura alguns trabalhos descrevendo implementações e técnicas para aumento da eficiência de SMCs para o Cell/B.E. [12, 20, 42, 110]. É importante notar neste ponto que uma SMC não garante consistência dos dados entre diferentes SPEs. Desta forma, um dado lido da posição compartilhada P, modificado e mantido lo-

calmente na SMC de um SPE, não é visto por outros SPEs. Ou seja, uma solicitação de leitura do mesmo endereço P por um segundo SPE não vai retornar o valor alterado pelo primeiro.

4.2.3 TL2

O sistema TL2 [27] é amplamente considerado o estado da arte de implementações em software de memória transacional [116, 80]. Esse sistema trabalha usualmente com granularidade de palavras, usando versionamento diferido e detecção de conflitos otimista. TL2 ainda usa o conceito de *relógio global* para garantir a consistência da execução especulativa e evitar transações zumbis.

Os metadados usados por TL2 são apresentados na Figura 4.2. O relógio global (`GCLOCK`) ① é um campo compartilhado (lido e escrito) por todas as transações e o meio pelo qual a consistência da execução é mantida. Cada transação mantém localmente um descritor ② armazenando seu *status* atual, o tempo lógico inicial (versão) e os conjuntos de dados lidos e escritos. Cada elemento do conjunto de leitura é um endereço de memória lido pela transação ③. Além do endereço, cada elemento do conjunto de escrita ④ precisa armazenar ainda o valor a ser escrito ⑤, já que a atualização é só feita no momento da efetivação. Os campos de endereço são mapeados através de uma função *hash* para um registro de posse em uma tabela compartilhada de versões ⑥, conhecida como *tabela de registros de posse* (*Ownership Record Table* – ORT) ⑦. Toda posição de memória lida e/ou escrita pela transação é mapeada para alguma entrada nessa tabela. O conteúdo de cada elemento da tabela depende do bit menos significativo, o *bit de trava* ⑧. Se o bit de trava for zero, então o conteúdo representa a versão corrente de todas as posições de memória mapeadas para a entrada. Caso contrário (bit menos significativo é um), a entrada está bloqueada e o seu conteúdo é um ponteiro para o descritor da transação que tem posse da respectiva entrada da tabela ⑨.

Os algoritmos para as principais primitivas de TL2 são discutidos a seguir. As primitivas de início de transação (`TxStart`) e escrita (`TxStore`) são apresentadas pelos algoritmos 4.1 e 4.2, respectivamente. Ambas são relativamente simples: no início, o tempo lógico `GCLOCK` é salvo no descritor da transação e servirá para validar cada leitura feita no decorrer de sua execução. A primitiva de escrita simplesmente adiciona no conjunto de escrita da transação o endereço de memória e o respectivo valor que lhe será atribuído durante a fase de efetivação.

As ações executadas a cada leitura transacional são descritas pelo algoritmo 4.3. Primeiramente é verificado se o endereço já está contido no conjunto de escrita (linha 1). Se este for o caso, então o último valor escrito no endereço é retornado (linha 2). Em seguida, o registro na ORT do endereço a ser lido é salvo em *orecPre* (linha 3). O valor

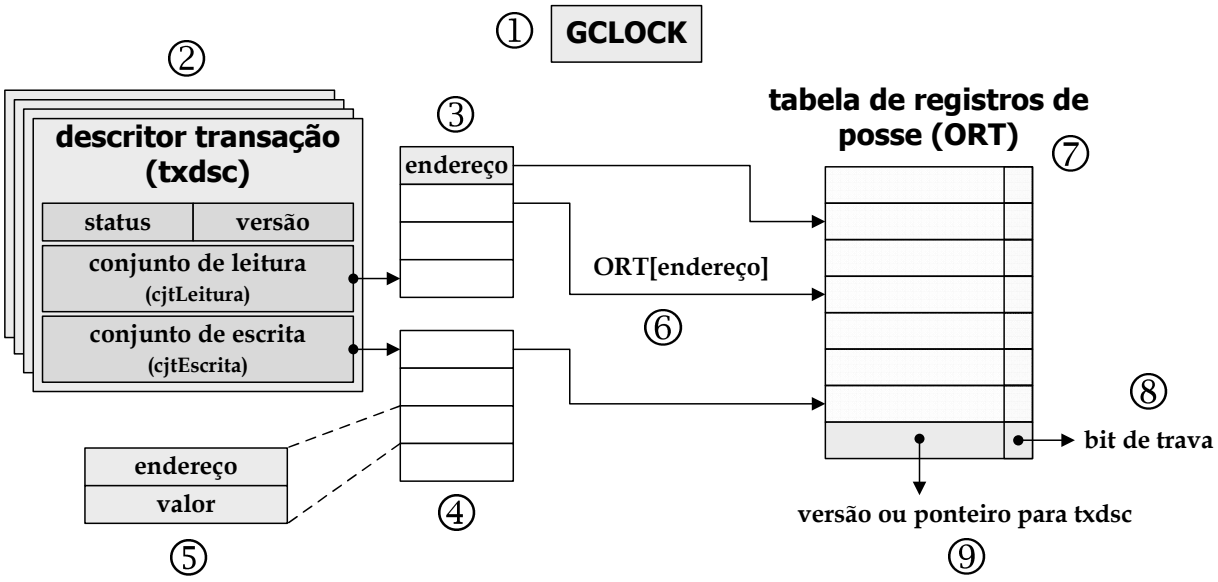

Figura 4.2: Metadados usados por TL2

Algoritmo 4.1 TxStart()

```
1: txdsc.versao ← GCLOCK
```

Algoritmo 4.2 TxStore(endereco, valor)

```
1: cjtEscrita.insere(endereco, valor)
```

então é lido da memória (linha 4) e novamente o respectivo registro é acessado na ORT (linha 5), sendo salvo em *orecPos*. Esses passos são necessários porque o valor na memória pode ter sido alterado entre a leitura do registro na ORT (linha 3) e a leitura do valor da memória (linha 4), requerendo assim uma outra leitura subsequente (linha 5). Se o registro na ORT não estiver bloqueado, os registros pré e pós leitura (*orecPre* e *orecPos*) forem iguais (indicando que o valor lido não foi alterado entre as linhas 3 e 4) e, ainda, a versão do endereço for menor ou igual ao tempo lógico inicial da transação (linha 6), então o endereço é adicionado ao conjunto de leitura e o valor lido retornado (linhas 7 e 8). Caso contrário, a transação é abortada (linha 10).

Finalizando, o algoritmo 4.4 apresenta os passos executados durante a efetivação da transação. O laço das linhas 1–5 é responsável por bloquear todos os registros na ORT relativos ao conjunto de escrita da transação. Para isso, primeiro é verificado se tal registro ainda não foi bloqueado e, caso tenha sido, se não foi a própria transação que o fez (o mesmo endereço pode aparecer mais de uma vez no conjunto de escrita). Após ter todo o conjunto de escrita bloqueado, a transação incrementa atomicamente o relógio global (linha 6) e valida o conjunto de leitura através do laço das linhas 7–9. A validação consiste

Algoritmo 4.3 TxLoad(endereco)

```
1: se cjtEscrita.contem(endereco) então
2:     retorna cjtEscrita.valor(endereco)
3: orecPre ← ORT[endereco]
4: valor ← MEMORIA[endereco]
5: orecPos ← ORT[endereco]
6: se orecPos não bloqueado E orecPos == oredPre E orecPos <= txdsc.versao então
7:     cjtLeitura.insere(endereco)
8:     retorna valor
9: senão
10:    cancela transação
```

em checar se a respectiva entrada não está bloqueada por outra transação ou se a versão do registro é maior que o tempo lógico da transação (linha 8). Se a validação falhar a transação é cancelada (linha 9). Caso contrário, a transação está logicamente efetivada e o intervalo entre as linhas 9 e 10 é o ponto de linearização da transação.

O laço das linhas 10 e 11 muda efetivamente a memória do sistema com os valores presentes no conjunto de escrita da transação e, como último passo, o laço das linhas 12–14 atualiza a versão de cada registro modificado e o desbloqueia. Note que, embora não esteja explícito no algoritmo, a operação de bloqueio do conjunto de escrita deve ser implementada através de uma instrução do tipo `CAS`. Caso a instrução falhe a transação deve ser abortada para evitar *deadlock*.

4.3 CellSTM

CellSTM é uma adaptação da TL2 para a arquitetura Cell/B.E.. As principais decisões da arquitetura de software se referem à forma como os metadados e primitivas transacionais são distribuídos entre os elementos da arquitetura. Estas decisões são essencialmente guiadas pelo fator desempenho, visto que a movimentação dos dados entre memória local e memória do sistema tem papel fundamental na implementação de quase todas as primitivas transacionais.

Considere, como exemplo, as possibilidades de implementação para a primitiva `TxLoad` apresentada pelo algoritmo 4.3. Dado que a primitiva potencialmente requer três acessos à memória do sistema, aparentemente seria mais viável fazer com que um SPE solicitasse tal serviço ao PPE e aguardasse o resultado, evitando as operações custosas de DMA. Há, no entanto, alguns fatores que impedem esse tipo de abordagem. Primeiramente, existem somente duas *threads* de execução concorrentes no PPE, limitando severamente

Algoritmo 4.4 TxCommit()

```
1: para cada elemento elem em cjtEscrita faça
2:     se ORT[elem.endereco] bloqueado E ORT[ele.endereco] ! = txdsc então
3:         cancela transação
4:     senão
5:         bloqueia ORT[elem.endereco]
6: novoGClock ← incrementoAtomico(GCLOCK)
7: para cada elemento elem em cjtLeitura faça
8:     se (ORT[elem.endereco] bloqueado E ORT[ele.endereco] ! = txdsc) OU
       ORT[elem.endereco].versao > txdsc.versao então
9:         cancela transação
10: para cada elemento elem em cjtEscrita faça
11:     MEMORIA[elem.endereco] ← elem.valor
12: para cada elemento elem em cjtEscrita faça
13:     ORT[elem.endereco].versao ← novoGClock
14:     desbloqueia ORT[elem.endereco]
```

o paralelismo quando todos os SPEs estiverem ativos. Além disso, como as operações de leitura são dominantes, tal abordagem aumentaria excessivamente o nível de contenção no PPE e faria com que os SPEs ficassem a maior parte do tempo ociosos, esperando resposta do PPE. Neste caso, a abordagem preferida (e implementada) consiste em deixar a cargo do SPE a realização de todo o algoritmo de leitura transacional.

Agora considere o caso da primitiva de efetivação (algoritmo 4.4). Neste caso em particular, os acessos à memória do sistema são constantes e, além disso, muitas construções de controle de fluxo são empregadas. Como a operação de efetivação é somente realizada uma única vez por transação e, como um SPE apresenta problemas de desempenho com instruções que alterem o fluxo do programa, a abordagem adotada consiste em deixar a cargo do PPE a realização da efetivação. Observe que o problema de contenção ainda pode ocorrer, mas espera-se que neste caso em menor proporção.

As seções a seguir detalham as decisões tomadas no projeto do CellSTM para os metadados e primitivas transacionais.

4.3.1 Metadados

Os metadados são organizados da seguinte forma: o `GCLOCK` e a ORT residem na memória do sistema, enquanto o descritor da transação reside na memória local (LS) de cada SPE. A ORT contém 1M registros e consome um total de 4MB (4 bytes por registro). Os conjuntos de leitura e escrita são armazenados localmente na LS, possuindo espaço para

2048 e 256 elementos, respectivamente. A implementação corrente aborta a execução caso esse tamanho limite seja excedido. Como o espaço da LS é limitado, uma solução mais geral para o problema do transbordamento seria migrar os conjuntos para a memória do sistema. É importante frisar, no entanto, que em nenhum dos experimentos realizados houve transbordamentos. Pesquisas revelaram que os tamanhos usados aqui são suficientes para a grande maioria das transações [22, 83].

A granularidade do versionamento usada por CellSTM é de 128 bytes, correspondendo ao tamanho da linha de cache do PPE. A principal justificativa para essa decisão é que uma SPE precisa realizar uma operação de DMA para buscar qualquer dado na memória do sistema. É sabido que o desempenho máximo é obtido quando o tamanho de uma transferência DMA é múltiplo de 128 [59].

Por fim, a implementação atual faz uso da SMC provida com o Cell SDK (*Software Development Kit*) [58]. Duas estruturas principais compõe a SMC: um *vetor de rótulos* (*tag array*) e um *vetor de dados* (*data array*). A cache é associativa por conjunto com 4 posições. Cada elemento do vetor de rótulos armazena a marca que identifica unicamente um determinado endereço dentro do conjunto, os bits de status e ainda o deslocamento dentro do vetor de dados que contém o valor associado ao endereço. O tamanho da linha de cache é de 128 bytes (como a granularidade usada por CellSTM) e existem um total de 64 conjuntos. As operações da SMC foram estendidas com suporte transacional conforme discutido na próxima seção.

4.3.2 Modelo de execução e primitivas transacionais

O modelo de execução adotado por CellSTM para uma aplicação transacional é ilustrado pela Figura 4.3. A aplicação é iniciada como uma única *thread* de execução no PPE ①. Essa *thread* é responsável por ler os parâmetros da linha de comando da aplicação, inicializar o ambiente de execução e disparar novas *threads* PPE ②. Cada *thread* PPE disparada é responsável por gerenciar um contexto de execução em um SPE ③, formando *pares de threads*. A comunicação entre cada par de *threads* é feita através do mecanismo de caixa de mensagem presente na arquitetura Cell/B.E. ④. Esse mecanismo é bloqueante e permite que uma mensagem de 32 bits seja transferida entre o PPE e o SPE.

As *threads* PPE de cada par são responsáveis por fornecer serviços de auxílio aos SPEs, como por exemplo alocação e liberação de memória dinâmica do sistema. A aplicação transacional é distribuída e executada pelos SPEs. Como o PPE tem suporte em hardware para somente duas *threads* simultâneas, é importante que as requisições feitas pelos SPEs sejam reduzidas, não provocando dessa forma sobrecarga de tarefas no PPE. Uma *thread* PPE geralmente fica em estado dormente e só é acordada quando a *thread* SPE do par requisita explicitamente alguma operação via caixa de mensagem. Após a execução da

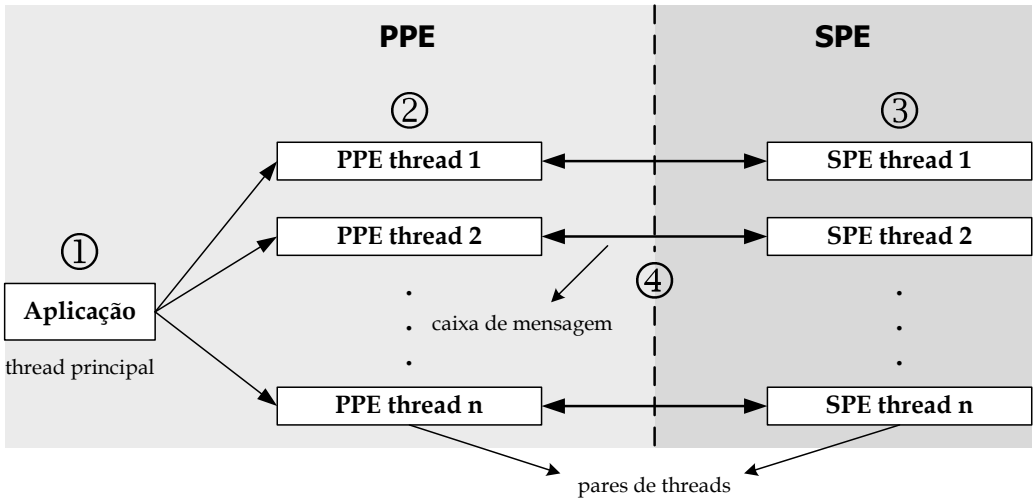

Figura 4.3: Modelo de execução do CellSTM

operação, a *thread* PPE volta novamente ao estado dormente.

As primitivas transacionais para início de transação (`TxStart`) e as barreiras de leitura e escrita (`TxLoad` e `TxStore`) são executadas totalmente no SPE. Os algoritmos são similares aos algoritmos 4.1, 4.2 e 4.3 já apresentados, a principal diferença sendo que todos os acessos ao estado compartilhado são realizados através de operações de DMA. Ou seja, os acessos ao `GCLOCK` em `TxStart` e à memória do sistema em `TxLoad` (linhas 3–5 do algoritmo 4.3) geram transações de DMA para trazer o conteúdo da RAM do sistema para a memória local do SPE.

Devido à grande similaridade entre as operações de leitura e escrita de uma SMC e as barreiras transacionais de leitura e escrita (por exemplo, as assinaturas das operações são idênticas), é comum pensar no sistema CellSTM como uma extensão da SMC provendo coerência de memória através de transações. De fato, a implementação atual estende a SMC disponibilizada junto ao Cell SDK. O principal aspecto a ser observado quando da extensão das operações de leitura e escrita em uma cache típica se refere às faltas na cache provocadas por tais operações. Quando uma falta de leitura ocorre, um valor novo é buscado na memória do sistema e também deve ser adicionado ao conjunto de leitura da transação após ser devidamente validado. Ainda, se a falta implicar a retirada de alguma linha da cache e subsequente escrita da linha na memória do sistema (por estar em estado modificado), então tal linha deve ser inserida no conjunto de escrita da transação. Por fim, a inicialização e finalização de cada transação necessita que a cache seja descarregada e que todas linhas sejam invalidadas.

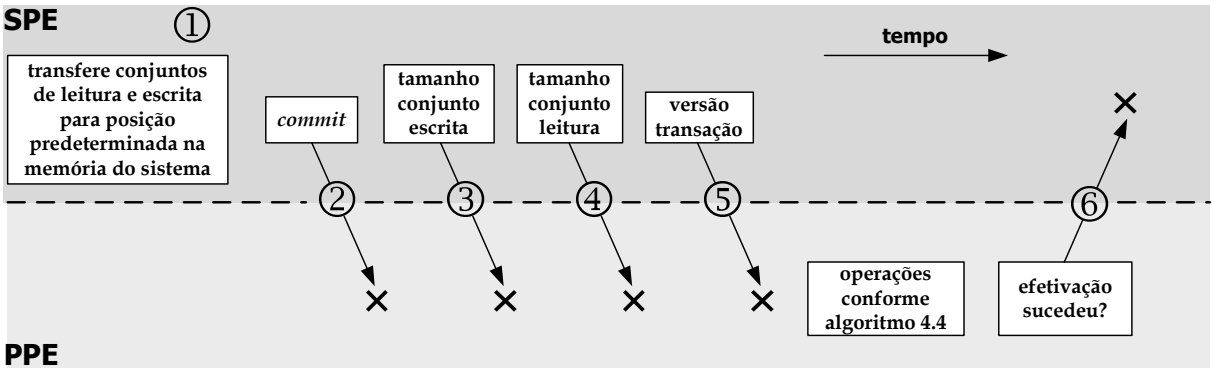

Figura 4.4: Procedimento de efetivação no CellSTM

Efetivação de transações

Um dos fatores mais críticos na implementação da CellSTM é o procedimento de efetivação. Como pode ser evidenciado pelo respectivo algoritmo apresentado na seção 4.2.3 (algoritmo 4.4), o processo exige sucessivos acessos à memória do sistema, além de possuir muitas construções de controle, as quais não são executadas eficientemente no SPE. Desta forma, o procedimento de efetivação é realizado em conjunto entre o SPE e o PPE, usando o esquema de pares de *threads* introduzido anteriormente.

