Avaliando Transacional Boosting para Haskell

Jonathas A. O. Conceição[1], **André R. Du Bois**[1]

[1]Universidade Federal do Rio Grande do Sul (UFPel)
Pelotas – RS – Brazil

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

Resumo. *Transactional Boosting oferece uma maneira de transformar ações linearmente concorrentes em ações transacionalmente concorrentes, possibilitando assim sua composição com outras ações transacionais. Este trabalho disponibiliza uma extensão do STM Haskell para a aplicação de trasacional boosting às funções, possibilitando que ações linearmente concorrentes de alto desempenho sejam adicionadas aos seus blocos transacionais.*

1. Introdução

Memórias Transacionais, do inglês *Software Transacional Memory* (STM), é uma alternativa de alto nível à sincronização por *locks*. Nas Memórias Transacionais, todo acesso à memória compartilhada é agrupado como transações que podem executar de maneira concorrente. Se não houve conflito ao acesso da memória compartilhada ao fim da transação um *commit* é feito, tornando assim o conteúdo da memória público para o sistema. Caso ocorra algum conflito um *abort* é executado em todo bloco descartando qualquer alteração à memória. Diferente da sincronização por *locks*, transações podem ser facilmente compostas e são livres de *deadlocks* [Harris et al. 2008].

Os conflitos ocorrem quando duas ou mais ações de escrita ou leitura são feitas no mesmo endereço de memória. Entretanto essa forma de detecção conflitos pode gerar falsos conflitos levando a uma perda de desempenho. Por exemplo onde há duas transações diferentes modificando partes diferentes de uma lista encadeada [Herlihy and Koskinen 2008]. Embora essas ações não conflitem, o sistema detecta um problema já que uma transação modifica memória que outra transação lê, esse tipo de detecção de conflito pode gerar grande perda de performance quando se utiliza certos tipos de estruturas encadeadas. Por outro lado, utilizando sincronização por *locks* ou algoritmos *lock-free*, pode-se alcançar um alto nível de concorrência ao custo de complexidade no código. *Transactional boosting* oferece uma maneira de compor ações transacionais com estruturas de dados concorrentes oferecendo assim uma solução para esses falsos conflitos do STM.

STM Haskell é uma biblioteca adicionada ao *Glasgow Haskell Compiler* (GHC) em 2008 e provê primitivas para o uso de memórias transacionais em Haskell. Utilizando uma biblioteca de memórias transacionais própria, toda escrita em Haskell, uma implementação de *Transactional Boosting* para STM Haskell foi feita para testar as vantagens de desempenho dessa técnica em uma linguagem puramente funcional [Du Bois et al. 2014]. Devido aos resultados promissores apresentados por esse estudo, o objetivo do corrente trabalho é disponibilizar uma primitiva de alto nível na implementação padrão do STM Haskell, permitindo assim a aplicação de *Transactional Boosting* de maneira nativa no compilador GHC.

2. Transactional Boosting

Transactional Boosting é uma técnica utilizada para transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes [Herlihy and Koskinen 2008], permitindo assim sua utilização dentro dos blocos atômicos do STM. Nessa técnica, os objetos são tratados como caixas-pretas e ambos conflitos e registros à memória são tratados logo que são encontrados. O processo de adicionar *transactional boosting* ao compilador GHC envolve duas partes: a criação de uma interface Haskell em alto nível, que provê uma primitiva que possa ser usada pelo programador para realizar o *boost* de uma função Haskell, e uma camada principal no *RunTime System* (RTS), no baixo nível do GHC, escrita em C.

A interface de alto nível é uma primitiva que permite aplicar o *Transactional Boosting* a uma função linearmente concorrente. Para que seja criado uma versão *boosted* da função, é necessário que essa tenha um inverso, assim o sistema STM terá os mecanismos necessários em caso de *commit* e *abort*. A primitiva então envolve a função numa ação STM permitindo sua chamada dentro do bloco transacional. A primitiva de *boost* tem o seguinte protótipo e argumentos:

```
boost :: IO(Maybe a) -> (a -> IO ()) -> IO () -> STM a
```

Os argumentos são (1) uma ação (do tipo **IO**(**Maybe** a)), i.e., a função original que vai ser executada; (2) Uma ação de desfazer (do tipo a $->$ **IO**()), usada para reverter a ação executada em caso de *abort*; (3) Um *commit* (do tipo **IO**()), que é usado para tornar público a ação feita pela versão *boosted* da função original.

