4

Programação de Máquinas *Multicore* usando Memórias Transacionais em *Software*

Timóteo Matthies Rico – tmrico@inf.ufpel.edu.br[1]
Rafael de Leão Bandeira – rlbandeira@inf.ufpel.edu.br[2]
André Rauber Du Bois – dubois@inf.ufpel.edu.br[3]
Maurício Lima Pilla – pilla@inf.ufpel.edu.br[4]

Resumo:

Memórias Transacionais são um paradigma relativamente recente para programação concorrente. As Memórias Transacionais permitem ao programador abstrair detalhes da execução e focar no desenvolvimento de algoritmos de mais alto nível, deixando para o sistema de execução a resolução de conflitos de acessos a variáveis compartilhadas. Neste curso, uma introdução aos principais conceitos de Memórias Transacionais será proporcionada. As principais questões de implementação e as construções utilizadas serão discutidas. Como explicação prática da aplicação de Memórias Transacionais, o curso concluirá com exemplos de programas concorrentes usando este mecanismo.

[1]Mestrando em Ciência da Computação na UFPEL.

[2]Mestrando em Ciência da Computação na UFPEL.

[3]Doutor em Ciência da Computação pela Heriot-Watt University (2005). Atualmente é professor na UFPEL. Sua pesquisa relaciona-se com linguagens de programação e processamento paralela e distribuída.

[4]Doutor em Ciência da Computação (UFRGS, 2004). Atualmente é professor na UFPEL, atuando nas áreas de processamento paralelo e distribuído e arquiteturas de computadores.

"Concurrent programming is notoriously tricky. [...] systems built using locks are difficult to compose without knowing about their internals. [...] Our main war-cry is ***compositionability****: a programmer can control atomicity and blocking behaviour in a modular way that respects abstraction barriers."*

T. Harris, S. Marlow, S. P. Jones e M. Herlihy

Composable Memory Transactions, PPoPP'05

4.1. Introdução

A computação paralela permite 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, tais como a necessidade de garantir a exclusão mútua das tarefas executadas paralelamente.

Memórias Transacionais é uma nova abstração que visa facilitar a programação paralela. Sua utilização permite ao programador focar em determinar *onde* a atomicidade é necessária, ao invés de preocupar-se com 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

O termo "Memória Transacional" foi cunhado em 1993 por Herlihy e Moss para designar *"uma nova arquitetura para microprocessadores que objetiva tornar a sincronização livre de bloqueio tão eficiente (e fácil de usar) quanto técnicas convencionais baseadas em exclusão mútua"* [HER 93]. Este termo define qualquer mecanismo de sincronização que utilize o conceito de transação para coordenar acessos aos dados compartilhados.

Embora memórias transacionais tenham recentemente se popularizado, a iniciativa de usar transações como forma de estruturação de programas concorrentes existe há mais de três décadas [LOM 77]. Todavia, o conceito não se popularizou na época principalmente por não se conhecer uma implementação eficiente. Além disso, os mecanismos de sincronização baseados em *locks* explícitos eram bastante pesquisados naquela época, contribuindo para que o trabalho de Lomet não obtivesse maior notoriedade [BAL 2009].

Este curso pretende apresentar uma introdução às memória transacionais, desde os aspectos de *design* até exemplos do seu uso para resolver problemas clássicos de programação concorrente. O público-alvo inclui alunos de graduação e pós-graduação na área de Computação que já tenham o referencial teórico básico de programação concorrente e paralela.

A estrutura deste curso divide-se da seguinte forma. Inicialmente, as principais características, propriedades e vantagens do uso de memórias transacionais são discutidas na Seção 4.2.. Após, são apresentados aspectos de linguagem e semântica na Seção 4.3.. Questões de implementação são abordadas na Seção 4.4.. Na Seção 4.5., exemplos de problemas clássicos de programação concorrente são abordados usando memórias transacionais. Finalmente, as conclusões são discutidas na Seção 4.6..

4.2. O que são Memórias Transacionais?

Programação concorrente não é exatamente uma atividade simples. Além de todas as questões que um programador deve enfrentar quando do desenvolvimento de um sistema sequencial, programas concorrentes possuem novos desafios. A concorrência no acesso a variáveis em sistemas com memória compartilhada pode levar a situações de não-determinismo, ou seja, onde uma determinada computação possa ter um resultado inesperado. Este não é um comportamento desejado pelo programador e, neste caso, é chamado de *condição de corrida*.

