Abordagem de Memória Transacional e Identificação de Local-Threads e Read-Only Memory Data

Alexandre Queiroz
RA 098321 - Instituto de Computação
Universidade Estadual de Campinas
alexqueiroz.blessed@gmail.com.br

Wilson Vendramel
RA 075642 - Instituto de Computação
Universidade Estadual de Campinas
pwvendramel@uol.com.br

RESUMO

Nas últimas décadas, o aumento dos transistores nos microprocessadores foi exponencial, seguindo a lei de Moore. Contudo o aumento do desempenho que havia acompanhado esse crescimento, deixou de ser verdade no início deste século devido ao aumento da dissipação do calor gerado pelo processamento, impedindo que sua velocidade pudesse ser aumentada. A tendência foi a criação dos processadores *multi-core* que passaram a integrar mais de um núcleo no mesmo *chip*. Para que haja um sincronismo entre os dados surgiu a técnica de Memória Transacional, mantendo assim a consistência da memória, podendo ser implementada tanto em hardware como software. Uma nova abordagem é a identificação dos dados de memória local ou apenas de leitura (constantes) para diminuir a geração de bloqueios, aumentando assim a eficiência do sistema. Este trabalho apresenta os fundamentos sobre Memória Transacional, *Benchmarks* para avaliação e cita uma abordagem da implementação desta técnica na identificação de *Local-Threads e Read-Only Memory Data*.

Palavras-Chave

Thread; Memória Transacional; Paralelismo; Memória Transacional Local; Benchmark.

1. INTRODUÇÃO

Até o final do século passado, o desempenho alcançado pelos microprocessadores crescia exponencialmente. Esse crescimento deu-se devido ao avanço da tecnologia que possibilitou a construção de transistores menores e mais rápidos. Esse panorama foi alterado no início desse século, com o aumento da energia dissipada.

Embora seja verdade que, segundo a lei de Moore [19], o número de transistores encapsulados continua crescendo, o desempenho do mesmo não acompanhou o mesmo ritmo. Isto deu-se pelo fator do aquecimento dos processadores, chegando a um limite que denominado barreira térmica de de 3.8GHz. A figura 1 mostra um crescimento na velocidade dos processadores [21] [22].

A tendência atual é desenvolver projetos mais simples, com frequências de operação menores e integração em um mesmo chip de dois ou mais núcleos de processamento, ou seja, o emprego de processadores *multi-core*, também conhecidos como como *single-chip multiprocessors*. Para manter a consistência dos dados compartilhados pelos processos executados por essa nova tecnologia, vem surgindo alternativas para controles de memória como a Memória Transacional.

Atualmente, a estratégia de implementação de Memória Transacional em Software é alvo de muita pesquisa devido à flexibilidade no desenvolvimento de novos algoritmos e por ser executada diretamente nos processadores atuais, enquanto a Memória Transacional em Hardware deve ser validada em simuladores.

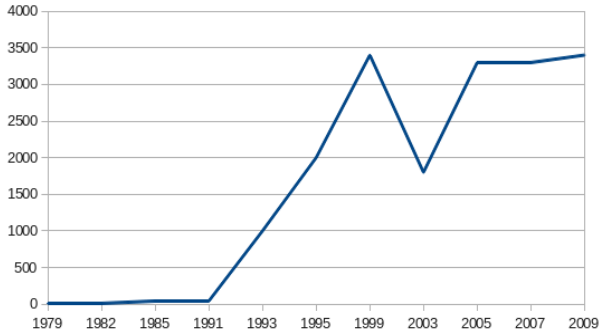

Figura 1: Velocidade dos processadores Intel [21][22].

Este trabalho apresenta os conceitos principais sobre Memória Transacional e os possíveis *Benchmarks* para sua avaliação. Além dos conceitos, este trabalho também cita uma proposta com o propósito de identificar transações que utilizam dados locais e apenas de leitura, não sendo necessário as implementações em software de bloqueios, mostrando assim a eficiência desta abordagem [6].

As outras seções do artigo apresentam os conceitos básicos sobre Paralelismo (Seção 2), Memórias Transacionais (Seção 3), e Métodos de Avaliação (Seção 4). A Seção 5 mostra o trabalho sobre Memória Transacional Local e sua respectiva Avaliação [6], e por fim, a conclusão do trabalho é descrita na Seção 6.

2. PARALELISMO

O termo paralelismo em computação é utilizado para indicar que mais de uma instrução será executada ao mesmo tempo por uma CPU. Existem diversas técnicas de paralelismo criadas para aumentar a performance dos computadores, aproveitando-se os tempos ociosos que podem ocorrer em determinadas instruções para executar outras neste mesmo intervalo de tempo. Segundo [10], existem paralelismos em nível de bits, instrução e tarefas onde cada uma tenta especular a melhor forma de execução de instruções em paralelo.

