SISTEMAS TRANSACIONAIS NA LINGUAGEM CMTJAVA

BANDEIRA, Rafael de Leão; DU BOIS, André Rauber; PILLA, Maurício Lima

Universidade Federal de Pelotas (UFPEL)
Centro de Desenvolvimento Tecnológico (CDTEC)
Programa de Pós-Graduação em Computação
{rdlbandeira, dubois, pilla}@inf.ufpel.edu.br

1. INTRODUÇÃO

Recentemente ocorreu uma transição histórica da computação sequencial para a paralela nos processadores usados em computadores pessoais, servidores e dispositivos móveis. Esta mudança marcou o fim de um período no qual o avanço na tecnologia de semicondutores e na arquitetura de computadores melhorava o desempenho dos processadores sequenciais a taxas anuais de 40% a 50% (LARUS; KOZYRAKIS, 2008). Esse período terminou quando os limites práticos da dissipação de energia de um chip foram esgotados, acabando com os ganhos contínuos na velocidade de *clock* e o paralelismo em nível de instrução se tornou mais vantajoso que o desenvolvimento de arquiteturas de processadores cada vez mais complexas.

Devido a este crescente interesse nas arquiteturas paralelas e da necessidade de escrever software para estas arquiteturas, é cada vez mais importante o desenvolvimento de abstrações para a programação concorrente. Memória transacional é um modelo de programação que fornece uma interface de mais alto nível para a escrita de programas *multicore*. Ela diminui consideravelmente a necessidade de lidar diretamente com aspectos da sincronização da execução paralela, diferentemente do que ocorre, por exemplo, quando da utilização de mecanismos baseados em exclusão mútua.

Transação é um bloco de operações de leitura e escrita na memória compartilhada, que satisfaz duas propriedades (HERLIHY; MOSS,1993; HARRIS et al, 2010):

Atomicidade: Quando uma transação termina sua execução é efetivada ou é cancelada. Se efetivada, os valores resultantes das operações realizadas ficam visíveis na memória compartilhada. Quando cancelada, qualquer valor parcialmente computado é descartado.

Isolamento: Transações parecem executar serialmente, ou seja, uma após a outra, pois transações concorrentes não observam o estado intermediário das demais. Isto é, o resultado da execução de diversas transações concorrentemente equivale ao resultado da execução destas em uma ordem serial qualquer.

CMTJava é uma extensão de Java para a programação de memórias transacionais (DU BOIS; ECHEVARRIA, 2009). A linguagem provê abstrações para programação paralela como os objetos transacionais. Esses objetos são acessados como em um programa sequencial, no entanto, o sistema transacional da linguagem garante a consistência do acesso concorrente.

No presente artigo são descritas as principais características da atual implementação do sistema transacional da CMTJava e também da nova implementação que encontra-se em desenvolvimento. Esse novo sistema é baseado no algoritmo da SwissTM (DRAGOJEVIĆ et al., 2009). Também realiza-se uma comparação dos detalhes de implementação desses dois algoritmos.

2. MATERIAL E MÉTODOS

O corrente sistema transacional da CMTJava foi inspirado no algoritmo *Transactional Locking II* (TL2) (DICE et al., 2006). Este algoritmo realiza detecção de conflitos tardia, isto é, efetua as computações em um log local à transação e no fim das operações tenta efetivar essas computações na memória principal. Essa abordagem ocasiona uma degradação de desempenho a medida que o tamanho das transações aumenta, visto que operações conflitantes são apenas identificadas no fim da execução da transação.

Devido a esse problema, foi iniciado o desenvolvimento de um novo sistema de gerenciamento de execução para as transações. Tal sistema emprega a detecção tardia apenas para conflitos de leitura/escrita e detecta conflitos de escrita/escrita de forma adiantada, ou seja, tão logo esse tipo de conflito ocorra. Essa técnica, conhecida como invalidação mista, foi utilizada na implementação da SwissTM.

O mecanismo de validação das transações aplicado pelos dois algoritmos é semelhante. Ambos utilizam um contador global de *commits*, que é incrementado por cada transação efetivada no sistema. Cada transação no início de sua execução lê o valor do contador global de *commits* e, para validar-se, compara este valor com o indicador de versão de cada atributo de memória. Se este valor de versão é maior que o valor do contador lido pela transação, então o atributo correspondente foi modificado após o início da execução da transação. Logo, a transação não é mais válida e deve descartar as operações realizadas, para posteriormente reiniciar sua execução especulativa.

A implementação anterior do sistema transacional da CMTJava é mais simples no que diz respeito ao controle de concorrência. Nesta apenas é necessário utilizar um *lock* para cada região de memória protegida. Em particular, é empregado um mapeamento de *locks* em nível de palavra. Também poderiam ser utilizadas outras associações entre locks e a memória compartilhada, como por exemplo um mapeamento por objeto ou bloco de palavras. Entretanto, essas políticas introduzem a possibilidade de detecção de falsos conflitos. Esse *lock* é utilizado pela transação ao tentar efetivar suas computações na memória. Para tanto, a transação necessita adquirir o *lock* associado a cada palavra de memória contida no seu conjunto de escrita.

