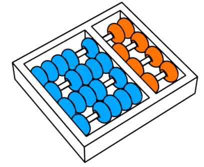
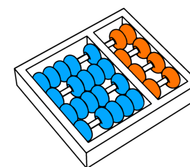


Daniel Henricus de Knegt Dutra Nicácio

“LUTS: A Light-Weight User-Level Transaction Scheduler”

CAMPINAS
2012



Universidade Estadual de Campinas
Instituto de Computação

Daniel Henricus de Knegt Dutra Nicácio

"LUTS: A Light-Weight User-Level Transaction Scheduler"

Orientador(a): **Prof. Dr. Guido Costa Souza de Araújo**

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Doutor em Ciência da Computação.

Este exemplar corresponde à versão final da Tese defendida por Daniel Henricus de Knegt Dutra Nicácio, sob orientação de Prof. Dr. Guido Costa Souza de Araújo.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

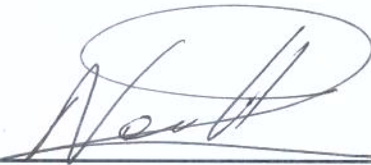
	Nicácio, Daniel Henricus de Knegt Dutra, 1984-
N512L	LUTS : a Light-Weight User-Level Transaction Scheduler / Daniel Henricus de Knegt Dutra Nicácio. – Campinas, SP : [s.n.], 2012.
	Orientador: Guido Costa Souza de Araújo.
	Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.
	1. Memória transacional. 2. Programação paralela (Computação). 3. Arquitetura de computador. 4. Linguagem de programação (Computadores). I. Araújo, Guido Costa Souza de,1962-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

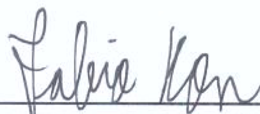
Título em inglês: LUTS : a Light-Weight User-Level Transaction Scheduler
Palavras-chave em inglês:
Transactional memory
Parallel programming (Computer science)
Computer architecture
Programming languages (Electronic computers)
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Nicolas Bruno Maillard
Sandro Rigo
Fabio Kon
Luiz Eduardo Buzato
Data de defesa: 31-08-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

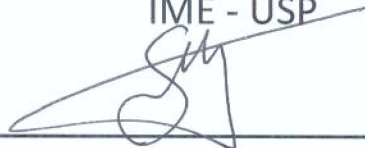
Tese Defendida e Aprovada em 31 de Agosto de 2012, pela Banca examinadora composta pelos Professores Doutores:



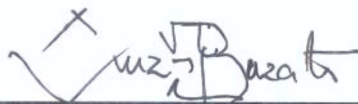
Prof. Dr. Nicolas Bruno Maillard
INFO - UFRGS



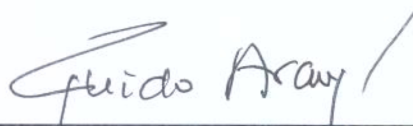
Prof. Dr. Fabio Kon
IME - USP



Prof. Dr. Sandro Rigo
IC - UNICAMP



Prof. Dr. Luiz Eduardo Buzato
IC- UNICAMP



Prof. Dr. Guido Costa Souza Araújo
IC - UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

LUTS: A Light-Weight User-Level Transaction Scheduler

Daniel Henricus de Knegt Dutra Nicácio

Agosto 2012

Banca Examinadora:

- Prof. Dr. Guido Costa Souza de Araújo (Orientador)
- Prof. Dr. Sandro Rigo
  Instituto de Computação - Universidade Estadual de Campinas (UNICAMP)
- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação - Universidade Estadual de Campinas (UNICAMP)
- Prof. Dr. Nicolas Maillard
  Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS)
- Prof. Dr. Fabio Kon
  Departamento de Ciência da Computação - Instituto de Matemática e Estatística - Universidade de São Paulo (IME/USP)

Abstract

Software Transaction Memory (STM) systems have been used as an approach to improve performance, by allowing the concurrent execution of atomic blocks. However, under high-contention workloads, STM-based systems can considerably degrade performance, as transaction conflict rate increases. Contention management policies have been used as a way to select which transaction to abort when a conflict occurs. In general, contention managers are not capable of avoiding conflicts, as they can only select which transaction to abort and the moment it should restart. Since contention managers act only after a conflict is detected, it becomes harder to effectively increase transaction throughput. More proactive approaches have emerged, aiming at predicting when a transaction is likely to abort, postponing its execution. Nevertheless, most of the proposed proactive techniques are limited, as they do not replace the doomed transaction by another or, when they do, they rely on the operating system for that, having little or no control on which transaction to run. This article proposes LUTS, a *Lightweight User-Level Transaction Scheduler*. Unlike other techniques, LUTS provides the means for selecting another transaction to run in parallel, thus improving system throughput. We discuss LUTS design and propose a dynamic conflict-avoidance heuristic built around its scheduling capabilities. Experimental results, conducted with the STAMP and STMBench7 benchmark suites, running on TinySTM and SwissTM, show how our conflict-avoidance heuristic can effectively improve STM performance on high contention applications.

Resumo

Sistemas de Memória Transacional em Software (MTS) têm sido usados como uma abordagem para melhorar o desempenho ao permitir a execução concorrente de blocos atômicos. Porém, em cenários com alta contenção, sistemas baseados em MTS podem diminuir o desempenho consideravelmente, já que a taxa de conflitos aumenta. Políticas de gerenciamento de contenção têm sido usadas como uma forma de selecionar qual transação abortar quando um conflito ocorre. No geral, gerenciadores de contenção não são capazes de evitar conflitos, tendo em vista que eles apenas selecionam qual transação abortar e o momento em que ela deve reiniciar. Como gerenciadores de contenção agem somente após a detecção de um conflito, é difícil aumentar a taxa de transações finalizadas com sucesso. Abordagens mais pró-ativas foram propostas, focando na previsão de quando uma transação deve abortar e atrasando o início de sua execução. Contudo, as técnicas pró-ativas continuam sendo limitadas, já que elas não substituem a transação fadada a abortar por outra transação com melhores probabilidades de sucesso, ou quando o fazem, dependem do sistema operacional para essa tarefa, tendo pouco ou nenhum controle de qual transação será a substituta. Esta tese apresenta o LUTS, *Lightweight User-Level Transaction Scheduler*, um escalonador de transação de baixo custo em nível de usuário. Diferente de outras técnicas, LUTS provê maneiras de selecionar outra transação a ser executada em paralelo, melhorando o desempenho do sistema. Nós discutimos o projeto do LUTS e propomos uma heurística dinâmica, com o objetivo de evitar conflitos, que foi construída utilizando os métodos disponibilizados pelo LUTS. Resultados experimentais, conduzidos com os conjuntos de aplicações STAMP e STMBench7, e executando nas bibliotecas TinySTM e SwissTM, mostram como nossa heurística para evitar conflitos pode melhorar efetivamente o desempenho de sistema de MTS em aplicações com alta contenção.

Agradecimentos

Eu gostaria de agradecer aos meus pais, que são os principais responsáveis pela pessoa que sou hoje. À minha mãe Lidu, que sempre esteve ao meu lado e por ser o meu porto seguro. Ao meu pai Paulo que, mesmo de longe, sempre se fez presente e possibilitou que eu pudesse seguir o caminho que escolhi. Também reservo um agradecimento especial ao meu irmão Lucas, o irmão mais massa que existe e meu melhor amigo. Estendo o meu muito obrigado à minha cunhada Luiza e à Beth Gomes, que completam essa família perfeita.

Agradeço à Pricila, que começou essa jornada sendo minha namorada e hoje é minha esposa. Obrigado Passion, por deixar os meus dias mais felizes, você foi parte essencial dessa etapa da minha vida e será de muitas outras que estão por vir.

Agradeço aos amigos de Juiz de Fora: Souza, Bruno, Bernard, Qualhada, Leitão, Giu, Eleusis, Bob, Pipa, DJ, Davis e Neto. É sempre bom (e sempre será) encontrar essa galera pra jogar card games, board games e até mesmo um futebol, e o principal: rir muito uns dos outros. Valew galera!

Também agradeço aos brothers de Campinas: Raoni, Auler, Gabs, Piga, Janjão, Cardosera, Bolaum, Ferrugem, César, Mário, Liana, George, Max, Faveri, Yang, Eco, Kleni, Baldas e Wesley. Esses são os caras que tornaram essa jornada divertida demais, foi muito da hora dividir o lab com vocês. Um agradecimento extra ao Baldas, com quem tive a honra de trabalhar em projetos de pesquisas e na escrita de diversos artigos.

Não posso deixar de citar os grandes amigos da república Tangamandápio: Thiago Senador, Fábio Fortes, Luciano Chaves e Cesar Chaves.

Agradeço aos meus gerentes dos estágios pelos quais passei, Alex Rosenberg da Sony Playstation e Magnus Christensson da Intel Sweden AB. A experiência profissional e de vida nesses estágios foi imensurável.

Agradeço aos professores do LSC: Borin, Côrtes, Ducatte, Sandro e Rodolfo, que contribuíram na minha formação como profissional.

Por fim, destaco minha gratidão ao meu orientador, Guido Araújo, que por ser um exemplo de professor e pesquisador me agregou importantes valores. Além disso, me ensinou como trilhar o caminho da vida acadêmica e a ser um profissional voltado para a

indústria. Também sou grato por ter me dado a oportunidade de alcançar um dos meus objetivos: trabalhar para renomadas empresas no exterior. Guido, tenho muito orgulho de ter sido seu orientando. Muito obrigado.

Sumário

Abstract **vii**

Resumo **viii**

Agradecimentos **ix**

1 Introdução **1**

1.1 Contribuições 5

1.2 Organização 6

2 Memória Transacional: Histórico e Mecanismos **7**

2.1 Perspectiva histórica 7

2.1.1 A barreira térmica 7

2.1.2 Paralelismo 8

2.2 Memória Transacional 13

2.2.1 A transação 13

2.2.2 Limitações 15

2.2.3 Implementações de MT 15

3 Trabalhos Relacionados **22**

3.1 Políticas de detecção de conflitos 22

3.2 Heurísticas para gerenciadores de contenção 23

3.3 Escalonadores de transações 25

3.3.1 Adaptive Transaction Scheduling - ATS 25

3.3.2 Shrink 26

3.3.3 CAR-STM 27

3.3.4 Steal-on-abort 27

3.3.5 M5-TM 28

3.3.6 Kernel based scheduler 29

3.3.7 McRT-STM 30

3.3.8 Contribuições desse trabalho 30

4 LUTS **32**

4.1 Visão Geral 32

4.1.1 Interface de Gerenciamento de *Threads* 32

4.1.2 Interface de Escalonamento 34

4.2 Uma Heurística Dinâmica para Prevenção de Conflitos 37

4.2.1 CILUTS - Transações Curtas 38

4.2.2 HASHLUTS - Transações Longas 39

5 Resultados Experimentais **43**

5.1 A importância de uma heurística 45

5.2 Speedup 48

5.3 `LUTS-dyn` com SwissTM 49

6 Conclusões **54**

6.1 Trabalhos Futuros 55

Bibliografia **57**

Lista de Tabelas

3.1 Resumo comparativo dos escalonadores de transações e sua principais características. O marcador X indica a presença da característica no escalonador de transações. A sigla "sw" significa software e a sigla "hw" significa hardware. 31

4.1 Resumo dos métodos de gerenciamento de *threads* e de escalonamento oferecidos pelo LUTS. 34

5.1 configuração do conjunto de aplicações STMBench7. 44

5.2 Duração das transações e nível de contenção (abortos/sucessos) para o conjunto de aplicações do STAMP. 47

Lista de Figuras

1.1 Alternativas de escalonamento: (a) totalmente concorrente (apenas uma transação tem permissão de realizar seu *commit*); (b) serialização; (c) parcialmente concorrente. 3

2.1 Duas tarefas que não podem ser executadas concorrentemente. 11
2.2 (a) Modelo de uma transação definida pelo programador. (b) Um conflito entre duas transações. 14
2.3 Exemplo no qual o modelo *eager* abortaria as transações T1 e T2, enquanto o modelo *lazy* abortaria apenas a transação T2. Não existe um consenso de qual modelo é o melhor, pois a eficiência de cada um depende da aplicação em questão. 16
2.4 Exemplo do funcionamento de um sistema de memória transacional em software com política de detecção de conflitos *lazy*. 21

4.1 LUTS provê um conjunto de métodos para escalonar tarefas e gerenciar *threads*, esses métodos podem ser utilizados tanto pela aplicação quanto pela biblioteca de memória transacional. 33
4.2 Mapeamento de *threads* do LUTS. O expedidor seleciona um RCE para ser executado e o mapeia para uma *thread* de sistema. 35
4.3 Informações contidas em um Registro de Contexto de Execução (RCE), necessárias para a troca de contexto realizada pelo LUTS. 36
4.4 Um exemplo ilustrando os metadados utilizados pela heurística e como a melhor transação a ser executada é selecionada dado um conjunto de transações ativas. 40
4.5 Pseudocódigo da heurística de previsão de conflitos com alta precisão e custo moderado. 41

5.1 Speedup da configuração `LUTS-dyn` em relação à `LUTS-pure` para o conjunto de aplicações STMBench7 (valores positivos são melhores). 46
5.2 Speedup do `LUTS-dyn` em relação ao `LUTS-pure` para o conjunto de aplicações do STAMP (valores positivos são melhores). 47

5.3 *Speedup* para as aplicações do STAMP. 51
5.4 *Speedup* para as aplicações do STMBench7. 52
5.5 Speedup do `LUTS-swiss` e `LUTS-tiny` em relação à implementação original de ambas bibliotecas para algumas aplicações do STAMP. 53

Capítulo 1

Introdução

Arquiteturas com vários núcleos renovaram o interesse em modelos de programação que proporcionam processamento paralelo eficiente. Entre as recentes propostas de abstrações de programação, Memória Transacional (MT) [41] tem atraído muita atenção. Recentemente, a indústria de processadores integrou sistemas de MT a seus processadores, reforçando a importância de MT para o avanço tecnológico [49, 50]. Uma transação em um sistema de MT é um bloco atômico de código que pode ser executado de forma isolada em relação ao restante do sistema, deixando o programador apenas com a tarefa de identificar o escopo das transações. As transações movem o fardo de sincronização entre as *threads* do programador para o sistema de MT, podendo evitar as principais dificuldades que a maioria dos programadores enfrentam ao programar máquinas com memória compartilhada.

Vejamos um exemplo clássico do uso de MT. Suponha um programa que execute tarefas em paralelo e essas tarefas acessam e modificam uma região de memória compartilhada. Vamos assumir que a região de memória compartilhada seja uma árvore. Tradicionalmente, a maneira mais simples de garantir às tarefas o acesso exclusivo a essa árvore é utilizar uma trava global. Portanto, antes de uma tarefa acessar a árvore ela adquire a trava e quando finalizar sua operação libera a trava. Essa abordagem é muito simples, pois o programador precisa apenas identificar os trechos de código que acessam a estrutura de dados compartilhada (regiões críticas) e delimitá-la com o uso de travas. Apesar de simples, essa abordagem não é eficiente, pois ela serializa todas as tarefas que acessam a estrutura de dados, limitando o grau de paralelismo da aplicação. Uma maneira mais eficiente é reduzir a granularidade das travas, por exemplo, criar uma trava individual para cada um dos nós da árvore. Dessa forma, as tarefas podem acessar a árvore concorrentemente desde que elas não acessem o mesmo nó. Entretanto, utilizar uma granularidade fina de travas aumenta drasticamente a complexidade de desenvolvimento do sistema, já que agora será possível ter condições de corrida, deadlocks e livelocks [93]. MT une o

melhor dos dois mundos. MT tem a simplicidade da trava global, pois o programador precisa apenas identificar as regiões críticas e delimitá-las com primitivas transacionais. E MT tem a eficiência da granularidade fina de travas, pois o sistema de MT fica responsável por verificar se duas transações acessam o mesmo dado na memória, e essa verificação é feita em nível de palavra, o que possibilita um alto grau de paralelismo.

Nessa tese, focamos nossa atenção em sistemas de Memória Transacional em Software (MTS), uma classe particular de sistemas de MT cuja implementação é completamente feita em software.

Apesar de sistemas de MTS melhorarem o desempenho de tarefas com alto nível de paralelismo, eles também podem piorar o desempenho consideravelmente em sistemas com alta taxa de contenção de dados. Contenção de dados ocorre quando mais de uma transação compete pelo mesmo dado e pelo menos uma dessas transações escreve nesse dado, causando um **conflito**. Quando um conflito ocorre, o sistema precisa decidir qual ação tomar, essa responsabilidade é atribuída ao **gerenciador de contenção** [37]. Para lidar com um conflito, o gerenciador de contenção normalmente aborta algumas das transações conflitantes e as reinicia posteriormente. Decidir qual transação abortar é parte de um conjunto de políticas de contenção utilizadas pelo gerenciador.

A maioria das pesquisas em gerenciadores de contenção procura desenvolver novas políticas de contenção, selecionando para abortar a transação que minimize o impacto no desempenho quando um conflito ocorre [36, 84, 90]. De forma geral, gerenciadores não são capazes de evitar conflitos, eles apenas selecionam qual transação abortar e quanto tempo esperar antes de reiniciar a transação. Os benefícios de reiniciar uma transação são difíceis de prever: ela pode reiniciar muito cedo, causando novos conflitos, ou muito tarde, deixando o processador vago e desperdiçando o potencial da execução concorrente. Como os gerenciadores tomam ações apenas depois de um conflito ser detectado, é difícil aumentar a frequência de transações finalizadas com sucesso.

Para ilustrar o problema considere a Figura 1.1 e um sistema de MT no qual os conflitos são detectados no momento em que eles ocorrem (em oposição a serem detectados apenas na fase de *commit*). As transações T_1 e T_2 serão executadas concorrentemente, mas possuem um conflito leitura/escrita. No cenário da Figura 1.1(a), T_1 primeiro escreve no endereço de memória compartilhada A e posteriormente o mesmo endereço é lido por T_2. Nesse ponto, um conflito leitura após escrita (RAW) [1] é detectado e o gerenciador de contenção precisa decidir qual transação abortar. Gerenciadores de contenção podem agir apenas após a ocorrência do conflito e, consequentemente, não são capazes de evitá-los.

Para lidar com o problema acima mencionado, técnicas baseadas em escalonamento foram propostas com a ideia de prevenir que transações que provavelmente vão conflitar executem em paralelo [5, 8, 25, 28, 57, 101]. O grande desafio encontrado por essas

[1] sigla em inglês para dependência *read after write*

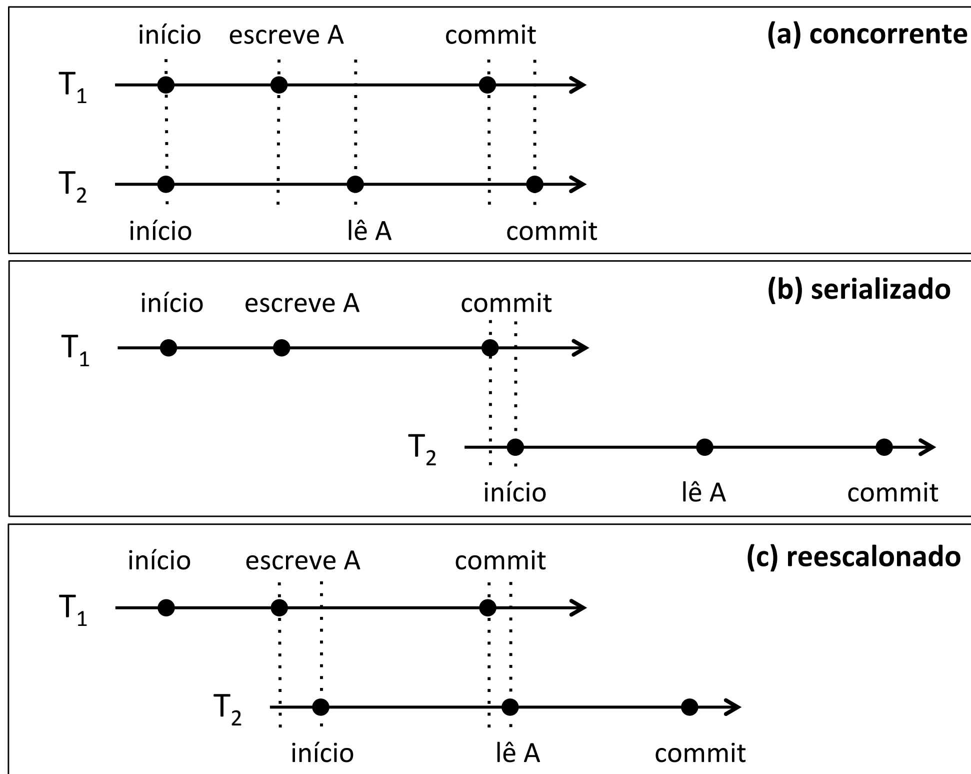


Figura 1.1: Alternativas de escalonamento: (a) totalmente concorrente (apenas uma transação tem permissão de realizar seu *commit*); (b) serialização; (c) parcialmente concorrente.

técnicas é conseguir decidir se uma transação irá causar um conflito antes de ela iniciar. As primeiras propostas [5, 25, 101] adotaram uma abordagem simples, na qual o nível de contenção do sistema é monitorado. Quando esse nível ultrapassa um limite pré-determinado, as transações são serializadas. Como uma transação só terá permissão de iniciar quando outra terminar sua execução, essa abordagem irá reduzir a contenção e evitar mais conflitos. Note porém, que isso pode piorar o desempenho do sistema, já que o paralelismo será limitado pela serialização.

