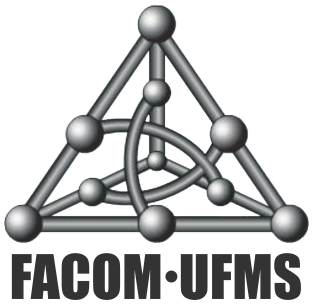
Dissertação de Mestrado

Uma plataforma para desenvolvimento e avaliação de estratégias para Memória Transacional em Software

Thales Farias Duarte

Orientação: Prof. Dr. Irineu Sotoma

Área de Concentração: Memória Transacional



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
22 de Agosto de 2012.

Pesquisa desenvolvida com suporte financeiro da Fundação de Apoio ao Desenvolvimento do Ensino, Ciência e Tecnologia do Estado de Mato Grosso do Sul (FUNDECT). Termo de Compromisso Nº: 044/10. Processo: 23/200.306/2009. Edital: Chamada FUNDECT Nº 15/2009 – POSGRAD - Mestrado.

Uma plataforma para desenvolvimento
e avaliação de estratégias para Memória
Transacional em Software

Este exemplar corresponde à redação final da dissertação devidadamente corrigida e defendida por Thales Farias Duarte e aprovada pela Banca Examinadora.

Campo Grande, 22 de agosto de 2012.

Banca Examinadora:

- Prof. Dr. Irineu Sotoma (FACOM/UFMS) - orientador
- Prof. Dr. Renato Porfírio Ishii (FACOM/UFMS)
- Prof. Dr. Ricardo Ribeiro dos Santos (FACOM/UFMS)
- Prof. Dr. Alexandro José Baldassin (UNESP)

Agradecimentos

Ao meu orientador Prof. Dr. Irineu Sotoma, que começou a me orientar no ano de 2007 em um projeto de iniciação científica, depois em 2009 no trabalho de conclusão de curso e a partir de 2010 neste trabalho de mestrado. Eu gostaria de agradecer todo apoio que ele me ofereceu durante estes 5 anos que trabalhamos juntos. Obrigado Professor Irineu, durante estes 5 anos o senhor não foi somente um orientador, mas um grande amigo.

À minha família e namorada que sempre me apoiaram durante todo mestrado.

Aos participantes da banca examinadora, que pacientemente leram o texto da dissertação e contribuíram para o aperfeiçoamento da pesquisa.

À FUNDECT (Chamada FUNDECT Nº 15/2009 – POSGRAD - Mestrado) pelo apoio financeiro.

Resumo

Atualmente existe um grande número de processadores multi-core, e para se obter melhor desempenho dos programas nestes processadores, o modelo de programação mais utilizado é o *multi-threaded*. Este modelo utiliza uma combinação de bloqueios para o controle de concorrência entre as threads, mas muitos problemas podem surgir com a utilização errada desta técnica, tal como *deadlocks* e *starvation*. Em reação a estes problemas apresentados, o modelo transacional de sincronização tem recebido atenção como um modelo de programação alternativo, usando como base o conceito de transação para garantir sincronismo entre threads concorrentes. Este modelo tira do programador a responsabilidade sobre o controle de sincronização, e problemas como *deadlocks* e *starvation* são evitados.

Neste trabalho implementamos o STM Builder, uma plataforma para desenvolvimento de estratégias para Memória Transacional em Software, que foi criada com a modularização da SwissTM. Junto com o STM Builder, desenvolvemos um framework de testes para a avaliação das implementações realizadas, que executou os testes e gerou todos os gráficos utilizados no trabalho automaticamente. O framework de testes utiliza os benchmarks STMBench7, STAMP e o Microbenchmark de Árvore Rubro-Negra. Os gráficos gerados pelo framework utilizam o gnuplot.

O STM Builder é capaz de testar várias variações no mesmo código, perdendo pouco tempo com novas implementações e testando diversas formas da mesma implementação rapidamente. No STM Builder há algumas variações do modelo transacional de sincronização em software da SwissTM. Implementamos e testamos: novos gerentes de contenção, como o *RandomizedRounds* (RR), gerenciamento de contenção de uma e duas fases, diferentes formas de recuo de uma transação, e a técnica de invalidação no momento da efetivação (*Commit-time invalidation*).

Implementamos uma plataforma para desenvolvimento e avaliação de estratégias para Memória Transacional em Software, que é formada pelo STM Builder e pelo framework de testes, que comprovou facilitar implementações e testes de novas ideias.

Palavras-chave: Memória Transacional, Memória Transacional em Software, STM, STM Builder, SwissTM

Abstract

Nowadays many processors are multi-core and, in order to obtain better performance of the programs in these processors, the programming model used is the multi-threaded. This model uses a combination of locks to control concurrency between threads, but many problems can arise with the wrong use of this technique, such as deadlock and starvation. In response to these problems presented, the transactional model of synchronization has received attention as an alternative programming model, using as a basis the concept of transaction to ensure synchronization between concurrent threads. This model takes the programmer's responsibility for the synchronization control, and problems such as deadlock and starvation are avoided.

In this work we implement the STM Builder, a platform for developing strategies for Software Transactional Memory, which was created from the modularization of SwissTM. Along with the STM Builder, we developed a test framework, which executed the tests and generated all graphics used on the work automatically. The test framework executes tests with benchmarks STMBench7, STAMP, and with Microbenchmark of Red-black Tree. The graphs generated by framework using the gnuplot.

The STM Builder is able to test several variations on the same code, wasting little time with new implementations and testing various ways of the same implementation quickly. In STM Builder there are some variations of the transactional model of SwissTM. We implemented and tested: new contention managers, such as RandomizedRounds, contention management of one and two phases, different ways to backoff a transaction, and the technique of Commit-time invalidation.

We have implemented a platform for developing and evaluating strategies for Software Transactional Memory, which is formed by the STM Builder and the test framework, which demonstrated to facilitate implementation and testing of new ideas.

Keywords: Transactional Memory, Software Transactional Memory, STM, STM Builder, SwissTM

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente existe um grande número de processadores multi-core, que usualmente possuem frequência de operação mais baixa, na tentativa de diminuir a quantidade de energia dissipada. Também integram em um mesmo chip dois ou mais núcleos de processamento na tentativa de extrair maior nível de paralelismo dos programas [31].

Ao adicionar novos núcleos de processamento não se aumenta a velocidade do processador, ou seja, o tempo em que uma instrução é executada não muda, mas melhora-se muito a vazão do processador, aumentando o número de instruções executadas em um período de tempo. Este fato implica que o modelo sequencial de programação deve ser mudado para um modelo paralelo.

O modelo de programação paralela é mais difícil que o modelo sequencial, pois é mais complicado produzir um código correto, robusto e seguro. Isto acontece pela dificuldade de se escrever e depurar o código [31].

Atualmente, o modelo de programação mais utilizado é o *multi-threaded* [31], onde normalmente os programadores dependem de uma combinação de bloqueios (*locks*) e condições, tais como monitores, para impedir o acesso simultâneo por threads diferentes ao mesmo dado compartilhado [23].

A técnica de combinação de bloqueios e condições possui muitos problemas. Os programadores devem decidir entre bloqueio de blocos (*coarse-grained locking*), em que uma estrutura de dados grande está protegida por um bloqueio único, e bloqueios refinados (*fine-grained locking*), em que um bloqueio está associado a cada componente da estrutura de dados. Bloqueio de blocos é mais simples, mas permite pequena ou nenhuma concorrência, impedindo o programa de explorar múltiplos núcleos de processamento. Pelo contrário, bloqueio refinado é substancialmente mais complicado devido a necessidade de garantir que threads tenham todos os bloqueios necessários para obter bom desempenho [20].

Em um ambiente multiprogramado, vários processos podem competir por um número finito de recursos. Um processo requisita recursos e se os recursos não estão disponíveis neste instante, o processo entra em estado de espera. Os processos em estado de espera podem nunca mais sair deste estado, pois os recursos requisitados estão retidos por outros

processos no estado de espera. Esta situação é um dos erros de implementação mais comuns utilizando bloqueios e denomina-se *deadlock* [33].

Em reação a estes problemas, um mecanismo de sincronização alternativo que vem despertando interesse por parte de pesquisadores nos últimos anos é a Memória Transacional (*Transactional Memory*, ou TM). O modelo transacional de sincronização tem recebido atenção como um modelo de programação alternativo, usando como base o conceito de transação para garantir sincronismo entre threads concorrentes. Este modelo tem como ideia facilitar a programação *multi-threaded* porque o programador não precisa se preocupar em garantir a sincronização como nas abordagens baseadas em bloqueios. Todo o controle de acesso à memória compartilhada é realizado automaticamente pelo sistema que implementa a Memória Transacional [23, 31].

Este projeto surgiu a partir de um trabalho de conclusão de curso, que implementou e realizou experimentos em um framework de Memória Transacional em Software (STM) denominado DSTM2 [20]. A partir do estudo do DSTM2, percebemos que existem várias estratégias de implementação do modelo transacional, cada estratégia implementada em um sistema STM diferente e algumas existem somente na forma de pseudocódigo com análises teóricas, sem avaliações experimentais.

Uma vez que cada sistema STM implementa uma estratégia diferente, fica muito difícil medir realmente na prática qual é a mais viável e a que possui melhor desempenho para uma certa carga de trabalho. A partir da implementação das principais estratégias em um mesmo sistema STM, podemos chegar a resultados consistentes do desempenho das estratégias em várias cargas de trabalho definidas por benchmarks. Com os estudos existentes, não é possível chegar a todas essas conclusões, porque todo o conhecimento está espalhado, existem muitas estratégias, mas implementadas em sistemas STM diferentes que não foram agrupadas para comparação.

Usaríamos o sistema STM SwissTM [10] para implementar as estratégias, que é um sistema atual, e é considerado o estado da arte dos sistemas STM, ou seja, o que possui o melhor desempenho [10]. Uma vez que a SwissTM é atualmente o melhor sistema STM, as implementações das estratégias neste sistema trazem resultados confiáveis.

Inicialmente, o objetivo era a implementação de estratégias alternativas para tentar encontrar uma combinação adequada de estratégias que melhorassem o desempenho dos sistemas STM. Logo no início da implementação da estratégia de Conhecimento de Dependência, fielmente como foi proposta no artigo [2], foi observado que estávamos adicionando inúmeros problemas de concorrência e grande sobrecarga na memória, causado pelas inúmeras estruturas de dados necessárias.

A partir destas observações iniciais, decidimos alterar a maneira de alcançar o objetivo principal deste projeto que é melhorar o estado da arte, ou seja, melhorar o desempenho da SwissTM. Para adicionar novas estratégias na SwissTM, grandes modificações na sua estrutura são necessárias e para isto foi realizada uma modularização da SwissTM.

A modularização consiste em adicionar à SwissTM o suporte para combinações de estratégias via parâmetros em um arquivo de configuração. Com a modularização é possível escolher qual gerente de contenção se deseja utilizar, quantas fases esse mecanismo

de gerenciamento terá, qual o tipo de tratamento de conflitos de leitura e escrita e inúmeros outros parâmetros. Com a modularização criamos um novo sistema STM, que denomina-se "STM Builder", em que pode-se facilmente adicionar novos mecanismos para a realização de testes e comparações justas.

Uma vez que a SwissTM possui compatibilidade com vários benchmarks, a modularização preservou essa compatibilidade para facilitar testes e evitar trabalho adicional. A avaliação de desempenho foi realizada usando benchmarks como o STMBench7 [18] e o STAMP [6] e Microbenchmarks [10, 21], por apresentarem vários tipos de cargas de trabalho e serem utilizados em testes de STMs por vários artigos [10, 9].

Após o desenvolvimento do STM Builder, implementamos e testamos: o gerente de contenção *RandomizedRounds* (RR) [32], gerenciamento de contenção de uma e duas fases, diferentes formas de recuo de uma transação, e a técnica de invalidação no momento da efetivação (*Commit-time invalidation*) [14].

Neste trabalho também desenvolvemos um framework de testes que é acoplado ao STM Builder. Este framework realizou todos os testes e gerou todos os gráficos utilizados no trabalho automaticamente, fazendo com que o STM Builder seja uma plataforma focada em desenvolvimento de novas ideias, onde o desenvolvedor não precisa se preocupar com muitos detalhes da plataforma nem com a geração de gráficos.

Mais precisamente, as contribuições deste trabalho são as seguintes:

- Desenvolvemos um novo sistema de memória transacional em software, o STM Builder, que é baseado na SwissTM.
- Junto com o STM Builder desenvolvemos um framework de testes, que também pode ser utilizado com outros STMs. O framework de testes executa e gera gráficos de testes automaticamente.
- Uma plataforma para desenvolvimento e avaliação de estratégias para Memória Transacional em Software, que é composta do STM Builder e do framework de testes.
- O gerenciamento de contenção de uma e duas fases possuem desempenho idênticos.
- O gerente de contenção *Greedy* possui desempenho melhor que o gerente de contenção *RandomizedRounds*.
- A técnica de recuo após um aborto é mais vantajosa do que a técnica de reiniciar a transação imediatamente após um aborto, e mais vantajosa que a técnica de recuar quando um conflito é encontrado.
- A melhor técnica de recuo após um aborto é aquela cuja a transação recua por um período proporcional ao número de seus abortos sucessivos, que é a técnica apresentada pela SwissTM.
- Implementamos uma nova ideia no STM Builder, a técnica de invalidação no momento da efetivação, do artigo [14]. Adaptamos a ideia para tentar torná-la viável

para o STM Builder. Mostramos que realmente é possível implementar e testar novas ideias no STM Builder, e apresentamos também uma nova heurística para a invalidação no momento da efetivação, a heurística iWork que teve melhor desempenho nos testes realizados que as heurísticas iFair e iPrio apresentadas pelo artigo [14].

O Capítulo 2 apresenta os conceitos básicos sobre memória transacional. O Capítulo 3 mostra em detalhes o STM Builder e as novas implementações realizadas no STM Builder, como a técnica de invalidação no momento da efetivação [14]. O Capítulo 4 apresenta em detalhes o Framework de Testes e discute os resultados de testes e comparações de desempenho entre o STM Builder e a SwissTM. O último capítulo resume as contribuições do trabalho e apresenta algumas sugestões de trabalhos futuros.

Capítulo 2

Conceitos Básicos

As seções a seguir fornecem os conceitos básicos sobre Memória Transacional e sobre Memória Transacional em Software, que é o enfoque deste trabalho.

2.1 Memória Transacional

Herlihy e Moss definiram o termo "Memória Transacional" como sendo uma nova arquitetura para multiprocessadores que objetiva tornar a sincronização livre de bloqueios tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua. A definição foi realizada em 1993 no artigo [22], e pode-se encontrar a tradução em português da definição na tese de doutorado [3].

A Memória Transacional utiliza o conceito de transação, um conceito muito utilizado em Sistemas de Banco de Dados. As transações são utilizadas na Memória Transacional para coordenar acessos à memória compartilhada. A definição de transação para o contexto deste trabalho é que uma transação é uma sequência de passos executados por uma única thread [23, 22].

Para entender melhor como as transações funcionam na Memória Transacional, é necessário saber como elas são executadas. Em Memória Transacional as transações executam especulativamente, ou seja, quando uma transação executa, ela faz tentativas de mudança no objeto. Se ela completa sua execução sem encontrar conflitos de sincronização, então é efetivada (*commit*), fazendo com que suas alterações sejam permanentes, ou ela aborta, descartando suas modificações.

Transações em Memória Transacional devem satisfazer a Opacidade, que é uma condição para a correção de TMs, que é certamente assegurada pela maioria das implementações de memória transacional. Ela é uma propriedade *safety* que captura os seguintes requisitos [17, 16]:

- Todas as operações realizadas por todas transações efetivadas parecem ter acontecido como se estivessem em algum momento único e indivisível, durante o tempo de

vida da transação;

- Não existem operações realizadas por qualquer transação abortada que sejam visíveis para outras transações;
- Todas as transações sempre observam um estado consistente do sistema.

Atualmente, existem três linhas de pesquisa em Memória Transacional. A primeira linha e mais antiga é a Memória Transacional em Hardware (*Hardware Transactional Memory* – HTM) que investiga um suporte em arquiteturas de computadores para execuções das transações. Execuções de transações em hardware possuem um alto desempenho, mas possuem uma grande desvantagem, que é a limitação de recursos [31]. Esta limitação é, principalmente, a quantidade de memória usada para armazenar os blocos que uma transação opera.

A segunda linha de pesquisa, e que é o enfoque deste projeto é a Memória Transacional em Software (*Software Transactional Memory* – STM), que visa um suporte de software para implementação e execução de sistemas transacionais. Tem o objetivo de aliviar a dificuldade de programação quando se utiliza mecanismos convencionais como bloqueios e variáveis de condição.

A terceira linha existente é a abordagem Híbrida de Memória Transacional que faz a combinação de HTM e STM. Nesta abordagem, uma transação pode ser executada em modo hardware ou em software. A vantagem de se executar em hardware é o alto desempenho alcançado, mas possui a desvantagem de ter recursos limitados. Já a execução em software possui recursos adicionais (grande quantidade de memória disponível), mas desempenho não muito bom. Normalmente, nesta abordagem, a execução de uma transação é iniciada em hardware, mas caso os recursos do hardware terminem, ela é reiniciada em software [31].

2.2 Memória Transacional em Software (STM)

A Memória Transacional em Software (STM) tem como um dos principais desafios a compreensão de como tornar STM eficaz. Este desafio resume-se em como projetar e avaliar mecanismos eficazes de tempo de execução para a sincronização transacional, incluindo como estabelecer e restaurar *checkpoints*, como detectar conflitos de sincronização e como garantir o progresso [20].

A principal meta da memória transacional é livrar o programador de preocupar-se com *starvation* (inanição), *deadlock*, e os inúmeros conflitos naturais, que são resultados dos bloqueios [23].

A Memória Transacional em Software possui as seguintes características:

- **Forma de Implementação**

Existem duas abordagens de implementação que prometem um bom desempenho para STM: STMs não bloqueantes que são livres de obstrução, e bloqueantes, que são STMs com base em bloqueios e que são livres de *deadlock* [23].

A implementação não bloqueante é livre de obstrução e assegura que todas as threads não podem ser bloqueadas por atrasos ou falhas de outras threads. Esta propriedade é mais fraca do que a sincronização livre de bloqueios, porque não garante progresso quando duas ou mais threads conflitantes estão executando concorrentemente [23]. A propriedade de ser livre de *deadlocks* da implementação com bloqueios não garante progresso se as threads param em seções críticas [23].

Ambas as STMs não bloqueantes livre de obstrução e bloqueantes livres de *deadlocks* não garantem progresso quando duas ou mais threads conflitantes estão executando concorrentemente. O progresso para transações conflitantes é garantido por um gerente de contenção, um mecanismo que decide quando uma thread pode prosseguir.

- **Granularidade**

A granularidade mostra qual é a unidade básica utilizada para o versionamento dos dados e a detecção de conflitos, que pode ser realizada por palavra ou objeto. O versionamento por palavra pode trazer uma grande desvantagem, que é o uso de muito mais espaço para guardar o registro de posse de cada palavra. Já o versionamento por objeto diminui esta sobrecarga de espaço, pois cada registro de posse é criado para um objeto inteiro.

O versionamento é realizado para garantir atomicidade. Uma transação precisa manter tanto a versão antiga quanto a versão corrente de um dado modificado durante sua execução. Caso a transação seja efetivada, o valor corrente deve-se tornar permanente e o antigo descartado. Caso contrário, o valor corrente deve ser descartado [31].

- **Tipos de leitura**

Existem três tipos de leitura:

  - Visível: na leitura realizada de forma visível, as transações têm conhecimento de todo objeto (posição de memória) lido(a) por outras transações e os conflitos são detectados imediatamente.
  - Invisível: a leitura realizada de forma invisível acontece quando uma transação que lê uma variável não escreve qualquer informação nos objetos de base (posições de memória) lidos(as). Assim, muitos processos podem executar concorrentemente transações que leem as mesmas variáveis, sem invalidar seus caches, e provocando grande vazão na execução das transações. No entanto, as transações que atualizarem as variáveis não sabem se existem quaisquer transações concorrentes que leem essas variáveis [17].
  - Semi-visível: na leitura realizada de forma semi-visível, uma transação que está lendo uma posição de memória (um objeto), pode determinar se outras

transações estão lendo a mesma posição de memória (mesmo objeto) e a quantidade de transações, mas não pode determinar especificamente quais transações estão lendo a mesma posição de memória [7].

2.2.1 Detecção de Conflitos

Um conflito ocorre quando transações concorrentes acessam a mesma posição de memória e ao menos um destes acessos é uma operação de escrita.

Existem três técnicas de detecção de conflitos. A primeira técnica é a *eager* (antecipada), onde a detecção de conflitos ocorre no instante em que os conflitos aconteceram [10].

A segunda técnica é a *lazy* (preguiçosa) onde a detecção do conflitos somente ocorre após a transação ser executada e tentar efetivar [10].

A terceira técnica é a *mixed invalidation* (invalidação mista). Este esquema de detecção de conflitos detecta conflitos escrita/escrita com a técnica *eager*, a fim de evitar que transações que são condenadas a abortar a execução desperdice recursos. Possui também outro esquema de detecção de conflitos de leitura/escrita *lazy*, para permitir mais paralelismo entre as transações [36, 10].

2.2.2 Gerentes de Contenção

Em muitas STMs, uma transação pode detectar quando está prestes a causar um conflito de sincronização. A transação requisitante consulta então um gerente de contenção. O gerente de contenção serve como um oráculo, que aconselha a transação se ela deve abortar imediatamente a outra transação, ou aguardar para permitir a outra transação a chance de completar. Naturalmente, nenhuma transação deveria aguardar indefinidamente uma outra [23].

Os gerentes de contenção são implementados usando políticas de resolução de conflitos. Algumas políticas para o gerente de contenção são:

- *Backoff*: Suponha uma transação A que está prestes a entrar em conflito com uma transação B. A transação A recua repetidamente com uma duração aleatória, duplicando o tempo esperado até algum limite. Quando o tempo limite é atingido, aborta B [23]. Recuo é fazer a transação parar a sua execução e aguardar algum mecanismo para voltar a executar. Este mecanismo normalmente é algum tempo determinado ou a espera por algum comportamento, por exemplo efetivar ou abortar, de pelo menos uma outra transação.

- Prioridade: Suponha uma transação A que está prestes a entrar em conflito com uma transação B. Cada transação tem um *timestamp* quando é iniciada. Se A tem um *timestamp* menor que B, ela aborta B, e caso contrário ela espera. Uma transação

que é reiniciada após um aborto mantém seu *timestamp* antigo, garantindo que todas as transações em algum momento concluam [23].