Para manter a semântica de um programa sequencial em um ambiente de programação concorrente, condições de corrida devem ser evitadas através de algum mecanismo de *sincronização*. Inicialmente, a sincronização foi realizada através da obtenção e liberação de *locks*, também chamados de *mutexes*. Um processo ou *thread* testa uma variável de forma atômica antes de acessar as variáveis desejadas (a *seção crítica*).

Em relação ao uso de *locks*, a memória transacional apresenta as seguintes vantagens [MCK 2010]:

- **Facilidade de programação**: A programação torna-se mais fácil porque o programador não tem a responsabilidade de garantir a sincronização, e sim em especificar quais blocos de código devem ser executados atomicamente, cabendo ao sistema transacional a implementação do mecanismo de sincronização.

- **Escalabilidade**: Transações que acessem um mesmo dado para leitura podem ser executadas concorrentemente. Também podem ser executadas de

modo paralelo as transações que modifiquem partes distintas de uma mesma estrutura de dados. Essa característica tem como vantagem garantir que mais desempenho seja obtido com o aumento do número de processadores porque o nível de paralelismo exposto é maior.

- **Composabilidade**: Transações suportam naturalmente a composição de código. Para criar uma nova operação baseando-se em outras já existentes, basta invocá-las dentro de uma nova transação. O sistema transacional irá garantir que tais operações sejam executadas de forma atômica.

Outras técnicas foram desenvolvidas para prover uma abstração maior na programação concorrente, como semáforos e monitores, mas estas ou não alcançam um nível de abstração tão elevado como memórias transacionais (semáforos), ou não oferecem uma potencialidade de explorar o paralelismo disponível (monitores). De qualquer forma, não abordam o problema da *composabilidade*, uma das principais vantagens apontadas pelos simpatizantes de memórias transacionais.

4.2.1. Conceito de Transação

O conceito de **transação**, criado e desenvolvido em sistemas de banco de dados, refere-se a um conjunto de operações que formam uma única unidade de trabalho lógica, satisfazendo as propriedades ACID (**A**tomicidade, **C**onsistência, **I**solamento e **D**urabilidade) [SIL 99].

A **propriedade de atomicidade** define que as instruções pertencentes ao escopo da transação serão todas efetivadas com sucesso ou todas descartadas. Quando a transação for efetivada com sucesso o resultado da execução é permanente, e diz-se que a transação sofreu *commit*. Quando ocorrer falha ou conflito na transação, diz-se que essa sofreu *abort*, e os valores pertencentes à execução da transação serão descartados.

A **propriedade de consistência** assegura a integridade dos dados, garantindo que uma transação levará o sistema de um estado consistente (pré-transação) a outro estado consistente (pós-transação).

O **isolamento** significa que transações concorrentes não poderão verificar resultados intermediários produzidos por outras transações, de modo que o resultado da execução de diversas transações seja equivalente ao resultado da execução dessas mesmas em ordem serial.

A **propriedade de durabilidade** indica que as mudanças efetivadas por transações são permanentes e resistentes a eventuais falhas no sistema.

O conceito de transação no contexto de memórias transacionais é basicamente o mesmo, com exceção para a propriedade de durabilidade. O conceito de

durabilidade pode ser ignorado em memórias transacionais, pois em seu contexto os dados serão armazenados em memória volátil, ao contrário de sistemas de banco de dados onde os dados são armazenados em memória persistente.

Se ao final da execução da transação não houver nenhum conflito com as demais transações em execução, as modificações são aplicadas à memória. Na ocorrência de conflitos, as modificações decorrentes da transação são descartadas e a transação é executada novamente. Por deixar a detecção de conflitos a cargo do sistema transacional, o programador apenas delimita as transações (seção crítica) e o sistema de execução transacional garante que essas sejam executadas de forma atômica e isolada.

4.2.2. Classificação de TM

Existem diversos modelos de memória transacional [HAR 2010]. O modelo de **Memória Transacional em *Hardware* (HTM)** 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)**, o sistema de execução transacional é totalmente implementado em *software*. Os sistemas de STM são o foco deste curso. 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. As **Memória Transacional Híbrida (HyTM)** suportam a execução da HTM mas voltam-se ao *software* quando os recursos de *hardware* são esgotados. As **Memórias Transacionais em *Software* assistidas por *Hardware* (HaSTM)** combinam STM com algum suporte arquitetural em *hardware* para acelerar partes da implementação em *software*.

