Memórias Transacionais

Luciano Jerez Chaves - 079759
Instituto de Computação
Universidade Estadual de Campinas
lchaves@ic.unicamp.br

RESUMO

O surgimento das arquiteturas computacionais multiprocessadas impôs novas dificuldades no desenvolvimentos de aplicativos. Dentre estas dificuldades está a necessidade de sincronização dos fluxos de execução de forma a garantir acessos consistentes à memória de dados que é compartilhada entre esses fluxos. As técnicas convencionais para garantir esta sincronização são baseadas no uso de bloqueios e semáforos, impondo sérias limitações quanto à quantidade de acessos concorrentes e não se mostrando proeminentes para solucionar o problema em questão. Nesse contexto, as memórias transacionais surgiram como uma alternativa aos mecanismos tradicionais de sincronização, tendo como base os conceitos de transações utilizadas pelos sistemas de gerenciamento de banco de dados. Este artigo apresenta uma introdução aos conceitos que envolvem as memórias transacionais, bem como uma análise qualitativa obtida a partir de um estudo empírico sobre a utilização desses sistemas.

Palavras-chave

Arquitetura de Computadores; Programação Paralela; Memória Transacional.

1. INTRODUÇÃO

Durante três décadas ou mais, os computadores evoluíram entre 40% e 50% ao ano, valores mais do que suficientes para atender as expectativas da maior parte das aplicações e dos usuários. Por conta disso, os desenvolvedores de *software* voltaram seus esforços para a programação sequencial, com o objetivo de aproveitar ao máximo os recursos disponíveis nos computadores utilizados nessa época. Entretanto, a era das melhorias exponenciais no desempenho dos computadores sequenciais chegou ao fim próximo do ano de 2004 [13]. Por conta de limitações físicas, se tornou impossível melhorar o desempenho dos processadores somente aumentando a frequência de operação, já que o consumo energético subiu consideravelmente e não existe método eficiente para dissipar todo o calor produzido pelos processadores.

Mesmo com a imposição destas limitações, a quantidade de transistores nos *chips* continuaram crescendo segundo a Lei de Moore[1]. A resposta dada pela indústria de processadores às limitações apresentadas foi o investimento na fabricação dos *chips* multiprocessados, com dois ou mais núcleos independentes que compartilham uma mesma hierarquia de memória. Esta arquitetura paralela permite que o desempenho dos computadores continue crescendo, pelo menos pelas próximas gerações, já que à medida que o número de processadores duplica, o potencial paralelismo no nível de instruções dobra, enquanto a frequência é mantida intacta.

Para tirar proveito deste ganho, é preciso extrair o paralelismo existente nas aplicações. Entretanto, a programação paralela tem se mostrado mais difícil do que a programação sequencial, já que os algoritmos são mais complexos de serem formulados e a prova de corretude nem sempre é alcançada com sucesso. Como exemplo dessas dificuldades, podemos citar o controle de acesso às variáveis (posições de memória) que são compartilhadas entre os diversos fluxos de execução. Um programa paralelo que faz uso dessas variáveis compartilhadas demanda por mecanismos de sincronização para garantir o acesso consistente às mesmas. Normalmente este controle é alcançado através de técnicas de exclusão mútua, que incluem o uso de bloqueios (*locks*), semáforos, filas, etc. Estas técnicas, quando corretamente utilizadas, garantem que dois fluxos distintos não alterem uma mesma posição de memória concorrentemente, mantendo a consistência dos dados durante as operações. Entretanto, elas reduzem o desempenho alcançado pelas aplicações, principalmente porque esses bloqueios não se fazem necessários em todos os momentos, já que os diversos fluxos de execução podem nunca concorrer por uma mesma posição de memória no mesmo instante de tempo.

Com o objetivo de facilitar o desenvolvimento e aumentar o desempenho dos aplicativos paralelos, surgiram os sistemas de memória transacional (*Transactional Memory* - TM). Este modelo de programação oferece uma abstração semelhante aos sistemas de transações utilizados em bancos de dados, com o objetivo principal de garantir a sincronização necessária para a execução correta das operações de leitura e escritas na memória sem a necessidade do uso de mecanismos bloqueantes, como os tradicionalmente utilizados [3].

[1]O número de transistores dos *chips* sofrem um aumento de 100%, pelo mesmo custo, a cada período de 18 meses.

Para apresentar as memórias transacionais, o restante deste artigo está organizado como segue. Na Seção 2 é introduzido o conceito de transação como utilizado na área de banco de dados. Na sequência, a Seção 3 apresenta os detalhes sobre o funcionamento das TMs e as diferentes abordagens para implementação. A Seção 4 traz uma análise qualitativa sobre o uso das memórias transacionas. Para finalizar, a Seção 5 contém as conclusões acerca deste trabalho.