Trabalhos posteriores [8, 28], também baseados em escalonadores, empregaram algum tipo de heurística para prever os conflitos de antemão. Por exemplo, o histórico de conflitos para cada par de transações pode ser gravado e usado para prever futuros conflitos. De volta ao exemplo da Figura 1.1, T_2 não terá permissão para iniciar se T_1 já estiver em execução. Todas as técnicas conhecidas até o momento irão simplesmente serializar a execução de T_2 (Fig. 1.1(b)) ao adquirir uma trava global.

Uma pergunta óbvia a ser feita é: serialização é realmente a melhor solução? Existem circunstancias onde a sobreposição de transações é certamente possível, mas gerar esse

escalonamento é difícil de ser feito. Considere o escalonamento mostrado na Figura 1.1(c). A transação T_2 lê a variável A logo após a transação T_1 realizar seu *commit* e, assumindo que nenhuma outra dependência exista, esse escalonamento é o melhor possível. Porém, posicionar a transação T_2 para que ela inicie exatamente nesse ponto não é uma tarefa fácil e pode envolver instrumentação de código pelo sistema de MTS, o que por sua vez pode gerar um *overhead* proibitivo. De fato, mesmo serializar transações de uma maneira eficiente é uma tarefa difícil. Considere por exemplo que a transação T_2 não acesse a variável A 10% das vezes que ela é executada. Nesse cenário, ambas as transações T_1 e T_2 poderiam executar concorrentemente sem qualquer conflito (assumindo que nenhuma outra variável compartilhada seja acessada pelas transações). Porém, se a heurística de predição de conflitos é baseada no histórico de conflitos, ela provavelmente irá falhar 10% das vezes. De forma geral, os programas tendem a exibir diferentes estágios de execução, fazendo com que seja muito difícil projetar uma heurística que funcione bem o tempo todo.

As atuais técnicas baseadas em escalonamento revelam uma limitação adicional: quando elas preveem que uma transação não deveria iniciar, elas recorrem exclusivamente à serialização. No exemplo da Figura 1.1, mesmo que tivesse uma terceira transação esperando para ser executada, ela não seria escalonada, independentemente se ela tivesse conflitos ou não com a transação T_1. Em outras palavras, as abordagens atuais não têm nenhum controle sobre o real escalonamento de transações senão liberar o processador e torcer para que o sistema operacional escalone uma *thread* com uma transação que não irá gerar conflitos.

Essa tese propõe um escalonador de transações em nível de usuário com baixo custo (LUTS) [2] e uma nova heurística para evitar conflitos que utiliza as funcionalidades do LUTS. LUTS implementa um escalonador cooperativo completo que não depende do escalonador do sistema operacional para realizar trocas de contexto das *threads* que estejam executando uma transação. A abordagem cooperativa do LUTS apresenta duas grandes vantagens quando comparadas com outros escalonadores de MT, considerados o estado da arte. Primeiro, ele lida com falso-paralelismo [25][3] de uma maneira elegante ao disparar uma quantidade de *threads* de sistema menor ou igual ao números de núcleos de processamento disponíveis e lidando com as *threads* excedentes internamente, totalmente transparente para o usuário e para o sistema operacional. Segundo, LUTS permite que o sistema de MT acesse de forma eficiente uma fila de transações em espera e troque a execução corrente por qualquer uma dessas transações. Isso permite o projeto de técnicas de escalonamento pró-ativas, o que não é possível com as abordagens atuais, que são

[2] sigla em inglês para *Lightweight User-level Transaction Scheduler*.

[3] Um cenário com falso-paralelismo é caracterizado por possuir mais *threads* do que número de núcleos de processamento disponíveis.

restritas à serialização. É importante notar que o LUTS é totalmente transparente para o usuário e para a aplicação, portanto, não é necessária nenhuma modificação no código fonte das aplicações que já estão adequadas a um sistema de MTS.

No cenário ideal de um sistema de MTS todos os núcleos de processamento estão sempre executando algum código (transacional ou não), e as execuções transacionais finalizam com sucesso. Com o uso do LUTS, o cenário ideal passa a ser factível, já que o LUTS, ao invés de serializar as transações, permite a substituição da transação fadada a abortar por outra transação que irá realizar seu *commit*. O quão próximo desse cenário o sistema estará irá depender de: (1) o nível de contenção da aplicação e (2) a heurística construída em cima do LUTS para prever os conflitos. Até a presente data, a heurística apresentada nesta tese possui os melhores resultados para sistemas de MTS.

1.1 Contribuições

As contribuições dessa tese são:

- Um escalonador cooperativo, projetado para evitar problemas de falso-paralelismo e prover uma interface simples e rica a ser usada por heurísticas que visam prever e evitar conflitos.

- Uma nova heurística que prevê e evita conflitos. Essa heurística utiliza as capacidades do LUTS para evitar o início de transações que provavelmente irão conflitar. Quando uma transação está prestes a iniciar, a heurística checa a sua probabilidade de conflitos com as transações já em execução. Se a probabilidade for alta, a heurística escolhe, da fila de transações em espera, uma transação com baixa probabilidade de conflito e a coloca em execução no lugar da transação com altas chances de conflito.

- Completa avaliação do LUTS através dos conjuntos de aplicações STMBench7 [38] e STAMP 0.9.10 [65]. Com uma boa diversidade de transações, LUTS apresenta bom ganho de desempenho quando comparado a trabalhos anteriores como o Shrink [28] e o ATS [101].

Este projeto de pesquisa resultou nas seguintes publicações:

- Daniel Nicacio, Alexandro Baldassin, and Guido Araujo. Transaction Scheduling Using Dynamic Conflict Avoidance In *International Journal of Parallel Programming*, aceito, a ser publicado em 2012/2013.

- Daniel Nicacio, Alexandro Baldassin, and Guido Araujo. LUTS: A Lightweight User-Level Transaction Scheduler. In *Algorithms and Architectures for Parallel Processing, volume 7016 of Lecture Notes in Computer Science*, pages 144-157. Springer Berlin / Heidelberg, 2011. [72]
- Daniel Nicacio and Guido Araujo. Reducing false aborts in STM systems. In *Algorithms and Architectures for Parallel Processing, volume 6081 of Lecture Notes in Computer Science*, pages 499-510. Springer Berlin / Heidelberg, 2010. [71]
- Daniel Nicacio, Alexandro Baldassin, and Guido Araujo. LUTS: A Lightweight User-Level Transactional Scheduler. Technical report, Institute of Computing, University of Campinas, December 2010. [73]
- Daniel Nicacio and Guido Araujo. Abortos falsos em sistemas de memória transacional em software. Technical Report IC-09-32, Institute of Computing, University of Campinas, September 2009. In Portuguese, 22 pages. [70]

1.2 Organização

O restante desse trabalho é organizado da seguinte forma:

- O Capítulo 2 apresenta o desenvolvimento da Computação e como chegamos ao ponto em que um novo paradigma de programação paralela se faz necessário.
- O Capítulo 3 reúne os principais trabalhos que têm como objetivo melhorar a taxa de transações que conseguem realizar seus *commits* e por consequência melhorar o desempenho do sistema de MTS. Em particular, damos especial atenção a trabalhos que implementam escalonadores de transações.
- O Capítulo 4 descreve detalhadamente o LUTS, um escalonador de transações em nível de usuário com baixo custo, e apresenta uma heurística para evitar abortos de transações. Essa heurística utiliza as funcionalidades fornecidas pelo LUTS.
- O Capítulo 5 apresenta os resultados experimentais comparando o desempenho de sistemas de MTS utilizando o LUTS e outros dois escalonadores, considerados o estado da arte.
- O Capítulo 6 resume os principais resultados desta tese e apresenta possíveis trabalhos futuros.

Capítulo 2

Memória Transacional: Histórico e Mecanismos

Este capítulo apresenta o que é um sistema de memória transacional, detalhando o seu funcionamento e suas peculiaridades. Também é intuito deste capítulo apresentar a motivação para esse estudo. A fim de entendermos a real importância dos sistemas de Memória Transacional para a Computação é preciso primeiro entender o que motivou a sua criação. Portanto, vejamos uma breve história de como chegamos aos sistemas de MT.

2.1 Perspectiva histórica

2.1.1 A barreira térmica

Em 1965 Gordon E. Moore constatou em seu artigo [67] que o número de dispositivos em um circuito integrado dobraria a cada ano, e previu que essa taxa se manteria nos próximos dez anos. Poucos anos depois sua teoria teve um leve ajuste, dizendo que o número de dispositivos em um circuito integrado dobraria a cada 18 meses. Sua teoria não só se confirmou como passou a guiar o avanço tecnológico até os dias de hoje. A indústria de processadores conseguiu manter esse ritmo, diminuindo o tamanho dos transistores e aumentando a frequência de operação dos chips.

Na última década a indústria esbarrou em uma forte oposição ao seu avanço tecnológico: a barreira térmica. Assim como os transistores diminuíam e a frequência dos processadores aumentava de forma exponencial, a potência dissipada também aumentava na mesma proporção [20]. Dessa forma, chegou-se ao extremo de não ser possível aumentar a frequência dos processadores, pois não havia como resfriá-los adequadamente. Em 2004 a indústria cancelou seus projetos baseados em uniprocessadores [32].

Com o intuito de superar o problema da barreira térmica e dar sobrevida à lei de Moore, a indústria apostou na computação paralela.

2.1.2 Paralelismo

Paralelismo é definido como uma forma de computação na qual vários cálculos são feitos simultaneamente, utilizando o princípio de que problemas grandes normalmente podem ser divididos em problemas menores, os quais podem ser resolvidos em paralelo [1]. Como discutido na seção anterior, limitações físicas que impossibilitam o aumento da frequência dos processadores e o crescente consumo de energia (e o consequente aumento da geração de calor) fizeram com que a computação paralela se tornasse o paradigma dominante em Arquiteturas de Computadores [6].

Tipos de paralelismo

Existem diferentes formas de realizar paralelismo em Computação:

- **Em nível de bit**: É uma forma de paralelismo baseada no aumento do tamanho da palavra utilizada pelo processador.
- **Em nível de instrução**: Se baseia em executar mais de uma instrução simultaneamente.
- **Paralelismo de dados**: Foca em distribuir os dados em diferentes nós computacionais do sistema.
- **Paralelismo de tarefas**: Foca em distribuir processos em execução (*threads*) em diferentes nós computacionais do sistema.

Em nível de bit: desde o advento da tecnologia VLSI (*very-large-scale integration*) na fabricação de chips em 1970 até 1986 essa foi a principal forma de avanço tecnológico em arquitetura de computadores. Ao aumentar o tamanho da palavra utilizada pelo processador, o número de instruções necessárias para fazer uma determinada computação diminui [17].

As arquiteturas passaram de 4-bit para 8-bit, depois para 16-bit e em seguida para 32-bit. As arquiteturas 32-bit se tornaram um padrão por praticamente duas décadas, até que foram lançadas arquiteturas 64-bit em 2003. Nos processadores de 32-bit/64-bit a largura do barramento de dados continua aumentando, hoje é possível transferir pelo menos 256 bits por rajada.

Apesar do grande avanço do paralelismo em nível de bit, os processadores de 8-bits, presentes em praticamente qualquer dispositivo eletrônico, continuam sendo responsáveis

por mais da metade do mercado (55%), sendo que apenas 10% dos processadores vendidos no mundo são 32-bit [97].

Em nível de instrução: também conhecida como ILP (*instruction-level parallelism*), essa forma de paralelismo tem como objetivo explorar a execução simultânea de instruções no processador. É função do projetista de processadores disponibilizar esse nível de paralelismo e em seguida o compilador fica responsável por escalonar as instruções de forma a tirar o máximo proveito do paralelismo oferecido pela arquitetura.

Existem várias técnicas para extrair paralelismo em nível instrução, e normalmente essas técnicas são usadas em conjunto [44]:

- ***Pipeline* de instruções**: A execução de uma instrução é dividida em diferentes estágios de execução, e diferentes estágios de diferentes instruções podem ser sobrepostos. Dessa forma, durante um único ciclo do processador várias instruções estão sendo executadas.

- **Arquiteturas superescalares**: Essas arquiteturas possuem unidades funcionais redundantes, dessa forma, elas são capazes de disparar e executar mais de uma instrução por ciclo. Note que as instruções continuam sendo disparadas de um fluxo sequencial e a dependência entre as instruções precisa ser verificada dinamicamente.

- **Execução fora de ordem**: O objetivo desse paradigma é usar os ciclos de processamento que seriam desperdiçados para executar uma instrução. Nesse paradigma, a ordem em que as instruções são executadas é dirigida pela disponibilidade dos dados para cada instrução e não pela ordem em que elas estão no programa. Portanto, se uma instrução está esperando por algum dado (ex: o dado não estava na memória cache), ela fica em espera e uma instrução seguinte que já tenha seus dados disponíveis é executada em seu lugar. As arquiteturas utilizam o algoritmo de Tomasulo [95] para implementar a execução fora de ordem.

- **Renomeação de registradores**: É uma técnica utilizada para evitar a serialização de operações imposta pelo reuso de registradores. Essa técnica é utilizada pelo algoritmo de Tomasulo na execução fora de ordem [95].

- **Execução especulativa**: Permite a execução de uma instrução mesmo sem ter a certeza se aquela instrução deveria ser executada [44]. Muito comum em instruções seguintes a uma instrução de desvio, enquanto o resultado do desvio é calculado, a instrução seguinte já é executada especulativamente. Para saber qual instrução executar, o alvo do desvio ou a instrução seguinte, a técnica de predição de desvios é utilizada.

Paralelismo de dados: em um sistema com múltiplos processadores e um único conjunto de instruções (SIMD - *Single Instruction Multiple Data*), o paralelismo é alcançado quando cada processador executa a mesma tarefa em diferentes partes do conjunto de dados [47].

Paralelismo de tarefas: em um sistema com múltiplos processadores, o paralelismo de tarefas é alcançado quando cada processador executa uma *thread* (ou processo) diferente num conjunto de dados, que poder ser o mesmo ou não. As *threads* podem executar o mesmo código ou códigos distintos. No caso geral, as *threads* comunicam entre si à medida que realizam suas tarefas.

Em 2001 foram lançados os primeiros processadores modernos baseados em paralelização em nível de *threads* (*Thread Level Paralelism* - TLP), permitindo a execução de múltiplas *threads* simultaneamente (*Simultaneous MultiThreading* - SMT) [96].

Programação Paralela

O primeiro fator a ser considerado durante o desenvolvimento de programas paralelos é a identificação de trechos de código passíveis de serem paralelizados. Seguindo a lei de Amdahl [2], o ganho máximo de desempenho através da paralelização é igual a:

$$1/(1-P)+P/N$$

onde P é a proporção de código que pode ser paralelizado (em relação ao tempo de execução), (1 - P) é a proporção que não pode ser paralelizada e N é o número de tarefas nas quais o código está sendo paralelizado. Portanto, se um programa tiver apenas 40% de código passível de paralelização em uma máquina com quatro processadores, o ganho máximo de desempenho será igual a 1.43x.

Programas que utilizam computação paralela são mais difíceis de escrever do que programas sequenciais [75]. A concorrência introduz novas possibilidades de erros durante a programação; desses erros, condições de corrida são os mais comuns. Comunicação e sincronização entre as tarefas costumam ser os principais obstáculos na obtenção de bom desempenho.

Em programas paralelos, as diferentes tarefas (normalmente identificadas por *threads*) precisam fornecer e receber informações de outras tarefas. Essa comunicação normalmente é feita através de memória compartilhada (em contraposição a sistemas de passagem de mensagens). O responsável por controlar o acesso das tarefas à essa memória compartilhada é o programador. Para entendermos melhor o problema, suponha as duas tarefas da Figura 2.1, elas executam o mesmo código: incrementar uma variável X. Quando executadas em paralelo, suas instruções podem ser executas de forma intercalada. Dessa forma,

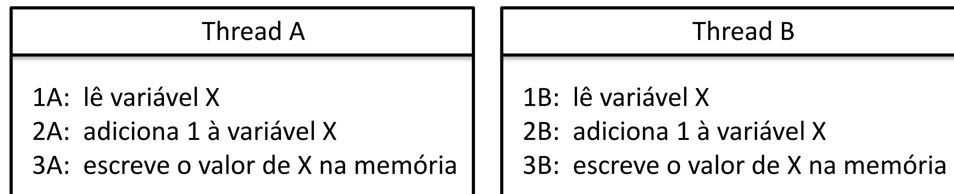


Figura 2.1: Duas tarefas que não podem ser executadas concorrentemente.

se a tarefa A executar a instrução 1A entre as instruções 1B e 3B (executadas pela tarefa B), o resultado da computação estará incorreto. O mesmo ocorreria se 1B fosse executada entre as instruções 1A e 3A. Essa situação é conhecida como "condição de corrida" (*race condition*) [69].

Portanto, é tarefa do programador identificar situações onde exista condição de corrida e garantir que as *threads* estejam sempre trabalhando com dados consistentes. Trechos de código onde existe condição de corrida são chamados de regiões críticas [93]. Portanto é preciso sincronizar o acesso das regiões críticas à memória compartilhada, em outras palavras, realizar exclusão mútua [93]. Os métodos de sincronização podem ser divididos em mecanismos bloqueantes e mecanismos não-bloqueantes.

Mecanismos bloqueantes

Os mecanismos bloqueantes são baseados em travas (*locks*) e variáveis de condição. O conceito de trava surgiu na década de 60 [21], e as variáveis de condição vieram em seguida, relacionadas ao conceito de monitor [48]. Por fim, outro termo muito utilizado é o semáforo, que é uma generalização de trava [24]. Esses conceitos existem há mais de 40 anos e continuam sendo os mais utilizados em programação concorrente.

A trava é utilizada para garantir que uma região crítica não seja executada em paralelo com outras regiões críticas que acessem o mesmo conjunto de dados. Portanto, quando uma *thread* está prestes a executar uma região crítica, ela primeiro tenta adquirir a trava. A trava é concedida a apenas uma *thread* por vez. Quando a *thread* acaba de executar a região crítica ela libera a trava para que outra *thread* possa adquiri-la. Dessa forma, uma região crítica delimitada por uma trava não é executada concorrentemente com outras regiões críticas associadas à mesma trava.

Dado um objeto compartilhado pelas *threads*, a maneira mais fácil de controlar os acessos a esse objeto é associar todas as operações sobre o mesmo a uma única trava. Porém, o resultado dessa abordagem pode ser a serialização de todos os acessos ao objeto, limitando o ganho de desempenho. A solução é diminuir a granularidade da trava, associando diferentes campos de um objeto a diferentes travas. Essa abordagem pode aumentar substancialmente o desempenho do código, mas também aumenta drasticamente a com-

plexidade de desenvolvimento do código. Note que usando uma trava para cada membro (ou conjunto de membros) do objeto pode gerar situações de *deadlock* ou *livelock* [93] caso exista uma relação de dependência entre os membros do objeto. Desenvolver um programa paralelo utilizando uma granularidade fina de travas é reconhecidamente uma tarefa muito difícil e geralmente feita apenas por especialistas.

Mecanismos não-bloqueantes

Mecanismos não-bloqueantes garantem que *threads* competindo por algum recurso compartilhado não tenham o seu progresso atrasado indefinidamente devido à exclusão mútua. Sendo assim, esse mecanismo não faz uso de travas para sincronizar tarefas. Os mecanismos não-bloqueantes podem ser classificados em três tipos de algoritmos:

- **Livre de espera (*wait-free*)**: É o tipo de algoritmo mais abrangente, para fazer parte deste grupo o algoritmo precisa garantir progresso de todas as *threads* e não apresentar casos de *starvation* [93], situação na qual o processo nunca obtém determinado recurso, impossibilitando o término do programa.

- **Livre de travas (*lock-free*)**: Esse grupo de algoritmos deve assegurar que ao menos uma *thread* tenha progresso, portanto ele permite *starvation*, mas é garantido que o programa como um todo tenha progresso (mesmo que necessite de muito tempo). Todos os algoritmos livres de travas também são livres de espera.

