UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Curso de Bacharelado em Ciência da Computação

Trabalho de Conclusão de Curso

Estudo de Instruções para Implementação de Suporte à Memória Transacional em Hardware

Frederico Peixoto Antunes

Pelotas, 2022

Frederico Peixoto Antunes

Estudo de Instruções para Implementação de Suporte à Memória Transacional em Hardware

Trabalho de Conclusão de Curso apresentado ao Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Gerson Geraldo H. Cavalheiro
Coorientador: Prof. MSc Fernando Emilio Puntel

Pelotas, 2022

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

A636e Antunes, Frederico Peixoto

Estudo de instruções para implementação de suporte à memória transacional em hardware / Frederico Peixoto Antunes ; Gerson Geraldo Homrich Cavalheiro, orientador ; Fernando Emilio Puntel, coorientador. — Pelotas, 2022.

54 f. : il.

Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) — Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2022.

1. Memória transacional. 2. Transactional synchronization extensions. 3. Stamp. 4. Memória transacional em software. 5. Memória transacional em hardware. I. Cavalheiro, Gerson Geraldo Homrich, orient. II. Puntel, Fernando Emilio, coorient. III. Título.

CDD : 005

Elaborada por Simone Godinho Maisonave CRB: 10/1733

Frederico Peixoto Antunes

Estudo de Instruções para Implementação de Suporte à Memória Transacional em Hardware

Trabalho de Conclusão de Curso aprovado, como requisito parcial, para obtenção do grau de Bacharel em Ciência da Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas.

Data da Defesa: 29 de novembro de 2022

Banca Examinadora:
Prof. Dr. Gerson Geraldo Homrich Cavalheiro (orientador)
Doutor em Informatique Systèmes et Communications pelo Institut National Polytechique de Grenoble.

Prof. Dr. Andre Rauber Du Bois
Doutor em Ciência da Computação pela Heriot-Watt University.

Prof. Dr. Marcelo Schiavon Porto
Doutor em Ciência da Computação pela Universidade Federal do Rio Grande do Sul.

Dedico este trabalho aos meus pais, Leonardo Augusto Antunes e Marge Peixoto, que sempre me apoiaram em todas minhas etapas da vida e sempre me incentivaram a estudar e de ir atrás do conhecimento e da ciência, e ao meu irmão, Henrique Peixoto Antunes, que apesar das implicâncias sempre está do meu lado quando preciso.

AGRADECIMENTOS

Agradeço, em primeiro lugar, o professor Gerson Cavalheiro por escolher me orientar no TCC, e me auxiliar em todas as etapas do trabalho, sempre indo muito além daquilo esperado de um orientador.

Ao meu coorientador no trabalho, Fernando Puntel, que me auxiliou no entendimento de diversas partes das memórias transacionais, e se disponibilizou de me ajudar e dar dicas em diversas partes do trabalho.

Aos professores Marcelo Porto e Andre DuBois, que avaliaram o meu trabalho, e apontaram diversos detalhes importantes para a melhora do trabalho.

Ao grupo GACI, por fornecer acesso à máquina usada para a execução e testes feitos neste trabalho, além de também darem suporte para soluções de problemas encontrados.

A amiga de longa data, Gilda Satte Alam, que se disponibilizou de fazer a revisão da escrita e do português no trabalho, por mais que não tenha havido tempo suficiente para ser feita a revisão completa.

Ao meu psiquiatra, Vinícius Afonso, que me auxiliou em conseguir me organizar e focar na escrita do trabalho, de forma que fosse possível terminar o trabalho a tempo.

E a todos meus amigos e colegas que me ajudaram a chegar neste ponto da minha vida acadêmica.

But there’s no sense crying
Over every mistake
You just keep on trying
Till you run out of cake
And the science gets done
And you make a neat gun
For the people who are
Still alive
— GLaDOS

RESUMO

ANTUNES, Frederico Peixoto. **Estudo de Instruções para Implementação de Suporte à Memória Transacional em Hardware**. Orientador: Gerson Geraldo H. Cavalheiro. 2022. 54 f. Trabalho de Conclusão de Curso (Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2022.

O modelo de Memórias Transacional (TM) oferece recursos interessantes para implementar controle de acesso a dados compartilhados em programas *multithread*. Este modelo de sincronização é oferecido tanto em software, *Software Transactional Memory* ou STM, como em hardware, *Hardware Transactional Memory* ou HTM. Existem diferentes ferramentas que implementam STM, entre as mais conhecidas encontram-se TinySTM e SwissTM. Processadores mais atuais, como alguns processadores da Intel e o IBM POWER8, oferecem, em sua arquitetura, instruções específicas que permitem a implementação de bibliotecas HTM. O uso destas instruções não é trivial (CAI; BLACKBURN; BOND, 2021) e a qualidade da implementação de uma biblioteca HTM depende da consideração de diferentes políticas de implementação dos mecanismos associados à gerência da TM.

No presente trabalho, o estudo consistirá em realizar uma implementação de HTM, utilizando a Transactional Synchronization Extension (TSX) dos processadores Intel e avaliar o desempenho de programas do *benchmark* STAMP com essa implementação. Os resultados de desempenho coletados são comparados com aqueles fornecidos pelo mesmo *benchmark* STAMP quando suportado por uma biblioteca STM. A materialização dos resultados é dada na análise do potencial dos recursos oferecidos pela arquitetura Intel TSX para construção de uma biblioteca verdadeiramente HTM.

Palavras-chave: Memória Transacional. TM. Memória Transacional em Software. STM. Memória Transacional em Hardware. HTM. Extensão de Sincronização Transacionals. TSX. STAMP.

ABSTRACT

ANTUNES, Frederico Peixoto. **Study of Intructions for the Hardware Transactional Memory Support Implementation**. Advisor: Gerson Geraldo H. Cavalheiro. 2022. 54 f. Undergraduate Thesis (Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2022.

The Transactional Memory (TM) model offers interesting resources to implement access control, to shared data, on multithread programs. This synchronization model is offered both in software, *Software Transactional Memory* or STM, as in hardware, *Hardware Transactional Memory* or HTM. There exists different tools that do implement STM, among the most well known there are the TinySTM and SwissTM. Newer processors, as are a few Intel CPUs and the IBM POWER8, offer, in their architecture, specific instructions that allow the implementation of HTM libraries. The use of these instructions is not trivial (CAI; BLACKBURN; BOND, 2021) and the quality of the implementation of a HTM library depends on the consideration of different implementation policies of the mechanisms associated to the TM management.

At the present paper, the study will consist of realizing a HTM implementation, utilizing the Transactional Synchronization Extension (TSX) of the Intel processors, and to evaluate the performance of the STAMP benchmark programs with that implementation. The collected performance results are compared with those provided by the same STAMP benchmark, when supported by a STM library. The results then come to fruition with the analysis of the potential of the resources offered by the Intel TSX architecture for the construction of a trully HTM library.

Keywords: Transactional Memory. TM. Software Transactional Memory. STM. Hardware Transactional Memory. HTM. Transactional Synchronization Extensions. TSX. STAMP.

LISTA DE FIGURAS

Figura 1 Testes de Hipótese para amostras de baixa contenção 41
Figura 2 Testes de Hipótese para amostras de alta contenção 42
Figura 3 Gráficos de tempo de execução para cada aplicação usada 42
Figura 4 Gráficos de speedup para cada aplicação usada 43
Figura 5 Testes de Hipótese para o segundo conjunto de execuções 44

Figura 6 Testes de Hipótese para amostras de baixa contenção 52
Figura 7 Testes de Hipótese para amostras de alta contenção 52
Figura 8 Testes de Hipótese para o segundo conjunto de execuções 52

LISTA DE TABELAS

Tabela 1 Resultados para rtm-bench com configurações padrão 39
Tabela 2 Parâmetros usados na execução do primeiro conjunto de testes . . 40
Tabela 3 Resultados da análise estatística dos tempos de execução no primeiro conjunto testado. 40
Tabela 4 Parâmetros usados na execução do segundo conjunto de testes . . 43
Tabela 5 Resultados da análise estatística dos tempos de execução no segundo conjunto testado. 44

Tabela 6 Resultados da análise estatística dos tempos de execução no primeiro conjunto testado. 53
Tabela 7 Resultados da análise estatística dos tempos de execução no segundo conjunto testado. 54

LISTA DE ABREVIATURAS E SIGLAS

TM	Transactional Memory
STM	Software Transactional Memory
HTM	Hardware Transactional Memory
TSX	Transactional Synchronization Extensions
TSX-NI	Transactional Synchronization Extensions New Instructions
TSXLDTRK	TSX Supend Load Address Tracking
HLE	Hardware Lock Elision
RTM	Restricted Transactional Memory
TAA	TSX Asynchronous Abort
TSX-FA	TSX Force Abort
STAMP	Stanford Transactional Applications for Multi-Processors
ISA	Instruction Set Architecture
TME	Transactional Memory Extension
ASF	Advanced Synchronization Facility
SGL	Single Global Lock
LUPS	Laboratory of Ubiquitous and Parallel Systems
GACI	Grupo de Arquiteturas e Circuitos Integrados

SUMÁRIO

1 INTRODUÇÃO . . . 14
1.1 Objetivos e Resultados Esperados . . . 16
1.2 Metodologia . . . 16
1.3 Organização do Texto . . . 18

2 MEMÓRIAS TRANSACIONAIS . . . 20
2.1 Conceito de Memória Transacional . . . 20
2.2 Memória Transacional em Hardware . . . 23
2.3 Suporte para HTM . . . 26
2.4 Conclusão do Capítulo . . . 29

3 TRABALHOS RELACIONADOS . . . 30

4 IMPLEMENTAÇÃO DA HTM E AVALIAÇÃO COM STAMP . . . 33
4.1 Metodologia . . . 33
4.2 Uso das instruções de hardware . . . 34
4.3 Coleta de Resultados . . . 36

5 RESULTADOS . . . 38
5.1 Validação das Saídas . . . 38
5.2 Análise de Desempenho . . . 39

6 CONCLUSÃO . . . 45
6.1 Trabalhos Futuros . . . 45

REFERÊNCIAS . . . 47

APÊNDICE A REPRODUTIBILIDADE . . . 51

APÊNDICE B RESULTADOS COMPLETOS . . . 52

1 INTRODUÇÃO

A área de programação paralela, concorrente e distribuída tem, desde sua origem, um de seus focos no desenvolvimento de tecnologias que permitam conseguir maior eficiência no código pelo uso de técnicas de programação que explorem estruturas de execução não sequenciais oferecidas pelas arquiteturas de computadores. Como resultado, a execução das instruções definidas por um programa se dá sobre o suporte de diversos fluxos de execução concorrentes. Porém, apesar de todos os avanços na área, nos últimos 30 anos houve uma estagnação nos métodos de implementação de técnicas de sincronização entre fluxos de execução concorrente no acesso a recursos compartilhados, sendo ubíquo o uso de métodos com seção crítica, como exclusão mútua (*mutex*) (HARRIS; LARUS; RAJWAR, 2010). Esses métodos funcionam identificando o início e o fim da seção crítica e adicionando um trancamento (*lock*) e um destrancamento (*unlock*) logo antes e após essa área.

A necessidade desses sistemas de controle dentro da programação paralela ou concorrente é ilustrada no comum exemplo das transferências bancárias, onde uma posição da memória que contém o valor da conta é compartilhado com dois ou mais processos, onde eles devem fazer a leitura do valor, modifica-lo refletindo a operação desejada e, só então, gravar de volta na mesma posição da memória o valor operado. Quando dois fluxos de execução concorrente desejarem fazer transações bancárias simultaneamente, existe a possibilidade de que um irá ler o valor contido na conta, e quando o outro processo for fazer a leitura do valor o primeiro ainda não terá retornado a sua modificação, causando que ao fim da execução a conta só irá representar a transação realizada pelo último processo a completar sua execução, e que a quantidade total de dinheiro existente no sistema tenha mudado. A maior parte dos programas paralelos irão utilizar dados de uma memória compartilhada executando funções muito mais complexas que uma subtração ou adição, portanto também muito mais longas, o que aumenta exponencialmente a probabilidade de conflitos ocorrerem no acesso a memória. Execuções mais longas também implicam que ao usar técnicas de sincronização bloqueantes, ocorrerá mais tempo ocioso nos processos paralelos que necessitam esperar o primeiro processo liberar o acesso à memória compartilhada.

