UNIVERSIDADE FEDERAL DE PELOTAS

Centro de Desenvolvimento Tecnológico

Programa de Pós-Graduação em Computação

Dissertação

Um sistema de detecção de conflitos com invalidação mista para a linguagem CMTJava

RAFAEL DE LEÃO BANDEIRA

Pelotas, 2013

RAFAEL DE LEÃO BANDEIRA

Um sistema de detecção de conflitos com invalidação mista para a linguagem CMTJava

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal de Pelotas, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. André Rauber du Bois
Co-orientador: Prof. Dr. Maurício Lima Pilla

Pelotas, 2013

Dados Internacionais de Publicação (CIP)

B214u Bandeira, Rafael de Leão
Um sistema de detecção de conflitos com invalidação mista para a linguagem CMTJava / Rafael de Leão Bandeira; André Rauber du Bois, orientador; Maurício Lima Pilla, co-orientador. – Pelotas, 2013.
74 f.

Dissertação (Programa de Pós-Graduação em Computação), Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas. Pelotas, 2013.

1.Memórias transacionais. 2.Programação concorrente. 3.STM. I. Bois, André Rauber du, orient. II. Pilla, Maurício Lima, co-orient. III. Título.

CDD: 005.133

Catalogação na Fonte: Leda Cristina Peres Lopes CRB:10/2064
Universidade Federal de Pelotas

Banca examinadora:

Prof. Dr. Adenauer Corrêa Yamin

Prof. Dr. Gerson Geraldo H. Cavalheiro

Profª. Drª. Juliana Kaizer Vizzotto

Dedico este trabalho à minha família, em especial à minha mãe.

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.
— MARTIN FOWLER

RESUMO

BANDEIRA, Rafael de Leão. **Um sistema de detecção de conflitos com invalidação mista para a linguagem CMTJava**. 2013. 74 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

A computação paralela permitiu um considerável ganho de desempenho na execução dos programas, dividindo-os em partes discretas resolvidas concorrentemente usando múltiplos recursos computacionais. Apesar dos seus benefícios, esse paradigma aumenta a complexidade no desenvolvimento dos algoritmos, pois é necessário levar em conta vários aspectos inexistentes na codificação de programas sequenciais, como por exemplo, garantir a exclusão mútua das tarefas executadas paralelamente.

Transações de memória são unidades de execução atômica. Sua utilização permite ao programador focar em determinar onde a atomicidade é necessária, ao invés dos mecanismos necessários para garanti-la. Com essa abstração, o desenvolvedor identifica as operações que formam uma seção crítica, enquanto que o sistema transacional determina como executar aquela seção crítica isoladamente em relação aos outros fluxos de execução do programa.

CMTJava (DU BOIS; ECHEVARRIA, 2009) é uma extensão de Java para programação com memórias transacionais. Neste trabalho descreve-se a implementação de um novo sistema para gerenciamento da execução concorrente das transações na CMTJava. O sistema transacional desenvolvido utiliza uma estratégia híbrida para detecção de conflitos, conhecida como invalidação mista (SPEAR et al., 2006). Conflitos entre transações de escrita são detectados de forma adiantada e conflitos entre uma transação de leitura e outra de escrita são detectados tardiamente. Na implementação antecedente, ambos tipos de conflitos são detectados ao final da execução da transação.

De acordo com resultados obtidos através de experimentos com árvores rubro-negras, o sistema de detecção de conflitos com invalidação mista proposto obtém melhor desempenho que o trabalho anterior em cenários onde o número de conflitos é grande. Em particular, o sistema transacional desenvolvido mostrou-se mais eficiente que a implementação anterior na maioria dos cenários testados, sendo até duas vezes mais eficiente para execução de transações de busca em uma árvore rubro-negra com 50000 elementos.

Palavras-chave: Memórias transacionais, programação concorrente, STM.

RESUMO

BANDEIRA, Rafael de Leão. **A conflict detection system with mixed invalidation for the CMTJava language**. 2013. 74 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação. Universidade Federal de Pelotas, Pelotas.

The parallel computing has allowed a considerable performance gain in programs execution, dividing them into discrete parts solved concurrently using multiple computational resources. Despite its benefits, this paradigm increases the complexity in the development of algorithms, because it is necessary to take into account several aspects not present in sequential programming, such as the necessity to ensure mutual exclusion of tasks performed in parallel.

Memory transactions are atomic execution units. Its use allows the programmer to focus on determining where atomicity is required, rather than the mechanisms needed to guarantee it. With this abstraction, the developer identifies the operations that form a critical section, while the transactional system determines how to execute that critical section isolatedly from other program threads.

CMTJava (DU BOIS; ECHEVARRIA, 2009) is a Java extension for programming with transactional memory. This work describes the implementation of a new system to manage the concurrent execution of transactions. The system developed employs a hybrid conflict detection strategy, called mixed invalidation (SPEAR et al., 2006). Write/write conflits are detected eagerly and read/write conflicts are lazily detected. On the former implementation, both conflict types are detected by the end of the transaction execution.

According to results obtained in a red-black tree microbenchmark, the mixed conflict detection strategy employed yields better performance than the previous work when the number of conflicts is high. In particular, the STM system developed is more efficient than the former implementation in most cases tested, being two times more efficient for executing search transactions in a red-black tree with 50000 elements.

Palavras-chave: transactional memory, concurrent programming, STM.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

HTM	Hardware Transactional Memory
HyTM	Hybrid Transactional Memory
LSA	Lazy Snapshot Algorithm
MT	Memória Transacional
RSTM	Rochester Software Transactional Memory
STM	Software Transactional Memory
TL2	Transactional Locking II

SUMÁRIO

1 INTRODUÇÃO

A computação paralela permitiu um considerável ganho de desempenho na execução dos programas, dividindo-os em partes discretas resolvidas concorrentemente usando múltiplos recursos computacionais. Essa abordagem trouxe uma série de benefícios em relação a programação sequencial, permitindo resolver problemas cada vez mais complexos e em tempo cada vez menor.

Apesar dos seus benefícios, esse paradigma aumenta a complexidade no desenvolvimento dos algoritmos, pois é necessário levar em conta vários aspectos inexistentes na codificação de programas sequenciais. Um exemplo disso é a necessidade de garantir a exclusão mútua das tarefas executadas paralelamente, que poderiam realizar acessos à memória compartilhada com condições de corrida. O modelo de programação *multithreaded* tradicional geralmente oferece um conjunto de primitivas de baixo nível, tal como *locks*, para garantir exclusão mútua (HARRIS et al., 2007). Contudo, o uso de técnicas baseadas em *locks* tem muitas desvantagens associadas como baixa escalabilidade, dificuldade de composição de código e a possibilidade de ocorrência de *deadlocks*.

Transações substituem *locks* com unidades de execução atômica. Desta forma, o desenvolvedor pode focar em determinar onde a atomicidade é necessária, ao invés dos mecanismos necessários para garanti-la. Com essa abstração, o programador identifica as operações que formam uma seção crítica, enquanto que o sistema transacional determina como executar aquela seção crítica isoladamente em relação aos outros *threads* (HARRIS et al., 2007).

O conceito de prover suporte a transações em hardware se originou em 1986 (KNIGHT, 1986), com o objeto de aumentar o desempenho de aplicações Lisp. A ideia foi popularizada por Herlihy e Moss (HERLIHY; MOSS, 1993), que propuseram um modelo de Memória Transacional em Hardware adaptando o conceito de transação de banco de dados e aplicando-o à operações concorrentes sobre memória compartilhada. Para dar suporte à transações em hardware, eles adicionaram novas instruções ao processador e uma memória cache transacional auxiliar.

Embora a pesquisa tenha sido iniciada no nível de hardware, as memórias transacionais ganharam notoriedade quando o conceito foi transposto para o software (SHAVIT; TOUITOU, 1995). Desde então, uma vasta gama de implementações de Memória Transacional em Software foi proposta, abrangendo as mais diversas linguagens, plataformas e domínios.

Na programação utilizando essa abordagem, todo o acesso à memória compartilhada é realizado dentro de transações e todas as transações são

executadas atomicamente em relação a outras transações concorrentes. Nesse paradigma, a tarefa de escrita de programas concorrentes é substancialmente diferente se comparada com a utilização de *locks* explícitos. Não é necessário adquirir nem liberar bloqueios, ou identificar quais as operações que devem executar concorrentemente, pois ambas atribuições são responsabilidade da implementação de memória transacional.

Dessa forma, há dois mecanismos principais que tal implementação precisa possibilitar (HARRIS; LARUS; RAJWAR, 2010). Primeiramente, ela precisa gerenciar o trabalho especulativo que cada transação realiza enquanto executa. Tipicamente isto é feito ou escrevendo os valores diretamente na memória e mantendo informação sobre os valores sobrescritos ou construindo um *buffer* privado, que guarda as atualizações que a transação fará na memória em caso de efetivação.

O segundo mecanismo a ser suprido pelo sistema transacional é uma maneira de assegurar o isolamento entre as transações. Isto é, o sistema precisa detectar e resolver os conflitos de forma que a execução concorrente das transações tenha o mesmo efeito da execução de uma após a outra, em uma ordem arbitrária. Tal condição pode ser garantida identificando possíveis conflitos enquanto as transações são executadas. Também é possível permitir a execução especulativa das transações, detectando e resolvendo conflitos quando estas tentam transpor os valores computados à memória.

CMTJava (DU BOIS; ECHEVARRIA, 2009) é uma extensão que permite a programação com memórias transacionais na linguagem Java. Essa extensão possuiu um sistema transacional, responsável por encontrar e resolver conflitos entre transações, que realiza a detecção de conflitos tardia. Isso significa que as transações são executadas especulativamente até o final e então ocorre uma verificação quanto a validade dessa execução. Essa abordagem é favorecida em aplicações com poucos conflitos e com transações pequenas. Em contrapartida, o desempenho do sistema tende a diminuir quando o número de conflitos é grande, visto que esses conflitos são detectados apenas ao final da execução do código da transação.

1.1 Objetivos do trabalho

O objetivo principal desse trabalho é a implementação de um novo sistema transacional para a CMTJava, responsável pela execução de transações e gerenciamento da concorrência. Esse novo sistema utiliza uma estratégia híbrida para detecção de conflitos, denominada invalidação mista (SPEAR et al., 2006). Neste esquema, a detecção de conflitos entre transações de escrita é feita de forma adiantada. Como nesse caso tipicamente é necessário abortar uma das transações, evita-se assim que a transação conflitante seja executada até o fim, para só então ser abortada. Já os conflitos entre uma transação de leitura e outra de escrita são detectados tardiamente. Isto permite maior paralelismo na execução, uma vez que esse tipo de conflito pode muitas vezes ser resolvido sem cancelamentos. No sistema transacional anterior, os conflitos são sempre detectados de forma tardia.

Também destacam-se como objetivos secundários:

- Realizar uma revisão bibliográfica do estado da arte em memórias

transacionais e, em particular, das estratégias de implementação utilizadas nos sistemas.

- Implementar em CMTJava um *benchmark* bastante utilizado para avaliar e comparar memórias transacionais: a árvore rubro-negra.
- Realizar testes para aferir o *speedup* obtido com o novo sistema e também comparar o presente trabalho com o sistema anterior.

1.2 Organização do Trabalho

O trabalho encontra-se dividido da seguinte maneira: no Capítulo 2 discute-se a programação de computadores *multicore* e os problemas associados ao uso de bloqueios. As memórias transacionais são destacadas como abstração de alto nível para implementar a sincronização e garantir exclusão mútua entre os fluxos de execução. Além disso, são apresentados os modelos de memória transacional e as construções básicas de programação usando transações: bloco atômico, *retry* e *orElse*.

No Capítulo 3 são descritos os principais requisitos de implementação de memórias transacionais, isto é, as possíveis opções de projeto de forma a garantir o isolamento e a atomicidade das transações. As estratégias de implementação discutidas são: controle de concorrência, versionamento de dados, detecção de conflitos, granularidade de conflitos e gerenciamento de contenção.

As principais características da linguagem CMTJava são relatadas no Capítulo 4. Posteriormente, no Capítulo 5, listam-se importantes implementações de memória transacional em software, são elas: Transactional Locking II, TinySTM, RSTM e SwissTM.

O sistema transacional desenvolvido é caracterizado em detalhes no Capítulo 6. Além de apresentar as características de implementação também são mostrados os resultados obtidos através de testes de desempenho realizados. Finalmente, no Capítulo 7 são apresentadas as conclusões desta Dissertação de Mestrado.

2 PROGRAMAÇÃO *MULTICORE*

As memórias transacionais surgiram como um novo modelo para o controle de concorrência na programação de máquinas *multicore*, superando muitas das dificuldades encontradas no uso de *locks*. Nessa abordagem, o acesso à memória compartilhada é feito em transações que executam de forma atômica em relação a outras transações concorrentes. O programador apenas precisa identificar e delimitar regiões críticas e o controle do acesso dessas fica por conta do sistema transacional.

Nesse capítulo são apresentados os conceitos básicos sobre programação concorrente na Seção 2.1. A Seção 2.2 versa sobre as dificuldades inerentes à programação com mecanismos baseados em exclusão mútua, os quais ainda representam o estado da arte para sincronização da execução concorrente.

Posteriormente, é introduzido o conceito de memórias transacionais na Seção 2.3, destacando as facilidades providas pela utilização desse mecanismo em detrimento aos bloqueios. Os principais modelos de memória transacional são descritos na Seção 2.4. Por fim, as construções transacionais básicas são mostradas na Seção 2.5. A Seção 2.6 contém as considerações finais deste capítulo.

2.1 Programação de Máquinas *Multicore*

Nos últimos anos, devido a limitações físicas na fabricação de semicondutores mais poderosos, o foco da indústria de processadores voltou-se das arquiteturas monoprocessadas para as providas de mais de um núcleo de processamento, as arquiteturas *multicore* (multinúcleo).

Um programa que é executado por apenas um fluxo de execução é chamado de programa sequencial. Nesse caso, existe somente um fluxo de controle durante a execução. Um programa concorrente é executado simultaneamente por diversos fluxos de execução que cooperam entre si, trocando dados ou realizando algum tipo de sincronização. Os fluxos de execução também são chamados *threads*.

Quanto à comunicação, os sistemas concorrentes podem ser divididos como de *memória compartilhada* ou de *troca de mensagem*. Nos sistemas de memória compartilhada, fluxos de execução interagem por meio de áreas de memória mutuamente acessíveis. Na abordagem de troca de mensagem, os processos interagem enviando e recebendo mensagens. Em um programa paralelo, múltiplos *threads* executam paralelamente dentre os processadores disponíveis.

2.2 Problemas no Uso de Bloqueios

O paradigma de programação concorrente é mais complexo que o sequencial, uma vez que além dos erros que aparecem em programas sequenciais podem ocorrer erros associados com as interações entre os processos. O mecanismo mais usado para o controle do acesso a seções críticas é o bloqueio.

Os bloqueios (*locks*) são usados para a implementar a sincronização entre os fluxos de execução de um programa concorrente. *Locks*, de forma sucinta, são variáveis booleanas que aceitam duas operações: *lock* e *unlock*. A operação de *lock* altera atomicamente o valor da variável booleana para verdadeiro. Caso a variável já possua o valor verdadeiro antes da alteração, a operação deve esperar o valor se tornar falso. Já a operação de *unlock* atribui incondicionalmente o valor falso à variável booleana.

No entanto, a utilização de técnicas baseadas em *locks* tem muitas desvantagens associadas como por exemplo (HERLIHY; MOSS, 1993; HARRIS; LARUS; RAJWAR, 2010):

Inversão de prioridade: processos de maior prioridade podem ser impedidos de executar pois um processo de menor prioridade possui um *lock* comum.

***Convoiyng*:** processo detentor de um *lock* pode perder o processador, fazendo com que outros processos aptos a executar fiquem bloqueados.

***Deadlock*:** pode ocorrer quando processos tentam adquirir o mesmo conjunto de *locks* em diferentes ordens, cada qual ficando bloqueado esperando para adquirir os demais bloqueios.

***Livelock*:** similar ao *deadlock*, exceto que os estados dos processos envolvidos mudam constantemente em função uns dos outros, mas nenhum progride.

***Starvation*:** ocorre quando um processo não consegue obter acesso a um recurso compartilhado e desse modo fica inabilitado de progredir.

Apesar de implementar corretamente o controle de acesso a dados compartilhados, a utilização de *locks* apresenta uma série de dificuldades ao programador como por exemplo (JONES, 2007):

- Adquirir poucos *locks*: é fácil esquecer de adquirir algum *lock* e assim permitir que dois *threads* entrem na mesma região crítica e modifiquem a mesma posição de memória.

- Adquirir muitos *locks*: além de inibir a concorrência a aquisição de bloqueios em excesso pode causar um *deadlock*.

- Adquirir o *lock* errado: a conexão entre o *lock* e o dado que ele protege geralmente existe somente na cabeça do programador, que pode adquirir ou liberar os *locks* errados.

- Adquirir *locks* na ordem errada: na programação baseada em bloqueios além de adquirir os *locks* certos, deve-se fazê-lo na ordem correta para evitar-se *deadlocks*.

Além dos problemas acima citados, há uma série de outras dificuldades características da utilização de bloqueios. Há um paradoxo na programação usando bloqueios no que diz respeito à granularidade do *lock*, ou seja, o tamanho da região protegida por um bloqueio. Grandes regiões de código protegidas por *locks*, reduzem o paralelismo, ocasionando gargalos seriais. Esses gargalos ocorrem quando um fluxo de execução tenta acessar uma região crítica que já está sendo acessado por outro *thread*. Nesse caso, esse fluxo deve esperar o outro liberar o bloqueio para acessar a região crítica. Porém, quando se reduz o tamanho da seção que cada bloqueio protege, tem-se um aumento de dificuldade na escrita do código e também uma maior possibilidade da ocorrência de erros como os descritos anteriormente.

Outro problema decorrente do uso de bloqueios é a dificuldade no reuso de código. O reuso de código é um artifício da Engenharia de Software que visa a criação de código a partir de blocos já implementados e testados, na composição de blocos mais complexos. Contudo, o código usando bloqueios não é reusável, ou seja, mesmo implementações corretas, quando combinadas, podem levar o programa a um estado inconsistente. Para remover esta inconsistência, o programador precisa sincronizar a nova operação com as operações existentes. Isso além de demandar conhecimento de detalhes do código já implementado, pode originar *deadlock* caso a alteração realizada faça com que diferentes bloqueios sejam adquiridos em ordem inversa (RIGO; CENTODUCATTE; BALDASSIN, 2007).

