Memória Transacional

Carla Négri Lintzmayer
RA 134042
Instituto de Computação
Universidade Estadual de Campinas
carla0negri@gmail.com

Eduardo Theodoro Bogue
RA 134049
Instituto de Computação
Universidade Estadual de Campinas
eduardotheodoro@gmail.com

Maycon Sambinelli
RA 134071
Instituto de Computação
Universidade Estadual de Campinas
msambinelli@gmail.com

RESUMO

Memória transacional (TM) é um mecanismo para controle de concorrência, análogo a transações em banco de dados, para controlar o acesso a dados compartilhados em sistemas concorrentes, sendo uma alternativa para a sincronização pelos mecanismos tradicionais. Neste contexto, uma transação é um pedaço de código que executa uma sequência de leituras e escritas que ocorrem em um único instante de tempo e cujo estado não é visível para as demais. Este trabalho apresenta conceitos do funcionamento de memórias transacionais, seus principais tipos de implementação, que fornecem tanto suporte em hardware (Memória Transacional em Hardware) quanto em software (Memória Transacional em Software), e uma discussão final indicando que, apesar dos benefícios provenientes das TMs, sua utilização na prática ainda não é viável devido às limitações que elas oferecem.

Palavras-chave

Memória Transacional; Memória Transacional em Hardware; Memória Transacional em Software; Memória Transacional Híbrida

1. INTRODUÇÃO

A cada lançamento de uma nova família de processadores, as aplicações sequenciais podiam sentir um aumento de desempenho proporcionado pelo aumento de velocidade e melhorias arquiteturais [26]. Em meados de 2004, fatores arquiteturais e tecnológicos limitaram o ritmo de crescimento do desempenho dos processadores [13].

No lado arquitetural, o ganho através do paralelismo em nível de instrução (ILP) atingiu seu limite; conforme mostrado por Wall [32], aplicações típicas possuem paralelismo de instruções reduzido na casa de quatro instruções, o que limita o ganho através de processadores super-escalares quando estes são capazes de disparar mais do que quatro instruções por ciclo de *clock* [26]. No lado tecnológico, problemas como superaquecimento e dissipação de potência inviabilizaram o aumento da frequência dos processadores.

A solução da indústria para resolver esses problemas foi criar os chamados "*chips multicore*", "*chips* multiprocessadores" ou "*many cores*" [23]. Essa nova geração de processadores opta por explorar o paralelismo em nível de *thread* (TLP, de *thread-level parallelism*).

No TLP, a *thread* é a unidade de concorrência, a comunicação se dá através de memória compartilhada e a sincronização é realizada através de algum mecanismo de exclusão mútua, tal como *locks*, *semáforos*, *monitores* ou *barreiras*. Porém a exclusão mutua pode gerar alguns problemas, como, por exemplo, quando um recurso está bloqueado e todas as outras *threads* ficam impedidas de utilizar este recurso, há uma serialização no processamento (já que uma *thread* impede a execução de outras), o que por sua vez diminui a vantagem de ter várias *threads* trabalhando simultaneamente e consequentemente diminui o desempenho. Além da diminuição de desempenho, o uso de exclusão mutua pode trazer problemas mais graves tais como *deadlocks*, *inversão de prioridade* e *convoying* [14].

Devido a esses problemas, poucos programas paralelos eficientes foram construídos [13], pois algoritmos paralelos são muito mais difíceis de se formular e de provar sua corretude do que os algoritmos sequenciais. Além disto, os modelos de programação, as linguagens de programação e as ferramentas disponíveis para a programação paralela são inferiores às disponíveis para programação sequencial [17].

A chave para a deficiência do paralelismo em nível de *thread* é a falta de abstração [17], que é extremamente importante para fazer com que os desenvolvedores foquem seus esforços em resolver o que realmente é importante para o problema. As linguagens de programação modernas suportam bem a abstração de computação sequencial através de funções e procedimentos, porém, elas não possuem um mecanismo capaz de abstrair a computação paralela.

Enquanto o paralelismo tem sido um problema difícil para a programação em geral, sistemas de banco de dados têm alcançado sucesso explorando o paralelismo [10], executando diversas *queries* simultaneamente em vários processadores sempre que possível, sem a necessidade do desenvolvedor se preocupar com o paralelismo [17].

Este sucesso é devido principalmente ao uso de transações. Uma transação especifica uma semântica na qual uma computação executa como se fosse a única acessando o banco de dados. Desta forma, o banco de dados garante que irá produzir resultados previsíveis independente da ordem de execução das transações.

Transações oferecem um comprovado mecanismo de abstração para o sistema de banco de dados, que o permite construir buscas paralelas de uma forma simples. Este mecanismo gerou a proposta de inserir transações nos modelos

de programação paralela, com o objetivo de simplificar a escrita de programas paralelos, além de solucionar os problemas ocasionados pelo modelo tradicional de sincronização. Assim surgiram os sistemas de memórias transacionais.

