Transactional Boosting no Glasgow Haskell Compiler

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

[1] Programa de Pós Graduação em Computação/CDTec
Universidade Federal de Pelotas (UFPel)

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

[2]Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Ouro Preto (UFOP)

rodrigo@decsi.ufop.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 assim a perda de desempenho de algumas aplicações. O objetivo deste trabalho é apresentar uma extensão do STM Haskell, bem como as modificações necessárias ao Run-Time System do compilador, para permitir o desenvolvimento de aplicações que utilizam Transactional Boosting, e assim apresentar a viabilidade desta técnica em Haskell.*

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 do endereço de memória público para o sistema. Caso ocorra algum conflito um *abort* é executado descartando qualquer alteração ao conteúdo da memória. Diferente da sincronização por *locks*, transações podem ser facilmente compostas e são livres de *deadlocks* [Harris et al. 2008].

Memórias Transacionais funcionam através da criação de blocos atômicos onde alterações de dados são registradas para detecção de conflitos. Um conflito ocorre quando duas ou mais transações acessão 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, Herlihy and Koskinen 2008]. Embora essas ações não conflitem, o sistema detecta um conflito já que 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

*O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES/Brasil 88882.151433/2017-01 e da Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul - FAPERGS

alcançar um alto nível de concorrência ao custo de complexidade no código. *Transactional Boosting* [Herlihy and Koskinen 2008] pode ser aplicado para transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes, oferecendo assim uma solução para esses falsos conflitos.

O objetivo deste trabalho é desenvolver uma biblioteca para *Transactional Boosting* em STM Haskell, permitindo assim a aplicação desta técnica de maneira nativa no *Glasgow Haskell Compiler* (GHC).

As contribuições deste artigo são:

- Desenvolvimento de uma nova primitiva que possibilita a aplicação da técnica de *Transactional Boosting* na biblioteca STM Haskell;
- Extensão do *RunTime System* do GHC para dar suporte a tal primitiva;
- Implementação de casos de uso da primitiva apresentada e análise do desempenho da mesma.

O artigo é organizado da seguinte forma: A Seção 2 descreve a biblioteca STM Haskell; na Seção 3 o conceito de *Transactional Boosting* é apresentado juntamente com a nova primitiva desenvolvida; a Seção 4 mostra três exemplos de aplicação da técnica; na Seção 5 as modificações necessárias no *RunTime System* do Haskell são descritas; a Seção 6 apresenta os resultados de alguns experimentos feitos para avaliar nossa implementação; na Seção 7 trabalhos relacionados são discutidos; por fim, na Seção 8 conclusões e trabalhos futuros são relatados.

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 como valores de primeira ordem. 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*, garantindo assim que a atomicidade (o efeito da transação se torna visível todo 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 assim realizar o *commit* ou *abort* da transação.