- **Livre de obstruções (*obstruction-free*)**: Um algoritmo pertencente a esse grupo garante que uma *thread* sempre progride se ela for executada isoladamente. Portanto, um algoritmo desse tipo deve ser capaz de detectar conflitos entre *threads*, ou seja, detectar se duas *threads* acessaram a mesma região de memória simultaneamente. Caso isso ocorra, o algoritmo também deve ser capaz de desfazer aquelas operações relacionadas à memória compartilhada. O processo de restaurar o estado anterior ao da execução das operações conflitantes é chamado de *rollback*. Todos os algoritmos livres de obstrução também são livres de travas.

Implementações que utilizam mecanismos não-bloqueantes normalmente se baseiam em instruções do próprio *hardware* para alterar um valor de forma atômica. A instrução mais utilizada nesse caso é o CAS (*Compare-and-Swap*).

Algoritmos não-bloqueantes também são reconhecidamente difíceis de implementar [64]. Ainda hoje, algoritmos não-bloqueantes para determinadas estruturas de dados são assuntos de artigos publicados em revistas e conferências.

2.2 Memória Transacional

O termo Memória Transacional (MT) surgiu em 1977, quando foi proposto por Lomet [56], porém ele só foi colocado em prática em 1993 quando Herlihy e Moss fizeram a primeira implementação de um sistema de Memória Transacional em Software (MTS) [46]. A partir de então MT se tornou um tópico constante em atividades de pesquisa.

O principal objetivo de MT é facilitar a programação paralela. Como vimos anteriormente, implementar programas concorrentes com desempenho satisfatório é uma tarefa complexa, normalmente feita apenas por especialistas. MT tenta conciliar a facilidade do uso de travas com granularidade alta e a eficiência das travas de granularidade fina.

MT permite ao programador definir trechos de código como uma transação, que podem ser executados atomicamente. Como mostrado no exemplo da Figura 2.2(a), as transações definidas pelo programador podem ser executadas em paralelo umas com as outras desde que elas não conflitem. Um conflito entre transações ocorre quando duas transações acessam uma mesma posição de memória e pelo menos um desses acessos é uma escrita. Quando ocorre um conflito, uma das duas transações envolvidas deve ser abortada. Qualquer computação realizada pela transação abortada é desfeita, voltando ao estado imediatamente anterior ao do início da transação, esse processo se chama *rollback*. A Figura 2.2(b) mostra um exemplo de conflito: duas transações (T1 e T2) executam em paralelo e T1 escreve na posição de memória X, quando T2 tenta ler essa mesma posição de memória ela verifica que o valor de X foi alterado após o seu início e portanto ele não é válido para o seu contexto, em seguida a transação T2 é abortada.

MT pode ser visto como um sistema otimista, pois mesmo sabendo que duas transações acessam informações de uma mesma estrutura de dados (por exemplo, uma lista ligada), ele permite que as transações executem em paralelo na esperança de que elas estejam acessando diferentes regiões da lista. Se for esse o caso, ambas as transações irão realizar seus *commits* com sucesso. Caso elas acessem exatamente a mesma informação da estrutura de dados uma das duas transações será abortada.

2.2.1 A transação

Transação é uma forma de execução emprestada da comunidade de banco de dados [35]. Nesse contexto podemos decompor a semântica de transações em quatro requisitos, normalmente denominados como propriedades "ACID": atomicidade, consistência, isolamento e durabilidade.

- **atomicidade**: assegura que todas as operações sejam executadas ou caso isso não seja possível, nenhuma operação é executada.

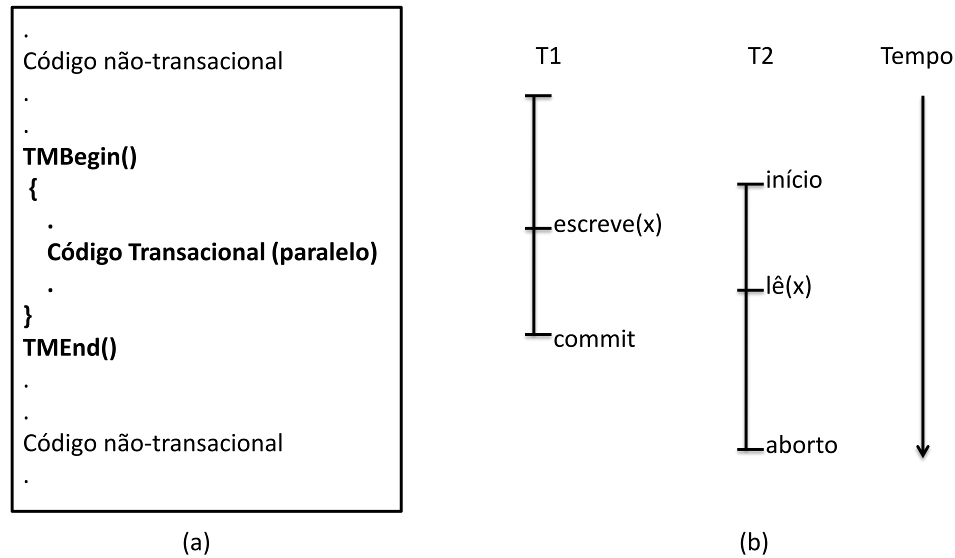


Figura 2.2: (a) Modelo de uma transação definida pelo programador. (b) Um conflito entre duas transações.

- **consistência**: uma transação sempre leva o sistema de um estado consistente para outro também consistente.
- **isolamento**: os resultados parciais de uma transação não são visíveis ao ambiente externo à transação.
- **durabilidade**: as mudanças efetivadas por uma transação são permanentes e resistentes a uma eventual falha do sistema.

Sistemas de MT garantem atomicidade e isolamento. A atomicidade garante que mudanças de estado provocadas por código de uma transação sejam indivisíveis pela perspectiva de qualquer outro código que esteja executando concorrentemente. Em outras palavras, qualquer código em execução só pode visualizar o estado do sistema imediatamente antes ou imediatamente depois de uma transação. O isolamento assegura que mudanças de estado provocadas por código concorrente não afetam o resultado da transação, ou seja, uma transação sempre produz o mesmo resultado que ela produziria se estivesse executando sem nenhuma outra tarefa concorrente.

Geralmente, MT não provê consistência e durabilidade. Consistência não é provida, pois ainda é responsabilidade do programador gerar código sem erros, garantindo o estado consistente do sistema ao longo de uma transação. Já em banco de dados uma transação é responsável por manter a consistência do sistema seguindo regras pré-estabelecidas (cascata, gatilhos, restrições, etc). Durabilidade também não é provida por um sistema de

MT. Em uma eventual falha do sistema (i.e. queda de energia), as mudanças efetivadas por uma transação podem se perder, já que as mesmas podem estar armazenadas em memória volátil.

Transações formam uma base de abstração para a programação paralela, elas podem ser vistas como blocos de código a serem combinados sem a necessidade de um profundo conhecimento sobre os mesmos. Elas podem ser comparadas a procedimentos e objetos, os quais podem ser combinados em códigos sequenciais [42].

2.2.2 Limitações

Transações em si não podem substituir toda a sincronização em programas paralelos [61]. Sincronização não se resume apenas à exclusão mútua, também é usada para coordenar a execução de tarefas independentes e garantir que uma tarefa espere até que outra termine. Considere por exemplo um cenário com uma tarefa produtora e outra consumidora, onde uma escreve um valor e a outra tarefa lê esse valor. Transações garantem que os acessos a essa variável estarão corretos e consistentes, mas as tarefas irão criar inúmeras situações de conflitos e muitos abortos ocorrerão. Essa situação de espera ocupada e com abortos está longe de ser a mais eficiente, uma melhor opção é a tarefa produtora sinalizar que a consumidora pode executar.

O desempenho de sistemas de memória transacional ainda não é satisfatório para serem adotados com uma solução definitiva [54, 74]. Sistemas de MTS ainda impõe *overhead* significativo em códigos rodando como uma transação. Sistemas de Memória Transacional em Hardware (MTH) podem diminuir o *overhead*, mas normalmente dependem de MTS para tratar transações longas.

No Capítulo 4, apresentamos a principal contribuição desse trabalho, um escalonador de transações chamado LUTS. LUTS tem como objetivo diminuir a contenção em sistemas de MTS e dessa forma melhorar o desempenho dos mesmos, fazendo com que MTS se aproxime de uma solução eficiente para programação paralela.

2.2.3 Implementações de MT

Sistemas de MT podem ser implementados em software (MTS) ou em hardware (MTH). Comum aos dois tipos é a característica otimista do sistema, ambos executam as transações supondo que elas não irão conflitar com outras transações, caso um conflito ocorra o sistema aborta umas das duas transações e reverte suas modificações (*rollback*). As implementações se baseiam em dois conceitos principais: versionamento de dados e detecção/resolução de conflitos.

O versionamento de dados consiste em gerenciar as versões de cada dado acessado por uma transação. Normalmente são mantidas duas cópias do dado: a versão original e a

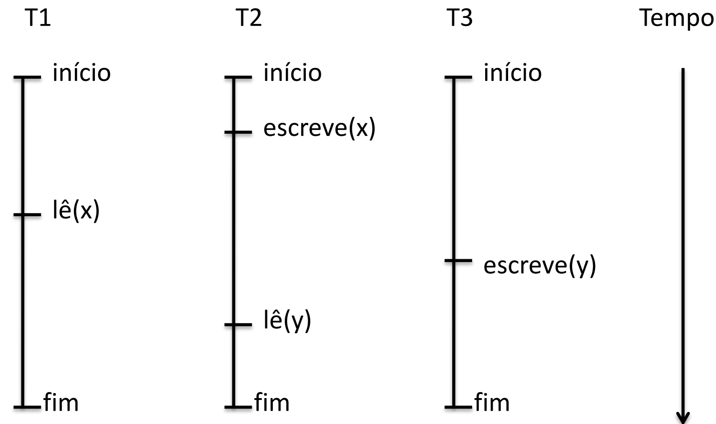


Figura 2.3: Exemplo no qual o modelo *eager* abortaria as transações T1 e T2, enquanto o modelo *lazy* abortaria apenas a transação T2. Não existe um consenso de qual modelo é o melhor, pois a eficiência de cada um depende da aplicação em questão.

versão modificada pela transação.

A detecção e resolução de conflitos são feitas da seguinte forma: cada transação mantém um conjunto de endereços de escrita e um conjunto de endereços de leitura, quando existe uma intersecção entre o conjunto de leitura e o conjunto de escrita de transações diferentes um conflito é detectado. Em outras palavras, um conflito entre transações ocorre quando duas transações acessam o mesmo endereço e pelo menos uma delas modifica o dado contido naquele endereço.

O versionamento de dados e a detecção de conflitos podem ser *eager* ou *lazy*. No modelo *eager*, os dados modificados são imediatamente alterados na memória principal e uma versão dos dados originais é mantida em um *undo log* caso o *rollback* seja necessário. Como os dados são imediatamente alterados, outra transação que acessar esse dado irá detectar o conflito imediatamente e abortar sua execução. A operação de efetivar as alterações de uma transação e deixá-las visíveis para o restante do sistema é denominado o *commit* da transação. No modelo *eager* a operação de *commit* é rápida, pois as alterações dos dados já foram refletidas na memória compartilhada, mas a operação de abortar é bastante custosa. No modelo *lazy*, as alterações nos dados são armazenadas em um buffer local da transação e são efetivadas somente na fase de *commit*, dessa forma, as transações só detectam um conflito após o *commit* de uma delas. Nesse modelo, o *commit* é mais custoso, enquanto o aborto é praticamente sem custos. Note que enquanto o modelo *eager* detecta conflitos o mais rápido possível, evitando computação que parece estar fadada a ser jogada fora, o modelo *lazy* pode evitar abortos desnecessários causados pelo *eager*.

A figura 2.3 ilustra um exemplo com 3 transações em execução: T1, T2 e T3. T1 conflita com T2 e T2 conflita com T3. Portanto, no modelo *eager* T1 é abortada e logo

em seguida T2 também é abortada devido ao seu conflito com T3. Já no modelo *lazy*, o conflito é detectado apenas na fase de *commit*, de modo que apenas T2 é abortada. Nesse exemplo o modelo *lazy* seria mais eficiente, mas não existe um consenso de qual modelo é melhor, pois a eficiência de cada um depende da aplicação em questão.

Memória Transacional em Software

Shavit e Touitou foram os primeiros a demonstrar que era possível implementar operações atômicas e livres de trava completamente em software, mas sua implementação exigia que cada um dos endereços acessados por transações fossem identificados previamente [87]. Em seguida Herlihy *et al.* implementaram o *Dynamic Software Transactional Memory* (DSTM), o primeiro sistema de MTS sem a limitação do trabalho de Shavit [45].

As implementações em software ficaram bastante populares, pois permitem grande flexibilidade em termos semânticos e podem ser testadas e utilizadas em processadores atuais, sem a necessidade de simulação. Existem dois tipos principais de implementações: não-bloqueantes e bloqueantes. Além disso, os sistemas em software se diferenciam principalmente na granularidade do versionamento de dados, que pode ser por objetos ou por palavras.

As primeiras implementações de MTS foram não-bloqueantes: DSTM (*Dynamic SYM*) [45], WSTM (*Word-based STM*) [40], OSTM (*Object-based STM*) [33], HaskellTM [42], ASTM (*Adaptative STM*) [58] e RSTM (*Rochester STM*) [59]. Dessas, apenas WSTM e HaskellTM usam granularidade por palavras, o restante utiliza granularidade por objetos.

Em 2006, Ennals mostrou que era possível implementar sistemas de MTS utilizando algoritmos bloqueantes, e que esse tipo de implementação tinha desempenho superior às implementações não-bloqueantes [29]. Ennals argumentou que as indireções usadas para acessar objetos sobrecarregavam a cache de dados, prejudicando o desempenho. A implementação com travas de Ennals fazia escritas utilizando um protocolo de travamento em duas fases [30] e as leituras eram feitas de forma otimista usando uma técnica já conhecida em banco de dados [53].

Após o trabalho de Ennals surgiram várias implementações bloqueantes: McRT-STM (Multi-core Run-time STM) [81], Bartok-STM [43], TL2 (Transactional Locking II) [23], TinySTM [31] e SwissTM [26]. Esses trabalhos seguiram o modelo proposto por Ennals e cada um deles adicionou novas características para aprimorar o sistema de alguma forma. Dentre esses trabalhos, TL2, TinySTM e SwissTM são consideras o estado da arte em sistemas de MTS [62, 92].

TL2, TinySTM e SwissTM utilizam um relógio lógico global para versionar os acessos à memória. Quando uma transação inicia sua execução, ela começa lendo o valor do relógio lógico e o armazena como sendo sua versão de leitura. Portanto, em toda leitura ou escrita, a versão do dado acessado é comparada com a versão de leitura da transação,

caso o valor do dado seja maior, a transação deve ser abortada, pois outra transação alterou a posição de memória após o início dessa transação. Isso garante que a transação está utilizando valores consistentes e não irá gerar algum comportamento indesejado (por exemplo, divisão por zero, loop infinito, acesso indevido de memória). Durante a efetivação das escritas, é adquirida uma trava para cada uma das posições alteradas, as versões das posições de leitura e escrita são novamente verificadas, o relógio lógico global é incrementado e cada posição de escrita é versionada com o novo valor do relógio lógico.

Os três sistemas acima mencionados (TL2, TinySTM e SwissTM) implementam o mesmo algoritmo com pequenas variações nas técnicas de programação utilizadas, a única diferença substancial entre eles se encontra na SwissTM, que escolhe dinamicamente qual política de detecção de conflitos a ser usada dependendo do padrão de leituras e escritas da transação.

A seguir apresentamos um exemplo detalhado de um sistema de MTS com política de detecção de conflitos *lazy* em execução, referencie a Figura 2.4 para acompanhar o exemplo. No exemplo temos três *threads* em execução e cada uma delas pode estar executando uma única transação em um dado instante de tempo. Quando a *thread* 1 inicia a transação Tx1 ❶, ela lê o relógio lógico global e atribui este valor à sua versão, nesse caso 100. Logo em seguida ❷, a *thread* 3 inicia Tx3 e também obtém o valor 100 para sua versão. Como nenhuma transação terminou sua execução entre o início de Tx1 e de Tx3, ambas possuem o mesmo valor de versão. No próximo momento ❸, Tx1 e Tx3 fazem leituras das variáveis `X` e `Y`, como a versão das variáveis é menor que a versão da transação que está realizando a leitura, a leitura é considerada consistente. Nesse caso assumimos que as variáveis possuem versão 90 porque elas foram escritas por outra transação no passado. As transações Tx1 e Tx3 fazem escritas nas variáveis `X, Y, Z` e `K` ❹, como a política de detecção de conflitos é *lazy*, essas escritas ficam armazenadas em um buffer local às transações. Agora Tx3 realiza seu *commit* ❺, que consiste em (i) verificar se a versão de todas as variáveis lidas continuam menor do que sua versão, nesse caso a versão de `X` continua menor do que 100; (ii) adquirir travas para cada variável do seu buffer de escrita e para o relógio lógico global; enquanto ela possui essas travas, ela incrementa o valor do relógio lógico global (101 após o incremento) e atribui esse valor à versão das variáveis do seu buffer de escrita, portanto a variável `K` recebe versão 101 e (iii) escrever os novos valores das variáveis do buffer de escrita na memória. O próximo evento do sistema é o início da transação Tx2 pela *thread* 2 ❻, Tx2 recebe versão 101, já que o relógio foi incrementado pelo *commit* de Tx3. Tx2 lê a variável `X` ❼, como a versão de `X` é menor do que 101, a leitura é válida. No próximo instante de tempo ❽, Tx1 realiza o seu *commit*, incrementando o relógio lógico para 102 e definindo a versão das sua variáveis de escrita (`X`, `Y` e `Z`) para 102. Por fim, a transação Tx2 tenta ler a variável `Y` ❾, porém, ela verifica que a versão de `Y` (102) é maior que a sua própria versão (101), o que caracteriza

um conflito entre transações. Nesse caso Tx2 aborta e reinicia sua execução, agora com versão 102.

No exemplo acima, caso a política de detecção de conflitos fosse *eager*, as escritas de cada transação seriam feitas diretamente na memória e as variáveis em questão ganhariam uma nova versão imediatamente. Dessa forma, Tx2 iria abortar no momento ❼, quando tentasse ler a variável `X` e constatar que `X` já possui uma versão maior que a sua própria versão, já que a versão de `X` foi atualizada pela escrita de Tx1.

Algumas implementações de MTS não seguiram completamente o modelo proposto por Ennals. No RingSTM [92] os conjuntos de escrita e leitura são representados por uma assinatura, essas assinaturas são armazenadas em uma estrutura de anel, a ordem das assinaturas nesse anel controla a ordem de efetivação das transações e os conflitos são detectados verificando a intersecção entre a assinatura da transação que está sendo efetivada e as outras assinaturas presentes no anel. O DASTM (*Dependence-Aware STM*) [79] faz com que um conflito escrita após leitura (*read-after-write*) não produza um aborto de imediato, a transação usa o valor modificado pela escrita e, caso a transação responsável pela escrita aborte, o aborto se propaga para as transações dependentes. Por fim, STM-Lite [62] dedica uma *thread* para validar e efetivar as transações, as transações enviam seus conjuntos de leitura e escrita para a *thread* gerente, e esta *thread* processa os conjuntos para detectar possíveis conflitos.

Memória Transacional em Hardware

Os primeiros trabalhos de MT totalmente em hardware começaram com os artigos de Knight [51] e Herlihy e Moss [46].

As implementações de MT em hardware normalmente consistem em realizar alterações ao protocolo de coerência de cache dos processadores para realizar o versionamento dos dados e a detecção de conflitos. Portanto, é adicionado um estado extra ao número de estados de cada linha de cache, esse novo estado é responsável por dizer se aquela linha de cache é válida ou não quando lida/escrita por uma transação. Vale ressaltar que, como consequência direta, a granularidade dos sistemas em hardware é por linha de cache e não por palavras ou objetos.

A maior vantagem de MTH é o desempenho. Em meados de 2008, o desempenho de um sistema em hardware era até quatro vezes mais rápido do que em software [54]. Porém, sistemas em hardware também possuem algumas desvantagens. A principal delas é que seus conjuntos de leitura e escrita estão sempre limitados ao tamanho da cache de dados, dessa forma, transações muito longas não são versionadas na cache e precisam de suporte em software para serem executadas.

As principais implementações de MTH são: TCC (*Transactional Coherence and Consistency*) [39], UTM (*Unbounded Transactional Memory*) [3], VTM (*Virtual Transactional*

Memory) [76], LogTM (*Log-based Transactional Memory*) [68], PTM (*Page-based Transactional Memory*) [16], OneTM [11], MetaTM [77], LogTM-SE [100], TokenTM [12] e DATM (*Dependence-Aware Transactional Memory*) [78]. A diferença entre esses trabalhos é a complexidade e quantidade de hardware necessário para implementá-los. Alguns deles também dão suporte a transações aninhadas.

