

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"

Pós-Graduação em Ciência da Computação

PhTM*: Uma Implementação Eficiente de Transações em Fases

João Paulo Labegalini de Carvalho

Rio Claro – SP
2016

João Paulo Labegalini de Carvalho

PhTM*: Uma Implementação Eficiente de Transações em Fases

Orientador: Prof. Dr. Alexandro José Baldassin

Dissertação de Mestrado elaborada junto ao Programa de Pós-Graduação em Ciência da Computação – Área de Concentração em Arquitetura de Computadores e Sistemas Distribuídos, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Rio Claro – SP
2016

Carvalho, João Paulo Labegalini de.

PhTM* : uma implementação eficiente de transações em fases / João Paulo Labegalini de Carvalho. -- São José do Rio Preto, 2016

60 f. : il., gráfs., tabs.

Orientador: Alexandro José Baldassin

Dissertação (mestrado) – Universidade Estadual Paulista “Júlio de Mesquita Filho”, Instituto de Biociências, Letras e Ciências Exatas

1. Computação. 2. Programação (Computação) 3. Sistema de transação (Sistemas de computação) 4. Sistemas híbridos. I. Baldassin, Alexandro José. II. Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Biociências, Letras e Ciências Exatas. III. Título.

CDU – 518.721

Ficha catalográfica elaborada pela Biblioteca do IBILCE
UNESP - Câmpus de São José do Rio Preto

PhTM*: Uma Implementação Eficiente de Transações em Fases

João Paulo Labegalini de Carvalho¹

07 de Junho de 2016

Banca Examinadora:

- Prof. Dr. Alexandro José Baldassin (Orientador)
Departamento de Estatística, Matemática Aplicada e Computação
Universidade Estadual Paulista – UNESP
- Prof. Dr. Márcio Bastos Castro
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina – UFSC
- Dr. Marcio Machado Pereira
Instituto de Computação
Universidade Estadual de Campinas – UNICAMP
- Prof. Dr. Aleardo Manacero Junior (Suplente)
Departamento de Ciência da Computação e Estatística
Universidade Estadual Paulista – UNESP
- Prof. Dr. Rodolfo Jardim de Azevedo (Suplente)
Instituto de Computação
Universidade Estadual de Campinas – UNICAMP

Rio Claro – SP
07 de Junho de 2016

¹Suporte financeiro pela FAPESP sob o processo 2014/00534-8.

RESUMO

As evidentes limitações em se explorar o aumento de frequência e otimização no nível de microarquitetura para obtenção de desempenho levaram a indústria de microprocessadores a buscar alternativas para suprir tal demanda. A solução encontrada foi construir arquiteturas com múltiplos fluxos de execução (*multicore*). No entanto, a utilização imediata do modelo paralelo exige que os programadores precisem explicitamente codificar as aplicações de forma a usar todos os núcleos, tarefa essa nada trivial e passível de erros que, muitas vezes, são difíceis de serem detectados. Um novo modelo de programação concorrente conhecido como memória transacional (*Transactional Memory* – TM) oferece abstrações para fácil codificação de aplicações e é capaz de explorar o paralelismo das arquiteturas *multicore* sem exigir conhecimentos da arquitetura por parte do programador. Porém, as implementações desse modelo, em *software* ou *hardware*, isoladamente apresentam problemas que limitam a escalabilidade das aplicações. Objetivando resolver tais problemas, foram propostos sistemas híbridos de transações em *hardware* e *software* para melhor exploração do paralelismo, denominado Memória Transacional Híbrida (*Hybrid Transactional Memory* – HyTM). A flexibilidade dos sistemas híbridos convencionais, mesmo permitindo executar transações em *hardware* e *software* simultaneamente, reduzem o desempenho de ambos seus componentes e tornam mais complexa a implementação da componente em *software*. Nesse sentido, este trabalho apresenta a primeira implementação eficiente (PhTM*) de um sistema de transações em fases, proposto como alternativa aos HyTMs convencionais. O PhTM* elimina a instrumentação das transações em *hardware*, necessária em outros HyTMs, e permite utilizar qualquer STM, sem nenhuma modificação, para executar as transações em *software*. A proposta é a primeira a utilizar o suporte transacional em *hardware* disponível em um processador real (não-emulado), para executar as transações em *hardware*. Os resultados mostram que o PhTM* foi capaz de escolher o modo mais adequado para todas as aplicações do STAMP utilizadas. De fato, o PhTM* apresentou o melhor desempenho entre os sistemas transacionais avaliados e foi cerca de 12% melhor que o HyTM-NOrec, sistema híbrido considerado *baseline* na avaliação de novos HyTMs.

Palavras-chave: Programação Concorrente. Memória Transacional. Sistemas Híbridos. Fases do Programa.

ABSTRACT

The limited performance gain due to micro-architectural optimizations and the modest increase in clock rate led the microprocessor industry to seek alternatives to single-core processors in order to keep up with the performance demand. The solution found was to build architectures with multiple execution flows (multicore). Nonetheless, the parallel model's immediate adoption requires programmers to explicitly code applications in a way that all cores are used. Yet this task has proven to be non-trivial and prone to errors, which are generally hard to detect. A new concurrent programming model known as transactional memory (TM) brings abstractions that ease the synchronization burdens of parallel coding and enables programmers to better exploit the parallelism of multicore architectures without knowing their details. However, software and hardware implementations of the TM model have proven to be limited when adopted in isolation. Hybrid transactional systems (HyTM) are the proposed solution to these problems, allowing applications to benefit from multicore machines. The flexibility of conventional hybrid systems, even though allowing simultaneous execution of hardware and software transactions, decrease the performance of both hardware and software and increase the complexity of the software component. In this direction, this work presents the first efficient implementation (PhTM*) of phase-based transactional system, a viable alternative to conventional HyTMs. PhTM* removes the additional instrumentation of hardware transactions, necessary in other HyTMs, and can use any STM library to execute software transactions. The proposal is the first to use hardware transactional support, available in a real (not emulated) processor, to run hardware transactions. The results show that PhTM* is capable of always choosing the best suited mode for all STAMP applications. In fact, PhTM* showed the best overall performance and was about 12% better than HyTM-NOrec, a *de facto* baseline in HyTM evaluation.

Keywords: Concurrent Programming. Transactional Memory. Hybrid Systems. Program Phase.

Lista de Figuras

2.1	Ciclo de vida das <i>threads</i> no modelo Fork-Join.	11
2.2	Exemplo de uma sequência que produz um resultado inconsistente (retirado de [7]).	13
2.3	Exemplo de uma sequência indecidível do ponto de vista de <i>B</i>	15
3.1	(a) Baseada em objetos. Baseado em palavras: (b) completamente associativo, (c) parcialmente associativo e (d) único (retirado de [30]).	24
3.2	Representação dos objetos do tipo <code>TMObject</code> e <code>Localizador</code> (retirado de [30]).	27
3.3	Abertura de objeto para escrita (retirado de [30]).	28
4.1	Comparação das abordagens de leitura da trava global (fonte [11]).	33
4.2	Transição entre os 5 tipos de transação no Invyswell (fonte [10]).	35
4.3	Modo rápido e misto no sistema RH-NOrec[45].	36
4.4	PhTM – (a) Configuração dos campos de estado e (b) automato de transição.	38
5.1	Speedup das aplicações STAMP em cada abordagem de paralelização. . . .	41
5.2	Autômato de transição do PhTM*.	43
5.3	Comparando o desempenho da versão automática (libitm) e manualmente anotada (macros) das aplicações STAMP.	47
6.1	Registradores de controle da PMU: (a) controle dos contadores programáveis; (b) controle dos contadores fixos; (c) controle global.	52
6.2	Árvore Rubro-Negra.	54
6.3	Lista Ligada.	54
6.4	Lista Ligada com 100% de buscas e 4 <i>threads</i>	55
6.5	STAMP – Comparação do <i>speedup</i> das aplicações usando cada um dos sistemas.	56

Lista de Tabelas

2.1	Comparação do uso de travas, semáforos e variáveis condicionais.	17
3.1	Leitura e escrita em um STM com ETL e versionamento ansioso.	25
3.2	Leitura e <i>commit</i> em um STM com ETL e versionamento preguiçoso. . . .	26
3.3	Operações de <i>lock</i> e <i>unlock</i> com omissão de travas em <i>hardware</i>	30
3.4	Um exemplo de implementação das operações TXSTART e TXCOMMIT	31
5.1	Porcentagem de ciclos e taxa de finalização por transação (RTM).	42
6.1	Aplicações do STAMP – parâmetros, domínio da aplicação e descrição. . .	49
6.2	Características dos Sistemas.	51
6.3	Eventos da PMU utilizados.	53
6.4	Porcentagem do tempo em cada modo para as aplicações do STAMP. . . .	58

SUMÁRIO

1	Introdução	7
1.1	Objetivos do Trabalho	8
1.2	Contribuições	9
1.3	Organização do Texto	9
2	Programação Concorrente	11
2.1	Pensar Serial vs. Pensar Paralelo	12
2.2	Sincronização	13
2.2.1	Problema de Consenso	14
2.2.2	Mecanismos Bloqueantes	15
2.2.3	Mecanismos Não-bloqueantes	16
3	Memória Transacional	19
3.1	Componentes de um Sistema de TM	20
3.1.1	Controle de Concorrência	20
3.1.2	Gerenciamento de Versões	21
3.1.3	Deteção e Resolução de Conflitos	21
3.2	Semântica e Progresso de Transações	22
3.3	Memória Transacional em Software	23
3.3.1	Gerenciamento dos Logs e Metadados	23
3.3.2	Implementações Baseadas em Travas	25
3.3.3	Implementações Não-Bloqueantes	27
3.4	Memória Transacional em Hardware	28
3.4.1	Memória Transacional no Haswell	29
3.4.1.1	Omissão de Locks em Hardware (HLE)	29
3.4.1.2	Memória Transacional Reduzida (RTM)	30
4	Trabalhos Relacionados	32
4.1	Melhorando o Desempenho de HTM	32
4.2	Sistemas Transacionais Híbridos	33
4.2.1	Sistemas Híbridos de Modo Local	34
4.2.2	Sistemas Híbridos de Modo Global	37
5	Uma Implementação Eficiente de Transações em Fases	39
5.1	Motivação	39
5.2	Avaliação Preliminar que Embasa a Motivação	40
5.3	PhTM*	42
5.3.1	Algoritmo	43
5.3.2	Aspectos de implementação	46

6	Avaliação Experimental da Proposta	48
6.1	Aplicações	48
6.1.1	Intset	48
6.1.2	STAMP	49
6.2	Sistemas Transacionais	50
6.2.1	RTM	50
6.2.2	NOrec	50
6.2.3	HyTM-NOrec	50
6.3	Ambiente Experimental e Metodologia	51
6.3.1	Perfilamento de Eventos nos Processadores Intel®	51
6.3.2	Metodologia	53
6.4	Avaliação Experimental	53
6.4.1	Intset	53
6.4.2	STAMP	56
6.5	Desafios e Limitações de Transações em Fases	58
7	Conclusão	60
7.1	Trabalhos Futuros	60
	Bibliografia	62

Capítulo 1

Introdução

O desenvolvimento tecnológico permite que indústria de microprocessadores construa máquinas cada vez mais rápidas, seguindo a tendência da Lei de Moore [47]. Nossa história recente revela que o rápido progresso da ciência se deve, principalmente, pela larga utilização de sistemas computacionais como ferramenta. Tal progresso, por sua vez, leva à necessidade de se resolver problemas ainda mais complexos. Essa relação simbiótica, que se traduz na crescente demanda por desempenho, levou a indústria de microprocessadores a modificar seus processadores tendo em vista atender essa necessidade.

Nos últimos 40 anos, a pilha de *software* alcançou maior desempenho a cada nova geração de microprocessadores, sem modificação do código-fonte, devido principalmente ao avanço de três áreas: frequência de *clock*, otimização de execução (ex.: ILP, *Instruction Level Parallelism*) e memória *cache*. No entanto, a observação dos efeitos *Power Wall* [8] desacelerou o aumento da frequência de operação dos processadores e reduziu a velocidade de transferência de dados entre a memória e o processador. Em um programa típico, o número de instruções independentes é geralmente reduzido [57] pela forma como o mesmo foi escrito, reduzindo as oportunidades de exploração do ILP. Aumentar a janela de instruções pode amenizar esse efeito, no entanto esta solução esbarra na *Memory Wall* [59]. Em resumo, os processadores de fluxo único de execução não serão capazes de atender a crescente demanda por desempenho. Como alternativa às abordagens baseadas em um único fluxo de execução, foram construídas as microarquiteturas com múltiplas unidades de execução (*multicore*).

A utilização de arquiteturas *multicore* traz desafios e a necessidade de se revolucionar a forma como pensamos sobre *software*. Os modelos e ferramentas atualmente disponíveis não são suficientes para permitir uma exploração significativa de todos os núcleos de execução, fazendo-se necessárias novas abordagens [55]. Um novo modelo de programação concorrente, conhecido como Memória Transacional (TM) [30], abstrai o controle de concorrência em sistemas de múltiplas linhas de execução (*threads*) usando o conceito de transação da área de Banco de Dados, introduzido por Eswaran et al. [25].

O investimento na área de TM não é restrito apenas à academia. Recentemente a indústria de microprocessadores mostrou interesse e investimentos concretos na área. A exemplo podemos citar o *Blue Gene/Q*TM [58], supercomputador destinado à computação de alto desempenho, lançado em 2012 pela IBM[®] que possui suporte em *hardware* para Memória Transacional (HTM). Outro exemplo de máquina com suporte a HTM é o processador *Haswell*TM, lançado pela Intel[®] em Junho de 2013. Diferentemente do supercomputador da IBM[®], o processador da Intel[®] pode ser empregado em uma gama maior de dispositivos, desde máquinas para computação de alto desempenho até dispositivos

móveis (*smartphones* e *tablets*).

O *Haswell*TM, processador que será utilizado nesse trabalho, estende a ISA (*Instruction-Set Architecture*) do *Sandy Bridge*TM (modelo anterior) implementando suporte em *hardware* para sincronização baseado no conceito de transações, denominado TSX (*Transactional Synchronization Extension*). Essa extensão é constituída de duas interfaces de programação, referenciadas no manual da Intel[®] por HLE (*Hardware Lock Elision*) e RTM (*Restricted Transactional Memory*) [3]. A primeira faz uso da execução especulativa tentando reduzir o custo da sincronização baseada em travas, omitindo as mesmas quando não há conflito de dados. Já a segunda é uma implementação reduzida do modelo transaccional. Em ambos os casos, a detecção de conflitos e o versionamento dos dados ocorre no nível de *cache* L1 e é realizado dinamicamente pelo *hardware*.

Apesar da grande flexibilidade dos sistemas de transações em software (STM), o *overhead* decorrente do gerenciamento das transações e barreiras de leitura e escrita podem tornar-se proibitivos para sua utilização em algumas aplicações. Nesse sentido, os sistemas em *hardware* (HTM) são uma alternativa interessante pois os custos de gestão das transações são desprezíveis, comparado ao dos STMs. Todavia, a maioria dos processadores implementam suporte para transações com garantia de melhor esforço (*best-effort*), ou seja, não garantem que todas as transações serão eventualmente efetivadas. A solução natural para a limitação de ambas encarnações (*software* e *hardware*) são os sistemas híbridos de TM (HyTM), que exploram as qualidades individuais dos HTMs e STMs ao permitir que transações executem em ambos os modos. Entretanto, em aplicações com pouca heterogeneidade no código executado pelas *threads*, o *overhead* introduzido para garantir a correta execução de transações em modos diferentes limita o potencial de aceleração de tais sistemas.

1.1 Objetivos do Trabalho

Sistemas de transações em fases (PhasedTM), diferente dos sistemas híbridos convencionais, executam as transações em modos escolhidos de acordo com as fases do programa. Em sistemas com fases, todas as transações executam em um mesmo modo global. Transações que desejam, ou necessitam, executar em um modo diferente do corrente precisam solicitar uma mudança de modo antes de iniciar. Apesar de parecerem muito restritivos, os sistemas de transações em fases apresentam um melhor desempenho que os HyTMs convencionais em aplicações com alta homogeneidade de código, pois não reduzem o desempenho de suas transações com a instrumentação adicional necessária na detecção de conflitos entre transações em modos diferentes.

O trabalho tem dois objetivos principais:

- Avaliar a viabilidade de sistemas de transações em fases, baseado na ideia de Lev et al. [40], propondo uma nova implementação que utiliza o suporte de TM disponível no processador *Haswell*TM para executar as transações em *hardware*.
- Contrastar o desempenho da implementação proposta com sistemas de transações em *software* e *hardware*, bem como o *estado da arte* dos sistemas híbridos.

O primeiro objetivo consiste em propor soluções aos problemas e desafios apontados por Lev et al. [40]. Os principais desafios são: (i) implementar políticas eficazes para guiar a mudança de fase; e (ii) implementar um mecanismo de mudança de fase que seja correto

e eficiente. O segundo objetivo propõe apresentar os sistemas de transações em fases, em particular o proposto neste trabalho (PhTM*), como alternativa viável aos sistemas híbridos convencionais. A principal motivação para executar transações em fases é que tais sistemas permitem executar as transações com quase o mesmo desempenho do sistema *puro* original. Isso é possível pois a instrumentação adicional, presente nos sistemas híbridos convencionais, pode ser eliminada uma vez que apenas transações no mesmo modo executam em paralelo. Inicialmente essa restrição de execução pode parecer muito constritiva, mas em aplicações com alta homogeneidade, a diferença de *overhead* na instrumentação torna os sistemas de transações em fases uma boa alternativa aos HyTMs convencionais.

1.2 Contribuições

Este trabalho atingiu todos objetivos propostos e teve seus resultados publicados a diferentes conferências. A seguir estão listados os trabalhos produzidos durante os estudos do mestrado.

1. *Caracterização de Desempenho e Transações em um Processador com Suporte para Execução Especulativa*. João P. L. de Carvalho e Alexandro Baldassin. *IV Escola Regional de Alto Desempenho de São Paulo* (ERAD-SP'15), 2015.
2. *Reavaliando o Conjunto de Aplicações STAMP em um Novo Hardware Transacional*. João P. L. de Carvalho, Rafael Pizzirani Murari e Alexandro Baldassin. *XVI Simpósio em Sistemas Computacionais de Alto Desempenho* (WSCAD'15), 2015.

O resumo estendido submetido à ERAD-SP'15 foi pioneiro ao mostrar a existência de uma transação dominante em todas as aplicações do STAMP. Posteriormente, esses resultados foram complementados e publicados na WSCAD'15. Esse trabalho analisou a correlação entre a taxa de efetivação das transações dominantes com o tamanho médio dos conjuntos de leitura e escrita, e projetou os resultados obtidos no *Haswell*TM para os demais processadores com suporte à execução especulativa. Estes dois primeiros trabalhos embasam a motivação do sistema proposto nessa dissertação. Os resultados que mostram a viabilidade dos sistemas de transações em fases e a comparação de desempenho com o *estado da arte* dos sistemas transacionais híbridos serão publicados em artigos futuros.

1.3 Organização do Texto

O texto desta tese está organizado em 6 capítulos. No Capítulo 2 é apresentado um estudo do cenário atual de programação concorrente. Em seguida, no Capítulo 3, a memória transacional é introduzida como uma alternativa viável ao uso de travas na sincronização de programas com múltiplas *threads*. Mais especificamente, são expostas as decisões de projeto de implementações em *software* e *hardware*. O estado da arte das propostas de melhoria de desempenho na utilização de HTMs e os principais sistemas híbridos também serão brevemente abordados no Capítulo 4. Este trabalho apresenta uma implementação eficiente de transações em fases, denominada PhTM*, no Capítulo 5. Como motivação da proposta, apresenta-se uma análise que contrasta o desempenho do suporte em *hardware* com o de uma biblioteca *software*. A análise detalhada das transações (Seção 5.2) revela

que as aplicações do STAMP possuem alta homogeneidade, o que as tornam mais amigáveis aos sistemas de transações em fases do que aos sistemas híbridos convencionais. A avaliação experimental, descrita no Capítulo 6, mostra que o sistema proposto é cerca de 12% melhor que o sistema híbrido HyTM-Norec [14]. Ao final da avaliação, são discutidos alguns desafios dos sistemas transacionais em fases (Seção 6.5). Por fim, são feitas algumas considerações finais e apresenta-se a conclusão no Capítulo 7.

Capítulo 2

Programação Concorrente

Programação concorrente é uma subárea da programação focada em desenvolver e implementar algoritmos que visam explorar mais eficientemente o paralelismo disponibilizado no sistema. Basicamente, existem dois modelos de programação concorrente: programação em memória compartilhada e em memória distribuída. O primeiro modelo engloba os algoritmos que executam em uma máquina *multicore* (ou multiprocessador) na qual a memória principal é acessível a todos os processadores. Já no segundo modelo os algoritmos executam em máquinas diferentes conectadas por uma rede, geralmente de alta velocidade, e a comunicação ocorre através da troca de mensagem.

Neste capítulo será discutido apenas o modelo de programação em memória compartilhada, visto que é no qual o modelo de Memória Transacional se baseia (vide Capítulo 3). No modelo de programação concorrente em memória compartilhada a unidade básica de execução é conhecida como *thread* (“linha de execução”). Um processo pode ser composto por uma ou mais *threads*, cada uma dessas *threads* possui contexto próprio de execução¹. No entanto o espaço de endereçamento é compartilhado entre as *threads* de um processo.

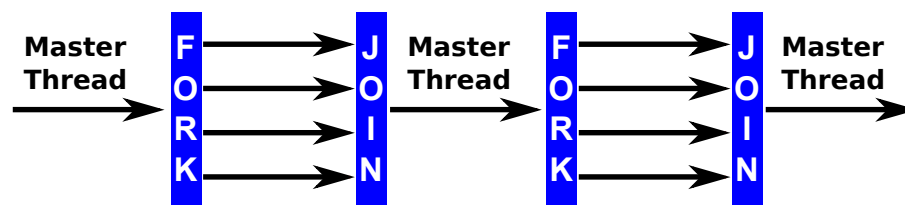


Figura 2.1: Ciclo de vida das *threads* no modelo Fork-Join.

O ciclo de vida mais simples das *threads* de uma aplicação concorrente pode ser representado pelo esquema da Figura 2.1. O esquema mostra a *thread* principal (*master thread*) disparando várias *threads* (*fork*) que executam e são sincronizadas ao final (*join*), quando a *thread* principal retoma a execução sequencial da aplicação. Esse é o esquema mais simples, porém um dos mais adotados. Outras aplicações podem possuir padrões diferentes no ciclo de vida das *threads*, por exemplo *threads* que não sincronizam ao final (*disjoint threads*). Ou seja, cabe ao programador decidir qual o padrão que melhor se adapta à cada aplicação. Cabe também ao programador optar por uma biblioteca como a Pthreads (POSIX *Threads*) [49], que permite gerenciar e sincronizar as *threads* manualmente, ou utilizar uma interface de programação de paralelização assistida por compilador como a OpenMP [12].

¹O sistema operacional fica encarregado de salvar os registradores e a pilha das *threads* quando um processo, ou *thread*, é retirado ou colocado em execução.

Na paralelização de um algoritmo, caso seja possível dividir os dados de maneira que cada *thread* acesse e modifique dados independentes, dizemos que o mesmo é embarçadamente paralelo (*embarrassingly parallel*). Essa tipo de algoritmo é bastante comum em aplicações científicas e de processamento gráfico. Mas caso as *threads* necessitem acessar e modificar regiões comuns de memória, se faz necessária a utilização de mecanismos de sincronização para garantir a consistência dos dados e do estado do programa. Usar correta e eficientemente tais mecanismos tem-se mostrado o real desafio da programação concorrente [55].

Na Seção 2.2 serão tratados alguns mecanismos de sincronização. A seção foi dividida em mecanismos bloqueantes (2.2.2) e mecanismos não-bloqueantes (2.2.3). Além da corretude do mecanismo de sincronização, outro aspecto importante a ser considerado é seu impacto no desempenho. Como será discutido, os mecanismos de sincronização limitam a ordem com que a execução ocorre e essa restrição pode introduzir gargalos decorrentes da serialização da execução.

