Memórias Transacionais

João Batista Correa G. Moreira, RA: 087331
Instituto de Computação
UNICAMP
Campinas, São Paulo
joao@livewire.com.br

ABSTRACT
O surgimento de arquiteturas computacionais multiprocessadas impõe novas dificuldades no desenvolvimento de software. Dentre estas dificuldades está a necessidade de sincronização dos fluxos de execução de forma a garantir acessos ordenados à memória compartilhada. A sincronização tradicionalmente é realizada com o uso de travas de exclusão mútua ou semáforos, porém estas estruturas são ineficientes e difíceis de usar. As memórias transacionais surgem como alternativa para a sincronização de um software paralelo, fornecendo uma abstração simples e garantia de consistência dos dados.

Keywords
Memória Transacional, Arquitetura de Computadores, Programação Paralela

1. INTRODUÇÃO
A evolução dos processadores seguiu por muito tempo um modelo baseado no aumento da freqüência do *clock* para conseguir um maior poder de processamento. Este modelo de evolução, no entanto, esbarrou em limitações físicas que impossibilitam a alimentação e o resfriamento adequado dos processadores que estariam por vir, exigindo novas alternativas para o desenvolvimento de computadores mais velozes. Buscando solucionar este problema, a indústria passou a desenvolver os chamados processadores *multicores*, com dois ou mais núcleos em um único *chip*. Esta solução mostra-se interessante uma vez que possibilita a continuidade da evolução dos processadores pelos próximos anos.

Para que os novos processadores sejam devidamente aproveitados, o software que eles executam precisa fazer uso adequado dos núcleos adicionais. Tradicionalmente um software é desenvolvido para executar seqüencialmente, o que exige a compreensão de novos paradigmas de programação por parte dos desenvolvedores que pretendem escrever aplicações paralelas. Além disto, se comparado a um software seqüencial equivalente, fatores como concorrência e não-determinismo tornam o software paralelo muito mais difícil de se programar. Os impactos gerados pelas arquiteturas paralelas e os problemas envolvidos na sua programação são descritos em [24].

O desenvolvimento de um software paralelo, como é feito hoje, deixa a cargo do programador todo o trabalho de sincronização dos fluxos de execução (*threads*). A sincronização das *threads* garante que o acesso aos dados compartilhados seja realizado de forma correta, garantindo a execução consistente do software. Os mecanismos utilizados pelos programadores para evitar que uma *thread* interfira inadequadamente na execução das demais *threads* concorrentes são travas de exclusão mútua (*locks*) e os semáforos.

Estas estruturas, *locks* e semáforos, controlam a execução de seções críticas nas diferentes *threads* de um programa. As seções críticas de um software dizem respeito a trechos de código que, quando executados em paralelo, podem tentar acessar recursos compartilhados de forma desordenada e levar a execução a estados inconsistentes de memória. Conforme descrito em [13], estes mecanismos não oferecem uma abstração adequada ao uso uma vez que todos os detalhes de implementação relacionados a sincronização são utilizados de forma explícita. Esta característica torna tais mecanismos difíceis de se usar corretamente, levando freqüentemente a problemas comuns em um software paralelo, como enfileiramentos e bloqueios mútuos (*deadlocks*).

Com o objetivo de facilitar o desenvolvimento de um software paralelo, surgiram os sistemas de memória transacional (*Transactional Memory, TM*). Esta proposta de modelo de programação fornece uma abstração semelhante aos sistemas de transações utilizados em bancos de dados, onde propriedades como atomicidade, consistência e isolamento são garantidas. Desta forma, os sistemas de memória transacional encarregam-se da sincronização de um software paralelo, possibilitando a execução de operações corretas de leitura e escrita na memória sem a necessidade do uso de mecanismos como *locks* e semáforos.

Com o uso em larga escala dos processadores *multicore*, o interesse pelas memórias transacionais aumentou bastante, resultando em uma grande quantidade de pesquisas sobre o assunto. Diferentes sistemas de memória transacional tem sido desenvolvidos e testados. Alguns destes sistemas consistem em bibliotecas de software, sendo chamados *Software Trans-*

actional Memories (*STM*). Outros sistemas contam com recursos de hardware para diminuir o impacto e atingir um melhor desempenho, compondo o que é chamado de *Hardware Transactional Memory* (*HTM*). Sistemas de memória transacional que combinam recursos oferecidos pelo *Hardware* com técnicas em *Software* são chamados *Hybrid Transactional Memories* (*HyTM*)

2. TRANSAÇÕES

Segundo [13], uma transação é uma seqüência de operações cuja execução é vista de maneira instantânea e indivisível por um observador externo. O conceito de transação é amplamente utilizado em sistemas de gerenciamento de banco de dados, garantindo quatro propriedades necessárias: Atomicidade, Consistência, Isolamento e Durabilidade.