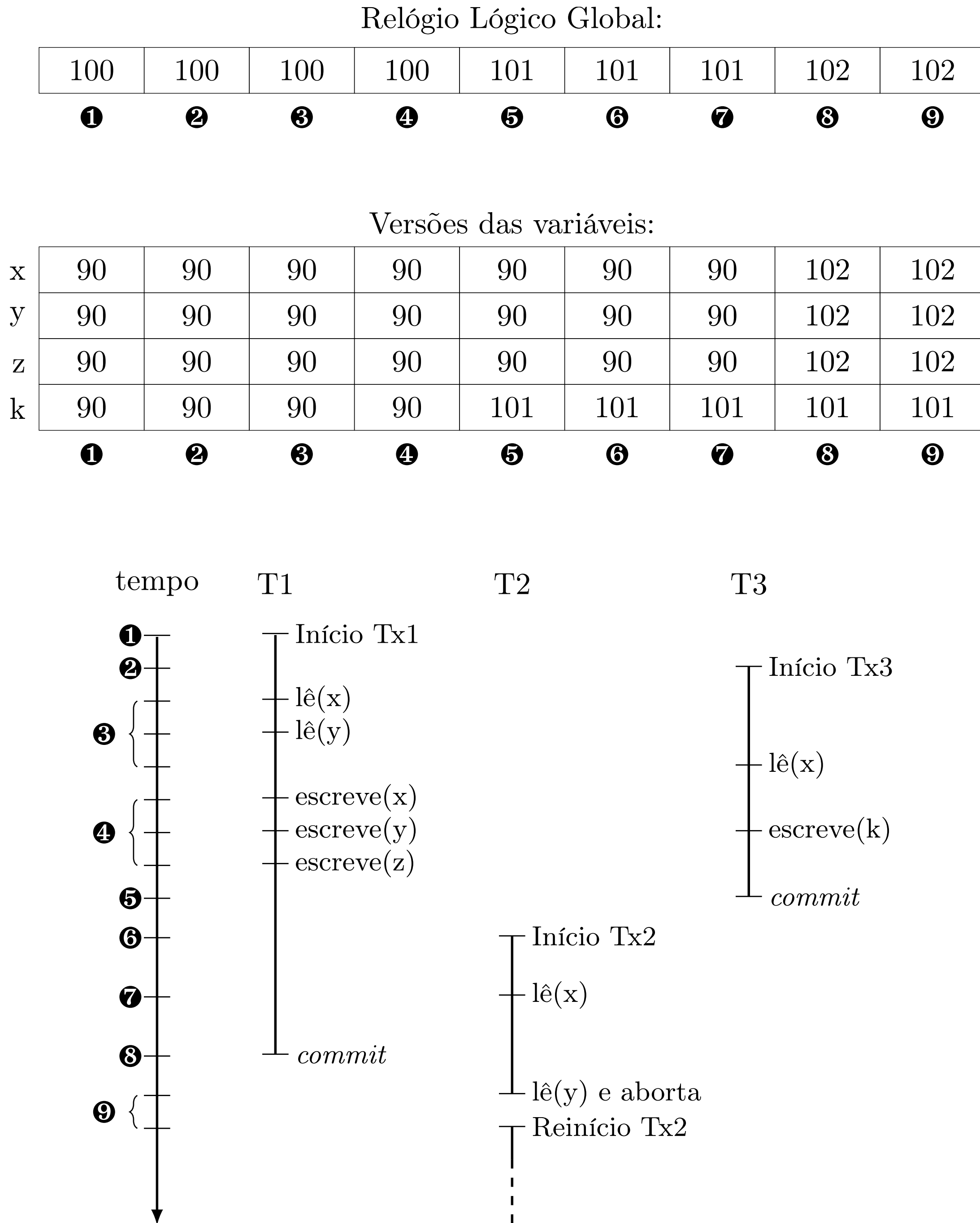
O objetivo em MTH continua sendo melhorar o desempenho e usar cada vez menos hardware em suas implementações. As atuais implementações ainda são consideradas muito complexas para serem adotas em um processador comercial. Além disso, a semântica de transações ainda não está madura o suficiente a ponto de justificar o investimento.

Memória Transacional Híbrida

Com a intenção de juntar as vantagens dos sistemas em software e em hardware, surgiram os sistemas híbridos. As primeiras abordagens utilizaram um hardware bem simples para rodar as transações, e quando uma transação não pudesse ser executada em modo hardware (i.e., a quantidade de dados versionados na transação é maior do que a quantidade suportada pela memória cache) ela passava a ser executada em modo software [18, 52]. O grande problema desse modelo é gerenciar transações que estão executando em modos diferentes. As soluções normalmente adotadas fazem com que as transações em modo hardware fiquem mais lentas.

Os modelos desenvolvidos posteriormente passaram a usar aceleração em hardware dos componentes mais custosos de MTS [66, 82, 88, 89]. Por exemplo, a validação é reconhecidamente um gargalo em MTS, e essa função pode então ser feita em hardware nos modelos híbridos.

Em 2009 a Sun chegou a anunciar um processador com suporte transacional [14], adotando uma abordagem híbrida [22], porém o projeto foi descontinuado meses depois. Recentemente a Intel anunciou que seus processadores Haswell darão suporte a MT, ele será capaz de converter programas que utilizam travas em programas transacionais e possuirá novas instruções para realizar início, *commit* e aborto de transações [50].



Relógio Lógico Global:

❶	❷	❸	❹	❺	❻	❼	❽	❾
100	100	100	100	101	101	101	102	102

Versões das variáveis:

	❶	❷	❸	❹	❺	❻	❼	❽	❾
x	90	90	90	90	90	90	90	102	102
y	90	90	90	90	90	90	90	102	102
z	90	90	90	90	90	90	90	102	102
k	90	90	90	90	101	101	101	101	101

Figura 2.4: Exemplo do funcionamento de um sistema de memória transacional em software com política de detecção de conflitos *lazy*.

Capítulo 3

Trabalhos Relacionados

Os primeiros artigos a abordar o desempenho de memória transacional, ou seja, a tentar aumentar a taxa de transações bem sucedidas por segundo, focaram na política de detecção de conflitos. Em seguida surgiram abordagens focadas no gerenciador de contenção. E por fim, surgiram trabalhos com técnicas pró-ativas para evitar conflitos de antemão. O trabalho desenvolvido nessa tese se encaixa no último tipo, onde prevemos que uma das transações prestes a executar irá abortar e evitamos que este conflito ocorra.

As próximas seções descrevem trabalhos com cada uma das abordagens citadas e apresentam um breve resumo com as características mais importantes de cada trabalho.

3.1 Políticas de detecção de conflitos

As políticas adotadas para detecção de conflitos em MTS podem ser *eager*, na qual o conflito é detectado o mais cedo possível, ou *lazy*, onde o conflito é detectado apenas na fase de *commit*. As duas abordagens têm suas vantagens, mas nenhuma é consistentemente superior à outra. Como discutido na Seção 2.2.3, enquanto a abordagem *eager* previne que uma transação execute desnecessariamente, a abordagem *lazy* pode evitar abortos, por exemplo quando a transação causadora do conflito também possui um conflito com uma terceira transação. Sistemas que utilizam *lazy* também são conhecidos como otimistas, por acreditarem que o conflito ainda pode ser resolvido antes do *commit* da transação, enquanto sistemas que utilizam *eager* são conhecidos como pessimistas. Mecanismos para implementar as duas abordagens foram apresentados em alguns trabalhos [83, 85, 98].

Existe sistemas híbridos que escolhem dinamicamente qual abordagem adotar de acordo com as características do programa. Dragojević *et al.* [26] utilizam detecção *lazy* para conflitos leitura/escrita e para conflitos escrita/escrita utilizam detecção *eager*. Outros trabalhos também mesclam características dos dois mecanismos para melhorar o desempenho do sistema [34, 88].

Por fim, existem trabalhos comparando o desempenho de cada política. Spear *et al.* [91] conclui que nenhum mecanismo de detecção de conflitos é consistentemente melhor em todas aplicações. Schoeberl *et al.* [86] considera, na média, o mecanismo *eager* supeior ao *lazy* para sistemas de MTS, mas para MTH o mecanismo *eager* é considerado muito custoso.

3.2 Heurísticas para gerenciadores de contenção

Outra forma de melhorar o desempenho de MT é aprimorar o gerenciador de contenção [37]. O gerenciador de contenção é responsável por decidir qual ação tomar quando um conflito entre transações ocorre.

Gerenciadores de contenção em sistemas de MTS foram primeiramente pesquisados no contexto de gerenciadores de contexto modulares, introduzidos por Herlihy *et al.* [45] para implementações não-bloqueantes de MTS. Devido à sua natureza modular, uma enorme quantidade de políticas de contenção foram sugeridas com o propósito de diminuir o número de conflitos e assim permitir o progresso do sistema [36, 37, 84, 90].

Implementações baseadas em travas geralmente empregavam heurísticas mais simples, como por exemplo, abortar a transação conflitante e atrasar o seu reinício utilizando um tempo de espera que aumenta exponencialmente a cada tentativa de reinício. Outros gerenciadores utilizam um sistema de prioridades entre as *threads*, a prioridade pode ser determinada utilizando diversos parâmetros, como a idade da transação, a quantidade de dados acessados pela transação, etc. Uma implementação de MTS que utiliza um gerenciador de contenção mostrou ganho de desempenho significativo em relação a um sistema sem o gerenciador de contenção. Ele também demonstra a sua capacidade de evitar *livelocks* [45, 58, 59]

Os principais gerenciadores de contenção são:

- **Polite** - O gerenciador Polite atrasa o reinício de uma transação conflitante por uma quantidade de tempo

  $$2^{(n+k)}$$

  nano-segundos, onde n é o número de vezes que o conflito ocorreu e k um parâmetro constante definido para cada arquitetura. Depois de um limite máximo de m tentativas, o gerenciador Polite aborta a outra transação conflitante. Os autores afirmam que obtiveram bons resultados com k = 4 e m = 22. Este gerenciador possui baixo custo e bons resultados, apresentando uma boa relação custo-benefício.

- **Karma** - O gerenciador Karma prioriza abortar transações que acabaram de iniciar, deixando que transações que já realizaram muito trabalho realizem o *commit*

sem serem interrompidas. Para medir a quantidade de trabalho realizado por uma transação, Karma conta a quantidade de objetos acessados pela transação. Em seguida Karma utiliza esse contador como a prioridade de uma transação, em caso de conflito, Karma aborta a transação com menor prioridade. Para ter garantias de progresso, uma transação que aborta mantém o valor de seu contador, de forma a ganhar mais prioridade à medida que sucessivos abortos aconteçam. O contador só é reiniciado após o *commit*. Após um aborto, Karma atrasa o reinício da transação por um período fixo de tempo. Apesar de ser mais custoso que o Polite, este gerenciador é mais inteligente ao dar prioridade às transações que estão em execução a mais tempo, pois um aborto dessa transação seria muito custoso.

- **Eruption** - Eruption é um gerenciador que combina as ideias do Karma e das técnicas `QOldDep` e `QCons` [55] para marcação de instruções na fila de instruções a serem executadas em uma arquitetura com execução fora de ordem. Nesta técnica a prioridade da transação bloqueada é adicionada à prioridade da outra transação conflitante, de forma a garantir que a mesma realize o *commit* o mais rápido possível e possibilite a execução da transação em espera. Este gerenciador é semelhante ao Karma, mas com uma heurística diferente para a atribuição de prioridades às transações.

- **Kindergarten** - Vagamente baseado na resolução de conflitos presente no problema *Drinking Philosophers* de Chandy e Misra [13], o gerenciador Kindergarten encoraja as transações a acessaram objetos alternadamente. O gerenciador mantém para cada transação uma lista (inicialmente vazia), durante um conflito, o gerenciador verifica se a transação inimiga está presente na lista, se sim, a transação conflitante é abortada, caso contrário, a transação inimiga é adicionada à lista e a transação dona da lista é atrasada por um período de tempo.

- **Timestamp** - Com o intuito de ser o mais justo possível com as transações, foi criado o gerenciador Timestamp, ele grava o tempo do sistema no início de cada transação. Em um conflito, a transação mais nova é abortada enquanto que a mais velha é marcada como uma possível transação zumbi. Se a transação abortada vier a conflitar novamente e a transação inimiga estiver marcada como zumbi, a transação zumbi é abortada. Transações ativas sempre limpam suas marcações zumbi assim que essas marcações são feitas. Assim como o Karma, este gerenciador é baseado em prioridades de transação, beneficiando transações que estão em execução a mais tempo, mas ao invés de rastrear a quantidade de objetos acessados pela transação, ele utiliza o tempo de execução da transação para definir qual a transação mais antiga.

- **Polka** - Após observarem que as técnicas Polite e Karma tinham os melhores desempenhos, foi criado um gerenciador que utiliza as melhores qualidades de cada um: Polka. Esse gerenciador combina o atraso exponencial do Polite com o contador de prioridades do Karma. O resultado é um gerenciador que atrasa uma transação abortada um número de vezes igual à diferença de prioridade entre as transações envolvidas, e a duração de cada atraso aumenta exponencialmente.

Apesar do progresso em gerenciadores de contenção, sistemas MTS ainda não eram capazes de antecipar a ocorrência de um conflito e assim aumentar a frequência de transações bem sucedidas [57, 101]. Pelo contrário, gerenciadores de contenção tradicionais privilegiam o uso de uma estratégia ação-reação, ao invés de tentar evitar os conflitos. Recentemente o foco passou para gerenciadores que se baseiam em escalonamento de transações. O trabalho dessa tese utiliza esse princípio. A seguir são mostrados outros trabalhos que também se basearam em escalonamento de transações.

3.3 Escalonadores de transações

3.3.1 Adaptive Transaction Scheduling - ATS

Yoo e Lee [101] descreveram o primeiro mecanismo de escalonamento, chamado *Adaptive Transaction Scheduling* (ATS). Seu foco é reduzir a re-execução de transações em cenários que apresentam alta contenção de dados. Para medir o nível de contenção do sistema, cada transação é associada a uma variável chamada Intensidade de Contenção (CI). CI é atualizada sempre que uma transação termina (*commit* ou aborto) através da fórmula:

$$CI_n = \alpha * CI_{n-1} + ((1 - \alpha) * CC)$$

onde CC é a contenção corrente e é definida como 0 em um *commit* e 1 em um aborto. Alpha é usado para decidir se é dado maior peso ao histórico recente da transação ou ao histórico antigo da transação, 0,3 foi o valor usado em seu artigo, dando mais importância ao passado recente da transação. Em todo início de transação, verifica-se o valor de CI, caso ele esteja abaixo de um limite pré-determinado (os autores afirmam que 0,5 é o valor com o qual obtiveram os melhores resultados), a transação inicia normalmente, caso contrário ela aciona o escalonador central e a transação é adicionada à uma fila de transações. As transações presentes na fila são serializadas pelo escalonador, ou seja, são executas uma de cada vez, evitando conflitos entre elas e diminuindo a contenção do sistema. Essa abordagem possui custo próximo de zero quando há paralelismo suficiente, porém, quando a taxa de conflitos aumenta, ela praticamente serializa a execução das transações.

ATS foi implementado tanto em sistemas de memória transacional em hardware quanto em software. Para a implementação em hardware foi utilizada o LogTM [68], nessa abordagem, o ATS conseguiu um ganho de desempenho de 1.97x. Em software, o ATS foi integrado ao RSTM [60] e conseguiu um ganho de desempenho de 1.4x.

ATS possui uma implementação muito eficiente, utilizando apenas uma trava global, o que torna essa solução muito atraente. Porém, serializar transações pode não ser a melhor resposta para alguns casos. A heurística para evitar conflitos desenvolvida nesta tese mostra que executar outra transação no lugar da transação que está prestes a abortar gera melhores resultados.

3.3.2 Shrink

Shrink [28] é outro escalonador que foca em evitar conflitos entre transações de antemão. Para escalonar as transações, o escalonador usa uma heurística chamada *serialization affinity*, na qual a probabilidade de serializar uma transação é proporcional à quantidade de contenção do sistema. Para prever conflitos, ele utiliza uma assinatura (um *bloomfilter*) [10] para guardar o padrão de acesso aos dados das últimas transações executadas naquela mesma *thread*. Mais especificamente, o conjunto de leitura é previsto pelo princípio da localidade temporal, no qual transações tendem a acessar a mesma estrutura de dados repetidas vezes. O princípio da localidade temporal não é eficaz para o conjunto de escritas, já que este é bem menor, portanto, para prever escritas, o escalonador utiliza o conjunto de escritas da última transação abortada e não consegue prever para transações que realizaram seus *commits* com sucesso. Antes de reiniciar uma transação, o Shrink compara as assinaturas de leitura e escrita das transações em execução com a assinatura da transação prestes a iniciar, se houver intersecção, é provável que a transação gere um conflito e a coloca numa fila se transações. O Shrink consegue prever corretamente 70% das leituras e escritas de uma transação.

Ao detectar que uma transação provavelmente irá abortar, o escalonador serializa essa transação. Para serializar as transações o escalonador utiliza uma trava global. Assim como o ATS, para evitar *overhead* desnecessário quando a contenção do sistema está baixa, o Shrink só ativa seu escalonador quando a contenção do sistema atinge um determinado limite.

Shrink foi integrado em dois sistemas de memória transacional em software: SwissTM [27] e TinySTM [31]. Resultados experimentais mostram que ele consegue bons ganhos de desempenho quando o número de *threads* em execução excede o número de CPUs disponíveis. Com o SwissTM, o ganho chegou a 55% no STMBench7 [38] e 120% no STAMP [65]. Com o TinySTM os ganhos foram ainda maiores, chegando a 200% no STAMP e 400% no STMBench7.

3.3.3 CAR-STM

CAR-STM [25] mantém uma fila de transações e uma *thread* (chamada de TQ *thread*) para cada núcleo de processamento disponível no sistema. Quando uma transação está prestes a iniciar, o distribuidor seleciona uma fila para inserir a transação e passa o controle da mesma para uma TQ *thread*. A ideia geral é inserir transações que conflitam umas com as outras na mesma fila, de forma que elas nunca executem em paralelo, evitando aborto de transações. Em tempo de execução, CAR-STM reage a um aborto colocando a transação abortada na mesma fila da TQ *thread* da outra transação conflitante, serializando a execução das duas transações. Note que, quando uma transação move para outra fila, todas as transações que estão na sua própria fila também são movidas, garantindo que as transações sejam executadas em ordem.

Além de reagir a conflitos, CAR-STM também é capaz de prevenir conflitos. Antes de iniciar uma transação, o escalonador é capaz de receber um apontador para um método que avalia a probabilidade de conflito. Este método retorna qual das transações em execução possui a maior probabilidade de conflito, baseado nesse resultado, o escalonador coloca a transação prestes a iniciar na fila de transações da *thread* que está executando a transação com maior probabilidade de conflito.

Infelizmente os autores não descrevem um possível método para calcular a probabilidade de conflito entre transações. Como mostrado nessa tese, uma heurística eficiente para determinar a probabilidade de conflitos entre transações desempenha um papel fundamental em um escalonador de transações.

CAR-STM foi implementado no RSTM [60] e utilizou o STMBench7 [38] para testá-lo e validá-lo. CAR-STM conseguiu significativos ganhos de desempenho quando comparado com a implementação original do RSTM, além disso, os resultados foram extremamente mais estáveis. Em cenários com leituras e escritas balanceadas, o *speedup* chega a 19x.

O trabalho desenvolvido nesta tese tem alguma semelhança com o CAR-STM, pois ambas as técnicas utilizam um número fixo de *threads* dedicadas às quais as transações são enviadas e executadas. Porém, CAR-STM se baseia em travas em nível de sistema e variáveis de condição para sincronizar as *threads* tradicionais com as *threads* transacionais. LUTS, por sua vez, é capaz de manter apenas *threads* transacionais, evitando o custo de sincronizações entre os diferentes tipos de *threads*.

3.3.4 Steal-on-abort

Ansari *et al.* [5] apresentaram uma técnica similar ao CAR-STM que também atua em tempo de execução, o nome de sua técnica é *Steal-on-abort*. Assim como no CAR-STM, a transação abortada é movida para uma fila da outra transação conflitante, evitando futuros conflitos. Diferentemente do CAR-STM, os autores investigaram diferentes es-

tratégias ao inserir uma transação na fila de outra *thread*, por exemplo, a transação pode ser inserida no início ou no final da fila.

Para medir o desempenho do *Steal-on-abort* foram usados três programas: lista ligada, árvore vermelha-preta e Vacation (pertencente ao Stamp). Apesar da estratégia de inserir a transação no início da fila ter se mostrado mais efetiva que as demais, nenhuma delas apresentou ganhos significativos em relação aos gerenciadores de contenção tradicionais (Polka e Priority).