2. TRANSAÇÕES

Para atingir um bom desempenho tanto em computadores sequenciais como paralelos, os sistemas de gerenciamento de bancos de dados (SGBDs) permitem a execução transparente de diversas consultas simultâneas à uma mesma base de dados, sem que o autor de uma consulta individual precise se preocupar com as demais consultas concorrentes. A ideia principal por trás desta organização são as transações. A transação especifica uma semântica de programação em que a computação é executada como se fosse a única computação acessando a base de dados. Outras computações podem ser executadas simultaneamente, mas o modelo restringe a interação entre elas para garantir a corretude dos dados. Como consequência, um programador que escreve um código para uma transação pode se apegar ao tradicional e simples modelo sequencial, além de obter os ganhos previstos pelo processamento paralelo, quando disponível [4].

Uma transação é uma sequencia de ações que devem se apresentar como indivisíveis e instantâneas à um observador externo. Elas são regidas por quatro propriedades básicas, conhecidas como propriedades ACID:

- A propriedade de **atomicidade** requer que todas as ações de uma transação sejam completadas com sucesso, ou que nenhuma delas seja efetivamente executada;
- A propriedade de **consistência** visa garantir que as modificações geradas por uma transação não levem o sistema a um estado incorreto. Por estado correto pode-se entender um estado em que todos os dados armazenados na memória possuem valores válidos;
- A propriedade de **isolamento** requer que as transações produzam o resultado correto independente de quais e quantas outras transações estão sendo executadas concorrentemente;
- A propriedade de **durabilidade** assegura que os resultados gerados por uma transação que já foi concluída sejam permanentes e estejam disponíveis para uso pelas próximas transações.

Um exemplo típico do uso de transações é uma operação bancária que realiza uma transferência financeira entre duas contas correntes. Esta transação primeiro debita um certo valor de uma conta para depois creditar este valor na outra. Se após o débito na primeira conta ocorrer alguma exceção e a computação for interrompida, a base de dados passará a apresentar um estado inconsistente. Para que isso não ocorra na prática, o mecanismo de transações garante que: ou as duas contas são alteradas, concluindo a transferência através de um *commit*; ou que nenhuma delas é modificada, através de um *abort* na transação. Para que isso seja possível, o programador precisa somente informar onde começa e termina cada transação. A partir daí, o SGBD assume o controle para executar as operações e garantir a semântica correta do sistema.

3. MEMÓRIAS TRANSACIONAIS

O conceito de memória transacional surgiu em 1977, quando Lomet observou que uma abstração similar às transações de banco de dados poderia se tornar um mecanismo de programação eficiente para garantir a consistência de dados compartilhados entre diferentes processos em execução em uma máquina multiprocessada [11]. Entretanto, a ideia apresentada por ele não era acompanhada de nenhuma implementação prática, e por conta disso caiu no esquecimento até 1993, quando Herlihy e Moss propuseram uma memória transacional implementada em *hardware* capaz de gerenciar estruturas de dados compartilhadas sem o uso dos tradicionais métodos bloqueantes [7].

Considerando o modelo de programação paralela tradicional, pode-se dizer que uma transação é o equivalente à uma seção crítica definida com o uso de bloqueios e semáforos, cujo objetivo é garantir que as leituras e escritas na memória realizadas pelos diversos fluxos de execução não resultem em um estado inconsistente. Todavia, esta nova abordagem utiliza uma sintaxe mais simples e fácil de compreender, além de evitar as serializações desnecessárias do modelo anterior. A Figura 1 exemplifica um trecho de código delimitado por um bloco atômico, responsável por identificar o início e o fim de uma transação.

```
atomic {
      if (x != NULL)
          x.function();
      y = true;
}
```

Figura 1: Definição sintática de uma transação.

Um sistema de TM é uma implementação em *hardware* ou *software* responsável por executar as transações, garantindo a corretude nas operações das mesmas. Essa execução deve ser feita como se as transações fossem completamente independente, sem interromper o fluxo de instruções com o uso dos bloqueios e dos semáforos, de maneira a diminuir a sobrecarga imposta pelas serializações desnecessárias. Ao mesmo tempo, o sistema monitora todos acessos à dados realizados pelas transações para detectar possíveis conflitos entre escritas e leituras numa mesma posição de memória. Casos algum conflito seja identificado, o sistema passa a gerenciá-lo, tomando as atitudes necessárias para corrigir as alterações indevidas realizadas pela aplicação. A técnica utilizada consiste em escolher uma das transações conflitantes para realizar o *commit* com sucesso, enquanto as demais serão abortadas e re-executadas. Caso contrário, quando não ocorrem conflitos, todas as transações são concluídas com sucesso e o aplicativo continua sua execução normal.

3.1 Características de implementação

Conforme descrito em [9], existem várias propostas distintas de sistemas de TM tanto na literatura quanto em uso em sis-

temas computacionais reais (algumas destas propostas serão apresentadas nas Subseções 3.2, 3.3 e 3.4). A seguir são apresentadas algumas das classificações mais comuns em relação às características de projeto dos sistemas de TM propostos até hoje.

