UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de São José do Rio Preto

Bruno Chinelato Honorio

Melhorando o Desempenho de Aplicações Transacionais Através de Anotações do Programador

Rio Claro - SP
2018

Bruno Chinelato Honorio

Melhorando o Desempenho de Aplicações Transacionais Através de Anotações do Programador

Orientador: Prof. Dr. Alexandro José Baldassin

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Geociências e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de Rio Claro.

Financiadora: FAPESP – Proc.2016/12103-7

Rio Claro – SP
2018

Honorio, Bruno Chinelato.
Melhorando o desempenho de aplicações transacionais através de anotações do programador / Bruno Chinelato Honorio. -- Rio Claro, 2018
52 f. : il., tabs.

Orientador: Alexandro José Baldassin
Dissertação (mestrado) – Universidade Estadual Paulista "Júlio de Mesquita Filho", Instituto de Geociências e Ciências Exatas

1. Ciência da computação. 2. Programação paralela. 3. Compiladores (Computadores) I. Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Geociências e Ciências Exatas. II. Título.

CDU – 518.721

Ficha catalográfica elaborada pela Biblioteca do IBILCE
UNESP - Câmpus de São José do Rio Preto

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Pós-Graduação em Ciência da Computação

Melhorando o Desempenho de Aplicações Transacionais Através de Anotações do Programador

Bruno Chinelato Honorio

9 de agosto de 2018

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Geociências e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de Rio Claro.
Financiadora: FAPESP – Proc.2016/12103-7

Banca Examinadora:

- Prof. Dr. Alexandro José Baldassin (Orientador)
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista – UNESP
- Prof. Dr. Daniel Carlos Guimarães Pedronette
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista – UNESP
- Prof. Dr. Márcio Merino Fernandes
  Departamento de Computação
  Universidade Federal de São Carlos – UFSCar

Rio Claro – SP
9 de agosto de 2018

AGRADECIMENTOS

Agradeço a FAPESP - Fundação de Amparo a Pesquisa do Estado de São Paulo pela bolsa concedida , processo 2016/12103-7.

RESUMO

Memória Transacional (*Transactional Memory* – TM) possibilita que programadores utilizem-se do conceito de *transação* na escrita de código concorrente. Nesse contexto, uma transação pode ser entendida como um bloco de instruções que é executado atomicamente e de forma isolada, ou seja, os estados intermediários no processamento de uma transação não são vistos pelas demais. Embora inicialmente confinada ao ambiente acadêmico, TM está se tornando cada vez mais popular. Prova disto é a adição de hardware transacional aos novos processadores da Intel e IBM, além de suporte para codificação de transações provido por compiladores como o GCC. A grande vantagem do modelo transacional é o maior nível de abstração fornecido ao programador, facilitando a escrita de programas concorrentes e evitando erros de sincronização famosos causados pelas travas (*locks*), como o *deadlock*. Infelizmente, o suporte em software para execução de transações ainda não provê desempenho muito bom. Em particular, o código transacional, produzido por compiladores e o sistema de tempo de execução associado, ainda pode ser considerado ineficiente. Nesta dissertação é realizado um estudo atualizado sobre a geração de código transacional do compilador GCC com o objetivo de encontrar a razão da deficiência de desempenho do compilador. O trabalho feito indica que uma das principais fontes de ineficiência são as barreiras de leitura e escrita inseridas pelo compilador. O problema dessa instrumentação acontece quando o compilador não consegue determinar, em tempo de compilação, se uma região de memória será acessada concorrentemente ou não, forçando o compilador a tomar uma decisão pessimista e instrumentar essa região de memória. Esse fenômeno é chamado de instrumentação excessiva. Para superar essas limitações, esta dissertação propõe uma nova construção de linguagem através de uma nova cláusula pragma que permite que programadores especifiquem quais regiões de memória não precisam ser instrumentadas. Para validar a nova cláusula pragma, esta dissertação conduziu experimentos usando o pacote STAMP, composto por aplicações transacionais. Os resultados obtidos mostram um grande ganho de desempenho para as aplicações que usaram o pragma proposto, com essas aplicações sendo até $7.2x$ mais rápidas que o código original gerado pelo GCC.

Palavras-chave: Memória Transacional, Programação Paralela, Compiladores, Otimização, Instrumentação Excessiva.

ABSTRACT

Transactional Memory (TM) allows programmers to utilize the concept of transaction for writing concurrent code. In this context, a transactioncan be extended as a block of instructions that is executed atomically and isolated, that is, the intermediate states of the processing of a transaction can not be seen by the other transactions. Although initially confined to the academic field, TM is becoming more popular. An evidence of this is the addition of transactional hardware to the new processors from Intel and IBM, as well as the support for transactional code provided by compilers such as GCC. The biggest advantage to the transactional model is the bigger level of abstraction provided to the programmer, making the process of writing parallel code easier, as well as avoiding famous synchronization errors caused by traditional locks, such as the deadlock problem. Unfortunately, the software support for execution of transaction still does not provide a good performance. In particular, transactional code, produced by compilers and the associated runtime system, can still be considered inefficient. This thesis performs an up-to-date study of the GCC transactional code generation and with the objective to find where the main performance losses are coming from. The study done indicates that one of the main sources of inefficiency is the read and write barriers inserted by the compiler. The problem of this instrumentation is that the compiler cannot determine, at compile time, if a memory region will be accessed concurrently or not, forcing the compiler to take a pessimist approach and instrument this memory region. This phenomenom is called Over-instrumentation. To overcome these limitations, this thesis proposes a new language construct through a new pragma clause that allows programmers to specify which memory regions do not need to be instrumented. To validade the new pragma clause, this thesis conducted experiments using the STAMP benchmark suite, composed of transactional applications. The obtained results show a great performance gain for applications that used the proposed pragma, with them being up to $7.2x$ faster than the original code generated by GCC.

Keywords: Transactional Memory, Parallel Programming, Compilers, Over-instrumentation, Optimization.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A utilização dos processadores *multicore* como solução para o reduzido ganho de desempenho e alta dissipação de calor utilizando paralelismo em nível de instrução (ILP - *Instruction Level Parallelism*) condicionou o desenvolvimento de novas aplicações necessitar da elaboração e codificação de algoritmos que levem em conta um modelo de computação paralelo.

Apesar da busca por alto desempenho agora ser direcionada por este novo paradigma, as ferramentas disponíveis para escrita e depuração de programas concorrentes ainda são muito precárias [24]. As principais construções paralelas ainda são baseadas em bloqueios (*locks*) e variáveis de condição (*condition variables*), que foram elaboradas nas décadas de 1960 e 1970, e creditadas a Dijkstra [4] e Hoare [14]. O fato desses modelos ainda estarem sendo utilizados não só evidencia uma certa ausência de novos modelos de programação paralela, mas também como eles têm se tornado obstáculos para o desenvolvimento de grandes aplicações, tanto em termos de desempenho, como na complexidade de programação.

Esse fator, por sua vez, fez com que ambas Academia e Indústria não poupassem esforços para desenvolver métodos mais apropriados para o desenvolvimento de software concorrente. Um dos resultados desses esforços é um mecanismo conhecido como *Memória Transacional* (*Transactional Memory* – TM) [9]. A memória transacional tem como principal conceito a transação: uma região de código que é executada atomicamente e de forma isolada. Desta forma, o programador precisa apenas se preocupar em explicitar o trecho de código que envolverá uma região compartilhada de memória, sendo então responsabilidade da implementação da estrutura transacional garantir atomicidade e isolamento.