Como uma alternativa que soluciona esse problema, no início da década de 1990 foi proposto o modelo de Memória Transacional (TM) (HERLIHY; MOSS, 1993). As memórias transacionais contornam a ociosidade de execuções paralelas por seguir um princípio de execução especulativa otimista, ou seja, partindo da assunção de que a maioria das execuções paralelas não interferirão entre si e portanto continuam a execução mesmo acessando uma memória compartilhada, porém também ainda garantido a atomicidade dos dados, ou seja, utilizando técnicas de verificação de conflitos para garantir a consistência do resultado final. Esse método de sincronização ainda se propõem a simplificar a implementação de programação concorrente, uma vez que ele se livra da necessidade do controle dos trancamentos e destrancamentos no código e o risco de *deadlocks* (situação onde um travamento, ou *lock*, nunca é destravado e portanto a execução não consegue concluir) e condições de corrida (situação onde uma memória compartilhada é acessada simultaneamente, seja por falha do programador de tranca-la ou por ser destrancada não intencionalmente) na execução.

Os dois trabalhos, publicados na década de 1990, que deram início ao que hoje é conhecido como Memória Transacional, Herlihy; Moss (1993) e Shavit; Touitou (1995), traçaram dois caminhos distintos de implementar o suporte à TMs sendo baseados em software ou em hardware. A Memória Transacional em Software (STM) é a implementação da técnica de TM em bibliotecas, que irão lidar com todo o processo de alocação da memória de cada transação, controle da verificação de conflitos e resolver o tratamento do *fallback* no caso de *aborts*. Ela possui a grande vantagem da acessibilidade, uma vez que não necessita a implementação ou acesso a um hardware com suporte a técnica, também sendo muito mais flexível de modificar de acordo com as suas necessidades e ser integrada em sistemas e linguagens já existentes. A Memória Transacional em Hardware (HTM) é a implementação de suporte diretamente no chip do microprocessador, fornecendo somente um conjunto de instruções de nível de máquina, e toda a implementação, testes e controle sendo feitos direto na unidade especializada do processador. A necessidade de que os processadores usem sua preciosa, e limitada, área de chip para implementar uma técnica que divide a opinião de especialistas (tendo Click e Cai; Blackburn; Bond como exemplos de um detrator e um apoiador da usabilidade da técnica, respectivamente) e, com uma mudança de paradigma de programação, que enfrenta a inércia dos programadores terem que mudar todo seu código para incorporá-la, faz com que o suporte das fabricantes seja uma escolha difícil de justificar. Consequentemente, a existência de poucos chips, advindos de poucas fabricantes, que fornecem o suporte, faz com que essa técnica seja de muito difícil acesso. Ainda assim, ela fornece vantagens consideráveis quando colocada contra a STM, sendo a principal, a eliminação da maior parte dos sobrecustos (*overheads*) necessários para a execução em software, que é um grande limitador na

capacidade de fazer TM ser uma técnica viável para o uso em situações reais.

Então, com esse trabalho, deseja-se validar a usabilidade dessas técnicas, esclarecer o que é necessário para, e como, usar as implementações de suporte à HTM, testar o processo de rodar programas com essas implementações e, ao final, verificar se existem indicações de seu uso se destacar em algum aspecto que aponte qual o futuro das HTMs.

As contribuições deste trabalho são:

- Identificação de computadores com suporte a HTM e instruções TSX
- Expansão sobre o conhecimento/entendimento do funcionamento da TSX após a sua desativação via *microcode*
- Desenvolvimento de uma biblioteca que implementa suporte de HTM na *benchmark* STAMP
- Validação do funcionamento atômico das implementações de memórias transacionais, incluindo o da implementação com TSX desenvolvido para este trabalho

1.1 Objetivos e Resultados Esperados

O trabalho tem como seu principal objetivo identificar o potencial das instruções em hardware TSX (*Transactional Synchronization Extension*), oferecido em algumas famílias da arquitetura Intel, para a construção de uma biblioteca que forneça suporte à Memória Transacional em hardware.

Para alcançar este objetivo, será desenvolvida uma versão do suporte à TMs para o *benchmark* STAMP (MINH et al., 2008), apoiado nas instruções TSX, e, será realizado um estudo do desempenho desta ferramenta, comparando sua performance com a fornecida por opções de bibliotecas STM, aplicadas na mesma *benchmark*.

Busca-se, então, alcançar uma compreensão do potencial existente no uso dos recursos para ser feita a implementação de HTM nas arquiteturas, identificando o potencial destas novas implementações de suporte à HTM no desenvolvimento de novas estratégias para memórias transacionais em hardware.

1.2 Metodologia

Este trabalho tem seu foco na exploração das instruções TSX (*Transactional Synchronization Extension*) oferecida em algumas famílias de processadores fabricados pela Intel. O esforço de programação terá início em identificar os conjunto de instruções oferecidas e como explorá-las diretamente na linguagem C. Estudos de caso da

utilização destas instruções documentados na literatura servirão como base para prototipação de uma biblioteca de serviços para implementação do suporte à HTM sobre programas concorrentes escritos em C.

A validação dos resultados é feita comparando, os resultados obtidos, com os oferecidos pela TinySTM (FELBER; FETZER; RIEGEL, 2008). A TinySTM é uma biblioteca STM amplamente aceita na comunidade da área, justificando sua adoção neste trabalho. Por resultados entende-se tanto analisar a diferença de tempos de execução obtidos como a saída dos programas. No atual estágio de amadurecimento da pesquisa o principal aspecto será observar as diferenças existentes entre as saídas obtidas é de grande relevância para atestar a correção da implementação realizada. A avaliação de resultados em termos de desempenho também é realizada, mas para oferecer dados complementares e direcionar ações futuras.

A avaliação será realizada explorando um dos principais *benchmarks* voltados aos ambientes de programação dotados de Memória Transacional, o *benchmark* STAMP. Este *benchmark* consiste em um conjunto de programas oferecendo diferentes padrões de acesso a dados na Memória Transacional, permitindo verificar um amplo espectro de casos. A validação das saídas com a biblioteca desenvolvida será comparada com a apresentada pela execução com TinySTM.

Embora secundário em termos de objetivo, a análise de desempenho foi conduzida utilizando métodos estatístico para validação dos dados coletados. Neste trabalho, o desempenho foi medido em termos de tempo de execução. O protocolo de validação estatística foi implementado na forma de um notebook no ambiente Google Colab(DI DOMENICO; CAVALHEIRO; LIMA, 2022). Este notebook foi implementado em Python, recebendo como entrada arquivos de dados contendo as amostras de tempo coletadas, sendo capaz de executar testes estatísticos, como Kolmogorov-Smirnov (MASSEY JR, 1951), T de Student (STUDENT, 1908), Shapiro-Wilk (SHAPIRO; WILK, 1965) e U de Mann-Whitney (MANN; WHITNEY, 1947), para validar e comparar os resultados. Os resultados deste notebook são apresentados na forma de gráficos, utilizados no corpo deste texto.

Em relação ao acesso às instruções TSX, cabe destacar que ela não se encontra presente em todos os processadores Intel. Neste trabalho foram utilizadas a plataforma Intel DevCloud, para prototipação e desenvolvimento, e uma máquina local[1] para coleta de tempos. A Intel DevCloud é uma plataforma na nuvem hosteada na nuvem e fornecida pela Intel, com acesso a uma expansiva lista de softwares e os mais recentes sistemas da Intel, que permite facilmente prototipar, testar e executar programas sendo desenvolvidos.

[1] Servidor Dell PowerEdge R440 (Intel Xeon Gold 5118 2.3G, 12C/24T, 10.4GT/s , 16.5M Cache, Turbo, HT (105W) DDR4-2400), identificado como G5 nos no laboratório GACI.

1.3 Organização do Texto

O restante do trabalho de conclusão está organizado da seguinte forma:

O capítulo 2 (Memórias Transacionais) aborda introduzir memórias transacionais de forma mais geral, com um foco teórico, e possui quatro sub seções. A sub seção 2.1 (Conceito de Memória Transacional) trata dos conceitos e princípios da técnica como um todo, além discorrer sobre memórias transacionais em software. Já a sub seção 2.2 (Memória Transacional em Hardware) trata sobre os conceitos teóricos e técnicos das memórias transacionais em hardware, porém sem se aprofundar em uma implementação específica e suas instruções. E a sub seção 2.3 (Suporte à TM em Hardware) foca em apresentar as diferentes implementações da técnica de HTM em sistemas, com o suporte desenvolvido e fornecido por diferentes fabricantes, mas em especial as características e conjunto de instruções da TSX da Intel. Por último a subseção 2.4 (Conclusão do Capítulo) conclui os conceitos de memória transacionais que foram abordados no capítulo, e reflete a relação deles com o trabalho.

O capítulo 3 (Trabalhos Relacionados) traz uma discussão sobre a literatura e o estado da arte da área, expondo trabalhos que se relacionam com este sendo desenvolvido.

O capítulo 4 (Implementação da HTM e Avaliação com STAMP) apresenta a etapa de desenvolvimento, testagem e execução das implementações de memórias transacionais no *benchmark* STAMP, e possui três sub seções. A sub seção 4.1 (Metodologia) trata de se detalhar melhor os elementos e ferramentas que são utilizados no desenvolvimento e execução da HTM implementada na STAMP. A sub seção 4.2 (Uso das instruções de hardware) trata das modificações que foram feitas utilizando as instruções de hardware e o como cada instrução é utilizada e se comporta na implementação. A sub seção 4.3 (Coleta de resultados) detalha os passos tomados para fazer a execução e coleta dos resultados puros, e para fazer a coleta dos resultados da análise estatística.

O capítulo 5 (Resultados) é aonde são apresentados os resultados das execuções e testes feitos no desenvolvimento do trabalho, aqueles obtidos seguindo as métricas e parâmetros delimitados na seção 4.3. Na subseção 5.1 (Validação das Saídas) é apresentado resultados e características que validam o funcionamento apropriado das execuções de memória transacional em hardware. Enquanto na subseção 5.2 (Análise de Desempenho) são apresentadas as tabelas e gráficos dos resultados obtidos com os testes, e é feita uma contextualização e discussão sobre o que o significado dos dados obtidos.

E no capítulo 6 (Conclusão) é feita a análise do desenvolvimento do trabalho, discutindo descobertas, dificuldades e o que se alcançou dos objetivos, também possuindo uma sub seção 6.1 (Trabalhos Futuros) onde se levanta qual o próximo passo para

pesquisa, com as portas que foram abertas por esse trabalho e questões que permanecem após o trabalho.

2 MEMÓRIAS TRANSACIONAIS

Esse capitulo introduz a fundamentação teórica das memórias transacionais, desde sua origem, teoria, métodos de implementação, casos de uso, e em especial é introduzido o elemento central do trabalho, as memórias transacionais em hardware, detalhando as suas características especificas, como se diferencia das outras abordagens de Memória Transacional, e quais são as tecnologias existentes que permitem o seu uso.

2.1 Conceito de Memória Transacional

Dentro da área de pesquisa de Sistemas de Banco de Dados, existe uma técnica que possibilita serem feitas consultas e atualizações no banco de dados (DB), de forma rápida e simples. Ela utiliza sistemas concorrentes e paralelos, sem a necessidade que o programador tenha que lidar de definir, ou fazer o controle, desse processo concorrente. Essa técnica é chamada transação, e funciona por fazer cada computação rodar independente das outras de forma sequencial, representando as alterações feitas no banco de dados, e utilizando algoritmos complexos no nível de sistema, para fazer a integração de todas essas transações no banco de dados. Esse conceito também define um conjunto de quatro propriedades importantes para bancos de dados, comumente conhecidos pelo acrônimo ACID, para Atômico, Consistente, Isolado e Durável (HARRIS; LARUS; RAJWAR, 2010).

Portanto, por muitos anos se debateu a possibilidade do uso de um método similar dentro das linguagens de programação, com (LOMET, 1977) sendo o primeiro a fazer essa observação, com sua proposta de ações atômicas. Entretanto, em seu artigo, Lomet somente demonstrou uma estrutura para construir a técnica quando implementada em SIMULA, sem se alcançar uma implementação com desempenho comparável com as técnicas de sincronização explícita.

As memórias transacionais foram ser apropriadamente definidas com (HERLIHY; MOSS, 1993), onde é feita a proposta de Memória Transacional como uma nova arquitetura de multiprocessadores capazes de fornecer uma implementação de sincro-

nizações não bloqueantes com desempenho competitivo ao de implementações bloqueantes. Neste trabalho foram usadas as terminologias e conceitos originados das transações de banco de dados, porém recontextualizadas para o paradigma de linguagens de programação, nos fornecendo novas definições para alguns deles. Portanto Herlihy; Moss define transação como "uma sequência finita de instruções de máquina, executadas por um único processo, satisfazendo as propriedade de seriação e atomicidade".

