Avaliação de desempenho do sistema de memória transacional de Clojure como biblioteca de sincronização na linguagem Java

Pablo César Calcina Ccori

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Marco Dimas Gubitoso

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CNPq

São Paulo, agosto de 2011

Avaliação de desempenho do sistema de memória transacional de Clojure como biblioteca de sincronização na linguagem Java

Esta versão definitiva da dissertação contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa realizada por Pablo César Calcina Ccori em 14/06/2011.

Comissão Julgadora:

- Prof. Dr. Marco Dimas Gubitoso (orientador) - IME-USP
- Prof. Dr. Francisco Reverbel - IME-USP
- Profa. Dra. Líria Sato - POLI-USP

Agradecimentos

A Deus

Resumo

Neste trabalho apresenta-se uma avaliação do desempenho da implementação de memória transacional da linguagem Clojure, utilizada como biblioteca de sincronização para uso em conjunto com outras aplicações dentro da máquina virtual de Java. É implementada uma camada de interface entre as estruturas de dados de Clojure e o *benchmark* STMBench7 e são discutidos alguns aspectos que geram sobrecarga no desempenho.
Palavras-chave: memória transacional em software, STM, Clojure.

Abstract

In this work a performance evaluation of Clojure transactional memory implementation is presented, using it as a synchronization library to work together with other applications on Java virtual machine. It is implemented an interface layer between Clojure data structures and STMBench7 benchmark, and issues about overhead in performance are discussed.
Keywords: software transactional memory, STM, Clojure

Sumário

Lista de Abreviaturas

STM	Memória Transacional em Software (*Software Transactional Memory*)
HTM	Memória Transacional em Hardware (*Hardware Transactional Memory*)
TM	Memória Transacional (*Memória Transacional*)
ACID	Atomicidade Consistência Isolamento Durabilidade (*Atomicity Consistency Isolation Durability*)
CAD	Projeto Assistido por Computador (*Computer Aided Design*)
CAM	Manufatura Assistida por Computador (*Computer Aided Manufacturing*)
CASE	Engenharia do Software Assistida por Computador (*Computer Aided Software Engineering*)
GUI	Interface Gráfica de Usuário (*Graphic User Interface*)

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No início da década passada a aplicabilidade da lei de Moore foi questionada, uma vez que na indústria de *hardware* não era mais possível aumentar o desempenho dos microprocessadores por meio de incrementos na frequência do relógio, devido a problemas físicos como: excesso de calor e de consumo de energia ou vazamento de voltagem (Dongarra *et al.*, 2007).

Atualmente os processadores de múltiplos núcleos (*multicore*) encontram-se disponíveis de maneira bastante comum, em servidores, computadores de escritório, computadores portáteis e até em dispositivos móveis.

Como indicam Minh *et al.* (2008), este uso crescente de processadores de múltiplos núcleos constitui um ponto de inflexão no desenvolvimento de software. A fim de aproveitar as vantagens destes novos processadores, os programadores se veem obrigados a desenvolver programas com processamento paralelo e lidar com questões de sincronização, condições de corrida, *deadlocks* entre outras. Neste cenário, o conceito de memória transacional aparece como uma possível solução para o gerenciamento da sincronização de um programa, sem que o programador necessite entrar em detalhes de baixo nível. Assim, diversos sistemas de memória transacional surgem para concretizar esta ideia de maneira eficiente, em diversas plataformas e linguagens de programação, tanto em forma de bibliotecas como de extensões de linguagens existentes.

Com a aparição de diferentes implementações de STM, surge a necessidade de fazer-se comparações de desempenho entre elas, e em relação a métodos tradicionais de sincronização, como *locks*.

As primeiras implementações destes sistemas de memória transacional foram realizadas em linguagens como C, C++, C#, Java e Haskell que compartem a característica de serem linguagens de tipos estáticos e, com exceção desta última, também linguagens imperativas.

Neste trabalho, prestou-se especial atenção à implementação de memória transacional na linguagem Clojure, que é uma linguagem funcional e de tipos dinâmicos. Clojure resolve elegantemente problemas de modificação de dados presentes em linguagens imperativas, e por ser dinâmica e concisa permite um desenvolvimento rápido de aplicações.

Uma característica desta linguagem que contribuiu para que se popularizasse rapidamente é a facilidade com que ela consegue interagir com o código Java existente, bem como a possibilidade de geração de código de *bytes*, interpretável pela máquina virtual de Java. Isto permite que código escrito na linguagem Clojure, seja exportado e utilizado dentro de outras aplicações escritas em Java.

Neste trabalho, pretende-se avaliar o desempenho do uso de bibliotecas com código inteiramente

escrito em Clojure (já que é uma linguagem concisa na qual pode-se escrever programas em poucas linhas de código), utilizando seu próprio sistema de sincronização, mas dentro da máquina virtual de Java, e em interação com outras aplicações escritas em Java.

Para tanto, utiliza-se a ferramenta de *benchmark* STMBench7, implementada na linguagem Java. STMBench7 fornece uma carga de trabalho realista e representativa.

1.1 Objetivos

Neste trabalho pretende-se avaliar o desempenho do sistema de memória transacional implementado na linguagem de programação Clojure.

Para tanto, fazemos uma comparação entre o desempenho da sincronização baseada no STM de Clojure (usada como biblioteca em Java) e a sincronização baseada em *locks*.

Para este fim utiliza-se a ferramenta de *benchmark* STMBench7, a qual fornece um conjunto variado de operações que constituem uma carga de trabalho representativa de sistema de *software* do mundo real.

1.2 Contribuições

Foi respondida a questão inicial levantada nos objetivos deste trabalho, acerca do desempenho da sincronização baseada em STM de Clojure (usada como biblioteca em Java) em comparação com a sincronização baseada em *locks*. O trabalho de comparação foi feito com a ajuda da ferramenta de *benchmark* STMBench7.

Foi implementada uma camada de interação entre o sistema de memória transacional da linguagem Clojure e a ferramenta de *benchmark* STMBench7. Isto permitirá que o sistema STM de Clojure possa ser comparado com outros sistemas de STM, desde que estes sejam também adaptadas para o uso com STMBench7.

Foi identificada uma questão de implementação do *benchmark* STMBench7 que, dependendo das circunstâncias da execução, pode alterar sensivelmente os resultados. Esta questão está relacionada com a forma de medir o tempo gasto por cada *thread* em execução: na implementação original utiliza-se o tempo total do sistema, em lugar de utilizar-se o tempo de usuário, sendo que este último representaria uma medida mais fiel do tempo empregado na execução, pois nele não se contabiliza o tempo gasto pelos processadores em outras tarefas.

1.3 Organização do Trabalho

No capítulo 2 se apresentam conceitos básicos de memória transacional. No capítulo 3 se descreve a linguagem Clojure, com ênfase na sua implementação de memória transacional e interação com Java. Feito isto, no capítulo 4 são apresentadas algumas ferramentas de *benchmark* para memória transacional, enfatizando-se STMBench7, que é o *benchmark* utilizado neste trabalho. Posteriormente, no capítulo 5 faz-se uma descrição da implementação realizada para criar uma interface entre o sistema de memória transacional de Clojure e STMBench7. Na sequência, no capítulo 6 são descritos os resultados deste trabalho. Finalmente, no capítulo 7 expõem-se as conclusões às quais se chegou e também algumas ideias para possíveis investigações futuras.

Capítulo 2

Memória Transacional em Software (*Software Transactional Memory*)

2.1 Transações em Banco de Dados

Transação é um termo amplamente utilizado na área de banco de dados. Uma transação é uma sequência de ações que aparenta ser executada de forma indivisível e instantânea (Silberschatz *et al.*, 2006). Transações são caracterizadas por possuírem os atributos: atomicidade, consistência, isolamento e durabilidade (ACID), descritos a seguir:

- **atomicidade:** esta característica garante que devem ser executadas ou todas as ações que compõem uma transação, ou nenhuma delas.
- **consistência:** estabelece que antes e depois de uma transação, os dados se encontram em um estado consistente. Esta característica é garantida quando a transação aborta: neste caso as modificações nos dados são desfeitas.
- **isolamento:** especifica que uma transação deve produzir o resultado esperado, independentemente de outras transações concorrentes em execução.
- **durabilidade:** esta propriedade garante que uma vez realizada a operação de *commit* pela transação, o resultado se torna permanente e disponível para todas as outras transações.

2.2 Memória Transacional

A memória transacional surgiu da observação das propriedades das transações em bancos de dados aplicada na coordenação de acesso concorrente a dados em memória principal. Para tanto, das quatro características descritas na seção 2.1, as três primeiras (ACI) são aplicadas; a propriedade **d**urabilidade não é importante para memória transacional já que esta memória é geralmente volátil.

Transações em memória resolvem muitos problemas, porém mantem a possibilidade de escrever programas incorretos.

2.2.1 Diferenças de transações em banco de dados com transações em memória

Como mencionado na seção 2.2, as transações em memória foram inspiradas nas transações em bancos de dados, porém, havendo sido concebidas com um objetivo distinto e tratando-se de um tipo diferente de memória, apresentam características particulares que as diferenciam, como é apontado por Larus e Rajwar (2006) e Chung *et al.* (2006):

- Os dados em bancos de dados residem em disco, cujo acesso é cerca de 4 ordens de magnitude mais lento do que o acesso a memória primária.
- As transações em bancos de dados são geralmente extensas e compostas de uma quantidade grande de instruções, o que faz com que, como apontou-se no item anterior, consumam mais tempo em ser executadas. As transações em memória, por sua vez, costumam ter menos instruções e seu tempo de execução é muito menor.
- As transações em memória não requerem armazenar os resultados do processamento, o que simplifica a implementação.
- A memória transacional deve coexistir com a tecnologia existente: linguagens de programação, paradigmas de programação, bibliotecas, programas e sistemas operacionais.

2.2.2 Resenha histórica

Muitos dos conceitos presentes em STM foram antecipados no trabalho de Lomet (1977), que propôs a idéia de procedimento atômico, chamado ação (*action*), com uma sintaxe mostrada a seguir:

```
<identifier> : action(<parameter-list>);
   await <predicate> then
   <statement-list>
end
```

Figura 2.1: *Sintaxe proposta por Lomet para construir procedimentos atômicos*

O corpo da ação é executado como qualquer rotina, mas de forma atômica e isolada, isto é, outras *threads* não podem ver estados intermediários da ação até que esta seja completada e realize a operação de *commit*. A execução da ação é realizada assim que a expressão em `predicate` seja avaliada como verdadeira. Também foi proposto o uso de dados compartilhados entre *threads* desde que sejam declarados com o atributo `shared`. O autor se isentou de fornecer uma implementação do modelo proposto.