A métrica amplamente utilizada para medir desempenho é o *speedup* (“aceleração”) com relação à execução sequencial. A fórmula mais conhecida é a enunciada na lei de Amdahl [5], como mostra a Equação 2.1. Ela define o *speedup* máximo, $S(p)$, obtido com p processadores em uma aplicação cuja fração do tempo serial sobre o tempo total é f_s . Ao extrapolarmos matematicamente a Equação 2.1 fazendo o número de processadores p se aproximar, no limite, do infinito, temos o apresentado na Equação 2.2.

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

$$\lim_{p \rightarrow \infty} S(p) = \frac{1}{f_s} \quad (2.2)$$

A Equação 2.2 revela que o *speedup* máximo está limitado pela fração serial do código, ou seja, para uma dado f_s , o *speedup* máximo é $1/f_s$, não importando quão grande seja o número de *threads*. Este resultado evidencia a importância em se reduzir a fração serial do código e ajustar a granularidade da sincronização.

No contexto de sincronização podemos dividir as situações em dois cenários [7]: (a) contenção, para o progresso de uma *thread* as demais *threads* tem seu progresso bloqueado; e (b) cooperação, o progresso de uma *thread* habilita o progresso das demais. O primeiro cenário está relacionado ao problema da *exclusão mútua*, e o segundo com o problema *produtor-consumidor* (vide [22]). No problema da *exclusão mútua* objetiva-se permitir que apenas uma *thread* execute uma região de código em um dado instante, ou seja, a *thread* executando a *região crítica* deve fazê-lo de maneira exclusiva. Já no problema produtor-consumidor existem no sistema dois tipos de agentes: o(s) produtor(es), que torna(m) disponível um recurso, e o(s) consumidor(es), que consome(m) esse recurso. A grande dificuldade é a sincronização eficiente dos acessos ao *buffer* compartilhado utilizado para armazenar o que é produzido pelo produtor para futuro consumo pelo consumidor.

2.1 Pensar Serial vs. Pensar Paralelo

A resolução de um problema envolve o processo de construção das abstrações de seus componentes (objetos), bem como a especificação do comportamento e comunicação entre eles (métodos). Até mesmo em uma linguagem não-orientada a objetos é possível estruturar o programa de maneira a mimetizar este modelo conceitual. Nesse modelo conceitual po-

demos descrever o comportamento do programa pela mudança de estado de seus objetos, produzida por uma sequência de invocações de seus respectivos métodos, considerando apenas seus estados antes e depois das invocações. Em outras palavras, se o estado de um objeto satisfaz um conjunto de condições antes da invocação de um método (pré-condições), então a invocação desse método garante que o estado deste objeto satisfará um outro conjunto de condições após a invocação deste método (pós-condições). Esse tipo de descrição é conhecido como *especificação sequencial* [39].

A especificação sequencial é uma ferramenta poderosa e suficiente para descrever corretamente a transição de estado de objetos sequenciais, ou seja, objetos de um programa sequencial (*single-threaded*). No entanto, quando um objeto sequencial é compartilhado por múltiplas *threads*, descrever precisamente seu estado entre as invocações de seus métodos poder ser impossível, devido à natural sobreposição das chamadas. Ou seja, é possível que durante uma chamada o método observe o objeto em um estado intermediário produzido por modificações incompletas de outra chamada concorrente. E ainda, a noção de ordem da mudança de estado intrínseca do modelo sequencial perde o sentido uma vez que diferentes permutações das chamadas são possíveis em um programa *multi-threaded*. Em resumo, uma semântica para objetos concorrentes se faz necessária para descrever como tais objetos devem se comportar na presença de múltiplas *threads*.

2.2 Sincronização

Nesta seção se discute a necessária sincronização do acesso a objetos concorrentes para satisfazer a corretude dos mesmos. Essa discussão será guiada pelo problema de consenso que está diretamente ligado a problemas de sincronização, e pode ser utilizado para avaliar a força de diferentes primitivas utilizadas para sincronizar objetos compartilhados. Os dois tipos de sincronização, bloqueante e não-bloqueante, bem como suas garantias de progresso também serão discutidos.

O acesso à mesma região de memória por duas ou mais *threads* deve ser sincronizado pois pode gerar inconsistências dependendo da ordem que as instruções das operações ocorrem. Por exemplo, vamos analisar o caso em que duas *threads* t_1 e t_2 devem incrementar uma variável compartilhada de endereço l em um. A operação de incremento geralmente ocorre em três passos: *load* do valor da variável para um registrador, incremento do registrador e *store* do valor incrementado do registrador no endereço da variável. Considerando a sequência ilustrada na Figura 2.2, podemos notar que o resultado gerado é incorreto, visto que o valor esperado em l é dois, e não um.

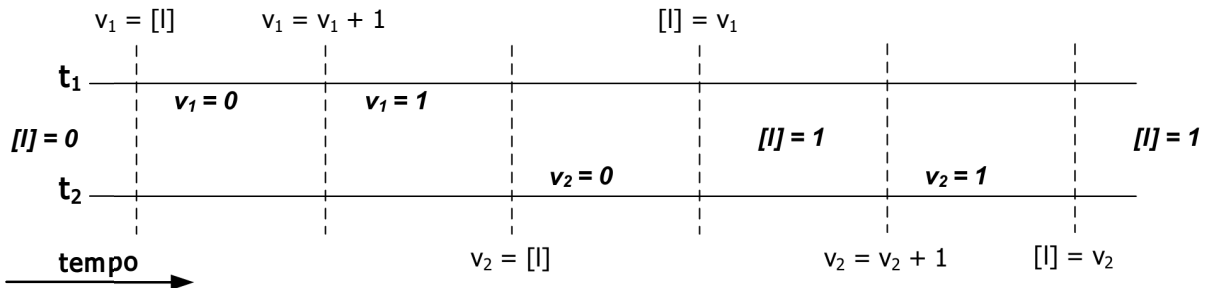


Figura 2.2: Exemplo de uma sequência que produz um resultado inconsistente (retirado de [7]).

Esta sequência ocorreu pois nenhum mecanismo de sincronização de acesso à região

l foi adotado, gerando uma condição de corrida (*race condition*). O uso da sincronização objetiva limitar a ordem em que as operações de diferentes *threads* são executadas de forma a impedir intercalações que produzam resultados inconsistentes. Note que uma expressão que incrementa uma variável parece ser atômica em uma linguagem de alto nível, no entanto mais de uma operação precisam ser executadas pelo processador para incrementar tal variável. A disparidade entre o que é expresso na linguagem e o que é realmente executado é responsável pelas principais dificuldades de um programador iniciante de programas concorrentes.

2.2.1 Problema de Consenso

O problema de consenso consiste em implementar o método `decidir(v)` tal que duas condições sejam satisfeitas para um número n de *threads*: (i) todas as *threads* decidem o mesmo valor quando chamam concorrentemente o método e (ii) o valor decidido (v_i) é o valor passado como parâmetro ao método por alguma *thread* i . Utilizando essa especificação de interface podemos formular as seguintes definições:

Definição 2.1: Uma classe $\mathcal{C}$ resolve o problema de consenso para n *threads* se existir uma implementação do método `decidir(v)` utilizando um número arbitrário de objetos de $\mathcal{C}$ e registradores atômicos².

Definição 2.2: O número de consenso de uma classe $\mathcal{C}$ é o maior número de *threads* n para o qual é possível implementar o método `decidir(v)`. Se tal número não existe, dizemos que o número de consenso da classe é *infinito*.

O problema de consenso está diretamente relacionado a sincronização, pois resolvê-lo é equivalente a decidir: qual *thread* pode entrar em uma região crítica; se há ou não a possibilidade de se consumir/produzir elementos de uma fila compartilhada e etc. Em resumo, se é possível resolver o problema de consenso entre um número arbitrário de *threads* é possível sincronizar o acesso dessas *threads* a uma região compartilhada de memória. A título de exemplo e para elucidar o porquê é errôneo pensar que é possível resolver o problema de consenso utilizando apenas registradores atômicos, consideremos o caso binário. Ou seja, queremos resolver o problema de consenso com entrada binária³ para duas *threads* A e B utilizando apenas registradores atômicos. Em outras palavras, é possível decidir qual foi a ordem de execução das *threads* apenas usando os valores de um número arbitrário de registradores atômicos.

Assuma, por contradição, que a classe de registradores atômicos resolve o problema de consenso binário entre as *threads* A e B . Partindo de um estado bivalente s , onde nenhuma das *threads* executou, a ordem de operações das *threads* deve ser decidível apenas nos valores dos registradores atômicos. Caso contrário, a classe de registradores atômicos não resolve o problema. Por convenção, quando A executa primeiro o valor decidido deve ser 0, caso contrário deve ser 1. A Figura 2.3 mostra duas sequências de operações que contradizem nossa suposição. Se B executa primeiro uma operação (lê ou escreve em um registrador atômico e compartilhado), representado pela transição $s \rightarrow s'$, e em seguida termina sua execução em isolamento, então no estado final e em s' o método `decidir(v)` deve retornar 1. Agora se A executa uma leitura, representada pela transição $s \rightarrow q$, e em seguida B realiza uma operação (no mesmo registrador que A ou não) e em seguida

²Registradores que permitem sua leitura e escrita atômicamente.

³Qualquer afirmação válida para entrada binária, por indução, vale para entradas de valor arbitrário.

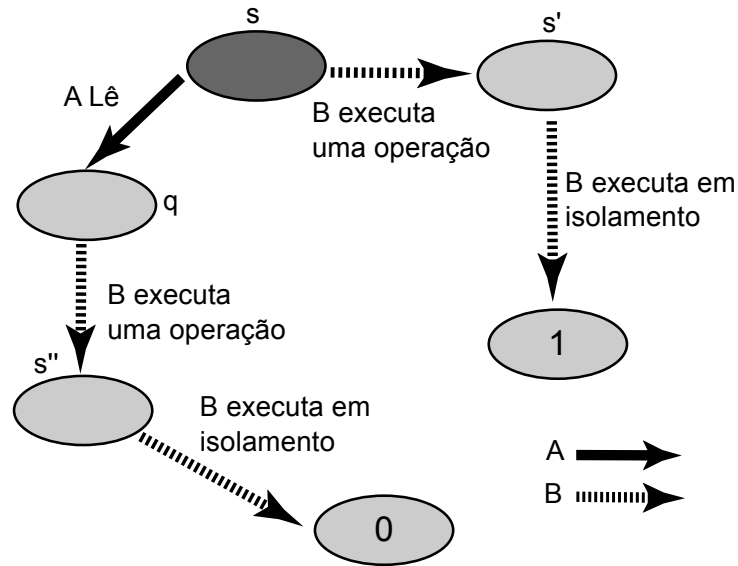


Figura 2.3: Exemplo de uma sequência indecidível do ponto de vista de *B*.

termina em isolamento, então como *A* executou primeiro o retorno do método `decidir(v)` deve ser 0 no estado final e em *s''*. Todavia, os estados *s'* e *s''* são indistinguíveis para *B* e logo *B* deve decidir o mesmo valor em ambos os estados. Isso é uma contradição, o que mostra que não é possível resolver o problema de consenso binário usando apenas registradores atômicos⁴.

Esse resultado é muito importante pois mostra que os processadores precisam fornecer primitivas mais poderosas do que apenas leitura e escrita atômicas para que seja possível implementar construções que sincronizem mais de 1 *thread*. De fato, os processadores modernos fornecem operações da classe RMW (*Read-Modify-Write*), como por exemplo `compareAndSwap` (CAS), que possuem valor de consenso infinito, ou seja, é possível sincronizar qualquer número de *threads* utilizando essas operações e registradores atômicos. Este resultado mostra que não é necessário fornecer todas as variantes de operações RMW, o que seria extremamente difícil de implementar e muito ineficiente.

2.2.2 Mecanismos Bloqueantes

Os mecanismos de sincronização bloqueantes são aqueles que bloqueiam a entrada de outras *threads* em um região crítica quando outra *thread* a está executando. As primitivas básicas de sincronização bloqueante são *travas* (*locks*) e *semáforos*. O objeto de trava *l* mais simples possível pode ser definido por um interface que contém os métodos `travar`, `l.lock()`, e `destravar`, `l.unlock()`. A implementação desses métodos garante a exclusão mútua do acesso à região protegida (seção/região crítica), ou seja, apenas uma *thread* conseguirá adquirir a trava. Antes de uma *thread* executar a região crítica a mesma chama o método `l.lock()` indicando que deseja obter acesso exclusivo à região crítica, e por consequência às variáveis protegidas pela trava *l*. Ao término da execução da seção crítica, a *thread* libera a trava chamando o método `l.unlock()`, o que permite que outras *threads* entrem e executem o trecho de código em questão.

O semáforo *s* é um objeto que pode ser definido simplesmente por dois métodos: `adquirir`, `s.acquire()`, e `liberar`, `s.release()`⁵. Analogamente às travas, quando um

⁴O número de consenso de registradores atômicos é 1.

⁵Esses métodos também são conhecidos por *wait* e *post*, respectivamente.

thread deseja executar uma seção crítica a mesma chama o método `s.acquire()`. Mas diferente do objeto *l* que tem valor binário (travado ou destravado), o objeto semáforo *s* tem seu valor de estado entre 0 e *c*, onde *c* é a capacidade do semáforo inicializada em seu construtor e que representa o número de *threads* que podem entrar simultaneamente na região crítica. Assim, a chamada do método `s.acquire()` decrementa esse valor de estado caso ele seja diferente de zero, caso contrário a *thread* espera até o valor ser não zero. Analogamente, o método `s.release()` incrementa o valor de estado de *s* caso o mesmo seja menor ou igual à $c - 1$, senão o mesmo permanece inalterado.

Pela descrição acima fica evidente que a associação entre objetos de trava, ou semáforo, e seções críticas pode ocorrer de maneira arbitrária e cabe a cada programador criar sua convenção e segui-la para garantir o funcionamento correto do mecanismo. Outras questões acerca do uso dessas primitivas são o *trade-off* entre desempenho e granularidade e o *trade-off* entre granularidade e complexidade e composição de código. O custo de primitivas como `Lock` e `Unlock` pode diminuir o desempenho caso a contenção seja alta. Logo, a tendência é de quanto menor a granularidade maior será o *overhead* e, por consequência, menor o desempenho. Mas se a granularidade for muito grossa, maior a fração serial do código e pela lei de Amdahl menor o desempenho.

O Código 2.1 exemplifica o uso de travas, semáforos e variáveis condicionais. No trecho de código da Tabela 2.1(c) uma trava *l* é usada para sincronizar o acesso a um contador global (`GlobalCounter`). Enquanto que no trecho da Tabela 2.1(a) os semáforos `empty` e `full` são usados para controlar a entrada na região crítica do produtor e consumidor⁶, respectivamente. O semáforo `mutex` é usado para controlar o acesso ao *buffer* compartilhado. O comportamento do código usando semáforos pode ser emulado utilizando travas e *variáveis de condição* (*conditional variables*). Essas variáveis condicionam a aquisição e liberação de uma trava de acordo com um predicado. Por exemplo, no trecho da Tabela 2.1(b) o *lock* somente é adquirido quando o *buffer* permite inserções ou remoções. O método `Queue.full()` indica ao consumidor quantos elementos existem na fila para serem consumidos, enquanto que o método `Queue.empty()` indica o número de vagas disponíveis no *buffer*. As variáveis condicionais `fullCond` e `emptyCond` são usadas para sinalizar, respectivamente, o consumidor e o produtor.

Os objetos básicos de sincronização bloqueante tem garantia de progresso denominada dependente. O progresso de um mecanismo é dito dependente se o mesmo não é garantido apenas pela implementação. Ou seja, certas condições externas devem ser satisfeitas. Por exemplo, uma implementação que usa travas pode entrar em *deadlock* caso a *thread* detentora da trava seja *preemptada*⁷ e nunca mais escalonada, devido à diferença de prioridade por exemplo. Dizemos que os mecanismos bloqueantes são dependentes da política de escalonamento do sistema operacional. Todavia é possível implementar construções utilizando esses mecanismos básicos que são livres de *deadlock* e *livelock*.

2.2.3 Mecanismos Não-bloqueantes

Os mecanismos não-bloqueantes são uma alternativa à abordagem bloqueante apresentada anteriormente. Além das liberdades discutidas a seguir, a principal vantagem da sincronização não-bloqueante são suas garantias de progresso independentes. Ou seja, é garantido que implementações livres de espera (*wait-free*) e de travas (*lock-free*) façam pro-

⁶Por brevidade apenas o código do método produtor é mostrada.

⁷O autor toma a liberdade de utilizar esse neologismo para indicar o ato de uma *thread* ser retirada de execução (*preemption*).

<pre> public Class Buffer<T> { private Semaphore empty; private Semaphore full; private Semaphore mutex; private Queue<T> b; public produce(T item) { empty.acquire(); mutex.acquire(); b.enq(item); mutex.release(); full.release(); } } </pre>	<pre> public Class Buffer<T> { private Lock l; private CondVar emptyCond; private CondVar fullCond; private Queue<T> b; public produce(T item) { l.lock(); while(b.full()){ fullCond.wait(l); } b.enq(item); emptyCond.signal(); l.unlock(); } } </pre>
(a) Semáforo	(b) Variáveis Condicionais

```

public Class GlobalCounter<T> {

    private Lock l;
    private T value;

    public T inc(){
        l.lock();
        this.value++;
        l.unlock();
    }
}

```

(c) Travas

Código 2.1: Comparação do uso de travas, semáforos e variáveis condicionais.

gresso independentemente de como as *threads* são escalonadas para execução. No entanto, mesmo que as abordagens bloqueantes possuam apenas garantias de progresso dependentes, como por exemplo serem livres de *starvation* e *deadlock*, não significa que devem ser evitadas. Pelo contrário, caso a preempção⁷ seja suficientemente rara durante as seções críticas, então uma implementação bloqueante e não-bloqueante seriam praticamente indistinguíveis.

A literatura [34] divide os mecanismos não-bloqueantes de acordo com o tipo de garantia de progresso, que pode ser: livre de espera, livre de trava e livre de obstrução. A seguir é feita uma breve descrição de cada garantia.

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.

É importante notar que a listagem está em ordem decrescente de complexidade de implementação e de robustez no que tange garantia de progresso. Note que ainda, um algoritmo *wait-free* é também um algoritmo *lock-free*, e assim por diante, mas o contrário não é sempre verdade. Uma ressalva se faz necessária quanto à sincronização livre de obstrução. Apesar de ser não-bloqueante, implementações livres de obstrução não possuem garantias de progresso independente. Em um cenário onde múltiplas *threads* competem por um objeto não garante-se que alguma *thread* fará progresso. Todavia a adoção de um simples mecanismo de *back-off*, no qual as *threads* conflitantes esperam um intervalo de tempo antes de voltar a competir pelo objeto compartilhado, geralmente é suficiente para sanar tal patologia.

Geralmente, as aplicações utilizando mecanismos não-bloqueantes atingem maior desempenho do que utilizando mecanismos bloqueantes pois a fração serial do código tende a ser menor. Mecanismos não-bloqueantes, por construção, não levam a *deadlocks* ou a problemas de inversão de prioridade, sendo implementados utilizando-se as instruções atômicas do *hardware*, como CAS (*Compare & Swap*). As dificuldades de implementação e correteza dos algoritmos não-bloqueantes foge do tema desse trabalho e portanto não serão tratadas.

A seguir é apresentada a Memória Transacional (TM), um modelo de programação concorrente que utiliza o conceito de transações para abstrair a sincronização dos acessos em regiões compartilhadas de memória.

Capítulo 3

Memória Transacional

O paralelismo ainda é um desafio no contexto geral de programação. No entanto, sistemas de banco de dados há décadas exploram eficientemente o desempenho de *hardware* paralelo. A resolução de várias consultas simultaneamente e em vários processadores são as principais características dos banco de dados para o grande desempenho alcançado. O modelo de programação abstrai o paralelismo permitindo mais liberdade ao programador, uma vez que o mesmo não precisa se preocupar com a sincronização de suas consultas. Em face do amplo sucesso e relativa simplicidade deste modelo conjecturou-se se o mesmo modelo de programação não poderia ser adotado para programação de aplicações de propósito geral.

Memória transacional (TM) é uma abstração conceitual de computação paralela inspirada no modelo de programação de banco de dados [30]. O modelo de TM, assim como o de bando de dados, é centrado no conceito de transação. Uma transação é uma sequência de operações que aparentam ser indivisíveis e cujas modificações ocorreram instantaneamente para um observador externo. O conceito de transação foi introduzido por Eswaran et al. [25] em 1976. Diversas abordagens de programação concorrente inspiraram-se no conceito de transação e operações atômicas para simplificar a codificação e sincronização de processos, como por exemplo o trabalho de Lomet [41]. No modelo de TM as operações em memória compartilhada são mapeadas em transações que possuem as seguintes propriedades: atomicidade, consistência e isolamento. Em banco de dados as transações possuem as propriedades que definem o acrônimo ACID (*Atomicity, Consistency, Isolation and Durability*). O último atributo, Durabilidade, não é considerado no modelo devido ao fato de as operações em memória transacional ocorrerem em memória volátil.

O significado das propriedades de transações em TM é elencado a seguir:

- Atomicidade: uma operação/transação em memória é dita atômica se seus estados intermediários não são observáveis por outros agentes no sistema. Essa propriedade é comumente conhecida como *Tudo ou Nada* (“*all or nothing*”), ou seja, o estado global do programa ou muda atomicamente, nesse caso se diz que a transação realizou um *commit*, ou a transação conflitou e foi cancelada, e nesse caso uma operação de *rollback* é necessária para desfazer as modificações realizadas pela transação.
- Consistência: sua definição exata é dependente de aplicação, mas em geral podemos dizer que a consistência é mantida quando as invariantes do programa são mantidas. Em essência, uma sequência de operações transacionais concorrentes que levaram o programa de um estado S_1 para o estado S_2 deve possuir ao menos uma sequência correspondente de operações sequenciais que também leve o programa do estado S_1 para o estado S_2 .

- Isolamento: define que as operações de uma transação devem acontecer como se a transação estivesse executando em isolamento. Em outras palavras, o resultado da execução de uma transação deve ser o mesmo caso ela tenha sido executada em paralelo ou em total isolamento das demais transações.

Transações em banco de dados (BD) e em memória transacional (TM) diferenciam-se em três aspectos: (i) em BD os dados residem no disco, onde cada acesso leva milhares de ciclos e permitem que milhões de instruções sejam executadas nesse intervalo. Em TM os dados são acessados da memória, levando centenas de ciclos, ou da *cache*, levando alguns ciclos. Ou seja, em BD podemos sobrepor mais computação por acesso do que em TM; (ii) em BD a representação e forma de armazenamento dos dados é independente de como os dados serão acessados, o que permite maior liberdade na implementação dos sistemas. E ainda, o acesso acontece apenas de uma forma, através de consultas. Por outro lado, em TM os dados podem ser acessados diretamente pelo programador e consolidar uma interface única de acesso é complexo e em muitas vezes impossível; (iii) sistemas de TM tem que se adaptar a uma grande variedade de bibliotecas, linguagens e infraestruturas já existentes. A curva de aprendizado será menor se as modificações forem menores nesses aspectos e se não for imposto um mundo fechado como em BD.

O restante deste capítulo discutirá os principais componentes dos sistemas de memória transacional (3.1). Também será abordado questões relativas à semântica e garantias de progressos das transações (3.2). Por fim, as decisões de projeto de sistemas em *software* (3.3) e *hardware* (3.4) serão discutidas através de exemplos e apresentações de propostas da literatura.

3.1 Componentes de um Sistema de TM

Nesta seção serão discutidas as decisões de projeto de um sistema de memória transacional. O sistema de TM é capaz de manter a consistência de suas transações controlando os acessos concorrentes na região compartilhada de memória. Para que o controle de concorrência (vide 3.1.1) seja possível dois mecanismos principais devem ser implementados. O primeiro é o mecanismo que gerencia o trabalho especulativo das transações. Essa atividade é denominada gerenciamento de versão (vide 3.1.2), ou versionamento. O segundo mecanismo deve ser capaz de manter o isolamento das transações, detectando e resolvendo seus conflitos (vide 3.1.3).