Existem três primitivas para o uso de *TVars*: (1) **newTVar** é utilizada para criar uma *TVar* que pode conter valores de um tipo a qualquer; (2) **readTVar** retorna o conteúdo de uma *TVar*; e (3) **writeTVar** escreve um valor em uma *TVar*. 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

```
-- 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.

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. Transactional Boosting em Haskell

Transactional Boosting [Herlihy and Koskinen 2008] é uma técnica proposta inicialmente no contexto de Programação Orientada a Objetos, como uma maneira de transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes, permitindo assim sua utilização dentro de blocos atômicos. Nas transações, esses objetos são tratados como caixas-pretas e *handlers* para lidar com efetivações e cancelamentos são registrados no sistema transacional. Dessa forma, para que seja criado uma versão *boosted* de uma ação realizada em um objeto, é necessário que essa tenha um inverso, para que o sistema STM possa lidar com o cancelamento de uma transação. Além disso, apenas operações comutativas podem ser executadas concorrentemente.

Neste trabalho, foi adicionada uma nova primitiva ao STM Haskell que permite a chamada de funções não transacionais escritas em Haskell dentro de uma ação STM, além disso a primitiva permite a adição de ações que serão executadas no caso de cancelamento ou no caso de efetivação da transação. A função de *boost* tem o seguinte protótipo:

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

Os argumentos são:

- Uma ação (do tipo **IO**(**Maybe** a)), que é a função original que vai ser executada. Quando a ação original é executada ela pode retornar um resultado do tipo a, ou se por algum motivo ela não pôde ser completada, e.g. um *lock* interno não pode ser adquirido, a função deve retornar **Nothing**.
- Uma ação de cancelamento (do tipo a **-> IO**()), usada para reverter a ação executada em caso de *abort*. Quando chamada, esta deve receber o valor retornado pela ação original para que seu efeito seja revertido.
- Uma ação de *commit* (do tipo **IO**()), que é usado para tornar público a ação feita pela versão *boosted* da função original.

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

Function Call	Inverse
generateID	noop

Commutativity

x <- generateID $\Leftrightarrow$ y <- generateID x $\neq$ y

x <- generateID $\not\Leftrightarrow$ y <- generateID x = y

Figura 2. Especificação do gerador de identificadores únicos.

4. Exemplos

Nesta Seção é apresentada a implementação em Haskell de três exemplos clássicos de *Transactional Boosting* [Herlihy and Koskinen 2008]. Estes exemplos são um gerador de identificadores únicos (Seção 4.1); Um Produtor e Consumidor (Seção 4.2) e Conjuntos (Seção 4.3).

4.1. Gerador de Identificadores Únicos

Um gerador de IDs únicos em STM pode ser problemático, sua implementação mais comum seria utilizando um contador compartilhado que é incrementado a cada chamada. Como diferentes transações estão acessando e incrementando um mesmo endereço de memória, o contador, o sistema de memória transacional da linguagem detectaria vários conflitos. Entretanto esses não são necessariamente conflitos. Desde que todos os retornos sejam diferentes não é necessário que os IDs sejam totalmente sequenciais.

Uma simples implementação *thread-safe* para fazer o gerador de IDs únicos seria utilizando uma instrução de *Compare-and-Swap* (CAS) disponível em diversas arquiteturas *multicore*. Haskell provê uma abstração chamada **IORef** para representar locais de memórias mutáveis. A biblioteca *atomic-primops* [Newton 2016] permite ao programador realizar operações CAS implementadas em Hardware com **IORef**s. Assim um gerador de IDs únicos pode ser implementado da seguinte forma:

```
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
```

A função generateID recebe como argumento a referência ao contador compartilhado e aplica o CAS até conseguir incrementá-lo. Usando *Transactional Boosting* o gerador terá que seguir a especificação da Figura 2. Usando a primitiva `boost` uma versão transacional do gerador de IDs pode ser implementada da seguinte forma:

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

Function Call	Inverse
offer buff x	tryPopL buff
x <- take buff	pushR buff x

Commutativity

offer buff x $\Leftrightarrow$ y <- take buff, buffer non-empty

offer buff x $\not\Leftrightarrow$ y <- take buff, otherwise

Figura 3. Especificação do Produtor-Consumidor.

```
commit = return ()
```

O novo gerador de IDs é agora uma operação transacional e pode ser chamado livremente dentro de transações. Quando executado, este simplesmente utiliza a versão CAS do gerador para incrementar o contador. Em caso de *commit* ou *abort*, nada precisa ser feito (ver Figura 2), por isso as ações estão vazias.

4.2. Produtor e Consumidor

O Produtor-Consumidor se trata de um problema onde há uma sequência de dados a serem processados por *threads* que se comunicam através de um *buffer* compartilhado.

O *buffer* deve oferecer então duas funções: offer, utilizada para adicionar um valor ao *buffer* e take para retirar um valor do *buffer*. Para implementar o *buffer* utilizando *Transactional Boosting* foi utilizado um *double-ended queue thread-safe*, seguindo a especificação da Figura 3.