2.1 Processos e threads

Processo é uma instância ou parte de um programa que está em execução. Um sistema operacional multi-tarefa pode executar mais de um processo ao mesmo tempo compartilhando o processador por meio de interrupções e trocas de contexto. Já *multi-core* podem executar processos em núcleos diferentes concorrentemente, sem a necessidade de compartilharem os mesmos recursos [10].

Threads são partes de processos designadas a execução de uma determinada tarefa. Elas normalmente são criadas pelos processos e compartilham os recursos do mesmo para serem executadas concorrentemente. A figura 2 mostra os processos ou *threads* sendo executados em CPUs S*ingle-core e* Dual-core [10].

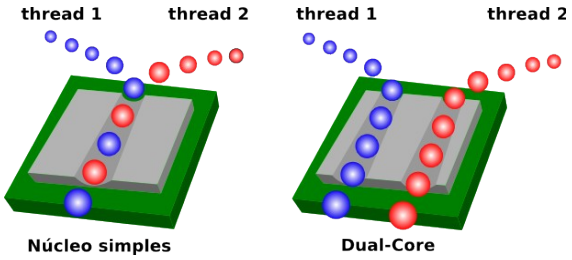

Figura 2: Processador Single-core e um Dual-core [21].

3. MEMÓRIA TRANSACIONAL

O conceito de transação surgiu no contexto de banco de dados [9]. O ápice de uma transação é agrupar diversas transações em um único conjunto de operações de tudo ou nada. Durante o processo, o estado de cada instrução é armazenado e não é compartilhado pelas demais. Se por alguma razão, uma transação não for concluída, as outras instruções não poderão alterar o banco de dados, conforme mostrado na figura 3 [7][9].

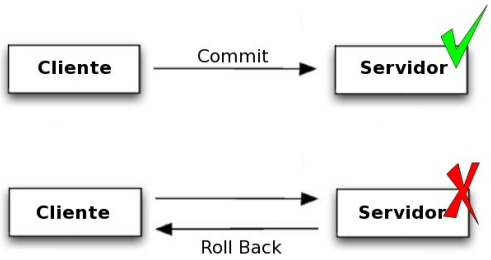

Figura 3: Operação de tudo ou nada em Banco de Dados [7][9].

De acordo com Gray and Reuter [9], uma transação é um conjunto de instruções executadas de forma sequencial por uma *thread*, e que satisfaça as seguintes propriedades (também conhecidas como ACID):

- *Atomicidade*: ou todas as instruções são executadas e a transação é efetivada (*commit*), transformando o resultado da computação como permanente, ou nenhuma instrução é executada e a transação é cancelada (*abort*), invalidando qualquer valor parcialmente computado. A efetivação representa o tudo, e o cancelamento denota o nada.
- Consistência: uma transação transforma o estado consistente do sistema (pré-transação) a outro estado consistente (pós-transação).
- *Isolamento*: as transações concorrentes não visualizam os resultados intermediários das outras transações. O resultado da execução das várias transações corresponde ao resultado da execução dessas mesmas transações em alguma ordem serial.
- *Durabilidade*: as mudanças realizadas pela transação permanecem e resistem a uma possível falha no sistema.

Para Herlihy e Moss [14], o conceito de transação adotado em Memória Transacional é quase o mesmo utilizado no contexto de banco de dados, exceto pela durabilidade já que as transações operam em memória volátil, não fazendo muito sentido satisfazer esta propriedade. Um adendo importante sobre a terminologia atômico é que a comunidade de Memória Transacional adota de forma semelhante à comunidade de linguagens de programação. A palavra atômico tem o sentido de isolamento para definir a não interferência entre duas ou mais transações.

O termo Memória Transacional (*Transactional Memory*, ou simplesmente TM) foi definido por Herlihy e Moss [14] como uma "nova arquitetura para multiprocessadores que visa tornar a sincronização livre de bloqueios de forma eficiente e fácil de usar quanto técnicas convencionais de exclusão mútua". Atualmente, TM é utilizada para designar os mecanismos de sincronização que utilize o conceito de transação para gerenciar acessos à memória compartilhada. O uso de transações em memória oferece as seguintes vantagens:

- Facilidade de codificação: o desenvolvedor não precisa garantir a sincronização. Ao invés disso, ele precisa especificar o que deve ser executado de forma atômica. A abordagem transacional é mais declarativa, pois define o código a ser executado atomicamente e o sistema de execução tem o papel garantir a sincronização.
- Escalabilidade: o sistema de execução possibilita que um mesmo método seja executado de modo concorrente além de verificar possíveis conflitos entre os acessos, garantindo paralelismo total nos cenários em que os acesos lidam com dados disjuntos.
- Composição: as transações suportam a composição de código. Uma nova operação com base em outras já existentes, basta englobá-las em uma nova transação (característica conhecida como aninhamento). O sistema de execução permite que as operações sejam executadas atomicamente.