2.3 Memórias Transacionais

Para contornar as dificuldades impostas pelo uso de bloqueios, foram desenvolvidas estratégias de controle de concorrência que não requerem a explicitação da exclusão mútua para o compartilhamento de recursos, chamadas não bloqueantes. Os algoritmos não bloqueantes podem ser classificados de acordo com seu tipo de garantia de progresso (FRASER; HARRIS, 2007):

- *Obstruction-freedom* é a garantia mais fraca: um *thread* realizando uma operação fará progresso enquanto não competir com outro *thread* para acessar algum dado.

- *Lock-freedom* adiciona a necessidade do sistema como um todo fazer progresso, mesmo se houver competição.

- *Wait-freedom* agrega a necessidade de que todos os *threads* farão progresso, mesmo que haja competição por recursos.

Apesar da sincronização *wait-free* ser a ideal, os mecanismos *lock-free* são suficientes para a grande maioria dos casos práticos. Memória transacional (MT) é uma técnica proposta para realizar sincronização *lock-free* tão eficiente e fácil de usar como meios convencionais baseados em exclusão mútua (HERLIHY; MOSS, 1993).

A principal abstração usada na escrita de programas concorrentes usado MT é a transação. O conceito de transação é proveniente dos sistemas de banco de dados, onde é aplicado há bastante tempo. No contexto de memória

compartilhada, uma transação é uma sequencia de instruções, incluindo leituras e escritas à memória, que satisfaz duas propriedades:

Atomicidade Quando uma transação termina sua execução, ou ela é efetivada (*commit*), fazendo com que todas as alterações realizadas na memória fiquem visíveis aos outros processos, ou ela é cancelada (*abort*), de modo que qualquer valor parcialmente computado seja descartado.

Isolamento Transações parecem executar serialmente quando possuem dados em comum, pois transações concorrentes não observam o estado intermediário de outras. Isto é, o resultado da execução de diversas transações concorrentemente equivale ao resultado da execução destas em uma ordem serial qualquer.

Na programação utilizando essa abordagem, todo o acesso à memória compartilhada é realizado dentro de transações e estas são executadas atomicamente em relação a transações concorrentes. A principal vantagem na programação de memórias transacionais é que o programador apenas delimita as seções críticas. Dessa forma não é necessário preocupar-se com a aquisição e liberação de *locks*, tarefa que se não for realizada de forma correta pode levar a problemas como *deadlocks*. O trabalho de realizar essa sincronização fica a cargo do sistema de memória transacional.

Outra importante vantagem da programação com MT é a composabilidade. Uma série de operações atômicas pode ser combinada, de forma que o resultado ainda será atômico. Isso não é válido para abstrações baseadas em *locks*. Mesmo utilizando dois trechos de código corretamente implementados com bloqueios, não é possível garantir que quando combinados o programa se manterá consistente.

Além de ser mais fácil de compor, o código transacional é mais escalável. A razão disto é a possibilidade de executar paralelamente transações que modificam partes disjuntas de uma mesma estrutura de dados ou até realizar operações de leitura em um mesmo dado concomitantemente.

2.4 Modelos de Memória Transacional

Existem diversos modelos de memória transacional. O modelo de Memória Transacional em Hardware (*HTM - Hardware Transactional Memory*) baseia-se no suporte arquitetural do processador para executar transações. Nesse modelo, o hardware é quem gerencia o versionamento de dados e os conflitos. Nas transações em hardware, os dados são armazenados nos registradores ou na memória cache, e esse dados são escritos na memória principal somente depois das transações efetivarem.

No modelo de Memória Transacional em Software (*STM - Software Transactional Memory*) o sistema de execução transacional é totalmente implementado em software. Os sistemas de STM são o foco deste trabalho. No entanto, grande parte do que é discutido contemplando transações em software também é válido para os outros modelos.

Além destas duas abordagens, existem também abordagens mistas. Memória Transacional Híbrida (*HyTM - Hybrid Transactional Memory*) suporta

a execução da HTM mas torna ao software quando os recursos de hardware são esgotados. Já a Memória Transacional em Software assistida por Hardware (*HaSTM - Hardware-assisted Software Transactional Memory*) combina STM com algum suporte arquitetural em hardware para acelerar partes da implementação em software (HARRIS et al., 2007).

2.5 Construções Transacionais Básicas

Como já discutido na Seção 2.2, na computação paralela é necessário a utilização de algum mecanismo para realizar a exclusão mútua no acesso a um dado compartilhado. Além disso, muitas vezes é necessário realizar a sincronização entre os diversos fluxos de execução constituintes de um programa, isto é, definir a ordem na qual os *threads* são executados.

Apesar dos mecanismos baseados em bloqueios possibilitarem a realização dessas tarefas, devido ao baixo nível de abstração característicos destes, o estudo de transações, assim como de outras abstrações de programação paralela, tem ganhado espaço.

Utilizando transações, ao invés de realizar explicitamente o controle de concorrência necessário para o acesso às seções críticas ou sincronização dos *threads*, utilizam-se construções transacionais. A seguir são descritas as principais construções transacionais, são elas: Bloco Atômico, Retry e OrElse.

2.5.1 Bloco Atômico

O comando `atomic`, ou em algumas linguagens `atomically`, serve para delimitar um bloco de código que deve ser executado atomicamente em relação a todos os outros blocos atômicos do programa. A Figura 1 demostra a definição de um bloco atômico que acessa as variáveis compartilhadas `x` e `y`. Uma outra possibilidade é a definição de funções ou métodos atômicos, como na Figura 2, cujo o corpo executa em uma instrução atômica implícita (HARRIS; LARUS; RAJWAR, 2010).

```
1 void Foo() {
2   atomic {
3     if (x != null)
4       x.foo();
5     y = true;
6   }
7 }
```

Figura 1: Exemplo de utilização de um bloco atômico

A principal distinção entre o bloco atômico e outras construções de programação concorrente, como monitores, é o fato de não ser necessário explicitar o recurso compartilhado a ser acessado. O bloco atômico sincroniza-se implicitamente com os outros blocos atômicos que acessam os mesmos dados.

No momento da execução, tem-se a impressão de que as transações estão sendo executadas de forma sequencial, mas na verdade os blocos atômicos são executados concorrentemente, usando algum esquema de sincronização entre

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

Figura 2: Outra sintaxe para definição de um bloco atômico

as transações. Na execução do bloco atômico as propriedades de atomicidade e isolamento são sempre garantidas. Para isso, o sistema de execução da MT conta com um esquema para detectar e resolver conflitos que venham a ocorrer na execução especulativa das transações (Seção 3.3). Esse mecanismo de detecção também necessita de alguma técnica de versionamento, seja para guardar o estado da memória antes do início da transação ou para conter os valores computados pela mesma (Seção 3.2).

2.5.2 Retry

Quando transações acessam um dado de forma conflitante, o sistema de tempo de execução detecta o conflito e resolve-o. Isto é feito geralmente abortando uma das transações e reexecutando-a, possivelmente após algum período de espera. Caso outro conflito ocorra, o sistema transacional realizará novamente esse processo, até que a execução da transação ocorra sem conflitos. No entanto, todo esse processo ocorre geralmente de forma transparente ao programador.

Contudo, há também uma maneira de fazer com que uma transação seja cancelada e reexecutada pelo código. A primitiva `retry` bloqueia uma transação, fazendo com que essa seja abortada e executada novamente. A transação somente será reexecutada quando algum dado associado a ela seja modificado por outro *thread*.

O exemplo da Figura 3 apresenta um trecho de código que realiza a remoção de um elemento de um `buffer`.

```
public int remover() {
   atomic {
      if (itens == 0)
         retry();
      itens--;
      return buffer[itens];
   }
}
```

Figura 3: Remoção bloqueante de um elemento de um *buffer*

Nesse método, utilizou-se a construção `retry` para implementar uma operação bloqueante de remoção. Quando o *buffer* estiver vazio, a transação chama `retry` e fica esperando até que um outro *thread* insira um elemento no *buffer*, para então tentar novamente a remoção.

2.5.3 OrElse

A construção `orElse` permite que transações sejam compostas como alternativas, sendo que somente uma transação será executada. Por exemplo, uma chamada para `orElse t1 t2` tentará primeiro executar `t1`, se `t1` chamar `retry` então `t2` será executada. Caso `t2` também chame `retry`, então todo o bloco atômico será novamente executado. Se `t1` terminar sua execução normalmente, `t2` não será executada.

Na Figura 4 foi utilizada a construção `orElse` para realizar a remoção de um elemento dentre dois *buffers*. O método `remover`, mostrado na seção anterior, chama *retry* caso o *buffer* esteja vazio. Dessa forma, esse trecho de código tentará a remoção de um elemento do `buffer1` e caso essa operação chame `retry`, tentará a remoção do `buffer2`. Caso o segundo *buffer* também esteja vazio, o bloco atômico ficará bloqueado até que um dos *buffers* seja modificado, para então novamente tentar a remoção.

```
1 atomic {
2   { x = buffer1.remover(); }
3 orElse
4   { x = buffer2.remover(); }
5 }
```

Figura 4: Remoção de um elemento dentre dois *buffers*

2.6 Considerações Finais

Neste capítulo foi discutida a programação de computadores multinúcleo. Para explorar a potencialidade dessas arquiteturas, é necessário escrever software que realize a sincronização das atividades concorrentes. Posteriormente, foram listados alguns dos principais problemas dos mecanismos baseados em exclusão mútua e apresentada as transações como alternativa capaz de facilitar a escrita de programas concorrentes.

As transações fornecem uma alternativa que retira do programador o encargo de lidar com detalhes de sincronização. O resultado disso é a escrita de programas concorrentes de forma mais rápida e com menos erros. Além do mais, o código gerado é mais fácil de ler, compor, depurar e também tende a ser mais escalável.

No próximo capítulo serão discutidas as estratégias de implementação de STMs. Estas estratégias ditam como o sistema transacional executa garantindo a atomicidade e o isolamento das transações. A escolha destas características está diretamente ligada ao desempenho das aplicações executadas.

3 ESTRATÉGIAS DE IMPLEMENTAÇÃO

No capítulo anterior demonstrou-se que as transações tem a capacidade de facilitar o desenvolvimento de programas concorrentes. Em contrapartida, para fornecer essas facilidades ao programador, as tarefas de sincronização da execução que seriam trabalho do desenvolvedor no caso do uso de outros mecanismos, como os de exclusão mútua, devem ser realizadas de forma transparente pela implementação da MT. Com isso, é necessário que o sistema de memória transacional gerencie a execução das transações escondendo os detalhes de sincronização do programador e ainda assim garantindo que as propriedades de atomicidade e isolamento sejam respeitadas.

Mesmo para um simples sistema de memória transacional, diversas alternativas de implementação são possíveis. Dentre os muitos requisitos de implementação, neste capítulo são discutidos os mais relevantes no desempenho geral do sistema, que são: Controle de Concorrência, Versionamento de Dados, Detecção de Conflitos, Granularidade de Conflitos e Gerenciamento de Contenção.

3.1 Controle de Concorrência

Um sistema de memória transacional requer sincronização para coordenar o acesso concorrente a dados. Essa sincronização é realizada com base em três eventos: ocorrência, detecção e resolução de conflitos. Um conflito ocorre quando duas transações realizam uma operação conflitante em uma mesma porção de dados. O conflito é detectado quando o sistema de MT subjacente descobre que um conflito ocorreu, e resolvido quando o próprio sistema ou código da transação realiza alguma ação para eliminar o conflito. Tipicamente, a resolução é feita atrasando ou abortando a execução de uma das transações conflitantes.

A ocorrência, detecção e resolução de um conflito pode ocorrer em diferentes momentos, mas não em uma ordem diferente, a menos que o sistema preveja ou antecipe conflitos (HARRIS; LARUS; RAJWAR, 2010). De maneira geral, existem duas abordagens de controle de concorrência:

Controle de concorrência pessimista: todos os três eventos ocorrem no mesmo ponto da execução. Quando uma transação vai realizar um acesso à memória, o sistema já adquire os bloqueios necessários para esse acesso, detectando e resolvendo eventuais conflitos. Esse tipo de controle de concorrência permite às transações garantir a posse do dado antes

de prosseguir, pois os bloqueios já foram adquiridos, impedindo outras transações de acessá-lo.

Controle de concorrência otimista: a detecção e resolução de um conflito pode ocorrer após a ocorrência. Esse tipo de controle de concorrência permite múltiplas transações acessarem dados concorrentemente e progredirem a execução mesmo na ocorrência de conflitos, contanto que o sistema transacional detecte e resolva esse conflitos antes da efetivação da transações.

Ambas formas de controle de concorrência demandam cuidado na implementação para assegurar que as transações progridam. A implementação do controle de concorrência pessimista deve evitar a ocorrência de *deadlocks*, por exemplo, se uma transação `T` detém um *lock* `L1` e necessita um *lock* `L2`, enquanto que uma transação `U` detém `L2` e solicita `L1`. Isso pode ser feito através da aquisição de bloqueios em uma ordem predeterminada e fixa, usando temporização ou alguma outra técnica de detecção de *deadlock* dinâmica.

A implementação do controle de concorrência otimista pode levar a *livelocks*. Por exemplo, se uma transação `T` escreve em `X`, esta pode conflitar com uma transação `U` e forçar o cancelamento desta. Portanto `U` pode reiniciar, escrever em `X`, forçando o cancelamento de `T`. Uma solução possível para evitar esse problema é utilizar um gerenciador de contenção (Seção 3.5) para atrasar a reexecução de transações em face de conflitos. Outra alternativa é garantir que uma transação é abortada somente diante de um conflito com uma transação que já foi de fato efetivada (HARRIS; LARUS; RAJWAR, 2010).

Se conflitos são frequentes, o controle de concorrência pessimista tende a ser mais efetivo. Já no caso em que conflitos são raros, o controle de concorrência otimista é geralmente mais rápido devido ao menor *overhead* com aquisição de bloqueios, além de aumentar a concorrência entre transações.

3.2 Versionamento de Dados

O versionamento de dados controla o armazenamento simultâneo dos novos dados, que serão visíveis caso a transação seja efetivada, e os dados antigos, guardados para o caso da transação abortar. Como no máximo um desses conjuntos de dados pode ser armazenado diretamente na memória, é necessário guardar o outro conjunto de dados paralelamente (MOORE et al., 2006). Esse versionamento é feito, basicamente, de duas formas:

- Versionamento adiantado (*eager versioning*) no qual a transação modifica o dado diretamente na memória e mantém um *undo-log* que guarda os valores que foram sobrescritos. Esse *log* é usado para restaurar o conteúdo inicial da memória, caso a transação venha a abortar. O versionamento adiantado implica o uso de controle de concorrência pessimista, pois, como os valores são escritos diretamente na memória, é necessário garantir a exclusão mútua desta atualização.

- Versionamento tardio (*lazy versioning*) onde a transação guarda os valores a serem escritos na memória em um *redo-log* privado, e as leituras primeiramente verificam esse *log* para garantir o isolamento da transação.

Ao terminar seu trabalho, ou a transação é efetivada e atualiza a memória a partir dos valores do *log*, ou aborta e simplesmente descarta o *redo-log*.

A Figura 5 ilustra o versionamento de dados adiantado, onde um *thread* atribui `15` a variável de memória `X`, que antes do início da transação continha o valor `10`. Cada parte da figura mostra um possível estado da transação. Em (a) é representado o estado inicial, antes da execução da transação. Em (b) a transação executa atribuindo a `15` à variável `X`. Com o versionamento adiantado, essa atribuição é realizada diretamente na memória e o valor anterior da variável compartilhada `X` é posto no *undo-log*. No cenário (c), que caracteriza uma situação de efetivação desta transação, apenas descarta-se o valor contido no *log*, pois a memória já encontra-se atualizada. Já em (d), onde é representado o cancelamento da transação, o valor contido no *undo-log* é primeiramente restaurado à memória, para então ser descartado.

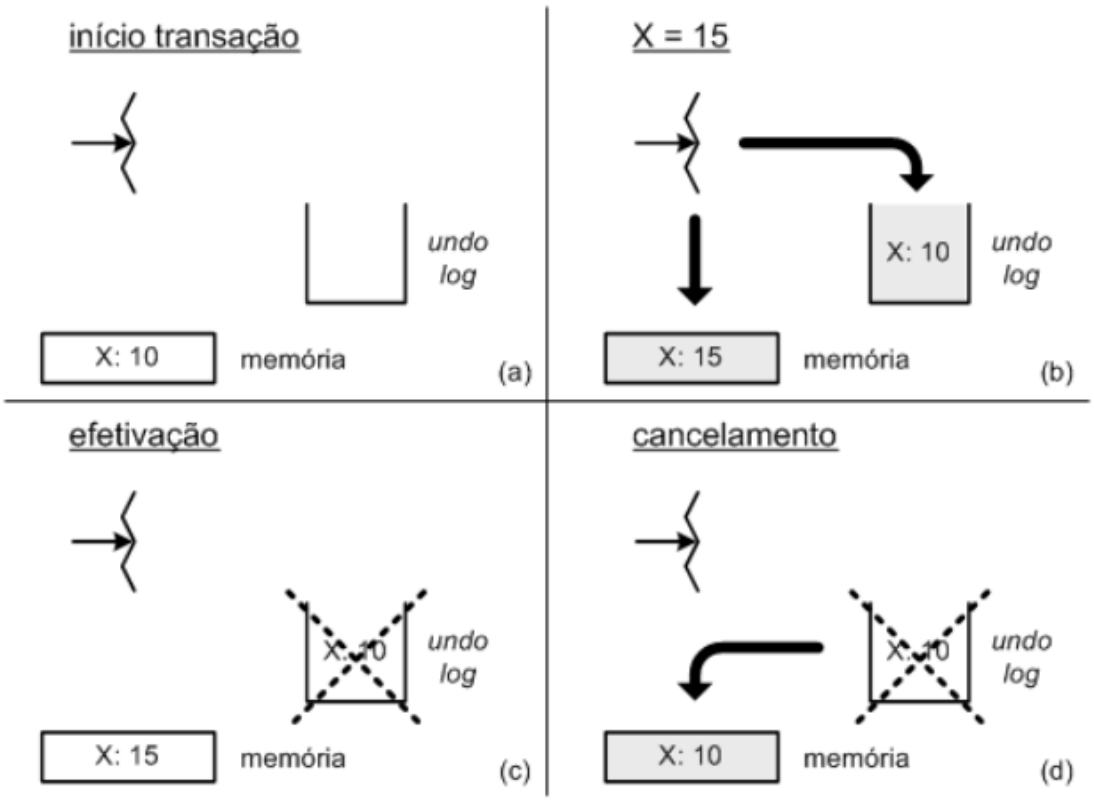

Figura 5: Versionamento de dados adiantado (RIGO; CENTODUCATTE; BALDASSIN, 2007)

Semelhantemente, na Figura 6 é caracterizada a mesma operação, utilizando versionamento de dados tardio. Em (a) é representado o estado inicial, antes da execução da atribuição do valor `10` à variável `X`. Quando a transação é executada em (b), o versionamento tardio escreve a modificação no *redo-log* e nada altera na memória principal. Em (c), no caso de efetivação desta transação, a memória é atualizada com o valor contido no *redo-log*, que é posteriormente descartado. No caso de cancelamento (d), não é necessária nenhuma operação a não ser o descarte do *log*.