O restante deste trabalho encontra-se dividido da seguinte forma. Na Seção 2, são apresentados os fundamentos e principais conceitos relacionados à memória transacional. As Seções 3 e 4 detalham as duas principais linhas de pesquisa, que são as abordagens para implementação em software e hardware, respectivamente. A Seção 5 aborda brevemente uma terceira linha de pesquisa, que utiliza transações em hardware e em software simultaneamente. Na Seção 6, é apresentada uma discussão sobre o uso de memória transacional na prática. Por fim, a Seção 7 conclui este trabalho.

2. MEMÓRIA TRANSACIONAL

Esta seção aborda os fundamentos e conceitos principais relacionados à memória transacional, apresentando exemplos e, de forma geral, mecanismos que devem ser providos por uma implementação.

2.1 Transações em Bancos de Dados

Apesar do paralelismo ser um problema difícil para a programação de propósito geral, dependendo muito do programador, os bancos de dados conseguem explorá-lo de maneira satisfatória [10]. Geralmente, bancos de dados são utilizados por vários usuários ao mesmo tempo, realizando várias ações de consulta e/ou atualização dos dados (como ocorre, por exemplo, em sistemas de agências bancárias, companhias de transporte ou supermercados). Um sistema gerenciador de banco de dados deve garantir que duas ou mais ações, mesmo que sejam sobre exatamente as mesmas entradas de dados, possam ocorrer simultaneamente sem que haja comprometimento desses dados e do resultado final esperado. Mais importante ainda nessas aplicações, é que o usuário do banco não precisa se preocupar com esse paralelismo.

No meio dos sistema de bancos de dados está o conceito de *transação*. Segundo Elmasri e Navathe [9], uma transação é "um programa em execução ou processo que inclui um ou mais acessos ao banco de dados, tais como leitura ou atualização dos registros do banco de dados". Uma definição mais interessante é dada por Harris *et al.* [13], que diz que uma transação é "uma sequência de ações que parecem indivisíveis e instantâneas para um observador externo". Assim, ou uma transação termina corretamente, e nesse caso diz-se que ela *efetiva* (do inglês, *commit*), ou então a transação *aborta* e nenhuma ação é realizada.

A vantagem das transações quando se trata de paralelismo é que uma transação especifica a semântica do programa no qual a computação executa como se ela fosse a única acessando o banco de dados e permite a *serializabilidade*, ou seja, elas executam simultaneamente, com certa permissão de interação, mas o resultado é o mesmo que seria se elas fossem executadas sequencialmente (em série).

Ainda no contexto dos bancos de dados, uma transação tem quatro propriedades que devem ser garantidas pelo sistema gerenciador do banco de dados [9]. Essas propriedades são conhecidas como ACID (devido às suas iniciais):

Atomicidade é a propriedade que garante que uma transação é atômica: ou ocorre (todas as suas ações constituintes terminam) ou não ocorre e, nesse último caso, não deixa "vestígios" de que foi executada (nenhuma ação constituinte acontece);

Consistência é a propriedade relacionada com o conceito de invariância: após uma transação, um sistema que era consistente deve ser mantido consistente;

Isolamento é a propriedade que garante que as transações não irão interferir umas com as outras enquanto estão executando;

Durabilidade é a propriedade que garante que quando a transação é efetivada, seu resultado será "permanente".

2.2 Transações em Memória

Será que o modelo de programação usado pelos bancos de dados pode ser usado no modelo de programação paralela em geral para facilitar a estruturação e sincronização dos processos? Em 1977, Lomet [18] observou que sim. A idéia básica é utilizar as propriedades das transações para coordenar leituras e escritas concorrentes que ocorrem em dados compartilhados de um sistema paralelo concorrente.

Diferente dos bancos de dados, que armazenam os dados em disco, programas normalmente armazenam seus dados em memória. Assim, a abstração do modelo dos bancos de dados para o modelo de programação paralela recebe um nome diferente, apresentado em 1993 por Herlihy e Moss [14]: *memória transacional* (TM, de *transactional memory*).

TMs podem ser vistas como sendo não-duráveis, pois os dados na memória não são mantidos após o término do programa [13]. Assim, para a TM considera-se apenas as outras 3 propriedades das transações: ACI. Atomicidade para garantir que toda computação ou terminará e efetivará seus resultados ou será abortada sem modificar nada; consistência para garantir um programa correto com dados em um estado previsível, consistente; e isolamento para garantir que os resultados obtidos são os mesmos que seriam obtidos caso não houvesse concorrência.

Pode-se dividir a pesquisa em memória transacional em duas principais linhas [26]. Uma é chamada de memória transacional em software (STM, de *software transactional memory*) [28, 29], que implementa o sistema de execução transacional em software, utilizando bibliotecas e compiladores, por exemplo. Outra é chamada de memória transacional em hardware (HTM, de *hardware transactional memory*) [2, 14], que implementa o sistema de execução transacional por meio de suporte arquitetural, modificando cache, barramento ou protocolos de coerência, por exemplo. Uma terceira possibilidade é chamada de memória transacional híbrida (HyTM, de *hybrid transactional memory*), que visa obter melhor desempenho utilizando características das duas anteriores. STM, HTM e HyTM são abordadas posteriormente neste trabalho, nas Seções 3, 4 e 5, respectivamente.