3.1.1 Controle de Concorrência

O controle de concorrência pode acontecer de duas formas, ditas *pessimista* e *otimista*. Essa nomenclatura vem da suposição que cada abordagem faz da frequência dos conflitos. A abordagem otimista supõe que os conflitos são pouco frequentes, enquanto que a abordagem pessimista assume que a frequência dos conflitos é maior. Na implementação pessimista o conflito, a sua detecção e resolução ocorrem no momento em que a transação tenta acessar uma região de memória compartilhada. Como esta abordagem supõe que os conflitos são frequentes, então a transação tenta obter posse exclusiva dos dados antes de proceder. Em cenários onde os conflitos são frequentes essa estratégia reduz o desperdício de trabalho, pois caso a verificação e resolução dos conflitos fosse diferida as modificações provavelmente seriam descartadas ao final da transação. No entanto, essa estratégia impede que outras transações acessem esses dados.

Em um sistema com controle de concorrência otimista a detecção e resolução de um conflito geralmente ocorrem de forma tardia. Nessa abordagem otimista, como se assume que os conflitos são pouco frequentes, não há necessidade de obter posse exclusiva dos dados antes de proceder. Em cenários onde os conflitos são pouco frequentes esta estratégia possibilita que mais transações prossigam e acessem os mesmos dados. Como a detecção e resolução são tardios, então duas transações conflitantes podem tentar realizar o *commit* de suas modificações. Nesse caso o sistema deve arbitrar e resolver o conflito. Na prática a detecção e resolução podem acontecer em qualquer instante entre a ocorrência do conflito e tentativa de *commit*. A abordagem otimista propicia maior liberdade de implementação permitindo que mais técnicas de resolução de conflitos possam ser empregadas.

3.1.2 Gerenciamento de Versões

O mecanismo de versionamento é responsável por gerenciar o trabalho especulativo das transações. Mais especificamente, gerenciar as versões corrente e especulativa dos dados. A versão corrente é a versão global do dado disponível a todas as transações, enquanto que a especulativa é a versão modificada pela transação ainda não finalizada. Existem duas estratégias de versionamento, ditas preguiçosa (*lazy*) e ansiosa (*eager*). No versionamento preguiçoso a versão especulativa permanece privada à transação (*redo-log*) até o momento do *commit*, que em caso de sucesso torna-se a nova versão corrente. Essa estratégia é vantajosa quando os conflitos são frequentes, pois quando um conflito ocorre basta descartar a versão especulativa (privada). Mas em cenários onde os conflitos são raros essa estratégia introduz um *overhead* devido à necessidade de transferir, atômica e, a versão especulativa no *redo-log* para a memória compartilhada.

No versionamento ansioso, a situação inversa ocorre, ou seja, o dado é alterado diretamente na memória compartilhada. Por essa razão é necessário utilizar o controle de concorrência pessimista nas escritas, pois a transação necessita de posse exclusiva da versão corrente para poder alterá-la diretamente. Como em caso de conflito a transação deve fazer o *rollback*, então é necessária uma estrutura para armazenar em um *buffer* privado (*undo-log*) a versão corrente. A grande vantagem dessa estratégia é que nenhum dado precisa ser transferido à memória no *commit*, já que se a transação atingiu o trecho de *commit* então nenhum conflito aconteceu. Todavia, em caso de conflito é necessário desfazer as modificações transferindo a versão corrente no *undo-log* para a memória compartilhada.

3.1.3 Detecção e Resolução de Conflitos

Outro componente importante de uma implementação de TM é o mecanismo de detecção de conflito, que pode ser caracterizado por três atributos: sua granularidade, momento no qual a detecção ocorre e a definição do que será considerado um conflito. A granularidade diz respeito a menor unidade na qual o mecanismo consegue detectar um conflito (ex.: nível de *cache* em HTMs e nível de objeto em STMs, *Software Transactional Memory*). O momento da detecção pode ser basicamente três: no momento do primeiro acesso a uma região previamente modificada (*eager detection*), uma ou mais vezes durante a execução (barreiras de validação) ou ainda durante a fase de *commit* (*lazy detection*). No que diz respeito à definição de conflito podemos também dividir em duas categorias: conflitos são considerados entre transações ainda não finalizadas ou são considerados somente entre transações finalizadas e não finalizadas.

Um mecanismo de detecção eficaz deve reduzir ao máximo o desperdício de trabalho

decorrente de sucessivos conflitos. Em uma implementação estática é necessário conhecer ou prever o comportamento de acesso à memória da aplicação para escolha da melhor forma de detecção. Porém na prática esse conhecimento ou predição podem ser impossíveis, visto que o padrão de acesso pode variar de aplicação para aplicação ou de acordo com a carga de trabalho de uma aplicação. As “patologias” de performance podem ser sanadas de duas formas: (i) utilizando um mecanismo híbrido que verifica os conflitos de forma diferente para cada acesso (ex.: detecção ansiosa para escritas e diferida para leituras); (ii) adaptativo no qual o mecanismo observa o comportamento de acesso e os conflitos das transações e determina dinamicamente qual estratégia utilizar.

Uma vez que um conflito ocorreu é responsabilidade do gerenciador de contenção tentar evitar que futuros conflitos ocorram. Diversas políticas de gerenciamento de contenção existem para não apenas resolver como também reduzir a chance de um futuro conflito ocorrer. A escolha da melhor política é guiada, assim como a escolha do mecanismo de detecção, pelas características da aplicação e da carga de trabalho da mesma. Permanece o desafio de conciliar diferentes políticas em uma mesma aplicação, que pode ter diversas fases cada uma com um comportamento e padrão de acesso diferente. A título de exemplo podemos citar duas políticas de resolução de conflito, conhecidas por “passiva” ou “suicida” e “*backoff*”. A primeira, como o nome diz, leva a transação que detectou o conflito a ser cancelada e imediatamente reiniciada. Já na segunda, ao invés de reiniciar imediatamente, espera-se um tempo aleatoriamente gerado antes de reiniciar (número esse gerado em um intervalo que pode crescer linear ou exponencialmente, por exemplo).

3.2 Semântica e Progresso de Transações

O objetivo de um modelo de programação é fornecer uma abstração simples e que facilite o entendimento, escrita e composição de código correto e eficiente. Com uma semântica simples é mais fácil projetar e utilizar ferramentas de programação e depuração. Diversas tentativas de definir uma semântica para transação existem, mas ainda não existe um consenso amplamente adotado. Em um sistema de TM a semântica deve descrever as interações entre as operações que constituem as transações, assim como a interação entre as transações propriamente ditas. Inicialmente foram considerados os conceitos semânticos de transações em banco de dados para construir uma semântica base para transações em sistemas transacionais. Com essa base estabelecida é possível evoluir esses conceitos semânticos e adaptá-los às especificidades das transações em TM.

Considerando apenas as interações entre operações que compõe as transações, como por exemplo leitura e escrita transacionais, podemos dizer que essas operações devem ser linearizáveis. Ou seja, seus efeitos devem ser percebidos como se tivessem ocorrido atômica e imediatamente entre o início e término da operação. No entanto a propriedade de linearizabilidade não define qual o significado das possíveis ordens sequenciais produzida pela intercalação dessas operações. Em essência, essa propriedade não é suficiente para definir uma semântica sequencial. A propriedade de corretude conhecida por serialização melhora essa primeira tentativa ao definir que: se existir uma transição do estado S_1 para S_2 , com as transações T_a e T_b executando concorrentemente, então deve existir a mesma transição com T_a e T_b executando serialmente.

Podemos fortalecer o critério de serialização ao restringirmos que: se T_a completou sua execução antes de T_b , então na ordem serial equivalente T_a deve aparecer antes de T_b . Essa restrição impede que ordenações inválidas ocorram, como por exemplo as operações de transação T_a executam antes das de T_b , mas T_b realiza o *commit* antes de T_a . Nessa

ordem, T_b não observará as modificações feitas por T_a apesar de ter executado depois. A serialização estrita (*strict serializability*) é suficiente para descrever as interações entre operações transacionais, mas não dá significado às interações entre operações transacionais e não-transacionais. Nesse caso podemos retomar o conceito de linearização, mas agora considerando cada transação como um operação indivisível. Dessa maneira as interações entre transações e operações não-transacionais devem ser linearizáveis, ou seja, elas devem ser percebidas atomicamente entre seu início e término.

Nenhuma dessas formalizações considera as interações entre transações finalizadas e canceladas. Guerraoui e Kapalka [29] formalizaram um conceito para lidar com essas interações, denominado *opacidade*. Opacidade é uma serialização estrita onde as transações canceladas também aparecem na ordem serial, assim como as transações finalizadas, exceto que suas modificações não se tornam globalmente visíveis.

Analogamente à semântica, podemos definir as garantias de progresso em um sistema transacional em dois níveis diferentes: (i) nível de TM, se há ou não garantias de transações diferentes executando as mesmas operações transacionais fazerem progresso; (ii) no nível de transações, se há ou não garantias de transações diferentes eventualmente terminarem com sucesso. A única maneira de garantir a liberdade das transações, mesmo quando alguma *thread*/transação sofre preempção, é implementando o sistema de TM com sincronização não-bloqueante. Porém, como discutido em 2.2.3, projetar e implementar algoritmos não-bloqueantes é uma atividade muito complexa. Em teoria uma implementação não-bloqueante pode ser muito mais eficiente que uma implementação bloqueante. Todavia vale ressaltar que ser não-bloqueante não garante que a implementação está livre de *livelock*, já que um gerenciador de contenção eficaz ainda se faz necessário. A questão se TM precisa ser não-bloqueante ou não ainda permanece aberta. Ennals [24] argumenta que TM não precisa ser não-bloqueante pois nos sistemas atuais a preempção dever ser preocupação do sistema de *runtime* ou do escalonador. Como será visto a seguir, a maioria dos sistemas de memória transacional em *software* e *hardware* são baseados em travas.

3.3 Memória Transacional em Software

As primeiras propostas de sistemas transacionais foram em *hardware*, especificamente a implementação de suportes arquiteturais para implementação de estruturas de dados não-bloqueantes [33]. No entanto, mesmo com os *overheads* introduzidos, a implementação em *software* de tais sistemas traz benefícios: (i) grande flexibilidade permitindo implementar uma gama maior de algoritmos sofisticados; (ii) facilidade de integração com sistemas e linguagens já existentes; (iii) implementações em *software* permitem fácil modificação e evolução, e (iv) STMs não estão limitados a recursos de *hardware* como memória *cache*, por exemplo.

3.3.1 Gerenciamento dos Logs e Metadados

Para o controle de concorrência é necessário associar metadados de controle às localizações de memória acessadas pelo programa. Basicamente existem duas abordagens, baseadas em objetos (*object-based*) e baseadas em palavras (*word-based*). Na abordagem baseada em objetos os metadados fazem parte dos objetos do programa e são armazenados e alocados juntamente com os mesmos. Já na abordagem baseada em palavras existem três esquemas: (i) completamente associativo, onde cada palavra é associada a um metadado distinto; (ii) parcialmente associativo, onde uma função *hash* que mapeia endereços a um mesmo

metadado e (iii) único, todos os endereços são associados a um único metadado. Uma representação do armazenamento de metadados é mostrada na Figura 3.1. Na Figura 3.1a é mostrada a abordagem baseada em objetos e nas Figuras 3.1b-d são mostradas os esquemas da abordagem baseada em palavras.

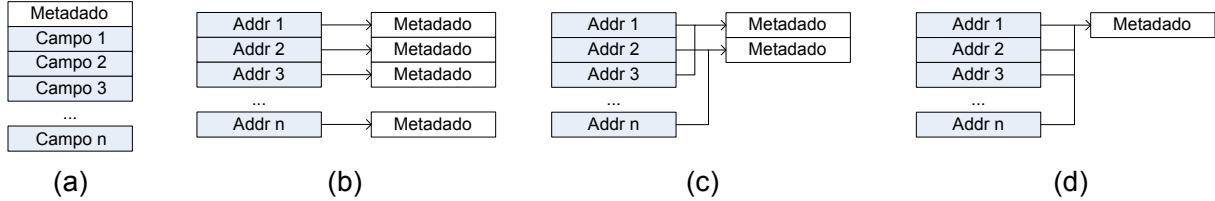


Figura 3.1: (a) Baseada em objetos. Baseado em palavras: (b) completamente associativo, (c) parcialmente associativo e (d) único (retirado de [30]).

A forma como os metadados são associados às posições de memória tem impacto direto no desempenho e consumo de memória da implementação. Por exemplo, pode-se reduzir o consumo de memória ao associar mais localizações a um mesmo metadado, mas também aumenta-se o número de falsos conflitos levando à perda de desempenho. O *overhead* de se mapear uma posição de memória a um metadado pode ser reduzido utilizando uma função *hash* mais simples. E ainda, as implementações que utilizam a abordagem baseada em objetos tornam os conflitos explícitos, uma vez que os metadados estão ligados diretamente aos objetos e pela ausência da indireção introduzida pela função de mapeamento [30].

O conjunto de leitura e escrita são estruturas que também fazem parte do projeto de implementação. Tais estruturas são necessárias para permitir a detecção de conflito, em implementações otimistas, ou liberar as travas durante o *commit*, em implementações pessimistas. Também é necessário armazenar os *logs* com o trabalho especulativo das transações: tanto o *redo-log* (vide 3.1.2), com modificações especulativas privadas que devem ser reescritas no *commit*, quanto o *undo-log*, com a versão corrente que deve ser restaurada em caso de *abort*. Os *logs* e os conjuntos de leitura e escrita podem ser armazenados juntos ou separadamente. Quando juntos o consumo de memória e a pressão sobre as *caches* é menor, mas se forem armazenados separadamente é possível customizar seus itens. Por exemplo, os conjuntos de leitura/escrita guardam as referências aos objetos e os *logs* apenas as referências para os campos modificados.

Os conjuntos de leitura e escrita, assim como os *logs*, são geralmente implementados utilizando estruturas encadeadas que crescem dinamicamente conforme o tamanho da transação. O caminhar nessas estruturas tem impacto direto no desempenho. Por exemplo, no caso de uma implementação com versionamento preguiçoso, o *redo-log* deve ser percorrido para que leituras subsequentes observem as modificações realizadas pela própria transação. Duas estratégias podem ser adotadas para evitar a busca. A primeira utiliza uma função que mapeia os objetos/endereços acessados às entradas no *log*, enquanto a segunda armazena nos metadados referências ao *log*. A validação do conjunto de leitura também pode ser otimizada se for mantido um filtro do conjunto de escrita de cada transação. Por exemplo, o filtro de Bloom [9] pode ser utilizado pois é constituído de um conjunto de *bits* que é setado indicando a presença de uma chave no conjunto. Dessa forma, essa estrutura permite verificar rapidamente se uma entrada no conjunto de escrita de uma transação está presente no conjunto de escrita das demais transações.

3.3.2 Implementações Baseadas em Travas

As implementações baseadas em travas podem ser classificadas em duas categorias: com metadados locais e com metadados globais. As implementações com metadados locais são aquelas que utilizam o armazenamento baseado em objetos (Figura 3.1a) ou baseado em palavras completamente ou parcialmente associativa (Figura 3.1b-c). Já as com metadados globais são implementadas utilizando uma abordagem similar à representada na Figura 3.1d.

Como estudo de caso será discutido duas formas de implementação baseadas em travas com metadados locais: número de versão local vs. global. Quando cada metadado possui um número de versão que é incrementado independentemente dizemos que o número de versão é local. Quando o próximo número de versão é obtido através de um relógio global (*global clock*), dizemos que o número de versão é global. Como já discutido neste capítulo (vide 3.1.2), as decisões de projeto são entre versionamento preguiçoso ou ansioso. Também deve ser decidido se as travas serão adquiridas no momento do primeiro acesso (*encounter-time locking*, ETL) ou durante o *commit* (*commit-time locking*, CTL). Se for utilizado ETL, os conflitos serão detectados entre transações finalizadas e não finalizadas e pode-se optar por qualquer tipo de versionamento. Caso seja utilizado CTL é necessário utilizar versionamento preguiçoso, pois para alterar diretamente a memória compartilhada é necessário antes adquirir a trava. Diferentemente da abordagem ETL, CTL pode detectar conflitos de forma tardia.

As travas e os números de versão são combinados em um objeto denominado trava versionada (*versioned lock*). Geralmente as travas versionadas são implementadas utilizando variáveis inteiras de 64bits onde o *bit* menos significativo indica trava liberada. Nesse caso, o valor dos demais *bits* indica a versão corrente do objeto que esse metadado se refere. Do contrário, trava bloqueada, os demais *bits* indicam qual transação detém o objeto. Uma implementação com esquema de trava ETL e versionamento ansioso será usado para exemplificar o funcionamento das implementações com versão local. O Algoritmo 3.1 mostra as operações de leitura e escrita transacionais, respetivamente, **readTX** e **writeTX**.

```

1: function READTX
2:   if obj ∈ tx.writeSet then
3:     return getValue(obj, field)
4:   else
5:     tx.readSet.add(obj)
6:     v ← STM.getVersion(obj)
7:     tx.readSet.setVersion(obj, v)
8:     return getValue(obj, field)

```

(a) Leitura transacional

```

1: function WRITETX
2:   if obj ∈ tx.writeSet then
3:     setValue(obj, field, value)
4:   else
5:     if STM.tryLock(obj, tx) then
6:       tx.undoLog.add(obj, field)
7:       tx.writeSet.add(obj, value)
8:     else STM.abort(tx)

```

(b) Escrita transacional

Algoritmo 3.1: Leitura e escrita em um STM com ETL e versionamento ansioso.

A operação de leitura (Algoritmo 3.1a) verifica se o objeto lido (*obj*) está no conjunto de escrita da própria transação (*tx*) e em caso afirmativo o valor mais recente do campo (*field*) sendo lido é retornado. Caso o objeto não esteja no conjunto de escrita, então o mesmo é adicionado ao conjunto de escrita juntamente com sua versão retornada pelo STM. Analogamente, a escrita (Algoritmo 3.1b) verifica se objeto já foi modificado pela própria transação e em caso afirmativo atualiza o valor do campo no conjunto de escrita. Caso contrário a transação tenta pegar a trava e em caso de sucesso adiciona o objeto

no conjunto de escrita e o valor corrente do campo no *undo-log*. Caso a transação não consiga pegar a trava ela se cancela.

No *commit* a transação valida suas leituras comparando a versão corrente com a versão no conjunto de leitura. Caso sejam as mesmas a transação libera as travas dos elementos no conjunto de escrita e finaliza com sucesso. Caso contrário desfaz as modificações utilizando os valores salvos no *undo-log* e se cancela. Como podemos observar, esse tipo de implementação não provê opacidade, já que a transação pode continuar executando como uma transação *zumbi* depois de ter lido um valor que foi modificado por outra transação não finalizada. A maneira mais simples de resolver a ausência de opacidade é adicionar a etapa de validação incremental a cada leitura. No entanto a complexidade da validação incremental é da ordem de $O(n)$, onde n é o número de leituras. Existem diversas alternativas de solução para o problema de opacidade em STMs com versão local, como por exemplo utilizar versão global que será discutido a seguir, mas essa discussão foge do tema deste trabalho. As implementações McRT-STM [52] e Bartok-STM [31] são exemplos de implementações baseadas em travas com número de versão local.

1: function READTx	1: function COMMITx
2: if obj $\in$ tx.redoLog then	2: if !STM.tryLock(tx.writeSet) then
3: return tx.redoLog.getValue(obj, field)	3: STM.abort(tx)
4: else	4: ▷ Incrementa relógio global
5: ▷ Checagem dupla para ter opacidade	5: tx.WV $\leftarrow$ STM.fetchAndAddClock()
6: v1 $\leftarrow$ STM.getVersion(obj)	6: ▷ Validação do conjunto de escrita
7: value $\leftarrow$ getValue(obj, field)	7: for all obj <i>in</i> tx.readSet do
8: v2 $\leftarrow$ STM.getVersion(obj)	8: if obj.version > tx.RV then
9: if STM.isLocked(v1)	9: STM.abort(tx)
10: v1 != v2	10: for all item <i>in</i> tx.redoLog do
11: v1 > tx.RV then STM.abort(tx)	11: setValue(obj, item)
12: tx.readSet.add(obj)	12: for all obj <i>in</i> tx.writeSet do
13: return value	13: STM.unlock(obj, tx.WV)
(a) Leitura transacional	(b) Escrita transacional

Algoritmo 3.2: Leitura e *commit* em um STM com ETL e versionamento preguiçoso.

Implementações com versão global garantem opacidade sem realizar validação incremental. A versão global é obtida de um contador global durante a inicialização. Essa versão é conhecida como versão de leitura e é utilizada para validar a transação durante sua execução e na etapa de *commit*. O Algoritmo 3.2 mostra as operações de leitura e *commit* de uma implementação com versão global, travamento no acesso (ETL) e versionamento preguiçoso. A operação de leitura é similar ao do Algoritmo 3.1a, com a diferença que o valor previamente escrito é retornado do *redo-log* ao invés de diretamente da memória compartilhada e da dupla verificação das linhas 6-8 e 10 do Algoritmo 3.2a. Essa verificação garante a opacidade das transações, pois caso a versão antes e depois da leitura sejam as mesmas, quer dizer que o valor lido é consistente. No *commit* (Algoritmo 3.2b) a versão global no relógio é incrementada atomicamente e escrita nos metadados dos objetos. Essa atualização invalida as transações que leram os objetos modificados. Uma possível otimização é evitar a “revalidação” do conjunto de escrita quando $\text{tx.WV} = \text{tx.RV} + 1$, nesse caso é garantido que nenhuma outra transação finalizou entre o início da transação e a alocação da versão de escrita (tx.WV).

3.3.3 Implementações Não-Bloqueantes

Como dito anteriormente, a proposta inicial do modelo de TM era facilitar a implementação de estruturas de dados não-bloqueantes. O grande desafio de implementar um sistema de TM com garantias não-bloqueantes é fazer com que múltiplas modificações sejam observadas atomicamente. E ainda, como uma transação T_1 não pode impedir outra transação T_2 de executar, T_2 deve ver todas as modificações realizadas por T_1 ou não ver nenhuma. A primeira implementação não-bloqueante de TM que não precisa da lista objetos/endereços antes de executar a transação foi apresentada por Herlihy et al. [32]. A implementação proposta utilizava indireção como forma de prover garantias não-bloqueantes. Nesta subseção será discutida apenas a abordagem de implementação baseada em indireção¹.

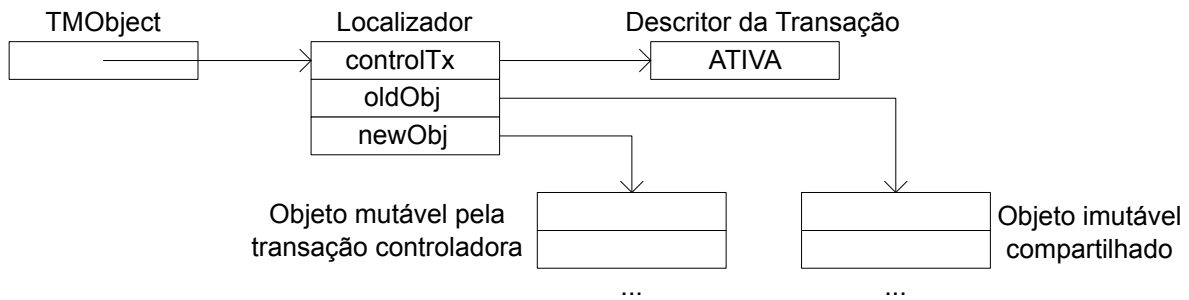


Figura 3.2: Representação dos objetos do tipo `TMOBJET` e `Localizador` (retirado de [30]).

Para utilizar esse sistema, os objetos do programa devem ser encapsulados em objetos especiais denominados `TMOBJETCS`. Esses objetos referenciam outro tipo de objeto imutável denominado localizador (*locator*). A Figura 3.2 ilustra o objeto localizador e seus referenciamentos. Um objeto localizador possui três campos: (i) uma referência ao descritor da transação (`controlTx`) com posse do objeto; (ii) uma referência para o antigo objeto (`oldObj`) e (iii) uma referência para o novo objeto (`newObj`). O valor corrente do objeto pode ser definido unicamente pelo seu respectivo objeto localizador, que por sua vez depende do status da transação em posse do objeto. O status da transação pode ser `ATIVA`, caso a transação ainda não tenha terminado, `ABORTADA`, caso a transação tenha sido cancelada ou experimentou um conflito, ou `FINALIZADA`, caso tenha completado com sucesso suas modificações. Dessa forma o valor corrente do objeto é dado pela referência `oldObj`, caso o valor do status seja `ATIVA` ou `ABORTADA`, ou pela referência `newObj`, caso o valor do status seja `FINALIZADA`.

