Uma Linguagem de Domínio Especifico com Suporte à Memória Transacional Distribuída

JERÔNIMO DA CUNHA RAMOS; MAURÍCIO LIMA PILLA; ANDRÉ RAUBER DU BOIS

LUPS / PPGC / CDTec / UFPel – {jdcramos, pilla, dubois}@inf.ufpel.edu.br

1. INTRODUÇÃO

Para tirar proveito do paralelismo disponibilizado pelos computadores atuais, os programas devem conter atividades que possam executar concorrentemente. Normalmente, em arquiteturas de memória compartilhada, isto é feito com o uso de *threads*, variáveis compartilhadas e *locks*. Porém, este modelo de programação impõe quase toda a complexidade de exploração do paralelismo e de tratamento das condições de corrida ao programador, o que torna-o propenso a erros, como *deadlocks*. A comunidade científica vem propondo alternativas a este modelo para facilitar e popularizar a programação concorrente, como é o caso das memórias transacionais (RIGO et al., 2007).

Memória transacional é uma abstração baseada na ideia de transações, semelhantes às de banco de dados. Uma transação de memória é uma sequência atômica de operações que modificam a memória e que podem ser executadas completamente ou abortadas. Desta maneira, executam como se estivessem modificando uma área de memória isolada do acesso de outras transações. Estas só conseguem ver o resultado após o *commit* (DU BOIS, 2008).

Existem muitas vantagens no uso deste modelo ao invés dos convencionais (*mutex*, semáforos, monitores, etc), uma delas é a facilidade de programação, pois basta que o programador utilize um bloco atômico para marcar os trechos onde podem ocorrer condições de corrida e o sistema transacional se encarrega de garantir a atomicidade da execução do bloco. Outras vantagens são ausência de *deadlock* e composabilidade.

O modelo de memórias transacionais tem se mostrado bastante promissor e largamente estudado para aplicação em máquinas multicore, no entanto, existem poucos trabalhos na área focados em arquiteturas distribuídas. O estudo destas arquiteturas continua importante nos dias de hoje, tanto por comporem uma grande parcela da computação de alto desempenho, quanto pela tendência de cada vez mais dispositivos computacionais se comunicarem entre si. Nas arquiteturas distribuídas, cada nodo possui seu próprio espaço de endereçamento de memória, fazendo com que a comunicação tenha que ser feita por troca de mensagens. Além disto, outra dificuldade clássica das arquiteturas distribuídas é a não existência de um relógio global para a ordenação dos eventos.

Acredita-se que este modelo também possa ser utilizado, com algumas adaptações, para facilitar a programação de arquiteturas distribuídas, porém os trabalhos anteriormente citados não preveem comunicação entre nodos nem ordenação de eventos. Propostas focadas em arquiteturas distribuídas são pouco numerosas, sendo as principais: Cluster-STM (BOCCHINO et al., 2008), DiSTM (KOTSELIDIS, 2008) e TFA (SAAD; RAVINDRAN, 2012). Neste sentido, o objetivo principal deste trabalho é propor a linguagem DCMTJava, uma linguagem de domínio específico embarcada baseada em mônadas, que contempla tanto transações locais quanto transações envolvendo objetos remotos.

2. METODOLOGIA

Foi realizada uma revisão bibliográfica para compreender os conceitos de memórias transacionais e arquiteturas distribuídas. O principal objetivo deste estudo foi identificar as características das implementações de cada modelo já existente e os principais desafios para estender algum destes modelos de forma que disponibilize suporte à transações distribuídas. Optou-se por estender a linguagem CMTJava e utilizar o algoritmo TFA para lidar com os objetos distribuídos. Após esta primeira etapa, deu-se início às implementações.