A infraestrutura transacional pode ser implementada em hardware (HTM), software (STM), ou de maneira híbrida (HyTM), com diferentes compromissos. O modelo transacional começou a ser explorado em 1993 por Herlihy e Moss [12], os quais propuseram uma implementação em hardware. As abordagens em software começaram a ser exploradas com maior sucesso em 2003, a partir do trabalho de Herlihy et al. [11]. Eles mostraram

como implementar transações dinâmicas utilizando-se da sincronização livre de obstrução (*obstruction-free synchronization*). A partir de então um grande número de trabalhos propuseram [6] outros tipos de implementação em software, dos quais a abordagem baseada no uso de travas e registros de posse se sobressaíram.

Embora atualmente os novos microprocessadores da Intel [16] e IBM [25, 17, 18] possuam suporte nativo para execução de transações em hardware, as abordagens em software continuam sendo importantes. Isso deve-se ao fato de o suporte em hardware ter sido adicionado como uma abordagem de melhor-esforço (*best-effort*), na qual não há nenhuma garantia da transação ser efetivada em hardware. Nesse caso, um componente em software é responsável pela garantia de progresso do sistema.

Atualmente alguns compiladores já possuem suporte para geração e depuração de código transacional. O compilador deve gerar o código para iniciar/finalizar a transação, além de instrumentar todo o acesso às variáveis compartilhadas. O compilador estudado neste trabalho que gera código transacional (i.e, GNU C Compiler – GCC) utiliza-se de uma interface binária de aplicação (*Application Binary Interface* – ABI) criada pela Intel [15]. Esse documento define uma interface entre o código gerado para arquiteturas IA32 e a biblioteca de memória transacional, de forma que o mesmo código de usuário possa ser ligado a diferentes bibliotecas transacionais em tempo de execução sem a necessidade de recompilação do código. A grande maioria das bibliotecas de TM atuais, como a TinySTM [7] e o NORec [3], aderem a essa ABI criada pela Intel.

Apesar dos esforços levantados na última década, memória transacional ainda não é usada normalmente no desenvolvimento de aplicações concorrentes comerciais. Uma das principais razões é por conta da ineficiência que os compiladores têm apresentado ao gerar código transacional graças ao fenômeno chamado instrumentação excessiva (over-instrumentation) [28, 5]. Ao gerar código transacional, um compilador instrumenta todos os acessos (leitura e escrita) à memória compartilhada através da inserção de barreiras no código gerado. Essas barreiras são usadas por bibliotecas de tempo de execução TM para detectar conflitos nas transações e realizar o controle de concorrência. Como compiladores geralmente têm informações limitadas durante o tempo de compilação, eles devem ser conservadores em sua análise e inserir barreiras mesmo em situações que não seria necessário.

1.1 Objetivos e Contribuições

De maneira a esclarecer o cenário atual das ineficiências dos compiladores ao gerar código transacional, este trabalho realiza uma análise completa de aplicações transacionais e propõe um novo suporte de linguagem para a omissão de barreiras. Em particular, este trabalho realizou uma exaustiva análise sobre o compilador GCC, usando diversas bibliotecas transacionais de tempo de execução. O suporte de linguagem criado foi feito

através de *pragmas* para elidir a instrumentação de variáveis que sofrem de instrumentação em excesso, permitindo que programadores experientes melhorem o desempenho de aplicações transacionais.

Este trabalho oferece as seguintes contribuições:

- Realiza um estudo atualizado sobre a geração de código transacional do compilador GCC com o objetivo de encontrar a razão da deficiência de desempenho do compilador;
- Propõe uma nova construção de linguagem que permite que programadores instruam o compilador a elidir barreiras para certas regiões de memória;
- Mostra o ganho de desempenho oferecido pela omissão de barreiras de algumas aplicações do benchmark STAMP.

Este texto está dividido da seguinte forma. O Capítulo 2 apresenta uma definição geral sobre memória transacional, incluindo seus benefícios e detalhes dos tipos de implementação em hardware e software. O Capítulo 3 descreve a ABI da Intel e o funcionamento do suporte transacional no GCC. O Capítulo 4 foca na implementação completa do novo suporte de linguagem proposto por esse trabalho. O Capítulo 5 mostra os resultados dos experimentos realizados e o Capítulo 6 é a conclusão do trabalho.

Capítulo 2

Memória Transacional

O conteúdo desse capítulo é uma visão geral do panorama atual de programação paralela e seus conceitos gerais, a definição de memória transacional e características de seu projeto, além dos diversos tipos de implementação.

2.1 Conceitos Gerais de Programação Concorrente

Para introduzir o conceito de memória transacional de maneira apropriada, alguns conceitos e observações devem ser apresentados antes. O modelo de programação paralela discutido nesse trabalho é o modelo baseado em memória compartilhada (diferente do modelo baseado em troca de mensagens) [13], já que este é o modelo empregado pelas transações em memória. Neste modelo, a unidade básica de execução é chamada de *thread*. Um processo pode ser composto por uma ou mais threads, com cada thread tendo seu próprio contexto de execução e todas as threads compartilhando o espaço de endereçamento do processo. Threads são assíncronas, onde cada uma possui diferentes velocidades de processamento e podem ser, a qualquer instante, interrompidas por tempo indeterminado.

Em geral, uma aplicação paralela é inicializada por uma única thread. Eventualmente, e de acordo com a lógica da aplicação, novas threads são disparadas. Se a aplicação não exigir que diferentes threads acessem dados em comum, a execução da aplicação pode prosseguir de maneira natural. Problemas desse tipo são conhecidos na literatura como embaraçosamente paralelos (*embarrassingly parallel*) [13] e ocorrem com mais frequência em aplicações científicas e gráficas. O grande desafio em programação paralela aparece em situações onde diferentes threads necessitam acessar a mesma região de memória. Nesta situação, para impedir que acessos concorrentes gerem inconsistências, são necessários mecanismos de sincronização. Utilizar esses mecanismos de forma eficiente apresenta-se como um grande desafio.

Duas ou mais threads acessando a mesma região de memória podem potencialmente gerar inconsistência de dados se o acesso não for realizado de maneira correta. Considere

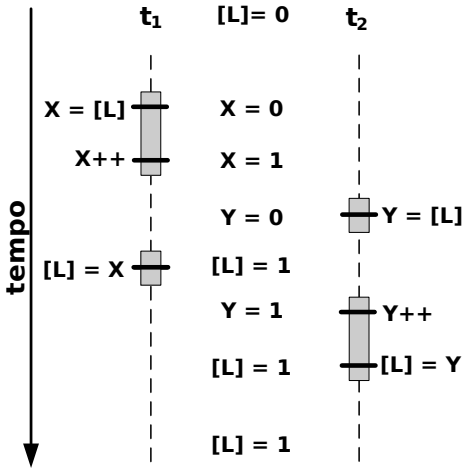

Figura 2.1: Sequência de operações levando a um resultado inconsistente para L

o seguinte exemplo ilustrado pela Figura 2.1: duas threads, t1 e t2, precisam incrementar em um (1) o valor contido em um endereço de memória L, inicialmente zero (0). Esta tarefa requer que o valor em L seja lido para uma variável local das threads. Cada thread deve incrementar a variável local e em seguida escrever em L novamente. Sem qualquer sincronização, é possível que a ordem de execução gerada acarrete no valor final em L ser inconsistente. Note que o valor final correto para L deveria ser dois (2), mas a sequência de execução gerou o valor um (1).

O problema se deve ao fato de o acesso ao dado compartilhado não ser controlado, gerando uma condição de competição (*race condition*). O objetivo da sincronização é restringir a ordem em que as operações de diferentes threads são executadas, evitando assim essas inconsistências.