2.3 Utilizando Memória Transacional

Pode-se comparar o uso de memória transacional *versus* o uso de mecanismos de sincronização tradicionais através de um exemplo simples: operações em agências bancárias. Supondo que exista um *array* denominado `conta`, no qual cada

entrada `conta[i]` é o valor armazenado na conta de número `i`, três procedimentos simples podem ser utilizados quando se trata de movimentações nessas contas:

(i) `Deposito(i, valor)`: soma a quantia `valor` à `conta[i]`;
(ii) `Retirada(i, valor)`: retira a quantia `valor` de `conta[i]`;
(iii) `Transfere(i_orig, i_dest, valor)`: move a quantia `valor` da conta `i_orig` para a conta `i_dest`.

Obviamente, um mesmo número de conta pode ser acessado por mais de um cliente do banco ao mesmo tempo; isso implica que um mecanismo de sincronização precisa estar presente. Uma possível implementação para esses três procedimentos utilizando *locks* é a seguinte:

```
 1 Deposito(i, valor) {
 2     lock(conta[i]);
 3     conta[i] = conta[i] + valor;
 4     unlock(conta[i]);
 5 }
 6 Retirada(i, valor) {
 7     lock(conta[i]);
 8     conta[i] = conta[i] - valor;
 9     unlock(conta[i]);
10 }
11 Transfere(i_orig, i_dest, valor) {
12     lock(conta[i_orig]);
13     lock(conta[i_dest]);
14     Retirada(i_orig, valor);
15     Deposito(i_dest, valor);
16     unlock(conta[i_dest]);
17     unlock(conta[i_orig]);
18 }
```

Um problema evidente desta implementação é que, se existem duas chamadas à função `Transfere`, por exemplo, das formas (i) `Transfere(1, 5, 300)` e (ii) `Transfere(5, 1, 400)`, e (i) é interrompida antes de executar a linha 15 para que (ii) seja executada (escalonamento), então (ii) vai obter o *lock* da conta `5` na linha 14 e vai precisar do *lock* da conta `1` na linha 15, enquanto (i) está com o *lock* da conta `1` e precisa do *lock* da conta `5`. Nessa situação tem-se um *deadlock* e as duas não conseguem mais executar.

Em Harris *et al.* [13] é apresentada uma interface de operações transacionais que podem ser suportadas tanto por HTM quanto por STM:

- Operações para gerenciamento de transações:
  - `void StartTx()`: começa uma transação na *thread* atual;
  - `bool CommitTx()`: tenta efetivar a transação, retornando *true* se for bem sucedida e *false* se falhar;
  - `void AbortTx()`: aborta explicitamente.
- Operações para acesso de dados:
  - `T ReadTx(T *address)`: faz leitura no endereço `address` de um valor do tipo `T`. O conjunto de locais que uma transação faz leitura é o seu *conjunto de leitura*;
  - `void WriteTx(T *address, T v)`: escreve no endereço `address` um novo valor `v` do tipo `T`. O conjunto de locais que uma transação faz escrita é o seu *conjunto de escrita*.

Assim, uma possível implementação utilizando TM para o procedimento `Deposito` citado anteriormente é:

```
1 Deposito(i, valor) {
2     do {
3         StartTx();
4         valor_atual = ReadTx(conta[i]);
5         WriteTx(conta[i], valor_atual + valor);
6     } while (!CommitTx());
7 }
```

onde `StartTx()` e `CommitTx()` delimitam o escopo da transação. Uma abordagem bastante utilizada [13] é prover blocos atômicos:

```
1 Deposito(i, valor) {
2     atomic {
3         conta[i] = conta[i] + valor;
4     }
5 }
```

e o compilador se encarrega de substituir `atomic{}` pelas operações delimitadoras e introduzir as operações `ReadTx` e `WriteTx` onde for necessário. As outras operações ficam:

```
 6 Retirada(i, valor) {
 7     atomic {
 8         conta[i] = conta[i] - valor;
 9     }
10 }
11 Transfere(i_orig, i_dest, valor) {
12     atomic {
13         Retirada(i_orig, valor);
14         Deposito(i_dest, valor);
15     }
16 }
```

2.4 Controles de Concorrência

De forma geral, existem dois mecanismos principais que uma implementação de TM precisa prover: (i) garantia de isolamento entre transações, detectando e resolvendo possíveis conflitos para que elas pareçam estar executando sequencialmente (serializabilidade); e (ii) gerenciamento do trabalho que uma transação faz enquanto ela executa (note que esse trabalho é uma tentativa – em outras palavras, a transação pode abortar). Abordagens para o primeiro mecanismo serão tratadas nesta e na próxima seção e abordagens para o segundo serão tratadas na seção 2.6.

