PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uma Avaliação Comparativa de Sistemas de Memória Transacional de *Software* e seus *Benchmarks*

Fernando Furlan Rui

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Luiz Gustavo Leão Fernandes

Porto Alegre
2012

R934a Rui, Fernando Furlan

Uma avaliação comparativa de sistemas de memória transacional de Software e seus Benchmarks / Fernando Furlan Rui. – Porto Alegre, 2012.

110 f.

Diss. (Mestrado) – Fac. de Informática, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.

1. Informática. 2. Processamento Paralelo. 3. Arquitetura de Computador. I. Fernandes, Luiz Gustavo Leão. II. Título.

CDD 004.22

Ficha Catalográfica elaborada pelo Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "Uma Avaliação Comparativa de Sistemas de Memória Transacional de *Software* e seus *Benchmarks*", apresentada por Fernando Furlan Rui como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Processamento Paralelo e Distribuído, aprovada em 29/03/2012 pela Comissão Examinadora:

Prof. Dr. Luiz Gustavo Leão Fernandes – Orientador — PPGCC/PUCRS

Prof. Dr.Fernando Gehm Moraes – — PPGCC/PUCRS

Prof. Dr. Rodrigo da Rosa Righi - — UNISINOS

Homologada em 08/06/2012, conforme Ata No. 012 pela Comissão Coordenadora.

Prof. Dr. Paulo Henrique Lemelle Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

Dedico este trabalho a minha família e esposa.

AGRADECIMENTOS

- À minha esposa e família que sempre deram todo o apoio para realização desta etapa;
- Aos pesquisadores de outras universidades que me ajudaram neste trabalho: Aleksandar Dragojevic (Escola Politécnica Federal de Lausana), Sungpack Hong (Universidade de Stanford) e Márcio Castro (Universidade de Grenoble);
- Ao meu orientador Luiz Gustavo Fernandes pela oportunidade de crescimento e o conhecimento adquirido;
- Aos meus colegas de GMAP Dalvan Griebler e Mateus Raeder por toda a ajuda prestada.
- Ao meu colega de mestrado Alexandre Seki pelo companherismo e amizade durante esse período.

Uma Avaliação Comparativa de Sistemas de Memória Transacional de *Software* e seus *Benchmarks*

RESUMO

Memórias Transacionais são consideradas por muitos pesquisadores como a mais promissora solução para resolver problemas de programação *multicore*. Esse modelo promete escalabilidade com pequena granularidade, enquanto elimina os problemas comuns nos mecanismos convencionais de *locks*, como *deadlocks* por exemplo. Durante esses quase vinte anos de pesquisas, diversos estudos foram realizados visando identificar abordagens para maximizar o uso de Memórias Transacionais no cenário de Programação Paralela. Além disso, diversos sistemas TM foram desenvolvidos em diferentes tipos de implementações, bem como novos *Benchmarks* foram propostos para avaliação e testes de sistemas TM.

Entretanto, apesar do avanço da área, Memória Transacional não é considerada uma solução pronta para comunidade científica devido às perguntas ainda não respondidas na literatura, tais como: "Como identificar se uma aplicação terá vantagem na utilização de memórias transacionais?" e "Porque algumas aplicações não se beneficiam com o uso de Memórias Transacionais?".

Esse trabalho realiza uma avaliação comparativa de um conjunto de sistemas e aplicações transacionais, apresentando o estado da arte atual, compreendendo os problemas existentes e identificando oportunidades de crescimento nos sistemas STM de maneira a contribuir para responder as perguntas ainda em aberto na comunidade científica.

Palavras-chave: Memórias Transacionais; Memórias Transacionais de *Software*; Paralelismo; Programação Paralela; Programação *Multicore*; Mecanismos de Controle de Sincronização.

A Comparative Evaluation of Software Transactional Memory Systems and Their Benchmarks

ABSTRACT

Transactional Memory is considered by many researchers to be one of the most promising solutions to address the problem of programming multicore processors. This model provides the scalability of fine-grained locking while avoiding common issues of tradicional mechanisms, such as deadlocks. During these almost twenty years of research, several studies were carried out to identify approaches in order to maximize the use of Transactional Memories in the Parallel Programming scenario. Furthermore, several TM systems have been developed in different types of implementations as well as new Benchmarks were proposed for evaluation and testing of TM systems.

However, despite advances in the area, Transactional Memory is not considered yet a ready solution by the scientific community, due to unanswered questions in the literature, such as: "How to identify if an application has an advantage using Transactional Memory?" and "Why some applications do not benefit from the use of Transactional Memories?",

This work presents a comparative evaluation of a set of transactional applications and systems, introducing the field current state-of-the-art, understanding the existing problems and identifying growth opportunities in the STM systems in order to contribute to answer the questions that remain open in the scientific community.

Keywords: Transactional Memory; Software Transactional Memory; Parallelism; Parallel Programming; Multicore Programming; Control Mechanisms Syncronization.

LISTA DE FIGURAS

Figura 5.1 Desempenho (*speed-up*) dos sistemas STM com o *benchmark* STAMP . . . 63
Figura 5.2 Desempenho (*speed-up*) das aplicações do STAMP com os sistemas STM (Parte 1) . . . 64
Figura 5.3 Desempenho (*speed-up*) das aplicações do STAMP com os sistemas STM (Parte 2) . . . 65
Figura 6.1 Desempenho (*speed-up*) dos sistemas STM com o *benchmark* EigenBench . 73
Figura 6.2 Desempenho (*speed-up*) das aplicações com o *benchmark* EigenBench (Parte 1) . . . 74
Figura 6.3 Desempenho (*speed-up*) das aplicações com o *benchmark* EigenBench (Parte 2) . . . 75
Figura 6.4 ApC (*Aborts per Commit*) do *benchmark* EigenBench . . . 75
Figura 7.1 Desempenho (*speed-up*) da aplicação Genome conforme Tabela 7.2 . . . 85
Figura 7.2 Desempenho (*speed-up*) da aplicação Intruder conforme Tabela 7.3 . . . 87
Figura 7.3 Desempenho (*speed-up*) da aplicação SSCA2 conforme da Tabela 7.5 . . . 89
Figura 7.4 Desempenho (*speed-up*) da aplicação Vacation conforme Tabela 7.6 . . . 90

LISTA DE TABELAS

Tabela 3.1 Principais parâmetros de entrada no *benchmark* EigenBench 43
Tabela 3.2 Derivação das características transacionais através do parâmetros de entrada 44
Tabela 3.3 Aplicações do *benchmark* STAMP 45
Tabela 3.4 Principais sistemas STM na literatura de Memórias Transacionais 51
Tabela 3.5 Principais *benchmarks* na literatura de Memórias Transacionais 52
Tabela 4.1 Processadores 56
Tabela 4.2 Níveis de confiança e Valor crítico 58
Tabela 4.3 Distribuições de população 59
Tabela 5.1 Sistemas STM selecionados para testes com *benchmark* STAMP 61
Tabela 5.2 *Speed-up* das aplicações STAMP com os sistemas SwissTM e RSTM 66
Tabela 5.3 Critérios de desempenho (speed-up) 67
Tabela 5.4 Resumo das características das aplicações STAMP 67
Tabela 5.5 Classificação das aplicações do *benchmark* STAMP conforme desempenho . 67
Tabela 6.1 Características das aplicações selecionadas do *benchmark* STAMP 73
Tabela 6.2 Variação do *speed-up* entre os sistemas STM 76
Tabela 7.1 Parâmetros de entrada da aplicação Genome no *benchmark* EigenBench . . 85
Tabela 7.2 Variação da característica Comprimento da Transação na aplicação Genome . 86
Tabela 7.3 Variação da característica Localidade Temporal na aplicação Intruder 87
Tabela 7.4 Parâmetros de entrada da aplicação SSCA2 no *benchmark* EigenBench . . . 88
Tabela 7.5 Variação da característica Conjunto de Trabalho (*Cold Array*) na aplicação SSCA2 88
Tabela 7.6 Variação da característica Conjunto de Trabalho (*Hot Array*) na aplicação Vacation 89
Tabela A.1 Número de execuções sequenciais no teste de desempenho com *benchmark* STAMP 99
Tabela A.2 Número de execuções do sistema SwissTM no teste de desempenho com *benchmark* STAMP 100
Tabela A.3 Número de execuções do sistema RSTM no teste de desempenho com *benchmark* STAMP 101
Tabela A.4 Número de execuções do sistema TinySTM no teste de desempenho com *benchmark* STAMP 102
Tabela A.5 Número de execuções do sistema TL2 no teste de desempenho com *benchmark* STAMP 103
Tabela A.6 Intervalo de confiança das execuções sequenciais no teste de desempenho com *benchmark* STAMP 103

Tabela A.7 Intervalo de confiança das execuções do sistema SwissTM no teste de desempenho com *benchmark* STAMP 104
Tabela A.8 Intervalo de confiança das execuções do sistema RSTM no teste de desempenho com *benchmark* STAMP 105
Tabela A.9 Intervalo de confiança das execuções do sistema TinySTM no teste de desempenho com *benchmark* STAMP 106
Tabela A.10 Intervalo de confiança das execuções do sistema TL2 no teste de desempenho com *benchmark* STAMP 107
Tabela A.11 Número de execuções sequenciais no teste de desempenho com *benchmark* EigenBench 107
Tabela A.12 Número de execuções do sistema SwissTM no teste de desempenho com *benchmark* EigenBench 108
Tabela A.13 Número de execuções do sistema RSTM no teste de desempenho com *benchmark* EigenBench 108
Tabela A.14 Intervalo de confiança das execuções sequenciais no teste de desempenho com *benchmark* EigenBench 109
Tabela A.15 Intervalo de confiança das execuções do sistema SwissTM no teste de desempenho com *benchmark* EigenBench 109
Tabela A.16 Intervalo de confiança das execuções do sistema RSTM no teste de desempenho com *benchmark* EigenBench 110

LISTA DE SIGLAS

ApC	*Aborts per Commit*
ECD	*Eager Conflict Detection*
EVM	*Eager Version Manangement*
HCD	*Hybrid Conflict Detection*
HTM	*Hardware Transactional Memory*
HyTM	*Hybrid Transactional Memory*
ISA	*Instruction Set Architecture*
LCD	*Lazy Conflict Detection*
LVM	*Lazy Version Management*
RMS	*Recognition, Mining and Synthesis*
SSCA2	*Scalable Synthetic Compact Applications 2*
STAMP	*Stanford Transactional Applications for Multi-Processing*
STM	*Software Transactional Memory*
TM	*Transactional Memory*
YADA	*Yet Another Delaunay Application*

SUMÁRIO

1. INTRODUÇÃO 23
1.1 Motivação e Objetivos . . . 24
1.2 Estrutura do Texto . . . 25

2. Memórias Transacionais 27
2.1 Motivação . . . 27
2.2 Conceitos Básicos . . . 30
2.3 Implementações . . . 31
2.3.1 STM . . . 32
2.3.2 HTM . . . 33
2.3.3 HyTM . . . 34
2.3.4 Conclusão . . . 35
2.4 Decisões de Projeto . . . 35

3. Estado da Arte 39
3.1 Sistemas STM . . . 39
3.1.1 SwissTM . . . 39
3.1.2 RSTM . . . 40
3.1.3 TL2 . . . 40
3.1.4 TinySTM . . . 41
3.1.5 Intel STM . . . 41
3.2 *Benchmarks* . . . 41
3.2.1 EigenBench . . . 42
3.2.2 LeeTM . . . 44
3.2.3 RMS-TM . . . 45
3.2.4 STAMP . . . 45
3.2.5 STMBench7 . . . 48
3.3 Trabalhos Relacionados . . . 49
3.3.1 *Benchmarks* e Avaliações de Sistemas . . . 49
3.3.2 Técnicas de *Tracing* e *Profiling* em Aplicações STM . . . 50
3.4 Resumo do Capítulo . . . 51

4. Metodologia 55
4.1 Arquitetura . . . 55

4.2 Análise Estatística 56
4.2.1 Confiabilidade dos Resultados 57

5. Teste de Desempenho com *Benchmark* STAMP 61
5.1 Metodologia 61
5.2 Resultados 62
5.3 Resumo do Capítulo 68

6. Teste de Desempenho com *Benchmark* EigenBench 71
6.1 Metodologia 71
6.2 Resultados 72
6.3 Resumo do Capítulo 80

7. Teste de Características Transacionais com *Benchmark* EigenBench 83
7.1 Metodologia 83
7.2 Resultados 84
7.2.1 Teste de Características Transacionais com Aplicação Genome 84
7.2.2 Teste de Características Transacionais com Aplicação Intruder 86
7.2.3 Teste de Características Transacionais com Aplicação SSCA2 87
7.2.4 Teste de Características Transacionais com Aplicação Vacation 89
7.3 Resumo do Capítulo 90

8. Conclusão 91

Referências 93

A. APÊNDICE 99
A.1 Teste de Desempenho com *Benchmark* STAMP 99
A.2 Teste de Desempenho com *Benchmark* EigenBench 99

1. INTRODUÇÃO

A tecnologia *multicore* tem-se mostrado capaz de solucionar o problema de alto desempenho sem o aumento do consumo de energia. Entretanto, para que seja utilizada toda a capacidade desta tecnologia, é necessário que as aplicações atingam um alto grau de concorrência [12, 35]. Os mecanismos de sincronização tradicionais possuem diversas limitações em termos de complexidade de código, baixa granularidade, pouca escalabilidade, dificuldade de gerenciamento e propensão a *deadlock* [12, 35, 68]. Além disso, a programação paralela para processadores *multicore* é algo não trivial. Aplicações necessitam explorar melhor as vantagens de tal tecnologia, facilitando a programação e gerenciando a sincronização de dados compartilhados de maneira mais efetiva.

Memórias Transacionais fornecem um mecanismo flexível e simples para programação paralela em processadores *multicore*, usando abstração de alto nível, ao contrário do mecanismo de *locks* [46, 68]. Dessa forma, o problema de sincronização de dados é passado para o sistema de Memória Transacional, que se torna responsável por garantir um correto funcionamento da aplicação (sem *deadlocks*, condições de corrida, etc.), explorando de fato, o paralelismo das aplicações. Memória Transacional é considerada por muitos pesquisadores como a mais promissora solução para resolver problemas de programação *multicore* [8].

Entretanto, ao se abstrair o mecanismo de sincronização para o sistema de Memória Transacional (TM - *Transactional Memory*), ocorre perda de desempenho, e importantes decisões de projetos precisam ser tomadas para minimizar tal impacto. Diversas propostas de Memórias Transacionais têm sido apresentadas por pesquisadores em diferentes maneiras de implementação, como *software* (STM - *Software Transactional Memory*), *hardware* (HTM - *Hardware Transactional Memory*) e híbrido (HyTM - *Hybrid Transactional Memory*). Dentre as possibilidades, a analisada neste trabalho é a STM, devido às seguintes vantagens [46, 68]:

- É mais simples de modificar e possui poucas limitações de arquitetura, em comparação ao *hardware*;
- Possui portabilidade da aplicação sem a necessidade de *hardware* específico;
- Pode possuir qualquer tamanho de transação ou tempo de duração, principal limitação de sistemas baseados em *hardware*.

Após quase vinte anos desde a primeira prosposta de Memória Transacional [43], ela ainda não é considerada uma solução final na área, devido a certas perguntas ainda não respondidas, tais como sobre as vantagens de uma aplicação na utilização de Memória Transacional ou sobre porque algumas aplicações não se beneficiam com o uso de Memória Transacional. Para resolver essas questões, pesquisadores estão buscando investigar profundamente os sistemas TM com o objetivo de identificar quais aplicações se beneficiam com eles, que características eles devem possuir para superar o desempenho de aplicações em comparação com mecanismos de *locks*, entre outros

resultados que podem ser obtidos para análise dos sistemas e das aplicações TM. Resultados atuais apresentados em [1, 18], indicam uma melhora no desempenho em relação a trabalhos mais antigos, embora ainda não seja suficientes se observado o contexto geral. Além disso, apesar da grande quantidade de implementações, bem como de *benchmarks* específicos existem poucos trabalhos que direcionam seu foco no entendimento de Memórias Transacionais. É um desafio para a área saber que tipo de aplicações são beneficadas com o uso de Memórias Transacionais, pois faltam ferramentas e mecanismos para analisar e compreender seu estado atual [9, 36].

1.1 Motivação e Objetivos

Há um crescente aumento de trabalhos de pesquisa sobre Memórias Transacionais. Segundo [35], com o passar dos anos, houve um crescimento no interesse de pesquisadores, inclusive vindo da indústria, por essa área. Importantes empresas como Intel, IBM e Microsoft estão investindo esforços para o desenvolvimento de soluções TM. Durante esses quase vinte anos de pesquisas (primeira proposta em [43]), diversos estudos foram realizados visando identificar abordagens para maximizar a programação paralela com o uso de Memórias Transacionais. Além disso, diversos sistemas TM foram desenvolvidos em diferentes tipos de implementação (*Hardware*, *Software* e Híbrido), e alguns *benchmarks* foram desenvolvidos exclusivamente para avaliação e testes desses sistemas. Entretanto, Memória Transacional não é considerada, ainda, uma solução pronta para comunidade científica, devido às perguntas ainda não respondidas pelas pesquisas realizadas até o presente momento. Algumas das perguntas não respondidas são listadas abaixo:

- Em relação ao desempenho, qual o estado atual dos sistemas STM propostos na literatura?
- Porque algumas aplicações não possuem desempenho satisfatório com Memórias Transacionais?
- Quais aplicações têm vantagem na utilização de Memórias Transacionais?
- Qual o impacto que as caraterísticas transacionais têm no desempenho das aplicações ao quando são utilizadas Memórias Transacionais?
- Qual as decisões de projeto utilizadas em sistemas de Memórias Transacionais possuem melhor desempenho?

Essas e outras perguntas são encontradas em trabalhos atuais de Memória Transacional. Segundo [36], apesar da grande variedade de implementações TM, ainda é um desafio saber que tipo de aplicações pode realmente tirar proveito de sistemas TM. Além disso, em [9], o autor salienta que, embora vários sistemas TM tenham sido propostos na literatura, ainda faltam ferramentas e mecanismos necessários para analisar e comparar os trabalhos propostos. O trabalho recente [8], questiona a capacidade de sistemas STM proporcionar bom desempenho e sugere o termo "*research toy*"(brinquedo de pesquisa) ao atual estado da área. Isso é consequência das promessas realizadas

sobre Memórias Transacionais há alguns anos atrás quando a área estava começando a ser explorada. Entretanto, implementações e propostas atuais não conseguem garantir o desempenho e a facilidade de uso prometidas.

Esse trabalho tem por objetivo realizar uma avaliação comparativa de um conjunto de sistemas e aplicações transacionais, buscando apresentar o estado atual da área, compreendendo as perguntas da comunidade na prática, identificando melhorias e oportunidades de crescimento nos sistemas STM de maneira a contribuir para a comunidade responder às perguntas ainda em aberto. Através de uma avaliação comparativa, é possível investigar o comportamento das aplicações para Memórias Transacionais, identificando suas características transacionais mais importantes, analisando seu desempenho, buscando entender seu comportamento. Também é possível investigar o comportamento dos sistemas STM atuais, identificando problemas de implementação e apontando melhorias e oportunidades de crescimento. Através da abordagem comparativa, é analisado o comportamento tanto das aplicações utilizadas, ao executar aplicações com diversos sistemas STM, bem como investigar os sistemas STM, ao comparar os resultados de aplicações entre os sistemas utilizados.

1.2 Estrutura do Texto

O trabalho está organizado da seguinte forma: este capítulo apresenta uma introdução geral sobre contexto de Memórias Transacionais, além da motivação e do objetivo para realização da trabalho. No Capítulo 2, é apresentado o conceito de Memórias Transacionais, de onde surgiu e como é utilizado. São também apresentadas as maneiras de se implementarem sistemas de Memórias Transacionais e as principais decisões de projeto necessárias para o desenvolvimento desses sistemas. O Capítulo 3 apresenta os trabalhos atuais sobre o tema, os principas sistemas STM propostos na literatura, os *benchmarks* propostos para testes, a avaliação desses sistemas e trabalhos relacionados ao tema. No Capítulo 4, é descrita a metodologia geral dos testes executados e os detalhes da arquitetura escolhida. É apresentado nesse capítulo o uso de métodos estatísticos para auxiliar na obtenção de resultados mais precisos nos testes executados. No Capítulo 5, é apresentado o teste de desempenho com o *benchmark* STAMP. No Capítulo 6, é apresentado o teste de desempenho com o *benchmark* EigenBench. O teste de características transacionais com o *benchmark* EigenBench é apresentado no Capítulo 7. Finalmente, o Capítulo 8 apresenta conclusões e possíveis trabalhos futuros.

2. Memórias Transacionais

O conceito de transação foi utilizado inicialmente na área de pesquisa de banco de dados [68]. Uma *Database Transaction* é uma sequência de ações indivisível e instantânea [35]. Desse modo, *Transaction Memory*, ou Memória Transacional, é a utilização do conceito de transação de Banco de Dados para o contexto de acesso a memória de um sistema. Memórias Transacionais são utilizadas para substituir regiões críticas protegidas por *lock*, em programas *multi-thread*, por transações (blocos atômicos) [68].

Memórias Transacionais são consideradas por muitos pesquisadores como a mais promissora solução para resolver problemas de programação *multicore*. Esse modelo promete escalabilidade com pequena granularidade, enquanto elimina os problemas comuns nos mecanismos convencionais de *locks*, como *deadlocks* por exemplo [8].

Nesse capítulo, é apresentado uma visão geral sobre Memória Transacional. Primeiramente, é apresentado a motivação para o uso de Memórias Transacionais (Seção 2.1). Após, são apresentados conceitos básicos, propriedades e comparações em relação ao mecanismo de *locks* (Seção 2.2). Em seguida, as diferentes maneiras de implementar Memórias Transacionais são mostradas na Seção 2.3. Por fim, são analisadas as decisões de projeto (*design choices*) para o projeto de Memórias Transacionais na Seção 2.4.

2.1 Motivação

A necessidade de alto desempenho, sem que se aumente o consumo de energia e calor, tornou-se uma importante área de estudos. A tecnologia *multicore* tem se mostrado capaz de realizar tanto desempenho quanto redução no consumo [35]. Processadores *multicore* são predominantes em servidores, *desktops* e também em sistemas embarcados. Entretanto, desempenho e o baixo consumo só podem ser obtidos se as aplicações atingirem um alto grau de concorrência [12]. Apesar do avanço da tecnologia, a programação paralela para processadores *multicore* é algo não trivial [68]. Aplicações necessitam de controle avançado de concorrência.

Um objeto concorrente é uma estrutura de dados compartilhada por processos concorrentes. Técnicas convencionais de implementação de tais estruturas são baseadas em seções críticas de código, para garantir que apenas um processo por vez possa acessar o objeto concorrente [42].

Uma estrutura de dados compartilhados (ou objeto concorrente) é denominada sem bloqueios (*lock-free*), se suas operações não necessitam de exclusão mútua. Segundo [43], estruturas *lock-free* eliminam problemas comuns associados a técnicas convencionais de bloqueio, como:

- **Inversão de prioridade:** ocorre quando processos de baixa prioridade são antecipados mantendo processos de maior prioridade bloqueados;
- ***Convoying:*** ocorre quando um processo perde o processador (como quando ocorre uma inter-

rupção ou devido ao limite de tempo excedido). Quando interrupções desse tipo acontecem, outros processos prontos para execução são impossibilitados de prossegir;

- ***Deadlock:*** pode ocorrer se processos decidem bloquear o mesmo conjunto de objetos em ordens diferentes.

Apesar de sua ampla utilização, *locks* possuem sérias limitações, salienta [12] ao comparar as granularidades utilizadas em mecanismos de *lock*. Segundo os autores, *locks* de alta granularidade que protegem uma grande quantidade de dados não são escaláveis. Bloqueiam objetos compartilhados que não são conflitantes, tornando-se uma fonte de disputa. Já *locks* de baixa granularidade (que protegem pequenas regiões de código) apresentam-se mais escaláveis, porém são difíceis de serem utilizados de uma maneira correta e eficaz. Podem, inclusive, introduzir problemas de engenharia de *software*, visto que as convenções de associação de *lock* tornam-se mais complexas e propensas a erros.

De modo complementar, observou-se em [35] alguns dos problemas enfrentados na utilização de mecanismos de sincronização tradicionais:

- São difíceis de gerenciar efetivamente, especialmente em sistemas complexos;
- Podem causar bloqueio, fazendo com que processos aguardem para liberação de um *lock*;
- São vulneráveis a falhas, como *deadlocks*;
- Limitam a escalabilidade e adicionam complexidade ao código fonte.

A Listagem 2.1 demonstra alguns dos problemas citados sobre o mecanismo de *lock* [65]. Neste exemplo, assume-se que o código necessita ser *thread-safe* e que múltiplas *threads* podem executá-lo concorrentemente.

Listagem 2.1: Código sem sincronização

```
long counter1;
long counter2;
time_t timestamp1;
time_t timestamp2;

void f1(long *r, time_t *t) {
        *t = timestamp1;
        *r = counter1++;
}

void f2(long *r, time_t *t) {
        *t = timestamp2;
```

```
        *r = counter2++;
}

void w1(long *r, time_t *t) {
        *r = counter1++;

        if (*r & 1)
                *t = timestamp2;
}

void w2(long *r, time_t *t) {
        *r = counter2++;

        if (*r & 1)
                *t = timestamp1;
}
```

O primeiro problema, observado em [65] refere-se a modelagem dos *locks*. Nesse código, a utilização de um simples *lock* (apesar de implementado corretamente) tornaria a execução lenta, visto que f1 e f2 possuem variáveis não conflitantes. Porém, a inclusão de dois *locks* também não é possível, pois nos outros dois métodos as variáveis acessadas são invertidas, sendo necessário, portanto, a inclusão de quatro *locks* (um para cada variável em questão). Neste ponto, é possível identificar a dificuldade na definição de quantos *locks* e onde estes serão colocados no código fonte.

Após a definição dos *locks*, o código ficaria conforme apresentado na Listagem 2.2 (apresentando apenas 2 métodos, os outros são análogos):

Listagem 2.2: Código com a inclusão de *locks*

```
void f1_1(long *r, time_t *t) {
        lock(l_timestamp1);
        lock(l_counter1);
        *t = timestamp1;
        *r = counter1++;
}

void w1_2(long *r, time_t *t) {
        lock(l _ counter1);
        lock(l _ timestamp1);

        *r = counter1++;
        if(*r&1)
```

```
                *t = timestamp2;

        unlock(l_timestamp1);
        unlock(l_counter1);
}
```

Apesar do código parecer pronto para execução, ainda pode-se identificar o problema de *deadlocks* se os métodos forem executados em *threads* separadas e em ordem diferente.

Através desse exemplo é possível verificar na prática a dificuldade na utilização de mecanismos de *locks*. Por fim, *locks* podem causar vulnerabilidades em falhas de *threads* e *delays*, inibir concorrência (pois precisam ser usados de maneira conservadora) e possuem ainda desvantagens em termos de consumo de energia [12].

2.2 Conceitos Básicos

Como citado anteriormente, o conceito de transação é herdado do conceito de *database transactions*. Assim como transações de banco de dados, Memórias Transacionais possuem propriedades de modo a garantir o real funcionamento das transações em seu contexto. Uma transação de banco de dados, possui as propriedades chamadas de ACID: atomicidade, consistência, isolamento e durabilidade [35]. Implementações gerais de Memórias Transacionais geralmente não fornecem as propriedades de consistência e durabilidade. A primeira, devido ao fato de memórias serem voláteis. A segunda, por sua vez, não é, geralmente, aplicada visto que Memórias Transacionais não possuem o conceito de metadados, não havendo regras de consistência a serem garantidas durante a transação. Portanto, para Memórias Transacionais, as seguintes propriedades são válidas:

- **Atomicidade:** refere-se à capacidade do sistema garantir que ou todas as tarefas de uma transação sejam executadas ou nenhuma delas seja.

- **Isolamento:** refere-se ao requisito que determina que outras operações não possam acessar os dados durante a execução da transação.

Abaixo são descritas algumas características do mecanismo de Memórias Transacionais, de maneira a exemplificar seu funcionamento [12, 68]:

- Antes do *commit* de uma transação, seu resultado não é visível fora da transação;

- Cada transação é executada em uma simples *thread* sem a aquisição de *lock*;

- Quando uma transação inicia, ela guarda um registro dos valores antigos, que pode ser restaurado se abortado;

- Uma transação não pode escrever diretamente na memória compartilhada. Em vez disso, seu resultado é armazenado em um *undo-log* ou um *write-buffer* mantido pelo sistema;

- Se a transação completar sem conflito com outra transação, é feito *commit* e seu resultado é armazenado na memória compartilhada. Caso contrário, se um conflito é detectado entre duas transações, uma delas fará *rollback*, restaurando o registro guardado anteriormente;
- Para detectar conflitos *read-write* ou *write-write*, as referências de memória são monitoradas.

Segundo [68], transações possuem diversas vantagens quando comparadas com o mecanismo de *locks*. Primeiramente, programadores não precisam se preocupar na eficiência e desempenho do esquema de *locks*. Segundo, é garantido que um objeto compartilhado está consistente mesmo após um evento de falha. Terceiro, transações são feitas naturalmente, o que torna o desenvolvimento de *software multicore* mais fácil.

