Memórias Transacionais: Prototipagem e Simulação de Implementações em *Hardware* e uma Caracterização para o Problema de Gerenciamento de Contenção em *Software*

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Fernando André Kronbauer e aprovada pela Banca Examinadora.

Campinas, 8 de janeiro de 2009.

Sandro Rigo (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Crisllene Queiroz Custódio – CRB8a 162/2005

Kronbauer, Fernando André.

K922m Memórias transacionais : prototipagem e simulação de implementações em hardware e uma caracterização para o problema de gerenciamento de contenção em software / Fernando André Kronbauer -- Campinas, [S.P. : s.n.], 2009.

Orientador : Sandro Rigo

Dissertação (Mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Memória transacional. 2. Arquitetura de computador. 3. Prototipagem rápida. 4. Simulação (Computadores). 5. Controle de concorrência. I. Rigo, Sandro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

(cqc/imecc)

Título em inglês: Transactional Memories: Prototyping and Simulation of Hardware Implementations and a Characterization of the Problem of Contention Management in Software

Palavras-chave em inglês (Keywords): 1. Transactional memories. 2. Computer architecture. 3. Rapid prototyping. 4. Simulation, Computer. 5. Concurrency control.

Área de concentração: Sistemas de Informação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Sandro Rigo (IC - Unicamp)
Prof. Dr. Paulo César Centoducatte (IC - Unicamp)
Prof. Dr. Luiz Cláudio Villar dos Santos (UFSC)

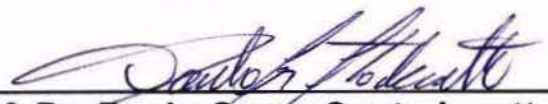
Data da defesa: 07/11/2008

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 07 de novembro de 2008, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Luiz Cláudio Villar dos Santos
Departamento de Informática e Estatística / UFSC.



Prof. Dr. Paulo Cesar Centoducatte
IC – UNICAMP.

Prof. Dr. Sandro Rigo
IC – UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Memórias Transacionais: Prototipagem e Simulação de Implementações em *Hardware* e uma Caracterização para o Problema de Gerenciamento de Contenção em *Software*

Fernando André Kronbauer[1]

Janeiro de 2009

Banca Examinadora:

- Sandro Rigo (Orientador)
- Paulo César Centoducatte
  Instituto de Computação - UNICAMP
- Luiz Cláudio Villar dos Santos
  Departamento de Informática e Estatística - UFSC
- Rodolfo Azevedo (Suplente)
  Instituto de Computação - UNICAMP
- Ivan Luiz Marques Ricarte (Suplente)
  Faculdade de Engenharia Elétrica e de Computação - UNICAMP

[1]Suporte financeiro de: Bolsa da CAPES (processo 01P-18945/2006) 2006–2007

Resumo

Enquanto que arquiteturas paralelas vão se tornando cada vez mais comuns na indústria de computação, mais e mais programadores precisam escrever programas paralelos e desta forma são expostos aos problemas relacionados ao uso dos mecanismos tradicionais de controle de concorrência. Memórias transacionais têm sido propostas como um meio de aliviar as dificuldades encontradas ao escreverem-se programas paralelos: o desenvolvedor precisa apenas marcar as seções de código que devem ser executadas de forma atômica e isolada — na forma de transações, e o sistema cuida dos detalhes de sincronização. Neste trabalho exploramos propostas de memórias transacionais com suporte específico em *hardware* (HTM), desenvolvendo uma plataforma flexível para a prototipagem, simulação e caracterização destes sistemas. Também exploramos um sistema de memória transacional com suporte apenas em *software* (STM), apresentando uma abordagem nova para gerenciar a contenção entre transações. Esta abordagem leva em consideração os padrões de acesso aos diferentes dados de um programa ao escolher o gerenciador de contenção a ser usado para o acesso a estes dados. Elaboramos uma modificação da plataforma de STM que nos permite realizar esta associação entre dados e gerenciamento de contenção, e a partir desta implementação realizamos uma caracterização baseada nos padrões de acesso aos dados de um programa executando em diferentes sistemas de computação. Os resultados de nosso trabalho mostram a viabilidade do uso de memórias transacionais em um ambiente de pesquisa acadêmica, e apontam caminhos para a realização de trabalhos futuros que aumentem a viabilidade do seu uso também pela indústria.

Abstract

As parallel architectures become prevalent in the computer industry, more and more programmers are required to write parallel programs and are thus being exposed to the problems related to the use of traditional mechanisms for concurrency control. Transactional memory has been devised as a means for easing the burden of writing parallel programs: the programmer has only to mark the sections of code that are to be executed in an atomic and isolated way — in the form of transactions, and the system takes care of the synchronization details. In this work we explore different proposals of transactional memories based on specific hardware support (HTM), developing a flexible platform for the prototyping, simulation and characterization of these systems. We also explore a transactional memory system based solely on software support (STM), devising a novel approach for managing the contention among transactions. This new approach takes into account the access patterns to different data in an application when choosing the contention management strategy to be used for the access to these data. We made modifications to the STM system in order to enable the association of the data with the contention manager algorithm, and using the new implementation we characterized the STM system based on the access patterns to the data of a program, running it on different hardware configurations. Our results show the viability of the use of transactional memories in an academic environment, and serve as a basis for the proposal of different directions to be followed in future research work, aimed at leveraging the use of transactional memories by the industry.

Agradecimentos

Eu gostaria de agradecer aos meus colegas da pós graduação, em especial àqueles com quem morei durante o tempo em que estive envolvido com o mestrado. Gostaria de agradecer ao meu orientador, pelo conhecimento passado, e por estar sempre disponível para esclarecer as minhas dúvidas.

Sumário

Resumo **v**

Abstract **vi**

Agradecimentos **vii**

1 Introdução **1**

2 Introdução a memórias transacionais **4**
2.1 Transações 4
2.2 Construções transacionais básicas 7
2.3 Espaço de Projeto 15
2.3.1 Implementação em *hardware* ou em *software* 15
2.3.2 Detecção e gerenciamento de conflitos 15
2.3.3 Granularidade de detecção de conflitos 16
2.3.4 Registros de escrita versus registros de recuperação 16
2.3.5 Isolamento forte versus isolamento fraco 17
2.3.6 Transações intermitentes ou contínuas 18

3 Plataforma de prototipagem de TM com suporte específico em *hardware* **19**
3.1 Trabalhos relacionados 19
3.1.1 Proposta inicial de Herlihy e Moss para a implementação de TM . . 19
3.1.2 TCC (*Transactional memory Coherence and Consistency*) 21
3.1.3 UTM (*Unbounded Transactional Memory*) 21
3.1.4 VTM (*Virtual Transactional Memory*) 22
3.1.5 LogTM 22
3.1.6 *Hybrid Transactional Memory* 23
3.1.7 *Bulk* HTM 23
3.1.8 Processador UltraSPARC T2 “Rock” 23

3.1.9 Plataformas de simulação e prototipagem 24
3.2 A plataforma de prototipagem desenvolvida 25
3.3 Primeiro estudo de caso: a proposta de Herlihy e Moss 28
3.3.1 Construção da plataforma 29
3.3.2 Avaliação do protótipo 30
3.4 Segundo estudo de caso: aninhamento de transações 38
3.4.1 Construção da plataforma 38
3.4.2 Avaliação do protótipo 41

4 Memória transacional com suporte em *software* e experimentos com gerenciamento de contenção 45
4.1 Trabalhos relacionados 45
4.1.1 STM de Shavit e Touitou 45
4.1.2 DSTM 46
4.1.3 WSTM 46
4.1.4 STM de Ananian e Rinard 46
4.1.5 TL 47
4.1.6 TinySTM 47
4.1.7 McRT-STM 47
4.1.8 RSTM 48
4.1.9 Outros trabalhos sobre gerenciamento de contenção 49
4.2 Trabalho realizado 50
4.2.1 Gerenciamento de contenção em RSTM 50
4.2.2 Modificações Propostas 51
4.2.3 Experimentos 53
4.2.4 Considerações finais 63

5 Conclusões 64

Bibliografia 66

Capítulo 1

Introdução

Conforme sistemas uniprocessados alcançam os limites de exploração do paralelismo de nível de instrução (ILP), e problemas como super aquecimento e alto consumo de potência limitam a utilização dos métodos tradicionais de aumento do poder de processamento, arquiteturas paralelas provêm uma alternativa realista para a construção de sistemas de alto desempenho, permitindo que programadores explorem o paralelismo em nível de linha de execução inerente a muitas tarefas. Como resultado, sistemas multiprocessados vêm tornando-se a norma em todos os níveis de computação, dos servidores aos sistemas embarcados, com destaque aos *chips* multiprocessados (CMPs) nos quais vários núcleos de processamento são postos em uma mesma pastilha de silício (*die*). Estes novos sistemas multiprocessados de memória compartilhada, apresentam ao mesmo tempo uma grande oportunidade e um grande desafio. A oportunidade é que núcleos de processamento independentes estarão disponíveis em um número sem precedentes, e o desafio é que um número cada vez maior de programadores será exposto aos problemas relacionados à programação paralela.

Para que linhas de execução (*threads*) possam cooperar na realização de trabalho útil, é necessário que se comuniquem de forma segura, ou seja, dados compartilhados entre as linhas precisam ser acessados de forma ordeira, coordenada e síncrona. Atualmente esta coordenação entre linhas de execução é de responsabilidade do programador, que possui à sua disposição somente mecanismos de baixo nível, como travas de exclusão mútua (*locks*) e semáforos, para prevenir que duas ou mais linhas de execução concorrentes interfiram umas com as outras. Mesmo linguagens modernas como Java e C# provêm uma construção de linguagem apenas um nível um pouco mais abstrato — o monitor, que evita acessos simultâneos aos dados internos de um objeto. Estes mecanismos de baixo nível de abstração são notoriamente difíceis de usar corretamente, e introduzem problemas variados, como inversão de prioridade, enfileiramento (*convoying*) e bloqueio mútuo (*deadlock*):

- *Inversão de prioridade*: quando uma linha de execução que adquiriu determinada trava de exclusão tem a permissão de execução temporariamente revogada (preempção) para que uma linha de maior prioridade seja executada, porém esta última precisa adquirir a mesma trava mantida pela primeira para poder executar, é possível que se forme um cenário de *inversão de prioridade*. Para tanto, é necessário que uma terceira linha de execução com prioridade intermediária também passe a executar, impedindo que a linha com prioridade baixa execute e libere a trava de exclusão necessária para a execução da linha de prioridade alta. Assim, há uma inversão de prioridades entre a linha de execução de prioridade média e a linha de prioridade alta.

- *Enfileiramento*: enfileiramento (*convoying*) pode ocorrer quando uma linha que adquiriu determinada trava de exclusão é retirada de execução, talvez por ter exaurido sua quota de processamento, por uma falha de página, ou por algum outro tipo de interrupção. Quando isto acontece, outras linhas capazes de executar mas que dependem da mesma trava não conseguem dar prosseguimento ao seu trabalho enquanto a trava não for liberada.

- *Bloqueio mútuo*: bloqueio mútuo (*deadlock*) pode ocorrer se linhas de execução distintas tentam adquirir um conjunto de travas em ordens diferentes. Um ciclo de dependências pode se formar, no qual cada linha de execução espera a outra liberar um recurso crítico, mas nenhuma libera o recurso que já adquiriu. Evitar bloqueios mútuos pode ser complicado, particularmente se o conjunto de travas a ser adquirido não for conhecido com antecedência.

Em geral, a programação paralela é considerada mais difícil que a programação seqüencial. Paralelismo e não determinismo aumentam em muito a quantidade de informações que um desenvolvedor de *software* deve manter em mente enquanto programa. Conseqüentemente, poucas pessoas são capazes de arazoar de forma consistente sobre o comportamento de um programa paralelo. O problema é agravado pelo fato de ferramentas de desenvolvimento e análise, que compensam a falibilidade humana ao sistematicamente identificar defeitos em programas, não conseguirem analisar o código de programas paralelos com a mesma eficiência com que o fazem com códigos seqüenciais. Erros que se manifestam durante a execução de um programa paralelo não são sempre facilmente reproduzíveis e muitas vezes sua origem é de difícil identificação. E ainda, o modelo de programação paralela em geral se encontra em um nível de abstração tão baixo que muitas vezes é mapeado diretamente a operações de baixo nível implementadas em *hardware*.

Uma estratégia alternativa à sincronização tradicional é a sincronização não-bloqueante, que provê exclusão mútua sem a utilização de travas de exclusão. Sistemas implementados de forma não-bloqueante possuem desempenho em geral melhor que sistemas que

utilizam sincronização bloqueante. No entanto, a programação não-bloqueante tradicional não somente é mais difícil que a programação com travas de exclusão, mas também aparenta ser mais suscetível a erros. Uma solução comum a estes problemas é encapsular a funcionalidade não-bloqueante em bibliotecas primitivas, porém esta estratégia limita a generalidade com que tal paradigma de programação pode ser utilizado pelos programadores.

Sistemas de memória transacional — TM (do inglês, *Transactional Memory*) — foram propostos como um modelo de programação genérico e flexível, que permite que linhas de execução leiam e escrevam posições de memória em uma única operação e de maneira atômica através de transações, da mesma forma com que transações em um banco de dados são capazes de modificar vários registros no disco atomicamente. Memórias transacionais se baseiam no conceito de transações atômicas capazes de prover sincronização mútua sem os detalhes complicados dos protocolos de sincronização convencionais.

No capítulo 2 discorremos a respeito das propriedades das transações que tornam as memórias transacionais um modelo atraente para o desenvolvimento de programas paralelos, e tratamos do espaço de projeto a ser considerado para o desenvolvimento de um sistema de TM. Como memórias transacionais ainda não são amplamente conhecidas, entendemos que merecem um capítulo individual para a sua introdução. As contribuições do nosso trabalho são descritas nos dois capítulos seguintes. O capítulo 3 trata da plataforma de prototipagem e simulação de sistemas de *hardware* CMP, desenvolvida ao longo da fase inicial de nosso trabalho, usada para o estudo de diferentes propostas de implementação de memórias transacionais com suporte *hardware* específico (HTM). Esta plataforma resultou na primeira publicação sobre TM do nosso instituto [27]. No capítulo 4 tratamos do uso de uma abordagem de implementação de memórias transacionais com suporte apenas em *software* (STM), e experimentos com uma técnica nova de gerenciamento de contenção entre transações conflitantes. Modificamos a biblioteca de STM de modo a permitir a associação entre dados e gerenciadores de contenção, e a partir desta implementação realizamos a caracterização da execução de um *benchmark* baseada nos padrões de acesso aos dados do programa, executado em sistemas de computação variados.

Como os capítulos 3 e 4 tratam de abordagens bem diferentes de implementação de memórias transacionais, resolvemos dividir os trabalhos relacionados e torná-los parte destes dois capítulos, colocando cada trabalho relacionado no capítulo onde é mais relevante. No capítulo final fazemos um resumo das contribuições do nosso trabalho à pesquisa na área de TM e apresentamos possíveis direções para o desenvolvimento de trabalhos futuros.

Capítulo 2

Introdução a memórias transacionais

Neste capítulo fazemos uma breve introdução a memórias transacionais, por se tratarem de um conceito bastante novo ao qual muitos leitores não estão familiarizados. Baseamos parte deste capítulo em um livro recentemente publicado, escrito por Larus e Rajwar [29], fazendo as devidas adaptações e acrescentando várias informações que julgamos faltar no livro citado. Alguns dos exemplos de código também foram inspirados nos exemplos publicados naquele livro.

2.1 Transações

Uma *transação* é uma seqüência de ações que parecem indivisíveis e instantâneas a um observador externo. O termo *transação* é comumente empregado pela literatura sobre bancos de dados e sistemas distribuídos. Uma transação em um banco de dados possui quatro atributos específicos: atomicidade perante falhas, consistência, isolamento e durabilidade — coletivamente conhecidas como propriedades ACID.

A propriedade de *atomicidade* requer que todas as ações que fazem parte de uma transação completem de forma bem-sucedida, ou então que nenhuma delas pareça ter executado. Não é aceitável que alguma ação de uma transação falhe e que esta termine de forma bem-sucedida. Tão pouco é aceitável que uma transação que falhou deixe para trás evidências de que tenha executado. Uma transação que terminou de forma bem-sucedida foi *completada*, enquanto que uma que falhou foi *abortada*. Chamamos esta propriedade de *atomicidade perante falhas*, para distinguir da noção de *atomicidade de execução*, que abrange elementos de outras propriedades ACID.

A propriedade seguinte é chamada de *consistência*. Uma transação pode modificar o estado do seu ambiente, por exemplo, dados em um banco de dados ou em memória. Estas mudanças devem deixar tal estado consistente, uma vez que transações subseqüentes começam a executar a partir deste estado modificado. Transações subseqüentes podem

não possuir nenhum conhecimento sobre quais transações executaram anteriormente, portanto não se pode esperar que uma transação execute apropriadamente se a anterior deixou o ambiente em um estado arbitrário. O significado de consistência depende inteiramente das semânticas de cada programa. Tipicamente, consiste de um conjunto de invariantes a respeito de estruturas de dados. Por exemplo, que **numConsumidores** contém o número de itens da tabela **Consumidores**, ou que a tabela **Consumidores** não contém entradas duplicadas.

A propriedade de *isolamento* requer que cada transação produza um resultado correto, independentemente de quais transações estejam executando concorrentemente. Uma transação não pode perceber o estado de execução intermediário de outras transações, pois de outra forma a propriedade de consistência poderia ser violada.

A propriedade final, *durabilidade*, requer que uma vez que uma transação seja completada, seus resultados se tornem permanentes (isto é, armazenados em um meio durável, como por exemplo um disco) e disponível a todas as transações subseqüentes. Esta propriedade não é importante em memórias transacionais, uma vez que dados em memória normalmente são transientes.

Em 1977, Lomet observou que uma abstração parecida com a das transações usadas em bancos de dados poderia ser um bom mecanismo de linguagem para assegurar a consistência dos dados compartilhados entre vários processos [30]. O artigo não descrevia uma implementação prática capaz de competir com a sincronização explícita, e a idéia permaneceu adormecida até que Herlihy e Moss, em 1993, propuseram a *memória transacional* com suporte de *hardware* como mecanismo para a construção de estruturas de dados livres de bloqueio (*lock-free*) [25]. Nos últimos anos, houve um grande aumento no interesse por métodos baseados tanto em *hardware* quanto em *software* para a implementação de memórias transacionais.

A idéia básica é bastante simples. As propriedades ACI das transações (*atomicidade*, *consistência* e *isolamento*) provêm uma abstração conveniente para a coordenação de leituras e escritas a dados compartilhados em um sistema com múltiplas linhas de execução ou processos. Acessos a dados compartilhados têm sua origem em computações executando em linhas de execução concorrentes que por sua vez executam em um ou mais processadores. Sem um mecanismo para coordenar estes acessos, leituras e escritas de várias computações podem sobrepor-se de forma a produzir resultados inconsistentes, incorretos e não determinísticos.

Um programa pode encapsular determinada computação em uma transação. A *atomicidade perante falhas* assegura que a computação termina de forma bem-sucedida e emite todos os seus resultados para a memória compartilhada ou então aborta. Além disto, a propriedade de *isolamento* assegura que a transação produz os mesmos resultados que produziria caso nenhuma outra transação estivesse executando concorrentemente.

Apesar de *isolamento* parecer ser a principal garantia oferecida por memórias transacionais, as demais propriedades, *atomicidade perante falhas* e *consistência*, também são importantes. Se o objetivo de um programador é construir um programa correto, então *consistência* é importante, uma vez que transações podem executar em ordens imprevisíveis. Seria difícil escrever código correto sem poder considerar que uma transação inicia sua execução em um estado consistente. A *atomicidade perante falhas* é portanto uma parte importante do processo que assegura a consistência. Se uma transação falhar, pode deixar os dados em um estado imprevisível que, se mantido, faria com que transações subseqüentes também falhassem. Além disto, o mecanismo usado para implementar a atomicidade perante falhas, retornando os dados ao seu estado anterior ao da execução da transação, revela-se extremamente importante para a implementação de outros tipos de mecanismos de controle de concorrência, como o bloqueio condicional de linhas de execução [24].

Transações em memória diferem das usadas em bancos de dados, e conseqüentemente requerem novas técnicas de implementação. As seguintes diferenças estão entre as mais importantes:

- Dados em um banco de dados residem em disco, em vez de memória. Acessos a discos levam de 5 a 10 ms, ou literalmente tempo suficiente para executar milhões de instruções. Bancos de dados podem portanto fazer um *trade-off* entre computação e acessos a disco. Memórias transacionais fazem acessos a memória, que incorrem em um custo de no máximo algumas centenas de instruções, e desta forma uma transação não pode executar muita computação a cada acesso a memória. Suporte em *hardware* é portanto mais atraente para TM que para sistemas de bancos de dados.