Uma diferença chave entre a técnica de transação de banco de dados e transação em linguagens de programação é que a computação feita nos bancos de dados tipicamente envolvem comunicação por rede ou acessos ao disco, enquanto as instruções executadas por um programa armazenam os seus dados em memória (cache), sendo essa a razão do nome dado à memórias transacionais (HARRIS; LARUS; RAJWAR, 2010).

Contudo, com a publicação do artigo de (SHAVIT; TOUITOU, 1995), houve uma nova reestruturação nas definições e nomenclaturas da área de Memória Transacional, sendo solidificados como são conhecidos esses conceitos hoje. A primeira grande mudança é a introdução do conceito de Memória Transacional em Software (STM), e a diferenciação dela da proposta de Herlihy e Moss, que é apontada como "uma engenhosa solução de hardware", mas sem ainda cunhar o nome Memória Transacional em Hardware (HTM), que é como hoje em dia é classificado implementações em hardware como a proposta por eles.

Dessa forma se tem que Memória Transacional, independente se implementada em hardware ou software, é um modelo de controle de concorrência proposto como alternativa ao modelo de seção crítica, o qual continua a ser usado até hoje, com a proposta de fornecer um maior nível de abstração, tentando solucionar o problema da complexidade de trabalhar com o controle de acesso à seção crítica, de forma que garante a consistência da memória compartilhada (JARDIM, 2021).

O seu funcionamento, também, sempre será baseado na definição de um trecho de código atômico executado por um processo, a transação. Durante a sua execução, é alocado um pedaço da memória usado unicamente por uma transação, e é armazenado nele todas as modificações que forem feitos, à valores da memória, por essa transação. O fim da execução da transação aciona uma verificação de conflitos, isto é, um algoritmo fará uma análise se houve alguma transação que esteja executando concorrentemente que tenha feito um acesso ao mesmo dado manipulado, comparando as memórias locais de cada transação. No caso de nenhum conflito ser verificado no teste, a transação é aceita, ou, no linguajar adotado no modelo de Memória Transacional, executa a operação *commit*, indicando que as modificações feitas durante a execução, que foram armazenadas localmente, são tornadas públicas, adicionadas na memória compartilhada. No caso de haver um conflito, é feito uma ação de *abort*,

ou seja, a execução de uma ou mais das transações conflitantes são interrompidas, todas as modificações armazenadas localmente nelas são descartadas, e a execução é reiniciada (RIGO; CENTODUCATTE; BALDASSIN, 2007).

É importante também destacar que, apesar deste ser o funcionamento típico do modelo de Memória Transacional, as ações tomadas em caso de conflito são dependentes de uma heurística que pode ser definida pelo programador. As políticas de um gerenciador de contenção são capazes de estabelecer se a verificação de conflitos deve ser feita cada vez que for feito um acesso à memória ou somente no fim da execução (adiantada ou preguiçosa) (JARDIM, 2021), decidir qual transação deve ser abortada na existência de conflito (a atacante, que verificou o conflito, ou a vítima, que possui conflito com a primeira) (DRAGOJEVIĆ; GUERRAOUI; KAPALKA, 2009), definir qual a ação específica performada por uma transação quando abortada (somente reexecutar, utilizar uma estratégia de *backoff* adaptativo, recorrer à sincronização com uso de um *lock* global, ou outra), e até utilizar técnicas mais sofisticadas para determinar o seu comportamento em diferentes conflitos. O uso de diferentes estratégias na gerência de contenção é importante pois ela tem uma relação direta com o desempenho de sistemas transacionais, podendo diminuir a ocorrências de *aborts* desnecessários, auxiliar transações mais importantes concluírem e, o talvez mais importante aspecto, garantir o avanço (*forward progress*) da execução.

Retornando à teoria de memórias transacionais em software, após serem esclarecido esses conceitos, agora é possível melhor explicar o seu funcionamento. As STMs são completamente implementadas em software, sendo capazes de serem usadas em processadores convencionais. Em software também é obtida uma maior flexibilidade na implementação das estratégias de gestão dos conflitos, permitindo que se use algoritmos mais sofisticados no controle de contenção, e que se suporte transações irrestritas (transações não limitadas por tamanho ou tempo, que causam os *aborts* de capacidade) (CAI; BLACKBURN; BOND, 2021). Porém, este tipo de implementação incorre em sobrecustos de execução não negligenciáveis, causados pela necessidade de executar diferentes operações para manutenção da Memória Transacional, como alocação da memória, acessos de leitura e escrita e controle de contenção sobre as transações. Esse impacto negativo, no desempenho, causado pelo *overhead*, é observado por Diegues; Romano; Rodrigues, ao ser feita a comparação de STMs com métodos convencionais, baseados em semáforos.

Para mitigar o impacto do sobrecusto de execução do suporte a STM no desempenho final, muito tem se pesquisado nos últimos anos. Em alguns casos, a literatura indica ter sido possível melhorar o desempenho para níveis razoáveis, em especial em sistemas que possuem compiladores otimizadores integrados, desenvolvendo interfaces que focam em diferentes características. Exemplos de ferramentas no estado da arte da implementação em software de Memória Transacional: uma in-

terface otimizada para cenários de alta contenção (SwissTM (DRAGOJEVIĆ; GUERRAOUI; KAPALKA, 2009)) e uma otimizada para reduzir custos de instrumentação (TinySTM (FELBER; FETZER; RIEGEL, 2008)) (DIEGUES; ROMANO; RODRIGUES, 2014). Mesmo com esses avanços, ainda não é claro se a melhora para um nível prático é possível sem suporte de HTM (LARUS; RAJWAR, 2007).

A implementação da Memória Transacional em Hardware é objeto de discussão na Seção 2.2.

Ainda há mais uma técnica de implementação de memórias transacionais, que pode ser vista como uma terceira abordagem de implementação, ainda não mencionada, que é as memórias transacionais híbridas (HyTM). A característica fundamental dela é de que as transações são feitas utilizando um conjunto único de primitivas que são traduzidas pelo compilador, tanto para instruções de STM, quanto de HTM, e o *contention manager* faz a troca da execução em hardware para software quando encontra transações maiores (RIGO; CENTODUCATTE; BALDASSIN, 2007)(DRAGOJEVIĆ; GUERRAOUI; KAPALKA, 2009). Ela não é comumente utilizada, e, como mostrado em Diegues; Romano; Rodrigues, com os métodos atuais elas possuem resultados muito inferiores a implementações inteiramente feitas em software ou hardware.

2.2 Memória Transacional em Hardware

Diferentemente de quando é falado de STMs, na implementação das memórias transacionais em hardware, ou HTM, é imprescindível que o processador tenha uma arquitetura que disponibilize suporte à essa técnica, tendo em vista que o conceito gira em volta da remoção de overheads fazendo partes da execução diretamente em uma unidade especializada que será mais rápida pra task especifica, não gastará recursos do processamento, e terá um acesso mais direto à memória (podendo também diretamente se aproveitar das estratégias de manipulação e substituição da memória, já disponíveis na arquitetura do processador).

Ter essa unidade em hardware, que implementa as partes fundamentais da Memória Transacional, garante que o desempenho das execuções irão depender significantemente menos de otimizações do compilador para se obter desempenho desejável, além de garantirem, também, um maior isolamento das transações, sem que seja necessário modificar o comportamento de acesso à memória em partes não transacionais do código. Entretanto, com essa dependência da implementação em hardware, se perde a flexibilidade fornecida em software que permite facilmente iterar, corrigir e evoluir o funcionamento da memória transacional, e que resulta em implementações mais sofisticadas e especializadas para cada situação.

O suporte a HTM deve, também, fornecer um conjunto de instruções de máquina

(um ISA) que permita que o software se comunique com o processador, para se fazer o uso das funções implementadas. Essas instruções são responsáveis pela primeira camada de abstração disponível, em um computador, para acessar o processador, com as instruções representando um, ou uma série de, opcode da ação, seguido pelo valor de um operando explícito. Cada processador possuirá um conjunto próprio de instruções, com semântica e funcionamento próprio (outra característica que dificulta para desenvolvedores implementarem suporte delas em seus sistemas), e existem duas categorias que englobam as instruções de HTM: explicitamente transacionais e implicitamente transacionais (HARRIS; LARUS; RAJWAR, 2010).

Implementações de HTM explicitamente transacionais fornecem instruções para indicar o início e fim da região transacional do código, bem como também fornecendo instruções que indicam quais acessos à memória serão transacionais, assim tendo que as regiões da memória acessadas durante uma transação sem ser indicada como tal, não seguirão os protocolos de Memória Transacional. Já as implementações implicitamente transacionais somente necessitam o uso de instruções que indicam os limites da região transacional, com todos acessos à memória feitos dentro dela sendo tratados como transacionais. Transações explícitas fornecem uma flexibilidade maior na programação de TMs, permitindo diminuir os conjuntos de leituras e escritas na Memória Transacional, e reduzindo o risco de *aborts* de capacidade (*aborts* causados pelo despejo de uma transação da cache), mas, também, aumentando a complexidade do seu uso. Enquanto isso, transações implícitas são muito mais fáceis de se trabalhar, e permitem o uso de bibliotecas já existentes, sem modificações, portanto sendo preferida, e a abordagem existente na grande maioria das implementações atuais. Diversas dessas implementações atuais também expandem as transações implícitas, fornecendo instruções que permitem identificar trechos não transacionais dentro das transações. Nestes casos, tendo partes do código, dentro da transação, que não serão adicionados ao conjunto de leitura na memória da transação. A extensão TSXLDTRK, da Intel, é um exemplo de uma expansão como essa, em transações implícitas.

Outra característica, que funciona de forma completamente diferente em HTMs e STMs, é o armazenamento dos conjuntos de leitura e escrita, das transações, na cache. Processadores atuais dispõem de diversas técnicas avançadas de controle da cache, técnicas que exploram características de localidade espacial e temporal da memória, e auxiliam a otimização da quantidade de *commits* alcançados pelas memórias transacionais. Uma alternativa, para aumentar ainda mais a taxa de sucessos, seria o uso de uma cache especializada somente para transações, como foi proposto no trabalho de (HERLIHY; MOSS, 1993). Porém, modificações nas estruturas de memória de processadores são de extrema complexidade e, portanto a expansão da memória principal é a técnica mais comumente aplicada no projeto das placas. Similarmente, a implementação de HTMs irrestritas, ou seja, HTMs em que as transações não são afe-

tadas pelas limitações físicas de armazenamento, necessitam mudanças substanciais aos subsistemas de cache e memória e, portanto, não existindo até então nenhum suporte comercial delas (CAI; BLACKBURN; BOND, 2021).

A utilização dessas estruturas de memória, dos processadores atuais, também necessitam cuidado adicional nas HTMs, pois é necessário garantir que a única cópia do conjunto de escrita não se perca, e, portanto, sendo necessário que linhas de cache suja sejam copiadas para a memória principal antes de serem sobrescritas, para que possam ser recuperadas em casos de *abort*.

Usualmente, as HTMs também possuem mais um nível de abstração acima das instruções de máquina, uma abstração no nível de software, que é adicionada na compilação. Essas abstrações irão fornecer as instruções que são usadas diretamente pelo programador, em seu código, e são normalmente disponibilizadas por intrínsecas. Intrínsecas sendo funções de uma biblioteca que o compilador implementa nativamente, sem a necessidade da sua declaração explícita . O GCC disponibiliza instruções de memórias transacionais, usando as intrínsecas das instruções TSX, fornecido no conjunto de intrínsecas da Intel.

Implementações de HTM ainda enfrentam outro grande desafio, causado pelo princípio de atomicidade garantido nas transações: elas não possuem garantia de progresso quando não fornecidas com alternativas em que as transações são garantidas sua execuções isoladas. Em outras palavras, as transações podem acabar indefinidamente bloqueadas por causa de padrões repetitivos que causam *aborts*, um processo conhecido como inanição (*starvation*). E sem que o controle de contenção forneça uma saída alternativa, ou *fallback*, não se tem como garantir que a execução irá progredir. Um *fallback* comumente fornecido em implementações de HTM é o uso de bloqueios globais (SGL) para uma data transação após um número X de tentativas que resultaram em *aborts* (RIGO; CENTODUCATTE; BALDASSIN, 2007).