Essa definição de estado do objeto é o ponto crucial para que a transação possa atualizar múltiplos objetos atomicamente sem utilizar travas. Como todo objeto no conjunto de escrita da transação está encapsulado por um objeto localizador, que por sua vez referencia o status da transação, para que a transação efetive suas modificações basta que a mesma modifique seu status atomicamente de `ATIVA` para `FINALIZADA`. Para compreender a dinâmica envolvendo a checagem do status da transação no caso da escrita, observe a Figura 3.3. Para escrever em um objeto, a transação verifica se ela mesma tem a posse do objeto e nesse caso retorna `newObj`. Caso contrário a posse do objeto é tomada escrevendo o valor `ABORTADA` atomicamente no status da antiga transação e substituindo o antigo localizador. Nesse novo localizador a referência `controlTx` aponta para a nova transação e `newObj` referencia o valor escrito pela mesma. O valor da referência `oldObj` depende do status resultante da antiga transação. Se for `ABORTADA`, então `oldObj` aponta para `oldObj`

¹O autor recomenda o Capítulo 4 do livro de Harris et al. [30] para leitura suplementar de implementações que não são baseadas em indireção.

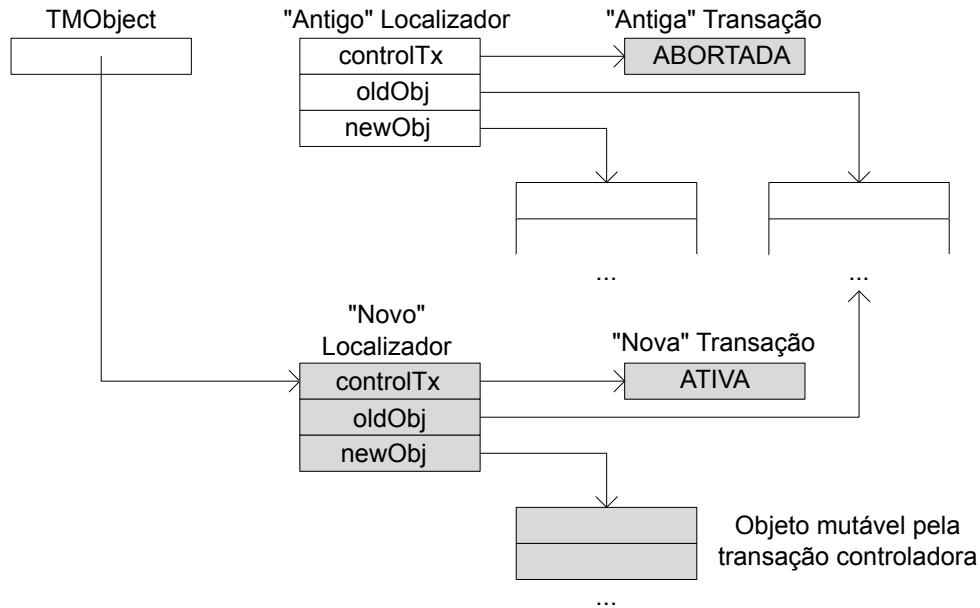


Figura 3.3: Abertura de objeto para escrita (retirado de [30]).

no novo localizador. Caso contrário será **FINALIZADA**, ou seja, a antiga transação efetivou suas modificações. sendo assim `oldObj` aponta para `newObj` do antigo localizador, que é o valor corrente do objeto.

3.4 Memória Transacional em Hardware

Apesar da implementação de um sistema de TM em *software* ser mais atrativo do ponto de vista da flexibilidade, implementá-lo em *hardware* pode ser mais vantajoso. Dentre as principais vantagens podemos citar o baixo *overhead* de inicialização e *rollback* das transações, maior desempenho em executar transações pequenas e médias (até centenas de linhas de *cache* [61]) e por possibilitar o reuso de bibliotecas de terceiros sem recompilação. As principais decisões de projeto de uma implementação de TM em *hardware* são: (i) identificação dos acessos transacionais e não transacionais; (ii) gerenciamento do conjunto de escrita e leitura; (iii) detecção e resolução de conflitos e (iv) preservação dos registradores arquiteturais.

A identificação dos acessos dentro de uma transação pode ser explícita, ou seja, as leituras e escritas transacionais devem utilizar instruções do sistema de TM ou pode ser implícita, na qual todas as operações entre o início e o fim da transação são consideradas transacionais. A principal vantagem da abordagem explícita é que somente os acessos necessários são versionados reduzindo o tamanho das transação. Por outro lado, transações implícitas são interessantes pois habilitam o reuso de bibliotecas sem exigir modificação do código ou recompilação. A proposta inicial de Herlihy e Moss [33] é um exemplo de implementação com identificação explícita. O suporte transacional do processador *Rock*TM [13] da Sun[®] é um exemplo de implementação com identificação implícita.

Existem propostas de duas abordagens de gestão dos conjuntos de leitura e escrita. A primeira abordagem propõe criar memórias *cache* transacionais para armazenar os acessos da transação. No entanto, adicionar *caches* transacionais cria uma nova fonte de onde os dados podem ser fornecidos aumentando muito a complexidade dos processadores modernos. E ainda, como esse nível da hierarquia de memória está no caminho crítico de

execução, não é trivial adicionar a lógica necessária sem impactar negativamente o desempenho do processador. Por essa razão, todas as implementações disponíveis comercialmente, como por exemplo o *System ZTM* [35] da IBM[®], utilizam uma segunda abordagem. Os acessos transacionais podem residir nas *caches* convencionais se forem adicionados *bits* às linhas de *cache* que diferenciem tais acessos de acessos não transacionais. Como dito, utilizar a mesma *cache* para manter os dois tipos de acessos simplifica o projeto da lógica do processador. Todavia, acessos não transacionais podem causar conflitos estruturais (conflitos “falsos”) quando uma linha de *cache* com dado transacional é selecionada para despejo (*eviction*).

Independentemente se a *cache* de dados é compartilhada ou não entre os acessos transacionais e não transacionais, modificações no protocolo de coerência são necessários para detecção dos conflitos. Pelo próprio mecanismo de invalidação do protocolo, que invalida as cópias de uma linha quando a mesma é modificada, a detecção de conflitos ansiosa é a configuração comumente presente nos processadores comerciais. Todavia processadores como o *Blue Gene/QTM* [58] da IBM[®] permitem ser configurados com detecção de conflitos preguiçosa. A resolução de conflito geralmente é a suicida, ou seja, a transação que detectou o conflito é cancelada. Em alguns casos, porém, pode estar à cargo do *hardware*, como no caso do *Blue Gene/QTM* que garante o progresso do programa.

Por fim, deve ser decidido se a preservação do estado arquitetural fica à cargo do *software* (compilador) ou do próprio *hardware*. A maioria das implementações com transações implícitas tem mecanismos de *checkpoint* que salvam o estado da arquitetura antes de iniciar a transação. Em caso de cancelamento o *hardware*, após desfazer as modificações da transação, restaura o estado dos registradores. Uma prática comum de implementação desse mecanismo de *checkpoint* é salvar os registradores arquiteturais em suas respectivas sombras (*shadowing*), que são registradores que fazem parte da microarquitetura e não são expostos ao programador. No entanto, processadores como o *Blue Gene/QTM* não implementam esse mecanismo e portanto exigem que o *software*, geralmente o compilador, faça esse gerenciamento dos registradores.

3.4.1 Memória Transacional no Haswell

Nesta subseção será apresentada a extensão na microarquitetura da 3ª geração de processadores *Sandy BridgeTM*, realizada para compor a 4ª geração de processadores Intel[®] com suporte à sincronização transacional, microarquitetura conhecida por *HaswellTM*. A extensão, anunciada como TSX (*Transactional Synchronization Extension*), é a implementação que permite executar regiões críticas na abstração de transações no modelo de memória transacional. Existem duas interfaces de programação nessa nova microarquitetura apresentadas a seguir: HLE, *Hardware Lock Elision* (3.4.1.1) e RTM, *Restricted Transactional Memory* (3.4.1.2).

3.4.1.1 Omissão de Locks em Hardware (HLE)

A omissão de *locks* em *hardware*, ou HLE, é a interface de programação que permite ao programador adicionar dicas (*hints*) para o módulo de execução omitir as escritas associadas à variável que protege a região crítica, podendo assim tentar executar a mesma com a trava liberada na visão global do sistema - a *thread* que omitiu a escrita vê a variável como ocupada, ou seja, é como se ela estivesse na região crítica de maneira exclusiva. Como a visão global é de “trava liberada”, as operações na região crítica são executadas especulativamente, ou seja, seu efeito é local e não é visível às demais *threads*. As *threads*

que executarem suas transações por completo, sem conflitos, realizam o *commit* e tornam globais suas modificações. Em caso de conflito, as modificações locais são descartadas.

Após o cancelamento, a região crítica/transação é reiniciada mas, ao reiniciar, a escrita na variável de trava é efetuada globalmente. Dessa maneira, as demais *threads* executando a mesma região crítica são canceladas e suas modificações locais também são descartadas. As *threads* reiniciam, porém como a escrita foi realizada globalmente, apenas a *thread* que realizou a escrita procede executando a transação, agora de maneira realmente exclusiva, enquanto as demais aguardam a liberação da trava.

Essa estratégia tenta explorar o paralelismo evitando a serialização das regiões críticas quando possível. Ou seja, se dinamicamente duas ou mais *threads* estão executando a mesma região crítica, mas não possuem conflito de dados, não há razão para privatizar o acesso à região crítica. Outra vantagem do uso de HLE é a portabilidade do código, visto que as modificações necessárias serão realizadas nas bibliotecas de sincronização. Essa característica não exige nem mesmo a recompilação de aplicações, visto que os prefixos são ignorados em máquinas que não suportam HLE.

```

1: function LOCK(l)
2:   while CAS(1,1,XACQUIRE)  $\neq$  0 do
3:     while ATOMIC_LOAD(l)  $\neq$  1 do
4:       THREAD_YIELD( )

```

(a) Operação de *lock*

```

1: function UNLOCK(l)
2:    $\triangleright$  Poderia ser somente  $l \leftarrow 0$  dependendo do modelo de consistência de memória
3:   ATOMIC_STORE(l,0,XRELEASE)

```

(b) Operação de *unlock*

Algoritmo 3.3: Operações de *lock* e *unlock* com omissão de travas em *hardware*.

A interface HLE não introduz novas instruções, apenas prefixos de instruções. Os prefixos adicionados são: XACQUIRE e XRELEASE. O Algoritmo 3.3 mostra as operações de *lock* e *unlock* utilizando o suporte à omissão de travas. O primeiro é associado à instrução de travamento (*lock* – ex.: `lock xchg`), enquanto que o segundo prefixa a instrução de destravamento (*unlock* – ex.: `mov`) [3]. O progresso do sistema é garantido com o uso de HLE visto que quando uma transação é cancelada, a mesma é executada de maneira exclusiva (trava ocupada). Os detalhes de implementação e causas de *abort* do uso de HLE são os mesmos do RTM e serão apresentados na Subseção 3.4.1.2.

3.4.1.2 Memória Transacional Reduzida (RTM)

A implementação de memória transacional reduzida é a resposta da Intel® frente ao recente interesse pela indústria em máquinas com suporte à execução transacional. Na microarquitetura *Haswell*TM, os conjuntos de leitura e escrita ocupam o nível L1 de *cache* privado aos *cores*, cujo tamanho típico é 32KB. Os processadores disponíveis possuem entre 2 e 6 *cores*, cada *core* com até 2 *threads*. Até o momento da escrita desse trabalho não foram revelados os detalhes da implementação. Porém, com base no que já foi publicado, podemos inferir que as principais modificações ocorreram no protocolo de coerência e nos *bits* das linhas de *cache*. O protocolo de coerência da *cache* L1 foi estendido para cancelar a transação que fizer uma requisição de coerência. Provavelmente *bits* de marcação de contexto transacional também foram adicionados. Um mecanismo de *checkpoint* do estado da arquitetura foi adicionado, visto que o programador não precisa realizar essa tarefa, como documentado no manual do desenvolvedor [3].

As instruções da interface RTM são: `xbegin`, `xend` e `xabort`. A primeira inicializa a execução especulativa e salva o estado atual da arquitetura e retorna o status. Além

disso recebe como parâmetro o endereço de *fallback* que será executado após um *abort*. A segunda realiza o *commit* do estado especulativo caso nenhum conflito seja detectado. Caso contrário o fluxo de execução é desviado para o endereço de *fallback*. A instrução **xabort** cancela a transação e define os *bits* [24:31] do registrador EAX com o valor do imediato passado como parâmetro. Os mesmos *bits* são definidos quando uma das situações de *abort* é alcançada. As razões identificáveis pelo valor desses *bits* são: conflito de dados, *overflow* da *cache* L1, nível máximo de aninhamento atingido, instrução de *debug* ou **xabort** executada. Outras situações levam ao *abort* de uma transação, como por exemplo chamada de sistema, mas o *hardware* não as rotula com um valor específico, apenas cancela a transação e cabe ao programador identificá-las.

<pre> 1: function TXSTART 2: while <i>true</i> do 3: if XBEGIN() = STARTED then 4: if ISLOCKED(<i>global_lock</i>) then 5: XABORT() 6: else 7: <i>retries</i> $\leftarrow$ <i>retries</i> + 1 8: $\triangleright$ Caminho de <i>fallback</i> 9: if <i>retries</i> > MAX_RETRIES then 10: LOCK(<i>global_lock</i>) 11: return </pre>	<pre> 1: function TXCOMMIT 2: if <i>retries</i> > MAX_RETRIES then 3: $\triangleright$ O <i>lock</i> foi travado pois não foi possível finalizar a transação em <i>hardware</i> 4: else 5: $\triangleright$ Tentativa de efetivação das modificações da transação 6: XEND() 7: <i>retries</i> $\leftarrow$ 0 </pre>
(a) Iniciar transação	(b) Finalizar transação

Algoritmo 3.4: Um exemplo de implementação das operações TXSTART e TXCOMMIT.

O Algoritmo 3.4 mostra um exemplo de implementação das operações TXSTART e TXCOMMIT utilizando as instruções da TSX. RTM é uma implementação de memória transacional de “melhor esforço” (*best effort*), ou seja, o *hardware* não garante o progresso do sistema. O caminho de *fallback* deve possuir a lógica necessária para garantir o progresso e a consistência do sistema. Outra limitação da implementação é a ausência de instruções de escrita não-transacionais, o que impossibilita a depuração direta do código. Além disso, instruções como *breakpoints* sempre causam o cancelamento das transações.

Capítulo 4

Trabalhos Relacionados

Este capítulo apresentará os trabalhos relacionados mais relevantes no contexto da proposta deste trabalho. A apresentação dos trabalhos foi dividida em duas subseções. Na Seção 4.1 estão os trabalhos que melhoram o desempenho dos HTMs através de otimizações nas transações e no caminho de *fallback*. Os sistemas híbridos serão discutidos na Seção 4.2.

4.1 Melhorando o Desempenho de HTM

O limitado espaço para armazenar o estado especulativo das transações restringe severamente o desempenho das aplicações que utilizam o suporte em *hardware* disponível nos processadores atuais [48]. Nesse sentido, o trabalho de Xiang et al. [60] propõem o particionamento das transações que possuem duas fases, denominadas **planejamento** e **finalização**. Na fase de planejamento as transações realizam operações de leitura e modificações em dados locais. Na fase de finalização as transações realizam as modificações em dados globais. Como as modificações na fase de planejamento são locais e, caso essas transações sejam tolerantes a valores inconsistentes [4], então pode-se mover a fase de planejamento para fora da transação. Dessa forma o tamanho da transação é reduzido, o que aumenta as chances da mesma ser efetivada, uma vez que apenas a fase de finalização é executada na transação em *hardware*. A principal dificuldade é encontrar um particionamento das transações, que pode não existir, limitando assim a aplicabilidade desta técnica.

Diegues et al. [21] melhoram o desempenho das transações em *hardware* utilizando técnicas de aprendizado de máquina, mais especificamente uma combinação do algoritmo UCB (*Upper Confidence Bounds*) [56] e da técnica de gradiente descendente (*Gradient Descent Exploration*) [51], similar ao *Hill Climbing*. O UCB foi utilizado para otimizar o decréscimo de tentativas em cancelamentos por capacidade. Já a exploração por gradiente descendente busca o número de tentativas ótimo para cada transação. O algoritmo **Tuner** [21] se baseia no número de ciclos necessários para efetivar a transação e no *feedback* do *hardware* para decidir o número ótimo de tentativas e quão rapidamente esse número deve decrescer em caso de cancelamento por capacidade. Indiretamente, o algoritmo calcula o grau de transitoriedade do cancelamento, representado pelo valor do decréscimo de tentativas. O **Tuner** mostrou-se bastante eficaz, todavia em cenários onde os cancelamentos por capacidade não são transientes foi observado queda no desempenho.

Uma otimização que permite a sobreposição das execuções do *fallback* irrevocável em *software* e várias transações em *hardware* é apresentada por Calciu et al. [11]. Ao invés de realizar a leitura da trava global no início da transação de forma adiantada, Calciu

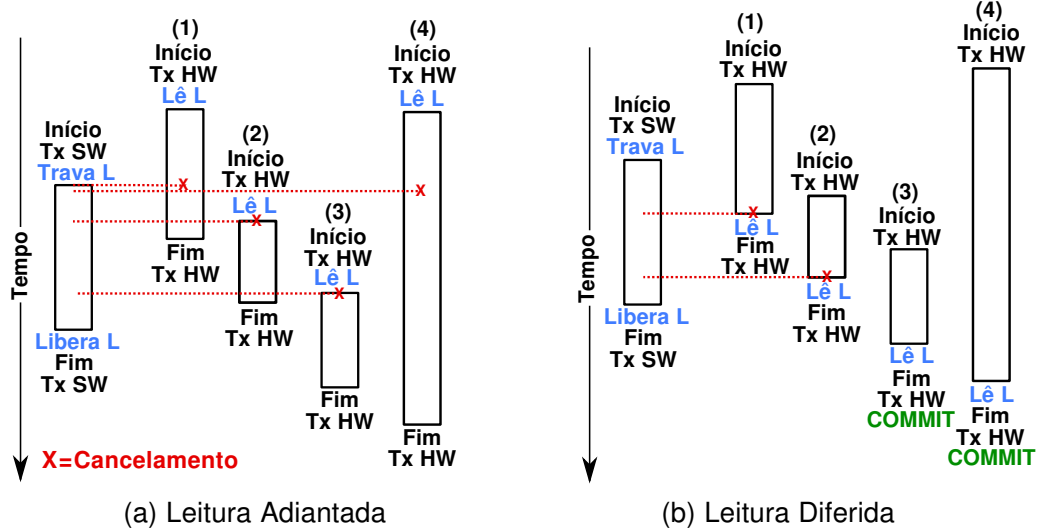


Figura 4.1: Comparação das abordagens de leitura da trava global (fonte [11]).

et al. propõem que a mesma seja diferida até a operação de *commit* da transação em *hardware*. A Figura 4.1 mostra a diferença entre a abordagem clássica, onde a trava é lida no início das transações 4.1(a), e a nova abordagem proposta por Calciu et al. 4.1(b). Essa ideia não é nova e foi baseada no mecanismo de *lazy subscription* proposto por Dalessandro et al. [14] como forma de reduzir o número de cancelamentos espúrios em sua implementação híbrida de memória transacional baseada no NOrec STM [15]. Como é mostrada na Figura 4.1, utilizando essa abordagem é possível que transações não conflitantes em *hardware* (transações 3 e 4) sejam efetivadas mesmo enquanto há uma transação em *software* executando, aumentando assim a exploração de paralelismo da aplicação. A trava global é utilizada no *fallback* para impedir que as transações em *hardware* efetivem suas modificações antes da transação em *software* do *fallback*. Isso é necessário para evitar inconsistências do tipo *lost-update* e violação da propriedade de serialização das transações: a transação em *software* tem ponto de serialização posterior a alguma transação em *hardware*, mas não visualizou as atualizações feitas pela última.

4.2 Sistemas Transacionais Híbridos

A apresentação dos trabalhos que propõem sistemas híbridos foi dividida em duas partes: Em 4.2.1 são discutidos os sistemas que permitem a execução simultânea de transações em modos diferentes (HTM ou STM), denominados HyTMs, e em 4.2.2 os sistemas nos quais apenas transações em um mesmo modo executam concorrentemente, denominados PhTMs. Como nos HyTMs as transações podem executar concorrentemente em modos distintos, então o modo de execução das transações é parte do estado local da mesma. Nos PhTMs o modo é parte do estado global do sistema, uma vez que as transações executam simultaneamente em um mesmo modo.

É importante notar que os dois tipos de sistemas híbridos permitem construir soluções complementares, e não alternativas, para os problemas de limitação de recursos e restrições de execução (ex.: instruções não permitidas) dos HTMs. Como mostrado por Shavit et al. [53], é impossível construir um sistema transacional híbrido sem instrumentação entre as transações em *hardware* e *software*, mesmo garantindo-se apenas a efetivação de transações que executam em isolamento (progresso sequencial). Portanto a escolha de qual sistema

híbrido é melhor dependente fortemente das características de cada aplicação como, por exemplo, existência ou não de fases.

4.2.1 Sistemas Híbridos de Modo Local

Kumar et al. [37] e Damron et al. [16] foram os primeiros a propor sistemas transacionais híbridos. Suas abordagens permitem executar concorrentemente transações em *hardware* e *software* pois os acessos das transações em *hardware* foram instrumentados para permitir a detecção de conflitos com as transações em *software*. Tal instrumentação não apenas torna a execução em *hardware* mais lenta, como também complica o projeto da biblioteca em *software* de TM, pois os metadados das transações em *software* devem estar acessíveis às transações em *hardware*. Novos sistemas híbridos surgiram com o desenvolvimento de STMs mais eficientes e o advento de implementações em *hardware* do modelo transacional. De fato, o STM NOrec de Dalessandro et al. [15], mesmo adicionando um potencial gargalo (o *write-back* é serializado) ao trocar o uso da tabela de travas por detecção de conflitos baseada em validação por valor e ao ordenar os *commits* usando um único valor de sequência (*sequence lock*), permitiu a criação de inúmeros sistemas transacionais híbridos.

Dalessandro et al. [14] foram os primeiros a contrastar diferentes implementações híbridas baseadas no STM NOrec. Os autores mostraram que apesar de uma implementação simples prover opacidade, ela pode fazer com que todas as transações em *hardware* sejam canceladas quando uma única transação em *software* efetiva (mesmo na ausência de conflito de dados). Uma abordagem mais sofisticada usando inscrição tardia do *sequence lock* (*lazy subscription*) também foi analisada, todavia a mesma requer mecanismo que garantam as propriedades de guarda (*safety properties*) como execução em “caixa-de-areia” (*sandboxing*) ou operações de leitura/escrita não-transacionais não presentes na maioria dos HTMs ¹. O HyTM-NOrec de Dalessandro et al. [14] é usualmente adotado como *baseline* em quase todos os trabalhos que propõe novos sistemas híbridos.

Calciu et al. [10] apresentam um sistema híbrido de memória transacional, denominado Invyswell, que utiliza o suporte disponível no processador *Haswell*TM da Intel[®]. Nele existem cinco tipos de transações: (i) SpecSW, transação em *software* baseado no STM InvalSTM [28]; (ii) BFHW, transação em *hardware* com anotações nas operações de leitura e escrita; (iii) LiteHW, transação em *hardware* sem anotações; (iv) IrrevocSW, transação em *software* irrevocável com anotação nas leituras e escritas e (v) SglSW, transação em *software* irrevocável e que executa em isolamento usando uma trava global. A Figura 4.2 mostra o diagrama de transição entre os cinco tipos de transação do Invyswell. A ausência de instrumentação nas transações LiteHW é possível pois elas somente finalizam caso não exista nenhuma transação em *software* executando. As transações de tipo (i)-(iv) utilizam filtros de Bloom [9] para a detecção de conflitos, que ocorre de maneira tardia durante o *commit*. Exceto pelas transações do tipo (v), as demais podem executar paralelamente. Transações de tipo (iv) permitem que transações do tipo (i)-(iii) executem em paralelo, mas ela impede que as mesmas finalizem antes.