Complementando, o modelo de programação em Memórias Transacionais oferece um novo meio de desenvolvimento de aplicações paralelas usando abstração de alto nível, ao contrário do mecanismo de *lock*. Esse modelo passa o problema de sincronização para o sistema de Memórias Transacionais, que é responsável por garantir que *deadlocks* não ocorram, condições de corridas sejam tratadas e *locks* sejam alocados em uma granularidade que explorem de fato o paralelismo das aplicações [46].

2.3 Implementações

Nesta seção são apresentadas as formas de implementação de Memórias Transacionais. É apresentado inicialmente um exemplo de código com Memórias transacionais.

Como apresentado na Seção 2.1, utilizou-se o mesmo código fonte para demonstrar o resultado com Memórias Transacionais [65]:

Listagem 2.3: Código com o uso de Memórias Transacionais

```
void f1_1(long *r, time_t *t) {
        tm_atomic {
                *t = timestamp1;
                *r = counter1++;
        }
}

void w1_2(long *r, time_t *t) {
        tm_atomic {
                *r = counter1++;

        if (*r & 1)
                *t = timestamp2;
        }
}
```

O código fonte com a utilização de Memórias Transacionais torna-se extremamente simples. Todo o controle de concorrência é realizado pelo sistema de Memória Transacional.

Uma vez que há um crescente interesse de importantes equipes de pesquisadores, bem como empresas internacionais, na pesquisa sobre TM, muitas soluções estão sendo propostas [35].

Memórias Transacionais podem ser implementadas em *Hardware (HTM)* [13, 14, 23, 29, 32] ou em *software (STM)* [2,16,25,40,44,51,66], ou com uma combinação híbrida de *Hardware/Software (HyTM)* [10, 46, 49, 50, 59].

Soluções HTM fornecem baixo *overhead* em comparação a soluções STM, mas possuem limitações de arquitetura. Estão começando a ser disponbilizadas, visto que os mecanismos necessitam de *hardware* específico, além de possuírem limitações de espaço na *cache*. Soluções STM, por outro lado, necessitam de mais *overhead* para gerenciar transações. Entretanto, transações podem utilizar qualquer tamanho ou tempo de duração, e ainda são independentes de *hardware*. Finalmente, soluções HyTM incorporam as vantagens de ambas as metodologias, usando HTM quando possível e recorrendo a STM se as transações forem maiores que a *cache* [25, 35].

Nas próximas seções são apresentadas as características de cada solução, identificando as vantagens e desvantagens encontradas nas pesquisas desenvolvidas até o momento.

2.3.1 STM

Geralmente, sistemas STM podem ser implementados como uma biblioteca de uma linguagem de programação ou diretamente no compilador [35]. A primeira alternativa, que é mais comum, precisa ser explicitamente adicionada no código fonte pelo programador através de blocos atômicos. Implementações como [2, 16, 25, 52, 66] utilizam essa abordagem de desenvolvimento. A segunda alternativa é a utilização de mecanismos TM na implementação do compilador da linguagem. Essa abordagem é utilizada por [21].

Segundo [8], a vantagem de uma solução STM para os programadores de sistema é que ela oferece flexibilidade na aplicação de diferentes mecanismos e políticas para essas operações. Além disso, *software* é mais fácil de ser modificado e possui pouca limitação de tamanho, como estruturas de *hardware* [35]. Possui, ainda, uma grande vantagem para usuários finais em relação a portabilidade de aplicações sem a necessidade de *hardware* específico. Em [25], soluções HTM possuem limitação de arquitetura, nas quais transações necessitam ser menores que o espaço da *cache*, tornando soluções HTM limitadas pelo tamanho das transações e também pelo tempo de execução. Em [2], os autores acreditam que, mesmo que o suporte a HTM seja amplamente utilizado no futuro, apenas aplicações com transações de escala pequena serão complementamente utilizadas em *hardware*. Enquanto isso, STM continuará sendo necessário para transações grandes.

Apesar das diversas pesquisas, soluções STM possuem algumas limitações de desempenho e atomicidade de transações. Segundo [57], muitas das pesquisas anteriores em STM possuam limitações em dois aspectos: primeiro, estudos utilizavam simulações e propostas com transações pequenas, gerando resultados com limitações; segundo, muitos dos sistemas utilizam soluções baseadas em bibliotecas de linguagens de programação. Uma abordagem com base em biblioteca torna

difícil lidar com cargas de trabalho complexas e grandes transações, pois necessitam de um controle maior do programador. Porém, para a maioria de propostas STM, o desempenho baixo é uma séria desvantagem [17].

Outro desafio em soluções STM refere-se a propriedade de isolamento. Em [68], os autores relatam tal desafio comparando os dois tipos de isolamento. Ao utilizar isolamento forte, o código não transacional não poderá ler/escrever em dados não armazenados, prejudicando o desempenho do sistema. Ao utilizar isolamento fraco, o isolamento será utilizado apenas em transações e o código não transacional poderá ler/escrever em dados não armazenados, trazendo assim alto desempenho, mas podendo gerar resultados não esperados.

Uma grande variedade de técnicas STM, inspiradas por algoritmos de banco de dados, foram explorados [2]. O grande desafio que enfrentam os pesquisadores STM é determinar a combinação adequada de estratégias que atendam aos requisitos de aplicativos concorrentes. Entretanto, muitos experimentos STM foram avaliados utilizando *benchmarks* caracterizados por pequenas transações, dados simples e uniformes, ou ainda acesso a dados padrão. Tais experimentos não representam os problemas encontrados em aplicações reais.

Finalmente, reduzir o *overhead* de soluções STM ao ponto de tornar-se viável é uma tarefa difícil, e resultados significativamente melhores precisam ser demonstrados [8]. De acordo com [63], mesmo com otimização de código pelo compilador, sistemas STM podem sofrer queda de 40% no desempenho em cada *thread* de execução.

2.3.2 HTM

Ao contrário das soluções STM, as soluções HTM implementam todas as suas funcionalidades em *hardware* [35]. A ideia básica desse sistema é modificar os protocolos de coerência de *caches* modernos, a fim de implementar transações.

Comparado com STM, HTM naturalmente possui vantagens de alto desempenho e atomicidade forte. Tipicamente, sistemas HTM utilizam *caches* em *hardware* para guardar dados de leitura e escrita para cada transação, e utilizam o protocolo de coerência para detectar conflitos entre transações concorrentes [68]. Utilizando *hardware*, os sistemas TM reduzem o *overhead* de mecanismos de *locks* de granularidade baixa, possuindo melhor desempenho não só em relação a soluções STM, mas também em sincronização baseada em *locks*. Além disso, garantem atomicidade forte, com nenhum ou pouco *overhead* adicional [17].

Embora sistemas HTM são viáveis por manter *overhead* baixo, existem limitações de tamanho e comprimento de transações em *hardware*. Estas limitações impedem os sistemas HTM de serem úteis na prática [60].

Um sistema HTM é tipicamente classificado pelas escolhas de versionamento e mecanismos de detecção de conflitos [61]. Uma abordagem utilizada é o versionamento ansioso (*eager versioning*) e a detecção de conflitos ansiosa (*eager conflict detection*). Nessa abordagem, os dados são armazenados diretamente na memória (sem armazenamento temporário) e as transações são abortadas assim que conflitos sejam identificados. Essa abordagem garante uma boa escalabilidade, porém é

propensa a *deadlocks/livelocks* e necessita de políticas de contenção. Já uma outra abordagem é a utilização de versionamento preguioso (*lazy versioning*) e detecção de conflitos preguiosa (*lazy conflict detection*). Essa abordagem é livre de *deadlocks*, mas, em contra partida, ela prejudica a escalabilidade da solução.

Em [5], os autores salientam que ainda não é claro que sistemas HTM de grande escala fornecerão implementações práticas, semanticamente aceitáveis e com um bom custo-benefício. Segundo eles, sistemas HTM possuem 3 principais problemas:

- **Promovem arquitetura ambiciosa:** fazendo com que o mercado exiga provas mais convincentes para garantir o investimento.

- **Existem políticas não explícitas:** decisões que não são atualmente compreendidas e que, ainda não possuem uma abordagem aceitável.

- **Suporte apenas a máquinas atuais:** programas projetados em sistemas HTM possivelmente não funcionarão em máquinas antigas.

Finalizando, sistemas HTM possuem outros desafios de implementação. O relacionamento entre transações e operações de entrada e saída (I/O) é um importante desafio de pesquisa [68]. Como operações desse tipo estão fora do escopo do sistema TM, é difícil utilizarmos o conceito de *rollback*. Outro desafio em sistemas HTM é o fato de não haver, ainda, um padrão do conjunto de instruções suportadas (ISA).

2.3.3 HyTM

Como observado nas Seções 2.3.1 e 2.3.2 existem limitações na implementação tanto de sistemas HTM como de sistemas STM. Apesar do desempenho de sistemas STM terem se tornado mais escaláveis com o aumento do número de núcleos em processadores *multicore*, o *overhead* desses sistemas em *software* é ainda significante [23]. Por outro lado, sistemas HTM apresentam diversas limitações impostas pelo *hardware*. Dessa forma, uma técnica para solucionar a limitação de recursos de *hardware* e o *overhead* gerado no *software* é a utilização dos sistemas híbridos (HyTM).

Propostas de sistemas puramente HTM possuem a vantagem de serem rápidas, mas são altamente ambiciosas e incorporam políticas não reconhecidas. Propostas de sistemas STM fornecem flexibilidade substancial de políticas, porém incluem um *overhead* significativo em versionamento e validação em relação a conflitos de transações [4].

Existem diversas formas de implementar sistemas híbridos. Em [50], é proposta uma abordagem de implementação em *software* que utiliza HTM para aumentar o desempenho e não depender apenas do sistema STM. Em [4], os autores propõem que o *hardware* serve apenas para otimizar o desempenho das transações que são fundamentalmente controladas por *software*. Já em [13], sugere-se que pequenas transações utilizem a *cache* e grandes transações utilizem *software*. Entretanto, a interação entre o sistema de *software* precisa ser da mesma maneira como executada em *hardware*.

Porém, o principal desafio de sistemas Híbridos é detectar conflitos entre transações iniciadas em *hardware* e transações iniciadas em *software*.

2.3.4 Conclusão

Como visto nas seções anteriores existem três implementações possíveis para TM, cada qual com suas vantagens e desvantagens. Dentre as possibilidades, a analisada neste trabalho é a STM, devido às seguintes vantagens [40, 44]:

- É mais simples de modificar e possui poucas limitações de arquitetura, em comparação ao *hardware*;
- Possui portabilidade da aplicação sem a necessidade de *hardware* específico;
- Pode possuir qualquer tamanho de transação ou tempo de duração, principal limitação de sistemas baseados em *hardware*.

Além disso, pesquisas recentes de HTM ainda indicam a limitação do tamanho da cache como um problema para implementações TM baseadas em Hardware. Em [11], o autor salienta que mais trabalhos são necessários para avaliar estratégias para solucionar problemas como *overflow* do tamanho da cache e transações que abortam repetidas vezes devido a conflitos de dados. A proposta HTM [14], por exemplo, não utiliza o sistema TM para transações que excedem o limite do *hardware*. Em [49], uma proposta HyTM, o autor salienta que sistemas híbridos funcionam corretamente se possuírem recursos de *hardware* suficientes, entretanto o desempenho é reduzido para execução linear se os recursos de *hardware* são excedidos.

2.4 Decisões de Projeto

Existem diversos sistemas de Memória Transacional propostos na literatura, cada uma apresenta suas características e mecanismos para controle das transações. Tais características são denominadas decisões de projeto (*design choices*), pois são definidas na fase de construção do sistema. Basicamente, as seguintes decisões de projeto são levadas em consideração na construção de uma sistema de Memória Transacional de *Software*:

- **Granularidade da transação (*transaction granularity*):** é a unidade de armazenamento na qual o sistema detecta o conflito. Para linguagens orientadas a objetos, é comum utilizar granularidade de objeto (*object granularity*), que detecta conflitos quando o estado do objeto é alterado. Existem ainda granularidade de palavra (*word granularity*) que utiliza uma palavra de memória como unidade de conflito, ou de bloco (*block granularity*) que utiliza um bloco de palavras de memória. A escolha na granularidade influência diretamente o desempenho do sistema TM. Essa característica também é conhecida na literatura por *word-based* (baseado em palavra) ou *object-base* (baseado em objeto). Um estudo sobre o tamanho ideal da

granularidade é apresentado em [2], onde testes são realizados com diferentes tamanhos de granularidade. O resultado apresenta granularidade de 4 palavras como o melhor em relação ao desempenho do que granularidade de blocos ou com 1 palavra em 4% a 5% respectivamente.

- **Gerenciamento de Versão: (*version management*):** é o mecanismo que gerencia versões diferentes do mesmo dado lógico: as novas versões atualizadas vindas de diferentes transações e a versão antiga com o dado original em caso de a transação falhar.

  Geralmente, existem dois tipos de gerenciamento de versões: Gerenciamento de versão preguiçoso (LVM - *Lazy Version Management*) e Gerenciamento de versão ansioso (EVM - *Eager Version Management*). Na literatura é possível encontrar variações para o termo *Version Management* como: *lazy-versioning* (versionamento preguiçoso) ou *eager-versioning* (versionamento ansioso).

  Em sistemas que utilizam LVM, versões antigas se mantêm em seu lugar original e novas versões são armazenadas em um *buffer* específico da transação. Quando a transação executa o *commit* a versão nova substitui a versão antiga e o endereço do dado novo é removido do *buffer* temporário. Quando a transação aborta, a versão nova é descartada do *buffer* temporário diretamente. Portanto, LVM é mais eficiente em transações que abortam. Além disso, múltiplas transações podem acessar objetos compartilhados concorrentemente, e cada uma manter uma versão privada em seu *buffer*. Finalmente, LVM permite leitura e escrita concorrentemente de uma mesmo dado.

  Em sistemas EVM, uma nova versão substitui versões antigas diretamente, enquanto a versão antiga é salva em um *buffer* temporário. Comparado com LVM, EVM reduz o custo de copiar todos os dados, pois a versão nova é armazena diretamente no endereço da versão antiga. Entretanto, isso impossibilita a leitura do objeto modificado por outras transações, limitando a concorrência. Com EVM, o *commit* de uma transação é simples, basta descarta os dados do *buffer* temporário. Quando uma transação aborta, a versão antiga é restaurada do *buffer* temporário e a nova versão é descartada. Portanto, EVM é mais eficiente para transações que executam com sucesso, especialmente quando o *commit* é frequente.

- **Detecção de conflito: (*conflict detection*):** um conflito acontece quando duas transações acessam o mesmo dado e pelo menos uma o modifica. Quando um conflito é detectado, o sistema TM busca solucionar o problema, geralmente, abortando uma das transações envolvidas.

  Geralmente existem, três tipos de detecção de conflito em sistemas TM: detecção ansiosa (*ECD - Eager Conflict Detection*), detecção preguiçosa (*LCD - Lazy Conflict Detection*) e detecção híbrida (*HCD - Hybrid Conflict Detection*). Na literatura é possível encontrar variações para o termo *Conflict Detection* como: otimista (que corresponde a preguiçosa) ou pessimista (que corresponde a ansiosa); lazy-acquire (aquisição preguiçosa) ou eager-acquire (aquisição ansiosa); *commit-time* (ao realizar o *commit*) e *encounter-time* (ao encontrar um conflito).

ECD detecta conflitos no momento que as transações acessam a memória enquanto LCD detecta conflitos quando a transação está para executar o *commit*. ECD sempre trabalha em conjunto com EVM, visto que, é necessário para garantir que apenas uma transação escreva uma nova versão no objeto. Portanto, o sistema deve detectar o conflito antecipadamente. Da mesma forma, LCD trabalha, geralmente, com LVM visto que, todas as atualizações são privadas e invisíveis para as demais transações. Portanto, o sistema detecta conflitos apenas no final de uma transação. A combinação LVM e ECD é utilizada em alguns sistemas HTM, entretanto é raro em sistemas TM baseados em *software*. Por outro lado, EVM não pode trabalhar com conjunto com LCD pois, apenas uma nova versão pode ser armazenada no objeto, por esse motivo, detecção de conflito precisa ser verifica antecipadamente para garantir que apenas uma transação escreva no objeto.

Alguns sistemas STM utilizam HCD, que combina detecção ansiosa e preguiçosa. Nesse modelo de detecção o gerenciamento de versão opera em EVM e utiliza ECD antes de uma transação modificar um dado, mas permite que transações leiam o dado compartilhado concorrentemente e também para atrasar a detecção de conflito antes do *commit*. SwissTM [2], por exemplo, utiliza uma versão de HCD (denominada *mixed acquire*) que possui deteção de conflito otimista para conflitos de leitura/escrita e detecção de conflito pessimista para conflitos de escrita/escrita. Ao detectar conflitos escrita/escrita antecipadamente, o sistema evita perder tempo e processamento com transações que irão abortar. E detectando conflitos leitura/escrita tardiamente possibilita uma maior concorrência.

Uma característica importante dos detectores de conflitos é a visibilidade das leituras na memória. Uma transação T que lê o dado x pode ser visível (*visible read*) ou invisível (*invisible read*) para outras transações que detectam x. Quando T é invisível, T necessita detectar conflitos no dado x se outra transação tentar escrever. Isso é chamado na literatura de validação do conjunto de leitura. O tempo necessário de um algoritmo de validação básico é proporcional ao tamanho do conjunto de leitura, porém existem mecanismos que diminuem esse tempo. Existem 2 algortimos de validação conhecidos na literatura: global commit counter heuristic [64] (utilizado no sistema RSTM) e *time-based scheme* [16, 52] (utilizado no SwissTM, TL2 e TinySTM).

- **Gerenciamento de Contenção: (*contention management*):** é o mecanismo que determina qual a transação envolvida em um conflito necessita ser desfeita. Quando um conflito acontece, as transações envolvidas são divididas em duas partes: atacantes e defensores. As atacantes são as transações requisitando acesso a memória compartilhada enquanto as defensoras são as transações que recebem a requisição das atacantes. Em um sistema com detecção ansiosa, a transação atacante é a que está tentando ler da memória compartilhada. Já no sistema com detecção preguiçosa, a transação atacante é a que está tentando realizar o *commit* de suas modificações.

  Quando um conflito acontece, a decisão de qual transação (atacante ou defensor) ganhará é

geralmente relacionada com políticas de detecção de conflito e gerenciamento de versão. Para escolher qual transação abortar, existem diversas medidas, como o tamanho da transação, o número de *aborts* anteriores, tempo decorrido da transação, entre outros. Diversos mecanismos foram prosposto para gerenciamento de contenção como: Passive, Polite, Karma, Polka e Greedy.

No mecanismo Passive, também conhecido por *timid* (tímido), transações que detectam um conflito de escrita se auto abortam. No mecanismo Polite [67], uma transação que descobre um conflito adia o acesso ao dado por um período determinado. Se o limite de tempo for ultrapassado, a transação atacante aborta a defensora. Esse mecanismos possibilita que muitos conflitos serjam resolvidos sem precisar abortar transações. No mecanismo Karma [67], existe uma tentativa de favorecer transações que estão a mais tempo executando. O mecanismo armazena o número de objetos transacionais acessados pela transação e dá prioridade para a transação com o maior número. No mecanismo Greedy [54], é atribuido para cada transação um valor *timestamp* ao iniciar. A transação com o menor timestamp sempre ganha. Uma importante característica deste mecanismos é, ao contrário de outros mecanismos, ele evita *starvation*. O mecanismo Polka mostrou o melhor desempenho em *benchmarks* de pequena escala [67], enquanto que Greedy apresenta melhores resultados com *benchmarks* de larga escala [2]. Por fim, o mecanismo *timid*, utilizado nos sistemas TinySTM e TL2, apresenta desempenho baixo para transações grandes [2].

- **Mecanismo de sincronização (*synchronization mechanism*):** uma importante característica de sistemas transacionais refere-se ao mecanismo de sincronização dos sistemas. É possível classificar os sistemas STM de 2 tipos: *Obstruction-free* e *Lock-based*. O primeiro tipo não utiliza nenhum mecanismo de bloqueio (como *locks*), e garante o progresso mesmo quando algumas transações são atrasadas (Prosposto em [44]). Já *Lock-based* implementa o protocolo de bloqueio de duas fases [28] e foi inicialmente utilizado em [62]. Sistemas *lock-based* possuem melhor desempenho que implementações *obstruction-free* [25].

3. Estado da Arte

Este capítulo apresenta o estado da arte de Memórias Transacionais relacionadas com o assunto deste trabalho. É apresentado na Seção 3.1, os sistemas STM mais citados na literatura de Memórias Transacionais. A Seção 3.2 apresenta um conjunto de *benchmarks* pesquisados. Nessa seção ainda é apresentado um detalhamento das aplicações utilizadas nos testes dos capítulos 5, 6 e 7. Por fim, a Seção 3.4 encerra o capítulo descrevendo os sistemas STM e os *benchmarks* selecionados para os testes deste trabalho.

3.1 Sistemas STM

O primeiro trabalho de Memórias Transacionais com foco exclusivamente em *software* foi [48]. Desde então vários trabalhos apresentam novos sistemas baseados em *software* e novos mecanismos para minimizar o *overhead* gerado pelos sistemas STM. Atualmente é possível visualizar um grande avanço nos sistemas desde a primeira proposta e novos trabalhos e aprimoramentos estão surgindo na área. Nas seções seguintes são apresentados os principais sistemas STM da literatura.

3.1.1 SwissTM

SwissTM [2] é um sistema STM recente que foi desenvolvido a partir de diversos experimentos com as principais decisões de projeto STM. SwissTM é baseado em *locks* (*lock-based*) e em palavras (*word-based*) e possui 2 abordagens para detecção de conflito: deteção de conflito otimista (*commit-time*) para conflitos de leitura/escrita e detecção de conflito pessimista (*encounter-time*) para conflitos de escrita/escrita. Ao detectar conflitos escrita/escrita antecipadamente, o sistema evita perder tempo e processamento com transações que irão abortar. E detectando conflitos leitura/escrita tardiamente possibilita uma maior concorrência. Além disso, possui um gerenciador de contenção de duas fases que garante o progresso de transações longas enquanto não sobrecarrega transações curtas. O gerenciador de contenção muda dinamicamente da politica Polite [67], que favorece transações pequenas, para a política Greedy [54], que desempenha melhor com transações grandes. Por fim, SwissTM utiliza leituras invisíveis (*invisible reads*) e utiliza esquema baseado em tempo para validação do conjunto de leituras (time-based scheme) [16, 52]. O sistema foi desenvolvido para desempenhar satisfatoriamente com aplicações complexas com grandes cargas de trabalho mas também foi pensado para aplicações menores e com estruturas de dados simples. Esse fator é importante pois, muitas vezes aplicações complexas possuem tanto transações grandes com um grande volume de dados, mas também transaçoes curtas com acessos simples na memória. Desde sua criação em 2009 tem se aprimorado e vem obtendo resultados melhores ao longo dos anos, possuindo sua última versão de Agosto de 2011. SwissTM apresenta resultados satisfatórios em diversos trabalhos apresentados, como em [1, 2]. SwissTM é *open-source* e pode ser encontrado no endereço `http://lpd.epfl.ch/transactions/wiki/doku.php`.

3.1.2 RSTM

RSTM [66] foi desenvolvido na Universidade de Rochester em 2006. Nas primeiras versões o sistema era baseado em objeto (*object-based*) e livre de obstruções (*obstruction-free*). Além disso, utilizava heurística de contador de commits global (global commit counter heuristic) [64]. RSTM suportava leituras visíveis (*visible reads*) e invisíveis (*invisible reads*) e suportava detecção de conflitos ansiosa (*eager*) e preguiçosa (*lazy*) (4 variações de algoritmos). Por fim, RSTM suportava uma variedade de gerenciadores de contenção como Polka [67], Polite [67], Greedy [54], entre outros.

Atualmente o sistema RSTM suporta múltiplos algoritmos (aproximadamente 30 algoritmos baseados em palavra (*word-based*) e baseado em *locks* (*lock-based*). Cada algoritmo possui ainda, suas características próprias de detecção de conflito e gerenciamento de contenção além de mecanismos avançados específicos para cada objetivo. O algoritmo padrão, utilizado neste trabalho, é chamado NoRec (Ver [31]). O algoritmo possui versionamento preguiçoso (lazy-versioning) e aquisição também preguiçosa (*lazy-acquire*) e é baseado em palavra (word-based). NORec combina 3 idéias chave: (1) um *sequence-lock* global [30]; (2) um conjunto de escrita indexado [37]; e (3) detecção de conflito baseado em valor [37].

O sistema RSTM possui sua última versão de Julho de 2011. Desde a primeira versão até agora, o grupo criador do RSTM tem desenvolvido diversos algoritmos para incorporação do sistema. Além disso, novos mecanismos estão sendo propostos e utilizados em novos algoritmos do sistema. RSTM é *open-source* e pode ser encontrado no endereço `http://code.google.com/p/rstm/`.

3.1.3 TL2

TL2 [16] é um sistema STM baseado em *locks* (*lock-based*). TL2 possui uma variação do algoritmo de versionamento de relógio global (*global version-clock*) do algoritmo original (TL) proposto em [59]. Com essa variação, os autores solucionaram diversos problemas de segurança presentes em sistemas STM baseados em *locks* além de garantir que o código opere apenas em estados de memória consistentes. Utiliza esquema baseado em tempo para validação do conjunto de leituras (time-based scheme) [16, 52], aquisição/versionamento preguiçoso e leitura invisível (*invisble reads*). O gerenciador de contenção é tímido, sempre aborta transações atacantes (com um possível atraso). Um dos objetivos do TL2 é oferecer execução eficiente para transações somente leitura. Para isso algumas etapas são necessárias para reduzir o custo ao executar tais transações. Por outro lado, mais etapas são necessárias para executar transações com escrita, o que pode significar um custo alto no desempenho de aplicações.

O sistema TL2 possui sua última versão de Setembro de 2008. Esse sistema STM, é um dos mais citados na literatura dentre os apresentados nesse trabalho. TL2 é *open-source* e pode ser encontrado no endereço `http://stamp.stanford.edu/`.

3.1.4 TinySTM

TinySTM [52] é um sistema STM baseado em *locks* e em palavras (*word-based*) também conhecido na literatura. Utiliza versionamento global (contador compartilhado como relógio) para controlar conflitos entre transações e *locks* para proteger endereços de memória compartilhada. TinySTM utiliza aquisição/versionamento ansioso e leitura invisível (*invisble reads*). Assim como no TL2, TinySTM utiliza um vetor de *locks* compartilhados para gerenciar o acesso a memória. Cada *lock* cobre uma porção da memória e são mapeados em uma função *hash*. O gerenciador de contenção é tímido, assim como no sistema TL2.

O sistema TinySTM foi proposto em 2008 e possui sua última versão de Novembro de 2011. Entretanto, os testes realizados neste trabalho utilizam a versão de Setembro de 2008, visto que foram desenvolvidos anteriormente a liberação da versão mais atual. Esse sistema STM, é bastante citado na literatura. TinySTM é *open-source* e pode ser encontrado no endereço `http://tmware.org/tinystm`.

3.1.5 Intel STM

Intel STM [21] consiste em um compilador C/C++ e uma biblioteca STM em tempo de execução. O compilador interage com toda a memória compartilhada (leituras/escrita) dentro de transações através de barreiras. O escopo do compilador inclui linguagem de extensão para especificar e definir seções atômicas. A linguagem de extensão inclui diversas construções e palavras chaves para lidar com transações. A biblioteca STM em tempo de execução suporta a geração de estatísticas transacionais simples que pode ajudar o usuário a entender o desempenho das aplicações e encontrar gargalos. Algumas das informações são: número de vezes que uma transação executou, número de tentativas de execução de uma transação devido a conflito e quantidade de *bytes* de leitura e escrita transacionais.

A primeira versão do sistema Intel STM foi proposta em 2006 [6] com o nome de McRT-STM. Atualmente está no protótipo 4.0 (2009) e se chama Intel C++ STM *Compiler*. Esse compilador precisa de licença para uso e pode ser encontrado no endereço `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`.

3.2 *Benchmarks*

Muitos pesquisadores desenvolveram *benchmarks* e *microbenchmarks* para avaliação de sistemas STM. *Microbenchmarks* são usualmente bem estruturados e utilizam um conjunto pequeno e regular de transações [58]. São fáceis de desenvolver, parametrizados e com estrutura de dados simples. Além disso, são úteis para isolar casos particulares em sistemas STM e são fáceis de portar para outras linguagens/arquiteturas [9]. Transações pequenas e simples são importantes para testar a mecânica do sistema STM e comparar detalhes de baixo nível de diferentes implementações [2]. Entretanto, eles não representam a carga de trabalho de uma aplicação completa. Aplicações completas possuem

transações que constistem em muitas operações em grande volume de dados e ainda possuem muitas operações paralelas e até mesmo sequenciais entre as transações [9, 58]. Segundo [55], medir o desempenho de um STM em um ambiente excessivamente simplificado pode ser não informativo e, no pior caso, pode causar enganos, pois pode orientar pesquisadores a otimizar aspectos irrelevantes de suas implementações. Muitos *microbenchmarks* foram propostos [6,10,16,44,50,59] na literatura e amplamente utilizados para avaliação de sistemas STM.