Já na nova implementação são utilizados dois *locks* versionados por região de memória, também empregando mapeamento em nível de palavra. Cada atributo de um objeto transacional *m* possui um *lock* de leitura (*r-lock*) e outro de escrita (*w-lock*). *w-lock* é adquirido adiantadamente pela transação a cada operação de escrita, para evitar que outras transações modifiquem *m*. Quando bloqueado, o *lock* de escrita contém o identificador da transação detentora e um valor nulo quando livre. Já *r-lock* é adquirido apenas em tempo de efetivação. A finalidade disto é evitar que outras transações leiam o valor de *m* e assim observem um estado inconsistente da memória, enquanto uma transação está sendo efetivada. Quando livre, *r-lock* contém um número de versão de *m*. Esse número é utilizado para identificar se o valor de *m* não foi modificado desde o início da transação. Quando uma transação é efetivada, além de liberar os *locks* de escrita e leitura, também modifica o valor de *r-lock* para o novo valor do relógio global.

Após cada operação de leitura ou escrita ocorre uma validação para garantir que as transações sempre observam um estado consistente do sistema. Esta propriedade é conhecida como opacidade. Como realiza a aquisição de

locks de forma adiantada, a nova implementação deve liberar os *locks* adquiridos caso a transação seja abortada, tentando adquiri-los novamente ao ser reexecutada.

3. RESULTADOS E DISCUSSÕES

A detecção adiantada encontra conflitos tão logo ocorram, reduzindo o trabalho desperdiçado por transações conflitantes. No entanto, alguns falsos conflitos são encontrados. Em contrapartida, a detecção tardia é mais otimista, no sentido que deixa as transações executarem até o final para detectar conflitos. Dessa forma, alguns conflitos são resolvidos sem cancelamento mas quando isso não é possível, o desperdício de trabalho é maior, comparado a detecção adiantada.

Nesse contexto, a técnica de invalidação mista surge como uma ótima alternativa, pois beneficia-se das vantagens dos dois meios anteriores. Conflitos de escrita/escrita são detectados de maneira adiantada, evitando que um longo tempo seja gasto até o conflito ser detectado e resolvido. Conflitos de leitura/escrita, que muitas vezes podem ser resolvidos sem cancelamentos, são detectados tardiamente, assim aumentando o paralelismo.

Como o trabalho de implementação do sistema transacional com detecção mista de conflitos encontra-se em andamento, comparações de desempenho ainda não puderam ser realizadas. Quando finalizado, pretende-se realizar medições de desempenho dos dois sistemas com transações de diferentes tamanhos, além de diferentes cenários de execução, como por exemplo variando o nível de contenção do sistema e número de núcleos de processamento disponíveis.

4. CONCLUSÕES

As memórias transacionais tem se mostrado um modelo de programação promissor em comparação aos mecanismos baseados em exclusão mútua. Transações fornecem uma abstração de mais alto nível para a escrita de programas concorrentes, deixando o programador concentrado no algoritmo, ao invés de na sincronização da execução. Além disso, memórias transacionais fornecem uma melhor relação entre escalabilidade e esforço de implementação. Apesar de algoritmos usando primitivas de baixo nível para sincronização obterem melhor desempenho, impõem o custo de uma grande complexidade de programação.

Com a difusão de transações para programação paralela, a tendência é o aumento da complexidade das estruturas de dados e do tamanho das transações executadas. Isso faz com que os sistemas de memória transacional sejam cada vez mais exigidos no que diz respeito a desempenho. Para alcançar um desempenho que torne essas aplicações viáveis, é necessário aprimorar os sistemas transacionais tornando-os cada vez mais eficientes.

Após a finalização da implementação do novo algoritmo, o próximo passo do projeto é a realização de testes para a comparação do desempenho das implementações dos sistemas de transações da CMTJava.

5. AGRADECIMENTOS

Agradecemos aos projetos PRONEX/FAPERGS/CNPq GREEN-GRID Computação de Alto Desempenho Sustentável e Composição de Ações Transacionais (Pesquisador Gaúcho/FAPERGS) pelo apoio nessa pesquisa.

6. REFERÊNCIAS BIBLIOGRÁFICAS

DICE, D.; SHALEV, O.; SHAVIT, N. Transactional locking II. **Proc. of the 20th Intl. Symp. on Distributed Computing**. 2006.

DRAGOJEVIĆ, A.; GUERRAOUI, R.; KAPALKA, M. Stretching transactional memory. **Sigplan Notices, 44**. 2009.

DU BOIS, A. R.; ECHEVARRIA, M. A Domain Specific Language for Composable Memory Transactions in Java. **DSL '09: PROCEEDINGS OF THE IFIP TC 2 WORKING CONFERENCE ON DOMAIN-SPECIFIC LANGUAGES.** Berlin, Heidelberg. Springer-Verlag, p.170–186. 2009.

HARRIS, T.; LARUS, J.; RAJWAR, R. **Transactional Memory, 2nd Edition**. Morgan and Claypool Publishers, 2nd edition, 2010.

HERLIHY, M.; MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. **SIGARCH Comput. Archit. News**, New York, NY, USA,v.21, n.2, p. 289–300, 1993.

LARUS, JAMES; KOZYRAKIS, CHRISTOS. **Transactional Memory**. *Commun. ACM* 51, 7. New York, NY, USA. 2008.