Um sistema de TM pode ser implementado tanto em hardware quanto em software. A Memória Transacional em Hardware (*Hardware Transactional Memory*, ou HTM) desenvolve o suporte arquitetural para execução de transações. A Memória Transacional em Software (*Software Transactional Memory*, ou STM) desenvolve o sistema de execução transacional em software. Além dessas duas estratégias de implementação, existe uma terceira denominada Memória Transacional Híbrida que apesar de ter basicamente sua implementação em software, esta abordagem é apoiada por mecanismos de hardware que ajudam a melhorar o seu desempenho. A figura 4 mostra um exemplo de

processos compartilhando dados e sendo executados num sistema TM [17].

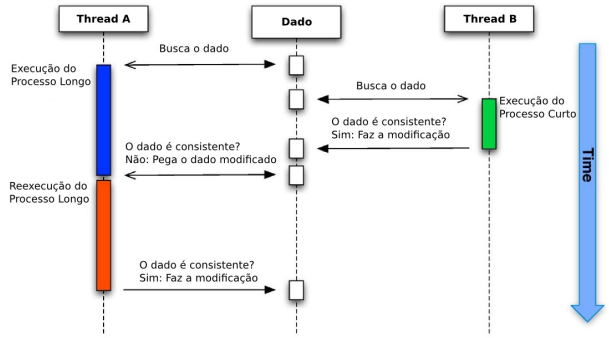

Figura 4: Threads em memória transacional [17].

As subseções seguintes apresentam maiores detalhes sobre as estratégias HTM e STM.

3.1 Memória Transacional em Hardware

O primeiro sistema de TM implementado em hardware foi apresentado por Kinight [16] em 1986. A estratégia HTM geralmente faz o versionamento e controle de conflito por meio de modificações na cache e no protocolo de coerência. Os aspectos principais da HTM são o isolamento forte e a granularidade de linha de cache.

Entre as principais implementações em hardware, pode-se elencar: TCC (*Transactional Coherence and Consistency*) [13], UTM (Unbounded TM) [1], VTM (Virtual TM) [24], LogTM (Log-based TM) [20], PTM (Page-based TM) [5]. Todas essas propostas suportam virtualização de espaço e tempo, sendo que as diferenças entre elas estão no hardware extra exigido e na estratégia utilizada para versionamento e controle de conflitos.

3.2 Memória Transacional em Software

A estratégia STM foi proposto por Shavit e Touitou [25] em 1995 como uma alternativa de solução implementada totalmente em software, e que está voltada ao mecanismo de hardware de Herlihy e Moss [14] de 1993. Entre as vantagens em utilizar o STM, pode-se citar que há uma maior flexibilidade de implementação e exploração de semântica transacional; e que permite a execução direta em processadores atuais. Os aspectos principais da STM são:

- Forma de implementação: utiliza propriedade não-bloqueante ou bloqueante;
- Granularidade de acesso: determina qual a unidade base utilizada para versionamento de dados e detecção de conflitos (classificada como palavra ou objeto);
- Nível de integração: aponta se a STM é disponibilizada como uma biblioteca , ou se está integrada a um sistema de execução e/ou compilador.

Entre as implementações STM mais conhecidas são, pode-se mencionar: DSTM (Dynamic STM), WSTM (Word-based STM), ASTM (Adaptive STM), RSTM (Rochester STM). Todas estas possuem a forma de implementação não-bloqueante. Além disso, todos trabalham com a granularidade de objeto, exceto o WSTM que utiliza granularidade de palavra.

3.3 Versionamento de dados

O versionamento procura gerenciar diferentes versões dos dados acessados pelas transações. Ele possui duas versões para cada dado, sendo a versão original e a especulativa (alterada). Se a transação for bem sucedida, os dados especulados tornam-se os valores válidos. Caso ocorra uma falha, os dados especulados são descartados e os valores originais são mantidos. Existem dois tipos de versionamento: direto e diferido [3].

- direto: os dados especulados são mantidos na memória enquanto os originais são mantidos em um único *log*;
- diferido: os dados originais são mantidos na memória e os especulados são mantidos em um *buffer*.

A figura 5 apresenta um exemplo de versionamento direto e diferido, onde a váriável X é alterada por um processo [3].

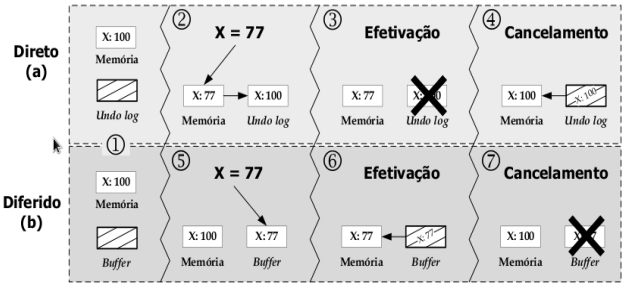

Figura 5: Versionamento direto e diferido [3]

3.4 Detecção de conflitos

As transações geralmente mantém um *set* de leituras com endereços do elementos lidos e um *set* de escrita dos que foram alterados. Se houver uma intersecção entre eles, significa que houve um conflito e as transações acessaram o mesmo elemento. Da mesma forma que o versionamento, a detecção de conflitos possui duas versões: pessimista e otimista [3].

