Memórias Transacionais

Nelson Luiz Geromel
Ra:958097
Instituto de Computação
UNICAMP

nlgromel@gmail.com

RESUMO

O surgimento de sistemas multiprocessadores e a promessa de ganhos consideráveis de desempenho através de processamento paralelo, tornou crítico o problema de compartilhamento de recursos, especificamente relacionado ao acesso às memórias. Em algum momento dois ou mais processadores terão que compartilhar dados existentes na memória. Vários processadores alterando um mesmo local da memória simultaneamente levará a dados inconsistentes nos processos executados por eles.

As técnicas convencionais baseadas em *lock* apresentam sérias limitações quando a quantidade de acessos concorrentes desses processadores aumentam, e não se mostraram promitentes para a resolução do problema, mesmo com significativas alterações. Nesse contexto surge o conceito de memória transacional, inspirado e semelhante ao conceito de transações em Bancos de Dados, e que pretende superar as limitações das técnicas baseadas em *lock*.

1. INTRODUÇÃO

Há alguns anos atrás, o aumento de desempenho dos processadores estava diretamente relacionado ao aumento da frequência de *clock*. Contudo, nos últimos anos a frequência dos processadores tem se mantido estável. Com o natural aumento nas demandas por processamento colocando cada vez mais tarefas para serem executadas, é imprescindível o aumento da capacidade das unidades processadoras.

Para aumentar a capacidade de processamento dos sistemas, uma solução foi incorporar outros processadores trabalhando em paralelo, com a premissa de executar diversos itens de processamento simultaneamente com elevado desempenho. Essa nova solução acarretou também em outros problemas e desafios, especificamente relacionados ao controle de acesso aos recursos que devem ser compartilhados por esses processadores. Esses problemas tem sido a causa das promessas de alto desempenho ficarem abaixo das expectativas, e para resumir, podem ser definidos numa palavra: concorrência.

O ponto crítico nessa nova forma de computação é o complexo processo de sincronizar acessos concorrentes a memória compartilhada. Uma técnica para lidar isso é chamada de Exclusão Mútua, que na prática bloqueia (*lock*) os locais na memória utilizados por um processo, impedindo que outros processadores alterem esses locais tornando-os inconsistentes.

Esse método provoca uma maior utilização dos processadores que precisam ficar verificando se os locais na memória necessários para o processamento deles estão disponíveis. Além disso, as técnicas baseadas em *lock* evidenciaram-se limitadas quando o número de acessos concorrentes aumenta. Como solução mais promissora para resolver este problema, surge então o mecanismo de memória transacional. Esse mecanismo, relativamente novo, tem por objetivo substituir o modelo antigo baseado nas convencionais técnicas de *locking*.

Conforme citado anteriormente, técnicas baseadas em *locking* apresentam uma série de problemas, entre eles, a inversão de prioridades, falhas de proteção e *deadlock*. Inversão de prioridades que ocorre quando um processo de baixa prioridade mantém um *lock* a um recurso necessário por um processo de alta prioridade. Falha de proteção ocorre quando um processo mantendo um *lock* sofre algum tipo de interrupção e outros processos ficam incapazes de progredir. *Deadlock* que ocorre quando um processo fica a espera de um recurso mantido por outro processo que por sua vez aguarda um recurso mantido pelo primeiro. Adaptar o mecanismo de *lock* para sistemas multiprocessadores incorre em grandes dificuldades de implementação e não parece ser o caminho mais promissor.

Memórias Transacionais baseiam-se num conceito semelhante ao utilizado na área de bancos de dados para controle de transações, sendo voltadas para sistemas multiprocessadores, e que pretende fazer a sincronização dos acessos à locais de memória sem a utilização de *locks*, de modo que ela seja tão eficiente e fácil de usar quanto, ou mais quando o número de acessos concorrentes crescem, que as técnicas convencionais baseadas em *locks*. A intenção é simples, maximizar a concorrência nos acessos. Ela também simplifica a programação de concorrência provendo execução atômica de um determinado bloco de operações.