Herlihy e Moss (1993) cunharam o termo **memória transacional** (*transactional memory*) e propuseram um modelo para dar suporte a TM em *hardware* como mecanismo para construir estruturas de dados sem usar *locks*. A idéia apresentada é estender o conjunto de instruções do microprocessador que acessam a memória, adicionando as instruções:

- `load-transactional`: carrega um valor de uma localização da memória compartilhada em um registro privado.
- `load-transactional-exclusive`: carrega um valor de uma localização da memória compartilhada em um registro privado e a indica como sendo de "utilização proviável".

- `store-transactional:` escreve um valor de um registro privado em uma localização de *cache*, porém o valor é visível às outras *threads* somente após uma operação sucedida de *commit*.

- `commit:` tenta fazer permanentes as alterações na memória se não houver conflito com outras transações. A instrução devolve o estado de sucesso ou de falha, neste último realiza a operação de `abort`. Esta operação não requer comunicação com outros processadores nem escrita de dados na memória.

- `abort:` descarta todas as modificações.

- `validate:` testa o estado atual da transação e devolve *falso* se a transação foi abortada ou *verdadeiro* no caso contrário.

Shavit e Touitou (1995) introduziram o termo **memória transacional em software** (*software transactional memory*) e propuseram a primeira implementação, na qual as localidades de memória a ser utilizada são declaradas de forma antecipada, isto é, estática. Assim, é possível atribuir a cada localidade de memória (*word*) um registro com a informação da transação que tem permissão para modificá-la; o armazenamento desta relação leva um alto consumo de memória. Neste modelo evita-se a possibilidade de *deadlock* realizando a aquisição das localidades de memória em ordem ascendente, verificando também se estas já pertencem a alguma transação, em cujo caso é realizado um processo de *rollback*. Uma vez que uma transação adquire a posse de todas as localidades de memória que necessita, pode continuar a execução até o final, sem necessidade de *rollback*.

O primeiro sistema de STM dinâmico (DSTM) foi proposto por Herlihy *et al.* (2003). Este sistema não necessita declarar com antecedência a memória a ser usada, melhorando assim a implementação de Shavit e Touitou. Neste trabalho foi introduzido o conceito de gerenciador de contenção (*contention manager*) cuja função é resolver os conflitos entre transações e decidir qual delas irá continuar a execução. Outra característica aqui introduzida é a possibilidade de liberar um objeto pertencente a uma transação antes de realizar a operação de *commit* com o objetivo de poupar o tempo de validação ao final da transação, porém é o programador o responsável por antecipar possíveis conflitos com outras transações.

Por sua vez, Harris e Fraser (2003) descreveram WSTM (Word-granularity STM). Este sistema foi o primeiro a ser integrado em uma linguagem de programação, adicionando a palavra reservada `atomic` à linguagem Java, que engloba um conjunto de sentenças a serem executadas de forma atômica.

Para uma revisão mais profunda dos sistemas de memória transacional propostos na literatura, recomenda-se Larus e Rajwar (2006) e Nasir (2009).

2.3 Construções Básicas de STM

Diferentes formas de adaptar STM aos ambientes de programação existentes foram propostas na literatura, tais como bibliotecas e extensões da linguagem, porém a maioria delas têm em comum as características que são apresentadas a seguir:

2.3.1 Bloco atômico

Um bloco atômico delimita uma sequência de instruções a serem executadas dentro de uma transação, como é ilustrado na figura 2.2.

```
void transferir( Account de, Account para, double quantidade ){
  atomic {
    de.retirada( quantidade );
    para.deposito( quantidade );
  }
}
```

Figura 2.2: *Exemplo de uso de bloco atômico*

É importante notar que o escopo do bloco atômico é determinado dinamicamente e abrange todo o código executado enquanto o controle está dentro do bloco, isto é, o código dos métodos `retirada` e `deposito` é executado transacionalmente.

2.3.2 Declarações

Em alguns sistemas de STM o programador necessita declarar os métodos que serão executados dentro de transações, ou ainda declarar duas versões do mesmo método, para serem executadas dentro e fora de um contexto transacional, como se mostra na figura 2.3.

```
public atomic int transferencia();
public int transferencia();
```

Figura 2.3: *Declaração de duas versões de um método, para ser usado dentro e fora de uma transação*

Adicionalmente é possível declarar os dados a serem compartilhados entre transações, como se ilustra na figura 2.4.

```
Account shared conta_corrente;
```

Figura 2.4: *Exemplo de declaração de dado compartilhado entre transações*

Porém esta estratégia delega ao programador a tarefa de comprovar a correção do programa, pois a omissão de uma declaração pode acarretar efeitos não desejados, como *data races* (Larus e Rajwar, 2006). Não obstante é possível realizar algum tipo de análise do programa para detectar estes casos, como os propostos por Harris *et al.* (2006) e Blanchet (2003).

2.3.3 Instrução *retry*

Harris *et al.* (2005) propuseram a instrução *retry*, com o objetivo de coordenar a execução das transações.

Na especificação de transações em memória não há uma ordem específica para a execução concorrente de transações, assim, a construção *retry* serve para determinar sob que condições uma transação sera executada. Um caso comum seria quando uma transação necessita do resultado de outra transação. Na figura 2.5, reproduz-se o exemplo proposto no trabalho original:

```
atomic {
   if (buffer.isEmpty()) retry;
   Object x = buffer.getElement();
   ...
}
```

Figura 2.5: *Exemplo de uso da instrução* retry

2.3.4 Instrução *orElse*

A instrução *orElse* foi proposta por Harris *et al.* (2005) com o intuito de compor duas transações como alternativas, de tal forma que a segunda transação só é executada caso a primeira não termine a execução.

Na figura 2.6 apresenta-se um exemplo de uso da instrução *orElse*.

```
atomic  {
   //Process A
   {
      obj1.CalculateMatrixMulti();
   }
   orElse
   //Process B
   {
      obj2.CalculateMatrixMulti();
   }
}
```

Figura 2.6: *Exemplo de uso da instrução* orElse

2.3.5 Exceções

Uma exceção é uma condição inesperada ou ao menos inusual durante a execução de uma programa que não é facilmente manuseável no contexto local (Scott, 2005). Uma exceção lançada dentro de uma transação que sai do escopo da transação pode terminar ou abortar a transação. Quando uma transação termine, esta tenta fazer *commit* e as alterações realizadas nos dados permanecem. A exceção também poderia causar que a transação seja re-executada. Uma exceção capturada dentro de uma transação não a termina.

2.4 Questões de projeto de sistemas de memória transacional em *software*

Na seção 2.2 foi explicado o conceito de memória transacional, descrevendo a aparição de sistemas tanto de *hardware*, quanto de *software*. Nas seções subsequentes deste capítulo se dará ênfase à memória transacional em *software* (STM).

2.4.1 Granularidade

Em memória transacional, a granularidade refere-se à unidade de armazenamento de dados sobre a qual o sistema de memória transacional detecta conflitos (Larus e Rajwar, 2006), esta pode ser a nível de palavra, de bloco[1] e de objeto.

Estos diferentes níveis de granularidade apresentam algumas vantagens e desvantagens como apontam Adl-Tabatabai *et al.* (2007):

Granularidade no nível de objeto facilita a escrita de código, ao ser próxima ao raciocínio do programador em ambientes orientados a objetos e reduz a sobrecarga de tempo e espaço de armazenamento na detecção de conflitos. Por sua vez, apresenta a possibilidade de produzir falsos conflitos, por exemplo se duas transações tentarem acessar diferentes posições de um vetor.

Já na **granularidade no nível de palavra e de bloco** permite-se um compartilhamento de dados mais fino, assim por exemplo, é possível ter acessos simultâneos aos elementos de um vetor, facilitando a concorrência, porém produz uma sobrecarga maior para guardar informação sobre os dados. Adicionalmente, em algumas linguagens o programador não tem acesso a nível de *word* ou bloco, o que dificulta eventuais otimizações no código.

2.4.2 Atualização direta e postergada

Uma transação que completa sua execução com sucesso deve atualizar os valores originais, para isto existem duas abordagens: atualização direta (*direct update*) e atualização postergada (*deferred update*) (Larus e Rajwar, 2006).

Com **atualização postergada** a transação guarda uma cópia privada do objeto a ser modificado e ao realizar a operação de *commit*, esta atualiza o valor original com o valor privado. A cópia privada é descartada se a transação aborta. Este tipo de atualização acarreta um alto consumo de memória e foi utilizado pelas primeiras implementações de STM.

Com **atualização direta**, a transação modifica diretamente o valor dos objetos, como consequência é necessário armazenar uma cópia do valor original para restaurá-lo caso a transação realize a operação de *abort*. Este sistema requer um mecanismo para "desfazer" as alterações feitas nos dados, assim como de um controle de concorrência para os acessos a memória. Esta abordagem aparenta ser mais eficiente.

2.4.3 Isolamento forte e fraco

Como foi discutido na seção 2.2, uma das propriedades básicas de uma transação em memória é o isolamento, isto é: a execução de uma transação é independente de outras que possam estar sendo executadas. Blundell *et al.* (2006) identificam dois tipos de isolamento: isolamento forte e isolamento fraco.

Grossman *et al.* (2006) discutem algumas questões referentes à semântica do isolamento.

Com **isolamento fraco**, uma referência a memória utilizada fora de uma transação pode não seguir os protocolos do sistema de Memória Transacional. Consequentemente a referência pode devolver um valor inconsistente ou alterar a execução correta da transação, dependendo da implementação.

[1] O termo bloco refere-se a bloco de *word*s consecutivos e não a bloco de código, como é usado no resto deste trabalho

Para evitar estos problemas é necessário assegurar que o acesso não transacional não entre em conflito com o acesso transacional, o que pode conseguir-se fazendo com que os acessos não se sobreponham seja no tempo ou na localização de memória.

Os tipos de dados são uma importante ferramenta para a identificação de erros no compartilhamento de dados. Por exemplo, o código executado numa transação pode ser restringido para acessar dados de tipo transacional. O acesso a dados deste tipo fora de uma transação seria facilmente identificável como erro. Esta é a abordagem adotada pelo sistema STM de Haskell (Jones, 2007) e de Clojure (VanderHart e Sierra, 2010), onde as variáveis mutáveis são acessíveis apenas dentro de uma transação.

Com **isolamento forte** todos os acessos a dados compartilhados fora de um bloco atômico são convertidos automaticamente em transações individuais, isto porém não garante que a execução concorrente seja correta, pois uma delimitação incorreta do bloco atômico pode levar a comportamento não desejado.