```
offer :: TBBuffer a -> a -> STM ()
offer (TBBuffer c ioref) v = boost ac undo commit
  where
  ac = do pushL c v
    return (Just ())
  undo _ = do mv <- tryPopL c
    case mv of
      Just v -> return ()
  commit = do v <- readIORef ioref
    ok <- atomCAS ioref v (v+1)
    if ok then return () else commit
```

O **TBBuffer** se trata de um *dequeue* e um **IORef** que conta o tamanho do *buffer*. A função offer usa **pushL** para adicionar um valor ao *queue*. Se a transação abortar o valor colocado no *buffer* deve ser removido utilizando o **tryPopL**. Em caso de *commit* só resta atualizar o tamanho da fila para tornar o novo elemento visível à *thread* consumidora. A função take utiliza **tryPopR** para consumir os dados do *buffer*:

```
take :: TBBuffer a -> STM a
readFifo (TBBuffer c ioref) = atomically (boost ac undo commit)
  where
  ac = do
    size<-readIORef ioref
    if size == 0 then return Nothing
      else do mv <- tryPopR c
```

```
      case mv of
        Just v -> return (Just v)
  undo v = pushR c v
  commit = do v <- readIORef ioref
    ok <- atomCAS ioref v (v-1)
    if ok then return () else commit
```

Se não há elementos o suficiente no *buffer* o **Nothing** resultante irá disparar um *abort* na transação. Caso contrário a função irá decrementar o contador do *buffer* e consumir o valor. A ação de *undo* deverá incrementar o contador e devolver o valor ao *buffer*.

4.3. Conjuntos

Uma implementação de conjuntos normalmente oferece três funções, *add*, *remove* e *contains*.

Function Call	**Inverse**
add set x / False	noop
add set x / True	remove set x / True
remove set x / False	noop
remove set x / True	add set x / True
contains set x / _	noop

Commutativity

add set x / _ $\Leftrightarrow$ add set y / _, x $\neq$ y

remove set x / _ $\Leftrightarrow$ remove set y / _, x $\neq$ y

add set x / _ $\Leftrightarrow$ remove set y / _, x $\neq$ y

add set x / False $\Leftrightarrow$ remove set x / False $\Leftrightarrow$ contains set x / _

Figura 4. Especificação da estrutura de conjuntos.

Para a versão *boosted* de conjuntos apresentada aqui foi utilizada uma lista encadeada *thread safe*, descrita em [Sulzmann et al. 2009]. Na implementação é importante garantir que se uma transação está trabalhando num elemento, nenhuma outra transação vai utilizar o mesmo elemento (vide Figura 4). Isso pode ser alcançado utilizando *key-based locking*, implementado 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], dentre eles, o algoritimo de *lock fino* foi utilizado para esta implementação de conjuntos.

Para adicionar um elemento ao conjunto deve-se adquirir o *lock* associado ao elemento e então inserí-lo na lista encadeada. Como a lista pode conter elementos duplicados é necessário verificar se o elemento já não está contido antes de inserí-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*:

```
add :: IntSet -> Int -> STM Bool
add (Set alock list) element = boost ac undo commit
  where
    ac = do ok <- AbstractLock.lock alock element
      case ok of
        True -> do found <- List.find list element
```

```
            if found then return (Just False)
              else do List.addToTail list element
                      return (Just True)
          False -> do return Nothing
      undo v = do case v of
          True -> do List.delete list element
                     AbstractLock.unlock alock element
                     return()
          False -> AbstractLock.unlock alock element >> return()
      commit = AbstractLock.unlock alock element >> return ()
```

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*.