O mecanismo de memória transacional é implementado através de extensões ao protocolo de coerência de cache. Antes de descrever como funciona e como é implementada a memória transacional, é necessário descrever melhor o conceito de transação.

2. O CONCEITO DE TRANSAÇÃO

Uma transação pode ser definida como uma seqüência finita de operações executadas por um único processo, segundo Herlihy e Moss [1]. Os problemas de sincronização encontrados por sistemas multiprocessadores que possuam várias aplicações ou

aplicações multi-threaded (aplicações com várias linhas de execução simultâneas) são similares aos encontrados em controle de concorrência em transações dos bancos de dados relacionais.

Transações são tradicionalmente usadas já há muito tempo para gerenciar acessos em sistemas de banco de dados, com eficácia comprovada. Imagine uma simples operação de transferência bancária entre duas contas correntes, que debita um certo valor de uma, e credita na outra. Suponha que após debitar da primeira conta ocorra um problema abortando a tarefa e o valor não seja creditado na outra conta. Isso representa uma inconsistência de dados inaceitável em bancos de dados. Esse problema é análogo ao de concorrência em sistemas multiprocessadores. Para evitar isso, o mecanismo de transações garante que para o caso acima descrito, as duas contas correntes são alteradas ou nenhuma delas será e a transferência será abortada. Na realidade, os sistemas de bancos de dados garantem que suas transações satisfaçam quatro propriedades que podem ser resumidas numa sigla: ACID, onde as letras representam as seguintes propriedades:

A - Atomicidade - todas as operações de uma transação são executadas conjuntamente (através de um commit efetivo) ou nenhuma delas é realizada, retornando ao estado inicial através de um mecanismo de roll-back ou abort.

I – Isolamento- Cada transação embora executando de forma concorrente com outras, parece estar sendo executada sozinha.

C - Consistência - Corresponde a um conjunto de operações executadas de forma correta. Início a transação a partir de um estado correto e termino deixando um estado também correto.

D- Durabilidade - Modificações executadas pelas transações devem sobreviver a falhas.

No caso de sistemas multiprocessadores, consistência e atomicidade são importantes para garantir que o programa funcione de forma semanticamente correta, fazendo que os dados continuem condizentes com o esperado nas aplicações executadas em cada processador. Locais compartilhados devem ser acessados com o mecanismo de isolamento para assegurar que sua execução termine. Contudo, o conceito de durabilidade pode ser desconsiderado devido ao aspecto transitório e volátil da memória.

Outro termo importante quando o assunto é transação refere-se à “Serializabilidade”, ou seja, as transações parecem que são executadas serialmente, dando a impressão que as operações de uma transação não estão intercaladas/escalonadas com operações de outras transações, mesmo que existam várias transações em execução simultaneamente.

3. COMO FUNCIONA E COMO É IMPLEMENTADA A MEMÓRIA TRANSACIONAL

3.1 A implementação

Para entender o mecanismo da memória transacional é necessário porém a descrição de alguns termos. Utiliza-se o termo *read set* para descrever os locais na memória lidos podendo ser

um local compartilhado. Utiliza-se o termo *write set* para representar os locais na memória lidos (de modo exclusivo) e escritos. O termo *data set* é utilizado para representar a união dos *read set* e *write set*.

O mecanismo de memória transacional oferece para acesso à memória as seguintes instruções primitivas e de manipulação do estado da transação:

- *Load Transacional (LT)* que faz a leitura de um valor de um local compartilhado na memória, colocando-o num registrador.
- *Load Transacional Exclusive (LTX)*, faz a leitura de um local compartilhado da memória, colocando-o num registrador, de forma que este local na memória provavelmente irá ser atualizado.
- *Store Transacional (ST)* escreve o valor presente num registrador para um local na memória, de forma que este valor não será visível para outros processadores até a transação efetuar um *commit*.
- *Commit* – tenta fazer que todas as alterações de uma transação sejam efetivadas. Um *commit* é bem sucedido se nenhuma outra transação alterou qualquer local no seu *data set* e nenhuma outra transação leu qualquer local do seu *write set*. Se o *commit* falha, todas as alterações no seu *write set* são descartadas.
- *Abort* – desfaz as alterações de uma transação que não conseguiu efetuar um *commit*.
- *Validate* – testa o *status* de uma transação. Um comando *Validade* que retorna *True* indica que a transação ainda não foi abortada, embora ela posteriormente possa ser. Caso contrário o *Validade* retorna *False* indicando que a transação foi abortada.