2.2 O que é Memória Transacional

A existência de novas abordagens de programação paralela, como a memória transacional, deve-se ao fato das dificuldades apresentadas pelo método tradicional de programação paralela, que utiliza bloqueios. As subseções a seguir apresentam as dificuldades da programação paralela utilizando bloqueios e a definição geral de memória transacional, além das vantagens que a mesma apresenta sobre os métodos tradicionais de programação paralela.

2.2.1 Programação Paralela Utilizando Bloqueios

As primitivas de sincronização bloqueantes são essencialmente baseadas em bloqueios (*locks*) e variáveis de condição (*condition variables*). Ambas as primitivas têm cerca de 40 anos desde sua criação [4] [14].

O uso de bloqueios para realizar sincronização requer muita atenção do programador,

visto que este deve identificar corretamente a região de memória compartilhada e aplicar a operação de bloqueio, garantindo então somente a uma *thread* o acesso à região de memória, forçando as demais *threads* a aguardarem pela disponibilidade de acesso. O acesso é liberado somente após a liberação do bloqueio pela *thread* que está acessando a região de memória.

Essa exigência requerida pelo uso de bloqueios torna a programação paralela uma atividade difícil e laboriosa, principalmente para programadores inexperientes. Algumas das dificuldades que são apontadas pelo uso de bloqueios são [13] :

Associação entre bloqueio e dado - não é clara a natureza da relação entre os bloqueios e os dados que devem ser protegidos. É necessário dizer tanto *o que* deve ser sincronizado quanto *como* essa sincronização é feita. Deve ser adotada uma convenção e garantir que essa seja seguida, dificultando a manutenção do ćodigo.

Granularidade e desempenho - associar um único bloqueio com todas as operações em um objeto é uma maneira fácil de garantir corretude mas pode acabar serializando todos os acessos ao objeto. Por exemplo, um único bloqueio protegendo as operações de inserir e consultar elementos em uma lista não permite que diferentes threads acessem o objeto concorrentemente.

Instabilidade - Como vários bloqueios podem ser usados ao mesmo tempo, a ordem com que eles são adquiridos é extremamente importante. Threads diferentes adquirindo dois ou mais bloqueios em ordem inversa podem causar uma dependência de espera circular, um cenário conhecido como *deadlock*. Outras instabilidades incluem: (i) *inversão de prioridade*, em que uma thread de baixa prioridade toma posse do bloqueio, sofre preempção pelo sistema operacional e impede que threads de alta prioridade prossigam; e (ii) *convoying*, uma generalização do caso de inversão de prioridade - o bloqueio de uma thread causa enfileiramento e bloqueio de outras threads.

Composição - construir novas operações a partir de outras preexistentes é extremamente difícil com bloqueios. Considere o caso de uma operação que deva mover um item entre duas filas. Em princípio essa operação poderia ser composta através dos métodos de remoção e inserção, sendo requerido atomicidade (ou seja, não pode ser percebida a ausência do item em uma fila antes que o mesmo seja inserido na outra). No entanto, o uso de bloqueios torna complicada tal composição sem que o encapsulamento do objeto seja quebrado ou ainda que novos bloqueios seja introduzidos.

Essas dificuldades expõem a escassez nas ferramentas disponíveis para o desenvolvimento de aplicações concorrentes. Novas abordagens e abstrações que facilitem o projeto, desenvolvimento e verificação de sistemas concorrentes são necessárias.

2.2.2 Memória Transacional

Uma das alternativas para mecanismos de sincronização concorrentes, especialmente para aqueles baseados em bloqueios, é a abordagem chamada de **Memória transacional** (*Transactional Memory* - TM) [9].

Memória transacional usa como principal abstração o conceito de transação para estruturar e facilitar a escrita de programas concorrentes em memória compartilhada. A ideia de transação nasceu e se desenvolveu no contexto de sistemas de banco de dados [8]. Nesse domínio, uma transação é vista como um grupo de operações satisfazendo as seguintes propriedades (conhecidas como ACID):

- Atomicidade - ou todas as operações são executadas, ou então nenhuma o é. A presença de alguma falha durante a execução da transação faz com que os resultados das operações produzidos até aquele ponto sejam descartados;
- Consistência - uma transação leva o sistema de um estado consistente (pré-transação) a outro estado consistente (pós-transação);
- Isolamento - os resultados parciais decorrentes do processamento de uma transação não são vistos por outra transações;
- Durabilidade - as mudanças efetivadas pela transação são permanentes e resistentes a uma eventual falha no sistema.

O conceito de transação no contexto de TM é praticamente o mesmo, com exceção da última propriedade: ela só se aplica em estudos de memória não volátil, onde se mostra uma propriedade chave para seu funcionamento. Como este não é o foco do trabalho, onde trabalhamos com memória volátil, é natural que essa propriedade seja descartada.

Transações em memória aumentam o nível de abstração no que se refere à sincronização em programas concorrentes, oferecendo as seguintes vantagens [19]:

Facilidade de programação - o programador não precisa se preocupar em *como* garantir a sincronização, e sim especificar o que deve ser executado atomicamente. Note que no caso dos bloqueios é necessário fazer a associação entre um bloqueio e o respectivo dado protegido. A abordagem transacional é muito mais declarativa nesse sentido, deixando o *como* para o sistema de execução.

Potencial ganho em desempenho - o fato de a sincronização estar a cargo do sistema de execução permite explorar paralelismo de forma mais agressiva. No caso dos bloqueios, o programador precisa adotar uma abordagem conservadora (ou pessimista) porque estaticamente não é possível prever todas as possíveis intercalações e regiões de memória acessadas. Já com transações, o sistema de execução pode permitir que um mesmo método seja executado concorrentemente (de forma otimista) e verificar se há

conflitos entre os acessos. Isso permite total paralelismo nas situações em que os acessos são a dados disjuntos.

Composição - o maior nível de abstração fornecido pelas transações permite naturalmente a composição de código. Para criar uma nova operação com base em outras preexistentes basta englobá-las em uma nova transação. Isso é possível porque uma nova transação pode ser iniciada sob o contexto de outra, característica conhecida como aninhamento.

2.3 Características de um Projeto de Memória Transacional

O projeto de um sistema de TM é constituído, basicamente, de quatro componentes [9]: blocos atômicos, mecanismo de controle de concorrência, versionamento de dados, e estratégias de detecção e resolução de conflitos.

2.3.1 Blocos Atômicos

Do ponto de vista do programador, um bloco atômico é uma sequência de instruções executado de maneira indivisível, sem qualquer tipo de interrupção por instruções fora do bloco atômico. A Figura 2.2 mostra um exemplo de um bloco atômico. Note que no exemplo, a única tarefa do programador foi, através do bloco atômico, denotar o que deve ser sincronizado, e não como a sincronização é efetuada.

