PROPOSTA DE UM ESCALONADOR DE TRANSAÇÕES PARA O GHC

RODRIGO MEDEIROS DUARTE[1]; ANDRÉ R. DU BOIS; MAURÍCIO LIMA PILLA[2]; RENATA H.S. REISER[3]

[1] Universidade Federal de Pelotas 1 – rmduarte@inf.ufpel.edu.br
[2]Universidade Federal de Pelotas – {pilla,dubois}@inf.ufpel.edu.br
[3]Universidade Federal de Pelotas – reiser@inf.ufpel.edu.br

1. INTRODUÇÃO

Memórias Transacionais (MT) são uma abstração para a programação concorrente, onde a sincronização entre *threads* é realizada através de transações, parecidas com as realizada em bancos de dados (RIGO, 2007) . Este modelo de sincronização isenta o programador de controlar a sincronização, pois o sistema transacional realiza todo o processo, dessa forma MT evita que o programador cometa erros clássicos de programação, como os conhecidos *deadlocks*.

Para realizar a sincronização entre *threads*, MTs usam dois conceitos que são o (I) versionamento de dados e (II) detecção de conflitos. O primeiro esta ligado a forma como as transações tratam os dados especulativos da transação, já o segundo a forma como as transações garantem isolamento entre si, uma transação não pode ver o resultado intermediário da computação de outra. Um conflito ocorre quando duas ou mais transações tentam acessar um mesmo dado na memória e pelo menos um destes acessos é de escrita. Para resolver conflitos, MT usa um Gerenciador de Contenção (DEMSKY,2010), este detecta as transações conflitantes reiniciando as mesmas.

Essa tarefa de reiniciar as transações não é a melhor alternativa (NINÁCIO,2013), devido ao fato de que transações conflitantes podem acabar sendo reiniciadas ciclicamente por recorrência dos mesmos conflitos, principalmente se a contenção de memória for muito elevada. Isso leva ao desperdício de recursos computacionais, pois um ou mais processadores estarão executando código desnecessário. Em virtude disso, o foco das pesquisas tem direcionado o tratamento de conflitos para o escalonador. A nível de escalonador é possível serializar transações conflitantes para evitar que as mesmas conflitem novamente.

STM Haskell é uma extensão da linguagem funcional Haskell que inclui a abstração de MT. Aliada à abstração do paradigma funcional, STM Haskell prove facilidades para o desenvolvimento de programas concorrentes (MARLOW,2013).

Glasgow Haskell Compiler (GHC) é um compilador, interpretador e máquina virtual para a linguagem Haskell. Este já possui em sua máquina virtual a implementação de STM.

O Run Time System (RTS) do GHC, utiliza um gerenciador de contenção tímido, ou seja, sempre aborta a transação que detecta o conflito. Assim nenhuma informação é passada ao escalonador a respeito de conflitos, com isso o escalonador simplesmente recebe o código de uma transação conflitante para reexecução, incapacitando uma tomada de decisão para evitar novos conflitos.

Com base no exposto, este artigo traz como proposta a modificação do escalonador do RTS do GHC, fazendo com que o mesmo passe a reconhecer transações e as reescalone de forma a evitar novos conflitos.

2. METODOLOGIA

Para a formação desta proposta foi estudada a implementação do RTS do GHC, bem como o estudo de trabalhos publicados na área sobre escalonadores

de transações. Assim uma breve introdução a STM Haskell é fornecida e depois é discursado a respeito do GHC e dos algoritmos de escalonamento estudados.

2.1. STM HASKELL

Em STM Haskell é definido um tipo especial de variável (`TVar`) que só pode ser lida e modificada por duas operações que são `readTVar :: TVAr a → STM a` e `writeTVar :: TVar a → a → STM ()`. Atenta-se para o fato de que ambas funções possuem tipo de retorno `STM a`. Isso indica que estas operações só podem ser executadas dentro de uma função atômica, `atomically :: STM a → IO a`. Essa função obriga que o programador execute as transações dentro de uma ação atômica, evitando assim que o programador insira erros como condições de corrida em sua implementação, pois o acesso externo a TVar fora de uma função `atomically` desrespeita a lei de tipos de Haskell, retornando um erro já no momento da compilação do código.