Considerando o modelo de programação paralela tradicional, pode se dizer que uma transação é equivalente às seções críticas definidas com o uso de *locks* e semáforos, garantindo que uma *thread* não interfira no fluxo de execução das demais, porém evitando serializações desnecessárias e definidas com uma sintaxe mais simples e fácil de compreender.

Quando duas transações tentam acessar os mesmos locais de memória simultaneamente, diz-se que estas transações entraram em conflito e as operações realizadas por uma delas não deve ser aplicada ao sistema. Esta característica é implementada através do uso de mecanismos de execução especulativa ou por mecanismos de *roll-back*. Em uma execução especulativa, as transações são executadas e seus efeitos são armazenados em um *buffer*, só sendo aplicados ao sistema caso a transação seja concluída sem ocorrência de conflitos. Nos mecanismos de *roll-back*, uma transação é executada e os seus efeitos são aplicados diretamente ao sistema, porém o estado de memória anterior ao início da transação é armazenado de forma que o sistema possa recuperá-lo caso um conflito seja detectado. Alguns mecanismos de *roll-back* utilizam *logs* que armazenam apenas as operações executadas na transação em vez de guardar todo o estado de memória anterior a transação.

Por **Atomicidade** entende-se que todas as operações de uma transação executam com sucesso ou nenhuma das operações é executada. Ao final de uma transação, todos os seus efeitos são aplicados ao sistema através de um *commit* efetivo, ou então são descartados através de um *abort*.

Consistência garante que as mudanças geradas por uma transação não levarão o sistema a um estado incorreto. Por estado correto pode-se entender um estado em que todos os dados armazenados em memória são valores certos.

Isolamento garante que cada uma das transações produzirá um resultado correto, independente de quantas e quais transações sejam executadas simultaneamente. Esta propriedade garante que o sistema não apresente erros gerados pelo uso incorreto de memória compartilhada, como pode ocorrer em um software paralelo que não possua um sistema adequado de sincronização

Durabilidade assegura que os resultados gerados por uma transação concluída sejam permanentes e estejam disponíveis para as próximas transações. Esta propriedade não é importante para sistemas de memória transacional, uma vez que os dados em memória são considerados transientes.

Um bloco de código atômico precisa ser delimitado de forma a identificar quais instruções fazem parte da transação especificamente. Um exemplo de bloco de código em que se define uma transação pode ser visualizado na Figura 1:

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

Figure 1: Definição de um bloco atômico

3. MEMÓRIA TRANSACIONAL

A idéia geral por trás dos sistemas de memória transacional consiste no uso de estruturas para definição das transações. O sistema de memória transacional se encarrega por executar as transações, detectar possíveis conflitos e gerencia-los, caso existam. Conforme descrito em [13], cada sistema de memória transacional possui um conjunto de características próprias, especificando como este sistema lida com as transações. Os mecanismos utilizados na detecção e tratamento de conflitos, no isolamento de memória, bem como estruturas para definição das transações podem variar bastante em cada implementação.

Algumas das classificações das principais características de um sistema de memória transacional são:

- Detecção de conflitos
  - *Eager* : O sistema detecta conflitos existentes entre transações durante sua execução, isto é, antes da realização dos *commits*.
  - *Lazy* : O sistema detecta os conflitos quando uma transação encerra sua execução e tenta realizar o *commit*.
- Isolamento
  - Fraco: Um acesso a memória executado fora de uma transação não entra em conflito com acessos a memória executados dentro de uma transação, podendo eventualmente corromper a execução do programa. Este modelo induz todos os acessos a memória compartilhada a serem feitos dentro de uma transação.
  - Forte: Converte todas as operações fora de uma transação em pequenas transações individuais, executando todos os acessos a memória compartilhada como se fossem transações. Neste modelo, referências de memória externas a uma transação entram em conflito com referências de memória executadas dentro de uma transação.
- Gerenciamento de versões
  - *Eager*: Mantém o estado de memória anterior a transação em um *buffer* ou *log*, aplicando os valores computados na transação imediatamente. Esta estratégia beneficia operações de *commit*, porém torna operações de *abort* mais complicadas e lentas.

