UMA BIBLIOTECA DE STM HASKELL COM VERSIONAMENTO ADIANTADO

RODRIGO MEDEIROS DUARTE; ANDRÉ R. DU BOIS; GERSON GERALDO H. CAVALHEIRO

UFPEL- Universidade Federal de Pelotas
LUPS – Laboratory of Ubiquitous and Parallel Systems
{rmduarte, dubois, gerson.cavalheiro}@inf.ufpel.edu.br

1. INTRODUÇÃO

Uma das maiores dificuldades no desenvolvimento de programas paralelos é manter a correta sincronização do acesso aos dados compartilhado entre *threads* concorrentes. Os métodos tradicionais utilizando bloqueios exigem muita experiência e habilidade do programador para evitar problemas como condição de corrida e *deadlocks* (LEE, 2006). Afim de aumentar a abstração e facilitar a programação paralela um novo modelo de sincronismo, conhecido como memória transacional, tem sido desenvolvido. Neste novo modelo, todo o sincronismo entre *threads* concorrentes é realizado através do sistema transacional, eximindo o programador deste processo.

STM Haskell é uma extensão da linguagem funcional Haskell que fornece a abstração de memórias transacionais para a programação paralela.

Neste artigo apresentamos **LSTM** (lups software transactional memory), uma nova implementação de STM Haskell, totalmente desenvolvida utilizando a linguagem Haskell. Esta nova implementação é a primeira para Haskell utilizando versionamento de dados adiantado e detecção de conflitos adiantada. Seu desempenho é comparado ao de outras duas implementações, a TL2 também desenvolvida em Haskell, mas com versionamento tardio e detecção de conflitos tardia e a STM Original, desenvolvida em C e com versionamento tardio e detecção de conflitos tardia. Resultados mostraram que a **LSTM** é mais eficiente que a TL2 e apresenta melhor escalabilidade que a STM original.

2. METODOLOGIA

2.1 **– Memórias Transacionais:**

Para garantir o correto sincronismo entre *threads*, memórias transacionais utilizam transações parecidas com as transações de bancos de dados (RIGO, 2007). Transações garantem atomicidade e isolamento (propriedades herdada de banco de dados), através de duas características, o versionamento de dados e a detecção de conflitos.

O Versionamento de dados pode ser de dois tipos, adiantado e tardio. No versionamento adiantado os dados especulativos gerados pela transação são armazenados diretamente na memória e os dados correntes são armazenados em um *undo log*, Caso a transação consiga efetivar, os dados já se encontram na memória e nada mais precisa ser feito, mas se houver um conflito, os dados do *undo log* são repostos na memória e o valor especulativo é descartado.

Já no versionamento tardio, os dados especulativos são armazenados em um *buffer* local da transação. Caso a transação seja efetivada os dados do *buffer*

FAPERGS/PqG (11/1065-1), PRONEX/FAPERGS/CNPq (10/0042-8).
Bolsista FAPERGS/PROBIC

são gravados na memória. Em caso de conflito, os dados especulativos (*buffer*), são simplesmente descartados.

Versionamento adiantado apresenta melhor desempenho em casos de baixa contenção (conflitos), pois os valores especulativos já se encontram na memória o que facilita a efetivação da transação. Porém em casos de alta contenção, este modelo não é eficiente por ter o custo extra de desfazer as alterações realizadas na memória em caso de cancelamento da transação. Já com o versionamento tardio em casos de alta contenção, este modelo é mais eficiente, pois basta descartar o valor do buffer local.

Um conflito ocorre quando pelo menos duas transações acessam um mesmo dado na memória e um destes acessos é de escrita. A detecção de conflitos também pode ser realizada de forma adiantada e tardia.

Na detecção adianta, um conflito é detectado no momento que uma transação acessa um dado na memória. Se o mesmo dado esta sendo acessado por outra transação, o conflito é detectado e a transação reiniciada. Já na detecção tardia, somente no momento da efetivação (*commit*), é realizado a verificação de conflito.

É importante salientar que, utilizando-se versionamento adiantado a detecção de conflitos também deve ser adiantada para evitar inconsistência na computação realizada. Percebe-se também que detectar conflitos de forma adiantada evita que se utilize um ou mais *cores* com computação desnecessária, porém pode-se cancelar transações que, dependendo do progresso de outras, poderiam ser efetivadas normalmente.