- pessimista (*pessimistic* ou *eager conflict detection*) onde é detectado assim que acontece;
- otimista (*optimistic* ou *lazy conflict detection*) onde o conflito é detectado apenas no final da transação.

A abordagem pessimista pode economizar tempo evitando que a transação seja executada e depois descartada. A otimista porém, faz com que o processo termine na "esperança" de que não seja necessário descartar o processo e reexecutá-lo novamente. A figura 6 exibe um esquema onde dois processos t_1 e t_2 desejam utilizar a mesma posição de memória denotada como X [3].

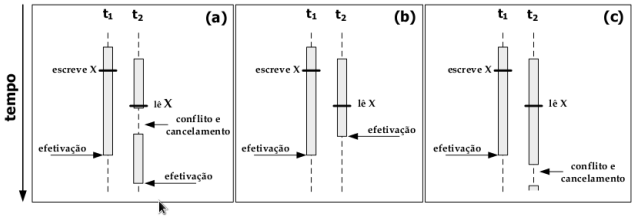

Figura 6: Cenários para detecção de conflitos otimista e pessimista [3].

3.4.1 Sistema dinâmico de detecção de conflitos

Geralmente as implementações de TM envolvem apenas uma dessas detecções para resolver a especulação. Uma técnica desenvolvida por Lupon *et al*. [18], procura explorar as duas abordagens para melhorar o desempenho especulativo de detecção de conflitos denominada DynTM que trabalha de forma dinâmica. As operações de DynTM consistem na troca de mensagens entre os núcleos do processador, marcando vetores se houver algum conflito. Cada núcleo envia uma mensagem de *AbortTx* quando detecta algum conflito e aguarda as mensagens de *AbortAck* de todos os núcleos que usam o mesmo endereçamento de memória, utilizando um *timestamp* a fim de verificar qual a transação que deve ser descartada. Cada núcleo deve ter uma simples TMS (*Transaction Mode Selector*) para definir o melhor método de execução para cada transação, isto é, deve decidir entre usar o método otimista ou pessimista, não sendo determinística a sua utilização.

Além disso, esse método é capaz de alternar sua execução, trocando de otimista para pessimista, ou vice-versa, se perceber que um conflito poderá ocorrer se continuar com o mesmo método não descartando assim a transação corrente.

Lupon *et al*. [18] verificaram uma eficiência em ganho de velocidade em 34% frente aos tradicionais métodos que exploram apenas um tipo de abordagem.

4. AVALIAÇÃO DE TM

Como método de avaliação de memória transacional, a exemplo dos processadores, surgiram alguns *benchmarks* para medir o desempenho dessas implementações. A maioria são citados por Kestor *et al*. [15] em sua proposta (RMS-TM: *A Transactional Memory Benchmark for Reconigntion, Mining and Synthesis Application*) onde foi criado o RMS-TM que traz diversos tipos de aplicacoes para as diversas situações de TM como *abort/commit* e operações de I/O em transações aninhadas:

- TM micro-benchmarks [8] trabalha com uma única estrutura de dados como tabelas *hash*, listas ligadas e arvores binárias;
- SPLASH-2 [26] é uma coleção de aplicativos composta por oito aplicações completas e quatro de *kernels*. As aplicações de *kernel* são utilizadas para analisar o tempo das seções críticas. Esse *benchmark* foi criado para aplicações que utilizam pouca sincronização de mensagens;
- STM- Bench7 [10] apresenta uma proposta baseada no *benchmark* 007 que realiza operações complexas em estruturas de dados não-triviais avaliando as transações de virtualização. Essa proposta apresenta referências de diversos *benchmarks* com diversos tipos de comportamentos;
- LeeTM [2] é um conjunto de *benchmarks* baseados em algoritmos Lee's. Compara diversos tipos de *lock* e implementações transacionais;
- STM Haskell [23] é composto por 10 aplicações que possuem códigos de tamanhos diferentes e são implementados na linguagem de programação Haskell, o que permite simular diversas situações, mas não é adequado para sistemas híbridos;
- STAMP [4] é um conjunto de referências de diversos *benchmarks* que possuem vários tipos de execução e comportamento; e
- WormBench [27] que traz um conjunto de aplicações de síntese transacional desenvolvidas em C, altamente configurável para criar cenários e comportamentos diferentes de TM.

Dentre os *benchmarks* citados, o mais referenciado e de maior relevância é o STAMP, apresentando várias partes de programas críticos para medição dos comportamentos e velocidade das propostas em memória transacional.