Para avaliações completas de sistemas STM é necessário utilizar *benchmarks* que representem aplicações reais. As próximas seções apresentam os principais *benchmarks* propostos na literatura.

3.2.1 EigenBench

EigenBench [58] é um *microbenchmark* leve e poderoso para avaliação de sistemas de memórias transacionais. EigenBench fornece uma compreensão mais profunda do desempenho de TM através de exploração das características Eigen (características ortogonais que representam a base do entendimento de uma aplicação transacional), avaliando, assim, detalhes das aplicações que não são facilmente alcançáveis pelos demais *benchmarks* existentes. O *benchmark* baseia-se nessas características para avaliar aplicações de maneira isolada, ou seja, é possível isolar uma das características sem alterar as demais e assim medir o impacto da mudança através de testes de desempenho. As características Eigen são descritas com detalhes abaixo:

1. ***Concurrency*** **(Concorrência):** número de *threads* executando concorrentemente;

2. ***Working-set size*** **(Conjunto de trabalho):** tamanho da memória utilizada frequentemente;

3. ***Transaction lenght*** **(Comprimento da transação):** número de acessos compartilhados por transação. Por acesso compartilhado é definido operações de leitura/escrita que precisam ser protegidas por transações. Essa característica é diferente da habitual *transaction size* (tamanho da transação) que indica o tamanho total de uma transação;

4. ***Pollution*** **(Poluição):** fração das operações de escrita compartilhada pelo número de acessos compartilhados ($P = \frac{\text{escritas compartilhadas}}{\text{acessos compartilhados}}$). Essa característica mede o quão frequente ocorre escritas dentro de transações. Com ela é possível medir se uma aplicação tem mais escritas do que leituras em acessos compartilhados.

5. ***Temporal locality*** **(Localidade temporal):** probabilidade de existir endereços repetidos por acesso compartilhado. Essa caracteristica indica se as escritas/leituras em uma transação são em vários endereços na memória ou sempre em um mesmo endereço. Isso auxilia a identificar a complexidade dentro das transações;

6. ***Contention*** **(Contenção):** probabilidade de um transação conflitar com outra transação. Essa característica é bastante utilizada na literatura;

7. ***Predominance* (Predominância):** fração de ciclos compartilhados pelo total de ciclos da aplicação ($P = \frac{\text{ciclos compartilhados}}{\text{total de ciclos}}$). Por ciclo compartilhado é definido como um ciclo de execução que possui um acesso compartilhado. Essa característica indica a quantidade de ciclos não transacionais existentes na aplicação. Por exemplo, se uma aplicação possui 10 ciclos no total, sendo 8 desses ciclos compartilhados, temos a equação $P = \frac{8}{10} = 0.8$, o que indica 80% de ciclos transacionais na aplicação. Essa característica varia de 0.0 a 1.0, mas também pode ser expressada como baixa, média ou alta predominância;

8. ***Density* (Densidade):** fração de ciclos de acesso não compartilhado executados fora de transações pelo número total de ciclos não compartilhados ($D = \frac{\text{ciclos não compartilhados}}{\text{total de ciclos não compartilhados}}$). Essa característica mede a quantidade de ciclos de operações não transacionais dentro de transações. Por exemplo, se uma aplicação possui 10 ciclos não compartilhados sendo que, 2 desses são executados fora de transações, temos a equação $D = \frac{2}{10} = 0.2$, ou seja 80% das operações não transacionais são realizadas dentro de transações. Essa característica varia de 0.0 a 1.0, mas também pode ser expressada como baixa, média ou alta densidade;

Além disso, com o *benchmark* é possível simular aplicações reais através de um conjunto de parâmetros de entrada que derivadas definem as caracteristicas ortogonais da aplicação. Em [58], o *benchmark* é utilizado para simular o compartamento de aplicações do *benchmark* STAMP. Apesar de não ser o foco do *benchmark*, simular o comportamento de outras aplicações auxilia na comparação de resultados de desempenho. A Tabela 3.1 apresenta os principais parâmetros de entrada que são utilizados para simular o comportamento de aplicações transacionais.

Tabela 3.1: Principais parâmetros de entrada no *benchmark* EigenBench

Nome	Descição
N	Número de *threads*
A1	Tamanho do Array 1 (*Hot Array*)
A2	Tamanho do Array 2 (*Mild Array*)
A3	Tamanho do Array 3 (*Cold Array*)
R1	Leituras/transação no *Hot Array*
W1	Escritas/transação no *Hot Array*
R2	Leituras/transação no *Mild Array*
W2	Escritas/transação no *Mild Array*
R3o	Leituras/transação no *Cold Array* entre transações
W3o	Escritas/transação no *Cold Array* entre transações
R3i	Leituras/transação no *Cold Array* dentro de transações
W3i	Escritas/transação no *Cold Array* dentro de transações
LCT	Localidade temporal

O *benchmark* divide a memória em 3 *arrays* distintos:

- *Hot Array: array* compartilhado entre todas as *threads*. As *threads* acessam os dados concorrentemente;

- *Mild Array: array* acessado dentro de transações, porém, é divido para que cada *thread* acesse sua porção, ou seja, acessos simultâneos não causam conflitos;
- *Cold Array: array* é particionado como o *Mild Array*, porém é utilizado somente para acesso não transacional.

Através do conjunto de parâmetros apresentados na Tabela 3.1 é possível derivar as características Eigen. A Tabela 3.2 apresenta a derivação das características ortogonais.

Tabela 3.2: Derivação das características transacionais através do parâmetros de entrada

Característica	Parâmetros EigenBench
Concorrência	N
Comprimento de transação	R1+R2+W1+W2=(T_{len})
Localidade temporal	LCT
Conjunto de trabalho	A1+A2+A3=(Mem_{size})
Poluição	(W1+W2)/T_{len}
Predominância	Derivação não trivial
Contenção	Derivação não trivial
Densidade	Derivação não trivial

As três características não derivadas apresentadas na Tabela 3.2 são complexas e dependem de informações adicionais do sistema utilizado. Entretanto, o trabalho [58] apresenta pouca informação a respeito.

EigenBench é *open-source* e pode ser encontrado no endereço `http://ppl.stanford.edu/eigenbench/`.

3.2.2 LeeTM

Lee-TM [33] é um *benchmark* baseado no algoritmo de roteamento de circuitos Lee que oferece carga de trabalho grande e realísta. O algoritmo seleciona pares de pontos (por exemplo, de um circuito integrado) como entrada e produz rotas sem interseção entre eles. Enquanto as transações de Lee-TM são significativas em tamanho, elas apresentam padrões de acesso muito regular - cada transação primeiramente lê uma grande número de localidades (procurando por caminhos adequados) e após atualiza um pequeno número delas (criação de um caminho). Além disso, o *benchmark* usa objetos muito simples (cada um pode ser representado como uma váriavel inteira). É interessante notar que o STAMP contém um aplicativo (chamado Labyrinth) que utiliza o mesmo algoritmo de Lee. Entretanto, Lee-TM usa dados de entrada do mundo real o que o torna mais realísta do que o Labyrinth. Lee-TM inclui dois conjuntos de entrada: memória e circuitos de placas mãe.

A limitação deste *benchmark* é o padrão de acesso regular presente nas transações e o fato de ter foco específico em uma aplicação apenas, não exercitando assim, toda a amplitude dos comportamentos de um sistema STM. Lee-TM é *open-source* e pode ser encontrado no endereço `http://apt.cs.man.ac.uk/projects/TM/LeeBenchmark/`.

3.2.3 RMS-TM

RMS-TM [20] é uma suíte de aplicações para *benchmark* de Memórias Transacionais composto de 7 aplicações reais para mineração, reconhecimento e síntese (*Recognition, Mining and Synthesis* (RMS)). Além de incluir os principais problemas das pesquisas com TM como transações aninhadas, chamadas de sistema e operações de entrada e saída (I/O), as aplicações fornecem uma variedade de transações pequenas e grandes com conjuntos de leitura/escritas grandes e pequenos bem como níveis de contenção pequeno, médio e grande. A suíte ainda possui versões das aplicações com *locks*, o que possibilita a comparação de versões transacionais com versões utilizando mecanismos tradicionais de sincronização. Além das características citadas acima, o *benchmark* suporta sistemas STM e HTM e tem uma boa escalabilidade.

A limitação desta suíte é o tamanho das transações que não é grande comparado com outros *benchmarks* bem como o tempo em transações também não é alto. RMS-TM é *open-source* e pode ser encontrado no endereço `http://www.bscmsrc.eu/software/rms-tm`.

3.2.4 STAMP

STAMP [9] é uma suíte de aplicações para *benchmark* de sistemas STM. A suíte consiste em oito aplicações diferentes e dez cargas de trabalho. Aplicações STAMP representam cargas de trabalho do mundo real para diversas áreas como bioinformática, engenharia, computação gráfica e aprendizado de máquina. A Tabela 3.3 apresenta as aplicações do STAMP bem como sua área e uma breve descrição.

Tabela 3.3: Aplicações do *benchmark* STAMP

Aplicação	**Área**	**Descrição**
Bayes	aprendizado de máquina	Aprender estruturas de redes Bayesianas
Genome	bioinformática	Executar sequenciamento de genes
Intruder	segurança	Detectar intrusões de redes
Kmeans	mineração de dados	Implementar agrupamento K-means
Labyrinth	engenharia	Rotear caminhos em um labirinto
SSCA2	científica	Criar representação de grafos
Vacation	processamento online	Emular um sistema de reserva de viagens
YADA	científica	Implementar o refinamento de Delaunay

Para que seja possível realizar análises sobre os mais diversos sistemas de Memórias Transacionais, é necessário um *benchmark* com as seguintes características [9]:

- **Amplitude:** precisa utilizar aplicações e algoritmos que tenham benefício na utilização de Memórias Transacionais;

- **Dimensão:** precisa cobrir uma grande quantidade de variações no *design* dos sistemas TM, como tamanho de transações, tamanho do grão, políticas de decisão de *aborts*, etc;

- **Portabilidade:** precisa ser compatível com um grande número de sistemas TM, cobrindo *software*, *hardware* e soluções híbridas.

Com base nessas características foi criado o *benchmark* STAMP. O STAMP possui um conjunto de 8 aplicações com 30 diferentes configurações e dados de entrada que exploram uma variedade de comportamentos transacionais. O *design* deste *benchmark* é baseado nas características citadas, visto que baseou-se nelas para a seleção/implementação das aplicações:

- **Amplitude:** as aplicações foram escolhidas de acordo com sua dificuldade de paralelismo e buscando diferentes tipos de algoritmos em diferentes áreas do conhecimento. No conjunto de aplicações há 7 áreas diferentes, como por exemplo, bioinformática, engenharia e segurança;

- **Dimensão:** no conjunto de aplicações presente, existe uma ampla cobertura de comportamentos transacionais, tais como diversos graus de contenção, transações pequenas e grandes e diferentes tamanhos de conjuntos de leitura e escrita. Além disso, existem aplicações que utilizam transações de granularidade alta e utilizam uma grande parcela do tempo das aplicações dentro de transações;

- **Portabilidade:** no STAMP é possível utilizar sistemas TM de qualquer classe (*hardware*, *software* e híbrido), e o código é facilmente portável.

Para que se possa ter uma visão mais aprofundada das aplicações utilizadas, e que se possa analisar posteriormente os resultados obtidos nos sistemas TM, é necessário conhecer detalhadamente as aplicações do *benchmark*, bem como suas características transacionais. A seguir, serão apresentadas as aplicações, mostrando sua área do conhecimento, seu funcionamento e suas caraterísticas transacionais.

1. **Bayes:** aplicação que implementa um algoritmo para aprendizado de estruturas de redes bayesianas através de dados observados. A rede bayesiana é representada por um grafo acíclico dirigido, com um nodo para cada variável e uma aresta para as condições de dependências entre as variáveis. O algoritmo incrementalmente aprende as dependências analisando os dados observados. Transações são usadas para proteger o cálculo e a adição de uma nova dependência. Como o cálculo utiliza a maior parte do tempo de execução, a aplicação ocupa quase todo o tempo em transações grandes e o conjunto de leitura/escrita também é grande. Além disso, há uma quantidade grande de contenção, pois os subgrafos mudam frequentemente;

2. **Genome:** aplicação da área de bioinformática que visa reconstruir um genoma original através da comparação de diversos segmentos de DNA. O programa é dividido em duas fases. A primeira fase é responsável em criar um conjunto único de segmentos, visto que podem existir duplicatas no conjunto inicial. Na segunda fase do algoritmo, cada processo tenta remover um segmento de uma reserva global de segmentos não comparados e adicioná-lo à sua partição de segmentos combinadas atualmente. Transações são utilizadas em ambas as fases, ao adicionar

no conjunto único de segmentos para permitir acesso concorrente e também na reserva global de segmentos, pois *threads* poderão tentar remover o mesmo segmento. As transações e os conjuntos de escrita e leitura, nessa aplicação, possuem tamanho moderado. Adicionamente, quase todo o tempo de execução é transacional e a quantidade de contenção é pequena;

3. **Intruder:** sistema de detecção de Intrusão de Redes baseado em assinaturas, que varre pacotes comparando com um conjunto de assinaturas de intrusão já conhecidas. Essa aplicação, da área de Segurança de Redes, processa pacotes em paralelo e é composta de 3 fases: captura, remontagem e detecção. A fase de remontagem é a fase crítica da aplicação, na qual é necessário utilizar sincronização de granulidade alta, o que acarreta em perda de desempenho. Na versão TM dessa aplicação, as fases de captura e remontagem são encapsuladas em transações. A aplicação possui transações pequenas e seu nível de contenção oscila de médio a alto dependendo da frequência da fase de remontagem. Além disso, possui uma quantidade moderada de tempo de execução total em transações;

4. **Kmeans:** essa aplicação é utilizada para particionar dados em subconjuntos relacionados. Cada *thread* processa uma parcela de dados iterativamente e armezena em um *cluster* central. A versão TM adiciona uma transação para proteger a atualização do *cluster* central, que ocorre durante cada iteração. A quantidade de contenção entre as *threads* depende do valor de subconjuntos. Com diversos subconjuntos ocorrerão menos conflitos, visto que menos provavelmente os dados serão guardados no mesmo subconjunto. O tamanho da transação e dos conjuntos de leitura e escrita são pequenos nessa aplicação. Por fim, a maior parte do tempo é utilizada para calcular o subconjunto a ser utilizado, e nesse momento não há transação, pois os dados são locais de cada *thread*. Portanto, o tempo gasto em transações é relativamente pequeno;

5. **Labyrinth:** essa aplicação implementa uma variação do algoritmo de Lee [15]. A principal estrutura de dados é uma grade tridimensional uniforme que representa o labirinto. Na versão paralela, cada *thread* utiliza um ponto de início e de fim no qual ele deve se conectar por um caminho de pontos da grade adjacentes ao labirinto. O cálculo do caminho e sua adição à rede global do labirinto são encapsulados em uma única transação. No geral, quase todo tempo de execução do labirinto é tomado pelo cálculo do caminho, e a quantidade de operações de leitura e escrita é proporcional ao número total de pontos do labirinto. Devido a isso, a aplicação possui transações grandes e um conjunto grande de leitura e escrita. A quantidade de contenção é alta devido ao grande número de acessos transacionais a memória;

6. **SSCA2:** é uma aplicação composta por quatro núcleos que operam em um multigrafo dirigido, ponderado e grande. Esses quatro núcleos são utilizados em aplicações que vão desde biologia computacional até segurança. No STAMP, o foco é no primeiro núcleo, que é responsável por construir uma estrutura de dados em grafo usando *arrays* adjacentes e auxiliares. A versão transacional dessa aplicação possui *threads* adicionando nodos no grafo em paralelo

e utiliza transações para proteger o acesso aos *arrays* adjacentes. Visto que as operações são pequenas, pouco tempo de execução é gasto em transações. Além disso, o tamanho das transações e dos conjuntos de leitura e escrita são pequenos também. A quantidade de contenção é relativamente pequena pois a grande quantidade de nodos leva a infrequentes atualizações simultâneas no mesmo *array* adjacente;

7. **Vacation:** essa aplicação emula um sistema de reserva de viagens. Todo o sistema é implementado como um conjunto de árvores que mantém controle dos clientes e suas reservas para vários itens de viagem. Durante a execução, várias *threads* clientes executam um número de sessões que interagem com o banco de dados do sistema de viagens. Cada uma dessas sessões de clientes é encapsulada em uma transação para garantir a validade do banco de dados. Devido a isso, a aplicação gasta uma grande quantidade de tempo de execução em transações e suas transações possuem tamanho médio com conjuntos de leitura e escrita moderados. Níveis baixos a moderados de contenção entre os segmentos podem ser criados, aumentando a fração de sessões que modificam grandes porções do banco de dados;

8. **YADA:** a aplicação implementa o algoritmo de Ruppert para o refinamento de malhas de Delaunay [26]. As estruturas de dados básicas são um grafo que armazena todos os triângulos da malha, um conjunto que contém os segmentos de fronteira da malha, e uma fila de tarefas que contém os triângulos que precisam ser refinados. Em cada iteração, um triângulo compacto é removido da fila de trabalho, e uma reorganização dos triângulos é realizada sobre a malha, e quaisquer novos triângulos compactos resultantes são adicionados à fila de trabalho. O acesso a fila de trabalho é encapsulado por uma transação, assim como o refinamento do triângulo. Como quase todo o tempo de execução é gasto no cálculo de reorganização dos triângulos, a aplicação possui transações grandes e gasta quase todo o tempo em transações. Além disso, possui conjuntos grandes de leitura e escrita e uma quantidade moderada de contenção.

A limitação desse *benchmark* são o tamanho das transações, que são pequenas comparado com os demais *benchmarks*. STAMP é *open-source* e pode ser encontrado no endereço `http://stamp.stanford.edu/`

3.2.5 STMBench7

STMBench7 [55] é um *benchmark* sintético onde suas cargas de trabalho visam representar aplicações orientadas a objetos realistas e complexas que são alvos importantes para STM. Ele apresenta uma grande variedade de operações (de operações pequenas somente leitura até operações grandes que modificam grande quantidades de estrutura de dados) e carga de trabalho (de cargas contendo, principalmente, transações de somente leitura até transações com escrita dominante). A estrutura de dados utilizada pelo STMBench7 é diversas vezes maior do que outros típicos *benchmarks* STM. Além disso, suas transações são maiores e acessam um maior número de objetos. STMBench7 é inerentemente baseado em objeto e suas implementações utilizam a biblioteca padrão. Pequenas modificações são necessárias para utilizá-lo em STMs baseados em palavra (*word-based*).

Este *benchmark* é conhecido, na literatura, por testar os sistemas STM em busca de falhas. Visto que seu diferencial é sua grande estrutura de dados e suas transações longas, o *benchmark* é, geralmente, utilizado para testes de estresse. O trabalho [3] apresenta testes realizados com STMBench7 em conjunto com diferentes sistemas STM onde todos falharam antes de finalizar a execução.

Apesar de ser um *benchmark* mais robusto o mesmo é focado em um domínio de aplicação específico e não foi projetado para exercitar toda a amplitude dos comportamentos dos sistemas STM. STMBench7 é *open-source* e pode ser encontrado no endereço `http://lpd.epfl.ch/transactions/wiki/doku.php?id=stmbench7`.

3.3 Trabalhos Relacionados

Na literatura de Memórias Transacionais de *Software* é possível encontrar inúmeras publicações tanto com foco em sistemas e em novos algoritmos para solucionar os principais problemas da área, como também aplicações com foco em avaliar os mecanismos e as principais decisões de projeto dos sistemas atuais. Apesar disso, poucos trabalhos apresentam a visão geral da área, apresentando o contexto geral e focando no desempenho de sistemas e aplicações, que apresenta ser, de acordo com os trabalhos analisados, um dos principais problemas da área. Além disso, falta trabalhos com foco na avaliação dos principais sistemas STM presentes na literatura, assim como falta entender as principais caraterísticas de aplicações transacionais. Os trabalhos de avaliação existentes, geralmente, são voltados para testes de um *benchmark* ou sistema STM proposto e observações são realizadas com foco na proposta apresentada. Tais trabalhos, possuem a avaliação apenas para comprovação de desempenho e não como investigação do comportamento do sistema ou *benchmark*. Por conseguinte, ainda que existam proposta de novos sistemas e *benchmarks* que possuam avaliações de desempenho e comportamento, trabalhos com propostas similares a deste trabalho ainda são raros na literatura.

Entretanto, a comparação com os diferentes trabalhos propostos na literatura que possuam avaliações similares as apresentadas neste trabalho são de grande importância para substanciar os resultados deste trabalho. As seções seguintes apresentam trabalhos científicos que possuem relação com a proposta deste trabalho. Na Seção 3.3.1 são apresentados os trabalhos que exibem avaliações de *benchmarks* e/ou sistemas STM que possuem similaridade com os testes aqui apresentados. A Seção 3.3.2 apresenta trabalhos que utilizam mecanismos avançados de análise de execuções de aplicações STM.

3.3.1 *Benchmarks* e Avaliações de Sistemas

Diversas propostas de sistemas STM tem sido apresentadas na literatura, e para cada proposta é apresentado testes de desempenho utilizando apenas *benchmarks* ou comparativos com outros sistemas. Entretanto diversas prospostas de sistemas apresentam testes com *microbenchmarks* que não representam uma aplicação transacional realista, conforme descrito na Seção 3.2 deste capítulo.

Nesta seção são apresentados os trabalhos que possuem testes similares com os testes realizados neste trabalho e que, são citados ao longo do texto. Apesar dos trabalhos apresentarem objetivos diferentes do apresentado neste trabalho, a comparação dos resultados são de grande importância para reforçar e complementar as conclusões realizadas.

Cascaval *et al* [8], apresenta uma comparação de desempenho com os sistemas TL2 [16], Intel STM [6] e XL [53]. As aplicações utilizadas na comparação são Kmeans, Genome e Vacation do *benchmark* STAMP e outros dois *microbenchmarks*.

Minh *et al* [9], propôs o *benchmark* STAMP composto por 8 aplicações de diferentes domínios e com 10 cargas de trabalho. Neste trabalho é realizado uma comparação com o próprio *benchmark* e um simulador para HTM e STM.

Dragojević *et al* [2], propôs uma nova implementação STM chamada SwissTM. Neste trabalho é apresentado uma comparação utilizando os *benchmarks* STMBench7 [55], Lee-TM [33], STAMP [9] e um *microbenchmark*. Além disso, foram utilizados os sistemas TinySTM [52], TL2 [16] e RSTM [66].

Rui *et al* [18], apresenta uma comparação de desempenho com o *benchmark* STAMP. Foram utilizados para esse teste os sistemas TL2 e TinySTM.

Dragojević *et al* [1], apresenta uma comparação de desempenho recente utilizando o sistema SwissTM e os *benchmarks* STMBench7 [55] e STAMP [9] e um conjunto de *microbenchmarks*. Neste trabalho porém, o código do SwissTM foi adaptado para realizar testes com suporte a privatização e a um compilador STM.

Hong *et al* [58], propôs um novo *benchmark* que apresenta um novo conceito de características transacionais. O *benchmark*, através de parâmetros de entrada, simula o comportamento de aplicações do STAMP [9] e realiza uma comparação de desempenho das versões simuladas com as versões originais.

3.3.2 Técnicas de *Tracing* e *Profiling* em Aplicações STM

Conforme citado na Seção 1.1, o objetivo deste trabalho é, inicialmente, entender o comportamento de sistemas STM e suas aplicações e também, compreender os problemas atuais da área e identificar melhorias e oportunidades de crescimento nos sistemas STM de maneira a contribuir para a comunidade. Para tanto, pesquisadores estão buscando investigar os sistemas STM em camadas mais baixas do sistema, através de técnicas de *tracing* e *profiling*.

Ansari *et al.* [34], modificou uma implementação STM para coleta de informações relevantes durante a execução de aplicações. É apresentado um conjunto de 12 métricas para caracterizar aplicações TM, tais como *Speed-up* e ApC (*Aborts per Commit*), métricas utilizadas nesse trabalho. Foram selecionadas três aplicações do *benchmark* STAMP e o *benchmark* Lee-TM para investigar as métricas e compreender o impacto no sistema utilizado.

Lourenço *et al.* [24] propôs um *framework* de monitoramento com baixo *overhead*, desenvolvido especificamente para monitoramento de aplicações TM. O *framework* coleta eventos transacionais

e insere em um arquivo de *log*. Além disso foi desenvolvido uma ferramenta de visualização para apresentar os dados transacionais coletados.

Castro *et al.* [36], propôs uma abordagem para coleta de informações relevantes sobre transações baseado em uma biblioteca compartilhada do linux que monitora eventos das transações de uma aplicação e armazena em um arquivo de *log* para posterior análise. Foram monitorados eventos de *commits* e *aborts* e analisado os resultados de três sistemas TM.

Zyulkyarov *et al.* [19], propôs uma série de técnicas para traçar o perfil de aplicações TM, que fornecem informações sobre o tempo perdido causado por transações abortadas. O trabalho explora 3 direções distintas: (1) técnicas para identificar potenciais conflitos; (2) técnicas para identificar as estruturas de dados envolvidas nos conflitos; (3) técnicas de visualização para apresentar resumidamente como as *threads* utilizando seu tempo e quais transações conflitam mas frequentemente.

A proposta deste trabalho difere dos trabalhos acima pois foca em uma avaliação geral tanto de aplicações como de implementações STM sem a necessidade de alteração de código e abrangendo um conjunto abrangente de sistemas/aplicações. Essa abordagem tem por vantagem utilizar uma variedade maior de sistemas STM, ao contrário dos trabalhos [24, 34] que alteram o código de um único sistema STM para coleta de informações. Adicionalmente, a abordagem sugerida não acrescenta *overhead* nas execuções ao realizar a análise, em oposto aos trabalhos [19, 36] que aumentam o tempo de execução das aplicações devido ao acréscimo de operações adicionais para a coleta de dados. Por fim, é possível avaliar um maior número de sistemas STM e aplicações, visto que nenhum modificação é necessária, ao contrário dos trabalhos apresentados que utilizam poucas aplicações/sistemas para coleta de informações. Através dessa abordagem, é possível analisar o comportamento tanto das aplicações utlizadas, ao executar aplicações com diversos sistemas STM, bem como investigar os sistemas STM, ao comparar os resultados de aplicações entre os sistemas utilizados.

3.4 Resumo do Capítulo

Este capítulo apresentou o estado da arte de Memórias Transacionais com foco em *Software* (sistemas STM). Foram apresentados os principais sistemas STM propostos na literatura. A Tabela 3.4 resume os sistemas apresentados, a versão atual e a data da última atualização.

Tabela 3.4: Principais sistemas STM na literatura de Memórias Transacionais

STM	Versão Atual	Última atualização
Intel STM	4.0	2009
RSTM	v7	2011
SwissTM	2011-08-15	2011
TinySTM	1.0.3	2011
TL2	0.9.6	2008

Dentre os sistemas apresentados nas seções anteriores os selecionados para utilização neste

trabalho são:

- **SwissTM**: um sistema STM eficaz e com ótimos resultados na literatura [1]. Possui mecanismos avançados para reduzir o *overhead* e foi elaborado a partir de diversos experimentos com as principais decisões de projeto de sistemas STM. Além disso, possui atualização constante e a última versão é do ano corrente;

- **RSTM**: um sistema STM que possui diversas publicações e diversas citações em outros trabalhos na literatura. Foi criado em 2006, porém apresenta características e mecanismos atuais para redução do *overhead*, visto que é atualizado constantemente. Possui última versão do ano corrente;

- **TL2**: apesar de não possuir atualização recente, é um dos mais importantes sistemas STM proposto na literatura, possuindo diversas citações;

- **TinySTM**: sistema STM também muito citado na literatura. Apresenta características semelhantes ao TL2, não é atualizado frequentemente, porém tem versão final do ano corrente (não utilizada neste trabalho devido a diferenças de datas).

Com relação aos sistemas STM escolhidos, todos são baseados em biblioteca. O Intel STM Compiler, apesar de ser um STM atual conhecido na literatura, não foi selecionado pois não se encaixa com o padrão dos outros sistemas utilizados. Primeiramente, dentre os *benchmarks* utilizados ou não tem suporte para o sistemas baseados em compilador ou a adaptação é não trivial. Em segundo lugar, o sistema necessita licença para utilização, possuindo apenas período de avaliação de 30 dias. Apesar da possibilidade de avaliação, o foco do trabalho é utilizar sistemas STM disponíveis para a comunidade, onde o código fonte é disponibilizado para que melhorias sejam acrescentadas e utilizadas por toda a comunidade. Por fim, diversos autores questionam a utilização desse tipo de sistemas devido a quantidade de *overhead* adicionada [8, 35, 57].

Foram apresentados também, os principais *benchmarks* propostos na literatura. A Tabela 3.5 resume os *benchmarks* apresentados, a versão atual e a data da última atualização.