- Guloso (*Greedy*): Suponha uma transação A que está prestes a entrar em conflito com uma transação B. Cada transação tem um *timestamp* quando é iniciada. A aborta B se ou A tem *timestamp* menor do que B, ou B está esperando por uma outra transação. Esta estratégia elimina correntes de espera de transações. Tal como na política de prioridade, cada transação em algum momento completará [15, 23].

- Agressivo (*Aggressive*): Suponha uma transação A que está prestes a entrar em conflito com uma transação B. Quando A entra em conflito, sempre B é abortada [25].

- Passivo (*Passive*): Suponha uma transação A que está prestes a entrar em conflito com uma transação B. Quando A entra em conflito, sempre A é abortada [35]. Este gerente de contenção também pode ser denominado Tímido (*Timid*) [10].

- *RandomizedRounds*: Suponha uma transação A que está prestes a entrar em conflito com uma transação B. Cada transação escolhe um número discreto uniformemente aletório no intervalo $[1, m]$, onde m representa o número de núcleos, no início e após todo aborto. A aborta B se A tem um número aleatório menor do que B. A transação B que foi abortada não deve ser reiniciada até que a sua adversária, a transação A efetive ou aborte [32].

- *Karma*: Cada transação mantém um registro da quantidade de trabalho que tem realizado, que é mantido após cada aborto da transação e zerado após a efetivação. A transação que tenha realizado mais trabalho tem mais prioridade. O *Karma* considera o número acumulado de blocos abertos por uma transação como a quantidade de trabalho realizada pela transação, ou seja, sua prioridade. Neste projeto consideramos como trabalho realizado o número de escritas realizadas por cada transação. Suponha que uma transação A entre em conflito com uma transação B. Se A tem prioridade menor do que B, o gerente espera por um determinado período de tempo para ver se B terminou. Uma vez que o número de tentativas mais a prioridade atual de A for superior à prioridade de B, A aborta B. Se no momento do conflito a prioridade da transação A é maior que a prioridade da transação B, A aborta B [25, 26].

- *Polite*: O gerente de contenção *Polite* utiliza *backoff* exponencial para resolver os conflitos encontrados. Suponha que uma transação A entra em conflito com uma transação B. A transação A faz tentativas de acesso por um período de tempo proporcional à 2^{n+k} ns, onde n é o número de tentativas que foram necessárias até o momento e k é uma constante para ajuste da arquitetura. Depois de um máximo de m tentativas, o gerente *Polite* incondicionalmente aborta a transação B [25, 26].

- Polka: Este gerente de contenção une as melhores características do gerentes de contenção *Karma* e *Polite*. Ele combina o *backoff* exponencial do *Polite* com o mecanismo de acumulação de prioridade do *Karma*. Quando uma transação A entra em conflito com uma transação B, o Polka recua um número de intervalos

Características	RSTM	TinySTM	TL2	SwissTM
Forma de Implem.	livre de obstrução	bloqueios	bloqueios	bloqueios
Granularidade	objeto	palavra	palavra	palavra
Leitura	visível e invisível	invisível	invisível	invisível
Detecção de conflitos	eager e lazy	eager	lazy	mixed inval.
Gerente de Cont.	Polka, Karma e outros	Passivo	Passivo	Duas Fases

Tabela 2.1: Características dos sistemas STM considerados o estado da arte.

igual à diferença de prioridades entre a transação A e a transação conflitante B. A única diferença entre o Polka e o Karma, no entanto, é que o comprimento desses intervalos de *backoff* aumentam exponencialmente como no *Polite* [26].

- Duas Fases: É um gerente de contenção de duas fases distintas, que não causa sobrecarga em transações de somente leitura e de escrita e leitura curtas utilizando a primeira fase com o gerente de contenção Passivo, enquanto favorece o progresso das transações que tenham realizado um número significativo de atualizações utilizando o gerente de contenção *Greedy* como segunda fase [10].

2.2.3 O Estado da Arte de Sistemas STM

Atualmente o estado da arte dos sistemas STM é a SwissTM, segundo o artigo [10], que comparou a SwissTM com outros sistemas STM que eram considerados o estado da arte, o RSTM (versão 3) [28], a TL2 [8] e a TinySTM [12]. A avaliação realizada no artigo foi bem criteriosa, com a utilização de Microbenchmarks [10, 21] para avaliação com cargas de trabalho em menor escala, que são implementações de estruturas de dados como a árvore rubro-negra, do STMBench7 [18], que é um benchmark sintético de STM que modela uma grande escala de cargas realistas de trabalho e do benchmark STAMP [6], que consiste de 8 representantes de diferentes aplicações com cargas de trabalho do mundo real. O desempenho da SwissTM também supera implementações com o código sequencial em uma ampla gama de cargas de trabalho e em duas diferentes arquiteturas multi-core segundo o artigo [9]. Os sistemas STM considerados o estado da arte possuem as características apresentadas pela Tabela 2.1.

A SwissTM, por ser o estado da arte atualmente, está sendo utilizada em vários projetos de pesquisa:

- Um novo escalonador, chamado Shrink, que prevê os acessos futuros de uma thread com base nos acessos passados, e dinamicamente serializa as transações baseado na previsão para prevenir conflitos, foi implementado na SwissTM. As experiências realizadas em benchmarks realistas, como STMBench7 e o STAMP, mostraram que o Shrink melhorou o desempenho da SwissTM [11].

- O artigo [24] combina programação relativista e memória transacional em software de uma maneira que leva o melhor dos dois mundos: leituras com baixa sobrecarga e linearmente escalável (programação relativista) que nunca entram em conflito com

escritas e acesso escalonável disjuntos para escritas paralelas (STM). A combinação destas duas técnicas foram implementadas na SwissTM. Nos testes houve pouca diferença de desempenho entre o STM derivado da SwissTM e a SwissTM original.

- O TMUnit é um framework que visa facilitar a especificação das cargas de trabalho de memória transacional. Ele permite aos usuários rapidamente especificar critérios para avaliar e comparar TMs, bem como testes de unidade para validar o comportamento de uma implementação específica. Ele executa uma carga sintética de trabalho, escrita em uma linguagem de domínio específica, em uma TM dedicada e registra estatísticas de desempenho e resultados do teste [19]. Desenvolvedores podem usar o TMUnit para verificar o comportamento e o desempenho das suas TMs. O artigo que apresenta o TMUnit [19] traz testes de desempenho e semântica aplicados na SwissTM e em outros STMs.

- O artigo [5] apresenta uma biblioteca baseada em STM para Scala, a CCSTM. Ela foca em ajudar os programadores a construir, otimistamente, algoritmos e estruturas de dados concorrentes, enquanto limitam-se a técnicas de implementação que não interferem com os componentes do sistema que não são utilizados. A CCSTM utiliza o algoritmo da SwissTM que detecta conflitos escrita/escrita antecipadamente (*eager*).

- O artigo [37] descreve o sistema TransFinder que detecta automaticamente regiões atômicas na memória compartilhada em programas SPMD (*Single Program, Multiple Data*). Este artigo utiliza a SwissTM para testes e comparações.

Uma vez que este projeto começou em março do ano de 2010, versões novas do RSTM e da SwissTM surgiram, onde foram mantidas as implementações originais dos STMs e foram adicionadas novas implementações.

O RSTM, que atualmente está na versão 5, é um pacote C++ que contém treze diferentes implementações de bibliotecas STM, possui compatibilidade com o STAMP, traz oito microbenchmarks implementados e vários gerentes de contenção. No RSTM atualmente há implementações STM com granularidade por objetos ou palavras, leitura visível ou invisível e detecção de conflitos *eager* ou *lazy* [28].

A versão atual da SwissTM foi lançada em 15 de agosto de 2011 com suporte a leituras invisíveis e visíveis com a implementação EpochSTM, e possui a capacidade de alternar dinamicamente entre diferentes esquemas de detecção de conflitos com a implementação Dynamic [1].

2.3 Benchmarks

Benchmarks para a área de computação são programas especificamente escolhidos para medir desempenho [29]. A seguir são apresentados os benchmarks que foram utilizados para avaliar o STM Builder e a SwissTM.

2.3.1 Microbenchmarks

Os microbenchmarks são benchmarks pequenos, que normalmente representam uma estrutura de dados como uma lista ou uma árvore rubro-negra. A sua utilização é uma forma predominante de medir o desempenho de STMs. Neste trabalho foi utilizado o microbenchmark de árvore rubro-negra, que consiste de transações curtas que inserem, pesquisam e removem elementos da estrutura de dados. A SwissTM e o STM Builder trazem este microbenchmark implementado [10, 21].

2.3.2 STMBench7

O STMBench7 [18] é um benchmark para avaliação de implementações de software de memória transacional. Mais especificamente, ele tem como objetivo produzir um conjunto de cargas de trabalho que:

- Correspondam a aplicações orientadas a objeto realistas e complexas que se beneficiam de *multi-threading*;
- Não dependam de qualquer STM particular ou de linguagem de programação;
- Sejam fáceis de usar e forneçam resultados que possam ser facilmente interpretados.

O STMBench7 pode ser visto como um "teste de estresse" para implementações de software de memória transacional, uma vez que inclui casos de teste que são conhecidos por ser um desafio, como percursos longos e estruturas de objetos complexos. Ele aproxima-se de como geralmente programas CAD (*Computer Aided Design* – Projeto Assistido por Computador), CAM (*Computer Aided Manufacturing* – Fabricação Assistida por Computador) e CASE (*Computer-Aided Software Engineering* – Engenharia de Software Auxiliada por Computador) trabalham.

É importante, contudo, notar que, embora os parâmetros do STMBench7 permitam emular uma grande variedade de cargas de trabalho, o benchmark é, claramente, não representativo de todas as aplicações possíveis.

O STMBench7 emula três tipos da carga de trabalho [18]: dominada por leitura (*read-dominated*), onde 90% das operações são somente leitura; dominada por leitura-escrita (*read-write dominated*), onde 60% das operações são somente de leitura; e dominada por escrita (*write-dominated*), onde 10% das operações são somente de leitura.

Ele é projetado de tal forma que deve ser fácil usá-lo com qualquer implementação de STM. Em qualquer caso, alguém pode utilizar o STMBench7 com qualquer STM sem entrar em detalhes de como o benchmark trabalha, ou como suas operações são implementadas. O seu desenvolvimento foi realizado com o intuito de tentar separar genericamente partes independentes de STM do código do STMBench7 tanto quanto possível [18].

Atualmente o STMBench7 possui implementações em Java e C++ que fornecem suporte a inúmeros STMs diferentes. Neste projeto é utilizada a versão em C++, que já

possui suporte a SwissTM e funciona corretamente com o STM Builder, pois ele mantém a compatibilidade da SwissTM [4].

2.3.3 STAMP

O STAMP (*Stanford Transactional Application for Multi-Processing*) é um benchmark abrangente para a avaliação de sistemas de memória transacional. O STAMP é desenvolvido na linguagem C e pode ser facilmente executado em muitas classes de sistemas de memória transacional, incluindo hardware (HTM), baseadas em software (STM), e projetos híbridos [6].

Este benchmark inclui oito aplicações e trinta variações de parâmetros de entrada e conjuntos de dados, a fim de representar diversos domínios de aplicações e abranger uma vasta gama de casos de execução transacionais (o uso frequente ou raro de transações, grandes ou pequenas transações, alta ou baixa contenção, entre outros). Uma breve descrição de cada aplicação do benchmark é apresentada abaixo:

- Bayes: Esta aplicação, do domínio do aprendizado de máquina, implementa um algoritmo para aprender a estrutura de redes Bayesianas a partir de dados observados. Possui transações longas e grandes conjuntos de leitura e escrita, resultando em alta contenção.
- Genome: Esta aplicação, do domínio de bioinformática, executa o sequenciamento de genes e possui transações de tamanho médio, com conjuntos de escrita e leitura médios. Nesta aplicação há baixa contenção.
- Intruder: Esta aplicação, do domínio de segurança, detecta invasões na rede escaneando pacotes de rede e comparando contra um conhecido conjunto de assinaturas de intrusão. Possui transações curtas, com conjuntos de escrita e leitura médios e esta aplicação possui alta contenção.
- Labyrinth: Esta aplicação, do domínio de engenharia, determina caminhos no labirinto e possui transações longas que tem grandes conjuntos de leitura e escrita. Possui alta contenção por causa do grande número de acessos transacionais na memória.
- SSCA2: Esta aplicação, do domínio científico, cria uma representação eficiente de um grafo. As transações do SSCA2 são curtas, seus conjuntos de leitura e escrita são pequenos, e a aplicação tem baixa contenção.
- Yada: Esta aplicação, do domínio científico, refina uma malha de Delaunay. Possui longas transações, com conjuntos de leitura e escrita grandes e uma quantidade média de contenção.
- Kmeans: Esta aplicação, do domínio de mineração de dados, fornece uma classificação de informações de acordo com os dados fornecidos. As transações do kmeans são curtas, seus conjuntos de leitura e escrita são pequenos e possui baixa contenção.

- Vacation: Esta aplicação, do domínio de processamento de transações online, emula um sistema de reservas de viagens. Possui transações de comprimento médio, com conjuntos de escrita e leitura de tamanho médio e possui contenção baixa ou média.

Capítulo 3

STM Builder

Este Capítulo apresenta o novo sistema STM, denominado STM Builder, desenvolvido a partir da modularização da SwissTM e do EpochSTM utilizando técnicas de programação orientada a objetos. Ele tem suporte para combinações de diversas estratégias via parâmetros em um arquivo de configuração. Este novo sistema foi desenvolvido com o intuito de ser fácil de usar, de adicionar novos recursos e de realizar testes e comparações justas.

Este sistema permite a escolha do gerente de contenção, do número de fases desse mecanismo de gerenciamento, da função de transição de cada fase do gerenciamento de contenção, e do tipo de tratamento de conflitos de leitura e escrita. Além disso, traz a possibilidade de mudar o tamanho da tabela de *locks* do sistema, assim como a função de Hash que acessa esta tabela. Mais detalhes destes mecanismos serão apresentados nas próximas seções.

3.1 Visão Geral do STM Builder

O sistema traz implementado um microbenchmark de árvore rubro-negra herdado da SwissTM e do EpochSTM [1], e tem suporte para testes utilizando o benchmark STMBench7 e STAMP.

A versão atual do STM Builder é baseada na versão da SwissTM lançada em 15 de agosto de 2011 com suporte a leitura invisíveis e visíveis com a implementação EpochSTM.

A configuração da execução do STM Builder é realizada por um arquivo de configuração (arquivo `config` na pasta raiz do sistema) como o exemplo da Figura 3.1. Para se ter uma nova execução do sistema com uma configuração diferente, não é necessário recompilar o sistema, basta mudar o arquivo de configuração. Desta maneira, a realização de testes fica simples, mesmo com as inúmeras possibilidades de configuração que o STM Builder tem. No arquivo de configuração do STM Builder, a entrada *OUTPUT COMPLETE* define uma impressão completa dos dados gerados pela execução do sistema, e para uma saída minimalista deve-se utilizar a entrada *OUTPUT MINIMUM*, que é útil

para o Framework de Testes (ver Seção 4.1).

A Figura 3.2 traz o diagrama de classes da SwissTM, em que a classe `TxMixinv` representa a implementação das transações da SwissTM, onde toda a implementação é realizada dentro da mesma classe e do mesmo arquivo, fazendo com que modificações sejam difíceis de serem realizadas. A Figura 3.3 traz o diagrama de classes do EpochSTM, onde diferentemente da implementação da SwissTM, o gerente de contenção é implementado separadamente e a classe `TransactionEpoch` implementa as transações. Ambos diagramas de classes trazem somente alguns métodos das classes apresentadas, os demais métodos não foram colocados para deixar o diagrama mais sucinto.

Os diagramas de classes da SwissTM e do EpochSTM são bem simples se forem comparados com o diagrama de classes do STM Builder, apresentado pela Figura 3.4. O diagrama de classes do STM Builder apresenta a modularização da SwissTM e do EpochSTM, e tem como meta ser mais simples e facilitar a implementação de estratégias existentes do modelo transacional de sincronização em software.

Os três diagramas de classes apresentados trazem em comum a dependência do pacote *Common*. Na linguagem de programação C++ não existem pacotes, mas utilizamos esta representação para facilitar o entendimento e não necessariamente todas as implementações presentes no pacote *Common* são realizadas em classes. O STM Builder, a SwissTM e o EpochSTM utilizam constantes, estatísticas, logs, operações atômicas, *locks* e outras implementações em comum. A SwissTM e o EpochSTM se diferenciam um pouco na implementação da gerência de memória, mas o STM Builder possui a implementação das duas gerências e consegue fazer a seleção da gerência de memória correta. O que diferencia o STM Builder dos outros dois STMs é como a estrutura das transações são organizadas, pois ambos têm a mesma base de sustentação.

```
1 CMS 2
2 OUTPUT MINIMUM
3 CM Passive
4 TF TF1
5 CM GreedySwissTM
6 CMRESTART NO
7 VERSION_LOCK_TABLE_SIZE 22
8 MODEL TransactionInvisible
```

Figura 3.1: Exemplo do arquivo `config` do STM Builder, está sendo escolhido o gerenciamento de contenção de duas fases e o *model* `TransactionInvisible`.

3.2 Gerenciamento de Contenção

A SwissTM utiliza um gerente de contenção de duas fases que não causa sobrecarga em transações somente de leitura (*read-only*) e de escrita e leitura curtas (*read-write*), enquanto favorece o progresso das transações que tenham realizado um número significativo de atualizações. Basicamente, as transações que são curtas ou somente de leitura utilizam o gerente de contenção *Passive* (Passivo). A SwissTM considera transações curtas, aquelas que realizam poucas escritas. Ela mantém o número de escritas que cada transação tem realizado e quando atinge um limite, ela troca de fase e de gerente de contenção

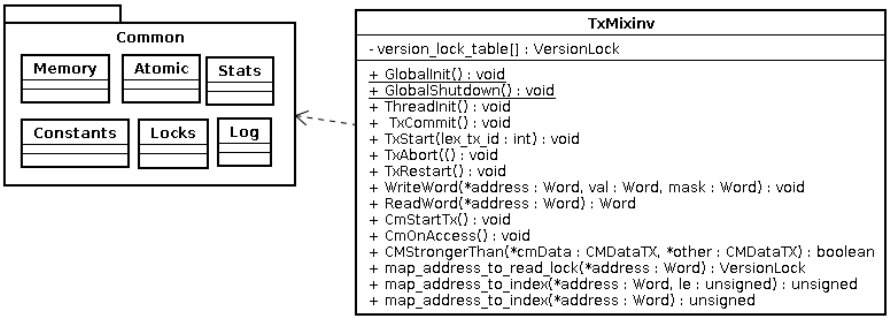


Figura 3.2: Diagrama de classes da SwissTM.

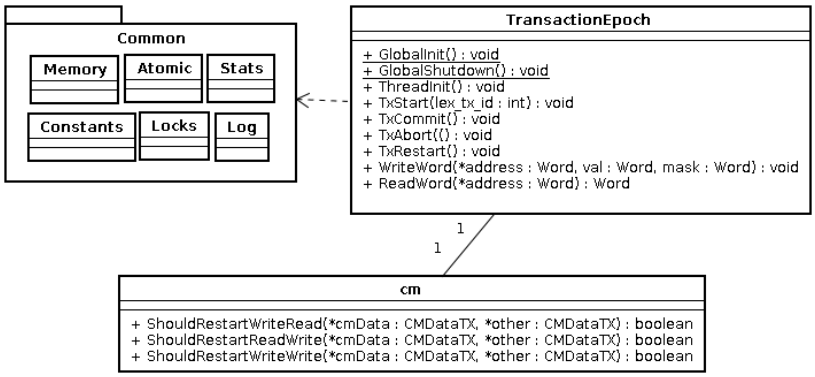


Figura 3.3: Diagrama de classes do EpochSTM.

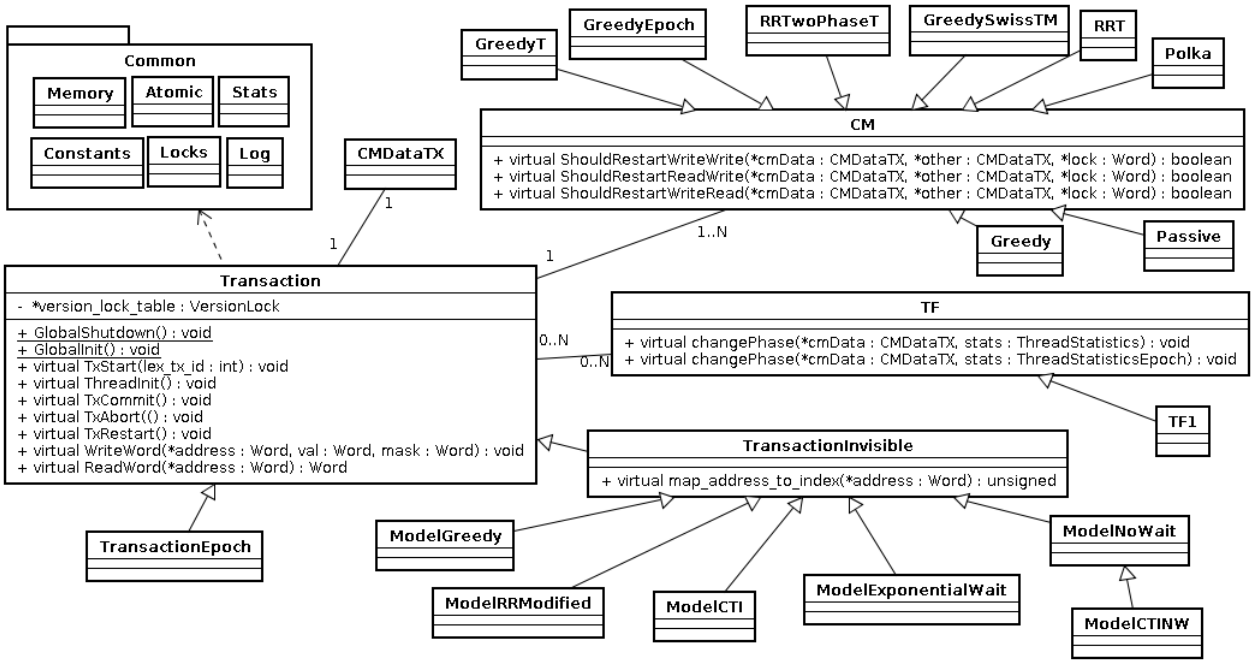


Figura 3.4: Diagrama de classes do STM Builder.

e considera que a transação não é mais curta. Para o STM Builder este mecanismo é chamado de *Transition Function* (TF). As transações que são mais complexas selecionam dinamicamente o gerente de contenção *Greedy*, que envolve mais sobrecarga, mas favorece essas transações, prevenindo *starvation*. O gerente *Greedy* idêntico ao da SwissTM será chamado de *GreedySwissTM* no restante do texto. Além disso, as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional ao número de seus abortos sucessivos [10]. O gerenciamento de contenção da SwissTM é implementando totalmente dentro da classe `TxMixinv`, como mostrado no diagrama de classes da Figura 3.2, totalmente acoplado com o código das transações e dificultando qualquer tipo de modificação.