O RTS pode ser visto como três grandes subsistemas: Armazenamento, responsável pelo *layout* de memória e *garbage collector*; Execução, como o código Haskell é executado; O Escalonador, que gerencia threads e dá suporte *multicore*. As implementações desse trabalho aconteceram principalmente nas partes de Armazenamento e Execução. Quanto ao Armazenamento temos os *Heap Objects*, um aspecto central do armazenamento das funções em execução do RTS. Estas estruturas de dados seguem o *layout* da Figura 1. A primeira parte desses objetos é chamado de *Info Pointer*, que aponta para o *entry code* da estrutura; A segunda é o *Payload* onde ficam os dados carregados pela estrutura; O *entry code* é um código estático responsável por avaliar o *Heap Object*. Imediatamente antes do *entry code* encontra-se a *Info Table* do objeto, que contém informação utilizada pelo RTS. Na parte de Execução temos as chamadas *Primitive Operations*, estas são operações que por alguma impossibilidade ou por uma questão de desempenho são implementadas diretamente no RTS, e este é o caso da maioria das funções do STM.

Para dar suporte ao *boost* um novo *Heap Object* foi criado para armazenar a função a ser executada, o *StgBoostSTMFrame*. A estrutura adicionada possui referências em seus campos para as funções passadas para lidar com o *abort* e *commit*. A primitiva então irá executar a ação passada, armazenar as referências às funções auxiliares numa instância do *StgBoostSTMFrame* e por fim associa-lo à transação atual. Em caso de *commit* ou *abort* do bloco as funções associadas à primitiva de *boosting* são executadas.

3. Exemplo

Tomemos como exemplo para a aplicação de *Transacional Boosting* uma estrutura de conjuntos, como em [Herlihy and Koskinen 2008]. Uma implementação de conjuntos

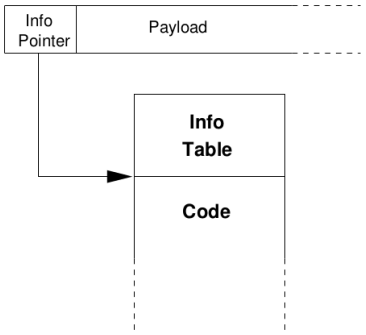

Figura 1. *Layout* de um *Heap Object* [Marlow and Peyton Jones 1998].

normalmente oferece três funções, *add*, *remove* e *contains*. Para a versão *boosted* de conjunto apresentada aqui foi utilizada uma lista encadeada *thread safe*. Na implementação é importante garantir que se uma transação está trabalhando num elemento, nenhuma outra transação vai utilizar o mesmo elemento. Isso pode ser alcançado utilizando uma tabela hash para associar um *lock* para cada elemento do conjunto. Vários modelos de tabela hash *thread safe* em Haskell foram apresentados em [Duarte et al. 2016], deles o algoritmo de *lock fino* foi utilizado para nossa implementação.

Para adicionar um elemento ao conjunto devemos adquirir o *lock* associado ao elemento e então inseri-lo na lista encadeada. Como a lista pode conter elementos duplicados é necessário verificar se o elemento já não está contido antes de inseri-lo. Se um elemento foi inserido e a transação abortar, o elemento deve ser removido e o *lock* liberado. Caso a transação termine sem conflitos é necessário apenas liberar o *lock*. Para remover um elemento é preciso adquirir o *lock* associado e então deletar o elemento da lista. Para reverter um *remove* o elemento deve ser devolvido ao conjunto e o *lock* liberado. O *commit* assim como antes precisa apenas liberar o *lock*. Por fim o *contains* precisa apenas realizar o *lock* no elemento e então conferir se ele está na lista. Tanto para o *commit* como para o *abort* o *contains* precisa apenas liberar o *lock* adquirido.