4.3. Linguagem e Semântica

No contexto de memórias transacionais, as seguintes construções geralmente são usadas: *(i) atomic*, *(ii) retry* e *(iii) orElse*.

4.3.1. Construção *atomic*

A construção *atomic* representa o bloco atômico, o qual é responsável por delimitar o escopo que deve ser executado por uma transação [HAR 2003]. Uma grande vantagem do bloco atômico é que não é necessário criar manualmente variáveis específicas para controlar e bloquear a seção crítica de um programa, diferentemente de outras construções como *locks*. O sistema transacional será responsável por garantir a sincronização do bloco, deixando a encargo do programador apenas especificar quais serão os blocos atômicos e não *em como* sincronizá-los.

Na Figura 4.1 mostra-se um exemplo de uso do bloco atômico, sendo delimitado pela palavra *atomic* (linha 2 até linha 6).

```
1 public void deposito(double quantia) {
2   atomic{
3     if (quantia > 0) {
4       this.saldo = this.saldo + quantia; //Seção Crítica
5     }
6   }
7 }
```

Figura 4.1: Bloco atômico.

A composição de transações é ilustrada na Figura 4.2, através de um método que transfere um valor entre duas contas bancárias. Pelo fato da operação ser efetuada de forma atômica e isolada, é garantido que nenhuma outra transação verá o estado intermediário no qual o valor sacado de uma conta (linha 3) ainda não tenha sido depositado na outra (linha 4). Caso haja conflito em alguma das operações (saque ou depósito), o sistema transacional abortará toda transação, descartando as alterações em ambas as operações.

```
1 public void transferencia(ContaBancaria contaOrigem, ContaBancaria
      contaDestino, Double quantia){
2   atomic{
3     contaOrigem.saque(quantia);
4     contaDestino.deposito(quantia);
5   }
6 }
```

Figura 4.2: Composição em memória transacional.

4.3.2. Construção *retry*

A construção *retry* permite que uma transação seja cancelada, desfazendo todas as suas ações intermediárias [HAR 2005]. Quando algum dado lido pela transação que chamou *retry* é modificado, ela é reexecutada. Por exemplo, considerando a remoção de elemento de um *buffer*. Sempre que um *buffer* estiver vazio a transação invoca a construção *retry*, fazendo com que a transação seja cancelada e reexecutada quando a variável itens for modificada por outra transação.

A fim de comparar uma construção transacional com outra abordagem de sincronização, o mesmo exemplo de remoção de elemento de um *buffer* será implementado em Java utilizando monitores, conforme mostra na Figura 4.4. O uso da primitiva *synchronized* indica que o processo/*thread* que invocar o método remover deverá obter o bloqueio associado ao objeto *buffer*, antes de prosseguir com a operação de remoção. A aquisição e liberação de bloqueios é implícita em Java, quando o método for qualificado como *synchronized*. O método *wait* serve para garantir que não seja removido um elemento de um *buffer* vazio e o método *notifyAll* notifica os outros processos/*threads* que um elemento foi removido do *buffer*.

```
1 public int remover() {
2   atomic {
3     if (itens == 0){
4       retry;
5     }
6     itens--;
7     return buffer[itens];
8   }
9 }
```

Figura 4.3: Remoção de elemento de *buffer* codificada com a construção *retry*. Fonte: [RIG 2007].

A partir dos dois exemplos mostrados nas Figuras 4.3 e 4.4, observa-se um problema comum com bloqueios: baixa concorrência e, consequentemente, baixo desempenho. Como um bloqueio associado ao objeto *buffer* é adquirido ao invocar o método remover, qualquer outro método do objeto que eventualmente seja invocado será bloqueado até que o método libere o bloqueio. Ou seja, duas ou mais operações não acontecerão concorrentemente no *buffer* mesmo quando há uma possibilidade de paralelismo. Por exemplo, é possível que uma operação de inserção ocorra simultaneamente a uma de remoção se ambas acessarem elementos disjuntos no *buffer*. Portanto, o uso de transações consegue explorar mais paralelismo, pois a detecção de conflitos é feita de forma dinâmica (em tempo de execução).

```
1  public synchronized int remover() {
2    int resultado;
3
4    while (itens == 0){
5          wait();
6    }
7    itens--;
8    resultado = buffer[itens];
9    notifyAll();
10   return resultado;
11 }
```

Figura 4.4: Remoção de elemento de buffer codificada em Java com monitor. Fonte: [RIG 2007].