O EpochSTM utiliza o gerente de contenção *Greedy* e o *Passive*, mas é utilizado somente um de cada vez. O EpochSTM implementa separadamente da transação o gerente de contenção, como mostrado no diagrama de classes da Figura 3.3. O que diferencia o gerenciamento de contenção do EpochSTM e do STM Builder é que no STM Builder não é necessário recompilar todo o sistema para fazer uma execução com um gerente diferente, e o STM Builder adicionou a possibilidade de se utilizar várias fases de gerenciamento de contenção com o algoritmo do EpochSTM. A implementação *Greedy* do EpochSTM é diferenciada, e no STM Builder denominamos *GreedyEpoch*.

O STM Builder traz todos os mecanismos de gerenciamento de contenção da SwissTM, mas com uma grande flexibilidade de uso e podendo adicionar novos recursos. Com o STM Builder estes mecanismos podem ser utilizados no EpochSTM.

Implementar um novo gerente de contenção no STM Builder é muito simples e para maiores detalhes veja o Apêndice A.1. Para utilizar os gerentes de contenção desejados em uma execução do sistema, é necessário configurar o arquivo `config`. Primeiramente é necessário informar a quantidade de gerentes que será usado, que pode ser realizado como a entrada *CMS 2* da Figura 3.1, que informa ao sistema que serão utilizados dois gerentes de contenção e a execução terá duas fases. Para indicar o nome do gerente utilizado, deve-se colocar a entrada CM seguida do nome do gerente. No exemplo da Figura 3.1, temos as entradas *CM Passive* e *CM GreedySwissTM*, onde informa pela ordem de colocação no arquivo de configuração que o gerente *Passive* será utilizado na primeira fase do gerenciamento e o *GreedySwissTM* na segunda.

Sempre que o STM Builder é executado com o gerenciamento de contenção com mais de uma fase, é necessário ter as *Transition Functions* (Funções de Transição - TF) que são responsáveis pela transição de uma fase para outra. Não é sempre necessário pular para a fase seguinte e o sistema permite ao implementador que selecione qualquer fase desejada do gerenciamento de contenção. Esta flexibilidade permite a possibilidade de implementação de inúmeras variações de técnicas de gerenciamento de contenção, apenas combinando gerentes. Sempre que o gerenciamento de contenção possui mais de uma fase, é obrigatório o uso de pelo menos uma *Transition Function* e é permitido utilizar quantas forem necessárias. Na Figura 3.1 foi indicado a utilização de dois gerentes, e duas fases, então é obrigatório indicar no arquivo `config` pelo menos uma *Transition Function*, a qual é indicada por *TF TF1*, onde TF1 é o nome da *Transition Function* utilizada. Detalhes de como implementar uma *Transition Function* podem ser vistos no Apêndice A.2.

Uma possibilidade bem interessante que o STM Builder também traz é a possível reinicialização dos dados locais dos gerentes de contenção que estão em *CMDataTX* após a transação ser abortada. Esta possibilidade pode ser útil para testar técnicas novas de gerenciamento de contenção. Na Figura 3.1, a entrada *CMRESTART NO* informa que a reinicialização não é utilizada. Caso seja necessária a utilização desta opção, substitua a entrada anterior por *CMRESTART YES*.

3.2.1 Gerentes de Contenção Implementados

O STM Builder traz alguns gerentes de contenção básicos implementados, que herdamos da SwissTM e EpochSTM para fazer a implementação inicial do sistema. Os gerentes de contenção são:

- *Passive*: É a implementação do gerente *Passive* apresentado no Capítulo 2.
- *GreedySwissTM*: É uma implementação do gerente *Greedy*, que chamamos de *GreedySwissTM* pois é igual ao gerente *Greedy* da SwissTM. Considere as transações A e B, se A entra em conflito com B e o gerente de contenção é chamado e A aborta B, neste momento o que diferencia o *GreedySwissTM* do *Greedy* original é que ele não necessita saber que foi A que causou o aborto de B para implementar o recuo da transação B. Outra diferença é que o *GreedySwissTM* é preparado para o gerenciamento de contenção duas fases.
- *GreedyEpoch*: É uma implementação do gerente *Greedy*, que chamamos de *GreedyEpoch* pois é igual ao gerente *Greedy* do EpochSTM.

3.3 Models

No STM Builder, um *model* é a implementação das transações, com seus métodos de leitura, escrita, efetivação, aborto, *rollback*, entre outros vários métodos existentes para execução de uma transação. Neste sistema, o *model* de cada transação é instanciado em tempo de execução para selecionar o que foi desejado para a execução, conforme o arquivo de configuração. Como pode-se observar na Figura 3.1, o *model* utilizado é o `TransactionInvisible` que foi selecionado pela entrada *MODEL* `TransactionInvisible` do arquivo de configuração. Um detalhe muito importante do STM Builder é que ele possui dois *models* bases, o `TransactionInvisible` que possui leitura invisível e o `TransactionEpoch` que possui leitura visível. Os dois *models* são filhos da classe Transaction, para que se possa fazer a seleção dos *models* em tempo de execução (veja o diagrama de classes da Figura 3.4). Todas as transações do sistema utilizam o mesmo sistema local de gerenciamento de contenção, mas a função de Hash para acessar a Tabela de *Locks* somente existe nos *models* com leitura invisíveis.

Implementar um novo *model* no sistema é simples, pois projetamos o sistema para receber novos *models*, para que os testes e comparações de novas técnicas de implemen-

tação de STMs sejam facilitadas. Veja o Apêndice A.3 para maiores detalhes de como implementar um novo *model*.

O STM Builder oferece outros *models* além dos dois *models* bases do sistema, para mais detalhes dos *models* do STM Builder veja a Subseção 3.4.2.

3.3.1 Tabela de Locks e Função Hash

O STM Builder com leituras invisíveis traz a mesma tabela de *locks* utilizada na SwissTM, em que cada palavra da memória é mapeada para uma entrada na tabela de *locks*. O mapeamento de cada palavra da memória é realizado com o que chamamos de função de Hash. Originalmente, a SwissTM e o STM Builder usam a tabela de *locks* com 2^{22} entradas e a função de Hash faz com que cada entrada da tabela de *locks* seja para quatro palavras consecutivas de memória. Ter várias palavras consecutivas de memória mapeadas para a mesma entrada na tabela de *locks* pode resultar em um falso conflito, quando palavras sem referências da memória ficam trancadas juntas, que resulta em aumento das taxas de abortos, mas isso não causa nenhum problema na corretude dos algoritmos da SwissTM [10].

Modificação da Tabela de Locks e da Função Hash

O STM Builder traz a possibilidade de alterar o tamanho da tabela de *locks* e a função de Hash utilizada, mas esta alteração somente funciona com leituras invisíveis, ou seja, somente funciona com *models* que utilizem como base o *model* `TransactionInvisible`. Para alterar o tamanho da tabela de locks, basta adicionar no arquivo `config` a entrada VERSION_LOCK_TABLE_SIZE seguida do tamanho desejado da tabela, como pode ser visto na Figura 3.1. Caso seja adicionada a entrada VERSION_LOCK_TABLE_SIZE 22, como no exemplo da Figura 3.1, a tabela de *locks* terá 2^{22} entradas. Aumentar o tamanho da tabela de *locks* leva a um melhor desempenho do sistema em alguns testes por causa da diminuição dos falsos conflitos. Modificar a função de hash utilizada é simples, veja o Apêndice A.4 para maiores detalhes.

3.3.2 Models Implementados

O STM Builder traz dois *models* bases implementados, que herdamos da SwissTM e EpochSTM para fazer a implementação inicial do sistema. Os *models* são:

- TransactionInvisible: Este é o *model* base do sistema com leituras invisíveis e corresponde à execução da SwissTM, tendo todos os métodos de execução da transação iguais ao da SwissTM. O *model* `TransactionInvisible` utiliza a técnica de detecção de conflitos *mixed invalidation*. Como na SwissTM, as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional ao número de seus abortos sucessivos. Este *model* pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT, RRT. E

pode ser utilizado com as combinações a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM, Passive + RRTwoPhaseT.

- TransactionEpoch: Este é o *model* base do sistema com leituras visíveis e corresponde a execução do EpochSTM, tendo todos os métodos de execução da transação iguais ao EpochSTM. O EpochSTM tenta implementar uma abordagem de leituras visíveis que não possui conjuntos locais de leitura e escrita, para reduzir as despesas gerais do sistema. Este *model* pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyEpoch. Após adaptar este *model* para funcionamento como um novo *model* base para o STM Builder, realizamos alguns testes com este *model* no STM Builder e também com a implementação original do EpochSTM, após encontrar problemas e em conversas com o autor do EpochSTM, ele nos aconselhou a não utilizar o algoritmo, porque não funciona perfeitamente. A adaptação do EpochSTM como um *model* base do STM Builder não foi desperdício de tempo, pois percebemos que é possível criar novos algoritmos como base, não somente a SwissTM.

3.4 Novas implementações realizadas no STM Builder

Nas próximas subseções serão apresentadas as novas implementações realizadas no STM Builder.

3.4.1 Novos Gerentes de Contenção

O STM Builder traz vários gerentes de contenção implementados, mas a Figura 3.4 não apresenta todos os gerentes de contenção do STM Builder por uma questão de economia de espaço. Os novos gerentes de contenção implementados são descritos abaixo:

- *Aggressive*: É a implementação do gerente *Aggressive* apresentado no Capítulo 2.
- *Polite*: É a implementação do gerente *Polite* apresentado no Capítulo 2.
- *Karma*: É a implementação do gerente *Karma* apresentado no Capítulo 2.
- Polka: É a implementação do gerente *Polka* apresentado no Capítulo 2.
- *Greedy*: É implementado da forma que foi proposto em [15], sua definição já foi apresentada no Capítulo 2. Considere as transações A e B, se A entra em conflito com B e o gerente de contenção *Greedy* é chamado e A aborta B, neste momento a implementação do *Greedy* necessita saber que foi A que causou o aborto de B para implementar o recuo da transação B.
- *GreedyTwoPhase*: O gerente *GreedyTwoPhase* é uma pequena adaptação do gerente *Greedy* para execução com o gerenciamento de contenção de duas fases.

- *GreedyT*: O gerente *GreedyT* é uma pequena variação do *Greedy* para utilizar o *model* TransactionInvisible. Considere as transações A e B, se A entra em conflito com B e o gerente de contenção é chamado e A aborta B, neste momento o que diferencia o *GreedyT* do *Greedy* original é que ele não necessita saber que foi A que causou o aborto de B para implementar o recuo da transação B.

- *RandomizedRounds* (RR): É implementado da forma que foi proposto em [32], e sua definição já foi apresentada no Capítulo 2. Considere as transações A e B, se A entra em conflito com B e o gerente de contenção RR é chamado e A aborta B, neste momento a implementação do RR necessita saber que foi A que causou o aborto de B para implementar o recuo da transação B.

- RRT: O gerente *RandomizedRoundsT* (RRT) é uma pequena variação do RR para utilizar os *models* ModelRRModified e TransactionInvisible. Considere as transações A e B, se A entra em conflito com B e o gerente de contenção é chamado e A aborta B, neste momento o que diferencia o RRT do RR original é que ele não necessita saber que foi A que causou o aborto de B para implementar o recuo da transação B.

- *RRTwoPhase*: O gerente *RRTwoPhase* é uma pequena adaptação do gerente RR para execução com o gerenciamento de contenção de duas fases.

- *RRTwoPhaseT*: O gerente *RRTwoPhaseT* apresenta algumas diferenças em comparação ao gerente *RRTwoPhase* para execução com os *models* ModelRRModified e TransactionInvisible. Considere as transações A e B, se A entra em conflito com B e o gerente de contenção é chamado e A aborta B, neste momento o que diferencia o *RRTwoPhaseT* do *RRTwoPhase* é que ele não necessita saber que foi A que causou o aborto de B para implementar o recuo da transação B.

3.4.2 Novos Models

O STM Builder traz vários *models* implementados, mas a Figura 3.4 não apresenta todos os *models* do STM Builder por uma questão de economia de espaço. Todos os novos *models* são desenvolvidos com base no `TransactionInvisible` e possuem esquema de detecção de conflitos *mixed invalidation*. Os *models* são descritos abaixo:

- ModelGreedy: Neste *model* transações que abortam devido a conflitos de escrita/escrita não são reiniciadas até que a transação que causou o aborto efetive ou aborte. A estratégia de recuo deste *model* é diferente para implementar totalmente a estratégia do gerente de contenção Greedy. Ele pode ser utilizado com o gerente de contenção de uma fase *Greedy* e com a combinação entre os gerentes Passive e GreedyTwoPhase que necessitam da utilização da função de transição TF1.

- ModelRR: Este *model* foi desenvolvido para implementar totalmente a estratégia do gerente de contenção *RandomizedRounds*, em que um número discreto uniformemente aleatório de cada transação é escolhido no início e após todo aborto e as

transações que abortam devido a conflitos de escrita/escrita não são reiniciadas até que a transação que causou o aborto efetive ou aborte. Ele pode ser utilizado com o gerente de contenção de uma fase RR (*RandomizedRounds*) e com a combinação entre os gerentes Passive e RRTwoPhase que necessitam da utilização da função de transição TF1.

- ModelRRModified: Ele é uma pequena modificação no *model* base `TransactionInvisible` em que o gerente *RandomizedRounds* escolhe o número discreto uniformemente aleatório de cada transação no início e após todo aborto e mantém o mecanismo em que as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional ao número de seus abortos sucessivos. Ele pode ser utilizado com o gerente de contenção de uma fase RRT (*RandomizedRoundsT*) e com a combinação entre os gerentes Passive e RRTwoPhaseT que necessitam da utilização da função de transição TF1.

- ModelRRPriority: Este *model* é similar ao ModelRR, se diferenciando somente no momento em que incorpora prioridade à transação que foi abortada, diminuindo sempre pela metade o intervalo em que o número discreto uniformemente aleatório é gerado. Ele pode ser utilizado com o gerente de contenção de uma fase RR (*RandomizedRounds*) e com a combinação entre os gerentes Passive e RRTwoPhase que necessitam da utilização da função de transição TF1.

- ModelRRPriorityModified: Este *model* é similar ao ModelRRModified, se diferenciando somente no momento em que incorpora prioridade a transação que foi abortada, diminuindo sempre pela metade o intervalo em que o número discreto uniformemente aleatório é gerado. Ele pode ser utilizado com o gerente de contenção de uma fase RRT (*RandomizedRoundsT*) e com a combinação entre os gerentes Passive e RRTwoPhaseT que necessitam da utilização da função de transição TF1.

- ModelNoWait: Este *model* não possui o mecanismo em que as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional ao número de seus abortos sucessivos. Neste *model*, após um aborto, a transação é reiniciada imediatamente. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT, Polite, Karma e Polka. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

- ModelExponentialWait: Neste *model* as transações que abortam devido a conflitos de escrita/escrita recuam por um período que aumenta exponecialmente conforme o número de abortos. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT, Polite, Karma e Polka. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

- ModelCTI: Este *model* implementa a invalidação no momento da efetivação com a heurística iWork, que é descrita na Subseção 3.4.3. Neste *model* as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional

ao número de seus abortos sucessivos. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

- ModelCTIiFair: Este *model* é desenvolvido com base no `ModelCTI`, mas implementa a invalidação no momento da efetivação com a heurística iFair, que é descrita na Subseção 3.4.3. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

- ModelCTIiPrio: Este *model* é desenvolvido com base no `ModelCTI`, mas implementa a invalidação no momento da efetivação com a heurística iPrio, que é descrita na Subseção 3.4.3. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

- ModelCTINW: Este *model* é desenvolvido com base no `ModelNoWait`, mas implementa a invalidação no momento da efetivação com a heurística iWork, que é descrita na Subseção 3.4.3. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT, Polite, Karma e Polka. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

- ModelCTIiFairNW: Este *model* é desenvolvido com base no `ModelCTINW`, mas implementa a invalidação no momento da efetivação com a heurística iFair, que é descrita na Subseção 3.4.3. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT, Polite, Karma e Polka. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

- ModelCTIiPrioNW: Este *model* é desenvolvido com base no `ModelCTINW`, mas implementa a invalidação no momento da efetivação com a heurística iPrio, que é descrita na Subseção 3.4.3. Ele pode ser utilizado com o gerenciamento de contenção de uma fase com os seguintes gerentes: *Passive*, *Agressive*, GreedyT, Polite, Karma e Polka. E pode ser utilizado com a combinação a seguir para o gerenciamento de contenção de duas fases junto com a função de transição TF1: Passive + GreedySwissTM.

3.4.3 Invalidação no momento da efetivação (*Commit-time invalidation*)

O artigo [14] apresenta uma implementação eficiente da técnica invalidação no momento da efetivação, uma estratégia de detecção de conflitos em que os conflitos transacionais são encontrados comparando a memória de uma transação que está efetivando

contra a memória de transações em execução. A invalidação no momento da efetivação difere da técnica mais comum utilizada que é a validação no momento da efetivação em que todos os conflitos de transações que estão efetivando com transações em execução são encontrados e resolvidos antes que a transação seja efetivada. Conflitos são enviados para o gerente de contenção (CM), o processo que decide quais as transações devem prosseguir, para a resolução. O CM utilizado na técnica de invalidação no momento de efetivação resolve os conflitos: (1) abortando todos os conflitantes com transações em execução, (2) abortando a transação que está efetivando, ou (3) paralisando a transação que está efetivando até que as transações em execução conflitantes tenham efetivado ou abortado. Através deste mecanismo, a invalidação no momento da efetivação pode notavelmente aumentar o *throughput* da transação, quando comparado com validação no momento da efetivação para cargas de trabalho com contenção.

Para compreender melhor as diferenças das técnicas de invalidação e validação no momento da efetivação, considere o cenário onde uma transação escreve na variável X e N transações posteriormente lêem o valor de X. O STM usando validação no momento da efetivação e detecção de conflito *lazy* na escrita, com sucesso valida a transação escritora na sua ação de efetivação. O escritor vai então atualizar X com o valor global e o número da versão e efetivar. No entanto, esse comportamento fará com que todos os N leitores abortem. Quando os leitores alcançarem suas ações de efetivação, eles serão obrigados a abortar porque sua visão de X será inconsistente com a memória principal, devido a efetivação da transação que escreveu em X. Assim, a validação no momento da efetivação elimina toda a concorrência entre os leitores e um escritor, que é um problema sério se N é grande, como é em grande parte das cargas de trabalho e sistemas.

Agora, considere a invalidação no momento da efetivação para o mesmo cenário anterior. Quando a transação que escreve em X chega na sua ação de efetivação, ela verifica todas as N transações em execução para encontrar conflitos. Cada conflito é enviado ao CM que, com base no número de rivais, que são transações de somente leitura, pode fazer uma decisão informada para abortar a transação de escrita e permitir a efetivação concorrente dos leitores de N. Quando N é grande, esse comportamento drasticamente aumenta o *throughput* da transação.

A InvalSTM é um STM que foi implementado no artigo [14] e implementa a invalidação no momento da efetivação para todos os conflitos. A InvalSTM também usa detecção de conflito de escrita *lazy*, possui leituras visíveis, é baseada em bloqueios e utiliza versionamento de dados por palavra.

Para a InvalSTM executar invalidação no momento da efetivação, suas transações devem ser impedidas de adicionar elementos novos na memória para o seu conjunto de leitura e escrita enquanto uma transação efetiva. Sem esta restrição, um elemento conflitante na memória pode ser adicionado ao conjunto de leitura e escrita de uma transação em execução depois dela ter sido considerada livre de conflitos. Para evitar esse comportamento indesejado, a InvalSTM associa um bloqueio em cada transação. Antes de executar a invalidação no momento da efetivação, a InvalSTM adquire os bloqueios transacionais de todas as transações em execução para assegurar que a fase de invalidação será realizada sem modificações estranhas nos conjuntos de leitura e escrita de transações em execução.

Além de um bloqueio por transação, a InvalSTM usa dois bloqueios globais: um bloqueio na efetivação e outro na execução. O bloqueio na efetivação limita a fase de efetivação para uma única transação. O bloqueio em execução é usado para limitar as alterações da lista de transações em execução para uma única thread.

O processo de invalidação no momento da efetivação, começa na InvalSTM quando os bloqueios de efetivação e execução são adquiridos. Isso garante que nenhuma outra transação pode efetivar ou ser iniciada enquanto uma transação está efetivando. Em seguida, os bloqueios de efetivação e execução associados às transações são adquiridos em um ordem sequencial para evitar *deadlock*. Em seguida, são identificados os conflitos que a transação que está efetivando têm com as transações em execução. Se os conflitos existirem, ao CM é enviado o lote de conflitos e é permitido que ele tome a decisão sobre quais transações são abortadas ou paradas.

Um detalhe importante do projeto da InvalSTM é que as transações em execução podem avançar durante todo o processo de invalidação no momento da efetivação. As duas exceções são (1) transações não podem simultaneamente efetivar enquanto outra transação já está efetivando e (2) elas não podem adicionar novos elementos de memória em seus conjuntos de leitura e escrita.

Na InvalSTM foram testadas três variantes do CM: iFair (invalidação justa), iPrio (invalidação priorizadas) e iAggr (invalidação agressiva).

O iAggr garante que a primeira transação que entra na fase de efetivação termine. Ele demonstra como a invalidação no momento da efetivação executa quando ela não usa as informações sobre os conflitos para fazer decisões informadas. Em outras palavras, iAggr capta a diferença da operação de detecção de conflitos invalidação e validação.

O iPrio associa uma prioridade a cada transação. A prioridade da transação é aumentada cada vez que aborta e é reiniciada a cada efetivação. Uma transação pode efetivar, se tiver a mais alta prioridade entre todas as transações conflitantes em execução. A transação também pode efetivar se ela tem o tamanho do conjunto de leitura maior que todas as transações em execução ou seus conjuntos de leitura e escrita tem tamanho maior do que a média do tamanho do conjunto de leitura e escrita de todas as transações conflitantes mais suas prioridades acumuladas.

O iFair associa uma prioridade a cada transação e levanta e redefine a prioridade da transação da mesma maneira que o iPrio. Ao contrário do iPrio, uma transação pode efetivar se o tamanho do seu conjunto de leitura e escrita é maior que a média ponderada do tamanho da leitura e da prioridade das transações em execução. Uma transação também pode efetivar, se o tamanho do seu conjunto de leitura e escrita é maior do que o tamanho da leitura de qualquer transação em execução. Além disso, o tamanho do conjunto de leitura das transações em execução é 10^2 x maior do que o tamanho do conjunto de leitura da transação que está efetivando e sua prioridade é 2^3 x maior, iFair irá abortar a transação que está efetivando em favor da prioridade mais elevada, maior na transação em execução.