Com todos esses fatores que devem ser considerados no projeto de memórias transacionais, quando adicionado em um *chipset*, além de todas as considerações com limitações que vão no projeto de qualquer processador (tamanho, consumo energético, dissipação de calor), e com a usabilidade comercial ainda não bem definida, faz com que o mercado de plataformas que suportem HTM seja ainda muito pequeno.

Por muitos anos, HTMs só eram disponíveis em processadores customizados, como os utilizados pela Azul Systems no início dos anos 2000 (CLICK, 2019), na tentativa do uso de elisão automática dos *locks* do Java. Acesso comercial a processadores com HTM só ocorreu em 2010, quando a IBM lançou o zEnterprise EC12, seguido, em anos subsequentes, pelo Blue Gene/Q e os processadores POWER8 e POWER9 (NAKAIKE et al., 2015) (IBM, 2019). Mesmo havendo esse advento do acesso comercial, ele ainda ficou relegado ao mercado de supercomputadores. Por volta de 2010, Sun Microsystems e AMD também demonstraram ter projetos de suporte a HTM em

desenvolvimento, com os processadores Rock e o conjunto de instruções ASF, mas ambas nunca lançando oficialmente processadores com o suporte (DICE et al., 2009) (AMD, 2009). A acessibilidade de HTMs só se tornou realmente possível em 2013, quando a Intel lançou a Extensão de Sincronização Transacional (TSX), sua tecnologia de HTM, em seus processadores *commodity*, concebendo os chamados HTMs *commodity* (DIEGUES; ROMANO; RODRIGUES, 2014). Um processador *commodity* é, como definido por (GRAHAM; SNIR; PATTERSON, 2004), aquele projetado para um mercado amplo e produzido em larga escala, em contraste com processadores customizados.

2.3 Suporte para HTM

O uso de HTM necessita de um hardware que possua implementado em sua arquitetura suporte à técnica, fornecendo um conjunto de instruções em linguagem de máquina que permita indicar ao hardware dedicado para mudar para um modo de execução diferente, ou retornar informações específicas. E como mencionado ao fim da subseção anterior, existiram alguns esforços de algumas empresas diferentes, mas a grande maioria falhou ou acabou nem saindo do papel, muito pela postura tímida das fabricantes de não investir e disponibilizar mais amplamente a técnica. Apesar disso, a Intel durante a década de 2010 decidiu ir contra a tendência das outras empresas, ofertando recursos para exploração na área de memória transacionais em hardware em diversos de seus processadores comerciais. Com essa oferta, a Intel disponibilizou uma extensão do conjunto de instruções, a *Transactional Synchronization Extension*.

A extensão TSX fornece dois conjuntos de instruções HTM, os quais oferecem princípios de implementação diferentes, a Hardware Lock Elision (HLE) e a Restricted Transactional Memory (RTM). Em algumas situações, esse conjunto de instruções também é chamado de TSX-NI, onde NI significa *New Instructions*, com TSX-NI sendo uma referência especificamente a RTM, que é o conjunto das TSXs que introduz novas instruções (INTEL, 2020).

A HLE, *Hardware Lock Elision*, se propõe como um conjunto de instruções de método retro compatível com programas e sistemas que não possuem suporte à HTM, usando a técnica de elisão. Esse conjunto possui apenas duas instruções, XAQUIRE e XRELEASE, que na implementação devem ser adicionadas imediatamente antes da instrução que fornece o bloqueio a uma seção crítica, e imediatamente após a instrução que libera o bloqueio. Assim, quando a HLE é disponibilizada pelo sistema, o programa evade os bloqueios, criando uma transação. Nos casos em que o sistema não possui o suporte, as instruções somente serão ignoradas pelo programa, e entrarão na região crítica normalmente. A não elisão do bloqueio também ocorre nas situações em que o sistema determina que a transação deve usar o *fallback*, ou seja,

em que ela deve seguir o trajeto alternativo em que a execução é sequencial.

A RTM, *Restricted Transactional Memory*, é um conjunto de instruções um pouco mais avançado e que, diferentemente da HLE, permite mais liberdade nas definições das transações, ficando sob responsabilidade do programador determinar o controle de contenção e o código de *fallback*, que será usado alternativamente a transação quando, após uma detecção de *abort*, o controlador de contenção definir o fim da transação.

Em 2018, a Intel divulgou que havia sido identificada uma falha de segurança no sistema de memória do processador (INTEL, 2021). Com uma atualização de *microcode* das arquiteturas provendo o recurso TSX, foi desabilitado o suporte de HLE em inúmeros processadores, sendo também implementada uma medida de mitigação no sistema da RTM que severamente afetava o desempenho e número de *aborts* da técnica. Esta nova versão do TSX foi denominada TSX Force Abort, ou TSX-FA (COSTA, 2020). Porém, essa não foi a única vulnerabilidade identificada no sistema de memória das HTMs da Intel. Em 2019 foi divulgada a vulnerabilidade TSX Asynchronous Abort (TSX AA) (INTEL, 2019). Essa vulnerabilidade fez com que eventualmente em 2021 a Intel publicasse um novo microcode, dessa vez desabilitando por completo o suporte da HLE de todos os processadores e deprecando a tecnologia. Nesta medida, o RTM foi também desabilitado por padrão, sendo, no entanto, possível reabilitá-lo com a modificação de algumas flags MSR adicionadas pela Intel com a atualização (INTEL, 2021). A Intel lançou um novo microcode em março de 2022, expandindo algumas das medidas tomadas em 2021 (INTEL, 2022), e também com o Kernel do Linux adicionando medidas próprias de mitigação, os quais podem ser desabilitados com o uso de comandos específicos oferecidos em nível de kernel (LINUX, 2019).

O trabalho já possuía o interesse da implementação com o uso das instruções RTM, e após essas descobertas, o foco foi mudado para ser totalmente nas RTMs. Essa, que apesar de também estar atualmente num limbo, ainda parece ser o foco da Intel para desenvolvimentos futuros.

Portanto, aprofundando o conhecimento do funcionamento das RTMs, vê-se que ela possui três principais instruções: XBEGIN, XEND, and XABORT (com XTEST sendo uma instrução adicional – também disponível na HLE – que consulta se o processadores está executando alguma transação no momento). Adicionalmente ela possui sete códigos de status, que são representados pelo estado de diferentes bits do registrador EAX.

A instrução intrínseca _xbegin inicia uma transação e retorna o status dela. Caso seja o primeiro xbegin a ser executado, ele também é a instrução que indicará para o processador entrar em modo de execução de transações. O funcionamento dessa função intrínseca é idêntico a uma execução em código *assembly* de move de 0xFFFF para o registrador EAX, a chamada da instrução de máquina xbegin com um pulo para

uma *label* definida na linha seguinte, e por último passando o valor do EAX para uma variável status. A criação da *label* é feita para que o status possa ser modificado e retornado pelas instruções xend e xabort como será descrito em breve.

A instrução intrínseca _xend encerra a transação mais recente/mais interna e, para o caso de o comando ser o último/mais externo xend, ele finaliza a execução de transações e automaticamente é feita a tentativa de *commit* dos valores das transações. No caso de falha no *commit*, o processador irá descartar todas modificações feitas na memória e nos registradores durante a execução, o registrador EAX será atualizado com um valor que indica o tipo de erro que causou o *abort*, e o processador irá usar o endereço/*label* do primeiro xbegin como endereço de *fallback* para retomar a execução. A modificação do EAX e o pulo pro *label* do xbegin garante que o status seja atualizado para visualizar que ocorreu um *abort* e o que causou ele. Em código *assembly* essa instrução somente faz a chamada da instrução xend de baixo nível.

A instrução intrínseca _xabort fará um *abort* forçado na execução de transações. A chamada da função xabort deve receber um argumento de tipo constante, imm, e fará com que o registrador EAX seja atualizado para indicar a causa do *abort* como sendo pela chamada da instrução, com os 8 bits menos significativos do registrador recebendo o valor do argumento imm8, um byte de controle imediato. Igualmente a um *abort* natural, os registradores de todas transações são ignorados e a execução é retomada do endereço de *fallback* indicado na *label* do primeiro xbegin. Em código *assembly* essa instrução somente faz a chamada da instrução xabort de baixo nível.

Numa implementação de um código, além do uso do xbegin e xend para delimitar o inicio e fim do trecho que se deseja uma transação executando em paralelo. Também é recomendado que sejam usados condicionais com os status da transação, para se acompanhar os números de *abort* e, principalmente, para fornecer uma alternativa de *fallback* para *aborts* na execução. Essas delimitações serão o controle de contenção, e poderão ser tão simples quanto na primeira falha diretamente executar a transação com sincronização bloqueante, como tão complexas quanto sistemas com cálculo de probabilidade de uma transação abortar, tempos de penalidade pra execuções de algumas transações, entre outros métodos mais sofisticados.

A Intel, apesar das falhas de segurança que levaram a desativação das instruções TSX, adicionou em suas documentações de conjuntos de instruções, no fim de 2020 (INTEL, 2020), uma nova expansão para o conjunto TSX, a TSX Load Address Tracking (TSXLDTRK). Esse novo conjunto introduz duas novas instruções, _xsusldtrk e _xresldtrk, que introduzem a possibilidade de implementar comportamento de transações explícitas a TSX. XSUSLDTRK faz a suspensão do monitoramento e armazenamento dos valores das variáveis da transação, enquanto XRESLDTRK retoma ambos. Colocando em outras palavras, o funcionamento é como se fosse o de um *mutex* reverso, ao invés de identificar áreas do código em que se possui variáveis que

não podem ser manipuladas por processos concorrentes, aqui se identifica as áreas do código em que se garante que nenhuma variável compartilhada será modificada. Essa técnica é especialmente importante para diminuir a quantidade de *aborts* de capacidade, ou seja, diminuir as falhas causadas pelos dados da transação terem sido despejados da cache.

Apesar da listagem da TSXLDTRK, o futuro das implementações de HTM da Intel ainda é incerto, com nenhum dos processadores entre Comet Lake, Alder Lake e Raptor Lake tendo acesso as instruções (sendo provável que eles possuam os núcleos fisicamente (GERASIMOV; BOIS; DAVIES, 2021)), nem com reativação pelas MSRs. A TSXLDTRK foi anunciada para ser lançada com os processadores Saphire Rapids, que após diversos atrasos será lançada em janeiro de 2023 (sendo o planejamento original para 2021), e com esse lançamento provavelmente ficará mais claro quais os planos da Intel para a tecnologia. Fora a Intel, ainda recentemente, em 2019 a ARM adicionou em sua documentação as intrínsecas para *Transactional Memory Extension* (TME), indicando que eles também possa explorar HTMs *Commodity* no futuro (ARM, 2021).

2.4 Conclusão do Capítulo

Neste capítulo foi apresentada a fundamentação teórica do modelo de Memória Transacional. Deste modelo, foram caracterizadas as principais formas de implementação, em software e em hardware, sendo discutido mais longamente o suporte para implementação de HTM em arquiteturas Intel. Embora as primeiras introduções do suporte à Memória Transacional em hardware tenha sido há algumas gerações de processadores, o fato é que a tecnologia não é explorada efetivamente. Um dos motivos pode estar associado aos aspectos transitórios de sua implementação. Este trabalho dedica-se a avaliar o potencial do uso da tecnologia tal como ela se encontra disponibilizada no momento.

3 TRABALHOS RELACIONADOS

Esse capítulo trata de discutir alguns trabalhos relacionados das literaturas de memórias transacionais e memórias transacionais em hardware.

(JARDIM, 2021), (JARDIM et al., 2021) e (PEREIRA et al., 2020) são trabalhos que se complementam, possuindo uma análise da mesma área específica e também resultados similares, e portanto serão aqui analisados de forma conjunta. Estes trabalhos tem o objetivo de fazer uma implementação de Memória Transacional em software no OpenMP, e analisar o desempenho da execução para que possam avaliar a viabilidade do seu uso na prática, substituindo os modelos já disponíveis atuais.

A implementação de STM nestes trabalhos é uma extensão ao OpenMP, de forma não invasiva, sendo possível usar TM sem mudar a forma de expressão do modelo OpenMP, nem sobre-escrever o funcionamento de nenhum outro elemento dele, permitindo que a implementação seja aplicada sem afetar o funcionamento de nenhum código OpenMP. Manter a forma de expressão é principalmente importante para manter a facilidade da adaptação para os programadores, e é uma característica que outras implementações existente, como Nebelung, OpenTM e "Wong", não possuem.

A cláusula '*transaction*', é o elemento que foi adicionado nessa extensão, e serve para especificar que aquele pedaço de código do bloco de comandos será uma transação atômica e deverá ser executado em paralelo.