4.3.3. Construção *orElse*

A construção *orElse* permite que transações sejam compostas como alternativas para execução, de modo que somente uma transação será executada entre várias [HAR 2005].

O *orElse* recebe como argumento duas transações. A segunda transação será executada somente se a primeira chamar *retry*. Se ambas chamam *retry* então todo o bloco atômico é executado novamente. Por exemplo, supondo um sistema que remova elementos de um *buffer* no qual o método de remoção pode ser bloqueado (invocar *retry*) se o *buffer* estiver vazio, assim como o exemplo na Figura 4.3. Um exemplo interessante usando *orElse* para esta situação é mostrado na Figura 4.5. Neste código, o sistema transacional tentará remover elementos entre dois *buffers*. Caso o *buffer* 1 estiver vazio o sistema tentará remover o elemento do *buffer* 2. Caso os dois *buffers* estiverem vazios, o bloco atômico será bloqueado até que ao menos um dos *buffers* contenha um elemento para poder ser removido.

```
1  public void exemploOrElse() {
2    atomic {
3      { x = buffer1.remover(); }
4      orElse
5        { x = buffer2.remover(); }
6    }
7  }
```

Figura 4.5: Uso da construção *orElse*. Fonte: [RIG 2007].

4.4. Detalhes de Implementação

As primeiras implementações de memórias transacionais em *software* são não-bloqueantes [HER 2003, HAR 2003, FRA 2004, HAR 2005, MAR 2005]. Este tipo de implementação exclui qualquer uso de *locks*, já que esses podem induzir estado de espera [BAL 2009]. No entanto, constatou-se através de pesquisas que as abordagens de STMs não-bloqueantes apresentam baixo desempenho se comparadas à utilização de *locks* explícitos [ENN 2006]. A partir desse importante resultado a maioria das implementações propostas passaram a utilizar *locks*, de maneira implícita.

O sistema transacional garante a atomicidade e isolamento baseado em dois mecanismos chave: *(i)* versionamento de dados; e *(ii)* detecção e resolução de conflitos.

4.4.1. Versionamento de dados

O versionamento de dados lida com o gerenciamento das diferentes versões dos dados (originais e especulativas) acessados por uma transação. A versão original corresponde ao valor do dado antes do início da transação. A versão especulativa representa o valor intermediário sendo trabalhado pela transação. Caso a transação seja efetivada, os valores especulativos tornam-se os valores correntes e são armazenados, e os valores originais são descartados. Do contrário, descarta-se os especulativos e mantém-se os originais, e a transação é reexecutada [BAL 2009].

Existem duas formas de realizar o versionamento de dados: *(i)* adiantado ou direto (*eager versioning* ou *direct update*); e *(ii)* atrasado ou deferido (*lazy versioning* ou *deferred update*) [HAR 2010].

No **versionamento de dados adiantado**, os valores especulativos são armazenados diretamente na memória, enquanto que os valores originais são armazenados em um *undo log*. No caso de uma transação ser abortada, os dados que foram gravados na memória inicialmente deverão ser convertidos para suas versões antigas (originais), através do *undo log*.

No **versionamento de dados atrasado**, o armazenamento de novos dados é realizado em um *buffer* local, e a memória continua com os valores originais. Os dados especulativos são transferidos do *buffer* para a memória somente quando a transação for efetivada.

Ilustra-se na Figura 4.6 ambos os tipos de versionamento de dados. Inicialmente, o conteúdo da variável X na memória corresponde ao valor 100 e os respectivos *undo log* e *buffer* estão vazios. Quando uma transação atribui o valor 77 à variável X, no versionamento adiantado altera-se imediatamente o conteúdo

da memória e armazena-se o valor original localmente (*undo log*), enquanto no versionamento atrasado somente salva-se o valor especulativo no *buffer*. No momento da efetivação (ou *commit*) da transação, a única ação necessária no versionamento adiantado é invalidar o *undo log*, pois o dado especulativo já está na memória.