É necessário controlar a concorrência pois acessos a dados podem causar conflitos. Existem três eventos relacionados a esses conflitos, que podem ocorrer em tempos diferentes mas não em ordens diferentes [13]:

Ocorrência Um conflito ocorre quando duas transações realizam operações escrita-escrita ou escrita-leitura no mesmo dado;

Detecção A detecção ocorre quando o sistema de TM determinar que o conflito ocorreu. Este tópico é detalhado na próxima seção;

Resolução O conflito é resolvido quando o sistema de TM realizar alguma ação para evitá-lo.

De forma geral, o controle de concorrência tem duas abordagens [13]:

1. Pessimista: considera que os três eventos ocorrem ao mesmo tempo, isto é, quando a transação está para acessar um local, o sistema detecta o conflito e o resolve. Isso permite que uma transação peça um bloqueio do dado que ela irá utilizar. Deve-se tomar um cuidado extra nessa abordagem para evitar que haja *deadlock*;
2. Otimista: permite que o conflito ocorra e depois realiza sua detecção e resolução; isso permite que várias transações "erradas" continuem executando. É claro que o sistema de TM deve detectar e resolver o(s) conflito(s) antes que a transação efetive. Nessa abordagem, por sua vez, deve-se tomar cuidado para evitar *livelock*.

2.5 Detecção de Conflitos

Para garantir a serializabilidade, uma solução simples é permitir que somente uma transação seja executada em um determinado instante. No entanto, este comportamento não é apropriado por não explorar nenhum paralelismo. Com a concorrência, aparecem os conflitos.

Conforme já visto, um dos principais mecanismos que uma implementação de TM deve prover é garantir a detecção e resolução dos possíveis conflitos que as transações geram enquanto elas executam. Conforme visto na seção anterior, a detecção de conflitos está diretamente relacionada com o controle de concorrência.

Utilizando um controle de concorrência pessimista, a detecção do conflito é de certa forma simples e direta, pois um bloqueio pode ser adquirido apenas quando ele não está em um estado de conflito com outra *thread* [13].

Na abordagem de controle de concorrência otimista existem diversas técnicas para a detecção de conflitos descritas na literatura [13]. Pode-se classificar a maioria destas técnicas para detecção de conflitos em três dimensões ortogonais:

Granularidade Por exemplo, em sistemas HTM, os conflitos podem ser detectados no nível de linhas de cache, ou em sistemas STM, eles podem ser detectados no nível de objetos completos;

Tempo A detecção pode ocorrer

(i) quando a transação declara sua intenção em acessar um dado (detecção de conflitos ansiosa);

(ii) quando a transação chama uma operação de validação (que verifica se os dados lidos ou atualizados por ela foram modificados por outra transação); ou

(iii) quando a transação tenta efetivar, devendo procurar por conflitos em todos os locais que ela acessou (detecção de conflitos preguiçosa).

Tipo de acesso Se uma transação ativa T_a lê de um local onde a transação ativa T_b escreveu, tem-se um conflito. Usando a detecção de conflitos por *tentativa*, ele é detectado assim que ocorre a leitura. Usando a detecção por *efetivação*, ele é detectado apenas quando T_a ou T_b efetivam.

Na prática, utiliza-se mecanismos ansiosos em conjunto com a detecção por tentativa, e mecanismos preguiçosos em conjunto com a detecção por efetivação [13].

2.6 Versionamento

Transações concorrentes estão constantemente realizando "tentativas" de escritas. É necessário que haja algum mecanismo para gerenciar essas tentativas, pois em um dado momento, existirão diversas "versões" dos dados (a versão antiga e a versão modificada por uma transação – lembrando que várias transações podem acessar o mesmo dado). Existem duas abordagens gerais para o controle de versões (ou versionamento) [13, 26]:

Versionamento Ansioso Uma transação modifica o dado diretamente na memória e guarda o valor antigo em um *undo-log*; se ela abortar, o valor será escrito de volta. Assim, há necessidade de um controle de concorrência pessimista, pois o bloqueio de um local é necessário, já que a transação vai escrever nele diretamente;

Versionamento Preguiçoso As tentativas de escritas são armazenadas em um *redo-log* que é privado à transação e são atualizadas na memória apenas se e quando a transação efetiva. Se ela abortar, o *redo-log* é simplesmente descartado.

3. TM EM SOFTWARE

As STMs oferecem algumas vantagens sobre HTMs [26], entre elas:

- Permitem a implementação de uma grande variedade de algoritmos mais sofisticados;
- O software é mais fácil e barato de se modificar do que o hardware;
- Podem ser integradas mais facilmente com sistemas existentes e características de linguagens, tal como o coletor de lixo.

Shavit e Touitou [29] cunharam o termo de memória transacional em software em 1995, quando propuseram uma solução totalmente baseada na utilização em software para o mecanismo em hardware desenvolvido por Herlihy e Moss [14]. Este sistema mantém, para cada palavra em memória compartilhada, um registro de posse, que aponta para o registro da transação que é detentora da palavra. Para se efetivar uma transação, deve-se obter a aquisição de posse de todas as palavras a serem alteradas, seguindo pela efetuação das alterações e liberação da posse das palavras.