2.2 **- STM Haskell**

STM Haskell é uma extensão da linguagem Haskell que fornece a abstração de memórias transacionais para a programação paralela.

Haskell é ideal para implementar memórias transacionais porque o sistema de tipos de Haskell consegue separar as ações que possuem efeito colaterais das que não possuem. Nem todo o tipo de ação pode ser executada dentro de uma transação (operações de entrada e saída por exemplo). Dentro de uma transação, somente operações de modificam a memória podem ser realizadas. Para isso STM Haskell define um tipo de variável transacional (`TVar a`). Essa variável só pode ser modificada por duas primitivas, a `readTVar` e `writeTVar`. As operações realizadas por estas primitivas só podem ser executadas dentro de um bloco `atomically`. Desta forma, o sistemas de tipos de Haskell garante que as ações realizadas por estas primitivas sobre uma `TVar` sejam sempre protegidas pelo bloco `atomically`.

Haskell fornece outras duas primitivas `retry` e `orElse`, A primitiva `retry` possui a propriedade de abortar uma transação e reexecutar novamente quando pelo menos uma das variáveis lidas pela transação tenha sido modificada e, `orElse` recebe como argumento duas ações transacionais e devolve uma outra ação transacional que faz uma escolha entre as ações que recebeu.

2.3 **– Implementação**

A implementação da **LSTM** foi baseada no algoritmo da TinySTM (FELBER, 2008) com algumas modificações para adaptar o algoritmo na linguagem Haskell.

Assim como a Tiny, a granularidade de detecção de conflitos é em nível de palavra. Dessa forma cada posição de memória (no caso da **LSTM**, cada `TVar`) é

protegida por um *lock.* Esse *lock* é adquirido quando uma transação tenta escrever em uma variável transacional. Se alguma transação já se encontra de posse deste *lock*, então a transação detecta o conflito e é abortada. O *lock* de cada `TVar` é um *lock* versionado (HARRIS, 2010). i.e., se está livre contém o número da versão da `TVar`, se está bloqueado, contem o *id* da transação que possui o *lock*. A versão de cada `TVar` é definida por um relógio global, o qual é utilizado para controlar a ocorrência de conflitos entre transações.

3. RESULTADOS E DISCUSSÃO

Para os testes foi utilizado o Haskell STM Benchmark (PERFUMO, 2007). Os testes constaram de 20 execuções de cada aplicação utilizando as três implementações (STM Original, TL2 e LSTM), em uma maquina core i7, com sistema operacional Ubuntu 12.04 e compilador Haskell GHC 7.4.1. As aplicações utilizadas foram a SI, LL, BT, HT, TC e CCHR-Sudoku.

A implementação da STM Original (que é escrita em C) vem junto com Haskell Platform, TL2 (DU BOIS, 2011) é baseado no algoritmo da TL2 (DICE, 2006) e também é escrita em Haskell. Ambas comparadas com a **LSTM**, possuem versionamento tardio e detecção de conflitos tardia.