A Figura 4.4 mostra a sequência das atividades que constituem o processo de efetivação e as tarefas executadas por cada um dos componentes envolvidos. No primeiro passo da efetivação, o SPE transfere todo o conteúdo dos conjuntos de leitura e escrita da transação para uma área reservada na memória do sistema ①. Essa área é alocada pelo PPE e comunicado ao SPE no início da transação. Em seguida, o SPE envia uma sequência de mensagens para o PPE informando que o processo de efetivação deve ser iniciado ②, o tamanho do conjunto de escrita ③ e leitura ④, e ainda a versão da transação para checagem de consistência ⑤. O PPE recebe cada uma das mensagens e as armazena localmente (indicado pelo `X` na figura). Após receber todas as informações necessárias à efetivação, o PPE então executa o algoritmo 4.4. Durante esse tempo, o SPE fica bloqueado em estado de baixo consumo de energia até que o PPE envie uma mensagem indicando o êxito ou não da efetivação ⑥. Em caso afirmativo, nada mais necessita ser feito pelo SPE e a execução do programa prossegue normalmente. Do contrário, a transação é abortada e reiniciada.

4.4 Resultados experimentais

O ambiente de execução no qual os resultados desta seção foram gerados é composto por um Sony PlayStation 3 rodando a distribuição Fedora 7 com o sistema operacional

Linux e o kit de desenvolvimento Cell SDK 3.1. A frequência de operação da plataforma Cell/B.E. utilizada é de 3.2GHz e, do total de 8 SPEs originalmente disponíveis, somente 6 estão disponíveis para software de aplicação. Um dos aceleradores é desabilitado de fábrica pela Sony para aumentar o aproveitamento (*yield*) no processo de fabricação. O outro acelerador é reservado e de uso exclusivo do sistema operacional.

Duas categorias de aplicações são usadas nos experimentos. Na primeira são empregadas microaplicações que objetivam estressar o sistema transacional e verificar seu comportamento sob condições extremas. Note, desta forma, que tais microaplicações não representam aplicações típicas encontradas em sistemas reais. A segunda categoria é composta pela aplicação Genoma, usada no sequenciamento de genes. Genoma é usada para verificar o comportamento esperado de CellSTM no caso de aplicações relativamente complexas.

Em todos os experimentos apresentados a seguir são usadas duas versões para cada aplicação: uma transacional e outra com trava global. A versão transacional é usada para averiguar o comportamento de CellSTM, enquanto a outra versão mostra o comportamento esperado da aplicação quando codificada com uma trava global ao invés de transações. A versão transacional é instrumentada explicitamente (primitivas `TxStart`, `TxLoad`, `TxStore` e `TxCommit`), enquanto a versão com trava equivalente é obtida simplesmente substituindo cada par `TxStart` e `TxCommit` pelas instruções de obtenção e liberação de uma trava global, respectivamente. Ainda, as barreiras de leitura e escrita são omitidas e substituídas pelas instruções equivalentes da própria arquitetura.

Os resultados apresentados nas próximas seções representam a média de 5 execuções. O desvio padrão para as microaplicações e Genoma ficou abaixo de 10% e 4% do valor da média, respectivamente.

4.4.1 Microaplicação

A microaplicação utilizada nos primeiros experimentos é semelhante à adotada por Herlihy *et al.* [54]. Um conjunto de inteiros, chamado `IntSet`, é mantido em memória compartilhada e dois tipos principais de operações podem ser invocados sobre esse conjunto: (1) atualização – uma *thread* pode solicitar a *inserção* ou *remoção* de um elemento no conjunto; (2) inspeção – um dado elemento é *buscado* dentro do conjunto. Note que uma operação de atualização necessita alterar o estado compartilhado, enquanto uma de inspeção somente efetua leituras.

Duas versões diferentes do conjunto de inteiros são empregadas nos experimentos. A primeira, chamada de `IntSet-LL`, implementa as operações de atualização e inspeção usando uma estrutura de lista ligada. Já a segunda, chamada aqui de `IntSet-HT`, é implementada através de uma *hashtable*. Cada execução de `IntSet` pode ser configurada

através dos seguintes parâmetros:

- *Número de operações:* total de operações (atualização e inspeção) a ser realizada no experimento;
- *Faixa de valores:* faixa válida de valores usada nas operações de atualização e inspeção. Quando uma operação é executada, o valor passado como parâmetro é escolhido aleatoriamente dentro desta faixa;
- *Tamanho inicial:* número inicial de elementos pertencentes ao conjunto de inteiros. Esse conjunto é populado pela *thread* principal antes das *threads* SPEs serem criadas;
- *Porcentagem de operações de atualização/inspeção:* este parâmetro permite variar a razão entre as operações que escrevem e as que somente leem estado compartilhado. Desta forma é possível analisar como o sistema escala conforme o nível de contenção é aumentado.

A métrica usada para reportar os resultados do `IntSet` é a de operações executadas na unidade de tempo. Para cada experimento, o número de operações é escolhido e mantido fixo, e o tempo total gasto na execução é anotado. A faixa de valores usada em todos os experimentos é 2^{16}. Os valores para o tamanho inicial do conjunto e a porcentagem de atualizações possuem duas configurações: 128 e 768 para o tamanho, e 20% e 50% para atualização. Essa variação permite averiguar o comportamento do sistema em cenários em que a computação exigida por uma operação e o grau de contenção variam.

`IntSet-LL`

O primeiro par de resultados é mostrado na Figura 4.5 para `IntSet-LL`, usando um tamanho inicial de 128 elementos. O gráfico (a) reporta os resultados para a taxa de atualização de 20%, enquanto no gráfico (b) uma taxa de 50% é usada. Note que em ambos os casos a implementação com STM tende a escalar, principalmente até 4 SPEs, enquanto a baseada em trava não. No cenário (a), com apenas 20% de atualizações, a versão transacional sempre apresenta melhora já que muitas transações apenas leem o estado compartilhado e podem, desta forma, ser executadas concorrentemente. O mesmo não ocorre no caso com trava, já que a trava global impede qualquer tipo de paralelismo.

No caso específico do cenário (b), a versão transacional apresenta uma queda de desempenho a partir de 5 SPEs. Neste cenário, o nível de contenção no sistema é grande já que existem muitas transações (em torno da metade do total) efetuando operações que alteram o estado compartilhado simultaneamente. Desta forma, a taxa de cancelamento de transações é muito alta, fazendo com que o número de operações executadas por segundo decresça.

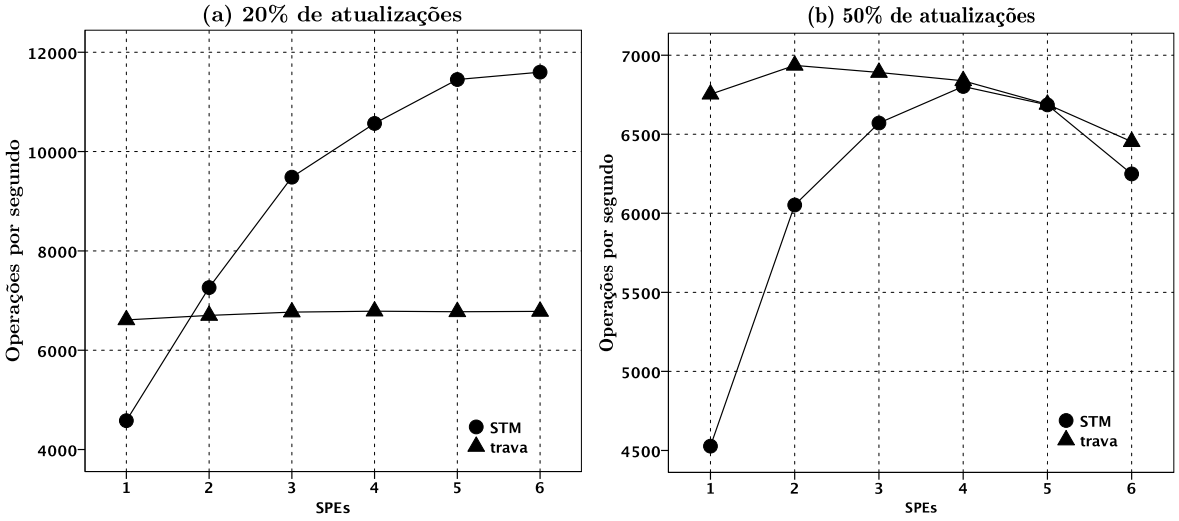

Figura 4.5: `IntSet-LL` com um conjunto inicial de 128 elementos

Note ainda que para o caso com apenas um SPE, a versão com trava é superior porque o custo de instrumentação das leituras e escritas pela STM é grande. Conforme mais SPEs são usados, mais trabalho pode ser executado em paralelo pelas transações de forma especulativa, enquanto o uso da trava global restringe o paralelismo. Especificamente no cenário (a), a versão transacional supera a com trava a partir de 2 SPEs.

Os resultados reportados pela Figura 4.6, ainda para `IntSet-LL`, são para um tamanho inicial de 768 elementos. Primeiramente observe que no cenário (a), com 20% de atualizações, a versão transacional ainda segue a mesma tendência anterior, mas agora supera a implementação com trava a partir de 3 SPEs. Novamente pode ser notado que o sistema continua a escalar nessa condição.

Já no cenário (b), com uma taxa de 50% de atualizações, a versão transacional não chega a superar a versão com trava global, embora continue escalando. A partir de 4 SPEs essa tendência diminui mas, ao contrário do que foi exposto anteriormente para o caso da Figura 4.5 (b), o sistema não apresenta queda no número de operações. Embora a taxa de cancelamentos ainda cresça, o fato do tamanho do conjunto ser maior faz com que o tempo médio efetivo de execução gasto pela transação também aumente, amortizando o custo das primitivas transacionais e não causando um impacto tão negativo quanto no caso anterior.

`IntSet-HT`

O próximo conjunto de gráficos, dado pelas Figuras 4.7 e 4.8, é relativo à aplicação `IntSet-HT`. Na Figura 4.7, a *hashtable* é populada com 128 elementos. Note que nenhuma

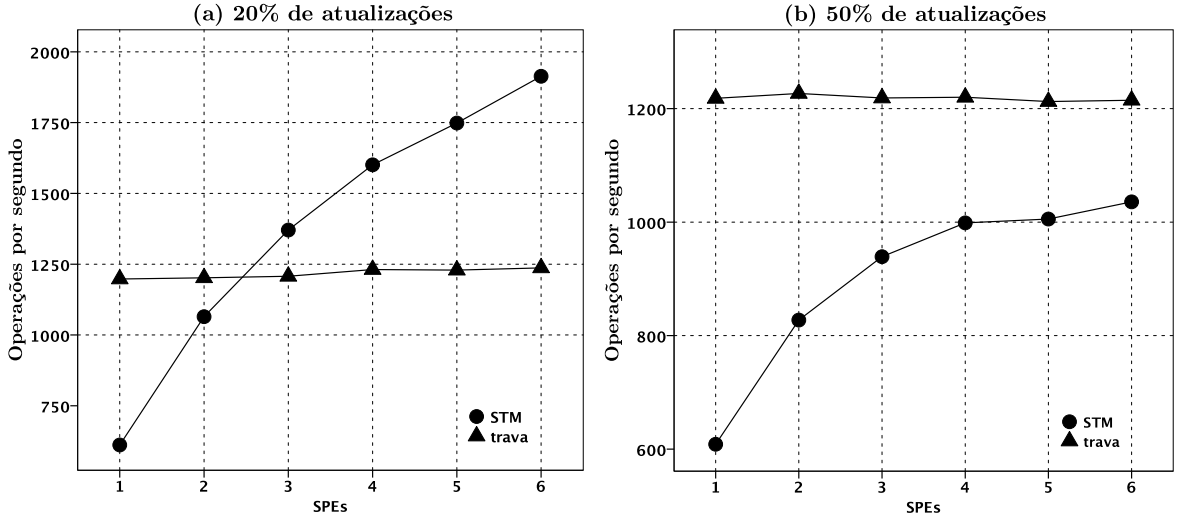

Figura 4.6: `IntSet-LL` com um conjunto inicial de 768 elementos

das implementações escalam: enquanto a implementação com trava tem ligeiro ganho de 1 para 2 SPEs, a com STM praticamente se mantém estável quando a taxa de atualização usada é de 20% (a) e tem uma pequena perda de desempenho se a taxa é de 50% (b).

A situação é praticamente a mesma quando a *hashtable* é populada com 768 elementos, como mostrado na Figura 4.8. O principal motivo para a diferença de desempenho observada entre as implementações com trava e STM está no esforço computacional de cada operação: como as operações de inserção, remoção e busca são rápidas na *hashtable*, o tempo gasto para iniciar e finalizar uma transação domina o tempo total. Na prática, espera-se que uma transação tenha uma granularidade maior do que a exibida pelas operações da *hashtable*, de forma tal que o custo da inicialização e finalização possa ser amortizado. De fato, o próximo experimento destaca essa característica através da aplicação Genoma. Uma das estruturas de dados usada por essa aplicação é justamente a *hashtable*.

4.4.2 Genoma

A aplicação Genoma usada neste experimento é uma versão da implementação distribuída junto com o pacote STAMP [83]. O objetivo do algoritmo é combinar uma série de segmentos de genes de forma a reconstruir o gene original. Os parâmetros de entrada da aplicação são:

- *Tamanho do gene* (`-g`): inicialmente, um *gene de referência* é formado aleatoriamente. O tamanho desse gene, ou seja, o número de nucleotídeos que o compõe, pode ser especificado através desse parâmetro;

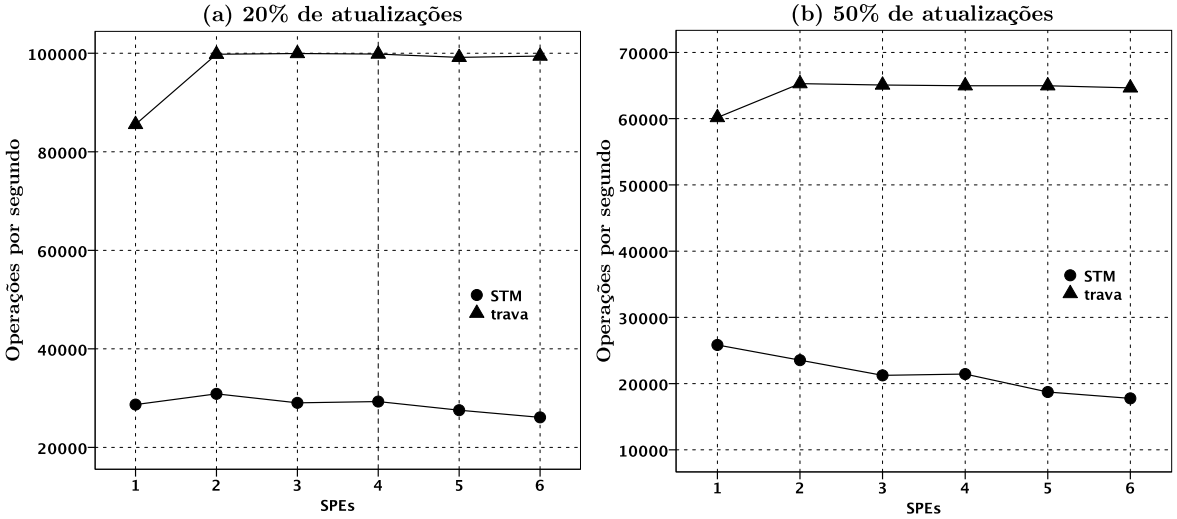

Figura 4.7: `IntSet-HT` com um conjunto inicial de 128 elementos

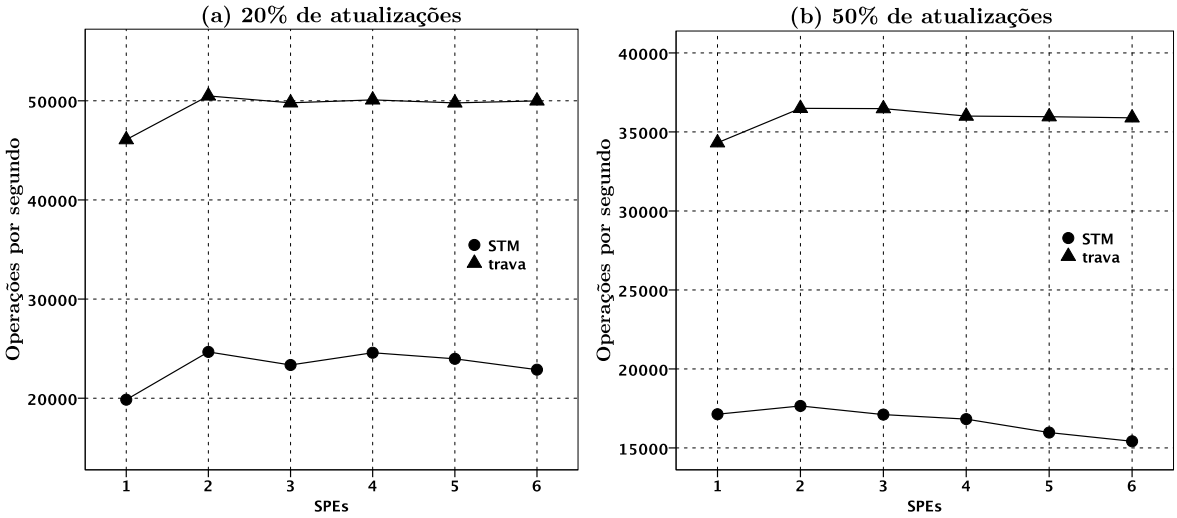

Figura 4.8: `IntSet-HT` com um conjunto inicial de 768 elementos

- *Número de segmentos* (`-n`): especifica a quantidade de segmentos de genes que devem ser usados no processo de sequenciamento. Os segmentos também são gerados aleatoriamente, sendo garantido que exista pelo menos uma combinação que reproduza o gene de referência;

- *Tamanho do segmento* (`-s`): determina o tamanho de cada segmento usado no processo de recombinação.

