Implementação de Transactional Boosting no Glasgow Haskell Compiler

Jonathas A. O. Conceição[1], **André R. Du Bois**[1], **Renata H. S. Reiser**[1]

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

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

Resumo. *Este trabalho tem como objetivo disponibilizar uma nova primitiva de Transactional Boosting para o compilador e interpretador Glasgow Haskell Compiler visando resolver problemas de desempenho no uso de Memórias Transacionais. Para esta implementação são necessárias algumas alterações diretamente no RunTime System do Glasgow Haskell Compiler.*

1. Introdução

Um grande problema para a programação paralela atualmente é a crescente complexidade dos programas. Um dos causadores dessa complexidade é a utilização de *Locks* que são recursos necessários para o controle de concorrência, porém de grande complexidade. Software Transactional Memory (STM) trata-se de uma abstração para programação paralela que visa a simplificação do código de programas paralelos. Para isso o controle de concorrência, entre outras coisas, é feito todo pela máquina virtual simplificando assim a confecção de um programa além de evitar por completo problemas como o *DeadLock* [Harris et al. 2008].

Memórias Transacionais funcionam através da criação de blocos atômicos onde alterações de dados feitas às memórias são registradas para detecção de conflitos. Se num bloco atômico não houve conflitos um *commit* é feito, tornando assim o conteúdo deste endereço de memória público para o sistema. Caso ocorra algum conflito um *abort* é executado em todo bloco revertendo qualquer alteração ao conteúdo da memória. Os conflitos ocorrem quando duas ou mais ações de escrita ou leitura são feitas no mesmo endereço de memória, entretanto essa forma de encontrar conflitos pode, em alguns casos, gerar falsos conflitos levando a uma perda de desempenho. Para solucionar o problema de falsos conflitos na STM, técnicas de *transactional boosting* podem ser aplicadas para transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes [Herlihy and Koskinen 2008].

Haskell é uma linguagem de programação funcional de alto nível que possui várias técnicas de abstração para programação paralela. STM foi implementada ao *Glasgow Haskell Compiler* (GHC) em 2008 [Harris et al. 2008], desde então as pesquisas em memórias transacionais tem-se expandido bastante. O GHC é um compilador e interpretador, de código aberto, para a linguagem funcional Haskell. Ele funciona em várias plataformas como Windows, Mac, Linux, a maioria das distribuições de Unix e várias arquiteturas de computador. Quando um código compilado com o GHC é chamado ele é executado juntamente do *RunTime System* (RTS). O RTS trata-se de um sistema, escrito principalmente em C, que roda juntamente do programa dando suporte para várias

funcionalidades de baixo nível como *garbage collector*, suporte a transações, exceções, escalonamento, controle de concorrência, entre outras coisas.

O objetivo deste trabalho é adicionar uma primitiva já existente de *transactional boosting* [Du Bois et al. 2014] ao RTS do GHC. A primitiva em questão foi implementada em Haskell utilizando uma biblioteca de memórias transacionais própria [Du Bois 2011], e apresentou resultados promissores. Com a primitiva incorporada ao RTS o desempenho deve ser ainda maior.

2. Transactional Boost

Transactional Boost 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 da STM. Nas transações os objetos são tratados como caixas-pretas e ambos conflitos e registros à memória são tratados logo que são encontrados.

A função que esta sendo adicionada poderá ser utilizada para aplicar o *Transactional Boost* a uma dada ação. Para que seja criado uma versão *boosted* da ação é necessário que essa tenha um inverso, assim o sistema STM terá os mecanismos necessários em caso de *commit* e *abort*. A função de *boost* recebe como argumentos:

$boost :: IO(Maybe\ a) \text{ -> } (Maybe\ a \text{ -> } IO()) \text{ -> } IO() \text{ -> } STM\ a$

- Uma ação (do tipo $IO(Maybe\ a)$), a função original que vai ser executada.
- Um ação de desfazer (do tipo $Maybe\ a \text{ -> } IO()$), usada para reverter a ação executada em caso de *abort*.
- Um *commit* (do tipo $IO()$), que é usado para tornar público a ação feita pela versão *boosted* da função original.

boost retorna então uma nova ação STM que pode ser usada dentro dos blocos atômicos para executar a ação original.

Função Chamada	**Inversa**
generateID	noop

Comutatividade

x <- generateID ⇔ y <- generateID	x ≠ y
x < -generateID ⇔ y <- generateID	x = y

Figura 1. Especificação do gerador de IDs únicos [Du Bois et al. 2014]

Para exemplificar como função de *boost* funciona tomemos como exemplo um gerador de IDs únicos. Um gerador de IDs únicos em STM pode ser problemático, uma vez que é implementado como um contador compartilhado que é incrementado a cada contagem. Como diferentes transações estão acessando e incrementando um mesmo endereço de memória, o contador, implementações de memória transacional detectaria conflitos de escrita e leitura e um *abort* seria chamado em pelo menos uma das transações. Entretanto esses não necessariamente são conflitos, desde que todos os retornos sejam diferentes, não é necessário que os IDs sejam totalmente sequenciais.