4. Experimentos e Resultados

Os experimentos foram executados numa máquina com processador Intel Core i7, frequência de 3.40GHz, 4 cores físicos e 4 lógicos, 8GiB de memória RAM. O sistema operacional foi um Ubuntu 14.04, a versão compilada do GHC foi a 7.10.3.

Para testar a biblioteca de conjuntos implementada usando *transactional boosting* foi utilizado uma lista de 2000 operações geradas aleatoriamente, bem como um conjunto inicial de 2000 elementos. Utilizando o mesmo número de threads e *cores* cada *thread* recebia uma lista de 2000 operações. Dois tipos de listas eram gerados em execuções separadas: Lista de leitura, contendo 40% de operações de `add` e `remove` mais 60% de operações de `contains`; Uma lista de escrita contendo 75% de operações de `add` e `remove` mais 25% de `contains`, permitindo uma visualização dos falsos conflitos na variação do desempenho. No gráfico é apresentado com tempo em escala logarítmica as médias de 30 execuções para os dados números de cores e threads. Pela Figura 2, pode-se observar que a utilização do *Transactional Boosting* resulta numa estrutura de conjuntos de desempenho bem superior em comparação à alternativa feita puramente com o STM Haskell.

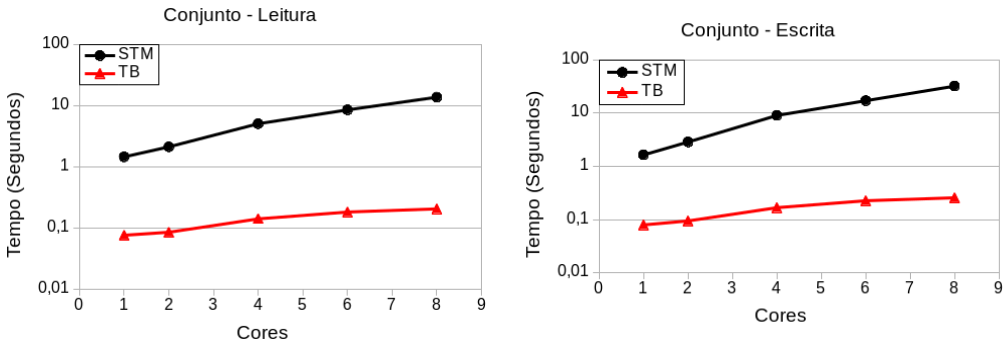

Figura 2. Tempo de execução de 2000 operações para cada core (A esquerda: 40% add e remove + 60% contains; A direita: 75% add e remove + 25% contains).

5. Conclusões e Trabalhos Futuros

A primitiva desenvolvida neste estudo, oferece uma maneira direta de compor objetos linearmente concorrentes com objetos transacionalmente concorrentes, de uma maneira que independe de qualquer tratamento do STM. *Transactional Boosting* é uma alternativa de baixo nível ao controle de concorrência e requer que o programador preserve as propriedades da estrutura transacional, além disso seu mau uso pode resultar em problemas como *deadlock*s. Entretanto se utilizada por programadores experientes pode ser usada para desenvolver bibliotecas concorrentes de alto desempenho capazes de serem adicionadas aos blocos transacionais oferecendo uma solução ao problema de falsos conflitos.

Para trabalhos futuros visamos um estudo da Semântica Formal e uma análise categórica da primitiva para uma melhor formalização da sua utilização em Haskell e a implementação de casos de teste mais robustos.

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*. Springer International Publishing.

Duarte, R. M., Du Bois, A. R., Pilla, M. L., and Reiser, R. H. S. (2016). Comparando o desempenho de implementações de tabelas hash concorrentes em haskell. *Revista de Informática Teórica e Aplicada*, 23(2).

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

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

Marlow, S. and Peyton Jones, S. (1998). The new ghc/hugs runtime system.