Tabela 3.5: Principais *benchmarks* na literatura de Memórias Transacionais

Benchmark	Versão Atual	Última atualização
EigenBench	0.8.0	2010
Lee-TM	1.0	2008
RMS-TM	1.0	2009
STAMP	0.9.10	2008
STMBench7	20110815	2011

Ao contrário de alguns sistemas STM apresentados, os *benchmarks* não possuem atualizações frequentes e muitos não foram atualizados desde sua criação. O *benchmark* STAMP, por exemplo,

foi atualizado pela última vez em Setembro de 2008 e continua sendo amplamente citado e, principalmente, utilizado pela comunidade científica. Por outro lado, atualizações são importantes para suportar novos sistemas STM, novas necessidades de avaliação e também para correção geral de *bugs*. Em relação ao suporte a sistemas STM, o *benchmark* RMS-TM não possui suporte a sistemas STM baseados em biblioteca. O *benchmark* STMBench7 não suporta a versão atual do sistema RSTM e as modificações para o suporte envolvem modificação de ambos os códigos fontes. O *benchmark* Lee-TM suportava os sistemas TinySTM, TL2 e RSTM, porém não possui atualização recente e apresentou problemas ao executar nos sistemas com versões atuais (incluindo SwissTM). Por fim, o *benchmark* EigenBench foi modificado para suportar o sistema RSTM e atualizado para suporte a última versão do sistema SwissTM.

Dentre os *benchmarks* apresentados, os selecionados para utilização neste trabalho são:

- **EigenBench**: apresenta inovação na avaliação de aplicações e sistemas STM através de características que representam a base do entendimento de uma aplicação transacional. É importante para entender o comportamento transacional atual das aplicações, bem como explorar as características do sistemas STM através de testes de características ortogonais isoladas;

- **STAMP**: *benchmark* completo que apresenta 8 aplicações de diferentes domínios. É amplamente utilizado na literatura. Apesar de não possuir transações tão grandes quanto outros *benchmarks* apresentados, o conjunto de oito aplicações com diferentes cargas de trabalho e grande volume de dados de entrada reduzem o impacto. Além disso, é extremamente adaptável a novos sistemas STM e possui versão sequencial das aplicações do *benchmark*.

A listagem acima apresenta os *benchmarks* que melhor se encaixam com o perfil e objetivo deste trabalho. Dentre os sistemas excluídos, alguns não possuem compatibilidade com os sistemas selecionados e outros não se encaixam no objetivo do trabalho. O RMS-TM, por exemplo, é um ótimo *benchmark* pois além de possuir 7 aplicações que abrangem diversos tamanhos de transação, tempos em transação e quantidade de contenção possui, ainda, características pouco exploradas como transações aninhadas, chamadas de sistemas e chamadas para outras bibliotecas dentro de transações. Além disso, o *benchmark* ainda possui versão com *locks* para todas as aplicações, o que é um diferencial para comparação entre as abordagens paralelas. Entretanto, apesar das vantagens, o *benchmark* não pode ser utilizado devido a incompatibilidade com sistemas baseados em biblioteca. Por outro lado, Lee-TM possui acesso regular a memória nas transações, o que não é necessariamente uma característica presente em aplicações reais. Além disso, o *benchmark* não possui suporte a novos sistemas STM e não possui uma abordagem simples para suportar modificações que possibilitem o suporte. Por fim, o *benchmark* STMBench7 apresenta caracaterísticas exigentes, visto que é o *benchmark* que mais estimula os sistemas STM. Entretanto, o mesmo tem foco em testes de estresse. Além disso, apresentou incompatibilidade com as últimas versões do sistema RSTM e não apresenta versão sequencial para comparação de resultados.

4. Metodologia

Conforme apresentado na Seção 1.1 a motivação deste trabalho é apresentar, de maneira clara, uma análise mais completa sobre Sistemas de Memórias Transacionais de *Software* bem como seus *benchmarks* presentes na literatura. Por análise completa, entende-se analisar o desempenho dos sistemas STM, através de execuções com diversas aplicações TM, de maneira crítica, tentando identificar o porque do comportamento das aplicações. Essa análise visa identificar caminhos para contribuir para a comunidade responder as perguntas propostas na Seção 1.1, que são de grande importância para o futuro dos sistemas de Memórias Transacionais.

Segundo [36], ainda é um desafio saber que tipo de aplicações pode realmente tirar proveito de sistemas TM. O trabalho recente [8] ainda questiona a capacidade de sistemas STM proporcionar bom desempenho devido aos resultados de baixo desempenho apresentados no trabalho. Conforme identificado por [8, 36] é fundamental investigar o motivo do baixo desempenho em algumas execuções com aplicações TM e ainda saber identificar previamente quais aplicações podem se beneficiar do paradigma.

Nos próximos capítulos são apresentados os testes executados, a metodologia utilizada em cada testes e os resultados obtidos. Na Seção 5 são apresentados os testes de desempenho com o *benchmark* STAMP, na Seção 6 são apresentados os testes de desempenho com o *benchmark* EigenBench e na Seção 7 são apresentados os testes específicos focados em características isoladas de aplicações transacionais.

Para a realização dos testes, é necessário identificar a arquitetura, geralmente utilizada, em testes de trabalhos relacionados na literatura. Além disso, devido a grande quantidade de testes realizados é importante utilizar algum mecanismos para validar as execuções e garantir confiabilidade dos resultados. Portanto, a Seção 4.1 apresenta um levantamento das arquiteturas utilizadas para testes de desempenho de sistemas STM na literatura, bem como a arquitetura utilizada neste trabalho e informações sobre o equipamento utilizado. Por fim, é apresentado, na Seção 4.2, as técnicas e fórmulas da estatística utilizadas para garantir a confiabilidade dos resultados gerados neste trabalho.

4.1 Arquitetura

Para a realização dos testes foram coletados os processadores utilizados nos diversos trabalhos estudados, buscando identificar o padrão de arquitetura que melhor se encaixe com o perfil de Memórias Transacionais. O foco desta área é arquiteturas *multicore*. Os trabalhos utilizam, em geral, computadores com dois ou mais *cores*. A Tabela 4.1 apresenta um resumo dos computadores utilizados para testes em diferentes trabalhos realizados com Memórias Transacionais. Os testes realizados nos trabalhos citados vão desde teste de sistema STM, *benchmarks*, testes de desempenho, entre outros.

De maneira geral, a quantidade de 8 *cores* é a configuração mais comum. Através dessa coleta

Tabela 4.1: Processadores

Autores	Descrição do Processador	Processadores	Total de *cores*
Cascavel *et al.* [8]	1 processador *quad-core* 2.3 GHz	1	4
Felber *et al.* [52]	1 processador *octa-core* 2.0 GHz	1	8
Hong *et al.* [58]	2 processadores *quad-core* 2.33 GHz	2	8
Castro *et al.* [35]	3 processadores *hexa-core* 2.66 GHz	3	18
Dragojević *et al.* [2]	4 processadores *dual-core* 2.4 GHz	4	8
Ansari *et al.* [33]	4 processadores *dual-core* 2.4 GHz	4	8
Kestor *et al.* [20]	4 processadores *dual-core* 3.2 GHz	4	8
Lourenço *et al.* [24]	8 processadores *dual-core* 2.8 GHz	8	16
Marathe *et al.* [66]	16 processadores 1.35 GHz	16	16
Dice *et al.* [16]	16 processadores 1.2 GHz	16	16

é possível determinar o computador com perfil semelhante para executar os testes prospostos nesse trabalho. A máquina selecionada para os testes é uma Dell PowerEdge R610 composto por 2 processadores Intel Xeon *quad-core* E5520 2.27 GHz *Hyper-Threading*, 16GB de memória e 150GB de HD. A máquina totaliza 8 *cores* físicos e mais 8 *cores* virtuais. A máquina selecionada se encaixa perfeitamente com o padrão utilizado nos demais trabalhos coletados. É importante ressaltar que apesar da máquina suportar *Hyper-Threading*, o recurso não foi utilizado, ou seja somente foram utilizados *cores* físicos. Portanto, os testes deste trabalho foram executados com 1, 2, 4, 6 e 8 *cores*.

4.2 Análise Estatística

Este trabalho utiliza métodos estatísticos para substanciar os resultados obtidos nos testes executados. Isso torna os resultados mas próximos da realidade e garante uma confiabilidade dos números apresentados. Ao executar testes de desempenho como os descritos neste trabalho o valor final para uma única execução pode variar significativamente. Ao executar o mesmo teste diversas vezes chegamos a um conjunto de resultados possíveis. A diferença entre os valores obtidos pode variar de acordo com a aplicação executada. Além disso, execuções podem variar de resultado devido a outros fatores, tais como:

- Variações do ambiente: qualquer interrupção ou comando adicional do sistema operacional poderá influenciar no resultado final.
- Valor *outlier*: um outlier (ou valor discrepante) é um valor que se localiza muito distante de quase todos os outros valores. No caso específico deste trabalho, dado um um conjunto de execuções onde o valor de uma execução é muito diferente dos demais resultados. Esse resultado diferente é um valor discrepante ou (*outlier*).

Na literatura da área de estatística, testes como os descritos nesse trabalho devem ser executados mais do que 30 vezes. Porém dependendo da variação dos valores nas execuções mais execuções

tornam-se necessárias [39].

Na seção seguinte serão utilizados termos específicos da área de estatística o que difere com os termos utilizados nas demais seções de testes deste capítulo. Para minimizar essa diferença na terminologia e auxiliar o entendimento da estatística nesse trabalho são listados abaixo os termos específicos da área e sua equivalência nesse trabalho.

- População: é o conjunto de dados utilizados nos cálculos. Nesse contexto, é infinito pois podemos realizar tantos testes quanto necessários, não há um limite de execuções;
- Amostra: é um membro da população, representa uma execução;
- Média amostral: é a média dos valores do conjunto de amostras. Nesse trabalho utiliza-se no mínimo 31 execuções (31 amostras). A média amostral é portanto a média das 31 execuções realizadas.
- Desvio padrão amostral: é o desvio padrão dos valores do conjunto de amostras. Nesse trabalho utiliza-se no mínimo 31 execuções (31 amostras). O desvio padrão amostral é portanto o desvio padrão das 31 execuções realizadas.
- População normalmente distribuída: é uma população onde os dados são próximos e que não possua nenhum valor *outlier* (valor fora do padrão dos demais valores).

Para complementar a utilização da estatística nesse trabalho a próxima seção apresenta mais detalhes do conceito de confiablidade, descrevendo as fórmulas e os conceitos necessários para os testes de desempenho.

4.2.1 Confiabilidade dos Resultados

Confiabilidade é o grau de confiança de um resultado, ou seja, indica o quão podemos confiar que um resultado está correto. O nível de confiabilidade é representado através de porcentagem. Na literatura, é comum utilizar um nível de confiabilidade de 90%, 95% e 99%, sendo 95% o mais comum [39]. Através de fórmulas estatísticas é possível estimar a quantidade ideal (ou mínima) de execuções dos testes realizados garantindo assim resultados mais confiáveis. De maneira geral, quanto mais execuções ocorrerem mais confiável fica o resultado final. A fórmula 4.1 é utilizada para determinar a quantidade de amostras necessárias (execuções) para garantirmos uma confiabilidade de 95%:

$$n = [\frac{z_{\alpha/2} \cdot \sigma}{E}]^2 \qquad (4.1)$$

onde,

$z_{\alpha/2}$ = escore z crítico com base no nível de confiança desejado
E = margem de erro desejada
σ = desvio padrão da população

n = quantidade de execuções necessárias

Para encontrar o z crítico é necessário consultar a Tabela 4.2 retirada de [39]. O valor *E* indica o erro máximo desejado. Esse valor deve ser atribuído conforme a variação desejada dos resultados, ou seja, um erro desejado de 1 segundo indica que 1 segundo de diferença entre os resultados é um valor aceitável. Por fim, o desvio padrão da população não pode ser obtido visto que o número de execuções tende a infinito. Para obter esse valor é necessário executar um teste piloto com 31 execuções (n $>$ 30) e coletar o desvio padrão desses resultados. Esse desvio padrão é chamado de amostral e pode substituir o desvio padrão da polução desde que n $>$ 30 e a população é normalmente distribuída. Esse desvio padrão é utilizado para calcular o verdadeiro valor de **n**, a quantidade de execuções.

Tabela 4.2: Níveis de confiança e Valor crítico

Nível de confiança	Valor crítico $z_{\alpha/2}$
90%	1.645
95%	1.96
99%	2.575

Neste trabalho os valores selecionados para cada variável descrita na fórmula 4.1 são:
$z_{\alpha/2}$ = 1.96 (Nível de confiança de 95%)
E = 0.5 segundos (Admite um erro máximo de meio segundo)
σ = valor variável (Depende da execução das aplicações)

Assim temos a seguinte fórmula pré-definida 4.2 para utilização no próximo capítulo onde será apresentado os testes. Conforme citado anteriormente é necessário realizar um teste piloto com 31 execuções para obtermos o desvio padrão amostral. Dependendo do resultado obtido na fórmula 4.1 mais execuções podem ser necessárias. É importante ressaltar que mesmo obtendo valores inferiores a 31 na fórmula 4.2 os testes utilizaram 31 execuções no mínimo.

$$n = [\frac{1.96 \times \sigma}{0.5}]^2 \tag{4.2}$$

Além de estimar as execuções ainda é possível calcular o intervalo de confiança de um resultado indicando o intervalo de variação daquele valor para mais ou para menos. Esse intervalo é importante pois indica uma faixa de valores onde o resultado final estará contido. Segundo [39], intervalo de confiança consiste em uma faixa (ou intervalo) de valores, em vez de apenas um único valor. Um intervalo de confiança está associado a um nível de confiança, tal como 95%. O nível de confiança nos dá a taxa de sucesso do procedimento usado para construir o intervalo de confiança.

Para calcular o intervalo de confiança é necessário selecionar uma distribuição. A distribuição é selecionada conforme a população (resultados) e ela é importante pois as fórmulas de cálculo mudam de acordo com a distribuição escolhida. A Tabela 4.3 apresenta as duas distribuições possíveis e os

critérios de escolha de cada uma.

Tabela 4.3: Distribuições de população

Distribuição	Critérios de escolha
normal (z)	σ conhecido e n $>$ 30
t	σ desconhecido e n $>$ 30

Como visto anteriormente todos os testes realizados nesse trabalho possuem n $>$ 30, portanto ambas as distribuições podem ser escolhidas conforme esse critério. Porém, não é possível termos σ (desvio padrão da população) conhecido visto que o número de execuções tende ao infinito. Portanto, a distribuição escolhida é a distribuição *t* (também conhecida como *t* de *Student*).

A fórmula 4.3 é utilizada para determinar o intervalo de confiança. Para determinar o intervalo de confiança é necessário calcular o erro encontrado na média dos resultados, ou seja, a diferença de valores dos resultados coletados. Para calcular o erro utiliza-se a fórmula 4.4.

$$\bar{X} - E < \mu < \bar{X} + E \tag{4.3}$$

onde,

$\bar{X}$ = média amostral

E = margem de erro

$$E = t_{\alpha/2} \cdot \frac{s}{\sqrt{n}} \tag{4.4}$$

onde,

$t_{\alpha/2}$ = é o valor crítico

s = desvio padrão amostral

n = número de amostras

Para calcular o valor crítico é necessário consultar a tabela da distribuição t, presente em [39]. Por exemplo, para um número de n = 31 e o nível de confiança = 95% chega-se a um $t_{\alpha/2} = 2.042$. No apêndice A são apresentados os valores de $t_{\alpha/2}$ para conjunto de execuções.

5. Teste de Desempenho com *Benchmark* STAMP

Este capítulo descreve os testes de desempenho com o *benchmark* STAMP. Na Seção 5.1 é apresentada a metodologia, descrevendo-se como o testes são executados e seus objetivos. Na Seção 5.2 são apresentados os resultados e uma análise crítica com observações sobre os testes executados. Por fim, na Seção 5.3 é apresentado um resumo com as considerações finais e as contribuições deste capítulo.

5.1 Metodologia

O teste com o *benchmark* STAMP tem o objetivo inicial de avaliar o desempenho dos principais sistemas STM propostos na literatura de maneira neutra, ou seja, sem apresentação de nenhuma prosposta de sistema ou algoritmo para sistemas STM. Apesar de diversos trabalhos na literatura realizar testes com o *benchmark* STAMP, não existe material com uma avaliação neutra dos principais sistemas STM da atualidade. Outro importante objetivo do teste é identificar comportamentos no desempenho das aplicações presentes no *benchmark*. Visto que, o STAMP é composto por diversas aplicações de domínios e comportamentos diferentes, o objetivo é analisar o desempenho dessas aplicações em conjunto com suas características transacionais. Tal análise tem como propósito identificar o comportamento das transações e seus motivos.

Para a realização dos testes, são selecionados os quatro sistemas STM mais relevantes propostos na literatura conforme seleção apresentada na Seção 3.4. A Tabela 5.1 resume os sistemas utilizados e apresenta a última versão de cada um e a data da última versão.

Tabela 5.1: Sistemas STM selecionados para testes com *benchmark* STAMP

STM	Versão	Data
TL2	0.9.6	2008
TinySTM	1.0.0	2009
RSTM	v7	2011
SwissTM	2011-08-15	2011

Os critérios de avaliação para seleção dos sistemas STM são sua relevância na literatura (quantidade de citações) e as últimas atualizações. Como exemplo, há o SwissTM, que tem sua última versão de Agosto de 2011, mas que não é o sistema mais citado dentre os selecionados. Em contrapartida, o TL2 não possui versão recente, porém, é citado em diversos trabalhos na literatura.

Conforme apresentado na Seção 3.2 o STAMP é composto por 8 aplicações diferentes e apresenta a mais completa variação de carga de trabalho e comportamentos transacionais entre os *benchmarks* atuais de STM. Além disso, é composto por aplicações de diferentes áreas como computação gráfica, bioinformática, segurança e mineração de dados, conforme apresentado na Tabela 3.3.

Para garantir a confiabilidade dos resultados gerados, todas as aplicações são executadas mais de 30 vezes. Na Seção 4.2.1 são apresentados os cálculos realizados que indicam a quantidade de execuções por aplicação do STAMP. Os testes foram executados em uma máquina *multicore* descrita na Seção 4.1.

Para realizar a análise dos resultados, a seguinte métrica é utilizada:

- ***Speed-up*:** o cálculo de *speed-up* mostra o quão bem os sistemas STM escalam com um determinado número de *core*. É calculado da seguinte forma: fração do tempo sequencial dividido pelo tempo paralelo ($Speed - up = \frac{\text{tempo sequencial}}{\text{tempo paralelo}}$). O resultado bom ou ruim é dependente do número de *cores* utilizados. Para uma execução com dois *cores*, um *speed-up* $= 2$ é o valor ideal; com quatro *cores* é esperado um *speed-up* $= 4$ e assim por diante. Isso significa que o tempo paralelo é 2/4 vezes mais rápido que o sequencial. Um *speed-up* alto depende não somente do sistema STM, mas também da aplicação transacional.

O STAMP é facilmente configurável para diversos sistemas de Memórias Transacionais. Por padrão, o sistema utilizado é o TL2, visto que a versão x86 da aplicação foi portada pelo próprio grupo do STAMP. Para executar as aplicações utilizando outra implementação de sistema TM, é necessário modificar o arquivo `common/Defines.common.mk`, alterando o caminho da chave STM para a localização da biblioteca a ser utilizada.

As aplicações presentes no STAMP possuem diversos parâmetros a serem utilizados de modo a explorar o paralelismo dos sistemas STM, além de possibitar diversos tipos de execuções. Nos testes realizados, as aplicações são executadas conforme configuração padrão apresentada em [9].

Conforme a listagem de parâmetros acima, todos as aplicações executaram com os maiores tamanhos possíveis de entrada [9], com exceção da aplicação Bayes, que apresentou problemas na execução com a biblioteca TL2 e rodou em uma versão intermediária. Além dos parâmetros citados, todas as aplicações recebem como parâmetro o número de *threads* na qual o código deverá executar.

Para todas as aplicações são executadas a versão sequencial, presente na distribuição do STAMP (1 *thread*), e até 4 versões STM: 2, 4, 6 e 8 *threads*. Os sistemas SwissTM e RSTM são executados com 2, 4, 6 e 8 *threads*, enquanto os sistemas TL2 e TinySTM são executados com 2, 4 e 8 *threads*. O *benchmark* STAMP possui uma limitação que impossibilita a execução com 6 *threads* nos sistemas TL2 e TinySTM. A seção seguinte apresenta os resultados desse teste bem como a análise dos resultados obtidos.

5.2 Resultados

Essa seção apresenta os resultados para os testes de desempenho com o *benchmark* STAMP. As Figuras 5.1, 5.2 e 5.3 exibem os resultados do teste. A Figura 5.1 apresenta os resultados com ênfase nos sistemas STM. Nela é possível ver o desempenho geral dos sistemas com todas as aplicações do *benchmark*. As Figuras 5.2 e 5.3 apresentam os resultados com ênfase nas aplicações, mostrando um gráfico por aplicação. Os dados são apresentados de duas formas diferentes para

facilitar a interpretação dos resultados dependendo do objetivo. A Figura 5.1 tem por objetivo a análise de desempenho dos sistemas testados, enquanto que as Figuras 5.2 e 5.3 têm ênfase no comportamento das aplicações.

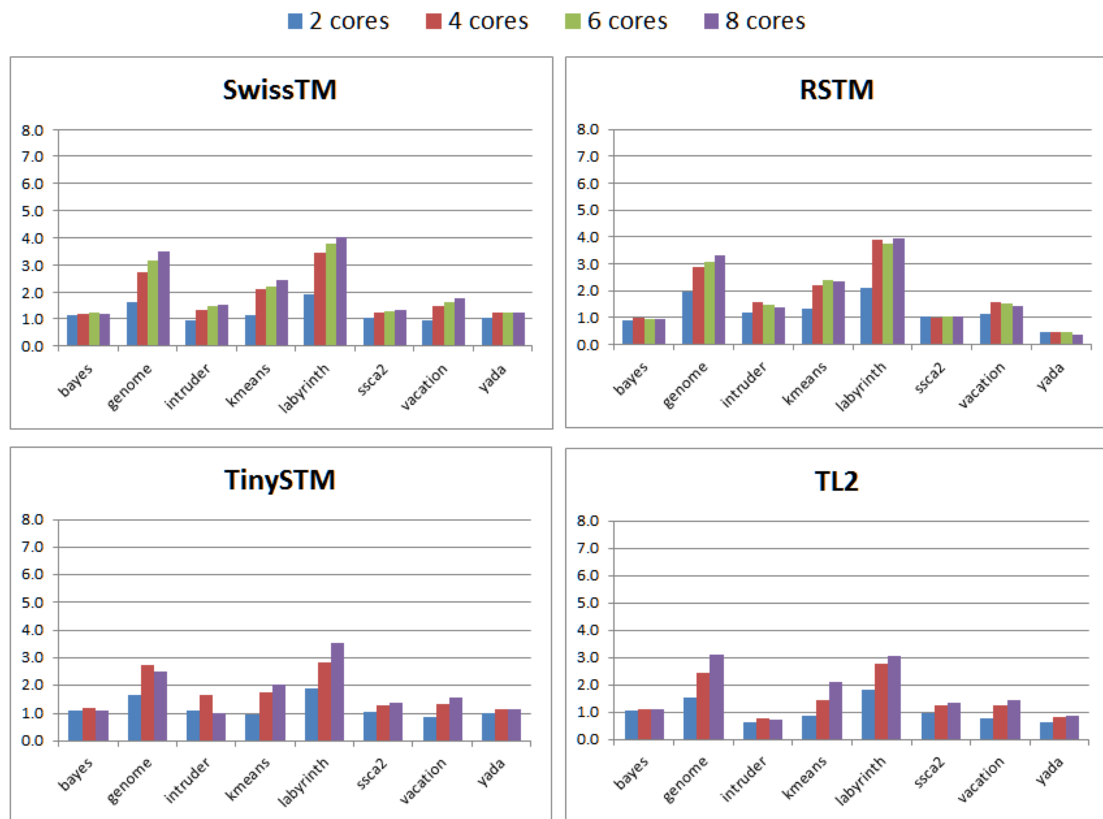

Figura 5.1: Desempenho (*speed-up*) dos sistemas STM com o *benchmark* STAMP

Em relação à comparação do desempenho geral dos sistemas testados (Figura 5.1), SwissTM e RSTM apresentam desempenho superior em relação aos outros dois. Essa constatação era esperada devido as atualizações frequentes das ferramentas, visto que ambas possuem última versão finalizada em 2011. As atualizações frequentes e recentes demonstram o esforço para se alcançarem melhores resultados, o que é comprovado na Figura 5.1. Entre os dois sistemas, o SwissTM possui vantagens em diversas configurações de *cores* e aplicações. O sistema que apresenta o pior desempenho nos testes é o sistema TL2. Além de não possuir atualizações recentes desde sua criação (2006), o sistema utiliza detecção de conflito preguiçosa, que é comumente conhecida na literatura como tendo o pior desempenho entre esse tipo de mecanismo [2].

Na análise específica do desempenho dos sistemas, é possível observar problemas ao escalar com seis e oito *cores*. Os resultados apresentados indicam que os sistemas, independentemente daquele utilizado, em geral, ainda possuem dificuldades em escalar acima de quatro *cores*. Adicionalmente, todos os sistemas apresentam problemas ao executar aplicações como SSCA2, Yada e Bayes, visto que elas apresentam resultados muito ruins e sem nenhuma escalabilidade. Por fim, aplicações como Intruder, Vacation e Kmeans apesar de terem resultados superiores às aplicações citadas, não

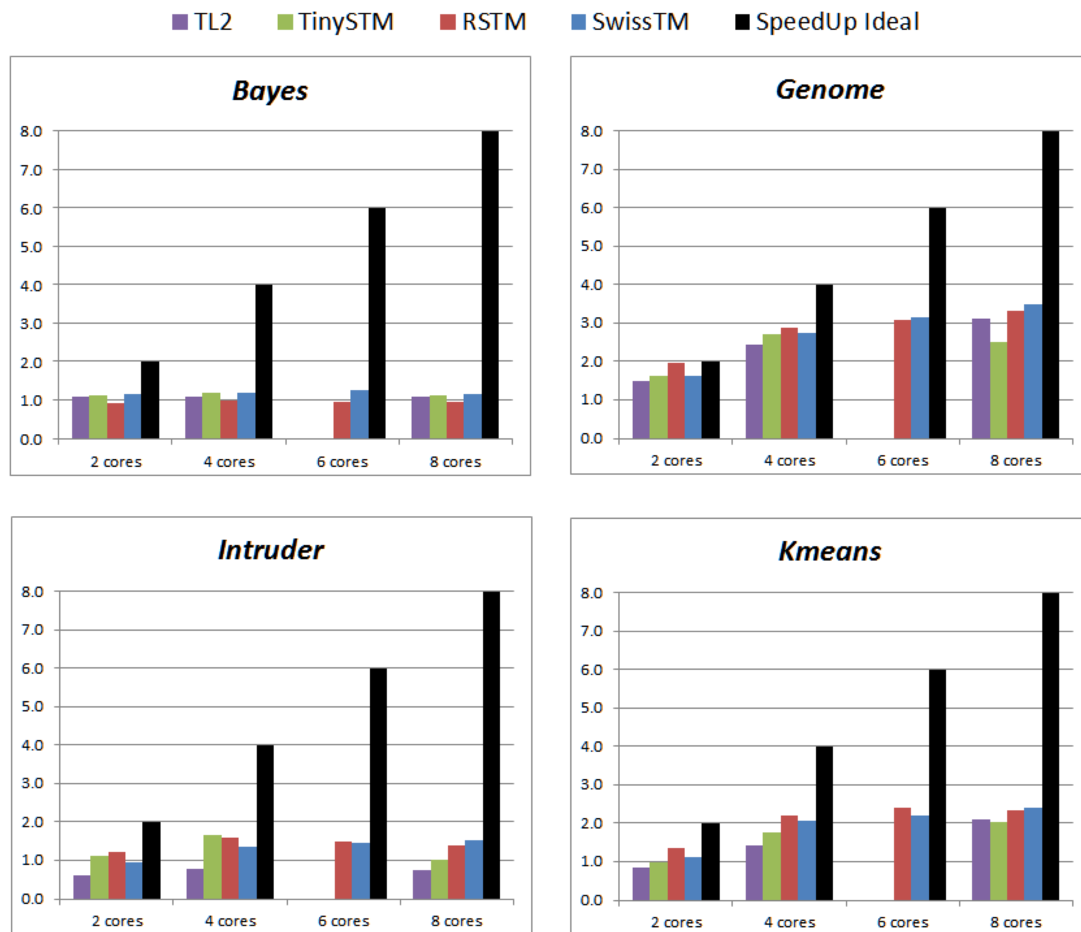

Figura 5.2: Desempenho (*speed-up*) das aplicações do STAMP com os sistemas STM (Parte 1)

apresentam resultados muito melhores e apresentam *speed-ups* insatisfatórios. É preciso investigar as características transacionais de tais aplicações em busca de motivos para o baixo desempenho. Uma importante observação, que justifica em parte o baixo desempenho geral dos sistemas STM, é o fato de algumas aplicações não serem totalmente transacionais. Isso significa que, é possível, identificar partes de código fonte não transacionais, ou seja, as aplicações não são 100% preparadas para serem executadas de maneira transacional e por isso, não são esperados valores de *speed-up* próximos do ideal. Contudo, mesmo possuindo partes não transacionais, os resultados apresentados são insatisfatórios, e ainda é necessário investigar as características transacionais de tais aplicações em busca de motivos para o baixo desempenho.