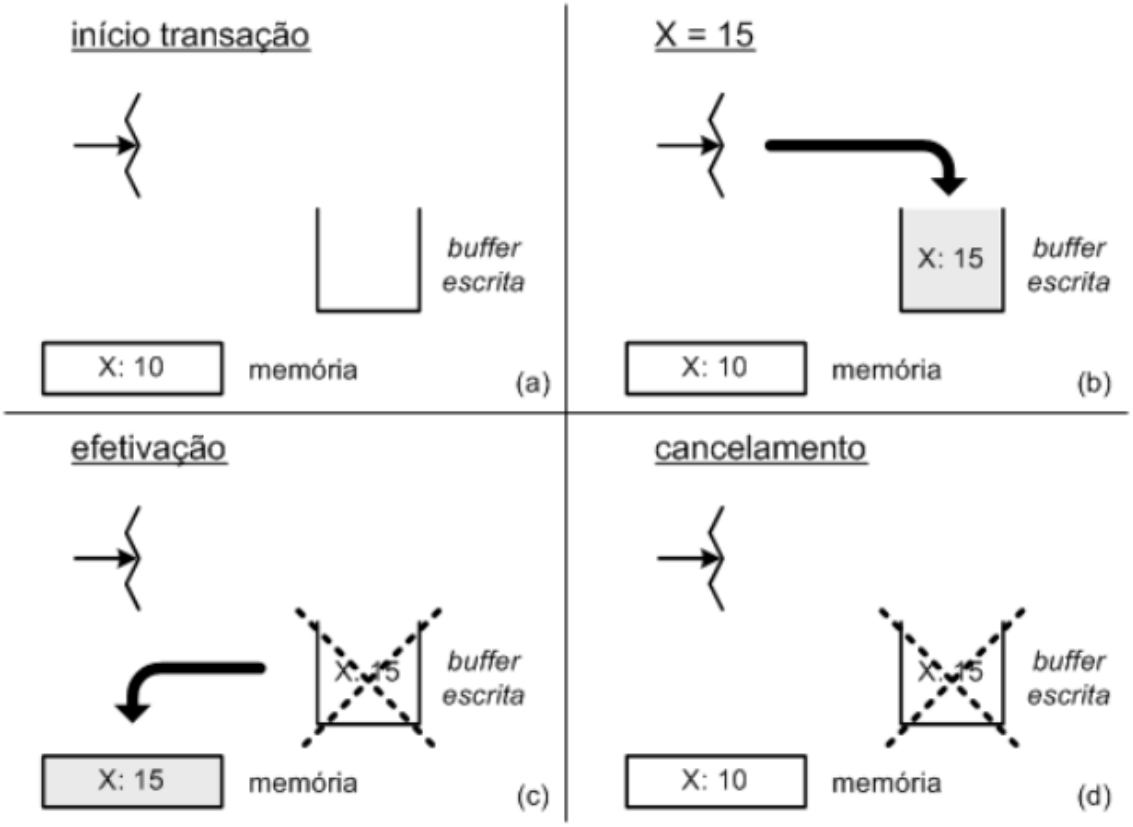

Figura 6: Versionamento de dados tardio (RIGO; CENTODUCATTE; BALDASSIN, 2007)

O versionamento adiantado é mais eficiente do que o versionamento tardio em caso de *commit*. A razão disso é que nesse mecanismo, quando uma transação é efetivada, os valores já estão na memória principal. O versionamento tardio guarda os valores modificados em um *log*, e precisa atualizar a memória no momento da efetivação. No entanto, o versionamento tardio é mais eficiente numa situação de cancelamento, uma vez que neste tipo de política de gerenciamento de versões se faz necessário apenas descartar o *redo-log* e reiniciar a transação com um *log* novo.

3.3 Detecção de Conflitos

Outro importante conceito nos sistemas de MT é a detecção de conflitos, responsável por assegurar o isolamento das transações. Para garantir a serialização, uma solução simples seria permitir a execução de somente uma transação num dado instante. No entanto, tal comportamento não exploraria nenhum paralelismo.

A maioria dos sistemas de memória transacional necessita de um mecanismo para manter controle das posições de memória que uma transação leu ou escreveu. No mecanismo mais utilizado, cada transação armazena os endereços acessados em um conjunto de leitura (*read set*) e em um conjunto de escrita (*write set*), de acordo com o tipo de acesso à memória. A detecção de conflitos

indica uma sobreposição entre o conjunto de escrita de uma transação com o conjunto de escrita ou de leitura de outra transação concorrente (RIGO; CENTODUCATTE; BALDASSIN, 2007).

De forma análoga ao versionamento, a detecção de conflitos pode ser realizada de maneira adiantada (*eager conflict detection*) ou tardia (*lazy conflict detection*). Além disso, dada uma transação em execução qualquer, os conflitos podem ser detectados em relação as demais transações em execução num dado momento, ou somente entre as transações já efetivadas, configurando os seguintes cenários (HARRIS; LARUS; RAJWAR, 2010):

Detecção adiantada entre transações em execução: nesse caso, conflitos são detectados quando duas transações modificam um mesmo dado. A resolução dos conflitos deve impedir *livelocks*, por exemplo, utilizando gerenciamento de contenção.

Detecção adiantada em relação a transações efetivadas: nessa abordagem, as transações procedem concorrentemente e conflitos são percebidos quando uma delas tenta efetivar. Situações de *livelock* são facilmente evitadas usando a política "*committer wins*", isto é, a transação que está em trabalho de efetivação tem a prioridade quanto a ocorrência de conflitos, e as demais transações conflitantes são abortadas.

Detecção tardia entre transações em execução: transações que competem no acesso a um dado executam concorrentemente. Quando alguma tenta efetivar o conflito é então detectado e resolvido de alguma maneira, em geral esta resolução fica a cargo do gerenciador de contenção.

Detecção tardia em relação a transações efetivadas: nesse caso, uma transação pode continuar executando mesmo que outra tenha efetivado na memória um valor conflitante. Como consequência, a primeira transação vai desperdiçar o trabalho realizado, pois está condenada a abortar.

Utilizando controle de concorrência pessimista a detecção de conflitos é direta. A razão disso é que o *lock* que protege a área de memória sendo acessada pode ser adquirido somente quando não estiver em posse de outro *thread*. Com controle de concorrência otimista, uma grande variedade de técnicas para detecção de conflitos pode ser empregada. Nesse sistemas, geralmente utiliza-se uma operação de validação, na qual a transação corrente verifica caso conflitos tenham ocorrido.

A detecção adiantada encontra conflitos antes, reduzindo o trabalho desperdiçado por transações conflitantes. Em contrapartida, alguns falsos conflitos são encontrados. Inversamente, a detecção tardia é mais otimista, no sentido que deixa as transações executarem até o final para detectar conflitos. Dessa forma, alguns conflitos são resolvidos sem cancelamentos. Quando isso não é possível, o desperdício de trabalho é maior comparado a detecção adiantada.

A seguir é ilustrada a desvantagem da detecção adiantada. A Figura 7 apresenta dois pares de transações (T e U) que acessam uma variável compartilhada X. Em ambos cenários, a detecção adiantada encontraria um conflito na leitura de X por T, uma vez que a transação U escreveu anteriormente

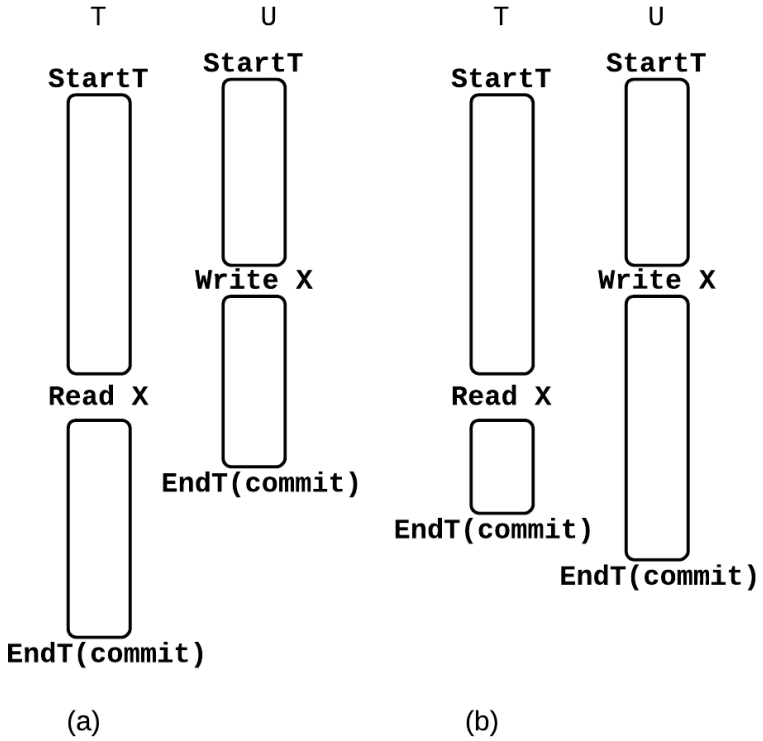

Figura 7: Execução de dois pares de transações. Tanto a detecção adiantada quanto tardia encontraria um conflito no primeiro par (a). No segundo par (b), a detecção adiantada encontraria um conflito e a tardia permitiria a efetivação das duas transações (HARRIS et al., 2007)

nesta variável. A abordagem tardia também detectaria um conflito no caso (a), pois a transação U faz o *commit* antes, implicando que T deveria ter usado o resultado dessa operação de escrita. Entretanto, para o caso descrito em (b), a detecção tardia permitiria a efetivação de ambas transações. Isto ocorre pois T faz o *commit* antes e dessa forma a leitura realizada não utiliza o resultado da operação `write X`, efetuada pela transação U.

A desvantagem da detecção tardia é apresentada na Figura 8. Utilizando detecção tardia, a transação T despende o tempo compreendido entre t_2 (efetivação de U) e t_3 (efetivação de T) realizando um trabalho que é condenado a ser abortado. Isto porque dado a efetivação de U, a execução de T não é mais válida. Com a técnica de detecção de conflitos adiantada, quando T tenta escrever em X (t_1) percebe que o atributo já encontra-se em poder de outra transação e aborta imediatamente, evitando o execução desnecessária do restante da transação.

Uma dificuldade encontrada na detecção adiantada é a eliminação de sucessivos conflitos entre um conjunto de transações. Isso requer que a memória transacional antecipe a decisão de qual das transações envolvidas em um conflito tem mais probabilidade de ser efetivada no futuro, o que não é uma tarefa trivial.

Uma abordagem interessante que extrai as vantagens das estratégias anteriores é a invalidação mista (*mixed invalidation*) (SPEAR et al., 2006). Nessa estratégia os conflitos de escrita/escrita são detectados de forma adiantada, pois

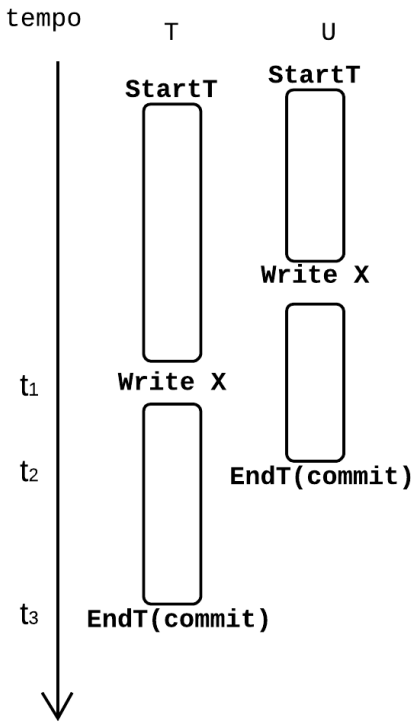

Figura 8: Desvantagem da detecção tardia

no máximo uma transação pode conseguir efetivar dada a ocorrência desse tipo de conflito. Conflitos de leitura/escrita são detectados tardiamente, visto que ambas transações podem efetivar desde que a transação de leitura o faça primeiro.

Outra abordagem híbrida bastante interessante, é descrita por WELC; JAGANNATHAN; HOSKING (2004). Sob baixa contenção, conflitos são detectados de forma adiantada, usando versionamento de dados adiantado. Em situações de alta contenção, passa-se a utilizar detecção de conflitos tardia, usando versionamento de dados tardio.

3.4 Granularidade de Conflitos

A granularidade de detecção de conflitos diz respeito a unidade de armazenamento na qual um sistema transacional detecta a ocorrência de conflitos. Em outras palavras, necessita-se associar algum metadado de controle de concorrência com as posições de memória que o programa acessa. Esse metadado pode ser um *lock* ou uma estrutura mais complexa, contendo por exemplo um número de versão ou um ponteiro. Em implementações de STM, os níveis de detecção mais recorrentes são:

Nível de objeto: um metadado é mantido para cada objeto que o programa instancia. Dessa forma, todos os atributos de um objeto estão associados a mesma unidade de sincronização.

Nível de palavra: nesse caso um metadado é relacionado a cada palavra de

memória, ao invés de objetos inteiros. Assim, no máximo uma transação detém a posse de cada palavra em um dado tempo.

Nível de bloco de palavras: nessa abordagem uma lista de número fixo de metadados é utilizada. As palavras de memória são mapeadas nessa lista através de uma função *hash*, podendo ocorrer o mapeamento de mais de uma palavra em um mesmo metadado.

A granularidade no nível de objeto é, em geral, de mais simples implementação. A Figura 9 mostra um exemplo dessa abordagem, onde é adicionado um metadado separado para cada objeto, que é usado para controlar o acesso a cada atributo desse objeto. Uma alternativa eficiente é implantar o metadado no cabeçalho do objeto. Isso reduz as faltas de cache e possibilita um único acesso ao metadado para uma série de acessos a diferentes atributos de um mesmo objeto (HARRIS; LARUS; RAJWAR, 2010). No entanto, esse método causa a detecção de falsos conflitos, no caso em que duas transações acessam atributos distintos de um mesmo objeto.

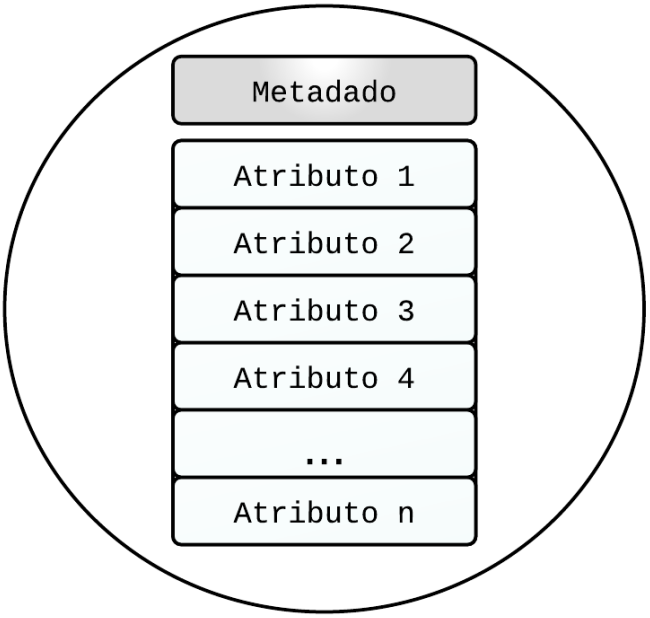

Figura 9: Detecção de conflitos no nível de objeto

A utilização de um metadado por palavra (Figura 10) elimina a ocorrência de falsos conflitos. Entretanto, impõe um *overhead* muito alto em termos de espaço e tempo para monitorar acessos à memória. A detecção em nível de blocos de palavras (Figura 11) provê melhor utilização da memória, pois demanda menos espaço que a granularidade a nível de palavra. No entanto, apesar de diminuir o *overhead* de gerenciamento, a granularidade em nível de blocos possibilita a detecção de falsos conflitos, assim como na granularidade de objeto.

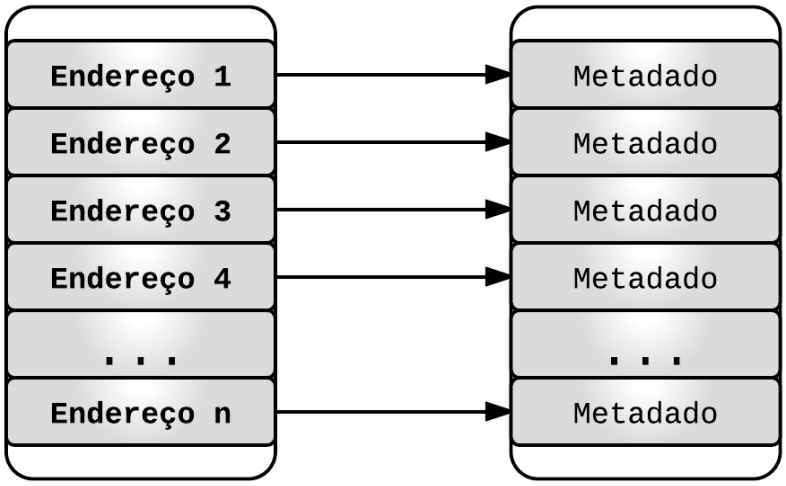

Figura 10: Detecção de conflitos no nível de palavra

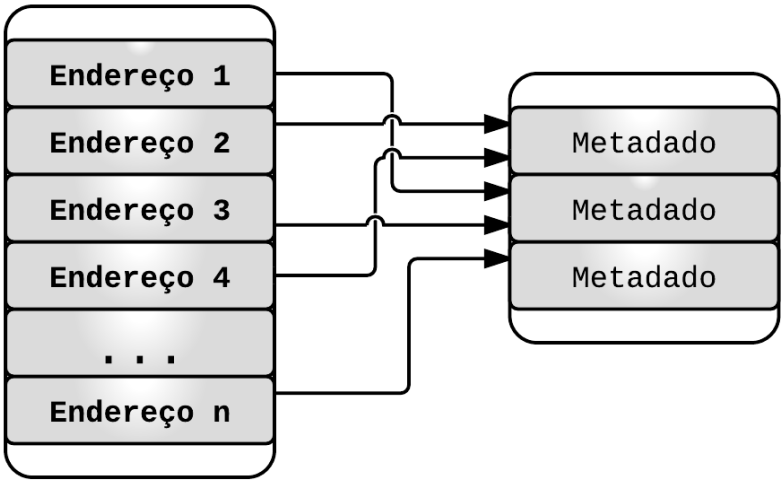

Figura 11: Detecção de conflitos no nível de bloco de palavras

3.5 Gerenciamento de Contenção

Após detectado o conflito, muitos sistemas possuem um gerenciador de contenção, o qual implementa uma ou mais políticas de resolução de conflitos. Quando um conflito ocorre, essas políticas ditam caso deva-se abortar a transação que detectou o conflito `TA`, a outra transação conflitante `TB`, e caso atrasar ou não uma das transação. Dentre as diversas políticas possíveis, destacam-se:

Tímido – A transação `TA` se aborta e reexecuta. É o mecanismo mais simples de gerenciamento de contenção.