Implementamos a técnica de invalidação no momento da efetivação no STM Builder, pois em testes realizados com o InvalSTM, mesmo com toda a sobrecarga adicionada pelos

bloqueios, o STM foi mais do que 3x mais rápido do que a TL2 para certas cargas de trabalho com alta contenção [14].

Implementação da Invalidação no momento da efetivação no STM Builder

Implementar a Invalidação no momento da efetivação da mesma maneira que a InvalSTM, seria totalmente inviável. Adicionar detecção de conflito de escrita lazy, leituras visíveis e todos os bloqueios que foram utilizados, traria uma grande perda de desempenho ao novo *model* do STM Builder.

Decidimos implementar no STM Builder, um *model* onde a ideia geral de invalidação é utilizada no momento da efetivação da transação. No STM Builder, no momento em que a transação está efetivando, ela verifica se o conjunto de leitura das demais transações em execução possui alguma posição de memória em que o bloqueio de escrita é da transação que está efetivando, ou seja, verifica se há um conflito w/r. Caso ocorra esses conflitos, conforme os dados coletados das transações conflitantes, um mecanismo decide se a transação que está efetivando irá efetivar realmente ou abortar. No STM Builder esta extensão da invalidação nunca faz com que a transação que está efetivando espere pelas demais transações conflitantes.

Transações de somente leitura, como na InvalSTM, não utilizam o mecanismo de invalidação e realizam zero operações de detecção de conflitos.

A implementação do mecanismo de invalidação foi realizada no ModelCTI, onde CTI é a abreviação de *Commit-time invalidation*. No ModelCTI cada transação passou a ter uma cópia do seu conjunto de leitura local e um inteiro representado o tamanho da cópia do conjunto de leitura. A cópia do conjunto de leitura existe para que não ocorra o problema de ter que parar todas as transações, no momento em que uma transação que está efetivando faz a leitura do conjunto de leitura das demais transações em busca de conflitos.

Decidimos implementar a cópia do conjunto de leitura como um vetor fixo com 10.000 posições. A utilização de um vetor trouxe a vantagem de não utilizar em nenhum momento do programa novos bloqueios, mas limitou a cópia do conjunto de leitura em 10.000 posições, que é um tamanho que foi ultrapassado poucas vezes durante os testes que realizamos no Capítulo 4.

Toda transação realiza a leitura de uma posição de memória utilizando o método `lerPalavra()` (linha 1 da Figura 3.5). A transação lê o dado de uma posição de memória (linha 2 da Figura 3.5), a transação que está lendo adiciona esta posição de memória no conjunto de leitura (linha 3 da Figura 3.5) e também na cópia do conjunto de leitura, mas antes de adicionar na cópia do conjunto de leitura ela verifica se o tamanho da cópia do conjunto de leitura não ultrapassou o limite de 10.000 posições (linha 5 da Figura 3.5). Após a verificação é adicionada a nova leitura na cópia do conjunto de leitura (linha 6 da Figura 3.5) e posteriomente é incrementado o tamanho do conjunto (linha 7 da Figura 3.5).

A utilização de um vetor deixou a implementação muito simples, de modo que a

limpeza necessária da cópia do conjunto de leitura quando a transação efetiva ou aborta, seja somente zerar o tamanho da cópia do conjunto de leitura.

A transação que está efetivando executa o método `cti` (linha 11 da Figura 3.5), que percorre a cópia do conjunto de leitura de todas as transações em busca de conflitos (linha 12 da Figura 3.5). Considere que a transação A está efetivando e vai ler a cópia do conjunto de leitura da transação B que está executando normalmente. A transação A realiza a detecção de possíveis conflitos comparando se a transação B leu alguma posição de memória em que o bloqueio de escrita é da transação A que está efetivando (linha 14 da Figura 3.5). Quando um conflito é encontrado (linha 14 da Figura 3.5), os dados necessários para os mecanismo de decisão são adquiridos (linha 15 da Figura 3.5). Após a verificação completa de conflitos com a transação B terminar, a transação A realiza os mesmos passos anteriores com todas as transações em execução em busca de conflitos. E após passar por todas, decide se vai efetivar ou abortar ela mesma (linha 19 da Figura 3.5). Em nenhum momento a transação A interfere na execução das demais transações.

O mecanismo de invalidação do STM Builder pode não detectar todos os conflitos existentes com a transação que está efetivando, porém faz uma predição e posteriormente aplica uma heurística para decidir.

Utilizamos três heurísticas diferentes, iFair (ModelCTIiFair), iPrio (ModelCTIiPrio) e iWork (ModelCTI). As heurrísticas iFair e iPrio são as mesmas do artigo [14] e apresentadas anteriormente. E criamos uma nova heurística, que chamamos de iWork, em que se a soma do tamanho do conjunto de leitura mais o de escrita, multiplicada pela quantidade de abortos mais um, for menor que o número de conflitos encontrados, a transação é abortada.

Nas heurísticas iFair e iPrio pode não ser necessário passar por todas as posições utilizadas do vetor com a cópia do conjunto de leitura das transações em execução, pois quando é encontrado um conflito, basta recolher os dados necessários da transação conflitante e seguir para a transação seguinte. Já na heurística iWork é necessário passar por todas posições utilizadas, pois nesta heurística é contabilizado o total de conflitos.

```
1  lerPalavra(endereço){
2      dado = lerDado(endereço);
3      log_leitura.adiciona(endereço);
4
5      se tamanho_copia_log_leitura < 10000 então{
6          copia_log_leitura[tamanho_copia_log_leitura] = endereço;
7          tamanho_copia_log_leitura++;
8      }
9  }
10
11 cti(){
12     para cada transação T do sistema{
13         para j = 0 até j < T.tamanho_copia_log_leitura passo j++ faça{
14             se foi detectado um conflito com a transação que esta efetivando e T.copia_log_leitura[j] então{
15                 coleta dados de T;
16             }
17         }
18     }
19     retorna decisão();
20 }
```

Figura 3.5: Pseudocódigo da Implementação do *Commit-time invalidation* no STM Builder.

Capítulo 4

Resultados das Comparações

Neste Capítulo apresentamos o framework de testes, os testes e as comparações realizados com STM Builder e a SwissTM.

4.1 Framework de testes

Após a realização de alguns testes com o STM Builder utilizando o Microbenchmark de árvore rubro-negra, o STMBench7 e o STAMP, foi gasto um grande tempo entendendo os benchmarks, aprendendo a executar cada opção dos benchmarks e implementando scripts para execução automatizada dos testes. Scripts para execução automatizada são necessários, pois são realizadas milhares de execuções e é custoso fazer uma a uma manualmente. Além dos scripts, foi necessário implementar um programa para interpretar centenas de arquivos de saídas, para podermos gerar o gráficos necessários para nossa análise.

Adicionar um novo gerente de contenção no STM Builder é muito simples. Vamos supor que para um gerente simples, você gaste algumas horas para implementar e ver que realmente ele funciona. Para avaliar realmente o desempenho deste gerente é necessário realizar todos os procedimentos já apresentados acima, e isso levaria alguns dias de implementação e outros dias com os testes rodando. Seria uma grande perda de tempo para testar um simples gerente de contenção.

Para facilitar testes e ter maior produtividade testando ideias novas, implementamos um framework de testes acoplado ao STM Builder, que também pode ser utilizado com outros STMs. Este framework de testes consiste em executar automaticamente os testes no Microbenchmark de árvore rubro-negra, no STMBench7 e no STAMP. Este framework de testes é muito simples de usar, pois basta indicar no arquivo de entrada o que se deseja executar, a quantidade máxima de threads utilizada em cada execução, quantidade de repetições desejada para os testes e quais benchmarks utilizar (os três juntos, ou por exemplo STMBench7 e o STAMP, ou somente o STAMP).

O framework de testes é uma ferramenta muito útil e tem como objetivo facilitar a

construção e testes de ideias para STMs.

4.1.1 Detalhes do Framework de testes

A implementação do framework foi realizada em quatro módulos distintos, como mostrado na Figura 4.1. Durante a implementação do framework foi necessário padronizar as saídas dos testes e alguns outros detalhes que serão abordados mais adiante.

Primeiro Módulo

O primeiro módulo é um programa interpretador do arquivo de entrada (implementado no Reader.h e Reader.cpp), como mostrado na Figura 4.1. Ele recebe como entrada o arquivo *input* (Figura 4.2-a), e gera como saída os arquivos de configuração necessários para a execução do STM Builder e o arquivo *output* (Figura 4.2-b). O arquivo *output* é a entrada dos demais módulos e é gerado somente neste primeiro módulo. Mais detalhes do funcionamento dos arquivos *input* e *output* (Figura 4.2-a e b respectivamente) podem ser vistos no Apêndice A.5.

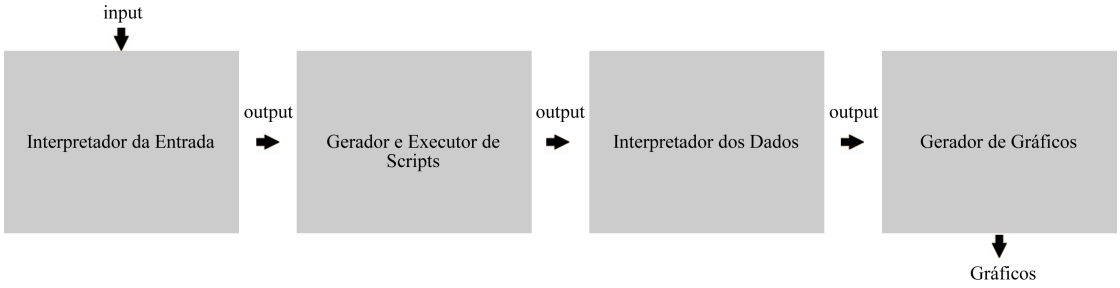


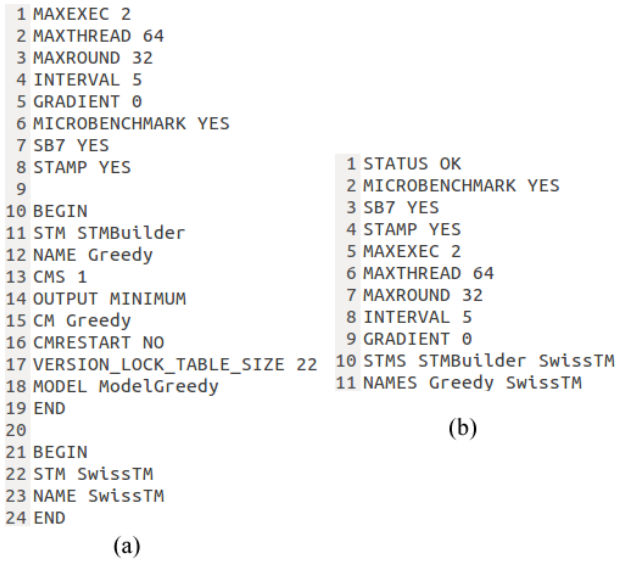
Figura 4.1: Framework de testes.

Segundo Módulo

O segundo módulo é um programa gerador e executor de scripts (implementado no Script.h e Script.cpp), como mostrado na Figura 4.1. Este programa recebe como entrada o arquivo *output*, gera todos os shell-scripts necessários para os testes e executa todos os scripts dos testes. O tempo de execução deste módulo é proporcional ao número de testes que se deseja executar.

Terceiro Módulo

O terceiro módulo é um programa interpretador dos dados (implementado no Data.h e Data.cpp) que recebe como entrada o arquivo *output* e interpreta os dados gerados pelos testes, pois ele conhece a localização dos dados. Este módulo verifica os arquivos de



```
MAXEXEC 2
MAXTHREAD 64
MAXROUND 32
INTERVAL 5
GRADIENT 0
MICROBENCHMARK YES
SB7 YES
STAMP YES

BEGIN
STM STMBuilder
NAME Greedy
CMS 1
OUTPUT MINIMUM
CM Greedy
CMRESTART NO
VERSION_LOCK_TABLE_SIZE 22
MODEL ModelGreedy
END

BEGIN
STM SwissTM
NAME SwissTM
END
```

(a)

```
STATUS OK
MICROBENCHMARK YES
SB7 YES
STAMP YES
MAXEXEC 2
MAXTHREAD 64
MAXROUND 32
INTERVAL 5
GRADIENT 0
STMS STMBuilder SwissTM
NAMES Greedy SwissTM
```

(b)

Figura 4.2: (a) Exemplo do arquivo *input*, (b) Exemplo do arquivo *output*.

saída dos testes, e caso seja encontrado algum erro, uma mensagem de erro é impressa no terminal que está executando o framework e uma saída negativa é plotada no gráfico para fácil identificação do erro, pois as métricas utilizadas somente utilizam números positivos.

O módulo também faz os cálculos para gerar como saída as médias das amostras com o intervalo de confiança de 95%. Os vários arquivos de saída gerados pelo terceiro módulo seguem o padrão que o *gnuplot* [13] exige.

Para que o terceiro módulo consiga interpretar os dados dos testes, foi necessário padronizar as saídas dos testes do Microbenchmark de árvore rubro-negra, do STMBench7 e do STAMP (para maiores detalhes veja o Apêndice A.6).

Quarto Módulo

O quarto módulo é um programa gerador de gráficos (implementado no Graphics.h e Graphics.cpp) que recebe como entrada o arquivo *output* e gera os gráficos para a avaliação dos resultados dos testes. Este módulo utiliza o *gnuplot* [13] para gerar os gráficos. Ele primeiramente cria os arquivos de configuração necessários para o *gnuplot*, e depois executa o *gnuplot* com estes arquivos de configuração e os dados do módulo anterior. São gerados gráficos de barras para a avaliação, que contêm plotado a média da amostra com o intervalo de confiança de 95%.

Para o Microbenchmark são gerados gráficos com a métrica *Commits*, que é o número de efetivações que cada execução teve. Decidimos utilizar esta métrica pois facilita a visualização dos resultados, pois quanto maior o número de efetivações, melhor o desempenho. Nos testes com o Microbenchmark de Árvore rubro-negra do framework de testes, o tempo de execução escolhido é de 10 segundos para cada variação no número de threads e atualização. No Microbenchmark são gerados três gráficos: com 20% de atualização (onde 20% das operações são inserções e remoções, e 80% são buscas), com 50% de atualização

Aplicação	Parâmetros
Bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1
Genome	-g16384 -s64 -n16777216
Intruder	-a10 -l128 -n262144 -s1
Labyrinth	-i random-x512-y512-z7-n512.txt
SSCA2	-s20 -i1.0 -u1.0 -l3 -p3
Yada	-a15 -i ttimeu1000000.2
Kmeans Low	-m40 -n40 -t0.00001 -i random-n65536-d32-c16.txt
Kmeans High	-m15 -n15 -t0.00001 -i random-n65536-d32-c16.txt
Vacation Low	-n2 -q90 -u98 -r1048576 -t4194304
Vacation High	-n4 -q60 -u90 -r1048576 -t4194304

Tabela 4.1: Parâmetros de entrada do STAMP.

(onde metade das operações são inserções e remoções, e a outra metade são buscas) e com 100% de atualização (onde todas as operações são inserções e remoções).

Para o STMBench7 são gerados gráficos com a métrica *op/s* (*throughput*), que significa o total de operações realizadas com sucesso, dividido pelo tempo gasto na execução. Decidimos utilizar esta métrica pois é a métrica padrão do STMBench7 e facilita a visualização dos resultados. Quanto maior for o número de *op/s*, melhor o desempenho da implementação testada. No STMBench7 são gerados três gráficos: com a carga de trabalho dominada por leitura (*Read Dominated*), onde 90% das operações são somente de leitura; com a carga de trabalho dominada por leitura-escrita (*Read-Write Dominated*), onde 60% das operações são somente de leitura; e com a carga de trabalho dominada por escrita (*Write Dominated*), onde 10% das operações são somente de leitura.

Para o STAMP são gerados gráficos com a métrica *Runtime(s)* (Tempo de execução), que significa o tempo total gasto na execução em segundos. Decidimos utilizar esta métrica pois é uma métrica justa e facilita a visualização dos resultados. Quanto menor for o tempo de execução, melhor o desempenho da implementação testada, diferente dos gráficos do Microbenchmark e do STMBench7. No STAMP são gerados 10 gráficos, um para cada variação da carga de trabalho, os parâmetros de entrada utilizados nos testes com o STAMP pode ser visualizado na Tabela 4.1.

Para cada teste do Microbenchmark, STMBench7 e STAMP plotamos outro gráfico com o número de abortos.

Execução do Framework de testes

Para executar o framework de testes basta configurar o arquivo *input*, compilar os STMs e os Benchmarks, sendo que para cada STM é necessário uma cópia separada do STMBench7 e do STAMP e é necessário seguir o padrão de nomeação das pastas, que é benchmark concatenado com stm (ex: STMBuilder, sb7STMBuilder, stampSTMBuilder). Após tudo compilado, basta executar o script *runningframework* que irá executar todos os módulos.

Todos os módulos do framework de testes podem ser executados separadamente. Esta divisão em módulos foi realizada com o intuito de facilitar o reaproveitamento de testes e de facilitar a extensão do framework de testes adicionando novos benchmarks.

A execução de cada módulo separadamente considera que todos os arquivos necessários para a execução de cada módulo individualmente existem, estão no local correto e com nomes corretos. Para executar os módulos separadamente, veja o Apêndice A.7.

4.2 Resultados e Comparações

Esta seção apresenta os resultados e comparações dos testes realizados com a SwissTM e o STM Builder.

Os testes foram realizados utilizando o Microbenchmark de árvore rubro-negra, o STMBench7 e o STAMP, sendo executados em um computador com dois processadores Intel Quad Core Xeon E5504 de 2.0 GHz (com 256 KB de cache L1, 1 MB de cache L2 e 4 MB de cache L3 cada), sistema operacional Linux Ubuntu versão 11.10 e com 16 GB de memória RAM.

Cada teste foi executado 20 vezes e o que é apresentado no trabalho é a média entre estas 20 execuções. Isto é realizado para que comportamentos inesperados do sistema operacional não causem influência nos resultados finais obtidos. Nos testes o número de threads que as implementações poderiam ter para executar é de 2^i com i variando de 0 a 3.

Todos os gráficos dos testes apresentam os resultados com o intervalo de confiança de 95% aplicado.

Todos os testes realizados e os gráficos com os resultados foram produzidos com a utilização do Framework de Testes acoplado ao STM Builder.

Foram realizados testes com as 13 implementações detalhadas abaixo:

- SwissTM: Nos testes utilizamos a SwissTM de 15/08/2011, a versão mais atual do STM. Utilizamos o algoritmo padrão da SwissTM, onde o esquema de detecção de conflitos é o *mixed invalidation* e utiliza o gerente de contenção de duas fases. Na primeira fase a transação utiliza o gerente de contenção *Passive* (Passivo) e após atingir um limite no número de escritas, o gerente de contenção troca para a segunda fase, onde a transações são mais complexas e utilizam o gerente de contenção *Greedy*. Além disso, as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional ao número de seus abortos sucessivos.

- STM Builder - Two Phase: Esta versão executada no STM Builder representa a SwissTM, e utiliza o *model* TransactionInvisible e o gerente de contenção de duas fases. Na primeira fase a transação utiliza o gerente de contenção *Passive* (Passivo) e na segunda o gerente *GreedySwissTM*. Esta configuração do STM Builder traz a Função de Transição TF1 que após a transação atingir um limite no número de

escritas troca o gerente de contenção dinamicamente para a segunda fase. A Figura 4.3-a traz o arquivo de configuração utilizado neste teste.

- STM Builder - RR Two PhaseT: Esta versão executada no STM Builder utiliza o *model* ModelRRModified e o gerente de contenção de duas fases. Na primeira fase a transação utiliza o gerente de contenção *Passive* (Passivo) e na segunda o gerente *RRTwoPhaseT*. Esta configuração do STM Builder utiliza também a Função de Transição TF1. A Figura 4.3-b traz o arquivo de configuração utilizado neste teste.

- STM Builder - RRT: Esta versão executada no STM Builder utiliza o *model* ModelRRModified e o gerente de contenção de uma única fase, o gerente *RandomizedRoundsT* (RRT). A Figura 4.3-c traz o arquivo de configuração utilizado neste teste.

- STM Builder - GreedyT: Esta versão executada no STM Builder utiliza o *model* TransactionInvisible e o gerente de contenção de uma única fase, o gerente *GreedyT*. A Figura 4.3-d traz o arquivo de configuração utilizado neste teste.

- STM Builder - Greedy: Esta versão executada no STM Builder utiliza o *model* ModelGreedy e o gerente de contenção de uma única fase, o gerente *Greedy*. A Figura 4.3-e traz o arquivo de configuração utilizado neste teste.

- STM Builder - Greedy-EW: Esta versão executada no STM Builder utiliza o *model* ModelExponentialWait e o gerente de contenção de uma única fase, o gerente *GreedyT*. A Figura 4.3-f traz o arquivo de configuração utilizado neste teste.

- STM Builder - Greedy-NW: Esta versão executada no STM Builder utiliza o *model* ModelNoWait e o gerente de contenção de uma única fase, o gerente *GreedyT*. A Figura 4.3-g traz o arquivo de configuração utilizado neste teste.

- STM Builder - Polka: Esta versão executada no STM Builder utiliza o *model* ModelNoWait e o gerente de contenção de uma única fase, o gerente *Polka*. A Figura 4.3-h traz o arquivo de configuração utilizado neste teste.

- STM Builder-CTI-iWork: Esta versão executada no STM Builder utiliza o *model* ModelCTI e o gerente de contenção de uma única fase, o gerente *GreedyT*. A Figura 4.3-i traz o arquivo de configuração utilizado neste teste.

- STM Builder-CTI-iFair: Esta versão executada no STM Builder utiliza o *model* ModelCTIiFair e o gerente de contenção de uma única fase, o gerente *GreedyT*. A Figura 4.3-j traz o arquivo de configuração utilizado neste teste.

- STM Builder-CTI-iPrio: Esta versão executada no STM Builder utiliza o *model* ModelCTIiPrio e o gerente de contenção de uma única fase, o gerente *GreedyT*. A Figura 4.3-k traz o arquivo de configuração utilizado neste teste.

