Uma Linguagem de Domínio Especifico Baseada em Mônada com Suporte à Memória Transacional Distribuída em Java

Jerônimo da Cunha Ramos, Maurício Lima Pilla, André Rauber Du Bois

Programa de Pós-Graduação em Computação (PPGC)
Universidade Federal de Pelotas (UFPel)

{jdcramos, pilla, dubois}@inf.ufpel.edu.br

Resumo. *Memórias transacionais (STM) facilitam a programação paralela. Porém, poucos são os trabalhos da área focados em arquiteturas distribuídas, sendo que estas normalmente apresentam maiores dificuldades de programação do que as arquiteturas centralizadas. Desta forma, o objetivo central deste trabalho é estender uma linguagem transacional para contemplar transações distribuídas, levando a facilidade das STM para as arquiteturas distribuídas.*

1. Introdução

Memória transacional é uma abstração para programação concorrente baseada na ideia de transações, parecidas com as 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 podem ser abortadas. Desta maneira, transações de memória 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 linguagens de programação que utilizam este modelo para controlar a concorrência. A principal vantagem 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 importantes são ausência de *deadlock*, maior possibilidade de exploração do paralelismo e composabilidade.

2. A Linguagem CMTJava

CMTJava [Bandeira 2013] é uma linguagem de domínio específico que estende Java para programação de memórias transacionais em software. 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 podem ser acessados somente através de métodos especiais de *get* e *set*, gerados automaticamente pelo compilador. Estes métodos contém parte do sistema transacional e retornam ações transacionais como resultado. Estas somente podem ser executadas dentro de blocos atômicos. Com isto, garante-se as propriedades de atomicidade e isolamento. A Figura 1 exemplifica a definição e a execução de uma transação na linguagem.

3. DCMTJava

A linguagem CMTJava [Bandeira 2013] contempla apenas transações locais. Já DCMTJava estende-a de forma a também dar suporte a transações envolvendo objetos distribuídos. A primeira extensão é baseada no algoritmo TFA, proposto

```
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)
        };
}
```

```
Account account = new Account();

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

Figure 1. Definição (esquerda) e execução (direita) de uma transação

em [Saad and Ravindran 2012]. Este algoritmo verifica conflitos a nível de objetos e possui o diferencial de movê-los através dos nodos, ao invés de mover instruções de controle, na tentativa de diminuir o *overhead* e amenizar os efeitos da latência de comunicação. Uma comparação entre TFA e outros dois trabalhos focados em memória transacional distribuída pode ser vista em [Ramos 2013].

As classes de objetos que podem ser movidos pela rede são marcadas com `implements RemoteTObject` para que o compilador as reconheça e gere seus métodos especiais. Estes métodos tem como objetivo facilitar a serialização e verificação de conflitos entre os nodos. Além disso, o sistema transacional foi estendido para realizar as operações distribuídas e possui um diretório, para a localização e envio dos objetos, como visto na Figura 2.

```
directory.registry("account1", (Object)new Account());

Account account = (Account) directory.openObject("localhost", "account1");
```

Figure 2. Exemplos de uso dos métodos `registry()` e `openObject()`

4. Considerações Finais

Este trabalho está em fase final de implementação e propõe uma linguagem de programação de domínio específico embarcada para transações distribuídas. Busca tirar proveito das diversas vantagens do uso de memórias transacionais ao invés dos modelos clássicos de sincronização em arquiteturas distribuídas. O próximo passo do trabalho é a realização testes para validação do modelo. Um possível trabalho futuro é a implementação de outras estratégias para lidar com objetos que não podem ser movidos, formando uma estratégia mista.

Referências

Bandeira, R. d. L. (2013). Um sistema de detecção de conflitos com invalidação mista para a linguagem CMTJava. Dissertação de mestrado, Programa de Pós-Graduação em Computação, UFPel, Pelotas, RS.

Du Bois, A. R. (2008). Memórias transacionais e troca de mensagens: Duas alternativas para a programação de máquinas multi-core. *ERAD-RS*, pages 43–76.

Ramos, J. d. C. (2013). Um estudo sobre memórias transacionais para arquiteturas distribuídas. Trabalho individual de mestrado, Programa de Pós-Graduação em Computação, UFPel, Pelotas, RS.

Saad, M. and Ravindran, B. (2012). Transactional forwarding: Supporting highly-concurrent stm in asynchronous distributed systems. In *Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 219 –226.