Primitivas para aplicação de Transactional Boosting no STM Haskell

Jonathas A. O. Conceição[1], André R. Du Bois[1], Rodrigo G. Ribeiro[2]

[1]Universidade Federal de Pelotas (UFPel)
Pelotas – RS – Brasil

{jadoliveira,dubois}@inf.ufpel.edu.br

[2]Universidade Federal de Ouro Preto (UFOP)
Ouro Preto – MG – Brasil

rodrigo.ribeiro@ufop.edu.br

Resumo. *Transactional Boosting é uma técnica que pode ser usada para transformar ações linearmente concorrentes em ações transacionalmente concorrentes, possibilitando assim sua utilização em blocos transacionais. Esta técnica pode ser utilizada para resolução de falsos conflitos, evitando a perda de desempenho de algumas aplicações. O STM Haskell atualmente não oferece as ferramentas necessárias para aplicação de tal técnica. O objetivo deste trabalho então é apresentar quatro novas primitivas para o STM Haskell, que em conjunto podem ser usadas para a aplicação do Transactional Boosting.*

1. Introdução

Software Transactional Memory (STM) é uma alternativa de alto nível ao sistema de sincronização por *locks*. Nela todo acesso à memória compartilhada é agrupado como transações que podem executar de maneira concorrente. Se não houve conflito no acesso à memória compartilhada, ao fim da transação um *commit* é feito, tornando assim o conteúdo dos endereços de memória público para o sistema. Caso ocorra algum conflito um *abort* é feito descartando qualquer alteração ao conteúdo da memória e a transação é recomeçada. Diferente da sincronização por *locks*, transações podem ser facilmente compostas e são livres de *deadlocks* [Harris et al. 2008].

STM funciona através da criação de blocos atômicos onde alterações de dados são registradas para detecção de conflitos. Um conflito ocorre quando duas ou mais transações acessam o mesmo endereço e pelo menos um dos acessos é de escrita. Entretanto, essa forma de detecção de conflitos pode, em alguns casos, gerar falsos conflitos levando a uma perda de desempenho. Um exemplo seria quando duas transações modificam partes diferentes de uma lista encadeada [Sulzmann et al. 2009]. Embora essas ações não conflitem, o sistema detecta um problema, pois uma transação modifica uma área de memória lida por outra transação. Este tipo de detecção de conflito pode ter grande impacto na performance quando se utiliza certos tipos de estruturas encadeadas. Por outro lado, utilizando sincronização por *locks*, ou mesmo algoritmos *lock-free*, programadores experientes podem alcançar um alto nível de concorrência ao custo de complexidade no código.

Transactional Boosting [Herlihy and Koskinen 2008] é uma metodologia que pode ser usada para resolver problemas de falsos em STM. A técnica permite, por exemplo, usar estrutura de lista encadeada *thread-safe* otimizada dentro da transação, porém

em caso de *abort* deve-se registrar ações para desfazer as modificações realizadas na lista, e em caso de *commit* deve-se executar ações que tornem as modificações visíveis para o sistema. A técnica consiste então em desenvolver junto à rotina de acesso a memória duas outras ações, uma para tornar público o resultado da rotina em caso de *commit*, e uma segunda para reverter alterações à memória para ser usada em caso de *abort*. A rotina à qual se deseja fazer o *Boosting* é adicionada a transação, e suas ações de *commit* e *abort* são postas como *handlers* para serem tratados ao fim da transação.

Entretanto, interface para instruções atômicas, *locks*, e várias outras opções de controle de concorrência, em Haskell são modeladas sobre a Mônada IO. Para garantir a exatidão na execução de uma transação, ações STM não podem ser livremente compostas à estas ações IO. Além disso, o STM Haskell não possibilita ao programador adicionar *handlers* para serem executados em caso de *commit* ou *abort*. O objetivo deste trabalho então é apresentar um conjunto de novas primitivas para o STM Haskell, permitindo ao programador a aplicação do *Transactional Boosting*, flexibilizando assim o uso de Memórias Transacionais na linguagem.

2. STM Haskell

STM Haskell [Harris et al. 2008] é uma biblioteca do *Glasgow Haskell Compiler* que provê primitivas para o uso de memórias transacionais em Haskell. O programador define ações transacionais que podem ser combinadas para gerar novas transações. O sistema de tipos da linguagem só permite acesso à memória compartilhada dentro das transações e transações não podem ser executadas fora de uma chamada ao **atomically**, assim, o *RunTime* da linguagem pode garantir que a Atomicidade (a transação se torna visível toda de uma vez) e Isolamento (durante a execução, uma transação não é afetada por outra) são sempre mantidos.

A biblioteca define um conjunto de primitivas para utilização de Memórias Transacionais em Haskell (Figura 1). Nela o acesso a memória compartilhada é feito através de variáveis transacionais, as *TVar*s, que são variáveis acessíveis apenas dentro de transações. Ao fim da execução de um bloco transacional, um registro de acesso às *TVars* é analisado pelo *RunTime System* para determinar se a transação foi bem-sucedida ou não, para então realizar o *commit* ou *abort* da transação.

As primitivas **newTVar**, **readTVar** e **writeTVar** são usadas respectivamente para criar, ler e escrever em *TVars*. As transações acontecem dentro da monada STM e essas ações podem ser compostas para gerar novas ações através dos operadores monádicos (bind (>>=), then (>>), e **return**), ou com a utilização da notação **do**. O **retry** e o **orElse** são primitivas que controlam a execução do bloco. **retry** é utilizada para abortar uma transação e colocá-la em espera até que alguma de suas *TVars* seja alterada por outra transação. **orElse** é uma primitiva de composição alternativa, ela recebe duas ações transacionais e apenas uma será considerada; se a primeira ação chamar **retry** ela é abandonada, sem efeito, e a segunda ação transacional é executada; se a segunda ação também chamar **retry** todo o bloco é reexecutado.