Já no versionamento atrasado, é necessário primeiramente mover o valor especulativo do *buffer* para a memória do sistema. Uma situação inversa acontece em caso de cancelamento da transação. Neste caso, no versionamento adiantado precisa-se restaurar as mudanças efetuadas na memória, movendo para esta o valor original presente no *undo log*. No versionamento atrasado só é necessário descartar os valores especulativos.

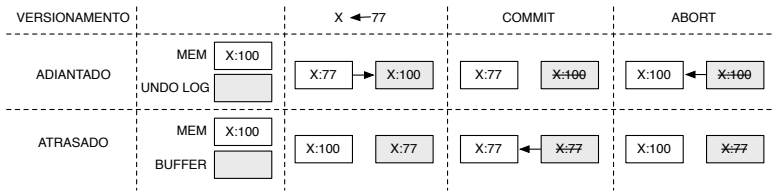

Figura 4.6: Exemplo ilustrando versionamento adiantado e atrasado. Fonte: [BAL 2009].

Conclui-se que o versionamento adiantado torna-se mais eficiente em casos em que a transação é confirmada, pois os dados especulativos já estão na memória. Por outro lado, quando a transação é cancelada, deve-se restaurar os valores originais para a memória, que estão armazenados no *undo log*.

No versionamento atrasado a situação é oposta, visto que transações confirmadas necessitam transferir os dados especulativos do *buffer* para a memória, enquanto que transações canceladas não necessitam de nenhuma operação.

4.4.2. Detecção de conflitos

Para detecção de conflitos, uma transação geralmente mantém um conjunto de leitura (*read set*) com os endereços dos dados lidos, e um conjunto de escrita (*write set*) com os endereços dos dados que foram alterados.

Há um conflito entre duas ou mais transações quando a intersecção entre o conjunto de leitura e o conjunto de escrita de transações diferentes é não vazia. Em outras palavras, quando no mesmo intervalo de tempo duas ou mais transações acessam o mesmo dado compartilhado e ao menos um dos acessos é de escrita [BAL 2009].

A validação dos conjuntos de leitura/escrita é uma tarefa crítica em sistemas de STMs. Com objetivo de realizar este processo o mais rápido possível, as implementações de STMs geralmente utilizam técnicas susceptíveis a respostas como falsos positivos, gerando conflitos desnecessários [HAR 2010].

O sistema transacional efetua a detecção de conflitos baseando-se na **granularidade de detecção**. Essa granularidade pode ser em três níveis [BAL 2009]:

- A **granularidade por nível de objetos** pode reduzir o *overhead* em termos de espaço e tempo para detecção de conflitos. No entanto, esse nível permite detectar falsos conflitos, ou seja, casos em que o sistema detectará conflitos quando duas transações operam em diferentes partes de um mesmo objeto.
- A **granularidade por nível de palavras** elimina a detecção de falsos conflitos, por outro lado, necessita-se de maior custo e espaço para a detecção.
- A **granularidade por nível de linha de *cache*** provê um acordo entre a frequência de detecção de falsos conflitos e o *overhead* em termos de tempo e espaço. No entanto, necessita-se de alterações na estrutura de *hardware* para prover a *cache* transacional, alterando seu protocolo de coerência.

Assim como no versionamento de dados, há dois modos de detecção de conflitos: *(i)* adiantado/pessimista (*eager conflict detection* ou *pessimistic conflict detection*); e *(ii)* atrasado/otimista (*lazy conflict detection* ou *optimistic conflict detection*) [HAR 2010].

No modo de **detecção de conflitos adiantado**, verifica-se a ocorrência de conflito no momento em que uma posição de memória é acessada. Desta forma pode-se evitar que uma transação condenada a abortar execute desnecessariamente, mas por outro lado, pode-se cancelar transações, que dependendo do progresso de outras, poderiam ser confirmadas com sucesso.

Para exemplificar este modo de detecção, os cenários mostrados na Figura 4.7 serão considerados. No caso 1, T1 e T2 manipulam conjuntos de dados disjuntos e portanto não há nenhum conflito. No caso 2 tem-se uma operação de escrita depois de uma leitura. A transação que escreveu (T2) faz com que a outra transação (T1) seja cancelada. O caso 3 mostra a situação em que, após ser cancelada, T1 volta a executar e por sua vez cancela a transação T2. Neste exemplo, pode-se notar que as transações T1 e T2 estão em estado de *livelock*, desta forma, a efetivação de tais transações é indefinida.