- Memórias transacionais não são duráveis, uma vez que dados na memória não sobrevivem ao término do programa. Isto simplifica a implementação de TM, uma vez que a obrigação de gravar os dados permanentemente em um meio durável quando uma transação completa complica consideravelmente a implementação de transações em bancos de dados.

- Memórias transacionais devem adaptar-se a um ambiente rico e complexo, repleto de linguagens e outros paradigmas de programação, bibliotecas, programas e sistemas operacionais. Para serem bem sucedidos, sistemas de memória transacional devem coexistir com a infraestrutura existente, mesmo que o objetivo de longo prazo seja suplantar com transações partes deste ambiente. Programadores acharão difícil adotar o novo paradigma de programação oferecido por memórias transacionais caso isto requeira mudanças pervasivas em linguagens de programação, bibliotecas e

sistemas operacionais, ou os limitem a um ambiente fechado, como bancos de dados, onde a única forma de acessar dados é através de transações.

2.2 Construções transacionais básicas

Nesta seção buscaremos descrever as características desejáveis em um sistema de memória transacional, seja sua implementação em *hardware* ou em *software*. Infelizmente, estes requisitos ainda são incompletos e pouco específicos, uma vez que de uma forma geral falta experiência aos pesquisadores para escolher entre as diferentes alternativas de projeto, e podemos apenas especular a respeito da complexidade de implementação e dificuldade de uso, por parte dos programadores, destes atributos apenas propostos ou incompletamente implementados.

Para simplificar, descreveremos apenas extensões básicas de uma linguagem parecida com Java. Estas extensões são propostas iniciais feitas por pesquisadores, e não uma proposta final de extensão de linguagem para dar suporte a memória transacional. Linguagens não seguras, como C ou C++, podem requerer implementações diferentes, no entanto as construções de linguagem provavelmente serão parecidas.

A maior parte da pesquisa sobre memórias transacionais tem seu foco em seu uso em programação paralela, portanto manteremos o foco neste aspecto, em vez de ocuparmonos com seu uso em recuperação na presença de erros, programação de tempo real, ou programação multitarefa. Em um programa paralelo, mais de uma linha de execução executa concorrentemente, o que requer um controle de concorrência para evitar que as linhas de execução acessem recursos compartilhados de forma conflitante e para coordenar as ações destas linhas. Memórias transacionais provêm mecanismos para controlar ambos os aspectos da concorrência.

Exclusão mútua é um mecanismo que evita que vária linhas de execução acessem um recurso compartilhado (por exemplo, um vetor, objeto, ou estrutura de dados) simultaneamente. Podemos ignorar o caso benigno de compartilhamento para leitura, no qual linhas de execução apenas lêem o recurso, uma vez que acessos concorrentes causam problemas apenas quando uma linha de execução modifica tal recurso. Quando isto acontece, outras linhas de execução podem ler valores inconsistentes ou ver estados intermediários da computação. Se várias linhas de execução modificam o recurso simultaneamente, então o resultado de qualquer linha pode ser sobrescrito, e o estado final do recurso pode não corresponder à computação de nenhuma linha. Travas de exclusão precisam ser conservadoras quando utilizadas para implementar a exclusão mútua. Mesmo que as operações sejam em sua maior parte leituras e escritas sejam infreqüentes, o programa precisa pagar o preço da sincronização.

Coordenação é outro motivo que leva ao controle de concorrência. Um programa pode

requerer que o trabalho de duas linhas de execução independentes seja coordenado de forma que, por exemplo, a linha 2 execute após a linha 1 executar determinada ação. Sem coordenação, as linhas de execução seguem independentemente, e a linha 1 pode produzir seu resultado ao mesmo tempo, ou mesmo depois, de a linha 2 executar.

Transações não são o único meio de controlar a computação paralela, no entanto muito do recente interesse por memórias transacionais deve-se à crença amplamente aceita de que transações oferecem um modelo de programação de mais alto nível e menos propenso a erros que alternativas mais bem conhecidas tais como travas de exclusão, semáforos e monitores.

A construção **atomic**, por exemplo, delimita um bloco de código que deve executar dentro de uma transação:

```
atomic {
  if (x != null) x.foo();
  y = true;
}
```

As semânticas desta construção serão discutidas a seguir, mas por enquanto consideraremos que o bloco executa sob as propriedades de atomicidade perante falhas e isolamento de uma transação. A transação resultante possui escopo dinâmico, isto é, encompassa todo o código executado enquanto o fluxo de controle está dentro do bloco **atomic**, independentemente de o código estar lexicograficamente cercado pelo bloco. Assim, por exemplo, o código na função **foo** também executa transacionalmente.

Um bloco **atomic** não nomeia recursos, sejam dados ou mecanismos de sincronização. Esta característica distingue as transações de construções programáticas anteriores, tais quais monitores, nos quais o programador explicitamente nomeia o dado protegido pela seção crítica. Também diferencia blocos **atomic** da sincronização baseada em travas de exclusão, na qual o programador especificamente nomeia a trava protegendo os dados. Nomear o recurso usado em uma abstração expõe detalhes de implementação, e viola a sua natureza auto-contida.

Transações, ao contrário, especificam o resultado desejado da execução (atomicidade perante falhas e isolamento), e confiam no sistema de memória transacional para implementá-lo. Como resultado, um bloco **atomic** permite que a abstração esconda os detalhes de implementação e que seja componível com respeito a estas propriedades. *Composição* é o processo de criação de *software* a partir de componentes — abstração na qual os detalhes internos são escondidos. A programação é muito mais difícil e suscetível a erros se o programador não pode depender da interface especificada por um objeto e deve entender sua implementação

Usar travas de exclusão em código de aplicativos para atingir exclusão mútua expõe muitos dos detalhes de implementação que gostaríamos de esconder. Se um método em uma biblioteca acessa uma estrutura de dados protegida pelas travas de exclusão A e B, então todo o código que invoca este método deve ter conhecimento sobre estas travas, para evitar executar concorrentemente com uma linha de execução que possa adquiri-las em ordem oposta [30]. Se uma nova versão da biblioteca também adquire a trava C, esta mudança pode surtir efeito sobre todo o programa que utiliza a biblioteca.

Um bloco **atomic** atinge o mesmo efeito de proteção aos dados de um programa escondendo no entanto o mecanismo usado. Um método de biblioteca pode acessar uma estrutura de dados de forma segura em uma transação, sem se preocupar a respeito de como as propriedades transacionais serão mantidas. O código que usa a biblioteca não precisa saber sobre a transação ou sua implementação, e tão pouco se preocupar sobre seu isolamento de outras linhas de execução concorrentes. O método da rotina desta forma é componível.

O bloco **atomic** não faz mágicas, no entanto. Ainda é possível escrever código incorreto [7]:

Linha de execução 1:

```
atomic {
  while (!flagA);
  flagB = true;
}
```

Linha de execução 2:

```
atomic {
  flagA = true;
  while (!flagB);
}
```

O código no bloco **atomic** está incorreto porque os laços não terminarão a não ser que a respectiva *flag* possua estado verdadeiro quando o bloco começa a executar. Este exemplo foi publicado para ilustrar a dificuldade envolvida na conversão manual de código com sincronização explícita para o uso de transações, mas também serve para ilustrar que blocos **atomic** por si só não servem para garantir que o programa termine ou produza resultados corretos.

Compor dois blocos **atomic**, como neste exemplo, não garante corretude ou mesmo que o programa termine. Outros mecanismos ou lógica são necessários para arrazoar sobre estas propriedades. Blocos **atomic** tornam este raciocínio mais fácil, uma vez que pré e pós-condições aplicam-se a toda a seqüência transacional de operações, mesmo quando os blocos executam concorrentemente.

Algumas linguagens podem também definir funções ou métodos transacionais, cujo corpo executa em uma expressão implicitamente transacional:

```
atomic void foo {
  if (x != null)
    x.foo();
  y = true;
}
```

⇔

```
void foo {
  atomic {
    if (x != null)
      x.foo();
    y = true;
  }
}
```

Da perspectiva de um programador, um bloco **atomic** pode ter três resultados possíveis. Se a transação completa, seus resultados tornam-se parte do estado do programa visível ao código executado fora do bloco **atomic**. Se a transação aborta, deixa o estado do programa não modificado. Se a transação não termina, seu resultado é indefinido.

Se permitirmos que blocos atômicos sejam aninhados, e se uma transação aninhada for abortada enquanto que a transação mais externa não está em conflito com nenhuma outra transação, então poderíamos diminuir o trabalho necessário para executar as transações, reiniciando apenas aquela mais interna:

```
atomic {
  // ...
  atomic {
    // ...
  }
  // ...
}
```

Uma construção do tipo **tryatomic** [2] poderia ser utilizada caso haja o interesse em tentar executar um bloco de código atomicamente, mas sem a necessidade de garantias de que a transação seja completada. Caso a transação seja abortada, passamos diretamente à próxima instrução após a transação sem tentar executá-la novamente:

```
tryatomic {
  // ...
}
```

Outras construções foram propostas para fornecer caminhos alternativos para a conclusão de uma transação. A construção **orElse**, proposta por Harris et al., [24] pode ser usada para definir um bloco de código transacional que é executado caso o bloco transa-

cional que o precede imediatamente não puder ser concluído:

```
atomic {
  // ...
} orElse {
  // ...
}
```

Tais blocos podem ser encadeados em um número arbitrário. Caso o último bloco da cadeia também falhar, e se esta construção não estiver aninhada dentro de outra construção **atomic ... orElse**, o sistema de memória transacional tenta executar a transação novamente a partir do primeiro bloco **atomic**. Caso contrário (isto é, se a construção estiver aninhada), o sistema transfere o controle para o próximo bloco **orElse** da transação mais externa, para tentar os caminhos alternativos no nível acima na hierarquia de transações.

Para a coordenação de transações, Harris et al. [24] introduziram o comando **retry**:

```
atomic {
  if (buffer.isEmpty())
    retry;
  Object x = buffer.getElement();
  ...
}
```

Uma transação que executa um comando **retry** aborta e então re-executa. **retry** é um mecanismo geral que permite que uma transação abandone sua computação corrente por uma razão arbitrária e re-execute na esperança de produzir um resultado diferente. Diferentemente da sinalização explícita, com variáveis de condição e operações **signal/wait**, ou sinalização implícita, com o comando de predicado **waituntil**, usado com monitores, **retry** não nomeia a transação com a qual deseja-se coordenar ou os recursos sob os quais coordenar. Harris et al. sugeriram atrasar a execução até que o sistema detecte mudanças em um ou mais valores que a transação tenha lido em sua execução anterior, de forma que seus resultados possam ser diferentes durante a próxima execução. Isto transforma **retry** em um mecanismo potencialmente eficiente para coordenar transações através de bloqueio condicional.

Se combinados, a construção **atomic ... orElse** e o comando **retry** possibilitam que o bloqueio condicional seja componível. Por exemplo, se o método **remove** de uma lista bloqueia enquanto não houver elemento disponível, então é possível que o programador escreva código que remova um elemento de qualquer lista de um determinado conjunto,

de forma semelhante ao comando **select** de sistemas Unix [24]:

```
class List {
  int elms[];
  int numElms;
  ...
  atomic int remove(){
    if(numElms == 0)
      retry;
    return elms[--numElems];
  }
}

int someMethod(List l1, List l2){
  atomic {
    return l1.remove();
  } orElse {
    return l2.remove();
  }
}
```

Neste exemplo, se o bloco **atomic** não for completado por causa da execução do comando **retry** no método **remove**, então o bloco **orElse** de **someMethod** terá uma chance de executar e remover um elemento da segunda lista, caso esta não esteja vazia. Notamos que o método **someMethod** por si só também é facilmente componível, ou seja, é possível utilizar novamente a construção **atomic ... orElse** que em um bloco da construção utilize **someMethod** para tentar remover um elemento das listas **l1** ou **l2**, e em outros blocos tente realizar operações em outras estruturas de dados quaisquer, como caminhos alternativos:

```
int yetAnotherMethod(){
  List l1, l2, l3, l4, l5;
  ...
  atomic {
    return someMethod(l1, l2);
  } orElse {
    return l3.remove();
  } orElse {
    return someMethod(l4, l5);
  }
}
```

Notamos que o comando **select** de sistemas Unix no entanto não compõe com a mesma facilidade que a construção **atomic ... orElse**. Programadores Unix precisam aprender técnicas de programação desajeitadas, agrupando todos os descritores de arquivos pelos quais esperar, invocando um comando **select** de alto nível, e despachando os resultados para o cliente apropriado [24].

O sistema de tempo de execução de memórias transacionais, ou o programa, podem abortar uma transação. O primeiro caso é um mecanismo de implementação invocado caso o acesso de uma transação a um recurso entra em conflito com os acessos de outra transação ou se a transação fica bloqueada por muito tempo esperando por um recurso. Em ambos os casos, o sistema aborta e re-executa a transação, na esperança de que o problema não se repita. Em geral, este processo de reexecução não é visível ao programa, exceto talvez pela manifestação de baixo desempenho.

Estes abortos diferem de um induzido pelo programa — os quais podem ocorrer explicitamente através do uso de um comando de aborto ou exceção. Quando o sistema aborta uma transação, ele a re-executa para que tenha outra oportunidade de completar. Quando o programa aborta a transação, o fluxo de controle passa para o comando após o bloco **atomic** ou para o código responsável por lidar com a exceção.

Transações não são uma panacéia. Ainda é (muito) fácil escrever programas incorretos, mesmo com memórias transacionais. Por exemplo, Flanagan e Qadeer desenvolveram a análise de atomicidade, que encontra os locais onde um programa libera uma trava de exclusão (ou termina uma transação) muito cedo [18]. Consideremos um exemplo do problema:

```
class StringBuffer ... {

  private int count;

  ...

  public StringBuffer append(StringBuffer sb) {
    int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length)
      expandCapacity(newcount);
    sb.getChars(0, len, value, count);
    count = newcount;
    return this;
  }

  public atomic int length() { return count; }

  public atomic void getChars(...) { ... }

}
```

O método **append** não possui uma transação. Apesar de as operações em **length** e **getChars** serem transacionais, o método **append** também deveria executar em uma transação para garantir que o tamanho do *buffer* não mude entre a chamada a **length** e o momento no qual os caracteres são copiados para o novo *buffer*.

Qual é o resultado de uma transação? Em propostas de implementação de linguagens feitas até o momento, uma transação é um comando, portanto ou modifica o estado do programa ou modifica o fluxo de controle. De forma geral, o estado de um programa inclui não somente variáveis e estruturas de dados no processo executando a transação, mas também a comunicação com entidades fora do processo. O último caso inclui ações como a criação e modificação de arquivos em disco, comunicação entre processos através de *pipes*, ou mesmo comunicação via TCP/IP — que afetam o ambiente onde a computação executa. Muitos sistemas de memória transacional evitam esta complexidade, proibindo operações de entrada e saída em transações.

Uma abstração de programação com semânticas simples e claras aumenta as chances de escrevermos código correto e auxilia-nos na detecção de erros com a ajuda de ferramentas de desenvolvimento. As semânticas de memórias transacionais ainda não foram

formalmente especificadas, apesar de algumas tentativas terem sido feitas. A maioria dos artigos assumem o critério de corretude usado em bancos de dados (serialidade) ou em estruturas de dados concorrentes (linearidade). Ambos os critérios especificam alguns aspectos de um bloco **atomic**, mas nenhum especifica as semânticas de transações aninhadas, construções de linguagem como **retry** ou **orElse**, ou a interação entre o código dentro de blocos **atomic** e o código não transacional fora destes blocos.

2.3 Espaço de Projeto

Várias propostas de implementação de memórias transacionais têm sido feitas nos últimos anos. As propostas divergem em muitos aspectos, e ainda não se estabeleceu um consenso sobre quais são as melhores abordagens. A seguir discorremos sobre algumas das escolhas a serem feitas ao se projetarem um sistema de memórias transacionais.

2.3.1 Implementação em *hardware* ou em *software*

Um projeto de implementação de TM pode ser baseado apenas no uso de bibliotecas de *software* e convenções de programação, ou receber suporte especial em *hardware*. TMs baseadas apenas em *software* possuem a vantagem de funcionarem em *hardware* legado, além de serem bastante flexíveis quanto aos demais detalhes de implementação, como políticas de controle de contenção, tamanho de transação (em número de acessos à memória) ou tempo de execução de uma transação. Já propostas de TM com suporte direto em *hardware* têm a vantagem de diminuirem o *overhead* das transações (se comparado à execução de uma única linha de execução, sem custos de sincronização).

2.3.2 Detecção e gerenciamento de conflitos

Transações lêem e escrevem objetos compartilhados. Duas transações conflitam entre si se acessam o mesmo mesmo objeto e pelo menos um dos acessos é uma escrita. Para que uma transação possa escrever em um objeto, primeiramente precisa adquiri-lo. A aquisição de um objeto é o "gancho" que permite a detecção de conflitos: torna as transações que escrevem visíveis umas às outras, bem como às transações que lêem tais objetos.

Aquisições podem ocorrer a qualquer momento, a partir do acesso inicial aos objetos até a confirmação final da transação. Aquisições realizadas quando do primeiro acesso para escrita a um objeto são chamadas de aquisições imediatas. Aquisições feitas somente durante o processo de confirmação da transação são chamadas de tardias. Aquisições tardias permitem que implementações de TM especulem mais, e dão mais oportunidade para que transações conflitantes executem em paralelo. O paralelismo entre uma transação

que escreve em um objeto e um grupo de transações que lêem este mesmo objeto pode ser 100% aproveitável se a transação que escreve completar por último. O paralelismo entre transações que escrevem a um mesmo objeto possui natureza mais puramente especulativa: apenas uma destas transações pode ser concluída, no entanto não há uma forma geral de saber qual delas **deve** completar [33].

Uma vez que leitores e escritores se tornam visíveis, escolher as circunstâncias nas quais "roubar" um objeto (e desta forma abortar a transação que o adquiriu previamente) ou esperar até que este recurso seja liberado é um problema a ser resolvido pelo sistema de gerenciamento de contenção. O gerenciamento de contenção não afeta a corretude da implementação de um sistema de memória transacional, apenas o seu desempenho. Algumas propostas de sistemas de TM possuem um método fixo para gerenciamento de contenção, ao passo que outras propostas tratam o problema como um aspecto modular do sistema, podendo ser alterado para melhorar o desempenho de um programa sob determinada carga de trabalho.

2.3.3 Granularidade de detecção de conflitos

O conflito entre transações pode ser detectado com granularidade fina, no nível do tamanho de palavra da arquitetura do processador, ou com uma granularidade maior, como linhas de cache ou objetos inteiros em uma linguagem de programação orientada a objetos de alto nível. Os *trade-offs* na escolha da granularidade de detecção de conflitos são parecidos com aqueles que levamos em consideração ao especificar o tamanho de linhas de cache em um sistema multiprocessado. Quanto menor a granularidade, menor a ocorrência de falsos conflitos e conseqüentemente maior a exploração de paralelismo. Quanto maior a granularidade, no entanto, torna-se mais fácil amortizar os custos de operações transacionais, assim como é mais fácil amortizar os custos de buscar uma linha de *cache* maior na memória, que pelo princípio de localidade conterá outros dados que serão acessados logo em seguida. Algumas abordagens tentam utilizar granularidades de detecção de conflitos diferentes com base no padrão de acessos aos objetos compartilhados, por exemplo, usando granularidade do tamanho de palavra da arquitetura para acessar vetores de tipos primitivos da linguagem de programação, e usar a granularidade do tamanho dos objetos da linguagem para os demais acessos [1].

2.3.4 Registros de escrita versus registros de recuperação

Os dados escritos especulativamente sob uma transação devem ser mantidos em algum lugar antes de se tornarem disponíveis através da conclusão da transação. Se os dados sendo escritos são mantidos nas mesmas posições que ocuparão na memória física quando

a transação for completada, então é necessário manter registros de recuperação (*undo-logs*) para recuperar os valores iniciais caso a transação venha a ser abortada. Isto em geral torna o processo de conclusão da transação bastante rápido, pois os valores a serem escritos transacionalmente já estão em suas posições corretas, porém torna o cancelamento de transações mais lento uma vez que os registros de recuperação precisam ser processados e os valores antigos das posições de memória precisam ser restaurados. Notamos que uma abordagem de implementação que utiliza registros de recuperação pressupõe que a aquisição de objetos para escrita realiza-se de forma imediata, ou seja, durante a primeira escrita ao objeto durante a transação.