```
remove :: IntSet -> Int -> STM Bool
remove (Set alock list) element = boost ac undo commit
  where
    ac = do ok <- lock alock element
      case ok of
        True -> do v <- List.delete list element
                   return (Just v)
        False -> return Nothing
    undo ok = do case ok of
        True -> do List.addToTail list element
                   AbstractLock.unlock alock element
                   return ()
        False-> AbstractLock.unlock alock element >> return ()
    commit = AbstractLock.unlock alock element >> return ()
```

Por fim o *contains* precisa apenas adquirir o *lock* do 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. Por sua simplicidade o *contains* teve o seu código omitido neste artigo.

5. Implementação

Para a proposta de *Transactional Boosting* em [Herlihy and Koskinen 2008] é necessário que o sistema de memória transacional da linguagem permita a definição de *handlers* para quando uma transação realizar o *abort* ou *commit*, entretanto essa não é uma funcionalidade oferecida pelo *STM Haskell*. Para este trabalho uma extensão do *STM Haskell* e do RTS do GHC foi feita para que o compilador oferecesse suporte nativo para *Transactional Boosting*.

A implementação da primitiva se divide em duas partes. Uma interface Haskell em alto nível, que provê a função `boost`, e uma camada principal no *RunTime System* (RTS) escrita em C e em Cmm (um *assembly* de alto nível utilizado na implementação do RTS [Ramsey et al. 2005]).

5.1. Implementação da primitiva `boost`

A implementação em Haskell da primitiva é responsável por lidar com o sistema de tipos da linguagem, executar a ação original e passar dados do alto nível para o baixo nível:

```
boost :: IO(Maybe a) -> (a -> IO ()) -> IO () -> STM a
boost iomac undo commit = STM (\s -> expression s)
  where
    expression = unIO $ iomac >>= \mac -> case mac of
      Just ac -> IO (\s -> (boost# (return ac) undo commit) s)
      Nothing -> IO (\s -> (abort# s))
```

A função começa executando a ação à ser aplicado o *boost* (iomac). Se o resultado for um **Just** ac a ação foi bem sucedida, neste caso a primitiva **boost#** é chamada para registrar no RTS ações para o *commit* e *undo* da ação executada. Caso o resultado seja um **Nothing** a ação não pode ser executada agora e a transação deve recomeçar. Ambas as funções **boost#** e **abort#** são *Primitive Operations* e serão explicadas em mais detalhe na Seção 5.2, o caractere # ao fim do nome da função indica uma chamada ao RTS no GHC. A seguir é apresentado o protótipo das *Primitive Operations* em Haskell:

```
boost# :: IO a -> (a -> IO ()) -> IO () -> STM a
abort# :: STM a
```

5.2. Implementação no RunTime System

O *RunTime System* (RTS) é uma biblioteca escrita predominantemente em C que é ligada a qualquer programa em Haskell compilado com o GHC. Ele provê suporte e infraestrutura para funcionalidades como *garbage collector*, transações, exceções, escalonamento, controle de concorrência, entre outras. O RTS pode ser visto como três grandes subsistemas: Armazenamento, responsável pelo layout de memória e *garbage collector*; Execução, responsável pela execução de código Haskell; O Escalonador, que gerencia threads e dá suporte *multicore*. As modificações feitas no RTS para este trabalho aconteceram principalmente nas partes de Armazenamento e Execução.

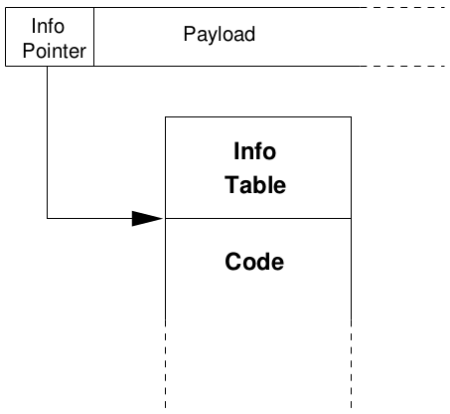

Figura 5. *Layout* de um *Heap Object* [Marlow and Peyton Jones 1998].

Heap Objects, são um aspecto central do armazenamento das funções em execução no RTS. Estas são estruturas de dados escritas em C e todo *Heap Object* segue o *layout* da Figura 5. 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 Peyton 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 o objeto.

Um tipo importante de *Heap Object* é o *Thread State Object* (TSO). Ele representa o estado atual de uma thread incluindo seu *stack* de execução. O *stack* consiste de uma sequência de *stack frames* onde cada *frame* corresponde a um *Heap Object*.

Para dar suporte à *Transactional Boosting* um novo *Heap Object* (StgBoostSTMFrame) foi criado para armazenar o resultado da função de **boost**, bem como a ação de *abort*. O *Heap Object* criado possui duas referências em seus campos: (1) ação de *abort*; (2) resultado da ação de boost executada, que é usado como argumento da ação de *abort*. O objeto tem o código a seguir:

```
typedef struct {
  StgHeader    header;
  StgClosure *tbAbort;
  StgClosure *tbResult;
} StgBoostSTMFrame;
```

Na parte de Execução temos as chamadas *Primitive Operations* (*PrimOps*), estas são operações que por alguma impossibilidade ou por uma questão de desempenho são implementadas diretamente no RTS, e este é o contexto onde a maioria das funções do STM Haskell se encontram. As *PrimOps* são escritas em Cmm, um assembly de alto nível que é compilado dentro do próprio GHC. Apenas códigos em Cmm pode manipular diretamente o *stack* do TSO ao qual pertencem e invocar novas execuções de transações.

A primitiva **boost#** é responsável por instanciar um novo *StgBoostSTMFrame* com suas respectivas referências; o *StgBoostSTMFrame* é então colocado na *stack* do TSO atual. As ações de commit, são armazenadas em uma lista associada ao TSO, juntamente com o estado da transação (conjuntos de leitura e escrita). Ao fim da transação, se ela for bem-sucedida, as ações de commit são executadas. O **abort#** por sua vez, quando chamado percorre o *stack* do TSO realizando as ações de *abort* necessárias para cada tipo de *stack frame* encontrado e então recomeça a transação. Três tipos de *frames* podem ser encontrados dentro dos blocos transacionais, são eles:

- StgCatchRetryFrame, o *frame* associado ao **orElse**. Neste caso somente as ações de abort do primeiro ramo serão executadas e a execução continua no segundo ramo do `orElse`
- StgBoostSTMFrame, o *frame* associado à ação de *Boost*. Quando encontrado sua ação de *abort* associada é executada.
- StgAtomicallyFrame, o *frame* associado ao **atomically**. Encontrá-lo no *stack* indica que a se retornou ao início da transação. Neste caso o registro de acesso à memória atual é descartado e a transação reexecutada do início.

A primitiva de **retry#** (*PrimOp* utilizado pelo **retry**) funciona de maneira semelhante ao **abort#**. Quando executado ele também percorre o *stack* associado a seu TSO para tratar dos casos específicos até chegar no StgAtomicallyFrame.

6. Resultados e Experimentos

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 usado foi o Ubuntu 14.04, a versão do GHC foi a 7.10.3.

A Figura 6 apresenta os resultados para três implementações do gerador de identificadores únicos: IDSTM que é uma implementação utilizando o STM Haskell puro; IDCAS que utiliza CAS para incrementar o contador; IDTB que que utiliza *Transactional Boosting*. Foram executados 10 milhões de chamadas de incremento ao identificador, dividindo igualmente as operações entre as threads disponíveis. Trinta execuções foram feitas, e o gráfico apresenta as médias de cada execução com o tempo em escala logarítmica. Como esperado, a versão usando *Transactional Boosting* adiciona um certo *overhead* em comparação com a versão usando apenas CAS, porém se comparado com o desempenho da ação transacional pura, a versão *boosted* se mostra uma alternativa muito mais eficiente.

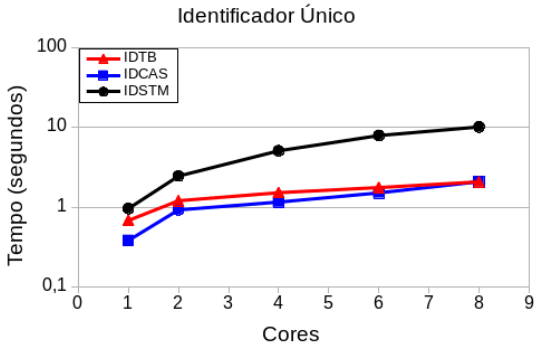