Na **detecção atrasada** o sistema detecta se há conflito somente no momento de efetivação da transação. Assim, permite-se que transações conflitantes prossigam na esperança do conflito não se efetivar de fato. De modo a exemplificar a detecção atrasada, os cenários mostrados na Figura 4.8 serão considerados. No

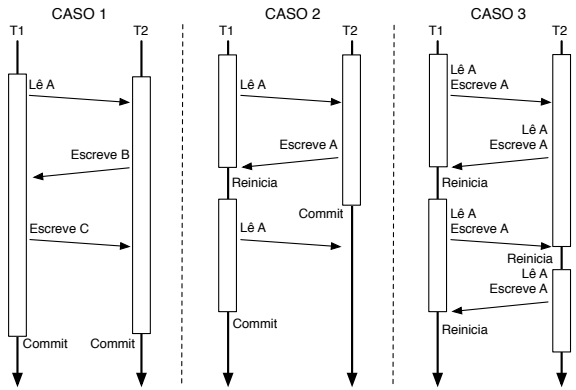

Figura 4.7: Detecção de conflitos em modo adiantado em cenários transacionais. Fonte: [RIG 2007].

primeiro caso, as transações acessam um conjunto de dados disjuntos, não ocasionando conflitos. No caso 2, T2 lê uma variável escrita por T1. Quando T1 sofre efetivação, T2 nota o conflito e é reiniciada. No caso 3 não ocorre nenhum conflito, pois T1 apenas lê uma variável que T2 está escrevendo e esta sofre efetivação após T1. O caso 4 mostra a situação em que, após ser cancelada, T1 volta a executar. No entanto, diferentemente da detecção adiantada, este caso não gera situação de *live-lock*, porém, poderia acontecer situação de postergação indefinida ou *starvation*. Se T1 fosse muito longa poderia ser sempre cancelada por outras transações e nunca conseguir efetivar suas alterações.

4.4.3. Resolução de conflitos

Em caso de conflito, um componente do sistema transacional, denominado gerenciador de contenção, é invocado para resolver os conflitos entre as transações e assim garantir o progresso do sistema. Este componente implementa uma ou mais políticas de resolução de conflitos, as quais ditam caso deva-se abortar a transação que detectou o conflito, as outras transações conflitantes, e caso atrasar ou não a reexecução das transações abortadas [HAR 2010a].

Algumas propostas de memórias transacionais possuem um método fixo para gerenciamento de contenção, ao passo que outras propostas tratam o problema como um aspecto modular do sistema, podendo ser alterado para melhorar o desem-

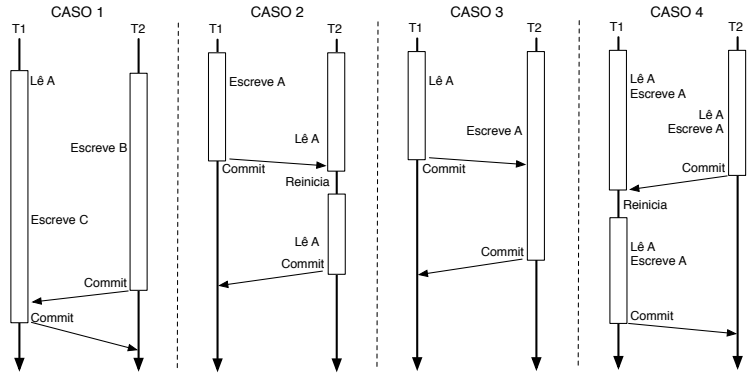

Figura 4.8: Detecção de conflitos modo atrasado. Fonte: [RIG 2007].

penho de um programa sob determinada carga de trabalho [KRO 2009]. Dentre as diversas políticas existentes, destacam-se:

- **Tímido:** a transação que detectou o conflito é cancelada e reexecutada imediatamente. É o mecanismo mais simples de gerenciamento de contenção.
- **Guloso:** o gerenciador de contenção guloso atribui um marcador de tempo (*timestamp*) a cada transação no início de sua primeira tentativa de execução transacional. Em caso de conflito, a transação que possuir o *timestamp* mais antigo prevalece. No entanto, se uma transação mais nova está bloqueada pela mais antiga e detecta que esta também está bloqueada, esperando por um recurso adquirido por outra transação, então a transação mais nova cancela a mais antiga e prossegue a execução.
- ***Backoff*:** a transação cancelada aguarda por um período (linear ou exponencial) de acordo com o número de seus cancelamentos para então ser reexecutada.
- ***Delay*:** a transação abortada será reexecutada somente após o *lock* que causou o conflito ser liberado.
- ***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, aborta imediatamente outra transação com menor prioridade. No entanto, se a prioridade de é menor, a transação espera por um período fixo de tempo.