2.4.4 Transações aninhadas

Uma transação aninhada é executada dentro do escopo de outra. Como observam Moravan *et al.* (2006), a necessidade de aninhar transações surge de forma natural ao tentar compor código transacional.

Moss e Hosking (2006) identificam dois tipos de transações: abertas e fechadas.

Nas **transações abertas** as mudanças realizadas sobre os dados se tornam visíveis para todas as outras transações no sistema assim que operação de *commit* é efetuada, mesmo que as transações circundantes estejam ainda em execução. Além disso, mesmo que a transação superior aborte, o resultado das transações aninhadas abertas prevalecerá após seu respectivo *commit*. No exemplo dado na figura 2.7, mesmo depois de abortar a transação externa, a variável x fica com o valor 3.

```
int  x = 1;

atomic  {

    x  = 2;

    atomic open {
         x = 3;
    }

    abort;
}
```

Figura 2.7: *Exemplo de transação aberta*

Quando uma **transação fechada** interna faz commit ou é abortada, o controle passa para a transação circundante. Contudo, as transações que não fazem parte do aninhamento veem estas mudanças apenas quando a transação aninhada mais externa realiza o processo de *commit*. No exemplo dado na figura 2.8 a variável x fica com o valor 2, pois a atribuição da transação interna se desfaz com o aborto.

```
int  x = 1;

atomic  {

    x  = 2;

    atomic closed {
        x = 3;
        abort;
    }
}
```

Figura 2.8: *Exemplo de transação fechada*

2.4.5 Controle de concorrência

Um sistema de controle de concorrência necessita sincronizar os acessos concorrentes aos dados. Existem duas grandes abordagens para o controle de concorrência.

O **controle de concorrência pessimista** permite ao objeto adquirir acesso exclusivo sobre os dados antes de usá-los, prevenindo assim acessos de outras *threads*. Isto pode levar a situações de *deadlock*, mas é evitável realizando o acesso exclusivo numa ordem predeterminada ou forçando uma das transações a abortar.

Com o **controle otimista de concorrência** a detecção e resolução do conflito acontece depois que o conflito acontece (e não ao mesmo tempo, como no controle pessimista). Assim, várias transações podem ter acesso aos dados simultaneamente e detectar os possíveis conflitos somente na hora de validar os dados para realizar o *commit*. O desempenho desta abordagem é maior se os conflito são pouco frequentes, em comparação com a sobrecarga gerada com o uso de *locks* no controle pessimista.

2.4.6 Detecção antecipada e tardia de conflitos

Como apontam Larus e Rajwar (2006) existem três pontos na execução de uma transação onde um sistema STM pode detectar um conflito:

- Durante a **aquisição** de um objeto compartilhado.
- Durante a **validação** dos objetos lidos ou modificados localmente com o objetivo de detectar se foram adquiridos por outra transação.
- Durante o ***commit***, quando a transação comprova que os objetos acessados e modificados ainda são válidos.

A detecção **antecipada** acontece durante as etapas de aquisição ou validação e tem como vantagem que reduce a quantidade de cálculo descartado quando o conflito é detectado, porém isto pode causar o aborto de uma transação que poderia haver concluido sua execução, por exemplo, consideremos as transações T_B,T_C em conflito com a transação T_C sobre objetos diferentes. T_B aborta assim que detecta o conflito com T_A, e T_A por sua vez aborta ao detectar o conflito com T_C, porém, T_B poderia haver terminado a execução com T_A abortada.

A detecção **tardía** poderia haver evitado este problema, encontrando o conflito na etapa de *commit*, porém, maximiza o cálculo descartado.

2.5 Problemas com STM

As implementações de STM se veem comprometidas por algumas questões semânticas, como mencionado por Cascaval *et al.* (2008).

- Interação de transações com código não transacional, em particular o acesso a dados compartilhados.
- Propagação consistente de exceções dentro de um contexto transacional.
- Interação com código que não pode ser transacionalizado e.g. entrada/saída.
- Garantir que todas as transações terminam a execução inclusive na presença de conflitos (*livelock*).

Capítulo 3

Clojure

Clojure é uma linguagem de programação funcional baseada em Lisp, com suporte nativo para concorrência, escrita para ser executada na Máquina Virtual de Java (JVM).

A versão inicial de Clojure apareceu em 2007, sendo que a primeira versão estável foi apresentada em maio de 2009. Desde então, Clojure foi ganhando uma grande popularidade que segundo VanderHart e Sierra (2010), é devida principalmente a três características importantes.

A primeira é o fato de resolver o problema de processamento paralelo com técnicas relativamente recentes como processamento baseado em agentes e memória transacional em software; sobre esta técnica discorreu-se no capítulo 2.

A segunda característica que contribuiu à popularidade de Clojure é o fato de ser uma linguagem funcional, dinâmica e concisa. Assim sendo, é possível escrever programas em uma fração do número de linhas que seria necessário com uma linguagem de tipos estáticos, como C ou Java. Boehm (1981) afirma que os programadores conseguem escrever aproximadamente o mesmo número de linhas de código por ano, independentemente da linguagem de programação, assim, o uso de Clojure produziria um incremento significativo na produtividade. Como exemplo, na figura 3.1 se mostra um código tomado da biblioteca Apache Commons que verifica se uma String está em branco, isto é, se é vazia ou consiste só de caracteres de espaço.

```
public class StringUtils {
  public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
              return false;
            }
    }
    return true;
  }
}
```

Figura 3.1: *Código em Java para determinar se uma String está em branco*

Na figura 3.2 mostra-se o código em Clojure para realizar a mesma tarefa.

```
(defn blank? [s] (every? #(Character/isWhitespace %) s))
```

Figura 3.2: *Código em Clojure para determinar se uma String está em branco*

Ademais, por ser uma linguagem funcional, a chamada de uma função em Clojure não produz efeitos secundários. (Scott, 2005). De acordo com Harris (2005), os efeitos secundários constituem uma das principais dificuldades na implementação de sistemas de memória transacional.

A terceira característica que ajudou a difundir rapidamente a linguagem Clojure é a plataforma Java. A Máquina Virtual de Java é um software de última geração, robusto, maduro, estável e rápido. Com milhares de bibliotecas de propósito específico desenvolvidas para esta plataforma. A integração de Java e Clojure é em dois sentidos: Clojure pode aproveitar todo este universo de bibliotecas existentes em Java, e ao mesmo tempo, pode exportar código de *bytes* interpretável pela Máquina Virtual de Java, isto é explicado com maior detalhe na seção 3.3.

Neste capítulo será apresentada a linguagem Clojure, enfatizando-se as propriedades de concorrência e de integração com Java, que constituem as principais motivações para ter escolhido a linguagem neste trabalho.

3.1 Programação Funcional

Programação funcional é um paradigma de programação no qual a saída de um programa é definida como uma função matemática das entradas, sem o conceito de estado interno ou de efeitos colaterais. (Scott, 2005). Algumas das principais características próprias de linguagens de programação funcional são:

3.1.1 Funções como valores de primeira classe

Chamam-se valores de primeira-classe aqueles que podem ser passados como parâmetro ou ser devolvido por uma função e podem ser armazenados numa estrutura de dados. Assim, é necessário que novos valores possam ser calculados em tempo de execução. No caso das funções como valores de primeira-classe, é necessário que o comportamento da função seja determinado de forma dinâmica. Na maioria das linguagens imperativas as funções são valores de segunda-classe, isto é, o valor pode ser passado como parâmetro mas não devolvido por uma função e armazenado numa variável.

Em Clojure, como em outros dialetos de Lisp, os programas são escritos utilizando as mesmas estruturas de dados que são usadas nos próprios programas, ou seja, o código como dado (*code-as-data*).

3.1.2 Polimorfismo

Polimorfismo é a possibilidade de uma função ser aplicada a múltiplos tipos de dados, os quais possuem características em comum (Scott, 2005).

As estruturas de dados em Clojure são implementadas a partir de uma interface comum, o que permite um polimorfismo rico na linguagem. Adicionalmente, é possível definir uma função com múltiplas implementações, determinando em tempo de execução qual delas usar, dependendo dos argumentos da função.

Tipo	Exemplo
Number	`29`
String	`"Exemplo"`
Boolean	`true`
Character	`\a`
Keyword	`:chave`
List	`'(1 2 3)`
Vector	`[1 2 3]`
Map	`{:chave valor :chave valor}`
Set	`#{1 2 3}`

Tabela 3.1: *Tipos de dados de Clojure*

3.1.3 Recursividade

As listas são importantes em linguagens funcionais por terem uma definição recursiva naturalmente, assim a função é aplicada ao primeiro elemento e recursivamente ao resto da lista. A recursividade é importante na programação funcional para realizar tarefas repetitivas, desde que não existem os efeitos colaterais.

3.1.4 Coleta de lixo

É o processo de identificar os objetos aos quais não existem mais referências, com o objetivo de reaproveitar a memória que estes ocupam. Linguagens funcionais geralmente implementam algum mecanismo de coleta de lixo.

3.2 A linguagem

Clojure é baseada em Lisp, e assim sendo herda todas as construções sintáticas desta linguagem. As listas constituem a estrutura de dados básica em Clojure. Estas são delimitadas por parênteses e os elementos são separados por espaços ou, opcionalmente, por vírgula. O primeiro elemento é o nome da função ou operador e os elementos restantes são os argumentos.

A seguir são apresentados os elementos básicos da linguagem Clojure, sintaxe, estruturas de dados, estruturas de controle e outras características.

3.2.1 Tipos e estruturas de dados

Clojure é uma linguagem com tipos de dados dinâmicos, isto é, o tipo de uma variável é comprovado em tempo de execução e não durante a compilação. Na tabela 3.1 é aprensentado um resumo dos tipos de dados nativos de Clojure.

