INSTITUTO DE PESQUISAS TECNOLÓGICAS DO ESTADO DE SÃO PAULO

Mauro Romano Trajber

Monitoração de Eventos Relacionados à Memória Transacional

São Paulo

Junho/2011

Mauro Romano Trajber

Monitoração de Eventos Relacionados à Memória Transacional

Dissertação de Mestrado apresentada ao Instituto
de Pesquisas Tecnológicas do Estado de São Paulo,
como parte dos requisitos para a obtenção do
título de Mestre em Engenharia de Computação.

Data da aprovação: _____/_____/_______

____

Prof. Dr. Marco Dimas Gubitoso (Orientador)
IME - Instituto de Matemática e Estatística

Membros da banca examinadora:

Prof. Dr. Marco Dimas Gubitoso (Orientador)
IME – Instituto de Matemática e Estatística

Prof. Dr. Paulo Sérgio Muniz Silva (Membro)
IPT – Instituto de Pesquisas Tecnológicas do Estado de São Paulo

Prof. Dr. Marcel Jackowski (Membro)
IME – Instituto de Matemática e Estatística

Mauro Romano Trajber

Monitoração de Eventos Relacionados à Memória Transacional

Dissertação de Mestrado apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo, como parte dos requisitos para a obtenção do título de Mestre em Engenharia de Computação.
Área de concentração: Engenharia de software

Orientador:
Prof. Dr. Marco Dimas Gubitoso

São Paulo

Junho/2011

Ficha Catalográfica
Elaborada pelo Departamento de Acervo e Informação Tecnológica – DAIT
do Instituto de Pesquisas Tecnológicas do Estado de São Paulo - IPT

T768m **Trajber, Mauro Romano**
Monitoração de eventos relacionados à memória transacional./ Mauro Romano Trajber. São Paulo, 2011.
55p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Engenharia de Software.

Orientador: Prof. Dr. Marco Dimas Gubitoso

1. Transação 2. Memória transacional 3. Software Transactional Memory - STM 4. Controle de concorrência 5. Sistema de monitoração 6. Tese I. Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Coordenadoria de Ensino Tecnológico II. Título

11-45 CDU 004.424.71(043)

RESUMO

A utilização de arquiteturas paralelas, não somente em sistemas científicos e corporativos, mas também em sistemas domésticos, fez com que a programação *multithreading* tenha se tornado cada vez mais popular, obrigando os programadores a enfrentarem os problemas relacionados aos mecanismos clássicos de controle de concorrência. A memória transacional foi proposta como uma maneira de facilitar o desenvolvimento de programas paralelos, oferecendo ao programador a possibilidade de apenas marcar os trechos de código que devem ser executados de forma atômica e isolada. Neste trabalho, é proposta uma modificação para o *kernel* do sistema operacional Linux capaz de monitorar eventos relacionados à memória transacional como transações iniciadas, abortadas e finalizadas com sucesso, tornando possível observar o comportamento de um código que faz uso de memória transacional sem a necessidade de interromper sua execução e com mínima intrusão no tempo de usuário. As informações produzidas por este sistema de monitoração são publicadas em um sistema de arquivos virtual para que fiquem disponíveis às ferramentas de análise e monitoração que não possuem permissão para acessá-las diretamente, auxiliando o desenvolvedor a decidir se uma abordagem de memória transacional é melhor que outra ou ainda se o uso de memória transacional é vantajoso em relação ao uso de mecanismos clássicos de controle de concorrência.

Palavras-chave: Transações, Memória Transacional, Controle de concorrência.

ABSTRACT

Monitoring events related to transactional memory

The use of parallel architectures, not only in scientific and corporate, but also in domestic systems, made the multithread programming more popular, forcing programmers to face the problems related to concurrency control. The transactional memory was proposed as a way for easing parallel program development, providing the programmer the possibility to only mark the sections of code that are to be executed in an atomic and isolated way. In this work, it's proposed a Linux kernel modification that is able to monitor transactional memory events like started, aborted and commited transactions, making it possible to observe the behaviour of a program that uses transactional memory without the need to interrupt its execution and with low overhead in user time. The information produced by this monitoring system is published in a virtual file system to make them available for analysis and monitoring tools that do not have access permissions to access them directly, helping the developer to decide whether some transactional memory aproach is better than other or if the use of transactional memory is advantageous over the use of classical mechanisms of concurrency control.

Keywords: Transactions, Transactional Memory, Concurrency control.

Lista de Ilustrações

3.1 Representação de uma chamada de sistema. 34

4.1 Arquitetura do sistema de monitoração TMEM. 39

5.1 Ciclos de *clocks* necessários para execução da chamada de sistema. 48

5.2 Duração da chamada de sistema em espaço de kernel. 49

5.3 Tempo em espaço de usuário e em espaço de sistema 50

5.4 Microbenchmarks de árvore rubro-negra com e sem TMEM 51

Lista de Tabelas

4.1 Arquivos em cada diretório /proc/PID 42

4.2 Saída do TMEM para o mau uso da API 46

5.1 Eventos de uma árvore rubro-negra com memória transacional 50

Lista de Listagens

2.1 Problemas de composição . . . 19
2.2 Incremento atômico baseado na proposta de (HERLIHY et al., 2003) . . . 20
2.3 Uso de `guarded atomic blocks` . . . 21
2.4 Uso da primitiva `retry` . . . 21
2.5 Sentenças condicionais . . . 22
2.6 Aninhamento de transações . . . 23
3.1 Exemplo de uma classe com suporte a transações . . . 31
3.2 Macros da biblioteca RSTM . . . 32
3.3 Exemplo de uma chamada de sistema . . . 35
3.4 Definição dos identificadores das chamadas de sistema . . . 35
3.5 Exemplo de uma chamada de sistema . . . 35
3.6 Registro na tabela que define chamadas de sistema . . . 35
3.7 Nova entrada no Makefile . . . 35
3.8 Função create_proc_entry . . . 37
4.1 Alteração da task_struct . . . 40
4.2 Representação dos eventos . . . 41
4.3 Chamada de sistema do TMEM . . . 41
4.4 TMEM acessando informações sobre o processo em execução . . . 42
4.5 Conteúdo do arquivo stm_events . . . 42
4.6 Nova entrada no vetor tgid_base_stuff . . . 43
4.7 Alteração da macro da biblioteca RSTM . . . 45
4.8 Uso incorreto da API . . . 46

Lista de abreviaturas e siglas

CAS *Compare and swap*

HTM *Hardware Transactional Memory*

HyTM *Hybrid Transactional Memory*

MMU *Memory Management Unit*

RSTM *Rochester Software Transactional Memory*

SGBD Sistema Gerenciador de Bancos de Dados

STM *Software Transactional Memory*

TM *Transactional Memory*

TMEM *Transactional Memory Event Monitor*

Sumário

1 INTRODUÇÃO 12

1.1 Motivação 12

1.2 Objetivo 14

1.2.1 Objetivos específicos 14

1.3 Contribuições 14

1.4 Trabalhos relacionados 15

1.5 Organização 15

2 MEMÓRIA TRANSACIONAL 17

2.1 Introdução 17

2.1.1 Abstrações para programação paralela 17

2.1.2 Transações em memória e transações em bancos de dados 19

2.2 Construções transacionais 20

2.2.1 Introdução 20

2.2.2 Espera condicional 21

2.2.3 Sentenças condicionais 21

2.2.4 Aninhamento de transações 22

2.2.5 Ações irrevogáveis 23

2.3 Possíveis abordagens na construção de um sistema de TM 23

2.3.1 Implementação em hardware ou em software 24

2.3.2 Garantia de progresso 24

2.3.3 Controle de concorrência 25

2.3.4 Gerenciamento de versão 26

2.3.5 Detecção de conflitos 26

2.3.6 Gerenciamento de conflitos 27

2.4 Implementações de memória transacional com suporte em software 28

2.5 Depuração e monitoração 28

2.6 Quando a memória transacional é uma boa escolha 29

3 CONCEITOS FUNDAMENTAIS . . . 31
3.1 Implementação de memória transacional . . . 31
3.2 Modo usuário e modo kernel . . . 33
3.3 Chamadas de sistema . . . 33
3.3.1 Introdução . . . 33
3.3.2 Criando uma chamada de sistema . . . 34
3.4 Procfs . . . 36
3.4.1 Introdução . . . 36
3.4.2 Escrita no procfs . . . 36
3.4.3 Manipulando os arquivos no procfs . . . 37
4 SISTEMA DE MONITORAÇÃO TMEM . . . 38
4.1 Introdução . . . 38
4.2 Características . . . 39
4.2.1 Eventos monitorados . . . 40
4.3 Alterações no sistema operacional . . . 40
4.3.1 Concorrência em espaço de kernel . . . 41
4.3.2 Chamada de sistema . . . 41
4.3.3 Informações de saída . . . 42
4.3.4 TMEM como módulo do kernel . . . 43
4.4 Benchmark suites e microbenchmarks . . . 43
4.5 Alterações na biblioteca RSTM . . . 44
4.6 Extensibilidade . . . 45
4.7 Limites . . . 45
4.8 Possíveis usos . . . 46
5 ANÁLISE DOS RESULTADOS . . . 47
5.1 Introdução . . . 47
5.2 Avaliação da chamada de sistema . . . 47
5.2.1 Ciclos de clock . . . 47
5.2.2 Tempo . . . 48
5.3 Avaliação do sistema de memória transacional . . . 49
6 CONCLUSÕES E TRABALHOS FUTUROS . . . 52
6.1 Conclusões . . . 52

6.2 Trabalhos futuros 53
REFERÊNCIAS 54

1 INTRODUÇÃO

1.1 Motivação

Nas últimas décadas, o desenvolvimento de programas sequenciais foi suficiente para explorar os recursos dos processadores disponíveis. Grande parte do ganho de poder de processamento era alcançado aumentando a frequência de seus *clocks* exponencialmente, cerca de 40-50% ao ano. Porém, no início da última década percebeu-se que esta evolução teria um fim (OLUKOTUN; HAMMOND, 2005). Aumentar a frequência dos processadores tornou-se uma tarefa difícil devido a limitações físicas como alto consumo de energia e dissipação de calor.

Como alternativa a estas limitações, as arquiteturas paralelas tem se tornado cada vez mais populares, mas para que o software aproveite este aumento de eficiência é necessário que ele seja capaz de ser processado em paralelo. Uma maneira comum de se codificar programas que se beneficiam deste tipo de arquitetura é utilizar programação *multithreading*.

Infelizmente, mesmo depois de décadas de experiência com programação paralela, codificar programas paralelos ainda se mostra uma tarefa muito mais complicada do que codificar programas sequenciais. Algoritmos paralelos são mais difíceis de serem formulados e escritos corretamente do que algoritmos sequenciais. Um programa paralelo é mais difícil de ser projetado, escrito e depurado. Devido a sua execução não-determinística, falhas não-deterministicas ocorrem em programas paralelos, e estas falhas são mais difíceis de serem encontradas e corrigidas.

Além da dificuldade de projetar programas paralelos, a busca de defeitos por ferramentas que analisam e monitoram códigos é muito mais complexa quando o código analisado é paralelo.

Mecanismos de baixo nível de abstração como exclusão mútua e sincronização explícita são inadequados para construção de abstrações, pois sua natureza impede que sejam compostos em abstrações maiores e mais complexas (HARRIS et al., 2005). Um programa que usa uma abstração que contém sincronização explícita deve estar ciente destes detalhes, para evitar condições de corrida e *deadlocks*.

Enquanto paralelismo tem sido um problema difícil para programação, sistemas de bancos de dados tem explorado este recurso há décadas. Bancos de dados têm um bom desempenho executando várias consultas (*queries*) simultaneamente e em múltiplos processadores quando possível. Além disso, o modelo de programação de bancos de dados garante que um autor de um determinada consulta não precisa se preocupar com paralelismo. Muitos se perguntaram se o modelo usado em bancos de dados, com relativa simplicidade e grande sucesso, poderia também ser aplicado à programação.

No centro do modelo de programação de bancos de dados está a transação. Uma transação especifica uma semântica na qual uma computação é feita como se fosse a única que está acessando o banco de dados. Outras computações podem ser executadas simultaneamente, mas o modelo restringe a interação entre transações, de maneira que cada uma produza o mesmo resultado independentemente da ordem em que foram executadas. Como consequência, um programador que escreve um código de uma transação vive em um mundo mais simples e sequencial. Transações permitem que operações concorrentes acessem o mesmo banco de dados e ainda assim produzam resultados previsíveis e reproduzíveis.

Uma transação é uma sequência de ações que parecem indivisíveis e instantâneas para um observador externo, e para garantir tais características uma transação em banco de dados possui quatro atributos importantes: atomicidade, consistência, isolamento e durabilidade – conhecidos como ACID.

- *Atomicidade*: Garante que todas as ações que constituem uma transação sejam executadas com sucesso, ou então que nenhuma destas ações deixe rastros de sua execução.
- *Consistência*: O significado de consistência é depende do contexto onde é aplicado, geralmente são regras associadas a estruturas de dados. Por exemplo, um contador de registros em uma tabela deve ter exatamente a quantidade de registros, um atributo idade não deve ter um valor negativo. O fato é que o atributo consistência de uma transação garante que mesmo que os dados envolvidos sofram alteração sua consistência deve ser preservada. Uma transação deve iniciar e terminar em um estado consistente.
- *Isolamento*: Exige que uma transação não interfira em outras enquanto está sendo executada – independentemente de estarem ou não sendo executadas em paralelo. Esta propriedade faz com que transações sejam muito atrativas para modelos de programação concorrente.
- *Durabilidade*: Exige que uma vez que a transação finalize com sucesso (*commit*), seu resultado seja permanente e fique disponível para todas as transações subsequentes.

Um sistema transacional que implementa estas propriedades permite que um programador continue pensando sequencialmente sem se preocupar com concorrência. Transações são implementadas pelo Sistema Gerenciador de Bancos de Dados (SGBD), que esconde detalhes complexos de implementação por trás de uma interface simples.Transações também oferecem um mecanismo de abstração capaz de ser reutilizado na construção de computações mais complexas, fazendo com que a tarefa de compor transações seja simples e natural.

A popularização dos processadores *multicore* renovou o interesse em uma ideia antiga (LOMET, 1977), incorporar transações em modelos de programação usados para escrever programas paralelos. Esta ideia recebeu o nome de **memória transacional** – do inglês *Transactional Memory* (TM). Nesta pesquisa, Lomet observou que uma abstração similar a transações de bancos de dados poderia ser útil em linguagens de programação para garantir a consistência dos dados compartilhados entre vários processos. Seu trabalho não descreve uma implementação prática para esta tecnologia, mas em 1993 Herlihy e Moss propuseram (HERLIHY et al., 1993) uma implementação em hardware para estas transações. Mais tarde esta abordagem ficaria conhecida como *Hardware Transactional Memory* (HTM). Inspirados por este modelo, em 1995 Shavit e Touitou (SHAVIT; TOUITOU, 1995) propuseram uma alternativa capaz de

executar transações com suporte em software – *Software Transactional Memory* (STM) – sem a necessidade de utilização de uma arquitetura ou hardware específico.

Memória transacional é uma abordagem alternativa para coordenação de *threads* concorrentes e permite que um programa encapsule uma computação em uma transação. A atomicidade garante que uma transação complete com sucesso ou então seja abortada. O isolamento garante que uma transação produza sempre o mesmo resultado, independentemente de outras transações concorrentes.