A principal vantagem de se utilizar filtros de Bloom é que os filtros introduzem pouco *overhead* de espaço nas transações em *hardware*, visto que podem ser implementados de maneira a estarem alinhados com as linhas da *cache* e minimizar o impacto da detecção de conflitos entres transações dos modos SpecSW e BFHW. As transações do Invyswell, que não são irrevocáveis, requerem uma etapa de invalidação após o *commit*. A invalidação consiste em calcular a interseção entre os filtros de Bloom das transações em execução.

¹O IBM *Power8*TM permite tais operações usando o par de instruções transacionais *suspend* e *resume*.

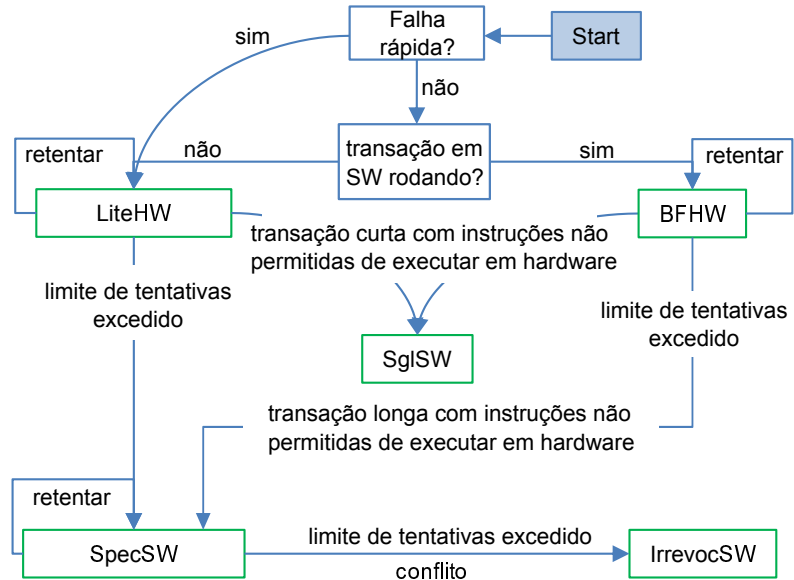


Figura 4.2: Transição entre os 5 tipos de transação no Invswell (fonte [10]).

Em caso de interseção não-vazia, o gerenciador de contenção decide quais transações serão invalidadas e posteriormente canceladas. O progresso do sistema é trivialmente garantido quando as transações executam em modos irrevocáveis. Nos demais modos o progresso é possível pelo uso de uma trava de finalização, que necessita ser verificada apenas ao final das transações em *hardware* (*lazy subscription*). Essa estratégia elimina os cancelamentos espúrios causados apenas pela aquisição da trava de finalização entre transações não conflitantes.

Alexander Matveev e Nir Shavit possuem três versões de um sistema híbrido de TM. Nelas existem basicamente três modos de execução: (i) rápido, transações em *hardware*; (ii) híbrido, transações em *software* aceleradas usando transações em *hardware* e (iii) lento, transações puramente em *software* com as etapas de validação e *write-back* executadas em isolamento. O primeiro trabalho publicado em [43] apresenta um sistema baseado no algoritmo TL2 [20], possuindo versionamento diferido e detecção de conflitos adiantada nas leituras e diferido nas escritas. Nele, cada transação inicia em modo rápido e apenas muda para o modo híbrido quando a mesma não pode ser efetivada, após um número predefinido de tentativas. Caso a transação em modo híbrido falhe a mesma muda para modo lento, que na fase de *commit* executa em isolamento a validação do conjunto de leitura e o *write-back*, causando o *abort* das transações executando.

No modo rápido, cada transação ao iniciar lê o valor do “relógio global” (*gclock*²), garantindo a consistência entre transações no modo rápido e transações nos modos híbrido e lento. Isso ocorre pois transações no modo híbrido ou lento avançam o *gclock* ao finalizar, causando o cancelamento das transações em modo rápido. A cada escrita a transação deve atualizar a versão no metadado referente à escrita, pois as transações em modo híbrido utilizam esses metadados para validar o conjunto de escrita.

O modo híbrido consiste no algoritmo TL2 com modificações apenas na rotina de finalização. Basicamente as modificações propostas em [43] executam as etapas de validação e *write-back* dentro de uma transação em *hardware*. Dessa forma se garante a consistência entre transações em modo rápido e híbrido. Originalmente essas duas etapas são divididas

²O autor assume a familiaridade dos leitores com o algoritmo TL2. Aos não-familiarizados sugere-se a leitura do artigo “Transactional Locking II” [20].

em quatro: (1) pega-se as travas referentes as modificações, (2) valida-se o conjunto de escrita, (3) *write-back* (4) libera-se as travas. Ao executar as etapas (1) e (2) em *hardware* o *overhead* das travas observados no algoritmo TL2 original é eliminado. Matveev et al. em [43] mostram uma variação desse modo híbrido onde apenas a etapa de *write-back* é executada dentro da transação em *hardware*, o que reduz o tamanho da transação e aumenta sua chance de efetivação.

O algoritmo proposto em [43] apresenta melhorias significativas com relação aos trabalhos percursores [37, 16] ao eliminar o acesso aos metadados nas leituras e na fase de *commit* das transação em modo rápido. No entanto ainda é necessário manipular os metadados em toda escrita. Visando eliminar a instrumentação dos acessos em modo rápido, Matveev et al. em [44] propõem um refinamento ao algoritmo anterior substituindo o STM TL2 pelo NOrec [15]. O NOrec é um STM que faz validação por valor e, diferente do TL2, não associa um número de versão a cada dado acessado. O valor do relógio global é utilizado para serializar a validação e *write-back* durante a fase de *commit*: quando ímpar, indica que há uma transação tentando efetivar suas modificações e impede o reinício e efetivação das demais transações. O algoritmo proposto em [44] também melhora o desempenho do anterior ao atrasar a leitura do *gclock* para o final da transação em *hardware*, reduzindo assim as chances de *abort* causados apenas pela atualização do relógio global.

O terceiro e mais recente sistema, denominado RH-NOrec [45], junto com o Invyswell [10] discutido anteriormente, representam o estado da arte em sistemas transacionais híbridos. Nesse sistema existe apenas dois modos, rápido e híbrido. A proposta inova ao construir um modo de execução híbrido com duas transações em *hardware*, denominadas **prefixa** e **sufixa**. A primeira realiza o maior número de leituras possível e finaliza quando uma operação de escrita é atingida ou quando realiza o *commit* da transação em modo híbrido. A **sufixa** começa quando a **prefixa** termina e executa até a finalização. Caso o número máximo de leituras seja atingido, a transação continua a executar leituras em *software*. As leituras realizadas na transação **prefixa** e em sua extensão em *software* são anotadas para verificar se o *gclock* mudou, em caso positivo a transação é cancelada.

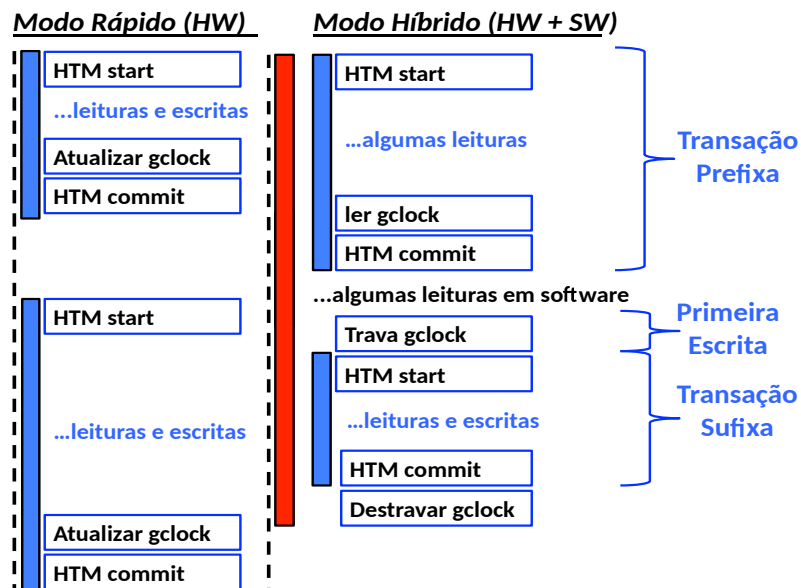


Figura 4.3: Modo rápido e misto no sistema RH-NOrec[45].

A Figura 4.3 mostra um esquema de execução de duas transações, uma no modo rápido e outra no modo híbrido, e detalha as transações **prefixa** e **sufixa**. A transação

sufixa executa as demais operações da transação em modo híbrido até o *commit*. Caso falhe, uma trava global é utilizada para serializar a execução da transação e garantir progresso. Essa trava global é lida no início de toda transação em *hardware* no modo rápido. O uso dessa trava global garante a consistência entre transações em modo rápido e híbrido. Já a verificação do *gclock* a cada leitura garante a consistência entre transações no modo híbrido. As transações **prefixa** e **sufixa**, quando executando em *hardware*, tem consistência garantida com relação a transação em modo rápido pelo próprio mecanismo de detecção de conflitos do *hardware*. A grande inovação dessa abordagem em dividir o modo híbrido em duas transações é permitir a execução de transações mais longas e que não caberiam nos limitados recursos do *hardware*.

4.2.2 Sistemas Híbridos de Modo Global

O único trabalho encontrado na literatura sobre sistemas híbridos de modo global foi publicado em 2007 por Lev et al. [40]. Os autores apresentam o PhTM (*Phased Transactional Memory*), um sistema transacional que propõe executar as transações em modos de acordo com as fases da aplicação. Tais fases são determinadas pela mudança de comportamento da aplicação [54], bem como pelas características de suas transações como, por exemplo, comprimento, nível de contenção e requisitos de capacidade especulativa. Como o objetivo desta dissertação é mostrar a viabilidade de sistemas transacionais baseados em fases apresentando uma nova e eficiente implementação baseada no PhTM (Capítulo 5), a seguir é discutido alguns detalhes da implementação proposta por Lev et al. [40]. Nenhum algoritmo ou pseudocódigo foi publicado deste então, portanto a discussão será baseada somente na descrição do artigo.

A implementação descrita em [40] possui dois modos de execução: modo *hardware puro* (HW), e um modo *software puro* (SW). Transações em HW executam usando o HTM disponível no processador, enquanto que transações em SW executam usando uma biblioteca STM convencional. Dado que a maioria dos HTMs disponíveis apresentam menor *overhead* do que as bibliotecas de STM para gerenciar as transações (iniciar, efetivar e cancelar), e visto que todos os acessos em HW são considerados transacionais (não há barreiras explícitas de leitura e escrita), então os caminhos executados em HW e SW são denominados, respectivamente, *caminho rápido* (*fast-path*) e *caminho lento* (*slow-path*). As transições entre modos e a escolha de qual caminho deve ser executado é guiada por três variáveis globais: **mode**, **deferredCount** e **undDeferredCount**. O sistema precisa ser capaz de acessar essas variáveis atomicamente, todavia os processadores atuais permitem apenas acessos atômicos de até 64 *bits*. Uma maneira de contornar essa limitação é armazená-las como campos (*bit-fields*) de uma variável de 64 *bits* (**modeIndicator**): um *bit* para o modo (0 – HW, 1 – SW), e os demais 63 *bits* divididos entre os dois contadores (**deferredCount** e **undDeferredCount**). A Figura 4.4(a) mostra uma possível configuração da variável **modeIndicator**.

Como mostra a Figura 4.4(b), o PhTM começa no modo HW (bit **mode** não setado). Quando uma transação em *hardware* é cancelada, a mesma incrementa um contador local com o número de tentativas (*retries*). Como reportado em [40], enquanto esse contador for menor que nove (**MAX_RETRIES**), o sistema permanece no modo HW ①. Caso esse limite seja ultrapassado, as transações que falharam repetidamente iniciam uma transição para o modo SW ②: cada transação cancelada incrementa **deferredCount** se, e somente se, o sistema ainda está em HW e seta **mode** se ainda não foi setado. Cada transação em HW, após iniciar, verifica se **modeIndicator** ainda indica que o sistema está em HW, caso

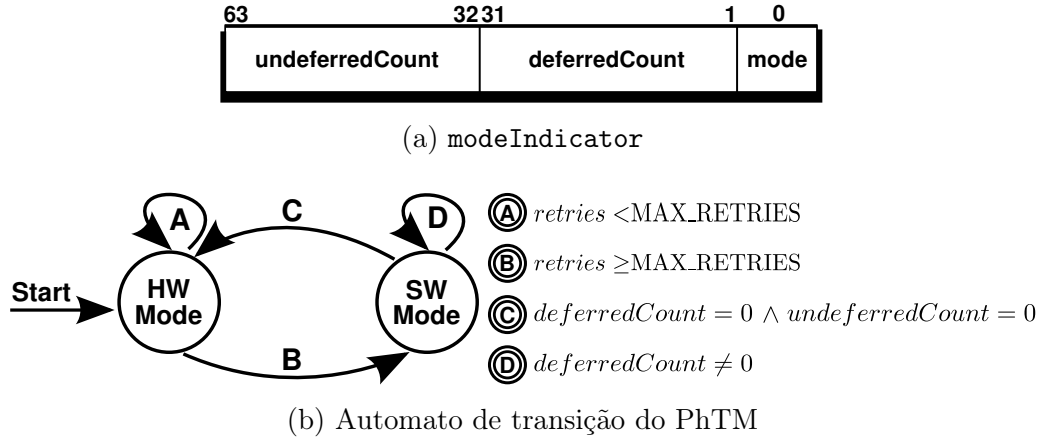


Figura 4.4: PhTM – (a) Configuração dos campos de estado e (b) automato de transição.

negativo a transação é cancelada. Assim uma transição de modo causa o cancelamento de todas as transações em *hardware*, pois a variável `modeIndicator` pertence ao conjunto de leitura de todas as transações em HW. As transações canceladas reiniciam e verificam qual caminho executar lendo o *bit mode* de `modeIndicator`.

O sistema permanece em SW enquanto `deferredCount` for diferente de zero ④. As transações *diferidas*, que incrementaram `deferredCount`, podem iniciar imediatamente e, ao efetivar, devem apenas decrementar `deferredCount`. As demais, *não-diferidas*, antes de iniciar, devem incrementar `undeferredCount` se, e somente se, `mode` ainda estiver setado e `deferredCount` não for zero. Caso contrário, devem esperar o sistema voltar para o modo HW. As transações não-diferidas simplesmente decrementam `undeferredCount` após o *commit*. A transação que decrementa `deferredCount` para zero e observar que nenhuma outra transação esta executando (`undeferredCount` igual a zero) força o sistema a mudar para HW ③.

Não é difícil observar que o PhTM possui duas limitações quanto às suas políticas de transição de modos. Primeiramente, apenas o número de cancelamentos em HW é considerado na mudança para SW, logo transações que não podem completar por limitações do *hardware* – pois excedem a capacidade especulativa, por exemplo – terá que falhar 9 vezes antes de ter a chance de completar em SW. Não obstante, transações em *hardware* podem falhar arbitrariamente (de forma *espúria*), então transações que poderiam ser efetivadas em HW serão forçadas a executar em SW. Já na outra limitação, o sistema volta para HW assim que todas as transações diferidas, potencialmente apenas uma, completam e não existam transações não-diferidas executando. Por consequência, o sistema pode começar a ir e voltar entre os modos em um curto espaço de tempo: uma transação cancelada muda o sistema para SW, inicia e efetiva nesse modo antes de alguma outra transação ter chance de executar, forçando assim uma mudança para HW.

A seguir é apresentado o PhTM*, um implementação eficiente de Memória Transacional em Fases (PhasedTM) que trata as desvantagens do PhTM de Lev et al. [40]. PhTM* evita mudanças desnecessárias para SW ao considerar não somente o número, mas também o tipo dos cancelamentos das transações em *hardware*. O PhTM* também reduz a frequência de mudanças *espúrias* para HW usando um mecanismo simples de *backoff*.

Capítulo 5

Uma Implementação Eficiente de Transações em Fases

Neste capítulo é apresentada a implementação do sistema proposto de transações em fases (PhTM*). Antes, porém, a motivação do trabalho é brevemente discutida (Seção 5.1). Em seguida, são apresentados resultados preliminares que embasam a proposta (Seção 5.2). A proposta, seu algoritmo e detalhes de implementação são apresentados na Seção 5.3.

5.1 Motivação

Recentemente, diversos fabricantes de processadores adicionaram suporte a execução especulativa em seus microprocessadores, por exemplo, a Extensão de Sincronização Transacional (TSX) adicionada a partir do modelo *Haswell*TM da Intel[®] e as instruções transacionais incorporada à arquitetura Power8 da IBM[®] [48]. Apesar de apresentarem menor *overhead* de gerenciamento de transações (iniciar, efetivar e reiniciar), o suporte adicionado na grande maioria dos processadores possui apenas garantias de “melhor esforço” (*best-effort*), o que significa que cabe ao programador fornecer um caminho de código alternativo (*fallback*) para garantir o progresso do sistema. Além disso, como as transações destes HTMs são limitadas pela capacidade especulativa disponível em *hardware*, detalhes da microarquitetura que deveriam ser transparentes ao programador, agora devem fazer parte do processo de implementação das aplicações que utilizam tal suporte.

Sistemas de transações em *software* (STM) foram extensivamente estudados [38] e já se provaram como alternativa viável ao uso de travas para sincronização [23], tanto no aspecto de facilidade de uso quando de desempenho. Todavia, alguns tipos de transações não são bem executados em *software* (ex.: transações curtas), pois estressam comportamentos que evidenciam o alto custo de gestão de transações em *software* e das barreiras de leitura e escrita. Os sistemas transacionais híbridos (HyTM) surgiram como forma de exploração do baixo *overhead* de transações em *hardware* e da capacidade ilimitada de transações em *software*. Para isso, as propostas convencionais (ex.: HyTM-NOrec [14]) instrumentam os acessos das transações em *hardware* para garantir a consistência entre transações de diferentes implementações (*software* e *hardware*) que executam simultaneamente.

Os sistemas híbridos convencionais, apesar de permitirem executar transações em *hardware* e *software* ao mesmo tempo, não apenas tornam as transações em *hardware* mais lentas, devido à instrumentação extra, como também complicam o *design* de seu componente de *software*, pois uma quantidade de metadados suficiente deve ser exposta ao *hardware*, adicionando assim mais *overhead* ao caminho de *software* e dificultando o pro-

jeto da biblioteca STM. Nesse sentido, o trabalho aqui apresentado objetiva avaliar a viabilidade de uma nova implementação de transações em fases (PhasedTM), baseada na implementação proposta por Lev et al. [40]. O sistema proposto (PhTM*), apresentado na Seção 5.3, diferente dos sistemas híbridos convencionais (ex.: HyTM-NOrec), executam as transações em modos de acordo com as fases da aplicação. O argumento é que aplicações que não possuem heterogeneidade de transações (vide Seção 5.2), beneficiam-se mais de transações em fases (vide Seção 6.4) pois não precisam pagar o *overhead* da execução simultânea de diferentes modos.

5.2 Avaliação Preliminar que Embasa a Motivação

A Figura 5.1 mostra o *speedup* obtido para cada aplicação do STAMP executada seguindo a metodologia que será apresentada na Seção 6.3. São mostrados os resultados utilizando memória transacional reduzida (RTM), biblioteca transacional em *software* (NOrec) e trava global (LOCK) executando com 1, 2 e 4 *threads*. O *speedup* foi calculado com relação ao tempo de execução da versão sequencial de cada aplicação. O último grupo de barras (**geomean**) mostra a média geométrica do *speedup* das aplicações obtido com cada uma das abordagens avaliadas. Apesar do processador utilizado permitir a execução com até 8 *threads*, limitou-se o número de *threads* nos experimentos pois a partir de 4 *threads* o desempenho do suporte em *hardware* piora, devido à alta taxa de substituição de linhas (*trashing*) na *cache* L1 compartilhada. Julgou-se mais justo comparar a biblioteca em *software* e a execução especulativa em *hardware* em cenários que ambos não são prejudicados por questões estruturais.

Podemos observar que o RTM proporcionou desempenho comparável e, na maioria dos casos, superior (6 de 7 aplicações) à abordagem LOCK. A redução do *overhead* de sincronização e redução da serialização proporcionados pela execução especulativa são as duas principais razões que justificam a melhora de desempenho. As aplicações **Labyrinth** e **Yada** não escalaram com relação à execução sequencial, e a aplicação **Yada** obteve desempenho inferior à execução com travas globais. Isso foi observado pois essas aplicações possuem transações muito longas, com grande *footprint* de memória, o que causa o cancelamento da maioria das transações por conflito de dados e, principalmente, por *overflow* da capacidade especulativa do *hardware* (*cache* L1 de 32KB e 8 vias).

Comparando o desempenho usando a biblioteca NOrec e o suporte do *Haswell*TM (RTM), podemos notar que o uso de STM proporcionou melhores ganhos em apenas 3 das 7 aplicações (**Genome**, **Labyrinth** e **Yada**). Por outro lado, o suporte em *hardware* propiciou maior desempenho que o STM em mais da metade das aplicações (4 das 7 aplicações), mais significativamente nas aplicações **Kmeans** e **SSCA2**. Esse resultado deve-se à característica das transações dessas aplicações, que são bem mais breves que as transações das demais aplicações. Essa característica permite que o número de cancelamentos por motivo de capacidade e por conflito sejam menores (quanto menor a transação, menor a probabilidade de conflito). Além disso, o baixo *overhead* de inicialização e *commit* das transações em *hardware* comparado ao do STM também tem impacto positivo no desempenho. Entretanto, na média (**geomean**) o desempenho da biblioteca em *software* (NOrec) é apenas ligeiramente superior (< 4%) ao do suporte em *hardware* (RTM).

A Tabela 5.1 mostra a porcentagem sobre o total do número de ciclos gasto e taxa de *commits* de cada uma das transações, quando executadas com RTM, das aplicações do STAMP na execução com 4 *threads*. Nela podemos observar um padrão: presença de uma transação que domina o tempo de execução. Esse padrão também foi observado nas

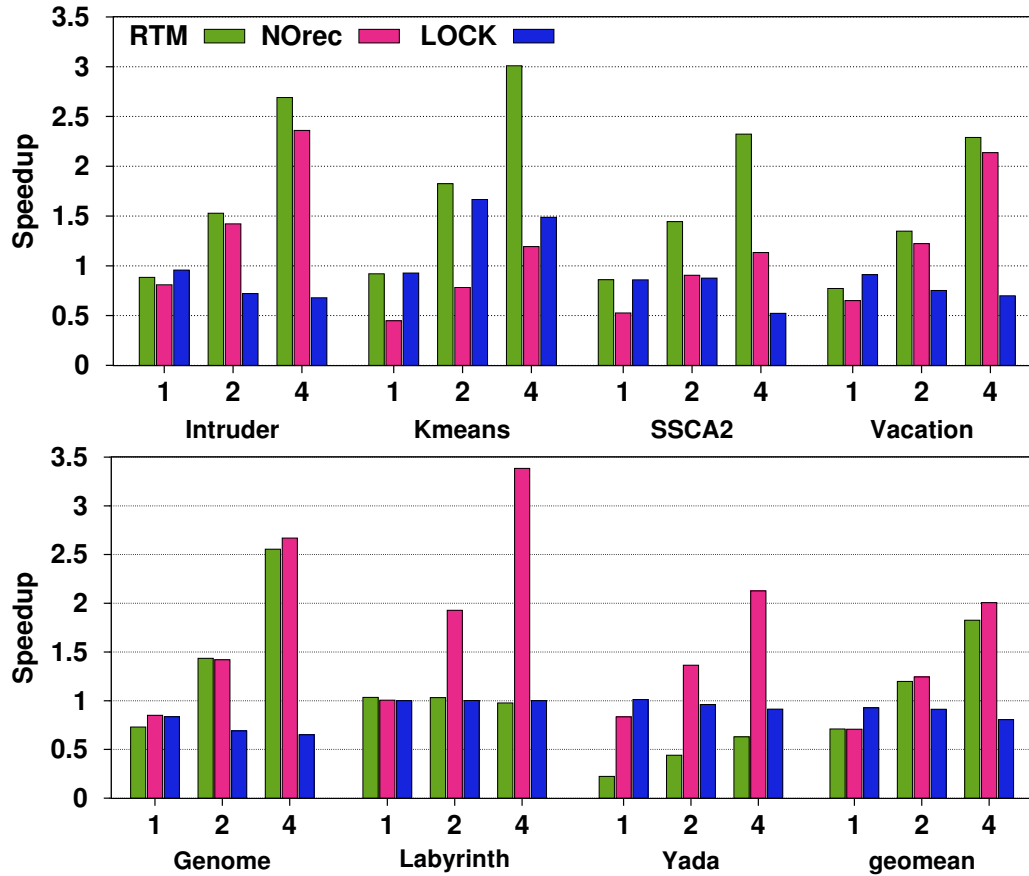


Figura 5.1: Speedup das aplicações STAMP em cada abordagem de paralelização.

execuções com a biblioteca em *software* NOrec. Isoladamente essa constatação apenas permite argumentar que existe uma possibilidade de ganho em desempenho caso a transação dominante fosse dividida em duas ou mais transações menores. Lembrando que transações menores possuem menor probabilidade de conflitar com outras, o que beneficia STMs e HTMs. E ainda, transações menores têm maiores chances de serem executadas nos limitados recursos do *hardware*. Todavia, se analisarmos esse resultado em conjunto com a taxa de *commits* das transações dominantes, podemos concluir que transações dominantes são realmente o gargalo que limita a exploração do paralelismo das aplicações.