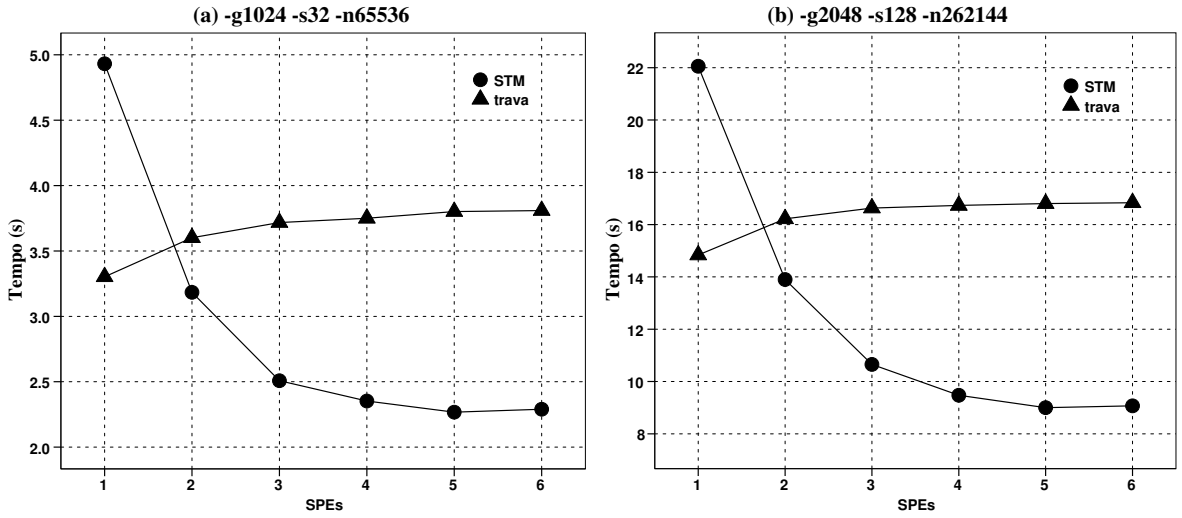

Figura 4.9: Resultados do Genoma para duas configurações

A Figura 4.9 mostra o resultado do Genoma para duas configurações diferentes, apresentadas pelos gráficos (a) e (b). Ambos mostram o tempo de execução da aplicação, em segundos, pelo número de SPEs usados. Note que, apesar do tempo de execução total variar de acordo com a configuração usada, a forma dos gráficos é muito semelhante. Novamente, pode ser observado que o custo extra gerado pela instrumentação do código faz com que STM perca para a versão com trava quando apenas 1 SPE é usado. No entanto, a partir de 2 SPEs a versão transacional claramente é a preferida pelo fato de escalar bem com o aumento do número de SPEs. Uma melhoria de um pouco mais de 2x é conseguida com 6 SPEs em ambas configurações.

4.5 Epílogo

O sistema CellSTM e a respectiva implementação abordados neste capítulo formam a primeira incursão do modelo transacional em arquiteturas assimétricas. Os resultados observados através de microaplicações e de uma aplicação mais complexa, Genoma, mostram-se promissores e, acima de tudo, apresentam evidências de escalabilidade do modelo transacional.

O trabalho discutido neste capítulo é resultado da colaboração com o colegas de mestrado Felipe Goldstein e Leonardo Garcia, sendo publicado em [41]. Este autor participou da idealização do algoritmo original, uma adaptação do sistema TL2 para a arquitetura Cell/B.E., e da implementação de uma nova versão das microaplicações apresentadas na Seção 4.4.1 deste capítulo. Os experimentos que geraram os resultados analisados neste capítulo foram refeitos recentemente por este autor e divergem em número absoluto dos

apresentados no artigo, embora as conclusões continuem válidas.

Capítulo 5

Um Sistema de STM Eficiente Voltado para Jogos

"Manual synchronization . . . is hopelessly intractible here [concurrency in gameplay simulation]. Transactions are the only plausible solution to concurrent mutable state."

Tim Sweeney, 2006

Neste capítulo é apresentado um sistema de STM específico para jogos. O sistema usa o conhecimento sobre o domínio da aplicação para acelerar a execução das transações. A Seção 5.1 apresenta a motivação para o trabalho. A Seção 5.2 descreve em termos gerais a abordagem adotada, enquanto as Seções 5.3 e 5.4 introduzem o modelo de programação e aspectos de implementação, respectivamente. A Seção 5.5 descreve o processo de paralelização de um jogo complexo e mostra os resultados obtidos com o sistema proposto. As considerações finais são feitas na Seção 5.6.

5.1 Motivação

O baixo desempenho das implementações em software do mecanismo transacional tem sido considerado um dos principais empecilhos contra sua popularização, conforme evidenciado na literatura recentemente [18]. Estudos com aplicações reais também sugerem que, embora facilitem a programação, o desempenho de transações em software ainda está aquém do desejado [128, 61].

Visando reduzir os custos impostos por STM, Herlihy e Koskinen [53] recentemente sugeriram uma metodologia para que objetos com alta concorrência possam ser usados transacionalmente: ao invés de detectar conflitos em granularidade baixa (usando conjuntos de leitura e escrita), conflitos passam a ser identificados entre métodos do objeto.

Embora ganhos de uma ordem de grandeza em relação à STM típicas sejam relatados, a metodologia requer implementações já eficientes para os objetos (livres de travas, por exemplo) e a existência de operações inversas para os métodos, já que uma transação pode vir a ser cancelada posteriormente.

Ao invés de apresentar uma implementação de STM para o caso geral, este capítulo adota uma abordagem alternativa: usar o conhecimento sobre o domínio da aplicação (no caso, jogos computacionais) para realizar uma implementação eficiente de STM. A solução adotada difere de implementações convencionais para STM em vários aspectos. Primeiro, as transações (aqui chamadas também de *tarefas*) nunca abortam. Isso permite que operações de entrada e saída sejam executadas normalmente dentro da transação sem as dificuldades encontradas em sistemas de STM típicos. Segundo, a consistência é mantida no nível de objetos, não no nível de transações. Ou seja, é possível realizar uma ordem de execução em que transações não sejam serializadas. E terceiro, o programador necessita especificar código para tratar os conflitos entre acessos concorrentes aos objetos compartilhados. Embora essa última característica pareça ser limitante, na prática isso não ocorre: mecanismos de resoluções para conflitos típicos já são providos pela infraestrutura.

5.2 Visão geral

Como dito, a solução apresentada neste capítulo leva em conta o conhecimento sobre o domínio da aplicação para relaxar algumas restrições e fornecer uma implementação eficiente de STM para jogos. Assim, esta seção introduz os conceitos e características importantes presentes em jogos computacionais que possibilitaram o desenvolvimento de tal STM.

A Figura 5.1 ilustra a estrutura geral de codificação de um jogo. Além de trivialmente conter código para inicialização e finalização, um jogo possui um laço principal (*laço de jogo*) executando ininterruptamente as seguintes ações: leitura dos comandos do jogador, atualização do estado do mundo e resposta aos comandos do jogador (geralmente audiovisual). O nível de complexidade de cada uma dessas ações pode variar bastante de jogo para jogo. O processamento da entrada pode considerar dispositivos sofisticados baseados em movimentos do corpo do usuário (como os existentes nos consoles Wii da Nintendo). O mundo em que o jogo se passa pode exigir modelos físicos complexos, contar com técnicas de inteligência artificial avançadas para emular o comportamento de objetos não controlados pelo jogador, e ainda levar em consideração a ação de outros jogadores que interagem remotamente através da rede. A forma mais comum de saída é via imagem e áudio. A imagem, em particular, pode exigir hardware complexo para renderização de objetos e cenários em três dimensões (3D). O jogo termina quando um objetivo específico

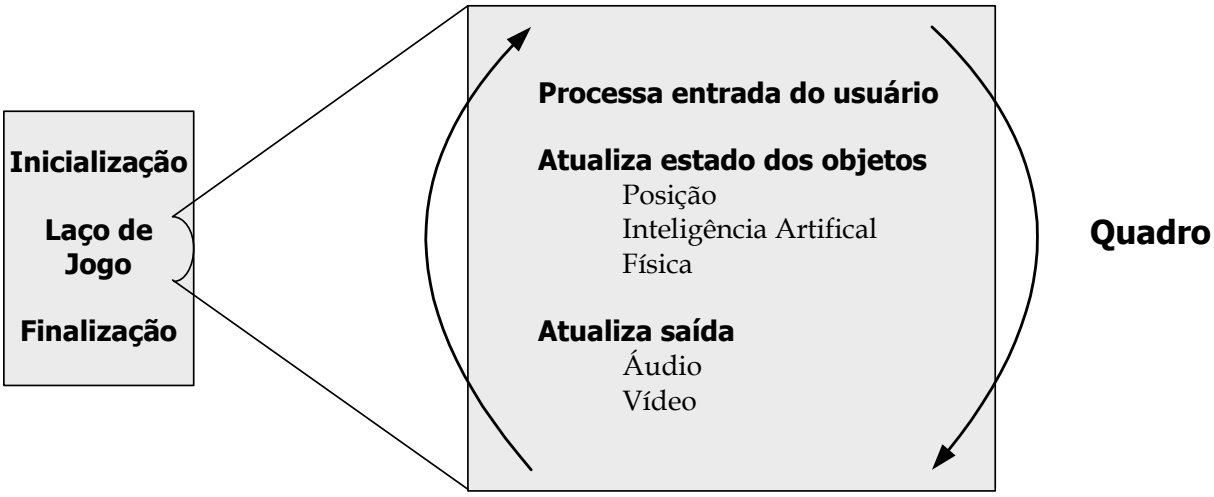

Figura 5.1: Estrutura geral de codificação de um jogo

é atingido pelo usuário ou este explicitamente cancela sua execução.

Para facilitar o desenvolvimento do texto, as seguintes definições são adotadas. Um jogo é composto por *objetos*. Ações executadas por ou sobre esses objetos são descritas por *tarefas*. Um *quadro de execução* consiste das tarefas executas durante uma iteração do laço de jogo. A velocidade com que quadros são processados é denotada pela *taxa de quadro* (*frame rate*), sendo *quadros por segundo* (*Frames Per Second* – FPS) a unidade de medição adotada.

As seguintes observações caracterizam o domínio de jogos computacionais e servem como requisitos para o desenvolvimento da STM:

- As tarefas que acessam objetos compartilhados são divididas em dois tipos, de acordo com o tipo de operação efetuada nesses objetos: (1) *leitora* – objetos compartilhados são apenas lidos, nunca modificados; (2) *modificadora* – objetos compartilhados podem ser modificados. Assim, uma tarefa que atualize a posição de um personagem no mundo é do tipo modificadora, enquanto uma outra responsável por renderizar uma cena é apenas leitora. Essa divisão é bastante comum em jogos. Por exemplo, o esqueleto de código gerado pelo arcabouço para desenvolvimento de jogos XNA da Microsoft [73] naturalmente requer que o programador especifique código para dois métodos principais: `Update` e `Draw`. O primeiro é responsável pela atualização dos estados dos objetos (e portanto corresponde a uma tarefa modificadora) e o segundo pela saída (claramente, tarefa leitora);

- Tarefas são executadas periodicamente em iterações do laço de jogo. Cada nova iteração consiste em uma nova sequência de execução dessas tarefas.

Construção	Contexto	Breve descrição
CreateReadonlyTask (task) *CreateUpdaterTask* (task)	Tarefa	Cria tarefa somente para leitura Cria tarefa para modificação
CreateBarrier (task1, task2)	Tarefa	Força ordenação entre tarefas
SharedValue <val-type> *SharedObject* <ref-type> *SharedCollection* <collection-type>	Objeto	Marcações usadas na declaração de objetos compartilhados para os tipos por valor, por referência e coleção
SharedValueWithPriority <val-type, enum> *SharedObjectWithPriority* <ref-type, enum>	Objeto	Marcações com resolução de conflito baseada em prioridades
get () *set* ()	Objeto	Métodos para leitura e escrita de variáveis cujo tipo é por valor
getReadOnly () *getForWrite* ()	Objeto	Métodos que retornam referências a objetos somente para leitura ou então escrita

Tabela 5.1: Principais construções propostas

Como essas observações orientaram a implementação da STM é explicado na Seção 5.4. Antes, porém, o modelo de programação é apresentado na próxima seção.

5.3 Modelo de programação

O sistema de STM é implementado como uma biblioteca para a linguagem `C#`. A API pode ser dividida, em termos mais gerais, entre as construções usadas para tarefas e as usadas para objetos. A Tabela 5.1 mostra as principais construções, o contexto do uso e uma breve descrição. As seções a seguir descrevem informalmente a API em termos de tarefas e objetos. Exemplos de uso são apresentados na Seção 5.3.3.

5.3.1 Tarefas

Uma aplicação deve ser pensada logicamente como um conjunto de tarefas. A primeira ação realizada pelo programador consiste em organizar o código em tarefas. É importante notar que, como uma tarefa é a unidade de execução concorrente básica, sua granularidade pode restringir a escalabilidade. O sistema de execução garante que os acessos concorrentes aos objetos compartilhados dentro de uma tarefa serão sempre consistentes, ou seja, uma tarefa é isolada das demais neste aspecto. Ao final de cada quadro, os acessos conflitantes devem ser resolvidos com a ajuda do programador (mais detalhes na Seção 5.3.2). Note que os termos *tarefa* e *transação* são usados de forma intercambiável neste contexto.

Uma vez identificadas, as tarefas devem ser registradas no sistema através do seguinte subconjunto da API:

CreateReadonlyTask (*task*)
: indica ao sistema de execução para criar uma tarefa que somente efetuará acessos de leitura ao estado compartilhado. O parâmetro *task* é simplesmente um bloco de código. Em `C#`, *task* é um tipo *delegate* que não aceita nem retorna nenhum valor. Um *delegate* é simplesmente um tipo que referencia um método.

CreateUpdaterTask (*task*)
: pede a criação de uma tarefa que pode eventualmente alterar estado compartilhado.

Com as tarefas registradas, o sistema de execução se encarrega de escalonar a execução de cada uma delas para cada quadro de forma a explorar o paralelismo. Os dados enxergados por cada tarefa em um quadro correspondem às modificações efetuadas pelas tarefas no quadro anterior. Por padrão, as tarefas são disparadas em alguma ordem não determinística. Às vezes é necessário impor alguma ordem de execução devido a dependência de dados entre tarefas. Para esses casos, o sistema de execução provê a seguinte facilidade:

CreateBarrier ($task_1$, $task_2$)
: a tarefa $task_2$ será executada somente após o término da tarefa $task_1$. Os valores alterados pela primeira são repassados para a segunda.

5.3.2 Objetos

Os objetos acessados concorrentemente por mais de uma tarefa devem ser explicitamente identificados e marcados. Como o sistema de tipos em `C#` distingue os tipos por valor dos tipos por referência, os seguintes esquemas de marcação são disponibilizados:

SharedValue $< var >$
: a variável *var*, cujo tipo é por valor, é declarada compartilhada.

SharedObject $< var >$
: a variável *var*, cujo tipo é por referência, é declarada compartilhada.

Uma variável cujo tipo é por valor é alocada diretamente na pilha. Já uma variável cujo tipo é por referência armazena apenas uma referência para o dado real, alocado no *heap* e sujeito à coleta de lixo. Além das marcações para os tipos por valor e referência, uma marcação para coleções também é introduzida por conveniência:

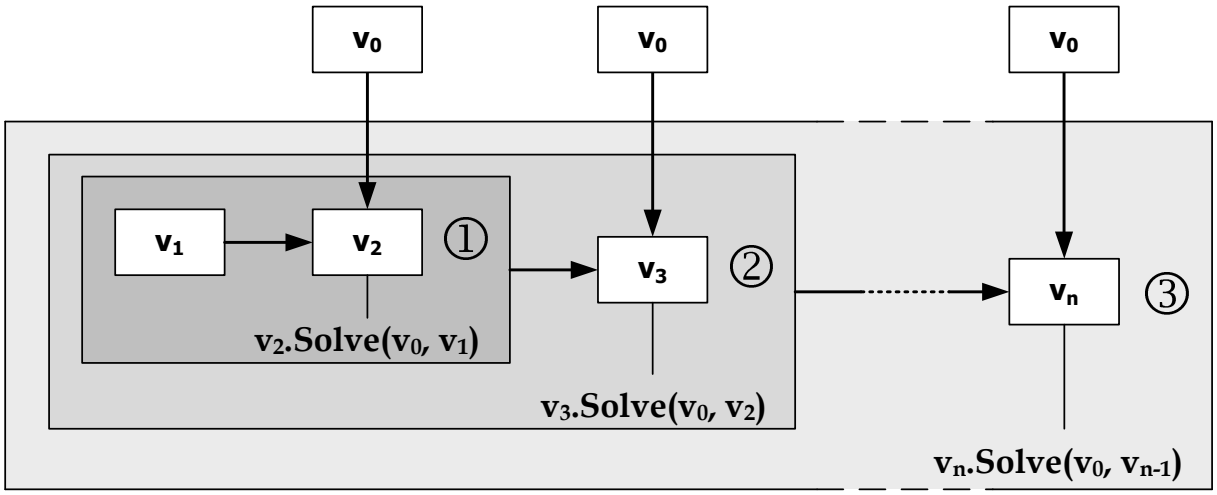

Figura 5.2: Mecanismo de resolução de conflitos padrão

SharedCollection < *var* >
: marcação usada para coleções de dados. Em `C#` toda coleção implementa a interface *ICollection*, e portanto *var* deve fazê-lo.

Antes de usar o objeto, o programador deve abri-lo explicitamente, de forma semelhante aos sistemas típicos de STM baseados em objetos. Para variáveis cujos tipos são por valor, os métodos *get* () e *set* () são usados. Já para as de tipos por referência, utilizam-se os métodos *getReadOnly* () e *getForWrite* (). Após abertos, os objetos podem ser acessados normalmente.

Ao término de cada quadro de execução, o sistema entra em uma fase denominada de *reconciliação*. Essa fase consiste em resolver conflitos de escrita a um mesmo objeto por pelo menos duas tarefas. O processo de reconciliação funciona da seguinte maneira. Cada objeto conflitante tem um conjunto de versões $V = \{v_0, v_1, v_2, ..., v_n\}$ correspondendo às versões do objeto no início do quadro, v_0, e subsequentes modificações, $V_t = \{v_1, v_2, ..., v_n\}$. As versões em V_t são réplicas de v_0, modificadas pelas tarefas $t_1, t_2, ..., t_n$. O sistema de execução invoca, para cada par de versões (v_i, v_{i+1}), $\forall i \in \{1, 2, ..., n\}$, o método *Solve*($v_0$, v_i) do objeto com a versão v_{i+1}. A resolução do conflito consiste em alterar a versão do objeto no qual *Solve* foi invocado (v_{i+1}). Essa versão é propagada para subsequentes resoluções (se existirem). A Figura 5.2 ilustra esse procedimento. Primeiramente, o conflito entre as versões v_1 e v_2 é resolvido por v_2 e sua versão repassada ①. O conflito entre v_2 e v_3 é agora resolvido por v_3 e, novamente, a versão resultante é repassada ②. O processo se repete para cada par, até que finalmente a versão resolvida final é obtida ③. Todo objeto compartilhado deve implementar uma interface chamada *ISolvable* que provê o método de resolução *Solve*(T original, T alterado), para um tipo predefinido T.