5. MEMÓRIA TRANSACIONAL LOCAL

Um sistema STM depende do compilador para implementação dos seus elementos bloqueantes e não bloqueantes, e identificar essas regiões não é uma tarefa muito trivial. A proposta de Dragojevic e Adl-Tabatabai [6] intitulada *Optimizing Transactions for Captured Memory* busca desenvolver uma melhor alternativa para identificar as transações locais de um sistema de TM.

Conforme visto na seção 2, uma transação tem que garantir isolamento, ou seja, precisa ser independente, não permitindo que as outras possam "ver" o que está ocorrendo. Desta forma, a memória, bem como os recursos de memória alocados dentro de uma transação não podem ser acessados por outras transações. Estes blocos são chamados de Memória Transacional Local.

Já que estes recursos de memória não podem ser acessados por outras transações, é certo que não serão elementos bloqueantes que podem causar *deadlocks*, não sendo necessário criar barreiras para os mesmos.

5.1 *Thread* Local de Dados

Normalmente, os ponteiros de memória em um STM são criados em modo compartilhado. Porém, se uma transação t_1 está sendo executada, as alocações locais não poderão ser acessadas até que ela termine sua execução. Logo o conceito de isolamento de TM garante que esses dados não serão acessados por qualquer instância t_i. Obviamente a barreira para esses elementos torna-se desnecessária.

A tarefa de identificar esses pontos de isolamentos não é muito fácil quando se trabalha com ponteiros. Por exemplo, quando um seguimento de memória pode ser acessado por uma função externa passado como argumento do tipo ponteiro. Além disso, uma *thread* local pode deixar de ser local neste caso. O compilador deve fazer uma análise de todo o código para tomar a decisão de colocar ou não barreiras para esse código. Na figura 7 é apresentado um trecho de código (do *benchmark* STAMP *Bayes*) que acessa uma *thread* local de dados [6]. O primeiro código aloca um vetor local *queryVectorPtr* e, posteriormente, o utiliza para passar argumentos para funções *TmpopulateQueryVectors* e *computeLocalLogLikelihood*. O mesmo vetor é usado em transações diferentes, não sendo acessado fora da *thread* de alocação.

```
list_iter_t it;
TMLIST_ITER_RESET(&it, taskListPtr);

if(TMLIST_ITER_HASNEXT(&it, taskListPtr)) {
  taskPtr = (learner_task_t*)TMLIST_ITER_NEXT(
  &it, taskListPtr);
  bool_t status = TMLIST_REMOVE(taskListPtr,
  (void*)taskPtr);
}
```

(a) Transaction-Local Stack

```
vector_t* queryVectorPtr = PVECTOR_ALLOC(1);

...

TMpopulateQueryVectors(..., queryVectorPtr, ...);
newBaseLogLikelihood = computeLocalLogLikelihood(
                          ..., queryVectorPtr, ...);
```

(b) Thread-Local Data

```
nodePtr = (list_node_t*)TM_SHARED_READ_P(
prevPtr->nextPtr);

if((listPtr->compare(nodePtr->dataPtr, dataPtr) != 0)
|| (nodePtr == NULL)) {
  return NULL;
}

return (nodePtr->dataPtr);
```

(c) Read-Only Data

Figura 7: Pedaço de código do STAMP Bayes [6]

5.2 Read Only Memory

Acessos à memória só de leitura não exigem elementos bloqueantes para leitura, mesmo se a memória é compartilhada entre *threads*. Assim como acontece com uma *thread* local de dados, uma região de memória pode alternar dinamicamente entre ser somente leitura, e de leitura e escrita, como por exemplo, alguns dados podem ser atualizados no início de um programa durante a inicialização, mas nunca mudou depois.

Declarando variáveis constantes em C/C++ com *const* não resolve totalmente o problema por razões semelhantes às *thread* local de dados. Além disso, o qualificador *const* pode ser simplesmente descartado quando os dados são acessados.

Se não fosse por esses casos, um qualificador *const* poderia ser usado para variáveis globais para informar ao compilador STM que a variável é somente leitura e que as barreiras não são necessárias quando acessá-lo. No entanto, é difícil para o compilador dizer se realmente um ponteiro aponta para alguma memória só de leitura. Por exemplo, se o endereço de uma variável global *const* é passado para uma função como um argumento de ponteiro, é impossível para o compilador compilar essa função para saber que este ponteiro aponta para memória só de leitura, a menos que uma análise de todo o programa muito complexa seja realizada. Em C ++, definindo-se um argumento *p* para a função *foo* como *const <tipo> * p* apenas garante que a função *foo* não mudará a memória apontada por *p*, mas não que *p* aponta para dados imutáveis. Não há outra maneira de expressar isso em C++.

