Memória Transacional

MO401A - Arquitetura de Computadores I

Carla Négri Lintzmayer
Eduardo Theodoro Bogue
Maycon Sambinelli

Instituto de Computação da UNICAMP
Mestrado em Ciência da Computação

11 de junho de 2012

Introdução

- A cada lançamento de uma nova família de processadores, as aplicações sequenciais podiam sentir um aumento de desempenho proporcionado pelo aumento de velocidade e melhorias arquiteturais [9].
- No lado tecnológico, problemas como superaquecimento e dissipação de potência inviabilizaram o aumento da frequência dos processadores.
- No lado arquitetural, o ganho através do paralelismo em *nível de instrução* (ILP) atingiu seu limite.

Solução

- A solução da indústria para resolver esses problemas foi criar os chamados "chips *multicore*", "chips multiprocessadores" ou "*many cores*" [8].
- Essa nova geração de processadores opta por explorar o paralelismo em nível de *thread* (TLP, de *thread-level parallelism*)

Thread Level Parallelism

- *Thread* é a unidade de paralelismo
- Comunicação através de memória compartilhada
- Sincronização realizada geralmente através de um mecanismo de exclusão mútua
  - *Locks*
  - Semáforos
  - Monitores
  - Barreiras

Problemas com a Exclusão Mútua

1. Serialização;
2. *Deadlock*;
3. Inversão de Prioridade;
4. *Convoying*.

Consequências

- Devido a esses problemas, poucos programas paralelos eficientes foram construídos [6].
- Algoritmos paralelos são mais difíceis de se formular e de provar sua corretude do que os algoritmos sequenciais [6].

A raiz de todo o mal

- Falta de abstração para programação paralela.

Banco de Dados

- Sistemas de bancos de dados têm alcançado sucesso explorando o paralelismo;
- Executam diversas *queries* simultaneamente sem a necessidade do desenvolvedor se preocupar com o paralelismo;
- Sucesso é devido às Transações.

Transações

- Bancos de Dados – 1976;

Transações

- Bancos de Dados – 1976;
- Uma transação é “uma sequência de ações que parecem indivisíveis e instantâneas para um observador externo”;
- Ela efetiva ou aborta;

Transações

- Bancos de Dados – 1976;
- Uma transação é “uma sequência de ações que parecem indivisíveis e instantâneas para um observador externo”;
- Ela efetiva ou aborta;
- Permitem serializabilidade;

Transações

- Bancos de Dados – 1976;
- Uma transação é “uma sequência de ações que parecem indivisíveis e instantâneas para um observador externo”;
- Ela efetiva ou aborta;
- Permitem serializabilidade;
- Propriedades que devem ser garantidas: Atomicidade, Consistência, Isolamento, Durabilidade.

Transações em Memória

- Podemos usar transações na programação de propósito geral?

Transações em Memória

- Podemos usar transações na programação de propósito geral?
- Lomet [1977] – sim;
- Herlihy e Moss [1993] – memória transacional (TM, de *transactional memory*);

Transações em Memória

- Podemos usar transações na programação de propósito geral?
- Lomet [1977] – sim;
- Herlihy e Moss [1993] – memória transacional (TM, de *transactional memory*);
- Propriedades visadas: ACI.

Transações em Memória - Exemplo

Agência Bancária: `conta`

- `Deposito(i, valor)`
- `Retirada(i, valor)`
- `Transfere(i_orig, i_dest, valor)`

Transações em Memória - Exemplo

```
 1 Deposito(i, valor) {
 2     lock(contas[i]);
 3     contas[i] = contas[i] + valor;
 4     unlock(contas[i]);
 5 }
 6
 7 Retirada(i, valor) {
 8     lock(contas[i]);
 9     contas[i] = contas[i] - valor;
10     unlock(contas[i]);
11 }
12
13 Transfere(i_orig, i_dest, valor) {
14     lock(contas[i_orig]);
15     lock(contas[i_dest]);
16     Retirada(i_orig, valor);
17     Deposito(i_dest, valor);
18     unlock(contas[i_dest]);
19     unlock(contas[i_orig]);
20 }
```

Transações em Memória - Exemplo

Interface de operações transacionais que podem ser suportadas tanto por HTM quanto por STM [6]:

- Operações para gerenciamento de transações:
  - `void StartTx();`
  - `bool CommitTx();`
  - `void AbortTx().`
- Operações para acesso de dados:
  - `T ReadTx(T *address)` - conjunto de leitura;
  - `void WriteTx(T *address, T v)`- conjunto de escrita.