- SwissTM-CTI-iWork: Implementação da invalidação no momento da efetivação diretamente na SwissTM de 15/08/2011, a versão mais atual do STM. Utilizamos o algoritmo padrão da SwissTM, onde o esquema de detecção de conflitos é o *mixed invalidation* e utiliza o gerente de contenção de duas fases. Na primeira fase a

transação utiliza o gerente de contenção *Passive* (Passivo) e após atingir um limite no número de escritas, o gerente de contenção troca para a segunda fase, onde a transações são mais complexas e utilizam o gerente de contenção *Greedy*. Além disso, as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional ao número de seus abortos sucessivos.

```
1 CMS 2
2 OUTPUT MINIMUM
3 CM Passive
4 TF TF1
5 CM GreedySwissTM
6 CMRESTART NO
7 VERSION_LOCK_TABLE_SIZE 22
8 MODEL TransactionInvisible
```

(a)

```
1 CMS 2
2 OUTPUT MINIMUM
3 CM Passive
4 TF TF1
5 CM RRTwoPhaseT
6 CMRESTART NO
7 VERSION_LOCK_TABLE_SIZE 22
8 MODEL ModelRRModified
```

(b)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM RRT
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelRRModified
```

(c)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  GreedyT
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL TransactionInvisible
```

(d)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  Greedy
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelGreedy
```

(e)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  GreedyT
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelExponentialWait
```

(f)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  GreedyT
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelNoWait
```

(g)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  Polka
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelNoWait
```

(h)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  GreedyT
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelCTI
```

(i)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  GreedyT
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelCTIiFair
```

(j)

```
1 CMS 1
2 OUTPUT MINIMUM
3 CM  GreedyT
4 CMRESTART NO
5 VERSION_LOCK_TABLE_SIZE 22
6 MODEL ModelCTIiPrio
```

(k)

Figura 4.3: Exemplo dos arquivos de configuração: (a) STM Builder - Two Phase, (b) STM Builder - RR Two PhaseT, (c) STM Builder - RRT, (d) STM Builder - GreedyT, (e) STM Builder - Greedy, (f) STM Builder - Greedy-EW, (g) STM Builder - Greedy-NW, (h) STM Builder - Polka, (i) STM Builder-CTI-iWork, (j) STM Builder-CTI-iFair e (k) STM Builder-CTI-iPrio.

4.2.1 Diferença de desempenho entre a SwissTM e o STM Builder

O STM Builder possui uma forma de execução que é totalmente baseada na SwissTM, o STM Builder - Two Phase. Para mostrar a diferença de desempenho entre as duas aplicações e apresentar a sobrecarga adicionada pelo STM Builder, esta subseção faz a comparação entre a SwissTM e o STM Builder - Two Phase.

Microbenchmark

Fica evidente nos gráficos das Figuras 4.4-(a, b e c) que a SwissTM foi bem mais rápida que o STM Builder - Two Phase, alcançando o seu pico de desempenho com 8 threads, onde chegou a ser mais de 20 vezes mais rápida a execução STM Builder - Two Phase em todos os testes do Microbenchmark. Atribuimos esta grande diferença de desempenho entre a SwissTM e o STM Builder - Two Phase nos testes com o microbenchmark à sobrecarga adicionada pelo STM Builder, por causa do uso constante de objetos alocados dinamicamente e do acoplamento dinâmico na chamada de métodos. Nos testes realizados

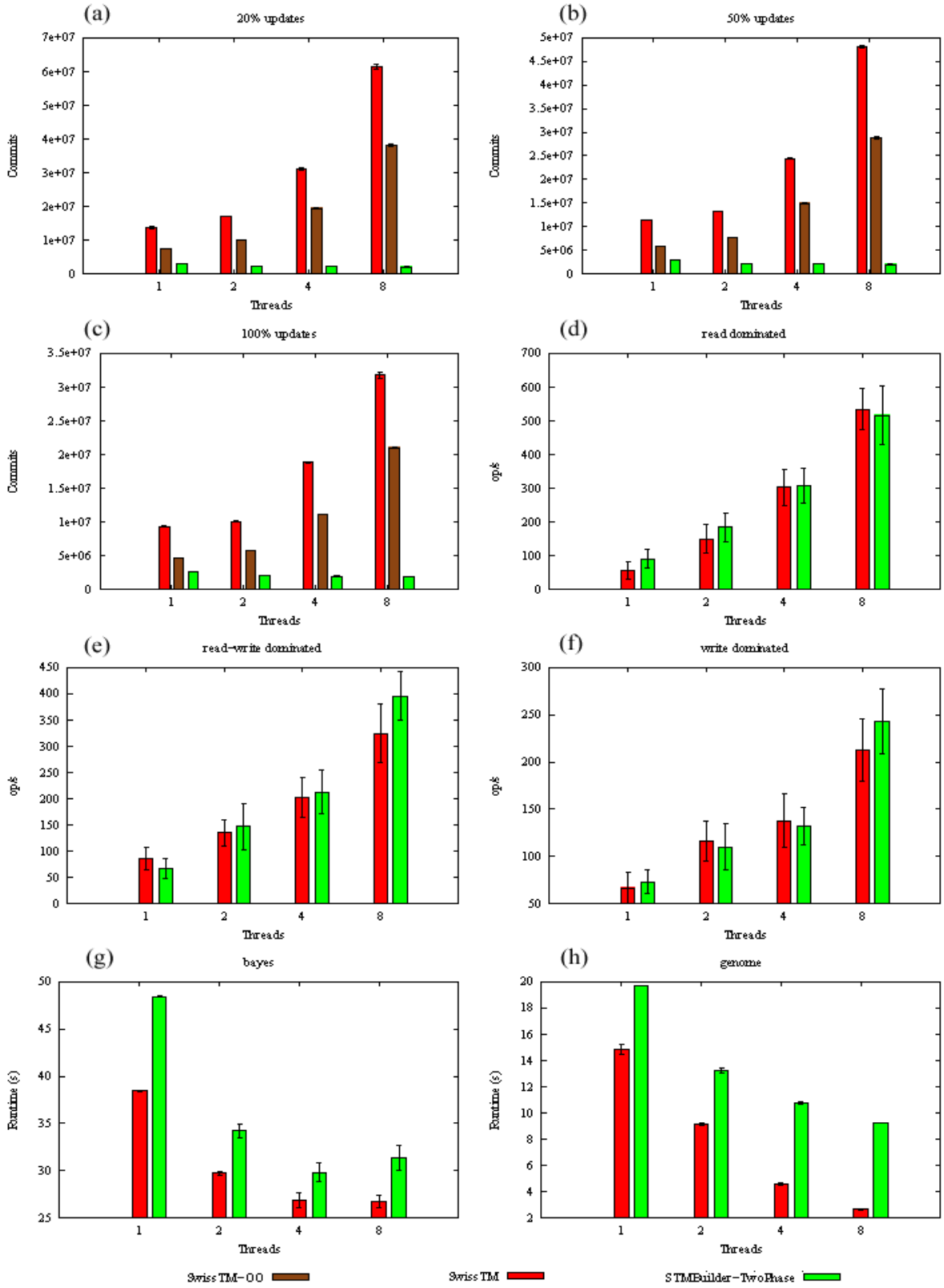


Figura 4.4: Gráficos com os resultados dos testes da diferença de desempenho entre a SwissTM e o STM Builder. Microbenchmark de Árvore rubro-negra: (a) 20%, (b) 50% e (c) 100% de atualização. STMBench7: (d) *Read*, (e) *Read-Write* e (f) *Write Dominated*. STAMP: Aplicações (g) Bayes e (h) Genome.

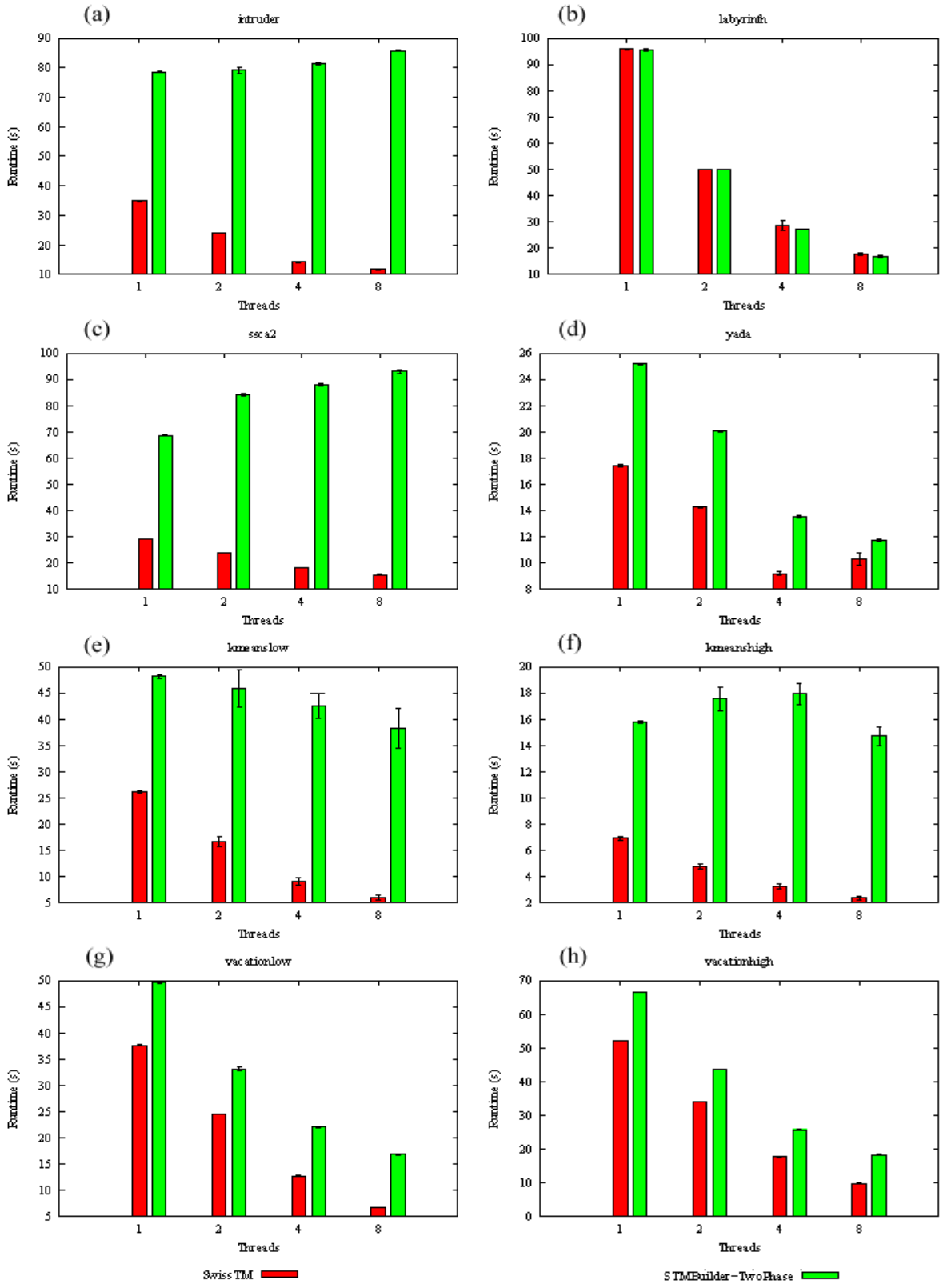


Figura 4.5: Gráficos com os resultados dos testes da diferença de desempenho entre a SwissTM e o STM Builder. STAMP: Aplicações (a) Intruder, (b) Labyrinth, (c) SSCA2, (d) Yada, (e) Kmeans Low, (f) Kmeans High, (g) Vacation Low e (h) Vacation High.

com o microbenchmark o efeito desta sobrecarga ficou bem evidente porque as transações são curtas.

Para verificar se a grande diferença de desempenho foi causada pelo grande uso de orientação a objetos, realizamos uma simples modificação na SwissTM, tornando alguns métodos virtuais como no STM Builder - Two Phase, desta maneira a SwissTM passa a ter um pouco de sobrecarga com acoplamento dinâmico na chamada de métodos. Esta pequena modificação da SwissTM, que chamamos de SwissTM-OO, foi cerca de 60% mais lenta que a SwissTM com 8 threads em todos os testes do Microbenchmark. Na implementação SwissTM-OO não foi colocada toda a sobrecarga existente no STM Builder - Two Phase, colocamos somente um pequeno recurso que utilizamos para comprovar que o uso de orientação a objetos degrada o desempenho nos testes com microbenchmarks.

STMBench7

Nos gráficos do STMBench7 (Figuras 4.4-(d, e e f)) fica claro que conforme o número de threads, escritas e a concorrência aumenta, o desempenho do STM Builder - Two Phase melhorou em relação a SwissTM. A sobrecarga anteriormente apresentada no Microbenchmark diminuiu conforme a complexidade do testes aumentaram no STMBench7, mostrando que o STM Builder - Two Phase possui um desempenho bom quando executa aplicações mais complexas.

Avaliando as médias dos gráficos das Figuras 4.4-(d, e e f) e considerando as intersecções entre os intervalos de confiança, o desempenho da SwissTM e do STM Builder - Two Phase são similares no STMBench7.

STAMP

Existe uma grande diferença de desempenho entre a SwissTM e o STM Builder - Two Phase, o que pode ser visualizado facilmente nos gráficos (Figuras 4.4-(g e h) e 4.5(a, c, d, e, f, g e h)). Somente na aplicação Labyrinth (gráfico da Figura 4.5-b) o STM Builder - Two Phase obteve um bom desempenho, sendo similar a SwissTM.

As aplicações do STAMP com transações longas, com exceção da aplicação Labyrinth, foram as que obtiveram a menor diferença de desempenho entre a SwissTM e o STM Builder - Two Phase. A SwissTM obteve a maior diferença de desempenho sendo 26% mais rápida que o STM Builder - Two Phase na aplicação Bayes (gráfico da Figura 4.4-g) com uma thread. E na aplicação Yada (gráfico da Figura 4.5-d) a SwissTM foi 47% mais rápida com 4 threads.

As aplicações com transações de tamanho médio tiveram uma diferença de desempenho mediana. A SwissTM foi no melhor caso 3,5 vezes mais rápida que o STM Builder - Two Phase na aplicação Genome (gráfico da Figura 4.4-h) com 8 threads. Nas aplicações Vacation Low e Vacation high (gráficos das Figuras 4.5-(g e h) respectivamente) a SwissTM foi no melhor caso 2,5 vezes e 85% mais rápida respectivamente com 8 threads.

O STM Builder - Two Phase possui seu pior desempenho quando as transações são

curtas. A SwissTM obteve a maior diferença de desempenho nas aplicações com transações curtas com 8 threads, onde foi 7,8 vezes mais rápida na aplicação Intruder, 6 vezes na aplicação SSCA2, 6,3 vezes na aplicação Kmeans Low e 7,4 vezes na Kmeans High (gráficos das Figuras 4.5(a, c, e e f) respectivamente).

Após a avaliação e comparações entre a SwissTM e o STM Builder - Two Phase, podemos chegar a conclusão de que o STM Builder possui uma grande sobrecarga em comparação a SwissTM, por causa do uso maior de orientação a objetos. O uso constante de objetos alocados dinamicamente e do acoplamento dinâmico na chamada de métodos do STM Builder fez com que ele fosse mais lento que a SwissTM na maioria dos testes. Nos testes com o microbenchmark, o STM Builder - Two Phase teve um péssimo desempenho, mas conforme a complexidade dos testes aumentaram, no STAMP e STMBench7, o STM Builder - Two Phase passou a ter um desempenho melhor.

Com aplicações com transações longas, a sobrecarga interfere menos nos resultados, a diferença de desempenho entre o STM Builder e a SwissTM é menor, chegando até ser nula como nos testes do STMBench7 e da aplicação Labyrinth do STAMP.

Nos testes fica comprovado que o STM Builder - Two Phase tem um desempenho ruim no cenário em que as transações são curtas. Este fato é comprovado nos testes com as aplicações Kmeans, SSCA2 e Intruder do benchmark STAMP e os testes do microbenchmark de árvore rubro negra.

4.2.2 Greedy x RandomizedRounds e gerenciamento de contenção de duas fases

Esta subseção apresenta os resultados e comparações dos testes realizados com o STM Builder - Two Phase, STM Builder - RR Two PhaseT, STM Builder - GreedyT e STM Builder - RRT. O objetivo desta subseção é mostrar:

1. Se existe diferença de desempenho do gerenciamento de contenção de uma e duas fases.
2. A diferença de desempenho entre os gerentes de contenção *Greedy* e *Randomized-Rounds* (STM Builder - GreedyT, STM Builder - RRT).
3. A diferença de desempenho entre os gerentes de contenção de duas fases *Greedy* e *RandomizedRounds* (STM Builder - Two Phase, STM Builder - RR Two PhaseT).

Microbenchmark

Os gráficos das Figuras 4.6-(a, b e c) mostram os resultados dos testes com o Microbenchmark. Avaliando os gráficos e considerando as intersecções entre os intervalos de confiança, podemos perceber que o desempenho das execuções com gerenciamento de contenção de uma fase (STM Builder - GreedyT e STM Builder - RRT) e duas fases (STM

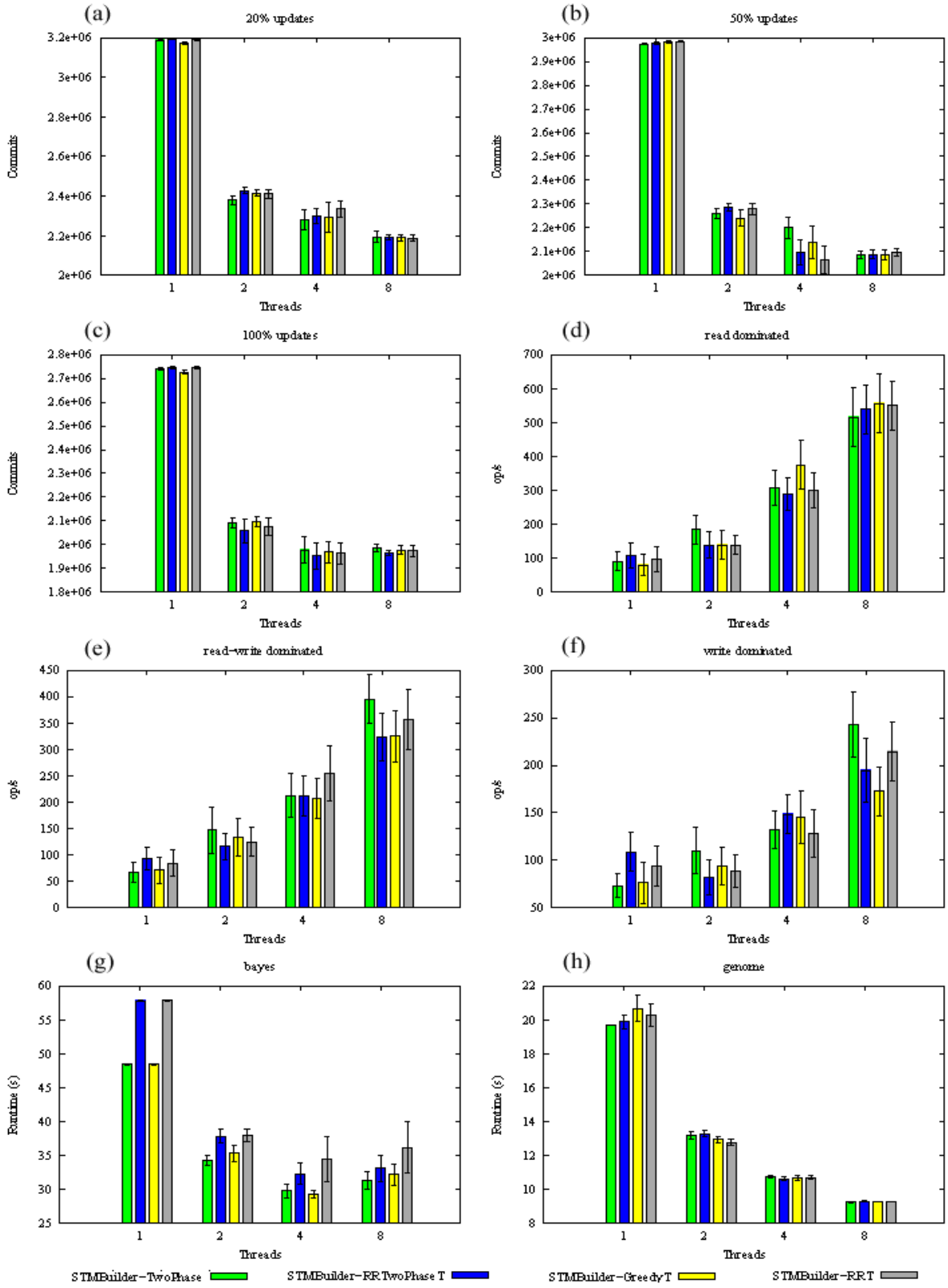


Figura 4.6: Gráficos com os resultados dos testes do gerenciamento de contenção. Microbenchmark de Árvore rubro-negra: (a) 20%, (b) 50% e (c) 100% de atualização. STMBench7: (d) *Read*, (e) *Read-Write* e (f) *Write Dominated*. STAMP: Aplicações (g) Bayes e (h) Genome.