Estruturas de dados persistentes

As estruturas de dados em Clojure são imutáveis, isto é, elas não mudam. São criadas com um valor e o mantém constante durante a existência do objeto. Assim, é garantido que o objeto possa ser usado em diferentes contextos, em múltiplas *threads* sem conflitos. Porém, naturalmente a lógica de programação requererá que o valor de uma estrutura de dados seja mudado. A abordagem

da programação funcional é criar um novo objeto a partir de uma cópia do original, preservando assim o valor antigo. Já que os objetos são imutáveis, eles podem compartilhar a parte comum da estrutura com o objetivo de poupar memória, como é mostrado nas figuras 3.3 e 3.4

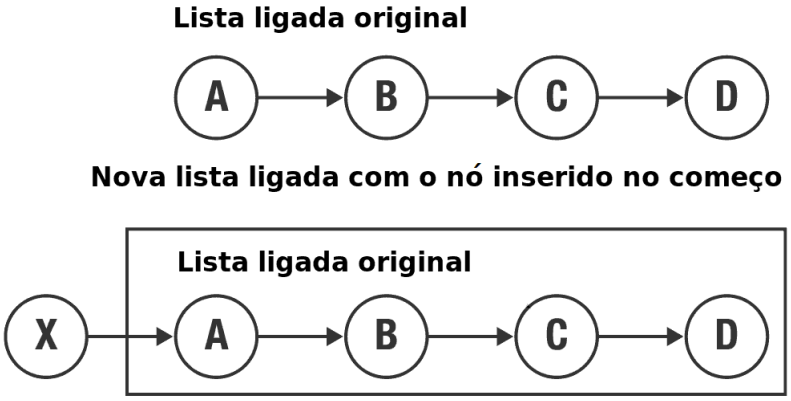

Figura 3.3: *Compartilhamento de estrutura entre duas listas ligadas (VanderHart e Sierra (2010) pág. 7)*

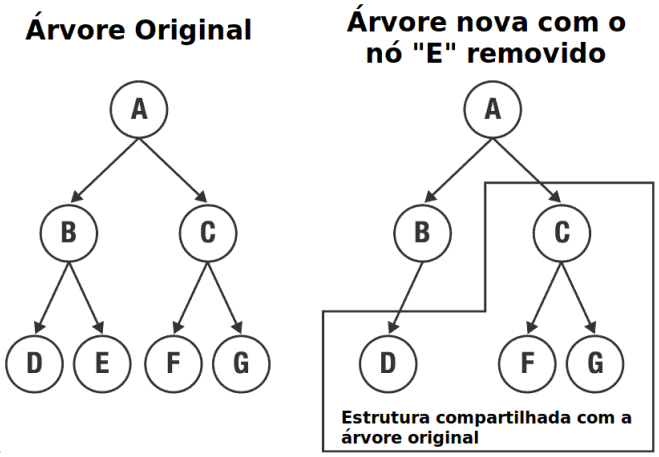

Figura 3.4: *Compartilhamento de estrutura entre duas árvores binárias ligadas (VanderHart e Sierra (2010) pág. 8)*

3.2.2 Referências `refs`

Uma `ref` é um tipo de variável que referencia um valor qualquer. Este valor pode ser lido utilizando a função `deref`, ou utilizando a macro `@`, como é ilustrado na figura 3.5.

O tipo de dados `ref` em Clojure é o único tipo de dados que pode ser mudado, e isto deve realizado dentro de uma transação, por meio da função `dosync`.

3.2.3 Funções

Em Clojure, uma chamada a função é uma lista cujo primeiro elemento é o nome da função e os demais elementos são os argumentos desta. Na figura 3.6 mostram-se alguns exemplos de chamada a função.

```
user=> (def valor (ref 5))
#'user/valor
user=> valor
#<Ref@2565a3c2: 5>
user=> (deref valor)
5
user=> @valor
5
```

Figura 3.5: *Exemplo do uso de* `deref`

```
;; Define uma lista
(def a '(1 2))

;; Soma de numeros
(+ 1 2 3 4 5 6 7 8)

;; Concatena as strings Ola e mundo, em seguida imprime
(println (str "Ola" "mundo" ))
```

Figura 3.6: *Exemplos de chamadas a funções em Clojure*

Definição de funções

A definição de uma função na linguagem Clojure é realizada por meio do uso da macro `def`. Na figura 3.7 é ilustrada a definição de uma função que determina se um numero é ímpar. Note-se o uso de comentário como parte da sintaxe, este comentário constitui a documentação da função quando é invocada a função `doc`, que devolve a documentação de uma função dada.

```
(defn impar? [n]
 "Devolve true caso n seja impar"
 (not (= (mod n 2) 0)))
```

Figura 3.7: *Exemplo de definição de funções em Clojure*

Funções anônimas

Em Clojure é possível definir funções sem nome (anônimas), que pode resultar prático em casos onde o propósito da função é claro e auto-explicativo, e dar-lhe um nome não ajude a leitura do código, mas chegue a dificultá-la. Outra motivação para usar uma função anônima é a possibilidade de utilizá-la somente num contexto local dentro de uma outra função e não globalmente. Na figura 3.8 mostra-se um trecho da implementação realizada neste trabalho. A função implementada, `getRange`, devolve uma lista de valores que se encontram dentro do intervalo determinado por `minKey` e `maxKey`. Utiliza-se uma função anônima dentro da função `filter` com o objetivo de avaliar se o um valor particular se encontra dentro do intervalo.

```
(defn ClojureSTMIndex-getRange
[#^br.usp.ime.backend.ClojureSTMIndex this minKey maxKey]

(vals (filter #(< (.getValue minKey) (key %) (.getValue maxKey))
              @(.state this))))
```

Figura 3.8: *Exemplo de implementação de uma função anônima em Clojure. A função devolve uma lista de valores dentro de um intervalo dado e faz parte da implementação proposta neste trabalho*

3.2.4 Controle de Fluxo

`if`

A sintaxe do `if` em Clojure é: (if condição ação-caso-verdadeiro ação-caso-falso). Na figura 3.9 é ilustrado o uso de `if`, extraindo um trecho da implementação realizada neste trabalho, em que, caso o nó atual não possua um nó pai (superAssembly nulo), devolve-se um número padrão (*NumAssmLevels*) e caso contrário é devolvido o nível superior decrementado em 1.

```
(if (nil? superAssembly)
   (. stmbench7.Parameters NumAssmLevels)
   (dec (. superAssembly getLevel) ))
```

Figura 3.9: *Exemplos do uso de `if`*

`do`

Em alguns programas surge a necessidade de executar código com "efeitos colaterais", tais como realizar operações de entrada/saída. Para realizar esta tarefa, deve utilizar-se a construção `do`, ou `dosync` como se verá na seção 3.4.1.

Na figura 3.10 se ilustra o uso de `do` para imprimir várias linhas na saída padrão.

```
(do
   (println "executando")
   (println "codigo")
   (println "com efeitos colaterais"))
```

Figura 3.10: *Uso da construção `do` para escrever texto na saída padrão*

Recursividade com `loop/recur`

A construção `loop`, junto com `recur` permite executar funções múltiplas vezes utilizando recursividade em cenários nos quais, numa linguagem imperativa utilizar-se-ia um laço, por exemplo `for`, `while`, `do/while` em Java.

Recursividade de cauda Devido a limitações de *hardware* existe um limite para o número de chamadas aninhadas a função (tamanho da pilha), o que constitui um problema em situações nas

quais o nível de recursividade é muito alto. Uma solução comum em linguagens funcionais é o uso de otimização de cauda, onde o compilador realiza uma otimização das chamadas recursivas de tal forma que elas não ocupam um espaço adicional na pilha. Geralmente o requisito para poder realizar esta otimização é que a chamada à função recursiva se encontre no final da expressão recursiva (na cauda).

recur Em algumas linguagens esta otimização é automática desde que a chamada recursiva se encontre na posição da cauda. Em Clojure esta otimização deve ser realizada explicitamente, utilizando a construção `recur` em lugar do nome da função recursiva. Na figura 3.11 é ilustrado o uso da construção `recur`.

```
(defn soma-n
  "soma os n primeiros numeros"
  ([n] (soma-n n 0 0 ))
  ([n atual soma]
    (if (< n atual)
      soma
      (recur n (inc atual) (+ atual soma)))))
```

Figura 3.11: *Chamada recursiva de uma função com otimização de cauda utilizando `recur`*

loop A construção `loop` de Clojure permite, ao ser usada juntamente com `recur`, declarar e chamar uma função anônima recursiva ao mesmo tempo, simplificando assim a escrita de código com recursividade de cauda. Na figura 3.12

```
(defn soma-loop-n
  "soma os n primeiros numeros"
  [n]
  (loop [atual 0 soma 0]
    (if (< n atual)
      soma
      (recur (inc atual) (+ atual soma)))))
```

Figura 3.12: *Chamada recursiva de uma função com otimização de cauda utilizando `loop` e `recur`*

3.3 Interoperabilidade com Java

Ao ser implementado em Java, Clojure se beneficia da riqueza e portabilidade desta plataforma e também da vasta quantidade de bibliotecas existentes tanto em software de código aberto como em software comercial. Assim, supera um problema apresentado em outras linguagens de programação recentes, onde em cada uma destas é necessário criar quase do zero um conjunto de bibliotecas para propósitos específicos e que geralmente não é possível utilizar fora da linguagem para a qual foram implementadas.

Clojure não vêm com bibliotecas embutidas para lidar com tarefas comuns, como E/S de arquivos, acesso a redes ou conexão com banco de dados. Já que afortunadamente, para qualquer

tarefa comum que se tenha em mente, quase sempre existe uma biblioteca Java que ajude em sua realização. A máquina virtual de Java vem com mais de 4000 classes, que abrangem desde redes até as Interfaces Gráficas de Usuário (GUI). Clojure foi concebido para fazer o trabalho conjuntamente com Java, da maneira mais simples possível.

Adicionalmente, o número de bibliotecas próprias de Clojure está crescendo rapidamente, porém, ele ainda é bem pequeno.

Por outro lado, Clojure permite a geração de classes Java, em forma de código de **bytes**, que podem interagir com outras aplicações dentro da plataforma Java. Na figura 3.13 mostra-se como exemplo um extrato da nossa implementação da classe `ClojureSTMAssembly` em Clojure, que herda da classe `Assembly` da API de STMBench7. Na figura é mostrada apenas a declaração da classe, que contém meta-dados utilizados para a geração de código.

```
(gen-class
 :name "br.usp.ime.backend.ClojureSTMAssembly"
 :implements ["stmbench7.core.Assembly"]
 :extends "br.usp.ime.backend.ClojureSTMDesignObj"
 :constructors {[int String int stmbench7.core.Module
   stmbench7.core.ComplexAssembly][int String int],
   [stmbench7.core.Assembly][]}
 :init init
 :state state
 :prefix "ClojureSTMAssembly-")
```

Figura 3.13: *Exemplo do uso de geração de classes Java em Clojure*

Na figura 3.14 mostra-se um trecho da nossa implementação na qual são usados meta-dados para dar uma dica do tipo de dados ao compilador e evitar que seja necessário inferir o tipo.

```
(defn ClojureSTMCompositePart-addAssembly
  [#^ClojureSTMCompositePart this
  #^BaseAssembly assembly]
  (dosync (alter (:usedIn @(.state this)) conj assembly)))
```

Figura 3.14: *Exemplo de uso de meta-dados para indicar ao compilador o tipo de dado do parâmetro (#^ClojureSTMCompositePart) que evita o acionamento do mecanismo de* reflection

3.4 Sistema de Memória Transacional de Clojure

Na sequência damos uma visão geral da implementação do sistema de memória transacional em software de Clojure.