– *Lazy*: Mantém o estado de memória anterior até o momento do *commit* efetivo. Este modelo beneficia operações de *abort*, porém aumenta o custo das operações de *commit*.

A Figura 2 ilustra um fluxograma da execução de uma transação em um sistema com detecção de conflitos *Lazy*.

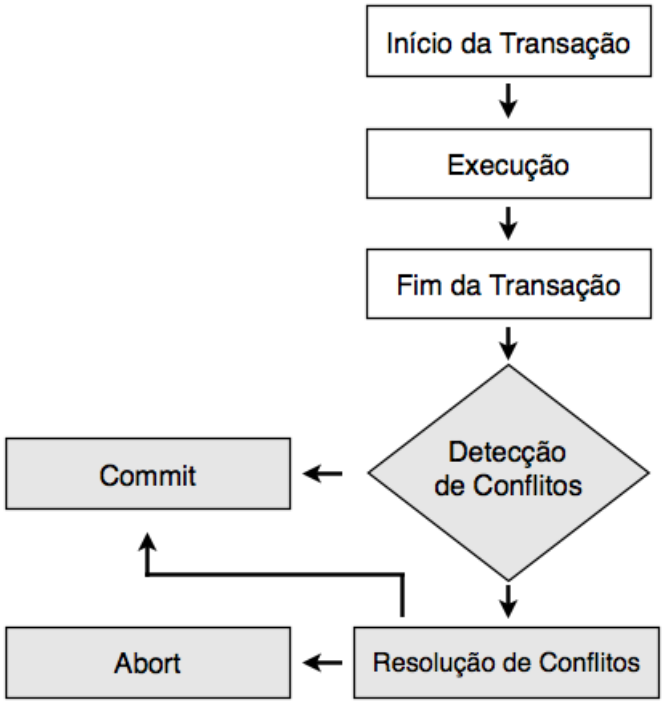

Figure 2: Fluxo de uma transação em um Sistema TM com detecção de conflitos *Lazy*

Outra importante característica de um sistema de memória transacional é o gerenciamento de contenção. O gerenciador de contenção é a parte do sistema responsável por lidar com os conflitos, aplicando diferentes políticas para decidir qual das transações conflitantes será abortada e qual delas efetuará o *commit*. Estas políticas, chamadas políticas de resolução de conflitos, consistem em regras que auxiliam na decisão a respeito de *aborts* e *commits* sobre transações conflitantes, de forma que esta escolha minimize o custo decorrente do conflito. Algumas políticas de gerenciamento de contenção são [4]:

- *Committer Wins*: Quando o conflito é detectado na fase de *commit* de uma transação, esta transação sempre é efetivada e as demais transações conflitantes são abortadas.
- *Requester Wins*: Quando uma transação realiza acesso a um endereço de memória compartilhado que esteja em uso por outra transação, esta continua executando e as demais transações conflitantes são abortadas.
- *Requester Stalls*: O sistema bloqueia transações que tentam realizar acesso a um endereço de memória compartilhado em uso por outra transação. Caso um potencial *deadlock* seja detectado, o sistema aborta esta transação.

Em um sistema de memória transacional, espera-se que o gerenciador de contenção seja capaz de garantir o fluxo contínuo do programa. Em [13] afirma-se que a política de gerenciamento de contenção possa ser especificada pelo programador para cada uma das transações ou ainda ser escolhida automaticamente pelo sistema de memória transacional.

A escolha ideal do sistema de memória transacional e, consequentemente, das suas características, depende diretamente da aplicação em que será utilizado. Um exemplo são os sistemas de memória transacional com detecção de conflitos *Eager*. Estes sistemas evitam que as transações executem de maneira desnecessária, detectando os conflitos de maneira antecipada. Por outro lado, estes sistemas impõem maior complexidade ao modelo de memória transacional utilizado, exigindo mais recursos computacionais. Normalmente a escolha do sistema de detecção de conflitos é feita com base nos *throughputs* observados em execuções com cada um deles.