Dentre as oito aplicações, apenas duas atingiram desempenho bom ao ideal em alguma configuração de *cores*: Genome e Labyrinth. A aplicação Labyrinth apresenta ótimo desempenho com dois *cores* em todos os sistemas. Já com quatro *cores*, apenas RSTM e SwissTM mantiveram desempenho bom. Já a aplicações Genome alcançou desempenho com dois *cores* ideal com RSTM e muito próximo com os outros sistemas. Acima de quatro *cores*, os sistemas SwissTM e RSTM apresentam melhor desempenho geral.

Na análise das Figuras 5.2 e 5.3 é possível notar um padrão de comportamento nas aplicações

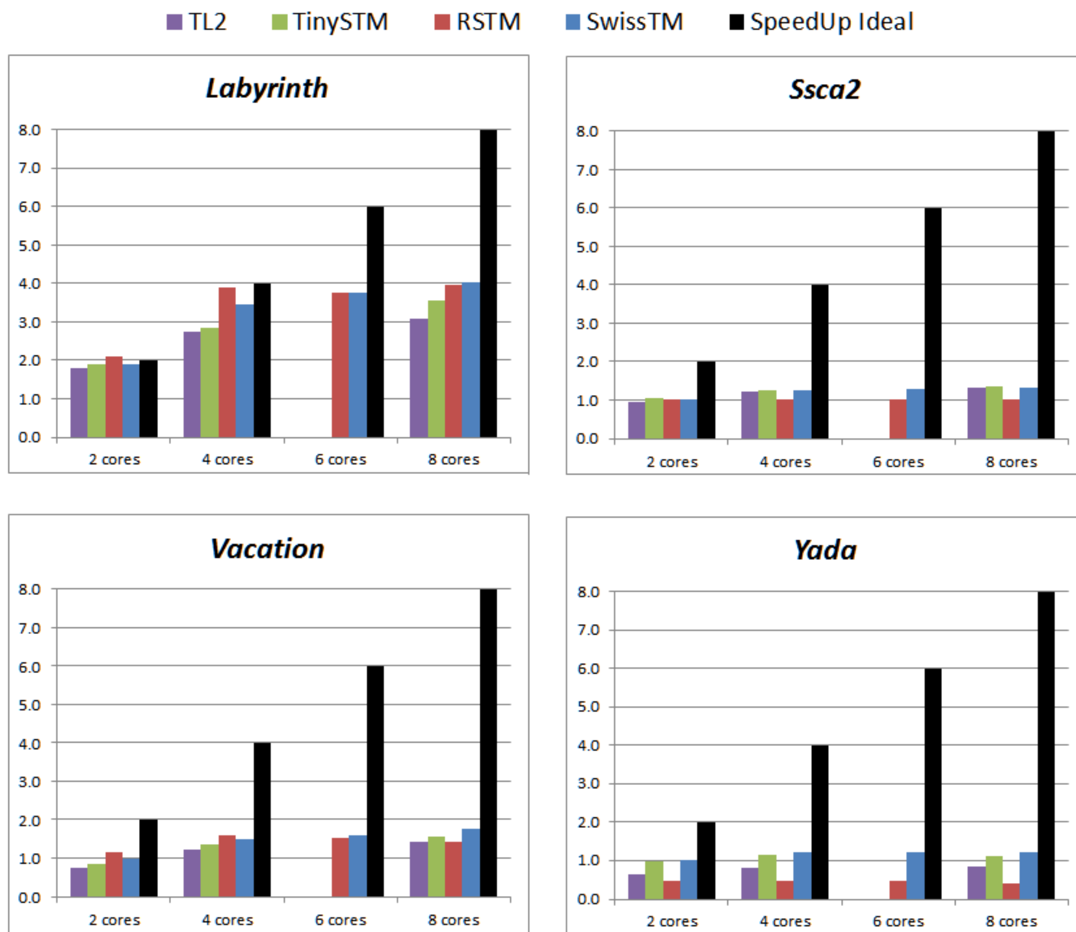

Figura 5.3: Desempenho (*speed-up*) das aplicações do STAMP com os sistemas STM (Parte 2)

com o conjunto de sistemas STM utilizados. Como exemplo, há as aplicações Bayes, SSCA2 e YADA que apresenta desempenho inferior em todos os sistemas. O mesmo acontece com aplicações com desempenho superior, que é o caso da aplicação Labyrinth, que apresenta resultado ideal com dois *cores* em todos os sistemas, e a Genome, que apresenta resultado bom com dois *cores*. Por padrão de comportamento entende-se que as aplicações tiveram resultados semelhantes entre os sistemas STM, apesar de existir diferenças de valores. Esse padrão pode ser observado em todas as aplicações e fica mais evidente na Tabela 5.2. A tabela mostra os resultados de apenas dois sistemas, apesar do padrão encontrado existir em todas os sistemas. A tabela apresenta somente 2 sistemas pois facilita a interpretação da tabela e também por ser os sistemas que obtiveram o melhor desempenho geral.

Além do padrão individual entre os sistemas, é possível, ainda, identificar um comportamento semelhante entre aplicações diferentes. Por exemplo, as aplicações Bayes, SSCA2 e YADA apresentaram resultados ruins em todos os sistemas STM: elas não ultrapassaram *speed-up* 1.4, independentemente da quantidade de *cores* utilizados. Foram identificados 3 grupos de aplicações, cada grupo com um comportamento distinto dos demais, nos resultados apresentados nas Figuras 5.2 e 5.3:

Tabela 5.2: *Speed-up* das aplicações STAMP com os sistemas SwissTM e RSTM

Aplicação	Sistema	2 *cores*	4 *cores*	6 *cores*	8 *cores*
Bayes	SwissTM	1.1546	1.1999	1.2408	1.1677
	RSTM	0.9147	0.9829	0.9331	0.9621
Genome	SwissTM	1.6229	2.7377	3.1434	3.4752
	RSTM	1.9564	2.8777	3.0914	3.3133
Intruder	SwissTM	0.9540	1.3480	1.4528	1.5261
	RSTM	1.2144	1.5832	1.4983	1.3848
Kmeans	SwissTM	1.1151	2.0802	2.2022	2.4123
	RSTM	1.3463	2.1996	2.3901	2.3530
Labyrinth	SwissTM	1.8994	3.4493	3.7695	4.0377
	RSTM	2.1018	3.8842	3.7640	3.9515
SSCA2	SwissTM	1.0292	1.2567	1.2847	1.3185
	RSTM	1.0301	1.0144	1.0305	1.0279
Vacation	SwissTM	0.9704	1.4783	1.6001	1.7645
	RSTM	1.1582	1.5862	1.5359	1.4387
YADA	SwissTM	1.0221	1.2130	1.2363	1.2277
	RSTM	0.4789	0.4796	0.4723	0.3932

1. **Aplicações com resultados ruins e sem escalabilidade:** esse grupo apresenta aplicações que tiveram resultado ruim em todos os sistemas STM e ainda não tiveram escalabilidade (ou escalabilidade mínima) com o aumento de *cores*;

2. **Aplicações com resultados baixos mas com escalabilidade:** esse grupo apresenta aplicações com desempenho um pouco superior ao grupo anterior e com um certo nível de escalabilidade.

3. **Aplicações com desempenho bom:** aplicações que tiveram bom desempenho no geral, mas não necessariamente em todos os *cores*, que é caso das aplicações Labyrinth e Genome.

De maneira a clarificar os conceitos de bom e ruim utilizados neste trabalho, a Tabela 5.3 apresenta os critérios utilizados para caracterizar o desempenho das aplicações. A tabela apresenta os níveis de desempenho, o valor mínimo do nível para cada configuração de *cores* e o critério do nível. Por exemplo, para considerar um resultado como 'Bom' em uma execução com dois *cores*, o *speed-up* deve estar acima de 1.6, ou seja, esse valor indica um desempenho superior a 80% do ideal (2.0 nesse caso). É importante ressaltar que um resultado abaixo do nível 'Baixo' indicam resultado ruim.

Para realizar um análise mais detalhada dos resultados dos testes é necessário entender o funcionamento e as características das aplicações envolvidas. A Seção 3.2.4 detalha cada aplicação do STAMP, enquanto a Tabela 5.4 resume suas principais características transacionais. A tabela

Tabela 5.3: Critérios de desempenho (speed-up)

Nível	**Valor mínimo**				**Critério**
	2 cores	**4 cores**	**6 cores**	**8 cores**	
Ideal	2.0	4.0	6.0	8.0	100%
Bom	1.6	3.2	4.8	6.4	acima de 80%
Razoável	1.2	2.4	3.6	4.8	acima de 60%
Baixo	1.0	2.0	3.0	4.0	acima de 50%

apresenta o tamanho da transação, o conjunto de leitura e escrita, o tempo gasto em transações e o nível de contenção de cada aplicação.

Tabela 5.4: Resumo das características das aplicações STAMP

Aplicação	Tamanho da transação	Conjunto L/E	Tempo em transações	Nível de contenção
Bayes	Grande	Grande	Alto	Alto
Genome	Médio	Médio	Alto	Baixo
Intruder	Pequeno	Médio	Médio	Alto
Kmeans	Pequeno	Pequeno	Baixo	Baixo
Labyrinth	Grande	Grande	Alto	Alto
SSCA2	Pequeno	Pequeno	Baixo	Baixo
Vacation	Médio	Médio	Alto	Baixo/Médio
YADA	Grande	Grande	Alto	Médio

Ao comparar as características transacionais presentes na Tabela 5.4 e os resultados apresentados nas Figuras 5.2 e 5.3, é possível identificar variações entre resultados e características. Pode-se observar que a aplicação Labyrinth possui características transacionais rigorosas (severidade alta) e, mesmo assim, apresentou os melhores resultados do teste, diferente da aplicação SSCA2 que possui severidade baixa e apresentou um desempenho muito abaixo do esperado. A Tabela 5.5 resume essa observação, apresentando as aplicações separadas em grupos de acordo com o comportamento, e indica a severidade das características transacionais conforme Tabela 5.4.

Tabela 5.5: Classificação das aplicações do *benchmark* STAMP conforme desempenho

Grupo	**Aplicação**	**Severidade das características**	**Descrição do grupo**
1	Bayes SSCA2 YADA	Alta Baixa Alta	Aplicações com resultados ruins e sem escalabilidade
2	Intruder Kmeans Vacation	Média Baixa Média	Aplicações com resultados baixos mas com escalabilidade
3	Genome Labyrinth	Média Alta	Aplicações com resultados bons

A partir da Tabela 5.5 conclui-se que as características apresentadas não são suficientes para investigar o desempenho das aplicações do STAMP e, portanto, um teste mais específico é necessário para entender o fluxo das aplicações. Além disso, é necessário identificar características transacionais mais específicas. De acordo com a Tabela 5.4, não é possível identificar se as aplicações possuem ou não acesso não transacional dentro das transações, como também não é possível saber se as transações de uma aplicação possui mais leituras do que escritas. Essas informações, bem como outras, são de grande importância para entender o funcionamento de uma aplicação transacional e a partir disso, comparar seu resultado com suas características.

Por fim, observa-se discrepância entre as características apresentas na Tabela 5.4 e os resultados obtidos. Nesse viés, as características apresentadas não são suficientes para compreender o desempenho de cada aplicação do STAMP. Para investigar novas características e analisar mais profundamente as aplicações, o teste do próximo capítulo utiliza uma nova maneira de caracterizar aplicações transacionais. As algumas das aplicações presentes neste teste serão simuladas para um novo teste de desempenho.

Confiabilidade

Conforme citado na Seção 4.2.1 este trabalho utiliza métodos estatísticos para determinar a quantidade de execuções necessárias para cada conjunto de aplicação e sistema STM e ainda para determinar o intervalo de confiança dos resultados. No Apêndice A.1 são apresentados os resultados de todas as execuções deste teste. As Tabelas A.1, A.2, A.3, A.4, A.5 exibem o número de execuções de cada aplicação enquanto as Tabelas A.6, A.7, A.8, A.9, A.10 exibem o intervalo de confiança de cada resultado.

5.3 Resumo do Capítulo

Conforme observado na Seção 5.2, as aplicações do *benchmark* STAMP apresentam desempenho baixo em todos os sistemas utilizados, com exceção da aplicação Labyrinth. Testes assim foram executados também em [1, 2, 8, 18, 58]. Em [8], foram obtidos resultados piores em relação aos apresentados neste trabalho. Na comparação dos sistemas utilizados, apenas o sistema TL2 está presente em ambos as pesquisas. Neste trabalho, o sistema TL2 apresentou resultados insatisfatórios, tais como os apresentados em [8]. Além disso, aqui foram utilizados sistemas STM mais modernos e criados/atualizados recentemente. Em [2], o autor não apresenta resultados em valores numéricos, mas uma comparação percentual de sistemas. É realizada uma comparação entre os sistemas SwissTM e TinySTM e entre os sistemas SwissTM e TL2. Em ambas as comparações o sistema SwissTM apresenta desempenho de 10 a 50% maior que os outros sistemas. Embora, nesse teste, o sistema SwissTM também possua desempenho superior a ambos, não é possível comparar os resultados, pois o trabalho não apresenta os parâmetros de execução utilizados nos testes. O trabalho [18] apresenta foco similar ao deste capítulo, propondo uma avaliação de sistemas STM com *benchmark* STAMP. Entretanto, não utiliza os principais sistemas STM apresentados aqui:

SwissTM e RSTM. No trabalho é possível identificar o mesmo comportamento encontrado nos testes. As características transacionais são incompatíveis com os resultados obtidos.

O trabalho [1] apresenta resultados semelhantes em algumas aplicações como SSCA2 e YADA. Entretanto, demais aplicações apresentam resultados muito superiores aos apresentados aqui. Com exceção da aplicação Bayes, que foi executada com parâmetros de entrada de tamanho médio, todas as demais aplicações apresentam os mesmos parâmetros de entrada em ambos os trabalhos. Uma possível diferença entre os testes são as modificações realizadas no sistema STM do trabalho, que possui diversas versões nas execuções do teste. Entretanto, a diferença de desempenho é muito significativa e não apresenta apenas como uma modificação de código. Por fim, o trabalho [58] também executa testes com aplicações do *benchmark* STAMP e apresenta resultados diferentes na maioria das aplicações de ambos os trabalhos, embora mais próximos dos resultados obtidos nos testes deste trabalho.

Ignorando as diferenças de resultados entre os trabalhos e assumindo os valores aqui mencionados como aceitáveis, pode-se concluir que o *benchmark* utilizado apresenta um conjunto de aplicações que exercitam de fato os sistemas STM. Através de inúmeras variações de cargas de trabalho e parâmetros, o *benchmark* é importante para avaliar tais sistemas visto que, ainda não é encontrado um sistema STM capaz de obter resultados satisfatórios em todas as aplicações. Isso indica o quão próximo o *benchmark* alcança aplicações reais. A investigação de aplicações, tal como a proposta deste trabalho, é importante para entender melhor seu desempenho e identificar oportunidades de melhoria nos sistemas STM.

Como observado na Seção 5.2, não são esperados resultados próximos do ideal devido ao fato de algumas aplicações não possuirem código 100% transacional. Entretanto, apesar dessa constatação, espera-se melhores resultados de sistemas STM com o *benchmark* STAMP no futuro devido a seguintes considerações:

1. Nenhum sistema STM obteve resultado satisfatório ao executar acima de 4 *cores*. Isso indica não ser um problema específico de um sistema, visto que todos os sistemas tiveram esse problema;

2. Aplicações não apresentam características transacionais tão exigentes (como tamanho de transação, por exemplo) comparados a outros *benchmarks* da literatura como STMBench7 [55] e Lee-TM [33];

3. Algumas aplicações, como Bayes, SSCA2 e YADA obtiveram resultados ruins e sem nenhuma escalabilidade. Apesar de não ser possível investigar o comportamento das aplicações mais profundamente, o resultado dessas aplicações indica deficiência nos sistemas STM.

Por fim, observa-se discrepância entre as características transacionais apresentadas pelo *benchmark* e os resultados obtidos. Além disso, as características transacionais são apresentadas como adjetivos, tais como: Grande, Pequeno, Médio e Alto. Esses adjetivos são difíceis de mensurar e

podem ser subjetivos. Por exemplo, algumas aplicações do *benchmark* STAMP apresentam transações grandes, porém, as mesmas são consideravelmente menores do que as utilizadas no *benchmark* STMBench7 [55].

Nessa perspectiva, é necessário identificar características transacionais que expliquem melhor as aplicações transacionais. Tais características precisam representar a aplicação de maneira mais completa possível, indicando atributos de aplicações que possam influenciar na compreensão do desempenho.

A principal contribuição deste capítulo é a avaliação imparcial dos principais sistemas STM propostos na literatura, avaliação ainda não realizada na área. Foi possível, através de testes com o principal *benchmark* para sistemas STM, identificar a superioridade da ferramenta SwissTM que, apesar de ser criada recentemente possui um conjunto de decisões de projeto eficaz. Esse conjunto foi amplamente pesquisado e explorado no trabalho [2]. Outra contribuição importante deste capítulo é sobre o *benchmark*, que exercita, de fato, os sistemas STM apesar de caracterizar as aplicações de maneira substancial. Por fim, outra contribuição é a constatação apresentada sobre os sistemas STM, que indica problemas ao utilizar muitos *cores* e não têm um desempenho satisfatório com determinados tipos de aplicações.

6. Teste de Desempenho com *Benchmark* EigenBench

Este capítulo apresenta os testes de desempenho com o *benchmark* EigenBench. Na seção 6.1, é apresentada a metodologia, descrevendo como o testes são executados e seus objetivos. Na Seção 6.2, são apresentados os resultados e uma análise crítica com observações sobre os testes executados. Por fim, na Seção 6.3, é apresentado um resumo com as considerações finais e as contribuições deste capítulo.

6.1 Metodologia

O teste de desempenho do *benchmark* EigenBench visa complementar o teste da Seção 5 através de uma análise também focada no desempenho dos sistemas STM e, principalmente, nas características transacionais de cada aplicação. No teste, é utilizado o *benchmark* proposto recentemente [58], que apresenta uma nova maneira de classificar as características transacionais de aplicações TM (ver mais na Seção 3.2.1). Além disso, com o *benchmark* é possível simular o comportamento de outras aplicações reais, como as utilizadas no *benchmark* STAMP. O objetivo principal deste teste é, portanto, utilizar as mesmas aplicações do teste anterior, porém, expandindo-se a definição das características transacionais de cada uma para realizar uma análise mais completa do desempenho. Apesar de não serem exatamente as mesmas aplicações, visto que são *benchmarks* diferentes, ao simular as aplicações do *benchmark* STAMP, obtêm-se o mesmo comportamento transacional das aplicações reais, conforme apresentado no trabalho [58].

As caracteristicas ortogonais propostas pelo *benchmark* são de grande importância para analisar o comportamento das aplicações e seu desempenho. Segundo [58], essas características ortogonais são a base do entendimento de uma aplicação transacional.

No teste em questão, são selecionados dois dos sistemas STM executados nos testes do capítulo anterior: SwissTM e RSTM. O critério de seleção é o desempenho obtido nos testes com o *benchmark* STAMP. Os motivos para a seleção de dois sistemas STM ao invés de apenas o melhor, são: (1) os sistemas apresentaram resultados bem similares em algumas aplicações no teste com *benchmark* STAMP; (2) ao utilizarem dois sistemas, é possivel comparar os resultados entre os sistemas para verificar comportamentos distintos; (3) o sistema RSTM é um sistema importante na literatura de Memórias Transacionais e não foi utilizado no trabalho [58].

Para utilizar o *benchmark* com os sistemas SwissTM e RSTM, são necessárias modificações no *benchmark*. O *benchmark* utiliza um arquivo .h (*header file*) para definir as chamadas do sistema STM. O pacote de distribuição do *benchmark* apresenta suporte para os sistemas SwissTM (versão 2009-09-10) e TL2. Para suportar a última versão do sistema SwissTM (versão 2011-08-15) pequenas alterações no arquivo .h são necessárias. Já para suportar o sistema RSTM, é necessário escrever o arquivo .h do zero e alterar o código do *benchmark*.

Para a realização deste teste são selecionadas 5 aplicações presentes no teste anterior, mencio-

nadas abaixo:

1. **SSCA2**: aplicação que apresenta desempenho ruim no teste com *benchmark* STAMP (Capítulo 5), apesar de possuir características transacionais com severidade baixa, conforme classificação da Tabela 5.5;

2. **Intruder e Vacation**: aplicações que apresentam desempenho baixo no teste com *benchmark* STAMP (Capítulo 5) e que possuem características transacionais com severidade média, conforme classificação da Tabela 5.5;

3. **Labyrinth e Genome**: aplicações que apresentam desempenho bom no teste com *benchmark* STAMP (Capítulo 5) e que possuem características transacionais com severidade alta e média, respectivamente, conforme classificação da Tabela 5.5.

Para garantir a confiabilidade dos resultados gerados, todas as aplicações foram executadas mais de 30 vezes. Na Seção 4.2.1 são apresentados os cálculos realizados que indicam a quantidade de execuções por aplicação do STAMP. Os testes foram executados em uma máquina *multicore* descrita na Seção 4.1.

Para realizar a análise dos resultados, as seguintes métricas são utilizadas:

- ***Speed-up*:** conforme descrito na Seção 5.1 o cálculo de *speed-up* mostra o quão bem os sistemas STM escalam com um número crescente de *cores*. Um alto *speed-up* depende não somente do sistema STM mas também da aplicação;

- ***Aborts per Commit* (ApC):** mostra o percentual de transações que efetuaram *aborts* por transações que efetuaram *commit*. Essa métrica auxilia na análise indicando quando uma execução teve excessos de *aborts*, o que indica perda de desempenho.

Para todas as aplicações são executadas a versão sequencial e 3 versões STM: 2, 4, 8 *threads*. A seção seguinte apresenta os resultados desse teste bem como a análise dos resultados obtidos.

6.2 Resultados

Conforme citado na seção anterior, este teste pretende analisar as características transacionais através de outro *benchmark*, porém, utilizando o mesmo perfil de aplicações. Isso é possível, pois o *benchmark* simula o comportamento das aplicações STAMP através de um conjunto de parâmetros de entrada.

Após a seleção das aplicações do STAMP na seção anterior, é necessário determinar as características transacionais de cada aplicação. As características Eigen são derivadas a partir do conjunto de parâmetros de entrada do *benchmark* EigenBench. O trabalho [58] apresenta os parâmetros de entrada que simulam o comportamento das aplicações utilizadas nesse teste além de apresentar as características transacionais das aplicações do STAMP. A Tabela 6.1 apresenta as características

Tabela 6.1: Características das aplicações selecionadas do *benchmark* STAMP

Característica	**SSCA2**	**Intruder**	**Vacation**	**Labyrinth**	**Genome**
Conjunto de trabalho	400 MB	20 MB	256 MB	16 MB	20 MB
Variação de memória	0..500 MB	0..200 MB	0..600 MB	0..16 MB	0..70 MB
Comprimento da transação	3	24	226	357	88
Variação do comprimento	3	3..680	1..1239	3..1688	1..4000
Poluição	33%	5%	2%	50%	5%
Localidade	0.33	0.52	0.59	0.77	0.58
Contenção	0.0005%	22%	0.2%	5%	0.5%
Predominância	Baixo	Baixo	Alto	Baixo	Alto
Densidade	Alto	Alto	Alto	Baixo	Alto

de cada aplicação do STAMP. Este trabalho utiliza o mesmo conjunto de parâmetros de entrada e caracaterísticas derivadas presentes no trabalho [58].

A Tabela 6.1 apresenta as características Eigen (em negrito) das aplicações do STAMP. As demais características são complementares às características conjunto de trabalho e comprimento da transação e foram separadas em outra linha para facilitar a visualização. As características conjunto de trabalho e comprimento da transação apresentam a média obtida.

Os resultados dos testes são apresentados de duas maneiras: primeiramente, a Figura 6.1 apresenta o desempenho focando nos sistemas STM. Por fim, as Figuras 6.2 e 6.3 apresentam o desempenho invidivual por aplicação, focando no desempenho em ambos os sistemas. A Figura 6.4 apresenta a métrica ApC das aplicações.

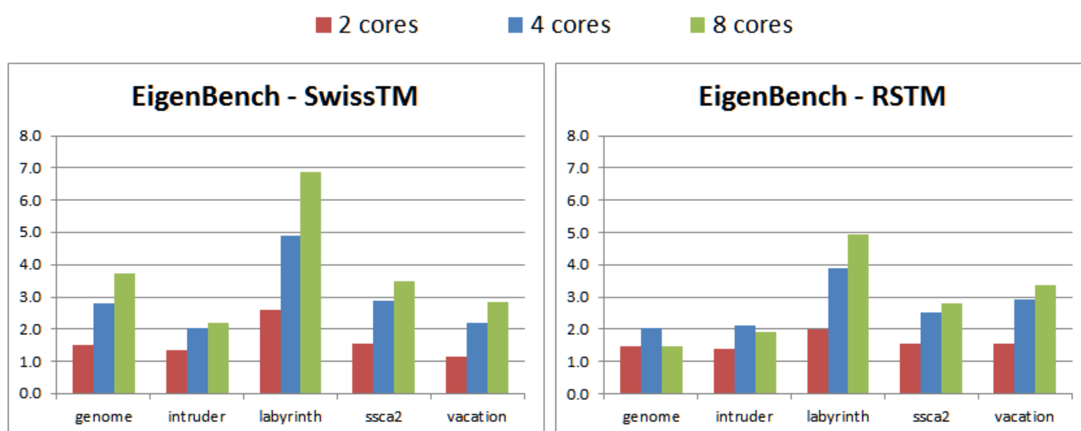

Figura 6.1: Desempenho (*speed-up*) dos sistemas STM com o *benchmark* EigenBench

Quando observados os gráficos da Figura 6.1 pode-se notar que os resultados estão superiores ao encontrado nas execuções com as aplicações reais do STAMP (Figuras 5.2 e 5.3). Entretanto, os valores apresentados na Figura 6.1 são semelhantes ao encontrado no trabalho original [58]. Uma das possibilidades para a diferença nos resultados é que o teste com o *benchmark* STAMP (Seção 5.2) utiliza os maiores valores de entrada possíveis, o que pode influenciar no *overhead* do sistema

STM e reduzir o desempenho das aplicações. O *benchmark* apresenta execuções das aplicações do STAMP, porém, não indica quais parâmetros de entrada foram utilizados nas aplicações originais para realizar a comparação das execuções. A aplicação Labyrinth apresentou valores acima do ideal pois seu código fonte favorece a programação paralela, eliminando problemas na versão paralela que degradam o desempenho na versão sequencial.

Conforme dito anteriormente, os valores apresentados são semelhantes aos encontrados no trabalho [58]. A aplicação Labyrinth apresenta valores muito próximos, enquanto a aplicação Genome apresenta diferença significativa apenas com 8 *cores*. As demais aplicações apresentam valores próximos, sendo que os resultados deste trabalho são um pouco maiores. Em relação a comparação entre os sistemas STM, é possível observar, novamente, uma superioridade do sistema SwissTM. Ele apresenta melhores resultados em 4 das 5 aplicações testadas, com desempenho além do ideal para a aplicação Labyrinth (com dois e quatro *cores*). Adicionalmente, é possível identificar a superioridade do sistema ao analisar a aplicação Genome, cuja diferença entre os sistemas é considerável.