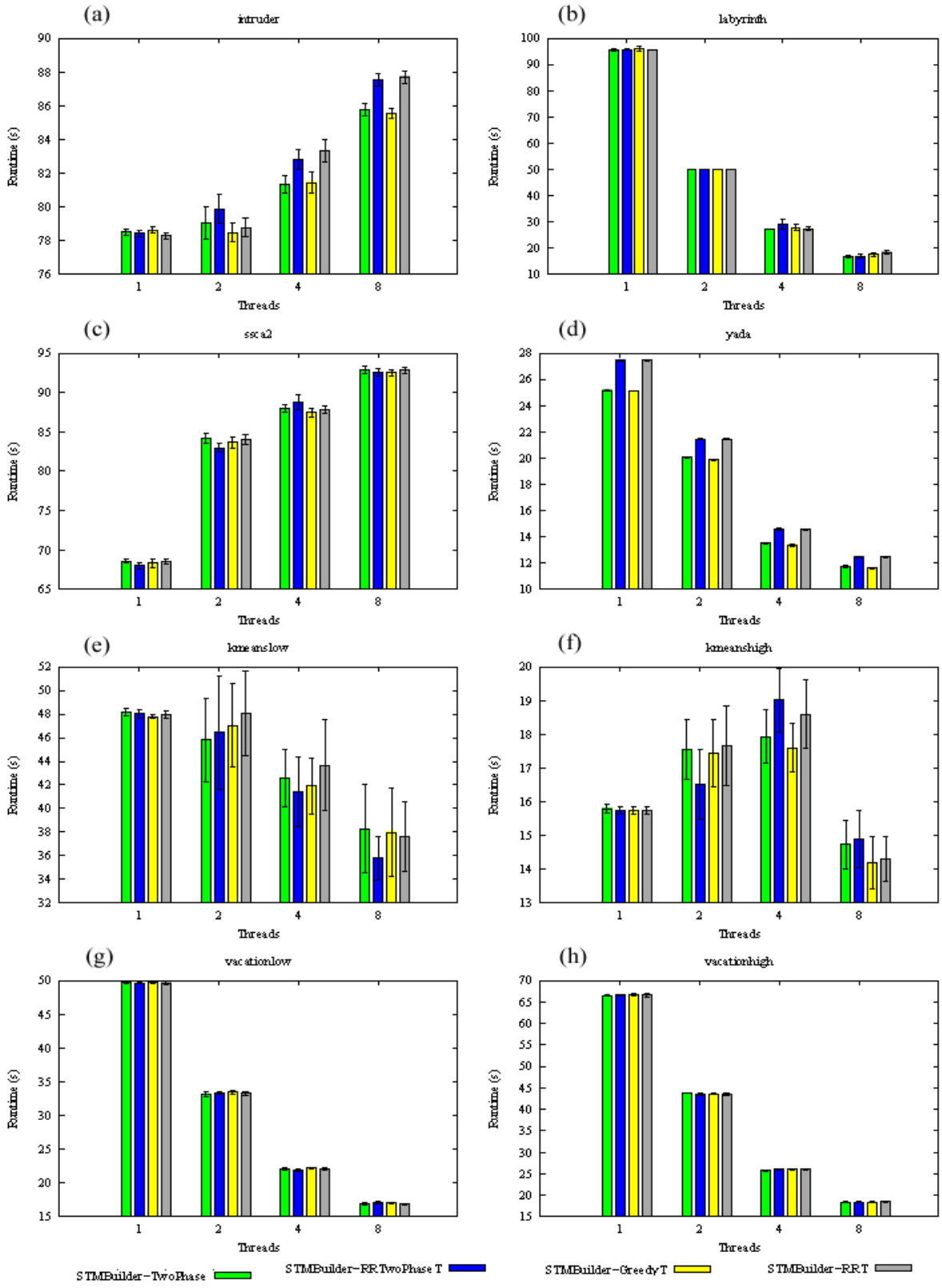


Figura 4.7: Gráficos com os resultados dos testes do gerenciamento de contenção. STAMP: Aplicações (a) Intruder, (b) Labyrinth, (c) SSCA2, (d) Yada, (e) Kmeans Low, (f) Kmeans High, (g) Vacation Low e (h) Vacation High.

Builder - Two Phase e STM Builder - RR Two PhaseT) são similares em todos os testes, não sugerindo diferença alguma de desempenho relevante entre as execuções comparadas.

STMBench7

Os gráficos das Figuras 4.6-(d, e e f) mostram os resultados dos testes com o STMBench7. Avaliando os gráficos e considerando as intersecções entre os intervalos de confiança, podemos perceber que o desempenho das execuções com gerenciamento de contenção de uma fase (STM Builder - GreedyT e STM Builder - RRT) e duas fases (STM Builder - Two Phase e STM Builder - RR Two PhaseT) são similares em todos os testes, não sugerindo diferença alguma de desempenho relevante entre as execuções comparadas. Nos testes com o STMBench7 o gerenciamento de contenção de uma e duas fases possuem desempenho similares.

STAMP

Os gráficos das Figuras 4.6-(g e h) e 4.7-(a, b, c, d, e, f, g e h) mostram os resultados dos testes com o STAMP. Avaliando o gerenciamento de contenção de uma fase (STM Builder - GreedyT e STM Builder - RRT) e considerando as intersecções entre os intervalos de confiança, as duas execuções possuem desempenho similar nas aplicações: Genome, Labyrinth, SSCA2, Kmeans Low, Kmeans High, Vacation Low e Vacation High (gráficos das Figuras 4.6-h e 4.7-(b, c, e, f, g e h)). A execução STM Builder - GreedyT é melhor que a execução STM Builder - RRT nas aplicações: Bayes, chegando a ser 19,5% mais rápida com uma thread (gráfico da Figura 4.6-g); Intruder, chegando a ser 3% mais rápida com 8 threads (gráfico da Figura 4.7-a); e Yada, chegando a ser 9,3% mais rápida com uma thread (gráfico da Figura 4.7-d).

Avaliando o gerenciamento de contenção de duas fases (STM Builder - Two Phase e STM Builder - RR Two PhaseT) e considerando as intersecções entre os intervalos de confiança, as duas execuções possuem desempenho similar nas aplicações: Genome, Labyrinth, SSCA2, Kmeans Low, Kmeans High, Vacation Low e Vacation High (gráficos das Figuras 4.6-h e 4.7-(b, c, e, f, g e h)). A execução STM Builder - Two Phase é melhor que a execução STM Builder - RR Two PhaseT nas aplicações: Bayes, chegando a ser 19,4% mais rápida com uma thread (gráfico da Figura 4.6-g); Intruder, chegando a ser 3% mais rápida com 8 threads (gráfico da Figura 4.7-a); e Yada, chegando a ser 9% mais rápida com uma thread (gráfico da Figura 4.7-d).

As execuções STM Builder - GreedyT e STM Builder - Two Phase são as melhores com o gerenciamento de contenção de uma e duas fases respectivamente. Avaliando as duas execuções e considerando as intersecções entre os intervalos de confiança, as duas execuções possuem desempenho similar nas aplicações do STAMP. Não há vantagem no uso do gerenciamento de contenção mais complexo com duas fases.

Nos testes com o STAMP ficou claro que as execuções que utilizam o gerente de contenção *Greedy* possuem desempenho melhor que as execuções com o gerente *RandomizedRounds*.

4.2.3 Diferença entre recuos

Esta subseção apresenta os resultados e comparações dos testes realizados com o STM Builder - GreedyT, STM Builder - Greedy, STM Builder - Greedy-EW, STM Builder - Greedy-NW e STM Builder - Polka. O objetivo desta subseção é mostrar:

1. Qual é o melhor tipo de recuo após um aborto (STM Builder - GreedyT, STM Builder - Greedy e STM Builder - Greedy-EW).
2. Se após um aborto é melhor executar um recuo ou reiniciar imediatamente (STM Builder - Greedy-NW, STM Builder - GreedyT, STM Builder - Greedy, STM Builder - Greedy-EW).
3. Se é melhor executar um recuo após um aborto (STM Builder - GreedyT, STM Builder - Greedy, STM Builder - Greedy-EW) ou quando um conflito é encontrado (STM Builder - Polka).

Microbenchmark

Os gráficos das Figuras 4.8-(a, b e c) mostram os resultados dos testes com o Microbenchmark. Avaliando os gráficos e considerando as intersecções entre os intervalos de confiança, podemos perceber que o desempenho de todas as execuções (STM Builder - GreedyT, STM Builder - Greedy, STM Builder - Greedy-EW, STM Builder - Greedy-NW e STM Builder - Polka) são similares em todos os testes.

No geral, os resultados do Microbenchmark não sugerem diferença alguma de desempenho relevante entre as execuções comparadas.

STMBench7

Os gráficos das Figuras 4.8-(d, e e f) mostram os resultados dos testes com o STMBench7. Avaliando os gráficos e considerando as intersecções entre os intervalos de confiança, podemos perceber que o desempenho de todas as execuções (STM Builder - GreedyT, STM Builder - Greedy, STM Builder - Greedy-EW, STM Builder - Greedy-NW e STM Builder - Polka) são similares em todos os testes.

No geral, os resultados do STMBench7 não sugerem diferença alguma de desempenho relevante entre as execuções comparadas devido ao intervalo de confiança grande, o que ocasionou intersecções entre todas as execuções.

STAMP

Os gráficos das Figuras 4.8-(g e h) e 4.9-(a, b, c, d, e, f, g e h) mostram os resultados dos testes com o STAMP. Avaliando os testes realizados no STAMP com as três

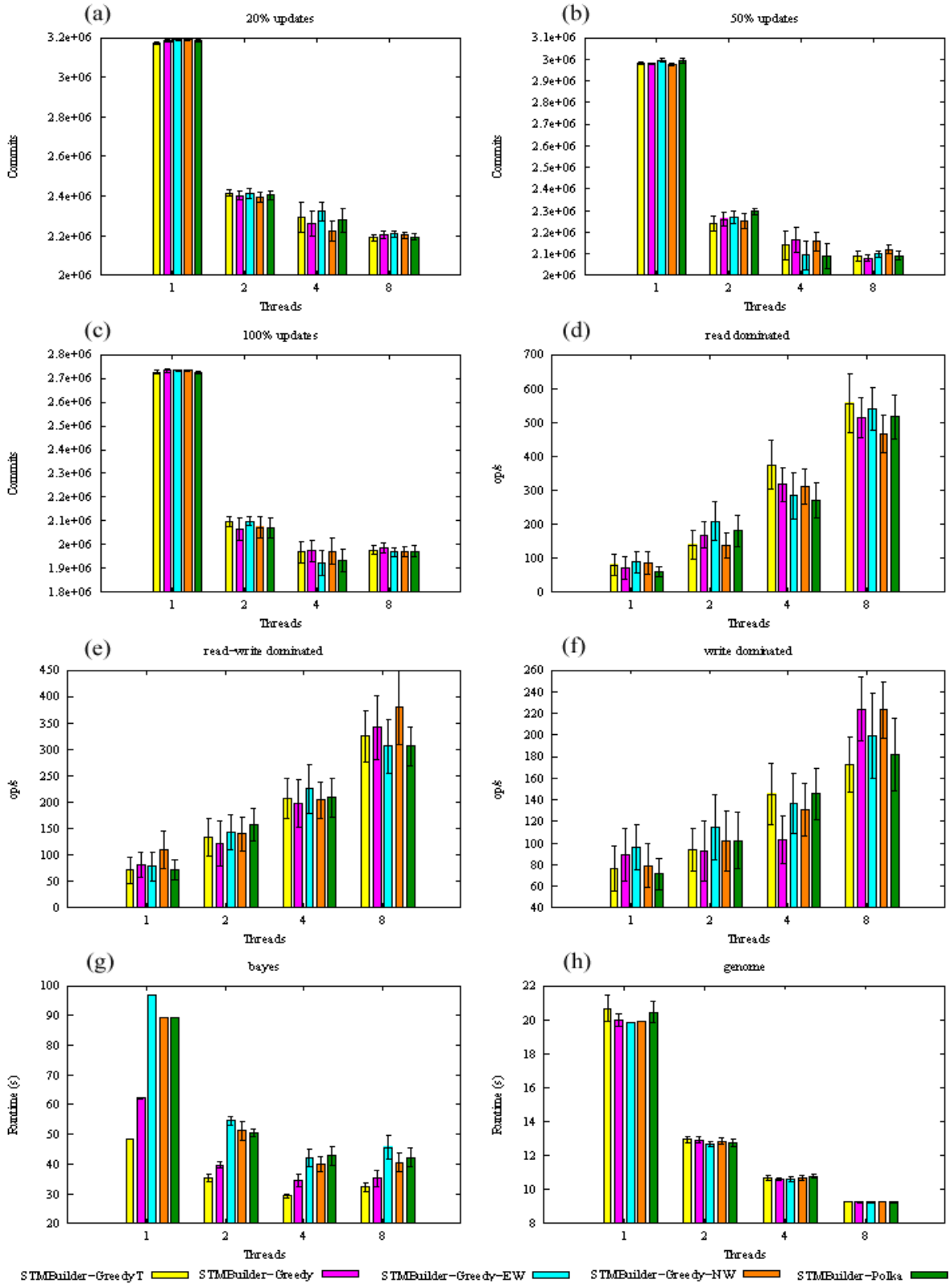


Figura 4.8: Gráficos com os resultados dos testes da diferença entre recuos. Microbenchmark de Árvore rubro-negra: (a) 20%, (b) 50% e (c) 100% de atualização. STMBench7: (d) *Read*, (e) *Read-Write* e (f) *Write Dominated*. STAMP: Aplicações (g) Bayes e (h) Genome.

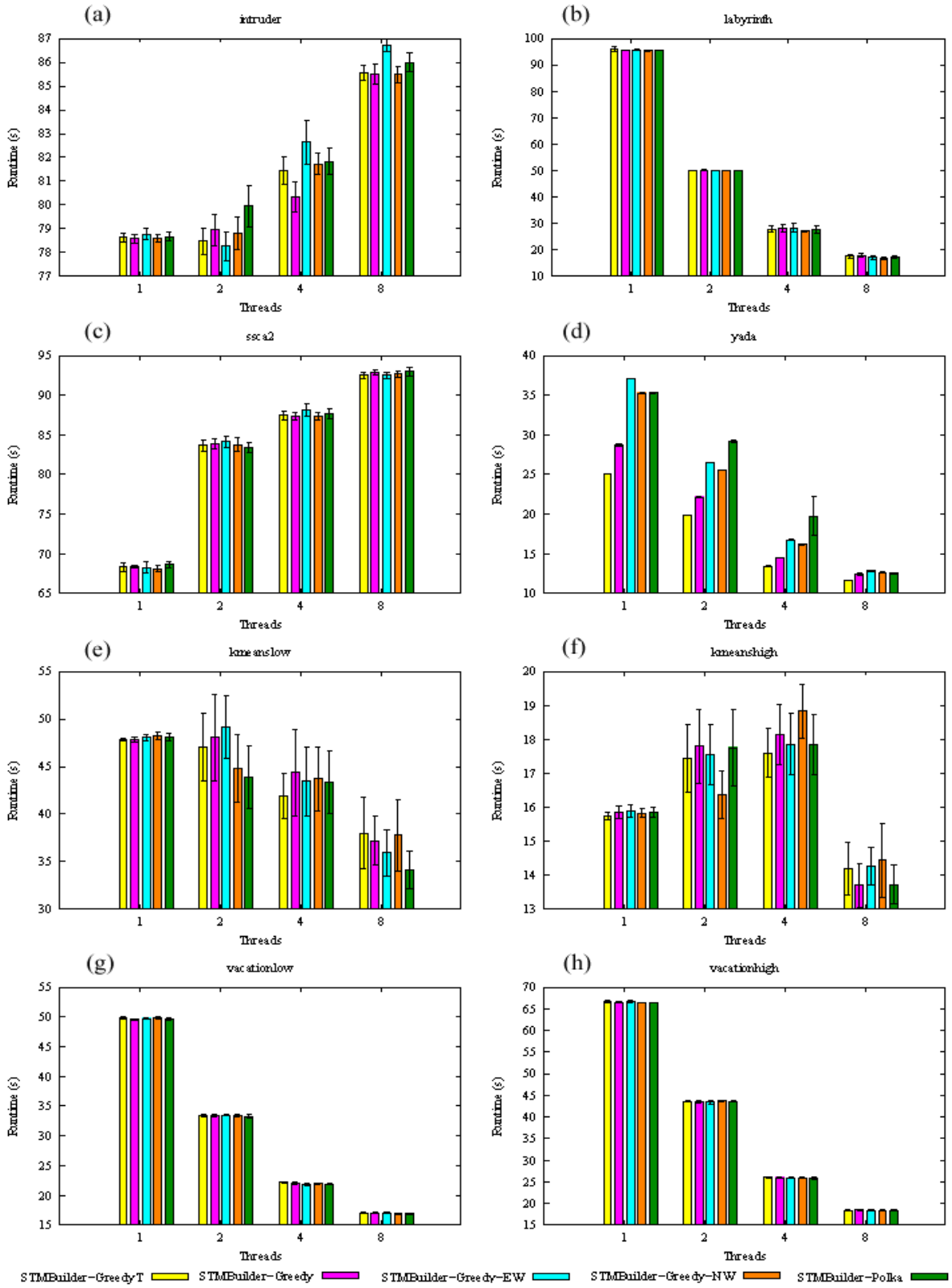


Figura 4.9: Gráficos com os resultados dos testes da diferença entre recuos. STAMP: Aplicações (a) Intruder, (b) Labyrinth, (c) SSCA2, (d) Yada, (e) Kmeans Low, (f) Kmeans High, (g) Vacation Low e (h) Vacation High.

execuções com recuo após um aborto (STM Builder - GreedyT, STM Builder - Greedy, STM Builder - Greedy-EW) e considerando as intersecções entre os intervalos de confiança, as execuções com recuo após um aborto possuem desempenho similar nas aplicações: Genome, Labyrinth, SSCA2, Kmeans Low, Kmeans High, Vacation Low e Vacation High (gráficos das Figuras 4.8-h e 4.9-(b, c, e, f, g e h)).

Na aplicação Intruder (gráfico da Figura 4.9-a) com 1, 2 e 8 threads as execuções com recuo após um aborto possuem desempenho similar, com 4 threads a execução STM Builder - Greedy é cerca de 2% melhor ques as execuções STM Builder - GreedyT e STM Builder - Greedy-EW, uma diferença de desempenho pequena, que fornece indício de que as execuções com recuo após um aborto possuem desempenho similar na aplicação Intruder.

A execução STM Builder - GreedyT é melhor que as execuções STM Builder - Greedy e STM Builder - Greedy-EW em duas aplicações com transações longas: Bayes, chegando a ser 28,4% mais rápida que a execução STM Builder - Greedy, e 2 vezes mais rápida que a execução STM Builder - Greedy-EW, ambas com uma thread (gráfico da Figura 4.8-g); Yada, chegando a ser 14,1% mais rápida que a execução STM Builder - Greedy, e 44,8% mais rápida que a execução STM Builder - Greedy-EW, ambas com uma thread (gráfico da Figura 4.9-d).

Como a execução STM Builder - GreedyT foi a melhor com recuo após um aborto no benchmark STAMP, vamos comparar a execução STM Builder - GreedyT com a execução STM Builder - Greedy-NW, para mostrar se após um aborto é melhor executar um recuo (STM Builder - GreedyT) ou reiniciar imediatamente (STM Builder - Greedy-NW).

Avaliando as execuções STM Builder - GreedyT e STM Builder - Greedy-NW, e considerando as intersecções entre os intervalos de confiança, as execuções possuem desempenho similar nas aplicações: Genome, Intruder, Labyrinth, SSCA2, Kmeans Low, Kmeans High, Vacation Low e Vacation High (gráficos das Figuras 4.8-h e 4.9-(a, b, c, e, f, g e h)). A execução STM Builder - GreedyT é melhor que a execução STM Builder - Greedy-NW em duas aplicações com transações longas: Bayes, chegando a ser 84,5% mais rápida com uma thread (gráfico da Figura 4.8-g); Yada, chegando a ser 40,4% mais rápida com uma thread (gráfico da Figura 4.9-d).

A execução STM Builder - GreedyT foi a melhor com recuo após um aborto no benchmark STAMP, vamos comparar a execução STM Builder - GreedyT com a execução STM Builder - Polka, para mostrar se é melhor executar um recuo após um aborto (STM Builder - GreedyT) ou ou quando um conflito é encontrado (STM Builder - Polka).

Avaliando as execuções STM Builder - GreedyT e STM Builder - Polka, e considerando as intersecções entre os intervalos de confiança, as execuções possuem desempenho similar nas aplicações: Genome, Labyrinth, SSCA2, Kmeans Low, Kmeans High, Vacation Low e Vacation High (gráficos das Figuras 4.8-h e 4.9-(b, c, e, f, g e h)).

Na aplicação Intruder (gráfico da Figura 4.9-a) com 1, 4 e 8 threads as execuções STM Builder - GreedyT e STM Builder - Polka possuem desempenho similar, com 2 threads a execução STM Builder - Polka é cerca de 2% melhor que a execução STM Builder - GreedyT, uma diferença de desempenho pequena, que fornece indício de que ambas

execuções possuem desempenho similar na aplicação Intruder.

A execução STM Builder - GreedyT é melhor que a execução STM Builder - Polka em duas aplicações com transações longas: Bayes, chegando a ser 84,5% mais rápida com uma thread (gráfico da Figura 4.8-g); Yada, chegando a ser 47,4% mais rápida com 4 threads (gráfico da Figura 4.9-d).

Nos testes realizados nesta subseção fica evidente que o melhor recuo após um aborto é realizado pela execução STM Builder - GreedyT, onde as transações que abortam devido a conflitos de escrita/escrita recuam por um período proporcional ao número de seus abortos sucessivos. Recuar após um aborto é mais vantajoso do que reiniciar a transação imediatamente, nas comparações entre a execução STM Builder - GreedyT e a execução STM Builder - Greedy-NW que não possui recuo em momento algum, o STM Builder - GreedyT obteve vantagem em alguns testes.

Recuar após um aborto é mais vantajoso do que recuar quando um conflito é encontrado, este resultado fica evidente nas comparações entre as execuções STM Builder - GreedyT e STM Builder - Polka no benchmark STAMP.

4.2.4 Invalidação no momento da efetivação (*Commit-time invalidation* - CTI)

Esta subseção apresenta os resultados e comparações dos testes realizados com o STM Builder - GreedyT, STM Builder-CTI-iWork, STM Builder-CTI-iFair e STM Builder-CTI-iPrio. O objetivo desta subseção é mostrar:

1. Qual é a melhor das três heurísticas testadas para a invalidação no momento da efetivação (*Commit-time invalidation* - CTI): iWork (STM Builder -CTI-iWork), iFair (STM Builder-CTI-iFair) e iPrio (STM Builder-CTI-iPrio).
2. Se a melhor heurística do CTI é melhor que a execução STM Builder - GreedyT.

Microbenchmark

Os gráficos das Figuras 4.10-(a, b e c) mostram os resultados dos testes com o Microbenchmark. Avaliando os gráficos e considerando as intersecções entre os intervalos de confiança, podemos perceber que o desempenho de todas as execuções do CTI são similares (STM Builder-CTI-iWork, STM Builder-CTI-iFair e STM Builder-CTI-iPrio).

A execução STM Builder - GreedyT é um pouco melhor que as execuções do CTI nos testes do Microbenchmark com uma thread. No geral, os resultados do Microbenchmark não sugerem diferença de desempenho relevante entre as execuções comparadas, o que nos leva a concluir que as execuções do CTI possuem desempenho satisfatório nos testes com o Microbenchmark.

