Transactional Boosting no Glasgow Haskell Compiler

JONATHAS AUGUSTO DE OLIVEIRA CONCEIÇÃO[1]; ANDRÉ RAUBER DU BOIS[2]

[1]*Universidade Federal de Pelotas – jadoliveira@inf.ufpel.edu.br*
[2]*Universidade Federal de Pelotas – dubois@inf.ufpel.edu.br*

1. INTRODUÇÃO

Software Transactional Memory (STM) é uma alternativa de alto nível ao sistema de sincronização por *locks*. Nele 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 dos endereços de memória acessados pela transação públicos para o sistema. Caso ocorra algum conflito um *abort* é executado em todo bloco descartando qualquer alteração ao conteúdo da memória. Diferente da sincronização por locks, transações podem ser facilmente compostas e são livres *dedeadlocks* (Harris et al. 2008).

Memórias Transacionais funcionam através da criação de blocos atômicos onde alterações de dados são registradas para detecção de conflitos. 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, em alguns casos, gerar falsos conflitos levando a uma perda de desempenho. Um bom exemplo seria onde há duas transações diferentes modificando partes diferentes de uma lista encadeada. Embora essas ações não conflitem, o sistema detecta um conflito já que uma transação modifica memória que outra transação leu. 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* pode ser aplicado para transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes (Herlihy et al. 2008), oferecendo assim uma maneira de compor uma solução eficiente para esses falsos conflitos.

STM Haskell foi implementada ao *Glasgow Haskell Compiler* (GHC) em 2008 (Harris et al. 2008) e 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; 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.

Utilizando uma biblioteca de memórias transacionais própria, toda escrita em Haskell (Du Bois et al. 2011), uma implementação de *Transactional Boosting* foi feita para testar as vantagens de desempenho e foram apresentados resultados promissores (Du Bois et al. 2014). O objetivo deste trabalho é disponibilizar uma primitiva de alto nível para o próprio STM Haskell, permitindo assim a aplicação de *Transactional Boosting* de maneira nativa no GHC. Para isso é necessário que uma extensão seja feita ao *RunTime System* da linguagem.

2. METODOLOGIA

Transactional Boost é uma técnica utilizada para transformar objetos linearmente concorrentes em objetos transacionalmente concorrentes (Herlihy et al. 2008), permitindo assim sua utilização dentro dos blocos atômicos do STM. Nas transações os objetos são tratados como caixas-pretas e ambos conflitos e registros à memória são tratados logo que são encontrados. Para a proposta de *Transactional Boosting* original em (Herlihy et al. 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.* Assim, foi necessário uma extensão da biblioteca e do RTS do compilador para tal implementação.

A função 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 primitiva então envolve a função numa ação STM permitindo assim sua chamada dentro do bloco transacional. A função de boost tem o seguinte protótipo e argumentos:

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

1. Uma ação (do tipo **IO(Maybe a)**), a função original que vai ser executada.
2. Um 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.

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

A implementação da primitiva se divide em duas partes. Uma interface Haskell em alto nível, que provê a função chamada para realizar o *boost* de uma ação, e uma camada principal no RTS no baixo nível do GHC. Dentro do RTS duas adições foram feitas: a primeira foi criar uma nova estrutura de dados responsável por armazenar os *handlers* necessários para o *boost* de uma ação; A segunda adição foi criar as funções responsáveis por interagir com a estrutura adicionada. A interface da função em Haskell é então responsável por lidar com o sistema de tipos da linguagem e assim fazer as chamadas para as novas funções do RTS permitindo a aplicação do *Transactional Boosting*.

Para exemplificar o uso da função de **boost** podemos tomar como exemplo a implementação de uma estrutura para o conceito matemático de conjuntos como em (Du Bois et al. 2014). Uma implementação de conjuntos normalmente oferece três funções, *add*, *remove* e *contains*.

Na implementação é importante garantir que se uma transação está trabalhando em um elemento, nenhuma outra transação vai utilizar o mesmo

elemento (vide Figura 1). Isso pode ser alcançado utilizando uma tabela hash para associar um *lock* para cada elemento do conjunto.

Função Chamada	Inversa
`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

Comutatividade

`add set x / _` ⇔ `add set y / _`, $x \neq y$

`remove set x / _` ⇔ `remove set y / _`, $x \neq y$

`add set x / _` ⇔ `remove set y / _`, $x \neq y$

`add set x / False` ⇔ `remove set x / False` ⇔ `contains set x / _`

Figura 1. Especificação da estrutura de conjuntos [Du Bois et al. 2014].