3.1 Software Transactional Memory

Um STM implementa transações não duráveis para a manipulação de dados compartilhados entre *threads*. A maioria dos STMs pode ser executada em um processador convencional, porém pouco se conhece a respeito do custo computacional adicional introduzido ao se utilizar tal recurso. As principais vantagens relacionadas ao uso de STMs são:

- Software é mais flexível que o hardware, permitindo variadas implementações de sofisticados algoritmos
- É mais fácil modificar e desenvolver software que hardware
- STMs são mais fáceis de se integrar com os sistemas de software existentes
- STMs possuem poucas limitações intrínsecas impostas por capacidades físicas do hardware, como *caches*

Quando um software é desenvolvido com o uso de uma STM, as seções críticas do seu código são demarcadas, identificando-as como transações. Parte do sistema é responsável por realizar a detecção de conflitos entre transações. Quando conflitos são detectados o gerenciador de contenção seleciona quais transações deverão ser abortadas e quais transações serão concluídas. Ao utilizar um sistema de memória transacional, o desenvolvedor só se preocupa com definir quais as seções críticas do seu código. Todo o trabalho adicional é realizado pelo sistema. Este modelo genérico de memória transacional, em que se baseiam a maioria das implementações, é descrito em [13].

O termo *Software Transactional Memory* surgiu em [21], publicado em 1995, onde se descreveu a primeira implementação de STM. Este sistema exigia que uma transação declarasse todos os acessos a memória que seriam realizados em seu início, solicitando automaticamente a posse dos *locks*

referentes aos dados. Este mecanismo introduziu um custo computacional significativo ao software.

O sistema de memória transacional RSTM é descrito em [20]. 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 as mesmas são 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 reexecutar. 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.

Em [7], um sistema de memória transacional chamado TL2 é descrito. Este sistema baseia-se na idéia de manter um *clock* 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 seguido, garantindo a consistência dos dados compartilhados. O *clock* global funciona como um *lock* que é adquirido e incrementado no final de uma execução especulativa de uma transação. As transações que só executam leituras são bastante facilitadas neste sistema, uma vez que só é necessário verificar se o *clock* global não foi acessado entre o início e o fim da transação.

Em [8] descreve-se um sistema de memória transacional que também utiliza *locks*. Este sistema, chamado TinySTM, utiliza um algoritmo similar ao utilizado no sistema TL2, descrito em [7], exceto por adquirir os *locks* sempre que uma variável é acessada. A execução de *benchmarks* demonstrou que esta característica proporcionou maior *throughput* em relação aos testes realizados com o sistema TL2. Em [8] também é demonstrada uma técnica utilizada para configuração dos parâmetros do sistema de memória transacional, onde um algoritmo de subida da colina é utilizado a partir de vários valores aleatórios.

3.2 Hardware Transactional Memory

Sistemas de memória transacional em Hardware consistem em sistemas que suportam a execução de transações em dados compartilhados, respeitando as propriedades Atomicidade, Consistência e Isolamento. O principal objetivo destes sistemas é atingir melhor eficiencia com menor custo. As principais vantagens relacionadas ao uso de HTMs são:

- HTMs conseguem executar aplicações com menor custo adicional que STMs.
- HTMs apresentam menor consumo de energia.
- HTMs existem de maneira independente aos compiladores e as características de acesso a memória.
- HTMs fornecem alta eficiência e isolamento forte sem a necessidade de grandes modificações nas aplicações

A execução comum de um programa baseia-se num modelo tradicional em que um contador de programa aponta para uma instrução, esta instrução é lida pelo processador, decodificada, executada e, quando concluida, o contador de programa é incrementado apontando para uma nova instrução que passará pelo mesmo ciclo. Em processadores que suportam execuções paralelas, cada uma das threads executando no processador executa a sequência descrita.

Apesar de manter o modelo de execução sequencial, os processadores executam diversas tarefas em paralelo. O processador aplica ao código sequencial vários mecanismos de controle de fluxo e previsão de dependências para poder executar suas instruções sequenciais em paralelo, fora da ordem prevista, e ainda assim produzir resultados corretos. Para cumprir esta tarefa, é necessário que o processador mantenha informações que possibilitem a correção do estado de execução no caso de uma predição incorreta.