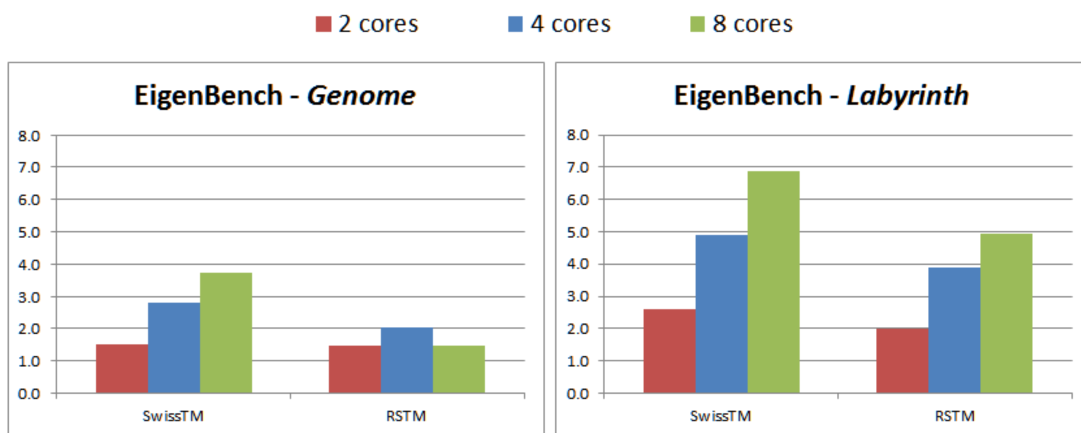

Figura 6.2: Desempenho (*speed-up*) das aplicações com o *benchmark* EigenBench (Parte 1)

De maneira geral, para investigar o comportamento de cada aplicação individualmente e identificar o motivo do desempenho apresentado, é necessário investigar separadamente a aplicação e suas características transacionais, bem como o sistema STM e os mecanismos utilizados (decisões de projeto) para o paralelismo da aplicação. Dentre as aplicações testadas nessa seção, todas apresentam resultados similares entre os sistemas STM utilizados. Apesar do sistema SwissTM apresentar resultados superiores em todas as aplicações, o comportamento das aplicações é semelhante independentemente das decisões de projeto dos sistemas STM. A única exceção é a aplicação Genome que apresenta uma diferença significativa entre os sistemas e é investigada separadamente por sistema, posteriormente. Dentre as aplicações que apresentam o mesmo comportamento, pode-se citar a aplicação Labyrinth, que teve desempenho bom nos dois sistemas STM, com uma certa vantagem no sistema SwissTM. Nas demais aplicações, a diferença de valores entre os sistemas é não significativa. A aplicação Intruder apresenta valores de *speed-up* entre 1.3 a 2.2 para o sistema SwissTM e 1.4 a 2.1 com o sistema RSTM. A aplicação Vacation apresenta valores de *speed-up*

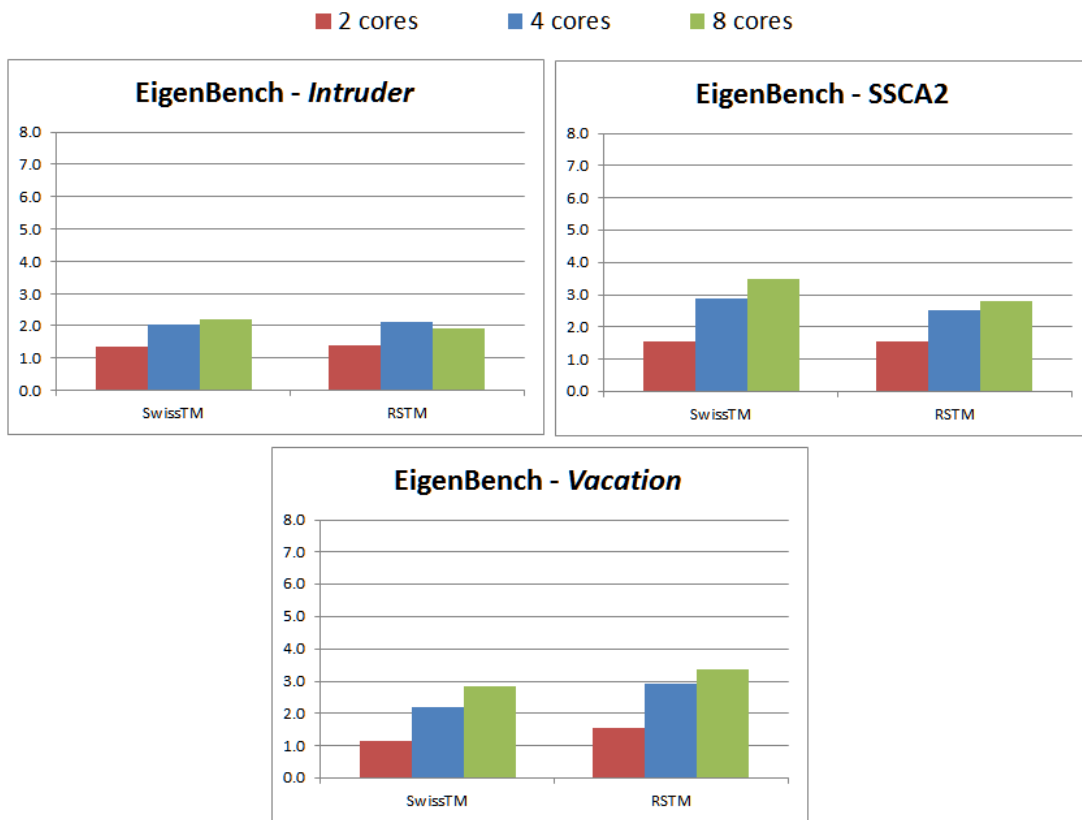

Figura 6.3: Desempenho (*speed-up*) das aplicações com o *benchmark* EigenBench (Parte 2)

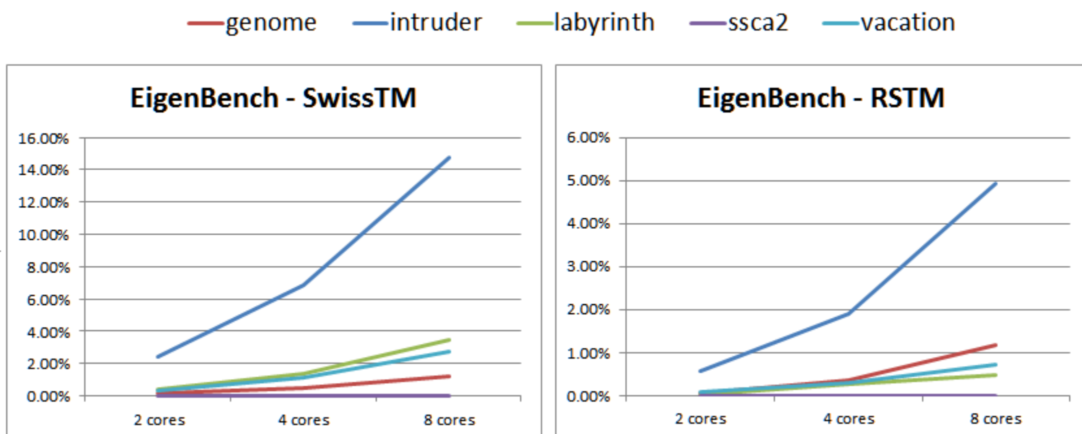

Figura 6.4: ApC (*Aborts per Commit*) do *benchmark* EigenBench

entre 1.1 e 2.8 (SwissTM) e 1.5 e 3.3 (RSTM). Finalmente, a aplicação SSCA2 variou seu *speed-up* entre 1.5 a 3.4 com o sistema SwissTM e 1.5 a 2.7 com o sistema RSTM.

A Tabela 6.2 auxilia na interpretação dessa conclusão, apresentando a variação de *speed-up* entre os sistemas testados. A tabela mostra os valores mínimos e máximos do *speed-up* para cada sistema STM independentemente da quantidade de *threads* executadas. Os valores estão arrendondados para facilitar a interpretação. A tabela não apresenta os valores da aplicação Genome, visto que a

mesma apresenta valores distintos entre os sistemas STM.

Tabela 6.2: Variação do *speed-up* entre os sistemas STM

Aplicação	**SwissTM**		**RSTM**	
	Mínimo	Máximo	Mínimo	Máximo
Intruder	1.3	2.2	1.4	2.1
Labyrinth	2.5	6.8	2.0	4.9
SSCA2	1.5	3.4	1.5	2.7
Vacation	1.1	2.8	1.5	3.3

Na Tabela 6.2, observa-se valores mínimos muitos próximos entre os sistemas STM, analisando cada aplicação individualmente. Da mesma forma, é possível observar o mesmo com os valores máximos, com exceção da aplicação Labyrinth, que apresenta valor máximo distante entre os sistemas STM. Essa distância encontrada deve-se ao fato do sistema RSTM não desempenhar satisfatoriamente ao utilizar 8 *cores*. O mesmo pode ser observado, em menor grau, nas demais aplicações. Além disso, o sistema SwissTM consegue desempenhar melhor com 8 *cores* devido ao seu sofisticado mecanismo de resolução de conflito, que previne abortar transações longas com frequência.

Através dessa análise conclui-se, que ambos os sistemas estão apresentando desempenho semelhante ao comparar a mesma aplicação. As características dos sistemas STM não impactam, nesse caso, na interpretação do desempenho das aplicações. Para tanto, somente as caracaterísticas das aplicações são levadas em consideração na análise individual de cada aplicação. A constatação sobre a semelhança nas execuções é nova na literatura, principalmente porque as últimas versões de ambos os sistemas são recentes e não existem trabalhos que apresentem uma avaliação sobre eles. Essa constatação é importante para indicar o crescimento dos sistemas STM na literatura, e pode-se concluir que os sistemas estão convergindo para sistemas STM com desempenho superior. Apesar desse ponto positivo, ainda é necessário investir nas ferramentas de STM para reduzir o *overhead* gerado com o aumento de *cores*. Ambos os sistemas possuem um desempenho inferior ao executar com 8 *cores*, por exemplo.

Para realizar a análise individual de cada aplicação, são utilizados todos os recursos disponíveis. São analisados os gráficos de desempenho (Figura 6.1, 6.2 e 6.3), a métrica ApC (Figura 6.4), as características ortogonais de cada aplicação (Tabela 6.1) e o código fonte. Na listagem abaixo, são apresentadas as principais considerações para cada aplicação:

- **Genome:** apresenta desempenho distinto entre os sistemas STM. O resultado com o sistema SwissTM apresenta diferença do resultado apresentado em [58], apenas com 8 *cores*. A aplicação possui diferentes comprimentos de transações, variando de 1 a 4000. Apesar do comprimento das transações mais frequente ser pequeno, algumas transações com comprimento maior foram utilizadas (investigadas a partir do código fonte), o que impactou no desempenho geral da aplicação. Predominância e densidade altas também influenciam no desempenho. É possivel analisar ainda que a aplicação é a única, dentre aquelas utilizadas,

a apresentar diferença significativa de desempenho entre os sistemas STM. SwissTM obteve resultados baixos, porém conseguiu escalar até *speed-up* 3.7 (8 *cores*), enquanto o sistema RSTM não obteve *speed-up* maior que 2.0 (4 *cores*). Esse comportamento ocorre devido as decisões de projeto utilizadas pelo sistema SwissTM, que consegue lidar satisfatoriamente tanto com transações curtas como com transações longas, que é o caso específico da aplicação Genome.

- **Intruder:** apresenta o pior desempenho entre as aplicações testadas, com *speed-up* máximo de 2.2 com o sistema SwissTM e 2.1 com sistema RSTM. Em comparação com o trabalho [58], apresenta *speed-ups* melhores com dois e quatro *cores* e pior com oito *cores*. A aplicação apresenta uma variedade de tamanhos de memória utilizada, além de possuir um nível de contenção considerável, conforme Tabela 6.1. Entretanto, analisando-se a tabela, algumas características apresentadas são semelhantes a aplicação Labyrinth que teve um bom desempenho. Apesar das semelhanças, o principal motivo para a queda no desempenho da aplicação é o nível alto de contenção que sobrecarrega o sistema STM e acaba aumentando o número de *aborts*. O percentual de *aborts* por *commit* comprova essa conclusão (Figura 6.4).

- **Labyrinth:** apresenta o melhor desempenho do teste. O resultado com o sistema SwissTM apresenta diferença do resultado apresentado em [58], com *speed-ups* melhores com dois e quatro *cores* e pior com oito *cores*. A aplicação Labyrinth possui uma distribuição uniforme no comprimento das transações, e a quantidade de memória utilizada frequentemente é pequena. Apesar de possuir tamanho das transações grandes (acima de 1600), elas possuem baixa densidade (muitas operações não transacionais dentro da transação) e baixa predominância (muitas operações não transacionais na aplicação), como pode ser observado na Tabela 6.1. A tabela informa, ainda, contenção baixa, que pode ser comprovado com o percentual baixo de *aborts* (Figura 6.4). Além disso, a característica localidade é alta (muitas operações utilizando o mesmo conjunto de endereços de memória dentro da transação). Todas essas características sugerem que não é o *overhead* do sistema STM que governa o desempenho da aplicação, mas sim a eficiência da detecção de conflito dos sistemas STM. Essa conclusão é comprovada pelos resultados dos gráficos da Figura 6.1, onde o sistema SwissTM obteve *speed-up* maior do que o ideal para dois e quatro *cores* e quase ideal para oito *cores*. Com sistema RSTM, apesar da limitação de escalibilidade, obteve resultado ideal para dois e quatro *cores* e não escalou satisfatoriamente com oito *cores* devido a suas limitações. Da mesma forma, o percentual de *aborts* para dois e quatro *cores* é insignificante para ambos os sistemas STM. o percentual de *aborts* com oito *cores* difere entre os sistemas mas não impacta no desempenho geral.

- **SSCA2:** apresenta desempenho baixo, um pouco superior ao encontrado nos testes com o *benchmark* STAMP (Seção 5.2). O resultado com o sistema SwissTM apresenta diferença do resultado apresentado em [58], com *speed-ups* melhores em todos os *cores*. Apesar da aplicação apresentar comprimento da transação curta, ela possui uma grande quantidade de

memória utilizada. Essa combinação reduziu o desempenho da aplicação. O interessante desta aplicação é que, ao contrário da Intruder, ela não apresenta percentual alto de *aborts*. Ou seja, a perda de desempenho dessa aplicação não está nos conflitos mas sim no *overhead* do sistema STM ao lidar ao mesmo tempo com comprimento de transações curto e muita memória.

- **Vacation:** apresenta desempenho baixo, um pouco superior ao encontrado nos testes com o *benchmark* STAMP (Seção 5.2). O resultado com o sistema SwissTM apresenta diferença do resultado de [58], com *speed-ups* melhores com quatro e oito *cores* e pior com dois *cores*. A aplicação Vacation possui a seguinte combinação de características que degradam seu desempenho: grande volume de memória e diversos comprimentos de transação possuindo valor médio de 226. Além disso, predominância e densidade alta auxiliam no comportamento dessa aplicação. Um fator interessante sobre os resultados dessa aplicação é a comparação entre os sistemas STM. Essa é a única aplicação que possui resultados superiores com o sistema RSTM, mesmo que essa diferença não seja tão grande. Não é possível determinar qual aspecto do sistema RSTM garantiu um melhor desempenho nessa aplicação. Entretanto, destaca a necessidade de uma nova avaliação de decisões de projeto como realizado em [2], que realiza testes de tentativa e erro utilizando diversas abordagens para as principais caraterísticas dos sistemas STM. Isso é importante para avaliar o comportamento das decisões tanto isoladamente, como em conjunto.

A partir das conclusões geradas através da análise individual das aplicações testadas, é possível também chegar a conclusões gerais sobre o comportamento geral do teste. Na listagem abaixo, são apresentadas as principais considerações do teste:

- **Baixo desempenho sem nível alto de *aborts*:** através do teste com a aplicação SSCA2, é possível concluir que não somente conflitos em transações é indício para a perda de desempenho, mas que outros fatores também contribuem para o desempenho desfavorável. No caso da aplicação SSCA2, a combinação transações de comprimento pequeno e grande quantidade de memória gerou *overhead* em ambos os sistemas testados, o que limitou o *speed-up* a valores muito abaixo do valor ideal em todas as configurações *cores*. O próprio sistema STM tem *overhead* interno ao lidar com com grande volume de memória, e aplicações livres de *aborts* (ou com níveis baixos) não são garantia de desempenho superior;

- **Desempenho semelhante entre os sistemas STM:** ambos os sistemas STM possuem desempenho semelhante ao se comparar a mesma aplicação (com exceção da aplicação Genome), não levando em conta os número de *speed-up* gerados, mas sim o comportamento geral. Como exemplo, temos a aplicação Labyrinth que desempenhou satisfatoriamente nos testes, apresentando ótimos resultados em ambas as ferramentas. Da mesma forma, a aplicação Intruder apresenta desempenho baixo em ambos os sistemas, com pequena variação de *speed-ups*. Esse comportamento é observado também no teste com *benchmark* STAMP,

o que é esperado, visto que são simulações das aplicações reais do *benchmark*. Isso não é necessariamente uma regra, mas apresenta indícios que ambas as ferramentas, que tiveram o melhor desempenho, estão convergindo para o mesmo lugar, indicando o crescimento dos sistemas STM na literatura.

- **Desempenho satisfatório:** a aplicação Labyrinth é um exemplo para mais uma constatação. Primeiramente, apresenta resultados excelentes com o sistema STM SwissTM, o que contradiz trabalhos que criticam o potencial de Sistemas de Memórias Transacionais de *Software*, como em [8]. Em comparação com esse trabalho, apenas o sistema TL2 é comum, pois os demais sistemas utilizados no trabalho são baseados em compilador. Além disso, o sistema TL2 teve o pior desempenho dos testes com *benchmark* STAMP e não possui mecanismos atuais para lidar com *overhead*. A aplicação RSTM também apresentou resultados ótimos para dois e quatro *cores*, porém perdeu desempenho com oito *cores*, conforme limitação já mencionada;

- **Influência de mais de uma característica:** pode-se observar nas conclusões individuais das aplicações que analisar o desempenho e identificar o motivo para o desempenho alcançado não é algo trivial, pois, geralmente, não há informações suficientes da aplicação para encontrar o real motivo. Foi identificado que um conjunto de característica é necessário para determinar o desempenho, pois dificilmente apenas uma característica é responsável pelos resultados. Adicionalmente, não somente as caracaterísticas citadas como principais tornam o desempenho baixo ou alto, mas também características coadjuvantes com diferentes pesos. Esse é o caso da aplicação Labyrinth, por exemplo, que tem como características principais para o bom desempenho: densidade, predominâcia e pouca memória. Porém, analisando a Tabela 6.1 é possível identificar mais três características importantes, que auxiliam no bom desempenho da aplicação: localidade temporal (que indica que as transações utilizam geralmente o mesmo conjunto de endereços de memória), contenção (que apresenta 5% apenas de chance de uma transação ter conflito) e poluição (que indica que apenas 50% das operações em acessos compartilhados é escrita);

- **Grande quantidade de memória:** pode-se observar que, em geral, aplicações com grande quantidade de memória têm baixo desempenho. A aplicações SSCA2 e Vacation são exemplos disso: ambas possuem grande quantidade de memória independentemente do comprimento das transações da aplicação (SSCA2 apresenta tamanho fixo e pequeno enquanto Vacation apresenta diversos comprimentos e comprimento longo na média). Essa constatação não pode ser expandido para qualquer aplicação de Memória Transacional mas é um indício de que é preciso mecanismos mais robustos para lidar com grande quantidade de memória nos sistemas STM.

Confiabilidade

Conforme citado na Seção 4.2.1 este trabalho utiliza métodos estatísticos para determinar a quantidade de execuções necessárias para cada conjunto de aplicação e sistema STM e ainda para determinar o intervalo de confiança dos resultados. No Apêndice A.2 são apresentados os resultados de todas as execuções deste teste. As Tabelas A.11, A.12, A.13 exibem o número de execuções de cada aplicação enquanto as Tabelas A.14, A.15, A.16 exibem o intervalo de confiança de cada resultado.

6.3 Resumo do Capítulo

Conforme observado na Seção 6.2, as aplicações simuladas do *benchmark* STAMP apresentam desempenho baixo em todos os sistemas utilizados, com exceção da aplicação Labyrinth. Testes como este foram executados também em [58]. Em [58], trabalho no qual esse teste foi beaseado, apresenta o mesmo teste com as cinco aplicações STAMP e o sistema SwissTM. Os resultados mostraram-se melhores do que o trabalho original. Com exceção das aplicações Genome e Labyrinth, que apresentam resultados menores em alguma configuração de *cores*, todas as demais aplicações apresentam resultados melhores. É possível listar duas justificativas para a diferença nos valores: (1) Os trabalhos executaram testes com versoes diferentes do sistema SwissTM (o trabalho [58] utiliza a versão 2009-09-10, enquanto este teste utiliza a última versão do sistema - versão 2011-08-15); (2) A arquitetura utilizada é diferente, o que pode influenciar no desempenho. Ambas as justificativas apresentam-se coerentes visto que a diferença de resultados é aceitável.

Em relação a comparação entre os sistemas STM, o sistema SwissTM mostrou, novamente, superioridade nas execuções, obtendo melhores resultados com quatro das cinco aplicações. As aplicações que mais chamaram a atenção, neste caso, foram Genome e Vacation. Genome, apresentou diferença significativa entre os sistemas, cujo *speed-up* com oito *cores* foi 3.7 e 1.4, para os sistemas SwissTM e RSTM respectivamente. Essa diferença considerável entre os sistemas indica ser consequência do comportamento da aplicação Genome, que apresenta variação entre comprimentos de transações curto e longo, conforme observado na Seção 6.2, e é melhor trabalhado no sistema SwissTM, devido a seus mecanimos voltados tanto para transações curtas como longas. A aplicação Vacation, foi a única aplicação onde o sistema RSTM teve vantagem nos resultados. Não é possível determinar o motivo da diferença, visto que a diferença não é significativa. Mas esse fato indica que o sistema SwissTM ainda precisar melhorar seu desempenho, visto que não é satisfatório em 100% das aplicações transacionais.

Através da nova caracterização das aplicações transacionais, foi possível obter melhores resultados ao analisar o comportamento das aplicações. O *benchmark*, apresenta características que melhor definem uma aplicação transacional, identificando comportamentos mais específicos, como são o caso das características densidade, predominância e localidade Temporal. Além disso, a possibilidade de simular outras aplicações foi de suma importância para este trabalho, pois permitiu uma análise gradual das aplicações do STAMP, iniciando com as aplicações reais e, posteriormente,

utilizando aplicações simuladas (com mesmo comportamento das originais) e utilizando uma nova maneira de caracterizar as aplicações de maneira mais detalhista.

Assim como observado na Seção 5.3, os sistemas STM ainda precisam melhorar no desempenho. Os resultados de algumas aplicações ficaram, novamente, muito abaixo do ideal, apesar desse comportamento ser o esperado, tendo em vista que as aplicações são simuladas e representam o comportamento das aplicações reais. Na comparação dos trabalhos, foram identificadas diferenças que, de acordo com as jusificativas apresentadas não são impactantes no resultado final, pois todas as aplicações apresentam o mesmo comportamento, independentemente do trabalho. Devido essas constatações, também reforçadas pelo trabalho [58], são esperados melhores resultados nos sistemas STM atuais.

A principal contribuição deste capítulo é a extensão do trabalho [58], ao suportar o sistema RSTM e realizar comparações com o sistema SwissTM. Foi possível, através de testes reforçar a superioridade da ferramenta SwissTM que apesar de ser criada recentemente possui um conjunto de decisões de projeto eficaz. Além disso, identificou-se similaridade entre os sistemas STM com o *benchmark* EigenBench, o que indica que os resultados apresentados em [58] são coerentes e, ainda, que o desempenho dos sistemas estão muito próximo, embora o SwissTM seja superior em determinadas aplicações. Outra contribuição importante deste capítulo é a avaliação detalhada de cada aplicação do *benchmark* STAMP através de uma nova ótica, utilizando-se características transacionais mais detalhadas e obtendo-se as características que mais interferem no desempenho de cada aplicação.

7. Teste de Características Transacionais com *Benchmark* EigenBench

Este capítulo apresenta os testes de características com o *benchmark* EigenBench. Na Seção 7.1, é apresentada a metodologia, descrevendo-se como o testes são executados e seus objetivos. Na Seção 7.2, são apresentados os resultados e uma análise crítica com observações sobre os testes executados. Por fim, na Seção 7.3, é apresentado um resumo com as considerações finais e as contribuições do capítulo.

7.1 Metodologia

O teste de características transacionais com o *benchmark* EigenBench visa complementar o teste do *benchmark* EigenBench (Capítulo 6), investigando-se de maneira mais isolada o impacto que cada característica transacional possui nas aplicações simuladas do *benchmark* STAMP. Isso nos permite averiguar o comportamento das aplicações de maneira controlada, alterando-se uma característica por vez e avaliando-se as mudanças ocorridas nos resultados. O teste da Seção 6 simula o comportamento das aplicações do STAMP e apresentam um novo conjunto de características para tais aplicações. Este teste tem por objetivo, explorar essas características transacionais, de modo a complementar a análise das aplicações e verificar o impacto (importância) que essas características possuem no comportamento da aplicação. O teste permite, inicialmente, validar os resultados obtidos, bem como encontrar informações adicionais não capturadas no teste anterior.

Para a realização desta etapa, são selecionadas as quatro aplicações que apresentam resultados baixos presentes no teste com *benchmark* EigenBench: Genome, Intruder, SSCA2 e Vacation. O objetivo desta seleção é justamente identificar oportunidades de melhoria no desempenho dessas aplicações, o que é algo mais complexo na aplicação Labyrinth, visto que ela apresenta *speed-up* maior que o ideal com dois e quatro *cores* e próximo do ideal com oito *cores*. Além disso, é selecionado o sistema que possui o melhor desempenho no teste, SwissTM, pois o foco deste teste é justamente as características da aplicação e não a comparação entre sistemas STM.

Os resultados deste teste são obtidos através da abordagem 'tentativa e erro', pois é preciso identificar como uma característica transacional irá impactar no desempenho da aplicação. Entretanto, as tentativas são guiadas a partir de todo o conhecimento adquirido sobre as aplicações utilizadas, ou seja, são analisados os resultados obtidos, a tabela de características Eigen (Tabela 6.1 e as observações individuais realizadas presentes na Seção 6.2. Todos esses recursos são utilizados para identificar qual característica transacional é importante para uma determinada aplicação e, a partir disso, isolar tal característica e analisar o impacto no desempenho. Devido à quantidade de execuções necessárias com a abordagem de tentativa e erro, esse teste não apresenta intervalo de confiança, apesar de ser executado no mínimo 30 vezes. Além disso, conforme apresentado nos testes com *benchmark* EigenBench (Apêndice A.2) as execuções com esse *benchmark* apresentam desvio padrão menor que 0.1 em todas as execuções realizadas.

Para realizar a análise dos resultados, a seguinte métrica é utilizada:

- ***Speed-up*:** conforme descrito na Seção 5.1 o cálculo de *speed-up* mostra o quão bem os sistemas STM escalam com um número crescente de *cores*. Um alto *speed-up* depende não somente do sistema STM mas também da aplicação;

Para todas as aplicações são executadas a versão sequencial e 3 versões STM: 2, 4, 8 *threads*. A seção seguinte apresenta os resultados desse teste bem como a análise dos resultados obtidos.

7.2 Resultados

Conforme citado na seção anterior este teste pretende investigar as caraterísticas transacionais de maneira isolada. Dentre as características analisadas na Seção 6.2 algumas mostraram-se relevantes para determinar o desempenho de aplicações. Como exemplo, pode-se citar: nível de contenção, que impactou o resultado da aplicação Intruder; localidade temporal, que auxiliou no bom resultado da aplicação Labyrinth; conjunto de trabalho, que impactou nas aplicações com grande quantidade de memória (SSCA2 e Vacation); e, por fim, comprimento das transações, que impactou aplicações com transações tanto curtas como longas (Genome). Dentre as características citadas, não é possível realizar testes com a característica contenção, pois não é possível derivar a fórmula utilizada no trabalho [58], que oferece pouca informação a respeito. As demais caraterísticas, tais como densidade e predominância, são também importantes e são citadas no teste da Seção 6.2. Suas derivações, são não triviais e dependem de fatores do sistema, não apresentadas no trabalho [58]. Portanto, com exceção de contenção, densidade e predominância, as demais características apresentadas são exploradas nesse teste.

Como citado anteriormente, o conjunto de modificações disponíveis para realização do teste de tentativa e erro é vasto. Para reduzir o escopo de parâmetros a serem modificados, é preciso analisar aqueles utilizados para a simulação das aplicações, suas características ortogonais e seu desempenho. Através dessa análise, é possível identificar oportunidades de melhoria em alguma característica e a partir disso, realizar testes e medir o desempenho. As próximas seções apresentam a análise inicial, as modificações realizadas e os resultados obtidos para cada aplicação selecionada.

7.2.1 Teste de Características Transacionais com Aplicação Genome

A aplicação apresenta desempenho baixo no teste com *benchmark* EigenBench (Capítulo 6). Conforme observado na Seção 6.2, a aplicação possui comprimento de transação curto no geral, mas possui também algumas transações com comprimento longo que, em conjunto com predominância e densidade altas influenciam no desempenho da aplicação. Analisando-se o conjunto de características ortogonais (Tabela 6.1), é possível identificar que a aplicação possui um *range* grande de comprimento de transações (1..4000). Além disso, conforme citado acima, a variação de transações curtas e longas influenciaram o desempenho. Observando-se, portanto, as características

ortogonais (Seção 3.2.1), identifica-se a possibilidade de modificar a característica comprimento da transação. A característica comprimento da transação pode ser derivada conforme Fórmula 7.1.

$$T_{len} = R1 + R2 + W1 + W2 \tag{7.1}$$

Os parâmetros `R1, R2, W1` e `W2` são de entrada do *benchmark* e são descritos na Tabela 3.1 da Seção 3.2.1. Os parâmetros de entrada da aplicação Genome são apresentados na Tabela 7.1. Cada linha da tabela representa uma *thread* a ser executada.

Tabela 7.1: Parâmetros de entrada da aplicação Genome no *benchmark* EigenBench

A1	A2*N	A3	R1	W1	R2	W2	R3o	W3o	R3i	W3i	LCT
			28	2	65	5	20	20	0	0	
256KB	2MB	256KB	145	5	340	10	80	80	0	0	512
			770	30	1660	40	500	500	0	0	

Como pode ser observado na Tabela 7.1 e na Equação 7.1, diversas modificações podem ser realizadas para alterar o comprimento da transação. Uma das possibilidades de modificação que melhora o desempenho da aplicação é a alteração de todos os parâmetros pertencentes a característica comprimento da transação. Visto que essa aplicação é configurada com 3 *threads* diferentes, é preciso identificar qual delas é relevante nesse caso. Após algumas tentativas, é identificado que as *threads* que apresentam comprimento da transação maiores são as que apresentam maior *overhead* no sistema STM. Portanto, as alterações no comprimento da transação são realizadas nas *threads* 2 e 3. A Tabela 7.2 mostra as modificações realizadas nesses parâmetros para melhorar o desempenho da aplicação. A tabela apresenta na primeira linha a versão original dos parâmetros e, nas demais linhas, as modificações realizadas.