Os experimentos feitos nestes trabalhos testaram três ferramentas de programação: TinySTM, GCC-TM e uma implementação de OpenMP, e foram executados nas mesmas máquinas para os três trabalhos, a Hydra e a Tekoha. A primeira coisa avaliada em todos foi a simplicidade de programação e número de linhas resultantes, e para avaliar isso, foram implementados programas do *benchmark* Cowichan, um *benchmark* mais apropriado para avaliação do nível de expressão, e não de desempenho. Porém, ao serem feitos os testes de desempenho, programas dessa benchmark foram alguns dos avaliados, em conjunto com testes de programas mais apropriados, como Bayes e K-means, duas aplicações que fazem parte do conjunto da *benchmark* STAMP, que são desenvolvidos exatamente para a avaliação de desempenho de ferramentas de memórias transacional. Ao final, os resultados quanto ao nível de abstração

foram bem positivos, mas os de desempenho, de forma geral, obtiveram resultados apenas satisfatórios, com muitos resultados que não puderam ser apropriadamente avaliados por não obedecerem a distribuição normal.

(DIEGUES; ROMANO; RODRIGUES, 2014) é um trabalho que visa esclarecer questões sobre a HTM, como sua performance e consumo energética quando comparado à diferentes técnicas bloqueantes, métodos de STM e híbridos. Também é definido no trabalho a intenção de identificar uma série de problemas de pesquisa em outros trabalhos das áreas de TM, além da intenção dele ser o maior estudo feito sobre TMs até então.

Para ser feita a pesquisa foram utilizados 13 (treze) mecanismos de sincronização diferentes, sendo seis métodos bloqueantes, quatro implementações de STM no estado da arte, uma implementação de HTM utilizando o conjunto de instruções TSX e dois métodos híbridos (que usam mecanismos de HTM e STM sinergeticamente). Foram testados e avaliados os resultados de performance e consumo energético destes métodos utilizando três conjuntos de aplicações: o benchmark STAMP, que já é o benchmark padrão para avaliação de implementações de TM; Memcached, um sistema de caching em memória; e algumas estruturas de dados concorrentes geralmente usadas na construção de aplicações paralelas.

Os resultados apresentados por este trabalho, de Diegues; Romano; Rodrigues, são de muito mais interesse para o trabalho que está sendo desenvolvido, por possuírem diretamente resultados do desempenho de implementação de HTM utilizando TSX. Eles foram resultados mistos, porém. Foi observado que para certas tarefas e tamanhos de tarefa, HTM obteve uma desempenho claramente superior a qualquer outro método de sincronização, enquanto que também foi observado que a implementação atual de HTM possui restrições que limitam o escopo em que eles podem ser superiores ao estado da arte de STM.

Dos trabalhos relacionados vistos neste capítulo, apenas um não utilizou o *benchmark* STAMP para a avaliação de desempenho das diferentes técnicas implementadas, um indicativo forte de que STAMP é o padrão de facto para testes da performance de memórias transacionais. Também quanto ao uso, ou não uso, de *benchmarks*, apenas um dos trabalhos não utilizou o Cowichan, porém neste caso isso aparenta ser por um principal motivo: diferente dos três trabalhos complementares, o trabalho de Diegues foca em outras avaliações que não quanto ao nível de abstração. Outro fator observado nos trabalhos que deve ser cuidado é a grande quantidade de casos com resultados que não seguem a curva normal, e portanto não podem ser comparados.

Os três trabalhos citados na primeira metade do capítulo, (JARDIM, 2021), (JARDIM et al., 2021) e (PEREIRA et al., 2020), foram desenvolvidos pelo grupo de pesquisa do LUPS, e apresentam um controle das implementações de memórias transacionais em software, e portanto pouco abordando as memórias transacionais em

hardware. Outros trabalhos de memórias transacionais desenvolvidos no LUPS também somente se limitaram a abordagens em software, fazendo com que esse trabalho seja o primeiro com foco no suporte em hardware, tendo a base dos conhecimentos teóricos das memórias transacionais, porém necessitando fazer toda a exploração inicial de características exclusivas das HTMs e de como implementar e executar elas. O trabalho de Diegues também tem uma importância grande, devido a ser focado em HTMs e implementações com uso das instruções TSX, que é exatamente o foco desse trabalho.

4 IMPLEMENTAÇÃO DA HTM E AVALIAÇÃO COM STAMP

Este capítulo foca em descrever o método de implementação de transações em hardware em aplicações reais, desenvolvendo uma biblioteca de macros que permita inserir instruções TSX em códigos com execução paralela, que será usada na compilação de um benchmark. O estudo de caso é realizado sobre o benchmark STAMP, onde uma biblioteca 'tm.h' define estruturas que podem ser facilmente trocadas entre diferentes métodos de controle de concorrência, e os resultados obtidos são avaliados com o uso de um algoritmo de análise estatística implementado em Python.

4.1 Metodologia

O benchmark STAMP, ou Stanford Transactional Applications for Multi-Processing, é uma suíte de *benchmarks* desenvolvida para avaliar sistemas de Memória Transacional, contendo oito aplicações com diversos parâmetros de entrada e conjuntos de dados, que permitem o teste de uma variedade de casos de execução transacional. Ele foi desenvolvido em 2008 para fornecer um conjunto para testes de memórias transacionais que não fossem apenas *microbenchmarks*, que não são capazes de apropriadamente representar o comportamento de aplicações no mundo real. Para guiar a criação do *benchmark* de forma que permita a avaliação adequada dos resultados de memórias transacionais e a comparação entre implementações de Memória Transacional em diferentes paradigmas, os desenvolvedores definiram três aspectos fundamentais que deveriam ser fornecidos pelo *benchmark*: largura, profundidade e portabilidade. Largura define que o *benchmark* deve possuir uma variedade de algoritmos com domínio de aplicação diferentes. Profundidade define que ele deve cobrir uma ampla gama de comportamentos transacionais, contemplando implementações com diferentes graus de contenção, duração das transações e tamanho dos conjuntos de escrita e leitura. E portabilidade define que ele deve ser capaz de facilmente implementar e executar diferentes classes de memórias transacionais, entre HTM, STM e HyTM (MINH et al., 2008).

As oito aplicações presentes na STAMP são: bayes, genome, intruder, kmeans,

labirynth, ssca2, vacation e yada. Cada uma dessas aplicações originando de uma diferente área de conhecimento, e possuindo características de transação de todo tipo. A garantia do último princípio delimitado foi feita com a implementação das aplicações utilizando *anotações* para definir as transações, com as mesmas anotações sendo definidas em uma biblioteca de macros, *tm.h*, que pode ser facilmente modificada e expandida para suportar diferentes sistemas e implementações.

As instruções TSX, como introduzidas na Seção 2.3, foram utilizadas na implementação com o uso da biblioteca de intrínsecas *immintrin.h*, e por proxy a biblioteca *rtmintrin.h*. Essas bibliotecas são as responsáveis pela definição das funções e variáveis usadas para chamar as instruções TSX no nível de máquina.

As máquinas utilizadas para desenvolvimento e experimentação neste trabalho foram os recursos oferecidos pela Intel DevCloud e por um servidor local. Os recursos da nuvem foram utilizados para desenvolvimento e prototipação e o servidor local para coleta de tempos de execução.

Para análise estatística dos resultados e avaliação de desempenho, foi implementado um notebook em Python na plataforma Google Colab. Este notebook foi inicialmente desenvolvido em um projeto do LUPS (Laboratory of Ubiquitous and Parallel Systems) para ser utilizado na avaliação dos resultados da execução do NAS Parallel Benchmark, e disponibilizado para o uso em outros trabalhos com as devidas adaptações dos arquivos de entrada (DI DOMENICO; CAVALHEIRO; LIMA, 2022). O programa lê de um arquivo de entrada que fornece as características de cada execução e o tempo decorrido nela, e utiliza algoritmos que implementam os testes de Kolmogorov-Smirnov, Shapiro-Wilk, T de Student e U de Mann-Whitney para determinar a normalidade dos tempos resultantes de um grupo de execuções, e a heterogeneidade entre os resultados de dois grupos. Os resultados dos testes são expressos pelo programa em uma tabela e através de gráficos da distribuição dos dados e da aceleração (*speedup*) obtida.

4.2 Uso das instruções de hardware

Para que a implementação de novas abordagens de memórias transacionais possam ser adicionadas a ela e aplicadas a todas suas aplicações de forma padronizada, a *benchmark* STAMP utiliza de um conjunto de *anotações* análogas a instruções e funções presentes, e/ou necessárias, na grande parte das memórias transacionais, e essas 'anotações' são definidas em uma biblioteca tm.h, onde elas são associadas a funções (ou por conjuntos de instruções) que implementam diferentes técnicas e abordagens de Memória Transacional. O conjunto de macros, presente dentro dessa biblioteca, que será usado na compilação das aplicações, é definido pelo uso de *flags* de compilação, com diretivas '#ifdef' sendo usadas para definir as macros apontadas

pelo identificados do compilador.

A implementação original da STAMP, mantida por Minh e Kozyrakis, apesar de possuir uma estrutura pronta para a adição de HTM na biblioteca, somente possuía uma implementação básica para o uso com simuladores de hardware. Algumas [poucas] re-distribuições do *benchmark* estenderam a biblioteca com suporte para HTM não simulado, e no intuito deste trabalho foi selecionado a versão mantida pelos pesquisadores da Universidade Técnica de Lisboa para ser usada como base da implementação. Essa versão da STAMP, nomeada htm-stamp (FILIPE, 2017), possui implementações fornecidas para a HTM da IBM e para o conjunto de instruções HLE da Intel, porém, não para as instruções RTM, ficando a base perfeita para o uso neste projeto.

Assim que foi iniciado o desenvolvimento em cima dessa versão, no entanto, foi possível identificar alguns problemas que precisavam ser corrigidos antes de dar prosseguimento na implementação. O primeiro, e mais inconsequente, é que o *script* de compilação e execução usado por eles foi desenvolvido na linguagem Perl, o que dificultava a modificação e expansão da automatização do *script*. Entretanto, o outro problema era na compilação com o uso do HLE, e exigiu um esforço de *debugging* para corrigir as partes necessárias.

Com a constatação de que o uso da HLE havia sido deprecado, além do intuito já existente de fazer a implementação da RTM, o foco se direcionou completamente em identificar quais seriam os equivalentes da RTM com os elementos de memórias transacionais abstraídos pela STAMP, além de identificar a forma mais apropriada de se implementar essas instruções.

As instruções TSX, e o conjunto das instruções RTM por conseguinte, são disponibilizadas no nível de máquina, necessitando que seja usada alguma função ou abstração para usa-las em linguagens de mais alto nível, tal qual C. Portanto duas abordagens foram encontradas em outras implementações para fazer essa interface com a linguagem de montagem. Uma delas criou uma biblioteca *rtm.h*, na qual é feita a definição de macros com o valor dos bits de status e de macros onde instruções de assembler inline são usadas, sendo indicados uma série de opcodes das instruções de TSX. A outra abordagem é a inclusão da biblioteca immintrin.h, uma biblioteca que faz as definições de todas outras bibliotecas de intrínsecas da Intel, gerando efetivamente as mesmas definições da abordagem com *rtm.h*.

Para se manter a simplicidade se optou por somente utilizar a biblioteca de intrínsecas, também se atentando que a flag -mrtm"deve ser adicionada na compilação. E com o acesso as intrínsecas foi possível ser feita uma primeira versão da implementação, onde TM_BEGIN( ), TM_END( ) e TM_ABORT( ) são definidos com a chamada de _xbegin( ), _xend( ) e _xabort( ). Porém a implementação ainda necessitava algum tipo de fallback para evitar que a execução possa acabar travada em tentativas indeterminadas, sem garantia que ela irá concluir, se definindo uma implementação com

uso de *locks* globais e número de tentativas, após alguma iteração.

4.3 Coleta de Resultados

Para a execução e coleta dos resultados, ainda foi necessário implementar dois *scripts* para automatizar o processo, após concluir a implementação do suporte à HTM. Um dos *scripts* (Script_htm-stamp.sh) foi feito para que com somente a execução dele fosse possível limpar e recompilar todas aplicações, executar cada uma delas 30 vezes, mudar os parâmetros da execução, e salvar as saídas para um arquivo de resultados específico de cada uma das aplicações. O outro *script* feito, trabalhava em cima desses arquivos de resultados, tratando as saídas para ficarem padronizadas no padrão que é usado pelo NPB_statistics, e também unindo todos resultados em um único arquivo.