2.3.2 Controle de concorrência

Um sistema transacional necessita de algum mecanismo de sincronização para mediar acessos concorrentes ao dado. Um conflito acontece quando uma mesma região de memória é acessada por duas ou mais transações, e pelo menos um desses acessos é de escrita. O controle de concorrência determina quando os conflitos são detectados e resolvidos. Esse controle pode ocorrer de duas maneiras, ditas *pessimista* e *otimista*. No controle pessimista, uma transação é cancelada assim que essa transação tenta acessar uma posição de memória já alterada, mas ainda não efetivada, por outra transação. Para o controle otimista, a transação é cancelada no momento da efetivação. Dessa forma, uma transação que acessa uma posição de memória já alterada, mas ainda não efetivada, deve realizar suas operações antes da posição de memória ser efetivada, se não ela é cancelada. Na prática, o controle de concorrência otimista pode ser implementado de forma que a detecção e resolução de conflitos possam ocorrer em qualquer instante entre o início e finalização da transação. Uma alternativa é utilizar barreiras de validação e realizar a detecção e resolução de conflitos de forma incremental.

```
atomic{
 x++;
}
```

Figura 2.2: Exemplo de um bloco atômico

2.3.3 Versionamento de Dados

Durante a execução de uma transação, as modificações em memória compartilhada implicam a existência de duas versões dos dados, ditas *corrente* e *especulativa*. O mecanismo de versionamento é o componente responsável por gerir essas duas versões. A versão dita corrente é a anterior à modificação feita pela transação, enquanto que a especulativa é o produto dessa modificação. O gerenciamento de versões, ou versionamento, pode ocorrer de duas formas: diferida/preguiçosa (*lazy*) ou direta/ansiosa (*eager*). Na primeira, a versão especulativa é armazenada em um *buffer* privado à transação (*redo-log*) e a versão corrente permanece na região compartilhada. No versionamento direto, a situação inversa ocorre, ou seja, a versão especulativa é armazenada na região global (o que implica necessidade de se utilizar detecção de conflitos adiantada para as escritas) e a versão corrente é armazenada em um *buffer* privado à transação (*undo-log*).

2.3.4 Detecção e resolução de conflitos

Para detecção de conflitos, uma transação geralmente mantém um conjunto de leitura, com os endereços dos elementos lidos, e um conjunto de escrita, com os endereços dos elementos que foram alterados. Há um conflito entre duas ou mais transações quando a intersecção entre o conjunto de leitura e o conjunto de escrita de transações diferentes é não vazia. Ou seja, quando duas ou mais transações acessam o mesmo elemento e um dos acesso é de escrita.

Os mecanismos de detecção de conflito diferenciam-se em três aspectos: (i) granularidade; (ii) momento no qual é realizada a detecção; e (iii) quais transações são consideradas na detecção. A granularidade diz respeito a menor unidade na qual o mecanismo consegue detectar um conflito. Geralmente implementações em software de TM (STM) detectam conflitos no nível de palavra ou objeto, enquanto implementações em hardware de TM (HTM) detectam em nível de linhas de cache. O momento da detecção pode ser basicamente três: (1) no momento do acesso a uma região previamente modificada (*eager detection* - detecção de conflitos adiantada); (2) uma ou mais vezes durante a execução (barreiras de validação), ou ainda (3) durante a fase de *commit* (*lazy detection* - detecção de conflitos adiada). Existem ainda duas abordagens quanto a quais transações são consideradas para detecção: conflitos são detectados entre transações finalizadas e não finalizadas ou somente entre transações não finalizadas. Geralmente os STMs não implementam isolamento entre acessos transacionais e não transacionais. Nessas imple-

mentações tais acessos não têm seus conflitos tratados, ficando a cargo do programador sincronizá-los. Porém, existem algumas implementações em software que possuem mecanismos como validação por valor que podem tratar conflitos entre acessos transacionais e não transacionais [9].

2.4 Tipos de implementação

Em termos gerais, as implementações do arcabouço transacional são comumente divididas em hardware, software e híbridas. Nas subseções a seguir, serão apresentados alguns detalhes gerais das implementações em HTM e STM, mostrando suas características gerais e a interface básica utilizada para sua utilização.

2.4.1 Implementação em Hardware (HTM)

As implementações em hardware, comumente conhecidas por HTM (*Hardware Transactional Memory*), estendem a cache do processador com bits transacionais e usam o protocolo de coerência de cache do barramento para garantir atomicidade e isolamento. Por proverem mecanismos em hardware, HTMs possuem naturalmente um bom desempenho. Por outro lado, elas sofrem com limitações de recursos, como pouco espaço em cache para manter conjuntos de leitura e escrita das transações. A tendência atual é prover um hardware transacional básico que suporte a maioria das transações (*best effort*), redirecionando a execução para software caso seja necessário. Essa abordagem é conhecida como híbrida, por utilizar tanto suporte em hardware quanto em software. Os novos processadores lançados recentemente pela Intel [16] e IBM [25, 17, 18] adotam essa estratégia.

Para exemplificar implementações HTM, será utilizada a extensão TSX (*Transactional Synchronization Extensions*) desenvolvida pela Intel. A TSX é uma extensão do conjunto de instruções da arquitetura x86 que adiciona suporte para memória transacional, acelerando a execução de aplicações concorrentes através da omissão de bloqueios. De acordo com diferentes benchmarks, a TSX pode prover ganhos de até 40% em workloads específicos. [27]

A TSX foi documentada em fevereiro de 2012, e lançada em junho de 2013 em específicos microprocessadores Intel baseados na microarquitetura Haswell [16]. A TSX oferece duas interfaces em software para criar regiões de código transacional. A interface **Hardware Lock Elision** (HLE) e a interface **Restricted Transactional Memory** (RTM). A HLE é uma interface baseada em prefixos de instruções com o intuito de ser compatível com processadores legados sem suporte para TSX. Para a HLE, duas novas instruções prefixo foram adicionadas: `XACQUIRE` e `XRELEASE`. Nessa interface, a execução é otimista, onde as transações são executadas sem adquirir os *locks*. No caso de uma transação ser cancelada, a execução é reiniciada a partir da instrução com o prefixo `XACQUIRE`, mas

agora executando como se ele não estivesse presente, adquirindo os *locks*. A RTM é uma implementação alternativa à HLE, onde essa interface não possui retrocompatibilidade aos processadores legados, mas oferece ao programador a flexibilidade de especificar um código de recuo caso a transação seja cancelada. Para isso, essa interface adiciona três novas instruções: `XBEGIN`, `XEND` e `XABORT`. A falha de uma transação redireciona o processador para o caminho do código de recuo, que é especificado pela instrução `XBEGIN`.

2.4.2 Implementação em Software (STM)

Como mencionado anteriormente, sistemas transacionais também podem ser implementados inteiramente em software (STM). A grande vantagem dessa abordagem é que ela permite flexibilidade na implementação dos algoritmos, principalmente em face da diversidade semântica existente. Além disso, as abordagens em software podem ser executadas diretamente na maioria dos processadores atuais que ainda não possuem suporte transacional em hardware. A grande desvantagem dessa abordagem é que ela, em termos de desempenho, perde para HTM. Os sistemas em software usam barreiras de leitura e escrita para interceptar os acessos aos dados compartilhados e manter os conjuntos de leitura e escrita.

Para exemplificar a interface básica de uma implementação em STM, será apresentada a API de STMs típicas que trabalham com granularidade de palavras por serem bastante difundidas atualmente. A seguir serão descritas as principais rotinas fornecidas por uma biblioteca transacional.

- `TxStart`: marca o início de uma nova transação.
- `TxCommit`: realiza uma tentativa de término da transação. Se a transação for efetivada com sucesso, os dados especulativos ficam visíveis para todo o sistema de forma atômica. Caso contrário, a transação é cancelada e o fluxo de execução retorna para a instrução seguinte ao comando `TxStart`.
- `TxAbort`: Cancela a transação atual. Essa primitiva é chamada implicitamente pela STM quando a operação `TxCommit` falha, mas pode também ser invocada explicitamente.
- `TxLoad` (*l*): A palavra contida na posição de memória *l* é lida e retornada.
- `TxStore` (*l, v*): O valor *v* é armazenado na posição de memória *l*.