Pesquisas sobre memória transacional em hardware (HTM), em software (STM) e até mesmo híbridas *Hybrid Transactional Memory* (HyTM) auxiliaram na criação de protótipos de implementações de sistemas de memória transacional que surgiram nos últimos anos, mas infelizmente apesar de trazer vários benefícios para o desenvolvimento de programas paralelos a TM não é uma panacéia. Mesmo utilizando esta tecnologia ainda é possível escrever programas paralelos incorretos e com desempenho ruim. Para solucionar este problema é necessário que o programador tenha em mãos ferramentas para monitorar seu código para que ele possa entender o comportamento de suas transações e os fluxos de execução de seu sistema. Uma maneira de atingir este objetivo é coletar informações obtidas em tempo de execução que podem ser utilizadas em uma análise mais aprofundada.

1.2 Objetivo

Este trabalho tem como objetivo monitorar, em tempo de execução e com baixa sobrecarga, os eventos relacionados à memória transacional em um sistema de STM e dispor as informações coletadas de forma conveniente para o usuário. A monitoração de eventos como transações finalizadas, transações abortadas e tentativas de reexecução, pode ajudar o desenvolvedor a entender melhor o comportamento de seu código em execução.

1.2.1 Objetivos específicos

A monitoração proposta nesta pesquisa é feita em espaço de sistema, onde os eventos são notificados por meio de uma chamada de sistema e as informações produzidas são publicadas no sistema de arquivos `procfs` para que fiquem disponíveis às ferramentas de análise e monitoração que não possuem permissão para acessá-las diretamente. Após a implementação deste sistema de monitoração, serão analisados os impactos no desempenho do sistema de memória transacional causados pela implementação desta nova funcionalidade.

1.3 Contribuições

As contribuições deste trabalho são as conclusões sobre a viabilidade do uso de um sistema de monitoração, que em tempo de execução, fornece informações a respeito de eventos relacionados a memória transacional. Estas conclusões foram obtidas por meio da análise do impacto causado no desempenho do sistema que implementa STM, avaliando-se a quantidade de ciclos de CPU e o tempo necessários para monitoração dos eventos. Além disso, como pro-

duto desta pequisa foi criado o *Transactional Memory Event Monitor* (TMEM), um sistema de monitoração de eventos relacionados à memória transacional.

1.4 Trabalhos relacionados

Em (LEV; MOIR, 2006) Lev e Moir propuseram uma maneira de depurar programas que utilizam memória transacional e em (ZYULKYAROV et al., 2010) foi proposta uma extensão para o depurador `WinDbg` para admitir programas que utilizam blocos atômicos (explicados mais adiante na seção 2.2) na linguagem C#. Estas pesquisas são focada no processo de depuração, que é executado durante o desenvolvimento de um programa. O diferencial deste trabalho em relação a estas pesquisas é que o sistema de monitoração proposto coleta as informações durante a execução dos programas que utilizam um sistema de STM e as publica para que ferramentas de análise possam observar o comportamento de programas que utilizam memória transacional sem a necessidade de interromper sua execução.

Em (LOURENCO et al., 2009), foi proposto um arcabouço para monitoração de sistemas de memória transacional, onde cada *thread* mantém as informações em uma região de memória privada, e quando o programa é finalizado todas estas informações são gravadas em em um único arquivo em disco. O diferencial deste trabalho em relação a esta pesquisa é que o sistema de monitoração proposto é executado pelo *kernel*, fazendo com que a maior parte do tempo seja computado como tempo de sistema. Além disso, é usado o sistema de arquivos *procfs*, que é um sistema de arquivos que reside em memória, para disponibilizar as informações sobre eventos, o que faz com que a escrita seja mais rápida do que em um dispositivo de bloco. Outra vantagem é que é possível ser notificado sobre um evento no momento em que ele ocorre e não somente quando o programa é finalizado.

1.5 Organização

Este trabalho está dividido da seguinte maneira:

- *Capítulo 1 - Introdução:* Neste capítulo foram apresentadas as motivações para memória transacional, os objetivos e contribuições deste trabalho.
- *Capítulo 2 - Memória Transacional:* Neste capítulo são apresentadas as principais características relacionadas a memória transacional e as pesquisas, protótipos e implementações mais relevantes para esta pesquisa.
- *Capítulo 3 - Conceitos fundamentais:* Neste capítulo são apresentadas as ferramentas que serão utilizadas para construção do sistema de monitoração e como esta pesquisa faz uso delas.
- *Capítulo 4 - Sistema de monitoração:* Neste capítulo são apresentadas a especificação da arquitetura e a implementação do protótipo do sistema de monitoração de eventos relacionados à memória transacional.

- *Capítulo 5 - Análise dos resultados:* Neste capítulo são apresentados e analisados os resultados dos testes de desempenho da implementação de STM com suporte a monitoração de eventos e os problemas encontrados durante o desenvolvimento do sistema de monitoração.
- *Capítulo 6 - Conclusões:* Neste capítulo é apresentado um resumo dos resultados desta pesquisa e serão apontados possíveis trabalhos futuros.

2 MEMÓRIA TRANSACIONAL

Neste capítulo serão apresentadas as principais características relacionadas a memória transacional e suas as principais abordagens. Serão apresentadas também as pesquisas, protótipos e implementações mais relevantes. Alguns conceitos contidos neste capítulo foram baseados na publicação de Tim Harris, James Larus, e Ravi Rajwar (HARRIS; LARUS; RAJWAR, 2010).

2.1 Introdução

A memória transacional é uma abordagem alternativa aos mecanismos clássicos de coordenação de linhas de execução (*threads*) concorrentes. Esta alternativa baseia-se na substituição dos mecanismos tradicionais de controle de concorrência como travas de exclusão mútua (*locks*) e monitores, e oferece uma semântica que permite que os programadores foquem-se onde seu código deve ser executado atomicamente e não em detalhes de baixo nível relacionados à concorrência.

As propriedades atomicidade, consistência e isolamento e durabilidade de uma transação são fundamentais para um sistema que implementa memória transacional. Embora o isolamento pareça ser a garantia mais importante, as outras propriedades, atomicidade e consistência também são importantes. A consistência é importante já que transações podem ser executadas em uma ordem imprevisível e seria difícil escrever programas sem poder assumir que uma transação vai iniciar em um estado consistente. Atomicidade é uma propriedade chave para garantir a consistência dos dados acessados e modificados por uma transação. Sem esta garantia, se uma transação falhar, ela poderia deixar os dados em um estado imprevisível e inconsistente, que se mantido, faria com as transações subsequentes também falhassem.

2.1.1 Abstrações para programação paralela

Uma das principais deficiências na programação paralela é a falta de mecanismos eficientes para abstração e composição – duas ferramentas fundamentais para lidar com complexidade. Uma abstração é uma visão simplificada de uma entidade, que captura as características fundamentais para entender e manipulá-la para um propósito particular. Abstrações escondem detalhes e complexidades irrelevantes, e permitem que humanos (e computadores) foquem nos aspectos relevantes de um problema específico. Composição é a capacidade de juntar duas entidades para formar uma segunda entidade maior e mais complexa, que por sua vez é abstraída em uma única entidade.

Linguagens modernas de programação suportam mecanismos poderosos de abstração e

também possuem bibliotecas que auxiliam a criação de programas sequenciais. Funções, por exemplo, oferecem uma maneira de nomear e encapsular uma sequencia de operações. Tipos abstratos de dados e objetos oferecem uma maneira de encapsular e nomear estruturas de dados mais complexas. Sistemas mais complexos como sistemas operacionais e bancos de dados relacionais também possuem níveis de abstração como memória virtual, sistema de arquivos, tabelas, colunas, etc.

Por outro lado, na programação paralela existem alguns mecanismos com direrentes níveis de abstração. Os mais usados ainda são mecanismos de baixo nível como *threads* e sincronização explícita, e são inadequados para construção de abstrações porque não é possível os compôr.

Por exemplo, considere uma tabela hash que possui métodos *thread-safe* `Insert` e `Remove`. Em um programa sequencial, cada uma destas operações pode ser uma abstração. Alguém pode utilizá-los sem conhecer detalhes da implementação da tabela hash. Agora, suponha que em um programa paralelo, nós queremos construir uma nova operação, chamada `Move`, que remove um item de uma tabela hash e o insere em outra tabela hash. O estado intermediário, em que nenhuma tabela contém o elemento, não deve ser visível por nenhuma outra *thread*. Não existe uma maneira de compor as operações `Insert` com o `Remove` já que elas travam a tabela hash somente durante suas execuções individuais. Para corrigir este problema é necessário criar novos métodos como `LockTable` e `UnlockTable`, o que quebraria a abstração expondo detalhes de implementação da tabela hash. Mais do que isso, estes métodos são propensos a erros. Um usuário que trava mais de uma tabela hash deve fazer isto de uma maneira ordenada e não se esquecer de destravá-las.

O mesmo problema pode acontecer em outras formas de composição. Suponha que uma função `f1` receba duas filas e espera por um elemento de qualquer uma destas filas, e faz o uso de uma função `A` bloqueante que recebe uma lista de filas e aguarda um elemento em qualquer uma delas. Uma segunda função `f2` deve fazer o mesmo com duas filas diferentes. Não é possível compor `f1` e `f2` para criar uma abstração maior capaz de esperar por um elemento em qualquer uma das quatro filas (Listagem 2.1). Para resolver este problema, programadores usam técnicas que prejudicam a modelagem de seu código, como passar as filas usadas nos níveis baixos de abstração (`f1` e `f2`) para uma `WaitAny` externa, que fica responsável em fazer um *call-back* para a função apropriada. Mais uma vez, duas abstrações corretas, `f1` e `f2`, não podem ser compostas em uma abstração maior; ao invés disso elas precisaram ter suas lógicas externalizadas.

Listagem 2.1: Problemas de composição

```
WaitAny(fila1, fila2) {
  if (fila1.empty() || fila2.empty()) {
    thread.wait();
  }
  // se nao estiver vazia consome...
}

f1(fila1, fila2) {
  WaitAny(fila1, fila2);
}

f2(fila3, fila4) {
  WaitAny(fila3, fila4);
}

compor() {
  // Impossivel esperar um elemento em
  // qualquer uma das quatro filas.
  // O codigo fica bloqueado logo na
  // primeira chamada a f1()
  f1(fila1, fila2);
  f2(fila3, fila4);
}
```

2.1.2 Transações em memória e transações em bancos de dados

Transações em memória diferem das utilizadas em bancos de dados, e consequentemente requerem novas técnicas de implementação. As diferenças listadas a seguir estão entre as mais importantes:

- Dados em um banco de dados geralmente residem em disco. Os acessos aos discos disponíveis atualmente levam de 3ms a 15ms, isto é tempo suficiente para o processador executar milhões de instruções. Bancos de dados podem, portanto, executar instruções enquanto acessam o disco. Sistemas que implementam memórias transacionais fazem acessos a memória, que resultam em um custo de no máximo algumas centenas de instruções, e desta forma uma transação não fica livre para executar mais instruções a cada acesso a memória.

- Memória transacional é normalmente volátil, não resiste ao fim do programa. Isso simplifica as implementações de memória transacional, já que a necessidade de gravar registros permanentemente no disco antes da transação ser efetivada torna o sistema de bancos de dados mais complexo.

- Um banco de dados provê um único modo de acessar os dados que ele gerencia (por meio do SGBD); consequentemente, quem implementa um banco de dados é livre para

escolher como representar seus dados, como controlar a concorrência aos dados, etc. Com memória transacional, o programador normalmente pode fazer acessos comuns à memória e também acessos transacionais.

- Para ser adotada amplamente a memória transacional precisa coexistir com a infraestrutura existente: linguagens, paradigmas, bibliotecas, sistemas operacionais. Programadores resistirão em adotar memória transacional caso sejam necessárias mudanças radicais nas linguagens de programação, bibliotecas ou sistemas operacionais – ou ainda, se for muito restritiva, limitando a apenas um ambiente fechado, exigindo que os dados sejam acessados somente por meio de transações, como nos SGBD.

2.2 Construções transacionais

Nesta seção serão apresentadas as características de algumas implementações, ideias e projetos propostos relacionados a memória transacional e que são resultado de pesquisas recentes.

2.2.1 Introdução

Tendo as propriedades fundamentais para um sistema que implementa memória transacional como ponto de partida, diversas implementações e funcionalidades diferentes têm surgido nos últimos anos. Naturalmente, os protótipos de memória transacional também se diferenciam na sua sintaxe. A abordagem proposta em (HERLIHY et al., 2003) utiliza métodos definidos em uma biblioteca que controlam as transações que estão em progresso bem como as regiões de memória acessadas por elas (Listagem 2.2). Outra implementação proposta em (HARRIS et al., 2005) introduziu a palavra chave `atomic` para representar uma transação. Os autores o chamaram de *bloco atômico* (Listagem 2.4). Nesta abordagem o programador fica responsável em apenas identificar as regiões do seu código que devem ser executadas atomicamente, e a implementação fica responsável por detectar e gerenciar as regiões de memória envolvidas em uma transação.

Listagem 2.2: Incremento atômico baseado na proposta de (HERLIHY et al., 2003)

```
1  void increment() {
2    do {
3      // inicia a transacao
4      StartTx();
5      // le a variavel 'counter' de forma transacional
6      ReadTx(counter);
7      // incrementa variavel 'counter' de forma transacional
8      WriteTx(counter + 1);
9      // finaliza a transacao
10   } while (!CommitTx());
11 }
```

2.2.2 Espera condicional

Uma construção básica de blocos atômicos não permite uma maneira de expressar esperas condicionais, onde uma linha de execução dentro de um bloco atômico precisa esperar até que certa condição seja satisfeita. Isto limita a aplicabilidade de blocos atômicos – fica impossível, por exemplo, escrever um código no modelo produtor-consumidor. Duas pesquisas propuseram maneiras diferentes de solucionar este problema: a primeira, (HARRIS; FRASER, 2003), com *guarded atomic blocks* e a segunda, (HARRIS et al., 2005) com um o comando `retry`. Analogamente, em (ADL-TABATABAI et al., 2006) foi proposta a implementação desta mesma idéia na linguagem Java.

Listagem 2.3: Uso de `guarded atomic blocks`

```
1 void consume(){
2   atomic (!buffer.isEmpty()) {
3     Object x = buffer.getElement();
4     ...
5   }
6 }
```

Com os *guarded atomic blocks*, a execução de uma transação é postergada até que a condição seja satisfeita. A semântica do comando `retry` é simples: se o `retry` for executado a transação é abortada e será executada novamente mais tarde. Seria ineficiente executar a transação imediatamente após o `retry` pois provavelmente o estado das condições ainda seria o mesmo. Em uma implementação eficiente a transação é abortada e só é executada novamente quando alguma das variáveis lidas anteriormente for modificada por outra transação. No exemplo listado em 2.4, se o comando `retry` for executado, a transação será executada novamente somente quando o `buffer` não estiver vazio. A primitiva `retry` se mostra mais flexível do que os *guarded atomic blocks* visto que a condição de guarda pode ser mais complexa do que uma simples expressão, como um laço (*loop*), por exemplo.

Listagem 2.4: Uso da primitiva `retry`

```
1 void consume(){
2   atomic {
3     if (buffer.isEmpty()) retry;
4     Object x = buffer.getElement();
5     ...
6   }
7 }
```

O código exibido na listagem 2.4 possui uma vantagem com relação aos mecanismos clássicos de controle de concorrência, pois não é necessário fazer sinalizações explícitas no modelo `wait/notify`.