Caso os dados escritos sob uma transação são mantidos em um registro de escrita (*write buffer*), tornaremos a confirmação da transação potencialmente mais lenta, dado que teremos de transferir os dados do registro para as posições da memória física correspondentes aos endereços dos objetos escritos. Outro problema a ser resolvido são as dependências do tipo RAW (*read-after-write*), ou seja, como detectar que uma leitura dentro de uma transação refere-se a um dado anteriormente escrito durante a mesma transação e que portanto deve ser procurado no registro de escrita.

2.3.5 Isolamento forte versus isolamento fraco

Resultados parciais de uma transação não devem ser percebidos por outras transações no programa. No entanto, uma implementação de TM pode permitir que resultados parciais de uma transação sejam visíveis pelo código que não esteja sendo executado sob nenhuma transação. Dizemos que tais implementações possuem isolamento *fraco*. A noção provavelmente mais intuitiva de isolamento, chamada de isolamento *forte*, define que um dado escrito sob uma transação que não tiver concluído não estará disponível a *nenhuma* outra porção de código, esteja ou não executando sob uma transação.

O isolamento forte pode ser difícil de ser implementado de forma eficiente, especialmente no caso de memórias transacionais sem suporte especial em *hardware*. Um sistema de TM precisa especificar e implementar consistentemente um dos tipos de isolamento, sob pena de "quebrar" o código de usuários que esperam um tipo de isolamento mas encontram outro. Tal problema foi primeiramente apontado por Blundell et al. [7], que demonstraram que transações não podem substituir de forma genérica seções críticas guardadas por travas de exclusão, e que programas escritos tendo em mente modelos de isolamento fraco podem não funcionar se o sistema implementar o modelo de isolamento forte. Notamos também que Blundell et al. em seu trabalho confundem os termos *isolamento* e *atomicidade*, utilizando erroneamente os termos *atomicidade forte* e *atomicidade fraca*.

2.3.6 Transações intermitentes ou contínuas

Quando transações são contínuas, consideramos cada acesso a memória que não esteja sendo realizado sob uma transação como sendo por si só uma transação curta que nunca é abortada. Um dos efeitos desta propriedade é que código não transacional é capaz de abortar transações conflitantes. Por outro lado, quando transações são intermitentes, apenas acessos à memória efetuados dentro de transações podem ser considerados como transacionais e são portanto capazes de conflitar com os acessos feitos por outras transações [9]. Transações contínuas são difíceis de implementar eficientemente em um sistema de TM sem suporte especial em *hardware*, uma vez que para tanto o sistema precisaria instrumentar cada acesso à memória feito fora de uma transação. Algumas propostas de implementação em *hardware* provêem suporte a transações contínuas, como TCC (vide subseção 3.1.2), UTM (subseção 3.1.3) e VTM (subseção 3.1.4).

Capítulo 3

Plataforma de prototipagem de TM com suporte específico em *hardware*

Neste capítulo apresentamos o trabalho desenvolvido para a criação de uma plataforma de prototipagem e simulação de sistemas de memória transacional com suporte específico em *hardware*. Primeiramente, fazemos uma análise dos trabalhos relacionados, com destaque à proposta inicial de implementação de TM, feita por Herlihy e Moss em 1993. Em seguida apresentamos a plataforma de prototipagem desenvolvida, bem como dois estudos de caso. O primeiro estudo de caso trata da implementação de um protótipo baseado na proposta de Herlihy e Moss e o segundo da extensão desta proposta para dar suporte a transações aninhadas.

Todos os experimentos apresentados neste capítulo foram conduzidos em uma máquina Linux padrão (*kernel* 2.6), equipada com dois processadores Intel Core 2 Quad a 2 GHz, e 4 GB de memória principal. O compilador usado foi compilador C/C++ versão 4.0 do GCC, e utilizamos o SystemC versão 2.1 [26].

3.1 Trabalhos relacionados

3.1.1 Proposta inicial de Herlihy e Moss para a implementação de TM

Para dar suporte a transações, seis novas instruções primitivas são adicionadas à arquitetura de instruções do processador (ISA) [25]. As instruções transacionais `LT`, `LTX` e `ST` são variantes das instruções de leitura (*load*) e de escrita em memória (*store*). Elas definem o conjunto de posições de memória lido sob uma transação (através de `LT`) e o conjunto de posições escrito (através de `LTX` e `ST`). Três outras instruções (`COMMIT`, `VALIDATE` e `ABORT`) lidam com o estado da transação. `COMMIT` torna o conjunto de dados escritos visível às

outras transações executando no sistema. Se um conflito é detectado, as escritas feitas pela transação são descartadas e um código de erro é retornado em um registrador (especificado ao invocar a instrução). `VALIDATE` verifica se o conjunto de dados lidos e escritos pela transação é consistente, e `ABORT` descarta todas as atualizações feitas pela transação. Não existe nenhuma instrução específica para delimitar o início ou o fim de uma transação. Uma transação é implicitamente iniciada quando um acesso transacional é feito à memória (através de `ST`, `LT` ou `LTX`). Uma vez iniciada, continua em execução até que uma instrução `COMMIT`, `ABORT` ou `VALIDATE` é executada (esta última no entanto só termina a transação caso retorne um código de falha).

O processador também requer dois marcadores adicionais: `TACTIVE`, indicando que uma transação está em execução, e `TSTATUS`, indicando se a transação em execução é válida ou se foi abortada. Herlihy e Moss [25] apresentaram vários aspectos referentes à implementação das propriedades transacionais (ACI). As técnicas seguintes foram sugeridas:

- **Versionamento de dados:** Escritas especulativas são realizadas através de uma pequena *cache* transacional. Uma cache normal armazena dados não transacionais. A qualquer momento, determinada posição de memória pode ser representada por uma entrada em somente uma das caches do processador.

- **Detecção/resolução de conflitos:** O protocolo de coerência de cache é modificado para dar suporte às operações transacionais. Rótulos especiais são adicionados a cada linha na cache transacional, e ciclos de barramentos adicionais são definidos. Algumas abordagens diferentes de resolução de conflitos podem ser utilizadas. O método proposto aborta qualquer transação que tentar revogar a permissão de acesso a um dado já concedida a outra transação.

- **Atomicidade e isolamento:** lógica adicional é introduzida para facilitar o processo de confirmação ou aborto de uma transação. Operações transacionais mantém duas entradas na cache transacional: uma com o rótulo `XCOMMIT`, guardando o valor antigo da linha, e outra marcada com o rótulo `XABORT`, armazenando o valor escrito especulativamente. Quando uma transação é confirmada, as entradas marcadas como `XABORT` são rotuladas como `NORMAL`, enquanto que as entradas marcadas como `XCOMMIT` recebem o rótulo `EMPTY`. No caso de um aborto, linhas marcadas com `XABORT` recebem o rótulo `EMPTY` e linhas marcadas com `XCOMMIT` são rotuladas com `NORMAL`. Uma vez que estas alterações são realizadas localmente na cache e em paralelo, atomicidade e isolamento são garantidos.

Esta proposta não prevê o aninhamento de transações nem tão pouco construções mais elaboradas como **orElse** ou **retry**. O programador é responsável por periodicamente va-

lidar o estado da transação e reiniciá-la quando detectar que foi abortada. Por armazenar as escritas especulativas de uma transação em uma cache separada, e por não virtualizar este recurso, corre-se o risco de que os recursos da plataforma sejam rapidamente exauridos, caso no qual a transação é automaticamente abortada. Além disto, durante uma troca de contexto de uma linha de execução, a transação também é abortada. Desta forma, o programador é encorajado a utilizar somente transações de curta duração e que toquem poucas posições de memória.

3.1.2 TCC (*Transactional memory Coherence and Consistency*)

TCC [22] foi proposta como uma alternativa ao uso do protocolo de coerência de sistemas de memória compartilhada. Em vez de utilizar o protocolo para manter as linhas de cache coerentes a cada acesso à memória, TCC armazena temporariamente todas as escritas de uma transação, arbitra a permissão para a realização destas escritas, e ao final da transação as transmite atomicamente para todo o sistema. Outros processadores que possuem cópias das mesmas linhas de cache devem invalidar suas cópias bem como as transações correntes. Todas as operações de acesso à memória ocorrem dentro de transações.

A proposta possui a seu favor o fato de permitir a implementação de isolamento forte e transações contínuas, o que por alguns é visto como a forma mais natural de lidar com a programação com memórias transacionais. A proposta realiza também a serialização da confirmação das transações, o que permite a paralelização especulativa de um programa seqüencial, como por exemplo a paralelização de um laço de repetição cuja independência do fluxo de dados entre uma iteração e outra não pode ser provada estaticamente. Por outro lado, a TCC propõe mudar de forma significativa o modelo de memória dos sistemas de computação atuais, o que é uma grande barreira para a sua adoção. Além disso, a serialização da confirmação das transações pode ser um gargalo em sistemas com grande número de processadores. Notamos no entanto que modificações foram feitas à proposta inicial, que permitem certo grau de paralelismo na confirmação de transações [11].

3.1.3 UTM (*Unbounded Transactional Memory*)

UTM [3] tenta resolver os problemas relacionados ao limite de recursos associados a memórias transacionais com suporte em *hardware*, permitindo que transações tenham espaço de trabalho igual ao tamanho da memória virtual, e que transações sobrevivam a trocas de contexto sem abortarem. Valores escritos transacionalmente são armazenados *in loco* na memória, ao passo que valores antigos são armazenados em registros na memória virtual. Para prover desempenho aceitável, caches podem ser utilizadas para manter os valores novos enquanto que os valores originais são mantidos na memória principal. No

entanto, UTM requer mudanças significativas nos processadores e hierarquia de memória. Os autores propuseram LTM (Large Tansactional Memory) como uma alternativa menos custosa que limita o tamanho da transação ao tamanho da memória física. Como em UTM, a cache mantém os valores mais novos enquanto o valor original é mantido na memória física. Quando o estado da transação excede o tamanho da cache, os valores excedentes transbordam para uma tabela de dispersão mantida na memória principal.

3.1.4 VTM (*Virtual Transactional Memory*)

VTM [38] foi proposta como um esquema que combina *hardware* e *software* com o objetivo de resolver as questões de limite de recursos de sistemas HTM. VTM estende uma proposta de HTM com um registro de leituras e escritas na memória principal, mantido via *software*, e *hardware* específico para verificar e lidar com as situações nas quais os recursos de *hardware* não são suficientes para manter o estado da transação. Quando tais recursos são exauridos, novos valores lidos ou escritos transacionalmente são armazenados no registro. VTM envia os dados que o *hardware* não comporta a este registro e possivelmente o examina seqüencialmente durante acessos subseqüentes à memória para verificar conflitos. VTM utiliza uma tabela de filtragem de 10 MB para detectar se a análise seqüencial do registro é necessária para a detecção de conflitos com outras transações. VTM otimiza os casos onde o *hardware* comporta a transação sem recorrer ao registro em memória, checando um contador e evitando os acessos à tabela de filtragem.

3.1.5 LogTM

LogTM [37] consiste em um esquema de memória que armazena valores transacionais *in loco* enquanto que os valores originais são postos em registros mantidos na memória privada de cada linha de execução. Ao modificar posições de memória *in loco*, LogTM otimiza a confirmação das transações, porque neste caso simplesmente descarta os registro de valores originais. O aborto de uma transação requer o processamento dos registros para restaurar os valores originais das posições de memória escritas especulativamente. Os recursos de *hardware* são naturalmente virtualizados porque LogTM permite que blocos de memória manipulados transacionalmente sejam substituídos na cache. LogTM foi idealizado sobre um protocolo de coherência de cache baseado em diretórios, adicionando bits a cada entrada do diretório para sinalizar quais blocos de memória foram lidos ou escritos transacionalmente e por quais processadores. Quando estes blocos são substituídos na cache antes da transação terminar, continuam pertencendo ao processador que os requisitou mantendo-se os bits transacionais ligados no diretório, de modo que conflitos sejam detectados quando outro processador tentar acessar os mesmos blocos de memória.

3.1.6 *Hybrid Transactional Memory*

Hybrid Transactional Memory [28] consiste em um esquema no qual transações podem ser executadas em modo de *hardware* ou *software*. Primeiramente, o sistema tenta executar a transação no modo de *hardware*, utilizando o esquema de detecção de conflitos baseado no protocolo de coerência de cache. Caso os recursos de *hardware* transacional sejam exauridos, a transação pode ser re-executada em modo de *software*. O sistema garante que conflitos entre transações executadas em diferentes modos são detectados e tratados. Enquanto que o modo de *software* incorre em um alto custo de trabalho adicional, em geral espera-se que seja utilizado apenas para garantir que transações grandes demais para serem executadas apenas com suporte de *hardware* tenham a chance de completar.

3.1.7 *Bulk* HTM

Esta proposta de implementação utiliza assinaturas de *hardware* para identificar as posições de memória lidas e escritas por uma transação [10]. Para confirmar uma transação, o processador adquire uma permissão global de confirmação e envia aos outros processadores suas assinaturas de leituras e escritas. Estas assinaturas são versões compactadas dos endereços de memória acessados, formando um superconjunto destes. Desta forma, falsos conflitos podem acontecer. A implementação não requer alterações às caches ou ao protocolo de coerência para dar suporte aos registros de escrita especulativas ou à detecção de conflitos tardia. Como em TCC (subseção 3.1.2), ao arbitrar pela permissão global de confirmação *Bulk* STM serializa as transações, o que pode ser um gargalo de desempenho. Ao mesmo tempo esta serialização possibilita técnicas de paralelização especulativa de código serial.

3.1.8 Processador UltraSPARC T2 “Rock”

O primeiro processador com suporte em *hardware* a memórias transacionais deverá estar disponível no mercado muito em breve [41]. Os projetistas foram bastante cautelosos ao adicionar tal suporte. Em vez de idealizarem um sistema que virtualize todos os recursos transacionais do processador e tente garantir que qualquer transação seja em algum momento concluída de forma bem sucedida, os projetistas buscaram um sistema mais simples que aborta transações nas situações que considera “difíceis”. Estas situações ocorrem em eventos como exceções ou interrupções assíncronas, execução de instruções não permitidas dentro de transações, ou a exaustão dos recursos do hardware que dá suporte às transações, entre outras [36]. Duas novas instruções são adicionadas ao conjunto de instruções do processador. A primeira serve para começar a transação, salvando o contexto da linha de execução e especificando o contador de programa (PC) onde fica o código a ser

executado em caso de aborto. A segunda instrução serve para confirmar a transação corrente. Aspectos mais complexos da implementação de sistemas de memória transacional (como gerenciamento de contenção) são deixados a cargo do *software*, e transações que não podem executar em modo de *hardware* podem ser executadas em modo de *software* como em uma abordagem híbrida.

3.1.9 Plataformas de simulação e prototipagem

A avaliação de sistemas de TM com suporte em *hardware* geralmente faz uso de plataformas de simulação (*execution-driven*). Plataformas de simulação dirigidas por execução compilam o código do programa a ser caracterizado para a arquitetura onde a simulação será executada, instrumentando o código para coletar informações relevantes à simulação, como por exemplo o tempo que cada instrução ou bloco básico levaria para executar na arquitetura que se deseja simular. Estas plataformas são adaptadas para atingir os requisitos de uma proposta de implementação em particular. A exata quantidade de trabalho envolvida no desenvolvimento destas plataformas é difícil de mensurar, uma vez que os artigos da área não focam na descrição dos detalhes sobre o ambiente de simulação que lhes serve de base. Algumas das plataformas mencionadas incluem Proteus [8], UVSIM [42] e GEMS [34].

Proteus foi utilizado para avaliar a primeira proposta de TM feita por Herlihy e Moss. Este simulador requer que programas sejam escritos em uma versão de C estendida e não captura os efeitos de caches locais de instruções ou dados. GEMS por sua vez utiliza Simics [31], um simulador de sistemas completos de uso comercial. Um módulo do GEMS, chamado Opal, captura informações de tempo baseando-se em uma estratégia de simulação *execution-driven* que coleta informações de temporização. Uma vez que Opal é fortemente amarrado à arquitetura de processador usada, portar a plataforma de simulação para outras arquiteturas requer bastante experiência e trabalho.

A plataforma de prototipagem desenvolvida neste trabalho estende a plataforma ARP (*ArchC Reference Platform*) [13]. ARP foi concebida para fornecer um ambiente para o desenvolvimento de protótipos de sistemas utilizando simuladores funcionais de processadores ArchC [5, 6], permitindo que mesmo usuários pouco habituados à modelagem de sistemas sejam capazes de entender e utilizar rápida e eficientemente os protocolos de comunicação do ArchC e SystemC [26], bem como metodologias relacionadas ao projeto de plataformas.

Um modelo de prototipagem da plataforma ARP especifica um conjunto de componentes e relações entre eles. A figura 3.1 mostra um diagrama de blocos com os elementos principais capturados pela plataforma. Setas indicam o fluxo de comunicação entre os módulos mestres e seus escravos. Por exemplo, o componente *processador* atua como um

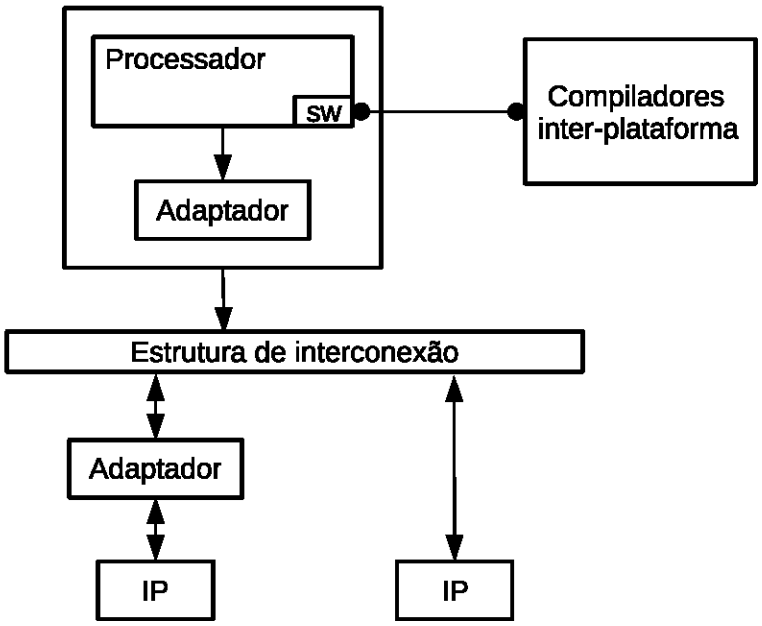


Figura 3.1: Plataforma de desenvolvimento.

mestre quando requisitando dados de um adaptador.

Os componentes do modelo são divididos entre *processadores*, *adaptadores*, *estruturas de interconexão* e outros componentes de propriedade intelectual (*IPs*). Para construir uma plataforma de prototipagem específica, é preciso definir quais componentes são necessários e como a interface entre eles é definida. Se um componente não está prontamente disponível, ele deve ser implementado seguindo os padrão TLM (*Transaction Level Modeling*) de SystemC. Adaptadores podem ser requeridos para a implementação de interfaces entre módulos que se comunicam usando protocolos diferentes. A natureza modular de SystemC também promove o reuso de componentes de uma plataforma em outra.

3.2 A plataforma de prototipagem desenvolvida

A plataforma de prototipagem reaproveita o método de integração definido pela ARP para organizar os modelos de componentes de forma que a compilação e execução da especificação do protótipo sejam automatizadas através de *Makefiles* e *scripts*. Cada modelo de plataforma deve especificar um arquivo de definição com informações sobre sua estrutura básica. A figura 3.2 mostra o arquivo de definições do protótipo de HTM que será discutido na seção 3.3. Neste exemplo, utilizamos um modelo de processador PowerPC (linha 1), um IP adicional para implementar as caches transacional e normal, (linha 2), um barramento para ligar as várias instâncias de processadores (linha 4) e

```
1. PROCESSOR := powerpc
2. IP        := transactional_cache
3. SW        := producer_consumer
4. IS        := simple_bus_ua
5. ADAPTOR   := simple_bus_2_cache_adapter \
6.              cache_2_processor_adapter
```

Figura 3.2: Um arquivo de definição.

alguns adaptadores (linhas 5 e 6). O aplicativo embarcado a ser executado na simulação também é especificado (linha 3).