Muitas vezes os conflitos podem ser resolvidos naturalmente através de um esquema de prioridades. Para esses casos, as seguintes declarações podem ser usadas:

SharedValueWithPriority $< var, enum >$
: A resolução de conflitos para a variável *var*, cujo tipo é por valor, é feita com base na enumeração *enum*. A enumeração lista um conjunto de prioridades $P = \{p_1, p_2, ..., p_n\}$ que podem ser usadas no momento em que o objeto é alterado com *set* (). Quando da resolução de conflitos, as versões do objeto que foram alteradas com prioridade p_i automaticamente são mantidas em relação às versões com prioridade p_{i+1}.

SharedObjectWithPriority <var, enum>
: O mesmo que *SharedValueWithPriority*, mas para tipos por referência. Neste caso, *getForWrite* () é usado no lugar de *set* ().

5.3.3 Exemplos

Esta seção apresenta alguns exemplos ilustrando o uso da API apresentada na seção anterior. É importante salientar que os exemplos apresentados aqui visam puramente ilustrar os aspectos de programação e, desta forma, não necessariamente representam a melhor forma de resolver determinado problema.

No primeiro exemplo, ilustrado pela Figura 5.3, cada quadro consiste na execução de quatro tarefas distintas apresentadas na Figura 5.3b. Três dessas tarefas (linhas 1–17) atualizam o estado de três objetos e de uma variável compartilhada *maxVelocity*, a qual armazena a maior velocidade alcançada por quaisquer desses objetos. A quarta tarefa (linhas 19–21) é responsável por imprimir a velocidade máxima na tela. A Figura 5.3a mostra as declarações do tipo estruturado usado para velocidade (linhas 1–12), da variável compartilhada *maxVelocity* (linha 14), e dos registros das quatro tarefas (16–19) apresentadas na Figura 5.3b. Note que as três primeiras tarefas alteram o valor de uma variável compartilhada, e portanto são registradas como modificadoras. Já a quarta tarefa é apenas de leitura. Note ainda que a declaração da variável compartilhada na linha 14 é feita de forma explícita, através da marcação *SharedValue* (já que estruturas são de tipo por valor em `C#`).

Como a variável compartilhada *maxVelocity* pode potencialmente ser alterada por mais de uma tarefa, a fase de reconciliação muito provavelmente será executada no final de cada quadro. Para que o processo de reconciliação funcione de forma esperada, o tipo estruturado *velocity* definido na Figura 5.3a deve implementar a interface *ISolvable* (linha 1), que requer a especificação do método *Solve* (linhas 7–11). Para este exemplo, a resolução do conflito consiste em manter a maior das velocidades (linhas 9–10).

```
1  struct velocity : ISolvable<velocity>
2  {
3    float myVelocity;
4
5    float Vel() { return myVelocity; }
6
7    void Solve(velocity v0, velocity vi)
8    {
9      if (vi.Vel() > myVelocity)
10       myVelocity = vi.Vel();
11   }
12 }
13
14 SharedValue<velocity> maxVelocity;
15
16 CreateUpdaterTask(task1);
17 CreateUpdaterTask(task2);
18 CreateUpdaterTask(task3);
19 CreateReadonlyTask(task4);
```

(a) Declarações

```
1  void task1() {
2    object1.update();
3    if (object1.isMoving())
4      maxVelocity.set(object1.velocity);
5  }
6
7  void task2() {
8    object2.update();
9    if (object2.isMoving())
10     maxVelocity.set(object2.velocity);
11 }
12
13 void task3() {
14   object3.update();
15   if (object3.isMoving())
16     maxVelocity.set(object3.velocity);
17 }
18
19 void task4() {
20   show(maxVelocity.get().Vel());
21 }
```

(b) Tarefas

Figura 5.3: Exemplo de codificação para tipo por valor

O segundo exemplo, dado pela Figura 5.4, usa prioridades para resolver potenciais conflitos. O problema é resolvido por quatro tarefas, especificadas na Figura 5.4b. Neste exemplo, a posição de um objeto *bola* é atualizada a cada quadro pela tarefa 1 (linhas 1–5). A tarefa 2 (linhas 7–11) checa colisões com outros objetos no cenário (linha 9) e, se for o caso, destrói o objeto (linha 10). Note que a checagem não necessita alterar o objeto, e portanto ele é aberto somente para leitura. Só na destruição é necessário abri-lo para escrita. A terceira tarefa (linhas 13–17) lê a entrada do usuário e, se solicitado, limpa a posição da bola no cenário. A última tarefa (linhas 19–23) imprime na tela a posição da bola. Note portanto que as primeiras três tarefas podem gerar conflitos. Para este exemplo um esquema de prioridades é usado. A atualização da tarefa 1 recebeu a menor prioridade, já que os efeitos das tarefas 2 e 3 invalidam os dessa modificação. Da mesma forma, o efeito da tarefa 3 prevalece sobre o da tarefa 2, e portanto têm uma prioridade maior.

A Figura 5.4a mostra a declaração da classe para o objeto bola (linhas 1–13), a definição das prioridades como uma enumeração (linha 15), a declaração do objeto bola (linhas 17–18) e o registro das quatro tarefas (linhas 20–23).

```
1  class OBall
2  {
3    int x, y;
4    int dirX, dirY;
5    int length, width;
6
7    void moveX() { x += dirX; }
8
9    void move() { y += dirY; }
10
11   void reset() { x = 0; y = 0; }
12   }
13 }
14
15 enum Prior {high, medium, low};
16
17 SharedObjectWithPriority<OBall,Order>
18       ball;
19
20 CreateUpdaterTask(task1);
21 CreateUpdaterTask(task2);
22 CreateUpdaterTask(task3);
23 CreateReadonlyTask(task4);
```

(a) Declarações

```
1  void task1()
2  {
3    ball.getForWrite(Prior.low).moveX();
4    ball.getForWrite(Prior.low).moveY();
5  }
6
7  void task2()
8  {
9    if (collide(ball.getReadonly())
10     destroy(ball.getForWrite(Prior.medium))
11 }
12
13 void task3()
14 {
15   if (userRequested())
16     ball.getForWrite(Prior.high).reset();
17 }
18
19 void task4()
20 {
21   print(ball.getReadonly().x,
22         ball.getReadonly().y);
23 }
```

(b) Tarefas

Figura 5.4: Exemplo de codificação usando prioridades

5.3.4 Considerações

Como pode ser percebido pelas seções anteriores, o programador tem um papel ativo: deve separar o código em tarefas, marcar os objetos compartilhados, fornecer métodos para resolução de conflitos e abrir os objetos para leitura ou escrita corretamente. Algumas dessas obrigações podem ser aliviadas com a ajuda de compiladores. Por exemplo, o manuseio de objetos compartilhados (abertura para leitura/escrita) poderia ser feito automaticamente por compiladores como acontece em alguns sistemas de STM. A divisão de tarefas e marcação dos objetos compartilhados constituem importantes decisões de projeto. O modelo sugerido aqui pode ser visto como uma forma de explicitar as intenções do programador e de documentar o código, fatores importantes em engenharia de software. Os mecanismos para resolução de conflitos podem ser reutilizados em outros cenários. Raramente a descrição de um novo esquema faz-se necessária.

De particular utilidade para o processo de partição em tarefas e de marcação do estado compartilhado é o paradigma de Modelo-Visão-Controle (*Model-View-Controller* – MVC) [66]. A metodologia MVC pode ser empregada da seguinte forma. A aplicação é dividida em módulos representando diferentes aspectos do jogo (áudio, física, rede, etc). Cada módulo possui exatamente um *controlador* e uma ou mais *visões*. Os objetos alterados por mais de um módulo constituem o *modelo compartilhado* da aplicação. O objetivo

do controlador é oferecer métodos de acesso ao modelo. A visão é uma representação do módulo para o mundo externo. Por exemplo, considere o módulo de tela. A visão deste módulo é uma representação gráfica do modelo compartilhado. O paradigma MVC tem várias nuances e é necessário certa prática para ser usado de forma efetiva. A Seção 5.5 relata como MVC ajudou na partição de um jogo real.

5.4 Aspectos de implementação

Esta seção descreve as principais decisões de implementação do sistema. A principal tarefa do sistema de execução é escalonar as tarefas de forma a explorar o máximo paralelismo possível sem provocar inconsistências. A principal técnica usada para implementação consiste na replicação de dados. O mesmo conceito foi anteriormente usado por Rinard e Diniz [104] no contexto de compiladores paralelizadores para reduzir a contenção em objetos compartilhados.

Note que as tarefas podem ser um de dois tipos: leitora ou modificadora. O programador registra as tarefas através da API fornecida na fase de inicialização do jogo. A primeira abordagem então é explorar o paralelismo entre as tarefas leitoras e modificadoras. Cada objeto possui duas réplicas, chamadas aqui de versões: (1) *mestre*, acessada no quadro atual por tarefas leitoras; e (2) *rascunho*, usada pelas tarefas modificadoras para efetuar as alterações necessárias no quadro atual. No final do quadro, uma fase conhecida como *propagação* é responsável por copiar os dados alterados da versão de rascunho para a mestre. Embora simples e de fácil implementação, essa abordagem não permite nenhum paralelismo entre as tarefas modificadoras.

A implementação real usa uma extensão da abordagem apresentada anteriormente. Ao invés de usar apenas duas versões, uma versão é criada para cada tarefa modificadora. Desta forma, as modificações podem prosseguir em paralelo porque estão isoladas pelo esquema de versionamento. As tarefas leitoras continuam usando uma versão dedicada (mestre) que não é alterada durante o quadro. Quando todas as tarefas do quadro são finalizadas, o sistema entra na fase de reconciliação que visa resolver possíveis conflitos. A fase de reconciliação usa o número de acessos concorrentes armazenado em cada objeto para decidir quais precisam ser reconciliados. O mecanismo de resolução de conflitos descrito na Seção 5.3.2 é então invocado para cada objeto, e o valor resolvido é transferido para a versão mestre. Note que se não houver nenhum conflito, a fase de reconciliação simplesmente atualiza a versão mestre com as mudanças efetuadas durante o quadro. A fase de propagação é executada como última etapa do quadro. Nesta fase, as versões de todo objeto alterado devem ser sincronizadas com a versão efetivada pela fase de reconciliação. A partir de então um novo quadro é iniciado.

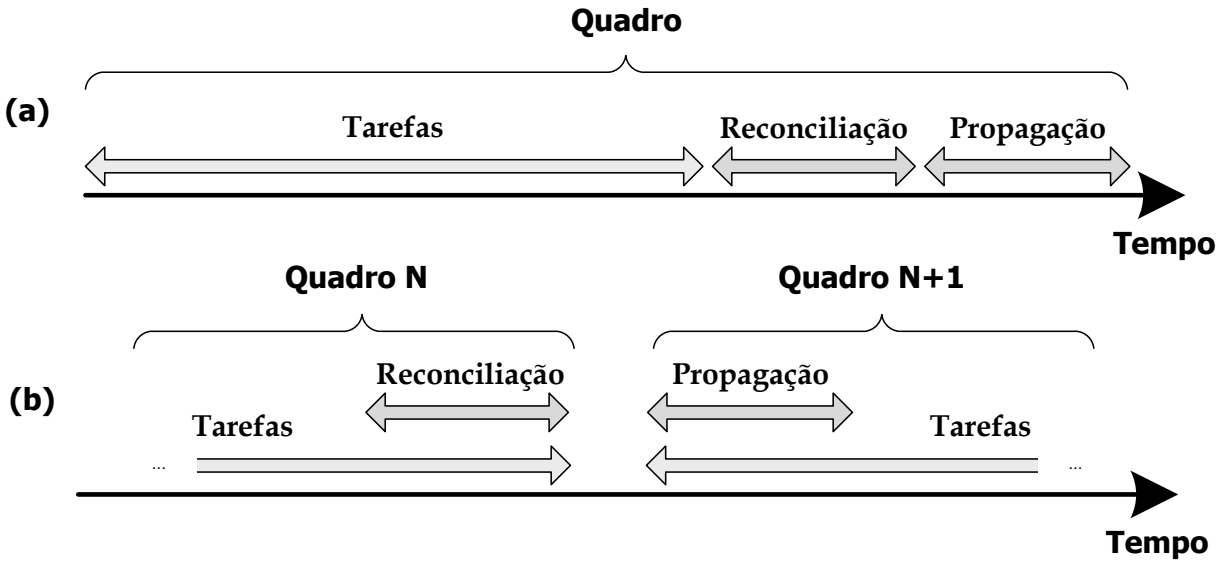

Figura 5.5: Fases de execução: (a) sequencial e (b) paralelo

5.4.1 Otimização

Na implementação descrita previamente, o sistema de execução possui três fases principais: execução das tarefas, reconciliação e propagação. Essas 3 fases são executadas sequencialmente, conforme mostra o item (a) da Figura 5.5. Além disso, a fase de propagação pode incorrer em alto custo devido ao elevado número de cópias necessárias. A otimização sugerida aqui visa explorar um potencial maior de paralelismo ao executar as fases de reconciliação e propagação em paralelo com a execução das tarefas, conforme ilustrado pelo item (b) da Figura 5.5. Uma técnica conhecida como *copy-on-write* é usada para reduzir a quantidade de cópias: ao invés de propagar as alterações no final de cada quadro, as tarefas modificadoras acessam a versão mestre para os acessos de leitura. A réplica somente é criada quando o primeiro acesso de escrita é feito. Note que a fase de propagação passa a ser executada somente no início do próximo quadro.

Para implementar a otimização, uma nova versão é adicionada a cada objeto, chamada de versão *sombra*. Na fase de reconciliação, a versão sombra é atualizada ao invés da versão mestre. A última etapa de cada quadro consiste agora em permutar as versões sombra e mestre. O novo quadro é iniciado e a fase de propagação agora é executada para sincronizar a versão sombra com a versão mestre. Como a versão sombra não é escrita por ninguém além da fase de reconciliação, a fase de propagação pode ser executada em paralelo com as tarefas do quadro. As tarefas modificadoras adotam o esquema de *copy-on-write* como descrito anteriormente. A fase de reconciliação é iniciada tão logo a fase de propagação e as tarefas modificadoras terminem.

O Algoritmo 5.1 mostra uma versão simplificada do laço principal de execução do sistema em pseudo-código. Duas listas são mantidas com as tarefas leitoras e modificadoras registradas (linhas 1–2). As versões então são alocadas (linha 3) e inicializadas (linhas 4–7). Repare que o número total de versões é igual a dois (mestre e sombra) mais uma para cada tarefa modificadora. Para simplificar o algoritmo, as versões são apresentadas como índices.

O laço principal (linhas 8–20) é iterado até que o usuário explicitamente cancele a execução. O comando `dispara` é usado para indicar a criação de uma nova *thread* que segue em paralelo (a chamada é não-bloqueante). O pseudo-código também usa o comando `espera` para indicar a suspensão de execução até que a *thread* especificada como argumento retorne (ou seja, a chamada é bloqueante). O laço principal dispara a *thread* de propagação (linha 9) para sincronizar as versões sombra e mestre. Em seguida, as tarefas leitoras (linhas 10–11), escritoras (linhas 12–13) e a de reconciliação (linha 14) são lançadas em paralelo. Note que a *thread* de reconciliação deve esperar pela finalização das *threads* com as tarefas modificadoras (não mostrado no pseudo-código) antes de iniciar o processamento. O sistema de execução então espera pela finalização das tarefas leitoras (linhas 15–16) e da fase de reconciliação (linha 17). No final do quadro, as versões sombra e mestre são permutadas (linhas 18–19).

5.5 Estudo de caso

Esta seção apresenta o processo de paralelização de um jogo real usando a API descrita na seção 5.3. O jogo é uma versão 3D do clássico *SpaceWars* (surpreendentemente chamado de *SpaceWars3D*) para a plataforma Windows, codificado em `C#`. O código original é distribuído com um livro que ensina programação de jogos em `C#` [124]. No total, o jogo é dividido em 42 arquivos que contêm cerca de 5000 linhas de código executável (excluindo linhas em branco, comentários, declarações de tipos, membros, *namespaces* etc).

De forma breve, o jogo se passa no espaço sideral onde duas naves devem atirar uma na outra. Umas das naves é controlada pelo usuário local. A outra pode ser controlada por um usuário remoto via rede, ou então pelo computador (no entanto nenhum algoritmo de inteligência artificial está implementado – a nave simplesmente fica em uma órbita predefinida). Asteroides foram introduzidos no jogo para aumentar o nível de processamento e prover maior potencial de paralelismo. A quantidade de asteroides é configurável e pode chegar aos milhares. As naves podem atirar nesses asteroides e quebrá-los em subpartes, aumentando ainda mais sua quantidade no universo do jogo.

O primeiro passo do processo de paralelização consistiu em reorganizar o código original usando o paradigma MVC, introduzido brevemente na Seção 5.3.4. O jogo foi dividido em cinco módulos principais, cada módulo originando diversas tarefas. Os módulos e as

Algoritmo 5.1 Laço principal do sistema de execução

```
1: listaLeitoras: lista de tarefas leitoras
2: listaModificadoras: lista de tarefas modificadoras
3: alocaVersoes(listaModificadoras.Tamanho)
4: versaoMestre ← 0
5: versaoSombra ← 1
6: para i = 0 até listaModificadoras.Tamanho faça
7:    listaModificadoras[i].versao = 2 + i
8: enquanto usuário não abortar execução faça
9:    dispara Propagacao(versaoMestre, versaoSombra)
10:   para cada tarefaLeitora em listaLeitoras faça
11:      dispara tarefaLeitora
12:   para cada tarefaModificadora em listaModificadoras faça
13:      dispara tarefaModificadora
14:   dispara Reconciliacao(versaoSombra)
15:   para cada tarefaLeitora em listaLeitoras faça
16:      espera tarefaLeitora
17:   espera Reconciliacao
18:   versaoSombra ← versaoMestre
19:   versaoMestre ← 1 − versaoSombra
20: fim-enquanto
```

principais tarefas são mostradas na Figura 5.6. O modelo contém os objetos compartilhados pelos diferentes módulos, como as naves, asteroides e projéteis. A visão é responsável por prover uma representação externa do jogo: o módulo de tela apresenta uma imagem gráfica, o de áudio gera som audível e o de rede gera os pacotes. Note que os módulos de física e entrada não possuem visões.