3.4.1 Transações

As transações são iniciadas com a macro `dosync`. Quando uma transação é iniciada dentro de outra, esta se une à transação exterior. Isto significa que todas as transações internas irão fazer *commit* quando a transação externa o fizer também. Este comportamento corresponde a transações fechadas, como foi visto na seção 2.4.4.

A implementação do sistema de STM de Clojure baseia-se principalmente no conceito de `ref`, que como foi visto na seção 3.2.2, é o único tipo de dados cujo valor pode ser modificado (como se verá a seguir, estritamente, o valor não é modificado, mas é trocado), e isto só pode acontecer dentro de uma transação.

Em caso de conflito durante a modificação de uma referência, uma transação pode suspender sua execução e voltar a tentar (*retry*) depois. A cada tentativa a transação armazena um novo "ponto de leitura", que serve para manter uma ordem total de todas as transações iniciadas. Inspecionando o código fonte da linguagem, observa-se que o número máximo de vezes que uma transação pode voltar a tentar sua execução é 10 mil.

Modificação de uma (`ref`)

Leitura (dereferenciamento) de uma `ref` Uma `ref` pode ser lida (dereferenciada) dentro ou fora de uma transação. Quando a leitura da `ref` acontece fora de uma transação, o último valor modificado com sucesso após uma operação de *commit* é devolvido e a leitura não causa bloqueios de outras *threads* que tentem ler ou modificar a `ref`.

Se a `ref` é usada dentro de uma transação o processo é mais complexo: cada referência possui um histórico de valores que foram atribuídos com sucesso por meio de uma transação, se a transação possui valores anteriores à transação atual, devolve-se o mais recente, caso contrário, a transação voltará a tentar a execução posteriormente.

Substituição de uma `ref` As operações feitas para substituir-se o valor de uma ref devem ser todas feitas dentro de uma transação. A tentativa de alteração do valor em um contexto não transacional, causará uma exceção. A fim de fazer ênfase no fato de que os valores na linguagem Clojure são imutáveis, utiliza-se a palavra substituir, em lugar de modificar. Assim, o valor de uma ref é atomicamente substituído por um outro valor.

As seguintes operações podem ser usadas para substituir-se o valor de uma `ref` com o valor de outra.

- `ref-set` Para especificar diretamente o valor de uma `ref`
- `alter` Para especificar o novo valor de uma `ref`, por meio da aplicação de uma função ao seu valor atual.
- `commute` Para substituir o valor de uma `ref`, com o resultado obtido por meio da aplicação de uma função comutativa[1] ao seu valor atual. Esta operação tem um impacto positivo no desempenho, pois, como as funções comutativas não tem uma ordem estrita, a linguagem não precisa garantir que as funções são executadas na ordem em que são chamadas. É responsabilidade do programador garantir que a função fornecida seja comutativa.

[1] No contexto de Clojure consideram-se funções comutativas aquelas cujos resultados não dependen da ordem das chamadas

Capítulo 4

Ferramentas de *benchmark*

Com o surgimento de diversos sistemas de STM, como foi exposto na seção 2.2.2 surge a necessidade de comparar o desempenho deles. Guerraoui *et al.* (2007) observam a dificuldade de se realizar uma comparação entre estes sistemas pelo fato de serem escritos em linguagens de programação diversas ou serem executados em ambientes de execução personalizados. Ademais, os autores reconhecem uma escassez de *benchmarks* que forneçam cargas de trabalho realistas.

Sim *et al.* (2003) definem um *benchmark* como um teste ou conjunto de testes utilizados para comparar o desempenho de técnicas ou ferramentas. Neste capítulo serão apresentadas algumas ferramentas de *benchmark* para sistemas de memória transacional em software, com ênfase em STMBench7, cuja escolha é explicada na seção 4.2.

4.1 *Benchmark* para memória transacional

Encontram-se na literatura diversas propostas de ferramentas de *benchmark* para STM, porém, nenhuma destas é amplamente utilizada. Nielsen e Kristiansen (2009) realizam uma revisão de diferentes ferramentas de *benchmark* para memória transacional, identificando duas categorias principais: micro e macro-*benchmark*, em função do tamanho destes. Na primeira categoria consideram-se os *benchmarks* cujas operações são aplicadas em uma só estrutura de dados, como árvores rubro-negras, *hash-maps* ou listas ligadas; já na categoria de macro-*benchmark* encontram-se aquelas que utilizam estruturas de dados mais complexas.

4.1.1 STAMP

O *benchmark* STAMP (*Stanford Transactional Applications for Multi-Processing*) é um conjunto de oito aplicações:

- **bayes**, que implementa um algoritmo para a aprendizagem da estrutura de uma rede Bayesiana a partir de dados observados.
- **genome**, que a partir de um grande número de segmentos de DNA e realiza comparações a fim de reconstruir o genoma original.
- **intruder**, um sistema de detecção de intrusão em redes, baseado em assinatura, que examina pacotes de rede para compararlos com um conjunto conhecido de assinaturas de intrusão.

- **kmeans**, implementa o algoritmo *k-means*, que agrupa objetos num espaço N-dimensional em k grupos.
- **labyrinth**, que implementa uma variante do algoritmo de Lee, similar ao Lee-TM, descrito na seção 4.1.3. A estrutura de dados principal é uma grade uniforme tridimensional que representa o labirinto.
- **ssca2** (*Scalable Synthetic Compact Applications 2*) é uma aplicação que realiza operações sobre um grande multigrafo dirigido e com pesos, cuja aplicação vai desde a biologia computacional até a segurança.
- **vacation** implementa um sistema de processamento de transações online que simula um sistema de reservas de viagens.
- **yada** (*Yet Another Dalaunay Application*) implementa o algoritmo de Ruppert para refinamento de redes (*mesh*).

Cada um deles consta de aproximadamente 1000 linhas de código e é apresentado junto com uma implementação transacional e sequencial, mas não com uma implementação baseada em *locks* (Minh *et al.*, 2008).

4.1.2 Wormbench

Zyulkyarov *et al.* (2008) propuseram Wormbench, um *benchmark* que foi projetado para ter uma carga de trabalho configurável que consistisse no percorrido de um conjunto de vários *worms* dentro de uma estrutura de dados *BenchWorld*, baseado no jogo *Snake*. Possui um conjunto de 15 operações predefinidas que junto com a variação dos parâmetros: *comprimento* e *tamanho da cabeça do* work podem gerar um número grande de configurações para o *benchmark*.

Wormbench foi implementado na linguagem de programação C#, em aproximadamente 940 linhas de código. A implementação do *benchmark* possui dois tipos de sincronização, uma com *locks* globais e a outra baseada em transações, por meio de blocos atômicos.

4.1.3 Lee-TM

Lee-TM é um *benchmark* baseado no algoritmo de roteamento de circuitos de Lee. Ansari *et al.* (2008) apontam este algoritmo interessante para avaliação de memória transacional porque é um exemplo de uma aplicação do mundo real que contem bastante paralelismo potencial devido ao número de caminhos que devem ser realizados em um circuito real, sendo difícil de implementar utilizando *locks*.

O *benchmark* Lee-TM consiste em cinco implementações do algoritmo de roteamento de Lee

- **Sequencial** sem sincronização
- ***Locks* com granularidade grossa**
- ***Locks* com granularidade média**
- **Transacional**
- **Transacional otimizado**

4.1.4 STMBench7

Esta ferramenta de *benchmark* foi proposta por Guerraoui *et al.* (2007) com o objetivo de fornecer um ambiente com cargas de trabalho realistas e é desenvolvida com mais detalhe na seção 4.2

4.1.5 Benchmark para Clojure

O *benchmark* para memória transacional proposto por Nielsen e Kristiansen (2009) merece uma menção especial, pois foi implementado na linguagem Clojure. O objetivo deste *benchmark* foi avaliar o desempenho de diferentes gerenciadores de contenção propostos pelos mesmos autores, com a finalidade de melhorar o sistema de memória transacional de Clojure.

4.2 STMBench7

STMBench7 é um *benchmark* criado para avaliar implementações de memória transacional que consiste em um conjunto de estruturas de dados e as operações sobre elas, que pretendem modelar uma ampla variedade de cenários de execução que sejam complexos, multi-*thread* e orientadas a objetos, isto é, que consigam representar aplicações do mundo real, como sistemas de CAD, CAM e CASE (Guerraoui *et al.*, 2007).

A implementação de STMBench7 foi baseada nas estruturas de dados do *benchmark* OO7 que será apresentado na seção 4.2.1

4.2.1 OO7 Benchmark

STMBench7 é baseado em OO7, um *benchmark* que foi projetado com o intuito de comparar sistemas de bancos de dados orientados a objetos (Carey *et al.*, 1994).

Estrutura de dados

Na figura 4.1 mostra-se a estrutura de OO7. Esta consiste em um conjunto de *módulos*, cada um contendo uma árvore de *componentes*. Os nós internos da árvore são chamados de *componentes complexos* e as folhas, de *componentes base*. Cada um destes últimos contem várias *partes compostas*. Cada *parte composta* possui um *documento* e conexões com *partes atômicas*, via objetos *conexão*. Todos os elementos da estrutura de dados possuem acesso ao respectivo nó superior, o que permite realizar percorridos de cima para baixo e de baixo para cima.

Operações sobre a estrutura

OO7 inclui três tipos de operações:

- **percorridos** percorrem a estrutura de dados de cima para baixo começando pelo nó raiz, ou de baixo para cima, começando a partir de uma *parte atômica* (nó folha) aleatória. Geralmente percorrem um subconjunto grande de todos os objetos compartilhados.
- **consultas** procuram um subconjunto de objetos da estrutura utilizando um índice.