2.2.3 Sentenças condicionais

Em algumas situações é necessário expressar alternativas na construção de uma transação. A construção `orElse`, proposta por Harris em (HARRIS et al., 2005), pode ser usada para

definir que uma transação seja executada caso a transação que a precede não seja concluída. Se `X` e `Y` são transações e `{X orElse Y}`, as seguintes regras são aplicáveis:

- Se a transação `X` é finalizada com sucesso, a primitiva `orElse` termina sem executar `Y`.
- Se `X` executa um comando `retry`, `X` é abortada e `Y` é executada.
- Se `Y` é finalizada com sucesso, `orElse` termina com sucesso.
- Se `Y` executa um comando `retry`, todo o comando `orElse` executa o `retry`

Suponha um código que resgata um elemento de qualquer uma de duas filas que possuem uma operação `getElement` transacional. Esta operação executará a primitiva `retry` caso a fila esteja vazia. Na listagem 2.5, é feita uma tentativa de resgatar um elemento da `fila1`; caso a fila esteja vazia e não seja possível resgatar um elemento, esta transação é abortada e é feita uma tentativa de resgatar um elemento da `fila2`. Se ambas filas estiverem vazias todo o código é executado novamente a partir do primeiro bloco `atomic`.

Listagem 2.5: Sentenças condicionais

```
1 atomic {
2   do {
3     x = fila1.getElement();
4   } orElse {
5     x = fila2.getElement();
6   }
7 }
```

2.2.4 Aninhamento de transações

Uma característica bem conhecida das transações utilizadas em bancos de dados é a possibilidade de aninhar transações. Esta característica é fundamental para compor transações, o que é essencial para construção de abstrações maiores e mais complexas. Existem três modos de aninhamento entre transações aninhadas (AGRAWAL; LEISERSON; SUKHA, 2006):

- *Flattened Nesting*: As modificações da sub-transação só serão visíveis quando a transação que a envolve terminar com sucesso. Abortar uma sub-transação faz com que a transação mais externa também seja abortada.
- *Closed Nesting*: Esta abordagem é similar a *flattened*, exceto quando a sub-transação aborta. Neste caso a transação que a envolve não é abortada e o controle volta para a transação externa. As modificações da sub-transação só serão visíveis quando a transação mais externa termine com sucesso. Quando uma sub-transação aborta, somente as modificações desta sub-transação são desfeitas.
- *Open Nesting*: Neste caso as sub-transações são consideradas independentes das transações que as envolvem. Assim que uma sub-transação termina com sucesso suas modificações são efetivadas e ficam visíveis para todas as outras transações imediatamente. Caso a sub-transação aborte, a transação que a envolve não sofre nenhum efeito.

Listagem 2.6: Aninhamento de transações

```
1 int x = 1;
2 atomic {
3   x = 2;
4   atomic {
5     x = 3;
6     abort();
7     ...
8   }
9 }
```

2.2.5 Ações irrevogáveis

Um dos maiores desafios na utilização de transações é a comunicação com entidades externas ao controle do sistema de memória transacional. Uma vez que transações podem abortar a qualquer momento é importante que toda ação dentro de uma transação possa ser desfeita. Por outro lado, sistemas operacionais modernos possuem diversos mecanismos como chamadas de sistema, manipulação de arquivos, acesso a bancos de dados, comunicação entre processos, comunicação em rede etc. Estas ações não são reversíveis. Sistemas que utilizam memória transacional são capazes de reverter somente operações em memória. Desta forma operações de entrada e saída não devem ser usadas dentro de uma transação, caso contrário esta ação poderá ser executada múltiplas vezes ou ser executada mesmo se a transação abortar.

Não existe mecanismo genérico para tratar estes tipos de operações. Algumas pesquisas (SPEAR; MICHAEL; SCOTT, 2008; SPEAR et al., 2008; WELC; SAHA; ADL-TABATABAI, 2008) propõem um tratamento caso-a-caso.

Um sistema de memória transacional pode, por exemplo, armazenar as operações como escrita em arquivos, até que a transação termine com sucesso e então escreva as alterações em disco. Similarmente, um sistema também pode armazenar os dados digitados por um usuário e reutilizá-los caso a transação aborte e precise ser executada novamente. Por outro lado, esta abordagem pode causar graves problemas de concorrência se o recurso estiver sendo acessado concorrentemente por outras *threads* ou processos.

Outra abordagem é permitir somente operações de entrada e saída se estas operações suportarem uma semântica transacional. Desta forma o sistema de memória transacional passa a confiar em uma outra camada capaz de reverter as modificações. Em (PORTER et al., 2009) foi proposta uma modificação do sistema operacional Linux, que implementa transações e faz com que seja possível implementar o sistema de arquivos `ext3` de forma transacional.

2.3 Possíveis abordagens na construção de um sistema de TM

Vários protótipos de memória transacional tem sido implementados nos últimos anos. Estes protótipos divergem em muitos aspectos, e ainda não se estabeleceu um consenso sobre quais são as melhores abordagens. Existem algumas decisões que devem ser feitas quando se constrói um sistema de memória transacional – técnicas de sincronização, granularidade de detecção de

conflitos, etc. Nesta seção serão abordados os possíveis aspectos relacionados a construção de um sistema de memória transacional.

2.3.1 Implementação em hardware ou em software

Um sistema de memória transacional pode ser baseado apenas no uso de bibliotecas ou em uma linguagem específica – *Software Transactional Memory* (STM)) – ou ser suportado por um hardware – *Hardware Transactional Memory* (HTM). Implementações de STM usam técnicas de implementação para as arquiteturas mais comuns. As implementações de HTM utilizam instruções e recursos específicos de algumas arquiteturas como o *cache coherence protocol* ou o buffer de escrita do processador. Isso faz com que estas implementações sejam possíveis em apenas algumas arquiteturas.

Implementações de STM possuem a vantagem de serem mais flexíveis e mais fáceis de serem modificadas em relação as implementações de HTM. As implementações de STM também são mais simples de serem integradas em linguagens de programação já existentes e possuem menos limitações relacionadas ao hardware, como tamanho do cache por exemplo.

Já as implementações de HTM, normalmente, são capazes de diminuir a sobrecarga associada a sincronização, detecção e gerenciamento de conflitos que existem nas implementações de STM.

2.3.2 Garantia de progresso

Quando uma transação é executada espera-se que ela progrida, e não apenas seja executada o mais rápido possível. Para isso, é necessário que um sistema de memória transacional ofereça alguma garantia de progresso para as transações. Neste contexto, progresso significa que o sistema de memória transacional deve garantir que uma transação seja executada até ser finalizada com sucesso (*commit*), não importando se outras *threads* também estão em execução. Quando uma *thread* executa um acesso transacional (leitura ou escrita), ela o faz por meio do sistema de memória transacional. Todo acesso precisa ser mediado para que o sistema de memória transacional possa sincronizar os acessos concorrentes aos recursos compartilhados, podendo assim garantir o progresso de todas as transações.

Esta sincronização pode ser feita de forma bloqueante, na qual uma trava é utilizada para evitar que outras *threads* acessem um recurso compartilhado. Desta forma os recursos são travados antes de serem acessados e destravados assim que a operação terminar. Neste caso um algoritmo de gerenciamento de contenção deve ser utilizado para evitar *deadlocks* – estes algoritmos serão explicados mais adiante na seção 2.3.6.

A sincronização também pode ser feita de forma não-bloqueante, e este caso três diferentes algoritmos que garantem o progresso das transações podem ser utilizados:

O primeiro algoritmo é conhecido como *wait-freedom* (livre de espera) (HERLIHY, 1991), que exige que uma *thread* progrida caso ela esteja em execução. Este tipo de implementação é mais agressivo e tem, geralmente, um desempenho pior sendo aplicada somente a alguns problemas específicos onde uma imparcialidade entre *thread* não se faz necessária.

O segundo é o *lock-freedom* (livre de travas), que exige que se alguma *thread* estiver em

execução então **alguma** *thread* progrida. Essa garantia é mais fraca do que a anterior, mas também não garante justiça entre as *threads*.

O terceiro algoritmo não-bloqueante que garante o progresso das transações é o *obstruction-freedom* (HERLIHY, 2003), que garante que uma *thread* faça progresso caso ela seja a única *thread* que esteja em execução. Uma *thread* que sofreu preempção não deve afetar o progresso de uma *thread* em execução. Este tipo de garantia, por si só, não previne um *livelock* – se uma *thread* T1 que sofreu preempção possui uma trava que a *thread* T2 precisa, T2 não possui opção a não ser aguardar a liberação da trava – neste caso também se faz necessário a utilização de um gerenciamento de contenção.

Existem pesquisas que defendem técnicas bloqueantes e não-bloqueantes. Ennals argumentou que *obstruction-freedom* não é uma propriedade importante para memória transacional e demonstrou que uma implementação de STM sem esta propriedade pode ter um desempenho melhor (ENNALS; ENNALS, 2006). Em (MARATHE; MOIR, 2008) os resultados mostraram que é possível melhorar o desempenho de sistemas de memória transacional que utilizam técnicas não-bloqueantes a ponto de terem o seu desempenho comparado ao de implementações bloqueantes.

2.3.3 Controle de concorrência

Existem duas abordagens para o controle de concorrência: controle pessimista e controle otimista. Estes controles atuam sobre os três eventos principais relacionados a conflitos. São eles:

- Um conflito **ocorre** quando duas transações realizam operações conflitantes sobre os mesmos dados, ou existe um acesso concorrente de escrita, ou uma escrita é feita por uma transação e uma leitura por outra.
- Um conflito é **detectado** quando o sistema de memória transacional determina que um conflito ocorreu. A detecção de conflitos será explicada com mais detalhes na seção 2.3.5.
- O conflito é **resolvido** quando o sistema de memória transacional ou o código da transação toma alguma ação para evitar que o conflito ocorra, geralmente abortando uma das transações conflitantes.

Com o *controle pessimista de concorrência*, todos os três eventos listados acima ocorrem no mesmo ponto de execução: quando a transação tenta acessar um dado compartilhado, o sistema detecta o conflito, e então o resolve. Este tipo de controle de concorrência permite que uma transação tenha acesso exclusivo a um dado compartilhado, impedindo que outras transações o acessem concorrentemente.

Com o *controle otimista de concorrência*, a detecção e resolução podem acontecer depois que um conflito ocorrer. Este tipo de controle de concorrência permite que múltiplas transações acessem os dados compartilhados paralelamente e continuem sendo executadas mesmo se um conflito ocorrer. O sistema de memória transacional irá detectar e resolver estes conflitos antes da transação terminar.

Se os conflitos são frequentes, então o controle pessimista de concorrência pode ser vantajoso: uma vez que a transação pode travar o acesso aos dados compartilhados, podendo assim

prosseguir até ser finalizada. Caso contrário, se os conflitos são raros, o controle otimista de concorrência pode ter um desempenho melhor, afinal não é necessário ter o custo de travar e liberar os acessos aos dados compartilhados, oferecendo assim uma possibilidade maior de paralelismo entre as transações.

2.3.4 Gerenciamento de versão

Sistemas de memória transacional precisam de mecanismos para gerenciar os dados que são escritos de forma especulativa por uma transação, onde a escrita em memória é feita em uma região temporária e caso a transação finalize com sucesso as mudanças feitas devem ser efetivadas, caso contrário devem ser desfeitas. Nas implementações de STM existem duas abordagens principais para gerenciar modificações efetuadas por uma transação.

A primeira abordagem é conhecida como gerenciamento prematuro (*eager version management*) (MOORE et al., 2006). Quando acontece uma operação de escrita, o dado compartilhado é travado, utilizando-se o controle pessimista de concorrência, e o seu valor é sobrescrito diretamente na memória. O valor antigo é gravado em um registro de recuperação chamado *undo-log*. Este registro permite que valores antigos sejam resgatados caso a transação aborte.

A segunda abordagem é conhecida como gerenciamento tardio (*lazy version management*), no qual a modificação feita pela transação só é efetivada quando a transação for completada. A transação mantém suas escritas em um registro que deve ser consultado a cada leitura feita pela transação. Quando a transação é finalizada com sucesso, os valores gravados neste registro são copiados para a memória. Se a transação abortar este registro é simplesmente descartado.

2.3.5 Detecção de conflitos

Com o controle pessimista de concorrência a detecção de conflitos é mais simples porque a trava garante que nenhuma outra *thread* fará acessos concorrentes a esta região de memória.

Por outro lado, se for utilizado o controle otimista de concorrência a detecção passa a ser mais complexa. Nestes sistemas existe um processo de validação de uma transação, no qual um possível conflito é detectado; se a validação for bem sucedida então a operação é executada.

A detecção de conflitos pode ser classificada quanto a granularidade e tempo.

No primeiro aspecto, granularidade, o conflito entre transações pode ser detectado com uma granularidade fina, no nível do tamanho da palavra da arquitetura como proposto por Harris em (HARRIS; FRASER, 2003), ou com uma granularidade maior usando, por exemplo, objetos em uma linguagem de programação orientada a objetos como prosposto em (HERLIHY et al., 2003).

Uma detecção de conflitos baseada no tamanho da palavra da arquitetura os conflitos são menos frequentes, mas por outro lado é necessário que o sistema de memória transacional trave e libere cada quatro bytes (supondo uma arquitetura de 32 bit) acessados por uma transacão, podendo prejudicar o desempenho de uma transação que acesse uma grande quantidade de dados como um objeto por exemplo.

Já com uma detecção de conflitos baseada em objetos, acessos concorrentes a atributos de

um mesmo objeto feitos por diferentes transações resultarão em falso um conflito, mas por outro lado o sistema de memória transacional não precisa travar e destravar cada palavra acessada pela transação, diminuindo assim a sobrecarga imposta por sistemas de memória transacional que utilizam detecção de conflito com uma granularidade no tamanho da palavra da aquitetura.

No segundo aspecto, tempo, um conflito pode ser detectado em três momentos diferentes:

- Quando uma transação tenta acessar um dado compartilhado. Esta abordagem também é conhecida como *eager conflict detection* (MOORE et al., 2006)
- Durante a validação, quando a transação verifica se os conjuntos de dados lidos e escritos foram modificados por outras transações. O processo de validação pode ocorrer a qualquer momento, e até mesmo várias vezes, durante a execução de uma transação.
- Somente ao tentar finalizar uma transação verificam-se os dados lidos e escritos. Esta abordagem também é conhecida como *lazy conflict detection* (MOORE et al., 2006)

2.3.6 Gerenciamento de conflitos

Duas transações são conflitantes se acessam a mesma variável e se pelo menos uma destas transações efetuar uma operação de escrita. Para que uma transação possa acessar uma variável, primeiramente precisa adquirir o acesso a ela. No momento de aquisição as transações são visíveis umas às outras. Os conflitos entre transações causam uma perda de desempenho em um sistema de memória transacional, e uma maneira de atenuar esta perda de desempenho é utilizar um *contention manager* (CM). Um CM implementa uma ou mais políticas de resolução de conflitos. Quando ocorre um conflito, estas políticas selecionam qual transação será prejudicada (abortará e será executada novamente mais tarde). Alguns estudos definem políticas de gestão de conflito (GUERRAOUI; HERLIHY; POCHON, 2005b; GUERRAOUI; HERLIHY; POCHON, 2005a; SCHERER III; SCOTT, 2005).

As estratégias mais comuns utilizadas por gerenciadores de conflito são:

- *Passiva*: A transação que tenta adquirir o acesso a uma variável conflitante é abortada e é executada novamente.
- *Educada*: A transação que tenta adquirir o acesso a uma variável conflitante aguarda por um período de tempo que aumenta exponencialmente. Após cada intervalo, a transação verifica se a outra transação já terminou de manipular a variável conflitante.
- *Karma*: Nesta estratégia o gerenciador de conflitos armazena o número de vezes que cada transação acessou determinada região de memória e assume este valor como uma prioridade. Uma transação que possui um *karma* (prioridade) menor do que outra tenta adquirir o acesso a uma variável *N* vezes (onde *N*: é a diferença entre as prioridades) antes de desistir e abortar.
- *Eruption*: Similar a *Karma*, exceto pelo fato de adicionar o *karma* da transação abortada ao *karma* da transação em execução, diminuindo assim as chances de uma terceira transação abortar a transação corrente.