Em geral, as tarefas são implementadas pelos controladores. O modelo e as visões encapsulam estado compartilhado e dados específicos do módulo, respectivamente. Uma descrição de cada módulo é apresentada a seguir:

Tela

O componente Tela encapsula os aspectos gráficos do jogo. A tarefa de renderização processa os elementos gráficos dos objetos e se comunica com a placa gráfica através da interface provida por DirectX. A tarefa de atualização de *status* se encarrega de coletar dados como o número de asteroides no universo e a velocidade do jogo (medida em quadros por segundo). Parâmetros e objetos gráficos específicos (como malhas e texturas) são encapsulados pela visão.

Rede

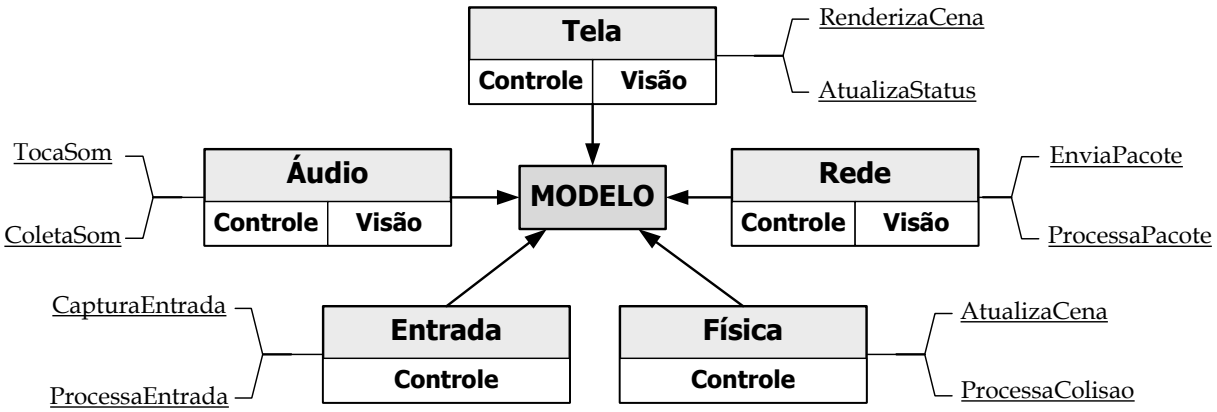

Figura 5.6: O jogo organizado como MVC e as principais tarefas

O módulo de rede processa mensagens vindas de outro jogador e periodicamente envia pacotes com as atualizações locais. Esse módulo ainda foi pouco desenvolvido (por questões práticas) e não é usado nos experimentos.

Física

O módulo de física possui duas tarefas principais. Na primeira, as posições dos objetos na cena são atualizados com base em suas velocidades e tempo decorrido desde o último quadro. A segunda tarefa é responsável pela checagem e processamento das colisões entre naves e asteroides no jogo. Como não há nenhum aspecto externo vinculado a esse módulo, o componente de visão inexiste.

Entrada

Este módulo é a interface pela qual o usuário atua efetivamente no jogo. As tarefas deste módulo são responsáveis por capturar os movimentos através de dispositivos como o teclado e mouse, e processar as respectivas atualizações. Por exemplo, o movimento do mouse muda a posição da nave no espaço. Note que essas alterações são feitas concorrentemente com as tarefas do módulo de física. Como naquele módulo, este também não apresenta uma visão.

Áudio

A primeira tarefa deste módulo é responsável por capturar os eventos gerados por cada objeto no jogo que corresponde a algum efeito sonoro. Por exemplo, o disparo de um projétil por uma das naves gera um evento que é então coletado pela tarefa. A outra tarefa produz efetivamente os efeitos sonoros em reação aos eventos gerados no quadro.

A divisão em módulos facilita o processo de especificação das tarefas e expõe o paralelismo em uma granularidade mais alta. As tarefas em cada módulo podem naturalmente

ser executadas em paralelo com as de outros módulos. Mesmo as tarefas dentro de um mesmo módulo podem seguir em paralelo. De fato, a única restrição de ordenação necessária no jogo foi entre as tarefas de captura e processamento das entradas, implementada com a API de barreira (*CreateBarrier*). A concorrência dentro de um módulo pode ser explorada iterativamente e a granularidade refinada. O sistema de execução fornece dados valiosos como duração de cada tarefa e taxa de conflitos que podem guiar o processo de paralelização. A tarefa de processamento de colisões do módulo de física pôde, desta forma, ser refinada gradativamente (os resultados são apresentados na Seção 5.5.1).

O processo de resolução de conflitos não apresentou muitas dificuldades. A maioria delas foi resolvida usando o esquema de prioridades já apresentado. Considere, como exemplo, o objeto nave. Em cada quadro, o objeto pode ser alterado pelas tarefas pertencentes ao módulo de entrada (mudar a posição da nave, atirar, etc) e física (atualizar posição com base na velocidade e checagem de colisões). Claramente, a tarefa de colisão obteve prioridade maior (não há porque alterar a posição da nave se ela foi destruída), seguida pela tarefa do módulo de entrada e a de atualização da cena.

Criar novos esquemas de resolução também não foi trabalhoso. Como exemplo, considere as explosões dos asteroides. Os objetos asteroides são mantidos em uma lista. Na versão mais eficiente do jogo, há três tarefas concorrentes checando por colisões. Quando elas acontecem os asteroides se dividem em partes menores. A explosão faz com que as dimensões do asteroide original sejam diminuídas, e ainda que novos asteroides menores sejam adicionados à lista de asteroides. Se mais de uma explosão ocorrer no mesmo quadro então o potencial de conflito é grande. A rotina desenvolvida para resolver esse conflito consiste simplesmente em fazer a união das versões conflitantes da lista.

5.5.1 Resultados

Os experimentos apresentados nesta seção foram conduzidos em uma máquina *quad-core* (Intel Q9550) com 2GB de memória RAM rodando a 2.83GHz. A placa gráfica é uma GeForce 8600 da NVidia. O sistema é executado sobre a plataforma virtual .NET da Microsoft e Windows Vista.

Os experimentos visam ilustrar e aferir as propriedades do sistema proposto com o jogo *SpaceWars3D*. Três tipos de experimentos são usados:

- Experimento A
  Neste experimento a versão sequencial do jogo é empregada, ou seja, existe somente uma réplica empregada para cada objeto.

- Experimento B
  Cada objeto possui duas réplicas, conforme discutido na Seção 5.4. O paralelismo

Experimento	Réplicas	FPS	Tempo médio (ms)	Ganho	Memória extra
A	1	39	25.2	1	1
B	2	59	17.0	1,48x	1,1x
C	9	76	13.1	1,92x	1,47x

Tabela 5.2: Resultados típicos para os três experimentos

neste caso é explorado somente entre as tarefas leitoras e modificadoras, ou seja, não é possível ter mais de uma tarefa modificadora executando concorrentemente.

- Experimento C
  Este experimento usa o sistema proposto, baseado em múltiplas réplicas. Paralelismo é permitido também entre as tarefas modificadoras.

A Tabela 5.2 apresenta valores típicos para algumas métricas nos diferentes tipos de experimentos. A segunda coluna mostra o número total de réplicas empregadas. Note que no experimento C um total de 9 réplicas são empregadas: uma para cada tarefa modificadora (total de 7), mais as duas usadas para as versões mestre e sombra. A terceira e quarta colunas representam a média do número de quadros por segundo (FPS) e o tempo médio em milisegundos gasto por quadro, respectivamente. Esses resultados foram coletados através de uma série de execuções idênticas para cada experimento, desempenhadas automaticamente por um *script* que simula o papel de um jogador. O ganho conseguido em desempenho em relação à versão sequencial (experimento A) para os experimentos B e C é apresentado pela quinta coluna. Finalmente, a sexta coluna mostra o custo máximo em memória para os experimentos B e C, quando comparados ao da versão sequencial.

Os resultados mostram que o experimento C obteve o melhor grau de concorrência (ganho em desempenho de aproximadamente 2x em relação à versão sequencial), mostrando a eficácia do sistema proposto. Como o tempo total gasto por quadro não pode ser menor do que a maior tarefa sequencial (no caso `RenderizaCena`), o experimento C obteve praticamente concorrência máxima para a aplicação em estudo e a máquina usada. O ganho em desempenho conseguido pelo experimento C deve ser considerado em face ao consumo extra exigido em memória (cerca de 1,5x em relação ao caso serial). Em sistemas *desktops* este custo extra não é fator crítico.

A Figura 5.7 ilustra um escalonamento típico das tarefas durante um quadro de execução para os três experimentos realizados. Algumas observações fazem-se necessárias. Primeiro, note que a versão sequencial não conta com as fases de reconciliação e propagação. A versão com duas réplicas necessita apenas da fase de propagação. Já a versão com nove réplicas executa tanto a reconciliação (no final do quadro) quanto a propagação

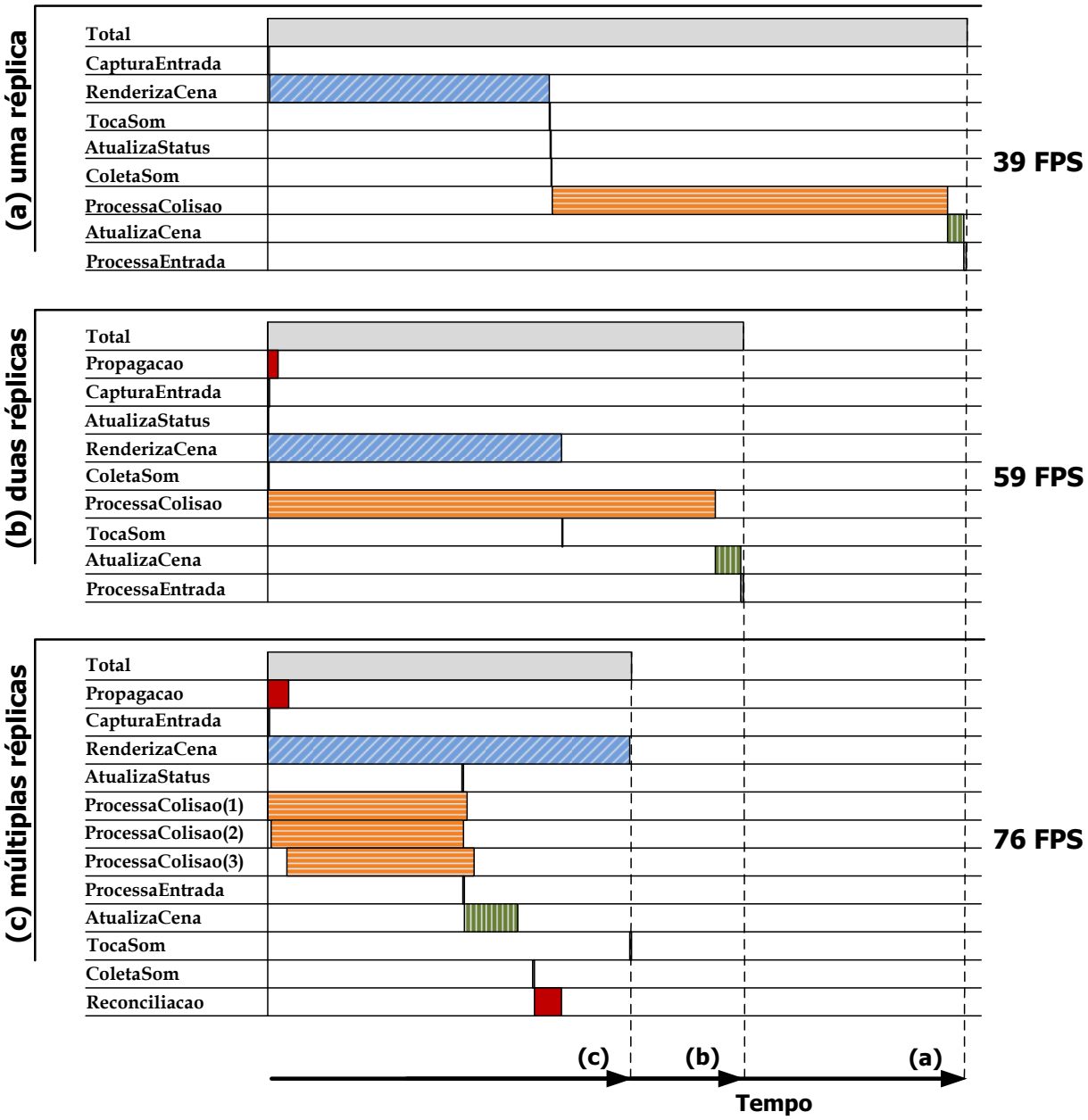

Figura 5.7: Escalonamentos típicos para as tarefas nos três experimentos

(no início do próximo quadro). Note também que, no caso de múltiplas réplicas, a tarefa que faz o processamento das colisões pôde ser subdividida em três tarefas que são executadas em paralelo.

5.6 Epílogo

Este capítulo adotou uma abordagem diferente de implementação para STMs que leva em consideração o domínio da aplicação. A principal observação que motivou a criação do sistema é o baixo desempenho observado em sistemas gerais de STM. O estudo de caso feito através de um jogo complexo mostrou a eficácia do sistema e revelou ganho de até 2x em relação à execução sequencial em uma máquina com quatro núcleos de execução.

Os resultados do trabalho apresentado neste capítulo foram publicados em [14]. Grande parte da pesquisa, implementação e coleta de resultados foram conduzidas quando este autor fazia estágio na sede da Microsoft Research, Redmond (EUA).

Capítulo 6

Caracterização Energética de STM

"If we were to follow Moore's Law . . . our next generation processor would have a power density close to that of the core of a nuclear reactor."

A. Lidow e G. Sheridan, 2003

Este capítulo apresenta uma caracterização para o consumo de energia em uma STM típica. A motivação para o trabalho é descrita pela Seção 6.1. As Seções 6.2 e 6.3 descrevem, respectivamente, a plataforma de simulação e o conjunto de aplicações usados no processo de caracterização. A metodologia empregada no trabalho é apresentada na Seção 6.4, seguida pelos resultados obtidos como parte da caracterização da STM na Seção 6.5. Uma estratégia é proposta na Seção 6.6 visando reduzir o consumo de energia nos momentos de espera. Por fim, as considerações finais são brevemente apresentadas na Seção 6.7.

6.1 Motivação

A avaliação de novas propostas de implementações para memória transacional (tanto em hardware como em software) tem sido invariavelmente guiada pelo desempenho. A maioria dos sistemas costumam medir o número de transações realizadas em alguma unidade de tempo, e usam o resultado dessa medida como critério de eficiência: quanto mais transações processadas por unidade de tempo, melhor candidato é o sistema. Essa tendência inicial é totalmente justificada pelo fato de que uma nova tecnologia deve mostrar pelo menos um desempenho comparável ao da tecnologia atual.

As implementações atuais ainda seguem essa tendência, ou seja, o desempenho é o principal (e muitas vezes o único) fator considerado. Devido ao fato do amadurecimento

da pesquisa em TM, este capítulo argumenta que outros fatores também devem ser considerados na avaliação de novas abordagens, principalmente o consumo de energia. É inegável a importância de técnicas para redução do consumo de energia em sistemas embarcados e, atualmente, esta tendência tem se espalhado também para *data centers* e sistemas *desktops* [62]. A questão energética tem portanto extrema relevância no projeto de sistemas computacionais atuais, e deve ser devidamente considerada e analisada em novas propostas de TM.

Visando suprir a deficiência na literatura com relação a análise de energia, este capítulo apresenta um trabalho pioneiro que visa caracterizar o consumo de energia em uma implementação típica de STM. O objetivo é elucidar os aspectos importantes para o projeto de algoritmos de STM, abrindo caminho para implementações que não só possuam bom desempenho, mas que também reduzam o consumo de energia. Os únicos trabalhos com alguma semelhança foram desenvolvidos no contexto de HTM, por Moreshet *et al.* [90] e Ferri *et al.* [35, 36]. Eles chegam a conclusão, usando uma implementação em hardware baseada na proposta de Herlihy e Moss [55], que para algumas aplicações a abordagem transacional pode ser mais eficiente do que travas em termos de energia. No entanto, as aplicações usadas pelos autores são extremamente simples e não condizem com o cenário de uso esperado para TM. A abordagem apresentada aqui difere em dois aspectos principais. Primeiro, este texto tem como alvo principal sistemas de STM. Segundo, as aplicações usadas para a caracterização fazem parte do pacote STAMP, construído especialmente para análise do comportamento de implementações de TM.

6.2 Plataforma de simulação

O processo de caracterização é conduzido em um ambiente simulado amplamente usado na literatura, conhecido como MPARM [74]. Um diagrama com os principais elementos da plataforma é mostrado na Figura 6.1. Os componentes da plataforma usados nos experimentos são configurados conforme mostra a Tabela 6.1.

Os dispositivos de interrupção e semáforo fornecem mecanismos de comunicação entre os processadores. O primeiro permite a um processador enviar um pedido de interrupção a outro, enquanto o segundo suporta operações atômicas no estilo de *Test & Set*. Ambos são mapeados diretamente em memória, bastando operações de leitura e escrita em endereços preestabelecidos para acessá-los. Duas observações a respeito da plataforma fazem-se necessárias. Primeiramente, as memórias empregadas (tanto privada como compartilhada) são baseadas na tecnologia SRAM (*Static Random Access Memory*). Como consequência, permitem acesso mais rápido e consomem menos energia se comparadas com a tecnologia DRAM (*Dynamic Random Access Memory*). Segundo, os acessos feitos à memória compartilhada não são retidos na cache, visto que a implementação do

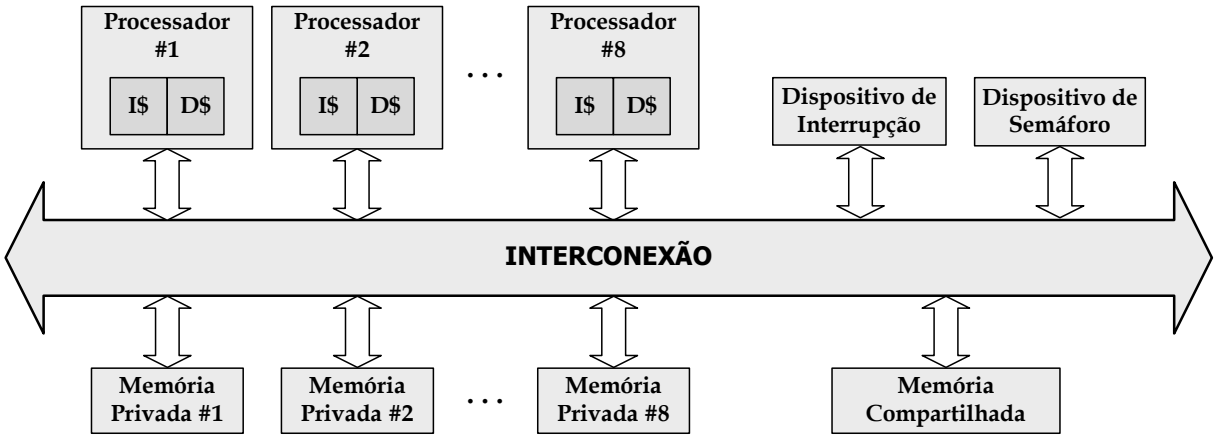

Figura 6.1: Plataforma de simulação usada no processo de caracterização

Componente	Configuração
Processadores	de 1 a 8 processadores ARMv7 – precisão de ciclos
I$	cache de instrução – 8KB, associatividade 4
D$	cache de dados – 4KB, associatividade 4, write-through
Interconexão	AMBA-AHB – precisão de sinal
Memória privada	uma para cada processador – SRAM, 12MB, precisão de sinal
Memória compartilhada	SRAM, 16MB, precisão de sinal

Tabela 6.1: Configuração da plataforma usada no processo de caracterização

barramento usada não possui suporte para coerência de cache.