Um conflito ocorre quando uma transação tenta obter posse de alguma palavra que já foi adquirida por outra transação. Existem duas principais desvantagens no sistema desenvolvido por Shavit e Touitou [29]: a primeira é que o conjunto de endereço dos dados que serão acessados precisa ser conhecido de antemão; a segunda é a necessidade de espaço extra para armazenar o registro de posse de cada palavra compartilhada, aumentando assim o espaço de armazenamento.

Para evitar este grande requerimento de espaço, Fraser [11] apresenta um sistema denominado FSTM (*Fraser Software Transaction Memory*), no qual o objeto é utilizado como unidade básica de armazenamento, diminuindo assim o *overhead* de espaço, pois o registro de espaço agora está associado a um objeto inteiro. Além disso, a FSTM, ao contrário do sis-

tema proposto por Shavit e Touitou [29], permite transações dinâmicas (ou seja, o conjunto de endereços acessados não precisa ser conhecido de antemão).

A manipulação de transações é feita através do uso de bibliotecas ou de uma API. O pseudo-código abaixo exibe a API utilizada no FSTM, onde as seis primeiras rotinas são utilizadas para manipulação das transações e as quatro últimas para manipulação dos objetos.

```
 1 stm *new_stm(int object_size);
 2 void free_stm(stm *mem);
 3 stm_tx *new_transaction(stm *mem);
 4 void abort_transaction(stm_t *t);
 5 bool commit_transaction(stm_tx *t);
 6 bool validate_transaction(stm_t *t);

 7 (stm_obj *, void *) new_object(stm *mem);
 8 void free_object(stm *mem, stm_obj *o);
 9 void *open_for_reading(stm_tx *t, stm_obj *o);
10 void *open_for_writing(stm_tx *t, stm_obj *o);
```

Os modelos iniciais, como o modelo de Fraser, utilizavam *sincronização livre de bloqueio*. Uma das características deste tipo de sincronização é que quando uma transação T_1 entra em conflito com uma transação T_2, T_1 deve ajudar T_2 a completar a sua tarefa.

Contudo, este tipo de abordagem nem sempre produz bons resultados e sua implementação é demasiadamente complicada [26]. Com isso, as abordagens que se seguiram para STM acabaram utilizando a chamada *sincronização livre de obstrução*, onde é suficiente garantir o progresso de uma transação na ausência de conflitos, ou em outras palavras, uma transação não precisa garantir o progresso das outras. Uma das principais abordagens com STM livre de obstrução é a de Marathe *et al.* [19].

Apesar das melhorias, o desempenho das abordagens não-bloqueantes ainda estava muito abaixo em comparação aos métodos de sincronização tradicionais, o que levou alguns pesquisadores a considerarem novas alternativas de implementação para as STMs [26]. Foi sugerido então a adoção de bloqueios como uma possível forma de aumentar o desempenho das STMs. Alguns dos fatores que a utilização de métodos bloqueantes dão vantagem são: maior possibilidade de otimizar do código, reduzir o número de cancelamentos de transações e gerência mais simplificada da memória.

Uma das abordagens bloqueantes para STM atuais é a de Saha *et al.* [28], que utiliza o chamado *bloqueio de versão*. Esta abstração combina um bloqueio de exclusão-mútua, que é utilizado para arbitrar entre escritas concorrentes, com um número de versão que é utilizado pelos leitores para detecção de conflitos. Se o bloqueio está disponível, então nenhuma transação tem escritas pendentes no objeto e o bloqueio de versão possui o número da versão atual do objeto. Caso contrário, o bloqueio se refere à transação que atualmente utiliza o objeto. Antes de efetuar a leitura de um objeto, a transação precisa salvar a versão atual do objeto em seu conjunto de leitura:

```
1 void OpenForReadTx(TMDesc tx, object obj) {
2     tx.readSet.obj = obj;
3     tx.readSet.version = GetSTMMetaData(obj);
4     tx.readSet++;
5 }
```

Antes de escrever em um objeto, a transação adquire a versão do bloqueio do objeto e então adiciona a referência do objeto e o número antigo da versão do objeto no conjunto de escrita. Antes de escrever em algum campo do objeto, o sistema precisa gravar o valor anterior do campo no *undo-log* da transação, para que assim a modificação possa ser revertida:

```
1 void LogForUndoIntTx(TMDesc tx, object obj,
int offset) {
2    tx.undoLog.obj = obj;
3    tx.undoLog.offset = offset;
4    tx.undoLog.value = Magic.Read(obj,offset);
5    tx.undoLog++;
6 }
```