A **verificação de conflitos** é a técnica utilizada pelos sistemas de TM para identificar conflitos de memória entre as transações em execução. As abordagens mais comuns são:

- Ansioso: neste modelo, o sistema de TM monitora em tempo real a execução de cada transação ativa, identificando a existência de conflitos imediatamente após elas acontecerem;
- Preguiçoso: com esta abordagem, o sistema de TM somente faz a verificação de possíveis conflitos ao final da execução de toda a transação, imediatamente antes da realização do *commit*.

O modelo ansioso aumenta a sobrecarga durante a execução da transação, porém é mais responsivo no sentido de que não é preciso esperar a transação terminar para identificar que ocorreu um conflito e que a mesma deve ser abortada. Entretanto, uma abordagem preguiçosa apresenta melhores resultados quando as transações são pequenas, já que o custo de monitorar cada acesso à memória não compensa as instruções que poderiam deixar de serem executadas desnecessariamente.

No que diz respeito ao **isolamento entre transações**, o sistema precisa identificar quais posições de memória serão considerados durante a verificação de conflitos entre as transações. As abordagens utilizadas são:

- Fraco: neste caso, o sistema de TM somente monitora os acessos à memória que são realizados por operações dentro das transações. Acessos externo às transações não são considerados e não provocam conflitos;
- Forte: já neste modelo, o sistema de TM monitora todos os acessos à memória, transformando de maneira transparente os acessos não inclusos nas transações em pequenas novas transações.

Com o isolamento fraco, grande parte da responsabilidade sobre o controle ainda cai sobre o programador que, para garantir a corretude da aplicação, precisa ter cuidado com acessos à memória fora das transações. Já com um isolamento forte, esta preocupação pode ser eliminada, ao custo de maior sobrecarga na execução das aplicações.

A estratégia de **resolução de conflitos** decide como o sistema de TM gerencia a memória para que possa ser capaz de abortar uma transação em andamento. As técnicas utilizadas são classificadas em:

- Ansioso: com esta abordagem, o sistema de TM armazena em um *buffer* o estado da memória antes do início da execução de cada transação. Assim, a memória pode ser modificada permanentemente à medida que a transação é executada.
- Preguiçoso: o sistema de TM armazena todas as modificações realizadas pela transação em um *buffer*, e somente altera a memória quando não ocorre nenhum conflito e a transação é concluída com sucesso.

O modelo ansioso privilegia as transações que executam sem ocorrência de conflitos, ao passo que quando é preciso abortar a transação, é custoso desfazer todas as alterações e retomar a memória com as características originais. Na resolução de conflitos preguiçosa, as transações que são abortadas não introduzem custo além de sua execução, mas todas as transações completadas com sucesso ainda precisa aguardar o tempo necessário para aplicar as modificações do *buffer* na memória antes de realizarem o *commit*.

Outra característica importante está relacionada com o **gerenciamento de contenção**. Esta é a parte do sistema responsável por gerenciar os conflitos, e decidir quais transações devem ser abortadas e quais irão efetuar o *commit*. Para isso, são utilizadas políticas de resolução de conflitos, que consistem em regras para auxiliar na decisão de maneira a minimizar a penalidade. Algumas dessas políticas de gerenciamento de contenção são:

- *Committer wins*: quando um conflito é identificado já na fase de *commit* de uma transação, essa transação será concluída e as demais transações conflitantes serão abortadas;
- *Requester wins*: quando uma transação realiza um acesso à um endereço de memória compartilhado que esteja em uso por outra transação, o sistema aborta as demais transações e permite que esta execute até o final;
- *Committer stalls*: quando uma transação realiza um acesso à um endereço de memória compartilhado que esteja em uso por outra transação, o sistema aborta esta transação, deixando a outra concluir com sucesso.

A configuração destas características depende da implementação. Há sistemas de TM que permite que o usuário utilize a configuração que julgar adequado para a aplicação em execução. Em outros sistemas, estas escolhas são realizadas pelo próprio sistema, sem intervenção humana.

Como já foi dito, um sistema de TM pode ser implementado tanto em *hardware* quanto em *software*. As memórias transacionais em *hardwares* (HTM) são implementadas através de alterações na arquitetura dos processadores, nas memórias *caches* (principalmente nos protocolos de coerência), e nos barramentos de instrução e dados. Por sua vez, as memórias transacionais em *softwares* (STM) proveem a mesma semântica de TM através de uma biblioteca dinâmica ou da própria linguagem de programação, e não requerem suporte avançado dos processadores (tipicamente, uma instrução do tipo *test and set* é suficiente). Uma terceira classificação são as memórias transacionais híbridas, que possuem em sua essência uma implementação em *software*, mas que é apoiada por mecanismos de *hardware* para melhorar o desempenho. Nas subseções a seguir serão apresentados mais detalhes de cada uma dessas vertentes no desenvolvimentos de sistemas de memória transacional.