Polite – A transação `TA` aguarda por um número fixo de intervalos de tempo. Ao final de cada intervalo, `TA` verifica se o conflito com `TB` ainda persiste. O

período do intervalo cresce exponencialmente, de acordo com o número de tentativas. Se o número máximo de intervalos for excedido TB é abortada.

Karma – Esse gerenciador tenta medir a quantidade de trabalho que uma transação realizou para tomar a decisão de abortá-la ou não. Apesar desta estimativa não ser trivial, o número de objetos que a transação acessou fornece uma boa aproximação. Nesse cenário, o gerenciador Karma usa como prioridade o número de acessos que cada transação realizou para decidir qual delas deve ser interrompida. A prioridade é calculada cumulativamente entre todas as tentativas, isto é, não é reiniciada quando a transação é abortada. Assim que uma transação é efetivada, sua prioridade é zerada. Em caso de conflito, TA aborta imediatamente outra transação com menor prioridade. No entanto, se a prioridade de TA é menor, a transação espera por um período fixo de tempo.

Greedy – A abordagem gulosa atribui um *timestamp* a cada transação no início de sua primeira tentativa de execução transacional. Em caso de conflito, a transação TA aborta a transação TB se TB possuir um *timestamp* mais novo que TA ou se TB já estiver esperando por outra transação.

Uma característica importante do método guloso é que elimina a possibilidade de ocorrência de *starvation*.

Polka – É uma combinação das políticas Polite e Karma. Esse gerenciador utiliza o *backoff* exponencial do Polite juntamente com o mecanismo de acumulação de prioridades do Karma. Como resultado, Polka atrasa por um intervalo de tempo igual a diferença de prioridades entre a transação TA e TB, sendo que o tamanho desse intervalo cresce exponencialmente.

A escolha entre os gerenciadores de contenção pode ser dependente do sistema de memória transacional, da carga de trabalho e do mecanismo de controle de concorrência usado pelo sistema (HARRIS et al., 2007). Por exemplo, se uma memória transacional usa detecção de conflitos adiantada, então o gerenciador de contenção deve fazer com que uma das transações conflitante espere tempo suficiente para a outra completar.

Além disso, um outro desafio no gerenciamento de contenção é como combinar algoritmos de gerenciamento de contenção dentro de um mesmo programa. Para cargas de trabalho simples, uma única estratégia pode ser apropriada para todos os *threads*. Em cargas de trabalho mais complexas, podem haver seções do programa que favorecem diferentes estratégias, seja variando durante o tempo ou concorrentemente para diferentes tipos de dados.

Conforme SCHERER III; SCOTT (2005), nenhum gerenciador de contenção supera os demais em todas as comparações, mas Polka alcança um bom desempenho mesmo nos casos em que é superado. O gerenciador Greedy tem melhor desempenho que Polka para cargas de trabalho de larga escala, mas tem baixo desempenho com transações curtas, pois todas as transações tem que incrementar, no início da execução, um contador compartilhado. Tal fato causa muitas faltas de cache e diminui significativamente a performance e a escalabilidade (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009).

3.6 Considerações Finais

Nesse capítulo foram discutidas as mais relevantes estratégias de implementação de um sistema de memória transacional. Como visto, para cada uma dessas estratégias, uma dada abordagem pode ser eficiente em determinados cenários mas inefetiva em outros.

O controle de concorrência pessimista é mais eficiente quando conflitos são frequentes. No caso em que conflitos são raros, o controle de concorrência otimista tende aumentar a concorrência entre transações. O versionamento adiantado é beneficiado por um cenário de baixa contenção, isto é, quando conflitos não são frequentes. Por outro lado, a técnica de versionamento tardio é mais efetiva quando da ocorrência frequente de cancelamentos.

A estratégia de detecção de conflitos adiantada pode evitar que uma transação continue executando desnecessariamente, uma vez que os conflitos são detectados mais cedo. No entanto, pode ocasionar perda de concorrência, cancelando transações que podiam ser efetivadas, dependendo do progresso das demais. Já a detecção tardia evita a perda de concorrência, pois conflitos são detectados apenas no fim da execução. Entretanto, pode gerar desperdício de trabalho no caso de uma transação condenada a abortar por um conflito, que é executada até o final.

Ambas estratégias impõem períodos de espera ou de desperdício. Esses períodos de tempo podem ser consideravelmente longos para transações grandes. Nesse contexto, a técnica de invalidação mista surge como ótima alternativa, pois beneficia-se das vantagens dos dois meios anteriores. Conflitos de escrita/escrita são detectados de maneira adiantada, evitando que um longo tempo seja gasto até o conflito ser detectado e resolvido. Conflitos de leitura/escrita são detectados tardiamente, assim aumentando o paralelismo.

A granularidade de detecção de conflitos afeta diretamente a performance de um sistema de MT. A redução do volume de metadados pode reduzir o *overhead* imposto na memória. Todavia, um baixo número de metadados pode levar a ocorrência de falsos conflitos entre transações que acessam diferentes porções de memória protegida pelo mesmo metadado.

Dentre as políticas de gerenciamento de contenção, há uma grande variedade de estratégias utilizadas para resolver os conflitos. O desempenho delas varia de acordo com o cenário, sendo que nenhuma das políticas de gerenciamento supera as demais em todas as configurações. No entanto, Polka apresenta o melhor desempenho no caso médio. Mesmo quando é superado por outra política, esse gerenciador alcança um bom desempenho, sendo uma boa escolha como gerenciador de contenção padrão.

No capítulo seguinte descreve-se a linguagem CMTJava, a qual permite as programação com memórias transacionais em Java.

4 CMTJAVA

CMTJava é um extensão de Java para programação utilizando o modelo de memórias transacionais, desenvolvida totalmente em software (STM). CMTJava é baseada em STM Haskell (HARRIS et al., 2005), uma extensão da linguagem funcional Haskell (HUDAK et al., 1992) para programação concorrente com memórias transacionais. Neste capítulo são apresentadas as principais características da linguagem. Um exemplo prático da aplicação da CMTJava é descrito, através da implementação de uma conta bancária acessada paralelamente.

Para apresentar a CMTJava, este capítulo divide-se em duas seções. Na primeira parte (Seção 4.1) são mostrados aspectos da linguagem referentes às construções que ela propõe para a programação concorrente. Na segunda parte (Seção 4.2), são descritos os requisitos de implementação necessários para dar suporte às construções propostas. As considerações finais do capítulo estão na Seção 4.3.

4.1 Características de implementação

A programação com memórias transacionais tem por objetivo simplificar o controle do acesso à memória compartilhada. CMTJava adiciona o suporte à transações de memória sem alterar drasticamente as características da linguagem Java. Em consequencia disto, os códigos produzidos são bastante semelhantes a um programa em Java puro.

Nesta seção descrevem-se as construções e abstrações utilizadas para o desenvolvimento de programas concorrentes utilizando CMTJava. Inicialmente são abordados os Objetos Transacionais, utilizados para definir objetos atômicos. Em seguida, explica-se a definição e composição de transações, através dos Blocos STMDO. Para facilitar o entendimento desses conceitos, utiliza-se como exemplo o problema de uma conta bancária que pode ser acessada concorrentemente.

4.1.1 Objetos Transacionais

Na CMTJava é utilizada uma abstração chamada de Objetos Transacionais (EDDON; HERLIHY, 2007). Nesse modelo, o programador declara determinados objetos como atômicos. Então, o sistema de memória transacional subjacente insere o código de sincronização e recuperação de erros no acesso a cada atributo do objeto.

A principal vantagem desta abordagem é a separação entre o código da aplicação e de controle de sincronização das transações. Ou seja, o código paralelo utilizando memórias transacionais fica muito semelhante ao código sequencial, pois detalhes da sincronização da execução não estão presentes no código definido pelo desenvolvedor. O mesmo não é possível afirmar-se sobre a sincronização baseada em *locks*, haja vista que o programador precisa instanciar e realizar operações utilizando os bloqueios para garantir o correto funcionamento do programa. O resultado disso é um programa paralelo no qual ficam misturados o código da aplicação e o código de sincronização, o que dificulta a legibilidade e manutenibilidade.

No caso da CMTJava, para definir um objeto transacional é necessário simplesmente estender a interface `TObject`. O compilador da linguagem se encarrega então de gerar os métodos para acesso a todos os atributos desse objeto. No problema da conta bancária, defini-se um objeto transacional `TAccount` (Figura 12). Esse objeto representa de forma simplista a abstração de uma conta, o qual possui como único atributo o saldo da conta.

```
1 class TAccount implements TObject{
2
3   private volatile Double balance;
4 }
```

Figura 12: Exemplo de objeto transacional

Tomando como exemplo o objeto transacional definido na Figura 12, o compilador geraria uma interface de acesso a esse objeto como a mostrada na Figura 13. Os métodos gerados retornam ações transacionais como resultado e são tipicamente chamados dentro dos blocos STMDO.

```
1 public STM<stm.Void> setBalance (Double n);
2
3 public STM<Double> getBalance();
```

Figura 13: Interface de acesso ao objeto `TAccount`

Para garantir que os atributos de um objeto transacional não poderão ser acessados fora de transações, foram utilizadas algumas convenções. Na classe que define um objeto transacional é permitida somente a declaração de atributos privados e métodos construtores. As demais operações sobre o objeto são definidas em uma classe separada, a qual estende o objeto transacional. Deste modo, a única forma de acesso a classe é através da interface gerada pelo compilador, descrita acima. Na subseção seguinte é mostrado como se dá o acesso aos objetos transacionais.

4.1.2 Blocos STMDO

Transações são compostas usando blocos STMDO. Os blocos STMDO em CMTJava são uma implementação da notação *do* disponível em Haskell.

```
class Account extends TAccount{

  public STM<stm.Void> deposit (Double n) {

    return new STMDO{
      Double balance <- this.getBalance();
      this.setBalance(balance + n)
    };
  }

  public STM<stm.Void> withdraw (Double n) {

    return new STMDO{
      Double balance <- this.getBalance();
      if(balance>=n)
        this.setBalance(balance - n)
      else
        STMRTS.retry()
    };
  }

  public STM<stm.Void> transfer(Account dest, Double n){

    return new STMDO{
      this.withdraw(n);
      dest.deposit(n)
    };
  }
}
```

Figura 14: Métodos de movimentação da conta

A notação $STMDO\{a_1; \ldots; a_n\}$ constrói uma ação STM que une pequenas operações $a_1; \ldots; a_n$ em sequência (DU BOIS; ECHEVARRIA, 2009).

O bloco STMDO funciona como um bloco atômico. O conjunto de operações componentes é executado de forma atômica e isolada das demais transações do sistema.

Na Figura 14 são apresentadas possíveis implementações de operações recorrentes de movimento de uma conta bancária: depósito, saque e transferência, respectivamente. Os blocos STMDO são utilizados para compor operações unitárias, formando uma operação maior, mas ainda assim mantendo a atomicidade na execução. Na CMTJava, para executar-se um bloco atômico utiliza-se o método `atomic` (Figura 15).

```
Account c = new Account();
atomic(c.deposit(100.0));
```

Figura 15: Execução de um bloco atômico

4.2 Implementação

A linguagem CMTJava propõe algumas abstrações de alto nível para a programação de memórias transacionais em Java, como os objetos transacionais e os blocos STMDO, descritos na seção anterior. Essas abstrações de alto nível são baseadas em uma linguagem de mais baixo nível composta de mônadas, *closures* e chamadas à bibliotecas que implementam transações. Logo, para a execução dos programas na linguagem é necessário transformar essas abstrações de alto nível para as construções de baixo nível disponíveis.

Na primeira parte da seção corrente descreve-se uma abstração de programação conhecida como função anônima ou *closure*. Em particular, é apresentada uma biblioteca que adiciona essa funcionalidade a linguagem Java. A Subseção 4.2.2 conceitua mônadas, uma formalização utilizada para descrever e combinar computações. Na Subseção 4.2.3 descreve-se a mônada para transações na CMTJava. Em seguida, na Seção 4.2.4 relata-se o processo de compilação de um programa em CMTJava.

4.2.1 Java *Closures*

Um *closure*, também conhecido como função anônima ou função lambda, é uma função que referencia variáveis livres no contexto léxico. Isto é, um closure pode referenciar variáveis não locais à função, ou seja, aquelas definidas fora do seu escopo. No entanto, o *binding* (ligação) desses valores é feito no momento da execução.

As funções anônimas estão presentes na maioria das linguagens de programação modernas, a exemplo de JavaScript, C# e Ruby. Até o momento da escrita deste trabalho, a linguagem Java não dispõe de closures em sua distribuição oficial. Para implementar CMTJava foi usada BGGA *Closures*[1], uma extensão Java que suporta *closures*.

Em BGGA, uma função anônima pode ser definida usando a seguinte sintaxe:

```
{ parâmetros formais => expressão}
```

`Parâmetros formais` e `expressão` são opcionais. Por exemplo:

```
{int x => x + 1 }
```

é uma função que recebe um inteiro e retorna seu valor incrementado em um. Uma função anônima pode ser invocada usando o método `invoke`:

```
String s = {=> "Hello!"}.invoke();
```

Uma função anônima também pode ser atribuída a variáveis:

```
{int => void} func = {int x => System.out.println(x)};
```

A variável `func` tem o tipo `{int => void}`, ou seja, é uma função de `int` para `void`. Esse *closure* recebe um valor inteiro e imprime-o. Funções também podem ser passadas como argumento em declarações de métodos Java.

[1] Disponível em: `http://www.javac.info/`

Um exemplo de *closures* em Java pode ser visto na Figura 16. Nesse exemplo, é definido um *closure* chamado `func` (Linha 3), que recebe um inteiro como parâmetro e retorna um inteiro como resultado. Posteriormente, o *closure* é invocado (Linha 5), imprimindo 3. O resultado é 3 pois no momento em que o *closure* é instanciado captura o valor da variável x no instante da execução.

```
1 public static void main(String[] args) {
2     int x = 1;
3     {int=>int} func = {int y => x+y };
4     x++;
5     System.out.println(func.invoke(1)); // vai imprimir 3
6 }
```

Figura 16: Exemplo de código com *closures*

Um *closure* pode usar variáveis de um escopo mesmo que esse escopo não esteja ativo no momento da invocação do *closure*. Por exemplo, se um *closure* é passado como argumento para um método, ele usará ainda as variáveis do escopo onde foi criado.

4.2.2 Mônadas

Apesar de bastante antigo, o conceito de mônadas popularizou-se pelo trabalho de Eugenio Moggi (MOGGI, 1991), que utilizou a noção de mônadas da teoria das categorias como forma de estruturar a descrição semântica de características como estado, exceção e continuação, em linguagens funcionais. Mônadas são uma forma de estruturar computações em termos de valores e sequências de computações usando esses valores. A mônada é usada para descrever computações que podem ser combinadas formando novas computações (NEWBERN, 2010).

Enquanto que as linguagens orientadas a objeto utilizam a abstração de dados para representar objetos e computações para comunicação entre esses, linguagens funcionais comumente usam mônadas como estrutura de programação para representar computações. Nas linguagens funcionais puras não existe a ocorrência de efeitos colaterais, que são usados nas linguagens imperativas para implementar entrada e saída de dados, exceções, estados, etc. Nesse contexto, as mônadas mostraram-se uma ótima solução para esse problema, pois permitem lidar com efeitos colaterais sem afetar a pureza da linguagem. Mônadas fornecem uma interface geral para computação sequencial, permitindo a construção de um fluxo de processamento em várias etapas. Isso permite a propagação de efeitos colaterais pelo retorno de funções, mantendo o modelo puramente funcional (BANDEIRA et al., 2011).

Apesar de adequarem-se muito bem ao propósito de lidar com efeitos colaterais em linguagens funcionais, a aplicação de mônadas vai muito além disso. Outras aplicações de mônadas podem ser vistas, por exemplo, na compreensão de listas e construção de parsers. Muitas vezes, o código com mônadas pode ser extremamente menor e mais legível que um código similar que faz uso de outra abstração.

4.2.3 A mônada STM

A mônada para ações STM é implementada como uma mônada de passagem de estados (DU BOIS; ECHEVARRIA, 2009; ECHEVARRIA, 2010). Essa mônada é usada para passar um estado pelas computações, onde cada computação retorna uma cópia alterada desse estado. No caso das transações, este estado é representado pelos metadados associados a transação, ou seja, seus *logs*. A mônada para transações da CMTJava é similar a utilizada no STM Haskell (HARRIS et al., 2005).

Uma mônada pode ser implementada como um tipo abstrato de dados que representa um contêiner para uma computação. Essas computações podem ser criadas e compostas usando três operações básicas: `bind`, `then` e `return`. Para uma mônada qualquer `m`, essas funções tem o seguinte tipo em Haskell:

```
bind ::  m a -> (a -> m b )-> m b
then ::  m a -> m b -> m b
return ::  a -> m a
```

O tipo `m a` representa uma computação dentro da monada `m` que quando executada produzirá um valor do tipo `a`. As funções *bind* e *then* são usadas para combinar computações em uma mônada. *Bind* executa seu primeiro argumento e passa o resultado para seu segundo argumento (uma função) produzindo uma nova computação. *Then* recebe duas computações como argumento e produz uma computação que irá executá-las uma depois da outra. A função *return* cria uma nova computação para um simples valor.