O arquivo de definições especifica os componentes da plataforma mas não provê informações sobre como eles são conectados. A instanciação e amarração dos componentes é feita através de um arquivo escrito em SystemC. Para compilar a plataforma, um *script* faz a varredura do arquivo de configuração e gera os arquivos objeto para cada um dos componentes especificados. Um simulador é gerado, que carrega os arquivos executáveis dos programas embarcados e inicia a simulação. Ao iniciar a simulação do protótipo, dados como o número de processadores, tamanho e número de linhas das caches, também são interpretados e configurados automaticamente.

Uma característica importante da plataforma de prototipagem está relacionada à escolha da linguagem de descrição de arquiteturas (ADL) ArchC [5, 6] para a definição da arquitetura dos processadores do modelo de plataforma simulada. Escolhemos ArchC principalmente por permitir que arquiteturas de processadores sejam modeladas em um nível de abstração bastante alto, além de prover um conjunto de ferramentas capazes de gerar automaticamente simuladores funcionais e programas de manipulação de arquivos binários (montadores, *linkeditores*) para a arquitetura alvo. Os simuladores gerados são compatíveis com o padrão SystemC TLM e podem ser facilmente integrados com outros componentes do sistema. O modelo de simulação é voltado à simulação das instruções da arquitetura alvo (*instruction driven*), o que provê maior flexibilidade na mudança e customização da arquitetura a ser utilizada pelo protótipo, ao contrário das abordagens *execution driven* que precisam estimar o tempo de execução de cada instrução na arquitetura sendo simulada.

Como exemplo da flexibilidade obtida ao empregarmos uma ADL, consideremos a descrição de uma nova instrução adicionada ao modelo da arquitetura PowerPC, mostrado na figura 3.3 (esta é uma das novas instruções adicionadas ao modelo de PowerPC empregado no estudo de caso discutido na seção 3.3). O comportamento da instrução `COMMIT` (linhas 1 a 7) é descrito em SystemC. Apenas duas linhas de código são requeridas: uma para a requisição de uma operação de confirmação da transação feita ao controlador de

```
1. void ac_behavior( commit ) {
2.
3.  response = MEM.readT(1, COMMIT);
4.  GPR.write(rt, response);
5.
6.  if (response == success) num_commit++;
7. }
```

Figura 3.3: Implementação da instrução `COMMIT` no modelo ArchC da arquitetura PowerPC.

cache (linha 3), e outra para guardar o resultado da requisição (sucesso ou falha) em um registrador de propósito geral (linha 4). Uma vez que o código é baseado em uma linguagem comum (SystemC/C++), torna-se fácil familiarizar-se com a implementação e realizar modificações. Por exemplo, podemos contar o número de transações confirmadas adicionando um contador (linha 6 da figura 3.3). Outras medidas úteis, como o número de abortos ou número de instruções transacionais executadas, podem ser obtidas com a mesma flexibilidade.

Por utilizarmos uma ADL torna-se também bastante fácil experimentar diferentes arquiteturas de processadores. Por exemplo, é possível explorar uma plataforma baseadas em processadores MIPS apenas alterando o processador no arquivo de definições mostrado na figura 3.2. Todos os modelos de processadores implementados em ArchC podem ser utilizados da mesma forma na plataforma de desenvolvimento.

Devemos notar no entanto que a plataforma de prototipagem não depende dos simuladores e linguagem ArchC. De fato, outros simuladores podem ser utilizados desde que possam se comunicar com o restante do sistema através do padrão SystemC TLM. Abordagens diferentes podem ser modeladas na plataforma de desenvolvimento ao integrarmos diferentes IPs, desde que as interfaces entre processadores, caches e memória principal sejam implementadas apropriadamente. Dada a maior interdependência entre processadores e hierarquia de memória, demandada pelos mecanismos necessários para dar suporte a diferentes propostas de HTM, possivelmente seja necessário criar novos adaptadores entre a cache e IPs, processadores e memória principal, no mesmo espírito dos adaptadores usados nas metodologias TLM e SLM (*System Level Design*).

A plataforma recebeu suporte para a realização de testes de regressão, o que facilita o teste das diferentes combinações de características que podem compor um protótipo. Por exemplo, um determinado protótipo de plataforma pode ser configurado com arquiteturas diferentes e quantidade de processadores diferentes. Caso realizarmos alguma alteração em algum dos componentes do protótipo, por exemplo na cache, é importante realizar

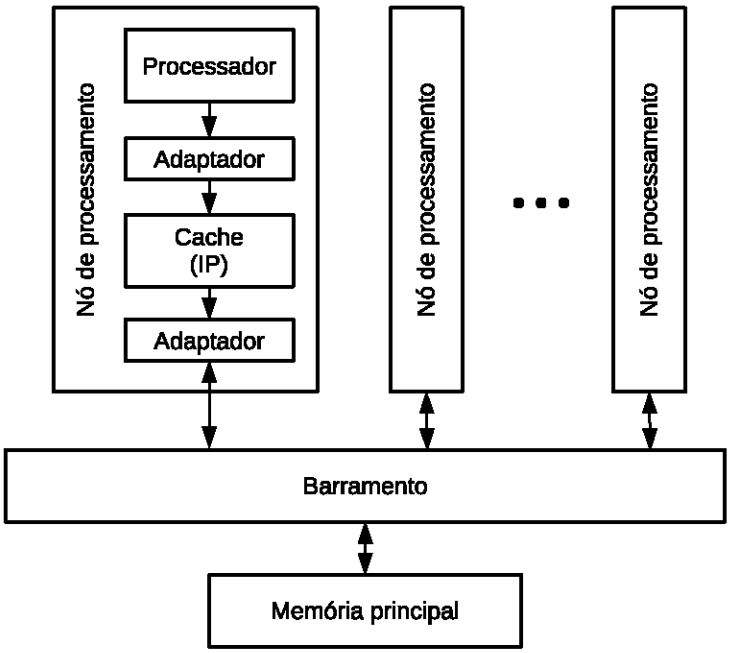


Figura 3.4: A plataforma TM.

testes que cubram boa parte das possíveis combinações de configurações. Se por exemplo um protótipo pode ser configurado com as arquiteturas MIPS, SPARC e PowerPC, e com uma quantidade de processadores igual as potências de 2 entre 2 e 32, teremos um total de 15 configurações a serem cobertas pelo teste de regressão. Estes testes podem ser customizados no *Makefile* da plataforma de prototipagem e no *Makefile* do *software* embarcado.

3.3 Primeiro estudo de caso: a proposta de Herlihy e Moss

Nesta seção descrevemos o protótipo de um sistema HTM baseado na proposta feita por Herlihy e Moss, descrita na seção 3.1.1. Adotamos a plataforma de prototipagem descrita na seção 3.2 para a implementação.

3.3.1 Construção da plataforma

Um sistema multiprocessado simétrico convencional (SMP) é empregado como a base do sistema. O meio de interconexão é um barramento localizado entre a cache privada de cada processador e o subsistema de memória principal (uma abordagem utilizando barramento compartilhado). A coerência de cache é realizada por um protocolo baseado em *snooping* e política de escritas do tipo *write-back*. Para simplificar a análise dos resultados, modelamos apenas caches de nível primário, sem perda de generalidade.

Cada vez que um dado não é encontrado na cache devemos realizar comunicação através do barramento. Esta comunicação é modelada no sistema com o custo de 50 ciclos de processador, de forma a representar a contenção do barramento. Modificamos o sistema base de acordo com a proposta inicial de Herlihy e Moss introduzida na seção 3.1.1. A figura 3.4 mostra o projeto da plataforma (seu arquivo de definição foi mostrado na figura 3.2). A seguir descrevemos os componentes principais da plataforma.

Processador. Adicionamos as novas instruções aos modelos funcionais das arquiteturas MIPS, SPARC e PowerPC. Este processo é bastante simples uma vez que os modelos são descritos em um nível de abstração bastante alto, como previamente mostrado no exemplo da figura 3.3. Também introduzimos novos tipos de mensagem TLM para permitir a comunicação adequada entre processador e o controlador da cache, uma vez que a responsabilidade da implementação do comportamento das novas instruções é compartilhada entre processador e cache. Os modelos de processadores emitem apenas uma instrução a cada ciclo e as instruções são executadas em sua ordem de emissão, sendo estas limitações atuais da linguagem ArchC.

Para facilitar o desenvolvimento do *software* embarcado, cada nova instrução foi mapeada a uma chamada de função C. A figura 3.5 ilustra a interface de programação para as instruções `LT` e `ST`, para a arquitetura PowerPC. Programadores desenvolvem código transacional empregando estas interfaces que por sua vez são traduzidas em sua forma correspondente de linguagem de montagem. Um exemplo do uso destas funções pode ser visto na figura 3.6.

Simuladores funcionais com suporte a TM são automaticamente gerados pelas ferramentas do ArchC. Para compilar o *software* embarcado a ser utilizado nas simulações, utilizamos compiladores GCC inter-plataforma (*cross-compilers*) que traduzem código C em código de montagem. O resultado é ligado para produzir o código binário a ser carregado na memória principal do protótipo quando a simulação está para começar.

Adaptadores. Dois adaptadores foram implementados: um entre cada processador e sua cache, e outro entre as caches e o barramento. Estes adaptadores foram projetados considerando-se a modularidade. Por exemplo, um processador pode ser facilmente

```
inline int LT(int address) {
  int return_val;
  asm volatile("lt %0,0(%1)" : \
        "=r" (return_val): "r" (address): "0");
  return return_val;
}

inline void ST(int address, int write_val) {
  asm volatile("st %0,0(%1)" : : \
        "r" (write_val), "r" (address): "0");
}
```

Figura 3.5: Interface de programação TM.

trocado, desde que um novo adaptador seja implementado para fazer interface com a cache.

Para reduzir o tempo de simulação, a arbitragem do barramento é realizada através de um objeto `sc_mutex`, colocado no adaptador entre as caches e o barramento. Um modelo de barramento com suporte direto aos protocolos de *snooping* e arbitragem teria levado a um desempenho de simulação inferior.

Caches e memória principal. A lógica de controle de cache e as caches normal e transacional foram encapsuladas dentro de um mesmo IP. Escolhemos manter ambas as caches no mesmo IP para facilitar a comunicação entre elas, evitando assim comunicação através do barramento quando uma requisita informações já carregadas pela outra. Este IP implementa duas interfaces, uma para a coerência de cache através de *snooping* e a outra para as requisições de memória feitas pelo processador. A cache normal é diretamente mapeada e a cache transacional é completamente associativa com 64 entradas, e ambas as caches possuem latência de acesso de um ciclo do processador. Vários parâmetros do IP podem ser configurados pelo projetista, tais como o número de linhas em cada cache e o tamanho das linhas.

3.3.2 Avaliação do protótipo

Para avaliar o protótipo de TM, utilizamos essencialmente os mesmos *microbenchmarks* discutidos no trabalho original de Herlihy e Moss. Todos os programas empregam um mecanismo de *back-off*, semelhante àquele empregado para reduzir a contenção por travas de exclusão mútua, para reduzir os números de abortos e aliviar a contenção do barramento. Os *benchmarks* seguintes foram implementados:

- **Contador compartilhado:** cada processador deve incrementar um único contador

compartilhado $2^{12}/n$ vezes (onde n é o número total de processadores). Uma vez que todos acessam a mesma variável, a contenção é bastante alta e não existe qualquer paralelismo inerente.

- **Produtor/Consumidor:** $n/2$ processadores produzem elementos enquanto os demais $n/2$ os consomem. Um total de 2^{12} operações são realizadas sobre um *buffer* FIFO inicialmente vazio

- **Lista duplamente encadeada:** metade dos processadores remove elementos do final de uma lista e os coloca em uma pilha de elementos livres, enquanto que a outra metade remove elementos da pilha e os coloca no início da lista. As duas operações (manipulação da lista e da pilha) são realizadas por transações diferentes. A simulação termina quando um total de 2^{12} operações é realizado.

A figura 3.6 mostra o código da operação de remoção da lista duplamente encadeada. Uma vez que a interface para as instruções transacionais é definida externamente, o mesmo código C pode ser utilizado para os protótipos utilizando simuladores funcionais tanto para MIPS, quanto para SPARC, quanto para PowerPC.

```
entry* list_deq(){
  entry *elm, *elm_next;
  unsigned backoff = BACKOFF_MIN;
  unsigned wait;

  while (1) {
    elm = (entry*)LTX(Head);
    if (VALIDATE() && elm != NULL) {
      elm_next = (entry*)LT(&elm->next);
      if (VALIDATE()) {
        if (elm_next == NULL) {
          ST(Tail, NULL);
        } else {
          ST(&elm_next->prev, NULL);
        }
        ST(Head, elm_next);
        if (COMMIT())
          break;
      }
    } else {
      ABORT();
    }
    wait = random() % (1 << backoff);
    while (wait--);
    if (backoff < BACKOFF_MAX)
      backoff++;
  }
  return elm;
}
```

Figura 3.6: Operação de remoção de elemento da lista.

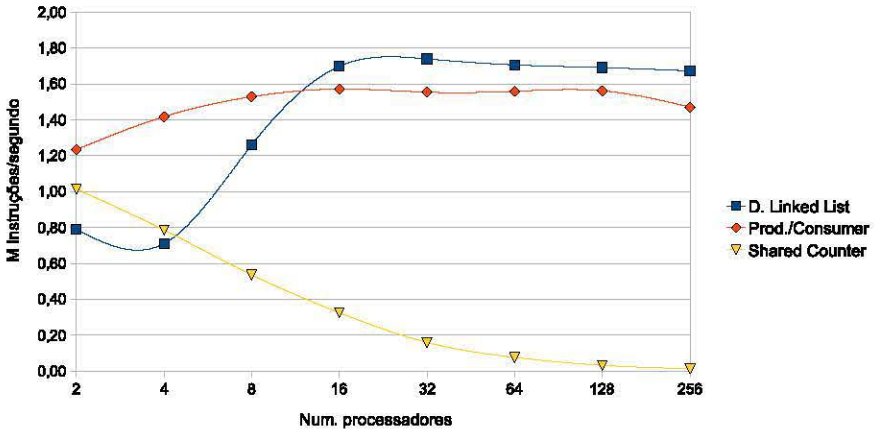


Figura 3.7: Velocidade de simulação (MIPS).

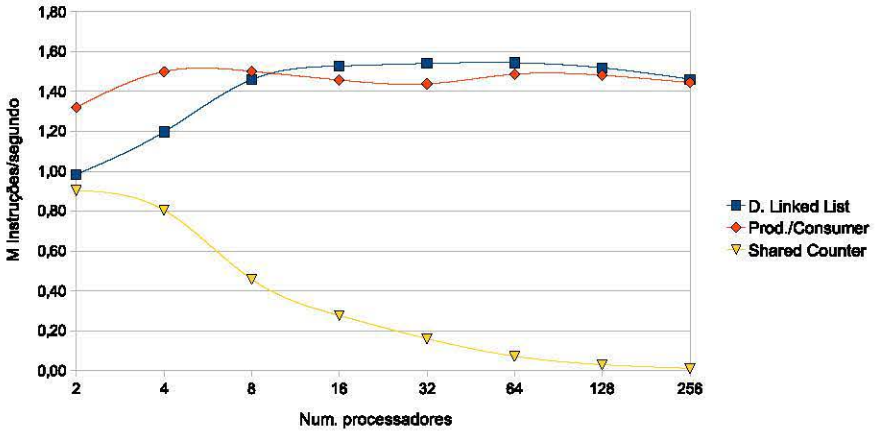


Figura 3.8: Velocidade de simulação (SPARC).

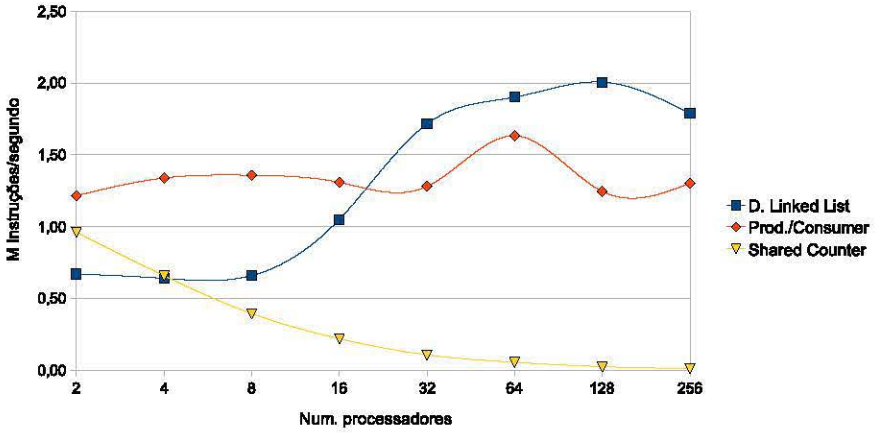


Figura 3.9: Velocidade de simulação (PowerPC).

Em todas as simulações fizemos variar o número de processadores simulados de 2 até 256. Cuidamos ao dimensionar o tamanho das caches de modo a evitar falhas por capacidade, e cada objeto compartilhado nos *benchmarks* foi cuidadosamente colocado em uma linha de cache distinta para evitar falsos conflitos.

Primeiramente medimos a velocidade de simulação dos *benchmarks* citados. As figuras 3.7, 3.8 e 3.9 mostram o número agregado de instruções por segundo para os protótipos com processadores MIPS, SPARC e PowerPC respectivamente. Podemos ver que, a partir de 16 processadores, a velocidade de simulação tende a estabilizar ou diminuir devido ao trabalho adicional causado pelas inicializações do SystemC ao início da simulação. Este fato é ainda mais evidente no caso do contador compartilhado, uma vez que este *benchmark* possui transações bastante curtas que não abortam com freqüência. Além disso, uma vez que este *benchmark* não exibe paralelismo, boa parte do tempo é desperdiçada devido a contenção no barramento.

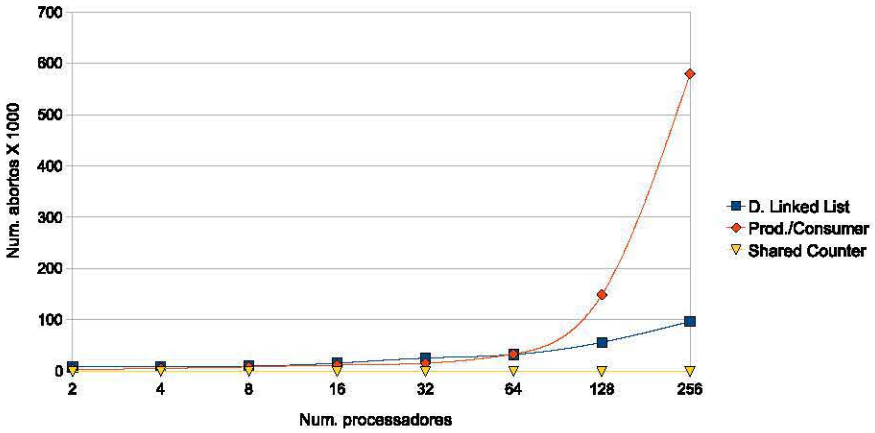


Figura 3.10: Número de abortos (MIPS).

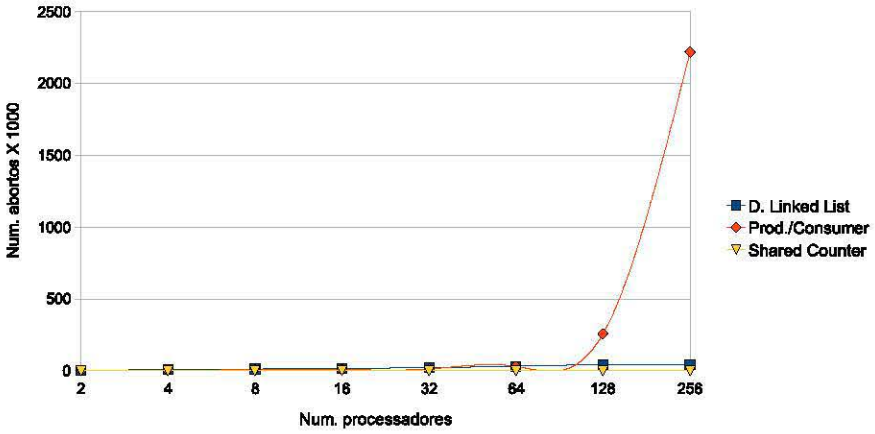


Figura 3.11: Número de abortos (SPARC).