Embora essa API possa ser usada diretamente por programadores para a implementação de aplicações transacionais, seu uso direto requer bastante atenção, já que todos os acessos de leitura e escrita à memória compartilhada devem ser adequadamente instrumentados através das respectivas barreiras. Desta forma, um uso mais eficiente seria usar

```
1  atomic{
2    x++;
3  }
```

Figura 2.3: Versão Original

```
1  if ((status = xbegin()) ==
       _XBEGIN_STARTED){
2     x++;
3     _xend();
4  } else {
5     // caminho em software
6  }
```

Figura 2.4: Versão instrumentada (HTM)

```
1  jmp_buf checkpoint;
2
3  setjmp(checkpoint);
4
5  TxStart(&checkpoint);
6
7  int temp;
8  temp = TxLoad((intptr_t *)(
       void *)&x);
9  temp++;
10 TxStore((intptr_t *)(void *)
       &x,(intptr_t)temp);
11
12 TxCommit();
```

Figura 2.5: Versão Instrumentada (STM)

a implementação da API como uma biblioteca de tempo de execução que pode ser invocada pelo compilador para implementar blocos atômicos (veja a Subseção 4.2.3 para ver essa abordagem). Esta é a abordagem utilizada pelo compilador GCC (*GNU Compiler Collection*), por exemplo.

As Figuras 2.4 e 2.5 apresentam os exemplos de instrumentação em hardware (Figura 2.4) e em software (Figura 2.5). A Figura 2.3 representa o código escrito pelo programador usando a construção `atomic`. Um compilador transformaria o código criado pelo programador em algo parecido com a versão instrumentada em software. A instrumentação em hardware mostra como o suporte em HTM funciona, onde neste caso, usando a interface RTM, é necessário sempre oferecer um caminho de recuo em software caso ocorra um cancelamento na transação em hardware.

Considerando a importância de STM até dentro do contexto de HTM, e seus problemas de ineficiência, as implementações em software de TM têm sido alvo de muita discussão nos últimos anos. No próximo capítulo será discutido o suporte que os compiladores oferecem atualmente para TM e seus problemas de otimização atuais.

Capítulo 3

Suporte para Memória Transacional em Compiladores

O compilador, dado um programa escrito através de construções transacionais, deve gerar o código para iniciar/finalizar a transação, além de instrumentar todo o acesso às variáveis compartilhadas. A maioria dos compiladores que geram código transacional (i.e, GNU C Compiler - GCC) utilizam-se de uma interface binária com a aplicação (*Application Binary Interface* - ABI) criada pela Intel [15]. Esse capítulo discute a especificação dessa ABI, o suporte presente nos compiladores da Intel e GCC, trabalhos relacionados a este, e resultados experimentais que expõem os problemas de eficiência do GCC.

3.1 Trabalhos Relacionados

Essa Seção tem como objetivo destacar alguns dos trabalhos que contribuíram e inspiraram a elaboração dessa dissertação. Os primeiros a descrever otimizações de compiladores para memória transacional foram Harris et al. [10] usando um compilador criado por seu grupo de pesquisa chamado Bartok. Otimizações apresentadas no texto incluem formas de evitar a instrumentação de objetos alocados recentemente e movimentações no código feitas pelo compilador para barreiras transacionais. Eles também discutem otimizações feitas em tempo de execução para objetos locais à transação para evitar instrumentação extra. Seguindo na mesma linha, o trabalho de Adl-Tabatabai et al. [1] propõe otimizações de tempo de compilação e execução para o compilador StarJIT. Começando a evidenciar os problemas de instrumentação causada por código transacional gerado pelo GCC, Wang et al. discutem as dificuldades de otimizar código transacional em linguagens como C [26]. Eles descrevem otimizações que realizam eliminação de barreiras redundantes, alinhamento de sub-rotinas transacionais, e inserção de pontos de verificação (*checkpointing*) em transações. O primeiro trabalho a reconhecer o fenômeno de instrumentação excessiva que ocorre em compiladores foi Yoo et al.[28]. Eles propuseram uma nova cons-

trução, `pragma tm_waiver`, que permite programadores especificarem se o compilador deve instrumentar um determinado bloco de código ou função. Essa ferramenta é similar à construção pragma apresentada neste trabalho, mas ela não permite programadores a especificarem exatamente quais variáveis não devem ser instrumentadas. O trabalho de Ruan et al. investiga a interação entre a instrumentação feita pelo compilador e o desempenho das transações [23]. Eles argumentam que o compilador deve considerar a plataforma que está gerando código para determinar quais otimizações aplicar.

Dragojević et al. [5] investigaram otimizações para memória capturada, ou seja, memória alocada dentro de transações que não conseguem escapar à transação alocadora. Acessos à memória capturada não requerem barreiras e portanto podem ser elididas pelo compilador. Uma análise de captura foi desenvolvida durante a compilação e a execução. Carvalho e Cachopo posteriormente estenderam a análise de captura para um ambiente gerenciado baseado na máquina virtual Java [2].

Apesar de técnicas baseadas em análise de captura (no nível do compilador) poderem providenciar ganhos de desempenho, o compilador ainda precisa ser conservador e pode não ser capaz de elidir todas as possíveis barreiras. Portanto, a solução proposta nessa dissertação depende dos programadores a informarem ao compilador quais partes da memória compartilhada não devem ser instrumentadas.

3.2 ABI de memória transacional da Intel

Documentada em 2009, a ABI de memória transacional da Intel tem o objetivo de definir uma interface entre o código gerado para arquiteturas IA32 e a biblioteca de memória transacional, de forma que o mesmo código de usuário possa ser ligado a diferentes bibliotecas transacionais em tempo de execução sem a necessidade de recompilação de código.

A ABI também mira ser a mais genérica possível, evitando restrições de implementações de bibliotecas diferentes. Quando documentada, a Intel também apresentou o desejo de oferecer o melhor desempenho possível para implementações de TM. Essa ambição requer que a ABI apresente o menor *overhead* possível. Porém, tal otimização irá requerer futuras revisões e na documentação atual a Intel define claramente que não cobiçou buscar a melhor otimização para ABI na versão inicial da especificação.

Essa primeira versão da ABI apresenta suporte para os sistemas operacionais Windows e Linux sendo executados em arquiteturas IA32 e Intel(R)64. As linguagens padrões suportadas são C e C++. Para as instruções da ABI não conflitarem com outras existentes, a Intel definiu uma nomenclatura geral para suas instruções, que será apresentada na subseção a seguir.

3.2.1 Nomenclaturas da ABI

A especificação provê uma nomenclatura que não conflita com o conjunto de símbolos de usuário. Todas os nomes das funções de biblioteca começam com o prefixo `_ITM_`.

As funções de leitura e escrita são construídas seguindo a seguinte pseudo expressão regular:

```
_ITM_[RW]{a[RW]}type | _ITM_RfWtype
```

A Tabela 3.1 apresenta a descrição de cada componente da nomenclatura. A Tabela 3.2 apresenta os tipos que o argumento *type* pode assumir.

Tabela 3.1: Nomenclatura para funções de leitura e escrita

Nomenclatura	Significado
`_ITM_`	O prefixo utilizado em todos os nomes da ABI.
`[RW]`	O caractere 'R' para leitura e 'W' para escrita.
`RfW`	Operação *Read for Write*, indicando uma operação de leitura seguida de uma de escrita
`{a[RW]}`	*String* opcional, onde 'aR' significa *After Read* e 'aW' significa *After Write*
type	Representa o tipo do argumento de acordo com a tabela 3.2

Tabela 3.2: Codificação dos tipos

Nomenclatura	Significado
`U[1248]`	Inteiro sem sinal com o tamanho de 1,2,4, ou 8 bytes.
`[FDE]`	Ponto flutuante com valor de tamanho 4 (float), 8 (double), ou 16 (long double).
`M64, M128, M256`	valores `__m64`, `__m128`, `__m256`. Tipos específicos criado pela Microsoft para uso no conjunto de instruções MMX
`CF`	Complexo composto de floats
`CD`	Complexo composto de doubles
`CE`	Complexo composto de long doubles.