Transações em Memória - Exemplo

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

Transações em Memória - Exemplo

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

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

Transações em Memória - Exemplo

```
 6 Retirada(i, valor) {
 7     atomic {
 8         contas[i] = contas[i] - valor;
 9     }
10 }
11
12 Transfere(i_orig, i_dest, valor) {
13     atomic {
14         Retirada(i_orig, valor);
15         Deposito(i_dest, valor);
16     }
17 }
```

Mecanismos

Dois mecanismos principais que uma implementação de TM precisa prover:

1. garantir isolamento – detectar e resolver conflitos;
2. gerenciar "tentativa" de trabalho da transação.

Controle de concorrência

- Garantir isolamento, pois acessos a dados podem causar conflitos;
- Eventos: ocorrência, detecção, resolução;

Controle de concorrência

- Garantir isolamento, pois acessos a dados podem causar conflitos;
- Eventos: ocorrência, detecção, resolução;
- Duas abordagens gerais para controle de concorrência:

  Pessimista: 3 eventos, bloqueio, *deadlock*;

  Otimista: conflito primeiro, *livelock*.

Detecção de conflitos

- Diretamente relacionado com o controle de concorrência:

  Controle Pessimista: adquire bloqueio apenas se não há conflito com outra *thread*;

  Controle Otimista: três classificações:

  - **Granularidade** nível de detalhe da informação;
  - **Tempo** detecção ansiosa, validação, detecção preguiçosa;
  - **Tipo de acesso** T_a leu onde T_b escreveu: por tentativa, por efetivação;

Versionamento

- Gerenciar tentativa de trabalho;
- Várias versões do mesmo dado;
- Duas abordagens para versionamento:

  Ansioso *undo-log*;

  Preguiçoso *redo-log*.

Memória Transacional em Software

Transação implementada puramente em software.
Suas principais vantagens são:

- Implementação de grande variedade de algoritmos sofisticados.

Memória Transacional em Software

Transação implementada puramente em software.
Suas principais vantagens são:

- Implementação de grande variedade de algoritmos sofisticados.
- Sotware mais barato de se modificar do que o hardware.

Memória Transacional em Software

Transação implementada puramente em software.
Suas principais vantagens são:

- Implementação de grande variedade de algoritmos sofisticados.
- Sotware mais barato de se modificar do que o hardware.
- Integrado mais facilmente com sistemas existentes.

Memória Transacional em Software

Shavit e Touitou [11] cunharam o termo quando propuseram uma solução totalmente baseada em software.

Características

Para cada palavra compartilhada, um registro de posse aponta para a transação detentora desta.
Para se efetivar a transação, esta deve obter posse de todas as palavras.

Desvantagens

Necessidade de um registro extra para cada palavra.
Conjunto de endereços de dados precisa ser conhecido de antemão.

Para diminuir o overhead de espaço, Fraser [5] propôs um sistema denominado FSTM (*Fraser Software Transaction Memory*).
O objeto é utilizado como unidade básica de armazenamento.

Vantagens

Diminui o overhead de espaço.
Conjunto de endereços de dados não precisa ser conhecido de antemão.

Memória Transacional em Software

A manipulação de transações é feita através de uma API.

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

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

Memória Transacional em Software

Uma das abordagens para STM atuais é a de Saha *et al.* [10], que utiliza o chamado *bloqueio de versão*.
Combina um bloqueio de exclusão-mútua, utilizado para arbitrariar escritas concorrentes, com um número de versão, utilizado pelos leitores para detecção de conflitos.

Características

Se o bloqueio está disponível, então nenhuma transação tem escritas pendentes no objeto, e o bloqueio de versão possui o número atual da versão do objeto.
Se o bloqueio não está disponível, o bloqueio se refere a transação que atualmente utiliza o objeto.

Antes de efetuar a leitura de um objeto, a transação precisa salvar a versão atual do objeto em seu conjunto de leitura.

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

Memória Transacional em Software

- Antes da escrita em um objeto, a transação adquire a versão do bloqueio do objeto e adiciona a referência e o número antigo da versão do objeto no conjunto de escrita.

Memória Transacional em Software

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

Memória Transacional em Software

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

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

Memória Transacional em Software

A operação de efetivação é realizada conforme o pseudo-código:

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

Memória Transacional em Hardware

- São TMs implementadas em hardware.
- A primeira proposta de HTM foi de Knight [7] em 1986.