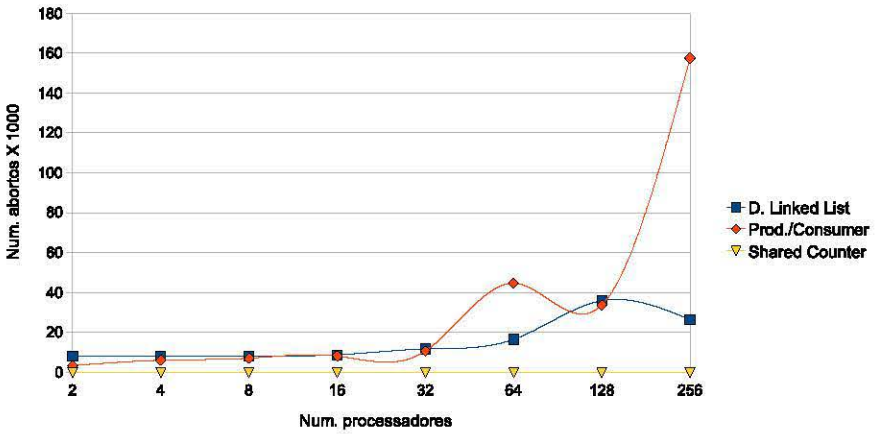


Figura 3.12: Número de abortos (PowerPC).

Também instrumentamos o simulador para contar o número de abortos de transação. As figuras 3.10 (MIPS), 3.11 (SPARC) e 3.12 (PowerPC) mostram os resultados obtidos. Em geral, o número de abortos aumenta significativamente com o aumento do número de processadores simulados, exceto pelo contador compartilhado, que possui transações bastante pequenas que executam várias vezes em seqüência em um mesmo processador antes que chegue algum evento do barramento requisitando a linha de cache onde o contador está localizado. O uso de *back-off* exponencial foi capaz de controlar o número de abortos apenas até certo ponto, sugerindo que talvez seja necessário o envolvimento de um escalonador que retire de execução linhas que não são capazes de executar devido à alta contenção por determinado recurso, escalonando estas linhas de execução novamente quando o recurso for liberado.

Já as figuras 3.13, 3.14 e 3.15 mostram o tempo do simulador tal qual retornado pela função **sc_simulation_time** da biblioteca SystemC. Podemos ver que o tempo de simulador sempre aumenta com o número de processadores simulados. Isto era de se esperar, uma vez que os *benchmarks* exibem pouco paralelismo e executam nenhum trabalho real além da comunicação de dados, servindo justamente para estressar o cenário de alta contenção, ponto fraco das abordagens tradicionais de sincronização.

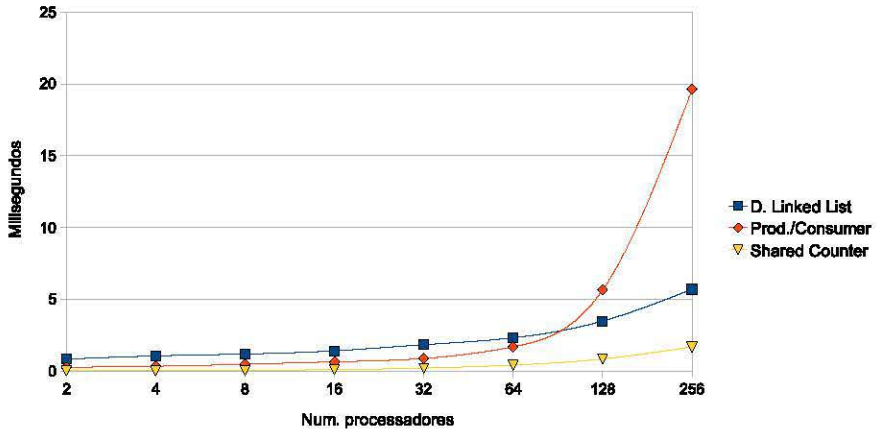


Figura 3.13: Tempo do simulador (MIPS).

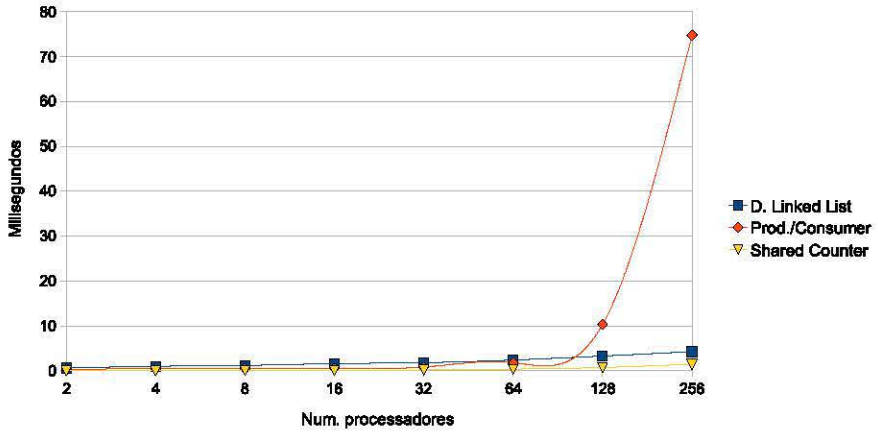


Figura 3.14: Tempo do simulador (SPARC).

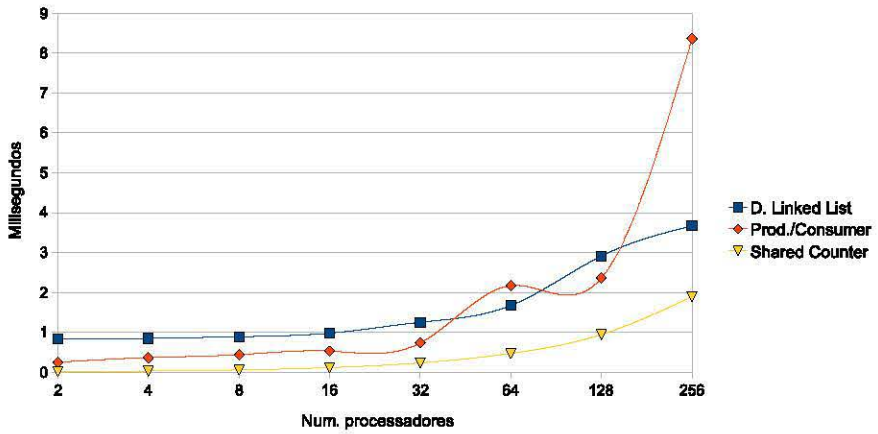


Figura 3.15: Tempo do simulador (PowerPC).

3.4 Segundo estudo de caso: aninhamento de transações

Nesta seção apresentamos outro protótipo desenvolvido usando a plataforma proposta, desta vez estendendo o protótipo descrito na seção 3.3 para dar suporte ao uso de transações aninhadas.

3.4.1 Construção da plataforma

Como na seção 3.3, o protótipo aqui descrito utiliza modelos de processadores ArchC para as arquiteturas MIPS, SPARC e PowerPC, tendo como base um sistema multiprocessado simétrico convencional (SMP). A memória e o barramento são os mesmos, e a coerência de cache também baseia-se no protocolo de *snooping*. Apenas as caches de nível primário são modeladas.

O modelo abstrato do protótipo assemelha-se em muito ao modelo descrito na seção 3.3, de modo que o diagrama de alto nível da figura 3.4 também se aplica a este estudo de caso. A seguir descrevemos os componentes principais deste protótipo.

Processador. Além das instruções definidas na proposta de Herlihy e Moss, adicionamos novas instruções aos modelos funcionais das arquiteturas de processador. A instrução `BEGIN_TRANS` é definida para determinar explicitamente o início de uma transação. A instrução `VALIDATE` não mais encerra a transação caso esta tiver abortado, mas apenas retorna o status da transação no nível de aninhamento corrente. Apenas as instruções `COMMIT` e `ABORT` são capazes te encerrar uma transação. Cada ocorrência de uma instrução `BEGIN_TRANS` inicia um novo nível de aninhamento de transações, e deve ser casada com uma instrução `COMMIT` ou `ABORT` para confirmar ou abortar a transação no nível de aninhamento corrente. Quando uma transação aninhada é confirmada, os efeitos de suas atualizações à memória são visíveis às outras transações somente quando a transação de nível mais alto daquela linha de execução é confirmada. Caso o número de níveis de aninhamento suportado pelo *hardware* for totalmente utilizado, novas transações aninhadas executam no contexto da transação que as deu origem, sem aborto e reinício independentes (uma abordagem chamada de *flattening*).

A instrução `GET_NEST` retorna o nível de aninhamento de transações corrente. A instrução `ABORT_N_N_D` (*Abort-no-nesting-decrease*) é usada para forçar o aborto de uma transação sem no entanto concluí-la. Esta instrução é necessária para a implementação de bibliotecas de *software* que precisam abortar a transação do código cliente e assim evitar possíveis bloqueios mútuos (*deadlocks*). Seu uso fica mais claro no exemplo da figura 3.16, que representa a inserção de um elemento em uma lista duplamente enca-

```
1.  void list_enq(entry *new){
2.    entry *old_tail;
3.    unsigned backoff = BACKOFF_MIN;
4.    unsigned wait;
5.    unsigned retries = 0;
6.
7.    while (1) {
8.
9.      BEGIN_TRANS();
10.
11.     ST(&new->next, NULL);
12.     ST(&new->prev, NULL);
13.
14.     old_tail = (entry*)LTX(Tail);
15.     if (VALIDATE()){
16.       ST(&new->prev, old_tail);
17.       if (old_tail == NULL){
18.         ST(Head, new);
19.       } else {
20.         ST(&old_tail->next, new);
21.       }
22.       ST(Tail, new);
23.       if (COMMIT())
24.         return;
25.     }else{
26.       ABORT();
27.     }
28.
29.     if( ! VALIDATE()){
30.       return;
31.     }
32.
33.     retries++;
34.
35.     if(retries > NUM_MAX_RETRIES && GET_NESTING_LEVEL() > 0){
36.       ABORT_NO_NESTING_DECREASE();
37.       return;
38.     }
39.
40.     wait = random() % (01 << backoff);
41.     while (wait--);
42.     if (backoff < BACKOFF_MAX)
43.       backoff++;
44.   }
45. }
```

Figura 3.16: Operação de remoção de elemento da lista, usando aninhamento de transações.

deada. Suponhamos que duas linhas de execução (L1 e L2) operem sobre duas listas encadeadas (A e B) com apenas um elemento em cada uma. L1 remove o elemento *a* de A em uma transação aninhada e a confirma. L2 remove o elemento *b* da lista B também em uma transação aninhada e a confirma. Notemos que as transações de nível mais alto das duas linhas de execução ainda não foram concluídas, e portanto *a* e *b* ainda estão sob posse de L1 e L2, respectivamente. Agora L1 tenta inserir *a* em B, em outra transação aninhada, mas não consegue porque o ponteiro que aponta para o começo de B está sob posse de L2. Da mesma forma, L2 tenta inserir *b* em A, mas não consegue dar prosseguimento à transação porque o ponteiro que aponta para o início de A pertence a L1. Se nada for feito, ambas L1 e L2 podem abortar suas transações de inserção por tempo indeterminado, e assim nunca concluir de forma bem sucedida. Na linha 35 do exemplo, testamos se um número pré-definido de tentativas da transação aninhada foi executado, e se estivermos executando dentro de uma transação a abortamos sem reduzir o nível de aninhamento corrente (linha 36), uma vez que o código cliente é responsável por casar a instrução `BEGIN_TRANS` por ele executada com uma instrução `ABORT`. Desta forma uma linha de execução dentre L1 e L2 aborta sua transação de mais alto nível e permite a outra linha concluir sua transação aninhada.

Caches. Como no protótipo descrito na seção 3.3, ambas as caches transacional e normal foram implementadas em um mesmo IP. A associatividade e o tamanho das linhas da cache normal podem ser configuradas pelo desenvolvedor do protótipo. A cache transacional por sua vez é completamente associativa e possui 64 entradas.

Para dar suporte a transações aninhadas, optamos por um esquema bastante simples que adiciona espaço a cada entrada na cache transacional para armazenar blocos de memória lidos ou escritos transacionalmente em um determinado nível de aninhamento. A cache transacional funciona como uma pequena matriz, onde cada linha representa um vetor com as diferentes versões de um bloco de memória lido ou escrito transacionalmente a cada nível. Quando uma transação aninhada é confirmada, as versões dos blocos de memória por ela acessados são integrados à transação no nível acima na hierarquia de transações. Escritas especulativas de uma transação aninhada são facilmente descartadas sem descartar as escritas da transação que a originou. Enquanto não muito eficiente em termos de utilização de espaço, consideramos a abordagem viável devido a sua fácil modelagem e rapidez de execução. E como a cache transacional possui poucas linhas e o número de níveis de aninhamento suportado em *hardware* é limitado, o tamanho final da cache transacional continua pequeno.

As configurações de memória, barramento e adaptadores são as mesmas para o protótipo da seção 3.3. Notamos que a questão de virtualização de recursos não foi tratada neste protótipo, de modo os programadores são encorajados a escrever transações que não to-

quem um número de posições de memória muito grande ou estendam a execução das transações por um tempo que exceda o período entre trocas de contexto entre linhas de execução.

3.4.2 Avaliação do protótipo

Buscamos avaliar o impacto que o aninhamento de transações traz ao desempenho de sistemas de memória transacional. O benefício principal do uso de aninhamento, como vimos no capítulo 2, seria a possibilidade de economizar o trabalho de re-executar toda uma transação, detectando conflitos da transação aninhada e re-executando somente esta.

Desenvolvemos dois *benchmarks* para avaliação deste protótipo:

- **Lista duplamente encadeada:** semelhante ao *benchmark* de lista encadeada do primeiro estudo de caso. Desta vez, no entanto, as transações de manipulação da lista e da pilha são aninhadas dentro de uma transação externa.
- **Árvores AVL:** são instanciadas duas árvores binárias balanceadas do tipo AVL. Metade dos processadores remove elementos da primeira árvore e os coloca na segunda, enquanto que a outra metade remove elementos da segunda árvore e os insere na primeira. As operações de inserção e remoção ocorrem em transações aninhadas dentro de outra mais externa. A operação de remoção faz uma busca por um elemento definido de forma pseudo-aleatória, com o intuito de removê-lo. Caso esta busca não encontre o elemento especificado, um segundo elemento próximo a este (localizado no nó que serviria como "pai" do elemento sendo buscado) é selecionado e removido em seu lugar. Desta forma, remoções sempre retiram algum elemento (a não ser que a árvore esteja completamente vazia), de forma a maximizar a possibilidade de conflitos.

Uma transação aninhada precisa ser reiniciada caso tiver abortado e caso a transação mais externa ainda for válida. No entanto, como vimos na subseção 3.4.1, uma transação aninhada não pode ser reiniciada por um número indeterminado de vezes, ou bloqueios mútuos poderão ocorrer. Espera-se que implementações de memória transacional resolvam este problema de forma transparente ao programador. Neste protótipo estamos lidando com memórias transacionais em um nível de abstração um pouco mais baixo, escrevendo manualmente código que idealmente seria gerado de forma automática por um compilador. Portanto, o desenvolvedor de *software* precisa especificar de forma programática por quantas vezes a transação aninhada deverá re-executar até desistir de executá-la, abortando a transação mais externa e começando tudo de novo. Nos *benchmarks* escritos para a avaliação deste estudo de caso buscamos estudar esta variável.

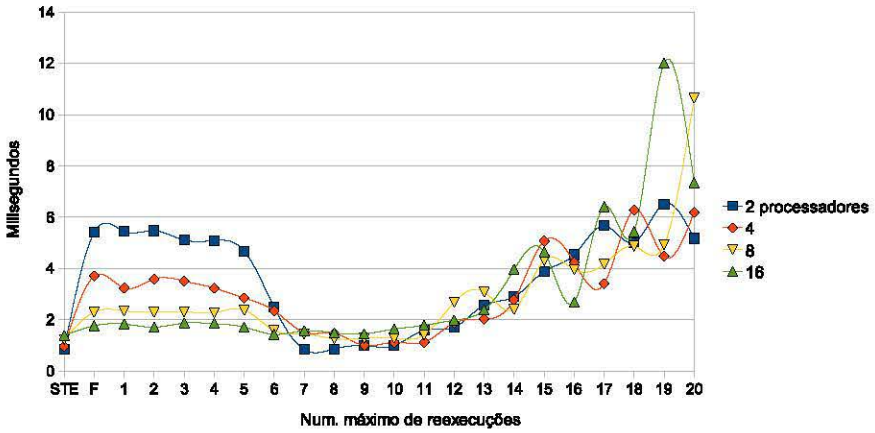


Figura 3.17: Tempo do simulador, lista duplamente encadeada. (MIPS).

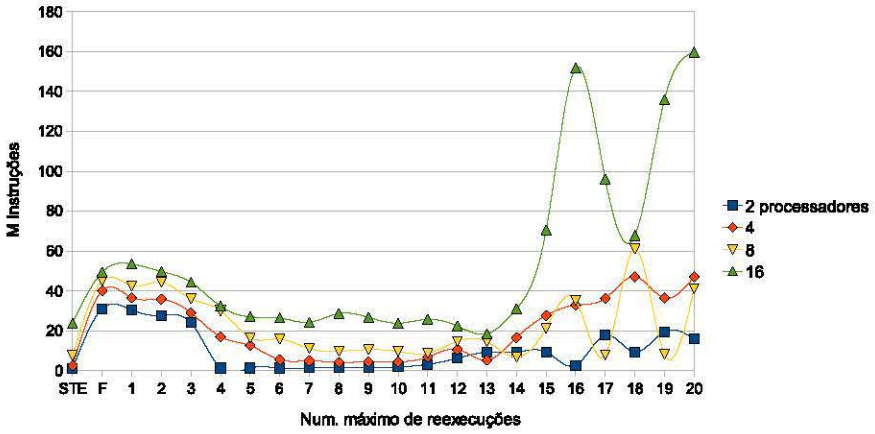


Figura 3.18: Tempo do simulador, lista duplamente encadeada. (SPARC).

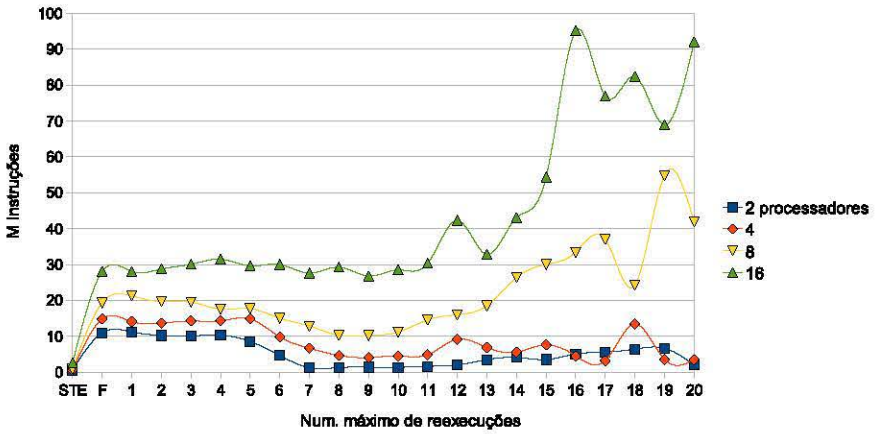


Figura 3.19: Tempo do simulador, lista duplamente encadeada. (PowerPC).

Nas figuras 3.17 até 3.19 apresentamos os resultados de tempo de simulador (retornado pela função **sc_simulation_time** da biblioteca SystemC) de simulações das arquiteturas MIPS, SPARC e PowerPC, respectivamente, para o *benchmark* da lista duplamente encadeada, e nas figuras 3.20 até 3.22 apresentamos os resultados para o *benchmark* das árvores AVL para as respectivas arquiteturas. Nos eixos horizontais fizemos variar o número de vezes que as transações internas são re-executadas em caso de aborto. Note que as primeiras duas posições neste eixo (*STE* e *F*) na verdade são exceções à regra. *STE* significa "Sem Transação Externa", e representa o cenário no qual as duas transações que cada processador executaria como internas não são envoltas por outra mais externa, sendo este o caso em que esperamos melhor desempenho em termos de tempo de simulador, dado que cada transação é confirmada de forma independente. *F* significa *Flattening*, ou seja, as duas transações internas na verdade formam uma com a transação externa, sem abortos e re-execuções independentes. Espera-se que este segundo caso tenha o pior desempenho, uma vez que as transações resultantes são maiores que as transações que as compõem, oferecendo maior potencial para conflitos. Apresentamos os dados de simulação para 2 até 16 processadores.

