IMPLEMENTAÇÃO DE TRANSACIONAL BOOSTING AO GLASGOW HASKELL COMPILER

JONATHAS AUGUSTO DE OLIVEIRA CONCEIÇÃO[1]; ANDRÉ RAUBER DU BOIS[2]; RENATA HAX SANDER REISER[3]

1 Universidade Federal de Pelotas – adoliv@inf.ufpel.edu.br;

2 Universidade Federal de Pelotas – dubois@inf.ufpel.edu.br;

3 Universidade Federal de Pelotas – reiser@inf.ufpel.edu.br

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*.

Memórias Transacionais trabalham registrando as alterações de dados feitos às memórias e fazendo uma comparação entre os dados registrados para detectar conflitos, se numa ação não houve conflitos um *commit* é feito, tornando assim o conteúdo deste endereço de memória público. Caso ocorra algum conflito um *abort* é executado revertendo qualquer alteração ao conteúdo da memória. Os conflitos ocorrem quando dois ou mais ações de escrita e 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 no STM, técnicas de *transactional boosting* podem ser aplicadas para transformar objetos linearmente concorrentes em objetos transaccionalmente concorrentes[Du Bois et al. 2014]. Nas transações os objetos são tratados como caixas-pretas e tanto conflitos quanto os registros à memória são tratados logo que são encontrados.

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 2006[Harris et al. 2006], 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, e a maioria das distribuições de Unix, e várias arquiteturas de computador. Quando um código compilado do GHC é chamado ele é executado juntamente do *RunTime System*. 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 *garbege colector*, suporte a transações, exceções, escalonamento, controle de concorrência, entre outras coisas.

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

2. METODOLOGIA

A função que esta sendo adicionada se trata de um *transactional boosting*, onde a operação a ser executada precisa ter um inverso. Com isso o sistema STM terá os meios necessários de tratar os dados em casos de *commit ou abort.* A tal primitiva é usada para criar uma nova versão de uma dada função Haskell, a função passada é então englobada em uma ação STM possibilitado sua chamada dentro de transações e também reverter suas ações em caso de um *abort*. Ela recebe como argumentos:

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

1. Uma ação(de tipo *IO(Maybe a)*), a função que será executada.
2. Uma ação de desfazer(de tipo *(Maybe a) -> IO()*), usada para reverter o que foi feito em caso ode um *abort*.
3. Uma ação de *commit*(de tipo *IO()*), usada para tornar público a ação feita bela versão *boosted* da função.

Para exemplificar como a 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 são necessariamente conflitos, desde que todos os retornos sejam diferentes, não é necessário que os IDs sigam uma ordem em específico.

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].

Uma forma simples e eficiente para fazer este gerador de IDS únicos seria utilizando uma operação de *Compare-and-Swap*. Haskell prove 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[Du Bois eal. 2014]:

```
type IDGer = IORef Int

newID :: IO IDGer
newID = newIORef 0

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

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

```
generateIDT B :: IDGer -> STM Int
generateIDT B idger = boost ac undo commit
  where
  ac = do
    Id <- U niqueIDCAS.generateID idger
    return (Just id)
  undo = return ()
  commit = return ()
```

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

3. RESULTADOS E DISCUSSÃO

O tipo de primitiva a ser implementada 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[GHC *Developer* Wiki et al. 2016], e este é o caso da maioria das funções de STM. Para a implementação da primitiva de *boost* uma expansão das estruturas de dados do STM é necessária, permitindo assim que uma transação possa carregar as funções que deverão ser executadas, bem como as ações necessárias para realizar o *undo* e o *commit* destas respectivas ações. O resultado obtido até o momento é este procedimento para adicionar esta nova primitiva.

Uma vez que as estruturas tenham sido devidamente incrementadas é necessário então implementar a função em C que ira tratar estes dados, a função em

Haskell que será utilizada pelo programador em alto nível, e implementada também a devida ligação entre a função de alto e baixo nível e suas respectivas PrimOps.

4. CONCLUSÃO

O procedimento para adesão de uma PrimOps, funcionamento da função em alto nível e funções do STM bem como suas respectivas estruturas de dados já foram entendidas e listadas para a continuidade do trabalho.

Para os passos futuros é necessário implementar a função em baixo nível e realizar testes com as estruturas de dados estendidas nos diferentes casos de funcionamento, isto é, em casos de *commit* e *abort*. Uma vez com a função devidamente adicionada ao RTS, serão feitos testes de desempenho e comparações com a primitiva puramente em Haskell, bem como a implementações que não utilizem *transactional boosting*.

4. REFERÊNCIAS BIBLIOGRÁFICAS

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

GHC Developer Wiki, (2016), Disponível em: <https://ghc.haskell.org/trac/ghc/wiki>. Acesso em: 21 de jul. 2016.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2006). Composable memory tranactions. *Commun*. *ACM.*