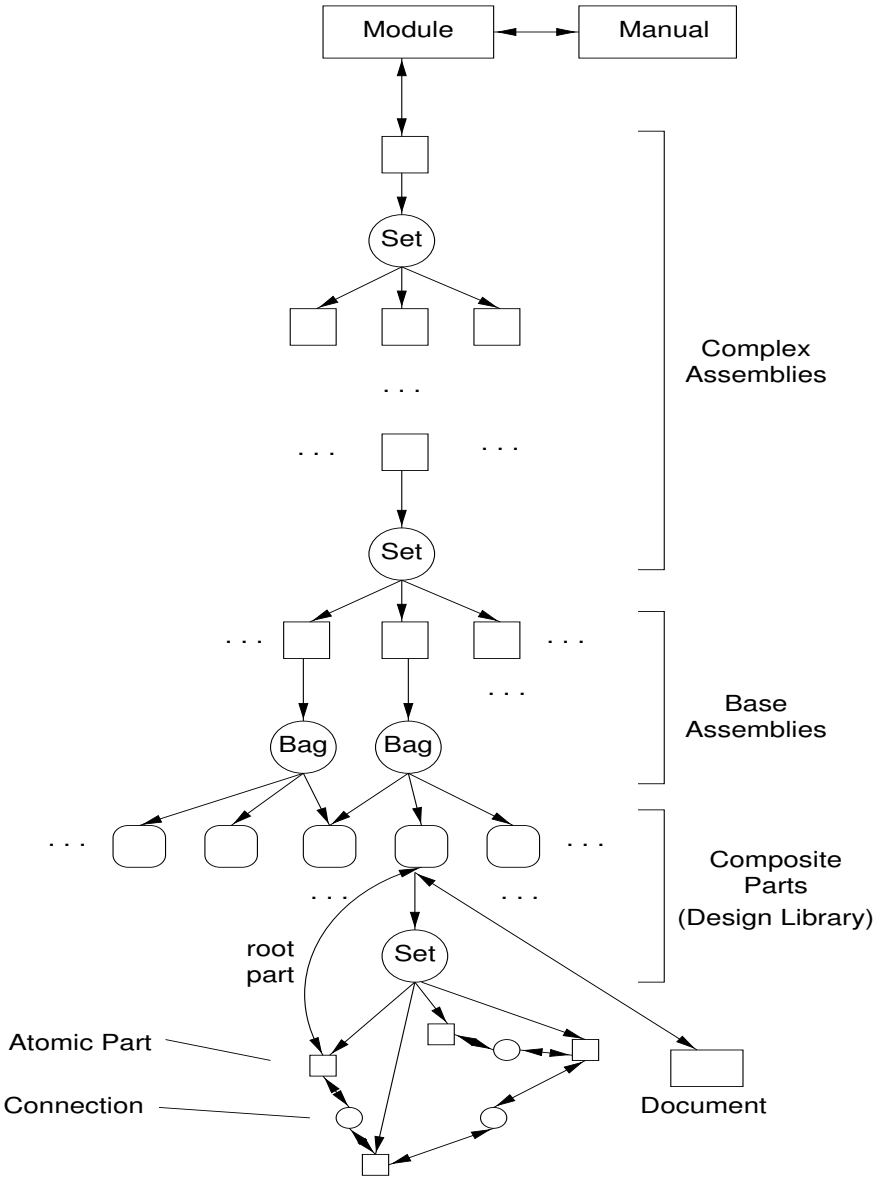

Figura 4.1: *Arquitetura do OO7 e STMBench7 (Guerraoui et al., 2007), pág 3*

- **modificações estruturais** criam e apagam *componentes base* e suas *partes compostas* descendentes.

4.2.2 Estrutura do STMBench7

STMBench7 utiliza a estrutura de dados de OO7, descrita na seção 4.2.1. Diferentemente de OO7, que foi projetado para avaliar o desempenho de transações isoladas, STMBench7 foi projetado levando-se em consideração a concorrência.

STMBench7 executa um conjunto de *threads* de forma concorrente, as quais realizam operações sobre a estrutura de dados descrita. Cada *thread* escolhe aleatoriamente uma operação de um total

de 45 que são implementadas no *benchmark*.

4.2.3 Operações do STMBench7

STMBench7 contem um total de 45 operações sobre a estrutura de dados utilizada. Estas operações estão agrupadas em quatro categorias, às quais são atribuídas algumas ponderações baseadas nas escolhas a mais alto nível, de forma que não fica sob a responsabilidade do usuário configurar todos os parâmetros destas operações. As categorias de operações contempladas neste *benchmark* são:

Percorridos longos Percorrem todos os componentes e/ou todas as partes atômicas. Algumas destas operações modificam *documentos* ou *partes atômicas*. Os nomes das operações desta categoria são marcadas com o prefixo T, como T1-T6 e adicionalmente Q6 e Q7.

Percorridos curtos Percorrem a estrutura escolhendo o caminho de forma aleatória, começando por um *módulo*, um *documento* ou uma *parte atômica*. Alguns destes utilizam índices. Um destes percorridos possui um comportamento diferente: itera sobre todos os *componentes* base e avalia alguns dos seus nós *partes compostas* descendentes. Os nomes das operações desta categoria são marcados com o prefixo ST, como ST1-ST10.

Operações curtas Escolhem um objeto ou conjunto de objetos na estrutura de dados e executam alguma operação sobre este objeto ou sobre sua vizinhança local. A escolha é realizada aleatoriamente ou atendendo a algum critério de busca, geralmente o próprio índice. Os nomes destas operações utilizam o prefixo OP, como OP1-OP15.

Operações de modificação de estrutura Aleatoriamente criam ou eliminam elementos da estrutura, ou enlaces entre elementos. Porém estas operações tem um alcance limitado, assim, não produzem uma degeneração na estrutura de forma significante. Os nomes destas operações são denotados pelo prefixo SM, como SM1-SM8.

4.2.4 Interface do STMBench7

STMBench7 consiste principalmente em uma complexa estrutura de dados e em operações realizadas sobre esta. Sendo que suas implementações encontram-se separadas, é possível estender o *benchmark* para interagir com qualquer implementação de STM.

Para tanto é necessário estender as estruturas de dados da implementação padrão de modo que se possa utilizar a sincronização baseada no sistema de STM a ser usado, bem como suas correspondentes fábricas.

A fim de estender estas estruturas de dados, a implementação de STMBench7 contém algumas anotações no código, como se mostra na sequência:

- **@Atomic** Objeto que pode ser lido ou modificado por uma transação
- **@Immutable** Objeto que não será modificado durante a execução do *benchmark*.
- **@Transactional** Método que será executado dentro de uma transação.

- **@ReadOnly** Método pertencente a um objeto @Atomic e que sobre este realiza apenas leitura.
- **@Update** Método pertencente a um objeto @Atomic e que sobre este opera modificações.

STMBench7 foi escrito em Java 5, e existe também uma implementação em C++.

Nota sobre a medição de tempos no STMBench7

Durante a avaliação e modificação de STMBench7, foi levantada uma questão de implementação que altera sensivelmente a medição dos resultados do *benchmark*. A observação consiste na forma de medir os tempos de execução de cada *thread*, o que se fazia utilizando-se a diferença simples entre o tempo final e o tempo inicial da execução de cada *thread*. Esta abordagem tem a desvantagem de incluir na medição o tempo gasto pelo sistema operacional em outros processos externos ao *benchmark*. Na versão 5 de Java, que é a versão utilizada nesta implementação, existe uma função que mede explicitamente o tempo de usuário (*user time*). Esta função é uma medida mais exata do tempo gasto por cada *thread* na sua execução. Como parte da implementação do presente trabalho, esta forma de medição de tempos foi modificada, introduzindo-se as mudanças anteriormente descritas.

Capítulo 5

Implementação

5.1 STMBench7 para Clojure

Como foi explicado no capítulo anterior, os *benchmarks* para STM existentes estão escritos principalmente nas linguagens Java, C, C++ e C#, já que estas linguagens são os principais destinos de implementação de sistemas de STM. Nielsen e Kristiansen (2009) apresentam um *benchmark* que a pesar de ter como foco a linguagem Clojure, não possui uma implementação que utilize *locks*, cuja comparação é um dos objetivos do nosso trabalho. Assim, mediante a ausência de um *benchmark* específico para a linguagem Clojure que se ajuste às nossas necessidades, neste trabalho fez-se uma adaptação do *benchmark* STMBench7, que foi apresentado na seção 4.2. O fato de STMBench7 ser objeto de interesse neste trabalho, é devido a duas características suas: A primeira, é a possibilidade de se realizar uma carga de trabalho que representa vários cenários de implementação de sistemas de software; a segunda característica importante é o por estar implementada na linguagem de programação Java, que é a mesma linguagem utilizada para a implementação de Clojure. Como foi apontado no capítulo 3, um fator que contribuiu na rápida popularização da linguagem Clojure é a facilidade de interação com código Java existente, como foi detalhado na seção 3.3 em Clojure é possível herdar classes Java existentes e ao mesmo tempo criar novas classes Java que possam ser utilizadas por outras aplicações. Estas duas propriedades foram utilizadas para a adaptação do STMBench7 que se realiza neste trabalho, como é explicado nas seções subsequentes.

5.2 Implementação

Nesta seção descreve-se a implementação realizada para dar suporte ao sistema de STM de Clojure no *benchmark* STMBench7.

5.2.1 Implementação das Fábricas (*Factories*)

Foram estendidas um total de três fábricas em STMBench7, com o objetivo de criar os objetos das classes que foram implementadas neste trabalho. As fábricas a seguir foram implementadas na linguagem Java, porém, as classes por elas criadas foram implementadas na linguagem Clojure, como se expõe na seção 5.2.2.

DesignObjFactory

Na classe `DesignObjFactory` são criados os componentes da estrutura de dados principal. O arcabouço fornece uma implementação padrão (que não se encontra pronta para trabalhar com *threads*) para cada uma das classes que compõem esta estrutura.

Na tabela 5.1 são resumidas as classes criadas por este `Factory`. As classes que implementamos encontram-se em negrito

Interface devolvida	Classe implementada	Implementação padrão
AtomicPart	**ClojureSTMAtomicPart**	AtomicPartImpl
BaseAssembly	**ClojureSTMBaseAssembly**	BaseAssemblyImpl
ComplexAssembly	**ClojureSTMComplexAssembly**	ComplexAssemblyImpl
CompositePart	**ClojureSTMCompositePart**	CompositepartImpl
Connection	ConnectionImpl	ConnectionImpl
Document	**ClojureSTMDocument**	DocumentImpl
Manual	ManualImpl	ManualImpl
Module	ModuleImpl	ModuleImpl

Tabela 5.1: *Classes implementadas na fábrica* DesignObj

BackendFactory

Na tabela 5.2 são resumidas as classes criadas por este `Factory`. As classes que implementamos encontram-se em negrito.

