PROJETO DE UM ESQUEMA DE DETECÇÃO DE CONFLITOS PARA CMTJAVA

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

Programa de Pós-Graduação em Computação
Centro de Desenvolvimento Tecnológico (CDTEC)
Universidade Federal de Pelotas (UFPEL)
Campus Universitário, s/nº
Caixa Postal 354 – 96010-900 – Pelotas – RS – Brasil
{rdlbandeira, dubois, pilla}@inf.ufpel.edu.br

1. INTRODUÇÃO

O paradigma de Memória Transacional foi proposto primeiramente em hardware (*Hardware Transactional Memory* - HTM) (HERLIHY; MOSS, 1993). No entanto, a maioria das HTMs suporta transações de tamanho fixo e não garantem o progresso das transações sob eventos específicos do sistema, como interrupções. Abordagens híbridas executam transações pequenas em hardware e transações maiores em software, ou aceleram algumas operações da STM (*Software Transactional Memory*) em hardware.

Atualmente, existe um crescente interesse no estudo das Memórias Transacionais em Software (SHAVIT; TOUITOU, 1995) como mecanismo de controle da concorrência para computação paralela, uma vez que esse método simplifica bastante a escrita de programas usando memória compartilhada, comparado com outros métodos como exclusão mútua.

Nessa abordagem, expressa-se a concorrência por meio de blocos de código chamados transações, que executam concorrentemente acessando a memória compartilhada. Ao serem detectados conflitos no acesso a esses dados, a transação é reexecutada. A simplicidade das memórias transacionais reside no fato de que o controle do acesso à memória é realizado pelo sistema transacional, ao invés de realizado pelo programador. Memórias transacionais eliminam muitas das dificuldades da programação baseada em *locks*, como *deadlocks* (pois *locks* não são manipuladas de forma explícita) e inversão de prioridade (HARRIS et al., 2005).

A linguagem CMTJava (DU BOIS; ECHEVARRIA, 2009) é uma extensão de Java para a programação de memórias transacionais. O sistema transacional da linguagem é baseado no algoritmo TL2. Esse algoritmo é amplamente utilizado e apresenta ótimo desempenho para transações pequenas, no entanto tem seu desempenho comprometido em transações maiores. O motivo do mau desempenho do TL2 é sua detecção tardia de conflitos de escrita/escrita, isto é, de duas transações que tentam escrever em uma posição de memória ao mesmo tempo, que faz com que transações condenadas por um conflito executem até o fim, desperdiçando recursos.

Este trabalho discute a implementação de um sistema transacional para a CMTJava de forma a melhorar o desempenho geral da linguagem, minimizando os problemas gerados no sistema atual por transações grandes, também melhorando o gerenciamento de contenção da linguagem.

2. METODOLOGIA

Dentre as diversas implementações de STMs, algumas estratégias de implementação impactam diretamente em seu desempenho e no modelo de programação usado.

O gerenciador de contenção (*contention manager*) é responsável por detectar conflitos dentre as transações concorrentes e resolvê-los. Ele decide o que uma transação (agressor) deve fazer caso conflite com outra transação (vítima). Possíveis soluções são: abortar o agressor, abortar a vítima, ou forçar o agressor a reexecutar depois de algum período. O esquema mais simples, chamado tímido, é sempre abortar o agressor, possivelmente com alguma temporização. A abordagem gulosa atribui um *timestamp* a cada transação. Em caso de conflito, a transação com maior *timestamp* é abortada. Uma característica importante do método guloso é que elimina a possibilidade de ocorrência de *starvation* (inanição).

Para verificar conflitos, uma transação precisa adquirir todas as posições de memória que atualiza. Essa aquisição pode ser em tempo de encontro (*encounter-time locking*), isto é, tão logo ocorra a primeira modificação, detectando conflitos imediatamente após sua ocorrência, ou em tempo de efetivação (*commit-time locking*). Também é possível a adoção de um esquema de detecção misto, que detecta conflitos de escrita/escrita adiantadamente e detecta tardiamente conflitos de leitura/escrita.

O algoritmo *Transactional Locking II* (TL2) (DICE et al., 2006) utiliza um relógio de versão global (*global version-clock*) que é incrementado para cada transação que escreve na memória, e é lido por todas as transações. O TL2 utiliza o esquema de aquisição de *locks* em tempo de efetivação e um gerenciador de contenção tímido.