- *Polka*: Uma combinação da estratégia Educada (Polite) com a *Karma* onde o *karma* da transação abortada é usado para calcular o tempo que esta transação aguardará.
- *Tempo de vida* O gerenciador simplesmente aborta a transação mais nova.

O gerenciamento de conflitos não afeta a eficácia de uma implementação de memória transacional, apenas seu desempenho. Outro desafio na área de gerenciamento de conflitos é criar uma maneira de combinar diferentes algoritmos em um mesmo programa e poder escolher a estratégia mais adequada em tempo de execução podendo beneficiar as transações que estão sendo executadas.

2.4 Implementações de memória transacional com suporte em software

Existem diversas implementações de memória transacional com suporte em software. A seguir serão listadas as mais relevantes para esta pesquisa.

- Rochester Software Transactional Memory (RSTM) (MARATHE et al., 2006) é um conjunto de implementações de sistemas de STM em C++. Atualmente são treze implementações diferentes de STM, e não é necessária nenhuma modificação no compilador. O RSTM suporta diversas arquiteturas (x86, SPARC, Itanium) e sistemas operacionais (Linux, Solaris, AIX, Windows, Mac OS X) e utiliza uma granularidade de detecção de conflitos no tamanho de um objeto.
- Transactional Locking II (TL2) (DICE; SHALEV; SHAVIT, 2006) é uma implementação de STM escrita em C que não necessita de modificações nas funções de alocação e liberação de memória (malloc/free) e utiliza uma granularidade do tamanho da palavra da aquitetura. Originalmente foi desenvolvida para arquitetura SPARC e sistema operacional Solaris, mas foi modificada para suportar as arquiteturas x86 de 32 ou 64-bit e sistema operacional Linux.
- TinyTM (FELBER; FETZER; RIEGEL, 2008) é uma implementação de STM que utiliza uma granularidade do tamanho da palavra da arquitetura. Esta implementação suporta arquiteturas de 32 ou 64-bit e diversos sistemas operacionais como Linux, BSDs, Mac OS X e Windows. Esta implementação utiliza uma biblioteca chamada `atomic_ops` para implementar operações atômicas.

2.5 Depuração e monitoração

Atualmente, todos os conceitos que envolvem memória transacional como blocos atômicos, transações finalizadas, transações abortadas, reexecução, etc. não são suportados pelas ferramentas que são utilizadas atualmente para depuração de programas sequenciais e paralelos.

Programas executados em paralelo são fundamentalmente mais difíceis de serem depurados do que programas sequenciais, pois são executados em uma ordem não-deterministica na qual

erros não-determinísticos podem acontecer. Depurar programas que utilizam memória transacional também não é uma tarefa simples. A implementação de memória transacional pode ter suporte em hardware ou em software, o que muda completamente o comportamento de uma ferramenta de depuração.

A depuração inevitavelmente atrasa a execução de um código. Este fato pode ser irrelevante quando o código depurado é sequencial, mas é crítico para um código paralelo. Se uma *thread* tem sua execução atrasada devido a instruções de depuração, os problemas relacionados a concorrência podem simplesmente não acontecer durante o processo de depuração.

Em (LEV; MOIR, 2006), Lev e Moir discutiram os desafios relacionados à criação de uma ferramenta de depuração para memória transacional. Em um trabalho mais recente Herlihy e Lev criaram um sistema chamado `tm_db` (HERLIHY; LEV, 2009) capaz de fornecer uma interface comum de depuração para diversas implementações de memória transacional. Já em (LEV; MOIR, 2006) Lev e Moir propuseram uma maneira de depurar programas que utilizam memória transacional e em (ZYULKYAROV et al., 2010) foi proposta uma extensão para o depurador `WinDbg` para suportar programas que utilizam blocos atômicos na linguagem C#.

Em (CHAFI et al., 2005) Chafi apresentou uma ferramenta de monitoração para sistemas de memória transacional com suporte em hardware que pode ser utilizada para rastrear e detectar, em tempo de execução, gargalos que prejudicam o desempenho de um *software* que utiliza memória transacional.

Nesta pesquisa é apresentado um sistema de monitoração de eventos relacionados à memória transacional com suporte em software. Este sistema de monitoração será responsável em capturar eventos como transações abortadas, transações finalizadas e tentativas de reexecução e publicá-las, por meio de uma chamada de sistema, no sistema de arquivos `procfs`, auxiliando o desenvolvedor a decidir se o uso de memória transacional é vantajoso para sua aplicação em relação ao uso de mecanismos clássicos de controle de concorrência. O desenvolvedor poderá também verificar se fez uso correto de memória transacional, se todas as transações estão sendo finalizadas ou se muitos conflitos ocorrem em determinado trecho de código protegido por uma transação, por exemplo.

2.6 Quando a memória transacional é uma boa escolha

Memória transacional pode ser utilizada para manipular estruturas de dados em memória compartilhada onde o uso de travas prejudica a escalabilidade. Um exemplo disto é a implementação de Zyulkyarov (ZYULKYAROV et al., 2009) de servidor para o jogo Quake usando memória transacional ao invés de travas. Ao modelar o efeito de um movimento de um jogador durante o jogo, a implementação baseada em travas precisava simular o efeito desta operação para determinar qual objeto precisaria ser travado. Feita a simulação, o jogo travaria os objetos, verificaria se o movimento do jogador ainda é válido, e então realizaria a ação. Com TM, a estrutura do código foi simplificada porque o passo que faz a simulação foi descartado.

O uso de memória transacional também é vantajoso em situações nas quais a modelagem e menutenibilidade de um código são mais importantes que seu desempenho. Um código paralelo escrito com memória transacional é mais fácil de ser entendido e não obriga o programador a modificar a modelagem de seu sistema somente para contornar as limitações dos mecanismos tradicionais de controle de concorrência.

Por outro lado, transações não são uma panacéia. Mesmo com memória transacional o programador ainda precisa decidir quais trechos de seu código poderão ser executados paralelamente e por este motivo ainda é fácil escrever programas concorrentes incorretos. É possível, por exemplo, escrever transações muito curtas: no caso de uma operação `Move`, podem ser tratadas as operações `Insert` e `Remove` em duas transações separadas, ao invés de uma transação que combinaria as duas operações. Pode-se também escrever transações muito longas: onde o estado intermediário entre duas transações precisa ser visível e o programador decide compor estas duas transações em apenas uma, impossibilitando tal funcionalidade. Também é muito fácil escrever transações de forma incorreta, iniciando uma transação mas esquecendo de finalizá-la (*commit*), como em um encadeamento, por exemplo.

Finalmente, e particularmente nas implementações iniciais, o desempenho do código executado por uma transação pode ser pior do que um código paralelo que utiliza os mecanismos clássicos para controle de concorrência. Um dos propósitos do paralelismo normalmente é o aumento de desempenho, portanto, antes de utilizar qualquer implementação de memória transacional o programador deve ter certeza de que o ganho de desempenho graças ao paralelismo supera as penalidades da sincronização impostos pela memória transacional, e ferramentas capazes de monitorar o comportamento das transações podem auxiliá-lo a tomar este tipo de decisão.

3 CONCEITOS FUNDAMENTAIS

Este capítulo introduz um conjunto de conceitos relacionados ao sistema de monitoração proposto nesta pesquisa.

3.1 Implementação de memória transacional

Para que seja possível comparar o desempenho de diversas implementações de STM a implementação de memória transacional que será utilizada como base para esta pesquisa é a *Rochester Software Transactional Memory* (RSTM) (MARATHE et al., 2006). A RSTM é um biblioteca composta por um conjunto de treze diferentes implementações de memória transacional, o que possibilita utilizar o sistema de monitoração em diferentes implementações de STM sem a necessidade de reescrever o código da aplicação. RSTM é uma implementação de memória transacional com suporte em software e implementada na linguagem C++, e isto faz com que a execução de chamadas de sistema seja mais simples e direta, sem a necessidade de passar por uma máquina virtual por exemplo.

A RSTM adota a granularidade de detecção de conflitos no tamanho de um objeto (seção 2.3.5) e oferece a possibilidade de utilizar implementações bloqueantes e não-bloqueantes (seção 2.3.2.

A biblioteca RSTM adiciona um cabeçalho único e que é válido por todo o ciclo de vida do objeto. Este cabeçalho é composto por dois ponteiros: um para o descritor da transação ao qual o objeto pertence e outro para a versão antiga do objeto. Uma transação possui autoridade sobre um objeto somente se o descritor da transação estiver com o estado *ACTIVE*. Se o descritor estiver com o estado *ABORTED* então a versão atual do objeto é a apontada como sendo a versão antiga do objeto. Caso o estado for *COMMITED* então o objeto está na sua versão correta. A troca de estado é feita utilizando uma instrução atômica do tipo comparar-e-trocar (*Compare and swap* (CAS))

Quando uma transação realiza escritas em um objeto, o faz na verdade em uma cópia deste objeto, retornada pelo método `clone()` obrigatoriamente implementado por todos objetos transacionais que são gerenciados pela RSTM.

Atualmente, a biblioteca RSTM admite somente programas que utilizam `pthreads`. Qualquer objeto compartilhado entre *threads* deve herdar a classe `stm::Object`. Esta classe adiciona os metadados necessários para um objeto transacional. Para garantir que os valores lidos e escritos em um objeto transacional estão em um estado consistentes, a RSTM faz esta validação com os métodos `getters` e `setters` dos objetos. Estes métodos devem ser gerados pela macro `GENERATE_FIELD`.

Listagem 3.1: Exemplo de uma classe com suporte a transações

```
1 class Counter : public stm::Object {
2     GENERATE_FIELD(int , value);
3     ...
4 }
```

Uma *thread* deve executar o método `stm::init()` antes de executar a primeira transação. Este método inicializa alguma regras relacionadas ao sistema de memória transacional, como por exemplo o gerenciador de contenção que será utilizado.

Dentro da RSTM um objeto compartilhado pode estar em quatro diferentes estados:

- Compartilhado: Um objeto compartilhado representa um objeto que ainda não foi acessado por nenhuma transação.
- Somente leitura: Representa um objeto que será utilizado somente para leitura em uma transação.
- Passível de escrita: Um objeto que pode ter seus atributos modificados pela transação corrente. Nenhuma mudança será visível a outras *threads* até que a transação seja finalizada com sucesso.
- Privatizado: É um objeto que será acessado (escrita ou leitura) por somente uma *thread* por vez.

Para representar estes estados a RSTM utiliza as respectivas classes: `stm::sh_ptr`, `stm::rd_ptr`, `stm::wr_ptr` e `stm::un_ptr`.

Quando uma transação tenta adquirir um objeto que está sendo manipulado por outra transação, um gerenciador de contenção é invocado para resolução deste conflito. (seção 2.3.6). A RSTM admite atualmente dezesseis algoritmos de gerenciamento de contenção, a escolha de qual gerenciador será usado é feita em tempo de compilação.

O escopo de uma transação é demarcado pelas as macros `BEGIN_TRANSACTION` e `END_TRANSACTION`. Caso seja necessário executar códigos de limpeza antes de uma transação abortada ser reexecutada, a RSTM provê uma macro chamada `ON_RETRY` (listagem 3.2) que deve ser usada dentro de uma transação. Esta macro não provê a mesma funcionalidade do comando `retry` visto na seção 2.2.2, esta macro apenas prove um escopo para que códigos relacionados ao tratamento de uma transação abortada sejam executados, podendo ser considerado análogo a captura e tratamento de exceções.

Listagem 3.2: Macros da biblioteca RSTM

```
1 BEGIN_TRANSACTION;
2 // codigo transacional
3 ON_RETRY {
4 // se reexecutar...
5 }
6 END_TRANSACTION;
```

3.2 Modo usuário e modo kernel

Arquiteturas de processadores modernos normalmente permitem que a CPU opere em pelo menos dois modos diferentes: *modo usuário* e *modo kernel* (nomeado também como *modo supervisor* e *modo privilegiado* em algumas literaturas). Instruções de hardware permitem a troca de um modo para o outro. Correspondentemente, existem áreas de memória virtual marcadas como sendo parte do *espaço de usuário* e outras como *espaço de kernel*. Quando está operando em modo usuário, a CPU pode somente obter acesso a regiões de memória que estão marcadas como sendo parte do espaço de usuário; neste caso uma tentativa de acesso à regiões de memória marcadas como sendo parte do espaço de *kernel* resultaria em uma exceção de hardware. Quando está operando em modo *kernel*, a CPU pode acessar os dois espaços, de usuário e de *kernel*.

Algumas operações são protegia das por hardware e podem ser executadas somente enquanto a CPU está operando em modo *kernel*, por exemplo: instruções para desligar a máquina, instruções para acessar a *Memory Management Unit* (MMU) ou dispositivos de I/O. Para aproveitar esta proteção por hardware, o sistema operacional é executado em modo *kernel*, garantindo assim que processos que estão sendo executados em modo usuário não interfiram nas regiões de memória do sistema operacional, ou que sejam capazes de executar operações sem o conhecimento do sistema operacional. Isto é feito para que o *kernel* possa manter o sistema seguro e livre de programas maliciosos que estão sendo executados em espaço de usuário.

3.3 Chamadas de sistema

3.3.1 Introdução

Uma chamada de sistema é um mecanismo controlado para comunicação entre espaço de usuário e espaço de *kernel*, permitindo que um processo requisite ao *kernel* operações de seu interesse. O *kernel* disponibiliza uma série de serviços aos processos através de chamadas de sistema. Estes serviços incluem a criação de um novo processo, leitura e escrita de arquivos, requisitar mais memória, criar novas *threads*, etc.

Uma chamada de sistema altera o modo de execução da CPU, do modo usuário para o modo *kernel*, assim a CPU pode obter acesso a regiões protegidas de memória. O sistema operacional define um número fixo de chamadas de sistema. Cada uma possui um identificador único e um conjunto de argumentos que definem as informações que serão transferidas do espaço de usuário para o espaço de *kernel*.

Existe uma interrupção de hardware (`INT 0x80` na arquitetura Intel®-32 bit) capaz de fazer a troca de modo de execução. Esta interrupção é executada quando um processo em modo usuário requisitou a execução de uma chamada de sistema. Em conjunto a esta interrupção, um registrador (EAX na arquitetura Intel®-32 bit) recebe um valor inteiro que identifica a chamada de sistema. Caso este identificador esteja definido como um identificador válido no sistema operacional, a chamada de sistema poderá ser executada. Arquiteturas Intel® e AMD® mais recentes admitem as instruções *sysenter* e *syscall* respectivamente, que provêem um método mais rápido para troca do modo usuário para o modo *kernel*; mais detalhes sobre estas instruções estão fora do escopo desta pesquisa.

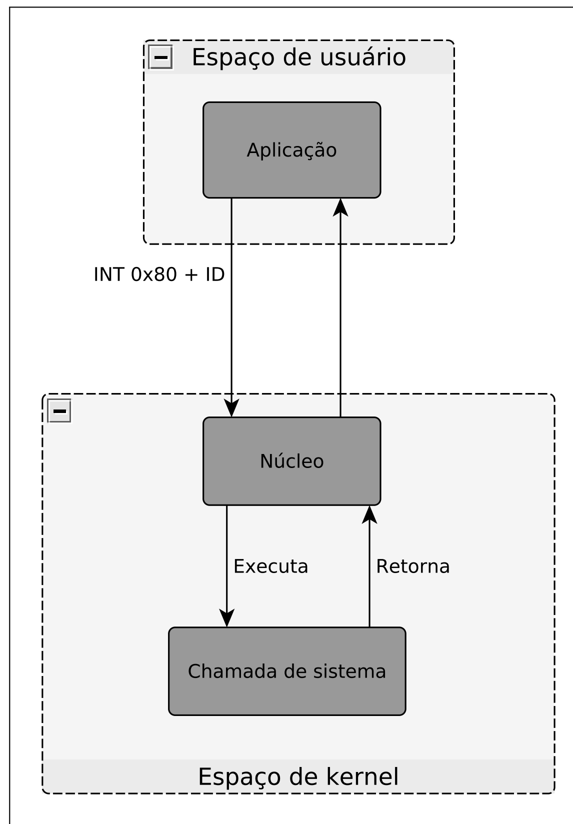


Figura 3.1: Representação de uma chamada de sistema.