3.2 Memórias Transacionais em Hardware

As primeiras propostas de implementações de TM foram desenvolvidas para suporte de *hardware*, como foi o caso de Tom Knight [8], que em 1986 apresentou a depositou a primeira patente por esta ideia inovadora. Entretanto, as HTMs só se popularizaram em 1993 com o trabalho de Herlihy e Moss [7]. O principal objetivo destes sistemas de TM é atingir o melhor desempenho com o menor custo, eliminando a necessidade de suporte dos compiladores e garantindo eficiência e isolamento com poucas modificações nas aplicações. Segundo [9], os sistemas de HTM podem ser divididos em três grupos distintos: (1) Abordagem percursoras, (2) HTMs limitadas e (3) HTMs ilimitadas. Essas categorias são apresentadas na sequência.

3.2.1 Abordagens percursoras

As abordagens percursoras consistem em trabalhos voltados para o suporte eficiente ao paralelismo através de técnicas e implementações em *hardware*. Estas abordagens fornecem grande inspiração para posteriores desenvolvimentos de HTMs, servindo como base e fundamentação para consequentes pesquisas.

Um exemplo desta categoria é o mecanismo proposto por Herlihy e Moss [7]. Nesse trabalho foram introduzidas novas instruções primitivas para manipulação de memória: o `Load Transactional` (`LT`), que lê o valor de uma posição de memória compartilhada em um registrador; o `Load Transactional Exclusive` (`LTE`), que lê o valor de uma posição de memória compartilhada garantindo o acesso exclusivo; e o `Store Transactional` (`ST`), que grava o valor de um registrador em uma posição de memória. A grande diferença das operações transacionais para suas equivalentes não-transacionais está no processo de escrita: as mudanças realizadas por uma transação só se tornam visíveis às demais após executar a instrução de `commit`. Antes de efetivar completamente as mudanças, porém, a instrução `commit` verifica se as posições de memória operadas transacionalmente pela aplicação foram modificadas concorrentemente, ou se outra transação leu algum valor modificado por esta. Em caso afirmativo, a operação falha e a instrução de `abort` é executada para desfazer as mudanças. Uma última instrução proposta é a `validate`, que testa para saber se a transação atual já abortou ou não, com o objetivo de verificar a validade dos valores lidos até então, evitando que uma aplicação execute operações ilegais.

Toda a implementação deste sistema é feita através de modificações no protocolo de coerência de *cache* proposto por Goodman [5]. A abordagem é manter duas *caches* em funcionamento: uma *cache* normal e uma *cache* transacional. As linhas de ambas as *caches* contém um dos quatro estados: `INVALID`, `VALID`, `DIRTY` e `RESERVED`; também presentes no protocolo de Goodman. O estado `INVALID` indica uma linha inválida, enquanto o estado `VALID` indica uma linha compartilhável com permissão de leitura que contém os mesmos dados que na memória principal. O estado `DIRTY` indica uma linha não-compartilhável com permissão de escrita e leitura cujo valor foi modificado em relação ao valor na memória principal. Por fim, o estado `RESERVED` indica uma linha igual a uma linha marcada `DIRTY`, exceto por seu conteúdo ainda não ter sido modificado. A Tabela 1 sumariza os quatro estados apresentados. Operações não-transacionais usam a *cache* normal da mesma forma que no protocolo de Goodman.

Table 1: Estados do protocolo de Goodman.

Estado	Acesso	Compart.	Modific.
`INVALID`	Nenhum		
`VALID`	Leitura	Sim	Não
`DIRTY`	Leitura/Escrita	Não	Sim
`RESERVED`	Leitura/Escrita	Não	Não

Já a *cache* transacional mantém adicionalmente para cada linha outro estado, que pode assumir um dentre quatro valores: `EMPTY`, `NORMAL`, `XCOMMIT` e `XABORT`. O estado `EMPTY` indica que uma linha de *cache* não está em uso e o estado `NORMAL` indica que ela contém dados modificados por uma transação bem-sucedida. Instruções `ST` sempre modificam as linhas para o estado `XABORT`. Essas mudanças não são propagadas à memória principal até o sucesso da transação, permanecendo invisíveis aos outros fluxos de execução. Uma operação de `COMMIT` bem-sucedida envolve a mudança em paralelo de estados de `XABORT` para `NORMAL` e de `XCOMMIT` para `EMPTY`. Se a transação é abortada, o contrário ocorre: linhas `XABORT` são marcadas como `EMPTY` e `XCOMMIT` são marcadas como `NORMAL`. A Tabela 2 sumariza os estados utilizados pela implementação deste sistema de HTM.

Table 2: Estados da memória transacional.