3.3.5 M5-TM

Blake *et al.* [8] fizeram um estudo mostrando que gerenciadores de contenção baseados em atrasos de transação não conseguem bom desempenho no conjunto de aplicações do STAMP. Além disso, eles identificaram que pequenos grupos de transações formam regiões de alta contenção que culminam no baixo desempenho. Por outro lado, eles afirmam que essas regiões são formadas por pequenos conjuntos de conflitos, os quais ocorrem de uma maneira bastante previsível. Sendo assim, os autores propõem uma estratégia dinâmica de gerenciamento de contenção que usa um histórico para identificar quando essas regiões serão executadas e as evita através do reescalonamento das transações envolvidas. Apesar de o seu trabalho focar sistemas de memória transacional em hardware, o escalonador é implementado em software em nível de usuário.

O escalonador consiste de um algoritmo totalmente distribuído no qual cada processador executa em paralelo quando uma transação deseja iniciar. Primeiramente, cada processador pesquisa quais transações estão em execução em todo o sistema, note que esse "*snapshot*" das transações em execução não é necessariamente consistente com o estado corrente do sistema, já que não é usado nenhum tipo de sincronização nessa pesquisa, pois qualquer mecanismo de sincronização é computacionalmente proibitivo nesse caso. Em seguida a *thread* verifica qual a probabilidade de sua transação conflitar com alguma transação em execução, essa informação é armazenada em uma tabela global que contém a probabilidade de conflitos entre cada par de transações existente no programa. Com esse resultado o escalonador decide se a transação deve iniciar, atrasar seu início ou trocar por outra *thread*. No caso de uma troca de *threads*, a *thread* em execução chama o método `pthread_yield()`, o qual deixa o sistema operacional responsável por colocar outra *thread* em execução.

Para avaliar o desempenho do escalonador, um sistema de memória transacional em hardware muito semelhante ao LogTM foi implementado no simulador de sistemas M5 [7]. Em um processador de 16 núcleos, o M5-TM conseguiu, em média, um ganho de desempenho de 85%. Além disso, a contenção foi reduzida de 4-5x.

LUTS apresenta semelhanças com o M5-TM, LUTS também consulta o *snapshot* do

sistema para tomar decisões, mas ao contrário do M5-TM, LUTS não delega a tarefa de trocar a transação para o sistema operacional, o próprio LUTS é capaz de selecionar qual a melhor transação a ser executada e em nível de usuário fazer a troca da transação. Infelizmente não é possível comparar o desempenho dos dois sistemas, já que o M5-TM é voltado para MTH, enquanto que o LUTS foca em MTS.

3.3.6 Kernel based scheduler

Maldonado *et al.* [57] afirma que deixar o escalonamento de *threads* transacionais a cargo do sistema operacional causa muita imprecisão do sistema e a consequente queda de desempenho em cenários com alta contenção. Este trabalho é o primeiro a investigar técnicas de escalonamento em nível de *kernel* para reduzir essa imprecisão e os custos de sincronização. Os autores apresentam e comparam diferentes estratégias, tanto em nível de usuário quanto em nível de *kernel*.

Na abordagem tradicional a *thread* é bloqueada usando espera-ocupada (*spinlock*). Outra opção apresentada é o uso de chamadas de sistema na qual a *thread* fica bloqueada e associada a uma variável de condição, mas o uso de chamadas de sistema é extremamente custoso, o que torna essa solução inviável. Para desenvolver uma técnica eficaz, o primeiro desafio a ser vencido foi criar uma comunicação eficiente entre as operações transacionais e o *kernel* sem depender de chamadas de sistema.

Seu algoritmo, chamado `Ser-k`, utiliza uma região de memória compartilhada entre a aplicação e o sistema operacional, essa região armazena uma tabela com um descritor de transação para cada *thread* do sistema. À estrutura de *thread* do sistema operacional é adicionado um ponteiro para sua respectiva entrada na tabela compartilhada. Com esta infraestrutura a transação se comunica com o sistema operacional apenas adicionando seu estado (início, conflito, aborto, *commit*) à sua estrutura da tabela compartilhada. Essas informações capacitam o *kernel* a serializar as transações conflitantes e reduzir a contenção do sistema.

Eles também descrevem uma técnica que reduz a prioridade de uma *thread* para cenários onde não determinismo possa ocorrer. A ideia central é que uma transação com muito controle de fluxo irá tomar um caminho diferente quando ela reiniciar após um aborto, e sendo assim, ela pode não causar o mesmo conflito novamente. Nesse caso, os autores afirmam que serializar a transação é uma atitude muito drástica, e propõem apenas reduzir a prioridade da *thread* que está executando a transação que acabou de abortar. Além disso, um mecanismo para aumentar o quantum de tempo de uma *thread* é utilizado para reduzir conflitos que envolvem transações muito longas.

Como discutimos nessa tese, deixar a cargo do sistema operacional o escalonamento de transações não é eficiente. Isso acontece porque o sistema operacional não tem conhe-

cimento de que está executando uma transação e sendo assim, não consegue selecionar tarefas que não vão conflitar entre si. Além disso, a troca de contexto realizada pelo sistema operacional é um pouco mais custosa do que a realizada em nível de usuário. Fazer com que o sistema operacional tenha ciência das transações é uma solução viável, porém essa solução não apresenta boa portabilidade.

3.3.7 McRT-STM

McRT-STM [81] é um módulo de sistema memória transacional em software construído em conjunto com o McRT: um sistema de execução *multi-core* experimental. O McRT-STM possui um escalonador que utiliza escalonamento cooperativo, ou seja, a aplicação auxilia nas decisões tomadas pelo escalonador. O escalonador executa em nível de usuário, tornando a iteração entre o sistema de MTS e o escalonador muito eficiente, já que o mesmo é feito através de chamadas de função.

Cenários com mais *threads* em execução do que o número de núcleos de processamento disponíveis apresentam falso-paralelismo [25], pois a constante troca de contexto degrada o desempenho do sistema, limitando o paralelismo. Para evitar este cenário, o escalonador desenvolvido pelos autores possui filas de tarefas para cada processador, e as trocas de tarefas em execução são feitas em nível de usuário. A aplicação pode definir o número de filas de tarefa e qual política de escalonamento será usada pelo escalonador. Além disso, cada tarefa também é capaz de conceder seu tempo de execução para outra tarefa e se colocar em uma das filas para terminar sua execução mais tarde.

McRT-STM tem algumas semelhanças com o LUTS, ambos implementam um escalonador cooperativo e ambos evitam cenários com falso-paralelismo ao usar *threads* em nível de usuário, deixando escalonador do sistema operacional ignorante ao fato de quantas *threads* estão realmente em execução. Porém, o LUTS vai além, implementando heurísticas de escalonamento, nas quais a aplicação é capaz de solicitar a execução de uma transação específica.

A Tabela 3.1 apresenta um resumo comparativo dos escalonadores de transações e suas principais características.

3.3.8 Contribuições desse trabalho

Uma distinção chave em relação a trabalhos anteriores é que o LUTS, ao invés de serializar as transações, torna possível para o escalonador a escolha de outra transação para ser executada no lugar da transação que está fadada a abortar. Nossos resultados mostram que substituir uma transação fadada a abortar por outra transação com melhores probabilidades de sucesso gera melhor desempenho do que apenas serializar as transações.

Tabela 3.1: Resumo comparativo dos escalonadores de transações e sua principais características. O marcador X indica a presença da característica no escalonador de transações. A sigla "sw" significa software e a sigla "hw" significa hardware.

Escalonadores	Características							
	serializa transações	só ativa em alta contenção	tipo	*threads* em nível de usuário	*threads* em nível de sistema	histórico de conflitos entre transações	histórico de leituras e escritas	dinâmico
ATS	X	X	sw/hw					
Shrink	X	X	sw				X	
CAR-STM	X		sw		X			
Steal on Abort	X		sw					
M5-TM	X		hw		X	X		
Kernel Based	X		sw	X	X			
McRT-STM	X		sw	X				
LUTS			sw	X		X		X

Esse trabalho é o primeiro (circa 2012) a apresentar uma heurística dinâmica para prever o futuro de uma transação e selecionar uma melhor opção a ser executada. A heurística é considerada dinâmica, pois ela se adapta de acordo com as características do programa, como por exemplo, o tamanho da transação (identificamos que o tamanho das transações influencia diretamente no sucesso da heurística de escalonamento).

Capítulo 4

LUTS

Este capítulo apresenta em detalhes o LUTS (*Lightweight User-Level Transaction Scheduler*), um escalonador de transações em nível de usuário e de baixo custo. LUTS disponibiliza um conjunto de métodos para escalonar transações, possibilitando o desenvolvimento de heurísticas para prever conflitos e melhorar o desempenho de sistemas de MTS.

4.1 Visão Geral

O projeto do LUTS é embasado em duas características principais: (1) o número de *threads* em nível de sistema não deve exceder o número de núcleos de processamento disponíveis e (2) um escalonador de transações deve ser capaz de trazer benefícios a um sistema de MTS. A primeira característica visa reduzir problemas de falso-paralelismo, enquanto que a segunda característica é guiada pela ideia de que mecanismos mais robustos para detectar e evitar conflitos podem ser criados se mais detalhes do escalonamento forem expostos ao sistema de MT.

A Figura 4.1 mostra a visão geral de um sistema baseado no LUTS. O LUTS é responsável por oferecer um conjunto de operações de escalonamento e de gerenciamento de *threads* para a aplicação e para a biblioteca de memória transacional. Esses serviços são resumidos na Tabela 4.1, e serão apresentados com mais detalhes nas próximas seções.

4.1.1 Interface de Gerenciamento de *Threads*

O LUTS provê uma interface simples para o desenvolvimento de programas com múltiplos dados (SPMD) [1] [19]. Isso inclui uma rotina para criar um número de *threads* trabalhadoras (`luts_init`) e a capacidade de instruir essas *threads* a iniciarem sua execução com o mesmo programa (`luts_start`). Note que mecanismos de sincronização, como travas e

[1] do inglês *Single Program Multiple Data*

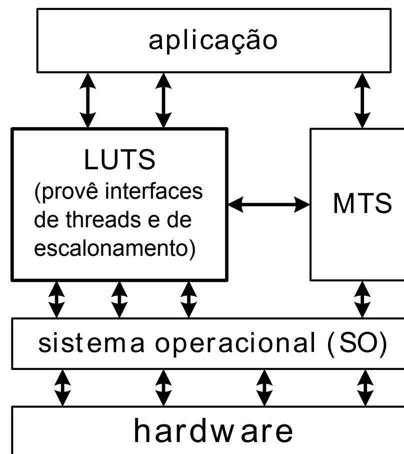


Figura 4.1: LUTS provê um conjunto de métodos para escalonar tarefas e gerenciar *threads*, esses métodos podem ser utilizados tanto pela aplicação quanto pela biblioteca de memória transacional.

variáveis de condição, não são disponibilizados, já que adotamos um modelo de transações implementado por uma biblioteca de MTS. Porém, há uma importante exceção: barreiras de sincronização. Dado que barreiras são comumente usadas em programação paralela, LUTS provê a rotina (`luts_barrier_wait`), responsável por bloquear o progresso das *threads* até que todas alcancem essa mesma barreira.

O LUTS não cria mais *threads* em nível de sistema do que o número de núcleos de processamento disponíveis na máquina. Em outras palavras, se `luts_init` for chamado com argumento 16 e a máquina possui apenas 8 núcleos de processamento, serão criadas apenas 8 *threads* em nível de sistema. É importante ressaltar que o gerenciamento de *threads* é totalmente transparente para a aplicação; do ponto de vista da aplicação as 16 *threads* foram criadas. Para lidar com esse cenário, o LUTS representa cada *thread* internamente usando Registros de Contexto em Execução (RCEs). Cada RCE encapsula o estado de uma das *threads* e o escalonador do LUTS fica responsável por designar um RCE para uma *thread* de sistema para que ele seja executado. Quando `luts_init` é invocado, LUTS cria o número necessário de RCEs e aloca o número ideal de *threads* de sistema através de chamadas de sistemas. Cada RCE é inserido em uma fila de contextos, podendo ser escolhido para execução posteriormente. O expedidor é responsável por selecionar um RCE da fila de contextos e mapeá-lo para uma *thread* de sistema, como mostrado na Figura 4.2. A execução paralela só começa quando a aplicação chamar o método `luts_start`.

A principal razão pela qual o LUTS adota esse esquema de gerenciamento de *threads* é para evitar potenciais problemas causados pelo falso-paralelismo. Em um sistema convencional, várias transações podem executar no mesmo núcleo de processamento, portanto,

Tabela 4.1: Resumo dos métodos de gerenciamento de *threads* e de escalonamento oferecidos pelo LUTS.

Interface de Gerenciamento de *Threads*	
Método	Descrição
luts_init(`num_threads`)	Inicia o sistema com o número de *threads* especificado
luts_start(`func, args`)	Após a chamada desse método, cada *thread* irá começar a executar a função `func` com os argumentos `args`
luts_barrier_wait()	Execução só progride depois que todas as *threads* atingirem essa barreira
luts_shutdown()	Limpa as estruturas utilizadas pelo LUTS e finaliza o processo
Interface de Escalonamento	
Método	Descrição
luts_yield()	O contexto corrente é inserido ao final da fila e o contexto do início da fila é colocado em execução
luts_getid(`ctxId`)	Retorna o identificador único do contexto em execução
luts_switch_from(`ctxId`)	O escalonador irá trocar o contexto corrente por outro contexto da fila que contenha um ID diferente do ID especificado por `ctxId`
luts_switch_to(`ctxId`)	O escalonador irá trocar o contexto corrente por outro contexto da fila que tenha um ID igual ao especificado por `ctxId`

o escalonador do sistema operacional pode substituir uma *thread* que esteja executando uma transação por outra *thread* que esteja executando outra transação. Ao fazer essa substituição no meio de uma transação, a probabilidade de um conflito da transação suspensa com outra transação aumenta drasticamente, limitando o desempenho do sistema como um todo. O projeto do LUTS visa executar no máximo uma transação por núcleo de processamento. Para atingir esse objetivo as seguintes ações são adotadas: (1) criar o número de *threads* de sistema igual ao número de núcleos de processamento disponíveis e (2) definir a afinidade de cada *thread* a um núcleo de processamento específico. Essas ações praticamente eliminam casos onde um único núcleo de processamento execute mais de uma *thread* transacional.

4.1.2 Interface de Escalonamento

Como mencionado anteriormente, cada RCE armazena o estado de uma única *thread*. Quando o número de RCEs é menor ou igual ao número de *threads* de sistema, o comportamento do sistema é similar ao comportamento de um sistema convencional: cada RCE é continuamente mapeado para uma única *thread* de sistema. Quando o número de RCEs é maior que o número de *threads* de sistema disponíveis, o expedidor seleciona um RCE e o mapeia para uma *thread* de sistema para ser executado. Quando um RCE é mapeado e começa a executar, só existem duas maneiras da sua correspondente *thread* de sistema ficar livre novamente: ou o trabalho é finalizado ou o RCE voluntariamente pede para ser substituído por outro RCE ao chamar um método provido pelo LUTS, por exemplo o

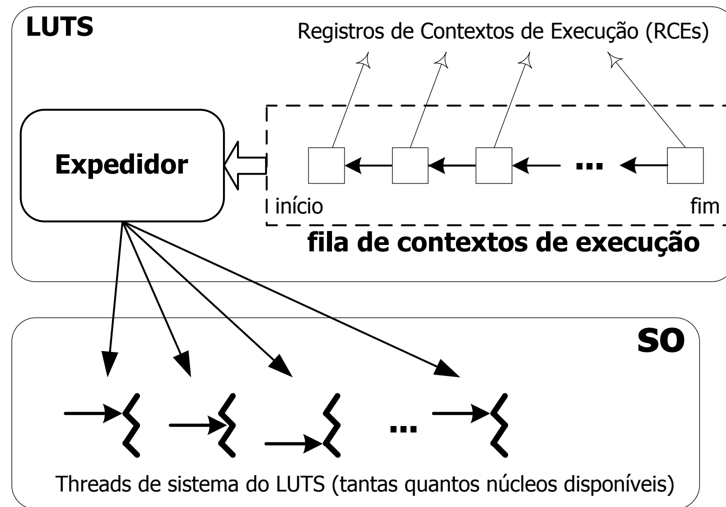


Figura 4.2: Mapeamento de *threads* do LUTS. O expedidor seleciona um RCE para ser executado e o mapeia para uma *thread* de sistema.

`luts_yield`. Dessa forma, o LUTS adota uma abordagem cooperativa de escalonamento.

Adotar uma abordagem cooperativa ao invés de uma abordagem baseada em preferências traz duas grandes vantagens no contexto desse trabalho. Primeiro, ela é muito eficiente e bastante simples de ser implementada, deixando a tarefa de portar o LUTS para diferentes bibliotecas de MTS extremamente simples. Segundo, a coordenação entre o escalonador e a biblioteca de MTS evita casos de falso-paralelismo e de preempção como discutido anteriormente. Dessa forma, o sistema não bloqueia uma *thread* que esteja executando uma transação e diminui a probabilidade de conflitos entre transações.

A rotina básica do escalonador é o `luts_yield`, que tem uma semântica muito similar à rotina `sched_yield` disponível em sistemas operacionais baseados no UNIX. Essa rotina força o contexto corrente a liberar a *thread* de sistema para que outro contexto seja executado. O contexto corrente é salvo e inserido ao final da fila de contextos mantida pelo LUTS, em seguida o contexto do início da fila é selecionado e mapeado pelo expedidor para a *thread* de sistema que acabou de ficar livre. O `luts_yield` também é utilizado internamente na implementação das barreiras de sincronização (`luts_barrier_wait`). Note que o suporte nativo a barreiras (por exemplo, o incluso na biblioteca `pthreads`) não pode ser utilizado, já que o número de RCEs pode ser maior que o número de *threads* de sistema. Dessa forma, um mecanismo de barreiras foi implementado com base no algoritmo *sense-reversing* proposto por Mellor-Crummey and Scott [63].

LUTS também permite que a biblioteca de MTS troque o contexto em execução por outro RCE ao chamar um dos outros dois métodos disponibilizados: `luts_switch_from` ou `luts_switch_to` (ver Tabela 4.1). Já que existe apenas uma transação por RCE,

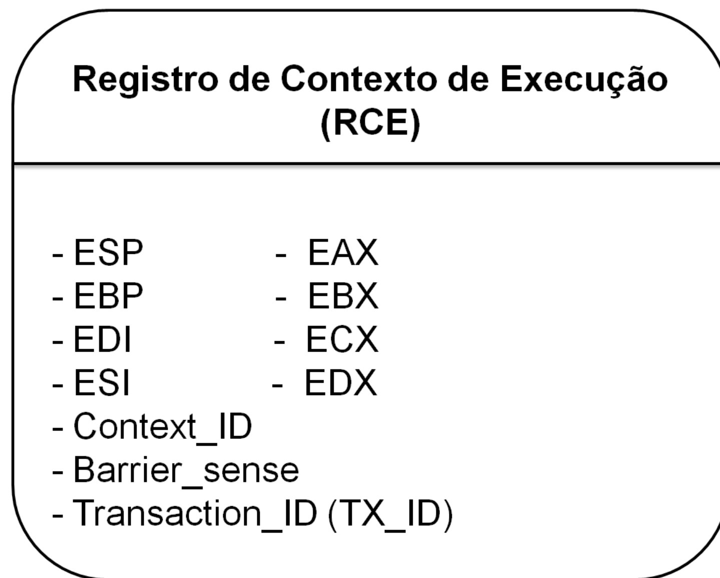


Figura 4.3: Informações contidas em um Registro de Contexto de Execução (RCE), necessárias para a troca de contexto realizada pelo LUTS.

esses métodos permitem que a biblioteca de MTS execute a transação que desejar. A transação desejada pode ser indicada por uma heurística capaz de indicar uma transação com baixa probabilidade de conflitos. Note que esses métodos são mais sofisticados do que simplesmente liberar a *thread* de sistema, pois com eles a biblioteca de STM tem mais controle sobre qual contexto deve ser executado no lugar daquele que acabou de liberar a sua *thread* de sistema. Para diminuir a contenção do sistema, uma heurística de prevenção de conflitos foi implementada utilizando esses métodos, mais detalhes podem ser encontrados na Seção 4.2.

Uma troca de contexto realizada pelo LUTS consiste em salvar o estado do processador em um RCE e mapear as informações de outro RCE no processador. As informações contidas em um RCE podem ser vistas na Figura 4.3. Os registradores de máquina são responsáveis por armazenar o contexto do processsador. `Context_ID` é usado para gerenciamento de interno do LUTS, `Barrier_sense` é usado caso a aplicação precise de sincronização entre *threads* e `Transaction_ID` diz qual transação o RCE está prestes a iniciar.