Após a interrupção de hardware o *kernel* do sistema operacional Linux executa a função `system_call` (localizada no arquivo `arch/x86/kernel/entry_32.S`) para manipular a execução da chamada de sistema. Para que seja mantido um padrão entre todas as chamadas de sistema, o *kernel* espera que todos os argumentos das chamadas de sistema sejam passados em registradores específicos, e uma das responsabilidades da função `system_call` é copiar os argumentos que estão nos registradores para a pilha. Logo em seguida é verificado se o identificador da chamada de sistema é admitido pelo *kernel*, em caso positivo a função apropriada é executada. O *kernel* do sistema operacional Linux define uma tabela de chamadas de sistema admitidas e seus respectivos identificadores na variável `sys_call_table`.

Ao final da execução, a chamada de sistema retorna o seu resultado para a função `system_call`, os valores relacionados a esta execução de chamada de sistema que estão na pilha do *kernel* são copiados para os registradores e a CPU volta para o modo usuário.

3.3.2 Criando uma chamada de sistema

Nesta seção é demonstrado como uma nova chamada de sistema pode ser adicionada ao sistema operacional Linux. O exemplo a seguir foi baseado na versão 2.6.36. Os nomes dos arquivos e diretórios citados a seguir podem variar de acordo com a versão do *kernel*, mas os conceitos permanecem.

O processo de adição de uma nova chamada de sistema consiste em três passos: Criar uma nova função, atualizar alguns cabeçalhos do *kernel* e adicionar uma nova entrada na tabela que define todas as chamadas de sistema admitidas pelo sistema operacional.

A criação da função que será a nova chamada de sistema é similar a criação de uma função comum. A diferença é o uso do modificador `asmlinkage` (Listagem 3.3), que diz ao compilador que, quando essa função é invocada todos seus argumentos devem ser armazenados na pilha e não em registradores. Normalmente o compilador, por questões de otimização, tenta fazer com que os argumentos passados para uma função sejam armazenados em registradores, mas para garantir que a chamada de sistema recupere os argumentos passados para a função em espaço de usuário corretamente, e a garantia de que os argumentos serão passados pela pilha é o uso do modificador `asmlinkage`.

Listagem 3.3: Exemplo de uma chamada de sistema

```
1 asmlinkage long sys_novachamada(int x) {
2   return x * 2;
3 }
```

É necessário também atualizar alguns cabeçalhos do *kernel*. Um dos cabeçalhos que deve ser atualizado é o `arch/x86/include/asm/unistd_32.h`. Neste arquivo são definidos os identificadores de todas as chamadas de sistema, bem como a quantidade total de chamadas de sistema admitidas. Para adicionar uma nova chamada de sistema é deve-se incrementar a quantidade total de chamadas de sistema e incluir uma entrada para a nova chamada de sistema e seu identificador, conforme listagem 3.4.

Listagem 3.4: Definição dos identificadores das chamadas de sistema

```
1 /* identificador para a nova chamada de sistema */
2 #define __NR_novachamada   338
3 /* total de chamadas de sistema admitidas pelo kernel */
4 #define NR_syscalls   339
```

O outro cabeçalho que deve ser alterado é o `include/linux/syscalls.h`. Este arquivo define a assinatura de todas as chamadas de sistema. Um exemplo de uma entrada de chamada de sistema neste arquivo é exibida na listagem 3.5.

Listagem 3.5: Exemplo de uma chamada de sistema

```
1 asmlinkage long sys_novachamada(int x);
```

O sistema operacional Linux possui uma tabela que define as chamadas de sistema admitidas. Esta tabela é definida no arquivo `arch/x86/kernel/syscall_table_32.S`. É necessário adicionar uma nova entrada nesta tabela para que o *kernel* saiba que a nova chamada de sistema é uma chamada válida.

Listagem 3.6: Registro na tabela que define chamadas de sistema

```
1   .long sys_novachamada
```

Por fim, deve-se incluir no arquivo `Makefile` o diretório onde está a implementação da nova chamada de sistema para que seja compilada e seja admitida pela nova compilação do *kernel*.

Listagem 3.7: Nova entrada no Makefile

```
1   core-y += kernel/ mm/ fs/ ... novachamada/
```

3.4 Procfs

3.4.1 Introdução

No sistema operacional Linux existem mecanismos para que o *kernel* e seus módulos enviem informações para os processos. Um destes mecanismo é um sistema de arquivos virtual conhecido como `procfs`. O `procfs` foi originalmente projetado para permitir acesso fácil a informações sobre processos, e atualmente é utilizado pelo sistema operacional para reportar qualquer informação interessante sobre o estado dos processos e do próprio sistema operacional para o espaço de usuário.

Apesar de ser um sistema de arquivos virtual e não ser associado a nenhum dispositivo de bloco — residindo somente em memória — o `procfs` tem a vantagem de ser estruturado como um sistema de arquivos comum, contendo diretórios e arquivos.

O sistema de arquivos `procfs` atua como interface entre as estruturas internas do *kernel* e o espaço de usuário. Ele pode ser usado tanto para obter informações sobre o sistema operacional como para alterar alguns parâmetros do *kernel* em tempo de execução.

O `procfs` contém, entre outras coisas, um subdiretório para cada processo em execução. Estes subdiretórios disponibilizam diversas informações sobre o estado do processo e os recursos utilizados por ele: como consumo de CPU e memória, descritores de arquivos abertos, argumentos recebidos ao ser executado, etc. Por estarem disponibilizadas em formato texto, estas informações podem ser lidas facilmente por um usuário humano ou por ferramentas de análise e monitoração como `top, ps, netstat, pstree`, que formatam estas informações para uma melhor visualização.

Similarmente as entradas no `procfs` relacionadas aos processos em execução, dados sobre o sistema operacional também ficam disponíveis no `procfs`. Informações sobre recursos de hardware disponíveis, como CPU e memória, carga do sistema, mensagens enviadas pelo *kernel*, módulos do *kernel* que estão carregados, *drivers* que estão sendo utilizados, etc.

Também é possível enviar informações para o *kernel* por meio do `procfs`. Certos aspectos relacionados ao comportamento do sistema operacional podem ser modificados em tempo de execução, sem a necessidade de recompilar o *kernel* ou mesmo reiniciar o sistema. Alguns arquivos abaixo do diretório `/proc/sys/` podem não somente ser lidos, mas também modificados, alterando algumas configurações do sistema operacional.

3.4.2 Escrita no procfs

Esta pesquisa não tem como objetivo nem a necessidade de permitir o envio de dados do espaço de usuário para o sistema operacional por meio do `procfs`. Portanto nesta seção será abordado apenas o caminho inverso – como o *kernel* pode enviar informações ao espaço de usuário utilizando o sistema de arquivos `procfs`.

Por ser um sistema de arquivos virtual e possuir particularidades, a escrita no sistema de aquivos `procfs` não é feita por meio do processo tradicional de escrita em arquivos. A leitura e escrita no `procfs` deve ser feita utilizando funções específicas para esta finalidade. A criação de uma nova entrada se dá por meio da função `create_proc_entry` (listada em 3.8). Esta função

recebe como argumentos o nome da nova entrada (nome do arquivo), modo de leitura e escrita (permissão) e um ponteiro para sua entrada raiz, se existir. Se bem sucedida, uma estrutura do tipo `proc_dir_entry` é retornada. Para esta pesquisa, um campo desta estrutura é mais relevante: o `read_proc`. Neste campo é passada um função de *callback* que será executada toda vez que esta nova entrada no `procfs` for lida.

Existem outras duas maneiras de manipular o sistema de arquivos `procfs`: como um sistema de arquivos comum e arquivos sequenciais (`seq_file`). No primeiro caso, a vantagem é poder utilizar funções mais avançadas como as que tratam permissões de arquivos. Já o uso de arquivos sequenciais é vantajoso caso o conjunto de dados disponibilizados seja muito grande. A implementação de `seq_file` permite que os objetos que representam a informação que será enviada ao `procfs` sejam iterados, e também possuí funções para formatação destas informações.

Listagem 3.8: Função create_proc_entry

```
1 struct proc_dir_entry *create_proc_entry(const char *name,
2                               mode_t mode,
3                               struct proc_dir_entry *parent);
```

3.4.3 Manipulando os arquivos no procfs

Arquivos abaixo do diretório `/proc` podem ser manipulados por *scripts* ou por programas que utilizam instruções comuns de leitura de arquivos. Apenas deve-se respeitar algumas restrições:

- Alguns arquivos são somente para leitura e não podem ser modificados. Isso se aplica a maioria dos arquivos abaixo do subdiretório `/proc/PID`
- Alguns arquivos podem ser lidos somente por um usuário ou processo que possuí crendenciais para isso. Por exemplo os arquivos abaixo de `/proc/PID` podem ser lidos pelo usuário que iniciou tal processo.
- Outros arquivos do `/proc` podem ser manipulados ou modificados somente pelo superusuário.

4 SISTEMA DE MONITORAÇÃO TMEM

O TMEM é um sistema de monitoração de eventos relacionados à memória transacional que permite que o sistema operacional admita novas informações sobre processos que estão em execução e que utilizam STM. O TMEM é projetado para monitorar, inicialmente, a biblioteca RSTM (MARATHE et al., 2006) que, como visto na seção 3.1, possui treze implementações diferentes de memória transacional.

Neste capítulo são apresentados os detalhes da implementação do TMEM para o sistema operacional Linux versão 2.6.39 e como ele pode auxiliar o desenvolvedor a entender melhor o comportamento de seu código em execução.

4.1 Introdução

Conforme visto no capítulo 2, o desenvolvimento de aplicações capazes de serem processadas em paralelo é uma tarefa mais complexa do que o desenvolvimento de aplicações sequenciais. Apesar de facilitar o desenvolvimento de programas desta natureza, a memória transacional não elimina a possibilidade de programas incorretos serem escritos. A codificação de programas paralelos (incluindo os que utilizam memória transacional) é mais difícil devido a fatores como coordenação e rastreabilidade de múltiplos fluxos concorrentes. Testar estes programas é uma tarefa ainda mais difícil, a execução não determinística dos fluxos de execução pode resultar em comportamentos inadequados. E como visto na seção 2.5, a depuração de programas deste tipo pode ser ineficiente quando se deseja rastrear problemas relacionados a concorrência, pois inevitavelmente atrasos na execução são introduzidos, alterando o comportamento da aplicação. O propósito principal de um sistema de monitoração como o TMEM é ajudar o desenvolvedor a entender o comportamento do seu código em execução podendo assim melhorar o desempenho do seu programa.

O TMEM auxilia o desenvolvedor a entender o comportamento de seu código paralelo que utiliza STM coletando e disponibilizando informações relacionadas a aplicação que está execução. Em um código *multithreaded* isto deve ser feito de maneira com que não sejam inseridas sincronizações desnecessárias e deve-se permitir que as transações sejam executadas normalmente, com mínima intrusão por parte da monitoração.

Durante a transição de programas *multithreaded* que usam mecanismos clássicos de controle de concorrência, como travas de exclusão mútua, para memória transacional um sistema de monitoração pode ser mais útil que um depurador. Assumindo-se que os programas que uti-

lizam travas estão corretos, não há necessidade do uso de um depurador. Por outro lado, como as regiões críticas usam *locks* como mecanismo de exclusão mútua e não foram projetadas para serem executadas concorrentemente, elas podem compartilhar dados desnecessariamente o que pode comprometer a eficácia ou escalabilidade do programa quando passar a utilizar memória transacional.

4.2 Características

Esta pesquisa não tem como objetivo fazer com que o sistema operacional seja capaz de manipular programas que utilizam memória transacional, como em (RAMADAN et al., 2007) onde foi proposta uma modificação para o *kernel* capaz de decidir entre transações e *locks*. No TMEM o sistema operacional Linux foi modificado para poder admitir novas informações sobre os eventos relacionados à TM e que ocorrem nos processos que estão em execução. Estas informações serão enviadas do espaço de usuário para o espaço de *kernel* por meio de uma chamada de sistema e serão disponibilizadas no sistema de arquivos `procfs`.

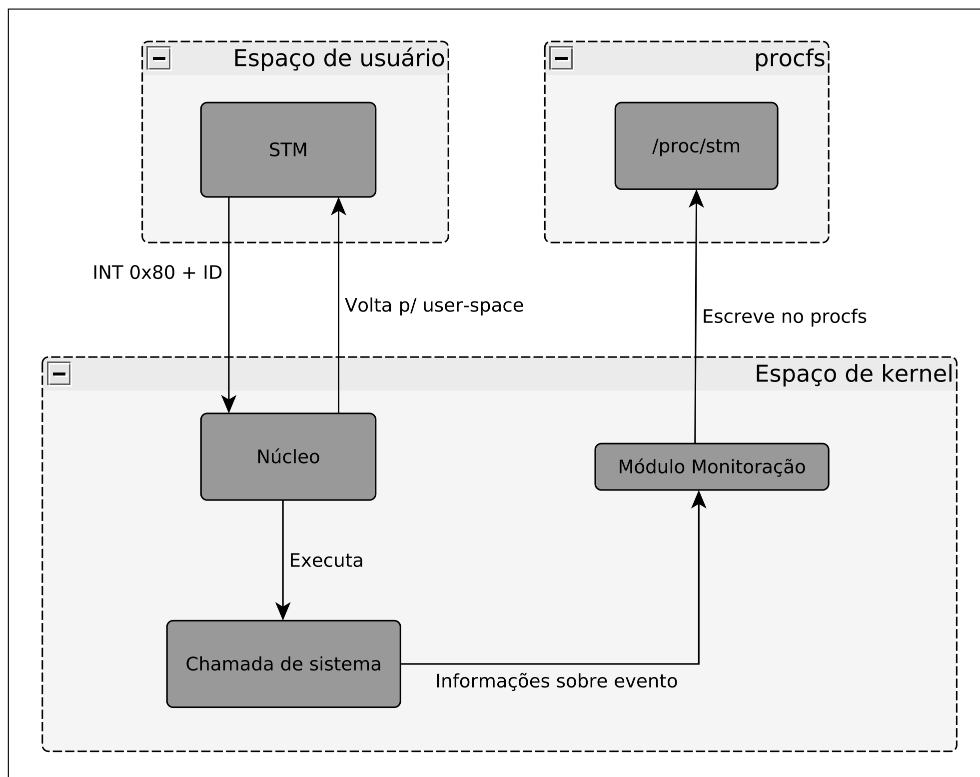


Figura 4.1: Arquitetura do sistema de monitoração TMEM.

A disponibilização das informações no sistema de aquivos `procfs` torna mais fácil a construção de ferramentas de análise e monitoração por parte do usuário final, pois é uma interface padronizada de acesso ao *kernel*, além de possuir a estrutura de um sistema de arquivos, facilitando a manipulação dos dados, como visto na seção 3.4.