A classe `STM` é implementada como na Figura 17. A classe é usada para descrever transações, e tem somente um atributo: uma função representando uma transação. A transação `STM<A>` é uma função que recebe um objeto do tipo `Trans` como argumento e retorna um objeto `TResult`. `Trans` representa o estado corrente da transação durante sua execução e `TResult` descreve o novo estado da transação depois de sua execução.

```
1 class STM<A>{
2
3    {Trans => TResult} stm;
4
5    public STM({Trans => TResult} stm){
6       this.stm = stm;
7    }
8 }
```

Figura 17: Implementação da classe `STM`

A classe `TResult` (Figura 18) possui três atributos. O primeiro atributo (`result`) é o resultado da execução da ação STM. O segundo (`newTrans`), representa o estado da transação. Esse atributo contém tanto o *log* de escrita quanto o *log* de leitura. O terceiro atributo (`state`) é o estado corrente da transação. A execução de uma ação STM pode levar a transação a três estados:

`ACTIVE`**:** a execução da ação foi válida e a transação pode continuar

```
1  class TResult<A>{
2
3    A result;
4    Trans newTrans;
5    Trans.Status state;
6  }
```

Figura 18: Classe `TResult`

`RETRY`: a construção `retry` (Seção 2.5.2) foi chamada e a transação deve ser abortada

`ABORTED`: a transação foi abortada devido a ocorrência de um conflito.

```
1  public static <A,B> STM<B> bind(STM<A> t, {A => STM<B>} f){
2
3    return new STM<B> ( {Trans t1 =>
4      TResult<A> r1 = t.stm.invoke(t1);
5      TResult<B> r2;
6      if (r1.state == Trans.Status.ACTIVE) {
7        STM<B> r3 = f.invoke(r1.result);
8        r2 = r3.stm.invoke(r1.newTrans);
9      } else {
10       r2 = new TResult(null, r1.newTrans, r1.state);
11     }
12     r2
13   });
14 }
```

Figura 19: Método `bind`

O método `bind`, descrito na Figura 19, é usado para compor ações transacionais. `bind` recebe como argumento uma ação `STM<A>` e uma função do tipo `A => STM<B>`, retornando como resultado uma nova ação `STM<B>`. O objetivo do método `bind` é combinar ações STM gerando novas ações. Inicialmente a ação `t` é executada recebendo a representação do estado do transação (`t1`) como argumento (Linha 4). Se a invocação de `t` for válida - seu estado é `ACTIVE` - então `f` é chamada gerando o resultado `STM<B>`. Caso contrário, a segunda ação não é executada e `bind` retorna `STMResult` com o estado `RETRY` ou `ABORTED`, de acordo com estado da execução de `t`.

O método `then` é implementado de forma similar (Figura 20). O método `then` é uma combinação sequencial: ele recebe como argumento duas ações STM e retorna uma ação que irá executá-las uma depois a outra.

Finalmente, o método `stmReturn` é usado para inserir um objeto qualquer `a` dentro da mônada STM. O método `stmReturn` (Figura 21) funciona igual ao método `return` de Java, só que para blocos STM. Ele recebe um objeto como argumento e cria uma simples transação que retorna esse objeto como resultado. Ele pode ser usado também para criar novos objetos dentro de uma transação.

```
public static <A,B> STM<B> then(STM<A> a, STM<B> b){

  return new STM<B> ( {Trans t1 =>
    TResult<A> r1 = a.stm.invoke(t1);
    TResult<B> r2;
    if (r1.state == Trans.Status.ACTIVE) {
      r2 = b.stm.invoke(r1.newTrans);
    } else {
      r2 = new TResult(null, r1.newTrans, r1.state);
    }
    r2
  });
}
```

Figura 20: Método `then`

```
public static <A> STM<A> stmReturn(A a){

  return new STM<A>({ Trans t1 => new TResult(a, t1,
  Trans.Status.ACTIVE)});
}
```

Figura 21: Método `stmReturn`

4.2.4 Compilação

Em um primeiro momento, as transformações do código em CMTJava para as construções de baixo nível eram feitas manualmente. Com o desenvolvimento da linguagem, essa tarefa foi tornando-se muito complexa, a ponto de limitar o desenvolvimento de programas um pouco mais elaborados. Posteriormente, foi desenvolvido um compilador para automatizar o processo de compilação dos programas em CMTJava. A ferramenta recebe códigos escritos em CMTJava e gera *bytecodes* Java, tendo como código intermediário a linguagem Java com *closures* (BANDEIRA; DU BOIS; PILLA, 2011).

Para realizar essa geração, o compilador é dividido em duas partes básicas: tradução e síntese. A tradução encarrega-se da realização de transformações no código e inserção de métodos e atributos que devem ser criados pelo compilador. Já a fase de síntese gera código executável, a partir do código intermediário.

Uma visão geral da compilação de um programa CMTJava pode ser vista na Figura 22. Mais detalhadamente, a tradução compreende as seguintes fases: análise léxica e análise sintática do código fonte em CMTJava, geração dos métodos *get* e *set* para todos os atributos das classes que implementam a interface `TObject` e a tradução dos blocos STMDO para chamadas de *bind* e *then* (etapas detalhas na sequencia do texto). Após essa fase, o código traduzido para Java puro + *closures* é compilado usando o compilador javac do BGGA *closures*.

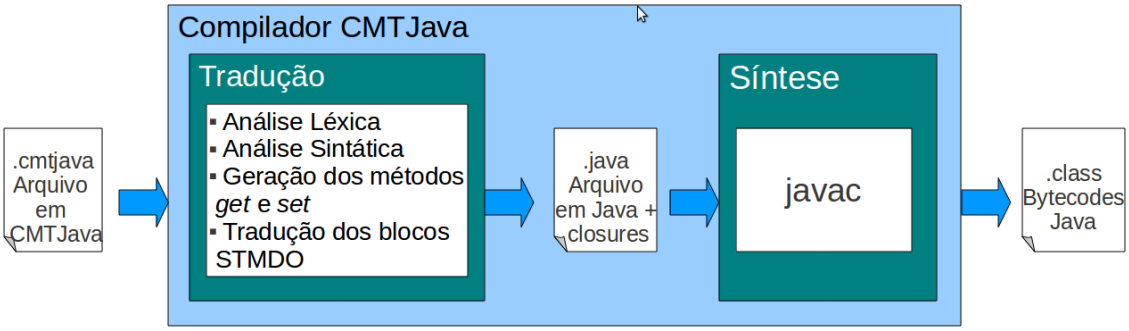

Figura 22: Processo de compilação de um arquivo na CMTJava

Análise do código

Para o desenvolvimento do compilador da CMTJava foi utilizado o ANTLR[2]. ANTLR (PARR, 2007) é um acrônimo para *ANother Tool for Language Recognition*, um sofisticado gerador de *parsers* que pode ser usado para implementar interpretadores, compiladores e outros tradutores.

Para analisar o código fonte, a ferramenta tenta derivar o programa de entrada de acordo com a gramática definida da linguagem. Essa gramática descreve tanto as regras sintáticas quanto léxicas. A análise semântica, uma verificação realizada na grande maioria dos compiladores, não é feita na etapa de geração. Essa análise é somente executada na etapa de síntese do código intermediário.

Caso o programa de entrada em CMTJava seja léxica e sintaticamente correto, então é efetuada a geração e tradução de código.

Geração da interface de acesso aos objetos transacionais

O acesso aos objetos transacionais na CTMJava é feito atômicamente dentro das transacões. Para garantir essa atomicidade, esses objetos proveem uma interface de acesso aos seus membros. Esta é a única forma de acesso a esses atributos.

Contudo, essa interface é diferente do que normalmente é definido para um objeto em uma linguagem de programação orientada a objetos. Em um programa sequencial, a interface do objeto tipicamente contém, para cada atributo de acesso externo, um método para modificar e outro para ler o valor do atributo. Nos objetos transacionais, os métodos da interface também são responsáveis pela sincronização do acesso concorrente.

Esse código de sincronização está diretamente ligado a implementação do sistema transacional. Isto é, os diferentes sistemas transacionais da CMTJava geram métodos diferentes. Entretanto, a interface é sempre a mesma. Desta forma os códigos de aplicação escritos são independentes do algoritmo usado para coordenação da execução concorrente.

Para gerar a interface de acesso aos objetos transacionais, o compilador da CMTJava também faz uso do StringTemplate[3]. StringTemplate é um motor de modelos (*templates*) para Java, que gera código fonte ou qualquer outra saída de

[2]Disponível em: `http://www.antlr.org/`
[3]Disponível em: `http://www.stringtemplate.org/`

texto formatado. *Templates* são documentos com lacunas que são preenchidas com parâmetros, passados ao *template* (PARR, 2007).

Geradores construídos com instruções de impressão (*prints*) espalhados pela gramática são mais difíceis de codificar, ler e mudar o código alvo. Em contrapartida, os geradores que utilizam *templates* especificam a estrutura da saída separadamente, facilitando sua definição e edição. Além disso, ANTLR carrega esses modelos em tempo de execução.

O benefício da utilização desse mecanismo é que para modificar a implementação da interface dos objetos transacionais não são necessárias modificações na gramática da linguagem. Apenas modifica-se o *template*.

Tradução dos blocos STMDO

Blocos STMDO são traduzidos para chamadas de *bind* e *then* usando as regras de tradução mostradas na Tabela 1. As regras de tradução dos blocos STMDO são muito similares as regras de tradução da notação *do* descrita em (HARRIS et al., 2005).

Tabela 1: Esquema de tradução dos blocos STMDO

CMTJava	Java + *closures*
`STM{ type var <- e; s}`	`STMRTS.bind(e, { type var => STM{ s }})`
`STM{ e; s }`	`STMRTS.then( e, STM{ s })`
`STM{ e }`	`e`

O compilador da linguagem transforma recursivamente cada sentença que compõe o bloco para chamadas da mônada STM. Por exemplo, a implementação da operação de depósito sobre a conta descritas na Figura 14, seria transformada para o código mostrado na Figura 23.

```
1 public STM<stm.Void> deposit (Double n) {
2
3   STMRTS.bind( this.getBalance(),{ Double balance =>
4     this.setBalance(balance + n) });
5 }
```

Figura 23: Método `deposit` traduzido para o código intermediário

4.3 Considerações Finais

No presente capítulo foram descritas as principais características da linguagem de programação concorrente com memórias transacionais CMTJava. Apresentou-se os requisitos da implementação e funcionamento interno da linguagem. Além disso também foram mostrados alguns exemplos de código na linguagem.

No próximo capítulo realiza-se um estudo dos principais trabalhos relacionados a CMTJava, e em particular a implementação do sistema para execução de transações descrito neste trabalho. No capítulo serão caracterizadas outras implementações de MT, discutindo as particularidades de cada uma.

5 TRABALHOS RELACIONADOS

Dentre a grande variedade de memórias transacionais em software, este capítulo apresenta alguns dos principais trabalhos da literatura. Devido ao enfoque deste trabalho, é dada atenção especial às características de implementação de cada um dos sistemas.

Nesse capítulo são descritos os seguintes sistemas de STM:*Transactional Locking II*, um dos algoritmos mais conhecidos e que influenciou grande parte das implementações atuais, TinySTM, que apesar de semelhante ao TL2 apresenta um desempenho superior na maioria dos casos, RSTM, que utiliza uma abordagem distinta das demais para detecção de conflitos e validação de transações e finalmente SwissTM, uma implementação mais recente, misturando características das anteriores e que mostra resultados bastante efetivos.

5.1 *Transactional Locking II*

O algoritmo *Transactional Locking II* (TL2) (DICE; SHALEV; SHAVIT, 2006) utiliza um relógio de versão global (*global version-clock*). Esse relógio é incrementado por cada transação que escreve na memória e lido por todas as transações.

No TL2, para cada aplicação ou estrutura de dados, é alocada uma coleção de *write-locks* versionados. Esses bloqueios possuem, além da indicação que o *locks* está ou não adquirido, um número de versão armazenado no restante da palavra de memória. Dentre as possíveis associações de blocos de memória compartilhada com os *locks*, destacam-se: *per object* (PO), onde um *lock* é atribuído por objeto compartilhado e *per stripe* (PS), onde aloca-se um vetor de *locks* e a memória é particionada usando alguma função *hash* para mapear os bloqueios na memória.

Uma transação de escrita, ou seja, aquela que escreve na memória compartilhada, realiza a seguinte sequencia de operações:

1. Leitura do relógio de versão global
2. Execução especulativa
3. Aquisição dos *locks* do conjunto de escrita
4. Incremento do relógio de versão global
5. Validação do conjunto de leitura

6. Efetivação e liberação dos *locks*

A seguir são detalhadas essas etapas:

1. **Leitura do relógio de versão global:** Carrega o valor corrente do relógio de versão global e armazena-o em uma variável local do *thread* chamada *read-version* (`rv`).

2. **Execução especulativa:** A transação é executada sem alterar a memória compartilhada. Um conjunto de leitura contendo os endereços carregados e um conjunto de escrita contendo endereço/valor a ser escrito na memória são mantidos. A operação de leitura primeiro verifica se o endereço encontra-se no conjunto de escrita, nesse caso a transação retorna o último valor escrito no endereço pela transação corrente. A cada leitura ocorre uma validação para verificar se o *write-lock* não foi modificado. Se a versão do *lock* é maior que `rv` então aquela região de memória foi modificada depois do *thread* atual realizar o passo 1, logo a transação é abortada.

3. **Aquisição dos *locks* do conjunto de escrita:** Os *locks* do conjunto de escrita são adquiridos em uma ordem conveniente. Caso algum não consiga ser adquirido a transação é abortada.

4. **Incremento do relógio de versão global:** Caso a aquisição dos *locks* seja bem sucedida, é feita uma operação *increment-and-fetch* (usando CAS por exemplo) do relógio de versão global gravando o número de versão de escrita local `wv`.

5. **Validação do conjunto de leitura:** Verifica-se para cada entrada no conjunto de leitura se o número de versão associado com o *write-lock* é menor ou igual a `rv`. Revalidando o conjunto de leitura, garante-se que as posições de memória não foram modificadas enquanto os passos 3 e 4 estavam sendo executados.

6. **Efetivação e liberação dos *locks*:** Para cada entrada no conjunto de escrita, armazena-se o novo valor contido no *log* e libera-se o *lock* associado modificando o número de versão do *lock* para a versão de escrita `wv`.

Como os *locks* são mantidos por um curto período de tempo, obtém-se um aumento de performance em situações de alta contenção. As transações somente leitura são executadas de forma mais simples, pois somente as etapas 1 e 2 são realizadas. Dessa forma, esse tipo de transação é altamente eficiente porque não constrói e nem valida o conjunto de leitura. A detecção do comportamento somente leitura pode ser feita tanto em tempo de compilação quanto em tempo de execução.

Por outro lado, o acesso a um relógio global pode comprometer o desempenho do sistema, principalmente para transações pequenas. Tal gargalo pode limitar consideravelmente a escalabilidade, serializando as transações.

5.2 TinySTM

TinySTM (FELBER; FETZER; RIEGEL, 2008) é uma implementação de memória transacional baseada no algoritmo LSA (RIEGEL; FELBER; FETZER, 2006). As características do LSA (*Lazy Snapshot Algorithm*) são bastante similares as do TL2, como por exemplo a utilização de um relógio global, sincronização baseada em *locks* e detecção de conflito a nível de palavra. A principal distinção é que, diferentemente do algoritmo TL2, o LSA adquire *locks* tão logo tente escrever em uma posição de memória. Essa detecção adiantada de conflitos evita que uma transação condenada a ser abortada continue executando e assim desperdiçando recursos.

Quanto a atualização da memória, TinySTM conta com duas políticas distintas: *write-through* e *write-back*. Com *write-through*, as atualizações são feitas diretamente na memória e os valores anteriores são armazenados em um registro (*undo log*) para serem restaurados em caso de aborto. Essa abordagem diminui o *overhead* de *commit*. No acesso *write-back*, os valores são armazenados em um *log* e escritos na memória em caso de efetivação, diminuindo os custos para abortar uma transação.

Como na maioria dos projetos de STM baseados em palavra, TinySTM conta com um *array* de *locks* para gerenciar o acesso concorrente a memória, onde cada *lock* cobre uma parte do espaço de endereçamento. Uma característica bastante interessante utilizada no TinySTM é o “ajuste” de alguns parâmetros em tempo de execução, como por exemplo o tamanho do vetor de *locks*. Tal característica, permite que o algoritmo se adapte a carga de trabalho e desta forma aumente sua performance.

5.3 RSTM

Rochester Software Transactional Memory System (RSTM) é uma rápida e não bloqueante biblioteca C++ que visa maximizar o *throughput* e fornece uma interface de programação simples (SPEAR et al., 2006). Como na maioria das outras STMs não bloqueantes, um objeto é acessado através de um cabeçalho, o qual permite as transações identificarem a última versão efetivada de um objeto e também a versão especulativa corrente.

Cada *thread* mantém um descritor da transação que indica o status (ativa/efetivada/abortada) da sua transação mais recente, junto com uma lista de objetos abertos para leitura e escrita. O programador pode especificar se as leituras são visíveis ou invisíveis. Na prática, leituras visíveis tendem a causar um aumento significante no tráfego de memória, levando estas a terem desempenho pior que leituras invisíveis.

RSTM utiliza um mecanismo heurístico de validação, o contador de *commit* global. Mais especificamente, cada objeto possui um contador de leitura complementado com um contador de conflitos global. Leitores incrementam e decrementam o contador por objeto no momento da abertura e da efetivação,

respectivamente. Escritores incrementam o contador de conflitos sempre que adquirem um objeto cujo contador de leitura é diferente de zero. Ao abrir um novo objeto, um leitor pode pular a validação incremental se o contador de conflitos global não foi modificado desde a última vez que o leitor o verificou.

5.4 SwissTM

SwissTM (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009) é uma implementação de MT que busca obter bom desempenho tanto para transações curtas e estruturas de dados pequenas, assim como para transações grandes e cargas de trabalho complexas. SwissTM detecta conflitos de escrita/escrita de forma adiantada, visando prevenir transações condenadas a abortar de continuarem executando e assim gastando recursos. Conflitos de leitura/escrita são detectados de forma tardia, para permitir de forma otimista mais paralelismo.