Além das funções de leitura e escrita, a ABI também define as funções _ITM_abortTransaction, _ITM_beginTransaction, _ITM_commitTransaction. Há várias outras funções auxiliares definidas pela ABI, porém essas não estão no escopo deste trabalho.

3.2.2 Exemplo de código

Considere o exemplo da Figura 3.1, retirado da documentação da ABI da Intel [15] e o respectivo código gerado na Figura 3.2. É interessante notar como o bloco atômico foi expandido no código gerado e como as operações de inicialização (linha 8), cancelamento (linha 32) e efetivação (linha 34), as operações de escrita (linha 29) e leitura (linha 25) foram substituídas pelas funções seguindo a nomenclatura discutida anteriormente.

A função `_ITM_beginTransaction` retorna um inteiro que define o que a transação irá fazer no início. Os códigos de retorno são 1 (executa o código instrumentado), 2 (executa código não instrumentado), 4 (salva variáveis locais que não serão instrumentadas), 8 (resgata variáveis locais que não foram instrumentadas) ou 16 (cancela transação). As macros de execução desses códigos são: *a_runInstrumentedCode*, *a_runUninstrumentedCode*, *a_saveLiveVariables*, *a_restoreLiveVariables*, e *a_abortTransaction*. Na linha 29,

```
1  int s;
2  int __declspec (tm_callable) f(int);
3
4  void Foo(void) {
5    int i;
6    for (i=0;i<10;i++){
7     atomic{
8       s += f(i);
9       if (s > 1000) __tm_abort;
10    }
11   }
12 }
```

Figura 3.1: Exemplo de código transacional

```
1  int s;
2  int f(int);
3
4  void foo(void){
5    _ITM_transaction * td = _ITM_getTransaction();
6   for (i=0;i<10;i++) {
7    //bloco atomico inicio
8    int doWhat = _ITM_beginTransaction (td,pr_instrumentedCode | &start_outer_loc);
9
10   if (doWhat & a_restoreLiveVariables) {
11    /* compilador resgata variaveis locais que nao
12       nao foram instrumentadas */
13   }
14
15   //abortar transacao
16   if (doWhat & a_abortTransaction) goto txn1_abort_label;
17
18
19   if ((doWhat & a_saveLiveVariables)) {
20    /* compilador salva variaveis locais que nao
21       serao instrumentadas
22    */
23   }
24
25   int sval = (int)_ITM_RU4 (td, (uint32_t *)&s);
26
27   sval += f_@TXN(i); // chama a funcao instrumentada
28
29   _ITM_WaRU4 (td, (uint32_t *)&s, (unit32_t)sval);
30
31   if (sval > 1000)
32    _ITM_abortTransaction(td,userAbort,&abort_loc);
33
34   _ITM_commitTransaction(td,commit_outer_loc);
35
36   txn1_abort_label:
37  }
38  return;
39 }
```

Figura 3.2: Exemplo de código gerado da Figura 3.1

a rotina de escrita chamada é a *write after read*, que significa que o compilador sabe que a escrita é dominada pela leitura, e utilizá-la ao invés da rotina convencional oferecerá um melhor desempenho, pois é uma das rotinas otimizadas de leitura e escrita que a ABI oferece. Já a rotina de leitura (linha 25) é a rotina comum porque não ocorreu leitura anterior desse endereço dentro da transação.

```
1  int remove(node_t *head, int item)
2  {
3   node_t *pred, *curr;
4   __transaction_atomic{
5     pred = head;
6     curr = head->next;
7     while(curr->key < item){
8      pred = curr;
9      curr = curr->next;
10    }
11
12
13    if (item == curr->key){
14     pred->next = curr->next;
15     free(curr);
16     return 1;
17    }
18    return 0;
19  }
20 }
```

Figura 3.3: Operação de remoção utilizando suporte transacional do GCC

3.3 Suporte para Memória Transacional no compilador GCC

O suporte para TM no GNU C Compiler (GCC) foi adicionado na versão 4.7. O suporte é considerado experimental, significando que a sua implementação ainda não está otimizada. A grande razão para a inclusão do mecanismo transacional é possibilitar que programadores consigam desenvolver novas aplicações utilizando-se desse mecanismo, habilitar a utilização de TM em códigos existentes e receber contribuições da comunidade para expandir e otimizar o suporte.

O suporte para memória transacional do GCC é composto de duas partes: (i) Anotações transacionais para permitir a instrumentação automática de acessos à memória e (ii) bibliotecas em tempo de execução.

3.3.1 Anotações Transacionais

O GCC oferece mecanismo para anotar blocos de código e atributos de função para seletivamente instrumentar acessos em uma função. Acessos à memória dentro de blocos `__transaction_atomic` são instrumentados com chamadas à biblioteca em tempo de execução que implementa barreiras transacionais usadas para detectar conflitos. Todo bloco `__transaction_atomic` é transformado em dois caminhos: um com as barreiras transacionais que serão usadas por STMs e outra sem barreiras que podem ser usadas com HTM ou uma estratégia de sincronização alternativa (como por exemplo, usando um *lock* global).

A Figura 3.3 mostra um exemplo de paralelização usando o suporte transacional do GCC para a rotina de remoção de um elemento da lista encadeada. Note que é possível utilizar uma sentença de retorno de procedimento dentro de uma transação (linhas 16 e 18), já que neste caso o compilador simplesmente interpreta o retorno como fim da transação.

A especificação suporta duas anotações principais de funções: `transaction_safe` e `transaction_pure`.

- `transaction_safe`: Anotação que define uma função ser *segura* para uma transação. Como mencionado anteriormente, em transações atômicas, somente são invocadas funções seguras. Se a função invocada faz parte do mesmo módulo de compilação da transação (mesmo arquivo fonte), o compilador pode inferir automaticamente se tal função é segura. Caso contrário, a função precisa ser anotada com o atributo `transaction safe`. Se uma função com essa anotação possui operações inseguras, um erro de compilação é emitido.

  Para toda função anotada com esse atributo, o compilador gera duas versões: uma instrumentada, com barreiras de leitura e escrita necessárias para permitir o cancelamento da transação, caso seja necessário, e uma versão não instrumentada que pode ser usada fora de transações. Caso uma função seja anotada com esse atributo e o compilador não encontre a versão instrumentada, um erro é emitido.

- `transaction_pure`: Funções anotadas com esse atributo dizem ao compilador que elas não causam nenhum tipo de efeito colateral, e portanto todos os acessos à memória dentro dessas funções não são instrumentados com barreiras.

3.3.2 Biblioteca Transacional em Tempo de Execução

A última peça para a execução de códigos transacionais no GCC é a biblioteca transacional em tempo de execução. A biblioteca transacional no GCC utilizada é chamada de **libITM**, uma implementação que segue as linhas da ABI da Intel. Sua documentação é semelhante à da ABI da Intel, tanto que a GNU resolveu utilizar a documentação da ABI da Intel como referência para a libITM, evidenciando apenas as diferenças entre elas. As mudanças abrangem restrições adicionadas, funções removidas, adicionadas, alteradas ou ainda não implementadas e o modelo de memória, que deve ser definido na ABI da libITM, diferentemente da ABI da Intel, que não exige um modelo de memória.