Os primeiros testes feitos com essa nova implementação de RTM na STAMP foram executados no ambiente da Intel DevCloud, porém, logo foi percebido que as instruções da TSX não estavam funcionando devidamente, isso sendo em virtude que, com a desabilitação das instruções como medida de segurança, a Intel também não disponibiliza acesso à elas no momento em suas plataformas na nuvem.

Para a execução efetiva dos testes da nova implementação, conseguiu-se obter acesso a uma máquina do grupo de pesquisa GACI com disposição das instruções TSX. Nela foram feitos algumas verificações iniciais pra confirmar a usabilidade, constatando que elas estavam disponíveis e ativadas, com a ressalva de que no nível do *kernel* também estava ativada uma função de mitigação das vulnerabilidades da TSX Async Abort, que limpa o *buffer* da CPU em transições de *Ring*, e que podem ter afetado os resultados obtidos (aumentando o número de *aborts*, por exemplo).

A máquina do GACI (Grupo de Arquiteturas e Circuitos Integrados) em que foram feitas as execuções é uma Intel(R) Xeon(R) Gold 5118, com BIOS na versão 1.3.7, e rodando em um Linux 5.4.0. A versão do GCC instalada e usada para a compilação das implementações é a 9.4.0.

Durante o tempo de acesso limitado, foram ainda feitos alguns ajustes finais nos códigos, agora tendo como testar o funcionamento dos elementos individuais. Foi executada, algumas vezes, a implementação, desenvolvida neste trabalho, usando diferentes parâmetros de entrada e quantidades de *retries*. Também foram executados alguns dos mesmos testes sequencialmente, e com implementações de STMs disponibilizadas em outra distribuição da STAMP, a OpenSTMP. Um último teste que também fui capaz de executar foi com a extensão de rtm-bench do Zixian Cai, onde o tamanho das transações é aumentado até o maior elemento em que o programa não é capaz de *commitar* nenhuma transação, com o sucesso dessa execução sendo mais uma demonstração do funcionamento apropriado das instruções de Memória Transacional.

Os resultados de todas essas execuções e testes que foram feitos, foram salvas em diferentes documentos de texto. Para os casos em que os resultados foram salvos corretamente e a execução funcionou como esperado, eles foram tratados com o *script* previamente mencionado, gerando um único arquivo de saída com todos os resultados devidamente tratados. Esses arquivos tratados foram processados pelo notebook de análise estatística. Os resultados da aplicação dos métodos estatísticos são catalogados, para permitir comparação de resultados entre as diferentes versões do programa.

5 RESULTADOS

Neste capítulo são discutidos os resultados obtidos. Inicialmente é realizada a validação das saídas produzidas pela biblioteca desenvolvida com as produzidas pelo TinySTM. Na seção seguinte, apresentada uma avaliação do desempenho em termos de tempos de execução.

Todos resultados obtidos que serão apresentados neste capítulo foram compilados e executados na máquina GaciG5[1]

5.1 Validação das Saídas

Devido a dificuldade de conseguir acesso a máquinas com as instruções necessárias, e do curto período de tempo que se teve quando foi possível utilizar uma, ajustes no código foram realizados durante os períodos de tempo em que os *benchmarks* estavam rodando. Isso ocasionou que o conjunto limitado de testes possíveis durante o tempo em que a máquina esteve disponível foi ainda reduzido pela necessidade de ajustes. Isso também fez com que não fosse possível analisar os resultados e fazer execuções com testes complementares, os quais teriam por objetivo confirmar ou refutar hipóteses sobre os comportamentos observados ou mesmo para compreender a razão de determinadas técnicas apresentarem resultados promissores.

Um grande problema, identificado nos resultados obtidos, foi que grande parte, se não todas execuções, não foram capazes de completar nenhuma transação com sucesso. A não existência de *commits* não é, por si só, suficiente para indicar que realmente existe algum erro na implementação ou execução, porém, com a persistência do problema nos testes feitos com números de tentativas consideravelmente maiores em cada transação, e com aplicações do *benchmark* que possuem baixa contenção, a situação aponta para um possível problema maior. Na revisão do código e dos resultados, também não foi possível identificar nenhuma falha saliente na implementação que explicaria esse problema, então a hipótese levantada é de que a mitigação no

[1] Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz. Skylake, 14nm. Cache L1d: 384KiB, L2: 12MiB, L3: 16.5MiB. DDR4-2400. Cores: 12, Threads: 24. TDP: 105 W; Sistema operacional Ubuntu 20.04.3 LTS. Linux 5.4.0-91-generic. x86-64

Tabela 1 – Resultados para rtm-bench com configurações padrão.

	Gaci G5
Baseline	[9.0896 -> 9.4101) -3.86
Reuse	[135.6942 -> 140.4795) -1.91

nível de *Kernel*, onde o *buffer* da CPU é esvaziado possa ser o motivo da proporção de *aborts* maior do que esperada.

Para garantir que as instruções estavam realmente ativadas, como indicado pelo sistema, e de que elas estavam funcionando corretamente, fazendo *commits* e *aborts*, foi usado a *benchmark rtm-bench* expandida por Cai; Blackburn; Bond, onde transações de tamanho incrementalmente maiores são iniciadas e imediatamente concluídas, com o intuito de testar o maior tamanho de blocos de leitura e de escrita em um dado processador. Os resultados desse teste, utilizando as configurações padrão, são apresentados na Tabela 1. Ela apresenta os resultados para duas formas de tratamento da cache, *baseline* e *reuse*, onde *baseline* representa que a transação irá usar uma memória que nunca foi usada por outra transação do mesmo tamanho, e *reuse* representa que a mesma memória será usada para todas transações de um mesmo tamanho. Dentre os três valores apresentados, o primeiro indica qual foi o tamanho, em Kbits, da maior transação que *commitou* ao menos uma vez. O segundo valor apresenta a próxima maior transação que ele tentou mas não teve nenhum sucesso. E, em cinza claro, o valor representa o logaritmo da frequência de *commits* na maior transação.

Esses resultados servem para validar o funcionamento das instruções RTM, entretanto, sendo também necessário validar o comportamento atômico da implementação desenvolvida e a validade das saídas da execução, demonstrando que, independente do tempo de execução, os resultados dos algoritmos para uma determinada entrada sempre resultam na mesma saída. Essa consistência dos dados pôde ser verificada em todas as saídas de todas execuções feitas, confirmando a validade dos dados.

5.2 Análise de Desempenho

Tendo os resultados validados e a confirmação de que as instruções de Memória Transacional estavam devidamente funcionando, tendo também corrigido os erros encontrados na implementação e testado ela com execuções menores de *debug*, foram então feitas as execuções dos testes com os parâmetros descritos na Tabela 2.

Dos resultados obtidos dessas execuções, foi feita uma seleção pra retirar resultados com falhas, e foram agrupados aqueles que possuíam os mesmos parâmetros de entrada, e, portanto, fazem sentido ser comparados. Então, a análise dos resultados,

Tabela 2 – Parâmetros usados na execução do primeiro conjunto de testes. Quase todos os valores são os mesmos indicados pelo benchmark para uso em execuções não simuladas.

bayes		-v32 -r4096 -n10 -p14 -i2 -e8 -s1
genome		-g16384 -s64 -n8486974
intruder		-a10 -l128 -n142144 -s1
kmeans	Lo	-m40 -n40 -t0.00001 -i inputs/random-n65536-d32-c16.txt
	Hi	-m15 -n15 -t0.00001 -i inputs/random-n65536-d32-c16.txt
labyrinth		-i inputs/random-x512-y512-z7-n512.txt
ssca2		-s18 -i1.0 -u1.0 -l3 -p3
vacation	Lo	-n2 -q90 -u98 -r1048576 -t2194304
	Hi	-n4 -q60 -u90 -r1048576 -t2194304
yada		-a15 -i inputs/ttimeu10000.2

Tabela 3 – Resultados da análise estatística dos tempos de execução no primeiro conjunto testado.

	RTM-1 - Low	RTM-50-Low	TinySTM - Low	SwissTM - Low
mean	0.15813333	0.71324324	0.09640000	0.10370000
KS	0.00212453	0.00000005	0.00008335	0.01528637
KS p>0.05	Sample Bad	Sample Bad	Sample Bad	Sample Bad
SW p	0.00012799	0.00000000	0.00000057	0.00002978
SW p>0.05	Sample Bad	Sample Bad	Sample Bad	Sample Bad
	RTM-1 - High	RTM-50 - High	TinySTM - High	SwissTM - High
mean	8.06883460	50.74735327	17.55868447	17.34454050
KS	0.09966094	0.00000001	0.17978156	0.46643610
KS p>0.05	Sample OK	Sample Bad	Sample OK	Sample OK
SW p	0.00091592	0.00000000	0.00772318	0.02126906
SW p>0.05	Sample Bad	Sample Bad	Sample Bad	Sample Bad

dessas comparações, foi feita, e é apresentada e discutida a seguir.

A Tabela 3 apresenta os resultados obtidos na execução da *benchmark* STAMP com o uso dos parâmetros descritos na Tabela 2. Os testes foram executados com a implementação de RTM desenvolvida neste trabalho usando 1 e 50 repetições da transação antes do uso do *fallback* (*lock global*), e também para as implementações de STM, Tiny e Swiss. A diferenciação de *Low* e *High* descrita na tabela identifica os resultados para as aplicações Kmeans e Vacation com parâmetros de baixa e alta contenção (resultados de outras aplicações aparecem agrupados com os resultados de baixa contenção). Os elementos apresentados são somente a média do tempo de execução, que permite a comparação, e os resultados dos testes de normalidade Kolmogorov-Smirnov e Shapiro-Wilk, para verificar a validade da comparação. Uma versão completa da tabela é encontrada no Apêndice B, Tabela 6.

Essa tabela 3, demonstra que somente foi possível obter confirmação da norma-

lidade, verificada com o uso do teste de Kolmogorov-Smirnov, nos resultados de alta contenção das implementações STM e RTM de uma repetição, e sem ser verificada com o uso do teste de Shapiro-Wilk. Sem possuir a validação da normalidade dos resultados não é possível fazer afirmações sobre a comparação dos resultados usando o Teste t de Student, necessitando fazer a utilização do teste de Mann-Whitney, que fará a comparação desconsiderando a homogeneidade necessária para o teste t.

As Figuras 1 e 2 apresentam os resultados dos testes de hipótese, T de Student e U de Mann-Whitney, que são usados para verificar a validade da comparação entre todos pares de implementações. Em todos as comparações foi verificada relação válida entre os resultados, com 95% de confiança, no uso do teste de Mann-Whitney. Também foi verificada uma relação válida dos dados nas comparações entre RTM-1, SwissTM e TinySTM para casos de alta contenção. Uma versão completa dos resultados é encontrada no Apêndice B, Figuras 6 e 7.

```
                 (RTM-1[Low]) x (RTM-50[Low])
      T-Test p          6.163134747051e-05
T-Test p<=0.05        Not normally distrib.
      U-Test p          1.479276809093e-12
U-Test p<=0.05                   U-Test OK

                 (RTM-1[Low]) x (TinySTM[Low])
      T-Test p          4.702264111288e-85
T-Test p<=0.05        Not normally distrib.
      U-Test p          6.737635900891e-12
U-Test p<=0.05                   U-Test OK

                 (RTM-1[Low]) x (SwissTM[Low])
      T-Test p          1.857006892332e-88
T-Test p<=0.05        Not normally distrib.
      U-Test p          1.001445308518e-11
U-Test p<=0.05                   U-Test OK

                 (RTM-50[Low]) x (TinySTM[Low])
      T-Test p          1.332504047012e-05
T-Test p<=0.05        Not normally distrib.
      U-Test p          1.173965955031e-12
U-Test p<=0.05                   U-Test OK

                 (RTM-50[Low]) x (SwissTM[Low])
      T-Test p          1.598941469713e-05
T-Test p<=0.05        Not normally distrib.
      U-Test p          1.580139554790e-12
U-Test p<=0.05                   U-Test OK

                 (TinySTM[Low]) x (SwissTM[Low])
      T-Test p          5.332981245073e-45
T-Test p<=0.05        Not normally distrib.
      U-Test p          7.371481068518e-12
U-Test p<=0.05                   U-Test OK
```

Figura 1 – Testes de Hipótese para amostras de baixa contenção

Portanto, é válido afirmar que a execução de menos tentativas da transação resultou em tempos de execução significativamente menores que os obtidos para as outras implementações em situações de alta contenção, porém, performou inferior que as STMs em situações de baixa contenção, mesmo com o *overhead* existente nessas implementação, que em teoria devem ser mais evidentes em execuções mais curtas. Essa diferença fica mais evidente na Figura 3, onde os resultados do tempo de cada uma das aplicações do STAMP são exibidos individualmente, com o gráfico de *speedup*, da Figura 4 também auxiliando visualizar os ganhos e perdas de tempo de cada implementação.