Figura 6. Tempo de execução do Identificar Único.

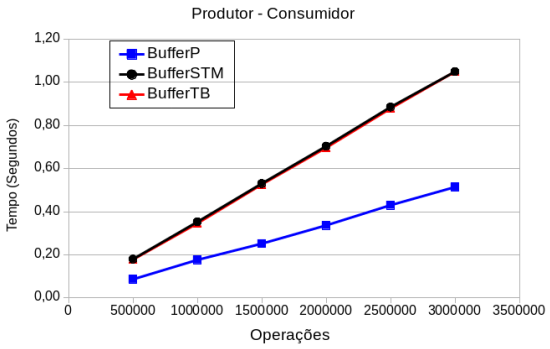

Figura 7. Tempo de execução do Produtor-Consumidor.

A Figura 7 por sua vez apresenta os experimentos com o Produtor e Consumidor. Nas execuções foram criadas duas threads, um produtor e um consumidor, para compartilhar o *buffer*. Três implementações são apresentadas: BufferSTM que utiliza apenas a biblioteca STM Haskell; BufferP é a implementação do *dequeue thread safe* e linearmente concorrente; BufferTB é a versão usando *Transactional Boosting*. Neste exemplo o *overhead* existente sobre a aplicação do *boost* torna o BufferTB equivalente ao BufferSTM em termos de tempo de execução. A biblioteca TChan STM, utilizada na versão puramente transacional, utiliza listas duplamente encadeadas, assim, com apenas um produtor e um consumidor, poucos falsos conflitos ocorrem; neste caso pode-se observar que o *overhead* da utilização do *boost* é equivalente ao *overhead* da própria camada STM, sendo o *Transactional Boosting* levemente mais rápido.

Para avaliação de um exemplo com grande quantidade de falsos conflitos, assim como do uso do **retry** e **orElse**, foram implementados dois exemplos que utilizam a estrutura de conjuntos descrita na Seção 4.3. Para ambos os exemplos, listas de 2000 operações foram geradas aleatoriamente para serem aplicadas sobre conjuntos inicializados com 2000 elementos.

No primeiro experimento, dois conjuntos diferentes foram inicializados, e todas as *threads* recebiam referências para ambos os conjuntos. Cada *thread* tenta realizar a operação no primeiro conjunto, caso a operação falhasse, e.g., tentar remover um elemento que não esta no conjunto, um **retry** era chamado e a *thread* tentava a mesma operação no segundo conjunto; se a operação no segundo conjunto também chamasse um **retry** todo o bloco abortava e a *thread* procurava uma nova operação. O gráfico da Figura 8 mostra a

média de 30 execuções em escala logarítmica.

No segundo experimento, 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; Lista de Escrita contendo 75% de operações de add e remove mais 25% de contains. No gráfico da Figura 9 mostra a média de 30 execuções em escala logarítmica.

Pelas Figuras 8 e 9 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. Isso acontece pois a detecção de conflitos feito pelo sistema de Memórias Transacionais gera uma excessiva quantidade de falsos conflitos lidando com estruturas encadeadas.

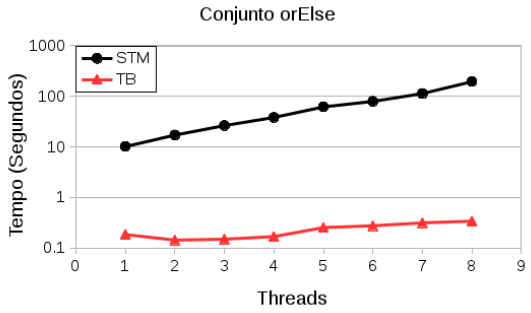

Figura 8. Tempo médio de execução de 2000 operações.

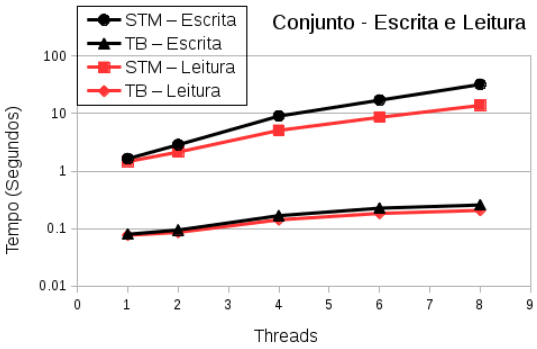

Figura 9. Tempo médio de execução de 2000 operações.

7. Trabalhos Relacionados

Outros trabalhos oferecem extensões para o STM Haskell que abordam o problema de perda de desempenho em casos de falsos conflitos. O **unreadTVar** [Sönmez et al. 2007] pode ser utilizado para melhorar o tempo de execução e uso de memória quando se atravessa uma estrutura transacional de dados encadeados. A primitiva pode ser usada para retirar do registro de leitura valores que podem gerar falsos conflitos. Similarmente em [Sulzmann et al. 2009] a primitiva **readTVarIO** é utilizada como uma maneira de ler *TVars* para percorrer uma estrutura encadeada sem que ela seja adicionada ao registro de leitura da transação.