CMTJava (BANDEIRA, 2013) é uma linguagem de domínio específico que estende a linguagem Java para programação de STM. Ela oferece a abstração de objetos transacionais e utiliza mônadas para a passagem dos metadados entre as operações de uma transação. Para definir um desses objetos basta estender a interface *TObject*, informando ao compilador CMTJava que este é um objeto transacional. Os atributos destes objetos poderão ser acessados somente através de métodos especiais de get e set, gerados automaticamente pelo compilador. Estes métodos retornam ações transacionais como resultado. Estas ações são composta usando um bloco *STMDO{a_1;...;a_n}* que une as operações $a_1;...;a_n$ em sequência (Figura 1) e só pode ser executado com a utilização do método *atomic (*Figura 2*).* Com isto, garante-se as propriedades de atomicidade e isolamento.

```
public STM<stm.Void> deposit (Double n) {

        return new STMDO{
                Double balance <- this.getBalance();
                this.setBalance(balance + n);
                Integer ops <- this.getOps();
                this.setOps(ops+1)
        };
}
```

Figura 1: Definição de uma transação em CMTJava

```
Account account = new Account();

STMRTS.atomic(account.deposit(INITIAL_VALUE));
```

Figura 2: Execução de uma transação em CMTJava

Algumas abstrações de alto nível da CMTJava, como os *TObjects* e os blocos STMDO são baseadas em uma linguagem de mais baixo nível composta de mônadas, *closures* e chamadas à bibliotecas que implementam transações. Logo, para a execução dos programas na linguagem é necessário transformar essas abstrações de alto nível para as construções disponíveis (BANDEIRA, 2013). Um exemplo desta transformação pode ser visto na Figura 3.

```
public STM<stm.Void> deposit (Double n) {

        return  STMRTS.bind( this.getBalance(),{ Double balance =>
                STMRTS.then( this.setBalance(balance + n),
                STMRTS.bind( this.getOps(),{ Integer ops =>
                this.setOps(ops+1)
        }) ) }) ;
}
```

Figura 3: Código da Figura 1 compilado para Java+Closures

A implementação atual da CMTJava (BANDEIRA, 2013) contempla apenas transações locais. O trabalho propõe a linguagem DCMTJava, que estenderá a CMTJava de forma a dar suporte a transações envolvendo objetos distribuídos. A primeira extensão será baseada no algoritmo TFA. Este algoritmo possui o diferencial de mover os objetos através dos nodos, ao invés de mover instruções de controle. Uma comparação entre TFA e outros dois trabalhos focados em memória transacional distribuída pode ser vista em (RAMOS, 2013).

Após a extensão, alguns *TObjects* poderão ser movidos de um nodo à outro, através da rede. Suas classes serão marcadas com *implements RemoteTObject* para que o compilador as reconheça e gere seus métodos especiais. Estes temcomo objetivo principal facilitar a serialização. Outra extensão necessária é a implementação de um sistema de diretório responsável por dar suporte às operações de registrar, mover e abrir os objetos remotos, quando isto for solicitado por alguma transação. Dessa forma, o sistema transacional também deve ser estendido para lidar com as operações distribuídas. As duas principais extensões são referentes à detecção de conflitos e ao *commit*, que devem levar em conta que o ambiente é distribuído. A princípio será utilizada uma estratégia de detecção tardia, onde os conflitos são checados apenas ao se tentar realizar o *commit*, pois esta estratégia tende a utilizar menos acessos à rede.

3. RESULTADOS E DISCUSSÃO

A fase de revisão bibliográfica já foi concluída e o trabalho encontra-se em fase de implementação. Dos resultados práticos, vale ressaltar a possibilidade de mover os *TObjects* entre máquinas e o registro e localização destes, sendo estes requisitos básicos para a implementação do sistema transacional distribuído.

Mover os *TObjects* foi um dificultante inesperado. Objetos comuns da linguagem Java podem ser serializados facilmente se implementarem a interface *Serializable*, porém, o mesmo não ocorre com os *TObjects*. O motivo disto é o uso de *Closures*, que não são serializáveis. Para contornar este problema, foi criada uma classe somente para representação dos dados dos *TObjetcs* e esta é serializável. Sendo assim, para enviá-los, primeiramente ele deve ser convertido em um objeto desta classe de representação e reconvertido em TObject quando for ser acessado.