O gerenciador de contenção usa o esquema tímido para transações curtas ou somente leitura, que aborta a transação tão logo encontre um conflito. Transações mais complexas trocam dinamicamente para o mecanismo guloso (*greed*), que apesar de envolver maior *overhead* favorece esse tipo de transação, prevenindo *starvation*.

A seguir são descritas as principais características da SwissTM:

Início da transação: Cada transação, ao iniciar, lê o contador global `commit-ts` e armazena-o localmente em `tx.valid-ts`.

Leitura: Ao ler uma posição de memória `addr`, a transação `T` primeiramente lê o valor de `w-lock` para detectar casos de leitura após escrita. Se `T` é detentora de `w-lock`, então `T` retorna o valor de seu *log* de escrita. Caso outra transação possua o *lock* ou este esteja livre, `T` então lê o valor de `r-lock`, o valor de `addr` e novamente o valor de `r-lock`. A transação `T` repete essas leituras até que duas condições sejam satisfeitas: (1) os valores de `r-lock` serem os mesmos, indicando que a transação leu valores consistentes de `r-lock` e `addr`, e (2) `r-lock` estar liberado. Quando livre, `r-lock` contem a versão atual de `addr`. Se essa versão é menor ou igual ao valor do contador local `tx.valid-ts`, indicando que `addr` não foi modificado desde o início ou da ultima validação de `T`, `T` retorna o valor de `addr` lido. Caso contrário, `T` revalida seu conjunto de leitura incrementado o contador local para o valor corrente do relógio global, no caso de sucesso na revalidação. Se a revalidação não for bem sucedida a transação é reexecutada.

Escrita: Ao tentar escrever numa posição de memória `addr`, a transação `T` primeiramente verifica se é detentora do *lock* de escrita correspondente a `addr`. Em caso afirmativo, `T` simplesmente atualiza o valor do endereço `addr` no seu *log* de escrita. Do contrário, `T` tenta adquirir esse *lock*. Caso não obtenha exito na aquisição, a transação pergunta ao gerenciador de contenção se deve reiniciar ou esperar a transação de posse do *lock* executar.

Validação: Para validar-se, `T` compara a versão atual de todas as posições de memória lidas até então com suas versões no ponto em que foram inicialmente lidas por `T`. Se existir alguma discrepância entre algum número de versão, a validação falha.

***Commit*:** Uma transação somente leitura `T` pode efetivar imediatamente, pois seu *log* de leitura é garantidamente consistente. Uma transação `T` que escreve na memória inicialmente bloqueia todos os *locks* de leitura de endereços que `T` precisa escrever. Então, `T` incrementa o valor de `commit-ts` e revalida seu *log* de leitura. Se essa validação não for bem sucedida, `T` descarta as modificações e é reexecutada. Em caso de sucesso, `T` percorre seu conjunto de escrita, atualizando seus valores correspondentes na memória, liberando e atualizando o valor dos *locks* de leitura e escrita.

Gerenciador de contenção: SwissTM utiliza um gerenciador de contenção de duas fases, onde a fronteira entre estas fases é dada pelo número de escritas na memória executadas até um determinado momento. Em caso de conflito, uma transação que está na primeira fase é abortada imediatamente. Caso as duas transações estejam na segunda fase, o que realizou mais trabalho é priorizada.

5.5 Comparação das implementações

Na Tabela 2 é feita uma comparação das principais características de implementação das STMs discutidas neste capítulo. É possível perceber que as implementações possuem bastante características em comum e que RSTM é a que mais se distingue das demais. Dentre as implementações analisadas, a maioria detecta conflitos no nível de palavra de memória, usando *locks* para sincronização. O gerenciador de contenção mais usado é o tímido, que é também o mais simples. No que diz respeito a detecção de conflitos, tem-se uma maior divergência, onde são utilizadas tanto abordagens puramente adiantadas ou tardias, quanto abordagens que combinam essas duas.

Tabela 2: Principais características de implementação dos sistemas de MT analisados

Característica	**TL2**	**TinySTM**	**SwissTM**	**RSTM**
Gran. detecção do conflito	Palavra	Palavra	Palavra	Objeto
Sincronização	*Locks*	*Locks*	*Locks*	Livre de obstrução
Gerenciador de contenção	Tímido	Tímido	Duas fases	Polka
Detecção de conflitos	Tardia	Adiantada	Mista	Mista

5.6 Considerações Finais

Dentro o grande número de implementações de STM disponíveis, este capítulo descreveu algumas delas. Essas implementações apesar de compartilharem algumas características, tem seu desempenho bastante diferente para determinados cenários de execução, como tipo e taxa de ocorrência de conflitos, tamanho das transações, dentre outros.

O algoritmo TL2 funciona bem para transações pequenas, devido ao esquema de aquisição de *locks* em tempo de efetivação. No entanto, perde em desempenho em transações maiores que eventualmente abortam por conflitos que são detectados tardiamente. Apesar do desempenho, esse algoritmo é bastante importante pela sua influência em outras implementações. Embora bastante similar ao TL2, TinySTM é mais eficiente em geral, devido a detecção de conflitos adiantada.

RSTM tem seu desempenho bastante comprometido em situações de acesso a objetos pequenos, devido ao *overhead* na detecção de conflito. Já SwissTM mostra-se bastante eficiente em diversos cenários, devido a sua capacidade de se adaptar a carga de trabalho. Essa eficiência se deve a adoção das estratégias mais eficientes das outras STMs. Isto é, a detecção adiantada de conflitos de escrita/escrita e tardia de conflitos de leitura/escrita, juntamente com a detecção de conflitos a nível de palavra, que impõe menor *overhead* que detecção a nível de objeto.

O próximo capítulo descreve o novo sistema transacional deselvolvido para a CMTJava.

6 SISTEMA DE DETECÇÃO DE CONFLITOS COM INVALIDAÇÃO MISTA

O objetivo principal deste trabalho é a implementação de um novo sistema transacional para a CMTJava. Tal sistema é responsável pela execução e gerenciamento de concorrência das transações na linguagem. Todavia, a interface da mônada para transações (Subseção 4.2.3) não é modificada, apenas sua implementação interna. Nesse capítulo é descrito em detalhes o algoritmo desenvolvido.

O primeiro sistema transacional desenvolvido para a linguagem (DU BOIS; ECHEVARRIA, 2009) foi baseado no sistema transacional da linguagem STM Haskell (HARRIS et al., 2005). Esse sistema possui implementação simples pois, para validar-se ou efetivar as computações, uma transação precisa adquirir um bloqueio global. Essa restrição acaba serializando as transações e assim limitando o paralelismo.

Posteriormente, foi implantada uma estratégia mais eficiente para execução das transações (ECHEVARRIA, 2010). O segundo sistema transacional foi desenvolvido com base no algoritmo TL2 (*Transactional Locking II*) (DICE; SHALEV; SHAVIT, 2006), apresentado na Seção 5.1 . Esse algoritmo realiza a detecção de conflitos de forma tardia. Isto é, a transação executa acessando a memória compartilhada, verificando apenas em tempo de efetivação se essa execução foi válida. Entretanto, esta estratégia muitas vezes degrada o desempenho do sistema. Transações conflitantes sempre executam até o final, para só então serem detectadas e reexecutadas. Tal fator é agravado a medida que o número de conflitos e o tamanho das transações cresce.

O novo sistema de detecção de conflitos desenvolvido neste trabalho baseia-se no algoritmo de (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009), descrito na Seção 5.4. Este algoritmo utiliza a estratégia de invalidação mista (SPEAR et al., 2006) para detectar e resolver conflitos. A detecção de conflitos entre transações de escrita é feita de forma adiantada. Como nesse caso tipicamente é necessário abortar uma das transações, evita-se assim que a transação conflitante seja executada até o fim, para só então ser abortada. Já os conflitos entre uma transação de leitura e outra de escrita são detectados tardiamente. Isto permite maior paralelismo na execução, uma vez que esse tipo de conflito pode muitas vezes ser resolvido sem cancelamentos (quando a transação de leitura é efetivada antes da transação de escrita).

Na primeira seção do capítulo é descrita uma visão geral do sistema desenvolvido, ou seja, as principais características de implementação.

Posteriormente, aprofunda-se essa discussão, apresentando fragmentos de código utilizados para implementar o suporte a objetos transacionais e transações (Seções 6.2 e 6.3, respectivamente). Os resultados obtidos com o novo sistema são discutidos na Seção 6.4. A Seção 6.5 descreve as observações finais sobre este capítulo.

6.1 Visão geral

O sistema utiliza um contador de *commits* global para identificar conflitos. Qualquer transação de escrita que é efetivada na memória incrementa o valor do contador global. Ou seja, se x é o valor do contador de *commits* em um dado instante de tempo, então x transações de escrita foram efetivadas no sistema até aquele momento.

Os objetos transacionais são dotados de indicadores de versão. Cada atributo de um TObject é provido de dois *locks* versionados: um *lock* de leitura e um *lock* de escrita. O *lock* além da função principal de bloquear leituras ou escritas no atributo, também é utilizado para guardar o indicador de versão do mesmo. Ao efetivar suas computações na memória, uma transação atribui o novo valor do contador global de *commits* a versão de cada atributo modificado. Esse novo valor é obtido após incrementar o valor do contador global.

No início da execução, toda transação lê o valor do contador global e armazena-o. Essa amostra é utilizada pela transação para verificar sua validade e é chamada de *timestamp* de validação local. A cada operação, seja de leitura ou escrita, uma transação T verifica se o indicador de versão do atributo é maior que o valor do seu *timestamp* de validação local. Em caso afirmativo, outra transação concorrente U alterou esse atributo depois do início da execução ou da última validação de T. Nesse caso, a validade da transação T deve ser verificada antes de prosseguir sua execução.

A Tabela 3 sintetiza as principais características de implementação da CMTJava, utilizando seu mais novo sistema transacional.

Tabela 3: Principais características de implementação da CMTJava

Característica	**CMTJava**
Gran. detecção do conflito	Atributo
Sincronização	*Locks*
Gerenciador de contenção	Tímido
Detecção de conflitos	Mista
Versionamento de dados	Tardio

6.2 Objetos transacionais

Como descrito na Subseção 4.1.1, o compilador da CMTJava gera a interface de acesso aos objetos transacionais. Essa interface contém o código de sincronização do acesso a cada atributo do objeto. Logo, a definição desses métodos varia para cada um dos sistema transacionais da linguagem, pois cada qual emprega uma estratégia de gerência de conflitos diferente. Os códigos apresentados a seguir são correspondentes ao sistema transacional desenvolvido nesta seção.

```
1  class TAccount implements TObject{
2
3    private volatile Double balance;
4    private FieldInfo<Double> balanceFieldInfo =
5      new FieldInfo<Double>({Double a => balance = a;});
6
7    public STM<stm.Void> setBalance(Double n){
8     ...
9    }
10
11   public STM<Double> getBalance(){
12    ...
13   }
14 }
```

Figura 24: Estrutura básica do código gerado pelo compilador para a classe `TAccount`

```
1  public class FieldInfo <A> {
2
3    {A => void} updateField;
4    AtomicMarkableReference<Long> wlock;
5    AtomicMarkableReference<Long> rlock;
6    Vector<Block> blockedThreads;
7    ...
8  }
```

Figura 25: A classe `FieldInfo`

Na Figura 24 é mostrado o código gerado para o objeto transacional `TConta`, definido na Figura 12. Além de gerar os métodos de get e set, que serão apresentados na sequencia do texto, também é gerada uma referência para um objeto do tipo `FieldInfo` (Linha 4). Este objeto é responsável por guardar os metadados de cada atributo, uma vez que a CMTJava realiza detecção de conflitos neste nível de granularidade. O classe `FieldInfo` (Figura 25) é composta por dados utilizados pelo sistema transacional para o controle de concorrência. `updateField` é um *closure*, utilizado para atualizar o valor do atributo em caso de efetivação.

Os *locks* versionados de escrita (`wlock`) e leitura (`rlock`) são representados por um objeto do tipo `AtomicMarkableReference`. Este objeto é formado por uma referência e um valor booleano. A razão da utilização desse tipo de dados é a possibilidade de modificar seus dois atributos de forma atômica. Além disso, também é possível realizar essa modificação atômica condicionalmente. Isto é, se e somente se os valores atuais de ambos atributos forem iguais aos dados fornecidos como parâmetro. O dado booleano é utilizado com a mesma semântica de um *lock* comum. Se o valor é igual a verdadeiro, o *lock* está bloqueado. Se falso, está livre. Já a referência é utilizada para associar a informação de versão ao valor booleano, isto é, ao *lock*. Quando o valor booleano é verdadeiro, a referência contém o identificador da transação que possui o *lock* em questão. Somente esta transação modificará o bit de posse para falso novamente. Enquanto isso não ocorrer, as demais transações falharão na operação de adquirir o *lock*. Quando o valor booleano é falso, `rlock` guarda na referência o indicador de versão do atributo. Para o *lock* de escrita, nenhuma informação é guardada na referência quando o *lock* encontra-se desbloqueado.

No vetor `blockedThreads` são armazenados os indicadores das transações que estão bloqueadas por aquele atributo. Tal informação é utilizada na implementação de método `retry`, descrita na Subseção 6.3.4. A seguir é apresentado o corpo dos métodos para leitura e escrita do objeto `TAccount` (Figura 24).

6.2.1 Leitura

O código para leitura do valor do saldo da conta bancária pode ser visto na Figura 26.

Ao ler o valor do atributo, inicialmente é verificado se a transação possui o *lock* de escrita associado ao mesmo (Linha 9). O fato da transação já possuir o *lock* de escrita indica que esta já modificou o atributo anteriormente. Logo, a transação deve ler o valor de seu *log* de escrita (Linha 11) e não o valor contido na memória. Se a transação não realizou uma escrita no atributo anteriormente, é efetuada a leitura do valor na memória (Linha 21). Esta leitura é acompanhada de uma validação: a transação verifica o lock de leitura duas vezes, uma antes (Linhas 15 e 18) e outra depois da leitura do valor do atributo (Linha 22). Esses passos são repetidos até que, para ambas leituras, os valores de versão lidos sejam iguais e que o *lock* esteja desbloqueado (Linha 23). Os valores iguais da referência em `rlock` indicam que a versão do atributo não mudou durante a leitura. O fato de `rlock` estar desbloqueado significa que nenhuma transação está sendo efetivada no sistema no momento da leitura e que a referência contida no *lock* é de fato a versão do atributo.

Após a leitura, a transação adiciona o atributo ao seu *log* de leitura (Linha 26). Caso o indicador de versão do atributo for maior que o valor do *timestamp* de validação da transação, a transação é validada (Linha 27).

6.2.2 Escrita

Para modificar o valor do atributo (Figura 27), a transação primeiramente verifica se detém o *lock* de escrita (Linha 7), assim como é feito na operação de leitura. Se o *lock* já está em posse da transação, coloca-se o novo valor no conjunto de escrita da transação (Linha 8).

```
1  public STM<Double> getBalance() {
2
3    return new STM<Double> ({Trans t =>
4      TResult r = null;
5      Double result;
6      boolean mark[] = {false};
7      Long version = balanceFieldInfo.wlock.get(mark);
8
9      if(balanceFieldInfo.wlock.isMarked() &&
10      threadId.equals(balanceFieldInfo.wlock.getReference())){
11       result = (Double)t.writeSet.get(balanceFieldInfo);
12       r = new TResult(result, t, STMRTS.ACTIVE);
13     }else{
14       boolean holder[] = {false};
15       Long version = balanceFieldInfo.rlock.get(holder);
16       while(true){
17         if(holder[0]==true){
18           version = balanceFieldInfo.rlock.get(holder);
19           continue;
20         }
21         result = saldo;
22         long version2 = balanceFieldInfo.rlock.get(holder);
23         if(version == version2 && holder[0]==false)
24           break;
25       }
26       boolean added = t.readSet.put(balanceFieldInfo, version);
27       if (!added || (version > t.readStamp && !t.extend())){
28         r = new TResult(null, t, STMRTS.ABORTED);
29       }else{
30         r = new TResult(result, t, STMRTS.ACTIVE);
31       }
32     }
33     r
34   });
35 }
```

Figura 26: Código para a leitura de um atributo de um objeto transacional

Se não escreveu no atributo previamente, a transação tenta adquirir o *lock* de escrita (Linha 11). Caso a aquisição seja bem sucedida, a transação adiciona o valor modificado ao seu *log* (Linha 13). Se a versão do atributo for maior que o *timestamp* de validação, a transação é revalidada. Caso a aquisição do *lock* ou a validação falhar, aborta-se a transação.

6.3 Transações

As transações são executadas através de uma chamada ao método `atomic` (Seção 4.1.2). Para executar uma ação transacional, o método `atomic` (Figura 28) cria um objeto do tipo `Trans`. Esse objeto representa o estado da

```
1  public STM<stm.Void> setBalance (Double n) {
2
3    return new STM<stm.Void>({Trans t =>
4      TResult r = null;
5      boolean mark[] = {false};
6      Long version = balanceFieldInfo.wlock.get(mark);
7      if(mark[0] && t.transId.equals(version)){
8        t.writeSet.put(balanceFieldInfo, n);
9        r = new TResult(new stm.Void(), t, Trans.Status.ACTIVE);
10     }else{
11       if(balanceFieldInfo.wlock.compareAndSet(null,
12       t.transId, false, true)){
13         t.writeSet.put(balanceFieldInfo, n);
14         if(balanceFieldInfo.rlock.getReference() > t.readStamp
15         && !t.extend()){
16           r = new TResult(null, t, Trans.Status.ABORTED);
17         }else
18           r = new TResult(new stm.Void(), t,
19           Trans.Status.ACTIVE);
20       }else{
21         r = new TResult(null, t, Trans.Status.ABORTED);
22       }
23     }
24     r
25   });
26 }
```

Figura 27: Código para a modificação de um atributo de um objeto transacional

transação durante a sua execução.

Basicamente, o objeto `Trans` (Figura 29) é composto do identificador da transação, do *timestamp* de validação e pelos conjuntos de leitura e escrita. O identificador da transação é igual ao valor do identificador do *thread* que a transação executa. Tal atributo serve para verificar se uma dada transação é detentora de algum *lock*. O *timestamp* de validação é atribuído no início da execução da transação mas também pode ser estendido caso a validação do conjunto de leitura seja bem sucedida (Subseção 6.3.1).