6.3 Conjunto de aplicações

As aplicações usadas na caracterização fazem parte do pacote STAMP, versão 0.9.10, disponibilizado pela Universidade de Stanford [83]. O pacote é constituído por 8 aplicações e tem cerca de 30 variações para os argumentos de linha de comando, visando representar diversos cenários transacionais com diferentes tamanhos de transação, tempos em transação, tamanhos do conjunto de leitura e escrita, e graus de contenção. As aplicações são escritas em linguagem `C`, em duas versões: sequencial e transacional. A versão transacional é basicamente a sequencial marcada com macros para indicar o início e fim das transações, além das barreiras de leitura e escrita.

A Tabela 6.2 mostra, para cada aplicação, os argumentos de linha de comando usados na caracterização, o domínio da aplicação e uma breve descrição sobre seu comportamento.

Aplicação	Argumentos	Domínio	Descrição
`genome`	`-g512 -s32 -n32768`	bioinformática	sequenciamento de genes
`kmeans`	`-m15 -n15 -t0.05 -irandom-n2048-d16-c16`	mineração de dados	algoritmo de clusterização
`ssca2`	`-s13 -i1.0 -u1.0 -l3 -p3`	científico	conjunto de operações em grafos
`vacation`	`-n4 -q60 -u90 -r16384 -t4096`	processamento de transações online	sistema para reserva de passagens
`bayes`	`-v16 -r1024 -n2 -p20 -i2 -e2`	aprendizado de máquina	aprendizado de estrutura de rede Bayesiana
`labyrinth`	`-i random-x48-y48-z3-n64`	engenharia	algoritmo de roteamento
`yada`	`-a20 -i633.2`	científico	algoritmo para refinamento de malhas Delaunay
`intruder`	`-a10 -l16 -n4096 -s1`	segurança	detecção de ataques em redes

Tabela 6.2: Aplicações do pacote STAMP – argumentos usados na caracterização, domínio e breve descrição

Existem pelo menos 3 conjuntos de argumentos para cada aplicação, recomendados de acordo com o sistema onde elas são utilizadas. Os argumentos usados neste trabalho são os indicados para sistemas simulados, com exceção do aplicação `bayes`. Para essa aplicação o número de variáveis teve de ser reduzido de 32 para 16, caso contrário a memória requerida excedia a disponível na plataforma.

O processo de adaptação das aplicações para o ambiente de simulação foi, na sua maioria, direto. Foi necessário um mínimo de conhecimento sobre o comportamento de cada uma delas, já que as variáveis compartilhadas tinham de ser explicitamente marcadas para que o compilador as mapeassem para a área de memória correspondente na plataforma.

No processo de simulação duas anomalias principais foram observadas durante a execução das aplicações. Na primeira, a versão transacional das aplicações `genome` e `kmeans` produzia resultados inválidos. A causa da anomalia deveu-se a otimizações feitas pelo compilador que invalidavam a semântica transacional. Foi observado que variáveis não transacionais (geralmente variáveis de indução em laços) eram movidas para dentro de uma transação. Quando uma transação era cancelada, tais variáveis não tinham seu valor restaurado, comprometendo o restante da execução. Para corrigir o problema as devidas variáveis foram marcadas com a palavra reservada *volatile*, impedindo que o compilador as movesse de posição no código.

O segundo caso de anomalia ocorreu com a versão transacional do programa `ssca2`, que gerava resultados inconsistentes. Após análise do código foi notado o uso incorreto de uma transação para computar o número de nodos do grafo. O código foi devidamente

corrigido e notificado aos mantenedores do STAMP, que reconheceram o erro.

6.4 Metodologia

A metodologia para caracterização aqui descrita visa determinar a energia consumida pelos componentes básicos de uma STM. Desta forma, é possível avaliar quais os elementos mais custosos e orientar o projeto do algoritmo de forma a otimizá-los. O sistema de STM é considerado como uma caixa preta, ou seja, os detalhes de implementação não são importantes para o processo de medição. A interface entre aplicação e STM é feita através da seguinte API:

- `TxStart` – marca o início de uma nova transação;
- `TxCommit` – finaliza a transação. Se a transação for efetivada com sucesso, os dados especulativos ficam visíveis para todo o sistema de forma atômica. Caso contrário, a transação é cancelada e o fluxo de execução retorna para a instrução seguinte ao comando `TxStart`;
- `TxAbort` – cancela a transação atual. Essa primitiva é chamada implicitamente pelo STM quando a operação `TxCommit` falha, mas pode também ser invocada explicitamente;
- `TxLoad` (l) – a palavra contida na posição de memória l é lida e retornada;
- `TxStore` (l, v) – o valor v é armazenado na posição de memória l;
- `TxStoreLocal` (l, v) – mesmo que `TxStore`, mas l indica uma posição de memória local. Essa interface, embora opcional, pode aumentar o desempenho do sistema se devidamente usada no lugar de `TxStore`;
- `TxAlloc` (n) – aloca dinamicamente n bytes e retorna a posição de memória base;
- `TxFree` (l) – desaloca a memória dinâmica previamente alocada na posição base l.

É importante observar que essa interface é padrão e fornecida pela maioria das implementações de STM baseadas em palavras. Nos experimentos descritos neste capítulo a STM TL2 [27] é usada. Os motivos para adoção da TL2 são: (i) código fonte (escrito em linguagem `C`) é distribuído publicamente e bastante acessível ; e (ii) essa STM é amplamente considerada o estado da arte em implementações bloqueantes. A TL2 pode ser configurada em dois modos (ambos usados nos experimentos): *TL2-lazy* – que usa versionamento diferido, e *TL2-eager* – que usa versionamento direto. Os algoritmos para

as principais primitivas de *TL2-lazy* são descritas no Capítulo 4. Note que a metodologia apresentada aqui não se restringe de forma alguma à TL2.

A medição do consumo de energia é dividida em três fases principais:

- Aplicação (identificado nos experimentos por `App`) – representa o gasto causado unicamente pelo código da aplicação;
- STM API – representa o custo causado pelas chamadas e processamento de cada primitiva do sistema de STM. Para evitar excesso de informação, as primitivas `TxStart`, `TxStoreLocal`, `TxAlloc`, `TxFree` são somadas e identificadas nos experimentos como `Outros`;
- Cancelamentos – quando uma transação é cancelada (ou seja, `TxAbort` é invocado), todo o custo acumulado desde o último `TxStart` é armazenado sob o termo `Rollback`. Após o cancelamento, o gerenciador de contenção pode optar por suspender a transação antes de reiniciá-la. Note que esse comportamento é dependente de implementação, mas a STM geralmente provê alguma forma da aplicação escolher como o gerenciamento de contenção é feito. No caso da STM usada nos experimentos, uma técnica conhecida como *backoff* é usada: após três cancelamentos consecutivos, uma transação é postergada por um tempo proporcional ao número de cancelamentos. São duas as formas básicas para o *backoff* : (1) exponencial – cada cancelamento aumenta o tempo de espera exponencialmente; e (2) linear - o tempo aumenta linearmente com o número de cancelamentos seguidos. O total de energia gasta com esse procedimento é identificado nos experimentos como `Backoff`.

Para que o consumo de energia possa ser medido é necessário instrumentar o código das aplicações. Três comandos básicos fornecidos pelo simulador são usados:

- `start_metric` – instrui o simulador a coletar estatísticas sobre a simulação;
- `stop_metric` – instrui o simulador a finalizar a coleta;
- `dump_fine_metric` (*id*) – todos os dados coletados até o último `stop_metric` são escritos em um arquivo de saída preestabelecido e identificados pelo parâmetro *id*.

A Figura 6.2 ilustra o processo de medição do consumo de energia para as principais fases. A medição de energia é ligada assim que a simulação é iniciada ①, sendo contabilizada como parte da aplicação. Quando alguma chamada à API da STM é invocada (`TxStart`) ②, a fase da aplicação termina: o custo energético é contabilizado (`App`), e a medição é novamente iniciada ③. A fase de medição é agora da STM ④ e termina assim que a chamada da API retorna: os gastos são salvos e atribuídos para a API (`TxStart`) ⑤.

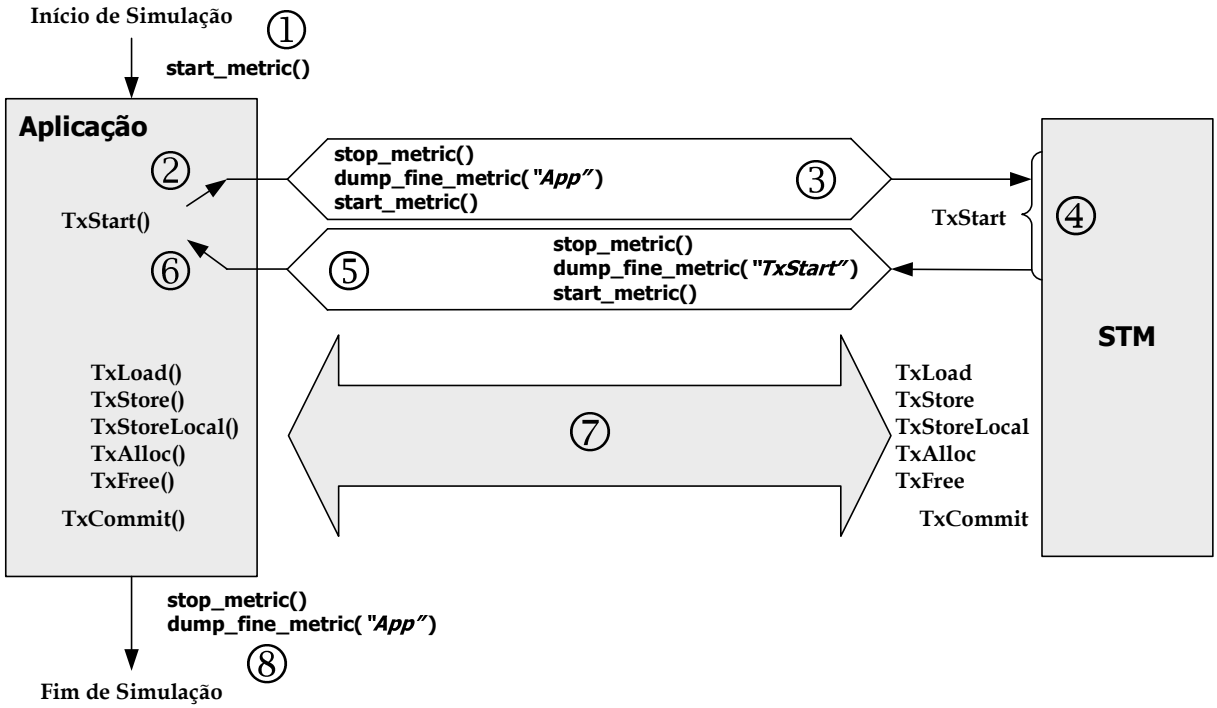

Figura 6.2: Processo de instrumentação para medição de energia

O fluxo de execução retorna à aplicação ⑥ e o custo energético é agora atribuído a esta. Durante a execução da aplicação novas chamadas serão feitas à STM, ocasionando novas mudanças de fases ⑦. Quando a aplicação chega ao final, a medição é encerrada e a simulação termina ⑧. As fases de cancelamentos não estão apresentadas na figura porque dependem da interação entre processadores. Quando uma transação é abortada, toda a energia acumulada desde o último `TxStart` é contabilizada como `Rollback`. Repetidos cancelamentos acionam o mecanismo de *backoff*, e toda a energia gasta nesta fase é contabilizada como `Backoff`.

Os dados coletados durante a simulação são pós-processados por vários *scripts* e resultam em tabelas sumarizando os custos devidos à aplicação e ao sistema de STM. A próxima seção caracteriza o comportamento enérgico das oito aplicações STAMP usando o sistema TL2 na plataforma MPARM.

6.5 Caracterização

6.5.1 Relação entre desempenho e energia

Para justificar a avaliação do consumo de energia é importante apresentar evidências que dissociem energia e desempenho, caso contrário o comportamento energético poderia ser

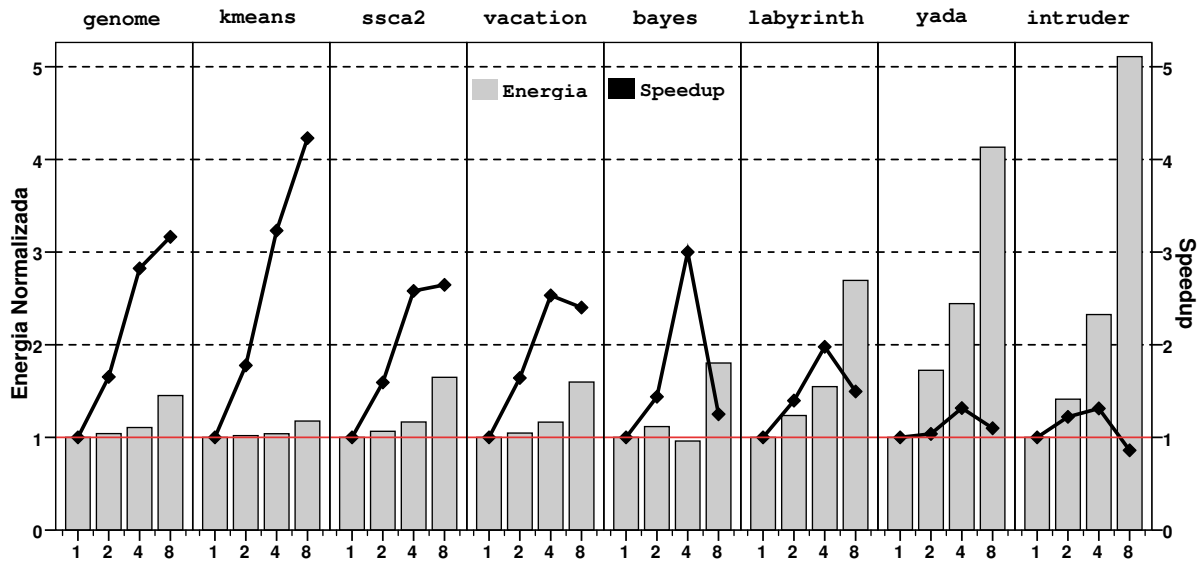

Figura 6.3: Valores para energia e desempenho usando a *TL2-lazy* e um número variável de processadores. Os resultados estão normalizados em relação ao caso transacional com apenas um processador

inferido diretamente a partir do desempenho. Visando destacar esse ponto, a Figura 6.3 apresenta os resultados de energia e desempenho para as aplicações STAMP usando a configuração *TL2-lazy*. Os valores estão normalizados em relação ao resultado da simulação com apenas um processador. A figura deixa claro que o aumento do gasto com energia nem sempre se deve ao tempo de execução maior: enquanto o valor de energia sempre aumenta com o número de processadores usados, o desempenho pode aumentar (como acontece nas aplicações `genome` e `kmeans`, por exemplo) ou mesmo cair (casos do `yada` e `intruder`) em diferentes proporções. O principal motivo para esse comportamento está relacionado aos cancelamentos provocados pela STM, como será visto logo adiante. Em geral, a Figura 6.3 mostra que, de fato, os valores para energia nem sempre podem ser inferidos diretamente a partir do tempo de execução. Os valores para a configuração *TL2-eager* são semelhantes aos da *TL2-lazy* e portanto foram omitidos.

6.5.2 Custo com 1 processador

O próximo passo da caracterização consiste em analisar o custo individual das primitivas transacionais quando apenas um processador é usado. Os valores obtidos, normalizados em relação à execução sequencial, são mostrados através da Figura 6.4 para as configurações *TL2-lazy* e *TL2-eager*. As seguintes observações podem ser feitas com base nessa figura:

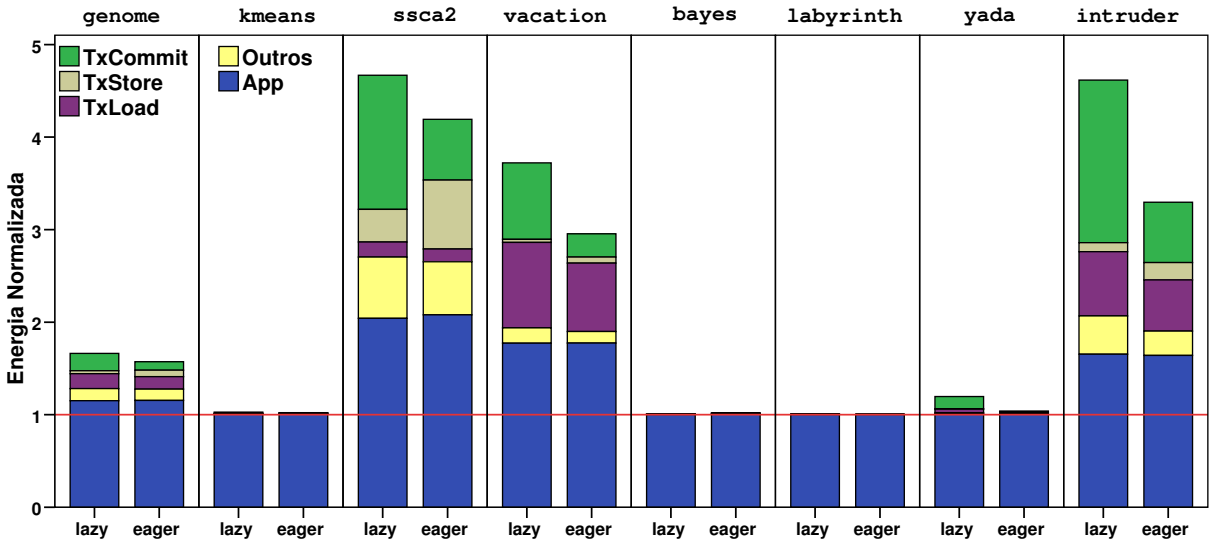

Figura 6.4: Custo energético por primitiva transacional para ambas *TL2-lazy* e *TL2-eager*, usando apenas um processador. Os resultados estão normalizados em relação à execução sequencial

- A técnica que usa versionamento direto (*TL2-eager*) tem um custo de energia ligeiramente menor do que a técnica com versionamento diferido (*TL2-lazy*). Mesmo assim, o custo total introduzido pelas transações pode chegar perto de 5x o do caso sequencial (casos de `ssca2` e `intruder`);