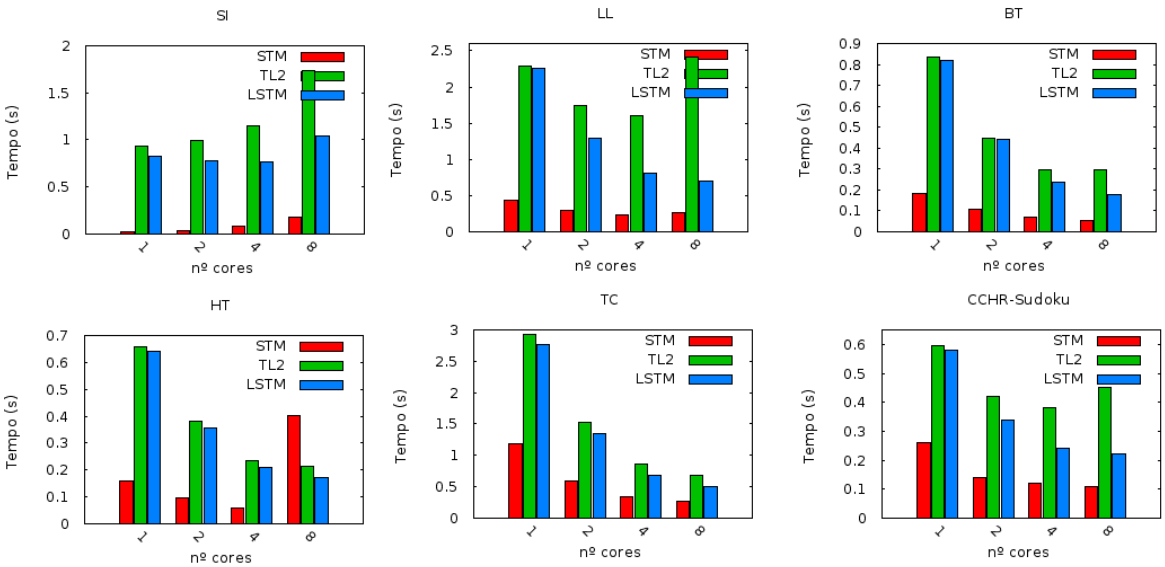

Figura 1

Os resultados da comparação entre as três implementações (Figura 1), mostram que o tempo de execução da **LSTM** foi menor do que o da implementação TL2 usando versionamento tardio. Já comparado a STM original que foi desenvolvida em C, o tempo de execução da **LSTM** foi maior, porém apresentou melhor escalabilidade em quase todos os casos.

Somente na aplicação HT, Tanto a **LSTM** como a TL2 apresentaram melhor resultado se comparado a STM original. Nas aplicações LL, BT e TC, o tempo de execução da **LSTM** se aproxima bastante do tempo de execução da STM Original.

4. CONCLUSÕES

A abordagem de versionamento adiantado e detecção adiantada da **LSTM** mostrou melhor desempenho que a abordagem de versionamento tardio e

detecção tardia da TL2. Esta diferença no desempenho está relacionada ao fato de que em uma abordagem de detecção adiantada, evita-se que se utilize um ou mais *cores* na execução de transações que estariam fadadas a abortar, liberando estes para executarem outras transações. Um exemplo disto está na execução da aplicação SI, nesta aplicação várias transações concorrem ao acesso de um mesmo dado na memória, gerando assim grande quantidade de conflitos. Neste *benchmark* pode-se ver que conforme se aumenta o número de *cores* se consegue aumento de desempenho com a **LSTM**, o que não ocorre nas outras duas implementações (TL2 e STM Original), porém quando se aumenta muito o número de *cores* utilizados, a taxa de conflitos aumenta e o desempenho acaba caindo para todas as implementações.

Apesar de apresentar um tempo maior do que o da STM Original, a **LSTM** apresentou melhor escalabilidade. Este resultado indica que, se a **LSTM** for implementada em uma linguagem mais eficiente, como C, poderia resultar em um desempenho melhor que a STM Original. Mais testes são necessários para comprovar esta hipótese.

5. REFERÊNCIAS BIBLIOGRÁFICAS

HARRIS, T., J. Larus, and R. Rajwar, “Transactional memory (synthesis lectures on computer architecture),” **Synthesis Lectures on Computer Architecture**. Morgan and Claypool, 2010.

FELBER, P., 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. ACM, 2008, pp. 237–246.

LEE, E. A., “The problem with threads,” **Computer, vol. 39**, pp. 33–42, May 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1137232.1137289

T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, “Composable memory transactions,” in Proceedings of the tenth **ACM SIGPLAN** symposium on Principles and practice of parallel programming. ACM, 2005, pp. 48–60.

PERFUMO, C., N. Sonmez, A. Cristal, O. S. Unsal, M. Valero, and T. Harris, “Dissecting transactional executions in haskell,” in In The Second **ACM SIGPLAN** Workshop on Transactional Computing (TRANSACT, 2007).

DU BOIS, A. R., “An implementation of composable memory transactions in haskell,” in Software Composition,ser. **Lecture Notes in Computer Science**. Springer Berlin Heidelberg, 2011, vol. 6708, pp. 34–50.

DICE, D., O. Shalev, and N. Shavit, “Transactional locking ii,” in Distributed Computing. **Springer**, 2006, pp. 194– 208.

RIGO S., C. P. and A., B. Memórias Transacionais, uma nova alternativa para programação concorrente, **WSCAD**, Anais Eletrônicos do WSCAD,2007.