SwissTM (DRAGOJEVIĆ et al., 2009) é uma implementação de Memórias Transacionais que busca obter bom desempenho tanto para transações curtas e estruturas de dados pequenas, assim como para transações grandes e cargas de trabalho complexas. SwissTM detecta conflitos de escrita/escrita de forma adiantada, de maneira a prevenir transações condenadas a abortar de continuarem executando e assim gastando recursos, e conflitos de leitura/escrita são detectados de forma tardia, para permitir de forma otimista mais paralelismo. O gerenciador de contenção usa o esquema tímido para transações curtas ou somente leitura, que aborta a transação tão logo encontre um conflito. Transações mais complexas trocam dinamicamente para o mecanismo guloso, que apesar de envolver maior *overhead* favorece esse tipo de transação, prevenindo *starvation*.

A Tabela 1 sintetiza as principais características de implementação dos sistemas SwissTM e TL2. Apesar de algumas semelhanças, como a detecção de conflitos em nível de palavra e sincronização baseada em locks, as implementações usam políticas diferentes para gerenciamento de contenção e detecção de conflitos.

Característica	**TL2**	**SwissTM**
Gerenciador de contenção	Tímido	Duas Fases
Detecção de conflitos	Tardia	Mista
Gran. detecção do conflito	Palavra	Palavra
Sincronização	*Locks*	*Locks*

3. RESULTADOS E DISCUSSÕES

HONG et al. (2010) apresenta uma análise usando EigenBench dos sistemas TL2 e SwissTM de três importantes características. O EigenBench é um *framework* de avaliação de sistemas de memória transacional. No que diz respeito ao tamanho da transação, TL2 é extremamente suscetível a esse fator, com mau desempenho para transações muito pequenas e também para valores acima de um limiar. Em contrapartida, SwissTM lida igualmente bem com todos os tamanhos de transação. Isso se deve ao esquema utilizado que aborta transações grandes tão logo conflitem com outra transação concorrente. Ao avaliar os efeitos do número de escritas transacionais, percebe-se que ao aumentar o número de escritas, a performance do TL2 cai mais drasticamente do que a do SwissTM, que tem uma performance geral melhor. A razão do melhor desempenho da SwissTM é a detecção otimista de conflitos leitura/escrita, e a detecção pessimista de conflitos escrita/escrita, comparado ao TL2. Quanto a escalabilidade dos dois sistemas, ambos mostram-se escalar bem com as transações não conflitantes embora o SwissTM mantenha menor *overhead*.

Tendo como base a implementação atual da linguagem, sem alterar sua interface e sem afetar os programas já escritos, serão modificadas as políticas de detecção e resolução de conflitos do sistema transacional da CMTJava e os resultados obtidos com esse novo gerenciamento serão comparados aos atuais.

4. CONCLUSÕES

O algoritmo TL2 funciona bem para transações pequenas, devido ao esquema de aquisição de *locks* em tempo de efetivação, no entanto perde em desempenho em transações maiores que eventualmente abortam por conflitos de escrita/escrita, pois estes são detectados tardiamente. Apesar do desempenho, esse algoritmo é bastante importante pela sua influência em outras implementações. Já a SwissTM mostra-se bastante eficiente em diversos cenários, devido a sua capacidade de se adaptar a carga de trabalho. Essa eficiência deve-se a adoção das estratégias mais eficientes de detecção de conflitos, gerando menor *overhead*.

Com a aplicação de tais estratégias na CMTJava espera-se melhorar seu desempenho geral e entender melhor a relação das características de implementação de memórias transacionais com as limitações e benefícios impostos por essas.

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.; MARLOW, S.; PEYTON-JONES, S.; HERLIHY, M. Composable memory transactions. **Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05**, pages 48–60, New York, NY, USA. ACM. 2005.

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.

HONG, S.; OGUNTEBI, T.; CASPER, J.; BRONSON, N.; C. KOZYRAKIS; K. OLUKOTUN. EigenBench: A simple exploration tool for orthogonal TM characteristics. **IEEE International Symposium on Workload Characterization**, 2010.

SHAVIT, N.; TOUITOU, D. Software transactional memory. **PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing**, p. 204–213, New York, NY, USA. ACM. 1995.