As outras duas categorias, declarações e anotações, são parte do *front-end* do suporte transacional do GCC, onde o código fonte é compilado e o arquivo objeto é gerado.

A Figura 3.4 mostra o processo de compilação e execução de uma aplicação transacional. Primeiramente o código fonte é compilado usando o suporte transacional do GCC, então os arquivos objetos gerados são ligados e o código executável é gerado. O executável

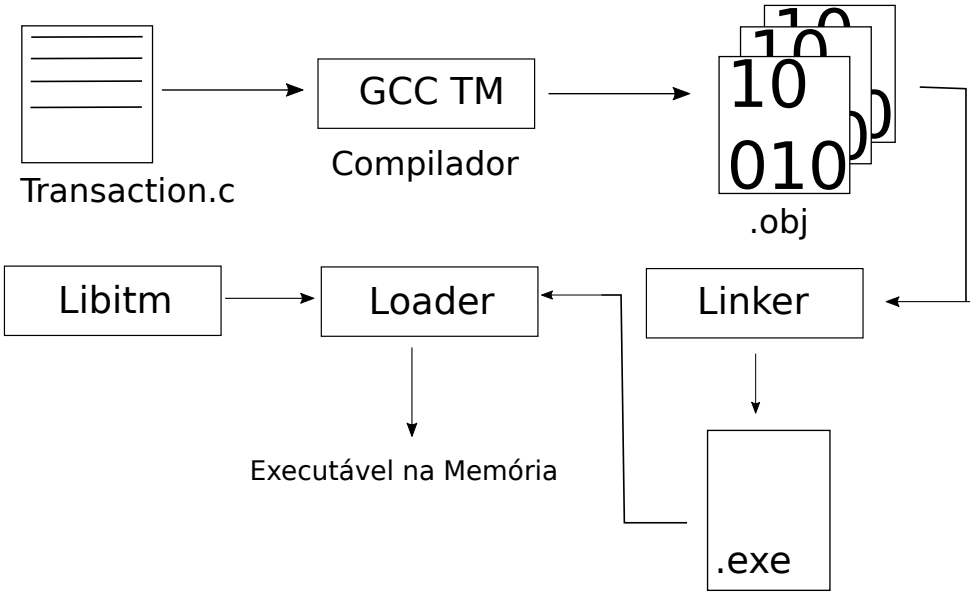

Figura 3.4: Processo de compilação e execução de uma aplicação transacional

é carregado pelo *loader*, que carrega o código executável, as pilhas do programa e dados são criadas e os registradores são iniciados. Além disso, o *loader* carrega a biblioteca transacional *libITM*, que executa dinamicamente todas as operações transacionais que são encontradas na execução. Essa figura é interessante por mostrar também o objetivo da ABI da Intel, em tornar a ligação das bibliotecas transacionais e o executável gerado totalmente independentes. Dessa forma, o executável pode ser carregado com diferentes bibliotecas transacionais.

3.4 Análise Experimental Sobre o Desempenho do Suporte Transacional do GCC

Como mencionado anteriormente, o suporte transacional do GCC ainda está em fase experimental, faltando um conjunto de otimizações para obter o desempenho desejado. Esta seção pretende apresentar algumas informações sobre as deficiências que o compilador apresenta, assim como uma comparação de desempenho de uma aplicação com duas versões: instrumentada manualmente e gerada pelo compilador GCC usando a biblioteca em tempo de execução NORec [3].

Para entender os problemas de desempenho do suporte transacional para o GCC, é importante compreender a grande fonte de ineficiência do próprio código transacional: adição de barreiras para leitura e escrita de variáveis compartilhadas. Esse problema é particularmente verdadeiro para implementações em software, já que cada invocação necessita acessar metadados da transação para checar a consistência da execução. Desta maneira, o compilador deve a todo custo conseguir otimizar o código que gera de forma a evitar ao máximo a instrumentação das leituras e escritas. Por exemplo, variáveis locais

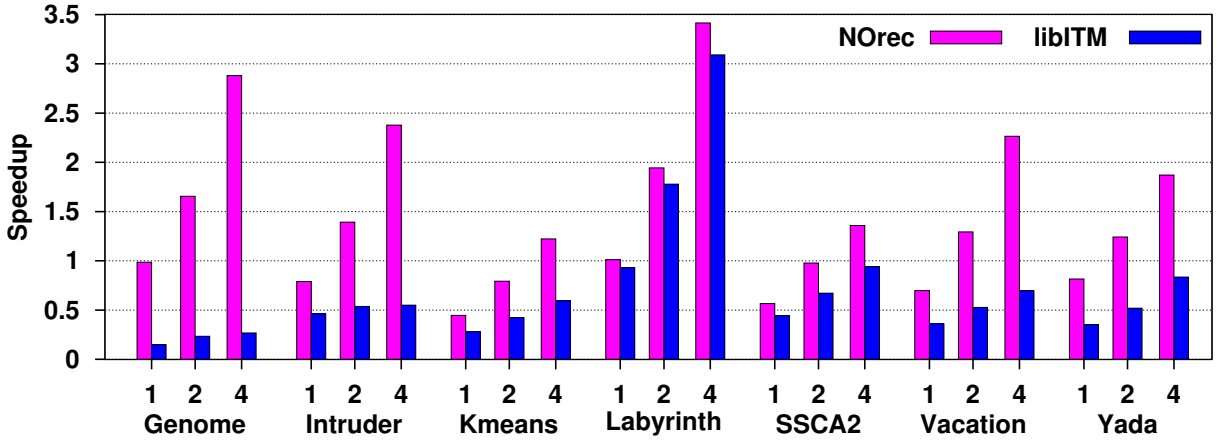

Figura 3.5: Comparação de desempenho entre a versão instrumentada manualmente (`NOrec`) e a gerada pelo compilador GCC (`libITM`). Ambas usam a STM conhecida como NOrec.

não necessitam de barreiras, uma vez que seu acesso possui escopo local à transação. Infelizmente nem sempre é fácil tomar essa decisão em tempo de compilação, forçando que o compilador seja pessimista e insira essas barreiras sempre que não seja possível decidir se o acesso é local ou global. Esse fenômeno é conhecido como instrumentação excessiva (Over-instrumentation).

Como exemplo do custo que pode ser adicionado pela instrumentação excessiva causada por compiladores, considere a Figura 3.5. Nela é feita uma análise de desempenho (realizada pelo grupo de pesquisa em paralelismo da UNESP), comparando o speedup das aplicações do pacote transacional conhecido como STAMP [22], com a versão gerada pelo compilador GCC (`libITM`) e a versão codificada manualmente (`NORec`). Ambas as abordagens utilizam como biblioteca em tempo de execução a STM conhecida como NORec, fazendo com que as comparações sejam justas. Nota-se pela figura que a versão instrumentada manualmente (versão fornecida pelo STAMP) é melhor que a gerada pelo compilador em todos os casos, particularmente para as aplicações `Genome`, `Intruder`, `Yada`.

Partindo da premissa que a diferença de desempenho entre as duas implementações ocorre por causa da instrumentação excessiva, este trabalho de mestrado oferece uma forma de eliminar essas instrumentações excessivas através de anotações feitas pelo programador, que serão responsáveis para guiar o compilador a não colocar barreiras em variáveis de escrita e leitura que não precisam delas.