As observações extraídas da Tabela 5.1 foram primeiramente observadas pelo autor [17] e, até então, eram inéditas na literatura. Esse resultado implica que aplicações com alta homogeneidade, semelhantes às aplicações STAMP, não serão beneficiadas pela capacidade dos sistemas híbridos convencionais de executar transações em modos diferentes simultaneamente. Por exemplo, nas aplicações *Labyrinth* e *Yada*, não haverá benefício algum em executar suas transações dominantes simultaneamente em *hardware* e *software*, pois o RTM não será capaz de efetivar na grande maioria das vezes. É importante notar que na literatura ([11, 21, 48]) nenhuma implementação baseada apenas em HTM acelerou as aplicações *Labyrinth* e *Yada*. E ainda, nenhum sistema híbrido ([45, 10]) acelerou a aplicação *Yada*.

Tabela 5.1: Porcentagem de ciclos e taxa de finalização por transação (RTM).

Aplicação	Total de Ciclos (%)					Taxa de Commits (%)				
	t1	t2	t3	t4	t5	t1	t2	t3	t4	t5
Genome	78,84	0,09	20,92	0,09	0,07	91,10	93,32	85,76	94,88	99,62
Intruder	5,25	91,59	3,15	–	–	94,32	75,18	95,43	–	–
Kmeans	81,40	18,59	0,00	–	–	94,73	98,96	99,42	–	–
Bayes	2,33	2,38	41,08	19,63	25,78	17,91	6,20	0,16	7,35	5,24
Labyrinth	7,79	92,21	0,00	–	–	41,96	0,00	70,83	–	–
SSCA2	0,00	0,00	100,0	–	–	100,0	87,50	99,91	–	–
Vacation	89,69	7,56	2,75	–	–	26,08	14,56	44,05	–	–
Yada	4,07	2,88	87,56	2,41	3,07	48,71	72,38	1,64	70,30	58,02

5.3 PhTM*

Nesta seção é apresentada o primeiro algoritmo de Memória Transacional em Fases (PhasedTM) projetado com um HTM real (não simulado) no *fast-path*. O algoritmo proposto (PhTM*) usa o *feedback* das transações em *hardware* para guiar a transição entre os modos. Antes de detalhar o algoritmo (Subseção 5.3.1), as características e o funcionamento do sistema serão discutidos em um nível mais alto de abstração. Após a apresentação do algoritmo, alguns detalhes e aspectos da implementação serão abordados em 5.3.2.

Similar ao PhTM de Lev et al. [40], tanto a transição entre os modos como a escolha de em qual caminho a transação deve executar são guiados por uma única variável global, `modeIndicator`. Mas diferente do PhTM, no PhTM* há três modos nos quais as transações podem executar: (a) modo *hardware* (HW); (b) modo *software* (SW); e (c) modo serial (GLOCK). Nos modos HW ou SW, as transações executam usando o suporte de TM disponível no processador ou usando qualquer biblioteca STM, respectivamente. Transações no modo GLOCK executam em posse de uma trava global, em total isolamento e irrevogavelmente. O PhTM* não requer nenhuma implementação específica de HTM ou STM. Pelo contrário, qualquer HTM que possui instruções explícitas para iniciar, efetivar e cancelar transações pode ser usado para implementar o *fast-path*. Pequenas modificações no *backend* do compilador são necessárias para usar o suporte disponível no *Blue Gene/Q*TM, visto que as transações são instrumentadas automaticamente.

Como mostra o autômato de transição do PhTM*, representado pela Figura 5.2, todas as transações do sistema iniciam em HW. O tipo e o número de cancelamentos das transações em *hardware* são usados para decidir qual modo escolher, SW ou GLOCK, caso a transação falhe repetidamente. O PhTM* apenas serializa as transações que foram canceladas repetidas vezes, mais do que nove ¹, por causa, não persistente, de conflitos ou *estouro* de capacidade especulativa ①. Essa política evita o problema de *starvation* – uma ou mais transações nunca efetivam – e permite um rápido retorno ao modo HW. A transição para GLOCK pode falhar pois outra transação mudou o sistema para SW. Como apenas um *thread* pode executar em GLOCK em um dado intervalo de tempo, então as demais *threads* precisam esperar a transação em GLOCK completar ②. No PhTM*, a política de transição para GLOCK objetiva serializar apenas transações curtas, então a quantidade de tempo esperando a transação em modo GLOCK é bem pequena (vide

¹Esse número foi escolhido pois é o mesmo utilizado na proposta de Lev et al. [40].

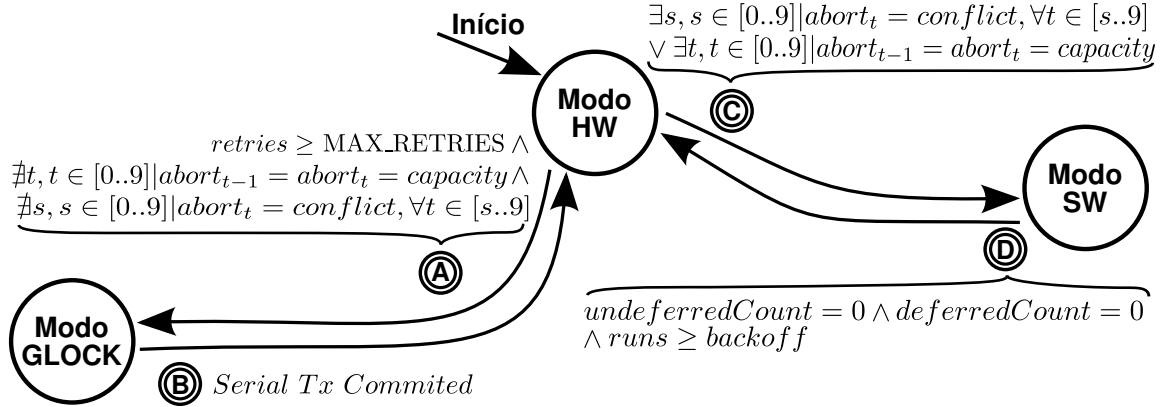


Figura 5.2: Autômato de transição do PhTM*.

Seção 6.5).

Uma transição HW→SW acontece assim que uma das duas seguintes condições é verdadeira: (i) uma transação foi cancelada duas vezes consecutivas por estouro de capacidade; (ii) uma transação falhou repetidas vezes e os cancelamentos foram persistentemente causados por conflitos de dados (C). No primeiro caso, as transações nunca efetivariam em HW pois HTMs possuem limite de capacidade especulativa [27]. A segunda condição é baseada no fato de que cenários de alta contenção são geralmente causados por transações longas: quanto mais longa a transação maior a probabilidade de conflitar com outras transações. A diferença de granularidade na detecção de conflitos, mais grossa em HTMs do que em STMs, e o fato de todos os acessos em *hardware* serem considerados transacionais, juntos fazem com que as transações em HW sejam mais suscetíveis a falsos conflitos do que transações executando em SW.

Uma vez que o sistema está em SW, as transações se comportam como no PhTM de Lev et al. [40]: transações não-diferidas apenas incrementam o contador `undeferrredCount` antes de iniciar se, e somente, o *bit mode* estiver setado e `deferredCount` for diferente de zero. Ao completar, as transações não-diferidas apenas decrementam `undeferrredCount`. Diferente do PhTM, no PhTM* as transações diferidas, que conseguiram incrementar `deferredCount` antes da transição HW→SW, somente decrementam `deferredCount` quando executam um número suficiente de vezes em SW (D). Tal número é determinado por um simples mecanismo de *backoff* que dobra o tempo em SW a cada transição HW→SW. Esse mecanismo considera o fato de que as transações que frequentemente mudam para SW o fazem pois o *hardware* não é capaz de efetivá-las. Apesar de simples, esse mecanismo de *backoff* proporcionou bons resultados (vide Seção 6.4).

5.3.1 Algoritmo

As transações do caminho rápido do PhTM*, como dito anteriormente, podem ser implementadas usando qualquer HTM com instruções que inicializam, efetivam e cancelam transações. As funções do caminho rápido, modos HW e GLOCK, são mostradas no Algoritmo 1. Após o início da transação em HW (linha 2), a mesma verifica se o modo do sistema não mudou desde a última leitura da variável `modeIndicator` (linha 4). Caso `modeIndicator` não seja zero, modo diferente de HW e contadores diferentes de zero, a transação se auto-cancela. Como essa verificação é feita dentro da transação em *hardware*, qualquer alteração em `modeIndicator`, ocasionada por uma transição para SW ou GLOCK, faz com que todas as transações em execução sejam canceladas. As transações

Algoritmo 1 Funções do caminho rápido para os modos HW e GLOCK

```

1: function STARTFASTPATHTx
2: RETRY:
3:    $s \leftarrow \text{STARTHTMTx}()$ 
4:   if TxSTARTED( $s$ ) then
5:     if modeIndicator = 0 then
6:       return FALSE
7:     else
8:       ABORTHTMTx()
9:   ▷ Fallback
10:  if isModeGLOCK() then
11:    WAITGLOCKRELEASE()
12:    goto RETRY
13:  if isModeSW() then
14:    return TRUE
15:  retries++
16:  if isCapacityAbortPersistent then
17:    SWITCHToSLOWPATH()
18:    return TRUE
19:  if retries  $\geq$  MaxRetries then
20:    if isConflictAbortPersistent then
21:      SWITCHToSLOWPATH()
22:      return TRUE
23:    else
24:       $s \leftarrow \text{CHANGEToGLOCKMode}()$ 
25:      if  $s = 0$  then return TRUE
26:      if  $s = 1$  then return FALSE
27:      else WAITGLOCKRELEASE()
28:    goto RETRY
29:  function SWITCHToSLOWPATH
30:    maxStmRuns  $\leftarrow 2 * \text{maxStmRuns} \bmod \text{MaxRuns}$ 
31:    numStmRuns  $\leftarrow 0$ 
32:    CHANGEToSWMode()
33:  function COMMITFASTPATHTx
34:    if isHTMGlockMine then
35:      ATOMICWRITE(modeIndicator, 0)
36:    else COMMITHTMTx()

```

canceladas retomam a execução no caminho de *fallback* (linha 9).

No *fallback* é necessário identificar o que causou o cancelamento da transação, que pode ter sido ocasionada pela interação natural das transações (ex.: conflito de dados), cancelamentos auto-induzidos (ex.: estouro de capacidade especulativa) ou mudança de modo. Caso o modo do sistema tenha mudado para GLOCK (linha 10), então as transações canceladas esperam a transação serial completar (linha 11) e tentam novamente executar em HW (linha 12). Alternativamente, se o sistema mudou para SW (linha 13), então a função `StartFastPathTx` retorna verdadeiro (linha 14) indicando que é necessário mudar de caminho. Caso contrário, o *feedback* do *hardware* é analisado para decidir se a transação cancelada deve ser retentada em HW, ou para qual modo o sistema deve mudar.

O sistema muda para SW assim que alguma transação é cancelada, duas vezes consecutivas, por estouro de capacidade especulativa (linhas 15-16). Essa persistência na causa do cancelamento indica que a transação não efetivará em HW pois excede os limites de especulação do *hardware*. O sistema também muda para SW se, após tentar `MaxRetries` vezes, alguma transação foi cancelada de forma persistente por conflito de dados. O PhTM* classifica o conflito como persistente se uma mesma transação foi cancelada s vezes consecutivas por conflito de dados, sendo $s \leq \text{MaxRetries}$. Essa condição é forte indicativo que o nível de contenção do sistema está alto e, como dito anteriormente, cenários de alta contenção geralmente acontecem na presença de transações longas, que são mais eficientemente executadas em SW. Caso uma transação exceda o limite de tentativas (`MaxRetries`), então a mesma tenta serializar no modo GLOCK. A transação pode não conseguir serializar por duas razões. Primeira, alguma outra transação mudou o sistema para SW (linha 25). Nesse caso a transação simplesmente muda para o caminho lento. E segunda, outra transação está sendo serializada (linha 27), e nesse caso a transação espera a transação serial concluir para reiniciar em HW.

O Algoritmo 2 mostra as funções que realizam ações pré-início e pós-finalização das transações no caminho lento. As transações diferidas podem iniciar imediatamente. No entanto, as transações não-diferidas, antes de iniciar, devem verificar se o modo do sistema não mudou desde a última leitura da variável `modeIndicator` (linhas 4-6). Tais transações mudam para o caminho rápido caso o modo seja HW, ou esperam a mudança para HW caso `deferredCount` seja zero (linhas 9-10). Caso contrário as transações não-diferidas incrementam atomicamente `undeferredCount` enquanto o modo é SW e `deferredCount`

Algoritmo 2 Funções do caminho lento para o modo SW

<pre> 1: function PreSlowPathStart(restarted) 2: if !deferredTx then 3: while !success do 4: indicator ← ATOMICREAD(&modeIndicator) 5: (mode, deferredCount, 0) ← indicator 6: if deferredCount = 0 ∨ mode == HW then 7: if restarted then 8: ATOMICDECUNDEFERREDCOUNT() 9: CHANGETOHWMODE() 10: return TRUE; 11: if restarted then return FALSE 12: expected ← SETMODE(indicator, SW) 13: new ← INCUNDEFERRED(expected) 14: success ← CAS(modeIndicator, 15: expected, new) </pre>	<pre> 16: return FALSE 17: function PostSlowPathCommit 18: if !deferredTx then 19: ATOMICDECUNDEFERREDCOUNT() 20: else 21: numStmRuns ← numStmRuns + 1 22: if numStmRuns < maxStmRuns then 23: return 24: ATOMICDECDEFERREDCOUNT() 25: deferredTx ← FALSE 26: indicator ← ATOMICREAD(modeIndicator) 27: (mode, deferredCount, 28: undeferredCount) ← indicator 29: if deferredCount = 0 then 30: CHANGETOHWMODE() </pre>
---	--

Algoritmo 3 Funções que implementam as transições entre os modos HW, SW e GLOCK

<pre> 1: function ChangeToSWMode 2: do 3: indicator ← ATOMICREAD(modeIndicator) 4: expected ← SETMODE(indicator, HW) 5: new ← INCDEFERRED(expected) 6: success ← CAS(modeIndicator, expected, new) 7: (mode, 0, 0) ← indicator 8: while !success ∧ mode = HW 9: if success then 10: deferredTx ← TRUE 11: do 12: indicator ← ATOMICREAD(modeIndicator) 13: expected ← SETMODE(indicator, HW) 14: new ← SETMODE(indicator, SW) 15: success ← CAS(modeIndicator, expected, new) 16: (mode, 0, 0) ← indicator 17: while !success ∨ mode ≠ SW </pre>	<pre> 18: function ChangeToHWMode 19: do 20: indicator ← ATOMICREAD(modeIndicator) 21: expected ← SETMODE(0, SW) 22: new ← SETMODE(0, HW) 23: success ← CAS(modeIndicator, expected, new) 24: while !success ∧ SELECTMODE(indicator) ≠ HW 25: function ChangeToGlockMode 26: do 27: mode ← ATOMICREADMODE(modeIndicator) 28: if mode = SW then return 0 29: if mode = GLOCK then return -1 30: expected ← SETMODE(0, HW) 31: new ← SETMODE(0, GLOCK) 32: success ← CAS(modeIndicator, expected, new) 33: while !success 34: return 1 </pre>
---	--

é diferente de zero (linhas 12-15). Analogamente à função que inicializa as transações no caminho rápido, a função `PreSlowPathStart` retorna verdadeiro se for necessário mudar de caminho devido à mudança de modo do sistema. Caso contrário retorna falso, indicando que a execução pode continuar pelo caminho lento.

A *flag* `restarted`, parâmetro da função `PreSlowPathStart`, indica se a transação foi reiniciada ou se está começando pela primeira vez, evitando atualizações desnecessárias da variável `modeIndicator`. Ao invés de incrementar e decrementar `undeferredCount` a cada reinício e cancelamento, as transações não-diferidas apenas incrementam `undeferredCount` na primeira vez que iniciam (linha 11), quando o sistema muda para HW (linhas 7-8) ou quando tais transações efetivam. As transações não-diferidas, depois que efetivam, devem simplesmente decrementar o `undeferredCount` (linha 19). Por outro lado, as transações diferidas decrementam `deferredCount` apenas se completaram um número (`maxStmRuns`) suficiente de vezes em SW (linhas 22-24).

As funções que realizam as transições são mostradas no Algoritmo 3. A transição HW→SW acontece da seguinte maneira. Primeiro a transação tenta se tornar uma transação diferida, incrementando `deferredCount` enquanto o modo ainda for HW (linhas 2-7). Caso a transação tenha sucesso, a mesma seta a *flag* privada `deferredTx` para depois saber se deve decrementar `deferredCount` quando efetivar. Em seguida, a transação troca atomicamente os *bits* `mode` de HW para SW se, e somente se, `mode` for HW. A transição HW→SW não requer nenhuma barreira de sincronização; todas as transações em HW

serão canceladas quando o valor de `modeIndicator` mudar. A transição HW→GLOCK acontece quando uma transação consegue escrever, atomicamente, o valor GLOCK no *bits* de modo (linha 32). Em caso de sucesso, a função `ChangeToGLOCKMode` retorna 1 (linha 34), indicando que a transação é *dona* da trava e pode executar em total isolamento. Caso contrário, retorna 0 se o modo for SW (linha 28) e -1 se outra transação estiver em posse da trava (linha 29).

O único ponto de sincronização necessário para a corretude do PhTM* é na transição SW→HW. Antes que essa transição aconteça é necessário que ambos contadores sejam iguais a zero. Se tal condição é satisfeita temos certeza de que todas as transações que iniciarem verão a alteração nos *bits* de modo. Dessa forma garante-se que se uma transação não-diferida tentar iniciar em SW depois da mudança para HW, a mesma mudará de caminho pois não conseguirá incrementar `undereferredCount`. O mesmo vale para a transição HW→SW: se alguma transação tentar iniciar em HW depois da mudança para SW, a mesma se auto-cancelará pois o valor de `modeIndicator` não será zero. Transações iniciadas antes da mudança de modo serão canceladas, no caso da transição HW→SW, ou fazem com que as transações efetivadas esperem pelas transações em execução, no caso da transição SW→HW. No caso da transição HW→GLOCK, como apenas uma transação executa em GLOCK, então a exclusão mútua é trivialmente garantida. Apesar da barreira (linhas 23-24), na prática os experimentos mostraram que o tempo total esperando é sempre menor que 1% do tempo de execução total das aplicações.

5.3.2 Aspectos de implementação

O Código 1 mostra como uma seção crítica deve ser anotada para usar o PhTM*. Este exemplo mostra o quão simples é usar o sistema. Cada seção crítica possui dois caminhos: o caminho rápido, executado quando o sistema está em modo HW ou GLOCK; e o caminho lento, executado somente no modo SW. A *thread* escolhe qual caminho executar ao ler, atomicamente, o modo global do sistema. Se o valor de `modeIndicator` não mudou entre a leitura atômica (linha 2) e o início da transação em HW ou SW, então a execução continua no caminho de acordo com o valor do *bit mode*. Transações em SW são reinicializadas usando um mecanismo de *checkpoint* (linha 9), que salva e restaura o contexto de execução da *thread*. Como dito, o PhTM* pode usar qualquer STM para executar transações no modo SW e não requer nenhuma modificação na biblioteca. `StartSTMTx` e `CommitSTMTx` são as funções da biblioteca STM que inicializam e efetivam as transações em *software*, respectivamente.

Código 1 Seção crítica usando PhTM*

```

1: readMODE:
2: mode ← ATOMICREADMODE(modeIndicator)
3: if mode = HW ∨ mode = GLOCK then
4:   modeChanged ← STARTFASTPATHTx()
5:   if modeChanged then goto readMODE
6:   ▷ fast-path (sem barreiras de leitura/escrita)
7:   COMMITFASTPATHTx()
8: else
9:   restarted ← SOFTWARECHECKPOINT()
10:  modechanged ← PRESLOWPATHSTART(restarted)
11:  if modeChanged then goto readMODE
12:  STARTSTMTx()
13:  ▷ slow-path (com barreiras de leitura/escrita)
14:  COMMITSTMTx()
15:  POSTSLOWPATHCOMMIT()
```

Código 2 Seção crítica usando a libitm

```

1: transactional atomic{
2:   ▷ código escrito pelo programador
3: }
```

Código 3 Código 2 pós-transformação

```

1: run ← _ITM_BEGINTRANSACTION(attr)
2: if run = runUninstrumentedCode then
3:   ▷ fast-path (sem barreiras de leitura/escrita)
4: else
5:   ▷ run = runInstrumentedCode
6:   ▷ slow-path (com barreiras de leitura/escrita)
7:   _ITM_COMMITTRANSACTION( )
```

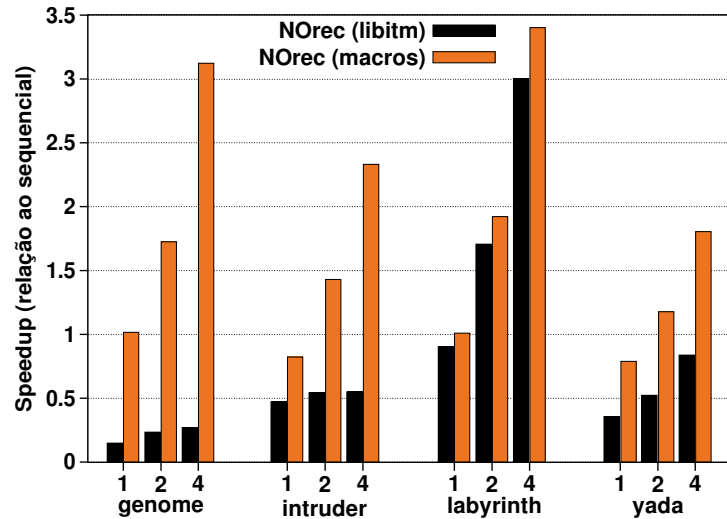


Figura 5.3: Comparando o desempenho da versão automática (libitm) e manualmente anotada (macros) das aplicações STAMP.

Em 2009, a Intel[®] propôs uma ABI (*Applications Binary Interface*) [2], denominada ITM (*Intel Transactional Memory*), com o objetivo de unificar a interface das bibliotecas transacionais, permitindo assim que o compilador faça a anotação dos acessos de maneira automática. Inicialmente, a ideia era implementar o PhTM* seguindo as especificações da ABI para geração da biblioteca dinâmica libitm, forma de distribuição das bibliotecas transacionais usando a ITM. Todavia, como mostra a Figura 5.3, o desempenho da versão anotada automaticamente (libitm) é muito inferior a da versão manualmente anotada (macros). A Figura 5.3 compara o desempenho obtido em 4 aplicações do STAMP [46] utilizando a biblioteca NRec [15], em sua versão automática e manualmente anotada.

Para não mascarar seu desempenho nos resultados mostrados a seguir, o PhTM* foi implementado de maneira que torna necessária a instrumentação manual das aplicações. Entretanto, uma vez sanado o problema de qualidade do código gerado é trivial adaptar o PhTM* para seguir a ABI ITM. A semelhança estrutural entre o Código 1, que usa o PhTM*, e o Código 3, versão automaticamente anotada do Código 2, é prova disso. O Código 2 mostra a sintaxe adotada pelo GCC, desde sua versão 4.7, para anotar transações.