A definição de um padrão para uma chamada de sistema capaz de disponibilizar informações sobre eventos relacionados à memória transacional permite que qualquer implementação de memória transacional seja adaptada com facilidade. Outra vantagem do TMEM é a sobre-

carga no tempo de usuário utilizado pela aplicação, já que a maior parte do tempo de tratamento dos eventos será computado como tempo de sistema.

A definição deste arcabouço permite a adaptação fácil para outros eventos relacionados à memória transacional e para qualquer outra implementação de memória transacional.

4.2.1 Eventos monitorados

Como memória transacional ainda é o foco de diversas pesquisas e novas implementações ainda surgem, existem diversos eventos entre as várias implementações de memória transacional. Alguns destes eventos são restritos a algumas implementações, outros eventos fazem parte da base de sistemas que utilizam transações e existem em todas as implementações de memória transacional. São eles: inicio de uma transação, transação finalizada com sucesso (*commit*) e transação abortada. Este último evento pode ser dividido em duas categorias: transações abortadas devido a um conflito ou por uma chamada explícita por parte do usuário. Por ser necessário instrumentar todas as treze implementações da biblioteca RSTM, o TMEM é capaz de monitorar os seguintes eventos:

- Transações iniciadas
- Transações finalizadas
- Transações abortadas

4.3 Alterações no sistema operacional

Para admitir novas informações sobre memória transacional, o *kernel* do sistema operacional Linux foi alterado. A estrutura de dados que representa um processo ou uma *thread* em execução, a (`task_struct`), passou a receber um vetor de uma nova estrutura capaz de representar eventos relacionados à memória transacional, assim cada novo processo ou nova *thread* manipulados pelo sistema operacional são capazes de armazenar informações sobre estes eventos.

Listagem 4.1: Alteração da task_struct

```
1 struct task_struct {
2         struct stm_struct stm_events[STM_BLOCKS_MAX];
3   ...
4 }
```

Esta nova estrutura é chamada de `stm_struct` (Listagem 4.2). Ela representa os eventos de apenas um bloco atômico e possui um contador para cada evento, contadores estes responsáveis em acumular a quantidade de ocorrências de cada evento. As razões para estes contadores serem do tipo `atomic_t` serão explicadas mais adiante na seção 4.3.1.

Listagem 4.2: Representação dos eventos

```
1 struct stm_struct {
2         int tx_id;
3         atomic_t commited;
4         atomic_t aborted;
5         atomic_t started;
6 }
```

4.3.1 Concorrência em espaço de kernel

Em um sistema que implementa memória transacional várias *threads* podem ser executadas em paralelo. Com o TMEM, assim que um evento relacionado à memória transacional acontece, a nova chamada de sistema é invocada e o processamento sai do espaço de usuário e passa ao espaço de *kernel*. Em um código *multithreaded* a execução da chamada de sistema também pode ser executada por diversas *threads* simultaneamente, fazendo com que os problemas de concorrência vistos no capítulo 1 existam também em espaço de *kernel*. Para evitá-los o sistema operacional Linux provê alguns mecanismos como semáforos, travas de exclusão mútua e *spinlocks* que podem ser usados em códigos internos ao *kernel*.

Como o TMEM tem como premissa não inserir sincronizações desnecessárias e deve permitir que as transações sejam executadas normalmente, com mínima intrusão por parte da monitoração, foi utilizada uma alternativa livre de travas: o tipo `atomic_t` em conjunto com as funções definidas em `asm/atomic.h`. Alguns processadores são capazes de realizar incrementos de maneira atômica e estas funcionalidades estão disponíveis neste arquivo. Isto é útil em casos simples, onde deseja-se incrementar ou decrementar contadores, como é o caso do TMEM. Estas operações de incremento e decremento são rápidas e em algumas arquiteturas podem ser traduzidas para apenas uma única instrução de máquina.

4.3.2 Chamada de sistema

Uma nova chamada de sistema foi incluída no sistema operacional, a `register_stm_event`. Ela tem a responsabilidade de recuperar a referência ao processo que a invocou em espaço de usuário, verificar se o processo ainda existe e se o identificador de eventos é válido. Se todos os argumentos forem válidos deve-se finalmente incrementar o contador correspondente ao evento ocorrido.

Esta nova chamada de sistema recebe como argumento os identificadores da transação e do evento ocorrido.

Listagem 4.3: Chamada de sistema do TMEM

```
1   int stm_event(int atomic_block_id, uint8_t stm_event_type);
```

Alguns detalhes de implementação da chamada de sistema também devem ser levados em consideração. O primeiro deles é que códigos em espaço de `kernel` referem-se ao processo em execução manipulando diretamente a variável `current` definida em `include/asm/current.h`,

que provê um ponteiro para a `task_struct` (*thread* ou processo) que está em execução, possibilitando assim que a chamada de sistemas `register_stm_event` manipule a estrutura que representa os eventos relacionados à TM, `stm_struct`.

Outro detalhe de implementação importante é que como viu-se na seção 4.3 a `task_struct` também representa as *threads* que estão em execução e o acesso à `trask_struct` por meio da variável `current` resultaria na criação de novos acumuladores. Para solucionar este problema foi necessário acessar o líder do grupo de *threads* (listagem 4.4), que é um ponteiro para a `task_struct` que representa o processo que iniciou as *threads* que estão em execução. Isto permite que a estrutura de dados manipulada pela chamada de sistema `register_stm_events` seja sempre a mesma para cada processo, mesmo quando uma invocação à chamada de sistema tenha sido feita por uma *thread*.

Listagem 4.4: TMEM acessando informações sobre o processo em execução

```
1        struct task_struct *task;
2        task = current->group_leader;
```

4.3.3 Informações de saída

Para cada processo em execução existe um diretório correspondente chamado de `/proc/PID`, onde *PID* é um número inteiro que identifica o processo. Dentro deste diretório existem diversos arquivos e informações sobre o processo. A tabela 4.1 lista alguns deles:

Tabela 4.1: Arquivos em cada diretório /proc/PID

Arquivo	Descrição
cmdline	Argumentos passados pela linha de comando
cwd	Link simbólico para o diretório corrente
environ	Varáveis de ambiente
exe	Link simbólico para o arquivo que gerou este processo
fd	Diretório contendo os links simbólicos para os descritores abertos por este processo
maps	Mapeamentos de memória
status	Informações como ID do processo, credenciais, uso de memória, sinais
task	Contém um subdiretório para cada thread em execução

O sistema de monitoração TMEM adiciona mais um elemento dentro deste diretório, o arquivo `stm_events`. As informações disponibilizadas neste arquivo estão em formato texto, o que facilita a leitura por humanos e por ferramentas de análise. Cada linha deste arquivo possui as seguintes informações: ID do bloco atômico, quantidade de transações iniciadas, finalizadas e abortadas.

A tabela 4.5 representa um possível conteúdo do aquivo `stm_events`:

Listagem 4.5: Conteúdo do arquivo stm_events

```
1  [AtomicBlock]   [Started]  [Finished]  [Aborted]
2  1                 12          10           2
3  2                 17          15           2
4  3                 100         80          20
```

Apesar de estar localizado em um sistema de arquivos virtual, a leitura do arquivo `stm_events` é feita normalmente. Um simples *script* pode auxiliar na leitura destas informações.

A escrita no procfs feita pelo TMEM não se dá do modo convencional, como descrito na seção 3.4.2. A escrita no subdiretório `/proc/PID` deve ser feita de maneira diferente. Isto se deve ao fato de que a criação dos arquivos contidos neste diretório acontece quando o sistema operacional cria um novo processo. Os arquivos e diretórios que serão criados para cada novo processo estão definidos no vetor `tgid_base_stuff`, que recebe uma lista contendo o nome do item, o seu tipo (arquivo, diretório), e uma função que será chamada quando o arquivo for lido em espaço de usuário. Para manter este modelo adotado pelo *kernel*, o TMEM adiciona um novo elemento ao vetor `tgid_base_stuff`, como pode ser visto na listagem 4.6.

O TMEM não tem a responsabilidade de remover o arquivo abaixo do diretório `/proc/PID` quando o processo for finalizado. A remoção do subdiretório `/proc/PID` e todo seu conteúdo já é feita automaticamente pelo *kernel* quando o processo é finalizado.

Listagem 4.6: Nova entrada no vetor tgid_base_stuff

```
1 static const struct pid_entry tgid_base_stuff[] = {
2   INF("stm_events", S_IRUGO, proc_pid_stm_events),
3   ...
4 }
```

4.3.4 TMEM como módulo do kernel

Um módulo do *kernel* tem a vantagem de ser carregado e removido conforme a necessidade, ampliando as funcionalidade do *kernel* sem a necessidade de reiniciar o sistema operacional. Este modelo seria vantajoso para o sistema de monitoração, mas as alterações listadas a seguir fazem com que não seja possível que o TMEM seja implementado como um módulo para o *kernel*.

- Criação de uma nova chamada de sistema (seção 4.3.2): Todas as chamadas de sistema admitidas pelo sistema operacional são definidas assim que ele é iniciado. O sistema operacional Linux não admite que novas chamadas de sistema sejam adicionadas e removidas enquanto o sistema está em funcionamento.
- Alteração da estrutura de dados `task_struct`, seção 4.3, listagem 4.1: Esta estrutura é utilizada em diversos trechos do *kernel* e principalmente no escalonador de processos e sua alteração faz com que a recompilação seja obrigatória.
- Alteração do vetor que define os arquivos e diretórios abaixo de `/proc/PID` (Listagem 4.2): A alteração desta estrutura também implica em uma recompilação de todo o *kernel*.

4.4 Benchmark suites e microbenchmarks

Atualmente os protótipos de memória transacional tem sido avaliados por meio de ferramentas de *microbenchmarks* ou aplicações paralelas contidas em *benchmark suites*. Nos *microbenchmarks* cada *thread* tenta executar, o mais rápido possível, o mesmo conjunto de operações

repetidamente em uma estrutura de dados compartilhada. Isto faz com que os *microbenchmarks* sejam úteis em situações onde deseja-se comparar o comportamento e o desempenho de uma implementação de memória transacional em diversas condições, como por exemplo a comparação entre *locks* e STM, ou quando deseja-se medir a escalabilidade de algoritmos diferentes de gerenciamento de contenção, ou ainda quando um novo comportamento é adicionado à implementação de memória transacional. Enquanto os *microbenchmarks* permitem avaliar características específicas de um sistema de memória transacional, eles não representam como aplicações reais se comportam ao utilizar este sistema. Já as *benchmark suites* são aplicações completas que representam sistemas mais realistas como os que são ou serão desenvolvidos e utilizados no dia-a-dia.

Como esta pesquisa tem como objetivo comparar o desempenho da biblioteca RSTM antes e depois da implementação do sistema de monitoração TMEM, é necessário apenas o uso de um *microbenchmark*. A implementação escolhida foi a implementação que acompanha a própria biblioteca RSTM, por admitir as treze implementações de memória transacional que fazem parte desta biblioteca.

4.5 Alterações na biblioteca RSTM

Como visto na seção 2.3.5 alguns sistemas de memória transacional diferem quanto a granularidade de detecção de conflitos. Algumas implementações utilizam um tamanho fixo, geralmente o tamanho da palavra da arquitetura, para garantir a consistência e detectar conflitos em regiões de memória. Estas implementações são conhecidas como *word-based* ou *block-based*. Existem também implementações de memória transacional que utilizam granularidade de detecção de conflitos do tamanho de um objeto, conhecidas como *object-based*.

Duas, das treze implementações de memória transacional que são distribuídas com a biblioteca RSTM são *object-based*, são elas: *RSTM* e *redo-lock*. A RSTM é a implementação não-bloqueante original descrita em (MARATHE et al., 2006). A implementação *redo-lock* é uma modificação do algoritmo RSTM que elimina um nível de indireção. Isto é feito usando um recurso de hardware conhecido como *alert-on-update*, que permite que uma *thread* seja notificada quando um dado específico é substituído ou invalidado no *cache*. Mais detalhes sobre esta implementação podem ser encontrados em (SPEAR et al., 2007).

Todas as treze implementações de memória transacional que fazem parte da RSTM foram alteradas para admitir o uso do sistema de monitoração de eventos TMEM. Estas alterações foram feitas de forma com que o seu uso seja transparente para o usuário, quem usa a RSTM não precisa saber como notificar o *kernel* a cada novo evento ocorrido.

Neste trabalho, o TMEM possui uma API simples que recebe o identificador do bloco atômico e o identificador do evento ocorrido. Cabe a implementação de memória transacional fazer a chamada à função com os argumentos corretos. Para facilitar o uso da nova chamada de sistema foram criadas funções, em espaço de usuário, que encapsulam parte desta logica e recebem apenas o identificador da transação. São elas: `register_transaction_start()`, `register_transaction_commit()`, `register_transaction_abort()`.

Foi realizada uma alteração na biblioteca RSTM para permitir que todos os blocos atômicos pudessem receber um número inteiro como identificador para que sejam identificados pelo sistema de monitoração. Para isso foi necessário alterar a implementação da macro que define

o início de um bloco atômico fazendo com que ela passasse a receber como argumento um número inteiro (listagem 4.7). Esta é a única alteração da biblioteca RSTM visível ao usuário.

É importante notar que a implementação do TMEM proposta nesta pesquisa não atribui um identificador automaticamente para cada nova transação, esta é uma responsabilidade do usuário. Esta implementação também não verifica se um identificar já está em uso por outro bloco atômico.

Listagem 4.7: Alteração da macro da biblioteca RSTM

```
1 // criacao de um bloco atomico com ID=10
2 BEGIN_TRANSACTION(10);
3 // codigo transacional
4 END_TRANSACTION;
```

Internamente, a biblioteca RSTM possui uma estrutura da dados que define metadados de cada transação 3.1. Foi adicionada à esta estrutura de dados um número inteiro que representa o identificador de cada bloco atômico, bem como seus respectivos métodos de acesso `get` e `set`. Assim que um novo bloco atômico é criado usando-se a macro `BEGIN_TRANSACTION(tx_id)`, esta estrutura de dados é acessada e o valor passado em `tx_id` é atribuído, por meio do método `setTxId()`, ao novo atributo.

A cada novo evento de transação iniciada, abortada ou finalizada, as funções que encapsulam a nova chamada de sistema são invocadas com os devidos argumentos, desta forma o uso do TMEM é transparente para os usuários da biblioteca RSTM.

4.6 Extensibilidade

O TMEM possui uma API simples e pode ser adaptado facilmente para admitir novos eventos ou monitorar outra implementação de memória transacional. Para esta última adaptação, basta que seja invocada a nova chamada de sistema no momento em que um evento acontecer, passando-se os identificadores de cada evento ocorrido e transação.

Já a adaptação que permite que o sistema de monitoração admita novos eventos exige que o código em espaço de *kernel* seja alterado. É necessário alterar a implementação da chamada de sistema, onde está concentrada a lógica responsável em incrementar os contadores de eventos para cada processo. Além de alterar a estrutura de dados `stm_struct`, que representa os eventos de cada transação executada (listagem 4.2).

4.7 Limites

Na implementação do sistema operacional Linux existem alguns limites relacionados a funcionalidades e manipulação de recursos, como por exemplo a quantidade de descritores de arquivos abertos, qual é o tamanho máximo de uma variável do tipo `int`, o quão grande pode ser a lista de argumentos de um programa, número máximo de *threads* que um processo pode executar, etc. O sistema de monitoração TMEM também possui alguns limites. O vetor que

representa os eventos ocorridos dentro de cada bloco atômico tem o seu tamanho definido pela variável `STM_BLOCKS_MAX`, que possuí um valor de 9.999 blocos atômicos por processo.