Interface devolvida	Classe implementada	Implementação padrão
IdPool	**ClojureSTMIdPool**	IdPoolImpl
Index	**ClojureSTMIndex**	IndexImpl
LargeSet	**ClojureSTMLargeSet**	TreeMapIndex

Tabela 5.2: *Classes implementadas na fábrica* Backend

OperationExecutorFactory

Interface devolvida	Classe implementada	Implementação padrão do arcabouço
OperationExecutor	**ClojureSTMOperationExecutor**	DefaultIdPoolImpl

Tabela 5.3: *Classes implementadas na fábrica* OperationExecutor

Outras classes implementadas

Ademais das classes anteriormente mencionadas, foram implementadass: ClojureSTMDesignObj, ClojureSTMAssembly e ImmutableCollectionAdaptor.

5.2.2 Implementação das Estruturas de Dados

As estruturas de dados fornecidas por STMBench7 precisam ser estendidas para dar suporte ao modelo de concorrência de cada Sistema de STM, para tanto, o *benchmark* utiliza anotações

de Java e com elas indica quais métodos das estruturas de dados devem ser modificados. Como foi descrito na seção 3.4.1, no sistema de STM de Clojure uma transação é executada por meio da função `dosync`. A linguagem Clojure fornece um conjunto de estruturas de dados imutáveis com suporte de modificação transacional. A fim de utilizar-se estas estruturas de dados, foram criadas classes que fazem a interface entre as estruturas de dados requeridas no STMBench7 e as estruturas de dados fornecidas pela linguagem Clojure. Estas classes foram escritas em Clojure e fazem uso de operações de sincronização nativas desta linguagem, como `dosync`, `alter`, `ref-set`, que foram descritas na seção 3.4.1.

5.2.3 Geração da biblioteca de classes

A linguagem Clojure fornece função `gen-class` que permite a geração de classes em código de *bytes*, que são interpretadas pela máquina virtual de Java e podem interagir com outras aplicações. Neste trabalho, foi utilizada esta função para gerar uma biblioteca com as classes implementadas para fazer uma interface entre as classes requeridas por STMBench7 e as fornecidas por Clojure. Esta biblioteca posteriormente foi importada na ferramenta de *benchmark* STMBench7 para realizar-se os testes deste trabalho.

Capítulo 6

Resultados

6.1 Testes realizados

Foi executado o **benchmark** STMBench7 com o conjunto de estruturas de dados implementados para fazer interface com o sistema de memória transacional de Clojure.

Como foi exposto no capítulo 4, STMBench7 vem configurado com três tipos de carga de trabalho: com predominância de operações de leitura, de escrita e de ambas. O *benchmark* foi executado utilizando as três configurações de sincronização, combinadas com três configurações do número de *threads*, em número de 2, 4 e 8.

Adicionalmente, para monitorar e identificar os pontos de maior sobrecarga, foi utilizada a ferramenta de *profiling* Yourkit.

6.1.1 Entorno utilizado

O STMBench7 modificado para dar suporte a Clojure foi executado num computador com as seguintes características: processador Intel(R) Core(TM) i7 2.67GHz com sistema operacional Ubuntu 10.04.2 LTS. A versão da máquina virtual de Java utilizada foi 1.6.0_23 para 64 bits.

6.1.2 Métricas

Para cada cenário de execução, foi considerada a medida de *operações por segundo*. Já que todos os testes foram executados com o mesmo parâmetro de tempo, a diferença principal entre eles radica no número de operações que conseguem executar neste tempo.

6.2 Análise dos resultados

A comparação do número de operações por segundo após a execução do *benchmark* mostra uma diferença substancial entre o desempenho utilizando *locks* de granularidade média e grossa, em comparação ao sistema de memória transacional de Clojure.

Nas figuras 6.1, 6.2 e 6.3, mostra-se a diferença entre o rendimento utilizando os três tipos de sincronização.

Com o objetivo de identificar os pontos causantes de tal sobrecarga foi utilizada a ferramenta de *profiling* Yourkit. Na tabela 6.2 mostram-se os métodos com maior sobrecarga.

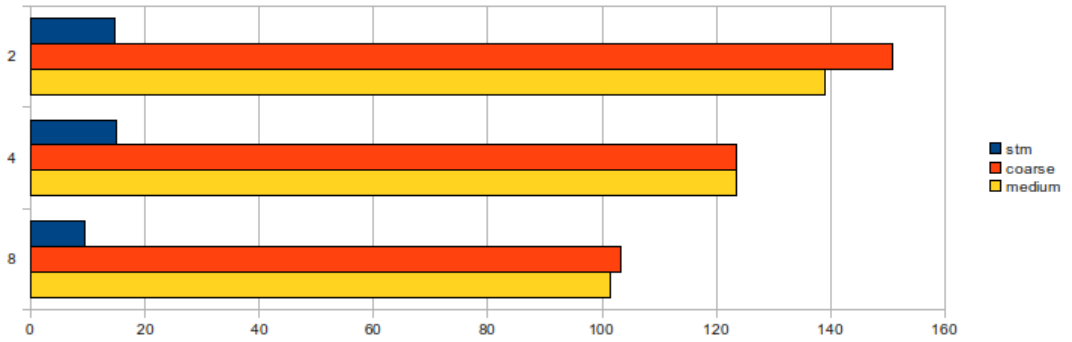

Figura 6.1: *Comparação de desempenho (operações por segundo) entre os três métodos de sincronização, utilizando carga de trabalho predominante de* ***leitura***

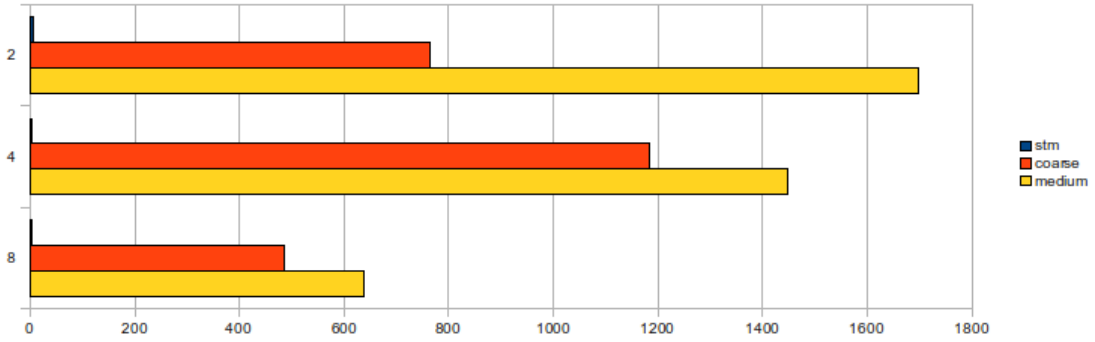

Figura 6.2: *Comparação de desempenho (operações por segundo) entre os três métodos de sincronização, utilizando carga de trabalho predominante de* ***escrita***

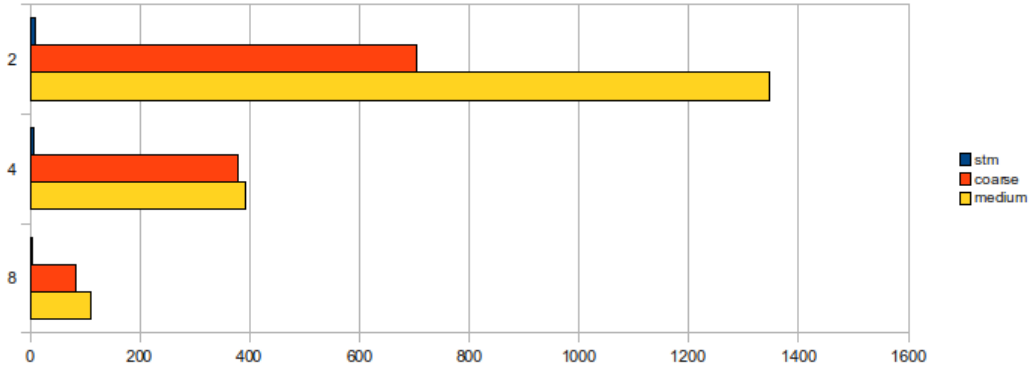

Figura 6.3: *Comparação de desempenho (operações por segundo) entre os três métodos de sincronização, utilizando carga de trabalho predominante de* ***leitura e escrita***

Na tabela 6.1 os métodos traverse da classe Traversal1 são os que consomem maior tempo. Esta classe é uma das operações de percorrido do *benchmark*, pelo que não causa surpresa que seja responsável do maior consumo de tempo. Sendo que um destes métodos chama o outro, pode se considerar como a mesma. Com uma sobrecarga menor, encontram-se os métodos de acesso aos nós filhos, que é uma operação muito comum no *benchmark*. Podemos concluir então que **o maior tempo consumido com esta estratégia de *locks* com granularidade média, é empregado pelas próprias operações do *benchmark***

	Método	% de tempo
M1	Traversal1.traverse(BaseAssembly)	99 %
M2	Traversal1.traverse(AtomicPart, HashSet)	95 %
M3	AtomicPartImpl.getToConnections()	28 %
M4	ImmutableCollectionImpl.<init>(Iterable)	28 %

Tabela 6.1: *Métodos que registram maior tempo de execução no benchmark utilizando* locks *com granularidade média*

Na tabela 6.2 os métodos etiquetados como S3, S4, S5, S6 e S7 são chamados dentro da mesma operação de percorrido que no caso anterior, porém, o método etiquetado como S8, `Ref.deref`, (`Ref.deref` chama a `LockingTransaction.doGet` e portanto pode se considerar o mesmo) é uma função de Clojure utilizada para dereferenciar uma `ref`. Como se viu na seção 3.4.1, este é o mecanismo de Clojure para realizar mudanças atômicas em dados. Este método constitui uma grande fonte de sobrecarga (29%).

Uma outra fonte de sobrecarga é observado no método `Reflector.getMethods()` (S12 na tabela) que utiliza *Java Reflection* para obter a lista de métodos de um objeto e verificar se é compatível com a chamada realizada. Isto também é uma grande fonte de sobrecarga durante o *benchmark* (25%). Ambas correspondem a funções básicas do Clojure.

Assim, como **principais responsáveis da sobrecarga** no *benchmark* com memória transacional, temos por um lado, o próprio mecanismo de **acesso às referências da linguagem Clojure**, e por outro, ao uso intensivo de **inferência de tipos** que utiliza a implementação da linguagem Clojure, considerando que na implementação realizada utilizaram-se sempre as dicas de tipo de dados para ajudar ao interpretador la linguagem na tarefa de inferir os tipos de dados.