HTMs baseiam-se em mecanismos que possibilitam a execução otimista (conflito não existe, a menos que seja detectado) de uma seqüência de instruções, detecção de eventuais conflitos no acesso aos dados e uso de *caches* para armazenar modificações temporárias que só serão visíveis após o *commit*. HTMs exploram vários mecanismos de hardware, como execução especulativa, *checkpoints* de registradores, *caches* e coerência de *cache*.

Segundo [13], os sistemas de memória transacional em hardware podem ser divididos em Abordagens percursoras, *Unbounded HTMs*, *Bounded/Large HTMs* e *Hybrid TMs/ Hardware Accelerated TMs*.

3.2.1 Abordagens percursoras

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

Em 1988 Chang e Mergen publicaram dois artigos [6, 17] descrevendo o *IBM 801 Storage Manager System*, em que suporte a *locks* para transações em bancos de dados seriam suportadas pelo hardware. Este é o primeiro caso conhecido de implementação de suporte a *locks* e transações em hardware. Caso um acesso a memória não encontrasse os respectivos *locks* em um determinado estado, uma função em software seria automaticamente invocada para realizar o gerenciamento do *lock* e garantir o sucesso da transação. Esta proposta incluía novos registradores ao processador para manter informações sobre as transações.

Knight descreve em [11] um sistema de paralelização especulativa de programas com apenas um fluxo de execução. O compilador seria responsável por dividir o programa em blocos de código chamados transações, assumindo que não exista nenhuma dependência de dados entre eles. Estas transações seriam executadas especulativamente e conflitos poderiam ser detectados através das *caches*.

O *Oklahoma Update protocol*, proposto por Stone et. all, é descrito em [23]. Este artigo propõe implementações em hardware para operações atômicas do tipo *read-modify-write* em um número limitado de locais de memória. Esta abor-

dagem é uma alternativa as seções críticas, cujo objetivo era simplificar a construção de blocos de código concorrentes e não bloqueantes, possibilitando a atualização automática de multiplas variáveis compartilhadas. Esta proposta incluía novas instruções que utilizariam novos registradores, chamados *Reservation Registers*, para apontar endereços e armazenar atualizações. Atualizações realizadas nestes endereços poderiam ser monitoradas através dos protocolos de coerência da *cache*.

Uma proposta similar ao *Oklahoma Update protocol* foi feita por Herlihy e Moss em [10]. Esta proposta incluía a adição de novas instruções e *cache* transacional para monitorar e armazenar dados transacionais. Este artigo é responsável pelo termo *Memória Transacional* e por iniciar o uso de mecanismos de *cache* para realizar sincronização otimista de dados compartilhados, ao contrário do *Oklahoma Update protocol*, que utilizava registradores no lugar de *caches* transacionais.

Ravi Rajwar e James R. Goodwin propuseram em [18] um sistema onde os *locks* de um software seriam reconhecidos e removidos pelo hardware, eliminando possíveis serializações. Este sistema executaria as transações de forma especulativa, isto é, todas as modificações geradas pela transação seriam escritas em um *buffer* local até que esta chegue ao fim. Quando a transação encerra, caso nenhum conflito seja detectado, as operações realizadas no *buffer* são aplicadas nos verdadeiros endereços de memória do sistema, tornando as modificações visíveis para os demais processos que compartilham o hardware. Executar as transações de maneira especulativa é uma estratégia utilizada em vários sistemas de memória transacional até hoje, inclusive em STMs, como pode ser observado em [20, 7, 8].

3.2.2 Bounded/Large HTMs

Alguns sistemas de HTM permitem que uma transação utilize informações além das fornecidas pela cache de dados. No entanto estes sistemas não permitem que uma thread seja migrada para outro processador durante sua execução ou sobrevivam a mudanças de contexto. Estes sistemas são chamados *Bounded/Larges HTMS*.

Transactional Coherence and Consistency (TCC) é o modelo de memória compartilhada descrito em [9]. Neste modelo, todas as operações são executadas dentro de transações. As transações são divididas pelo programador ou pelo compilador e podem possuir dependencias de dados. O hardware TCC executa estas transações em paralelo e, quando uma alteração é realizada na memória, esta é informada através de *broadcasts* aos demais processadores. Neste modelo, uma transação é executada especulativamente sem a realização de qualquer tipo de solicitação de *lock* na cache. As modificações são armazenadas em um *buffer* local. Múltiplas transações podem escrever no mesmo local de memória armazenado na *cache* local de maneira concorrente, porém, quando uma transação estiver pronta para realizar o *commit*, seu processador solicita permissão. Esta permissão define qual transação realizará o *commit*, sendo que apenas uma transação poderá realizá-lo por vez. Uma vez realizado o *commit*, esta transação envia um *broadcast* informando os outros processadores quais escritas foram feitas na memória. Estes processadores comparam o conjunto de escritas recebido no *broadcast* com seus próprios conjuntos de escrita, se houver um conflito, estas transações são abortadas e reexecutadas. Se uma transação excede as informações das *caches* locais, ela entra em modo não especulativo, solicitando o *lock* global para evitar que outras transações realizem *commits*.