Deve ser ressaltado que a principal contribuição desse trabalho não é o escalonador em si, mas a sua aplicação no domínio de memória transacional e a consequente capacidade de desenvolver novas heurísticas pró-ativas para prever conflitos. O projeto do LUTS pode ser visto como uma simplificação do escalonador *Scheduler Activations* [4].

4.2 Uma Heurística Dinâmica para Prevenção de Conflitos

O desenvolvimento de heurísticas pró-ativas para a prevenção de conflitos envolve o compromisso entre dois aspectos que se opõem. Se por um lado, a heurística deve ser o mais **precisa** possível, de forma que na maioria das vezes sua previsão é correta, por outro, o ***overhead*** em tempo de execução deve ser baixo o suficiente de modo a não afetar o desempenho do sistema ou anular os ganhos advindos da heurística.

Para prever conflitos com precisão, uma heurística necessita de muita informação do sistema como um todo e consequentemente, é necessário muito tempo para coletar e analisar toda essa quantidade de dados. Contudo, se uma transação é longa o suficiente para amortizar esse custo, então uma heurística mais elaborada pode apresentar um bom custo benefício, já que evitar conflitos dessas transações fará com que menos transações longas sejam abortadas e re-executadas.

Baseado na observação do parágrafo anterior, esse trabalho apresenta uma heurística dinâmica que foca em ter o melhor dos dois mundos. Uma heurística com baixo custo e menos precisa para transações curtas; e uma heurística muito precisa com um custo moderado para transações longas. A heurística dinâmica escolhe, em tempo de execução, a melhor estratégia a ser utilizada com base na execução das últimas 100 transações finalizadas com sucesso: se a média do tempo de execução das transações for maior que um valor pré-determinado, então a heurística com alta precisão é utilizada, caso contrário a heurística de baixo custo é a escolhida. Foram realizados testes com diversos valores (10, 25, 50, 100, 150, 200, 500 e 1000) para definir uma janela de transações ideal. O valor de 100 transações foi escolhido porque valores menores não apresentaram ganhos, enquanto que valores maiores não foram capazes de identificar diferentes fases do programa com exatidão. Isso se aplica principalmente aos programas do STAMP, que apresentam diferentes fases. Nas diferentes configurações do STMBench7 não ocorrem mudanças de fases.

Para simplificar a implementação e evitar custos extras, apenas um dos núcleos de processamento fica responsável por medir o tempo de execução das transações. Em outras palavras, apenas as últimas 100 transações executadas em um núcleo de processamento específico são usadas para calcular a média de tempo. Para coletar essa informação de tempo são utilizados contadores de desempenho em hardware, disponíveis em praticamente todos os multiprocessadores modernos, por exemplo, a Intel disponibiliza em seus processadores a instrução `rdtsc` para contar a quantidade de ciclos do processador [94].

Uma questão remanescente é saber como determinar o limite de tempo, ou seja, como distinguir entre uma transação longa e uma transação curta. Esse valor foi definido empiricamente ao executar os programas do STMBench7 [38] e STAMP [65], exceto o

Bayes. Os valores considerados incluem 10 mil, 50 mil, 100 mil, 500 mil e 1 milhão de ciclos. Para os experimentos apresentados no Capítulo 5 foi escolhido o valor de 100 mil ciclos, já que esse valor apresentou os melhores resultados.

Nas próximas duas seções, as heurísticas utilizadas para transações curtas (Seção 4.2.1) e para transações longas (Seção 4.2.2) são descritas. Ambas heurísticas fazem uso da capacidade do LUTS de trocar a execução de um contexto por outro indicado pela heurística, evitando assim a serialização da execução das transações.

4.2.1 CILUTS - Transações Curtas

O ideal seria que heurísticas nessa classe tivessem custo próximo de zero e uma precisão de previsão de conflitos razoável. A heurística utilizada nesse trabalho (CILUTS) é baseada no trabalho de Yoo e Lee, conhecido como *Adaptive Transaction Scheduling* (ATS) [101].

Nessa técnica, cada transação mantém internamente uma variável descrevendo sua probabilidade de conflito. Para quantificar essa variável, o conceito de Intensidade de Contenção (IC) dado pela Equação 4.1 é utilizado.

$$IC_n = \alpha \times IC_{n-1} + (1 - \alpha) \times CA \tag{4.1}$$

Inicialmente, IC tem o valor zero, e a equação é avaliada a cada término de transação: final com sucesso ou aborto. A Contenção Atual (CA) é definida como zero em um *commit* e como 1 em um aborto. O valor de α determina se deve ser dado mais valor ao histórico recente das transações ou se o resultado das transações mais antigas deve ter maior peso. Após testar diferentes valores de α (0,3, 0,5 e 0,75), 0,75 mostrou ser o valor mais eficiente para as aplicações utilizadas nesse trabalho, dando maior prioridade para o histórico mais remoto das transações.

Quando uma transação está prestes a iniciar sua execução, o algoritmo verifica se sua IC está acima de um determinado valor. Se for este o caso, uma ação pode ser tomada a fim de evitar que uma transação com alta probabilidade de conflito (e consequente aborto) inicie. Enquanto que a técnica ATS apenas serializa a execução da transação em questão, deixando o processador em espera ocupada, o LUTS acha mais apropriado substituir a transação em questão por outra transação. Para realizar essa substituição o LUTS chama o método `luts_switch_from`, o qual vai substituir a transação corrente (que ainda não iniciou) por uma transação com um ID diferente.

Para distinguir as transações, cada transação é identificada pelo endereço de sua primeira instrução. A eficácia dessa heurística é sensível à quantidade de diferentes transações do programa, ou seja, a quantidade de transações que correspondam a diferentes trechos de código fonte. No pior cenário, `luts_switch_from` não irá encontrar uma transação com ID diferente e irá apenas colocar em execução a transação corrente.

No melhor caso, o LUTS conseguirá colocar em execução outra transação e não gerará nenhum conflito.

4.2.2 HASHLUTS - Transações Longas

O custo computacional de uma heurística ocorre a cada operação de início, fim e aborto de uma transação, e esse custo é fixo em relação à duração da transação. Dessa forma, para transações longas, o custo de uma heurística é proporcionalmente menor do que para uma transação curta. Sendo assim, heurísticas usadas nesse cenário tendem a ser mais sofisticadas. Além disso, evitar abortos de transações longas traz mais benefícios do que evitar abortos de transações pequenas, já que o impacto no tempo de execução é maior nas transações longas (especialmente para sistemas de STM que utilizam técnicas de detecção de conflitos *lazy*).

Basicamente, o objetivo da heurística apresentada para esse cenário (HASHLUTS) é prever qual a melhor transação a ser executada dado um conjunto de transações ativas (que estão em execução). Uma transação é considerada a melhor candidata para execução quando sua probabilidade de conflitos é a menor entre as transações disponíveis para execução. Para ter essa informação nós armazenamos o histórico de conflitos de cada transação em uma tabela.

Para deixar claros os principais conceitos da HASHLUTS nós apresentamos um exemplo detalhado. Para acompanhar o exemplo, o leitor deve estar atento à Figura 4.4 e ao pseudocódigo da Figura 4.5.

A HASHLUTS utiliza três estruturas globais para armazenar os metadados necessários para a previsão de conflitos: um vetor de transações ativas (`activeTx`), uma tabela de conflitos (`conflictTable`) e um vetor que serve como um atalho para a melhor transação (`bestTx`). As estruturas são ilustradas por ❶, ❷ e ❸ na Figura 4.4, e pelas linhas 1 e 2 no pseudocódigo. O vetor `activeTx` mantém, para cada núcleo de processamento, o identificador (ID) da transação que está em execução naquele núcleo. A Figura 4.4 assume um sistema com 4 núcleos e 3 diferentes transações, representadas pelos IDs 1, 2 e 3. Se um núcleo de processamento não estiver executando uma transação, ou seja, um código não transacional, nós usamos o ID -1. O conjunto de transações ativas no sistema, ilustradas na Figura 4.4, é dado pelo vetor $\{-1, 1, 3, 2\}$ ❶. Portanto, o núcleo 0 não está executando qualquer transação e os núcleos 1, 2 e 3 estão executando as transações 1, 3 e 2 respectivamente.

Agora imagine que o núcleo 0 está prestes a iniciar uma transação. Para selecionar a melhor transação a ser iniciada o escalonador consulta a estrutura `conflictTable` ❷. Cada linha dessa tabela identifica um conjunto de transações; cada coluna quantifica a intensidade de conflito (ic) ao iniciar uma transação específica quando o respectivo

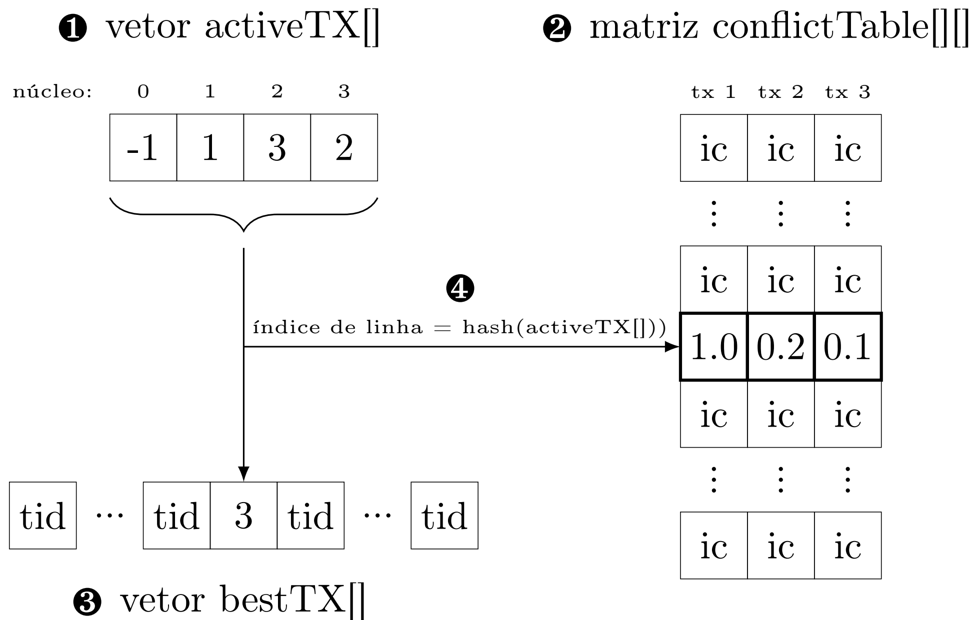


Figura 4.4: Um exemplo ilustrando os metadados utilizados pela heurística e como a melhor transação a ser executada é selecionada dado um conjunto de transações ativas.

conjunto de transações está ativo. Uma função *hash* é usada para mapear cada conjunto de transações ativas para um índice que indica uma linha da tabela, esse mapeamento é ilustrado por ❹ na Figura 4.4. Para esse exemplo específico, as intensidades de conflito ao iniciar as transações 1, 2 e 3 quando o conjunto de transações ativas é $\{-1, 1, 3, 2\}$ é de $1,0$, $0,2$ e $0,1$ respectivamente. Portanto, concluímos que o núcleo 0 deveria escolher a transação 3, já que ela possui a menor intensidade de conflito.

Note que consultar a tabela `conflictTable` todas as vezes que uma transação está prestes a iniciar pode adicionar um custo computacional indesejado ao sistema. Para evitar esse custo, nós utilizamos um vetor que resume quais as melhores transações a serem executadas para cada conjunto de transações ativas. O mesmo índice retornado pela função *hash* é utilizado para acessar o vetor `bestTx` ❸, o qual armazena o ID (tid) da melhor transação a ser executada, nesse caso a transação 3 (como esse vetor é atualizado será discutido logo adiante). Depois que uma transação é escolhida, o método `luts_switch_to` é chamado e o vetor de transações ativas `activeTx` é atualizado (linhas 11 a 15 da Figura 4.5 descrevem esse processo em pseudocódigo na operação *start*).

Agora suponha que a transação selecionada pelo núcleo 0 abortou após um conflito ter sido detectado (linhas 17 a 29 da Figura 4.5). Inicialmente, `activeTx` é atualizado para representar que nenhuma transação está em execução no respectivo núcleo (linha 18). Já que houve um conflito, é preciso aumentar a intensidade de conflito ao executar essa transação com o conjunto de transações ativas em questão (linhas 19 e 20). Note

```
1 double conflictTable[][];
2 int bestTx[];
3
4 upon stm_init
5     resetBestTx();
6     resetCT();
7   for each core i
8     activeTx[i] = INVALID;
9
10 upon start
11   int line_index = hash(activeTx);
12   int tx_id = bestTx[line_index];
13   luts_switch_to_id(tx_id);
14   updateActiveTx(thisCore, tx_id);
15
16 upon abort
17   updateActiveTx(thisCore, INVALID);
18   int line_index = hash(activeTx);
19   increaseIntCT(line_index, tx_id);
20   if(bestTx[line_index] == tx_id){
21     for each transaction tx {
22       if(conflictTable[line_index][tx] <
23           conflictTable[line_index][tx_id])
24       {
25         bestTx[line_index] = tx;
26       }
27     }
28   }
29
30 upon commit
31   updateActiveTx(thisCore, INVALID);
32   int line_index = hash(activeTx);
33   decreaseIntCT(line_index, tx_id);
34   if(conflictTable[line_index][tx_id] <
35       conflictTable[line_index][bestTx[line_index]])
36   {
37       bestTx[line_index] = tx_id;
38   }
```

Figura 4.5: Pseudocódigo da heurística de previsão de conflitos com alta precisão e custo moderado.

que o vetor de transações ativas não é necessariamente o mesmo de quando a transação iniciou. Assumindo que seja, então a antiga intensidade $(0, 1)$ de conflito precisa ser aumentada. Para esse exemplo, por fins didáticos, assumimos que a constante $0, 2$ seja sempre adicionada ao valor anterior (nossa implementação utiliza $0, 1$). Nesse caso, a nova intensidade de conflito da transação 3 deve ser aumentada para $0, 3$. Voltando à Figura 4.4, temos que a intensidade de conflito das transações 1 e 2 são $1, 0$ e $0, 2$ respectivamente. Portanto, o vetor `bestTx` precisa ser atualizado para refletir esse novo resultado, onde a transação 2 agora tem menor intensidade de conflito do que a transação 3. Essa ação pode ser vista no pseudocódigo nas linhas 21 a 29. Precisamos verificar se existe uma melhor transação apenas se a transação abortada era a transação indicada por `bestTx` (line 21). Nesse caso, é preciso checar cada coluna da tabela `conflictTable` para encontrar qual a

nova melhor transação (linhas 22 a 28).

Se ao invés de abortar, a transação finalizar com sucesso, é preciso diminuir sua intensidade de conflito (linhas 31 a 39 do pseudocódigo). Essa operação é similar à operação de aborto, exceto que não é preciso checar cada uma das colunas da tabela `conflictTable` para encontrar uma transação melhor. Durante um fim com sucesso de uma transação, a única maneira da melhor transação em `bestTx` ser alterada é se a transação que acabou de finalizar for diferente da transação em `bestTx` e, como consequência do final com sucesso, sua intensidade de conflito ficar menor que a intensidade de conflito da transação em `bestTx` (linha 34), se for este o caso, é feita a atualização de `bestTx` (linhas 35 a 39). Por exemplo, se a transação 2 fosse escolhida para execução ao invés da transação 3 (assumindo que não havia uma transação 3 disponível), então, na fase de finalização, a intensidade de conflito da transação 2 seria atualizada para $0,0$ e `bestTx` passaria a ter a transação 2 como a melhor transação a ser executada quando o conjunto de transações ativas fosse $\{-1, 1, 3, 2\}$.

Contrária à versão de transações curtas, essa heurística é mais elaborada e consequentemente mais pesada, tendo um custo computacional maior. Enquanto que para transações curtas usamos apenas uma variável global para armazenar a intensidade de contenção, para transações longas mantemos três estruturas de dados globais para prever qual a melhor transação a ser executada dado um conjunto de transações ativas no sistema. É importante notar que os acessos às estruturas `activeTx`, `bestTx` e `conflictTable` não são explicitamente sincronizados, portanto é possível ocorrer condições de corrida. Consideramos essas condições de corrida benéficas, já que elas podem gerar algumas imprecisões na heurística de previsão de conflitos, mas nunca geram um erro de execução. Nesse contexto, é preferível conviver com alguma imprecisão a pagar o altíssimo custo da sincronização de tarefas. Uma abordagem similar foi usada em um contexto parecido por Blake *et al.* [8].

Capítulo 5

Resultados Experimentais

Esse capítulo avalia o desempenho do LUTS e da heurística dinâmica de previsão de conflitos. Também é feita uma comparação da nossa heurística com outros trabalhos publicados que compartilham o mesmo objetivo.

Os experimentos foram realizados em uma mesma máquina com 4 processadores Intel Xeon E7-4860 2,27GHz (40 núcleos no total, com *hyper-threading* desligado) com 24MB de memória cache e 256GB de memória RAM. O sistema operacional da máquina é uma típica distribuição Red Hat GNU/Linux com o *kernel* versão 2.6.18-194. Todas as aplicações e bibliotecas utilizadas foram compiladas com o GCC 4.5.1. A avaliação faz uso dos conjuntos de aplicações STMBench7 [38] e STAMP 0.9.10 [65].

Para comparação de desempenho foram usadas quatro configurações. Para deixar a comparação o mais justa possível, todas as configurações utilizam a mesma biblioteca de MTS. As quatro configurações são:

- `Baseline`: a implementação original da biblioteca TinySTM (versão 1.0.0). Para os experimentos, a TinySTM foi configurada com a estratégia de detecção de conflitos *eager* e com o gerenciador de contenção `CM_SUICIDE`, que após um aborto reinicia a execução da transação imediatamente.

- `ATS`: uma implementação do escalonador de transações adaptável proposto por Yoo e Lee [101], o qual emprega uma única fila para todas as transações. Antes de executar os experimentos apresentados nessa tese, foi realizado um estudo de sensibilidade na variável α com valores $0,3$, $0,5$ e $0,75$, e no limite de contenção com valores $0,3$, $0,5$ e $0,7$. A combinação de $0,75$ para α e $0,7$ para o limite de contenção apresentou a melhor média de resultados para todas as aplicações e portanto foi utilizada neste experimento.

- `Shrink`: uma implementação do escalonador Shrink proposto por Dragojevic *et al.* [28]. O código fonte para a versão 0.9.5 da TinySTM foi retirado da página web

Tabela 5.1: configuração do conjunto de aplicações STMBench7.

Atributo	Valor
Travessias longas	falso
Tipo de carga de trabalho	dominada por leituras
Duração	5000ms
Tamanho	Pequeno, Médio, Grande e Enorme

dos autores [1] e adaptado para a versão 1.0.0. Os parâmetros do código não foram alterados: succ_threshold = 0.5, locality_window = 4, confidence_threshold = 3, `c1` = 3, `c2` = 2, `c3` = 1.

- `LUTS-dyn`: O escalonador proposto nessa tese em conjunto com a heurística dinâmica discutida na Seção 4.2. Lembrando que 100 mil ciclos de máquina são usados para distinguir uma transação curta de uma transação longa. Para transações curtas adotamos $\alpha = 0,75$ e limite de contenção $= 0,5$.

A configuração utilizada para o conjunto de aplicações do STMBench7 está listada na Tabela 5.1. O STMBench7 consiste de uma única aplicação que executa uma variedade de transações (existem 48 diferentes tipos de transações) sobre uma estrutura de dados. É possível definir diferentes parâmetros para cada execução do programa. Dentre eles, é possível escolher o tamanho da estrutura de dados, se é permitido transações que atravessam toda a estrutura de dados, a proporção de leituras e escritas em cada transação e por quanto tempo a aplicação irá executar. Ao final da sua execução é medida a quantidade de transações executadas com sucesso. Portanto, sua a eficiência é medida por transações bem sucedidas por segundo.

O STAMP é composto por 8 aplicações. Elas são: (1) Bayes, implementa um algoritmo para aprendizagem de redes bayesianas; (2) Genome, a partir de um grande número de segmentos de DNA, ela os combina para reconstruir o genoma original; (3) Intruder, faz o reconhecimento de padrões para identificar ameaças em uma rede; (4) Kmeans, esse algoritmo agrupa objetos de um espaço N-dimencional em K grupos; (5) Labyrinth, implementa uma variante do algoritmo de Lee [99], a estrutura principal é uma grade tridimencional, e cada *thread* tenta formar um caminho de um ponto de início a um ponto de fim conectando diferentes pontos da grade; (6) SSCA2, é formado por quatro núcleos que realizam operações sobre um grande multigrafo direcionado e com pesos nas arestas; (7) Vacation, emula um sistema de reserva de passagens; e (8) Yada, implementa o algoritmo de Ruppert [80] para o refinamento de malhas de Delaunay. Para as aplicações do STAMP foram usadas as configurações e conjuntos de dados de entrada recomendados em [65]. Os resultados para a aplicação Bayes foram omitidos, já que essa aplicação

[1] http://lpd.epfl.ch/site/research/tmeval

não possui um comportamento reproduzível e possui uma variância muito grande. Este comportamento já foi relatado na literatura [15].