3. Novas Primitivas

Para compor ações IO e STM em Haskell é importante ter em mente duas diferenças fundamentais em sua execução.

```
-- Execution control
atomically :: STM a -> IO ()
retry :: STM a
orElse :: STM a -> STM a -> STM a

-- Transactional Variables
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

Figura 1. Interface do STM Haskell.

```
abort :: STM a
performIO :: IO a -> STM a
addAbortHandler :: IO a -> STM ()
addCommitHandler :: IO a -> STM ()
```

Figura 2. Interface das novas primitivas.

1. Ações IO normalmente executam de maneira estritamente linear, onde as funções executadas por cada *Thread* são determinadas sequencialmente pelo programador; enquanto transações podem executar inúmeras vezes até que sejam validadas.
2. Transações em Haskell podem ser abortadas em qualquer ponto da execução. O Escalonador do *RunTime* eventualmente tenta verificar se a transação ainda é válida, ele o faz verificando se as TVars que a transação já leu ainda possuem o valor esperado, caso alguma delas já tenha sido alterada a transação é abortada prematuramente para evitar processamento desnecessário.

O STM Haskell oferece atualmente uma primitiva para compor ações IO às ações STM, o unsafeIOToSTM. Ao usar essa primitiva o programador deve ter em mente os Itens 1 e 2. Caso a ação IO tenha algum efeito colateral ele pode ser executado inúmeras vezes; e a transação pode ser abordada no meio das ações IO, limitado assim o uso de qualquer tipo de computação que adquira recursos que precisem ser liberados.

Propomos aqui então quatro novas primitivas que podem ser usadas para compor um conjunto maior funções IO aos blocos transacionais. **abort**, **performIO**, **addCommitHandler**, **addAbortHandler**, suas interfaces são apresentadas na Figura 2.

abort funciona de maneira muito semelhante ao **retry**, oferecido pelo STM Haskell, as diferenças entre as primitivas é que o **abort** não põe a transação para dormir, ela aborta e recomeça a transação imediatamente; além disso, caso seja chamado dentro de um **orElse**, o **abort** não altera o ramo atual de execução. Essa primitiva foi pensada para ser usada em conjunto com as ações IO e serve para recomeçar uma transação que, por motivos externos às variáveis transacionais, se tornou inválida, e.g., um recurso não pode ser adquirido e a transação não pode continuar.

performIO é uma primitiva para executar uma sequência de ações do tipo IO evitando que o sistema transacional aborte a transação antes dessas ações serem completadas. Diversas computações IO oferecidas pela linguagem fazem bloqueios de maneira

implícita, mas quando usadas dentro de transações podem ser canceladas no meio do processo resultado em *deadlocks* aleatórios. **performIO** pode então ser usada, por exemplo, para primitivas de *print* ou de leitura e escrita em arquivos de dentro de uma transação.

Por fim, **addCommitHandler** e **addAbortHandler** são um par de primitivas que, permitem ao programador adicionar tratamento aos possíveis cancelamentos e à eventual validação de uma transação. Estas primitivas podem ser usadas pelo programador para garantir propriedades de Atomicidade e Isolamento às operações IO compostas à transação. A primitiva **addAbortHandler** pode, por exemplo, apagar um arquivo criado, enquanto o **addAbortHandler** libera o *lock* que foi adquiridos para criação e escrita no arquivo.

Transactional Boosting [Herlihy and Koskinen 2008] e *Optimistic Transactional Boosting* [Hassan et al. 2014], oferecem metodologias sobre como esta composição entre IO e STM pode ser feita, em Haskell usando as primitivas apresentadas aqui ambas as técnicas podem ser implementadas.

4. Conclusões e Trabalhos Futuros

Apresentamos aqui algumas possíveis extensões para o STM Haskell, as primitivas e suas implementações são independentes entre si, mas juntas permitem aplicações de técnicas como *Transactional Boosting* em Haskell; trabalhos passados com *Transactional Boosting* já mostraram ser uma eficiente solução para o problema de falsos conflitos em STM [Herlihy and Koskinen 2008, Du Bois et al. 2014].

Apesar de possibilitar problemas como *deadlocks*, as primitivas apresentadas aqui, se utilizadas por programadores experientes, podem ser usadas para desenvolver bibliotecas transacionais de alto desempenho. Atualmente temos uma versão modificada do *RunTime* do *Glasgow Haskell Compiler* para dar suporte as primitivas apresentadas. Para trabalhos futuros visamos apresentar uma semântica formal para uso das primitivas, bem como apresentar exemplos de uso e testes de desempenho.

Referências

Du Bois, A., Pilla, M., and Duarte, R. (2014). Transactional boosting for haskell. In Quintão Pereira, F., editor, *Programming Languages*, volume 8771 of *Lecture Notes in Computer Science*, pages 145–159. Springer International Publishing.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2008). Composable memory transactions. *Commun. ACM*, 51(8):91–100.

Hassan, A., Palmieri, R., and Ravindran, B. (2014). Optimistic transactional boosting. In *ACM SIGPLAN Notices*, volume 49, pages 387–388. ACM.

Herlihy, M. and Koskinen, E. (2008). Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, New York, NY, USA. ACM.

Sulzmann, M., Lam, E. S., and Marlow, S. (2009). Comparing the performance of concurrent linked-list implementations in haskell. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 37–46. ACM.