Em [Harris 2007] os autores apresentam *Abstract Nested Transaction*, que também oferece uma solução para problemas de desempenho relacionados aos falsos conflitos em STM. Nele acessos à memória são armazenado num registro próprio para a detecção de conflitos. E em caso de conflito, inicialmente apenas a expressão que acessa dados conflitantes é reavaliada para checar se o resultado da expressão foi alterado desde a sua primeira execução, e apenas se houver uma alteração toda a transação é reexecutada.

Estes métodos acima apresentam maneiras de implementar novos tipos de dados com o objetivo de evitar falsos conflitos ou antecipar sua detecção e tratamento. O *Transactional Boosting*, por outro lado, permite a composição de estruturas linearmente concorrentes eficientes já existentes às ações transacionais. A primitiva de *Transactional Boosting* aqui apresentada foi inicialmente proposta em [Du Bois et al. 2014]. Porém a

implementação apresentada no artigo era toda feita em Haskell usando um sistema transacional também totalmente escrito em Haskell. Os resultados positivos apresentados por esse protótipo levaram os autores a buscar a implementação com suporte do RTS apresentada aqui.

8. Conclusões e Trabalhos Futuros

Neste artigo foi apresentada uma extensão do STM Haskell e do RTS do GHC que permite a aplicação da técnica de *Transactional Boosting* em programas escritos na linguagem Haskell. Além disso, foram apresentados três estudos de caso e seus respectivos desempenhos que validam a extensão apresentada.

O sistema aqui descrito oferece uma maneira simples de transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes que podem ser executados em transações. *Transactional Boosting* é uma técnica de baixo nível para controle de concorrência e pode resultar em problemas como *deadlocks*. Entretanto se empregada por programadores experientes pode ser usada no desenvolver bibliotecas transacionais de alto desempenho. Para trabalhos futuros visamos apresentar uma semântica formal para uso da primitiva.

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.

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):193–209.

Harris, T. (2007). Abstract nested transactions. In *TRANSACT 2007*.

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

Newton, R. R. (2016). Atomic-primops: A safe approach to cas and other atomic ops in haskell.

Ramsey, N., Jones, S. P., and Lindig, C. (2005). *The C– Language Specification Version 2.0*.

Sönmez, N., Perfumo, C., Stipic, S., Cristal, A., Unsal, O. S., and Valero, M. (2007). unreadtvar: Extending haskell software transactional memory for performance. *Trends in Functional Programming*, 8:89–114.

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.