Todos os resultados apresentados nesse capítulo correspondem à média de 50 execuções. Altman e Bland [9] mostram que intervalos de confiança são uma alternativa rigorosa para testes de hipótese. Seguindo essa linha de pesquisa, os gráficos de *speedup* da Seção 5.2 também incluem o intervalo de confiança de 95%. Em todos os experimentos o número de *threads* varia de 1 a 128.

5.1 A importância de uma heurística

O LUTS limita o número de *threads* criadas ao número de núcleos da máquina com o intuito de evitar conflitos resultantes de trocas de contextos. Portanto, é esperada uma melhora do desempenho do sistema mesmo quando as rotinas de escalonamento do LUTS não são utilizadas pela biblioteca de MTS. Um questionamento natural é se a abordagem básica é suficiente ou se uma heurística para previsão de conflitos é capaz de melhorar o desempenho do sistema ainda mais.

Para esclarecer esse aspecto, essa seção apresenta uma comparação entre o LUTS sem escalonamento de transações (`luts_switch_from` e `luts_switch_to` não são usados), referenciado como `LUTS-pure`, e a heurística `LUTS-dyn` discutida na Seção 4.2. As Figuras 5.1 e 5.2 mostram o *speedup* do `LUTS-dyn` em relação ao `LUTS-pure` para os conjuntos de aplicações STAMP e STMBench7 respectivamente.

Em um cenário ideal, a heurística construída sob o LUTS irá gerar ganhos de desempenho quando ambas a quantidade e a diversidade de transações do sistema forem grandes o suficiente. Portanto, esperamos ganhos em cenários com mais *threads* do que o número de núcleos (64 e 128 *threads*), já que esses cenários terão muitas transações disponíveis a serem escolhidas pelo escalonador. A Figura. 5.1 mostra que no STMBench7 esse cenário ocorre com exatidão. Para as configurações Big e Huge, há um ganho de desempenho assim que o número de *threads* atinge 32. Esse comportamento ocorre devido a dois importantes fatores: (i) esse conjunto de aplicações apresenta boa diversidade de transações (em torno de 40 diferentes transações podem existir no sistema); (ii) a duração das transações é maior nas configurações Big e Huge (já que a duração das transações aumenta quando o tamanho da estrutura de dados fica maior). Ainda mais importante, praticamente não há *overhead* quando o número de *threads* é 1, 2, 4, ou 8.

Para o conjunto de aplicações do STAMP, os resultados mostrados na Figura 5.2 são diversos. A Tabela 5.2 ajuda a entender esses resultados; para cada aplicação ela lista: (1) a duração da transação, o que indica qual heurística a `LUTS-dyn` adota a maior parte do tempo, ao relatar `médio` indicamos que tanto transações curtas quanto transações longas existem na aplicação; e (2) o nível de contenção, que corresponde à razão entre o número

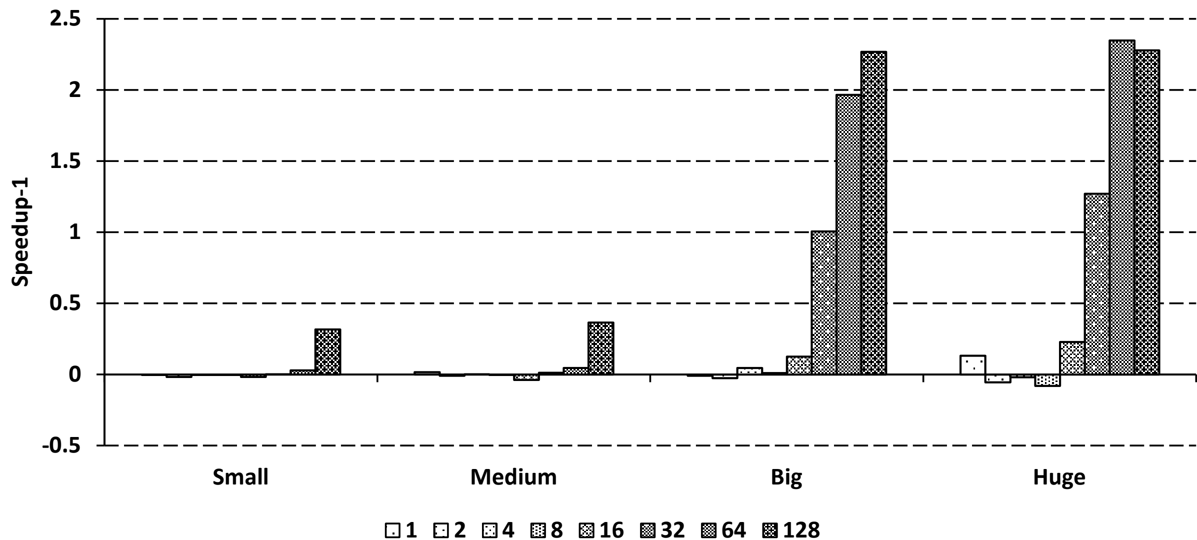


Figura 5.1: Speedup da configuração `LUTS-dyn` em relação à `LUTS-pure` para o conjunto de aplicações STMBench7 (valores positivos são melhores).

de transações abortadas e o número de transações finalizadas com sucesso. Essa razão foi medida ao usar o `LUTS-pure`. Também é muito importante notar que as aplicações do STAMP tendem a ter pouquíssimas transações diferentes ativas ao mesmo tempo. No geral, esse número não passa de três.

As aplicações do STAMP podem ser divididas em duas classes principais. A primeira é composta por Genome, SSCA2, Vacation e Yada. Como pode ser visto na Figura 5.2, a heurística `LUTS-dyn` não conseguiu apresentar melhoras em relação ao `LUTS-pure`. Mais especificamente, as três primeiras aplicações tem transações curtas e nível de contenção muito baixo (abaixo de 1), um cenário que não favorece a heurística `LUTS-dyn`. Por exemplo, na aplicação SSCA2, as transações são extremamente curtas (em torno de uma leitura transacional e duas escritas transacionais apenas), o que amplifica o *overhead* da heurística. A outra aplicação, Yada, é composta tanto por transações curtas quanto por transações longas, adicionando um custo extra à `LUTS-dyn` para realizar a troca entre as heurísticas de transações curtas e longas. Há dois importantes fatores para o *overhead* observado na aplicação Yada: (1) do total de 5 transações ativas, 3 são extremamente curtas (normalmente contém apenas uma leitura ou escrita transacional). Este é o pior cenário para a `LUTS-dyn`, já que o *overhead* de invocar os métodos de escalonamento providos pela interface do LUTS não é nada amortizado pelo tempo de duração da transação; e (2) o nível de contenção é muito alto (chegando a 250!). Portanto, o *overhead* da heurística é amplificado. Porém, é possível notar uma melhora com 128 *threads*. Nós suspeitamos que essa melhora ocorra nesse caso em particular porque existe uma grande quantidade de transações na fila a serem escolhidas pela `LUTS-dyn`, proporcionando muitas opções de

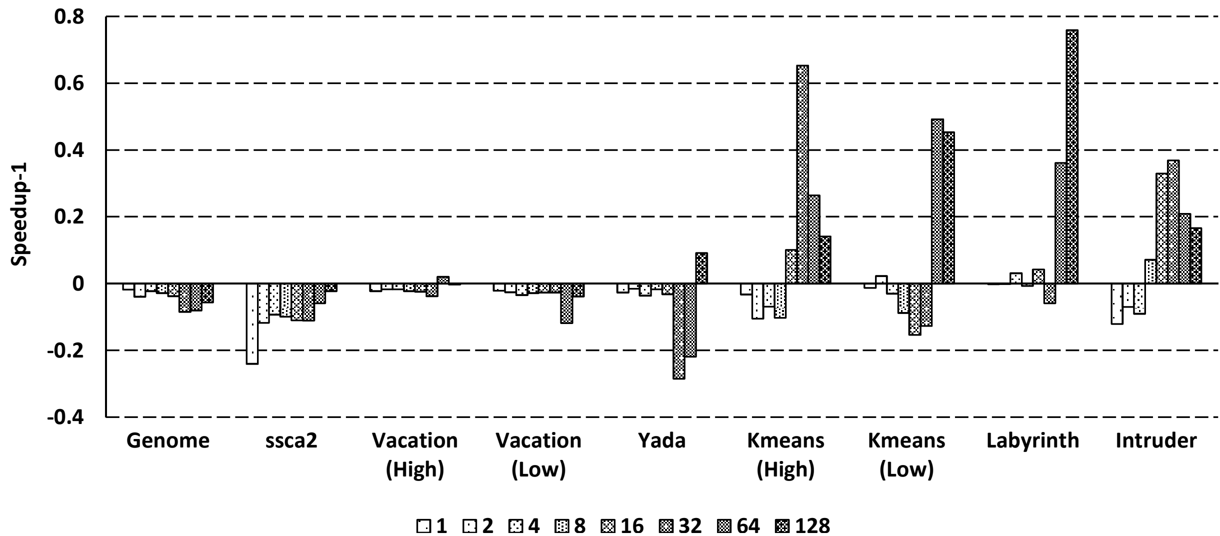


Figura 5.2: Speedup do `LUTS-dyn` em relação ao `LUTS-pure` para o conjunto de aplicações do STAMP (valores positivos são melhores).

Tabela 5.2: Duração das transações e nível de contenção (abortos/sucessos) para o conjunto de aplicações do STAMP.

Aplicação	Duração	Contenção (`LUTS-pure`)						
		2	4	8	16	32	64	128
Genome	curta	0.002	0.005	0.011	0.027	0.118	0.174	0.602
SSCA2	curta	4.5E-6	1.3E-5	3.7E-5	9.1E-5	17.5E-5	0.039	0.048
Vacation-high	curta	0.002	0.008	0.021	0.039	0.129	0.51	0.718
Vacation-low	curta	0.001	0.002	0.005	0.011	0.036	0.294	0.334
Yada	média	12.477	21.598	29.15	47.281	115.016	261.2	251.9
Kmeans-high	curta	0.715	1.695	5.066	12.909	25.819	83.582	88.002
Kmeans-low	curta	0.107	0.441	0.891	1.212	8.914	24.549	23.033
Labyrinth	longa	0.019	0.043	0.091	0.166	0.253	0.288	0.25
Intruder	curta	0.713	1.878	4.873	9.026	16.411	21.748	20.739

escalonamento.

A outra classe de programas consiste do Kmeans (high e low), Labyrinth e Intruder. Podemos observar uma importante melhora quando o número de *threads* é igual ou maior do que 32. Para o Kmeans e o Intruder, o nível de contenção trás várias oportunidades à `LUTS-dyn`. As transações curtas desses dois programas não são tão curtas quanto no SSCA2 e, dado um nível de contenção ideal, permite à `LUTS-dyn` melhorar o desempenho em até 60% (Kmeans-high com 32 *threads*). Note que a `LUTS-dyn` também melhora o desempenho do Labyrinth, mesmo ele tendo baixa contenção. A aplicação Labyrinth usa transações muito longas, beneficiando bastante heurísticas como a `LUTS-dyn` que visam evitar abortos de transações.

Já que descrevemos em detalhe a relação entre `LUTS-pure` e `LUTS-dyn` nessa seção, a fim de simplificar os gráficos, iremos omitir o resultado do LUTS sem o uso das rotinas

providas pela interface de escalonamento para as próximas seções.

5.2 Speedup

Nesta seção comparamos o *speedup* das duas melhores estratégias conhecidas para escalonamento de transações (`ATS` e `Shrink`) em MTS, descritas na Seção 3.3, com a `LUTS-dyn`, proposta nesta tese. Todos os resultados apresentados são normalizados em relação à `Baseline` com uma única *thread*.

Para o conjunto de aplicações STAMP, mostramos a aplicação Vacation apenas com a configuração `High`, já que os resultados para a outra variação (`Low`) são praticamente idênticos. A Figura 5.3 mostra o resultado para o STAMP. Primeiro, considere o cenário no qual o sistema tem pouca carga (número de *threads* igual ou menor do que 32). Podemos verificar que o `LUTS-dyn` não adiciona muito *overhead*, exceto para as aplicações Genome, Kmeans e Yada. Lembrando que, como discutido em seções anteriores, para essas aplicações, o `LUTS-dyn` também é inferior ao `LUTS-pure`. Para as aplicações Labyrinth, Vacation e SSCA2 o *overhead* é ínfimo. Podemos observar bons ganhos de desempenho no Intruder e Kmeans (High) com 16 e 32 *threads*. Olhando para as outras abordagens (`ATS` e `Shrink`), podemos verificar que elas também não conseguem melhor desempenho do que a `Baseline`, exceto para o Intruder e Kmeans (High).

Considere agora o cenário com muita carga (64 e 128 *threads*) onde podemos ter falso-paralelismo. Observamos que o `LUTS-dyn` melhorou o desempenho em relação ao `Baseline` em todas as aplicações, exceto o SSCA2. Como discutido anteriormente, transações no SSCA2 são extremamente curtas, impossibilitando qualquer melhora de desempenho. Note também que essa aplicação não escala com o número de *threads*. Nossa heurística apresentou o melhor resultado entre todas as técnicas comparadas para a aplicação Labyrinth, alcançando um *speedup* de 8,5x com 128 *threads*. Para o restante das aplicações, `LUTS-dyn` teve resultado semelhante ao `ATS` e `Shrink`. Apenas para o Genome (com 64 *threads*) notamos uma diferença considerável, devido principalmente à baixa contenção e às transação curtas presentes no Genome.

Devido à baixa diversidade de transações presentes no STAMP, esse conjunto de programas não consegue obter o melhor benefício que o LUTS consegue oferecer. Para que o `LUTS-dyn` seja eficiente, é preciso existir um grande número de transações ativas, de modo que o escalonador possa escolher a melhor transação a ser executada a cada momento. Esse cenário surge quando o sistema está com muita carga, mas mesmo assim, o STAMP não apresenta uma diversidade de transações suficiente. Normalmente, apenas três diferentes transações estão ativas ao mesmo tempo em cada aplicação.

Para a próxima série de experimentos optamos pelo conjunto de aplicações do STMBench7. Uma diferença chave entre o STAMP e o STMBench7 é a grande diversidade

de transações apresentada pelo último; em torno de 40 diferentes transações podem estar ativas concorrentemente. Sendo assim, é esperado que o `LUTS-dyn` apresente ganhos de desempenho substanciais nessas aplicações. A Figura 5.4 mostra que nossas expectativas estavam corretas. Foram usadas quatro configurações para a estrutura de dados utilizada no STMBench7: Small, Medium, Big e Huge. Ao aumentar o tamanho da estrutura de dados também aumentamos a duração das transações, forçando que existam transações longas.

Como os resultados mostram, `LUTS-dyn` foi, ou melhor, ou no mesmo nível do `Baseline`. Nossa heurística conseguiu *speedups* de até 10x (small), na qual o `ATS` conseguiu apenas 4x. Para estruturas de dados ainda maiores (Big e Huge) os ganhos são ainda mais evidentes, já que o `LUTS-dyn` mostra melhores resultados que o `Baseline` assim que o número de *threads* chega a 16. Transações são longas nas configurações Big e Huge, amortizando o *overhead* da heurística. Além disso, evitar o aborto de uma transação longa é mais efetivo do que evitar o aborto de uma transação curta, já que o aborto de uma transação longa desperdiça mais tempo de processamento. Nós acreditamos que esse aspecto também contribuiu para a melhora do `ATS` e `Shrink` na configuração Huge quando comparados com o Small, Medium e Big. Mas de uma maneira geral, o `ATS` e o `Shrink` não obtiveram bons resultados nos experimentos utilizando o STMBench7.

Resumindo, os resultados mostram que uma heurística para prever e evitar conflitos traz benefícios significativos para um o escalonador de transações. Além disso, vimos que a heurística `LUTS-dyn` adiciona muito pouco overhead ao sistema de MTS. Dessa forma, execuções com poucas *threads* (1 a 16) em execução não apresentam piora em relação à implementação base do sistema de MTS. Para cenários com mais *threads* em execução (32 a 128), podemos observar que o LUTS consegue ótimos resultados quando há uma boa diversidade de transações no sistema. Outro fator que beneficia o LUTS é a duração das transações, o LUTS consegue resultados ainda melhores em programas com transações mais longas.

5.3 `LUTS-dyn` com SwissTM

Esta seção tem dois objetivos. Primeiro, mostrar o quão fácil é adaptar o LUTS e a heurística `LUTS-dyn` para outra biblioteca de MTS baseada em palavras e descrever o processo de portar ambos para a SwissTM [27]. Segundo, quantificar o *speedup* obtido com esta rápida biblioteca de MTS e comparar com os resultados obtidos com a TinySTM. Como a SwissTM é reconhecidamente mais eficiente do que a TinySTM, estamos interessados em verificar se o *overhead* inserido pela nossa heurística será relativamente maior na SwissTM.

O primeiro passo para portar o LUTS e o `LUTS-dyn` para a SwissTM foi obter a

versão mais recente (2011-08-15) da biblioteca de MTS da página web dos autores. Como a biblioteca tem uma API bem definida, foi relativamente fácil encontrar as primitivas das transações que precisavam ser alteradas (início, aborto e *commit*). Não havíamos tido contato prévio com a SwissTM, mas fomos capazes de adaptar o `LUTS-dyn` em um único dia, usando uma versão antiga do Shrink como guia. Em seguida validamos nossa implementação ao executar o STAMP que já estava adaptado à SwissTM, o STAMP também está disponível na página web dos autores. A única complicação encontrada durante todo o processo foi devido ao fato da SwissTM estar implementada em C++, enquanto que nosso escalonador está implementado na linguagem C.

Desse ponto em diante usamos o termo `LUTS-swiss` para indicar a SwissTM com o suporte do `LUTS-dyn` e `LUTS-tiny` para denotar a versão do TinySTM com o suporte do `LUTS-dyn`. Para quantificar o *overhead*, procedemos da seguinte maneira: inicialmente, executamos todas as aplicações do STAMP (exceto Bayes) usando a SwissTM sem o nosso escalonador (a implementação original da SwissTM). Depois repetimos o experimento, mas agora com o `LUTS-swiss`, e calculamos o *speedup* para cada uma das aplicações em relação à implementação original. Por fim, repetimos esse processo para a TinySTM. Em seguida comparamos os resultados da `LUTS-swiss` e `LUTS-tiny` e verificamos se havia alguma diferença significativa para uma dada aplicação e um número de *threads*. Na maioria dos casos, nenhuma disparidade foi constatada.

Para ilustrarmos melhor, a Figura 5.5 mostra três aplicações (Genome, Intruder e Kmeans) com os seus respectivos *speedups* para 64 e 128 *threads*. Podemos notar que o desempenho da `LUTS-swiss` foi um pouco inferior do que a `LUTS-tiny` para as aplicações Genome e Intruder. Se todas as aplicações apresentassem o mesmo comportamento, poderíamos concluir que o *overhead* inserido pelo `LUTS-dyn` foi relativamente maior no SwissTM, contudo, `LUTS-swiss` apresenta melhor desempenho do que a `LUTS-tiny` quando analisamos os resultados da aplicação Kmeans, especialmente com a configuração High e com 128 *threads*. Portanto, concluímos que não há evidências para afirmar que `LUTS-dyn` adiciona mais *overhead* quando adaptada à `LUTS-swiss`.