A Figura 7.1 apresenta o desempenho das modificações apresentadas na Tabela 7.2. A versão 4 ilustra o comportamento inverso ao esperado, cujo valores de todos os parâmetros são dobrados de tamanho. As demais versões apresentam valores reduzidos uniformemente em busca de comprimento da transação menor.

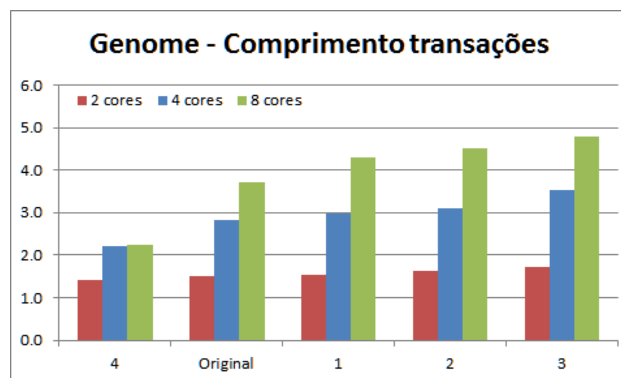

Figura 7.1: Desempenho (*speed-up*) da aplicação Genome conforme Tabela 7.2

Tabela 7.2: Variação da característica Comprimento da Transação na aplicação Genome

Versão	A1	A2*N	A3	R1	W1	R2	W2	R3o	W3o	R3i	W3i	LCT
				28	2	65	5	20	20	0	0	
Original	256KB	2MB	256KB	145	5	340	10	80	80	0	0	512
				770	30	1660	40	500	500	0	0	
				-	-	-	-	-	-	-	-	
1	-	-	-	-	-	-	-	-	-	-	-	-
				385	15	815	20	-	-	-	-	
				-	-	-	-	-	-	-	-	
2	-	-	-	73	3	170	5	-	-	-	-	-
				385	15	815	20	-	-	-	-	
				-	-	-	-	-	-	-	-	
3	-	-	-	73	3	170	5	-	-	-	-	-
				193	8	408	10	-	-	-	-	
				-	-	-	-	-	-	-	-	
4	-	-	-	290	10	680	20	-	-	-	-	-
				1540	60	3260	80	-	-	-	-	

Conforme observado na Figura 7.1, a redução do comprimento da transação nas *threads* 2 e 3 melhoram o desempenho da aplicação. Além disso, ess modificação comprova a constatação da Seção 6.2, em que o motivo do baixo desempenho é a variação de transações longas e curtas. A redução no comprimento de transações longas aumenta o desempenho significativamente. É importante ressaltar que o resultado encontrado na Figura 7.1 não é considerado ideal e tem o objetivo apenas de apresentar o impacto da característica no conjunto de características da aplicação Genome. As variações de *speed-up* entre as configurações testadas é pouco significativas; destaca-se, porém, o impacto que a variação no comprimento das transações tem no desempenho geral das aplicações.

7.2.2 Teste de Características Transacionais com Aplicação Intruder

A aplicação apresenta desempenho baixo no teste com *benchmark* EigenBench (Capítulo 6). Conforme observado na Seção 6.2, a aplicação possui nível de contenção considerável e transações curtas. Analisando-se o cojunto de características (Tabela 6.1), é possível identificar que o nível de contenção é o mais alto entre todas as aplicações. Entretanto conforme citado anteriormente, não é possível derivar a contenção das aplicações, e por isso, outra característica precisa ser explorada. Outra possibilidade de modificação para a melhora do desempenho da aplicação é a alteração da característica localidade temporal, que mostrou-se importante no teste com aplicação Labyrinth. A característica é mapeada diretamente com o parâmetro de entrada `LCT` e a Tabela 7.3 apresenta as modificações realizadas nesse parâmetro. A tabela apresenta a versão original na primeira linha e, nas demais, as modificações realizadas. O parâmetro `LCT` varia de 0 a 1024 com múltiplos de 256. Quanto maior o número, maior a probabilidade de utilização dos mesmos endereços de memória em

uma transação. Para cada configuração apresentada, uma linha representa uma *thread* que pode ser executada 1 ou mais vezes.

Tabela 7.3: Variação da característica Localidade Temporal na aplicação Intruder

Versão	A1	A2*N	A3	R1	W1	R2	W2	R3o	W3o	R3i	W3i	LCT
Original	1KB	1KB	64KB	8 18 20	2 2 10	0 27 180	0 3 20	2 4 10	2 4 10	0 0 0	0 0 0	256
1	-	-	-	-	-	-	-	-	-	-	-	512
2	-	-	-	-	-	-	-	-	-	-	-	768
3	-	-	-	-	-	-	-	-	-	-	-	0

A Figura 7.2 apresenta o desempenho das modificações apresentadas na Tabela 7.3. A versão 3 ilustra o comportamento inverso ao esperado, onde a localidade é menor do que na versão original. As demais versões incrementam o parâmetro LCT em busca de um desempenho superior.

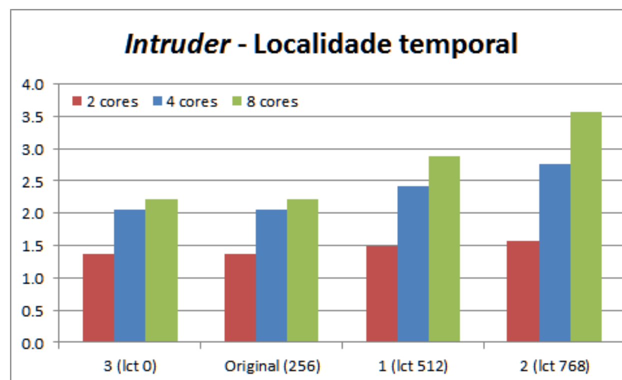

Figura 7.2: Desempenho (*speed-up*) da aplicação Intruder conforme Tabela 7.3

Conforme observado na Figura 7.2, a alteração da característica localidade temporal altera o desempenho da aplicação consideravelmente mudando de 2.2 a 3.5 o *speed-up* para oito *cores*. Entretanto, a característica tem efeito, mais significativo, a partir de quatro *cores*, visto que com dois *cores* os valores estão próximos. Isso sugere que a característica demonstra melhores resultados ao escalar com mais *cores*. Como citado na Seção 7.2.1, o resultado apresentado na Figura 7.2 não é considerado ideal e tem o objetivo apenas de apresentar o impacto da característica no conjunto da aplicação Intruder.

7.2.3 Teste de Características Transacionais com Aplicação SSCA2

A aplicação apresenta desempenho baixo no teste com o *benchmark* EigenBench (Capítulo 6). O principal motivo isso, conforme mencionado na Seção 6.2, é a grande quantidade de memória utilizada, considerada um dos fatores de queda no desempenho das aplicações. Além disso, ao se analisar a tabela das características ortogonais (Tabela 6.1), é possível identificar que o conjunto

de memória é grande (400 MB na média). Portanto, identifica-se a possibilidade de modificar a característica conjunto de trabalho. A característica pode ser derivada conforme Fórmula 7.2.

$$Mem_{size} = A1 + A2 + A3 \tag{7.2}$$

Os parâmetros `A1, A2` e `A3` são de entrada do *benchmark* e são descritos na Tabela 3.1 da Seção 3.2.1. Os parâmetros de entrada da aplicação SSCA2 são apresentados na Tabela 7.4. A aplicação apresenta apenas uma *thread* de execução que é executada 1 ou mais vezes.

Tabela 7.4: Parâmetros de entrada da aplicação SSCA2 no *benchmark* EigenBench

A1	**A2*N**	**A3**	**R1**	**W1**	**R2**	**W2**	**R3o**	**W3o**	**R3i**	**W3i**	**LCT**
32MB	8KB	64KB	1	1	1	0	15	0	0	0	0

Como pode ser observado na Tabela 7.4 e na Equação 7.2, diversas modificações podem ser realizadas para alterar o conjunto de trabalho. Uma das possibilidades de modificação que melhora o desempenho da aplicação é a alteração do parâmetro `A3` (Tamanho do *Cold Array* - array com dados não transacionais). A Tabela 7.5 apresenta as modificações realizadas nesse parâmetro para melhorar o desempenho da aplicação: é possível observar, na primeira linha, a versão original dos parâmetros e, nas demais linhas, as modificações realizadas.

Tabela 7.5: Variação da característica Conjunto de Trabalho (*Cold Array*) na aplicação SSCA2

Versão	**A1**	**A2*N**	**A3**	**R1**	**W1**	**R2**	**W2**	**R3o**	**W3o**	**R3i**	**W3i**	**LCT**
Original	32MB	8KB	64KB	1	1	1	0	15	0	0	0	0
1	-	-	32KB	-	-	-	-	-	-	-	-	-
2	-	-	16KB	-	-	-	-	-	-	-	-	-
3	-	-	8KB	-	-	-	-	-	-	-	-	-
4	-	-	128KB	-	-	-	-	-	-	-	-	-

A Figura 7.3 apresenta o desempenho das modificações apresentadas na Tabela 7.5. As versões 1, 2 e 3 decrementam o tamanho do *array* em busca de um desempenho melhor. Já a versão 4 ilustra o comportamento inverso, aumentando o tamanho do *array*.

Conforme observado na Figura 7.3, a alteração do tamanho de um dos *arrays* do *benchmark* é suficiente para melhorar o desempenho da aplicação, apesar da melhoria ser pouco significativa a partir da versão 2. É importante ressaltar que diversas outras variações podem ser realizadas para alterar o conjunto de trabalho. Aquela utilizada neste teste é apenas uma possibilidade e nos dá indício de que não apenas memória compartilhada em transações impacta no desempenho mas, também operações locais, que é o caso do parâmetro `A3`, que possui apenas dados não transacionais, conforme indica Tabela 3.1.

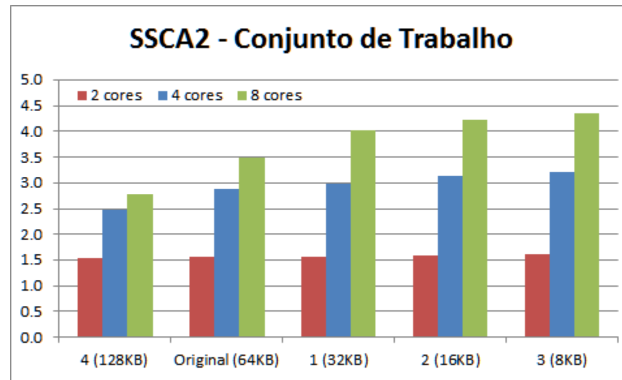

Figura 7.3: Desempenho (*speed-up*) da aplicação SSCA2 conforme da Tabela 7.5

7.2.4 Teste de Características Transacionais com Aplicação Vacation

A aplicação apresenta desempenho baixo no teste com o *benchmark* EigenBench (Capítulo 6). Assim como a aplicação SSCA2, esta aplicação apresenta conjunto de trabalho grande (256 MB na média), o que impacta consideravelmente no desempenho. Devido a essa similaridade com a aplicação SSCA2, identifica-se a possibilidade de modificar também a característica conjunto de trabalho, porém, utilizando outro parâmetro. Como visto na Seção 7.2.3, a característica conjunto de trabalho pode ser derivada conforme Fórmula 7.2.

Neste teste o parâmetro modificado é o `A1` (Tamanho do *Hot Array* - array com dados compartilhados, acessados concorrentemente) que representa o array principal da aplicação. A Tabela 7.6 apresenta a versão original e as modificações realizadas.

Tabela 7.6: Variação da característica Conjunto de Trabalho (*Hot Array*) na aplicação Vacation

Versão	A1	A2*N	A3	R1	W1	R2	W2	R3o	W3o	R3i	W3i	LCT
Original	512KB	32KB	64KB	195	5	45	2	0	0	0	0	0
1	256KB	-	-	-	-	-	-	-	-	-	-	-
2	128KB	-	-	-	-	-	-	-	-	-	-	-
3	64KB	-	-	-	-	-	-	-	-	-	-	-
4	1MB	-	-	-	-	-	-	-	-	-	-	-

A Figura 7.4 mostra o desempenho das modificações na aplicação Vacation. A versão 4 ilustra o comportamento inverso ao esperado, onde o valor de `A1` é dobrado de tamanho. As demais versões decrementam o tamanho do *array* em busca de melhor desempenho.

Conforme observado na Figura 7.4 a alteração do tamanho do *array* principal impactou no desempenho da aplicação, apesar de apresentar variação pouco significativa. Independentemente do desempenho não apresentar crescimento significativo entre os *cores*, a mudança no desempenho já é um indício da importância da característica no conjunto de caracteristicas da aplicação Vacation.

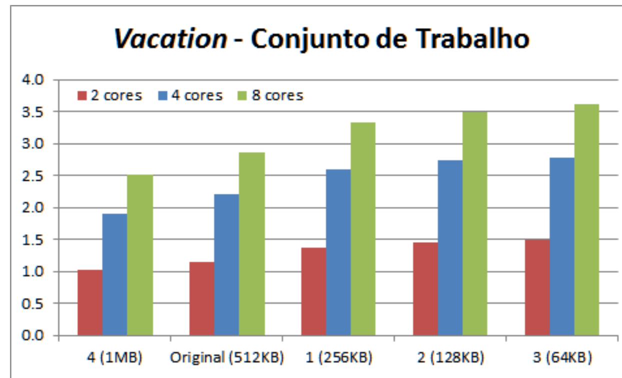

Figura 7.4: Desempenho (*speed-up*) da aplicação Vacation conforme Tabela 7.6

7.3 Resumo do Capítulo

Conforme observado na Seção 7.2, as modificações realizadas nas aplicações simuladas do *benchmark* STAMP melhoraram o desempenho das aplicações, o que substancia as observações realizadas em relação aos testes de desempenho do *benchmark* EigenBench (Seção 6.2). Como observado nas seções anteriores, os resultados encontrados após modificação dos parâmetros de entrada não são considerados ideais e tem o objetivo apenas de apresentar o impacto da característica no conjunto de características das aplicações.

Ao se modificar os parâmetros de entrada do *benchmark*, foi possível isolar as características transacionais e comprovar sua influência no desempenho das aplicações. O *benchmark* EigenBench, apresenta características que definem de maneira mais completa uma aplicação transacional, identificando comportamentos mais específicos. Entretanto, a impossibilidade de derivação de 3 das 8 características Eigen impossibilitou uma análise mais detalhada dessas características transacionais, que tiveram grande importância no desempenho das aplicações.

A principal contribuição deste capítulo, além da validação dos dados encontrados em [58], que oferece resultados similares aos apresentados neste trabalho, é a comprovação a respeito do desempenho das aplicações da Seção 6.2 através de modificações nas características transacionais das aplicações com resultados não satisfatórios. Esse é um importante passo para entender melhor o comportamento das aplicações transacionais. Quanto mais for entendido o comportamento das aplicações, mais será possível aperfeiçoar os sistemas STM e, consequentemente, obter melhores resultados no futuro. Essa contribuição é nova na literatura, visto que poucos trabalhos têm foco no entendimento das aplicações transacionais.

8. Conclusão

Memórias Transacionais fornecem um mecanismo flexível e simples para programação paralela em processadores *multicore*, usando abstração de alto nível, ao contrário dos tradicionais mecanismos de controle de sincronização, tais como *locks*. Através desse mecanismo, o problema de sincronização de dados é passado para o sistema de Memória Transacional, que se torna responsável por garantir o funcionamento correto da aplicação, evitando os principais problemas ao utilizar mecanismos de *locks* (*deadlocks*, condições de corrida, etc.) e, explorando de fato o paralelismo das aplicações.

Como apresentado neste trabalho, existe um aumento de trabalhos de pesquisa sobre Memórias Transacionais. Segundo alguns autores, com o passar dos anos houve um crescimento no interesse de pesquisadores e também de empresas em pesquisas com Memórias Transacionais. Esse interesse deve-se ao fato da simplificação que Memórias Transacionais possibilita aos programadores, além de melhorar a escalabilidade das aplicações.

Por entender a importância de Memórias Transacionais para a programação paralela em processadores *multicore*, este trabalho realizou uma avaliação comparativa de um conjunto de sistemas e aplicações transacionais, buscando apresentar o estado atual da área, compreendendo as perguntas ainda em aberto na comunidade e identificando oportunidades de melhorias nos sistemas de Memórias Transacionais de *Software*.

O resultado da avaliação realizada apresenta diversas contribuições para a comunidade científica, oferencendo um conjunto abrangente de resultados que são de grande importância para o crescimento da área. Como citado anteriormente, apesar de existir diversas propostas atuais de sistemas e algoritmos, existem poucos trabalhos na literatura que apresentam avaliações comparativas e outras abordagens para entender o funcionamento de aplicações e sistemas de Memórias Transacionais.

No primeiro teste realizado (Capítulo 5), foi possível realizar uma comparação completa e imparcial dos principais sistemas STM prospostos na atualidade. Diversas contribuições são apresentadas, como a eficiência do *benchmark* STAMP em avaliar os sistemas atuais. Os resultados apresentados demonstram a necessidade de melhorias nos sistemas STM em geral, visto que todos os sistemas apresentaram desempenho muito inferior com algumas aplicações. O teste apresenta, ainda, o sistema SwissTM como o que melhor desempenhou em todas as aplicações. Além disso, o sistema RSTM apresentou desempenho similar ao SwissTM em diversas aplicações, apesar de perder desempenho na medida que aumenta o número de *cores*. Por fim, identificou-se que para entender o comportamento de uma aplicação transacional é necessário caracterizar a aplicação de maneira mais detalhista, o que não ocorre com este *benchmkark*. A comparação dos resultados e as características transacionais das aplicações apresentaram discrepância e não foi possível realizar uma análise mais detalhada.

No segundo teste realizado (Capítulo 6), foi explorada uma nova maneira de caracterizar aplicações transacionais, o que contribuiu positivamente para uma análise mais detalhada do comportamento das aplicações. Foi possível realizar, através das caracaterísticas Eigen, observações sobre o

comportamento de cada aplicação testada. Além disso, foram apresentadas algumas constatações gerais como, o excesso de memória nas aplicações, que influenciou consideralvemente no desempenho. Por fim, foi utilizado o RSTM no *benchmark* EigenBench, para comparar novamente os resultados com o sistema SwissTM. Esse suporte adicional foi importante para novamente observar a diferenças entre os sistemas e o comportamento do sistema RSTM com o *benchmark* EigenBench. O sistema SwissTM apresenta novamente melhor desempenho na maioria das aplicações.

No terceiro e último teste (Capítulo 7), foram apresentadas modificações nas características transacionais das aplicações, comprovando as observações realizadas no capítulo anterior. Foram demonstrados na prática, a importância de entender as características transacionais das aplicações utilizadas. Essa contribuição é nova na literatura, visto que poucos trabalhos têm foco no entendimento das aplicações transacionais.

Em resumo, as principais contribuições deste trabalho são:

1. Avaliação imparcial e completa dos principais sistemas STM propostos na literatura, utilizando o *benchmark* para Memórias Transacionais mais completo e mais utilizado na literatura;

2. Identificação do sistema STM com melhor desempenho da literatura através de comparação completa utilizando 2 *benchmarks* importantes da área de Memórias Transacionais;

3. Análise detalhada das principais características transacionais do *benchmark* STAMP, identificando e comprovando as características mais relevantes nas aplicações do *benchmark*;

4. Identificação de melhorias nos sistemas STM atuais, obtidos através de análise dos resultados obtidos nos testes de desempenho com o conjunto de sistemas e aplicações STM.

Através do conjunto de testes realizados, foi possível entender melhor o estado atual da área de Memórias Transacionais. Apesar das contribuições apresentadas neste trabalho, bem como alguns trabalhos propostos na literatura, é preciso desenvolver mais trabalhos com foco no entendimento geral de Memórias Transacionais. Como oportunidade para o futuro, é possível estender os testes deste trabalho utilizando algum mecanismo de *tracing* como o proposto em [36]. O mecanimo de *tracing* associado ao conjunto de sistemas e aplicações transacionais utilizados neste trabalho poderão investigar as execuções em camadas mais baixas do sistema e assim identificar problemas difíceis de observar em resultados convencionais. Apesar do *overhead* criado, a utilização de mecanismos de *tracing* irá complementar os resultados obtidos, identificando oportunidades de melhoria nos sistemas STM de maneira mais detalhista.

Referências

[1] A. Dragojević, P. Felber, V. Gramoli e R. Guerraoui. 'Why STM Can Be More Than a Research Toy', *Commun. ACM*, vol. 54-4, Abril 2011, pp. 70–77.

[2] A. Dragojević, R. Guerraoui e M. Kapałka. 'Stretching Transactional Memory', In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2009, pp. 155–165.

[3] A. Dragojević, R. Guerraoui e M. Kapałka. 'Dividing Transactional Memories by Zero', In: Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2008.

[4] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, e M. L. Scott. 'An Integrated Hardware-Software Approach to Flexible Transactional Memory.', In: Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), 2007, pp. 104–115.

[5] A. Shriraman, V. Marathe, S. Dwarkadas, M. Scott, D. Eisenstat,C. Heriot, W. N. Scherer III e M. F. Spear. 'Hardware Acceleration of Software Transactional Memory', Relatório Técnico, Computer Science Department, University of Rochester, 2006, 10p.

[6] B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. C. Minh e B. Hertzberg. 'McRT-STM: A High Performance Software Transactional Memory System for a Multicore Runtime', In: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP, 2006, pp. 187–197.

[7] C. Blundell, E. C. Lewis e M. K. Martin. 'Deconstructing Transactional Semantics: The Subtleties of Atomicity', In: Proceedings of the 4th Workshop on Duplicating, Deconstructing and Debunking (WDDD), 2005, pp. 48–55.

[8] C. Cascaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, e S. Chatterjee. 'Software Transactional Memory: Why Is it Only a Research Toy?', *Queue*, vol. 6-5, Setembro 2008, pp. 46–58.

[9] C. C. Minh, J. Chung, C. Kozyrakis e K. Olukotun. 'STAMP: Stanford Transactional Applications for Multi-Processing', In: Proceedings of The IEEE International Symposium on Workload Characterization (IISWC), 2008, pp. 35–46.

[10] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis e K. Olukotun. 'An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees', In: Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA), 2007, pp. 69–80.

[11] C. Ferri, S. Wood, T. Moreshet, R. Bahar e M. Herlihy. 'Embedded-TM: Energy and Complexity-Effective Hardware Transactional Memory for Embedded Multicore Systems', *J. Parallel Distrib. Comput*, vol. 70-10, Outubro 2010, pp. 1042–1052.

[12] C. Ferri, T. Moreshet, I. Bahar, L. Benini e M. Herlihy. 'A Hardware/Software Framework for Supporting Transactional Memory in a MPSoC Environment', *SIGARCH Comput. Archit. News*, vol. 35-1, Março 2007, pp. 47–54.

[13] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. F. Leiserson e S. Lie. 'Unbounded Transactional Memory', In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005, pp. 316–327.

[14] C. Thacker. 'Hardware Transactional Memory for Beehive', Capturado em: http://research.microsoft.com/en-us/um/people/birrell/beehive/hardware transactional memory for beehive3.pdf, Dezembro 2011.

[15] C. Y. Lee. 'An Algorithm for Path Connections and its Applications', *Eletronic Computers*, vol. 10-3, Setembro 1961, pp. 346–365.

[16] D. Dice, O. Shalev e N. Shavit. 'Transactional Locking II', In: Proceedings of the 20th International Symposium on Distributed Computing (DISC), 2006, pp. 194–208.

[17] F. Chen, L. Hongwei, W. Xiaoqun, W. Dongxin e Y. Xiaozong. 'Hardware Transactional Memory in Multicore Processors', In: Proceedings of International Conference on Information Engineering and Computer Science, 2009, pp. 1 - 4.

[18] F. Rui e L. G. Fernandes. 'Avaliação de Bibliotecas STM com Benchmark STAMP', Relatório Técnico, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2011, 12p.

[19] F. Zyulkyarov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur e M. Valero. 'Discovering and Understanding Performance Bottlenecks in Transactional Applications', In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT, 2010, pp. 285–294.

[20] G. Kestor, V. Karakostas, O. Unsal, A. Cristal, I. Hur e M. Valero. 'RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems', In: Proceedings of the 2nd joint WOSP/SIPEW International Conference on Performance Engineering (ICPE), 2011, pp. 335–346.

[21] Intel Corporation. 'Intel C++ STM Compiler Prototype Edition 4.0', Capturado em: http://http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/, Dezembro 2011.

[22] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis e K. Olukotum. ‘The Common Case Transactional Behavior of Multithreaded Programs’, In: Proceedings of the 12th International Conference on High-Performance Computer Architecture (HPCA), 2006, pp. 266–277.

[23] J. Larus e C. Kozyrakis. ‘Transactional Memory: Is TM the Answer for Improving Parallel Programming?’, *Commun. ACM*, vol. 51-7, Julho 2008, pp. 80–83.

[24] J. Lourenço, R. Dias, J. Luís, M. Rebelo e V. Pessanha. ‘Understanding the Behavior of Transactional Memory Applications’, In: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD), 2009, pp. 1–9.

[25] J. Mankin, D. Kaeli e J. Ardini. ‘Software Transactional Memory for Multicore Embedded Systems’, In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), 2009, pp. 90–98.

[26] J. Ruppert. ’A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation’, *J. Algorithms*, vol. 18-3, Maio 1995, pp. 548–585.

[27] K. Jarvis. ‘Transactional Data Structures’, Tese de Doutorado, University of Manchester, 2011, 232p.

[28] K. P. Eswaran, J.N. Gray, R. A. Lorie e I. L. Traiger. ‘The Notions of Consistency and Predicate Locks in a Database System’, *Commun. ACM*, vol. 19-11, Novembro 1976, pp. 624–633.

[29] L. Ceze, J. Tuck, J. Torrellas e C. Cascaval. ‘Bulk Disambiguation of Speculative Threads in Multiprocessors’, In: Proceedings of the 33th Annual International Symposium on Computer Architecture (ISCA), 2006, pp. 227–238.

[30] L. Dalessandro, D. Dice, M. Scott, N. Shavit e M. F. Spear. ‘Transactional mutex locks’, In: Proceedings of the 16th International Euro-Par Conference on Parallel processing: (Euro-Par), 2010, pp. 2–13.

[31] L. Dalessandro, M. F. Spear e M. L Scott. ‘NOrec: Streamlining STM by Abolishing Ownership Records’, In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2010, pp. 67–78.

[32] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis e K. Olukotun. ‘Transactional Memory Coherence and Consistency’, In: Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA), 2004, pp. 102–114.

[33] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham e I. Watson. 'Lee-TM: A Non-Trivial Benchmark for Transactional Memory', In: Proceedings of the $8^(th)$ International Conference on Algorithms and Architectures for Parallel Processing, (ICA3PP), 2008, pp. 196–207.

[34] M. Ansari, K. Jarvis, C. Kotselidis, M. Lujan, C. Kirkham e I. Watson. 'Profiling Transactional Memory Applications', In: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009, pp. 11–20.

[35] M. Castro e A. Degomme. 'Transactional Memory: State of the Art and Trends', Relatório Técnico, LIG Laboratory, 2009, 45p.

[36] M. Castro, K. Georgiev, V. Marangozova-Martin, J. F Mehaut, L.G. Fernandes e M. Santana. 'Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures', In: Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2011, pp. 199–206.

[37] M. F. Spear, L. Dalessandro, V. J. Marathe e M. L. Scott. 'JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory', In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2009, pp. 141–150.

[39] M. F. Triola. 'Introdução à Estatística'. LTC, 2005, 656p.

[40] M. Moir. 'Practical Implementations of Non-Blocking Synchronization Primitives', In: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC), 1997, pp. 219–228.

[41] M. Herlihy. 'Wait-Free Synchronization', *ACM Trans. Program. Lang. Syst.*, vol. 13-1, Janeiro 1991, pp. 124–149.

[42] M. Herlihy. 'A Methodology for Implementing Highly Concurrent Data Objects', *ACM Trans. Program. Lang. Syst.*, vol. 15-5, Novembro 1993, pp. 745–770.

[43] M. Herlihy e J. Moss. 'Transactional Memory: Architectural Support for Lock-Free Data Structures', In: Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA), 1993, pp. 289–300.

[44] M. Herlihy, V. Luchangco, M. Moir e W. Scherer. 'Software Transactional Memory for Dynamic-Sized Data Structures', In: Proceedings of the 22th Annual Symposium on Principles of Distributed Computing (PODC), 2003, pp. 92–101.

[46] M. Quentin e P. Frédéric. 'Lightweight Transactional Memory Systems for Large Scale Shared Memory MPSoCs', In: Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference, 2007, pp. 432–435.

[47] N. Juristo e A. M. Moreno. 'Basics of Software Engineering Experimentation'. Kluwer Academic Publishers, 2001, 395p.

[48] N. Shavit e D. Touitou. 'Software Transactional Memory', In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, 1995, pp. 204–213.

[49] N. Sonmez, O. Arcas, O. Pflucker, O. S. Unsal, A. Cristal, I. Hur, S. Singh e M. Valero. 'TMBox: A Flexible and Reconfigurable 16-core Hybrid Transactional Memory System', In: Proceedings of the 19th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011, pp. 146–153.