Capítulo 6

Avaliação Experimental da Proposta

Neste capítulo é apresentada a avaliação experimental do sistema proposto. Antes, porém, serão detalhadas as aplicações (Seção 6.1) e os sistemas transacionais (Seção 6.2) utilizados. Em seguida, a configuração do ambiente experimental e a metodologia são detalhados (Seção 6.3). Ao final da avaliação experimental são discutidos alguns desafios de sistemas transacionais em fases.

6.1 Aplicações

A seguir são descritas as aplicações utilizadas na avaliação experimental.

6.1.1 Intset

O **Intset** é uma microaplicação projetada para medir o quão eficiente um mecanismo de sincronização é em diferentes cenários [32]. A microaplicação realiza um teste de *stress* em um conjunto de inteiros ao alternar, randomicamente, entre as operações de inserção, busca e remoção. O **Intset** pode ser configurado para usar quatro diferentes estruturas de dados: lista ligada, árvore rubro-negra, lista de saltos (*skip-list*) e tabela *hash* (*hashtable*). Neste trabalho mostramos os resultados do **Intset** implementado com lista ligada e árvore rubro-negra. Basicamente, além da estrutura de dados, outros quatro parâmetros controlam o teste de *stress*:

- *Número de threads*: informa o número de *threads* que devem executar, em paralelo, as operações do conjunto.
- *Tamanho inicial*: define quantos elementos o conjunto deve ter antes de iniciar o teste. Os elementos são adicionados pela *thread* principal antes de criar as *threads* que executaram as operações do conjunto.
- *Faixa de valores*: faixa dos valores permitidos nas operações do conjunto. As chaves das operações são uniformemente distribuídas nesta faixa.
- *Taxa de atualização*: porcentagem de operações de inserção e remoção sobre o total de operações. O valor 0% realiza o teste executando apenas operações de buscas e, por outro lado, o valor 100% executa apenas operações de inserção e remoção.

Como a microaplicação se vale de uma estrutura de dados para implementar o conjunto de inteiros, a métrica mais adequada para medir a eficácia dos mecanismos de sincronização é a vazão (*throughput*), número de operações por segundo. Em todos os experimentos neste trabalho o tamanho inicial do conjunto é 4096 elementos e o teste de *stress* foi executado por um segundo. Este número de elementos foi escolhido para tornar mais justa a comparação entre as abordagens em *software* e *hardware*, visto que a partir deste valor o número de cancelamentos por *overflow* de capacidade aumenta significativamente.

6.1.2 STAMP

O conjunto de aplicações STAMP [46] (*Stanford Transactional Applications for Multi-Processing*) é o padrão *de facto* na avaliação de sistemas de TM e vem sendo usado na maioria dos trabalhos desde 2008, quando foi disponibilizado pela Universidade de Stanford. Depois que os autores deixaram o projeto, a comunidade de TM assumiu a tarefa de manter as aplicações e corrigir os *bugs* encontrados. Neste trabalho foi usado a versão mais recente do repositório *tm-benchmarks* [1]. As aplicações foram escritas na linguagem C e suas transações foram anotadas, manualmente, com macros que mapeiam as funções de inicialização e efetivação das transações, bem como as barreiras de leitura e escrita. A Tabela 6.1 mostra o domínio e a descrição de cada aplicação. A tabela também mostra os valores dos parâmetros usando neste trabalho, valores estes recomendados para sistemas reais (vide Tabela IV em [46]).

Tabela 6.1: Aplicações do STAMP – parâmetros, domínio da aplicação e descrição.

Aplicação	Parâmetros	Domínio	Descrição
Bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1	aprendizado de máquina	aprendizado de estruturas de redes Bayesianas
Genome	-g16384 -s64 -n16777216	bioinformática	sequenciamento de genes
Intruder	-10 -1128 -n262144 -s1	segurança	detecção de ataques em redes
Kmeans	-m15 -n15 -t0.00001 -irandom-n65536-d32-c16	mineração de dados	algoritmo de clusterização
Labyrinth	-irandom-x512-y512-z7-n512	engenharia	algoritmo de roteamento
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3	científico	conjunto de operações em grafos
Vacation	-n4 -q60 -u90 -r1048576 -t4194304	transações <i>on-line</i>	sistema de reserva de passagens
Yada	-a15 -ittimeu1000000.2	científico	algoritmo para refinamento de malhas pelo critério de Delaunay

Como mostra a Tabela 6.1, o STAMP é constituído de 8 aplicações científicas de diversos domínios. Segundo os autores [46], o STAMP oferece aplicações com *workloads* mais *reais*, e cada aplicação possui diferentes características que diretamente influenciam o desempenho de sistemas transacionais. Apesar dessa aparente diversidade, cada aplicação foi paralelizada usando uma estratégia de programação comum. Primeiro, todos os parâmetros de entrada são processados sequencialmente, produzindo assim uma lista global de trabalho. Em seguida as *threads* são criadas para executar o laço principal da aplicação. Enquanto a lista de trabalho não for vazia, remova um item de trabalho da lista

e o processo. Se novos itens forem produzidos, os adicione à lista de trabalho. Pode-se observar que as aplicações do STAMP foram programadas usando o estilo SPMD (*Single Program Multiple Data*), o que torna as aplicações altamente regulares ¹ e homogêneas.

6.2 Sistemas Transacionais

A seguir são descritos os sistemas transacionais adotados neste trabalho. Os detalhes da implementação do algoritmo proposto foram descritos na Seção 5.3.

6.2.1 RTM

Implementação que usa o suporte em hardware disponível no *Haswell*TM. Cada transação pode tentar efetivar até 16 vezes. Nos experimentos foi observado que valores entre 5 e 16 geram, na média, os mesmos resultados. Caso o limite de 16 tentativas seja ultrapassado ou o cancelamento seja persistente, indicado por uma *flag* do *hardware*, a transação serializa usando uma trava global. O efeito de manada [19] (*lemming effect*) é tratado simplesmente atrasando o reinício de novas transações em HW até que a trava seja liberada. Apesar de simples, essa abordagem foi a que gerou os melhores resultados nos experimentos.

6.2.2 NOrec

O NOrec é um STM que não possui tabela de travas (*Ownership Record*, Orec. NOrec – “NO” + “orec”) proposto por Dalessandro et al. [15] em 2010. A versão utilizada nos experimentos faz parte do código desenvolvido pelo grupo de sincronização da universidade de Rochester e distribuído como parte do pacote RSTM [42]. Ao invés da tabela de travas, o NOrec utiliza uma trava global de sequência (*sequence lock*) para validação do conjunto de escrita. O NOrec detecta conflitos de forma adiantada e versiona os dados de maneira tardia. Durante o *commit*, a transação sendo efetivada impede as demais de efetivarem travando o *sequence lock*, o que simplifica a detecção de conflitos e cria um ponto de serialização trivial para a transação. Como a trava de sequência bloqueada, a transação revalida o conjunto de leitura e se, e somente se, o mesmo for válido realiza o *write-back* do conjunto de escrita.

6.2.3 HyTM-NOrec

Implementação da versão com duas variáveis (*2-location*) do HyTM-NOrec, sistema híbrido de TM proposto por Dalessandro et al. [14]. Na implementação usada as transações do caminho rápido (HW) executam usando TSX e as do caminho lento usando uma versão modificada do NOrec. As transações no HyTM-NOrec podem executar em caminhos diferentes simultaneamente e, para garantir que não ocorram falhas de *lost-update*, as transações em HW incrementam um contador global toda vez que efetivam. As transações em SW devem revalidar o conjunto de escrita toda vez que o valor do contador muda. Ainda assim é necessário evitar que uma transação em HW efetive depois que outra transação em SW tenha revalidado o conjunto de escrita. Tal cenário é evitado cancelando as transações em HW quando alguma transação em SW está fazendo *write-back*. Essa verificação pode acontecer de maneira tardia ou adianta, todavia mostramos os resultados apenas da

¹A aplicação **Bayes** é a única que possui certa irregularidade, mas devido a natureza aleatória do algoritmo e não pela forma como foi paralelizada.

inscrição adianta (*eager subscription*), pois nenhum HTM atual é capaz de implementar inscrição tardia (*lazy subscription*) de forma totalmente segura [18].

6.3 Ambiente Experimental e Metodologia

O trabalho objetiva validar a proposta de execução de transações em fases, contrastando seu desempenho com alguns sistemas transacionais (Seção 6.2) ao executar as microaplicações *Intset* e as aplicações do STAMP (Seção 6.1). Para isso, as aplicações e biblioteca utilizadas (ou implementadas) foram compiladas usando o GCC versão 4.8.3 (Abril/2014), que possui funções intrínsecas para as instruções RTM. Todo o código utilizado neste trabalho foi compilado com nível de otimização 3 (-O3). Objetivando evitar problemas de desempenho induzidos pelo alocador de memória [6], o alocador TCMalloc [26] foi modificado com as modificações sugeridas por Nakaike et al. [48].

Tabela 6.2: Características dos Sistemas.

Processador	CPU Tecnologia Hyperthreading	Intel® Core™ i7-4770 3.4GHz 22nm Sim, 2 <i>threads</i> por <i>core</i> .
Caches	L1	64Kb (32Kb L1d + 32Kb L1i), associatividade por conjunto de 8 vias, linhas de <i>cache</i> de 64 bytes
	L2	256Kb, associatividade por conjunto de 8 vias, não-inclusiva
	L3	8Mb, associatividade por conjunto de 16 vias, inclusiva e compartilhada por todos os <i>cores</i>
Memória	16Gb (8Gb x 2) DDR3 1333MHz	
Discos	SSD	120Gb SATA-3 6Gb/s
	HDD	500Gb SATA-2 7200RPM 3Gb/s, 64Mb cache

As aplicações foram executadas em uma máquina com processador Intel® Core™ i7-4770 com 4 *cores* e 2 *threads* por *core*. A Tabela 6.2 apresenta maiores detalhes do processador, como níveis e tamanhos das *caches* e frequência máxima de operação. Os experimentos foram conduzidos em uma instalação Linux típica, CentOS 6.5 *base server* e kernel versão 3.10. Como mostra a Tabela 6.2, o sistema conta com dois dispositivos de armazenamento não-voláteis, um SSD e um HDD. O sistema operacional, arquivos de configuração e ferramentas do sistema residem no SSD, enquanto que os arquivos dos usuários, binários das aplicações e arquivos de entrada ficam armazenados no HDD.

A seguir é discutida a unidade de monitoramento de eventos utilizada para obter os dados que embasam a motivação (Seção 5.2) e da avaliação experimental da proposta (Seção 6.4). Depois, a metodologia adotada nos experimentos é apresentada (Seção 6.3.2).

6.3.1 Perfilamento de Eventos nos Processadores Intel®

O monitoramento de desempenho foi introduzido na linha de processadores *Pentium*™ com a adição de contadores de desempenho [3]. Esses contadores são registradores específicos de modelo, ou MSR (*Model Specific Registers*) na sigla em inglês, que permitem extrair informações úteis para melhorar a performance do sistema e dos compiladores. Além do monitoramento de performance, os MSRs também facilitam a depuração de

aplicações bem como a configuração da CPU. Os contadores programáveis da PMU (*Performance Monitoring Unit*) se destacam dentre os MSRs disponíveis. A PMU é um conjunto de MSRs dedicados ao monitoramento de eventos relacionados ao desempenho do processador. Existem duas categorias de contadores: (i) os programáveis e (ii) fixos ou não-programáveis. Os contadores fixos são um facilitador no monitoramento de eventos arquiteturais. Eventos arquiteturais são eventos que o fabricante garante que estarão presentes nos novos modelos do processador. Por outro lado, os eventos não-arquiteturais podem, ou não, estarem presentes em seus sucessores. O número de contadores, bem como suas larguras são específicos de modelo. No caso do *Haswell*TM existem 4 contadores programáveis e 3 contadores fixos por *thread*, cada um com 48bits de largura.

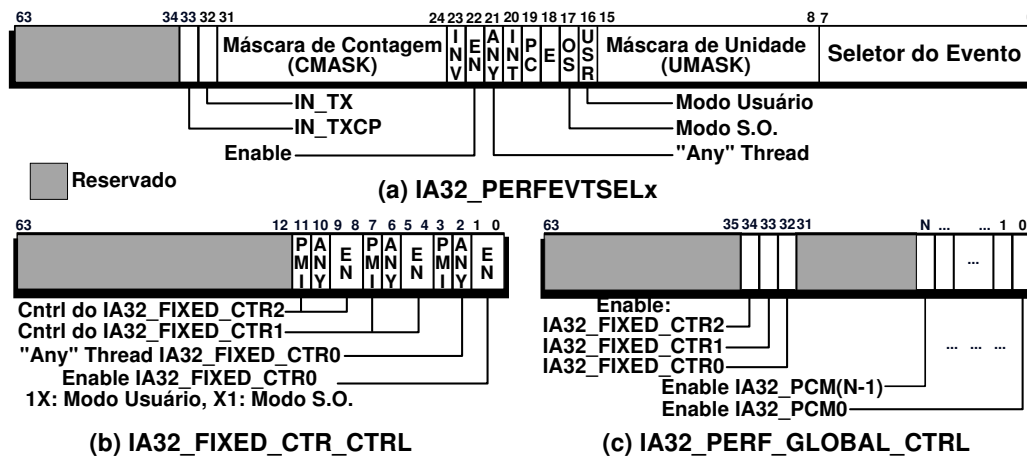


Figura 6.1: Registradores de controle da PMU: (a) controle dos contadores programáveis; (b) controle dos contadores fixos; (c) controle global.

A Figura 6.1 mostra uma representação dos registradores de controle dos contadores programáveis e fixos. A Figura 6.1a mostra os *bits* de controle de um contador programável², sendo que cada contador possui um registrador de controle. Para configurar um contador programável basta escrever o código de seleção e a máscara de unidade³ do evento desejado. O *bit* EN habilita quando “1” e desabilita quando “0” a contagem do evento configurado. O evento pode ser considerado apenas quando executando em modo usuário (*bit* USR *setado*), em modo kernel (*bit* OS *setado*) ou ambos. No *Haswell*TM e nos processadores com suporte ao TSX, os *bits* IN_TX e IN_TXCP são usados para contabilizar o evento apenas quando executando uma transação (IN_TX *setado*) ou apenas os eventos durante o cancelamento da transação (IN_TXCP *setado*). No perfilamento realizado apenas os *bits* USR, Any Thread e Enable foram *setados* para os eventos selecionados (vide Subseção 6.3.2). A Figura 6.1b mostra o registrador de controle dos contadores fixos. Sua configuração é análoga ao dos contadores programáveis, bastando habilitar a contagem no modo desejado (EN: 1x, modo usuário; x1, modo kernel). Como mostrado, todos os contadores fixos compartilham o mesmo registrador de controle, IA32_FIXED_CTR_CTRL. Os contadores somente são realmente habilitados quando seus respectivos *bits* são *setados* no registrador de controle global (Figura 6.1c). Esse registrador permite que múltiplos contadores sejam habilitados/desabilitados escrevendo-se em apenas um registrador.

²Para uma exposição detalhada é recomendado ao leitor o manual do desenvolvedor [3], Vol. 3 Capítulo 18.

³Esses valores estão tabelados no manual do desenvolvedor e são específicos de modelo.

6.3.2 Metodologia

Para comparar o sistema proposto (Seção 5.3) com os sistemas transacionais escolhidos (Seção 6.2) mediu-se o tempo de execução das aplicações (*Intset* e *STAMP*). Para tal, o código das aplicações foi instrumentado com chamadas para função `gettimeofday` da *libc* e, assim, calculou-se o tempo de execução do código paralelo. Além disso, as transações de cada aplicação foram instrumentadas para contabilizar os eventos da PMU. Os eventos selecionados estão brevemente descritos na Tabela 6.3 ⁴.

Tabela 6.3: Eventos da PMU utilizados.

Ciclos e Instruções	<code>INST_RETIRED.ANY</code>	Número de instruções executadas
	<code>CPU_CLK_UNHALTED.THREAD</code>	Número de ciclos transcorridos
Início e Término da Transação	<code>RTM_RETIRED.START</code>	Número de transações iniciadas (RTM)
	<code>RTM_RETIRED.COMMIT</code>	Número de transações finalizadas (RTM)
	<code>HLE_RETIRED.START</code>	Número de transações iniciadas (HLE)
	<code>HLE_RETIRED.COMMIT</code>	Número de transações finalizadas (HLE)
Causa dos Aborts	<code>TX_MEM.ABORT_CONFLICT</code>	Conflito ou <i>overflow</i> do <i>buffer</i> de leitura
	<code>TX_MEM.ABORT_CAPACITY_WRITE</code>	<i>Overflow</i> do <i>buffer</i> de escrita
	<code>XXX_RETIRED.ABORTED_MISC3</code>	Instrução não suportada executada
	<code>XXX_RETIRED.ABORTED_MISC5</code>	Outras razões (ex.: interrupção)

Os dados apresentados neste trabalho foram obtidos em duas etapas de execução. Na primeira, apenas mediu-se o tempo de execução da região paralela. Nessa etapa cada aplicação foi executada 20 vezes, número suficiente para reduzir a variância para menos de 1%. Na segunda etapa, a instrumentação das transações foi habilitada para medir o número de ciclos e instruções de cada transação, bem como os eventos listados na Tabela 6.3. O número de vezes que as transações iniciam e efetivam também foram contabilizadas nas execuções em *software* (NOrec) através dos contadores da própria biblioteca transacional. Nessa segunda etapa cada aplicação também foi executada 20 vezes.

6.4 Avaliação Experimental

Primeiramente é discutido o desempenho de cada sistema transacional usando a microaplicação *Intset* (Seção 6.4.1). Em seguida, a viabilidade do sistema proposto (PhTM*) é mostrada pelos resultados obtidos com as aplicações *STAMP* (Seção 6.4.2).

6.4.1 Intset

A Figura 6.2 mostra como o *throughput* (eixo y) do *Intset*, implementado usando uma árvore rubro-negra, escala com cada um dos sistemas avaliados: nossa implementação usando TSX (RTM), uma biblioteca em software (NOrec), a implementação do sistema proposto (PhTM*) e o sistema híbrido de Dalessandro et al. (HyTM-NOrec). Para cada nível de contenção, 0%, 20% e 60% atualizações, os sistemas foram avaliados executando com 1, 2 e 4 *threads* (eixo x). Claramente, a árvore rubro-negra é uma estrutura de dados *HTM-friendly* pois suas transações são curtas. É importante lembrar que as operações

⁴XXX é RTM ou HLE dependendo de qual suporte transacional deseja-se monitorar.

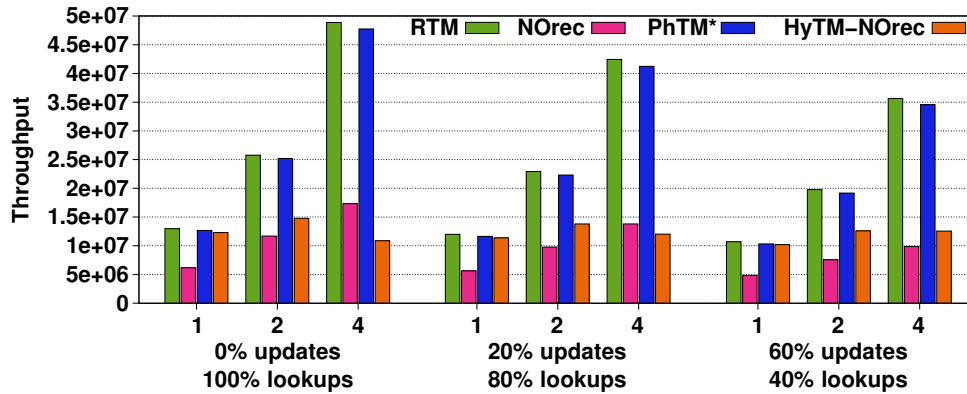


Figura 6.2: Árvore Rubro-Negra.

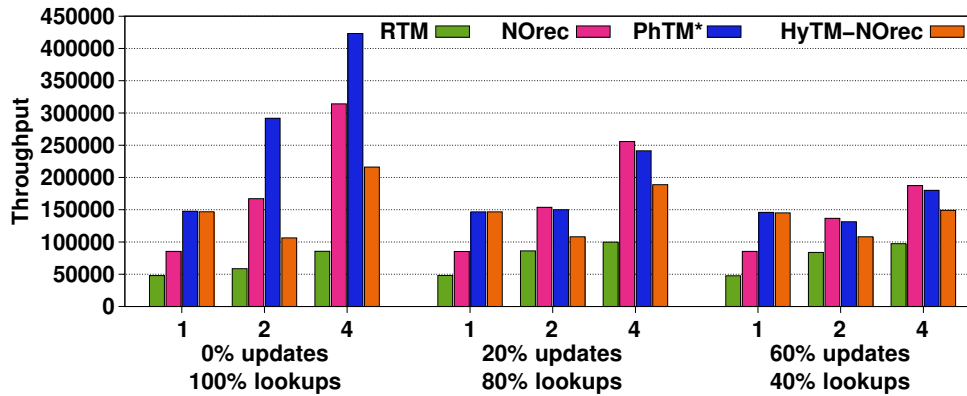


Figura 6.3: Lista Ligada.

em uma árvore rubro negra acessam, no máximo, o logaritmo na base 2 do número de elementos na árvore. Transações curtas reduzem a probabilidade de conflitos tanto em implementações de TM em *software* como em *hardware*, todavia nos STMs as barreiras de leitura/escrita e maior *overhead* para iniciar, efetivar e cancelar transações aumentam a fração serial do código. As curtas transações da árvore rubro-negra causam nenhum cancelamento por *overflow* de capacidade do *hardware*. De fato, as transações do RTM somente sofreram cancelamentos por capacidade se o número de elementos na árvore for maior que 2^l , sendo l o número de elementos que cabem na *cache* L1.

O PhTM* é capaz de escolher o melhor modo de execução (HW) em todos os cenários, como mostra a Figura 6.2. O HyTM-NOrec não desempenha tão bem quanto o PhTM* por três razões principais. Primeira, a cada efetivação em *software* o HyTM-NOrec cancela todas as transações em *hardware* devido a verificação adiantada da trava de sequência do caminho lento. Segunda, cada efetivação em *hardware* no HyTM-NOrec dispara uma revalidação das transações em SW, aumentando assim o tempo em transação e a probabilidade de conflito. E por último, o HyTM-NOrec não possui um modo que serializa as transações. Durante os experimentos foi observado que a serialização de transações curtas leva a um maior desempenho do que uma mudança para SW levaria no caso da árvore rubro-negra.