	Método	% de tempo
S1	LockingTransaction.run(Callable)	99 %
S2	Reflector.invokeMatchingMethod(...)	98 %
S3	Traversal1.traverse(ComplexAssembly)	89 %
S4	Traversal1.traverse(Assembly)	89 %
S5	Traversal1.traverse(BaseAssembly)	88 %
S6	Traversal1.traverse(CompositePart)	87 %
S7	Traversal1.traverse(AtomicPart, HashSet)	87 %
S8	clojure.lang.Ref.deref()	33 %
S9	clojure.lang.LockingTransaction.doGet(Ref)	29 %
S10	ClojureSTMAtomicPart_getToConnections()	29 %
S11	BaseOperation.addAtomicPartToBuildDateIndex()	27 %
S12	clojure.lang.Reflector.getMethods()	25 %

Tabela 6.2: *Métodos que registram maior tempo de execução no benchmark utilizando Clojure STM*

Capítulo 7

Conclusões

Com este trabalho pretendia-se saber se o uso de Clojure como uma biblioteca de sincronização na linguagem Java teria um desempenho favorável, em comparação com a sincronização com *locks*. Este experimento baseia-se na pretensão de se escrever bibliotecas inteiras na linguagem Clojure e poder utilizá-las na plataforma Java, com o que se aproveitaria a facilidade de escrever código em Clojure.

Chegou-se à conclusão de que a sobrecarga imposta pela linguagem Clojure utilizada como biblioteca é demasiado alta para tornar-se uma alternativa viável em implementações do mundo real, dado que a nossa motivação inicial era a de aproveitar o sistema de sincronização de Clojure.

Acreditamos que a utilização do sistema de memória transacional de Clojure possa ser usado na implementação de *provas de conceito* nas quais exista uma grande quantidade de código em Java a ser aproveitado. Em um cenário como este, o fato de implementar a sincronização na linguagem Clojure facilita enormemente um rápido desenvolvimento da aplicação.

7.1 Considerações finais

Atualmente um tema contínuo de pesquisa é o projeto de novas técnicas de memória transacional, em diversas linguagens e plataformas. Paralelamente a estas, encontram-se as pesquisas que buscam a formulação de ferramentas de *benchmark* que sirvam para avaliar o crescente número de sistemas de STM.

Os maiores desafios para um novo *benchmark* são: fornecer uma carga de trabalho suficientemente diversa para poder representar um maior espectro de aplicações do mundo real; e por outro lado, possuir implementações na maior quantidade de linguagens e/ou plataformas possíveis.

Assim, por exemplo, para um *benchmark* como STMBench7, que foi implementado na linguagem Java, ao querermos portá-lo para Clojure temos duas alternativas: reescrevê-lo completamente em Clojure ou aproveitar que Clojure funciona na plataforma Java e só implementar uma camada de interface entre ambos. Caso esta última opção seja menos custosa em termos de esforço de implementação, seria uma tarefa relativamente simples escrever uma camada de interface para outras linguagens executadas na plataforma Java, como Scala, Jyton, JRuby.

7.2 Sugestões para pesquisas futuras

Continuando com o raciocínio da seção anterior, a implementação realizada neste trabalho com o objetivo de criar uma interface entre STMBench7 e Clojure, pode ser tomada como inspiração para portar o *benchmark* a mais linguagens que funcionam na plataforma Java e assim fortalecê-lo.

A implementação feita neste trabalho, conjuntamente com o benchmark *STMBench7* pode ser diretamente usada para comparar o sistema de memória transacional (STM) de Clojure com outro sistema de STM implementado em Java, ou ainda, com o sistema de STM de uma outra linguagem, desde que esta seja previamente portada, como apontado anteriormente.

Referências Bibliográficas

Adl-Tabatabai *et al.*(**2007**) Ali-Reza Adl-Tabatabai, Christos Kozyrakis, e Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1189276.1189288. Citado na pág. 8

Ansari *et al.*(**2008**) Mohammad Ansari, Christos Kotselidis, Ian Watson, Chris Kirkham, Mikel Luján, e Kim Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. Em Anu Bourgeois e S. Zheng, editors, *Algorithms and Architectures for Parallel Processing*, volume 5022 of *Lecture Notes in Computer Science*, páginas 196–207. Springer Berlin - Heidelberg. URL http://dx.doi.org/10.1007/978-3-540-69501-1_21. Citado na pág. 24

Blanchet(**2003**) Bruno Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25:713–775. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/945885.945886. URL http://doi.acm.org/10.1145/945885.945886. Citado na pág. 6

Blundell *et al.*(**2006**) C. Blundell, E.C. Lewis, e M.M.K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):65. Citado na pág. 8

Boehm(**1981**) Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, New York. Citado na pág. 13

Carey *et al.*(**1994**) Michael J. Carey, David J. DeWitt, e Jeffrey F. Naughton. The OO7 benchmark. páginas 12–21. Citado na pág. 25

Cascaval *et al.*(**2008**) C. Cascaval, C. Blundell, M. Michael, H.W. Cain, P. Wu, S. Chiras, e S. Chatterjee. Software transactional memory: why is it only a research toy? *Communications of the ACM*, 51(11):40–46. Citado na pág. 11

Chung *et al.*(**2006**) J.W. Chung, C.C. Minh, A. McDonald, T. Skare, H. Chafi, B.D. Carlstrom, C. Kozyrakis, e K. Olukotun. Tradeoffs in transactional memory virtualization. Em *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, página 381. ACM. Citado na pág. 4

Dongarra *et al.*(**2007**) Jack Dongarra, Dennis Gannon, Geoffrey Fox, e Ken Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1). URL http://www.ctwatch.org/quarterly/articles/2007/02/the-impact-of-multicore-on-computational-science-software/. Citado na pág. 1

Grossman *et al.*(**2006**) D. Grossman, J. Manson, e W. Pugh. What do high-level memory models mean for transactions? Em *Proceedings of the 2006 workshop on Memory system performance and correctness*, página 69. ACM. Citado na pág. 8

Guerraoui *et al.*(**2007**) Rachid Guerraoui, Michal Kapalka, e Jan Vitek. Stmbench7: a benchmark for software transactional memory. Em *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, páginas 315–324, New York, NY, USA. ACM. ISBN 978-1-59593-636-3. doi: http://doi.acm.org/10.1145/1272996.1273029. URL http://doi.acm.org/10.1145/1272996.1273029. Citado na pág. xi, 23, 25, 26

Harris e Fraser(2003) T. Harris e K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):402. Citado na pág. 5

Harris *et al.*(2005) T. Harris, S. Marlow, S. Peyton-Jones, e M. Herlihy. Composable memory transactions. Em *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, páginas 48–60. ACM. Citado na pág. 6, 7

Harris(2005) Tim Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325 – 343. ISSN 0167-6423. doi: DOI:10.1016/j.scico.2005.03.005. URL http://www.sciencedirect.com/science/article/B6V17-4GCX1B5-2/2/7ec3ebeb6b11d7202fae3c01f27d8cda. Special Issue on Concurrency and synchonization in Java programs. Citado na pág. 14

Harris *et al.*(2006) Tim Harris, Mark Plesko, Avraham Shinnar, e David Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41:14–25. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1133255.1133984. URL http://doi.acm.org/10.1145/1133255.1133984. Citado na pág. 6

Herlihy *et al.*(2003) M. Herlihy, V. Luchangco, M. Moir, e W.N. Scherer III. Software transactional memory for dynamic-sized data structures. Em *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, páginas 92–101. ACM. Citado na pág. 5

Herlihy e Moss(1993) Maurice Herlihy e J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/173682.165164. Citado na pág. 4

Jones(2007) Simon P. Jones. *Beautiful Concurrency*. O'Reilly Media, Inc. ISBN 0596510047. URL http://research.microsoft.com/Users/simonpj/papers/stm/index.htm. Citado na pág. 9

Larus e Rajwar(2006) James R. Larus e Ravi Rajwar. *Transactional Memory*. Morgan & Claypool. Citado na pág. 4, 5, 6, 8, 10

Lomet(1977) D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/390017.808319. Citado na pág. 4

Minh *et al.*(2008) Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, e K. Olukotun. Stamp: Stanford transactional applications for multi-processing. Em *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, páginas 35–46. doi: 10.1109/IISWC.2008.4636089. Citado na pág. 1, 24

Moravan *et al.*(2006) M.J. Moravan, J. Bobba, K.E. Moore, L. Yen, M.D. Hill, B. Liblit, M.M. Swift, e D.A. Wood. Supporting nested transactional memory in LogTM. Em *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, página 370. ACM. Citado na pág. 9

Moss e Hosking(2006) J.E.B. Moss e A.L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201. Citado na pág. 9

Nasir(2009) Muhammad Nasir. Software transactional memory techniques. principles, design and implementation trade-offs. Dissertação de Mestrado, School of Computing, Blekinge Institute of Technology, Sweden. Citado na pág. 5

Nielsen e Kristiansen(2009) Peder R. L. Nielsen e Patrick T. Kristiansen. Benchmarking contention management strategies in Clojure's Software Transactional Memory implementation. Dissertação de Mestrado, Computer Science Department, Aalborg University, Denmark. Citado na pág. 23, 25, 29

Scott(2005) Michael L. Scott. *Programming Language Pragmatics, Second Edition*. Morgan Kaufmann. ISBN 0126339511. URL http://www.worldcat.org/isbn/0126339511. Citado na pág. 7, 14

Shavit e Touitou(1995) Nir Shavit e Dan Touitou. Software transactional memory. Em *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, páginas 204–213, Ottawa, Ontario, Canada. Citado na pág. 5

Silberschatz *et al.***(2006)** Abraham Silberschatz, Henry Korth, e S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA. ISBN 0072958863, 9780072958867. Citado na pág. 3

Sim *et al.***(2003)** Susan Elliott Sim, Steve Easterbrook, e Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. Em *in Proceedings of the 25th International Conference on Software Engineering*, páginas 74–83. Citado na pág. 23

VanderHart e Sierra(2010) Luke VanderHart e Stuart Sierra. *Practical Clojure*. Apress, Berkely, CA, USA, 1st edição. ISBN 1430272317, 9781430272311. Citado na pág. xi, 9, 13, 16

Zyulkyarov *et al.***(2008)** Ferad Zyulkyarov, Adrian Cristal, Sanja Cvijic, Eduard Ayguade, Mateo Valero, Osman Unsal, e Tim Harris. Wormbench: a configurable workload for evaluating transactional memory systems. Em *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, MEDEA '08, páginas 61–68, New York, NY, USA. ACM. ISBN 978-1-60558-243-6. doi: http://doi.acm.org/10.1145/1509084.1509093. URL http://doi.acm.org/10.1145/1509084.1509093. Citado na pág. 24