O crescimento exponencial do tempo de execução com o maior número de repetições da transação, como observado, também aparenta indicar que quase nenhuma, se não nenhuma, transação está concluindo de forma bem sucedida, gerando um re-

(RTM-1[High]) x (RTM-50[High])	
T-Test p	5.535443884005e-05
T-Test p<=0.05	Not normally distrib.
U-Test p	2.598295687016e-12
U-Test p<=0.05	U-Test OK

(RTM-1[High]) x (TinySTM[High])	
T-Test p	1.291906743302e-80
T-Test p<=0.05	T-Test OK
U-Test p	2.871949066320e-11
U-Test p<=0.05	U-Test OK

(RTM-1[High]) x (SwissTM[High])	
T-Test p	4.574344492313e-71
T-Test p<=0.05	T-Test OK
U-Test p	2.871949066320e-11
U-Test p<=0.05	U-Test OK

(RTM-50[High]) x (TinySTM[High])	
T-Test p	1.081496162054e-03
T-Test p<=0.05	Not normally distrib.
U-Test p	2.598295687016e-12
U-Test p<=0.05	U-Test OK

(RTM-50[High]) x (SwissTM[High])	
T-Test p	1.013995278489e-03
T-Test p<=0.05	Not normally distrib.
U-Test p	2.598295687016e-12
U-Test p<=0.05	U-Test OK

(TinySTM[High]) x (SwissTM[High])	
T-Test p	4.233386391873e-07
T-Test p<=0.05	T-Test OK
U-Test p	1.067005555706e-06
U-Test p<=0.05	U-Test OK

Figura 2 – Testes de Hipótese para amostras de alta contenção

Figura 3 – Gráficos de tempo de execução para cada aplicação usada

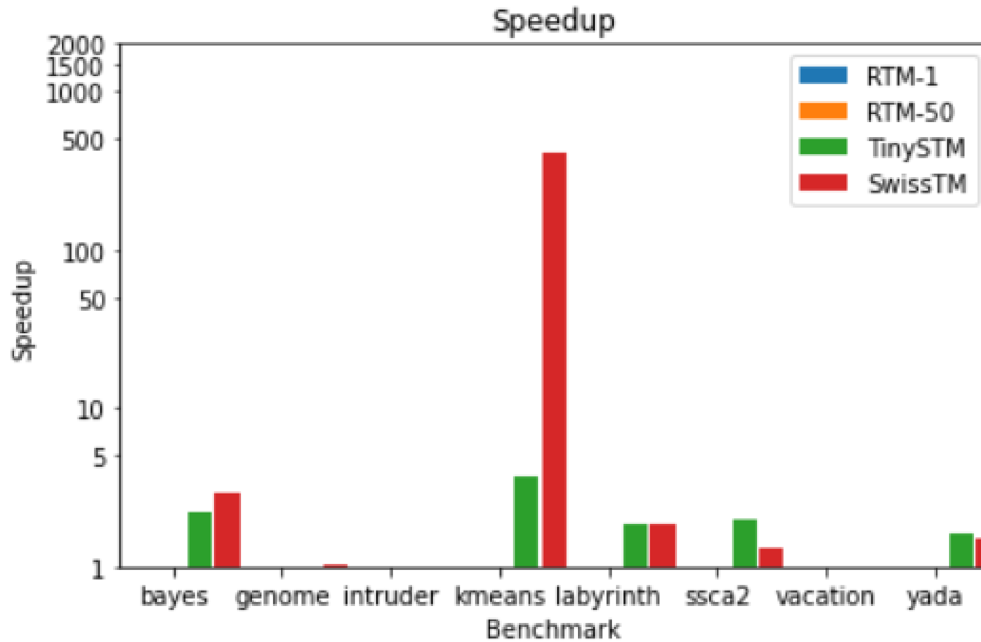

Figura 4 – Gráficos de speedup para cada aplicação usada

Tabela 4 – Parâmetros usados na execução do segundo conjunto de testes. Sendo todos os valores os mesmos que os indicados pelo benchmark para uso em execuções não simuladas.

bayes		-v32 -r4096 -n10 -p14 -i2 -e8 -s1 -t24
genome		-g16384 -s64 -n16777216 -t24
intruder		-a10 -l128 -n262144 -s1 -t24
kmeans	Lo	-m40 -n40 -t0.00001 -i inputs/random-n65536-d32-c16.txt -p24
	Hi	-m15 -n15 -t0.00001 -i inputs/random-n65536-d32-c16.txt -p24
labyrinth		-i inputs/random-x512-y512-z7-n512.txt -t24
ssca2		-s20 -i1.0 -u1.0 -l3 -p3 -t24
vacation	Lo	-n2 -q90 -u98 -r1048576 -t4194304 -c24
	Hi	-n4 -q60 -u90 -r1048576 -t4194304 -c24
yada		-a15 -i inputs/ttimeu10000.2 -t24

sultado que sempre será inferior à execução com o uso de técnicas de sincronização tradicionais. Para verificar a validade dessa hipótese, serão analisados os resultados obtidos em uma execução com parâmetros diferentes, para implementações de RTM com 5 repetições e execução sequencial, através do uso de *mutex*. Os parâmetros da nova execução são descritos na Tabela 4 e resultados na Tabela 5. Note que essas execuções também utilizaram 24 *threads* e, por consequência, não foram comparados com os resultados anteriores.

Todos os resultados obtidos nesse novo grupo de execuções obedeceram o fator de normalidade e aparentam ser extremamente próximos uns dos outros, quanto ao tempo de execução, porém como pode ser observado na Figura 4, as execuções de baixa contenção falharam ambos os testes de hipótese. Ainda assim, com os testes verificando o entrelaçamento dos dados, e da mesma forma demonstrando a execução

Tabela 5 – Resultados da análise estatística dos tempos de execução no segundo conjunto testado.

	RTM-5 - Low	Mutex - Low	RTM-5 - High	Mutex - High
count	30.0000000	30.0000000	30.0000000	30.0000000
KS	0.94953735	0.61377609	0.82179332	0.98435875
KS p>0.05	Sample OK	Sample OK	Sample OK	Sample OK
SW p	0.52263588	0.18199947	0.72744626	0.85255736
SW p>0.05	Sample OK	Sample OK	Sample OK	Sample OK

sequencial com tempos de execução inferiores aos obtidos na execução transacional. Uma versão completa dos resultados é encontrada no Apêndice B, Tabela 7 e 8.

	(RTM-5[Low]) x (Sequential[Low])
T-Test p	1.947179025543e-01
T-Test p<=0.05	T-Test Failed
U-Test p	2.088709239187e-01
U-Test p<=0.05	U-Test Failed

	(RTM-5[High]) x (Sequential[High])
T-Test p	3.815633616626e-02
T-Test p<=0.05	T-Test OK
U-Test p	4.757860204720e-02
U-Test p<=0.05	U-Test OK

Figura 5 – Testes de Hipótese para o segundo conjunto de execuções

6 CONCLUSÃO

O trabalho conseguiu navegar a literatura, muitas vezes confusa, e a documentação fraca das memórias transacionais em hardware, para conciliar essas informações e assim alcançar um melhor e mais acessível entendimento teórico e prático da área. Sendo esse esclarecimento de conceitos e de conhecimentos práticos um dos principais objetivos, que foi alcançado. E então fornecendo neste texto as informações, de HTM, preenchendo lacunas e elucidando conceitos, com a descrição de quais sistemas possuem ou não suporte a HTM e TSX, de quais os métodos de reativação e utilização das instruções TSX após o lançamento da atualização do *microcode*, e de outras características que ficavam nebulosas em outros textos.

Também se alcançou o objetivo de identificar o potencial e funcionamento das instruções TSX, e usa-las na implementação de uma biblioteca para o *benchmark* STAMP. Havendo a comprovação de que HTM pode ser utilizada como uma alternativa aos métodos bloqueantes de sincronização em sistemas concorrentes, não afetando o comportamento observado nas saídas de uma aplicação paralela. Além de confirmar a possibilidade da reativação e uso do conjunto TSX para a execução de programas HTM, mesmo após diversas medidas de segurança implementadas, em diferentes níveis do sistema, para mitigar os efeitos da falha de segurança TAA.

Além disso, se adquiriu o conhecimento de diversos detalhes menores sobre HTMs que não puderam ser mencionados no texto, porque não tinham relação direta com o assunto debatido, e se identificou algumas grandes questões ainda não respondidas sobre a área, e especialmente sobre as instruções TSX, que levantam diversas possibilidades para trabalhos futuros.

6.1 Trabalhos Futuros

Como recém mencionado, existem muitas pontas soltas nos assuntos da área, e não poucas delas deixadas pela baixa quantidade de documentação ou transparência, tanto dos pesquisadores quanto da Intel. Por exemplo, quais os efeitos das mitigações em nível de *kernel* no tempo de execução e comportamento de um código de HTM?

Os processadores da arquitetura Alder Lake possuem o núcleo de processamento da TSX ainda no *die*[1]? As instruções TSX serão reativadas com o lançamento dos processadores Saphire Rapids, que teoricamente introduzirão o novo conjunto de instruções TSXLDTRK? E qual o futuro para a implementação de HTM da ARM (TME)?

Trabalhos futuros também abrem a possibilidade de melhor testar características da implementação. Fazendo o teste e comparação de *fallbacks* alternativos, identificando se há diferença em performance entre o uso da biblioteca de intrínsecas da Intel ou declarando as instruções diretamente no assembly in-line, e implementando um registro do número e tipo dos *aborts* ocorridos na execução para melhor investigar as causas de *abort* mais comuns.

Ainda há interesse em um trabalho futuro que entre em contato com pesquisadores e engenheiros da área, para melhor esclarecer noções que não ficaram claras, e aparentam ser mais complicadas de se entender, além de sondar quais os prospectos que eles tem sobre a área (passado, presente e futuro).

[1] Documentação inicial parecia indicar que sim, porém caso tenha, não são re-habilitáveis por não possuir o *microcode* que adicionou os registradores que permitem reativar

REFERÊNCIAS

AMD. **Advanced Synchronization Facility - Proposed Architectural Specification**.

ARM. **Arm C Language Extensions**.

CAI, Z.; BLACKBURN, S. M.; BOND, M. D. Understanding and utilizing hardware transactional memory capacity. In: ACM SIGPLAN INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT, 2021., 2021. **Proceedings...** [S.l.: s.n.], 2021. p.1–14.

CLICK, C. **Cliff Click — The Azul Hardware Transactional Memory experience**. Disponível em: <https://www.youtube.com/watch?v=GEkeOHw87Sg>.

COSTA, G. **RPCS3 Inside Look**: A Deep-Dive into Hardware and Performance Scaling! Disponível em: <https://rpcs3.net/blog/2020/08/21/hardware-performance-scaling/>.

DI DOMENICO, D.; CAVALHEIRO, G. G. H.; LIMA, J. V. F. NAS Parallel Benchmark Kernels with Python: A performance and programming effort analysis focusing on GPUs. In: EUROMICRO INTERNATIONAL CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (PDP), 2022., 2022. **Anais...** [S.l.: s.n.], 2022. p.26–33.

DICE, D.; LEV, Y.; MOIR, M.; NUSSBAUM, D. Early Experience with a Commercial Hardware Transactional Memory Implementation. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 14., 2009. **Proceedings...** Association for Computing Machinery, 2009. p.157–168. (ASPLOS XIV).

DIEGUES, N.; ROMANO, P.; RODRIGUES, L. Virtues and limitations of commodity hardware transactional memory. In: PARALLEL ARCHITECTURES AND COMPILATION, 23., 2014. **Proceedings...** [S.l.: s.n.], 2014. p.3–14.

DRAGOJEVIĆ, A.; GUERRAOUI, R.; KAPALKA, M. Stretching transactional memory. **ACM sigplan notices**, [S.l.], v.44, n.6, p.155–165, 2009.

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.237–246.

FILIPE, R. **htm-stamp**. Disponível em: <https://github.com/riclas/htm-stamp>.

GERASIMOV, P.; BOIS, M. du; DAVIES, L. **Game Dev Guide for 12th Gen Intel® Core™ Processor**. Disponível em: <https://www.intel.com/content/www/us/en/developer/articles/guide/12th-gen-intel-core-processor-gamedev-guide.html>.