- As aplicações `kmeans`, `bayes`, `labyrinth` e `yada` têm custo adicional praticamente desprezível (`yada-lazy` apresenta um custo um pouco mais elevado). A explicação para `bayes`, `labyrinth` e `yada` decorre do fato dessas aplicações possuírem transações longas, o que amortiza o custo do uso das primitivas transacionais. O caso de `kmeans` é diferente: essa aplicação possui transações pequenas e que tomam apenas cerca de 5% do total do tempo de execução. Dessa forma, os custos das primitivas ficam diluídas e não são representativas;

- Algumas aplicações, principalmente `ssca2`, `vacation` e `intruder`, apresentam um custo elevado de energia devido ao código da aplicação (cerca de 2x em relação ao caso sequencial). A explicação para esse comportamento é que o compilador consegue otimizar bem o código no caso sequencial, mas não no caso transacional em decorrência da presença das barreiras transacionais. Este comportamento não deve ser visto como um fator limitante da nossa abordagem: nesse primeiro estágio estamos interessados em averiguar o comportamento de um sistema real, ou seja, sem alterar a forma como as aplicações são usualmente compiladas. Pretendemos

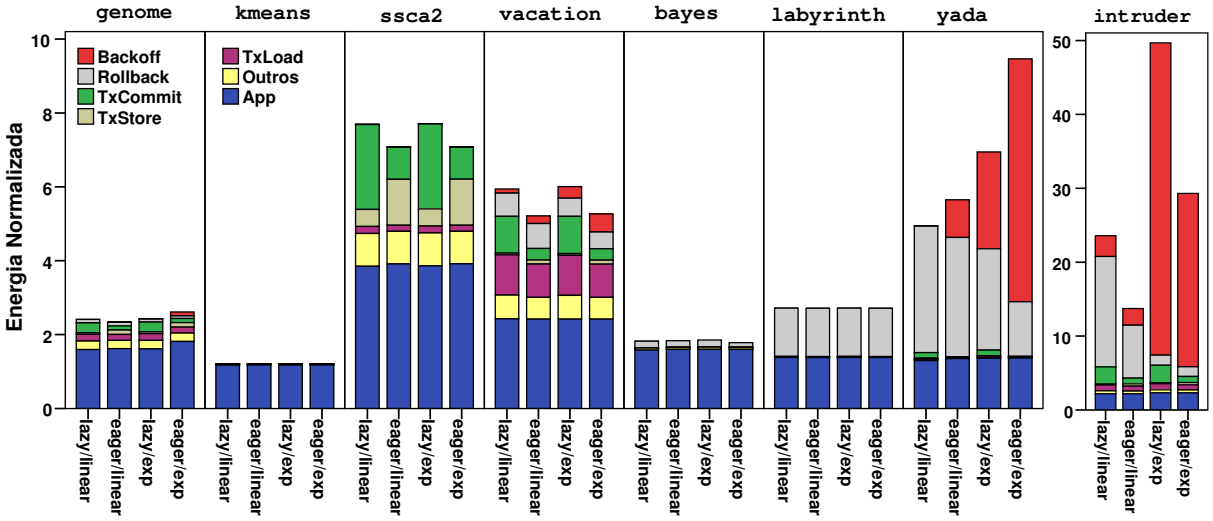

Figura 6.5: Custo energético por primitiva transacional para ambas *TL2-lazy* e *TL2-eager* e esquemas de espera linear e exponencial, usando 8 processadores. Os resultados estão normalizados em relação à execução sequencial

averiguar o impacto das otimizações em um estudo futuro;

- Os resultados dos custos para as primitivas `TxStore` e `TxCommit` estão coerentes com a configuração da TL2 usada. Note que para *TL2-lazy* o custo de escrita é menor do que *TL2-eager*, já que esse último necessita adquirir uma trava para cada palavra escrita. O comportamento é inverso para `TxCommit`: *TL2-lazy* tem um custo maior porque durante a efetivação é necessário adquirir uma trava para cada palavra no conjunto de escrita e mover o valores especulativos para a memória compartilhada, o que não é necessário no caso de *TL2-eager*.

6.5.3 Custo com 8 processadores

A análise anterior não mostra os custos devidos aos cancelamentos porque apenas 1 processador é usado. Para analisar este novo cenário, a plataforma de simulação foi configurada para 8 processadores e os experimentos foram refeitos. A Figura 6.5 mostra os resultados para esse novo lote de simulações. Note que agora cada configuração da TL2 também é diferenciada em relação ao tipo de *backoff* usado (linear e exponencial).

Os custos devidos aos cancelamentos evidenciados pela Figura 6.5 estão relacionados diretamente à taxa de cancelamento de cada aplicação. Essa taxa, descrita pela Tabela 6.3, mostra a porcentagem do total de transações que foram canceladas. Como pode ser visto, as aplicações com taxa de cancelamento baixa (`genome`, `kmeans`, `ssca2` e `bayes`)

Aplicação	Taxa de Cancelamento (%)			
	TL2-lazy		*TL2-eager*	
	linear	exponencial	linear	exponencial
`genome`	2	1.6	3.2	2.1
`kmeans`	10.5	7.5	15.5	10.3
`ssca2`	0.2	0.1	0.2	0.2
`vacation`	18.5	14.5	29.4	21
`bayes`	6.6	7.1	7.6	8.1
`labyrinth`	29.4	28.4	33.0	31.1
`yada`	51.1	31.2	95.6	48.6
`intruder`	55.6	24.9	59.9	38.4

Tabela 6.3: Taxa de cancelamento para a configuração com 8 processadores

praticamente não possuem gastos decorrentes de `Rollback` e `Backoff`. Apesar da taxa de `kmeans` ser um pouco mais acentuada, essa aplicação gasta pouco tempo em transações e portanto os gastos são diluídos (conforme já mencionado no caso com 1 processador).

As aplicações com taxa de cancelamento mais alta apresentam gastos em `Rollback` e `Backoff` em diferentes proporções. Primeiramente, note que `labyrinth` tem gasto desprezível com `Backoff`. Como essa aplicação possui transações grandes, o mecanismo de espera quase nunca é disparado (note que são necessários 3 cancelamentos sucessivos para acioná-lo) e portanto o custo de espera inexiste. A aplicação `vacation` tem taxa de cancelamento moderada, refletida nos respectivos custos de reexecução e espera. Os casos de `yada` e `intruder` são mais interessantes por apresentarem cancelamentos constantes. Nos casos em que o esquema de espera linear é usado, o custo de `Rollback` domina. Porém, quando o tempo de espera exponencial é empregado, o custo de `Backoff` passa a prevalecer. O gasto total em energia da aplicação `intruder` usando *TL2-lazy* e tempo de espera exponencial chega a cerca de 50x o do caso sequencial.

6.6 Otimização via DVFS

Esta seção apresenta uma estratégia para redução do consumo de energia em casos em que estados de espera são induzidos por cancelamentos de transações. A estratégia pode ser usada em qualquer STM na qual o gerenciador de contenção adote políticas de resolução de conflitos baseadas em espera. O objetivo principal da técnica é reduzir o consumo de energia em aplicações com alta taxa de cancelamento (como as aplicações `yada` e `intruder` vistas anteriormente), sem interferir negativamente no tempo de execução.

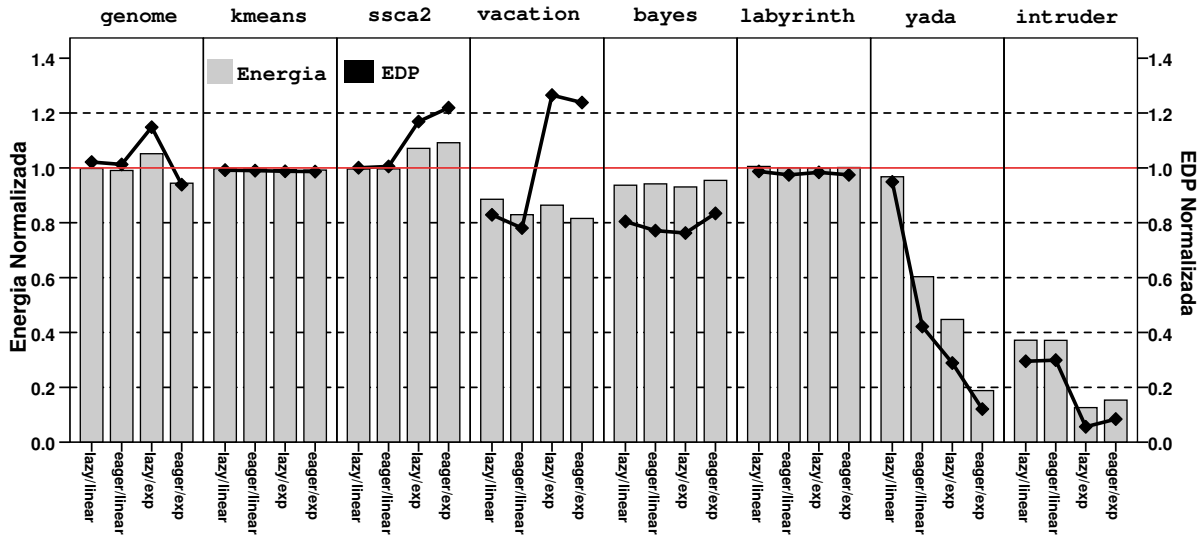

Figura 6.6: Valores para Energia e EDP resultantes da estratégia baseada em DVFS. Os resultados estão normalizados em relação aos da Figura 6.5

A estratégia desenvolvida é baseada em uma técnica conhecida como DVFS (*Dynamic Voltage and Frequency Scaling*) [62]. À plataforma original, apresentada na Figura 6.1, é adicionado um módulo que permite ao software modificar tanto a frequência quanto a voltagem de operação de um processador específico. A estratégia funciona da seguinte maneira. Ao entrar em modo `Backoff`, o processador correspondente tem sua frequência de operação e voltagem reduzidas de 200MHz para 1.56MHz e de 1V para 0.547V, respectivamente. Esta operação é rápida, requerendo apenas 1 ciclo por mudança. O processador então cumpre seu tempo em estado de espera (podendo ser tanto linear quanto exponencial) em modo de baixa potência. Quando o tempo de `Backoff` termina, ambas frequência de operação e voltagem originais são restauradas.

Para avaliar a eficácia da estratégia, os experimentos vistos na Seção 6.5.3 foram novamente executados usando o novo esquema baseado em DVFS. A Figura 6.6 mostra o resultado das simulações. É importante também que a técnica empregada não degrade o desempenho: a operação em baixa potência reduz o consumo, mas pode aumentar substancialmente o tempo de execução da aplicação. Desta forma, a Figura 6.6 apresenta os resultados não só em termos de energia, mas também através de uma métrica conhecida como EDP (*Energy-Delay Product*). Essa métrica é definida como o produto da energia pelo tempo de execução. Valores mais baixos para EDP indicam melhores resultados.

As seguintes observações são feitas com base na Figura 6.6:

- Para as aplicações com taxa de cancelamento média ou alta (`vacation`, `yada` e

`intruder`) a estratégia reduziu efetivamente o consumo de energia. Em média, a energia foi reduzida em 45%, chegando a 87% no caso do `intruder`. O EDP também seguiu essa mesma tendência para a versão linear de `vacation`, `yada` e `intruder` (chegando a 96% para este último), mas para a versão exponencial de `vacation` a técnica acabou afetando negativamente o EDP, já que o tempo de execução aumentou devido ao tempo de espera exponencial em baixa frequência decorrente dos cancelamentos;

- As versões com *backoff* exponencial das aplicações `genome`, `ssca2` também apresentaram comportamento negativo. Já as versões usando *backoff* linear quase nunca degradaram o EDP (ligeira exceção para o `genome`). As aplicações que não se beneficiaram da técnica (`genome`, `kmeans`, `ssca2` e `labyrinth`) são justamente aquelas com baixa taxa de cancelamento e que raramente entram em estado de espera. A aplicação `bayes` ainda foi beneficiada pela técnica, apesar do baixo grau de contenção.

Como resumo, pode se afirmar que a estratégia baseada em DVFS reduziu efetivamente o consumo de energia para as aplicações com alto grau de contenção. Além disso, para essas aplicações, o tempo de execução não foi afetado negativamente pela estratégia.

6.7 Epílogo

Este capítulo mostrou pela primeira vez o consumo de energia em um sistema de memória transacional em software considerado estado da arte. A caracterização usa o conjunto de aplicações do pacote STAMP e considera dois esquemas de versionamento diferentes para a STM: direto e diferido. Visando diminuir o EDP em aplicações com alto grau de conflitos, uma técnica baseada em DFVS é introduzida. Os resultados obtidos com o emprego dessa técnica mostram um ganho de EDP máximo e médio de 96% e 45%, respectivamente.

Este capítulo é baseado nos trabalhos [63, 13] publicados em colaboração com o colega de doutorado Felipe Klein. A ideia para caracterização da energia em STM nasceu após uma discussão informal entre ambos e da constatação que não havia nada neste sentido na literatura. Muitos dos conceitos que aparecem aqui são resultados de discussões conjuntas no laboratório que compartilham. Desta forma, a contribuição de cada um para o trabalho não pode ser determinada de forma pontual. No entanto, é possível dizer que este autor trabalhou efetivamente no processo de adaptação do sistema TL2 e das aplicações STAMP para a plataforma MPARM.

Capítulo 7

Conclusão

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Turing, 1950

A crescente popularização dos processadores com múltiplos núcleos tem renovado o interesse tanto por parte da academia como indústria por métodos mais simples, eficientes e produtivos em programação concorrente. Entendemos que esta tese apresenta contribuições importantes para uma das promessas mais relevantes dos últimos anos, conhecida como memória transacional. Mais especificamente, apresentamos três trabalhos pioneiros que abordam diferentes aspectos de sistemas com memória transacional em software, como a sua implementação, modelo de programação, desempenho e consumo de energia.

Começamos apresentando uma solução para programação paralela com memória compartilhada em arquiteturas assimétricas. Essas arquiteturas são notórias pela dificuldade de programação do software de sistema e aplicativo. Nossa abordagem focou especificamente na arquitetura Cell/B.E. usada nos consoles de vídeo game PlayStation 3 da Sony. O sistema proposto, chamado de CellSTM, usa transações para estender o conceito de cache gerenciada por software e fornecer, de forma transparente, uma visão consistente da memória compartilhada aos elementos de processamento da arquitetura.

Os experimentos realizados com CellSTM mostraram que o uso do modelo transacional em arquiteturas assimétricas também é promissor. Em especial, o experimento com a aplicação de sequenciamento de genes, Genoma, apresentou forte evidência de escalabilidade proporcionada pelas transações. Os experimentos também apontaram que aplicações típicas favorecidas pelo modelo transacional são aquelas que possuem: (1) tempo em transação moderado; e (2) nível de contenção de baixo para médio. Aplicações com transações pequenas dificilmente exibirão algum ganho, já que o custo extra introduzido pelas primitivas transacionais não pode ser amortizado efetivamente.

No segundo trabalho, usamos o conhecimento sobre o domínio da aplicação para propor um sistema de STM voltado para jogos computacionais. Nossa abordagem foi influenciada pelo baixo desempenho geralmente conseguido com implementações típicas de STM. A principal diferença da nossa proposta está no fato de que uma transação nunca é cancelada. Ao final de cada quadro de execução, o sistema detecta os conflitos nos acessos aos objetos compartilhados e delega ao programador a responsabilidade de resolvê-los. O modelo de programação proposto foi descrito de maneira informal e exemplos ilustraram seu uso.

Também apresentamos um estudo de caso usando um jogo complexo. Alguns aspectos de programação foram mencionados e resultados experimentais mostraram a eficácia do sistema na paralelização do jogo usado como objeto de estudo. Usando uma máquina com quatro núcleos de execução conseguimos um ganho de aproximadamente 2x em desempenho em relação a versão sequencial, com um custo adicional de memória em torno de 1,5x. O sistema proposto pode ser empregado em jogos que exibam uma quantidade razoável de tarefas (na ordem de dezenas) e que, além disso, não possuam tarefas com tempo excessivo de execução, uma vez que o paralelismo extraído é limitado diretamente por esse fator.

O último trabalho caracterizou, pela primeira vez, o consumo de energia em um sistema de STM considerado estado da arte. Apresentamos a plataforma de simulação adotada, as aplicações STAMP usadas nos experimentos e a metodologia para medição do consumo de energia das transações. O processo de caracterização cobriu uma ampla faixa de aplicações com diferentes propriedades no que se refere ao tamanho das transações, tempo em transação, tamanhos do conjunto de leitura e escrita, e níveis de contenção.

Com relação ao sistema de STM, duas configurações foram avaliadas: uma usando versionamento direto, e outra com versionamento diferido. Em geral, observamos que a versão usando versionamento direto possui uma ligeira vantagem em relação ao consumo de energia. Com base nos resultados da caracterização, propomos uma técnica baseada em DVFS visando melhorar o EDP nos cenários com alta contenção. Os resultados obtidos com o emprego de tal técnica mostraram um ganho de EDP máximo e médio de 96% e 45%, respectivamente. O processo de caracterização apresentado pode servir como guia para projetistas de algoritmos transacionais que visam melhorar não só desempenho, mas também o consumo de energia.

7.1 Trabalhos futuros

Devido ao caráter pioneiro dos trabalhos que formam o cerne desta tese, acreditamos que ainda há espaço para grandes aprimoramentos em todos eles. Esta última seção descreve o que pensamos ser possíveis trabalhos futuros.

O trabalho com o sistema CellSTM pode ser aperfeiçoado em diversas frentes. Pri-

meiramente, um número maior de aplicações deve ser adaptado ao sistema para que uma análise mais apurada do desempenho possa ser feita. Uma alternativa realista é adaptar o restante do pacote STAMP. Atualmente este processo é trabalhoso porque envolve a instrumentação manual de cada operação de leitura e escrita em memória compartilhada. Para que esta sobrecarga seja reduzida (ou mesmo eliminada), é vital que o compilador fique responsável por incluir automaticamente as barreiras de leitura e escrita. A versão atual do compilador GNU (*GNU's Not Unix*) `C` para os SPEs já consegue inserir automaticamente operações de DMA quando variáveis marcadas com o especificador `__ea` são referenciadas. Essas operações são feitas através de uma biblioteca que poderia ser estendida com o suporte transacional, eliminando desta forma um esforço significativo de codificação.