2.2. RTS DO GHC

O GHC é uma ferramenta gratuita e de código aberto desenvolvida pela universidade de Glasgow e contribuidores (PEYTON et al, 2016). Esta ferramenta esta implementada em duas linguagem diferentes. Uma em alto nível implementada em Haskell e outra em mais baixo nível implementada em C. A parte da implementação que é foco de estudo para este trabalho e a implementação de MT na máquina virtual e o escalonador, ambas desenvolvidas em C.

O escalonador do RTS do GHC usa o modelo *round robin*, onde a cada fatia de tempo de execução uma *thread Haskell* ou TSO (*Thread State Object*) é rê-escalonada, podendo esta *thread* ser enviada para outro processador que esteja ocioso. No escalonador tem-se uma distinção entre processadores físicos existentes no computador e processadores virtuais, os primeiros são chamados de *worker threads*, já o segundo são chamados de HEC (*Haskell Execution Context*) (MARLOW,2009). Ao começar uma execução, o escalonador pega um TSO da memória e coloca na fila de execução de um HEC, neste o escalonador atribui sua execução para um *worker thread*.

A implementação de MT no RTS é realiza através de uma estrutura de dados conhecida como TRec (*Transactional Record*) (HARRIS,2008). Um TRec é uma estrutura de dados que compõem um TSO, e armazena as informações e o estado de uma transação em uma *thread Haskell.*

Na implementação atual do escalonador do RTS, nenhuma informação sobre uma transação é passada para este, ou seja, quando uma transação aborta o escalonador simplesmente reexecuta o código da transação dentro da *thread* que a possui. Assim transações conflitantes que ocupem mais de um processador, podem acabar utilizando recursos de forma desnecessária, pois vão ser executadas muitas vezes antes de conseguir efetivar mudanças no estado do programa. Isso causa desperdício de recursos, deixando a execução mais onerosa.

2.3. ALGORITMOS DE ESCALONAMENTO DE TRANSAÇÕES

No trabalho de (DOLEV, 2008) é descrito CAR-STM, que é um escalonador de transações que pode operar de duas formas distintas. A primeira serializando as transações conflitantes, colocando estas na mesma fila de execução de um processador e a outra através de um modulo que compara os endereços das va-

riáveis acessadas pelas transações e faz uma previsão se pode haver ou não um conflito.

No trabalho descrito em (BLAKE, 2009), o escalonador utiliza uma tabela onde são armazenados os conflitos ocorridos entre as transação. Com base na taxa de conflitos descrita nesta tabela, o escalonador escolhe para qual processador é melhor escalonar a *thread*.

Em (DRAGOJEVIC, 2009), é apresentado um escalonador que prediz o futuro acesso de cada *thread* que executa uma transação, baseado no histórico de acessos passados e dinamicamente serializa as transações conflitantes. O histórico de uma transação é feita através da analise das variáveis acessadas presentes nos conjuntos de escrita e leitura de cada transação.

Já (YOO,2008), descreve um escalonador que utiliza um esquema de *backoff* exponencial. Quando o nível de conflitos aumenta além de uma taxa estabelecida, as *threads* são colocadas em uma fila de espera que espera um tempo exponencial para ser reescalonada. Assim que a taxa de conflitos diminui, o escalonador deixa de colocar as *threads* na fila de espera e passa a escalonar as *threads* normalmente, afim de voltar a explorar maior paralelismo.

Em (NICÁCIO,2013), é apresentado um escalonador que opera de duas formas distintas, uma para transações pequenas e outra para transações grandes. Quando em transações pequenas o escalonador utiliza heurística para definir a taxa de conflitos e definir se deve ou não serializar as transações. Já para transações grandes, o escalonador utiliza instrumentação extra. Esta instrumentação inserida nas transações não aumenta o custo da computação destas, isso pelo fato de que o custo para gerar a instrumentação ser irrelevante se comparado ao custo de execução da transação.

Concluindo esta revisão, em (MALDONADO,2010) é demostrado um escalonador simples, onde cada *thread* que possui uma transação conflitante é migrada para uma fila de espera, esta fica nesta fila aguardando até que a transação com quem conflitou seja efetivada, assim esta é reescalonada para a fila de execução do escalonador.