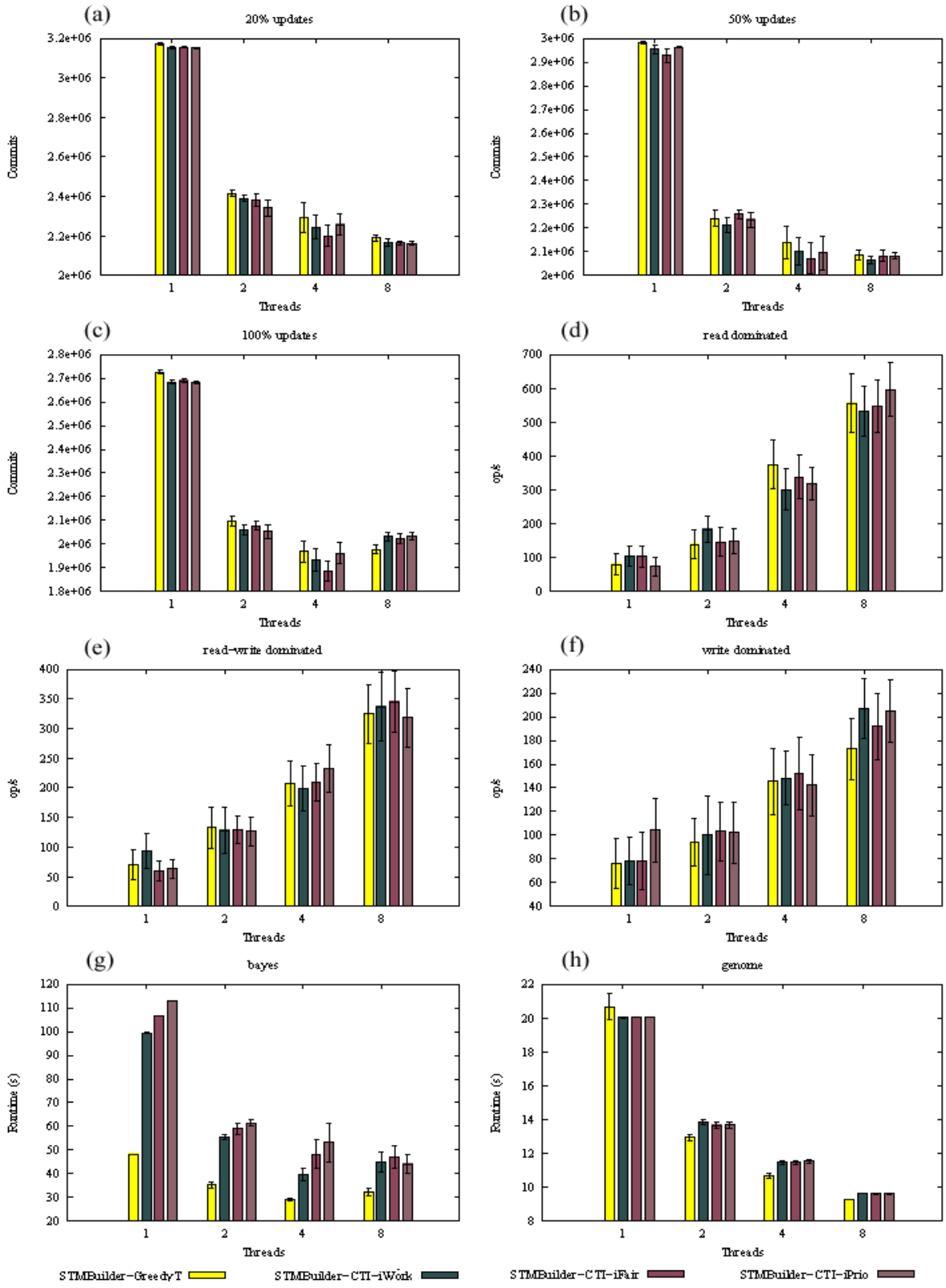


Figura 4.10: Gráficos com os resultados dos testes do CTI. Microbenchmark de Árvore rubro-negra: (a) 20%, (b) 50% e (c) 100% de atualização. STMBench7: (d) *Read*, (e) *Read-Write* e (f) *Write Dominated*. STAMP: Aplicações (g) Bayes e (h) Genome.

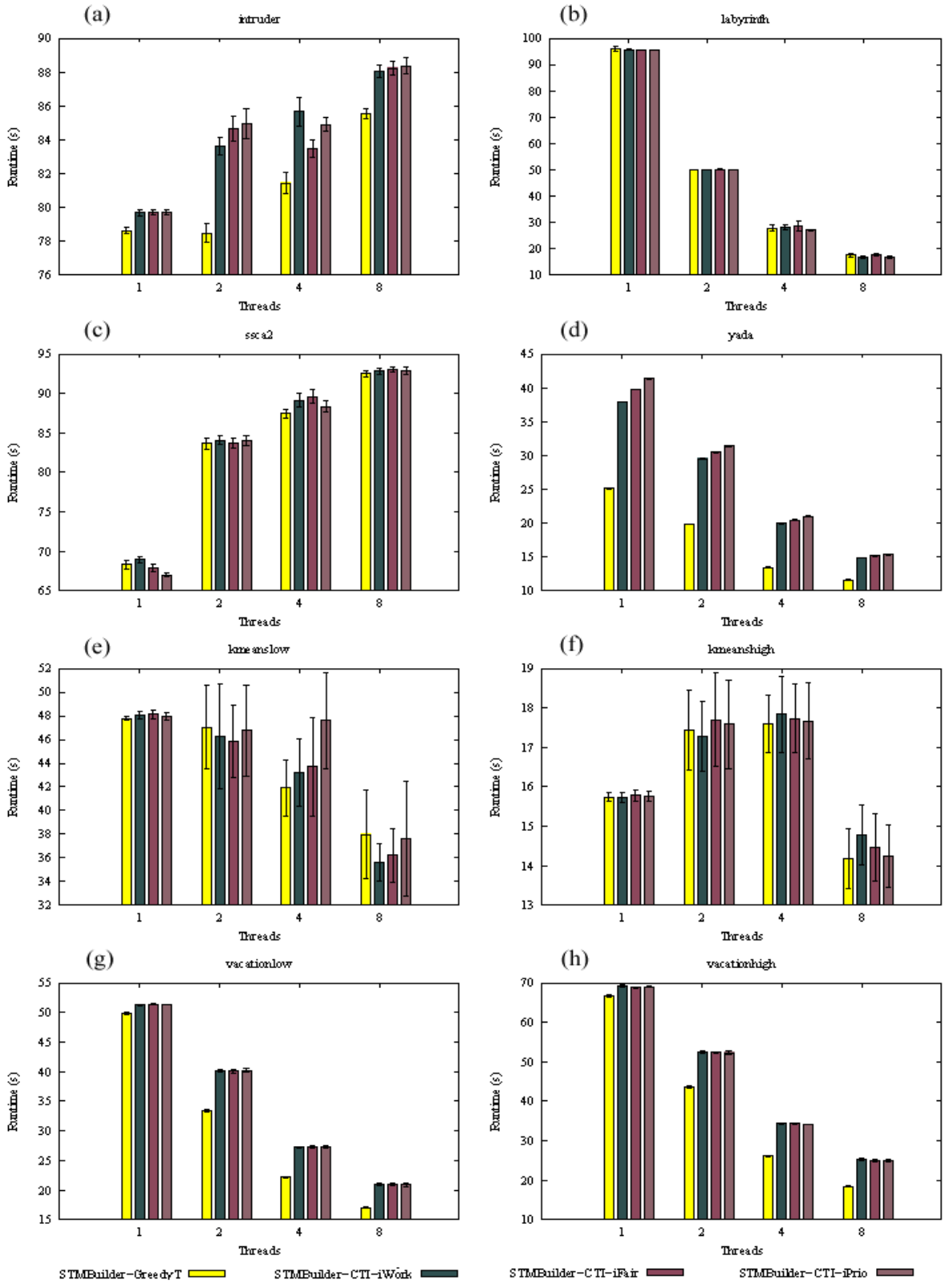


Figura 4.11: Gráficos com os resultados dos testes do CTI. STAMP: Aplicações (a) Intruder, (b) Labyrinth, (c) SSCA2, (d) Yada, (e) Kmeans Low, (f) Kmeans High, (g) Vacation Low e (h) Vacation High.

STMBench7

Os gráficos das Figuras 4.10-(d, e e f) mostram os resultados dos testes com o STMBench7. Avaliando os gráficos e considerando as intersecções entre os intervalos de confiança, podemos perceber que o desempenho das execuções do CTI (STM Builder-CTI-iWork, STM Builder-CTI-iFair e STM Builder-CTI-iPrio) e a execução STM Builder - GreedyT são similares em todos os testes.

Nos testes do STMBench7 não é possível definir qual heurística do CTI é melhor e não podemos afirmar que a execução STM Builder - GreedyT é melhor que as execuções do CTI, o que nos leva a concluir que as execuções do CTI possuem desempenho satisfatório nos testes com o STMBench7.

STAMP

Os gráficos das Figuras 4.10-(g e h) e 4.11-(a, b, c, d, e, f, g e h) mostram os resultados dos testes com o STAMP. Avaliando os testes realizados no STAMP com as três heurísticas do CTI (STM Builder-CTI-iWork, STM Builder-CTI-iFair e STM Builder-CTI-iPrio) e considerando as intersecções entre os intervalos de confiança, as execuções do CTI possuem desempenho similar nas aplicações: Genome, Labyrinth, SSCA2, Kmeans Low, Kmeans High, Vacation Low e Vacation High (gráficos das Figuras 4.10-h e 4.11-(b, c, e, f, g e h)).

Na aplicação Intruder (gráfico da Figura 4.11-a) com 1 e 8 threads as execuções do CTI possuem desempenho similar, com 2 threads a execução STM Builder-CTI-iWork é cerca de 2% melhor ques as execuções STM Builder-CTI-iFair e STM Builder-CTI-iPrio, e com 4 threads a execução STM Builder-CTI-iFair é cerca 2% melhor que as execuções STM Builder-CTI-iWork e STM Builder-CTI-iPrio. A diferença de desempenho encontrada entre as execuções do CTI na aplicação Intruder é pequena, e fornece indícios de que as execuções do CTI possuem desempenho similar na aplicação Intruder.

A execução STM Builder-CTI-iWork é melhor que as execuções STM Builder-CTI-iFair e STM Builder-CTI-iPrio em duas aplicações com transações longas: Bayes, chegando a ser 21,3% mais rápida que a execução STM Builder-CTI-iFair, e 33,7% mais rápida que a execução STM Builder-CTI-iPrio, ambas com 4 threads (gráfico da Figura 4.10-g); Yada, chegando a ser 5% mais rápida que a execução STM Builder-CTI-iFair, e 9% mais rápida que a execução STM Builder-CTI-iPrio, ambas com uma thread (gráfico da Figura 4.11-d).

Como a heurística iWork foi a melhor das três heurísticas do CTI no benchmark STAMP, vamos comparar a execução STM Builder-CTI-iWork com a execução STM Builder - GreedyT, para mostrar se o CTI trouxe ganho de desempenho para o STM Builder.

Avaliando as execuções STM Builder-CTI-iWork e STM Builder - GreedyT, e considerando as intersecções entre os intervalos de confiança, as execuções possuem desempenho similar nas aplicações: Labyrinth, SSCA2, Kmeans Low, Kmeans High (gráficos das Figuras 4.11-(b, c, e e f)).

A execução STM Builder - GreedyT é melhor que a execução STM Builder-CTI-iWork nas aplicações: Bayes, chegando a ser duas vezes mais rápida com uma thread (gráfico da Figura 4.10-g); Yada, chegando a ser 51% mais rápida com uma thread (gráfico da Figura 4.11-d); Intruder, chegando a ser 6,6% mais rápida com 2 threads (gráfico da Figura 4.10-h); Genome, chegando a ser 7,6% mais rápida com 4 threads (gráfico da Figura 4.11-b); Vacation Low, chegando a ser 23,3% mais rápida com 8 threads (gráfico da Figura 4.11-g); Vacation High, chegando a ser 37,3% mais rápida com 8 threads (gráfico da Figura 4.11-h).

As execuções do CTI possuem desempenho satisfatório no cenário onde as transações são curtas, os conjuntos de leitura e escrita são pequenos e a aplicação tem baixa contenção (SSCA2, Kmeans Low, Kmeans High), a única exceção é com a aplicação Labyrinth.

A melhor das três heurísticas testadas para a invalidação no momento da efetivação (*Commit-time invalidation* - CTI) foi a iWork (STM Builder-CTI-iWork), em que se a soma do tamanho do conjunto de leitura mais o de escrita, multiplicada pela quantidade de abortos mais um, for menor que o número de conflitos encontrados, a transação é abortada. Esta heurística é mais simples que as outras duas testadas (iFair e iPrio) e foi a que teve melhor desempenho geral no testes.

A tentativa de melhorar o desempenho do STM Builder - GreedyT e consecutivamente da SwissTM não foi alcançada com a implementação do CTI no STM Builder. As implementações de todas as heurísticas do CTI no momento da efetivação percorrem a cópia do conjunto de leitura de todas as transações em execução em busca de conflitos do tipo escrita/leitura, que causa uma sobrecarga a mais nas execuções do CTI, e gera uma perda de desempenho em relação a execução STM Builder - GreedyT.

4.2.5 Invalidação no momento da efetivação (*Commit-time invalidation* - CTI) na SwissTM

Esta subseção apresenta os resultados e comparações dos testes realizados com a SwissTM e a SwissTM-CTI-iWork. A SwissTM-CTI-iWork é a implementação do CTI diretamente na SwissTM original. Ela implementa a heurística iWork do CTI e utiliza o gerente de contenção de uma fase *GreedyT*.

Implementamos somente a heurística iWork do CTI, pois ela foi a que teve o melhor desempenho geral nos testes do CTI no STM Builder (Subseção 4.2.4).

O objetivo desta subseção é:

- Mostrar que é possível implementar um *model* do STM Builder diretamente na SwissTM. É um maneira de verificar se um *model* que foi implementado e testado supera o desempenho da SwissTM. Nos testes realizados na Subseção 4.2.4 o STM Builder-CTI-iWork, que implementa o CTI com a heurística iWork, não foi melhor que o desempenho da execução STM Builder - GreedyT, que é uma execução que se diferencia da SwissTM por não usar gerenciamento de contenção de duas fases,

utiliza somente o *GreedyT*. Implementamos a SwissTM-CTI-iWork para demonstrar que é possível implementar um *model* do STM Builder diretamente na SwissTM.

Os gráficos das Figuras 4.12-(a, b e c) mostram os resultados dos testes com o Microbenchmark. Avaliando os gráficos podemos perceber que o desempenho da SwissTM-CTI-iWork é inferior ao da SwissTM.

Os gráficos das Figuras 4.12-(d, e e f) mostram os resultados dos testes com o STMBench7. Avaliando os gráficos e considerando as intersecções entre os intervalos de confiança, podemos perceber que o desempenho da SwissTM-CTI-iWork é similar ao da SwissTM nos testes do STMBench7.

Os testes com o STAMP estão nos gráficos das Figuras 4.12-(g,h) e 4.13 (a, b, c, d, e, f, g, h). Avaliando os gráficos podemos perceber que o desempenho da SwissTM-CTI-iWork é similar ao da SwissTM na aplicação Labyrinth e inferior nas outras aplicações.

A SwissTM e a SwissTM-CTI-iWork são dois códigos independentes, qualquer outra variação que fizessemos na SwissTM, teríamos que gerar outro código independente para comparar com os testes já existentes. Além de gerar outro código independente, é necessário para cada implementação nova da SwissTM reconfigurar os Benchmarks.

Nesta subseção mostramos que implementar e testar novas ideias no STM Builder é bem mais fácil, pois nele pode-se testar diversas variações no mesmo código, perdendo pouco tempo com novas implementações e testando várias formas da mesma implementação rapidamente.

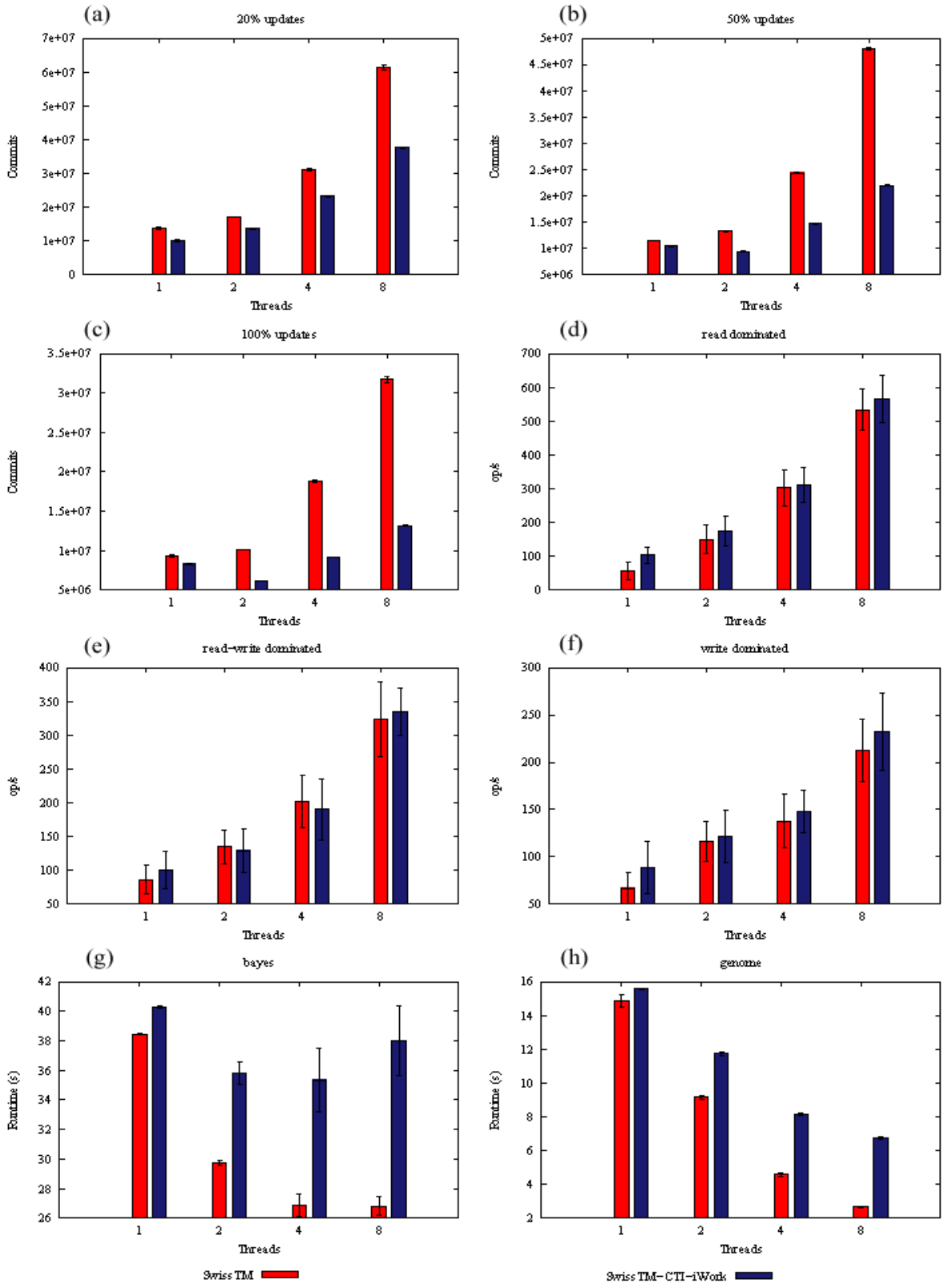


Figura 4.12: Gráficos com os resultados dos testes do CTI na SwissTM. Microbenchmark de Árvore rubro-negra: (a) 20%, (b) 50% e (c) 100% de atualização. STMBench7: (d) *Read*, (e) *Read-Write* e (f) *Write Dominated*. STAMP: Aplicações (g) Bayes e (h) Genome.

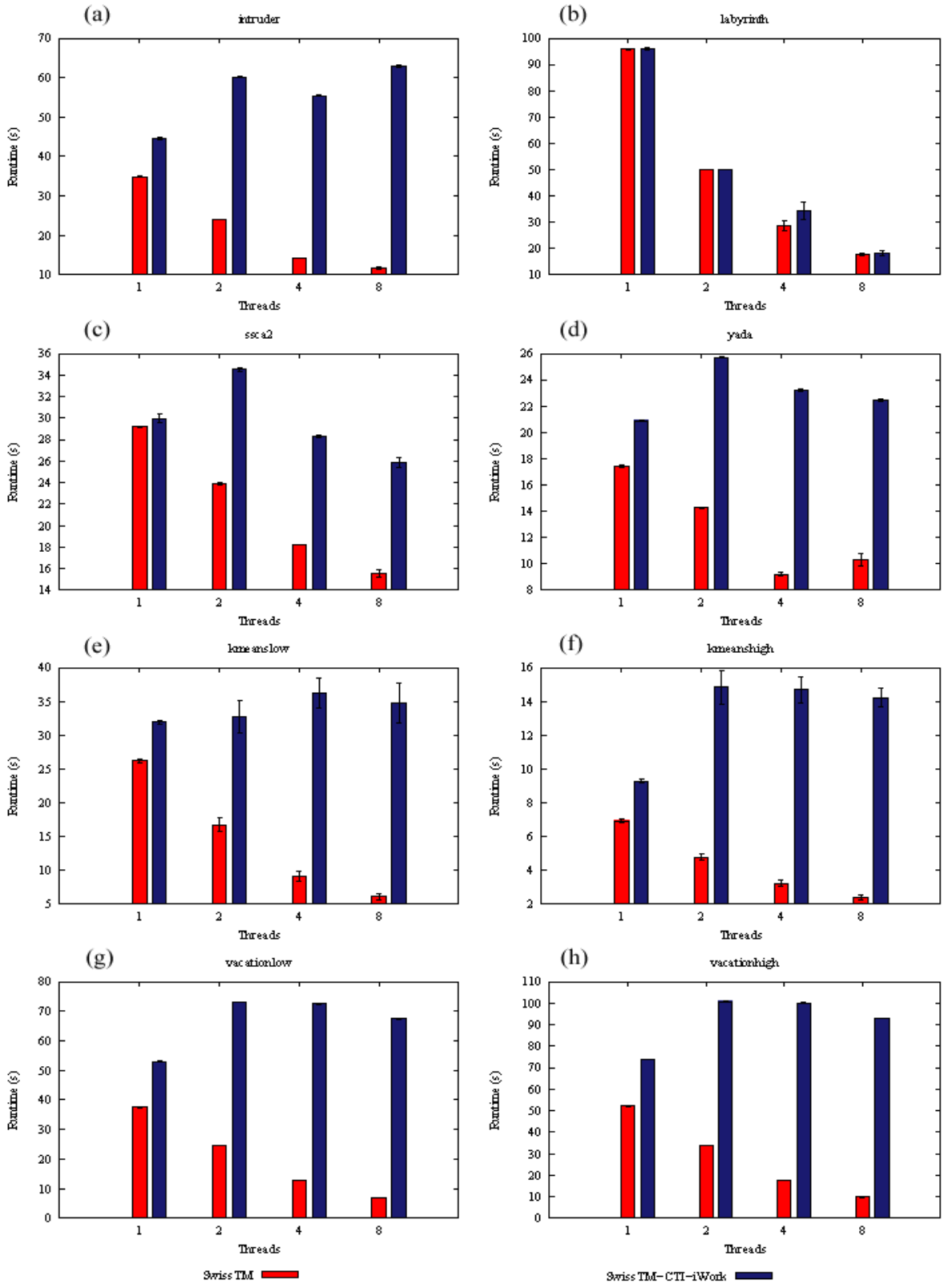


Figura 4.13: Gráficos com os resultados dos testes do CTI na SwissTM. STAMP: Aplicações (a) Intruder, (b) Labyrinth, (c) SSCA2, (d) Yada, (e) Kmeans Low, (f) Kmeans High, (g) Vacation Low e (h) Vacation High.