Estado	Significado
`EMPTY`	Não contém dados
`NORMAL`	Contém dados comitados
`XCOMMIT`	Descartar dados quando comitar
`XABORT`	Descartar dados quando abortar

A ideia dos estados transacionais é que um dado em *cache* deve estar, sempre que possível, em uma linha `XABORT` e em uma linha `XCOMMIT`. As linhas `XCOMMIT` contém sempre dados que foram permanentemente modificados, para que não seja sempre necessário acessar a memória principal quando uma transação aborta. Para que este sistema de HTM funcione corretamente, é preciso adaptar tanto o barramento quanto o processador para operar com os novos estados previstos na memória *cache*.

Algumas limitações do modelo apresentado são expostas: interrupções que ocorrem em uma transação causam aborto compulsório, enquanto uma troca de contexto provoca o reinício de todas as transações. Se o tamanho da memória *cache* não for suficiente para acomodar todos os blocos nos quais uma transação opera, ela será igualmente abortada. Adicionalmente, transações mais demoradas têm probabilidade maior de serem abortadas.

3.2.2 HTMs limitadas

Além das abordagens percursoras, alguns sistemas de HTM permitem que uma transação utilize informações além das fornecidas pela *cache* de dados. Entretanto, não é permitido que os diferentes fluxos de execução migrem de um processador para outro durante sua execução, bem como não pode haver trocas de contexto. Estes são os sistemas denominados HTMs limitados.

Um exemplo desta categoria é o sistema de HTM proposto por Hammond et al [6], denominado *Transactional Coherence and Consistency* (TCC). Neste modelo, todas as operações executadas pela aplicação deve pertencer a alguma transação, que pode ser definida pelo programador ou automaticamente pelo compilador.

A implementação deste sistema não utiliza os protocolos de coerência por linha de *cache* como visto anteriormente. Ao contrário, o TCC permite que diferentes fluxos de execução acessem concorrentemente a memória. Desta forma, cada transação é executada de maneira especulativa, e as alterações são provisoriamente armazenadas em um *buffer* local até que se decida pelo *commit* ou pelo *abort*. Quando uma transação está pronta para comitar, ela solicita um *token* global. Este *token* determina qual transação irá comitar, já que somente uma transação pode realizar as alterações globais na memória por vez. Assim que uma transação consegue o *token*, ela realiza o *commit* e faz um *broadcast* de todas as alterações para os outros processadores, que por sua vez irão comparar as informações recebidas com as alterações próprias para identificar os conflitos. Se ocorrer um conflito, esta transação deve ser abortada. Caso contrário, o processador continua ativo esperando sua vez de adquirir o *token*.

Outro sistema HTM limitado, proposto por Moore [12], descreve o sistema LogTM, cujos dados transacionais não se restringem a utilizar a *cache* local, mas também são transportados para os demais níveis da hierarquia de memória. Esta implementação utiliza um sistema de *logs* em *software* para armazenar os valores de locais de memória atualizados em uma transação, de forma que estes valores possam ser recuperados por uma rotina em *software* se a transação for abortada. Esta abordagem favorece transações que realizam o *commit* com sucesso.

3.2.3 HTMs ilimitadas

Os chamados HTMs ilimitados são sistemas de TM que permitem que as transações sobrevivam tanto a mudanças de contexto quanto à preempção para compartilhamento do processador. Este suporte é fundamental para que os sistemas de HTM sejam amplamente utilizados. Para atingir este objetivo, é necessário armazenar o estado das transações em um espaço de memória persistente, como o espaço de memória virtual.

O trabalho proposto por Ananian et al [1] apresenta o UTM: *Unbounded Transactional Memory*. Este sistema apresenta uma arquitetura que tenta desacoplar as informações de estado e verificação de conflito do *hardware* da *cache* e dos protocolos de coerência. Para que isso seja possível, o UTM introduz uma estrutura de dados denominada `XSTATE`, que é residente na memória e pode ser acessada por todo o sistema. O `XSTATE` armazena as informações sobre todas as transações do sistema, incluindo as leituras e escritas na memória. Para que a verificação de conflitos seja independente do protocolo de coerência, o UTM mantém um *bit* de acesso para cada bloco da memória no sistema de endereçamento virtual. Assim, as transações podem consultar estes *bits* para determinar a ocorrência de conflitos.

Em [15], Rajwar et al descrevem o sistema *TM Virtual Transactional Memory* (VTM). Neste sistema não é necessário que os programadores implementem detalhes em *hardware*, mas façam uso da virtualização dos recursos limitados, como *buffers* de *hardware* e períodos de escalonamento. Esta abordagem também possibilita que as transações sobrevivam a trocas de contexto e excedam os limites de memória impostos pelo *hardware*. Este mecanismo de virtualização é semelhante à forma como a memória virtual torna transparente o gerenciamento de memória física limitada para os programadores. O sistema VTM desacopla os estados de manutenção das transações e detecções de conflitos do *hardware* através do uso de estruturas de dados residentes na memória virtual da própria aplicação. Estas estruturas armazenam informações sobre as transações que excederam os limites da *cache* ou ainda que excederam o tempo de escalonamento.