Outra limitação da implementação apresentada nesta pesquisa e que deve ser observada é o tipo `atomic_t`, usado para garantir que a chamada de sistema seja *thread-safe* e não-bloqueante. Este tipo é um inteiro de 24-bit, o que limita o TMEM a uma quantidade de $2^{24} - 1$ para cada tipo de evento.

4.8 Possíveis usos

Como viu-se no capítulo 2, um programador que usa memória transacional ainda está sujeito a cometer erros que comprometam a eficácia ou o desempenho de seu programa. Com memória transacional a responsabilidade de decidir quais trechos de seu código serão executados paralelamente e de forma atômica ainda é do programador e por este motivo ainda é possível escrever programas concorrentes incorretos. É possível, por exemplo, escrever transações com uma granularidade muito grossa, onde um volume de dados manipulado pela transação é maior que o necessário, ou ainda criar uma transação onde são efetuadas mais operações que o necessário. Nestes casos o TMEM pode auxiliar o programador durante o desenvolvimento evidenciando a quantidade de transações finalizadas com sucesso para cada teste executado.

Outra maneira de escrever transações incorretamente é usando-se a API do sistema de memória transacional de maneira equivocada. Pode-se, por exemplo iniciar uma transação mas esquecer de finalizá-la. Pode-se, como na listagem 4.8, invocar o método `retry()` explicitamente sem necessidade, fazendo que a transação sempre seja abortada e nunca finalize com sucesso. Os dados fornecidos pelo TMEM evidenciam problemas deste tipo, como pode ser visto na tabela 4.2, onde a transação identificada pelo número 29 não foi finalizada com sucesso (*commit*) nenhuma vez.

Listagem 4.8: Uso incorreto da API

```
1  BEGIN_TRANSACTION(29);
2  stm::wr_ptr<Counter> wr(m_counter);
3  stm::retry(); // Uso errado da API
4  wr->set_value(wr->get_value(wr) + 1, wr);
5  END_TRANSACTION;
```

Tabela 4.2: Saída do TMEM para o mau uso da API

AtomicBlockID	Starts	Commits	Retries
29	248.842	0	497.679

Como visto na seção 2.3, existem diversas abordagens na construção de um sistema de memória transacional. Um sistema de monitoração pode auxiliar o desenvolvedor a definir qual a melhor configuração para seu sistema. Ainda durante o desenvolvimento o programador pode verificar, por exemplo, qual gerenciador de contenção é o mais adequado para o seu problema, neste sentido o TMEM pode auxiliar evidenciando a quantidade de transações abortadas e finalizadas com sucesso para cada execução.

5 ANÁLISE DOS RESULTADOS

Neste capítulo são apresentados os resultados dos experimentos realizados com o sistema de monitoração TMEM e as dificuldades encontradas durante sua implementação.

5.1 Introdução

Foram realizados testes funcionais e de desempenho durante e depois do desenvolvimento do TMEM.Para realizar os testes de desempenho, foram conduzidos testes isolados para chamada de sistema e testes do TMEM integrado à biblioteca RSTM. Detalhes de como estes testes foram conduzidos serão descritos nas seções 5.2 e 5.3, respectivamente.

Os testes foram feitos em um Intel Core 2 Duo de 2.13GHz executando o sistema operacional Linux versão 2.6.39 de 32-bit e compilador GCC versão 4.4.5. Os testes de desempenho conduzidos nesta pesquisa foram baseados no *microbenchmark* que acompanha a biblioteca RSTM versão 5.

5.2 Avaliação da chamada de sistema

Nesta pesquisa foram realizados dois tipos de testes para medir o desempenho da nova chamada de sistema criada pelo TMEM. O primeiro teste tem como objetivo medir a quantidade de ciclos de *clock* de CPU necessários para execução da chamada de sistema, podendo-se assim conhecer parte da sobrecarga imposta pelo sistema de monitoração. No segundo teste foi usado o tempo como métrica, onde foi medido o tempo em espaço de *kernel* necessário para execução da nova chamada de sistema.

5.2.1 Ciclos de clock

Para medir da quantidade de ciclos de *clock* necessários para execução da nova chamada de sistema foi utilizado registrador que armazena a quantidade de ciclos executados pela CPU desde a sua última inicialização, o `RDTSC`. Durante a execução deste teste podemos perceber que alguns eventos e comportamentos do hardware e do sistema operacional podem afetar o `RDTSC` e devem ser considerados.

O primeiro é o fato de que existe um contador para cada núcleo da CPU e não há garantias de que eles estão sincronizados e possuem o mesmo valor. Isto acontece porque em sistemas

operacionais modernos, para manter um limite térmico, podem ligar ou desligar núcleos da CPU individualmente ou, em caso de necessidade de mais desempenho podem até mesmo aumentar a frequência dos núcleos disponíveis. Para solucionar este problema foi necessário impedir que o escalonador de processos migre o processo para outro núcleo ou CPU.

Outro fator que deve ser levado em consideração ao utilizar-se o contador de ciclos de *clock* é o suporte a execução fora de ordem, onde processadores modernos são capazes de executar instruções fora da ordem em que aparecem no código fonte. Isto faz com que os ciclos de clock computados podem não ter sido gerados pelo código que deseja-se observar.

A troca de contexto também afeta diretamente testes que utilizam este registrador. Além da execução de outro processo que não é o que está sendo observado, a troca de contexto por si só também consome ciclos de clock.

Para medir a quantidade de ciclos de clock necessários, a nova chamada de sistema foi invocada de uma até mil vezes em um mesmo teste. Devido as possíveis variações que podem existir causadas pelos eventos e comportamentos do *hardware* e do sistema operacional, cada iteração do teste foi executada dez vezes e apenas o menor valor foi considerado.

Analisando o gráfico 5.1 nota-se que a quantidade de ciclos de *clock* necessários para execução da chamada de sistema cresce linearmente e é proporcional a quantidade de invocações feitas a nova chamada. Isto se dá a um fator próximo a 350 ciclos de clock adicionais por invocação.

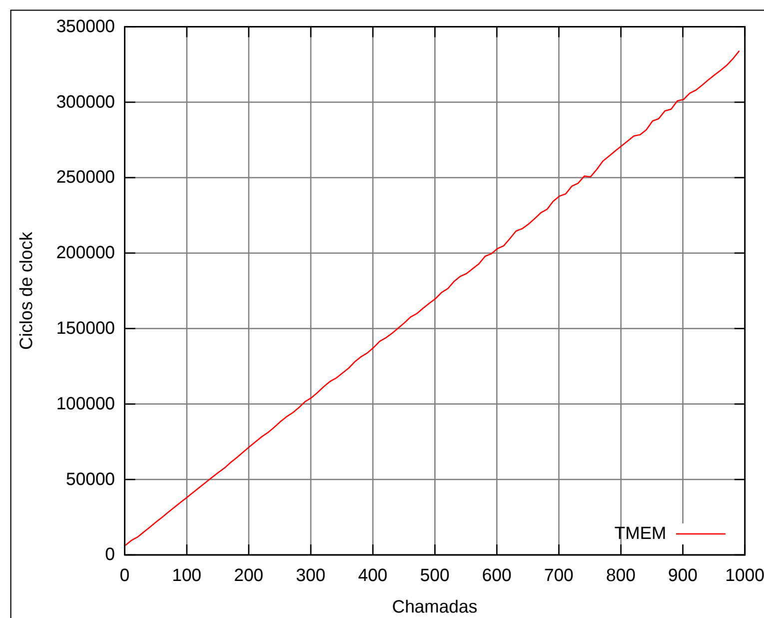


Figura 5.1: Ciclos de *clocks* necessários para execução da chamada de sistema.

5.2.2 Tempo

Para medir o tempo gasto pela chamada de sistema em espaço de usuário e em espaço de *kernel*, a nova chamada de sistema foi invocada de cem mil a um milhão de vezes. Observando a figura 5.2, nota-se que apesar das variações relacionadas aos eventos de hardware e de sistema operacional, o consumo de tempo cresce a medida em que cresce a quantidade de chamadas de

sistema. Em sistemas que executam 200.000 transações o tempo gasto em espaço de *kernel* é de aproximadamente 0.01 segundo e de aproximadamente 0.06 segundo para um milhão de transações. Isto faz com que o uso do TMEM seja aceitável em situações onde o desempenho não é fundamental, como durante o desenvolvimento de um sistema que utilizam memória transacional.

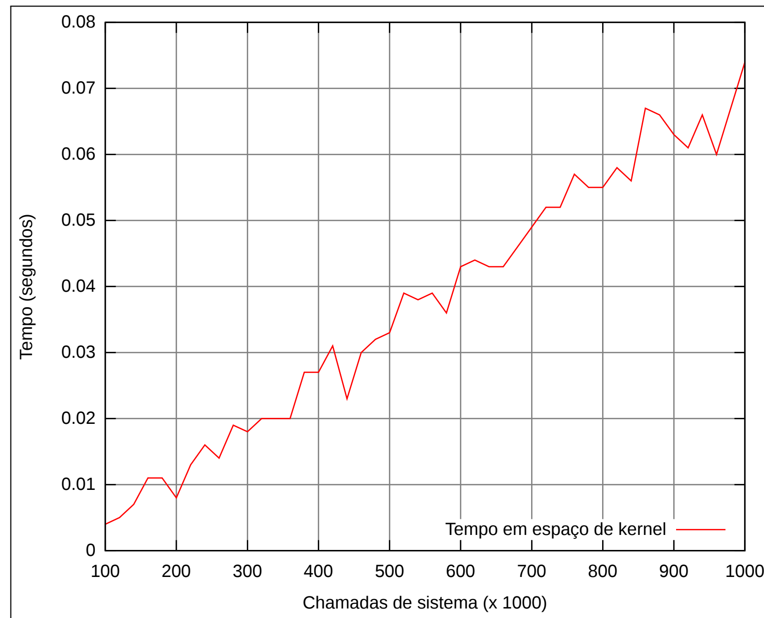


Figura 5.2: Duração da chamada de sistema em espaço de kernel.

É importante ressaltar que os teste citados nesta seção (5.2) foram feitos usando-se apenas um núcleo de CPU. Durante a utilização do TMEM em um sistema de memória transacional, o código usado pode ser *multithreaded* fazendo com que o tempo necessário para execução da chamada de sistema seja menor. Esta situação pode ser vista na seção 5.3.

5.3 Avaliação do sistema de memória transacional

Foram realizados testes na biblioteca RSTM para avaliar a sobrecarga causada pelo TMEM durante a sua utilização. Para isso os testes foram executados com o sistema de monitoração habilitado e desabilitado.

Para medir a intrusão causada pelo sistema de monitoração na implementação de memória transacional foram realizados experimentos com o *microbenchmark* que implementa uma árvore rubro-negra e que é distribuído com a biblioteca RSTM. Neste *microbenchmark* as transações inserem, buscam e removem elementos da árvore concorrentemente. Estes testes foram conduzidos variando-se o número de transações de cem mil a um milhão, sendo que o total de transações é a soma de transações finalizadas com sucesso.

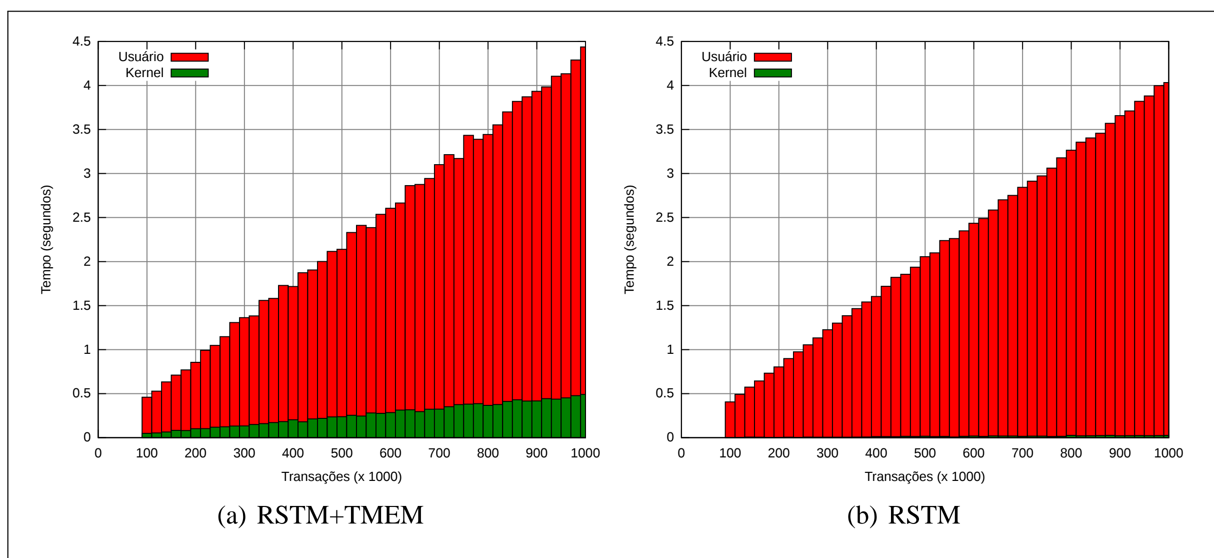
O resultado de uma execução de um *microbenchmark* que utiliza uma árvore rubro-negra pode ser visto na tabela 5.1. Com o uso do TMEM foi possível identificar quais eventos são mais frequentes para cada bloco atômico. Este teste foi executado por duas *threads* durante cinco segundos.

Tabela 5.1: Eventos de uma árvore rubro-negra com memória transacional

BlocoAtomicoID	Finalizadas	Abortadas
20 (busca)	1.948.875	7.789
24 (inserção)	1.891.228	17.148
25 (remoção)	1.889.822	12.565

Para verificar a intrusão no tempo imposta pelo sistema de monitoração mediu-se o tempo gasto em espaço de usuário e em espaço de *kernel* com o TMEM habilitado e desabilitado. O tempo gasto em espaço de *kernel* com sistema de monitoração desabilitado é baixo, variando de 0 segundo para cem mil transações a 0.026 segundo para um milhão de transações. Isto porque apenas operações comuns como alocação de memória e escalonamento de *threads* são executadas para este processo. Com o TMEM habilitado, o tempo é maior, variando de 0.049 segundo para cem mil transações a 0.491 segundo para um milhão transações. Isso se deve ao fato de que que grande parte do código necessário para monitoração concentra-se no *kernel*.

É importante notar que a intrusão imposta no tempo de usuário foi relativamente menor e que este permaneceu quase igual. Com o sistema de monitoração desabilitado o tempo gasto variou de 0.406 segundo para cem mil transações a 4.033 segundos para um milhão de transações, e de 0.459 a 4.438 segundos com o TMEM habilitado. Na figura 5.3 nota-se que o tempo gasto em espaço de usuário aumentou pouco com o uso do sistema de monitoração e que a maior parte da intrusão foi em espaço de *kernel*.