5.3 Análise de captura para Thread Local de Dados e Read Only Memory

É praticamente impossível decidir se certos dados são *thread* local ou somente de leitura em tempo de execução sem a ajuda do programador. Ao invés de tentar detectar automaticamente quais locais são *thread* local ou só de leitura, criou-se uma API onde o programador pode anotar as regiões de memória seguras ou não, sem barreiras STM. Quando um bloco de memória torna-se privado por meio de uma chamada para *addPrivateMemoryBlock*, ele é inserido no *log* de controle local de segmento de dados, que utiliza a mesma estrutura de dados e algoritmos em geral, como o *log* de alocação.

Quando um bloco de memória compartilhada torna-se novamente disponível, através de uma chamada de *removePrivateMemoryBlock*, ele é removido então do *log*.

A principal diferença entre esses dois registros é que o *log* dos blocos *read only* esvazia suas alocações a cada final da transação (*commit ou abort*), enquanto o *log* da *thread* local não é.

Enquanto as otimizações em tempo de execução podem reduzir o custo médio de barreiras, evitando barreiras completas, o tempo de execução pode introduzir custos adicionais. Em alguns casos, estes custos adicionais poderiam até mesmo levar à degradação de desempenho se o custo da análise de captura em tempo de execução supera o potencial de parcimônia de barreiras. Uma alternativa para realizar análise de captura em tempo de execução é fazê-lo em tempo de compilação, e reduzir algumas das barreiras desnecessárias em STM, sem custos adicionais em tempo de execução.

A análise de captura foi criada usando a análise de ponteiro, que determina se um ponteiro aponta para a memória alocada dentro da transação atual. Se acontecer, a referência do ponteiro não exige barreiras de STM. Da mesma forma que as técnicas de execução, a análise de captura do compilador pode tolerar falsos negativos e, portanto, podem usar análise de ponteiro que não é totalmente precisa, contanto que ele seja conservador.

A análise de captura de compilador foi baseada na análise do ponteiro padrão implementado no compilador Intel C++. Esse ponteiro apenas usa a análise intraprocedural, e se baseia na função *inlining* para estender os resultados da análise por meio de chamadas de função. Optou-se por não usar a análise inter-processual, uma vez que esta aumentaria o tempo de compilação e não seria plenamente aplicável quando as bibliotecas ligadas dinamicamente estão sendo utilizadas.

5.4 Avaliação de TM LOCAL

Para avaliação do tempo de execução deste método foi utilizado o *benchmark* STAMP 0.9.9, e os testes foram realizados em uma máquina *Dunnington* com quatro processadores Intel com seis núcleos cada um de 2,66 GHz e 16 GB de RAM com sistema operacional *Red Ha*t Linux versão 7. No entanto, o

STAMP só permite execuções de *threads* em potência de dois, o que permitiu utilizar apenas 16 *threads* no teste corrente.

Para estimar o número de barreiras necessárias, contou-se com um instrumentado manual de acessos nos programas de *benchmark* original STAMP. Embora este procedimento não forneça o número absoluto de barreiras mínimas necessárias para corrigir o código, ainda assim é uma boa base para estimativa do número de barreiras que o compilador STM acrescentou. Utilizou-se um algoritmo próprio baseado em árvore em tempo de execução para contar o número total de transações locais de acessos *heap* e *stack*. Esse algoritmo de tempo de execução baseado em árvore torna-se necessário pois detecta todos os acessos a transação locais de *heap* e pilha. O restante das barreiras STM geradas pelo compilador não são necessários pois não são transações locais. Essas barreiras podem ser acessos *read-only* ou segmentos locais de dados. Essas barreiras não podem ser reduzidas sem a ajuda do programador.

Notou-se que houve uma redução de barreiras introduzidas pelo compilador STM em todos os programas STAMP para leitura e gravação de acessos à memória. Os números mostraram oportunidades significativas de melhoria no desempenho com a maioria dos programas STAMP. Porém, o único que não apresentou barreiras redundantes foi o programa Labyrinth. Além disso, o número de barreiras que podem ser eliminadas utilizando esta técnica é muito maior para escrita que para leitura, o que sugere que o ganho pode ser melhorado, pois o custo de escrita é bem maior que o custo de leitura. Por fim, o número de leituras é muito maior que o de escrita para todos os programas STAMP, com exceção do Yada, mostrando que para este ainda pode ser reduzida 60% das barreiras existentes.

A Figura 8 mostra o número de barreiras que as diferentes técnicas de análise de captura podem remover [6]. A figura transmite dois pontos interessantes: (1) o único programa para o qual, em tempo de execução, remove um número menor de barreiras é o Yada, o que sugere que quase todo o potencial de análise de captura pode ser alcançado por meio do rastreamento de apenas algumas alocações de memória, (2) apesar de usar um algoritmo simples, a análise do compilador é bastante eficaz para a identificação e remoção de barreiras desnecessárias, onde, notou-se que apenas para o Yada foram removidas menos barreiras com a execução das técnicas de filtragem de *hashtable*.