A implementação da Memória Transacional é feita modificando o protocolo de coerência de cache. A idéia básica é atualizar as instruções convencionais utilizadas para a detecção de conflitos com as instruções definidas para a memória transacional para detectar e resolver conflitos em transações.

De modo geral a memória transacional pode ser implementada de forma a suportar conjuntamente instruções de memória não transacionais, embora o processo de interação entre a transacional e a não transacional requeira maior atenção, principalmente no que tange às circunstâncias que podem levar a transações serem abortadas ou ficarem num processo de inanição.

Em [1] sugere-se que uma alternativa para resolver conflitos seja simplesmente abortar a transação ou emitir um sinal de erro quando uma operação não transacional altera algum de seus endereços. A idéia básica é manter duas caches, uma regular para operações não transacionais e outra transacional, de forma que uma determinada entrada na cache pode ficar em uma ou outra, mas não em ambas.

3.2 O Funcionamento

Com relação à forma de utilização, as memórias transacionais são mais adequadas para atualização de pequenas porções de memória, ou seja, uma conjunto não muito grande de

operações. O processo consiste em efetuar leitura LT ou leitura exclusiva LTX de um certo conjunto de locais na memória compartilhada. Usar a instrução *Validade* para assegurar que os valores lidos nessas regiões são consistentes. Se o *Validade* falhar, volta-se a efetuar LT ou LTX. A instrução *Validade* é utilizada principalmente para se detectar quando um conjunto de valores na memória torna-se inconsistente.

Após isso, efetuar as modificações através da instrução ST para as regiões que devem ter valores alterados. Posteriormente, efetuar um *commit* para tornar as alterações efetivas. Se o *commit* falhar volta-se ao passo inicial de leitura, ou seja, LT ou LTX. O *commit* falha quando uma outra transação atualiza dados na região correspondente ao seu *data set*, onde o *commit* iria efetivar as operações da transação. Ele também falha quando outra transação faz leitura num dos locais da memória correspondente ao seu *write set*. Após o *commit*, pode-se considerar que a transação foi concluída.

Além dessas instruções a memória transacional ainda implementa alguns outros dispositivos que podem ser assim descritos: Para cada entrada da cache é associado um atributo, que pode ser implementado com dois bits apenas, onde é definido seu estado, os quais pode ter um dos seguintes rótulos INVALID, VALID, DIRTY, RESERVED, conforme definido em [1] e mostrado na tabela 1, descrita abaixo. Cada um desses rótulos tem um significado específico quanto às propriedades de permissão de acesso (*Access*), compartilhamento (*Shared*) e se foi modificado (*modified*). INVALID significa uma entrada de cache inválida. VALID indica uma entrada de cache com permissão de acesso somente para leitura, que é também definida como compartilhada, e possui dados idênticos na memória principal. DIRTY indica uma entrada com permissão de leitura e escrita, que não é compartilhada e com valor que foi modificado em relação à memória principal. RESERVED indica uma entrada com permissão de leitura e escrita, não compartilhada e que possui valores que não foram modificados.

Tabela 1 – Possíveis estados das entradas da cache. R significa permissão de leitura, e o W de escrita.