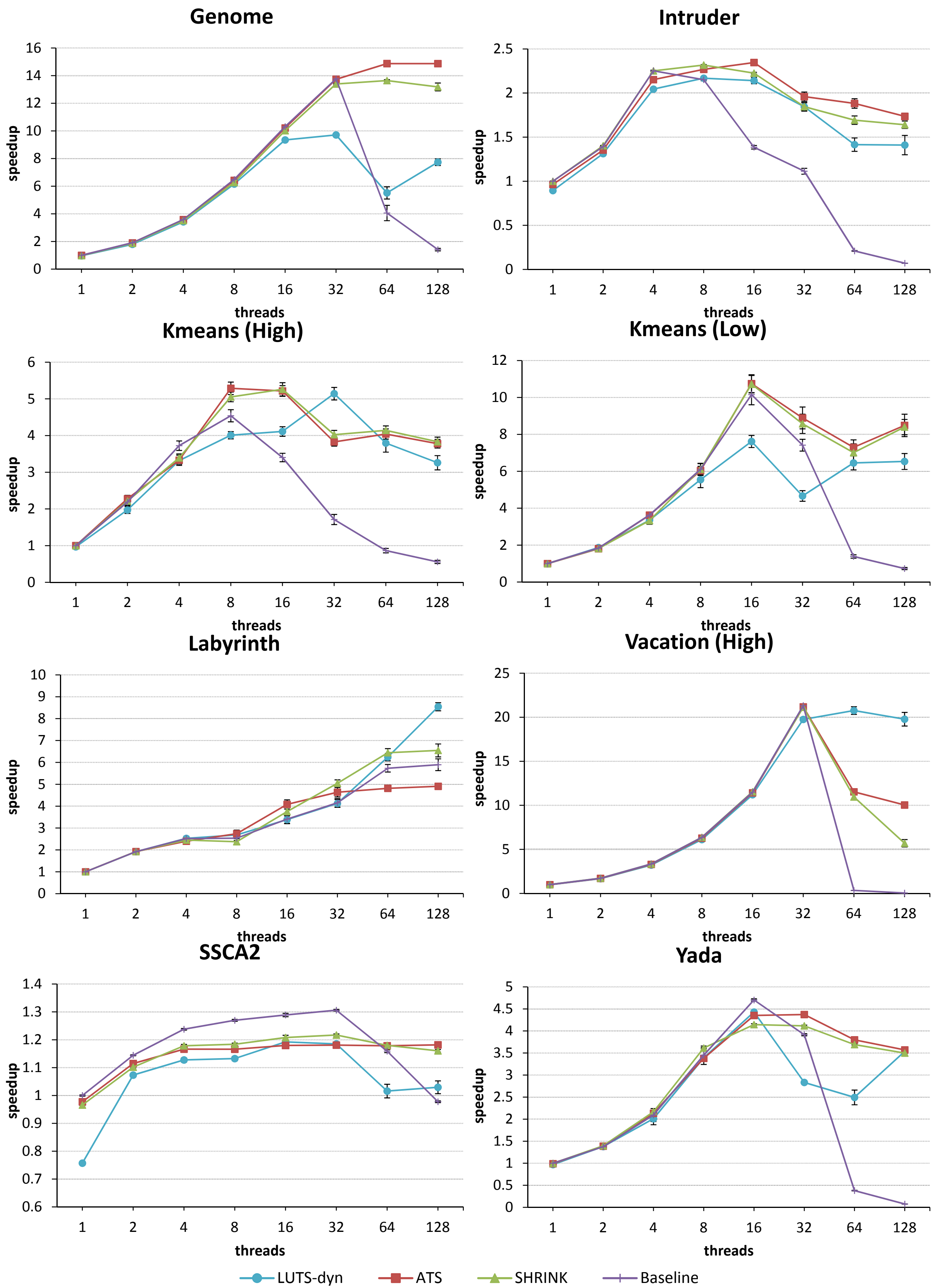


Figura 5.3: *Speedup* para as aplicações do STAMP.

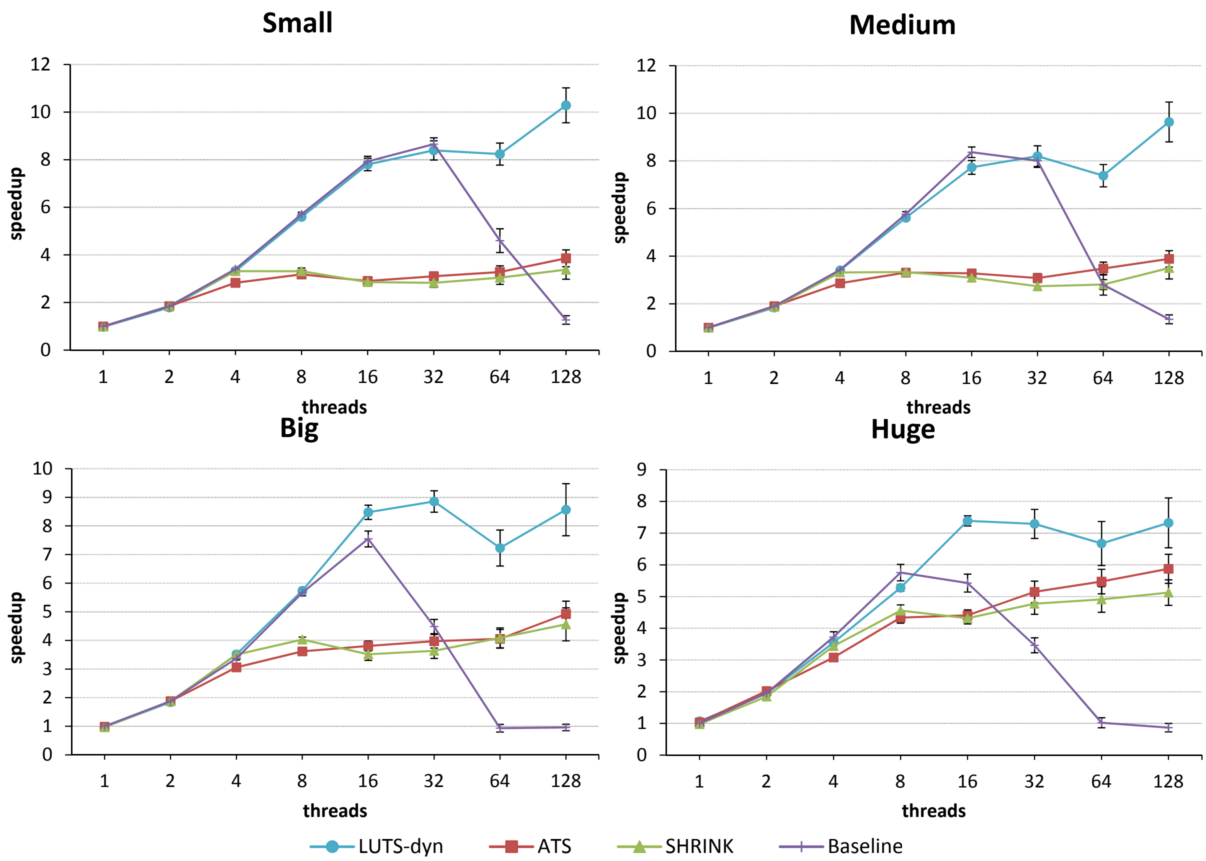


Figura 5.4: *Speedup* para as aplicações do STMBench7.

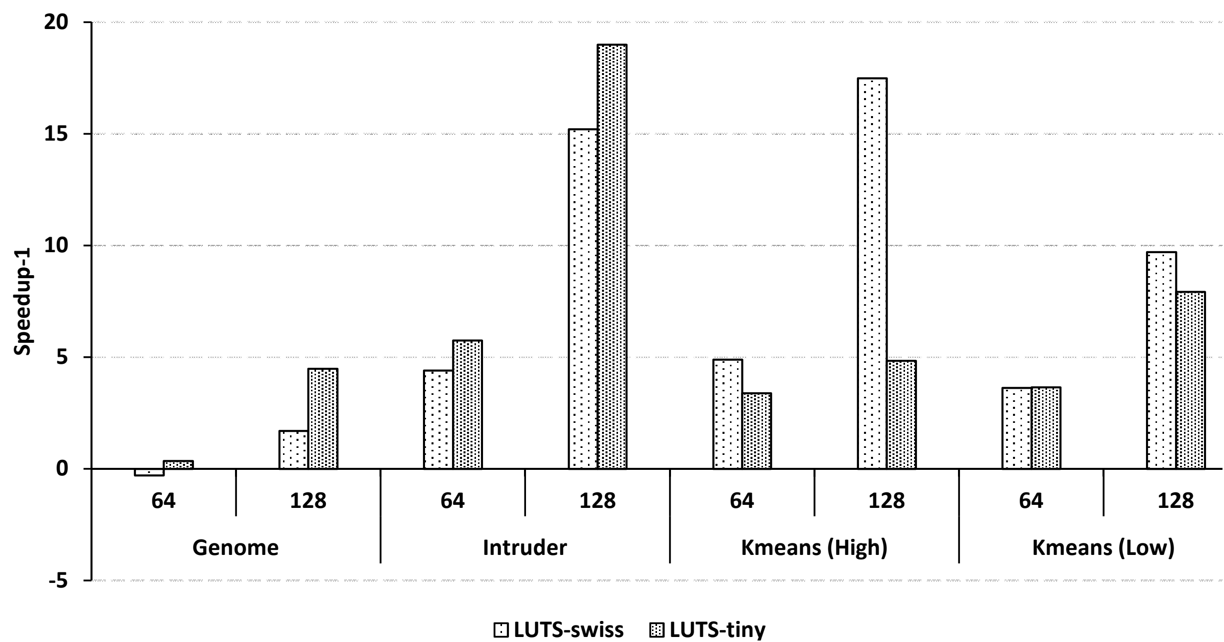


Figura 5.5: Speedup do `LUTS-swiss` e `LUTS-tiny` em relação à implementação original de ambas bibliotecas para algumas aplicações do STAMP.

Capítulo 6

Conclusões

Neste trabalho focamos nossa atenção em formas de melhorar o desempenho de sistemas de memória transacional em software. É conhecimento comum que sistemas de MTS não possuem bom desempenho em aplicações com alto nível de contenção, pois o alto índice de transações abortadas limita o progresso do sistema. Por isso, pesquisamos formas de evitar conflitos entre transações, fazendo com que abortos transacionais sejam menos frequentes.

Verificamos que as pesquisas nessa área iniciaram com novas políticas de contenção, evoluíram para gerenciadores de contenção e por fim chegaram aos escalonadores de transações pró-ativos, os quais tentam prever se uma transação irá abortar antes mesmo dela iniciar. Porém, todos os escalonadores até então se baseiam na serialização das transações após a previsão de um conflito, o que impossibilita explorar todo o paralelismo da aplicação. O nosso escalonador avança mais um passo, substituindo uma transação com alta probabilidade de aborto por outra transação com mais chances de sucesso.

Nós apresentamos o LUTS, um escalonador de transações em nível de usuário com baixo custo e uma heurística dinâmica para a previsão de conflitos que utiliza a interface de escalonamento disponibilizada pelo LUTS. A heurística, chamada `LUTS-dyn`, é capaz de alternar entre duas diferentes técnicas dependendo das características do programa em execução. Quando o programa está executando transações curtas, uma heurística com baixo custo, porém pouco precisa na previsão de conflitos entra em ação, e quando o programa estiver executando transações longas é utilizada uma heurística mais robusta e mais precisa na previsão de conflitos.

Avaliamos o LUTS em conjunto com nossa heurística dinâmica através de dois conjuntos de aplicações: STAMP e STMBench7. Para a maioria das aplicações, verificamos que nossa técnica provê ganhos de desempenho, especialmente em cenários com mais *threads* do que o número de núcleos disponíveis. Nossos melhores resultados foram obtidos no STMBench7 devido à sua grande diversidade de transações, o que favorece a nossa abor-

dagem de um escalonador de transações. Conseguimos *speedups* de até 10x onde outros trabalhos chegam a apenas 4x. Apesar de nosso principal resultado ter usado a biblioteca de TinySTM, nós também avaliamos nossa abordagem na biblioteca SwissTM, mostrando que nosso escalonador pode ser facilmente portado e apresenta bons resultados em ambos os sistemas.

Acreditamos que este trabalho é uma boa contribuição para que memória transacional dê mais um passo na direção de ser adotada pela indústria e utilizada em sistemas de programação paralela baseados em MTS.

6.1 Trabalhos Futuros

Memória Transacional ainda tem muito a amadurecer em várias direções. Nessa seção focamos em escalonadores de transações e apresentamos algumas ideias do que pode ser feito para que MT evolua ainda mais.

Existem poucos conjuntos de programas de teste para MT, e mesmo os conjuntos existentes apresentam pouca diversidade entre eles. Isso limita a pesquisa em MT, pois não é possível testar e validar técnicas utilizando um vasto e diversificado conjunto de testes. Características que ainda não estão sedimentadas nos atuais conjuntos de testes são:

- Transações com diferentes durações. Programas com as diferentes combinações de transações: curtas, médias e longas.
- Transações com diferentes padrões de leitura e escrita e diferentes proporções de leitura e escrita.
- Diversidade de transações em um mesmo programa. Esta tese mostra que para extrair o máximo de paralelismo de uma aplicação utilizando MT, é importante ter a maior quantidade possível de transações diferentes em um programa.
- Programas que reflitam o nível e o estilo de um programador comum, principal alvo de sistemas de MT.
- Programas que sejam usados por usuários no dia a dia. Atualmente os programas de MT focam apenas em programas científicos.

Apesar da grande quantidade de dados apresentados nesse trabalho, ainda há diversos experimentos a serem feitos com o LUTS. Por exemplo, usar os mesmos conjuntos de aplicações com uma proporção diferente de leituras/escritas. Em nossos testes os programas possuem maior carga de leituras. Em teoria, uma maior quantidade de escritas pode

gerar mais conflitos entre transações. Com isso o LUTS pode ser ainda mais eficiente, já que vão existir mais conflitos a serem evitados. Outro fator é a política de detecção de conflitos utilizada. Nos testes foi utilizada a política *eager*, também é preciso verificar qual o comportamento do LUTS com a política *lazy*.

O LUTS consegue trocar a transação a ser executada com muita eficiência, porém, em aplicações com poucas transações diferentes, é muito comum o caso em que não é possível achar outra transação para substituir a transação prestes a abortar, pois a fila transações disponíveis pode conter apenas um tipo de transação (mesmo ID). Nesses casos, é interessante recorrer a outras técnicas, como atrasar o início da transação ou até mesmo serializá-la para evitar o aborto (que pode ser bastante custoso).

Tendo em mente o atraso do início das transações, é preciso investigar a possibilidade de anotar o *timestamp* dos inícios, *commits*, leituras e escritas das transações e, baseado nessa informação, atrasar o início de algumas transações de forma a encontrar o encaixe perfeito entre elas e evitar que elas conflitem. Para anotar o *timestamp* das transações é possível utilizar contadores de hardware disponíveis nos processadores x86 mais recentes.

Por fim, também pensamos em trazer MT para outro nível de abstração. A ideia é deixar MT inerente às estruturas de dados, ou seja, a própria estrutura de dados estar ciente do tempo que cada operação sua gasta e conseguir a partir disso controlar os acessos de diferentes transações. Por exemplo, no caso de uma lista ligada, é possível saber quando uma transação passou por determinado nó e se ele foi modificado. Com essa informação a própria estrutura de dados pode permitir ou não que outra transação acesse a lista ligada, ou ainda melhor, decidir a partir de que momento a segunda transação pode acessar a lista de forma que quando ela alcançar o nó que foi modificado, a primeira transação já terá realizado o seu *commit* e não gerará um conflito.

Referências Bibliográficas

[1] G. S. Almasi and A. Gottlieb. *Highly parallel computing.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[2] 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.

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro*, 26:59–69, January 2006.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *SIGOPS Oper. Syst. Rev.*, 25:95–109, September 1991.

[5] Mohammad Ansari, Mikel Lujan, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-Abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 4–18, January 2009.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[8] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd ACM/IEEE International Symposium on Microarchitecture*, pages 156–167, December 2009.

[9] J Martin Bland and Douglas G Altman. Statistical methods for assessing agreement between two methods of clinical measurement. *International Journal of Nursing Studies*, 47(8):931–936, 2010.

[10] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[11] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 24–34, New York, NY, USA, 2007. ACM.

[12] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 127–138, Washington, DC, USA, 2008. IEEE Computer Society.

[13] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, October 1984.

[14] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29:6–16, March 2009.

[15] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European Conference on Computer Systems*, pages 27–40, April 2010.

[16] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 347–358, New York, NY, USA, 2006. ACM.

[17] D. Culler. *Parallel computer architecture: a hardware/software approach.* 1999.

[18] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[19] Frederica Darema. The SPMD model: Past, present and future. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 1, 2001.

[20] Vivek De and Shekhar Borkar. Technology and design challenges for low power and high performance. In *Proceedings of the 1999 international symposium on Low power electronics and design*, ISLPED '99, pages 163–168, New York, NY, USA, 1999. ACM.

[21] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9:143–155, March 1966.

[22] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 157–168, New York, NY, USA, 2009. ACM.

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

[24] Edsger Wybe Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, 1965.

[25] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th Annual Symposium on Principles of Distributed Computing*, pages 125–134, August 2008.

[26] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[27] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 155–165, June 2009.

[28] Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th Annual Symposium on Principles of Distributed Computing*, pages 7–16, August 2009.

[29] Robert Ennals and Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, 2006.

[30] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19:624–633, November 1976.

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

[32] Michael J. Flynn and Patrick Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, 25:16–31, May 2005.

[33] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, February 2004.

[34] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 101–110, New York, NY, USA, 2010. ACM.

[35] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[36] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *19th International Symposium on Distributed Computing*, pages 303–323, September 2005.

[37] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264, July 2005.

[38] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the 2nd European Conference on Computer Systems*, pages 315–324, March 2007.

[39] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[40] Tim Harris and Keir Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38:388–402, October 2003.

[41] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2nd edition, June 2010.

[42] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[43] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 14–25, New York, NY, USA, 2006. ACM.

[44] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[45] 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.

[46] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. pages 289–300, June 1993.

[47] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.

[48] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17:549–557, October 1974.

[49] IBM. Ibm unveils zenterprise ec12, a highly secure system for cloud computing and enterprise data, http://www-03.ibm.com/press/us/en/pressrelease/38653.wss#resource, 2012.

[50] James Reinders (Intel). Transactional synchronization in haswell, http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell, 2012.

[51] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[52] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[53] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6:213–226, 1981.

[54] James Larus and Christos Kozyrakis. Transactional memory. *Commun. ACM*, 51:80–88, July 2008.

[55] Dongning Liang. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 185–, Washington, DC, USA, 2001. IEEE Computer Society.

[56] 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.

[57] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, pages 79–90, January 2010.

[58] Virendra J. Marathe, William N. Scherer Iii, and Michael L. Scott. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

[59] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer Iii, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Dept. of Computer Science, Univ. of Rochester*, 2006.

[60] 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.

[61] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5:17–, July 2006.

[62] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[63] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[64] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[65] 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.

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

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

[68] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *in HPCA*, pages 254–265, 2006.

[69] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[70] Daniel Nicacio and Guido Araujo. Abortos falsos em sistemas de memória transacional em software. Technical Report IC-09-32, Institute of Computing, University of Campinas, September 2009. In Portuguese, 22 pages.

[71] Daniel Nicacio and Guido Araujo. Reducing false aborts in stm systems. In *Algorithms and Architectures for Parallel Processing*, volume 6081 of *Lecture Notes in Computer Science*, pages 499–510. 2010.

[72] Daniel Nicacio, Alexandro Baldassin, and Guido Araújo. LUTS: A lightweight user-level transaction scheduler. In *Proceedings of the 11th International Conference on Algorithms and Architectures For Parallel Processing*, pages 144–157, October 2011.

[73] Daniel Nicacio, Alexandro Baldassin, and Guido Araujo. LUTS: A Lightweight User-Level Transactional Scheduler. Technical report, Institute of Computing, University of Campinas, December 2010.

[74] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 43–52, New York, NY, USA, 2011. ACM.

[75] David A. Patterson and John L. Hennessy. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[76] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[77] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: transactional memory for an operating system. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 92–103, New York, NY, USA, 2007. ACM.

[78] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 246–257, Washington, DC, USA, 2008. IEEE Computer Society.

[79] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an stm. In *Proceedings of the 14th ACM SIGPLAN*

symposium on Principles and practice of parallel programming, PPoPP '09, pages 163–172, New York, NY, USA, 2009. ACM.

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

[81] 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 eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[82] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[83] Toufik Sarni, Audrey Queudet, and Patrick Valduriez. Real-time support for software transactional memory. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pages 477–485, Washington, DC, USA, 2009. IEEE Computer Society.

[84] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 240–248, July 2005.

[85] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[86] Martin Schoeberl, Florian Brandner, and Jan Vitek. Rttm: real-time transactional memory. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 326–333, New York, NY, USA, 2010. ACM.

[87] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[88] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. *SIGARCH Comput. Archit. News*, 36:139–150, June 2008.

[89] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 104–115, New York, NY, USA, 2007. ACM.

[90] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pages 141–150, February 2009.

[91] Michael F. Spear, Virendra J. Marathe, William N. Scherer Iii, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[92] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[93] A.S. Tanenbaum. *Sistemas operacionais modernos*. PRENTICE HALL BRASIL, 2006.

[94] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *ACM/IEEE Conference on Supercomputing*, 2004.

[95] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11:25–33, January 1967.

[96] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 533–544, New York, NY, USA, 1998. ACM.

[97] Jim Turley. The two percent solution, http://www.eetimes.com/discussion/other/4024488/the-two-percent-solution, 2012.

[98] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *In Proceedings of the Third ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2008.

[99] Lee C. Y. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, 1961.

[100] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.

[101] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, June 2008.