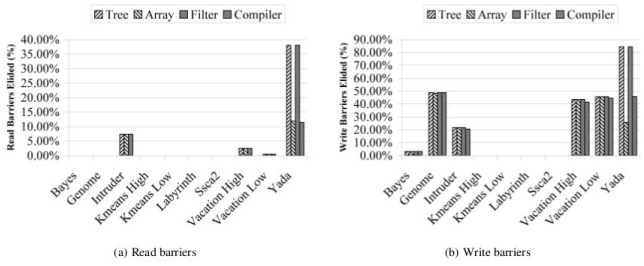

Figura 8: Barreiras removidas com diferentes técnicas [6].

5.5 Impactos no desempenho

A tabela 1 mostra a taxa de *abort-to-commit* para todos os programas do *benchmark* em execução em 16 *threads*, para todas as otimizações [6]. Esta relação é definida como a razão do número de transações abortadas, e repetido com o número de transações confirmadas. A tabela 1 ainda revela que as otimizações reduziram significativamente o número de *aborts* para todas as configurações de contenção do *benchmark* que resulta em uma melhoria de desempenho.

Tabela 1: Taxa de *abort-to-commit* em 16 threads [6].

	Baseline	Tree	Array	Filtering	Compiler
Bayes	95	0.86	1.2	0.94	0.67
Genome	0.05	0	0	0	0
Intruder	0.78	0.79	0.78	0.78	0.78
Kmeans High	1.6	1.6	1.6	1.6	1.6
Kmeans Low	0.66	0.67	0.68	0.65	0.67
Labyrinth	0.18	0.17	0.17	0.17	0.17
Ssca2	0	0	0	0	0
Vacation High	0.28	0.01	0.01	0.01	0.02
Vacation Low	0.24	0	0	0.01	0.01
Yada	1.7	1.6	1.7	1.6	1.6

Para avaliação da melhoria de desempenho, foi realizada a medição do tempo de execução de todos os programas do *benchmark* em um único segmento, e comparados os resultados de todas as otimizações de tempo de execução. Nos experimentos para otimização de tempo de execução, foi utilizada a estrutura de dados árvore para *log* de alocação e realizados em três diferentes configurações: (1) a verificação das transações locais em pilha e *heap* com barreiras de acessos de leitura e gravação, (2) a verificação transação-local de pilha e *heap* de acesso somente para escrita, e (3) a verificação de *heap* em transação local com barreiras apenas de escrita. Os resultados apresentados na figura 9 mostram que as otimizações de tempo de execução não degradaram o desempenho do *single-thread*, exceto para Kmeans e Yada [6]. Para Kmeans, isso se deveu à falta de oportunidades para diminuição de barreiras. Neste caso, a sobrecarga foi signifativa.

Para o programa Yada, a maioria das barreiras redundantes já estavam capturadas. Por esta razão, as otimizações em tempo de execução não melhoraram o desempenho, mas acrescentaram custos, degradando o desempenho.

Um ponto a ser notado é que quando desativadas as otimizações de tempo de execução para leitura e acessos em transação local da pilha, a sobrecarga do tempo de execução foi reduzida significativamente.

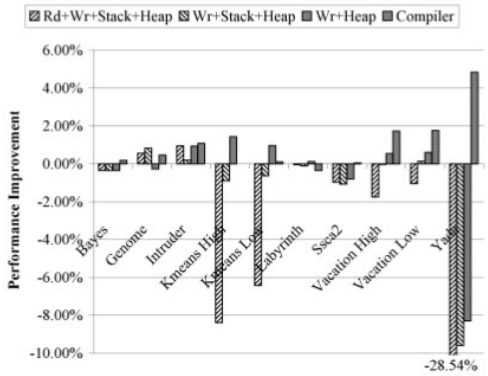

Figura 9: Otimizações em tempo de execução [6]

A otimização do compilador, por outro lado, não causou qualquer degradação de desempenho, como esperado, porque não introduziu qualquer custo de tempo de execução.

Os experimentos foram repetidos e mostraram grandes flutuações de desempenho nos programas do *benchmark*, que foram causadas por contenção. O desempenho melhorou em 14% para a configuração de contenção de alta e 18% para a configuração de contenção de baixa. As razões para estas melhorias são: o número relativamente elevado de diminuição de barreiras e a redução dos falsos conflitos. O experimento também mostrou que as otimizações do compilador, embora muito simples, tiveram um desempenho próximo ou melhor do que as otimizações de tempo de execução em quase todos os casos. No entanto, alguns programas do *benchmark* foram indiferentes à otimizações, dada a variedade vistas nas diversas medições.