Bloqueios pessimistas previnem conflitos em objetos que uma transação tenha escrito. Contudo, no tempo de efetivação, o tratamento de leituras requere grande atenção pois não existem tentativas para prevenir a ocorrência de conflitos de leitura-escrita durante o tempo de execução da transação. Se faz então necessário detectar estes conflitos na validação da efetivação, assegurando assim que a transação como um todo pareça executar de forma atômica. Em um pseudo-código, a operação de efetivação é descrita da seguinte forma:

```
 1 bool CommitTx(TMDesc tx) {
 2     //Checa o conjunto de leitura.
 3     foreach(entry e in tx.readSet){
 4         if(!ValidateTx(e.obj, e.version)){
 5             AbortTx(tx);
 6             return false;
 7         }
 8     }
 9     //Desbloquear o conjunto de escrita.
10     foreach(entry e in tx.writeSet){
11         UnlockObj(e.obj, e.version);
12     }
13     return true;
14 }
```

O primeiro laço valida cada entrada no conjunto de leitura, verificando se houve ou não um conflito no objeto envolvido. Assumindo que não tenha ocorrido, o segundo laço libera todos os bloqueios de escrita que foram adquiridos pela transação. Liberando o bloqueio da versão, é incremento o número da versão do objeto, sinalizando assim um conflito com qualquer leitor concorrente do objeto. Durante a validação, existem alguns casos a serem considerados, baseado nos valores salvos nos *logs* de transação, e no atual estado do bloqueio de versão do objeto. Se a validação é realizada com sucesso, então a transação é efetivada.

4. TM EM HARDWARE

Embora STMs possam ser bem flexíveis, HTMs podem oferecer várias vantagens chave [13]:

- HTM possui um *overhead* menor do que STM;
- HTM pode ser menos invasiva nos sistemas atuais; por exemplo, HTMs que são implícitas podem garantir as propriedades de transação para bibliotecas de terceiros que não foram escritas para sistemas transacionais.

HTMs devem executar da mesma maneira que STMs: elas devem identificar acessos a memória dentro de transações, gerenciar os conjuntos de leitura e escrita da transação, detectar e resolver conflitos, gerenciar o estado dos registrado-

res da arquitetura, efetivar e abortar transações.

A primeira coisa que a HTM deve fazer é identificar acessos à memória dentro de transações. Dependendo de como é essa identificação, ela pode ser classificada de duas formas [4]:

Transação Explicita São fornecidas instruções de acesso a memória especiais para acessos transacionais. Para escrita, por exemplo, pode ser fornecida uma instrução `store_transactional`, usando a instrução `store` para uma escrita não transacional;

Transação Implícita São fornecidas instruções apenas para demarcar o início e o final de uma transação, que geralmente são `begin_transaction` e `end_transaction`. Qualquer acesso a memória realizado dentro deste intervalo é feito transacionamente.

HTMs explícitas são mais flexíveis em relação ao acesso a memória, pois elas permitem que acessos transacionais e não transacionais sejam intercalados e assim os conjuntos de leitura e escrita de uma transação ficam menores. A maioria das HTMs possuem uma limitação em relação ao tamanho da transação, imposta pela capacidade de armazenamento interna do processador [2, 25], por isso a importância de se manter os conjuntos de leitura e escrita pequenos. Um ponto negativo das HTMs explícitas é o fato de que estas geralmente exigem que os fornecedores de biblioteca disponibilizem duas versões da mesma, uma para ser utilizada dentro de uma transação e a outra fora [13]. HTMs implícitas não sofrem deste problema, porém são menos flexíveis em relação ao controle de acesso a memória, pois tudo o que está dentro de uma transação é acessado transacionalmente.

Quando lidamos com transações em hardware, uma abordagem natural e muito utilizada para controlar os conjuntos de leitura e escrita é utilizar uma expansão da cache [13], pois esta já controla acessos de leitura e escrita. Assim, HTMs tem a vantagem de já possuir um mecanismo para controlar leituras e escritas.

A primeira proposta de HTM foi de Knight [15] e utilizava uma cache separada para controlar os conjuntos de leitura e escrita de uma transação. Atualmente, a abordagem mais popular é estender a cache existente para que ela passe a controlar os conjuntos [13], como é utilizada por Rajwar e Goodman [24] e Dice *et al.* [8]. Para tratar o conjunto de escrita, algumas abordagens estendem a cache atual [2, 3], enquanto outras utilizam um *buffer* separado para esta finalidade [6]. Zilles e Rajwar [34] apresentam um estudo sobre a eficiência de se usar a cache para controlar os conjuntos de leitura e escrita.

HTM também deve detectar se os conjuntos de leitura e escrita de transações executadas em paralelo conflitam ou se sobrepõem [22]. Se ela utilizar *buffers* locais para rastrear os conjuntos, então ela pode utilizar o protocolo de coerência de cache para detecção de conflitos [13].

A maioria das HTMs executam detecção de conflitos ansiosa e ao detectar um conflito o processador aborta a transação e passa o controle para um mecanismo de tratamento em software, para que este decida se a transação será reexecutada ou se será aplicada uma técnica de resolução de conflito [13].