Vantagens do HTM sobre STM

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

Classificação de acesso a memória dentro de transações

Transação Explicita Instruções especiais de acesso à memória.

```
1 LOOP:
2   begin_transaction()
3   load_transactional f4, conta[i]
4   load f6, desconto[i]
5   SUB f4, f4, f6
6   store_transactional conta[i], f4
7   commit_transaction()
8   ...
```

Transação Implícita Instruções para marcar início e fim.

```
1 LOOP:
2   begin_transaction()
3   load f4, conta[i]
4   load f6, desconto[i]
5   SUB f4, f4, f6
6   store conta[i], f4
7   end_transaction()
8   ...
```

Conjuntos de Escrita e Leitura

- O conjunto de leitura geralmente é tratado como uma extensão da cache.
- Para tratar o conjunto de escrita, algumas abordagens estendem a cache atual [1, 2], enquanto outras utilizam um buffer separado para esta finalidade [4].

Tratando *aborts*

- Quando uma transação aborta devemos restaurar o programa a um estado consistente.
  - Memória: versionamento preguiçoso.
  - Registradores:
    - Delegar para o software.
    - *shadow register*
    - *register renaming*

Limite das HTM

- Tamanho (estruturas do processador)
- Duração (não pode ser maior do que o *quantum*)

Limite das HTM

- Tamanho (estruturas do processador)
- Duração (não pode ser maior do que o *quantum*)
- Duas alternativas:
  1. usar *locks* para executar o código daquela transação; ou
  2. utilizar STM para aquela transação.

Limite das HTM

- Consequências das soluções anteriores:
  1. Perda das propriedades das transações;
  2. *Overhead* do STM
- Surgimento das HTM ilimitadas:
  - Conseguem executar transações maiores do que um *buffer*/cache de processador e/ou mais longas do que o *quantum*;
  - Implementação complexa e varia muito de uma implementação para outra.

Memória Transactional Híbrida (HyTM)

- HTMs e STMs possuem vantagens e desvantagens;
- Solução: híbrida.

Discussão

Memória transacional tem por objetivo reduzir substancialmente a dificuldade de escrever código correto, eficiente e escalável para programas concorrentes.
Como benefícios da TM, podemos citar:

- Facilidade na escrita de programas paralelos
- Facilidade em se conseguir bom desempenho paralelo
- Eliminação de deadlocks
- Facilidade em manter dados consistentes
- Tolerância a falhas

Discussão

Se a TM é tão boa e fácil, por que não é utilizada?

Discussão

Se a TM é tão boa e fácil, por que não é utilizada?

Desvantagens

As TMs, implementadas em software ou hardware, introduzem complexidade que limitam seus ganhos de produtividade esperado. Com isso, possui baixo incentivo de utilização e apenas uma pequena parcela do hardware suporta seu uso.

Discussão

Apesar dos benefícios, alguns pesquisadores discordam que ela possa ser utilizada em sistemas comerciais, e até a classificam como brinquedo de pesquisa [3].
Os principais problemas apontados são:

- I/O
- Transações Zumbis
- Overhead
- Transações limitadas e ilimitadas em HTM

Conclusão

Nesta apresentação, foram apresentados os seguintes tópicos:

- Introdução básica sobre memória transacional.
- Memória Transacional em Software.
- Memória Transacional em Hardware.
- Memória Transacional Híbrida.
- Discussão dos problemas e desafios de MT's.


Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin.
Making the fast case common and the uncommon case simple in unbounded transactional memory.
SIGARCH Comput. Archit. News, 35(2):24–34, 2007.


Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch.
Invisifence: performance-transparent memory ordering in conventional multiprocessors.
SIGARCH Comput. Archit. News, 37(3):233–244, 2009.


Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee.
Software transactional memory: Why is it only a research toy?
Queue, 6(5):46–58, September 2008.


Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière.
Evaluation of amd's advanced synchronization facility within a complete transactional memory stack.
In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.


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


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


Tom Knight.
An architecture for mostly functional languages.
In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.


Kunle Olukotun and Lance Hammond.
The future of microprocessors.
Queue, 3(7):26–29, 2005.


Sandro Rigo, Paulo Centoducatte, and Alexandre Baldassin.
Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente.
2007.
Minicurso apresentado no VIII Workshop on High Performance Computing System (WSCAD'07).


Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg.
Mcrt-stm: a high performance software transactional memory system for a multi-core runtime.
In *Proceedings of the 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM.


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