Nome	Access	Shared	Modified
INVALID	Nenhum		
VALID	R	Sim	Não
DIRTY	R, W	Não	Sim
RESERVED	R, W	Não	Não

A esse conjunto de propriedades que definem um estado, são acrescentadas ainda as *transactional tags*, mostradas na tabela 2, e que podem ter os seguintes rótulos, conforme protocolo definido por Goodman [2]: EMPTY para descrever que a entrada na cache está vazia, NORMAL contendo dados que efetuaram um *commit*, XCOMMIT com dados descartados quando comitado e XABORT com dados descartados quando abortado. As operações em cache tem duas entradas com esses valores. Esses valores são marcados como XCOMMIT e XABORT. Quando uma transação efetua um *commit* suas entradas são alteradas de XCOMMIT para EMPTY e de XABORT para NORMAL. Se a transação aborta, suas entradas são alteradas de XABORT para EMPTY e de XCOMMIT para NORMAL.

Tabela 2 – Transactional Tags

Nome	Significado
EMPTY	Não contém dados
NORMAL	Contém dados comitados
XCOMMIT	Descartar quando comitados
XABORT	Descartar quando abortados

O processo começa preenchendo uma entrada da cache. Para isso ele examina a cache transacional por uma entrada marcada com NORMAL ou XCOMMIT. Se não encontra, então ele passa a ocupar essa entrada. Se encontrar uma entrada marcada como NORMAL ele cria uma segunda cópia da entrada, fazendo ainda uma delas ser marcada como XABORT e a outra como XCOMMIT. As alterações são direcionadas apenas para as entradas com XABORT. As entradas com XCOMMIT contém dados que já estão na memória principal por isso quando a transação aborta não é necessário acessá-la.

Quando uma instrução ST ocorre, modificam-se as entradas rotuladas como XABORT e o valor antigo do dado deve ser preservado até o momento que a transação seja concluída com um *commit*. Após isso, os valores tornam-se visíveis aos outros processadores e as alterações são propagadas até a memória principal. Percebe-se que o rótulo XCOMMIT é utilizado praticamente apenas para aumento de desempenho. Com o valor antigo ainda na cache e o *status* como DIRTY significa que a entrada deve estar marcada como XCOMMIT ou então escrita na memória principal. O princípio básico é manter todas as tentativas de escrita sem propagá-las aos outros processadores ou para a memória principal, a não ser que a transação conclua com sucesso através de um *commit*. Sendo abortada a transação, todas as tentativas de escrita são abortadas e os valores anteriores preservados.

Quando uma cache transacional precisa de espaço para uma nova entrada, ela pesquisa por todas que possuem o valor EMPTY, depois pelas entradas com valor NORMAL e depois com XCOMMIT. No trabalho de [1], considerou-se uma cache transacional pequena e completamente associativa (*full associative*). Caso a entrada com XCOMMIT esteja também com o status DIRTY, então ela deve ser escrita na memória.

Com relação ao barramento, o funcionamento é descrito iniciando com o processador executando uma LT ocupando uma entrada na cache transacional. Se o estado da entrada é INVALID então o processador emite um READ. Se a resposta for BUSY, o processador aborta a transação, senão ele marca a entrada no estado VALID. No caso da execução de uma LTX, se a entrada não estiver marcada com os rótulos RESERVED ou DIRTY, o processador emite um READ. Se a resposta for BUSY a transação também é abortada, senão a entrada é marcada com o rótulo RESERVED para ser alterada posteriormente. Para a primitiva ST o funcionamento é o mesmo da instrução LTX, exceto pelo rótulo que é então marcado do DIRTY.

Referente ao processador, cada transação também mantém duas *flags* que são: TSTATUS e TACTIVE. TACTIVE indica se a transação está em progresso ou não. Se TACTIVE indica progresso, a variável TSTATUS descreve se uma transação está ativa ou não. Se a variável TACTIVE for igual a *true*, então a

instrução *Validade* retornará *true*. Tanto o COMMIT e o ABORT marcam a variável TACTIVE como *false* e variável TSTATUS como *true*.

Contudo a memória transacional sofre com alguns problemas. Se a cache não for grande o bastante para guardar todas as operações de uma transação ela será abortada. Transações mais demoradas tendem a ser abortadas com mais frequência, pois estão sujeitas num período de tempo maior à interferências nos seus dados, aumentando as chances de serem abortadas. Transações longas também, pois possuem mais dados, os quais podem ser utilizados por outros processos.