6. CONCLUSÃO

A utilização da Memória Transacional pode melhorar o desempenho dos aplicativos que exploram paralelismo, garantindo que os dados compartilhados da memória sejam utilizados de forma correta. Embora ainda seja experimental, novas implementações, principalmente de STM vem surgindo com o advento dos computadores pessoais *multi-core* aumentando o nível de abstração de paralelismo. Uma proposta interessante citada neste trabalho demonstrou um grande avanço em TM que identifica os blocos de dados locais em uma transação e valores constantes, fazendo com que haja uma diminuição do gargalo, evitando possíveis barreiras (bloqueios) desnecessários para acessos à esses endereçamentos de memória.

7. REFERÊNCIAS

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

[2] Ansari, M., Kotselidis, C., Watson, I., Kirkham, C., Luj´ n, M., and Jarvis, K., 2008. Lee-tm: A non-trivial benchmark suite for transactional memory. In *ICA3PP '08: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, pages 196–207, Berlin, Heidelberg. Springer-Verlag.

[3] Baldassin, A. J. D. 2009. Explorando Memória Transacional em Software nos Contextos de Arquiteturas Assimétricas, Jogos Computacionais e Consumo de Energia, Doctoral Thesis. Cod 000768401. Universidade Estadual de Campinas . Instituto de Computação., IC/UNICAMP.

[4] Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K., 2008. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*.

[5] Chuang, W., Narayanasamy, S., Venkatesh, G., Sampson, J., Biesbrouck, M. V., Pokam, G., Calder, B., and Colavin, O., 2006. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358.

[6] Dragojevic, A., Ni Y., Adl-Tabatabai, A., 2009. Optimizing transactions for captured memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (SPAA '09). ACM, New York, NY, USA, 214-222. DOI=10.1145/1583991.1584049 http://doi.acm.org/10.1145/1583991.1584049

[7] Elmasri R.; Navathe, S; Sistemas de Bancos de Dados - Fundamentos e Aplicações, 3 edição, LTC, 2002

[8] Ennals, R. J., 2004. Adaptive evaluation of non-strict programs. Technical report.

[9] Eswaran K. P., Gray J. N., Lorie R. A., and Traiger I. L., 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (November 1976), 624-633. DOI=10.1145/360363.360369 http://doi.acm.org/10.1145/360363.360369

[10] Goldstein, F. P. 2010., Um modelo de memória transacional para arquiteturas heterogêneas baseado em software Cache, Master Dissertation. Cod 000779116. Universidade Estadual de Campinas . Instituto de Computação., IC/UNICAMP.

[11] Gray, J. and Reuter, A. 1993. Transaction Processing: Concepts and Techniques. *Morgan Kaufmann Publishers*, Inc. 1993.

[12] Guerraoui, R., Kapałka, M., and Vitek, J., 2007. STMBench7: A benchmark for software transactional memory. In *Proceedings of the Second European Systems Conference EuroSys 2007*, pages 315–324. ACM.

[13] Hammond, L., Willey, M., and Olukotun, K. (1998). Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating System*s, pages 58–69.

[14] Herlihy, M. and Moss, J. E. B. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International* Symposium on Computer Architecture, pages 289–300.

[15] Kestor, G., Stipic, S., Unsal, O. S., Cristal, A., and Valero, M., 2009. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT '09: 4th Workshop on Transactional Computing*.

[16] Knight, T. 1986. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 105–112.

[17] Kraus, J. M., Kestler, H. A. 2010., A highly efficient multi-core algorithm for clustering extremely large datasets. In

BMC Bioinformatics 2010, **11:**169 doi:10.1186/1471-2105-11-169 = http://www.biomedcentral.com/1471-2105/11/169

[18] Lupon, M., Magklis, G., and Gonzalez, A., 2010. A Dynamically Adaptable Hardware Transactional Memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 27-38. DOI=10.1109/MICRO.2010.23 http://dx.doi.org/10.1109/MICRO.2010.23

[19] Moore, G. E. 1975., Progress in digital integrated electronics. In *International Electron Devices Meeting*, pp. 11–13, 1975.

[20] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A., 2006. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265.

[21] Morimoto, C. E., 2002. Hardware Manuel Completo. In *Guia do Hardware*, 2002. Last access 2011/06/19 - 14h30- http://www.hardware.com.br/livros/hardware-manual/evolucao-dos-processadores.html.

[22] Olukotun, K., Hammond, L. 2005., The future of microprocessors. *Queue*, 3(7):26–34, September 2005.

[23] Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T., and Valero, M., 2008. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 67–78, New York, NY, USA. ACM.

[24] Rajwar, R., Herlihy, M., and Lai, K., 2005. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505.

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

[26] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A., 1995a. The splash-2 programs: Characterization and methodological considerations. In *INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, pages 24–36. ACM.

[27] Zyulkyarov, F., Cristal, A., Cvijic, S., Ayguadé , E., Valero, M., Unsal, O. S., and Harris, T., 2008. WormBench: a configurable workload for evaluating transactional memory systems. In *MEDEA '08: Proc. 9th workshop on MEmory performance*, pages 61–68.