Uma forma simples e eficiente para fazer este gerador de IDs únicos seria utilizando uma operação de *Compare-and-Swap*. Haskell provê uma abstração chamada *IORef* para representar locais de memórias mutáveis. A biblioteca permite ao programador realizar operações de comparação e *swap* com *IORef* ao nível de máquina. Assim um gerador de IDs únicos pode ser gerado assim:

```
type IDGer = IORef Int

newID :: IO IDGer
newID = newIORef 0

generateID :: IDGer -> IO Int
generateID idger = do
  v <- readIORef idger
  ok <- atomCAS idger v (v+1)
  if ok then return(v+1) else generateID idger
```

Para o *Transactional Boosting* o gerador terá que seguir as especificações da figura 1. Usando a primitiva de *boost* a função pode ser implementada assim:

```
generateIDTB :: IDGer -> STM Int
generateIDTB idger = boost ac undo commit
  where
  ac = do
    newID <- generateID idger
    return(Just newID)
  undo _ = return()
  commit = return()
```

Agora o generateIDTB é uma operação sobre STM, que quando executada chama ac onde por sua vez simplesmente utiliza a versão CAS do gerador para incrementar o contador *IORef*. Neste exemplo se a houver um *commit* ou *abort* nada precisa ser feito, por isso as ações estão vazias.

3. Implementação

Para a implementação da primitiva de *boost* uma expansão da estrutura de dados do STM é necessária, permitindo assim que uma transação possa carregar a ação original, bem como o meio necessário para realizar o *undo* e o *commit* desta ação. No GHC essas estruturas são chamadas de *Heap Objects*, todo *Heap Object* segue o *layout* na figura 2. A primeira parte desses objetos é chamado de *Info Pointer*, que aponta para o *Info Table* da estrutura, a segunda é o *Payload* onde ficam os dados carregados pelo *Heap Object* [Marlow and Jones 1998]. A *Info Table* contém informações sobre o tipo de estrutura, informação essa utilizada principalmente pelo *garbage collector*, e também o código responsável por avaliar aquela estrutura.

O *Heap Object* que precisa ser alterado é o *StgAtomicallyFrame*, que é a estrutura posta na pilha de execução do Haskell quando a função *atomically* é chamada em alto nível. Nele é preciso estender o *Payload* para que carregue as ações extras e que lhe seja alterado o *Info Table*, para que utilize as ações extras de *commit* e *abort*.

Com o *StgAtomicallyFrame* estendido a primitiva de *boost* será responsável por colocar na pilha de execução do bloco atômico a ação original bem como seu *commit* e *abort*. Este tipo de função é chamada de *Primitive Operations* (PrimOps), estas são operações que por alguma impossibilidade ou por uma questão de desempenho são implementadas diretamente em C no RTS, e este é o caso da maioria das funções da STM.

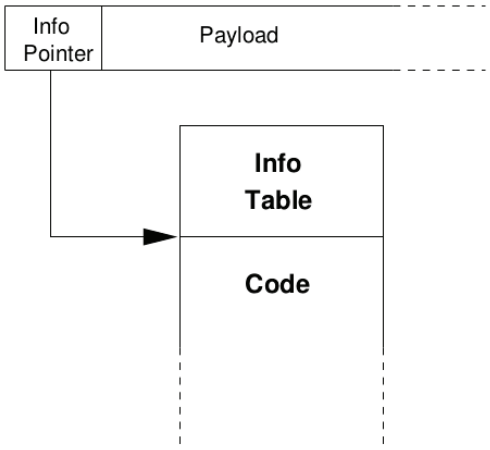

Figura 2. *Layout* de um *Heap Object* [Marlow and Jones 1998]

4. Estado Atual e Passos Futuros

Todo o procedimento para que a função proposta pudesse ser implementada, e o estudo do funcionamento das funções do STM no alto nível em Haskell e no baixo nível do RTS foram as primeiras etapas desse trabalho. Quanto a implementação já temos a primitiva implementada ao compilador e é possível executar uma ação externa agora dentro de um bloco STM. Atualmente estamos trabalhando na extensão das funções de *commit* e *abort*.

Os passos futuros deste trabalho são a realização de testes de desempenho da nova primitiva de *boost*, bem como a comparação com a versão original implementada puramente em Haskell.

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.

Du Bois, A. R. (2011). *An Implementation of Composable Memory Transactions in Haskell*, pages 34–50. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

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.

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