Se congelássemos a execução de um programa em um determinado momento, poderíamos dizer que o estado da execução de um programa é formado pelo estado dos registradores da arquitetura e pelo valores que o programa está armazenando na memória [30]. Quando uma transação aborta devemos restaurar o programa para um estado consistente de execução e isso geralmente significa restaurar os valores da memória e registradores para os valores anteriores ao da transação. Quanto à memória, podemos usar versionamento preguiçoso, mas infelizmente não podemos adiar a escrita nos registradores. Algumas das abordagens que são geralmente utilizadas para restaurar os valores dos registradores para seus estados consistentes são [13]: (i) delegar a responsabilidade de restaurar os registradores da arquitetura para o software; (ii) criar um registrador cópia (*shadow register*) de cada registrador da arquitetura; ou (iii) utilizar renomeação de registradores (*register renaming*). Delegar para o software restaurar o estado de execução é adequado apenas para pequenas transações, pois pode ser difícil para o compilador conseguir identificar corretamente como fazê-la [13].

Efetivar uma transação usando HTM requer fazer todas as atualizações de uma transação visíveis de uma forma instantânea, atômica. Para HTMs que utilizam *buffers* locais para armazenar as atualizações é necessário obter permissão de escrita em todos os endereços utilizados da transação, bloquear qualquer requisição subsequente de outros processadores a tais endereços, e finalmente gravar todas as atualizações do *buffer* na cache. HTMs que usam a cache como um *buffer* para os dados transacionais requerem um novo projeto de *status* da cache, como feito por Blundell *et al.* [3].

HTMs que utilizam estruturas do processador para fornecer transações, tais como *buffers* e caches, inserem complexidade em um sistema já complexo, além desses sistemas transacionais serem limitados pela estrutura do processador [2, 25]. Transações que possuem um tempo de execução maior do que o *quantum* do processador, ou cujo tamanho é maior do que a capacidade da estrutura fornecida pelo processador não podem ser executadas nessas implementações de HTM [25]. Quando uma transação não pode ser executada devido ao seu tamanho ou duração, existem duas alternativas [13]: (i) usar *locks* para executar o código daquela transação; ou (ii) utilizar STM para aquela transação.

A fim de evitar a perca das facilidades providas pelo uso de transações e evitar o *overhead* das STM, surgiu uma nova classe HTM, chamada HTM ilimitada [1, 33]. Essa classe consegue executar transações que são maiores do que um *buffers*/cache de processador e/ou mais longas do que o *quantum*. A implementação dessa nova classe de HTM é complexa e varia muito de uma implementação para outra. Algumas implementações de HTM ilimitadas são [1, 25, 33].

5. MEMÓRIA TRANSACIONAL HÍBRIDA

HTMs e STMs possuem suas limitações; por exemplo, HTMs geralmente não suportam transações que usam mais recursos do que o existente, e STMs apresentam altos *overheads* [31]. Mas ambas possuem suas vantagens; por exemplo, HTMs suportam isolamento forte por separar acessos transacionais de acessos não transacionais, e STMs podem ser facilmente modificadas para introduzir novas características [21].

Uma abordagem alternativa é a memória transacional híbrida (HyTM), que suporta a co-existência de transações em hardware e em software simultaneamente tentando obter o melhor de cada uma. Normalmente, HyTM suporta a execução de uma HTM e utiliza uma STM quando os recursos de hardware se excedem [12].

Trabalhos como os de Damron *et al.* [7], Minh *et al.* [21], para citar alguns, fornecem descrições de implementações de memórias transacionais híbridas, explicando, em geral, como funciona cada elemento do sistema (detecção e controle de conflitos, versionamento, controle dos conjuntos de escrita e leitura, etc.) e como se dá o relacionamento entre transações em software e em hardware.

6. DISCUSSÃO

Segundo Damron *et al.* [7], a TM tem por objetivo reduzir substancialmente a dificuldade de escrever código correto, eficiente e escalável para programas concorrentes. E isso pode ser atingido pelo fato dela permitir que o desenvolvedor descreva *o quê* deve ser executado atomicamente, ao invés de esperar que o desenvolvedor diga *como* atingir a atomicidade. Como benefícios da TM podemos citar [16]:

Facilidade de escrita de programas paralelos Os programadores não precisam se preocupar com a concorrência. Um experimento realizado por Rossbach *et al.* [27] comprovou que a memória transacional é menos suscetível a erro do que a programação utilizando mecanismos tradicionais de sincronização;

Facilidade em se conseguir bom desempenho paralelo Diferente dos *locks*, a serialização de uma transação depende apenas se as *threads* estão acessando os mesmos dados ao mesmo tempo;

Eliminação de deadlocks Transações podem ser abortadas a qualquer momento e a qualquer hora. Um *deadlock* pode ser evitado em TM abortando-se uma ou mais transações que dependem uma da outra e automaticamente reiniciando a transação abortada;

Facilidade de manter dados em um estado consistente Uma transação pode ser abortada a qualquer momento e suas alterações serão desfeitas automaticamente;

Evita *priority inversion* e *convoying* Se uma *thread* executando uma transação bloqueia outra *thread* (porque elas tentam acessar os mesmos dados), a transação pode ser abortada se ela tem baixa prioridade ou se ela é bloqueada em uma operação de latência longa;

Tolerância a falha Se a *thread* morre durante a execução da transação, a transação é automaticamente abortada deixando o dado em um estado consistente.