Alguns aspectos de implementação do sistema CellSTM podem ainda ser aprimorados. Como já comentado brevemente, caso os conjuntos de leitura e escrita excedam o limite de tamanho uma exceção é lançada e o programa é abortado. A solução mais geral para o caso do transbordamento exige que a memória global do sistema seja usada, já que a memória local do SPE é limitada em 256KB. Também acreditamos que um estudo mais detalhado dos algoritmos transacionais usados por CellSTM poderia ser realizado. Por exemplo, será que não compensaria serializar as efetivações, dado que o trabalho já é feito quase que totalmente pelo PPE e este permite somente a execução concorrente de duas *threads*? Uma consequência direta desta abordagem é que o problema da privatização [112] é eliminada naturalmente. Por fim, a escalabilidade de CellSTM deve ser aferida usando um sistema que suporte um número maior de elementos de processamento, como os servidores BladeCenter da IBM.

Com relação ao trabalho com a STM voltada para jogos, acreditamos ser importante definir formalmente a semântica das construções sugeridas e do modelo de execução. Muito recentemente, o trabalho conduzido por Koskinen *et al.* [65] apresenta uma formalização para o que eles chamam de *transações espessas* (*coarse-grained transactions*). Neste trabalho, eles chegam a formalizar alguns aspectos do sistema apresentado aqui e publicado em [14]. Acreditamos que a abordagem de especializar o sistema de STM para um determinado domínio tenha se mostrado valiosa. Neste sentido, seria interessante averiguar o quanto outros domínios se beneficiariam do mesmo tratamento. Em especial, estamos interessados em adotar o mesmo esquema para a paralelização do software de interface gráfica com o usuário. Atualmente, a maioria das abordagens para esse tipo de software usam um modelo sequencial baseado em um laço de eventos.

O processo de caracterização do consumo de energia em sistemas de STM abre caminho para uma série de otimizações dos algoritmos transacionais visando a redução do consumo energético, como a técnica baseada em DVFS apresentada. Seria de grande importância também caracterizar o consumo de energia quando travas são usadas no lugar de

transações, permitindo determinar a eficiência destas em relação às transações. Também seria de grande valia a caracterização de outras implementações de STM, especialmente as não-bloqueantes. Atualmente é aceito que as implementações bloqueantes de STM são mais eficientes em termos de desempenho quando comparadas às não-bloqueantes. Será que o mesmo valeria quando o consumo de energia é considerado?

O pouco que podemos enxergar no futuro da computação mostra que a programação concorrente terá papel fundamental e que há ainda muito por se fazer. Esperamos ter contribuído com uma fração da solução que os sistemas de memória transacional visam prover.

Referências Bibliográficas

[1] M. Abadi, A. Birrell, T. Harris e M. Isard. Semantics of transactional memory and automatic mutual exclusion. Em *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 63–74, janeiro de 2008.

[2] M. Abadi, T. Harris e M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. Em *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pp. 185–196, fevereiro de 2009.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha e T. Shpeisman. Compiler and runtime support for efficient software transactional memory. Em *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 26–37, junho de 2006.

[4] A. Agarwal e M. Cherian. Adaptive backoff synchronization techniques. Em *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 396–406, 1989.

[5] V. Agarwal, M. S. Hrishikesh, S. W. Keckler e D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. Em *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 248–259, 2000.

[6] K. Agrawal, J. T. Fineman e J. Sukha. Nested parallelism in transactional memory. Em *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pp. 163–174, fevereiro de 2008.

[7] K. Agrawal, I.-T. A. Lee e J. Sukha. Safe open-nested transactions through ownership. Em *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pp. 151–162, fevereiro de 2009.

[8] K. Agrawal, C. E. Leiserson e J. Sukha. Memory models for open-nested transactions. Em *Proceedings of the 2006 workshop on Memory system performance and correctness*, pp. 70–81, outubro de 2006.

[9] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. Em *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 483–485, abril de 1967.

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson e S. Lie. Unbounded transactional memory. Em *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pp. 316–327, fevereiro de 2005.

[11] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Lujan e K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. Em *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, pp. 196–207, 2008.

[12] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'Brien e K. O'Brien. A novel asynchronous software cache implementation for the Cell-BE processor. Em *20th International Workshop on Languages and Compilers for Parallel Computing*, pp. 125–140, outubro de 2008.

[13] A. Baldassin, F. Klein, G. Araujo, R. Azevedo e P. Centoducatte. Characterizing the energy consumption of software transactional memory. *IEEE Computer Architecture Letters*, agosto de 2009.

[14] A. Baldassin e S. Burckhardt. Lightweight software transactions for games. Em *First USENIX Workshop on Hot Topics in Parallelism*, março de 2009.

[15] C. Blundell, J. Devietti, E. C. Lewis e M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. Em *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 24–34, junho de 2007.

[16] C. Blundell, E. Lewis e M. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, julho-dezembro de 2006.

[17] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift e D. A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. Em *Proceedings of the 35th International Symposium on Computer Architecture*, pp. 127–138, junho de 2008.

[18] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras e S. Chatterjee. Software transactional memory: Why is it only a research toy? *Communications of the ACM*, 51(11):40–46, novembro de 2008.

[19] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer e M. Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, março/abril de 2009.

[20] T. Chen, T. Zhang, Z. Sura e M. G. Tallada. Prefetching irregular references for software cache on Cell. Em *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 155–164, abril de 2008.

[21] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder e O. Colavin. Unbounded page-based transactional memory. Em *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 347–358, outubro de 2006.

[22] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis e K. Olukotun. The common case transactional behavior of multithreaded programs. Em *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 266–277, fevereiro de 2006.

[23] P. Damron, A. Fedorova e Y. Lev. Hybrid transactional memory. Em *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 336–346, outubro de 2006.

[24] V. De e S. Borkar. Technology and design challenges for low power and high performance. Em *Proceedings of the 1999 international symposium on Low power electronics and design*, pp. 163–168, agosto de 1999.

[25] J. B. Dennis e E. C. V. Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, março de 1966.

[26] D. Dice, Y. Lev, M. Moir e D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. Em *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 157–168, março de 2009.

[27] D. Dice, O. Shalev e N. Shavit. Transactional locking II. Em *20th International Symposium on Distributed Computing*, pp. 194–208, setembro de 2006.

[28] D. Dice e N. Shavit. Understanding tradeoffs in software transactional memory. Em *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 21–33, março de 2007.

[29] E. W. Dijkstra. Cooperating sequential processes. Relatório Técnico EWD-123, Technological University, 1965.

[30] A. Dragojevic, R. Guerraoui e M. Kapalka. Stretching transactional memory. Em *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 155–165, junho de 2009.

[31] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao e M. Gschwind. Optimizing compiler for the CELL processor. Em *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pp. 161–172, setembro de 2005.

[32] R. Ennals. Software transactional memory should not be obstruction free. Relatório Técnico IRC-TR-06-052, Intel Research Cambridge, 2006.

[33] K. P. Eswaran, J. N. Gray, R. A. Lorie e I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, novembro de 1976.

[34] P. Felber, C. Fetzer e T. Riegel. Dynamic performance tuning of word-based software transactional memory. Em *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pp. 237–246, fevereiro de 2008.

[35] C. Ferri, T. Moreshet, R. I. Bahar, L. Benini e M. Herlihy. A hardware/software framework for supporting transactional memory in a MPSoC environment. *ACM SIGARCH Computer Architecture News*, 35(1):47–54, março de 2007.

[36] C. Ferri, A. Viescas, T. Moreshet, R. I. Bahar e M. Herlihy. Energy efficient synchronization techniques for embedded architectures. Em *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pp. 435–440, maio de 2008.

[37] M. J. Flynn e P. Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, 2005.

[38] K. Fraser. Practical lock-freedom. Relatório Técnico 579, University of Cambridge, fevereiro de 2004.

[39] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris e M. Valero. QuakeTM: Parallelizing a complex sequential application using transactional memory. Em *Proceedings of the 23rd international conference on Supercomputing*, pp. 126–135, junho de 2009.

[40] P. P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. Em *Proceedings of the 2001 International Solid-State Circuits Conference*, pp. 22–25, 2001.

[41] F. Goldstein, A. Baldassin, P. Centoducatte, R. Azevedo e L. A. G. Garcia. A software transactional memory system for an asymmetric processor architecture. Em *Proceedings of the 20th International Symposium on Computer Architecture and High Performance Computing*, pp. 175–182, outubro/novembro de 2008.

[42] M. Gonzalez, N. Vujic, X. Martorell, E. Ayguade, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien e K. O'Brien. Hybrid access-specific software cache techniques for the Cell BE architecture. Em *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 292–302, outubro de 2008.

[43] J. Gray e A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[44] R. Guerraoui e M. Kapalka. On obstruction-free transactions. Em *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 304–313, junho de 2008.

[45] R. Guerraoui e M. Kapalka. On the correctness of transactional memory. Em *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pp. 175–184, fevereiro de 2008.

[46] R. Guerraoui e M. Kapalka. The semantics of progress in lock-based transactional memory. Em *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 404–415, janeiro de 2009.

[47] R. Guerraoui, M. Kapalka e J. Vitek. STMBench7: A benchmark for software transactional memory. Em *Proceedings of the 2nd European Conference on Computer Systems*, pp. 315–324, março de 2007.

[48] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis e K. Olukotun. Transactional memory coherence

and consistency. Em *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 102–113, junho de 2004.

[49] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith e M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, maio/junho de 2007.

[50] T. Harris e K. Fraser. Language support for lightweight transactions. Em *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 388–402, outubro de 2003.

[51] T. Harris, S. Marlow, S. Peyton-Jones e M. Herlihy. Composable memory transactions. Em *Proceedings of the 10th Symposium on Principles and Practice of Parallel Programming*, pp. 48–60, junho de 2005.

[52] T. Harris, M. Plesko, A. Shinnar e D. Tarditi. Optimizing memory transactions. Em *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 14–25, junho de 2006.

[53] M. Herlihy e E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. Em *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pp. 207–216, fevereiro de 2008.

[54] M. Herlihy, V. Luchangco, M. Moir e W. N. Scherer. Software transactional memory for dynamic-sized data structures. Em *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pp. 92–101, julho de 2003.

[55] M. Herlihy e J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. Em *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 289–300, junho de 1993.

[56] M. Herlihy e N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[57] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, outubro de 1974.

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

[59] International Business Machines Corporation, Sony Computer Entertainment Inc. e Toshiba Corporation. *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*, abril de 2009.

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

[61] S. Kang e D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. Em *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pp. 15–24, fevereiro de 2009.

[62] S. Kaxiras e M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan & Claypool Publishers, 2008.

[63] F. Klein, A. Baldassin, G. Araujo, P. Centoducatte e R. Azevedo. On the energy-efficiency of software transactional memory. Em *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design*, pp. 1–6, setembro de 2009.

[64] T. Knight. An architecture for mostly functional languages. Em *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 105–112, 1986.

[65] E. Koskinen, M. Parkinson e M. Herlihy. Coarse-grained transactions. Em *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (a aparecer)*, 2010.

[66] G. E. Krasner e S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, agosto/setembro de 1988.

[67] R. Kumar, D. M. Tullsen, N. Jouppi e P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, novembro de 2005.

[68] S. Kumar, M. Chu, C. J. Hughes, P. Kundu e A. Nguyen. Hybrid transactional memory. Em *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pp. 209–220, março de 2006.

[69] H. T. Kung e J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, junho de 1981.

[70] J. Larus e C. Kozyrakis. Transactional memory. *Communications of the ACM*, 51(7):80–88, julho de 2008.

[71] J. R. Larus e R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[72] N. Leveson e C. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, julho de 1993.

[73] A. S. Lobao, B. P. Evangelista e J. A. L. de Farias. *Beginning XNA 2.0 Game Programming: From Novice to Professional*. Apress, 2008.

[74] M. Loghi, M. Poncino e L. Benini. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. pp. 410–406, abril de 2004.

[75] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. Em *Proceedings of an ACM conference on Language design for reliable software*, pp. 128–137, março de 1977.

[76] V. J. Marathe, W. N. Scherer e M. L. Scott. Adaptive software transactional memory. Em *19th International Symposium on Distributed Computing*, pp. 354–368, setembro de 2005.

[77] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer e M. L. Scott. Lowering the overhead of nonblocking software transactional memory. Em *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, junho de 2006.

[78] V. J. Marathe e M. Moir. Toward high performance nonblocking software transactional memory. Em *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pp. 227–236, fevereiro de 2008.

[79] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis e K. Olukotun. Architectural semantics for practical transactional memory. Em *Proceedings of the 33rd International Symposium on Computer Architecture*, pp. 53–65, junho de 2006.

[80] M. Mehrara, J. Hao, P.-C. Hsu e S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. Em *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 166–176, junho de 2009.

[81] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha e A. Welc. Practical weak-atomicity semantics for Java STM. Em *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 314–325, junho de 2008.

[82] M. M. Michael e M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Em *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pp. 267–275, maio de 1996.

[83] C. C. Minh, J. Chung, C. Kozyrakis e K. Olukotun. STAMP: Stanford transactional applications for multi-processing. Em *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 35–46, setembro de 2008.

[84] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis e K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. Em *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 69–80, June de 2007.

[85] G. E. Moore. Progress in digital integrated electronics. Em *International Electron Devices Meeting*, pp. 11–13, 1975.

[86] G. E. Moore. No exponential is forever: But "forever" can be delayed! Em *Proceedings of the 2003 International Solid-State Circuits Conference*, pp. 20–23, fevereiro de 2003.

[87] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill e D. A. Wood. LogTM: Log-based transactional memory. Em *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 254–265, fevereiro de 2006.

[88] K. F. Moore e D. Grossman. High-level small-step operational semantics for transactions. Em *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 51–62, janeiro de 2008.

[89] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift e D. A. Wood. Supporting nested transactional memory in logTM. Em *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 359–370, outubro de 2006.

[90] T. Moreshet, R. I. Bahar e M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. Em *Proceedings of the 2005 international symposium on Low power electronics and design*, pp. 331–334, agosto de 2005.

[91] J. E. B. Moss e A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, dezembro de 2006.

[92] S. G. Narendra. Challenges and design choices in nanoscale CMOS. *ACM Journal on Emerging Technologies in Computing Systems*, 1(1):7–49, março de 2005.

[93] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha e T. Shpeisman. Open nesting in software transactional memory. Em *Proceedings of the 12th Symposium on Principles and Practice of Parallel Programming*, pp. 68–78, março de 2007.

[94] M. Olszewski, J. Cutler e J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. Em *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pp. 365–375, setembro de 2007.

[95] K. Olukotun e L. Hammond. The future of microprocessors. *Queue*, 3(7):26–34, setembro de 2005.

[96] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson e K. Chang. The case for a single-chip multiprocessor. Em *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–11, outubro de 1996.

[97] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, outubro de 1979.

[98] K. Poulsen. Tracking the blackout bug. Online em SecurityFocus (http://www.securityfocus.com/news/8412), abril de 2004.

[99] M. K. Prabhu e K. Olukotun. Using thread-level speculation to simplify manual parallelization. Em *Proceedings of the 9th Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, junho de 2003.

[100] R. Rajwar, M. Herlihy e K. Lai. Virtualizing transactional memory. Em *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 494–505, junho de 2005.

[101] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari e E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. Em *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 92–103, June de 2007.

[102] H. E. Ramadan, C. J. Rossbach e E. Witchel. Dependence-aware transactional memory for increased concurrency. Em *Proceedings of the 41st ACM/IEEE International Symposium on Microarchitecture*, pp. 246–257, novembro de 2008.

[103] H. E. Ramadan, I. Roy, M. Herlihy e E. Witchel. Committing conflicting transactions in an STM. Em *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pp. 163–172, fevereiro de 2009.

[104] M. Rinard e P. Diniz. Eliminating synchronization bottlenecks in object-based programs using adaptive replication. Em *Proceedings of the 13th annual international conference on Supercomputing*, pp. 83–92, junho de 1999.

[105] B. Saha, A. Adl-Tabatabai e Q. Jacobson. Architectural support for software transactional memory. Em *Proceedings of the 39th annual ACM/IEEE International Symposium on Microarchitecture*, pp. 185 – 196, dezembro de 2006.

[106] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh e B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. Em *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pp. 187–197, março de 2006.

[107] F. T. Schneider, V. Menon, T. Shpeisman e A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. Em *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 181–194, outubro de 2008.

[108] M. L. Scott. Sequential specification of transactional memory semantics. Em *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, junho de 2006.

[109] M. L. Scott, M. F. Spear, L. Dalessandro e V. J. Marathe. Delaunay triangulation with transactions and barriers. Em *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 107–113, setembro de 2007.

[110] S. Seo, J. Lee e Z. Sura. Design and implementation of software-managed caches for multicores with local memory. Em *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, pp. 55–66, fevereiro de 2009.

[111] N. Shavit e D. Touitou. Software transactional memory. Em *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pp. 204–213, agosto de 1995.

[112] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore e B. Saha. Enforcing isolation and ordering in STM. Em *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 78–88, junho de 2007.

[113] A. Shriraman, S. Dwarkadas e M. L. Scott. Flexible decoupled transactional memory support. Em *Proceedings of the 35th International Symposium on Computer Architecture*, pp. 139–150, junho de 2008.

[114] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas e M. L. Scott. An integrated hardware-software approach to flexible transactional memory. Em *Proceedings of the 34th International Symposium on Computer Architecture*, pp. 104–115, June de 2007.

[115] M. F. Spear, M. M. Michael, M. L. Scott e P. Wu. Reducing memory ordering overheads in software transactional memory. Em *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 13–24, março de 2009.

[116] M. F. Spear, M. M. Michael e C. von Praun. RingSTM: Scalable transactions with a single atomic instruction. Em *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 275–284, junho de 2008.

[117] J. M. Stone, H. S. Stone, P. Heidelberger e J. Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel & Distributed Technology: Systems & Application*, 1(4):58–71, novembro de 1993.

[118] H. Sutter e J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[119] P. Tinker e M. Katz. Parallel execution of sequential Scheme with ParaTran. Em *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pp. 28–39, julho de 1988.

[120] D. M. Tullsen, S. J. Eggers e H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. Em *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392–403, junho de 1995.

[121] D. W. Wall. Limits of instruction-level parallelism. Relatório Técnico 93/6, Western Research Laboratory, novembro de 1993.

[122] C. Wang, W.-Y. Chen, Y. Wu, B. Saha e A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. Em *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 34–48, março de 2007.

[123] I. Watson, C. Kirkham e M. Lujan. A study of a transactional parallel routing algorithm. Em *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pp. 388–398, setembro de 2007.

[124] D. Weller, A. S. Lobao e E. Hatton. *Beginning .NET Game Programming in C#*. Apress, 2004.

[125] D. Wilner. Vx-Files: what really happened on Mars? Keynote at the 18th IEEE Real-Time Systems Symposium, dezembro de 1997.

[126] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh e A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. Em *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24–36, junho de 1995.

[127] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift e D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. Em *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pp. 261–272, fevereiro de 2007.

[128] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai e H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. Em *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 265–274, junho de 2008.

[129] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris e M. Valero. Atomic Quake: Using transactional memory in an interactive multiplayer game server. Em *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pp. 25–34, fevereiro de 2009.