Implementação de uma versão de STM Haskell com versionamento adiantado

Rodrigo M. Duarte[1], **André R. Du Bois**[1], **Gerson Geraldo H. Cavalheiro**[1]

[1]CDTEc - Centro Desenvolvimento Tecnológico
Bacharelado em Eng. Computação – Universidade Federal de Pelotas (UFPEL)
Gomes Carneiro 1 – 96.010-610 – Pelotas – RS – Brasil

{rmduarte, dubois, gerson.cavalheiro}@inf.ufpel.edu.br

Resumo. *Este artigo tem como objetivo apresentar um novo modelo de abstração para programação paralela conhecido como memórias transacionais, bem como sua aplicação na linguagem STM Haskell. O artigo também apresenta como proposta a implementação de uma versão diferente do STM Haskell, no intuito de explorar quais os impactos de uma diferente abordagem de versionamento de dados no desempenho e no consumo de energia.*

1. Introdução

Devido às dificuldades apresentadas na programação paralela utilizando bloqueios (*locks*) [Jones 2007, Rigo et al. 2007], um novo modelo de programação conhecido como memórias transacionais tem sido desenvolvido para facilitar a programação *multicore*. Memórias transacionais utilizam transações parecidas com as de bancos de dados para garantir sincronismo entre *threads*. Duas importantes características das memórias transacionais que foram herdadas de bancos de dados são a atomicidade e isolamento, características estas que fornecem maior abstração, facilitando a programação paralela, o reuso de código e propiciam maior escalabilidade [Rigo et al. 2007]. Uma extensão da linguagem funcional Haskell, conhecida como STM-Haskell, fornece suporte a programação paralela usando o modelo de memória transacional. Esta extensão torna Haskell uma excelente e fácil ferramenta para o desenvolvimento de programas paralelos.

A proposta apresentada neste artigo é uma implementação diferente de STM Haskell utilizando versionamento adiantado (vide seção 2.1) e, desta forma, realizar a comparação desta nova implementação com a já existente, que utiliza versionamento tardio. Este artigo está organizado como segue: Na seção 2 é contextualizado o que é memória transacional e suas características. A seção 3 apresenta o STM Haskell que é uma extensão da linguagem funcional Haskell implementando memórias transacionais e finalmente na seção 4, é apresentado de forma mais detalhada o trabalho a ser desenvolvido.

2. Memórias Transacionais

Memória transacional é um novo modelo de sincronização entre *threads* que aumenta o nível de abstração. Este modelo de programação concorrente usa como base o conceito de transações para garantir o sincronismo entre as *threads*.

Em memórias transacionais uma transação é uma sequência de operações que modificam a memória de forma atômica, ou seja, são executadas por completo (*commit*) ou

podem ser abortadas (*abort*) caso ocorra um conflito. Outro item importante é a garantia de isolamento, assim, transações não podem ver os resultados intermediários uma das outras. Dessa forma a execução de transações concorrentes equivale ao resultado da execução dessas mesmas transações em alguma ordem serial. Essas características (atomicidade e isolamento) são garantidas por duas propriedades que são o versionamento de dados e detecção de conflitos.

2.1. Versionamento de dados

Para garantir atomicidade, uma transação deve guardar tanto o dado corrente quanto o modificado até o momento do *commit*. Para isso, memórias transacionais usam dois tipos de versionamento, o adiantado e o tardio:

- **Versionamento adiantado:** os dados alterados pela transação são escritos diretamente na memória e os dados correntes são armazenados em um *undo log*. Caso não ocorra nenhum conflito, a única operação realizada deve ser descartar o valor do undo log, mas se caso ocorra um conflito, o valor do *undo log* deve ser restaurado na memória e a transação reiniciada.
- **Versionamento tardio:** operações de escrita em memória são armazenados em um *buffer* local. Se a transação conseguir realizar um *commit*, os valores do *buffer* devem ser escritos na memória, caso contrário se ocorrer um conflito, o valor do *buffer* local deve ser descartado e a transação reiniciada.

Versionamento adiantado apresenta melhor desempenho em casos de baixa contenção (conflitos), pois os valores especulativos já estão na memória, porém em casos de alta contenção este modelo de versionamento não é tão eficiente, pois neste caso a transação tem o custo extra de desfazer as alterações na memória. No entanto uma abordagem de versionamento tardio, em casos de alta contenção, é mais eficiente na efetivação (*commit*). Transações devem transferir seus dados locais para a memória, porem em cancelamentos não é necessário nenhuma intervenção extra.

2.2. Detecção de conflitos

Um conflito ocorre quando pelo menos duas transações estão acessando um mesmo dado na memória e pelo menos um destes acessos é de escrita. A detecção de conflitos também pode ser realizada de forma adianta (pessimista) e ou tardia (otimista)

- **detecção adiantada:** este tipo de detecção de conflito ocorre no momento em que uma transação realiza acesso a memória (leitura ou escrita). Se o mesmo dado em memória esta sendo acessado por outra transação, o conflito é detectado e a transação é reiniciada ou abortada.
- **detecção tardia:** neste método, os conflitos são detectados somente no momento do *commit*, ou seja, no final da execução da transação.