Capítulo 5

Conclusão

Neste Capítulo apresentamos as contribuições do trabalho e os trabalhos futuros.

5.1 Contribuições

Neste trabalho desenvolvemos um novo sistema de memória transacional em software, o STM Builder. Ele foi desenvolvido com o intuito de ser fácil de usar, de adicionar novos recursos e de realizar testes e comparações justas.

Junto com o STM Builder desenvolvemos um framework de testes, que também pode ser utilizado com outros STMs. Todos os testes realizados no Capítulo 4 foram executados com o framework de testes, que executou e gerou todos os gráficos automaticamente, restando apenas a necessidade de editar as figuras para o texto. Esta facilidade em executar testes fez com que testássemos uma grande quantidade de execuções, o que gerou inúmeros gráficos.

Nos testes realizados na Subseção 4.2.1 ficou claro que o STM Builder não possui bom desempenho no cenário em que as transações são curtas. A execução STM Builder - Two Phase teve um desempenho satisfatório em relação a SwissTM para benchmarks com simulação de aplicações do mundo real (STAMP e STMBench7), e caso uma das novas formas de execução do STM Builder supere o desempenho do STM Builder - Two Phase, consequentemente melhoraremos o estado da arte.

Comparamos o gerencimento de contenção de uma e duas fases nos testes realizados na Subseção 4.2.2, o que nos levou a conclusão de que o conceito de gerenciamento de contenção de duas fases apresentado pela SwissTM, o qual favorece o progresso das transações que tenham realizado um número significativo de atualizações, não trouxe benefícios ao STM Builder. O gerenciamento de contenção de uma e duas fases possuem desempenho idênticos. Nesta mesma subseção comparamos os gerentes de contenção *GreedyT* e *RandomizedRounds*, e podemos concluir que o gerente *GreedyT* foi melhor na maioria dos testes. A diferença de desempenho encontrada nos testes de gerenciamento de contenção foi pequena, o que mostra que focar o trabalho em melhorar um gerente de contenção

simples não resultará em grande ganho de desempenho, e outras maneiras devem ser encontradas para melhorar o desempenho de um STM.

Uma outra maneira simples de tentar melhorar o desempenho de um STM foi testado na Subseção 4.2.3, onde verificamos se é melhor ou não segurar o reinício de uma transação após o aborto ou quando um conflito é encontrado. Nos testes ficou comprovado que recuar após um aborto é vantajoso. A melhor técnica de recuo após um aborto é aquela cuja a transação recua por um período proporcional ao número de seus abortos sucessivos, que é a técnica apresentada pela SwissTM. Esta técnica é mais vantajosa do que reiniciar a transação imediatamente ou recuar quando um conflito é encontrado.

Na Subseção 4.2.4 tentamos melhorar o desempenho do STM Builder implementando uma nova ideia no sistema, a técnica de invalidação no momento da efetivação, do artigo [14]. Adaptamos a ideia para tentar torná-la viável para o STM Builder, mas o desempenho da nova implementação não foi satisfatório, e não conseguimos alcançar o objetivo de ganho de desempenho. Porém mostramos que realmente é possível implementar e testar novas ideias no STM Builder, e apresentamos também uma nova heurística para a invalidação no momento da efetivação, a heurística iWork que teve melhor desempenho nos testes realizados que as heurísticas iFair e iPrio apresentadas pelo artigo [14].

Implementar um *model* do STM Builder diretamente na SwissTM é um maneira de mostrar que um *model* que foi implementado e testado supera o desempenho da SwissTM, e consequentemente melhora o estado da arte. Nos testes realizados na Subseção 4.2.5 testamos a SwissTM-CTI-iWork, que implementa a técnica de invalidação no momento da efetivação com a heurística iWork diretamente na SwissTM, o desempenho desta implementação foi inferior ao da SwissTM como apresentado nos testes da Subseção 4.2.4. Realizamos esta implementação para mostrar que é possível, após encontrar uma execução boa do STM Builder, retirar toda a sobrecarga adicionada ao STM Builder para facilitar as implementações e testes, e ver o ganho real de desempenho.

Neste trabalho não conseguimos melhorar o desempenho da SwissTM, mas implementamos o STM Builder baseado na SwissTM, que comprovou facilitar implementações e testes de novas ideias em conjunto com o framework de testes. O STM Builder é capaz de testar diversas variações no mesmo código, perdendo pouco tempo com novas implementações e testando várias formas da mesma implementação rapidamente.

5.2 Trabalhos Futuros

Após a realização deste trabalho foram abertas várias possibilidades de trabalhos futuros:

- Implementar novas ideias de STM no STM Builder, como a Memória transacional com conhecimento de dependências (DATM - *Dependence-Aware Transactional Memory*) que foi proposta e implementada em HTM no artigo [30]. Uma versão de STM da DATM foi proposta posteriormente no artigo [2], onde foi implementada a *Dependence-aware software transacional memory* (DASTM) utilizando técnicas

da TL2 [8], mas modificadas para oferecer suporte a dependências e ao encaminhamento dos dados. Nos testes, a DASTM mostra um desempenho até 4,8 vezes melhor em benchmarks com alta contenção como o STAMP. A melhora de desempenho é diretamente atribuída à capacidade da DASTM de gerenciar dependências.

- Implementar novas *models* bases para o STM Builder, trazendo mais opções de gerenciamento de memória e detecção de conflitos.

- Implementações de STM variam muito em seus mecanismos subjacentes e a melhor escolha é sempre atrelada ao tipo de carga de trabalho que se vai executar. Durante uma execução, os tipos de cargas de trabalho podem mudar e uma melhor escolha da implementação de STM pode ser necessária em tempo de execução.

  O artigo [34] apresenta um sistema de baixa sobrecarga com adaptação entre implementações STM, que permite a escolha entre qualquer conjunto de algoritmos STM dinamicamente. Ele traz um sistema baseado no RSTM [28] e já traz 10 implementações STM diferentes e algumas políticas de troca entre as implementações STM. Os resultados dos testes realizados no artigo demonstram que a adaptatividade pode evitar os piores comportamentos, mesmo quando se usa apenas uma heurística simples para a troca de implementação, como abortos consecutivos. A exploração de novas políticas de adaptatividade é claramente necessária e estes resultados iniciais já demonstram a eficácia do mecanismo de adaptatividade.

  Implementar a adaptividade dinâmica no STM Builder é um dos nossos objetivos futuros, mas para isso ainda é necessário oferecer várias implementações de STM e criar um mecanismo de troca entre as implementações.

- Implementar o escalonamento de transações para decidir quando uma transação é executada. A implementação de um escalonador para o STM Builder poderá melhorar o desempenho do sistema. Grandes ganhos de desempenho foram alcançados por implementações de escalonadores como o Shrink: que prevê os acessos futuros de uma thread com base nos acessos passados, e dinamicamente serializa as transações baseado na previsão para prevenir conflitos [11]; e a LUTS: que é um escalonador de transações a nível de usuário, que controla efetivamente o nível de contenção no cenário de pseudoparalelismo, e fornece meios para aumentar o desempenho do sistema, evitando iniciar transações que provavelmente abortarão no futuro próximo [27].

Referências Bibliográficas

[1] SwissTM Website. URL: http://lpdserver.epfl.ch/transactions/wiki/doku.php?id=swisstm, April 2012.

[2] Hany E. Ramadan an Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an STM. *Symposium on Principles and Practice of Parallel Programming (PPoPP)- ACM*, 2009.

[3] Alexandro Jose Baldassin. *Explorando memória transacional em software nos contextos de arquiteturas assimétricas, jogos computacionais e consumo de energia.* PhD thesis, Universidade Estadual de Campinas . Instituto de Computação, 2009.

[4] The STMBench7 Benchmark. Url: http://lpd.epfl.ch/transactions/wiki/doku.php?id=stmbench7, June 2011.

[5] N. G. Bronson, H. Chafi, and K. Olukotun. CCSTM: A library-based stm for scala. In The First Annual Scala Workshop at Scala Days 2010, April 2010.

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

[7] 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.

[8] David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *DISC*, pages 194–208, 2006.

[9] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54:70–77, April 2011.

[10] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *SIGPLAN Not.*, 44:155–165, June 2009.

[11] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 7–16, New York, NY, USA, 2009. ACM.

[12] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[13] Gnuplot. Url: http://www.gnuplot.info/, August 2011.

[14] 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.

[15] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 258–264, New York, NY, USA, 2005. ACM.

[16] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[17] Rachid Guerraoui and Michal Kapalka. The Theory of Transactional Memory. *Bulletin of the EATCS*, (97), 2009.

[18] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: A benchmark for software transactional memory. *The European Professional Society on Computer Systems - ACM*, 2007.

[19] Derin Harmanci, Vincent Gramoli, Pascal Felber, and Christof Fetzer. Extensible transactional memory testbed. *J. Parallel Distrib. Comput.*, 70:1053–1067, October 2010.

[20] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. *In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pages 253–262*, 2006.

[21] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[22] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

[23] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, 2008.

[24] Philip W. Howard and Jonathan Walpole. A relativistic enhancement to software transactional memory. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 15–15, Berkeley, CA, USA, 2011. USENIX Association.

[25] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. *Concurrency and Synchronization in Java Programs (CSJP)*, 2004.

[26] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing.*, pages 240–248, 2005.

[27] Daniel Nicácio, 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 - Volume Part I*, ICA3PP'11, pages 144–157, Berlin, Heidelberg, 2011. Springer-Verlag.

[28] Department of Computer Science University of Rochester. RSTM. Home Page: http://www.cs.rochester.edu/research/synchronization/rstm, April 2012.

[29] David A. Patterson and John L. Hennessy. *Organização e Projeto de Computadores - A Interface Hardware / Software*. Morgan Kaufmann Publishers, 3rd edition, 2005.

[30] 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.

[31] Sandro Rigo, Paulo Centoducatte, and Alexandro Baldassin. *Memórias Transacionais - Uma nova alternativa Para Programação Concorrente*. Unicamp, October 2007.

[32] Johannes Schneider and Roger Wattenhofer. Bounds on contention management algorithms. *Theor. Comput. Sci.*, 412:4151–4160, July 2011.

[33] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts with Java*. John Wiley & Sons. Inc, 7th edition, 2007.

[34] Michael F. Spear. Lightweight, robust adaptivity for software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 273–283, New York, NY, USA, 2010. ACM.

[35] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Not.*, 44:141–150, February 2009.

[36] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the Twentieth International Symposium on Distributed Computing*, Sep 2006.

[37] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. Automatic atomic region identification in shared memory SPMD programs. *SIGPLAN Not.*, 45(10):652–670, October 2010.

Apêndice A

Detalhes da Plataforma

A.1 Implementação de um novo Gerente de Contenção no STM Builder

Implementar um novo gerente de contenção no STM Builder é muito simples. Cada transação possui uma instância local de *CMDataTX*, pois todas transações são filhas da classe base Transaction, como mostrado no diagrama de classes da Figura 3.4. *CMDataTX* é a classe onde ficam armazenados todos os dados que os gerentes de contenção precisam para funcionar corretamente. Quando é criado um novo gerente que necessite armazenar algum dado, este dado deve ser colocado em *CMDataTX*. Após ter colocado os dados no local correto, basta criar uma nova classe para seu gerente que herde a classe base *CM* e sobrescreva os métodos *ShouldRestartWriteWrite*, *ShouldRestartReadWrite* e *ShouldRestartWriteRead* que são responsáveis em decidir se a transação deve ou não abortar quando um conflito é encontrado. Cada método representa o tratamento para um tipo de conflito. Após ter criado o novo gerente, para o sistema selecionar o gerente correto, é necessário adicionar no construtor base da classe Transaction um novo comando condicional onde ocorre a seleção dos gerentes de contenção. Conforme mostrado na Figura 3.4, o STM Builder traz alguns gerentes de contenção implementados e os testes destes gerentes estão descritos no Capítulo 4.

A.2 Implementação de uma *Transition Function* no STM Builder

A implementação de uma *Transition Function* é similar as demais implementações. Basta criar uma nova classe para a sua *Transition Function* que herde a classe base *TF* e sobrescreva os métodos *changePhase* que são responsáveis pela mudança de fases dos gerentes. Na classe *TF* existem dois métodos *changePhase* que se diferenciam somente pelos parâmetros, o que possui o parâmetro ThreadStatistics deve ser utilizado somente com *models* com leituras invisíveis e o com o parâmetro ThreadStatisticsEpoch deve ser

utilizado somente com *models* com leituras visíveis. Após ter criado a TF, para o sistema selecionar a nova TF, é necessário adicionar no construtor base da classe Transaction um novo comando condicional onde ocorre a seleção das TFs. A utilização das TFs é um pouco mais complexa, pois o programador deve escolher onde as mudanças de fases ocorrerão, pois as mudanças de fases são atreladas ao esquema de gerenciamento utilizado. Para entender melhor veja o diagrama de classes do STM Builder na Figura 3.4.

A.3 Implementação de um novo Model no STM Builder

Para implementar um novo *model* no STM Builder, basta herdar um dos *models* bases do sistema, o `TransactionInvisible` ou o `TransactionEpoch`, sobrescrever os métodos que necessite refazer para o novo *model*, que basicamente correspondem aos métodos virtuais *TxCommit*, *Rollback*, *TxAbort*, *TxRestart*, *WriteWord*, *ReadWord* e alguns outros. Estes métodos são virtuais pois são acoplados dinamicamente na execução da transação. Um detalhe muito importante na implementação do novo *model* é que seu construtor base deve sempre chamar o construtor base do *model* base que ele herdou, para que os gerentes de contenção e as *Transition Functions* sejam criadas para o respectivo *model*. O último passo, para o sistema selecionar o *model* correto para execução, é adicionar na classe *Tls* um novo comando condicional nos métodos *GlobalMemoryManagerMalloc*, *GlobalMemoryManagerFree*, *GlobalInit*, *GlobalShutdown* e *ThreadInit* onde ocorre a seleção do *model* da transação.

A.4 Modificação da Função Hash do STM Builder

Modificações na função de Hash utilizada somente funciona com leituras invisíveis, ou seja, somente funciona com *models* que utilizem como base o *model* `TransactionInvisible`. Para modificar a função de hash utilizada, basta criar um novo *model* que herde o *model* base `TransactionInvisible` e sobrescrever o método *map_address_to_index*.

A.5 Detalhes dos arquivos de entrada do Framework de Testes

A Figura 4.2-a traz um exemplo do arquivo *input* que é a entrada do primeiro módulo. Neste arquivo é indicado o que se deseja executar no framework de testes. O arquivo *input* possui as seguintes entradas:

- MAXEXEC: Indica o número de testes que serão executados, e no exemplo da Figura 4.2-a serão executados 2 testes.
- MAXTHREAD: Indica o número máximo de threads que serão executadas em cada teste, e no exemplo da Figura 4.2-a os testes terão no máximo 64 threads. O número

de threads nos testes variam em potência de dois, ou seja, para o nosso exemplo que tem no máximo 64 threads, os testes serão executados com 1, 2, 4, 8, 16, 32 e 64 threads.

- MAXROUND: Indica o número máximo de repetições de cada teste, no exemplo da Figura 4.2-a cada teste será executado 32 vezes.

- INTERVAL: Indica o intervalo de tempo em segundos entre cada execução de um STM, por padrão este tempo é de 5 segundos.

- GRADIENT: Indica a utilização de um gradiente para as cores das barras dos gráficos. Suponha uma execução do framework de testes com 6 testes (MAXEXEC 6) e a entrada GRADIENT 33. Isto significa que as três primeiras barras dos gráficos gerados pelos testes terão a mesma cor com variações no tom (gradiente linear), a quarta, quinta e sexta barra terão uma outra cor com variação no tom. As barras dos gráficos são geradas na ordem em que as entradas das execuções foram colocadas no arquivo *input*. O número máximo de variações de uma mesma cor que o framework de testes permite é 9 e somente realiza as variações com 6 cores diferentes, por este motivo a maior entrada permitida é GRADIENT 999999 que corresponde a 54 testes. Não é necessário que seja indicado o gradiente para todos os testes, por exemplo, com 6 testes é permitida uma entrada como GRADIENT 22, onde as duas primeiras barras terão um gradiente de uma cor, as duas seguintes terão um gradiente de outra cor, e as duas últimas terão cores distintas. Caso não seja desejado utilizar o gradiente como na Figura 4.2-a, coloque a entrada GRADIENT 0 que o framework de testes colocará uma cor para cada barra dos gráficos.

- COLORS: Indica a utilização de cores personalizadas no formato RGB para as barras dos gráficos. Não é obrigatório o uso desta entrada, mas caso seja utilizada é necessário indicar uma cor para cada teste. Suponha uma execução do framework de testes com 6 testes (MAXEXEC 6), caso seja desejado personalizar as cores utilize como exemplo a entrada COLORS 2F4F4F FF0000 00FF00 0000FF FFFF00 FF00FF, onde é indicado 6 cores no formato RGB sem o # e em hexadecimal.

- MICROBENCHMARK: Indica se o Microbenchmark de árvore rubro-negra será utilizado nos testes. Esta opção funciona somente com o STM Builder, a SwissTM e o EpochSTM. A entrada MICROBENCHMARK YES indica que será utilizado o microbenchmark e a entrada MICROBENCHMARK NO indica que não será utilizado.

- SB7: Indica se o STMBench7 será utilizado nos testes. A entrada SB7 YES indica que será utilizado o STMBench7 e a entrada SB7 NO indica que não será utilizado.

- STAMP: Indica se o STAMP será utilizado nos testes. A entrada STAMP YES indica que será utilizado o STAMP e a entrada STAMP NO indica que não será utilizado.

- Execução: Cada STM que será executado deve ser indicado no arquivo *input*, tendo o início da indicação com um BEGIN e o término com END. Entre o BEGIN e o END deve ser colocado obrigatoriamente as entradas:

- STM: Indica a pasta em que está o STM, a entrada STM deve ser seguida do nome correto da pasta. No exemplo da Figura 4.2-a, entre as linhas 10 e 19 a entrada STM STMBuilder indica a execução do STM Builder e entre as linhas 21 e 24 a entrada STM SwissTM indica a execução da SwissTM.
- Name: Indica o nome sem conter espaços da execução que aparecerá nos gráficos. No exemplo da Figura 4.2-a, entre as linhas 10 e 19 a entrada NAME Greedy indica o nome desta execução do STM Builder, e entre as linhas 21 e 24 a entrada NAME SwissTM indica o nome da execução da SwissTM.
- Arquivo de Configuração do STM Builder: No exemplo da Figura 4.2-a, entre as linhas 13 a 18, está incluída uma cópia do arquivo de configuração do STM Builder referente a execução desejada. O arquivo de configuração é necessário somente para o STM Builder, os demais STMs não têm a necessidade desta entrada como é mostrado na execução da SwissTM (linhas 21 a 24).

Após interpretar o arquivo de entrada *input* e gerar os arquivos de configuração necessários, o primeiro módulo gera como saída o arquivo *output*. Para a execução com o arquivo de entrada *input* da Figura 4.2-a, o arquivo gerado é *output* da Figura 4.2-b. O arquivo *output* tem a entrada STATUS, que se for ERROR, não executa os demais módulos porque encontrou um erro no arquivo *input*, mas se for OK executa. Tem a entrada STMS, que indica as pastas dos STMs que serão executados e estão na ordem em que foram colocados no arquivo *input*. A entrada NAMES indica os nomes das execuções que aparecerão nos gráficos, na ordem em que foram colocados no arquivo *input*. As demais entradas são as mesmas do arquivo *input*.

A.6 Padrão de saída dos Benchmarks do Framework de Testes

A Figura A.1-a é a padronização utilizada para o Microbenchmark de árvore rubro-negra, onde a linha 1 é uma identificação do teste que foi executado, a linha 2 é o número de *Commits* (Efetivações) e a linha 3 é o número de abortos. Cada execução do Microbenchmark terá como saída somente estes dados.

A Figura A.1-b é a padronização utilizada para o STMBench7, onde a linha 1 é uma identificação do teste que foi executado, a linha 2 é o *throughput* e a linha 3 é a taxa de abortos. Cada execução do STMBench7 terá como saída somente estes dados.

A Figura A.1-c é a padronização utilizada para o STAMP, onde a linha 1 é uma identificação do teste que foi executado, a linha 2 é o tempo de execucão do testes, a linha 3 é o número de *Commits* (Efetivações) e a linha 4 é o número de abortos. Cada execução do STAMP terá como saída somente estes dados.

Para implementar a padronização é necessário modificar a saída do STMBench7, do STAMP, da SwissTM, do EpochSTM e de qualquer outro STM que se deseje utilizar. Não é necessário fazer modificações no STM Builder, pois a saída minimalista dos dados

que ele possui está no padrão utilizado (entrada OUTPUT MINIMUM do arquivo de configuração).

```
1 ==== STM Builder - CMs: 1 - CM 1: Greedy - MODEL: ModelGreedy ======
2 Commit: 2145696
3 Abort: 214
```

(a)

```
1 ==== STM Builder - CMs: 1 - CM 1: Greedy - MODEL: ModelGreedy ======
2 Throughput: 163
3 Aborts: 47613.4
```

(b)

```
1 ==== STM Builder - CMs: 1 - CM 1: Greedy - MODEL: ModelGreedy ======
2 Time: 0.022592
3 Commit: 5912
4 Abort: 5
```

(c)

Figura A.1: Padrão de saída dos testes: (a) Microbenchmark de árvore rubro-negra, (b) STMBench7, (c) STAMP.

A.7 Execução individual dos Módulos do Framework de Testes

Para executar os módulos separadamente siga os passos a seguir:

Primeiro Módulo: Execute o programa *reader* para gerar o arquivo *output* e os arquivos de configuração do STM Builder.

Segundo Módulo: Execute o programa *scripts* para gerar os shell-scripts necessários, e depois execute o script *test* para executar os testes. É útil executá-lo separadamente para testar uma implementação rapidamente.

Terceiro Módulo: Execute o programa *data* para interpretar os dados gerados pelos testes. É útil executá-lo separadamente para unir testes realizados separadamente.

Quarto Módulo: Execute o programa *graphics* para criar os arquivos de configuração necessários para o *gnuplot* e depois, execute o script *graphicsgenerator* para gerar os gráficos utilizando o *gnuplot*. É útil executá-lo separadamente para unir testes realizados separadamente. Para unir testes, é necessário executar o terceiro e o quarto módulo.