4.4.4. Níveis de isolamento

O nível de isolamento está relacionado à interação entre código transacional e código não transacional. Existem dois tipos de isolamento: **atomicidade forte** (*strong atomicity*) e **atomicidade fraca** (*weak atomicity*) [BAL 2009].

No modelo com **atomicidade forte**, o acesso a um dado compartilhado fora de uma transação é consistente com os acessos efetuados ao mesmo dado dentro da transação.

Em contrapartida, no modelo com **atomicidade fraca**, o acesso a um mesmo dado compartilhado dentro e fora de uma transação pode causar condição de corrida e portanto o resultado não é determinístico.

4.5. Programando com Memórias Transacionais

Esta Seção apresenta a solução de dois problemas clássicos de concorrência usando memórias transacionais: *O jantar dos Filósofos* e o problema do *Produtor-Consumidor*. Nos exemplos não foi usada nenhuma implementação específica de TM, apenas as primitivas discutidas na Seção 4.3.. Os exemplos podem ser facilmente convertidos para alguma biblioteca de TM, como por exemplo TL2 [DIC 2006], TinySTM [FEL 2008] e SwissTM [DRA 2009].

4.5.1. Exemplo de produtor-consumidor

O problema do produtor-consumidor pode ser descrito como o compartilhamento de um *buffer* de armazenamento. Um conjunto de processos ou *threads* produtores insere itens no *buffer*, enquanto que outro conjunto de consumidores retira itens.

Implementações usando *locks* e semáforos são complexas, pois devem abordar a sincronização para as seções críticas e ao mesmo tempo garantir que não vão ocorrer *deadlocks* entre diferentes sincronizações. Implementações com monitores

são simples, mas serializam todos os acessos ao *buffer*, o que não permite explorar eficientemente o paralelismo dos *multicores* modernos.

Na implementação com memória transacional da Figura 4.9 são definidas transações para acessar as variáveis compartilhadas que definem a capacidade do *buffer* e o próprio *buffer*. O sistema de execução é responsável por determinar as transações que podem ser executadas em paralelo, liberando o programador dessa preocupação.

```
1  class Buffer{
2    private Vector buffer;
3    private int primeiro;
4    private int ultimo;
5    private int itens;
6    private int capacidade;
7
8    public Buffer(){
9      //implementação suprimida
10   }
11
12   public void insere(Item a){
13     atomic{
14       if (itens==capacidade){
15         retry;
16       }
17       ultimo = (ultimo+1)%capacidade;
18       buffer.insertAt(ultimo, a);
19       itens++;
20     }
21
22   }
23
24   public Item retira(){
25     Item b;
26
27     atomic{
28       if (itens==0){
29         retry;
30       }
31       b = (Item) buffer.getElement(primeiro);
32       primeiro = (primeiro+1)%capacidade;
33       itens--;
34     }
35   }
36 }
```

Figura 4.9: Produtor e consumidor usando *atomic* e *retry*.

4.5.2. Jantar dos Filósofos

O *Jantar dos Filósofos* é um problema clássico de programação concorrente que foi proposto por Dijkstra em 1965. Cinco filósofos estão sentados em uma mesa redonda e cada um possui um prato de comida em sua frente. Para poder comer, um filósofo necessita de dois garfos, só que existe somente um garfo entre cada prato. Toda a vez que sente fome, o filósofo tenta pegar dois garfos, um à direita e outro à esquerda para comer. Se conseguir pegar os dois garfos então o filósofo come um pouco, coloca os garfos novamente na mesa e volta a pensar.