Em [19], Zilles e Baugh apresentam uma implementação alternativa para o sistema VTM. Nesta implementação são descritas extensões no *software* e no conjunto de instruções para permitir que as transações VTM aguardem determinados eventos para reiniciar sua execução eficientemente através de comunicação com o escalonador, que é orientado a pausá-las temporariamente e compensá-las em seguida. Esta implementação adiciona novas instruções, novo suporte a interrupções de *software* e amplia as estruturas de metadados da implementação VTM original.

3.3 Memórias Transacionais em Software

As memórias transacionais em *software* surgiram em 1995, como uma alternativa flexível para a implementação dos sistemas de TM nos processadores existentes. Elas consistem de sistemas de *software* que implementam transações não duráveis para a manipulação de dados compartilhados entre fluxos de execução. A maioria dos STMs podem ser executados em processadores convencionais, porém pouco se conhece a respeito do custo computacional introduzido ao se utilizar tal recurso.

As principais vantagens dos STMs sobre os HTMs são: (1) o *software* é mais flexível, permitindo implementações de algoritmos mais sofisticados; (2) o *software* é mais fácil de ser modificado e evoluído; (3) os STMs podem se integrar com mais facilidade aos sistemas existentes; e (4) os STMs possuem menos limitações intrínsecas impostas pelas estruturas fixas de *hardware*.

O primeiro artigo apresentando um STM foi escrito por Shavit e Touitou [17]. A abstração de programação proposta por eles requer que uma transação declare, à *priori*, todas as posições de memória que podem ser acessadas pela transação. Esta estratégia provoca uma sobrecarga significativa de memória, já que uma palavra é reservada para cada palavra utilizada pela aplicação. Com essas informações, o STM é capaz de adquirir a posse de cada uma dessas posições de memória através do uso interno de bloqueios. Se uma transação conseguir a posse das posições de memória que deseja modificar, então ela pode executar até o fim sem se preocupar com possíveis *roll-backs*. Além disso, o STM pode adquirir posse das posições de memória em uma ordem pré-definida, o que previne a ocorrência de impasses (*deadlocks*). Esta proposta de STM oferece garantia de que todas as transações eventualmente conseguirão ser executadas, mesmo se

acontecer falhas ou re-escalonamento da aplicação.

O sistema de memória transacional RSTM, proposto por Scott et al, é descrito em [16]. Este sistema foi feito para programas implementados na linguagem C++ e que utilizam *pthreads* no seu processo de paralelização. Este sistema implementa o conceito de objetos transacionais, ou seja, se um objeto é transacional ele pode executar transações com a garantia de que se elas forem abortadas, os valores alterados durante sua execução serão restaurados. Quando duas ou mais transações entram em conflito, um sistema de gerenciamento de contenção seleciona uma para ser concluída, enquanto força as demais a aguardar ou re-executar. O sistema de gerenciamento de contenção do sistema RSTM é baseado no algoritmo Polka, que privilegia transações que já executaram uma maior quantidade de instruções.

Um outro sistema de memória transacional por *software* bastante conhecido é o TL2, proposto por Dice et al em 2006 [2]. Este sistema baseia-se na ideia de manter um relógio global de versões de cada uma das variáveis que são acessadas no escopo de uma transação. Sempre que uma leitura ou escrita é realizada, um algoritmo de verificação de versão é executado, garantindo a consistência dos dados compartilhados. O relógio global funciona como um bloqueio que é adquirido e incrementado no final da execução especulativa de uma transação. As transações que só executam leitura são bastante facilitadas neste sistema, uma vez que só é necessário verificar se o relógio global não foi modificado entre o início e o fim da transação.

3.4 Memórias Transacionais Híbridas

Frente às dificuldades de se implementar um sistema de TM em *hardware* com suporte ilimitado às transações, surgiu uma abordagem alternativa que consiste em implementar uma memória transacional em *software* que tenha suporte direto do *hardware*, de maneira a aumentar o ganho de desempenho.

O sistema proposto por Lie em 2004 [10] descreve um sistema de memória transacional híbrido de *hardware* e *software* em que as transações são, em princípio, executadas pelo *hardware*. Porém, se estas transações não puderem ser executadas dessa forma, elas são reiniciadas e executadas como uma transação em *software*. Neste sistema, duas versões do código das transação são geradas: uma para a execução em *hardware* e outra para execução em *software*. Neste caso, é preciso que o *hardware* possa se comunicar com o *software* para detectar possíveis conflitos entre as transações em execução nessas diferentes esferas.

Outro sistema de memória transacional híbrido é o *Rochester Transactional Memory* (RTM), proposto por Shriraman [18], e que utiliza mecanismos HTM apenas para acelerar uma STM. Esta abordagem apresenta um novo conjunto de instruções através do qual é possível que uma STM controle cada um dos mecanismos da HTM, executando tarefas específicas da STM diretamente em *hardware*. Este sistema permite, por exemplo, que um *software* controle quais linhas de *cache* devem ser transacionalmente monitoradas e mantidas expostas aos protocolos de coerência de *cache* para a correta execução da transação.