O conjunto de leitura guarda um indicador de todos os atributos lidos pela transação. No conjunto de escrita são mantidos os atributos modificados. Como na CMTJava é aplicado o versionamento de dados tardio, a memória só é modificada após o *commit*. Qualquer computação especulativa é realizada sobre os *logs*. Quando a transação é efetivada, os valores do *log* de escrita são transpostos à memória.

A invocação da transação, além de alterar o estado do objeto `Trans`, retorna um valor do tipo `TResult` (Figura 28, Linha 5). `TResult` contém o status da execução. Se o status é `ACTIVE`, o sistema tentará efetivar os valores modificados na memória. Se o *commit* for bem sucedido, a transação retorna o resultado da execução da transação (Linha 13). A operação de *commit* é descrita na Subseção 6.3.2.

```
1  public static <A> A atomic(STM<A> c){
2
3    while(true){
4      Trans t = new Trans();
5      TResult<A> r = a.stm.invoke(t);
6      switch(r.state){
7        case RETRY:{
8          t.retry();
9          break;
10       }
11       case ACTIVE:{
12         if (t.commit()) {
13           return r.result;
14         }
15         break;
16       }
17       case ABORTED:{
18         t.rollback();
19         break;
20       }
21       default: break;
22     }
23   }
24 }
```

Figura 28: O método `atomic`

```
1  public class Trans {
2
3    Long validationStamp;
4    WriteSet writeSet;
5    ReadSet readSet;
6    Long transId;
7    ...
8  }
```

Figura 29: Classe `Trans`

Quando algum conflito relativo a concorrência das transações é identificado, o status da transação é `ABORT`. Transações abortadas são incondicionalmente reexecutadas. Entretanto, antes disso é necessário eliminar todo o trabalho até então realizado pela transação (Linha 18). Essa operação é conhecida como `rollback`, discutida na Subseção 6.3.3.

Se o status da execução da transação é `RETRY`, então a execução do programa chamou o método `retry`. Essa chamada é sempre definida pelo programador, tipicamente para evitar que uma transação seja reexecutada sem sucesso múltiplas vezes (Subseção 2.5.2). A implementação do `retry` é descrita na Subseção 6.3.4.

6.3.1 Validação

Para verificar se uma transação é válida, compara-se a versão corrente de todos os atributos lidos pela transação com a versão no momento da leitura. Essa informação da versão no momento da leitura é guardada no *log* de leitura da transação. Caso haja uma diferença entre as versões e a transação não esteja fazendo *commit*, isto é, de posse do *lock* de leitura, a transação não é válida. Na Figura 30 é mostrado o algoritmo de validação.

```
1  private boolean validate(){
2
3    for(Map.Entry<FieldInfo, Long> entry:readSet.log.entrySet()){
4      FieldInfo fieldInfo = entry.getKey();
5      Long entryVersion = entry.getValue();
6
7      boolean mark[] = {false};
8      Long version = (Long)fieldInfo.rlock.get(mark);
9      if(!version.equals(entryVersion)){
10       if(!mark[0] || !version.equals(transId)){
11         return false;
12       }
13     }
14   }
15   return true;
16 }
```

Figura 30: Algoritmo de validação

Em caso de sucesso na operação de validação, o *timestamp* de validação da transação é estendido para o valor corrente do contador global (Figura 31).

A validade da transação é verificada sempre que a versão de algum atributo acessado transacionalmente é maior que a *timestamp* de validação da transação. Isto é, quando uma outra transação foi efetivada na memória depois do início ou da última validação da transação.

```
1  public boolean extend(){
2
3    long globalTimestamp = VersionClock.getReadStamp();
4    if(validate()){
5      validationStamp = globalTimestamp;
6      return true;
7    }
8    return false;
9  }
```

Figura 31: A operação `extend`

6.3.2 Commit

Se a transação executou todas as suas operações e seu estado é ativo, então a mesma realiza a transposição dos valores do *log* de escrita para a memória principal. Esta operação é realizada no método `commit` (Figura 32). Ao efetivar suas computações na memória, a transação primeiramente verifica se realizou somente operações de leitura (Linha 3). Transações somente leitura podem ser efetivadas imediatamente, pois seu conjunto de leitura está sempre consistente. Isso é garantido pelas validações realizadas a cada operação.

Uma transação que realiza alguma escrita adquire os *locks* de leitura de todos os atributos alterados (Linha 7). Essa aquisição serve para que nenhuma operação de leitura nos atributos do conjunto de escrita seja realizada enquanto o *commit* é feito. Nesse caso, outra transação concorrente observaria um estado inconsistente da memória, violando o isolamento das transações. Isto é, as transações concorrentes só conseguirão ler algum dos valores modificados pela transação realizando o *commit*, quando todos os objetos alterados por ela forem alterados na memória. Os *locks* de escrita já encontram-se de posse da transação, pois são adquiridos de forma adiantada, a cada operação de escrita.

De posse dos *locks*, a transação incrementa o valor do contador de *commits* global e compara esse valor com sua versão local (Linhas 16 e 17). Se o novo valor do contador foi igual a versão da transação mais um, então nenhuma transação foi efetivada no sistema depois do inicio da execução ou da última validação. Se houver uma diferença maior entre estes valores, a transação valida seu *log*. Quando a validação não é bem sucedida, a transação libera todos os *locks* adquiridos (Linhas 19 e 20), cancelando a efetivação.

Em caso de sucesso na validação ou desta não ser necessária, a transação percorre seu conjunto de escrita atualizando os valores dos atributos na memória. Em seguida, libera os *locks* de escrita e leitura (Linha 25 à 33). Ao *lock* de leitura é atribuída a nova versão do contador de *commits* global. Por fim, são acordadas as transações bloqueadas nos atributos que a transação fazendo o *commit* atualizou (Linha 36). Isto é, aquelas que chamaram `retry` após ler algum desses atributos.

6.3.3 Rollback

Como vai adquirindo *locks* ao longo da execução, a transação necessita liberá-los em determinado momento. Essa liberação se dá tanto no caso em que a transação é abortada quanto após a efetivação das computações.

No método `rollback`, descrito na Figura 33, a transação libera todos os *locks* de escrita que detém.

6.3.4 Retry

A primitiva `retry` (Subseção 2.5.2) bloqueia uma transação, fazendo com que essa seja abortada e executada novamente, tão logo algum dado lido seja modificado por outra transação. A operação `retry` (Figura 34) valida a transação e adquire os *locks* de escrita dos atributos do conjunto de leitura. Se além de ler, a transação também escreveu em um atributo, a aquisição do *lock* de escrita não é necessária, pois a transação já detém o mesmo. Se todos *locks* de escrita forem obtidos, a transação se adiciona à fila de *threads* bloqueadas em cada

```
public <A> boolean commit(){

  if(writeSet.log.size()==0){
    return true;
  }

  for(FieldInfo field : writeSet.log.keySet()){
    Long ref = (Long)field.rlock.getReference();
    if(!field.rlock.compareAndSet(ref,transId,false, true)){
      releaseReadLocks();
      rollback();
      return false;
    }
  }

  Long writeVersion = VersionClock.getWriteStamp();
  if(writeVersion > validationStamp + 1){
    if(!validate()){
      releaseReadLocks();
      rollback();
      return false;
    }
  }

  for(Map.Entry entry : writeSet){

    FieldInfo<A> key = (FieldInfo<A>) entry.getKey();
    A newvalue = (A)entry.getValue();
    key.updateField.invoke(newvalue);

    key.rlock.set(writeVersion, false);
    key.wlock.set(null, false);
  }

  for(FieldInfo field : writeSet.log.keySet()){
    field.wakeupBlockedThreads();
  }

  return true;
}
```

Figura 32: Método `commit`

atributo do seu conjunto de leitura. Finalmente, a operação `retry` libera os *locks* de escrita adquiridos, assim permitindo o progresso de outras transações.

6.4 Resultados

Para avaliar o desempenho da implementação do novo sistema transacional, realizou-se um experimento com árvore rubro-negra. Esta seção primeiramente

```
1 public void rollback(){
2
3   for (FieldInfo f : writeSet.log.keySet()) {
4     f.wlock.set(null, false);
5   }
6 }
```

Figura 33: `Rollback`

descreve a estrutura de dados implementada e depois relata os testes realizados, juntamente com os resultados obtidos.

6.4.1 Árvore rubro-negra

Esta estrutura de dados é bastante utilizada para avaliar implementações de STM, dado seu bom desempenho nesses sistemas. A razão disso é que a travessia na árvore tem complexidade $O(log\, n)$, além da propriedade da árvore se manter balanceada. Com essas características, as árvores rubro-negras mostram-se estruturas propícias à paralelização.

Após o desenvolvimento do novo sistema para execução das transações, foi dado início a implementação da estrutura de árvore rubro-negra em CMTJava. A árvore rubro-negra é um tipo especial de árvore binária, onde os nodos são dotados de cor. Além disso, a árvore rubro-negra possui as seguintes propriedades:

1. Um nó é vermelho ou preto
2. A raiz é preta
3. Todas as folhas são pretas
4. Ambos os filhos de um nó vermelho são pretos
5. Todo caminho de um dado nó para qualquer de seus nós folhas descendentes contem o mesmo número de nós pretos

A base da estrutura de dados é definida pelo nó da árvore (Figura 35). O nó contém o dado, referências para os nós filhos e para o nó pai e a informação de cor.

Em linhas gerais, representa-se a árvore como mostrado na Figura 36. A árvore possui um nó raiz, o qual possibilita chegar em qualquer nó da árvore e um nó sentinela. Esse nó sentinela é usado para representar todos os nós folha da árvore, assim diminuindo a utilização de memória. Ou seja, todas as referência dos nós internos para os nós folha apontam para esse nó sentinela, o qual não guarda nenhum valor.

A Figura 37 apresenta um trecho da classe onde são definidas as operações na árvore. Em particular, descreve-se a inserção de um elemento. Uma inserção começa pela adição de um nó de forma semelhante a uma inserção em árvore binária. Essa operação utiliza um método auxiliar `insert(TreeNode root, Integer value)` que insere um valor na árvore,

```
1  public void retry(){
2
3    Block block = new Block();
4    boolean errorOnValidation = false;
5
6    if(!validate()){
7      errorOnValidation=true;
8    }else{
9      for(FieldInfo fieldInfo : readSet.log.keySet()){
10       boolean mark[] = {false};
11       Long version = (Long)fieldInfo.wlock.get(mark);
12       if(!mark[0] || !transId.equals(version)){
13         if(!fieldInfo.wlock.compareAndSet(null, transId, false,
14         true)){
15           errorOnValidation=true;
16           break;
17         }
18       }
19     }
20   }
21
22   if(!errorOnValidation){
23     for(FieldInfo f : readSet.log.keySet()) {
24       f.addBlockedThread(block);
25     }
26   }
27
28   for(FieldInfo fieldInfo : readSet.log.keySet()){
29     boolean mark[] = {false};
30     Long version = (Long)fieldInfo.wlock.get(mark);
31     if(mark[0] && transId.equals(version)){
32       fieldInfo.wlock.set(null, false);
33
34     }
35   }
36
37   if(!errorOnValidation){
38     block.block();
39   }
40 }
```

Figura 34: Método `retry`

partindo de um dado nó. Esse método retorna o nó criado ou um nó nulo, caso o valor já esteja na árvore. Se o nó foi inserido com sucesso, este é pintado em vermelho. Diferentemente de uma árvore binária, onde sempre adiciona-se nós nas folhas, na árvore rubro-negra as folhas não contém informação. Logo, foi inserido um nó vermelho na parte interna da árvore. Após inserir o novo nó, é necessário verificar se conjunto de propriedades da árvore rubro-negra foi violado. Nesse caso são efetuadas rotações e ajustes de cor até que a

```
public class TreeNode implements TObject{

  private Integer value;
  private TreeNode left;
  private TreeNode right;
  private TreeNode parent;
  private Color color;
  ...
}
```

Figura 35: Código para descrever os nós da árvore

```
public class TRedBlackTree implements TObject{

    private TreeNode root;
    private static TreeNode EMPTY;
    ...
}
```

Figura 36: Definição da árvore rubro-negra em CMTJava

árvore satisfaça todas suas propriedades. Após a remoção de um elemento, também pode ser necessário efetuar operações na árvore para garantir suas propriedades.

```
public class RedBlackTree extends TRedBlackTree{

  public STM<Boolean> insert(Integer value){

    return new STMDO{
      TreeNode empty <- this.getEMPTY();
      TreeNode root <- this.getRoot();
      TreeNode node <- this.insert(root, value);
      if(node == null){
        STMRTS.stmReturn(false)
      }else {
        node.setColor(Color.RED);
        fixRBTree(node);
        STMRTS.stmReturn(true)
      }
    };
  }

  public STM<Boolean> remove(Integer value){...}
  public STM<Boolean> find(Integer value){...}
  ...
}
```

Figura 37: Operações sobre a árvore

6.4.2 Experimento realizado

O teste consistiu em realizar operações, sorteadas aleatoriamente, em uma árvore rubro-negra pré-inicializada. Foram efetuadas operações de travessia na árvore (busca por elementos) e de atualização (inserção e remoção de elementos). Nos testes, variou-se o número de *threads* e a relação entre operações de busca e atualização. A variação do número de *threads* visa verificar a escalabilidade dos sistemas transacionais testados: a implementação descrita neste trabalho e o sistema predecessor, com base no algoritmo TL2. As alterações no número de operações de atualização tem o objeto de verificar o comportamento dos sistemas, a medida em que aumentam os conflitos.

Além disso, foram analisados 3 diferentes tamanhos iniciais da árvore: 10.000, 100,000 e 500.000 nós. Essa variação no tamanho inicial busca avaliar como os sistemas se comportam a medida que o tamanho dos conjuntos de leitura das transações aumentam. O tempo de inicialização da árvore não foi considerado no experimento.

Para mensurar o desempenho, verificou-se o número de operações realizadas, nos diferentes cenário descritos acima. Os testes foram executados utilizando um computador com processador de 4 núcleos com *HyperThreading*, 8 GB de memória RAM e sistema operacional Linux.

Nas Figuras 38, 39 e 40 são apresentados os resultados dos testes realizados em árvores inicializadas com 10.000, 100.000 e 500.000 elementos, respectivamente. Para cada caso, varia-se o número de operações de atualização. Os cenários testados são: 0% (somente buscas), 20% e 50% de operações de atualização na árvore.

O novo sistema transacional é mais eficiente nos casos onde o tamanho da árvore é maior. Entretanto, quando o tamanho da árvore é pequeno, o número de conflitos diminui e a detecção adiantada acaba sendo menos efetiva. Para o menor tamanho inicial da árvore analisado, o algoritmo desenvolvido tem desempenho levemente inferior ao sistema de execução com base no algoritmo TL2.

A medida que se utilizam estruturas de dados maiores, a contenção no sistema aumenta e com ela o número de conflitos cresce. Nessa conjuntura, o sistema transacional desenvolvido consegue realizar mais operações por unidade de tempo. Em particular, o sistema de detecção de conflitos com invalidação mista proposto mostrou-se até duas vezes mais eficiente para execução de transações de busca em uma árvore rubro-negra com 50000 elementos. Isso se deve ao fato do algoritmo utilizar uma política mais eficiente para detecção de conflitos, que evita a execução de transações condenadas a abortar.

6.5 Considerações Finais

Neste capítulo foi descrito em detalhes o novo sistema transacional da linguagem CMTJava. Esse novo sistema utiliza uma estratégia híbrida para detecção de conflitos, denominada invalidação mista. Também relatou-se os resultados de desempenho obtidos com um experimento com árvores rubro-negras. No próximo e último capítulo abordam-se as conclusões desta dissertação de mestrado.

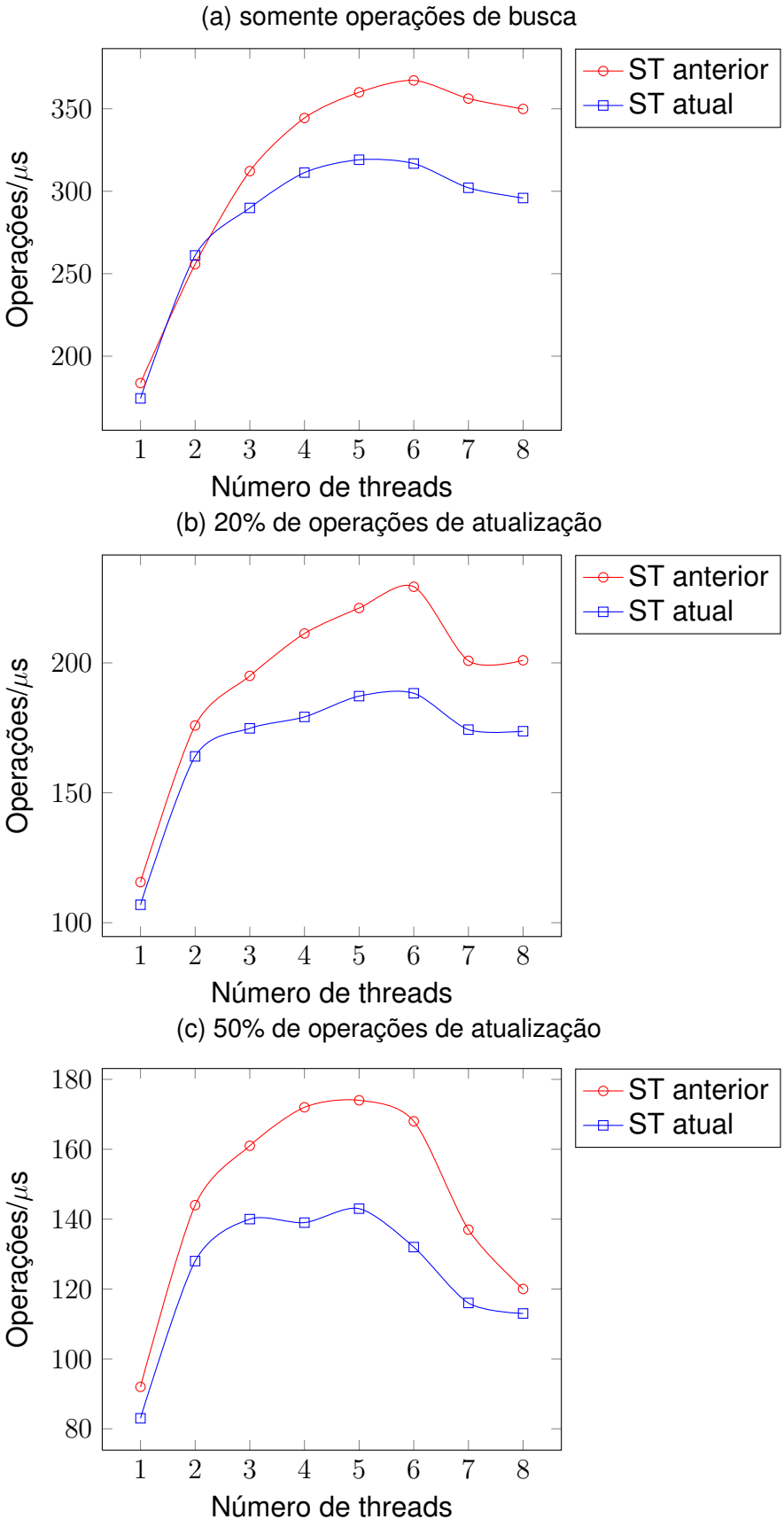

Figura 38: Número de operações realizadas em uma árvore rubro-negra inicializada com 10^4 elementos