Podemos verificar que no *benchmark* da lista duplamente encadeada o número ideal de re-execuções das transações internas fica entre 7 e 11. Na maior parte das vezes, re-executando as transações cerca de oito vezes serve para trazer o desempenho para muito próximo do ideal. Notamos que o potencial para bloqueios mútuos entre duas linhas de execução existe somente quando a lista possue apenas um elemento, o que dificilmente ocorre dado que vários elementos estão disponíveis para as linhas. Este *benchmark* específico consegue tirar proveito de forma eficiente dos mecanismos de aborto e re-execução independentes das trasações aninhadas.

Já no caso das árvores AVL, o aninhamento de transações não traz nenhum ganho de desempenho. Notamos que o potencial para bloqueios mútuos entre as transações é muito maior que no *benchmark* da lista ligada, dado que remoções e inserções freqüentemente causam rebalanceamento das árvores, o que exige acesso exclusivo a vários elementos das estruturas. Neste caso, como as o potencial para bloqueios é grande e a única solução é abortar uma das transações externas e reiniciá-la, a re-execução das transações aninhadas somente causa retrabalho que será descartado de qualquer forma. Neste caso específico, o aninhamento de transações com aborto e re-execuções independentes não trouxe melhora no desempenho.

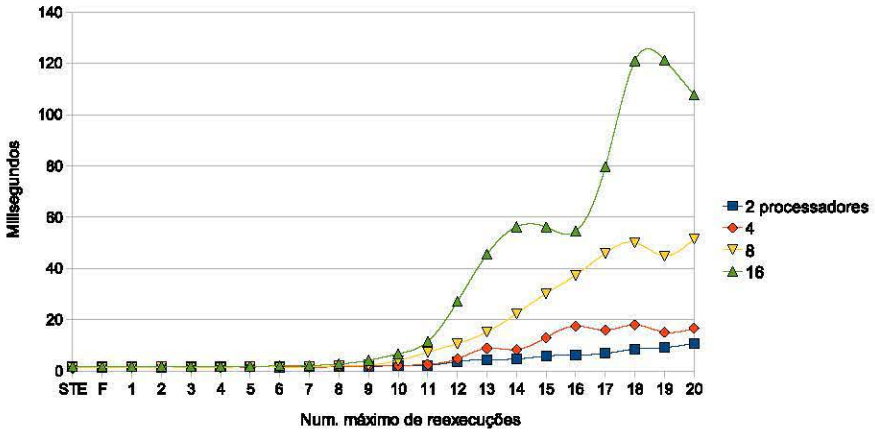


Figura 3.20: Tempo do simulador, árvores AVL. (MIPS).

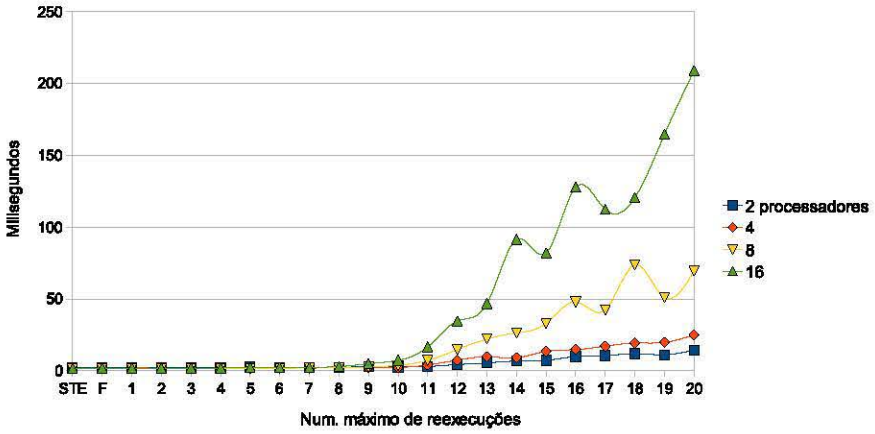


Figura 3.21: Tempo do simulador, árvores AVL. (SPARC).

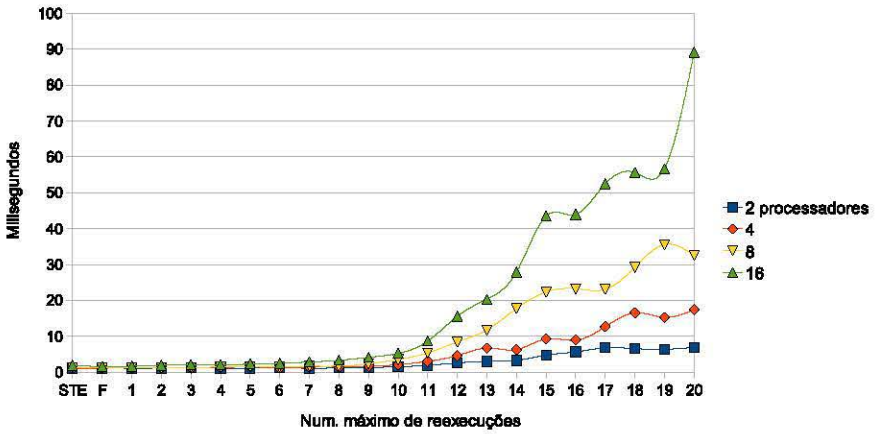


Figura 3.22: Tempo do simulador, árvores AVL. (PowerPC).

Capítulo 4

Memória transacional com suporte em *software* e experimentos com gerenciamento de contenção

Neste capítulo apresentamos as modificações feitas a uma implementação de memória transacional baseada em *software* (descrita em detalhes na subseção 4.1.8), sem suporte específico em *hardware*, com o intuito de possibilitar a experimentação com diferentes estratégias de gerenciamento de contenção entre transações.

As modificações propostas permitem um controle muito maior por parte do programador sobre os gerenciadores de contenção a serem utilizados pelas transações. Permitem ao programador associar um gerenciador específico a cada transação, bem como amarrar a estratégia de gerenciamento de contenção aos dados de um aplicativo baseando-se nos padrões de acesso a estes dados. Conduzimos experimentos avaliando a implementação e apresentamos resultados para uma variedade de sistemas de computação.

4.1 Trabalhos relacionados

4.1.1 STM de Shavit e Touitou

O trabalho pioneiro de Shavit e Touitou, publicado em 1995, cunhou o termo *software transactional memory* [40]. O modelo de programação para esta implementação requer que o programador indique as posições de memória lidas ou escritas por uma transação antes de esta ser iniciada. Ao iniciar a transação, as posições de memórias a serem lidas ou escritas são adquiridas em uma ordem específica (por exemplo a ordem crescente dos endereços) para evitar bloqueios mútuos. Então a transação executa até o final, sem risco de ser abortada, liberando no momento da conclusão as posições de memória por ela

tocadas. A granularidade de detecção de conflitos é do tamanho da palavra da arquitetura, o que incorre em um custo adicional de espaço de memória, uma vez que para cada palavra de dados precisamos de uma outra palavra para servir-lhe de trava de exclusão.

4.1.2 DSTM

Dynamic Software Transactional Memory (DSTM) [35] recebeu este nome por ser uma das primeiras implementações de memória transacional em *software* que não requeriam que o programador especificasse ao início da transação quais posições de memória seriam tocadas. DSTM introduziu a idéia de implementar o gerenciamento de contenção como um objeto explícito, encapsulando nele a política necessária para resolver conflitos entre transações. Implementada como uma biblioteca acessível a código C++ e Java, serviu como base para o desenvolvimento de vários trabalhos subseqüentes, com RSTM, SXM [20], ASTM [32] e OSTM [19].

4.1.3 WSTM

Word granularity STM (WSTM) [23] é uma implementação de memória transacional completamente integrada a uma linguagem de programação, trazendo para a linguagem Java a construção **atomic**. Apesar de estar integrada a uma linguagem de programação orientada a objetos, a granularidade de detecção de conflitos é feita no nível de palavra da arquitetura do processador, e poderia facilmente dar suporte a linguagens procedurais. Como DSTM, o programador não precisa especificar as posições de memória a serem tocadas pela transação ao seu início.

4.1.4 STM de Ananian e Rinard

Os autores propõe uma implementação de STM com isolamento forte, um atributo que a distingue as demais propostas [4]. Além disto, neste sistema acessos feitos por código não transacional podem abortar transações concorrentes, ou seja, o sistema usa transações contínuas. A maioria das outras propostas de STM implementam isolamento fraco e transações intermitentes, pois não requerem a instrumentação de código não transacional. Ananian e Rinard reduziram o trabalho adicional requerido pela instrumentação utilizando um valor sentinela nas posições de memória acessadas pelas transações. O monitoramento da sentinela tem custos baixos o suficiente para ser realizado a cada acesso à memória. A proposta foi implementada em Java usando granularidade de detecção de conflitos no nível de objetos, e executada em uma arquitetura de computador simulada.

4.1.5 TL

Transactional Locking (TL) usa travas de exclusão para guardar objetos escritos transacionalmente [14]. O sistema permite que as travas sejam adquiridas tanto durante o primeiro acesso a um objeto transacional quanto durante a confirmação da transação. O sistema pode realizar detecção de conflitos no nível de palavras da arquitetura, regiões de memória ou objetos da linguagem. O uso de travas de exclusão simplifica a implementação do sistema se comparado à implementação de STM não bloqueantes, o que pode melhorar o desempenho. Estes resultados servem para apoiar um trabalho anterior por Ennals, que também utilizava travas de exclusão pra implementar uma STM bloqueante, mas que na sua época foi considerado controverso demais para ser publicado em congressos científicos [16].

4.1.6 TinySTM

TinySTM é uma implementação de STM bastante simples, derivada do algoritmo baseado em travas de exclusão usado por TL [17]. Ela faz detecção de conflitos com granularidade do tamanho da palavra da arquitetura do processador. O sistema de resolução de conflitos entre transações é bastante simples: uma transação aborta quando tenta acessar algum recurso já adquirido por outra. Por sua simplicidade, é surpreendente o seu desempenho se comparado ao de outras implementações de STM [15], muitas vezes superando-as.

4.1.7 McRT-STM

Desenvolvido para executar no ambiente de simulação McRT, da Intel, este sistema possui algumas características interessantes. Ele atualiza diretamente as posições de memória escritas por uma transação, restaurando o valor antigo caso a transação abortar. Isto resulta na diminuição do custo da confirmação das transações e facilita a resolução de dependências do tipo RAW (*read-after-write*). Como TL, McRT-STM utiliza travas de exclusão mútua para guardar as posições de memória sendo escritas. Quando uma linha de execução não consegue adquirir uma trava de exclusão, é posta na lista de espera do sistema operacional até que a trava seja liberada novamente, e temporizadores são usados para detectar e resolver bloqueios mútuos. McRT-STM implementa transações tanto em C/C++ quanto em Java, suportando detecção de conflitos no nível de objetos ou linhas de cache.

4.1.8 RSTM

RSTM é uma implementação de memória transacional baseada em *software* (STM) e implementada como uma biblioteca C++. Em nosso trabalho exploramos a versão 3 da biblioteca RSTM. RSTM possui uma interface de programação baseada em *smart pointers* e *templates*, que tem como objetivo reduzir a complexidade de programação e capturar vários erros de programação comuns [12]. RSTM também permite ao desenvolvedor experimentar diferentes tipos de implementações internas (*backends*) de memória transacional sem a necessidade de reescrever o código de aplicação. RSTM possui duas implementações internas. A primeira é não-bloqueante e utiliza uma única indireção para acessar dados transacionais. A segunda é bloqueante, sem níveis de indireção e baseada em registros de gravação (*redo-logs*).

A implementação não-bloqueante da biblioteca RSTM utiliza apenas um nível de indireção para acessar dados agregando informações adicionais a cada objeto transacional. Mais especificamente, dois novos campos são adicionados a cada objeto transacional: um apontando para o descritor da transação ao qual o objeto pertence e o outro apontando para a versão antiga do objeto. Um objeto está sob propriedade de uma transação se o descritor de transação para o qual aponta possuir o status "ativo" (*ACTIVE*). Se o descritor tiver o estatus "abortado" (*ABORTED*), então a versão atual do objeto é aquela apontada como sendo a versão "antiga". Caso o status for "confirmado" (*COMMITED*), então o objeto já está em sua versão correta. Um objeto transacional é acessado através de um objeto de cabeçalho especial, desta forma implicando em apenas um nível de indireção para acessos. Uma vez que uma transação tenha lido um objeto, podemos ter certeza de que a versão lida não irá mudar. Quando uma transação realiza escritas em um objeto, o faz na verdade a uma cópia privada, retornada pelo método *clone* obrigatoriamente implementado por todos os objetos transacionais. Se a implementação utiliza leituras invisíveis, a validação de todos os objetos lidos ou escritos desde o início de uma transação precisa ser realizada incrementalmente cada vez que um objeto é aberto para leitura ou escrita. A implementação da biblioteca também dá suporte a leitores visíveis.

A implementação interna baseada em registros de gravação (*redo-logs*) também adiciona dois campos a cada objeto transacional. Um objeto não pertence a nenhuma transação se o primeiro campo se comportar como um número de versão ímpar e se o segundo campo for nulo. De outra forma, o objeto foi adquirido, e o primeiro campo aponta para o descritor da transação que adquiriu o objeto e o segundo campo aponta para o registro de gravação. Um caso especial ocorre quando o primeiro campo possui o valor 2, e o dono do objeto está presumivelmente copiando o registro de gravação para o objeto. Objetos portanto ficam inacessíveis durante a aplicação do registro de gravação, e qualquer transação que deseja acessá-los precisa esperar. Diferentemente da implementação não-bloqueante, objetos transacionais são acessados diretamente, sem serem referenciados através de um

objeto de cabeçalho. O leitor deve notar que um objeto aberto para leitura não é imutável, uma vez que a aplicação do registro de gravação ocorre *in loco*, e portanto uma transação precisa validar incrementalmente os objetos por ela abertos a cada acesso, e não somente quando da sua abertura. Além do método *clone*, objetos transacionais precisam implementar também uma operação de aplicação do registro de gravação (*redo*)

Em ambas as implementações internas da biblioteca (a bloqueante e a não-bloqueante), uma transação é confirmada utilizando uma instrução atômica do tipo *comparar-e-trocar* (CAS) para trocar o status da transação de *ACTIVE* para *COMMITED* em seu descritor. A implementação bloqueante, no entanto, também requer uma operação de CAS que opera sobre duas palavras do processador ao mesmo tempo, para a aquisição dos objetos transacionais, enquanto que a implementação não bloqueante requer um CAS simples para adquirir um objeto ou aplicar uma versão nova do mesmo ao seu cabeçalho. Ambas as implementações bloqueante e não-bloqueante suportam aquisições tardias ou imediatas. Se uma transação tenta adquirir um objeto que foi adquirido por outra, um gerenciador de contenção é invocado para arbitrar o conflito.

4.1.9 Outros trabalhos sobre gerenciamento de contenção

O gerenciamento de contenção entre transações em memória foi extensivamente estudado em outros trabalhos publicados [39, 21, 20]. Dentre estes trabalhos, o de maior interesse em nosso contexto é um sobre gerenciamento polimórfico de contenção [20]. No trabalho referido, diferentes gerenciadores de contenção podem ser associados a diferentes transações de uma forma parecida com a apresentada em nosso trabalho, apesar de os autores não explorarem a noção de associação de diferentes gerenciadores de contenção aos dados baseando-se nos padrões de acesso a estes. Em vez disso, os autores propõem a adaptação da estratégia de gerenciamento de contenção baseando-se na variação da carga de trabalho — mais precisamente, escolhendo o gerenciador de contenção baseando-se no número de linhas de execução ativas no programa.

ASTM explora outros aspectos da adaptação do sistema de memória transacional baseando-se na carga de trabalho do programa [32]. Explora quatro dimensões diferentes do espaço de projeto de um sistema de TM: aquisições imediatas versus aquisições tardias, o método para aquisição de objetos, a estrutura de meta-dados, e diferentes semânticas não-bloqueantes para transações. O sistema de TM adapta-se ao longo destas quatro dimensões em tempo de execução para atingir as necessidades da aplicação. ASTM não explora o uso ou adaptação de diferentes gerenciadores de contenção de acordo com os padrões de acesso ao dados de um programa.

4.2 Trabalho realizado

4.2.1 Gerenciamento de contenção em RSTM

Como em DSTM [35], ASTM [32] e SXM [20], o gerenciamento de contenção é tratado como um aspecto modular do sistema. Cada transação é associada a um objeto que representa a sua estratégia de gerenciamento de contenção corrente. Este objeto possui métodos que casam o ciclo de vida de uma transação (**onBeginTransaction**, **onTryCommitTransaction**, **onTransactionCommitted**, e **onTransactionAborted**), métodos que casam os diferentes eventos que ocorrem devido a interações com objetos transacionais (**onContention**, **onOpenRead**, **onOpenWrite** e **onReOpen**), e um método para decidir se deve ou não abortar uma transação conflitante (**shouldAbort**). A estratégia de gerenciamento de contenção não pode ser mudada no decorrer de uma transação, uma vez que os métodos invocados em função do ciclo de vida da transação são em geral utilizados para inicializar e atualizar os dados internos do objeto que representa a estratégia de gerenciamento.

Quando uma transação detecta que um objeto ao qual deseja acesso foi adquirido por outra transação, a primeira decide se espera a segunda liberar o objeto ou se a aborta e toma-lhe o recurso. O método **shouldAbort** do gerenciador de contenção da primeira transação é invocado, passando como parâmetro o gerenciador de contenção da segunda transação para que a decisão por esperar ou abortar a segunda transação seja tomada.

Várias estratégias de gerenciamento de contenção foram propostas na literatura [39, 21]. De interesse em nosso trabalho são os gerenciadores *Aggressive*, *Greedy*, *Eruption*, *Highlander*, *Karma*, *Killblocked*, *Polka*, *Whpolka* e *Polkaruption*. Outras estratégias de gerenciamento foram implementadas em RSTM, mas não as descreveremos aqui por por terem apresentado desempenho consideravelmente inferior no *benchmark* apresentado em nosso trabalho, se comparados aos gerenciadores citados.

O gerenciador de contenção *Aggressive* é o mais simples de todos: ele prontamente aborta qualquer transação conflitante. O gerenciador *Greedy* usa um marcador de tempo (*timestamp*), adquirido pela transação quando de sua primeira tentativa de execução, para determinar sua "idade". Se duas transações estão em conflito, em geral a mais velha prevalece. Mas se a transação mais jovem está bloqueada pela mais velha e detecta que esta também está bloqueada, esperando por um recurso adquirido por outra transação, então a transação mais jovem aborta a mais velha e toma-lhe o recurso. O gerenciador *Killblocked* marca a transação como bloqueada quando esperando por algum recurso transacional. Se em uma tentativa subseqüente de abrir o mesmo objeto a transação adversária também está bloqueada, a adversária é abortada. De outra forma a transação continua esperando por um certo número de intervalos de tempo fixos, tentando acessar o mesmo objeto, até que desiste de esperar e aborta a transação conflitante.

O gerenciador de contenção *Karma* dá prioridade a uma transação baseando-se no número de objetos acessados por ela desde sua primeira tentativa de execução. A contagem de objetos acessados é portanto reiniciada quando a transação termina de forma bem sucedida. Uma transação de prioridade menor espera por um certo número de intervalos de tempo fixos, caso tentar acessar um objeto adquirido por outra transação de prioridade mais alta, mas se o número de tentativas para acessar um recurso excede a diferença entre as prioridades das duas transações, a transação de prioridade mais baixa aborta a de prioridade mais alta e toma-lhe o recurso. O gerenciador *Polka* é muito parecido com o *Karma*, a diferença sendo que a transação bloqueada espera por intervalos de tempo exponencialmente crescentes, com um componente randômico. O gerenciador *Eruption* também deriva de *Karma*, com a diferença de que a transação bloqueada adiciona sua prioridade a da transação conflitante para que esta tenha a possibilidade de vencer conflitos com outras adversárias, terminar sua execução e liberar o recurso o quanto antes. *Polkaruption* combina os princípios de *Polka* e *Eruption*: como *Polka*, *Polkaruption* usa o número de objetos acessados para determinar a prioridade de uma transação e faz esperas exponencialmente maiores (com um componente randômico) quando a transação está bloqueada. Como *Eruption*, adiciona sua prioridade à prioridade da transação conflitante, para que esta termine o quanto antes e libere o recurso sob conflito. *Highlander* também se baseia nos princípios de *Polka*, mas quando uma transação aborta sua adversária, adiciona a prioridade da adversária a sua. Outra estratégia de gerenciamento de contenção baseada em *Polka* é a *Whpolka*, na qual objetos abertos para escrita tem peso maior ao incrementar a prioridade da transação.