4. ANÁLISE QUALITATIVA

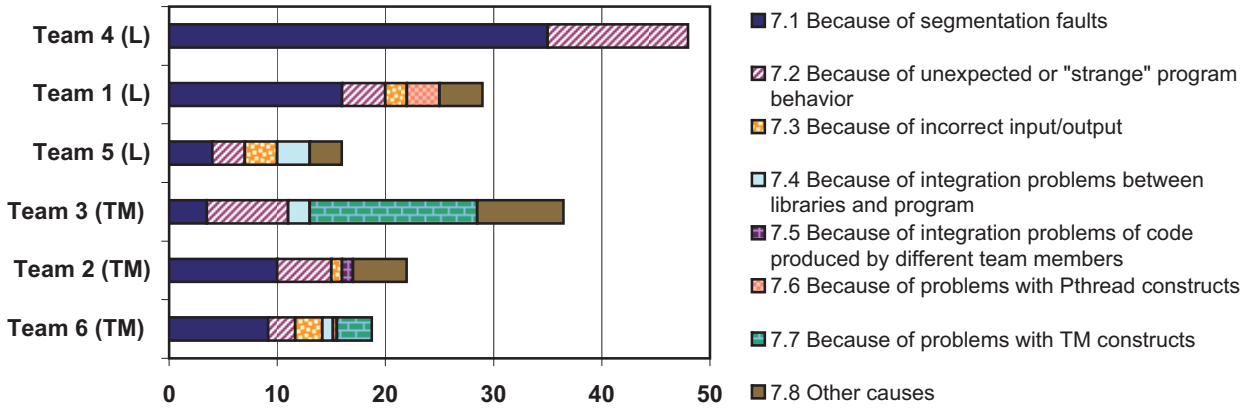
As memórias transacional surgiram com a promessa de simplificar a programação paralela através da substituição dos tradicionais mecanismos bloqueantes pelo uso de transações atômicas. Um estudo conduzido por Pankratius et al [14] tentou avaliar a usabilidade e o desempenho dos sistemas de TM através de um estudo de caso realizado com programadores reais. Para isso, doze estudantes, trabalhando em duplas, escreveram um sistema de busca paralelo em C/C++ durante um laboratório com duração de quinze semanas. Três grupos, escolhidos aleatoriamente e denominados de equipes TM, implementaram seus sistemas de busca utilizando o sistema de memória transacional em *software* desenvolvido pela Intel, além de utilizarem *pthreads* em conjunto. Por sua vez, os outros três grupos (denominados de equipes normais) implementaram seus sistemas usando apenas *pthreads*.

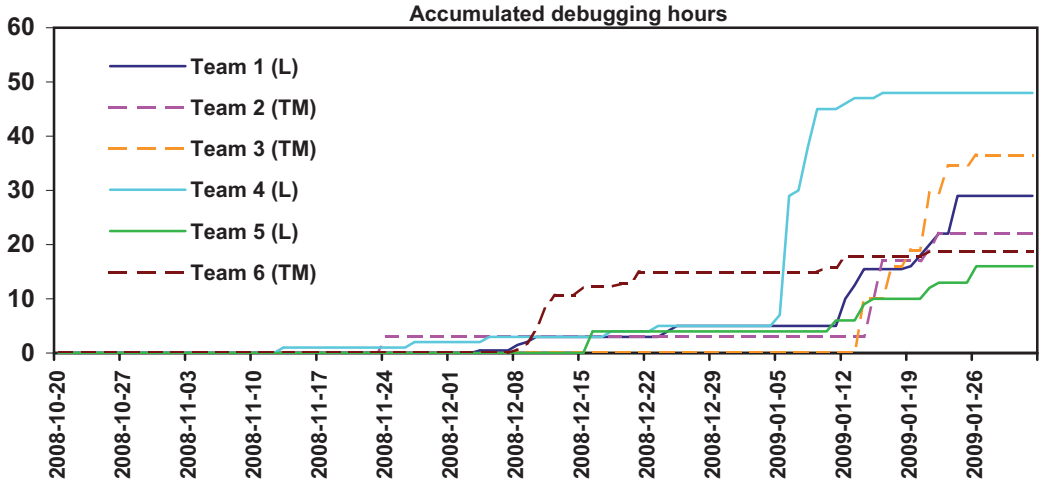
O estudo apresenta informações sobre diversos indicadores, incluindo o desempenho do aplicativo desenvolvido, a qualidade de código, as métricas utilizadas, bem como o esforço e as impressões dos programadores sobre as tecnologias utilizadas. As evidências empíricas destacam e confirmam as promessas que acompanham os sistemas de TM e, ao mesmo tempo, apontam para problemas existentes. A equipe de TM vencedora gastou menos esforço global em relação à equipa normal vencedora, além de alcançarem o melhor desempenho dentre todos os sistemas. Além disso, os vencedores que utilizaram TM foram os primeiros a ter um protótipo do sistema de busca paralela, quatro semanas mais cedo do que os demais grupos. Um dos principais benefícios dos sistemas de TM que foi observado diz respeito à depuração de código: comparado com as equipes normais, as equipes de TM gastaram menos da metade do tempo em depuração de falhas de segmentação. A Figura 2, retirada de [14], ilustra estas informações.