É importante perceber que versionamento de dados adiantado necessita que a detecção de conflitos também o seja desta forma, para evitar inconsistência nos dados computados pelas transações. Porém o versionamento de dados tardio não obedece a esta regra, o mesmo pode usufruir de ambos os tipos de detecção de conflitos. Para resolver um conflito, as implementações de memórias transacionais utilizam um gerenciador de contenção (*Contention Management*) [Guerraoui et al. 2005], o qual tem a finalidade de decidir qual transação deve ser abortada ou reiniciada através de regras especificas onde, dependendo do tipo de regra adotada pelo gerenciador, pode-se conseguir melhor desempenho por diminuir a ocorrência de abortos desnecessários. Alguns

modelos de gerenciadores de contenção mais utilizados são o tímido, guloso e backoff [Demsky and Dash 2010].

3. STM Haskell

STM Haskell é uma extensão da linguagem funcional Haskell que fornece a abstração de memórias transacionais para a programação paralela [Harris et al. 2005].

Uma linguagem funcional como Haskell é ideal para implementar memórias transacionais por dois motivos [Harris et al. 2005]:

- O tratamento de tipos consegue separar as ações que possuem efeito colateral das que não possuem. Nem todo o tipo de ação pode ser executada dentro de uma transação, por exemplo operações de entrada e saída. O sistema de tratamento de tipos de Haskell consegue garantir que somente as ações corretas serão executadas dentro de uma transação, evitando assim que o programador cometa erros durante a programação.
- Em Haskell, a maioria das computações são puras, ou seja, não possuem nenhum tipo de efeito colateral. Esse tipo de ação não modifica a memória e não precisa ser acompanhada pelo sistema transacional. Essas ações nunca precisam ser desfeitas em caso de uma transação abortar.

Dentro de transações, apenas operações que modificam a memória podem ser executadas. Para isso STM Haskell prove um tipo especifico de variável transacional `TVar a`. Esta tipo de variável pode ser modificada por duas primitivas especificas:

```
readTVar ::  TVar a -> STM a
writeTVar ::  TVar a -> a -> STM ()
```

A primitiva `readTVar` recebe como argumento uma variável do tipo `TVar` e retorna como resposta uma ação transacional do tipo STM que quando executada, retorna o valor contido na variável `TVar`. A primitiva `writeTVar` serve para escrever um valor em uma variável do tipo `TVar a`

A única maneira de executar estas ações envolvendo STM é utilizando a primitiva `atomically`

```
atomically::STM a -> IO a
```

A primitiva `atomically` faz com que as transações sejam executadas atomicamente em relação as outras transações, garantindo atomicidade e isolamento [Jones 2007].

STM-Haskell também possui outras duas primitivas de alto nível, `retry` e `orElse`. `retry` possui a propriedade de abortar uma transação e executar ela novamente no momento que pelo menos uma das variáveis lidas pela transação tenha sido modificada. A primitiva `orElse` permite escolher entre os resultados computados por transações. `orElse` recebe como argumento duas transações e devolve uma outra ação transacional que faz uma escolha entre as ações que recebeu como argumento. Exemplo:

```
orElse t1 t2
```

Primeiramente a transação `t1` é executada, se esta abortar ela então é descartada e a segunda transação passa a ser executada. Se t2 também abortar, então o conjunto completo das duas transações é reexecutado.

4. Proposta

A implementação atual de STM Haskell possui versionamento e detecção de conflitos tardia e um gerenciador de contenção tímido. O objetivo deste trabalho é a implementação de uma versão de STM Haskell utilizando versionamento e detecção de conflitos adiantada, tomando como base o algoritmo da *TinySTM* [Felber et al. 2008]. Com a implementação desta nova versão do STM Haskell, serão realizados testes de desempenho e consumo de energia de ambas implementações (atual e a proposta), sobre diferentes aplicações utilizando o STM Haskell Benchmark [Perfumo et al. 2007].

A implementação desta nova versão de STM Haskell também será usada em trabalhos futuros para o estudo da implementação de diferentes abordagens de gerenciadores de contenção. O objetivo e conseguir um conjunto completo de implementações de memórias transacionais e desta forma realizar testes em abordagens mais amplas, verificando como seria o desempenho de programas paralelos sobre diferentes abordagens de versionamento de dados e detecção de conflitos.

5. Agradecimentos

O primeiro autor gostaria de agradecer a FAPERGS pela bolsa PROBIC, concedida ao projeto. Este trabalho é financiado pelos projetos FAPERGS/PRONEX/CNPq GREEN-GRID e FAPERGS/PESQUISADOR GAÚCHO CMTJava.

Referências

Demsky, B. and Dash, A. (2010). Evaluating contention management using discrete event simulation. In *Fifth ACM SIGPLAN Workshop on Transactional Computing (April 2010)*.

Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM.

Guerraoui, R., Herlihy, M., and Pochon, B. (2005). Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264. ACM.

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

Jones, S. (2007). Beautiful concurrency. *Beautiful Code: Leading Programmers Explain How They Think*, pages 385–406.

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

Rigo, S., Centoducatte, P., and Baldassin, A. (2007). Memórias transacionais: Uma nova alternativa para programação concorrente. In *Minicursos do VIII Workshop em Sistemas Computacionais de Alto Desempenho, WSCAD 2007*.