[50] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir e D. Nussbaum. 'Hybrid Transactional Memory', In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006, pp. 336–346.

[51] P. Felber, C. Fetzer, U. Mueller, T. Riegel, M. Suesskraut e H. Sturzrehm. 'Transactifying Applications Using an Open Compiler Framework', In: Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing, 2007.

[52] P. Felber, C. Fetzer, e T. 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), 2008, pp. 237–246.

[53] P. Wu, M. M. Michael, C. V. Praun, T. Nabkaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. Mergen, X. Shen, M. F. Spear, H. Y. Wang e K. Wang. 'Compiler and Runtime Techniques for Software Transactional Memory Optimization', *Concurr. Comput. : Pract. Exper.*, vol. 21-1, Janeiro 2009, pp. 7–23.

[54] R. Guerraoui, M. Herlihy e B. Pochon. 'Toward a Theory of Transactional Contention Managers', In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC), 2005, pp. 258–264.

[55] R. Guerraoui, M. Kapałka e J. Vitek. 'STMBench7: A benchmark for Software Transactional Memory', In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), 2007, pp. 315–324.

[57] R. Yoo, N. Yang, A. Welc, B. Saha, A. Adl-Tabatabai e H. Lee. 'Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough', In: Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures (SPAA), 2008, pp. 265–274.

[58] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis e K. Olukotun. 'Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics', In: Proceedings of the International Symposium on Workload Characterization (IISWC), 2010, pp. 1–11.

[59] S. Kumar, M. Chu, C. J. Hughes, P. Kundu e A. Nguyen. 'Hybrid transactional memory', In: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP, 2006, pp. 209–220.

[60] S. Lie and K. Asanovic. 'Hardware Support for Unbounded Transaciontal Memory', Dissertação de Mestrado, Massachusetts Institute of Technology, 2004, 111p.

[61] S. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar e R. Balasubramonian. 'Scalable and Reliable Communication for Hardware Transactional Memory', In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008, pp. 144–154.

[62] T. Harris e K. Fraser. Revocable Locks for Non-Blocking Programming, In: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2005, pp. 72–82.

[63] T. Harris, M. Plesko, A. Shinnar e D. Tarditi. 'Optimizing memory transactions', In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2006, pp. 14–25.

[64] T. Riegel, P. Felber e C. Fetzer. 'A Lazy Snapshot Algorithm with Eager Validation', In: Proceedings of the 20th International Symposium on Distributed Computing (DISC), 2006, pp. 284–298.

[65] U. Drepper. 'Parallel Programming with Transactional Memory', *Queue*, vol. 6-5, Setembro 2008, pp. 38–45.

[66] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III and M. L. Scott. 'Lowering the Overhead of Non-blocking Software Transactional Memory', In: Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), 2006.

[67] W. N. Scherer III e M. L. Scott. 'Advanced Contention Management for Dynamic Software Transactional Memory', In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC), 2005, pp. 240–248.

[68] X. Wang, Z. Ji, C. Fu e M. Hu. 'A Review of Transactional Memory in Multicore Processors', *Information Technology Journal*, vol. 9-1, 2010, pp. 192–200.

A. APÊNDICE

Este apêndice apresenta os resultados dos testes conduzidos ao longo do trabalho. Os testes são separados em seções onde é apresentado os número de execuções de cada teste bem como o intervalo de confiança de cada aplicação. Além de resultados, são apresentados valores intermediários utilizados para cálculo do número de execuções e do intervalo de confiança.

A.1 Teste de Desempenho com *Benchmark* STAMP

As Tabelas A.1 até A.10 apresentam os dados do teste de desempenho do *benchmark* STAMP. As Tabelas A.1, A.2, A.3, A.4, A.5 exibem o número de execuções de cada aplicação além dos dados intermediários utilizados para se alcançar o número de execuções ideal. A Fórmula 4.2 presente no Capítulo 4.2.1 calcula a quantidade de amostras (execuções) mínima para cada aplicação. Já as Tabelas A.6, A.7, A.8, A.9, A.10 exibem o intervalo de confiança de cada resultado. Os demais valores presentes nas tabelas são valores intermediários utilizados na Fórmula 4.3 e na Fórmula 4.4 presentes no Capítulo 4.2.1.

Tabela A.1: Número de execuções sequenciais no teste de desempenho com *benchmark* STAMP

Aplicação	Média	Desvio Padrão	Número de execuções
bayes	37.6833	0.3208	31
Genome	13.0486	0.6331	31
Intruder	24.4200	0.0479	31
Kmeans	16.1419	0.2657	31
Labyrinth	77.2797	1.0277	31
SSCA2	16.8842	0.0957	31
Vacation	21.7300	0.0399	31
YADA	24.5667	0.5461	31

A.2 Teste de Desempenho com *Benchmark* EigenBench

As Tabelas A.11 até A.16 apresentam os dados do teste de desempenho do *benchmark* EigenBench. As Tabelas A.11, A.12, A.13 exibem o número de execuções de cada aplicação além dos dados intermediários utilizados para se alcançar o número de execuções ideal. A Fórmula 4.2 presente no Capítulo 4.2.1 calcula a quantidade de amostras (execuções) mínima para cada aplicação. Já as Tabelas A.14, A.15, A.16 exibem o intervalo de confiança de cada resultado. Os demais valores presentes nas tabelas são valores intermediários utilizados na Fórmula 4.3 e na Fórmula 4.4 presentes no Capítulo 4.2.1.

Tabela A.2: Número de execuções do sistema SwissTM no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Desvio Padrão	Número de execuções
Bayes	2	32.6364	2.2095	76
Genome	2	8.0404	0.3597	31
Intruder	2	25.5978	0.1968	31
Kmeans	2	14.4758	2.2288	77
Labyrinth	2	40.6860	0.0758	31
SSCA2	2	16.4057	0.2184	31
Vacation	2	22.3927	0.2145	31
YADA	2	24.0361	0.5045	31
Bayes	4	31.4052	1.5175	36
Genome	4	4.7664	0.1393	31
Intruder	4	18.1160	0.1362	31
Kmeans	4	7.7598	1.1179	31
Labyrinth	4	22.4047	0.2827	31
SSCA2	4	13.4353	0.0427	31
Vacation	4	14.6996	0.2561	31
YADA	4	20.2529	0.1342	31
Bayes	6	30.3709	2.1591	72
Genome	6	4.1512	0.0799	31
Intruder	6	16.8092	0.1295	31
Kmeans	6	7.3300	1.4354	32
Labyrinth	6	20.5014	0.4233	31
SSCA2	6	13.1429	0.0182	31
Vacation	6	13.5804	0.0635	31
YADA	6	19.8719	0.1255	31
Bayes	8	32.2719	1.9638	60
Genome	8	3.7548	0.0537	31
Intruder	8	16.0017	0.0802	31
Kmeans	8	6.6914	0.7070	31
Labyrinth	8	19.1393	0.3649	31
SSCA2	8	12.8054	0.0595	31
Vacation	8	12.3153	0.0087	31
YADA	8	20.0100	0.1932	31

Tabela A.3: Número de execuções do sistema RSTM no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Desvio Padrão	Número de execuções
Bayes	2	41.1995	1.7265	46
Genome	2	6.6697	0.1006	31
Intruder	2	20.1090	0.0510	31
Kmeans	2	11.9901	1.5782	39
Labyrinth	2	36.7676	0.2074	31
SSCA2	2	16.3916	0.0252	31
Vacation	2	18.7621	0.0824	31
YADA	2	51.2956	1.2825	31
Bayes	4	38.3378	2.1065	69
Genome	4	4.5344	0.0379	31
Intruder	4	15.4246	0.0685	31
Kmeans	4	7.3386	1.2553	31
Labyrinth	4	19.8959	0.3135	31
SSCA2	4	16.6440	0.0742	31
Vacation	4	13.6994	0.0682	31
YADA	4	51.2251	1.0429	31
Bayes	6	40.3864	2.4489	50
Genome	6	4.2209	0.0591	31
Intruder	6	16.2981	0.0291	31
Kmeans	6	6.7536	1.2459	31
Labyrinth	6	20.5313	0.5350	31
SSCA2	6	16.3852	0.0116	31
Vacation	6	14.1481	0.1616	31
YADA	6	52.0203	1.0767	31
Bayes	8	39.1685	2.3547	46
Genome	8	3.9383	0.0446	31
Intruder	8	17.6338	0.0503	31
Kmeans	8	6.8602	0.7922	31
Labyrinth	8	19.5573	0.6557	31
SSCA2	8	16.4257	0.0178	31
Vacation	8	15.1044	0.0602	31
YADA	8	62.4773	1.0716	31

Tabela A.4: Número de execuções do sistema TinySTM no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Desvio Padrão	Número de execuções
Bayes	2	34.1965	0.5707	31
Genome	2	7.9597	0.0988	31
Intruder	2	21.8865	0.0872	31
Kmeans	2	16.4707	3.7271	50
Labyrinth	2	40.4959	0.0849	31
SSCA2	2	16.3273	0.0236	31
Vacation	2	25.1516	0.4460	31
YADA	2	24.7443	0.0809	31
Bayes	4	31.8379	1.6825	44
Genome	4	4.7885	0.0657	31
Intruder	4	14.7170	0.0677	31
Kmeans	4	9.0859	1.3020	31
Labyrinth	4	27.2387	0.3313	31
SSCA2	4	13.3216	0.0377	31
Vacation	4	16.0653	0.0835	31
YADA	4	21.1956	0.0613	31
Bayes	8	34.0699	2.7041	60
Genome	8	5.2246	1.1402	31
Intruder	8	23.7831	0.3189	31
Kmeans	8	7.9447	1.4793	31
Labyrinth	8	21.7605	1.0332	31
SSCA2	8	12.3761	0.0190	31
Vacation	8	13.8474	0.8566	31
YADA	8	21.8608	0.1081	31

Tabela A.5: Número de execuções do sistema TL2 no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Desvio Padrão	Número de execuções
Bayes	2	35.2003	0.9686	31
Genome	2	8.6779	0.1034	31
Intruder	2	39.2532	0.2486	31
Kmeans	2	19.2169	0.7616	31
Labyrinth	2	42.9328	0.1450	31
SSCA2	2	17.8055	0.1333	31
Vacation	2	28.9353	0.1142	31
YADA	2	39.0096	1.8766	55
Bayes	4	34.2096	1.5836	39
Genome	4	5.3791	0.1285	31
Intruder	4	31.5565	0.1282	31
Kmeans	4	11.2933	1.9711	31
Labyrinth	4	28.0224	1.7377	47
SSCA2	4	13.8913	0.0948	31
Vacation	4	17.8664	0.0472	31
YADA	4	29.7412	0.2171	31
Bayes	8	35.1180	0.1507	31
Genome	8	4.2044	0.0487	31
Intruder	8	33.3928	0.1747	31
Kmeans	8	7.7234	0.6228	31
Labyrinth	8	25.1817	1.2443	31
SSCA2	8	12.6170	0.0158	31
Vacation	8	15.1518	0.0290	31
YADA	8	28.7321	0.6124	31

Tabela A.6: Intervalo de confiança das execuções sequenciais no teste de desempenho com *benchmark* STAMP

Aplicação	Média	Valor Crítico	Erro	Intervalo de confiança
Bayes	37.6833	2.042	0.1176	$37.5656 < \mu < 37.8009$
Genome	13.0486	2.042	0.2326	$12.8161 < \mu < 13.2812$
Intruder	24.4200	2.042	0.0176	$24.4024 < \mu < 24.4376$
Kmeans	16.1419	2.042	0.0976	$16.0443 < \mu < 16.2395$
Labyrinth	77.2797	2.042	0.3775	$76.9022 < \mu < 77.6571$
SSCA2	16.8842	2.042	0.0351	$16.8490 < \mu < 16.9193$
Vacation	21.7300	2.042	0.0147	$21.7153 < \mu < 21.7447$
YADA	24.5667	2.042	0.2006	$24.3662 < \mu < 24.7673$

Tabela A.7: Intervalo de confiança das execuções do sistema SwissTM no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Valor Crítico	Erro	Intervalo de confiança
Bayes	2	32.6364	1.992	0.5049	$32.1315 < \mu < 33.1413$
Genome	2	8.0404	2.042	0.1319	$7.9085 < \mu < 8.1724$
Intruder	2	25.5978	2.042	0.0722	$25.5256 < \mu < 25.6700$
Kmeans	2	14.4758	1.992	0.5060	$13.9698 < \mu < 14.9817$
Labyrinth	2	40.6860	2.042	0.0278	$40.6582 < \mu < 40.7138$
SSCA2	2	16.4057	2.042	0.0801	$16.3256 < \mu < 16.4858$
Vacation	2	22.3927	2.042	0.0787	$22.3140 < \mu < 22.4713$
YADA	2	24.0361	2.042	0.1850	$23.8511 < \mu < 24.2211$
Bayes	4	31.4052	2.032	0.5139	$30.8912 < \mu < 31.9191$
Genome	4	4.7664	2.042	0.0511	$4.7153 < \mu < 4.8174$
Intruder	4	18.1160	2.042	0.0500	$18.0661 < \mu < 18.1660$
Kmeans	4	7.7598	2.042	0.4100	$7.3498 < \mu < 8.1698$
Labyrinth	4	22.4047	2.042	0.1037	$22.3010 < \mu < 22.5083$
SSCA2	4	13.4353	2.042	0.0157	$13.4196 < \mu < 13.4509$
Vacation	4	14.6996	2.042	0.0939	$14.6057 < \mu < 14.7935$
YADA	4	20.2529	2.042	0.0492	$20.2036 < \mu < 20.3021$
Bayes	6	30.3709	1.994	0.5074	$29.8635 < \mu < 30.8783$
Genome	6	4.1512	2.045	0.0293	$4.1218 < \mu < 4.1805$
Intruder	6	16.8092	2.045	0.0476	$16.7617 < \mu < 16.8568$
Kmeans	6	7.3300	2.04	0.5176	$6.8123 < \mu < 7.8476$
Labyrinth	6	20.5014	2.045	0.1555	$20.3459 < \mu < 20.6568$
SSCA2	6	13.1429	2.045	0.0067	$13.1362 < \mu < 13.1496$
Vacation	6	13.5804	2.045	0.0233	$13.5571 < \mu < 13.6037$
YADA	6	19.8719	2.045	0.0461	$19.8258 < \mu < 19.9180$
Bayes	8	32.2719	2.004	0.5081	$31.7638 < \mu < 32.7799$
Genome	8	3.7548	2.045	0.0197	$3.7351 < \mu < 3.7746$
Intruder	8	16.0017	2.045	0.0295	$15.9722 < \mu < 16.0312$
Kmeans	8	6.6914	2.045	0.2597	$6.4318 < \mu < 6.9511$
Labyrinth	8	19.1393	2.045	0.1340	$19.0053 < \mu < 19.2733$
SSCA2	8	12.8054	2.045	0.0218	$12.7835 < \mu < 12.8272$
Vacation	8	12.3153	2.045	0.0032	$12.3121 < \mu < 12.3185$
YADA	8	20.0100	2.045	0.0710	$19.9390 < \mu < 20.0810$

Tabela A.8: Intervalo de confiança das execuções do sistema RSTM no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Valor Crítico	Erro	Intervalo de confiança
Bayes	2	41.1995	2.014	0.5127	$40.6868 < \mu < 41.7122$
Genome	2	6.6697	2.042	0.0369	$6.6328 < \mu < 6.7066$
Intruder	2	20.1090	2.042	0.0187	$20.0903 < \mu < 20.1277$
Kmeans	2	11.9901	2.042	0.5160	$11.4740 < \mu < 12.5061$
Labyrinth	2	36.7676	2.042	0.0761	$36.6915 < \mu < 36.8436$
SSCA2	2	16.3916	2.042	0.0092	$16.3823 < \mu < 16.4008$
Vacation	2	18.7621	2.042	0.0302	$18.7319 < \mu < 18.7923$
YADA	2	51.2956	2.042	0.4704	$50.8252 < \mu < 51.7660$
Bayes	4	38.3378	1.997	0.5064	$37.8314 < \mu < 38.8442$
Genome	4	4.5344	2.042	0.0139	$4.5205 < \mu < 4.5483$
Intruder	4	15.4246	2.042	0.0251	$15.3995 < \mu < 15.4497$
Kmeans	4	7.3386	2.042	0.4604	$6.8782 < \mu < 7.7990$
Labyrinth	4	19.8959	2.042	0.1150	$19.7809 < \mu < 20.0109$
SSCA2	4	16.6440	2.042	0.0272	$16.6168 < \mu < 16.6712$
Vacation	4	13.6994	2.042	0.0250	$13.6744 < \mu < 13.7244$
YADA	4	51.2251	2.042	0.3825	$50.8426 < \mu < 51.6076$
Bayes	6	40.3864	2.014	0.6975	$39.6889 < \mu < 41.0839$
Genome	6	4.2209	2.042	0.0217	$4.1992 < \mu < 4.2426$
Intruder	6	16.2981	2.042	0.0107	$16.2874 < \mu < 16.3088$
Kmeans	6	6.7536	2.042	0.4569	$6.2967 < \mu < 7.2105$
Labyrinth	6	20.5313	2.042	0.1962	$20.3351 < \mu < 20.7275$
SSCA2	6	16.3852	2.042	0.0043	$16.3810 < \mu < 16.3895$
Vacation	6	14.1481	2.042	0.0593	$14.0888 < \mu < 14.2074$
YADA	6	52.0203	2.042	0.3949	$51.6254 < \mu < 52.4151$
Bayes	8	39.1685	2.014	0.6992	$38.4693 < \mu < 39.8677$
Genome	8	3.9383	2.042	0.0164	$3.9219 < \mu < 3.9547$
Intruder	8	17.6338	2.042	0.0184	$17.6154 < \mu < 17.6523$
Kmeans	8	6.8602	2.042	0.2905	$6.5696 < \mu < 7.1507$
Labyrinth	8	19.5573	2.042	0.2405	$19.3168 < \mu < 19.7977$
SSCA2	8	16.4257	2.042	0.0065	$16.4192 < \mu < 16.4322$
Vacation	8	15.1044	2.042	0.0221	$15.0823 < \mu < 15.1265$
YADA	8	62.4773	2.042	0.3930	$62.0843 < \mu < 62.8703$

Tabela A.9: Intervalo de confiança das execuções do sistema TinySTM no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Valor Crítico	Erro	Intervalo de confiança
Bayes	2	34.1965	2.042	0.2093	$33.9872 < \mu < 34.4058$
Genome	2	7.9597	2.042	0.0362	$7.9235 < \mu < 7.9960$
Intruder	2	21.8865	2.042	0.0320	$21.8545 < \mu < 21.9185$
Kmeans	2	16.4707	2.014	1.0616	$15.4091 < \mu < 17.5322$
Labyrinth	2	40.4959	2.042	0.0312	$40.4647 < \mu < 40.5270$
SSCA2	2	16.3273	2.042	0.0086	$16.3187 < \mu < 16.3360$
Vacation	2	25.1516	2.042	0.1636	$24.9880 < \mu < 25.3152$
YADA	2	24.7443	2.042	0.0297	$24.7147 < \mu < 24.7740$
Bayes	4	31.8379	2.021	0.5126	$31.3253 < \mu < 32.3505$
Genome	4	4.7885	2.042	0.0241	$4.7644 < \mu < 4.8126$
Intruder	4	14.7170	2.042	0.0248	$14.6922 < \mu < 14.7418$
Kmeans	4	9.0859	2.042	0.4775	$8.6084 < \mu < 9.5634$
Labyrinth	4	27.2387	2.042	0.1215	$27.1172 < \mu < 27.3603$
SSCA2	4	13.3216	2.042	0.0138	$13.3078 < \mu < 13.3354$
Vacation	4	16.0653	2.042	0.0306	$16.0347 < \mu < 16.0959$
YADA	4	21.1956	2.042	0.0225	$21.1731 < \mu < 21.2181$
Bayes	8	34.0699	2.004	0.6996	$33.3703 < \mu < 34.7694$
Genome	8	5.2246	2.042	0.4182	$4.8065 < \mu < 5.6428$
Intruder	8	23.7831	2.042	0.1170	$23.6662 < \mu < 23.9001$
Kmeans	8	7.9447	2.042	0.5426	$7.4022 < \mu < 8.4873$
Labyrinth	8	21.7605	2.042	0.3789	$21.3815 < \mu < 22.1394$
SSCA2	8	12.3761	2.042	0.0070	$12.3691 < \mu < 12.3830$
Vacation	8	13.8474	2.042	0.3142	$13.5332 < \mu < 14.1615$
YADA	8	21.8608	2.042	0.0397	$21.8212 < \mu < 21.9005$

Tabela A.10: Intervalo de confiança das execuções do sistema TL2 no teste de desempenho com *benchmark* STAMP

Aplicação	*cores*	Média	Valor Crítico	Erro	Intervalo de confiança
Bayes	2	35.2003	2.042	0.3552	$34.8450 < \mu < 35.5555$
Genome	2	8.6779	2.042	0.0379	$8.6399 < \mu < 8.7159$
Intruder	2	39.2532	2.042	0.0912	$39.1619 < \mu < 39.3445$
Kmeans	2	19.2169	2.042	0.2793	$18.9372 < \mu < 19.4967$
Labyrinth	2	42.9328	2.042	0.0532	$42.8795 < \mu < 42.9860$
SSCA2	2	17.8055	2.042	0.0489	$17.7565 < \mu < 17.8544$
Vacation	2	28.9353	2.042	0.0419	$28.8933 < \mu < 28.9772$
YADA	2	39.0096	2.009	0.5083	$38.5012 < \mu < 39.5179$
Bayes	4	34.2096	2.042	0.5178	$33.6918 < \mu < 34.7275$
Genome	4	5.3791	2.024	0.0467	$5.3319 < \mu < 5.4263$
Intruder	4	31.5565	2.042	0.0470	$31.5094 < \mu < 31.6036$
Kmeans	4	11.2933	2.042	0.7229	$10.5693 < \mu < 12.0173$
Labyrinth	4	28.0224	2.042	0.5176	$27.5048 < \mu < 28.5400$
SSCA2	4	13.8913	2.014	0.0343	$13.8565 < \mu < 13.9261$
Vacation	4	17.8664	2.042	0.0173	$17.8491 < \mu < 17.8837$
YADA	4	29.7412	2.042	0.0796	$29.6614 < \mu < 29.8209$
Bayes	8	35.1180	2.042	0.0553	$35.0627 < \mu < 35.1733$
Genome	8	4.2044	2.042	0.0179	$4.1865 < \mu < 4.2223$
Intruder	8	33.3928	2.042	0.0641	$33.3287 < \mu < 33.4570$
Kmeans	8	7.7234	2.042	0.2284	$7.4947 < \mu < 7.9521$
Labyrinth	8	25.1817	2.042	0.4563	$24.7247 < \mu < 25.6387$
SSCA2	8	12.6170	2.042	0.0058	$12.6112 < \mu < 12.6228$
Vacation	8	15.1518	2.042	0.0106	$15.1411 < \mu < 15.1624$
YADA	8	28.7321	2.042	0.2246	$28.5072 < \mu < 28.9570$

Tabela A.11: Número de execuções sequenciais no teste de desempenho com *benchmark* EigenBench

Aplicação	*core*	Média	Desvio Padrão	Tamanho da amostra
Genome	2	2.6606	0.0048	31
Intruder	2	1.7542	0.0149	31
Labyrinth	2	11.3363	0.0366	31
Vacation	2	6.3897	0.0025	31
SSCA2	2	5.4528	0.0038	31

Tabela A.12: Número de execuções do sistema SwissTM no teste de desempenho com *benchmark* EigenBench

Aplicação	*cores*	Média	Desvio Padrão	Número de execuções
Genome	2	1.7635	0.0116	31
Intruder	2	1.2836	0.0023	31
Labyrinth	2	4.3832	0.0602	31
SSCA2	2	3.5013	0.0156	31
Vacation	2	5.5121	0.0180	31
Genome	4	0.9445	0.0033	31
Intruder	4	0.8564	0.0058	31
Labyrinth	4	2.3112	0.0431	31
SSCA2	4	1.8884	0.0153	31
Vacation	4	2.8980	0.0080	31
Genome	8	0.7157	0.0050	31
Intruder	8	0.7954	0.0021	31
Labyrinth	8	1.6511	0.0280	31
SSCA2	8	1.5629	0.0061	31
Vacation	8	2.2332	0.0462	31

Tabela A.13: Número de execuções do sistema RSTM no teste de desempenho com *benchmark* EigenBench

Aplicação	*cores*	Média	Desvio Padrão	Número de execuções
Genome	2	1.8052	0.0109	31
Intruder	2	1.2506	0.0124	31
Labyrinth	2	5.6753	0.0208	31
SSCA2	2	3.5079	0.0062	31
Vacation	2	4.0696	0.0229	31
Genome	4	1.2940	0.0108	31
Intruder	4	0.8256	0.0033	31
Labyrinth	4	2.9172	0.0631	31
SSCA2	4	2.1694	0.0047	31
Vacation	4	2.1904	0.0179	31
Genome	8	1.8201	0.0231	31
Intruder	8	0.9241	0.0039	31
Labyrinth	8	2.2935	0.0454	31
SSCA2	8	1.9512	0.0102	31
Vacation	8	1.8865	0.0251	31

Tabela A.14: Intervalo de confiança das execuções sequenciais no teste de desempenho com *benchmark* EigenBench

Aplicação	*cores*	Média	Valor Crítico	Erro	Intervalo de confiança
Genome	2	2.6606	2.042	0.0017	$2.6589 < \mu < 2.6624$
Intruder	2	1.7542	2.042	0.0055	$1.7488 < \mu < 1.7597$
Labyrinth	2	11.3363	2.042	0.0134	$11.3228 < \mu < 11.3497$
SSCA2	2	5.4528	2.042	0.0014	$5.4514 < \mu < 5.4542$
Vacation	2	6.3897	2.042	0.0009	$6.3888 < \mu < 6.3907$

Tabela A.15: Intervalo de confiança das execuções do sistema SwissTM no teste de desempenho com *benchmark* EigenBench

Aplicação	*cores*	Média	Valor Crítico	Erro	Intervalo de confiança
Genome	2	1.7635	2.042	0.0043	$1.7592 < \mu < 1.7677$
Intruder	2	1.2836	2.042	0.0008	$1.2828 < \mu < 1.2845$
Labyrinth	2	4.3832	2.042	0.0221	$4.3611 < \mu < 4.4052$
SSCA2	2	3.5013	2.042	0.0057	$3.4956 < \mu < 3.5071$
Vacation	2	5.5121	2.042	0.0066	$5.5055 < \mu < 5.5187$
Genome	4	0.9445	2.042	0.0012	$0.9432 < \mu < 0.9457$
Intruder	4	0.8564	2.042	0.0021	$0.8542 < \mu < 0.8585$
Labyrinth	4	2.3112	2.042	0.0158	$2.2954 < \mu < 2.3270$
SSCA2	4	1.8884	2.042	0.0056	$1.8828 < \mu < 1.8940$
Vacation	4	2.8980	2.042	0.0030	$2.8950 < \mu < 2.9009$
Genome	8	0.7157	2.042	0.0018	$0.7139 < \mu < 0.7175$
Intruder	8	0.7954	2.042	0.0008	$0.7946 < \mu < 0.7961$
Labyrinth	8	1.6511	2.042	0.0103	$1.6409 < \mu < 1.6614$
SSCA2	8	1.5629	2.042	0.0022	$1.5606 < \mu < 1.5651$
Vacation	8	2.2332	2.042	0.0169	$2.2163 < \mu < 2.2501$

Tabela A.16: Intervalo de confiança das execuções do sistema RSTM no teste de desempenho com *benchmark* EigenBench

Aplicação	*cores*	Média	Valor Crítico	Erro	Intervalo de confiança
Genome	2	1.8052	2.042	0.0040	$1.8012 < \mu < 1.8092$
Intruder	2	1.2506	2.042	0.0045	$1.2461 < \mu < 1.2552$
Labyrinth	2	5.6753	2.042	0.0076	$5.6676 < \mu < 5.6829$
SSCA2	2	3.5079	2.042	0.0023	$3.5057 < \mu < 3.5102$
Vacation	2	4.0696	2.042	0.0084	$4.0612 < \mu < 4.0780$
Genome	4	1.2940	2.042	0.0040	$1.2900 < \mu < 1.2979$
Intruder	4	0.8256	2.042	0.0012	$0.8243 < \mu < 0.8268$
Labyrinth	4	2.9172	2.042	0.0231	$2.8940 < \mu < 2.9403$
SSCA2	4	2.1694	2.042	0.0017	$2.1677 < \mu < 2.1711$
Vacation	4	2.1904	2.042	0.0066	$2.1838 < \mu < 2.1969$
Genome	8	1.8201	2.045	0.0085	$1.8116 < \mu < 1.8286$
Intruder	8	0.9241	2.045	0.0014	$0.9227 < \mu < 0.9256$
Labyrinth	8	2.2935	2.045	0.0167	$2.2769 < \mu < 2.3102$
SSCA2	8	1.9512	2.045	0.0037	$1.9474 < \mu < 1.9549$
Vacation	8	1.8865	2.045	0.0092	$1.8773 < \mu < 1.8957$