GRAHAM, S. L.; SNIR, M.; PATTERSON, C. A. Getting up to speed. **The future of supercomputing. Report of National Research Council of the National Academies Sciences**, [S.l.], 2004.

HARRIS, T.; LARUS, J.; RAJWAR, R. Transactional memory. **Synthesis Lectures on Computer Architecture**, Wisconsin, v.5, n.1, p.1–263, 2010.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In: COMPUTER ARCHITECTURE, 20., 1993. **Proceedings...** [S.l.: s.n.], 1993. p.289–300.

IBM. **POWER9 Processor User's Manual (Version 2.1)**.

INTEL, c. **Intel® Transactional Synchronization Extensions (Intel® TSX) Asynchronous Abort**. Disponível em: <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/intel-tsx-asynchronous-abort.html>.

INTEL, c. **Intel® Architecture Instruction Set Extensions Programming Reference**. [S.l.]: Intel Corporation, 2020.

INTEL, c. **Performance Monitoring Impact of Intel® Transactional Synchronization Extension Memory Ordering Issue**. [S.l.]: Intel Corporation, 2021.

INTEL, c. **Intel® Transactional Synchronization Extension (Intel® TSX) Disable Update for Selected Processors**. [S.l.]: Intel Corporation, 2022.

JARDIM, A. D. **Uma extensão à OpenMP para suporte à memória transacional**. 2021. Tese (Doutorado em Ciência da Computação) — Universidade Federal de Pelotas.

JARDIM, A. D. et al. An extension for Transactional Memory in OpenMP. In: BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 25., 2021. **Anais. . .** [S.l.: s.n.], 2021. p.58–65.

LARUS, J. R.; RAJWAR, R. Transactional memory. **Synthesis Lectures on Computer Architecture**, [S.l.], v.1, n.1, p.1–226, 2007.

LINUX, K. **Intel® Transactional Synchronization Extensions (Intel® TSX) Asynchronous Abort**. Disponível em: <https://www.kernel.org/doc/html/latest/x86/tsx_async_abort.html>.

LOMET, D. B. Process Structuring, Synchronization, and Recovery Using Atomic Actions. , New York, NY, USA, v.2, n.2, p.128–137, mar 1977.

MANN, H. B.; WHITNEY, D. R. On a test of whether one of two random variables is stochastically larger than the other. **The annals of mathematical statistics**, [S.l.], p.50–60, 1947.

MASSEY JR, F. J. The Kolmogorov-Smirnov test for goodness of fit. **Journal of the American statistical Association**, [S.l.], v.46, n.253, p.68–78, 1951.

MINH, C. C.; CHUNG, J.; KOZYRAKIS, C.; OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In: IEEE INTERNATIONAL SYMPOSIUM ON WORKLOAD CHARACTERIZATION, 2008., 2008. **Anais. . .** [S.l.: s.n.], 2008. p.35–46.

NAKAIKE, T. et al. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, ZEnterprise EC12, Intel Core, and POWER8. **SIGARCH Comput. Archit. News**, [S.l.], v.43, n.3S, p.144–157, 2015.

PEREIRA, H. et al. Estudo da Viabilidade de uma Interface para Memórias Transacionais em OpenMP. In: XXI SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2020. **Anais. . .** [S.l.: s.n.], 2020. p.37–48.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias transacionais: Uma nova alternativa para programaçao concorrente. In: MINICURSOS DO VIII WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD, 2007. **Anais. . .** [S.l.: s.n.], 2007.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples)†. **Biometrika**, [S.l.], v.52, n.3-4, p.591–611, 12 1965.

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 1995. **Proceedings. . .** [S.l.: s.n.], 1995. p.204–213.

STUDENT. The probable error of a mean. **Biometrika**, [S.l.], p.1–25, 1908.

APÊNDICE A REPRODUTIBILIDADE

Todos arquivos usados no desenvolvimento deste trabalho podem ser encontrados em: https://github.com/hpfred/TCC_Arquivos/.[1]

- Para reproduzir a execução dos testes da implementação com RTM, acesse o submódulo HTM-Stamp_Modified, e execute o script shell "Script_htm-stamp.sh". E para modificar a execução, o script fornece duas listas de elementos com todas as aplicações da STAMP e com três métodos de sincronização do paralelismo, só necessitando comentar ou descomentar aqueles que deseja rodar. Modificações dos parâmetros de entrada também são feitos no script, entre as linhas 100 e 203.[2]
- Para reproduzir a execução dos testes das implementações em STM, acesse o submódulo OpenSTMP-modified, e execute o script shell "Script_TinySTM.sh". A estrutura é similar a do "Script_htm-stamp.sh", possuindo listas das aplicações e das implementações de STM, e trechos para a definição dos parâmetros de execução.
- Para reproduzir os testes rtm-bench, do Zixian Cai, acesse o repositório e rode `python run_ismm21.py Baseline Reuse`, ou siga as instruções indicadas no README.[2]
- Para reproduzir os testes estatístico, acesse o submódulo NPB_statistics-Frederico. Na pasta "Teste-Tratamento"estará um script Python "tratamento.py", mova todos os arquivos de resultados gerados pelos testes com "HTM-Stamp_Modified"e "OpenSTMP-modified"e rode `python tratamento.py`. Mova o arquivo "tratado"que é gerado para uma pasta "Arquivos"que deve estar na pasta "Colab Notebooks"que o Google cria automaticamente no Drive.

[1] Uma snapshot da página do github também pode ser encontrada em: https://archive.org/web/

[2] Execuções de RTM necessitam um processador com suporte de TSX e modificações de registradores MSR

APÊNDICE B RESULTADOS COMPLETOS

	(RTM-1[Low]) x (RTM-50[Low])	(RTM-1[Low]) x (TinySTM[Low])	(RTM-1[Low]) x (SwissTM[Low])
KS OK?	No	No	No
T-Test p	6.163134747051e-05	4.702264111288e-85	1.857006892332e-88
T-Test stat	-4.53492248	328.81079534	268.46354627
T-Test p<=0.05	Not normally distrib.	Not normally distrib.	Not normally distrib.
U-Test p	1.479276809093e-12	6.737635900891e-12	1.001445308518e-11
U-Test stat	0.00000000	900.00000000	900.00000000
U-Test p<=0.05	U-Test OK	U-Test OK	U-Test OK

	(RTM-50[Low]) x (TinySTM[Low])	(RTM-50[Low]) x (SwissTM[Low])	(TinySTM[Low]) x (SwissTM[Low])
KS OK?	No	No	No
T-Test p	1.332504047012e-05	1.598941469713e-05	5.332981245073e-45
T-Test stat	5.03924998	4.97961226	-44.41693179
T-Test p<=0.05	Not normally distrib.	Not normally distrib.	Not normally distrib.
U-Test p	1.173965955031e-12	1.580139554790e-12	7.371481068518e-12
U-Test stat	1110.00000000	1110.00000000	0.00000000
U-Test p<=0.05	U-Test OK	U-Test OK	U-Test OK

Figura 6 – Testes de Hipótese para amostras de baixa contenção

	(RTM-1[High]) x (RTM-50[High])	(RTM-1[High]) x (TinySTM[High])	(RTM-1[High]) x (SwissTM[High])
KS OK?	No	Yes	Yes
T-Test p	5.535443884005e-05	1.291906743302e-80	4.574344492313e-71
T-Test stat	-4.57058499	-340.88587149	-291.13558161
T-Test p<=0.05	Not normally distrib.	T-Test OK	T-Test OK
U-Test p	2.598295687016e-12	2.871949066320e-11	2.871949066320e-11
U-Test stat	0.00000000	0.00000000	0.00000000
U-Test p<=0.05	U-Test OK	U-Test OK	U-Test OK

	(RTM-50[High]) x (TinySTM[High])	(RTM-50[High]) x (SwissTM[High])	(TinySTM[High]) x (SwissTM[High])
KS OK?	No	No	Yes
T-Test p	1.081496162054e-03	1.013995278489e-03	4.233386391873e-07
T-Test stat	3.55427743	3.57720585	5.72086270
T-Test p<=0.05	Not normally distrib.	Not normally distrib.	T-Test OK
U-Test p	2.598295687016e-12	2.598295687016e-12	1.067005555706e-06
U-Test stat	1110.00000000	1110.00000000	780.00000000
U-Test p<=0.05	U-Test OK	U-Test OK	U-Test OK

Figura 7 – Testes de Hipótese para amostras de alta contenção

	(RTM-5[Low]) x (Sequential[Low])	(RTM-5[High]) x (Sequential[High])
KS OK?	Yes	Yes
T-Test p	1.947179025543e-01	3.815633616626e-02
T-Test stat	1.31262167	2.12188836
T-Test p<=0.05	T-Test Failed	T-Test OK
U-Test p	2.088709239187e-01	4.757860204720e-02
U-Test stat	535.00000000	584.00000000
U-Test p<=0.05	U-Test Failed	U-Test OK

Figura 8 – Testes de Hipótese para o segundo conjunto de execuções

Tabela 6 – Resultados da análise estatística dos tempos de execução no primeiro conjunto testado.

	RTM-1 - Low	RTM-50-Low	TinySTM - Low	SwissTM - Low
count	30.0000000	30.0000000	30.0000000	30.0000000
mean	0.15813333	0.71324324	0.09640000	0.10370000
std	0.00084591	0.73444623	0.00055377	0.00069041
error (95%)	0.00026241	0.20384881	0.00017179	0.00021418
min	0.15700000	0.35300000	0.09600000	0.10300000
25%	0.15800000	0.35500000	0.09600000	0.10300000
50%	0.15800000	0.35700000	0.09600000	0.10400000
75%	0.15800000	0.35700000	0.09700000	0.10400000
max	0.16000000	2.25300000	0.09800000	0.10500000
KS stat	0.32928937	0.46948033	0.39828295	0.27801561
KS	0.00212453	0.00000005	0.00008335	0.01528637
KS p>0.05	Sample Bad	Sample Bad	Sample Bad	Sample Bad
SW stat	0.81561947	0.48656350	0.66927099	0.78072971
SW p	0.00012799	0.00000000	0.00000057	0.00002978
SW p>0.05	Sample Bad	Sample Bad	Sample Bad	Sample Bad
	RTM-1 - High	RTM-50 - High	TinySTM - High	SwissTM - High
count	30.0000000	30.0000000	30.0000000	30.0000000
mean	8.06883460	50.74735327	17.55868447	17.34454050
std	0.07509578	56.02583192	0.12975196	0.15426650
error (95%)	0.02329597	15.55021806	0.04025123	0.04785605
min	7.97327700	23.36720200	17.38628100	17.14186900
25%	8.01037900	23.54291700	17.44124400	17.21117700
50%	8.02558100	23.64089700	17.52216600	17.32612800
75%	8.10682000	24.27746600	17.60505800	17.38662600
max	8.21485600	188.70525100	17.82436600	17.81947600
KS stat	0.21768535	0.48627767	0.19473709	0.14981348
KS	0.09966094	0.00000001	0.17978156	0.46643610
KS p>0.05	Sample OK	Sample Bad	Sample OK	Sample OK
SW stat	0.85799658	0.49827093	0.89850789	0.91609687
SW p	0.00091592	0.00000000	0.00772318	0.02126906
SW p>0.05	Sample Bad	Sample Bad	Sample Bad	Sample Bad

Tabela 7 – Resultados da análise estatística dos tempos de execução no segundo conjunto testado.

	RTM-5 - Low	Mutex - Low	RTM-5 - High	Mutex - High
count	30.0000000	30.0000000	30.0000000	30.0000000
mean	46.82930643	46.70609250	56.59813103	56.51356410
std	0.31677747	0.39392897	0.14572012	0.15757172
error (95%)	0.09826967	0.12220336	0.04520482	0.04888138
min	46.11788700	45.60514900	56.28825400	56.19578000
25%	46.57983400	46.42393100	56.51786000	56.40603700
50%	46.84768100	46.77810400	56.58098400	56.51611000
75%	47.01332800	46.90058100	56.64980100	56.59235400
max	47.49371900	47.68687900	56.97457300	56.85947600
KS stat	0.09014939	0.13325698	0.11010598	0.07909818
KS	0.94953735	0.61377609	0.82179332	0.98435875
KS p>0.05	Sample OK	Sample OK	Sample OK	Sample OK
SW stat	0.96939051	0.95119727	0.97652072	0.98104280
SW p	0.52263588	0.18199947	0.72744626	0.85255736
SW p>0.05	Sample OK	Sample OK	Sample OK	Sample OK