Large Transactional Memory é uma implementação de memória transacional descrita em [1]. Esta implementação permite que linhas da *cache* transacional sejam alocadas em uma região reservada de memória local, sem que a transação seja abortada. Esta característica permite que uma transação acesse conjuntos de dados maiores que o espaço disponível na *cache*.

Em [15] descreve-se 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 favoreceu transações que realizam o *commit* com sucesso.

Em [5] descreve-se um sistema HTM que não utiliza *caches* e protocolos de coerência de *cache* para detectar os conflitos entre as transações. Uma nova implementação, chamada *Bulk* utiliza os endereços modificados na geração de uma assinatura que em seguida é enviada através de um *broadcast* para os demais processadores no momento do *commit*.

3.2.3 Unbounded HTMs

Os chamados *Unbounded HTMs* são sistemas de memória transacional que permitem que as transações sobrevivam a mudanças de contexto, podendo continuar sua execução ainda que esta tenha sido interrompida. Para atingir este objetivo, é necessário armazenar os estados das transações em um espaço de memória persistente, como o espaço de memória virtual.

Unbounded Transactional Memory (UTM) é o sistema descrito em [1]. Este sistema é capaz de sobreviver a mudanças de contexto e exceder limites de *buffer* em hardware. O UTM propõe modificações na arquitetura, ampliando-a de forma a desacoplar os estados de manutenção das transações e detecções de conflito das *caches*. Para manter o estado transacional fora do hardware, este sistema introduz uma estrutura de dados residente em memória chamada XSTATE, que armazena informações (como conjuntos de escrita e leituras) de todas as transações no sistema. Para manter a verificação de conflitos independente da coerência de *cache*, este sistema mantém bits de acesso com informações sobre processos em cada um dos blocos de memória. Através destas informações, que poderiam ser acessadas por uma transação, seria possível detectar conflitos.

Em [19] descreve-se o sistema TM *Virtual Transactional Memory (VTM)*. Neste sistema não seria necessário que programadores implementassem detalhes a respeito do hardware, através 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 a forma como 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 transações que excederam os limites da *cache* ou ainda que excederam o tempo de escalonamento.

Em [26] uma implementação alternativa para o sistema VTM é proposta. Nesta implementação são descritos extensões em software e no conjunto de instruções para permitir que as transações VTM aguardem determinados eventos e reiniciar sua execução eficientemente através de comunicação com o escalonador, que é orientado a pausa-las temporariamente, porém compensa-las em seguida. Esta implementação adiciona novas instruções, novo suporte a interrupções de software e amplia as estruturas de metadados da implementação VTM original.

3.2.4 Hybrid/Hardware Accelerated TMs

Sistemas de memória transacional que se baseiam em STMs como um ponto de partida, porém utilizam mecanismos de HTM para manter a eficiencia são chamados Híbridos ou acelerados por hardware. Esta abordagem garante a flexibilidade, mantendo as transações desacopladas do hardware através do uso de implementações em software (STMs) aplicados a mecanismos limitados de hardware (HTMs).

O sistema *Lie* é proposto em [14] e descreve um sistema de memória transacional híbrido de hardware e software, em que as transações são executadas como transações de hardware porém, se estas transações não puderem ser executadas desta forma, elas são reiniciadas e executadas como uma transação em software. Neste sistema, duas versões do código das transações é gerado, um para transações em hardware e outro para transações em software. O código gerado para transações em hardware executa ainda algumas verificações em software, permitindo que estas transações detectem conflitos com transações que estejam executando em software.

Outro sistema de memória transacional em que as transações são inicialmente executadas como transações de hardware porém, caso não possam ser executadas como tal, são reiniciadas e executadas como uma transação de software, é descrito em [12]. Através desta abordagem, é possível manter a eficiência de uma transação em hardware sempre que possível, porém recorrer a transações de software se o hardware for insuficiente.