4.2.2 Modificações Propostas

Foi necessário realizar algumas modificações na biblioteca RSTM de modo a permitir que diferentes gerenciadores de contenção possam ser associados a diferentes transações ou a diferentes objetos transacionais. Primeiramente, tentamos inserir um campo adicional em cada objeto transacional para denotar o gerenciador de contenção a ele associado, e fazer com que a transação detectasse qual gerenciador de contenção estava associado ao primeiro objeto aberto pela transação e utilizar este gerenciador pelo restante de sua execução. Este campo adicional teria semânticas de acesso semelhantes aos demais campos do objeto definidos pelo programador, e portanto os métodos **clone** e **redo** precisariam levá-lo em consideração ao criar clones e aplicar registros de gravação. Esta abordagem demonstrou-se inviável em função do trabalho adicional inserido nos métodos **clone** e **redo**. Também tentamos utilizar herança para introduzir o campo citado apenas em objetos transacionais específicos, mas então foi preciso utilizar conversão de tipos dinâmica ao consultar o objeto a respeito de qual gerenciador de contenção estava associado a ele,

```
1.  class RBTree
2.   {
3.   private:
4.     stm::cm::CMEnum m_cm;
5.
6.     stm::sh_ptr<RBNode> sentinel;
7.
8.   public:
9.     RBTree(stm::cm::CMEnum cm)
10.        : m_cm(cm),
11.          sentinel(new RBNode())
12.      { }
13.
14.    virtual bool lookup(int val) const
15.      {
16.        BEGIN_TRANSACTION_CM(m_cm);
17.
18.        // ...
19.
20.        END_TRANSACTION;
21.      }
22.
23.    // ...
24.
25.  };
```

Figura 4.1: Segmento de código da árvore vermelha-e-preta.

o que pareceu ser outra fonte significante de trabalho adicional. E de qualquer forma, um teste adicional ainda deveria ser realizado a cada abertura de objeto pela transação, para determinar se o mesmo era o primeiro sendo acessado.

Decidimos então usar uma abordagem mais simples e direta. Permitimos ao programador associar uma estratégia de gerenciamento de contenção a uma transação, modificando a *macro* que delimita seu início (**BEGIN_TRANSACTION**), fazendo com que recebesse como parâmetro um valor enumerado capaz de identificar o gerenciador de contenção a ser usado. O gerenciador pode desta forma ser associado ao objeto que encapsula a estrutura de dados transacional, e diferentes gerenciadores de contenção podem ser associados a diferentes estruturas de dados ou mesmo a diferentes operações em uma mesma estrutura de dados. Cada linha de execução possui um vetor com gerenciadores de contenção pré-alocados, e o parâmetro passado a **BEGIN_TRANSACTION** é usado

para configurar o gerenciador de contenção a ser utilizado pela transação ao começo de sua execução. Na figura 4.1 mostramos o idioma de programação descrito no contexto da implementação de uma árvore binária balanceada. Note que na linha 4 definimos um campo que armazena o tipo do gerenciador de contenção associado à estrutura de dados. Este campo é inicializado no construtor (linha 10) e usado para especificar o gerenciador de contenção em operações de busca na estrutura (linha 16). Note que cada instância de árvore balanceada no programa pode ser associada com diferentes gerenciadores de contenção.

O trabalho adicional inserido é bastante pequeno e pago somente uma vez, no início da transação. Também precisamos realizar pequenas limpezas no código dos gerenciadores de contenção, para permitir a interação entre diferentes implementações de gerenciadores (lembremo-nos de que **shouldAbort** pode receber como parâmetro gerenciadores de tipos arbitrários). Estas limpezas de fato trouxeram pequena melhora ao desempenho da biblioteca, se comparado com a implementação original de RSTM.

Para compilar e executar a versão baseada em registros de gravação em arquiteturas x86 de 32 bits, precisamos portar a biblioteca para funcionar nesta plataforma, uma vez que o time de desenvolvimento da versão 3 de RSTM dá suporte a esta implementação interna somente para a plataforma SPARC. Notamos no entanto que a versão 4 da biblioteca, recentemente disponibilizada, possui suporte a ambas as implementações internas (bloqueante e não-bloqueante) tanto em arquiteturas SPARC quanto x86.

4.2.3 Experimentos

Para avaliar as modificações feitas à biblioteca de STM, concebemos um *benchmark* consistindo em estruturas de dados com diferentes padrões de acessos. Para isolar componentes capazes de interferir na avaliação, as estruturas de dados devem ser do mesmo tamanho e tipo, a única diferença sendo o nível de contenção encontrado pelas transações ao acessá-las.

Escolhemos aproveitar a implementação de árvores balanceadas do tipo vermelha-e-preta encontrada junto aos *benchmarks* disponíveis na implementação original da biblioteca RSTM. Árvores balanceadas exibem grande potencial de desempenho para acessos paralelos a diferentes partes de uma mesma árvore, ao mesmo tempo que provêm a oportunidade de conflitos entre transações quando atualizações precisam rebalancear as subárvores, propagando modificações na estrutura de dados desde os nós-folha até a raiz. Criamos uma tabela de dispersão de tamanho fixo contendo árvores balanceadas. Operações sobre a estrutura de dados composta precisam primeiramente encontrar a árvore apropriada para o operando. Isto é feito dividindo o operando pelo número de árvores, e usando o resultado para indexar a tabela de dispersão.

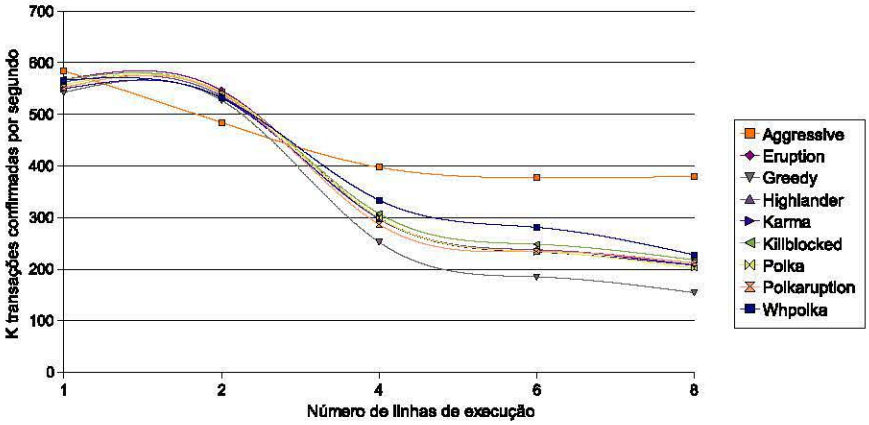


Figura 4.2: Core 2 Duo, não-bloqueante, alta contenção.

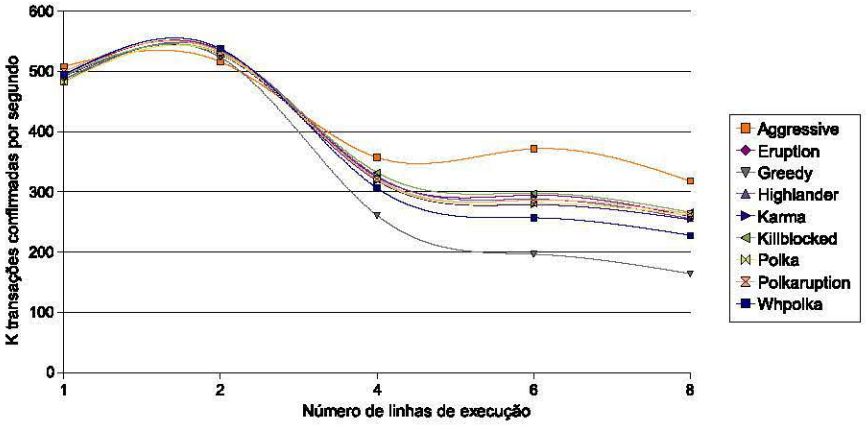


Figura 4.3: Core 2 Duo, não-bloqueante, baixa contenção.

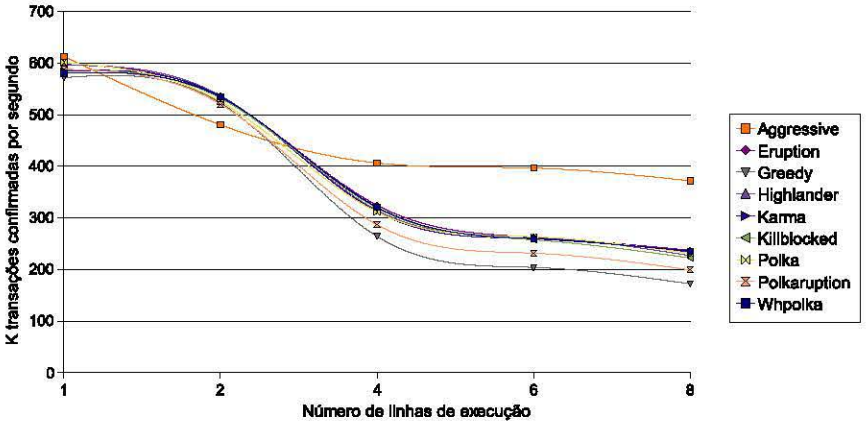


Figura 4.4: Core 2 Duo, bloqueante, alta contenção.

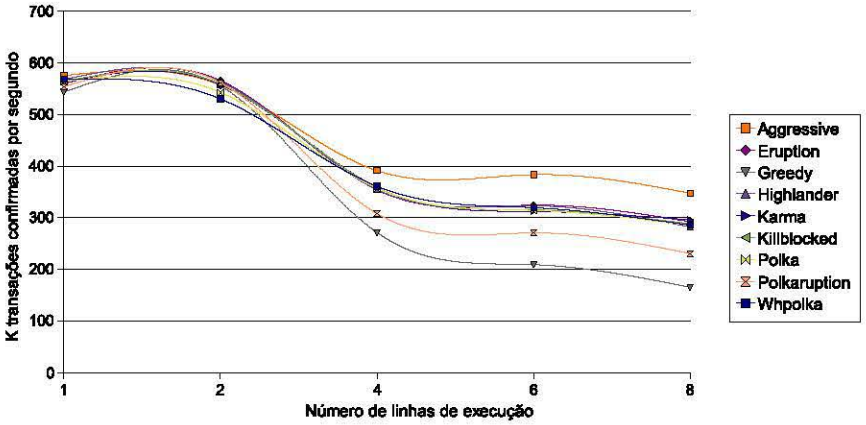


Figura 4.5: Core 2 Duo, bloqueante, baixa contenção.

Para assegurar diferentes padrões de acesso a diferentes estruturas de dados, geramos os operandos com diferentes probabilidades. Há uma chance aproximada de 50% de que o número gerado seja mapeado à primeira árvore, enquanto os 50% restantes são igualmente distribuídos entre as demais árvores binárias. Portanto, a primeira árvore possui um padrão de acesso de alta contenção, enquanto as demais são acessadas sob baixa contenção. Conduzimos experimentos com 2 até 6 árvores de baixa contenção, e como os resultados se mostraram consistentes ao longo deste espectro, escolhemos mostrar apenas os resultados para 1 árvore binária de alta contenção e 4 árvores de baixa.

Todos as variações do *benchmark* foram executadas três vezes por períodos de 60 segundos, e as médias aritméticas foram tiradas. A estratégia de alocação de memória utilizada foi a pilha de memória com coleta automática de lixo, as heurísticas de contador global de finalização de transações foram desligadas e a privatização de dados feita através de barreiras unidirecionais transacionais (*transactional fences*). A estratégia de validações utilizada foi a de leitores invisíveis, com aquisições imediatas. Mapeamos um total de 256 elementos a cada árvore, isto é, para uma árvore de alta contenção e 4 de baixa, os operandos encontram-se na amplitude de 0 até 1279. Os tipos de operações foram particionados igualmente, sendo um terço buscas, um terço inserções e um terço remoções. Executamos os *benchmarks* em ambas as implementações internas da biblioteca RSTM, a não-bloqueante e a bloqueante. Três sistemas de computação diferentes foram utilizados para as execuções. O primeiro, um sistema baseado no processador Intel Core 2 Duo a 2.8 GHz, com 2 GB de memória RAM. O segundo, um sistema baseado no processador Intel Core 2 Quad, a 2.4 GHz e com 4 GB de RAM. O terceiro, um sistema com dois processadores Intel Core 2 Quad a 2GHz e 4 GB de RAM. Todos os sistema utilizaram como sistema operacional uma distribuição Linux padrão (*kernel* 2.6).

Avaliamos o melhor gerenciador de contenção para uso sob alta e baixa contenções, e também em um cenário de contenção mista. Primeiramente executamos experimentos para apenas uma árvore balanceada, sob alta contenção, e apresentamos os resultados nas figuras 4.2, 4.4, 4.6, 4.9, 4.12 e 4.14. Então executamos os experimentos com as árvores de baixa contenção, e mostramos os resultados para as execuções com quatro estruturas de dados nas figuras 4.3, 4.5, 4.7, 4.10, 4.13 e 4.15. O próximo passo seria identificar os melhores gerenciadores de contenção para alta e baixa contenção e então associar os melhores gerenciadores às estruturas de dados baseando-se nos padrões de acesso a estas estruturas de dados.

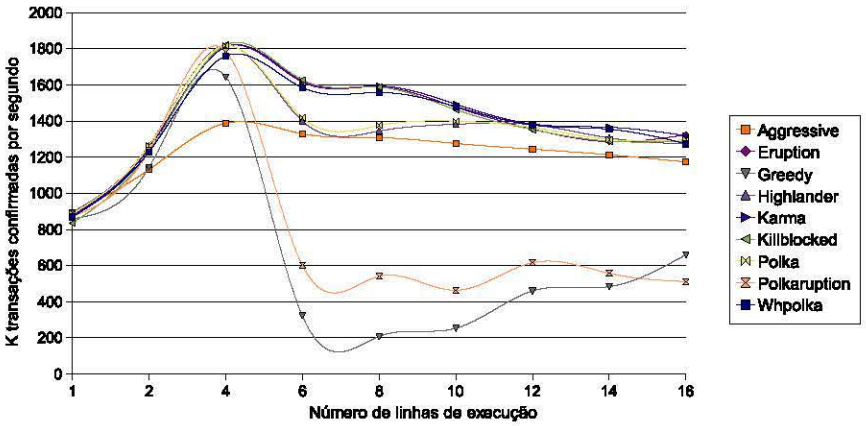


Figura 4.6: Core 2 Quad, não-bloqueante, alta contenção.

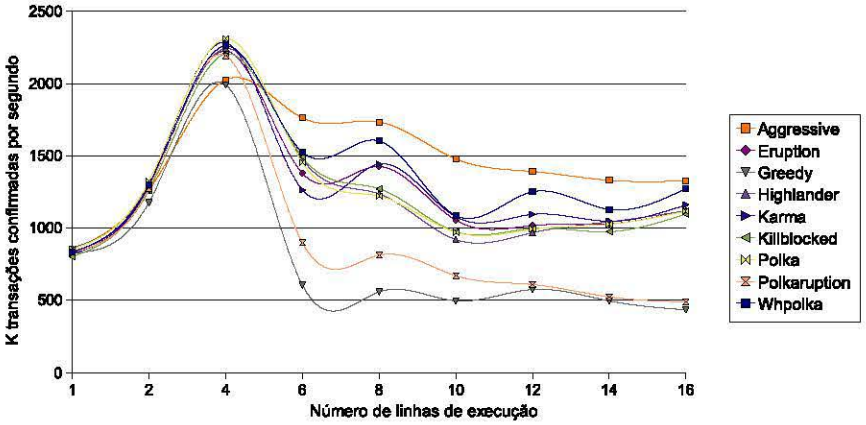


Figura 4.7: Core 2 Quad, não-bloqueante, baixa contenção.

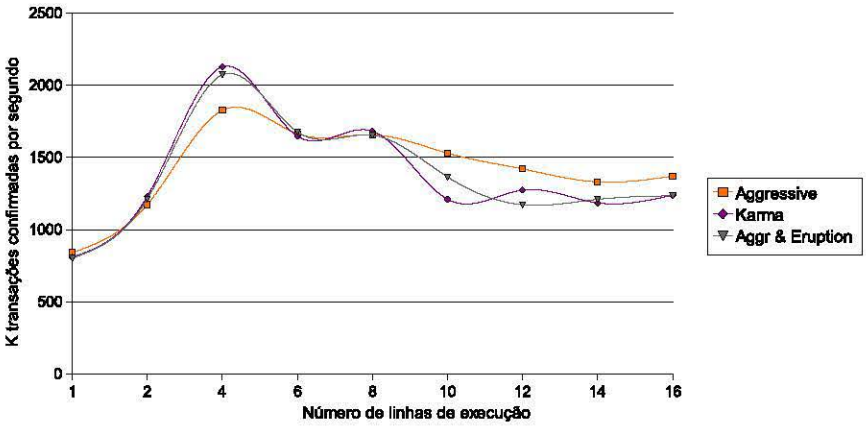


Figura 4.8: Core 2 Quad, não-bloqueante, contenção mista.

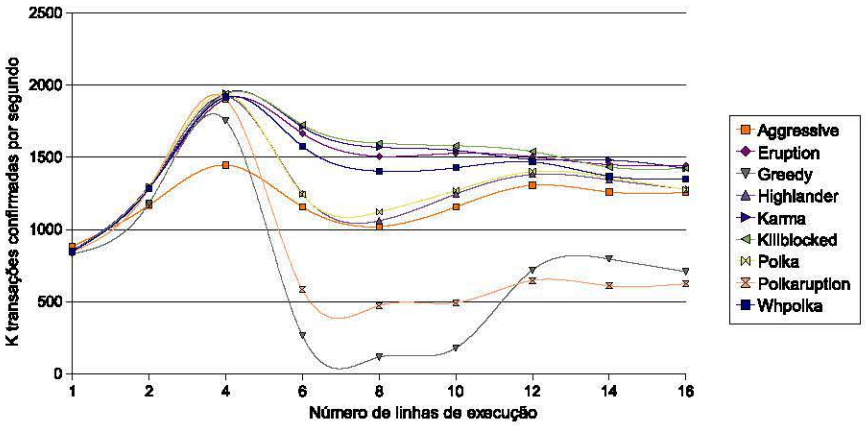


Figura 4.9: Core 2 Quad, bloqueante, alta contenção.

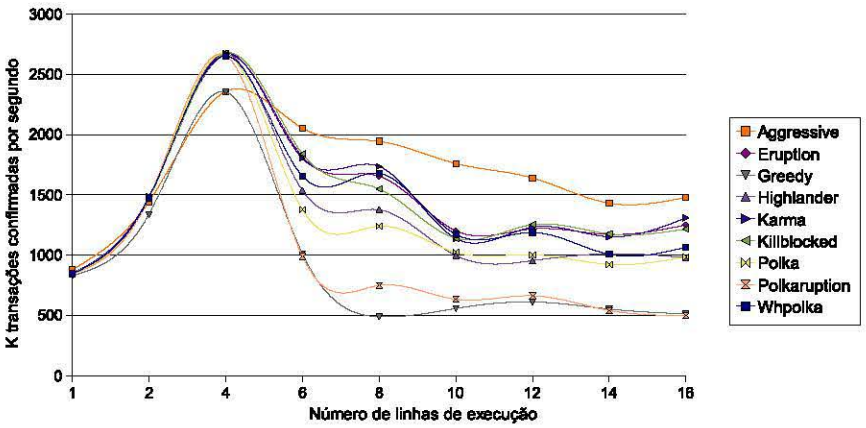


Figura 4.10: Core 2 Quad, bloqueante, baixa contenção.

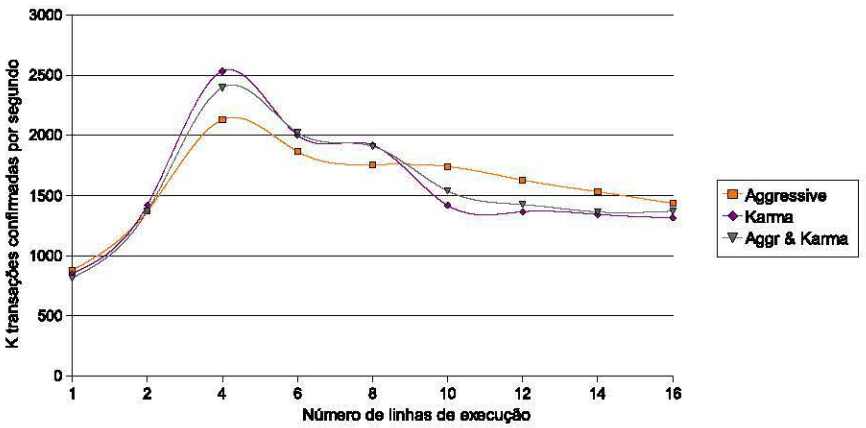


Figura 4.11: Core 2 Quad, bloqueante, contenção mista.