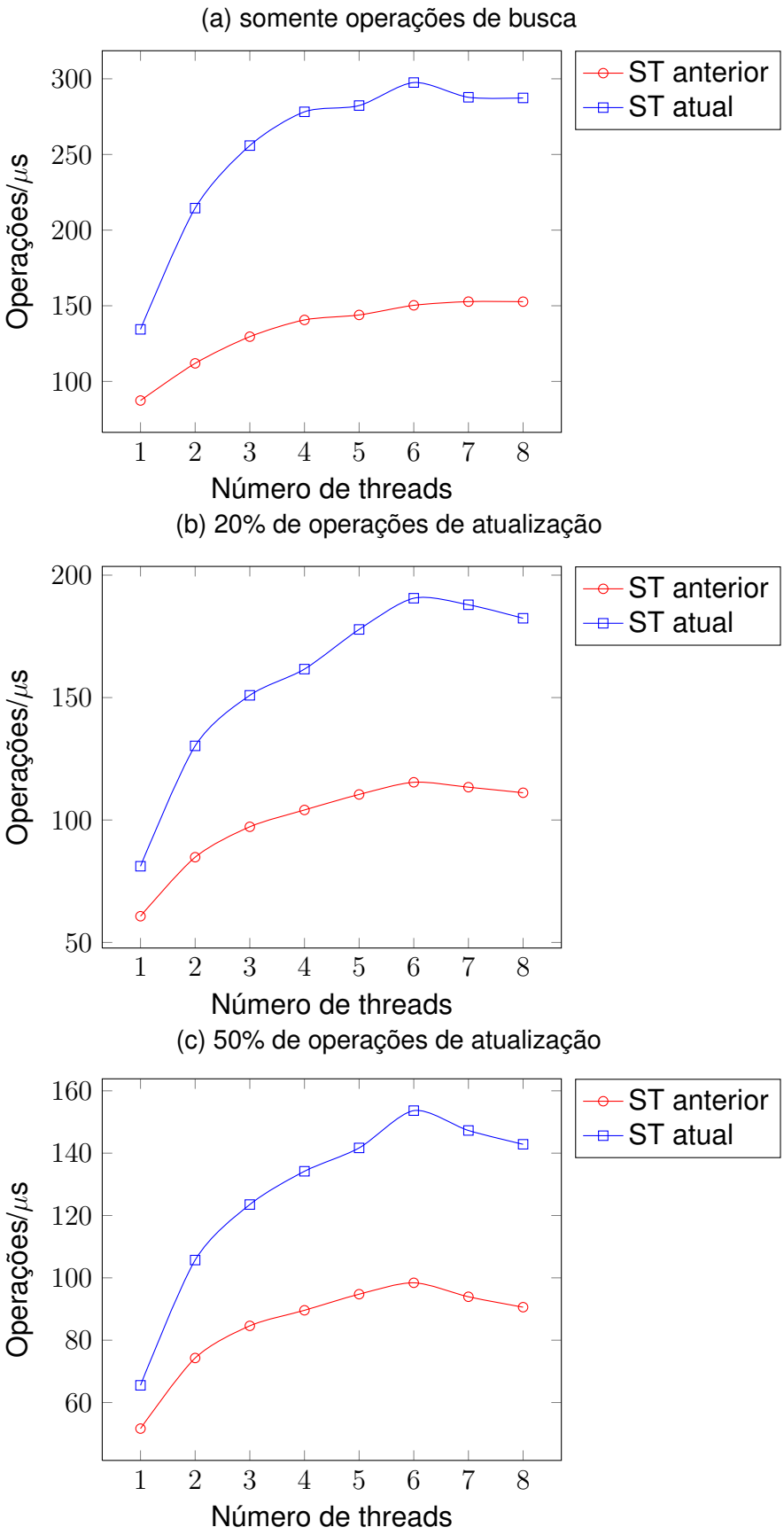

Figura 39: Número de operações realizadas em uma árvore rubro-negra inicializada com 10^5 elementos

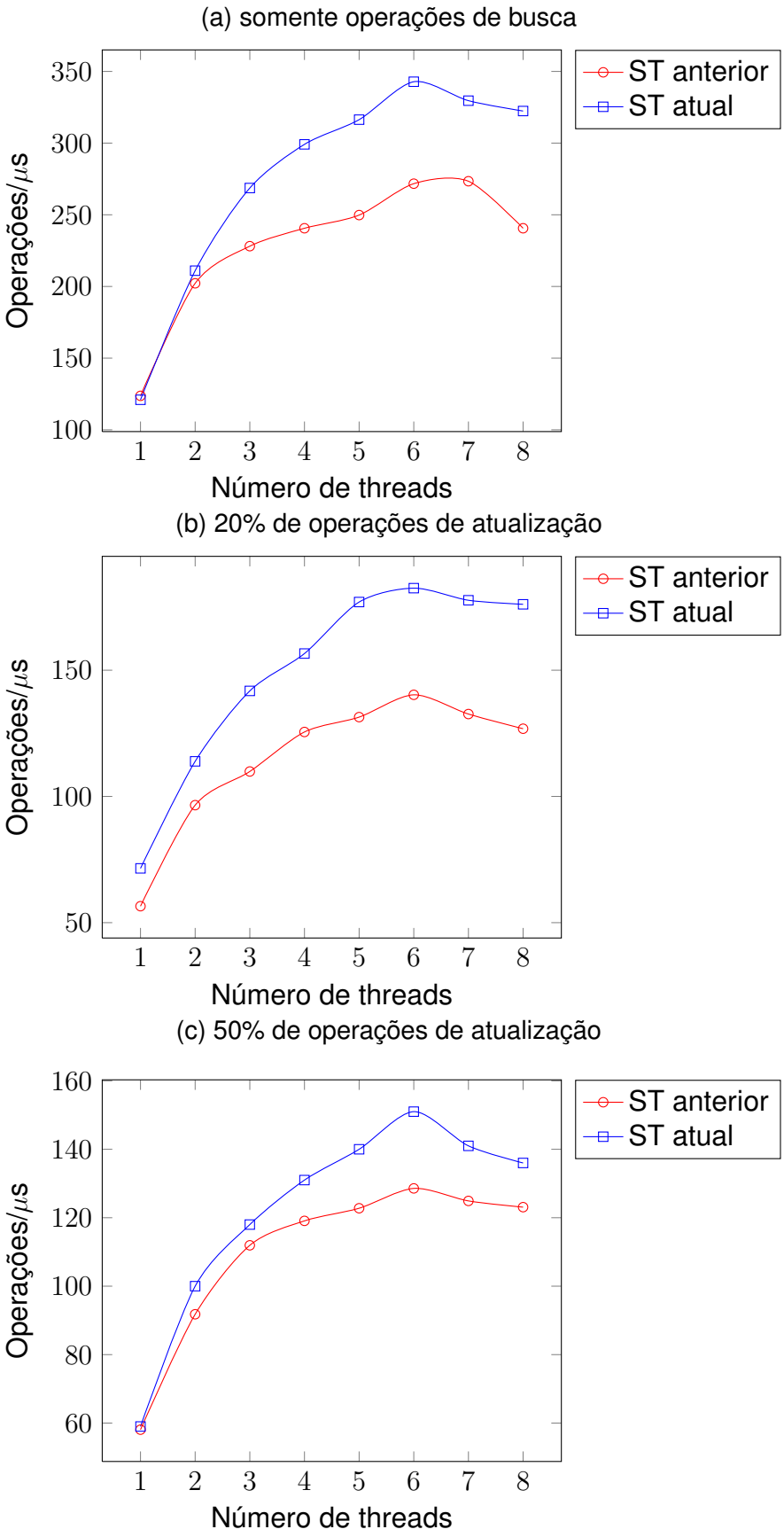

Figura 40: Número de operações realizadas em uma árvore rubro-negra inicializada com $5 * 10^5$ elementos

7 CONCLUSÕES

Neste trabalho foram discutidos os principais desafios da programação *multicore* e também algumas das diversas desvantagens da utilização de bloqueios como mecanismo de sincronização de um programa concorrente. Além disso, mostrou-se que a abstração de transações pode tornar a tarefa de escrita de código para arquiteturas multinúcleo bem menos complexa. Na programação utilizando esta abordagem, todo o acesso à memória compartilhada é realizado dentro de transações que são executadas respeitando as propriedades da atomicidade e isolamento.

As memórias transacionais tem se mostrado como um modelo de programação promissor em comparação aos mecanismos baseados em exclusão mútua. Embora algoritmos usando primitivas de baixo nível pra sincronização obtenham melhor desempenho, impõem uma grande complexidade de desenvolvimento. Memórias transacionais fornecem uma melhor relação entre escalabilidade e esforço de implementação. Ou seja, MT propicia uma abstração de mais alto nível para a escrita de programas concorrentes, deixando o programador concentrado no algoritmo, ao invés de na sincronização da execução.

Apesar dessa técnica facilitar a codificação de software concorrente, a programação desses sistemas não é simples. É necessário realizar, de forma transparente ao programador, tarefas que seriam responsabilidade deste. Isto tudo da maneira mais eficiente possível. No corrente trabalho foram discutidas as principais características de implementação de memória transacional. Também foram examinadas as consequências de tais requisitos no desempenho desses sistemas. Ficou evidenciado que a escolha de determinada estratégia em detrimento a outra está diretamente relacionada as características das aplicações como: tamanho das transações, carga de trabalho, predominância de leituras ou escritas, dentre outras. Logo, nenhuma estratégia supera as demais em todos cenários. Para finalizar a revisão bibliográfica, apresentou-se algumas implementações de sistemas de memória transacional em software, onde foram analisadas os requisitos de implementação utilizadas por cada uma.

O foco do trabalho foi caracterizar o projeto de um sistema transacional para a linguagem CMTJava, através da adaptação do algoritmo de (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009). Além disso, foi implementada uma estrutura de dados recorrente na avaliação de STMs: a árvore rubro-negra. A mesma foi utilizada para validar e comparar o sistema transacional desenvolvido com a implementação anterior. De acordo com o experimento, o sistema transacional desenvolvido supera o anterior na maioria dos casos testados. Em particular,

o algoritmo para coordenação das transações implementado se beneficia em cenários onde a taxa de contenção é maior e as transações são grandes.

Com a difusão das transações para a programação concorrente, a tendência é o aumento da complexidade das estruturas de dados e do tamanho das transações executadas. Isso faz com que os sistemas de memória transacional sejam cada vez mais exigidos no que diz respeito a desempenho. Nesse contexto, o trabalho desenvolvido mostra-se adequado a esta nova realidade, pois tem bom desempenho em cenários complexos de execução.

A seguir são discutidos os trabalhos futuros e citados os artigos publicados durante o desenvolvimento deste projeto.

7.1 Trabalhos futuros

Apesar do bom desempenho obtido com a sistema transacional desenvolvido, otimizações podem tornar esse desempenho ainda melhor. A principal delas é a adição de políticas mais eficientes de gerenciamento de contenção. Atualmente, o sistema transacional utiliza a estratégia tímida, que aborta e imediatamente reexecuta a transação que detecta um conflito. A implantação do gerenciador de contenção de duas fases (DRAGOJEVIć; GUERRAOUI; KAPALKA, 2009), deve melhorar a vazão da linguagem. Esta abordagem altera a política de gerenciamento de acordo com o número de operações realizadas pela transação.

Outro trabalho de importância para o projeto CMTJava é o desenvolvimento de mais aplicações teste e avaliação do desempenho destas na linguagem.

7.2 Artigos publicados

Ao longo do desenvolvimento deste projeto, alguns trabalhos foram publicados descrevendo as experiências adquiridas:

- Mônadas em Java. Workshop-Escola de Informática Teórica – WEIT. 2011.
- Projeto de um esquema de detecção de conflitos para CMTJava. XIII Encontro de Pós-Graduação - ENPOS. UFPel. 2011.
- Implementação de um esquema de invalidação mista na linguagem CMTJava. 12ª Escola Regional de Alto Desempenho - ERAD. Fórum de Pós-Graduação. 2012.
- Sistemas transacionais na linguagem CMTJava. XIV Encontro de Pós-Graduação - ENPOS. UFPel. 2012.
- Avaliação do novo sistema de execução de transações para CMTJava. 13ª Escola Regional de Alto Desempenho - ERAD. Fórum de Pós-Graduação. 2013.

REFERÊNCIAS

BANDEIRA, R. L.; DU BOIS, A. R.; PILLA, M. **Compilador para a linguagem CMTJava**. Monografia de conclusão de curso — Universidade Federal de Pelotas. 2011.

BANDEIRA, R. L.; DU BOIS, A. R.; PILLA, M.; VIZZOTTO, J. K. Mônadas em Java. In: WORKSHOP-ESCOLA DE INFORMáTICA TEóRICA, 2011. **Anais...** [S.l.: s.n.], 2011. p.105–108.

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional Locking II. In: IN PROC. OF THE 20TH INTL. SYMP. ON DISTRIBUTED COMPUTING, 2006. **Anais...** [S.l.: s.n.], 2006.

DRAGOJEVIć, A.; GUERRAOUI, R.; KAPALKA, M. Stretching transactional memory. **Sigplan Notices**, [S.l.], v.44, 2009.

DU BOIS, A. R.; ECHEVARRIA, M. A Domain Specific Language for Composable Memory Transactions in Java. In: DSL '09: PROCEEDINGS OF THE IFIP TC 2 WORKING CONFERENCE ON DOMAIN-SPECIFIC LANGUAGES, 2009, Berlin, Heidelberg. **Anais...** Springer-Verlag, 2009. p.170–186.

ECHEVARRIA, M. G. **Uma Linguagem de Domínio Específico para Programacão de Memórias Transacionais em Java**. 2010. Dissertação (Mestrado em Ciência da Computação) — Universidade Católica de Pelotas.

EDDON, G.; HERLIHY, M. Software Transactional Objects. **TRAMP 2007 - IBM Workshop on Transactional Memory and Programming Technologies**, [S.l.], 2007.

FELBER, P.; FETZER, C.; RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 13., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.237–246. (PPoPP '08).

FRASER, K.; HARRIS, T. Concurrent programming without locks. **ACM Transactions on Computer Systems (TOCS)**, [S.l.], v.25, n.2, may 2007.

HARRIS, T.; CRISTAL, A.; UNSAL, O. S.; AYGUADE, E.; GAGLIARDI, F.; SMITH, B.; VALERO, M. Transactional Memory: An Overview. **IEEE Micro**, Los Alamitos, CA, USA, v.27, p.8–29, 2007.

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

HARRIS, T.; MARLOW, S.; PEYTON-JONES, S.; HERLIHY, M. Composable memory transactions. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.48–60. (PPoPP '05).

HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.21, n.2, p.289–300, 1993.

HUDAK, P.; WADLER, P.; BRIAN, A.; FAIRBAIRN, B. J.; FASEL, J.; HAMMOND, K.; HUGHES, J.; JOHNSSON, T.; KIEBURTZ, D.; NIKHIL, R.; JONES, S. P.; REEVE, M.; WISE, D.; YOUNG, J. Report on the programming language Haskell: A non-strict, purely functional language. **ACM SIGPLAN Notices**, [S.l.], v.27, 1992.

JONES, S. P. **Beautiful Concurrency**. [S.l.]: O'Reilly Media, Inc., 2007.

KNIGHT, T. An architecture for mostly functional languages. In: LFP '86: PROCEEDINGS OF THE 1986 ACM CONFERENCE ON LISP AND FUNCTIONAL PROGRAMMING, 1986, New York, NY, USA. **Anais...** ACM, 1986. p.105–112.

MOGGI, E. Notions of computation and monads. **Information and Computation**, [S.l.], v.93, p.55–92, 1991.

MOORE, K. E.; BOBBA, J.; MORAVAN, M. J.; HILL, M. D.; WOOD, D. A. Logtm: Log-based transactional memory. In: HPCA '06: PROC. 12TH INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 2006. **Anais...** [S.l.: s.n.], 2006. p.254–265.

NEWBERN, J. **All About Monads**. Disponível em: <http://www.haskell.org/haskellwiki/All_About_Monads> Acesso em: 22 de Janeiro de 2013.

PARR, T. **The Definitive ANTLR Reference**: Building Domain-Specific Languages. [S.l.]: Pragmatic Bookshelf, 2007. (Pragmatic Programmers).

RIEGEL, T.; FELBER, P.; FETZER, C. A lazy snapshot algorithm with eager validation. In: IN PROCEEDINGS OF THE 20TH INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING (DISC'06, 2006. **Anais...** [S.l.: s.n.], 2006. p.284–298.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente. In: MINICURSOS DO VIII WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD 2007, 2007. **Anais...** [S.l.: s.n.], 2007.

SCHERER III, W. N.; SCOTT, M. L. Advanced contention management for dynamic software transactional memory. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.240–248. (PODC '05).

SHAVIT, N.; TOUITOU, D. Software transactional memory. In: PODC '95: PROCEEDINGS OF THE FOURTEENTH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 1995, New York, NY, USA. **Anais...** ACM, 1995. p.204–213.

SPEAR, M. F.; MARATHE, V. J.; III, W. N. S.; SCOTT, M. L. Conflict detection and validation strategies for software transactional memory. In: IN PROC. OF THE 20TH INTL. SYMP. ON DISTRIBUTED COMPUTING, 2006. **Anais...** [S.l.: s.n.], 2006.

WELC, A.; JAGANNATHAN, S.; HOSKING, A. L. Transactional monitors for concurrent objects. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 2004. **Proceedings...** Springer-Verlag, 2004. p.519–542. (Lecture Notes in Computer Science, v.3086).