Esse problema possui uma solução muito simples usando memórias transacionais, como pode ser visto na Figura 4.10. Primeiramente o filósofo tenta adquirir os dois garfos, fazendo uma chamada à primitiva *retry* caso algum esteja ocupado. Depois de comer, basta liberar os garfos que haviam sido adquiridos no primeiro bloco atômico.

```
1  class Filosofo{
2
3    Garfo garfoDireito;
4    Garfo garfoEsquerdo;
5
6    Philosopher(Garfo gd, Garfo ge)
7      garfoDireito = gd;
8      garfoEsquerdo = ge;
9    }
10
11   public void run(){
12     while(true){
13       atomic{
14               if(garfoDireito.get() || garfoEsquerdo.get()){
15                       retry;
16       }
17               else
18               {
19         garfoDireito.set(true);
20                 garfoEsquerdo.set(true);
21       }
22     }
23
24     System.out.println("Comendo....");
25
26     atomic{
27               garfoDireito.set(false);
28               garfoEsquerdo.set(false);
29     }
30   }
31 }
```

Figura 4.10: Filósofos jantantes com transações.

4.6. Conclusões

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.

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, estes impõem uma grande complexidade de desenvolvimento. Memórias transacionais fornecem uma melhor relação entre escalabilidade e esforço de implementação. Ou seja, memórias transacionais propiciam 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.

Os autores se mostram otimistas em relação à adoção de memórias transacionais para os futuros sistemas *manycores*, onde a complexidade na exploração de paralelismo deve aumentar ainda mais.

4.7. Bibliografia

[BAL 2009] BALDASSIN, A. J. **Explorando memória transacional em software nos contextos de arquiteturas assimétricas, jogos computacionais e consumo de energia**. 2009. Tese (Doutorado em Ciência da Computação) — Universidade Estadual de Campinas, Campinas, SP, Brasil.

[DIC 2006] DICE, D.; SHALEV, O.; SHAVIT, N. Transactional locking ii. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING (DISC 2006), 20., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.194–208.

[DRA 2009] DRAGOJEVIC, A.; GUERRAOUI, R.; KAPALKA, M. Stretching transactional memory. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2009., 2009, New York, NY, USA. **Proceedings...** ACM, 2009. p.155–165. (PLDI '09).

[ENN 2006] ENNALS, R. **Software transactional memory should not be obstruction-free**. [S.l.]: Intel Research Cambridge Tech Report, 2006. (IRC-TR-06-052).

[FEL 2008] 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).

[FRA 2004] FRASER, K. **Practical lock-freedom**. [S.l.]: University of Cambridge, 2004. (Relatório Técnico 579).

[HAR 2005] HARRIS, T. et al. Composable memory transactions. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.48–60.

[HAR 2003] HARRIS, T.; FRASER, K. Language support for lightweight transactions. In: ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 18., 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.388–402. (OOPSLA '03).

[HAR 2010] HARRIS, T.; LARUS, J.; RAJWAR, R. Transactional memory. **Synthesis Lectures on Computer Architecture**, p.1–263, 2010.

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

[HER 2003] HERLIHY, M. et al. Software transactional memory for dynamic-sized data structures. In: PRINCIPLES OF DISTRIBUTED COMPUTING, 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.92–101. (PODC '03).

[HER 93] HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In: OF THE 20TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1993, New York, NY, USA. **Proceedings...** ACM, 1993. p.289–300. (ISCA '93).

[KRO 2009] KRONBAUER, F.; RIGO, S. Experimentos com gerenciamento de contenção em uma memória transacional com suporte em software. **Anais do X Simpósio em Sistemas Computacionais WSCAD-SSC**, São Paulo, SP, Brasil, p.44–51, 2009.

[LOM 77] LOMET, D. Process structuring, synchronization, and recovery using atomic actions. In: ACM SIGPLAN NOTICES, 1977. **Anais...** [S.l.: s.n.], 1977. v.12, n.3, p.128–137.

[MAR 2005] MARATHE, V. J.; Scherer III, W. N.; SCOTT, M. L. Adaptive software transactional memory. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING, 19., 2005, Cracow, Poland. **Proceedings...** [S.l.: s.n.], 2005. p.354–368. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[MCK 2010] MCKENNEY, P. E. et al. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. **SIGOPS Oper. Syst. Rev.**, New York, NY, USA, v.44, p.93–101, August 2010.

[RIG 2007] RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. **Memórias transacionais**: uma nova alternativa para programação concorrente. [S.l.]: In Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, 2007.

[SIL 99] SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Sistema de banco de dados, 3o edição**. [S.l.]: Editora Pearson, 1999.