3.3 MEMÓRIA TRANSACIONAL EM HARDWARE

As implementações de memória transacional propostas por [1] eram basicamente em hardware. Sistemas com memória transacional em hardware geralmente fornecem um conjunto de instruções que definem os limites e a forma de manipulação das transações. Ela tem a vantagem de possuir um alto desempenho, contudo não oferecem muita flexibilidade e não mostra de forma clara um caminho de migração para software que deve rodar sobre máquinas antigas. Requer também que programadores estejam atentos para limitações específicas de uma plataforma, tamanho de cache, tamanho das transações, falhas de página, troca de contexto, entre outros. Eles falham também no suporte a vários aspectos das modernas linguagens de programação e de sistemas operacionais tais como chamadas de bibliotecas, sincronização condicional, chamadas de sistemas I/O e exceções em tempo de execução.

Além disso memória transacional em hardware é semanticamente insuficiente para suportar as propostas de linguagens recentes e sistemas em *runtime* que fornecem um modelo de programação concorrente de fácil uso. Para que sistemas de memória transacional em hardware tornem-se úteis e consequentemente com ampla aceitação é necessário esconder essas limitações anteriormente citadas, além de definir interfaces claras entre o hardware e o software transacional.

3.4 MEMÓRIA TRANSACIONAL EM SOFTWARE

Memória transacional em software disponibiliza aos programadores construções para delimitar operações transacionais e implicitamente cuidar da corretitude de acessos concorrentes em dados compartilhados. A memória transacional em software descrita por Shavit e Touitou [6] foi motivada pela necessidade de uma implementação mais fácil e portátil, nos computadores existentes naquela época. A implementação em software possibilita flexibilidade para explorar diferentes semânticas e o desenvolvimento sobre o hardware convencional.

A memória transacional em software é implementada através de mecanismos de *lock* e outros, que possam prover Exclusão Mútua, geralmente já existentes nos computadores, ou seja, o hardware convencional como atualmente implementado.

O Software transacional agrupa um conjunto de operações numa transação e tenta efetuar um *lock* em todos os endereços dos locais de memória que participam da transação. Se não consegue o *lock* num local, então a transação aborta e desfaz todos os *locks* efetuados antes. Se ele consegue efetuar todos os *locks*, então executa a transação e libera os *locks* efetuados. Na prática o software transacional trabalha apenas com transações estáticas, isto é, que dependem de acessos com uma seqüência estabelecida de locais de memória.

A memória transacional em software apresenta como vantagens a facilidade de implementação e portabilidade, pois pode ser implementada em todos os conjuntos de equipamentos existentes. Sua grande desvantagem é possuir alto *overhead*. Além disso não é plenamente esclarecido como são detectados e resolvidos os conflitos entre acessos transacionais e não transacionais.

3.5 MEMÓRIA TRANSACIONAL HÍBRIDA

Ambas as abordagens de hardware e software para memórias transacionais possuem vantagens e desvantagens. Reconhecidamente abordagens em hardware colocam muitas restrições sobre os programadores, mas tem a vantagem de sua velocidade. Abordagens em software mostraram-se mais flexíveis, contudo muito lentas devido ao seu *overhead*. Intuitivamente, o caminho natural parece ser a combinação de ambas as abordagens com a intenção de aproveitar as suas virtudes. Atualmente parece que grande parte das pesquisas apontam para esse caminho, e grande numero de propostas estão sendo produzidas.

Na proposta de Lie e Ananian [5] as transações sempre começam trabalhando sobre memória transacional em hardware. Se ela abortar então é tentada a execução em memória transacional em software. Dessa forma a transação pode se beneficiar do desempenho de execução no hardware transacional e caso ocorra alguma problema de aborto ela pode ser executada em software transacional, neste caso sofrendo o problema do *overhead*. Entretanto, é melhor executar de modo mais lento, a não executar nada daquela transação.