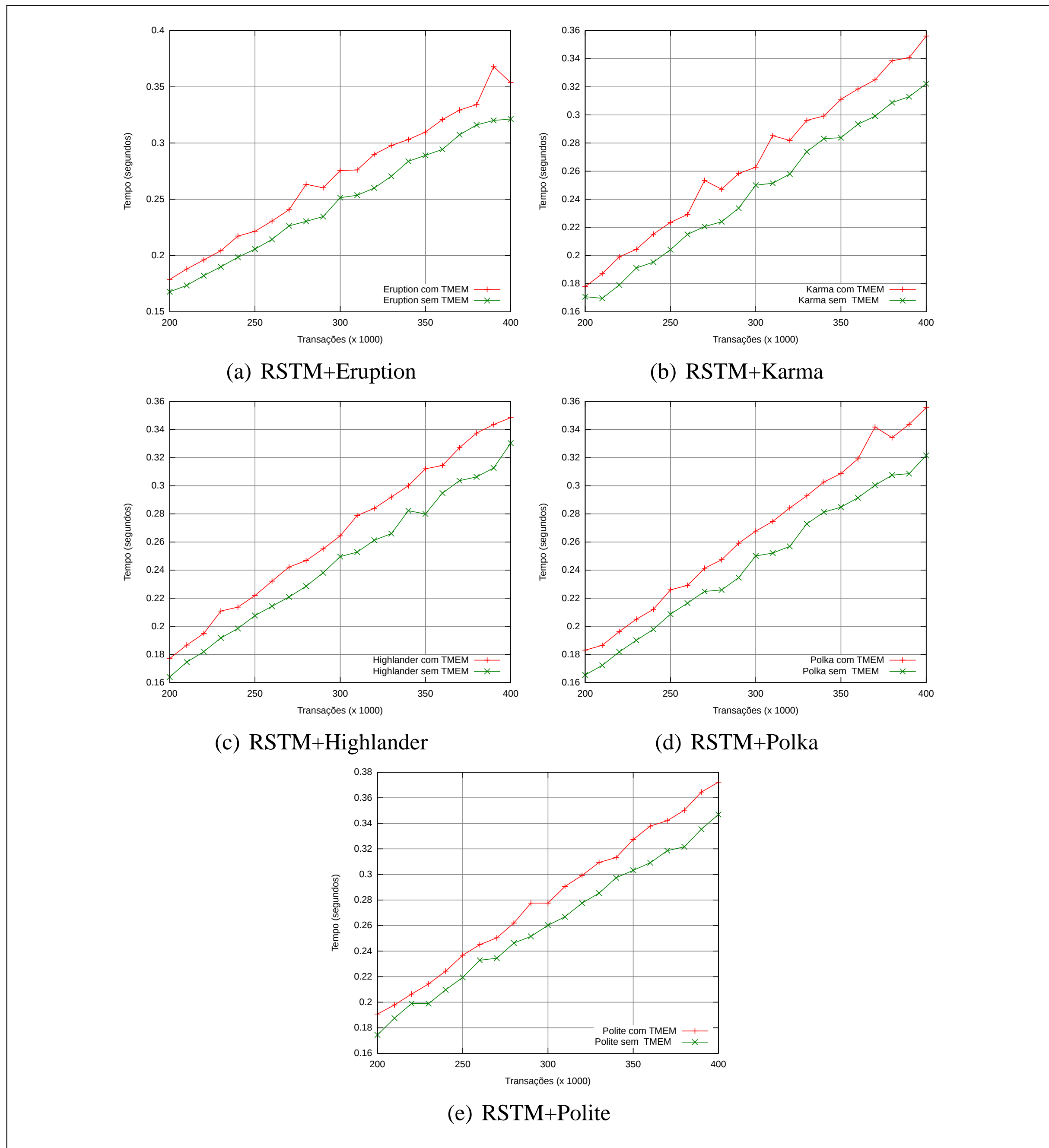


(a) RSTM+TMEM (b) RSTM

Figura 5.3: Tempo em espaço de usuário e em espaço de sistema

O tempo gasto varia de acordo com a quantidade de eventos ocorridos, quanto maior a quantidade de eventos, mais tempo o TMEM gasta para registrá-los. Na seção 2.3, viu-se que existem diversas abordagens na construção de um sistema de memória transacional.Em algumas situações onde é possível alterar o controle de concorrência sem comprometer a eficácia do programa pode-se, por exemplo, usar um controle pessimista de concorrência (seção 2.3.3), onde uma transação tem acesso exclusivo a um dado compartilhado, impedindo que outras transações o acessem concorrentemente. Isso reduz a quantidade de conflitos e consequentemente a quantidade de eventos, fazendo com que a intrusão imposta pelo TMEM seja ainda menor.

Como visto na seção 2.3.6, o uso de um gerenciador de contenção para solucionar conflitos causados entre as transações pode aumentar ou reduzir o número de transações abortadas e reexecutadas, aumentando ou reduzindo o número de eventos. A escolha correta de um gerenciador de contenção, pode fazer com que o tempo gasto pelo sistema de monitoração seja menor. Na figura 5.3 pode-se verificar que apesar da penalidade causada pelo sistema de monitoração, todos os gerenciadores de contenção ainda comportam-se da mesma maneira.



(a) RSTM+Eruption

(b) RSTM+Karma

(c) RSTM+Highlander

(d) RSTM+Polka

(e) RSTM+Polite

Figura 5.4: Microbenchmarks de árvore rubro-negra com e sem TMEM

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo é apresentado um resumo dos resultados desta pesquisa e serão apontados possíveis trabalhos futuros.

6.1 Conclusões

Ao longo do desenvolvimento deste trabalho vimos que apesar de sistemas de memória transacional solucionarem alguns dos principais problemas relacionados ao desenvolvimento de software concorrente, programas que usam esta estratégia ainda podem comportar-se de maneira errada. Para auxiliar o desenvolvedor que utiliza memória transacional apresentamos no capítulo 4 o TMEM, um sistema de monitoração de eventos relacionados à memória transacional capaz de monitorar programas que utilizam memória transacional sem a necessidade de interrompê-los e sem impor novas sincronizações entre as *threads*. O TMEM foi implementado nesta pesquisa para uma implementação de memória transacional em software e viu-se que sua extensibilidade para monitorar outros eventos e outras implementações de memória transacional é bastante simples.

No capítulo 5 foram apresentados os testes e vimos que a intrusão imposta pelo sistema de monitoração no tempo de usuário é baixa e que a maior parte do tempo gasto é computado como tempo de sistema. Na seção 4.8, viu-se que durante o desenvolvimento de um programa que utiliza memória transacional o TMEM pode auxiliar o desenvolvedor a encontrar transações problemáticas, como por exemplo transações que são iniciadas mas não são finalizadas - o que pode indicar o uso incorreto da API do sistema de memória transacional. Pode-se também encontrar transações que sofrem muitos conflitos e consequentemente são abortadas muitas vezes e isto pode indicar que a transação está com uma granularidade maior que necessária, além de dar a possibilidade ao desenvolvedor de comparar o desempenho de suas transações a cada modificação feita durante o desenvolvimento. Como visto na seção 2.3, existem diversas abordagens na construção de um sistema de memória transacional e como que cada uma delas pode afetar o desempenho de um programa e um sistema de monitoração como o TMEM também pode auxiliar na escolha correta de um gerenciador de contenção, por exemplo. Também foi visto como é possível criar uma chamada de sistema que incrementa contadores sem a necessidade de travas ou de sincronização, e isto pode ser útil para outros sistemas onde é necessário criar uma chamada de sistema *thread-safe* e não-bloqueante.

Como memória transacional ainda é o foco de diversas pesquisas, acreditamos que este trabalho pode auxiliar desenvolvedores e pesquisadores a tomarem decisões sobre quais estratégias

são mais adequadas ao usar e desenvolver sistemas de memória transacional.

Este trabalho totalizou mais de 1600 linhas de código. O leitor interessado nas implementações produzidas durante este estudo pode acessar o endereço `http://github.com/trajber/tmem` para obter mais informações.

6.2 Trabalhos futuros

Como ainda não existe um consenso entre os pesquisadores sobre quais abordagens são mais adequadas, a área de memória transacional ainda é o foco de diversas pesquisas e ainda existe muito trabalho a ser feito sobre monitoração de programas que utilizam esta abordagem para construção de programas concorrentes.

Como trabalhos futuros relacionados ao sistema de monitoração proposto nesta pesquisa, sugerimos que a monitoração seja ampliada para admitir informações sobre o conjunto de dados acessados pelas transações, podendo-se assim rastrear posições de memória conflitantes, verificar falsos conflitos e ter a exata noção do volume de dados manipulados por cada transação. Recomendamos também como trabalho futuro, o uso das informações fornecidas pelo sistema de monitoração pelo próprio sistema de memória transacional durante a execução do programa, onde o sistema de memória transacional seja capaz de tomar decisões para aumentar seu desempenho ou otimizar sua execução, e isto pode ser feito trocando-se algoritmos de gerenciamento de contenção dinamicamente, por exemplo.

Outra recomendação é verificar a viabilidade de adaptar o TMEM para uma linguagem que admita memória transacional e seja executada pela máquina virtual Java, como exemplo Clojure, e disponibilizar as informações sobre transações por meio de extensões JMX.

Algumas modificações também podem ser feitas no TMEM. A remoção da limitação imposta pelo tamanho do vetor que representa as transações no *kernel* usando-se, ao invés de um simples vetor, estruturas de dados como listas ligadas, por exemplo. Pelos motivos vistos na seção 4.2.1, o sistema de monitoração proposto nesta pesquisa monitora apenas os eventos comuns entre todas implementações de memória transacional e em um uso real é desejável que todos os eventos ocorridos sejam monitorados, adicionalmente a esta modificação deve-se verificar se a quantidade de novos eventos monitorados pode prejudicar o desempenho do sistema de memória transacional. Pode-se também modificar a implementação de memória transacional para que cada bloco atômico receba automaticamente um identificador, excluindo assim a possibilidade do usuário identificar dois ou mais blocos com o mesmo número. Outra modificação interessante é implementar níveis de monitoração onde seja possível definir apenas um conjunto de eventos que devem ser notificados ao sistema operacional.

REFERÊNCIAS

ADL-TABATABAI, A.-R. et al. Compiler and runtime support for efficient software transactional memory. In: **PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation**. New York, NY, USA: ACM, 2006. p. 26–37. ISBN 1-59593-320-4.

AGRAWAL, K.; LEISERSON, C. E.; SUKHA, J. Memory models for open-nested transactions. In: **MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness**. New York, NY, USA: ACM, 2006. p. 70–81. ISBN 1-59593-578-9.

CHAFI, H. et al. Tape: a transactional application profiling environment. In: **ICS '05: Proceedings of the 19th annual international conference on Supercomputing**. New York, NY, USA: ACM, 2005. p. 199–208. ISBN 1-59593-167-8.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional locking ii. In: **In Proc. of the 20th Intl. Symp. on Distributed Computing**. [S.l.: s.n.], 2006.

ENNALS, R.; ENNALS, R. **Software transactional memory should not be obstruction-free**. [S.l.], 2006.

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In: **Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)**. [S.l.: s.n.], 2008.

GUERRAOUI, R.; HERLIHY, M.; POCHON, B. Polymorphic contention management. In: **DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing**. [S.l.]: LNCS, Springer, 2005. p. 303–323.

GUERRAOUI, R.; HERLIHY, M.; POCHON, B. Toward a theory of transactional contention managers. In: **PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing**. New York, NY, USA: ACM, 2005. p. 258–264. ISBN 1-59593-994-2.

HARRIS, T.; FRASER, K. Language support for lightweight transactions. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 38, n. 11, p. 388–402, 2003. ISSN 0362-1340.

HARRIS, T.; LARUS, J.; RAJWAR, R. **Transactional Memory, 2nd Edition**. [S.l.]: Morgan and Claypool Publishers, 2010. ISBN 1608452352.

HARRIS, T. et al. Composable memory transactions. In: **PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming**. New York, NY, USA: ACM, 2005. p. 48–60. ISBN 1-59593-080-9.

HERLIHY, M. Wait-free synchronization. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 13, n. 1, p. 124–149, 1991. ISSN 0164-0925.

HERLIHY, M. Obstruction-free synchronization: Double-ended queues as an example. In: **In Proceedings of the 23rd International Conference on Distributed Computing Systems**. [S.l.]: IEEE Computer Society, 2003. p. 522–529.

HERLIHY, M.; LEV, Y. tm_db: A generic debugging library for transactional programs. In: **PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques**. Washington, DC, USA: IEEE Computer Society, 2009. p. 136–145. ISBN 978-0-7695-3771-9.

HERLIHY, M. et al. Software transactional memory for dynamic-sized data structures. In: **PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing**. New York, NY, USA: ACM, 2003. p. 92–101. ISBN 1-58113-708-7.

HERLIHY, M. et al. Transactional memory: Architectural support for lock-free data structures. In: **in Proceedings of the 20th Annual International Symposium on Computer Architecture**. [S.l.: s.n.], 1993. p. 289–300.

LEV, Y.; MOIR, M. Debugging with transactional memory. In: **Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing**. [S.l.: s.n.], 2006.

LOMET, D. B. Process structuring, synchronization, and recovery using atomic actions. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 12, n. 3, p. 128–137, 1977. ISSN 0362-1340.

LOURENCO, J. et al. Understanding the behavior of transactional memory applications. In: **Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging**. New York, NY, USA: ACM, 2009. (PADTAD '09), p. 3:1–3:9. ISBN 978-1-60558-655-7. Disponível em: <http://doi.acm.org/10.1145/1639622.1639625>.

MARATHE, V. J.; MOIR, M. Toward high performance nonblocking software transactional memory. In: **PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming**. New York, NY, USA: ACM, 2008. p. 227–236. ISBN 978-1-59593-795-7.

MARATHE, V. J. et al. Lowering the overhead of nonblocking software transactional memory. In: **Dept. of Computer Science, Univ. of Rochester**. [S.l.: s.n.], 2006.

MOORE, K. E. et al. Logtm: Log-based transactional memory. In: **in HPCA**. [S.l.: s.n.], 2006. p. 254–265.

OLUKOTUN, K.; HAMMOND, L. The future of microprocessors. **Queue**, ACM, New York, NY, USA, v. 3, n. 7, p. 26–29, 2005. ISSN 1542-7730.

PORTER, D. E. et al. Operating systems transactions. In: **SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles**. New York, NY, USA: ACM, 2009. p. 161–176. ISBN 978-1-60558-752-3.

RAMADAN, H. E. et al. Metatm/txlinux: transactional memory for an operating system. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 35, p. 92–103, June 2007. ISSN 0163-5964. Disponível em: <http://doi.acm.org/10.1145/1273440.1250675>.

SCHERER III, W. N.; SCOTT, M. L. Advanced contention management for dynamic software transactional memory. In: **PODC ’05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing**. New York, NY, USA: ACM, 2005. p. 240–248. ISBN 1-59593-994-2.

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: **PODC ’95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing**. New York, NY, USA: ACM, 1995. p. 204–213. ISBN 0-89791-710-3.

SPEAR, M. F.; MICHAEL, M.; SCOTT, M. L. Inevitability mechanisms for software transactional memory. In: **TRANSACT ’08: 3rd Workshop on Transactional Computing**. [S.l.: s.n.], 2008.

SPEAR, M. F. et al. Nonblocking transactions without indirection using alert-on-update. In: **Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures**. New York, NY, USA: ACM, 2007. (SPAA ’07), p. 210–220. ISBN 978-1-59593-667-7. Disponível em: <http://doi.acm.org/10.1145/1248377.1248414>.

SPEAR, M. F. et al. Implementing and exploiting inevitability in software transactional memory. In: **ICPP ’08: Proceedings of the 2008 37th International Conference on Parallel Processing**. Washington, DC, USA: IEEE Computer Society, 2008. p. 59–66. ISBN 978-0-7695-3374-2.

WELC, A.; SAHA, B.; ADL-TABATABAI, A.-R. Irrevocable transactions and their applications. In: **SPAA ’08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures**. New York, NY, USA: ACM, 2008. p. 285–296. ISBN 978-1-59593-973-9.

ZYULKYAROV, F. et al. Atomic quake: using transactional memory in an interactive multiplayer game server. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 44, n. 4, p. 25–34, 2009. ISSN 0362-1340.

ZYULKYAROV, F. et al. Debugging programs that use atomic blocks and transactional memory. In: **PPoPP ’10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming**. New York, NY, USA: ACM, 2010. p. 57–66. ISBN 978-1-60558-877-3.