Para adicionar um elemento ao conjunto devemos adquirir o *lock* associado ao elemento e então inseri-lo na lista. 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*. Essas ações são então passadas para a função de **boost** para criação da versão *transactional* da ação. 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.

3. RESULTADOS E DISCUSSÃO

Temos o suporte a *Transacional Boosting* já adicionado ao sistema de tempo de execução do GHC e estamos desenvolvendo diferentes testes para medir as vantagens e desvantagens da utilização do *Transactional Boosting*. Apresentamos aqui os resultados de um desses exemplos já implementados, a estrutura de conjuntos apresentada na seção anterior. 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 as medições de tempo a biblioteca *System.Time* da linguagem foi utilizada.

Para testar a estrutura de conjunto listas de 2000 operações foram geradas aleatoriamente, bem como um conjunto inicial de 2000 elementos. Para as execuções o número de thread e cores foram os mesmos nas execuções e 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*.

Duas implementações são comparadas nos gráficos da Figura 2, uma utilizando o STM puro e outra com o *Transactional Boosting*; A versão utilizando STM puro utiliza uma lista encadeada transacional para o armazenamento dos dados; A versão que utiliza o *Transactional Boosting* (TB) é a implementação

descrita na seção 2. 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 em uma estrutura de conjuntos de desempenho bem superior em comparação à alternativa feita puramente com o STM Haskell. Isso acontece principalmente por conta do alto número de falsos conflitos associados a se percorrer uma lista encadeada utilizando os recursos nativos da STM.

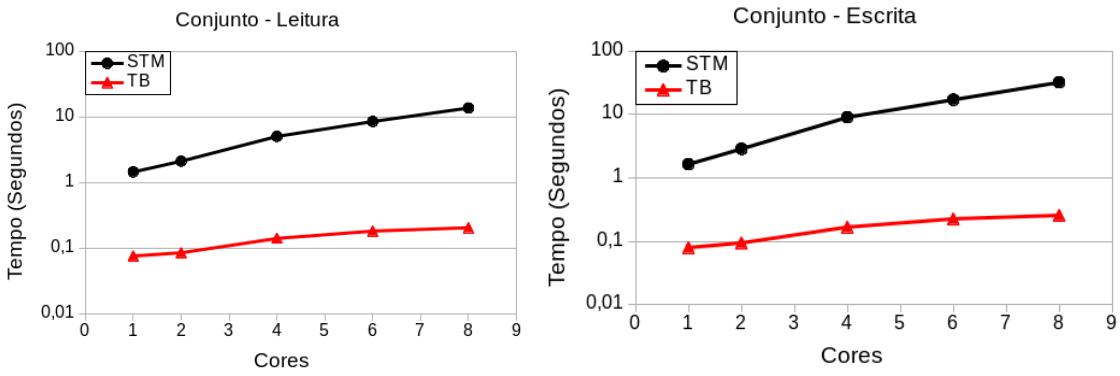

Figura 2. Tempo de execução de 2000 operações para cada core (A esquerda: 40% add e remove} + 60% contains; A direita: 75% add e remove + 25% contains).

4. CONCLUSÕES

A primitiva para *Transactional Boosting* descrita neste artigo oferece uma maneira direta de compor objetos linearmente concorrentes com objetos transacionalmente concorrentes, de uma maneira que independe de qualquer tratamento do STM mas ainda preservando suas propriedades. *Transactional Boosting* é um tratamento de baixo nível ao controle de concorrência e requer que o programador preserve a atomicidade e isolamento da estrutura transacional, além disso seu mau uso pode resultar em problemas como *deadlock*.

Entretanto se utilizada por programadores experientes pode ser usada para desenvolver bibliotecas concorrentes de alto desempenho capazes de serem compostos com blocos transacionais oferecendo uma solução à perda de desempenho por falsos conflitos.

5. REFERÊNCIAS BIBLIOGRÁFICAS

Du Bois, A.R. (2011) An Implementation of Composable Memory Transactions in Haskell. **Lecture Notes in Computer Science**, vol 6708. Springer, Berlin, Heidelberg, p. 34–50, 2011.

Du Bois, A. R., Pilla, M. L., and Duarte, R. Transactional Boosting for Haskell. **Lecture Notes in Computer Science**, vol 8771. Springer, p. 145–159, 2014.

Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. Composable memory transactions. **Commun. ACM**, vol. 51, p.91–100, New York, NY, USA. ACM. 2008.

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