3.6 TRABALHOS ATUAIS

Grande parte das atuais propostas parecem apontar para o desenvolvimento abordagens híbridas aproveitando os aspectos positivos de cada uma.

Uma proposta defendida por Shriraman e outros [4], na qual hardware é usado para acelerar a implementação da memória transacional, controlada fundamentalmente por software. Essa implementação consiste de um novo protocolo transacional MESI (TMESI), acompanhando o software de memória transacional. A concepção básica é utilizar TMESI para reduzir o *overhead* da cópia de dados, sem introduzir um novo algoritmo para o protocolo de coerência de cache. A única alteração no protocolo snoop e um sinal "*threatenet*" análogo ao sinal "*shared*" existente. Ela deixa para o software as políticas de gerenciamento de

contenção, *deadlock*, granularidade de dados, permitindo o experimento de diferentes políticas.

Nos trabalhos realizados por Smaragdakis e outros [3] defende-se que uma possível integração entre as técnicas baseadas em *lock* e de memória transacional ainda não foram suficientemente exploradas. A idéia deste trabalho é transformar *locks* em seções atômicas. Eles propõem a implementação de um mecanismo adaptativo que transforme um conjunto de *locks* em transações, e seu retorno para *locks* regulares. O ponto principal deste trabalho é uma análise de viabilidade que dinamicamente decida quando trocar/transformar a técnica baseada em *lock* para a técnica transacional.

4. CONCLUSÕES

Inevitavelmente os sistemas processadores e multiprocessadores serão submetidos para demandas sempre crescentes de execução de novos processos. Aumentar o número de processadores está sendo a solução mais indicada como resposta para este aumento de demanda. Consequentemente, o aumento do número de processadores aumenta também a concorrência aos dispositivos.

A memória transacional surge como solução adequada para tratar o problema de concorrência causado pelos acessos simultâneos aos meios compartilhados. Memórias transacionais satisfazem as propriedades de atomicidade e serializabilidade, bem como outras, que por sua vez conseguem manipular eficazmente acessos concorrentes, evitando os problemas mais comuns associados com as técnicas não transacionais, como *deadlock* e inversão de prioridades. Seu principal atrativo está ligado ao seu desempenho em relação às técnicas convencionais, quando a quantidade de acessos concorrentes aumenta significativamente.

Variadas formas de implementação mostraram-se possíveis e foram propostas, baseadas em hardware, software e híbridas, cada qual com suas vantagens e desvantagens. Contudo, parece que as propostas mais plausíveis incorrem na combinação de ambas abordagens, de software e hardware.

De concreto, o futuro parece apontar inevitavelmente para a programação concorrente com a utilização de memória transacional, afim de aproveitar aspectos inerentes das novas arquiteturas com várias unidades processadoras, além do paralelismo cada vez mais presentes nos diversos dispositivos dos computadores.

5. REFERENCIAS BIBLIOGRÁFICAS

- [1] Herlihy, M., and Moss, J., 1992, Transactional Memory: Architectural support for lock-free data structures. Technical Report 92/07, Digital Cambridge Research Lab, One Kendall Square, Cambridge MA 02139, Dezembro de 1992.
- [2] Goodman, J. R., Vernon, M. K, and Woest, P. J., 1993, Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors, In Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pages 64-75, Abril de 1989.
- [3] Smaragdakis, Y., Kay, A., Behrends, R. and Young, M., 2008. General and Efficient Locking without Blocking, Memory Systems Performance and Correctness – MSPC 08, Seattle, Washington, 02 de Março de 2008.
- [4] Shriraman, A., Marathe, V.J., Dwarkadas, S. and Scott, M.L., 2006., Hardware Acceleration of Software Transactional Memory, TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, Ontario, Canadá Junho de 2006.
- [5] Lie, S. and Ananian C. S., 2004. Na Integrated Hardware-Software Approach to Transactional Memory, MIT Computer Science and Artificial Intelligence Laboratory, Fevereiro de 2004.
- [6] N. Shavit and D. Touitou, "Software transactional memory," Distributed Computing, vol. 10, pp. 99-116, 1997.