3. RESULTADOS E DISCUSSÃO

Neste trabalho foram estudados diferentes tipos de escalonadores de transações, desde implementações simples até mais elaboradas. Das implementações estudadas, a mais simples para ser utilizada na modificação do RTS do GHC é a CAR-STM. Esta apresenta algumas semelhanças entre as estruturas já presentes na implementação do GHC, facilitando assim a modificação do escalonador, sendo que usando esta modificação como base, pode-se depois realizar modificações mais complexas. Para alcançar tal objetivo, considera-se a modificação da estrutura do TSO, para que este passe a fornecer informações ao escalonador do estado das transações, para que este possa tomar ações quando as transações conflitam. Como a implementação do escalonador já possui tratamento de roubo de tarefas, quando uma transação conflitar, esta pode simplesmente notificar ao escalonador com quem está conflitou e pedir para ser migrada para o HEC que detectou o conflito.

4. CONCLUSÕES

Para a realização da proposta deste trabalho será feita a modificação do escalonador do GHC com o objetivo de que este passe a reconhecer transações e assim possa tomar ações a respeito de conflitos. O Objetivo principal desta pro-

posta é fazer com que programas desenvolvidos usando STM Haskell passem a presentar melhor desempenho em sistemas de alta contenção de memória.

5. REFERÊNCIAS BIBLIOGRÁFICAS

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias Transacionais: Uma Nova Alternativa para a Programação Concorrente. Em: **MINICURSOS DO VIII WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO**, WSCAD 2007, 2007, Gramado,RS,Brasil. Anais SBC, 2007.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. Proactive Transaction Scheduling for Contention Management. In: ND ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 42., 2009, New York, NY, USA. **Proceedings. . . ACM**, 2009. p.156–167.(MICRO 42)

DRAGOJEVIC, A.; GUERRAOUI, R.; SINGH, A. V.; SINGH, V. Preventing Versus Curing: Avoiding Conflicts in Transactional Memories. In: ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 28., 2009, New York, NY, USA. **Proceedings. . . ACM**, 2009. p.7–16. (PODC 09).

YOO, R. M.; LEE, H.-H. S. Adaptive Transaction Scheduling for Transactional Memory Systems. In: TWENTIETH ANNUAL SYMPOSIUM ON PARALLELISM IN ALGORITHMS AND ARCHITECTURES, 2008, New York, NY, USA. **Proceedings. . . ACM**,2008. p.169–178. (SPAA '08).

NICÁCIO, D.; BALDASSIN, A.; ARAÚJO,G. Transaction Scheduling Using Dynamic Conflict Avoidance. International Journal of Parallel Programming, v.41, n.1, p.89–110, 2013.

MALDONADO, W.; MARLIER, P.; FELBER, P.; SUISSA, A.; HENDLER, D.; FEDOROVA, A.; LAWALL, J. L.; MULLER, G. Scheduling Support for Transactional Memory Contention Management. Em: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 15., 2010, New York, NY, USA. **Proceedings. . . ACM**, 2010. p.79–90. (PPoPP '10).

MARLOW, S. **Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming.** CA, USA: O'Reilly Media, Inc., 2013.

PEYTON-JONES, S.; MARLOW, S. et al. **Glasgow Haskell Compiler**. Disponível em https://www.haskell.org/ghc/. Acesso em: Maio de 2016.

DEMSKY, B.; DASH, A. Evaluating contention management using discrete event simulation. In: **FIFTH ACM SIGPLAN WORKSHOP ON TRANSACTIONAL COMPUTING** (APRIL 2010), 2010, 2010

MARLOW, S.; PEYTON JONES, S.; SINGH, S. Runtime Support for Multicore Haskell. **SIGPLAN** , New York, NY, USA, v.44, n.9, p.65–78, Aug. 2009.

HARRIS, T.; MARLOW, S.; JONES, S. P.; HERLIHY, M. Composable memory transactions. **Commun. ACM**, New York, NY, USA, v.51, p.91–100, August 2008.

DOLEV, S.; HENDLER, D.; SUISSA, A. CAR-STM: Scheduling-based Collision Avoidance and Resolution for Software Transactional Memory. In: TWENTY-SEVENTH ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, 2008, New York, NY, USA. **Proceedings. . . ACM**, 2008. p.125–134. (PODC '08).