O sistema *Rochester Transactional Memory (RTM)*, descrito em [22], 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.

3.3 Análise de consumo de energia em Sistemas de Memória Transacional

A eficiência de uma memória transacional pode ser medida de diferentes formas. A principal delas é chamada *throughput*, e diz respeito a quantidade de transações bem-sucedidas em um intervalo definido de tempo. Outra medida importante é o *speedup*, que diz respeito ao quão mais veloz um programa executou em proporção ao seu tempo de execução anterior. Estudos mais recentes apresentam a medida do consumo de energia como um parâmetro adicional, levando em consideração se estes sistemas aumentam o diminuem o consumo médio de uma aplicação.

Em [3] a plataforma MPARM foi utilizada como simulador para análise de consumo de energia em MPSoCs (*Multiprocessor System on Chips*). Neste trabalho, modelos de consumo de energia foram incorporados a cada um dos componentes de hardware, possibilitando que a plataforma observe os valores de consumo de energia em cada um dos ciclos simulados.

Em [16], um estudo a respeito dos efeitos gerados pelo uso de memórias transacionais em relação ao consumo de energia é apresentado. O sistema proposto é semelhante ao sistema apresentado em [25], sendo capaz de identificar momentos de alta contenção e serializar a execução das transações para diminuir a quantidade de transações mal-sucedidas. Nos resultados apresentados, pode-se perceber que as memórias transacionais reduzem significativamente o consumo de energia, sendo que no sistema que utiliza serialização de transações em momentos de alta contenção este consumo foi ainda menor.

Em [2], uma abordagem a respeito de metodologias de medição do consumo de energia são expostas, demonstrando técnicas para diferenciar o consumo de energia médio da aplicação e o consumo introduzido pelo uso do sistema de memória transacional.

4. CONCLUSÃO

Como se pode observar, o uso de sistemas de memória transacional se mostra uma alternativa interessante para a programação de um software paralelo. Através desta metodologia é possível manter uma abstração de alto nível, evitando que os programadores sejam obrigados a lidar e entender detalhes intrínsecos ao hardware. Além de facilitar a programação através da introdução de uma abstração mais amigável, estes sistemas também possibilitam maior aproveitamento do recursos disponíveis, uma vez que diferentes *threads* não se bloqueiam a menos que um conflito seja realmente detectado, característica que não existe em implementações com *locks* e semáforos.

Por ser uma idéia relativamente nova, pode-se observar que existem diversas implementações que tiram proveito de características de hardware, software ou ambas. A grande quantidade de pesquisas, implementações e publicações a respeito do tema denotam ainda a importância deste novo campo, expondo o fato de que ainda não se encontrou um modelo ideal ou padrão para implementações de sistemas de memória transacional.

References

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

[2] A. Baldassin, F. Klein, P. Centoducatte, G. Araujo, and R. Azevedo. A first study on characterizing the energy consumption of software transactional memory. *Relatórios Técnicos IC*, April 2009.

[3] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. Mparm: Exploring the multi-processor soc design space with systemc. *J. VLSI Signal Process. Syst.*, 41(2):169–182, 2005.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. *IEEE Micro*, 28(1):32–41, 2008.

[5] R. P. Case and A. Padegs. Architecture of the ibm system/370. *Commun. ACM*, 21(1):73–96, 1978.

[6] A. Chang and M. Mergen. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst.*, 6(1):28–50, February 1998.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. 2006.

[8] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[9] 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. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.

[10] 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.

[11] T. F. Knight. An architecture for mostly functional languages. *ACM Lisp and Functional Programming Conference*, pages 105–112, August 1986.

[12] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220. ACM Press, 2006.

[13] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

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

[15] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *12th Int. Symp. on High-Performance Computer Architecture*, pages 254–265, February 2006.

[16] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy-aware microprocessor synchronization: Transactional memory vs. locks. In *4th Workshop on Memory Performance Issues (held in conjunction with the 12th International Symposium on High-Performance Computer Architecture)*, 2006.

[17] G. Radin. The 801 minicomputer. In *1st Int. Symp on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, 1982.

[18] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[19] 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.

[20] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 107–113, 2007.

[21] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

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

[23] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, 1993.

[24] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[25] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178, New York, NY, USA, 2008. ACM.

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