Apesar da TM ser vista por muitos pesquisadores como a solução mais promissora para os problemas da programação paralela, outros pesquisadores discordam que um dia ela possa ser utilizada em sistemas comerciais e até a classificam como um brinquedo de pesquisa [5]. Os principais problemas apontados pelos críticos da TM são:

I/O A maneira como I/O deve ser abordada dentro de transações ainda não é clara, e não há um consenso de qual deveria ser seu comportamento [12];

Transações Zumbis São transações que leram um valor obsoleto ou um ponteiro de memória que as levou a um laço infinito, por seguir um fluxo incorreto [20];

***Overhead* da STM** Em geral, STM possui um *overhead* maior do que o da HTM e o da programação usando mecanismos de sincronizações tradicionais como *locks* [5];

Transações limitadas e ilimitadas em HTM HTMs executam apenas transações pequenas devido ao pouco espaço de armazenamento. As ilimitadas resolvem esse problema, mas são difíceis de serem implementadas [12];

Atomicidade fraca em STM Implementações típicas de STM não detectam conflitos entre acessos transacionais e não-transacionais. Assim, STM possui uma semântica de *atomicidade fraca* e a tarefa de garantir que não ocorra um conflito entre um acesso transacional e um não transacional é do programador [5];

Recuperação de Memória Algumas implementações de STM proíbem a recuperação de memória acessada transacionalmente de ser reutilizada arbitrariamente, por exemplo através de `free` e `malloc`. Nessas implementações, a alocação e desalocação de memória acessada transacionalmente deve ser tratada de forma especial [5].

As TMs, implementadas em software ou hardware, introduzem complexidade que limitam seus ganhos de produtividade esperado, fazendo assim com que o incentivo para a utilização de TMs seja baixo e criando assim uma justificativa para que apenas uma pequena parcela do hardware atual suporte seu uso.

7. CONCLUSÃO

Este artigo apresentou uma introdução básica sobre TM, desde o contexto histórico e problemático que levou ao surgimento da proposta de transações para programação de propósito geral, passando pela concepção do termo memória transacional, suas propriedades e abordagens em software, hardware e hibrida, até uma discussão dos problemas e desafios associados com TM.

Referências

[1] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th Intl. Symp. on High-Performance Computer Architecture*, pages 316–327, feb 2005.

[2] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[3] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. *SIGARCH Comput. Archit. News*, 37(3):233–244, 2009.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, 2006.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, Sept. 2008.

[6] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth,

M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, 2006.

[8] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. *SIGPLAN Not.*, 44(3):157–168, 2009.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edition, 2003.

[10] P. Felber, C. Fetzer, R. Guerraoui, and T. Harris. Transactions are back—but are they the same? *SIGACT News*, 39(1):48–58, 2008.

[11] K. Fraser. Pratical lock-freedom. Technical Report 579, University of Cambridge, 2004.

[12] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, may-jun 2007.

[13] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, Madison, USA, 2nd edition, 2010.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[15] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[16] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th ACM SIGPLAN Symp. on Principles and practice of parallel programming*, pages 209–220, New York, NY, USA, 2006. ACM.

[17] J. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, Madison, USA, 1st edition, 2007.

[18] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGOPS Oper. Syst. Rev.*, 11(2):128–137, 1977.

[19] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, sep 2005.

[20] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *SIGPLAN Not.*, 44(6):166–176, June 2009.

[21] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2):69–80, 2007.

[22] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: log-based transactional memory. *Intl. Symp. on High-Performance Computer Architecture*, 0: 254–265, 2006.

[23] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[24] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[25] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. *SIGARCH Comput. Archit. News*, 33(2):494–505, 2005.

[26] S. Rigo, P. Centoducatte, and A. Baldassin. Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente. 2007. Minicurso apresentado no VIII Workshop on High Performance Computing System (WSCAD'07).

[27] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3.

[28] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[29] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3.

[30] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[31] E. Vallejo, S. Sanyal, T. Harris, F. Vallejo, R. Beivide, O. Unsal, A. Cristal, and M. Valero. Hybrid transactional memory with pessimistic concurrency control. *International Journal of Parallel Programming*, 39:375–396, 2011.

[32] D. W. Wall. Limits of instruction-level parallelism. Technical report, Western Research Laboratory, Nov. 93.

[33] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtmse: Decoupling hardware transactional memory from caches. *High-Performance Computer Architecture, International Symposium on*, 0:261–272, 2007.

[34] C. Zilles and R. Rajwar. Implications of false conflict rate trends for robust software transactional memory. In *Proceedings of the IEEE 10th Intl. Symp. on Workload Characterization*, pages 15–24, sep 2007.