A Figura 6.3 mostra o desempenho do *Intset* usando lista ligada. Em todos os cenários, o NOrec é melhor que o RTM, comportamento contrário do observado na implementação que usa árvore rubro-negra. Esse comportamento deve-se ao fato de todas operações na lista possuírem uma fase de busca que adiciona os elementos acessados no conjunto de leitura das transações. Essa característica pode ser um problema para o RTM uma vez que suas transação são limitadas pelo tamanho da *cache* L1. Também é importante ob-

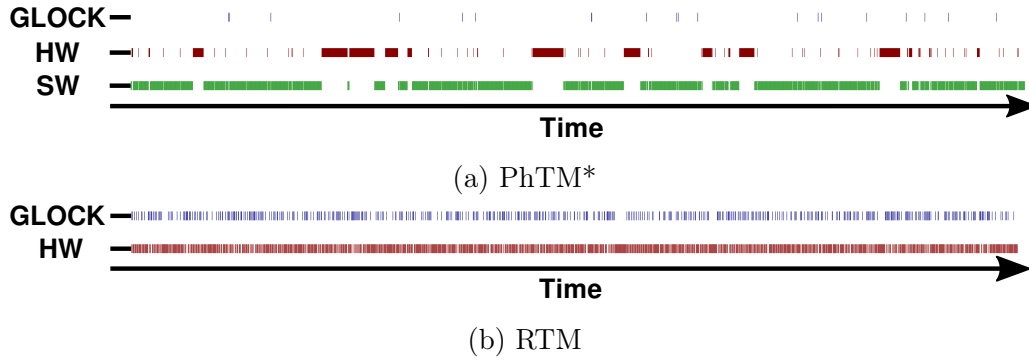


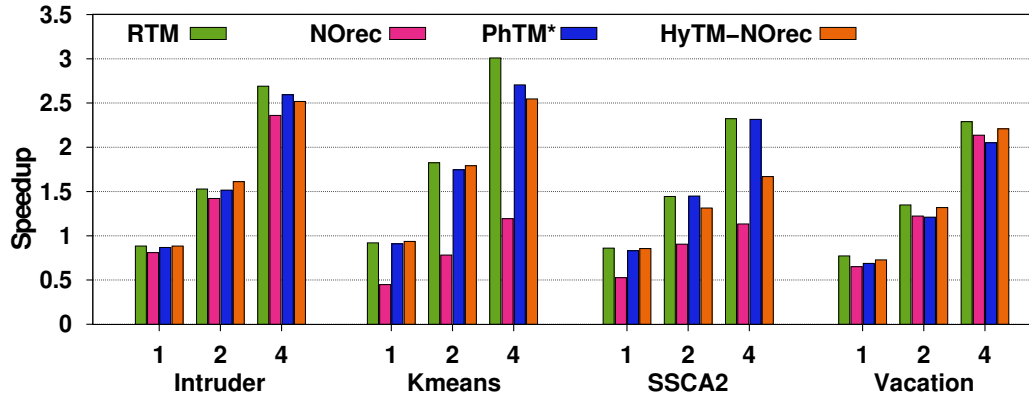
Figura 6.4: Lista Ligada com 100% de buscas e 4 *threads*.

servar que transações em *hardware* podem ser canceladas por capacidade mesmo quando a *cache* não estiver totalmente populada, uma vez que os elementos já acessados podem ser *despejados* da *cache*.

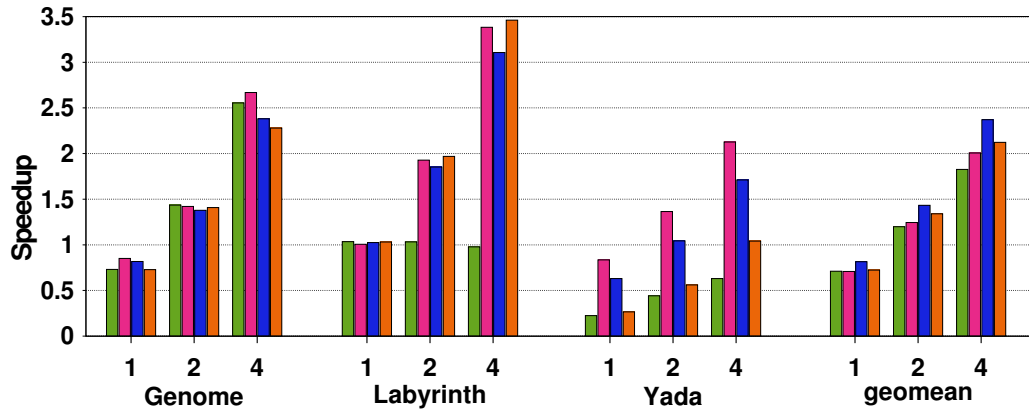
Novamente, o PhTM* é capaz de escolher o melhor modo de execução (SW) considerando os casos com 20% e 60% de atualizações. A pequena diferença, menor que 6%, de *throughput* obtido com PhTM* e o NOrec mostra que a latência para detectar o melhor modo, bem como o *overhead* da mudança são desprezíveis. O HyTM-NOrec não desempenha tão bem quanto o PhTM* porque suas transações no caminho rápido devem falhar 9 vezes antes de mudarem para SW, enquanto que apenas 2 cancelamentos consecutivos por capacidade levam o PhTM* a mudar para o caminho lento.

Um resultado interessante é observado na configuração com 100% de operações de busca e nas execuções com apenas uma *thread*, pois mostra que o PhTM* é melhor que NOrec, RTM e HyTM-NOrec. Nos experimentos foi observado que as transações do RTM sempre falhavam quando mais de 1024 elementos eram acessados, devido a cancelamentos por *overflow* de capacidade. Como no *Intset* as chaves das operações de busca e atualização são uniformemente distribuídas, na média 75% das operações (4096 – 1024 de 4096) não efetivarão em *hardware* e serão serializadas. Esse fato explica por que ambos PhTM* e HyTM-NOrec são melhores que RTM e NOrec em todas as execuções com apenas uma *thread*. Em outras palavras, 75% das transações que antes falhariam em HW agora são executadas em SW e as 25% restantes, transações curtas que executam pior em SW, agora são executadas em HW. O mesmo raciocínio explica por que o PhTM* é melhor que o NOrec com 100% de operações de busca executadas com qualquer número de *threads*.

A Figura 6.4 deixa claro o porquê do *hardware* ajudar o PhTM* a apresentar maior desempenho que o NOrec, mesmo que o RTM não seja capaz de completar a maioria das transações do *Intset* usando lista ligada. A Figura 6.4a mostra a distribuição do tempo em cada um dos modos do PhTM* (eixo y) durante a execução da aplicação (eixo x). Como podemos observar, existem períodos nos quais 100% das transações em *hardware* efetivam. É importante lembrar que neste cenário as transações somente são canceladas por capacidade e apenas dois cancelamentos consecutivos fazem o PhTM* mudar para SW. Tais períodos são suficientemente longos para fazer o PhTM* desempenhar melhor que o RTM e NOrec juntos. Por outro lado, como mostra a Figura 6.4b, no RTM as transações ficam indo e voltando para o modo GLOCK devido aos constantes cancelamentos por capacidades. Esse comportamento mostra porque o RTM possui o menor desempenho. Realmente, mais de 40% do tempo o RTM está serializando as transações, enquanto que menos de 0,4% das transações no PhTM* são serializadas. O HyTM-NOrec não apresenta um bom desempenho nesse cenário pois, nos períodos que o HW é o melhor modo, a



(a) Aplicações do STAMP que o HTM é melhor que o STM.



(b) Aplicações do STAMP que o STM é melhor que o HTM.

Figura 6.5: STAMP – Comparação do *speedup* das aplicações usando cada um dos sistemas.

interferência entre as transações em HW e SW faz com que tanto o número de conflitos do *hardware* quanto o número de revalidações do *software* sejam maiores.

6.4.2 STAMP

Nesta seção a viabilidade do sistema proposto é mostrada contrastando seu desempenho com o desempenho dos sistemas avaliados, como feito anteriormente para o *Intset* (Seção 6.4.1), mas agora com as aplicações do STAMP [46]. A Figura 6.5 mostra o *speedup* (eixo y), em relação ao tempo sequencial, de 7 aplicações do STAMP executadas com 1, 2 e 4 *threads* (eixo x). O último grupo de barras (*geomean*) é a média geométrica do *speedup* de todas as aplicações. A aplicação *Bayes* não foi usada por causa de sua conhecida alta variação no tempo de execução [50]. Similar ao observado na microaplicação *Intset*, existem dois grupos de aplicações do STAMP: um no qual o RTM apresenta o melhor desempenho (*Intruder*, *Kmeans*, *SSCA2* e *Vacation*) e outro onde o NOrec é o melhor (*Genome*, *Labyrinth* e *Yada*).

As aplicações que o RTM é mais eficaz (Figura 6.5a) apresentam transações curtas com quase nenhum cancelamento por capacidade e baixas taxas de conflitos. Em particular, ambos *Kmeans* e *SSCA2* possuem número de cancelamento por capacidade próximo de zero, uma vez que apenas poucos endereços de memória são acessados em cada transação. De fato, menos de 7% (*Kmeans*) e 1% (*SSCA2*) de suas transações são canceladas, respectivamente. Além disso, o tempo entre as transações do *Kmeans* e *SSCA2* é suficientemente longo para minimizar a probabilidade de conflitos. Devido ao *overhead* das barreiras de

leitura/escrita e o alto custo em gerenciar as transações em *software*, o NOrec apresenta um desempenho ligeiramente inferior ao RTM nas aplicações *Intruder* e *Vacation*. Em ambas aplicações, o custo de gestão das transações é maior porque o tempo entre transações é praticamente zero, o que aumenta a probabilidade de conflito e não ajuda a mitigar o custo de gestão das transações (iniciar, efetivar e reiniciar).

O NOrec é mais eficaz nas aplicações *Genome*, *Labyrinth* e *Yada* (Figura 6.5b) por dos principais motivos. Primeiro, algumas instruções não são permitidas de executar nas transações em *hardware* (ex.: chamadas de sistema) e sempre causam o cancelamento da transação que tenta executá-las. De fato, a porcentagem de transações canceladas por causa de tais instruções é maior que 14%, 22% e 33% nas aplicações *Yada*, *Genome* e *Labyrinth*, respectivamente. Tal restrição de instruções, obviamente, não é imposta quando as transações executam em *software*. Segundo e mais importante, como todos os acessos em uma transação em *hardware* são considerados transacionais (não há barreiras explícitas de leitura/escrita), as transações no RTM são mais suscetíveis a cancelamentos por capacidade e conflito que o NOrec, no qual os acessos estão manualmente instrumentados. Realmente, mais de 70% das transações do *Yada* são canceladas por conflito de dados quando executadas com RTM, valor bem maior que os 18% observados quando a mesma aplicação é executada com NOrec. Uma diferença similar foi observada na aplicação *Labyrinth*, na qual menos de 8% das transações são canceladas por conflito com NOrec, mas com RTM mais de 27% das transações conflitam. No caso particular da aplicação *Yada*, Kobayashi et al. [36] também notaram que esse versionamento em excesso faz com que seja impossível para os HTMs de melhor esforço atuais acelerarem a *Yada* sem modificar como a aplicação foi paralelizada.

Como a Figura 6.5 mostra, o PhTM* e o HyTM-NOrec aceleraram quase que igualmente 4 de 7 aplicações do STAMP (*Genome*, *Intruder*, *Kmeans*, e *Vacation*). Todavia, o PhTM* apresentou um desempenho ligeiramente melhor (até $\approx 5\%$) em 3 dessas 4 aplicações, especificamente o *Genome*, *Intruder*, e *Kmeans*. Nesses casos, o espaço para melhorias é pequeno devido a homogeneidade das aplicações STAMP ou o desempenho dos sistemas em *hardware* e *software* são próximos. Nesse caso, não importa se as transações executam em diferentes modos simultaneamente (HyTM-NOrec) ou se executam em fases (PhTM*). Ou ainda, se um dos modos é melhor que o outro, então executar transações em um modo diferente do melhor não melhora o desempenho. Essas observações explicam a pequena diferença de desempenho entre os sistemas híbridos e os sistemas *puros* observada nessas aplicações.

Uma diferença de desempenho mais expressiva foi observada em 3 aplicações: *Labyrinth*, *SSCA2* e *Yada*. Realmente, o PhTM* é mais de 30% e 50% melhor que o HyTM-NOrec nas aplicações *Yada* e *SSCA2*, respectivamente. Por outro lado, o HyTM-NOrec é cerca de 13% melhor que o PhTM* na aplicação *Labyrinth*. Em particular, a aceleração do PhTM* é bem mais próxima da alcançada pelo RTM na aplicação *SSCA2* principalmente porque essa aplicação tem suas transações canceladas na grande maioria das vezes por causas espúrias ($\approx 91\%$), então a política do PhTM* sempre escolhe serializar tais transações, assim como o RTM, ao invés de mudar para SW. Na aplicação *Yada*, a diferença mais pronunciada entre o desempenho do PhTM* e o NOrec deve-se à dificuldade em obter informações para decidir se as transações devem continuar executando em SW, ou voltar para HW. Como o NOrec é cerca de 2x melhor que o RTM na aplicação *Yada*, então toda vez que o PhTM* volta para HW, devido ao fim do período de *backoff*, o impacto negativo no tempo de execução é maior. A aplicação *Labyrinth* foi a única na qual o HyTM-NOrec foi melhor que o NOrec ($< 2\%$). De fato, essa foi a única aplicação do

Tabela 6.4: Porcentagem do tempo em cada modo para as aplicações do STAMP.

Sistema	RTM		PhTM*		
Aplicação/Modo	HW	GLOCK	HW	SW	GLOCK
Genome	84.19	15.81	59.42	40.53	0.05
Intruder	93.76	06.24	63.68	36.05	0.27
Kmeans	99.75	00.25	79.60	20.40	0.00
Labyrinth	00.00	100.0	02.22	97.68	0.10
SSCA2	70.00	30.00	99.98	00.02	0.00
Vacation	92.14	07.86	18.80	81.07	0.13
Yada	50.39	49.61	07.80	92.02	0.18
Média	70.03	29.97	47.21	52.54	0.11

STAMP a se beneficiar da capacidade do HyTM-Norec de executar transações em modos diferentes ao mesmo tempo. Mas, como mostra a Figura 6.5, o PhTM* é o melhor, na média (geomean), com ganho de cerca de 12% em relação ao HyTM-NOrec.

6.5 Desafios e Limitações de Transações em Fases

Os resultados anteriores (Seção 6.4) demonstram que o PhTM* é uma alternativa viável e competitiva aos sistemas híbridos convencionais e que é possível implementar transações em fases sem pagar os altos custos de barreiras de sincronização. De fato, como mostra a Tabela 6.4, o tempo que as *threads* esperam nas duas barreiras do PhTM* (SW→HW e GLOCK→HW) representa, na média, menos de 0.2% do tempo de execução total das aplicações STAMP. A Tabela 6.4 mostra, para cada aplicação STAMP executada com 4 *threads*, a porcentagem do tempo despendido em cada modo do RTM e PhTM* (HW, SW e GLOCK). A porcentagem do tempo no modo serial foi reduzido significativamente em todas aplicações, de 29.97% (RTM) para 0.11% (PhTM*). Em particular, o PhTM* permitiu que as aplicações *Labyrinth* e *Yada* executassem a maior parte do tempo em SW, ao invés de serializarem com a trava global.

Como apontado por Lev et al. [40], existem basicamente três desafios na implementação de transações em fases: (i) identificar os modos mais eficientes para cada fase do programa; (ii) implementar transições rápidas e corretas entre os modos; e (iii) decidir quando mudar de modo. O PhTM* ilustra uma das formas de tratar os dois primeiros desafios de forma eficaz e eficiente. O problema para decidir quando mudar de modo mostrou-se o mais desafiador dos três. Realmente, as políticas de transição do PhTM* são mais eficazes em detectar quando permanecer em HW ao decidir serializar as transações curtas. A parte difícil é decidir quando voltar para HW uma vez que o sistema está no modo SW. A política de transição para SW, apesar de ser capaz de detectar o início das fases de capacidade e alta contenção, não é capaz de detectar quando tais fases terminam: as transações executam em SW até o fim do período de *backoff*. Essa capacidade somente será possível uma vez que seja possível determinar a diferença entre os conjuntos especulativos usados nas transações em HW e SW. É importante notar que não basta contar o número de elementos nos conjuntos das transações em SW, pois existe diferença de instrumentação nos caminhos executados em HW e SW. Como todos os acessos em HW são monitorados para detectar conflitos (transações implícitas) e transações em HW possuem limite de

capacidade, então esse tipo de informação é essencial para detectar quando uma fase SW, induzida pela diferença de instrumentação e limites de capacidade, termina.

O argumento mais forte contrário ao sistemas de transações em fases é o suposto impacto negativo no desempenho em cenário no qual uma única transação requisita executar no modo mais lento (ex.: *software*). Os resultados apresentados neste trabalho (Seção 6.4) mostram que nem a microaplicação **Intset** quanto as aplicações do STAMP exibiram cenários onde uma única transação longa requisitou que o PhTM* mudasse para SW. Isso se deve a forma como as aplicações foram construídas (SPMD), que não favorece a heterogeneidade das transações. Portanto, a maioria das aplicações possuem transações com requisitos de capacidades bem definidos e um sistema baseado em fases como o PhTM* oferece uma solução competitiva quando comparada com sistemas híbridos convencionais. Ademais, no único caso que foi observado mudança de capacidade (**Intset** com 100% de buscas), o PhTM* foi capaz de detectar tal mudança e apresentou desempenho superior as abordagens HTM, STM e HyTM avaliadas.

Capítulo 7

Conclusão

Este trabalho apresentou uma breve discussão do cenário atual da programação concorrente para introduzir a Memória Transacional (TM), um dos modelos de programação mais estudados nos últimos 10 anos. Mais especificamente, foram analisadas as principais decisões de projeto e desafios na implementação de sistemas transacionais. As limitações dos sistemas em *software* e o carácter de melhor esforço da maioria dos suportes em *hardware* motivaram a proposta de soluções híbridas (HyTM). Os HyTMs objetivam explorar o melhor de cada implementação ao permitir a execução de transações em *hardware* e *software*, simultaneamente. Entretanto, nos HyTMs o desempenho das transações em *hardware* é reduzido pela instrumentação adicional. E também, a implementação da componente em *software* torna-se mais complexa.

Como alternativa aos sistemas híbridos convencionais, este trabalho apresenta a primeira implementação eficiente de transações em fases que utiliza suporte transacional em *hardware* disponível em um processador *real* (não-emulado). Ao executar as transações em modos de acordo com as fases do programa, os sistemas em fases eliminam a instrumentação das transações em *hardware* e permitem utilizar as bibliotecas STM sem nenhuma modificação. O PhTM* é o primeiro sistema de transações em fases a utilizar o *feedback* das transações em *hardware* para implementar políticas eficazes de mudança de modo. Ao propor a serialização de transações curtas e considerar a persistência dos cancelamentos por capacidade e conflito de dados, o PhTM* foi capaz de escolher o modo mais adequado (HW ou SW) tanto em uma microaplicação (*Intset*) quanto em todas as aplicações do STAMP avaliadas. Na média, o PhTM* obteve desempenho cerca de 12% maior que o HyTM-NOrec, sistema híbrido considerado *baseline* na avaliação de novos HyTMs.

7.1 Trabalhos Futuros

Pelo carácter pioneiro da implementação e apesar dos bons resultados obtidos, ainda assim pode-se destacar dois pontos centrais que certamente podem ser melhorados. Primeiro, o maior desafio encontrado foi decidir quando voltar para o modo HW. Pela dificuldade em determinar o fim de uma fase de capacidade enquanto no modo SW, atualmente o PhTM* utiliza uma abordagem indireta que aumenta o tempo em SW a cada transição HW→SW. Uma melhoria nesse sentido seria introduzir métricas ou formas de detectar o fim das fases de capacidade induzidas pela limitação especulativa do *hardware*. O outro ponto é que em apenas uma das configurações de uma aplicação (*Intset* usando lista ligada e 100% de buscas) foi observado uma mudança de fase significativa na qual o PhTM* foi o melhor entre todos os sistemas avaliados. A escassez de aplicações dificultou

a avaliação do PhTM* nesses cenários. Portanto, encontrar novas aplicações e usá-las para reavaliar a proposta certamente complementariam os resultados obtidos neste trabalho.

Bibliografia

- [1] Transactional Memory Benchmarks and Applications. <https://code.google.com/archive/p/tm-benchmarks/>. Last access 2016-04-25.
- [2] *Intel[®] Transactional Memory Compiler and Runtime Application Binary Interface*, May 2009.
- [3] *Intel[®] 64 and IA-32 Architectures Software Developer's Manual*, Jan 2013. Vol. 3B.
- [4] Yehuda Afek, Hillel Avni, and Nir Shavit. Towards consistency oblivious programming. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, pages 65–79, Berlin, Heidelberg, 2011. Springer-Verlag.
- [5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.
- [6] Alexandro Baldassin, Edson Borin, and Guido Araujo. Performance implications of dynamic memory allocators on transactional memory systems. pages 87–96, February 2015.
- [7] Alexandro José Baldassin. *Explorando memoria transaccional em software nos contextos de arquiteturas assimétricas, jogos computacionais e consumo de energia*. PhD thesis, Universidade Estadual de Campinas (UNICAMP), December 2009.
- [8] Luiz Andre Barroso and Urs Holzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, December 2007.
- [9] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.
- [10] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 187–200, New York, NY, USA, 2014. ACM.
- [11] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing*, 2014.
- [12] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*, volume 10. MIT press, 2008.

- [13] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. *IBM Journal of Research and Development*, 51(5):559–572, September 2007.
- [14] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 39–52, March 2011.
- [15] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, pages 67–78, January 2010.
- [16] Peter Damron, Alexandra Fedorova, and Yossi Lev. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, October 2006.
- [17] João P. L. de Carvalho, Rafael Pizzirani Murari, and Alexandro Baldassin. Reavaliando o Conjunto de Aplicações STAMP em um Novo Hardware Transacional. In *WSCAD 2015*, Florianópolis, Santa Catarina, Oct 2015.
- [18] Dave Dice, Timothy L Harris, Alex Kogan, Yossi Lev, and Mark Moir. Hardware extensions to make lazy subscription safe. *arXiv preprint arXiv:1407.6968*, 2014.
- [19] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, March 2009.
- [20] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *20th International Symposium on Distributed Computing*, pages 194–208, September 2006.
- [21] Nuno Diegues and Paolo Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219. USENIX Association, 2014.
- [22] Edsger Wybe Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, 1965.
- [23] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, April 2011.
- [24] Robert Ennals. Software transactional memory should not be obstruction free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006.
- [25] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

- [26] Sanjay Ghemawat and Paul Menage. TCMalloc : Thread-caching malloc. <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>. Acessado em Junho de 2012.
- [27] Bhavishya Goel, Ruben Titos-Gil, Atul Negi, Sally A McKee, and Per Stenstrom. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 615–624. IEEE, 2014.
- [28] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 101–110, April 2010.
- [29] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pages 175–184, February 2008.
- [30] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2 edition, June 2010.
- [31] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.
- [32] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.
- [33] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report CRL-92-7, Cambridge Research Lab, November 1992.
- [34] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.
- [35] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM system z. In *Proceedings of the 45th ACM/IEEE International Symposium on Microarchitecture*, pages 25–36, December 2012.
- [36] Tetsu Kobayashi, Shigeyuki Sato, and Hideya Iwasaki. Efficient use of hardware transactional memory for parallel mesh generation. In *Parallel Processing (ICPP), 2015 44th International Conference on*, pages 600–609. IEEE, 2015.
- [37] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, March 2006.
- [38] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

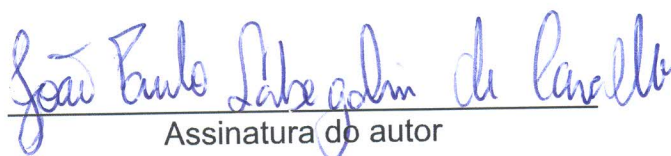
- [39] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, June 2007.
- [40] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased Transactional Memory. In *Workshop on Transactional Computing (Transact)*, 2007.
- [41] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, March 1977.
- [42] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.
- [43] Alexander Matveev and Nir Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 11–22, New York, NY, USA, 2013. ACM.
- [44] Alexander Matveev and Nir Shavit. Reduced Hardware NOrec: An opaque obstruction-free and privatizing HyTM. In *Transact 2014 Workshop*. Citeseer, 2014.
- [45] Alexander Matveev and Nir Shavit. Reduced Hardware NOrec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM.
- [46] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.
- [47] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, April 1965.
- [48] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 144–157, New York, NY, USA, 2015. ACM.
- [49] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. "O'Reilly Media, Inc.", 1996.
- [50] Wenjia Ruan, Yujie Liu, and Michael Spear. STAMP need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*, 2014.
- [51] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 1995.

- [52] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 187–197, March 2006.
- [53] Nir Shavit. Inherent Limitations of Hybrid Transactional Memory. In *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363, page 185. Springer, 2015.
- [54] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review*, 36(5):45–57, 2002.
- [55] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.
- [56] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, 1998.
- [57] David W. Wall. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV*, pages 176–188, New York, NY, USA, 1991. ACM.
- [58] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, September 2012.
- [59] Wm A Wulf and Sally A McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.
- [60] Lingxiang Xiang and Michael L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015*, pages 76–86, New York, NY, USA, 2015. ACM.
- [61] Craig Zilles and Ravi Rajwar. Implications of false conflict rate trends for robust software transactional memory. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 15–24, September 2007.

TERMO DE REPRODUÇÃO XEROGRÁFICA

Autorizo a reprodução xerográfica do presente Trabalho de Conclusão, na íntegra ou em partes, para fins de pesquisa.

São José do Rio Preto, 20/06/2016


Assinatura do autor