Fizemos com que o número de linhas de execução variasse ao longo das execuções do *benchmark*, de uma linha até o número de linhas de execução igual a quatro vezes o número de núcleos de processamento no sistema. Como Dragojevic argumenta [15], não podemos esperar que um sistema de TM aumente seu desempenho quando o número de linhas de execução excede o número de núcleos de processamento, mas seu desempenho deve ao menos degradar de forma amena sob estas circunstâncias, uma vez que usuários de um programa escrito com memória transacional na maioria das vezes não poderão fazer um ajuste fino da aplicação de acordo com o número de núcleos de processamento disponíveis. Também argumentamos que o número de núcleos do sistema é apenas um limite máximo do número de núcleos que um programa terá disponível a qualquer momento, e que em um ambiente multitarefa é bem provável que um aplicativo escrito com memória transacional terá de competir com outros aplicativos (possivelmente também paralelos) pelos núcleos de processamento do sistema.

As figuras 4.2 até 4.5 mostram os resultados para as execuções do *benchmark* no sistema Intel Core 2 Duo. As primeiras duas figuras são os resultados para a implementação interna não bloqueante, enquanto que as duas seguintes são para a implementação bloqueante baseada em registros de gravação. Como o melhor gerenciador de contenção foi o mesmo tanto no cenário de alta quanto no cenário de baixa contenção — *Aggressive*, não tentamos executar os testes em um cenário de contenção mista.

Resultados semelhantes são apresentados para o sistema com dois processadores Intel Core 2 Quad (com um total de oito núcleos de processamento), nas figuras 4.12 até 4.15. Podemos ver que tanto sob alta quanto sob baixa contenção um mesmo gerenciador apresentou o melhor desempenho, desta vez o *Killblocked*. Nossa expectativa era encontrar um gerenciador que tivesse desempenho melhor sob baixa contenção e outro que apresentasse desempenho superior sob alta contenção. Novamente, não executamos o *benchmark* em um cenário de contenção mista.

Os resultados para o sistema com um processador Core 2 Quad (com um total de 4 núcleos de processamento) são mostrados nas figuras 4.6 até 4.11. Sob alta contenção (figuras 4.6 e 4.9), quatro gerenciadores apresentaram o melhor desempenho. Estes foram *Killblocked*, *Karma*, *Eruption* e *Whpolka*. Sob baixa contenção (figuras 4.7 e 4.10), o gerenciador *Aggressive* ofereceu o melhor desempenho de uma forma geral, especialmente quando o número de linhas de execução excede o número de núcleos de processamento. No cenário de contenção mista (figuras 4.8 e 4.11), as diferentes árvores balanceadas foram associadas aos melhores gerenciadores de contenção de acordo com os respectivos níveis de contenção. Nas figuras que representam o cenário de contenção mista, apresentamos apenas os melhores resultados para torná-las mais fáceis de interpretar, sem as poluir visualmente. Mais precisamente, apresentamos os melhores resultados para quatro linhas de execução, para quatro até oito linhas, e para oito até dezesseis linhas de execução.

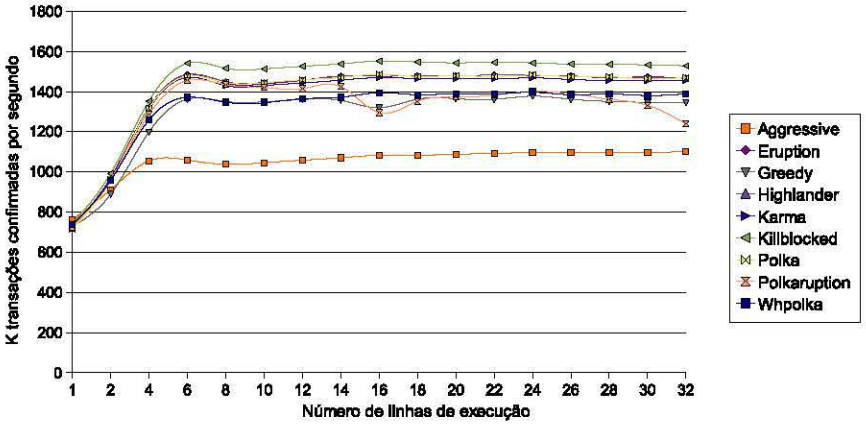


Figura 4.12: Dual Core 2 Quad, não-bloqueante, alta contenção.

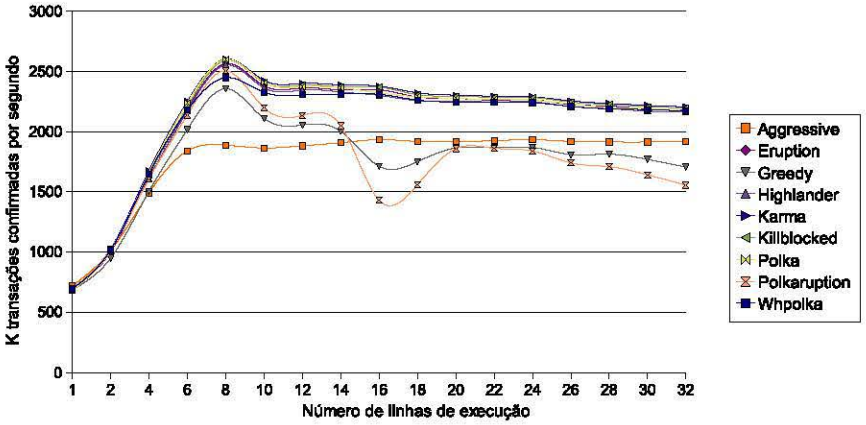


Figura 4.13: Dual Core 2 Quad, não-bloqueante, baixa contenção.

Podemos ver que, no caso do sistema com um processador Core 2 Quad, misturar diferentes gerenciadores de contenção produz um bom impacto no desempenho do *benchmark* quando este executa com quatro até oito linhas de execução, isto é, quando o número de linhas excede em pouco o número de núcleos de processamento. Com oito até dezesseis linhas, o esquema de gerenciadores mistos tem desempenho melhor que o esquema utilizando apenas um daqueles gerenciadores melhores para baixa contenção (*Killblocked*, *Karma*, *Eruption* e *Whpolka*), uma vez que o desempenho degrada de forma mais amena, mas o gerenciador *Aggressive* executa melhor que todos estes.

Notamos que conhecer o padrão de acesso às estruturas de dados não foi o suficiente para determinar os melhores gerenciadores de contenção para uso na aplicação, uma vez que os resultados mostraram alta variação nos diferentes sistemas de computação testados. Também, conhecer as diferentes configurações de *hardware* não foi suficiente porque, como podemos ver, os padrões de acesso aos dados de um programa possuem um papel importante ao decidir a melhor estratégia de gerenciamento de contenção a ser usada. Isto serve para reafirmar resultados encontrados anteriormente, que indicam que não há um gerenciador de contenção ideal a ser utilizado como uma escolha padrão [20]. Como os experimentos demonstraram, nossa implementação introduz custos adicionais de execução baixos, o que permite que seja utilizada naqueles casos em que diferentes gerenciadores de contenção executam melhor quando associados a diferentes dados ou transações. Aparentemente, nenhuma análise estática será suficiente para determinar o melhor gerenciador de contenção a ser usado em um programa, e encontrar o perfil de execução da aplicação em apenas uma configuração de *hardware* não irá funcionar para determinar o melhor gerenciador a ser utilizado se o sistema de computação mudar mesmo que apenas um pouco. A determinação dinâmica do perfil de execução do programa, bem como o ajuste dinâmico dos gerenciadores de contenção, parecem ser a resposta para o problema.

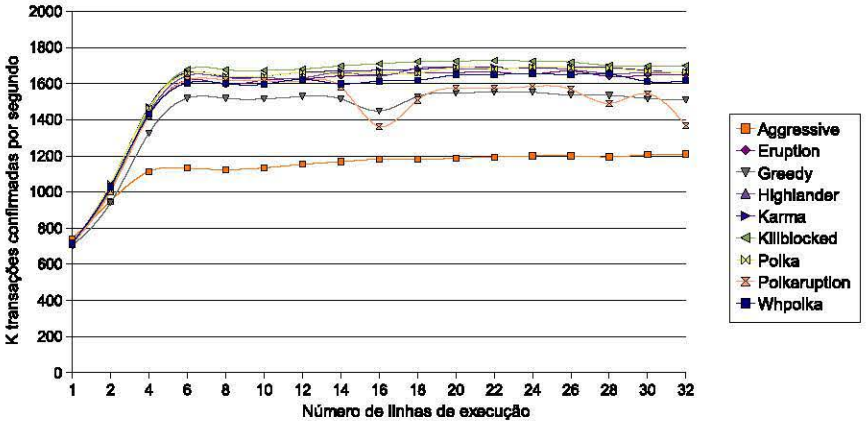


Figura 4.14: Dual Core 2 Quad, redo-lock, alta contenção.

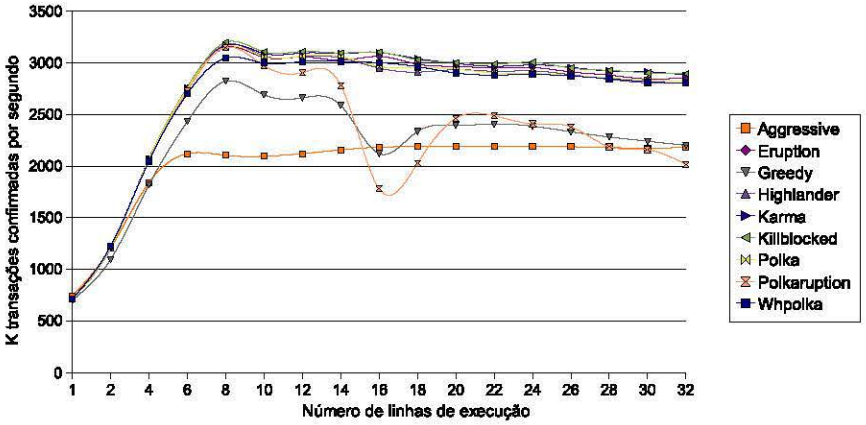


Figura 4.15: Dual Core 2 Quad, redo-lock, baixa contenção.

4.2.4 Considerações finais

Diferentes trabalhos têm sido realizados tentando encontrar bons algoritmos de propósito geral para o problema de gerenciamento de contenção, mas parece pouco provável que programas escritos com memória transacional possuem um caso geral do problema. Pelo contrário, parece mais provável que o gerenciamento de contenção, bem como outros aspectos do sistema, deverão receber um ajuste fino para alcançar bons níveis de desempenho, ao menos nos cenários que demandam níveis de desempenho bastante altos, como sistemas de processamento transacional *on-line* e bancos de dados. Em alguns casos nós poderemos permitir que o programador faça o ajuste fino de forma manual, e em outros desejaremos que o sistema o faça de forma automática.

O gerenciamento de contenção é uma área interessante para a busca pela melhoria de desempenho, uma vez que não traz impacto sobre a corretude do sistema, e existe uma grande quantidade de estratégias dentre as quais escolher. Podemos não somente querer escolher o melhor gerenciador padrão para todo o programa, mas podemos desejar escolher gerenciadores individuais que agreguem o melhor desempenho a cada uma das transações do aplicativo e a cada dado sendo acessado sob diferentes cargas de trabalho. O quão longe queremos levar a flexibilidade para a escolha de estratégias de gerenciamento depende do preço que queremos pagar por isto. Se os mecanismos que habilitam esta flexibilidade forem complicados demais, provavelmente favoreceremos alternativas mais simples.

Neste capítulo mostramos uma forma de fazer o ajuste fino do gerenciamento de contenção em programas escritos com TM, associando o gerenciador aos dados acessados no aplicativo. Mostramos que a implementação demanda custos de execução adicionais bastante baixos. Como demonstramos, o nível de concorrência no programa e os padrões de acesso aos seus dados podem ter um impacto significativo no melhor gerenciador de contenção a ser usado no aplicativo, um aspecto não investigado em trabalhos anteriores. Podemos permitir que o programador confie em um gerenciador padrão para o programa todo ou dar-lhe condições para escolher programaticamente alguma das várias estratégias, baseando-se nos padrões de acesso e variações de carga de trabalho.

Capítulo 5

Conclusões

No decorrer deste trabalho tivemos contato com diferentes propostas de implementação de memórias transacionais, com suporte de *hardware* específico e com suporte de bibliotecas de *software*. Vimos como o espaço de projeto é amplo, e que os pesquisadores ainda não chegaram a um consenso sobre as semânticas de uso de memórias transacionais e como implementá-las.

No capítulo 3 apresentamos uma plataforma de prototipagem de sistemas multiprocessados usada para o estudo, simulação e caracterização de sistemas de memórias transacionais. A infraestrutura desenvolvida foi útil para testarmos novas idéias de implementação de TM, acrescentando aninhamento de transações a uma proposta de implementação pré-existente. Ao longo do desenvolvimento da plataforma, pudemos validar e aperfeiçoar o protocolo de comunicação recentemente acrescentado à plataforma ArchC, baseado no padrão TLM (*Transaction Level Modeling*). Tivemos no entanto bastante dificuldade para desenvolver *benchmarks* para os protótipos, uma vez que o modelo de programação utilizado mapeia diretamente as novas instruções inseridas nas arquiteturas de processador, em um nível de abstração evidentemente bastante baixo.

Também tivemos a oportunidade de estudar no uma proposta de memória transacional implementada em uma biblioteca de *software*, conforme apresentado no capítulo 4. Vimos como escolher a melhor estratégia de gerenciamento de contenção pode ser bastante difícil, e apresentamos um fator que influencia na decisão. Ao fazermos variar o padrão de acesso a diferentes dados no programa, verificamos que o gerenciador de contenção mais apropriado para aqueles acessos também poderia mudar. O estudo de uma abordagem de STM também foi bastante importante para aprofundar nosso conhecimento sobre memórias transacionais, principalmente porque ainda não há consenso entre os pesquisadores sobre quais abordagems de implementação são mais apropriadas (em *hardware*, *software* ou híbrida).

As contribuições deste trabalho foram:

- a construção de uma plataforma de prototipagem e simulação que nos auxiliou na construção de dois protótipos de sistema HTM;
- a demonstração de como a modelagem voltada à simulação das instruções da arquitetura alvo (*instruction driven*) provê maior flexibilidade de modelagem de sistemas HTM permitindo ao mesmo tempo boa velocidade de simulação;
- a caracterização de uma variável usada para o ajuste fino do desempenho do nosso protótipo de HTM com suporte a aninhamento de transações;
- a modificação de uma implementação de STM visando a associação de diferentes algoritmos de gerenciamento de contenção às transações e aos diferentes dados em um programa;
- uma caracterização para o problema de gerenciamento de contenção entre transações levando em consideração os padrões de acesso aos dados em um programa paralelo.

Com o estudo de um sistema STM pudemos lidar com o problema de gerenciamento de contenção entre transações baseado nos padrões de acesso aos dados. Ficou evidenciado que a escolha do melhor gerenciador de contenção é um problema bastante difícil de ser feito na prática, e que somente com o levantamento do perfil de execução do programa no sistema de computação no qual este executará pode dar uma resposta mais próxima do ideal.

A plataforma de prototipagem desenvolvida levou à publicação de um artigo de nossa autoria, o primeiro sobre memórias transacionais publicado por nosso instituto [27].

Como trabalhos futuros, sugerimos que um compilador seja modificado para transformar as construções de alto nível exemplificadas no capítulo 2 diretamente em código de máquina contendo as instruções transacionais, no caso de sistemas HTM, ou contendo as chamadas de biblioteca, no caso de soluções STM. Desta forma poderemos concretizar a facilidade de programação que se espera de sistemas de memória transacional. Recomendamos também, como trabalho futuro, a construção de novos *benchmarks* para avaliar de forma mais profunda as idéias levantadas ao longo do desenvolvimento deste trabalho.

Entendemos que várias das restrições dos protótipos de HTM desenvolvidos podem ser atacadas, e recomendamos isto como trabalhos futuros. Por exemplo, métodos de virtualização de recursos podem ser desenvolvidos, ou a implementação de construções transacionais mais elaboradas, como o bloqueio condicional por meio de transações.

Referências Bibliográficas

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, New York, NY, USA, June 2006. ACM Press.

[2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 2008.

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, February 2005. IEEE Computer Society.

[4] C. Scott Ananian and Martin Rinard. Efficient Object-Based Software Transactions. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*. San Diego, CA, USA, October 2005. In conjunction with OOPSLA'05.

[5] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC Architecture Description Language. *International Journal of Parallel Programming*, 33(5):453–484, October 2005.

[6] Alexandro Baldassin, Paulo Cesar Centoducatte, and Sandro Rigo. Extending the ArchC Language for Automatic Generation of Assemblers. In *Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pages 60–68, Washington, DC, USA, October 2005. IEEE Computer Society.

[7] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. June 2005. In conjunction with ISCA'05.

[8] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: a High-Performance Parallel-Architecture Simulator. In *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 247–248, New York, NY, USA, June 1992. ACM Press.

[9] Brian D. Carlstrom, JaeWoong Chung, H. Chafi, Chi Cao Minh, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Transactional Execution of Java Programs. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005. In conjunction with OOPSLA'05.

[10] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, June 2006. IEEE Computer Society.

[11] H. Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. TAPE: A Transactional Application Profiling Environment. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 199–208, New York, NY, USA, June 2005. ACM Press.

[12] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In *Second ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, USA, August 2007. In conjunction with PPoPP'07.

[13] Bruno de Carvalho Albertini. Um framework de desenvolvimento de plataformas e um mecanismo de depuração baseado em reflexão computacional. Master's thesis, Universidade Estadual de Campinas, Instituto de Computação, Campinas, SP, Brazil, 2007.

[14] David Dice and Nir Shavit. What Really Makes Transactions Faster? In *First ACM SIGPLAN Workshop on Transactional Computing*. Ottawa, Ontario, Canada, June 2006. In conjunction with PPoPP'06.

[15] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Dividing Transactional Memories by Zero. In *Third ACM SIGPLAN Workshop on Transactional Computing*. Salt Lake City, UT, USA, February 2008. In conjunction with PPoPP'08.

[16] Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, January 2006.

[17] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246, New York, NY, USA, February 2008. ACM Press.

[18] Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 338–349, New York, NY, USA, June 2003. ACM Press.

[19] Keir Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[20] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 303–323, New York, NY, USA, September 2005. LNCS, Springer.

[21] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264, New York, NY, USA, July 2005. ACM Press.

[22] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software. *IEEE Micro*, 24(6):92–103, November-December 2004.

[23] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 388–402, New York, NY, USA, October 2003. ACM Press.

[24] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, June 2005. ACM Press.

[25] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, May 1993. ACM Press.

[26] IEEE. *IEEE Standard SystemC®Language Reference Manual*. IEEE Computer Society, New York, NY, USA, 2005.

[27] Fernando Kronbauer, Alexandro Baldassin, Bruno Albertini, Paulo Centoducatte, Sandro Rigo, Guido Araujo, and Rodolfo Azevedo. A Flexible Platform Framework for Rapid Transactional Memory Systems Prototyping and Evaluation. In *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping*, pages 123–129, Washington, DC, USA, May 2007. IEEE Computer Society.

[28] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, New York, NY, USA, March 2006. ACM Press.

[29] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[30] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGOPS Operating Systems Review*, 11(2):128–137, April 1977.

[31] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[32] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 354–368, New York, NY, USA, September 2005. LNCS, Springer.

[33] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, and Michael L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. June 2006. In conjunction with PLDI'06.

[34] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood.

Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, November 2005.

[35] Maurice Herlihy and Victor Luchangco and Mark Moir and William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[36] Mark Moir, Kevin Moore, and Dan Nussbaum. The Adaptive Transactional Memory Test Platform: A Tool for Experimenting with Transactional Code for Rock. In *Third ACM SIGPLAN Workshop on Transactional Computing*. Salt Lake City, UT, USA, February 2008. In conjunction with PPoPP'08.

[37] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 254–265, Washington, DC, USA, February 2006. IEEE Computer Society.

[38] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, June 2005. IEEE Computer Society.

[39] William N. Scherer III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 240–248, New York, NY, USA, July 2005. ACM Press.

[40] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, USA, August 1995. ACM Press.

[41] Marc Tremblay and Shailender Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 82–83, Washington, DC, USA, February 2008. IEEE Computer Society.

[42] Lixin Zhang. UVSIM User Manual (version 1.0). Technical Report UUCS-03-011, University of Utah, 2003. http://www.cs.utah.edu/research/techreports/.