Todas as equipes normais tentaram melhorar a escalabilidade de seus sistemas através de um ajuste detalhista dos bloqueios utilizados, mas foi difícil para eles alcançar um desempenho escalável. Apenas uma equipe (com mais de 1600 bloqueios) conseguiu um programa paralelo evolutivo. O estudo mostra que os bloqueios e as memórias transacionais foram empregadas com sucesso, de forma complementar, pois eles não precisam ser considerados como alternativas, excluindo um o outro.

Por outro lado, as equipes de TM tiveram mais problemas para ajustar o desempenho dos sistemas de busca, isso porque o desempenho das memórias transacionais é bem mais difícil de prever do que a programação apenas com *pthreads*. As evidências mostram também que, para usufruir plenamente dos benefícios das TMs em aplicativos desenvolvidos na linguagem C++ é preciso um refinamentos na linguagem, além do uso de ferramentas de depuração com suporte para ajuste de desempenho. Para a maioria das equipes de TM, foi difícil entender o comportamento do seu programa, dificultando um pouco o desenvolvimento. O estudo de caso mostrou que, mesmo com o uso dos sistemas de memória transacional, a programação paralela continua a ser difícil e, por isso, a busca por novos recursos de linguagem de programação paralela deve continuar pelas próximas geração de desenvolvedores.

7) Time spent on debugging





	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	
Team 5 (L)	4	3	3	3	0	0	0	3	16
Team 1 (L)	16	4	2	0	0	3	0	4	29
Team 4 (L)	35	13	0	0	0	0	0	0	48
Team 6 (TM)	9	3	3	1	0	0	3	0	19
Team 2 (TM)	10	5	1	0	1	0	0	5	22
Team 3 (TM)	4	8	0	2	0	0	16	8	38
sum all	**78**	**36**	**9**	**6**	**1**	**3**	**19**	**20**	**172**
	45%	21%	5%	3%	1%	2%	11%	12%	100%
sum L	**55**	**20**	**5**	**3**	**0**	**3**	**0**	**7**	**93**
	59%	22%	5%	3%	0%	3%	0%	8%	100%
sum TM	**23**	**16**	**4**	**3**	**1**	**0**	**19**	**13**	**79**
	29%	20%	5%	4%	1%	0%	24%	16%	100%

Figura 2: Tempo gasto em depuração de código no experimento realizado em [14].

5. CONCLUSÃO

O uso de sistemas de memória transacional tem se mostrado uma alternativa viável para o desenvolvimento de aplicativos paralelos. Através destes sistemas, é possível alcançar uma abstração capaz de aliviar os programadores da tarefa de compreender e lidar com detalhes do *hardware*, isso tudo através de uma interface bem mais amigável entre a máquina e o usuário. Com o uso das memórias transacionais é possível melhorar o desempenho dos aplicativos, principalmente porque os sistemas de TM tentam eliminar as serializações tradicionalmente utilizadas nos atuais métodos de sincronização de aplicativos paralelos. Dessa forma, somente nos casos onde conflitos são detectados, há alguma penalização na aplicação.

Pesquisas em memórias transacionas são muito promissoras devido a grande poder e simplicidade de seu modelo de programação e a seu potencial de alto desempenho. Dessa forma, esta área vem atraindo interesse crescente nos últimos anos, e é ainda hoje um alvo em movimento.

6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, 2006.

[2] D. Dice, O. Shalev, and N. Shavit. *Distributed Computing*, chapter Transactional Locking II, pages 194 – 208. Springer, 2006.

[3] U. Drepper. Parallel programming with transactional memory. *Queue*, 6(5):38–45, 2008.

[4] A. K. Elmagarmid, editor. *Database transaction models for advanced applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[5] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, 1989.

[6] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

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

[9] J. R. Laus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.

[10] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, 2004.

[11] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, New York, NY, USA, 1977. ACM.

[12] K. E. Moore. *Log-based transactional memory*. PhD thesis, Madison, WI, USA, 2007. Adviser-Wood, David A.

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

[14] V. Pankratius, A. R. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? results from an empirical study. Technical report, University of Karlsruhe, 2009.

[15] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[16] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC '07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, pages 107–113, Washington, DC, USA, 2007. IEEE Computer Society.

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

[18] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisentat, C. Heirot, W. N. Scherer, and M. F. Spear. Hardware accelerarion of software transactional memory. In *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[19] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and nontransactional actions. In *1st ACM Sigplan Workshop on Languages, Compilers and Hardware Support for Transactional Computing*, 2006.