O registro e localização dos objetos está sendo realizada de maneira distribuída, onde cada nodo possui um diretório que contem a tabela dos objetos locais e recebe as requisições via *socket*. Além disto, a tabela também contém o endereço atual dos objetos que foram criados no nodo e foram movidos para outros nodos. O acesso ao diretório é dado principalmente através de dois métodos, um para registrar um novo objeto e outro para abrir um objeto existente. O método de *registry* recebe o *ID* que identificará o objeto e uma representação serializável dos dados do mesmo. Já o método *openObject* recebe o *ID* do objeto desejado e o *IP* do nodo que criou o objeto, após executado, retorna uma representação do objeto.

O próximo passo de implementação é a alteração do sistema transacional para que utilize o algoritmo TFA. Concluída esta etapa, será possível realizar experimentos para verificar o desempenho do modelo proposto.

4. CONCLUSÕES

Este trabalho propõe uma linguagem de programação de domínio específico embarcada para transações distribuídas, utilizando mônadas para

representar as transações. O modelo de programação com memória transacional tem se mostrado bastante promissor, com diversas vantagens sobre os modelos clássicos de sincronização, porém as arquiteturas distribuídas apresentam características que não são levadas em consideração pela maior parte das implementações de STM, tais como comunicação e ordenação dos eventos sem relógio global, sendo estes abordados por este estudo.

Este trabalho engloba tanto a elaboração de novas construções da linguagem, quanto extensão do sistema transacional e implementação de um diretório. Busca tirar proveito das diversas vantagens do uso de memórias transacionais ao invés dos modelos clássicos de sincronização em arquiteturas distribuídas.

Dando continuidade ao trabalho, as implementações serão concluídas e serão realizados testes para validação do modelo. Um trabalho futuro é a implementação de outras estratégias para lidar com objetos que não podem ser movidos, que poderiam vir a integrar a linguagem, formando uma estratégia mista.

5. REFERÊNCIAS BIBLIOGRÁFICAS

BANDEIRA, R. L. **Um Sistema de Detecção de Conflitos com Invalidação Mista para a Linguagem CMTJava.** 2013. Dissertação (Mestrado em Computação) – Programa de Pós-graduação em Computação, UFPel.

BOCCHINO, R. L.; ADVE, V. S.; CHAMBERLAIN, B. L. Software Transactional Memory for Large Scale Clusters. **Proceedings of the 13th ACM SIGPLAN Symposium on PPoPP,** New York, p.247-258, 2008.

DU BOIS, A. R. Memórias Transacionais e Troca de Mensagens: Duas Alternativas para a Programação de Máquinas Multi-Core. **SBC, P. A. (Ed.). Escola Regional de Alto Desempenho,** p.43-76, 2008.

DU BOIS, A. R.; ECHEVARRIA, M. A. Domain Specific Language for Composable Memory Transactions in Java. **DSL '09: Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages,** p.170-186, 2009.

KOTSELIDIS, C.; ANSARI, M.; JARVIS, K.; LUJÁN, M.; KIRKHAM, C.; WATSON, I. DiSTM: A Software Transactional Memory Framework for Clusters. **Proceedings of 37th International Conference on Parallel Processing,** Washington, p.51-58, 2008.

RAMOS, J. C. **Um Estudo Sobre Memórias Transacionais para Arquiteturas Distribuídas.** 2013. Trabalho Individual (Mestrado em Computação) – Programa de Pós-graduação em Computação, UFPel.

RIGO, S.; CENTODUCATTE, P.; BALDASSIN, A. Memórias Transacionais: Uma Nova Alternativa para Programação Concorrente. **Minicursos do VIII WSCAD,** 2007.

SAAD, M. M.; RAVINDRAN, B. Transactional Forwarding: Supporting Highly-Concurrent STM in Asynchronous Distributed Systems. **Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium,** p.219-226 , 2012.