3.4.1 Como Ocorre a Instrumentação em Excesso?

Para ilustrar como a instrumentação em excesso ocorre, considere o exemplo hipotético apresentado pela Figura 3.6. Esse exemplo é bem parecido com o que acontece com a aplicação `Yada`, do pacote de aplicações STAMP [22]. Nele, a função `Foo` (linhas 1–13) cria uma transação (através da palavra-chave `transaction_atomic` suportada pelo

GCC) (linha 2) que aloca memória dinâmica (linha 4) e, após algum processamento, é passada como parâmetro para um procedimento chamado `Bar` (linha 8). Note que esse trecho de memória alocado serve apenas para processamento local à transação, e portanto não precisaria ser instrumentado. O procedimento `Bar` (linhas 15–27) é anotado com o atributo `transaction_safe` do GCC, significando que tal rotina pode ser chamada por uma transação (como realmente é feito na linha 8), e que portanto o compilador deve gerar uma versão transacional. O laço das linhas 18–25 percorre a lista ligada, acessando um elemento qualquer (linha 20) e fazendo alguma operação com esse elemento (não mostrado). Do ponto de vista de `Foo`, a lista ligada `pm` é local mas, como o compilador não tem essa informação, todo o acesso de leitura e escrita aos elementos da lista devem ser instrumentados em `Bar`.

```
1  void Foo() {
2    __transaction_atomic {
3
4      list_t *pm = (list_t *) malloc(SIZE*sizeof(list_t));
5
6      // ...
7
8      Bar(pm);
9
10     // ...
11
12   }
13 }
14
15 static __attribute__((transaction_safe))
16 void Bar(list_t *p) {
17
18   while (p->next != NULL) {
19
20     int a = p->v;
21
22     // ...
23
24     p = p->next;
25   }
26
27 }
```

Figura 3.6: Exemplo hipotético com alocação local de memória dinâmica

A Figura 3.7 mostra, de forma simplificada, o código em linguagem de montagem gerado pelo compilador para o código da Figura 3.6. As linhas 1–13 referem-se ao procedimento `Foo`, enquanto as linhas 15–42 ao procedimento `Bar`. Neste trecho de código, as palavras em negrito indicam chamadas à ABI transacional para iniciar uma transação (linha 2), alocar memória transacionalmente (linha 5), efetivar a transação (linha 11), e as barreiras de leituras usadas para versionar os acessos ao elemento da lista ligada (linha 23), ao próximo ponteiro (linha 30), além do próximo elemento encadeado (linha 37).

```
1   FOO:
2     40080d: callq  400660 <_ITM_beginTransaction>
3     // ...
4     40083a: mov    $0x640,%edi
5     40083f: callq  400690 <_ITM_malloc>
6     400844: mov    %rax,-0x8(%rbp)
7
8     400848: mov    -0x8(%rbp),%rax
9     40084c: mov    %rax,%rdi
10    40084f: callq  400870 <BAR>
11    400854: callq  400640 <_ITM_commitTransaction>
12    400859: leaveq
13    40085a: retq
14
15  BAR:
16    // ...
17    // while ...
18    40087c: jmp    4008a1 <BAR+0x31>
19    40087e: mov    -0x18(%rbp),%rax
20
21    // int a = p->v;
22    400882: mov    %rax,%rdi
23    400885: callq  4006b0 <_ITM_RU4>
24    40088a: mov    %eax,-0x4(%rbp)
25    40088d: mov    -0x18(%rbp),%rax
26    400891: add    $0x8,%rax
27
28    // p = p->next;
29    400895: mov    %rax,%rdi
30    400898: callq  4006a0 <_ITM_RU8>
31    40089d: mov    %rax,-0x18(%rbp)
32    4008a1: mov    -0x18(%rbp),%rax
33    4008a5: add    $0x8,%rax
34
35    // ... (p->next != NULL)
36    4008a9: mov    %rax,%rdi
37    4008ac: callq  4006a0 <_ITM_RU8>
38    4008b1: test   %rax,%rax
39    4008b4: jne    40087e <BAR+0xe>
40
41    4008b6: leaveq
42    4008b7: retq
```

Figura 3.7: Código em linguagem de montagem gerado para o exemplo da Figura 3.6

```
1   static __attribute__((transaction_safe))
2   void Bar(list_t *p) {
3     #pragma tm elidebar {p}
4     {
5       while (p->next != NULL) {
6         int a = p->v;
7         // ...
8         p = p->next;
9       }
10    }
11  }
```

Figura 3.8: Exemplo hipotético anotado com o pragma proposto

Esse exemplo mostra que, de fato, o compilador instrumenta todo o acesso à lista ligada. Como todo acesso implica a chamada a um procedimento da biblioteca transacional, fica claro que o custo comparado ao código sem versionamento é realmente alto. Como o compilador não tem como saber que a lista que está sendo acessada em `Bar` é local, ele insere as barreiras transacionais a todos os acessos de leitura e escrita relacionados à variável.

Para resolver esse problema de instrumentação, o pragma proposto deve ser inserido como na Figura 3.8. Dessa forma, as barreiras em excesso serão elididas e o desempenho da aplicação deve melhorar significativamente.

Capítulo 4

Implementação de Anotações Para Código Transacional Gerado Pelo GCC

Este capítulo apresenta como o novo suporte de linguagem para omissão de barreiras, a principal contribuição deste trabalho, foi implementado.

4.1 Sintaxe do Pragma

Para representar as anotações, foram usadas as diretivas pragma. Conhecidas por sua utilização na popular API de programação paralela OpenMP, as diretivas pragma oferecem uma forma facilmente legível de especificar variáveis e o escopo que elas tem efeito. A estrutura do pragma criado para especificar as variáveis que não vão ser instrumentadas tem a sintaxe apresentada na Figura 4.1.

Um pragma pode ter um *espaço* definido além de seu *nome*. O espaço determina o grupo de operações a quem ele pertence. O espaço criado foi chamado `tm`, em referência ao nome da abordagem *transactional memory*. O nome para essa operação chama-se `elidebar`, em referência a *elide barriers*, que em português significa omitir barreiras. É possível criar novos pragmas usando o mesmo espaço `tm`, facilitando a manutenção de pragmas relacionados à memória transacional. Da maneira como foi implementado, é possível ter pragmas aninhados, múltiplos pragmas no mesmo nível, pragmas fora da transação (obrigatoriamente na linha anterior), ou pragmas dentro da transação. A Figura 3.8 mostra a utilização do pragma em um exemplo hipotético.

```
#pragma tm elidebar (variáveis)
{
 //escopo afetado
}
```

Figura 4.1: Sintaxe do pragma criado

4.2 Representações Intermediárias

O GCC compila códigos-fonte de várias linguagens, sendo elas: C, C++, Objective-C, Fortran, Ada e Go. Cada linguagem possui um *front-end* no compilador, onde ocorre a análise léxica , sintática , e semântica. Como essas diversas linguagens devem gerar executáveis em diferentes arquiteturas, o GCC precisa oferecer uma forma de representar o código de cada linguagem de uma maneira independente de qualquer arquitetura ou linguagem. A representação intermediária (RI) permite que um só compilador ofereça múltiplos *front-ends* para várias linguagens terem códigos gerados em múltiplas arquiteturas. Após a geração da RI, ocorre a geração do código de máquina.

O GCC possui três representações intermediárias, que são geradas na seguinte ordem: GENERIC, GIMPLE e RTL. A Figura 4.2 mostra o processo de compilação do GCC e a passagem pelas representações intermediárias (RI).

GENERIC: A estrutura de dados central usada pela RI é uma árvore. As operações realizadas dentro da RI sempre serão em volta de árvores sendo usadas como entrada e saída dessas operações. O objetivo principal da GENERIC, a primeira transformação do código dependente de linguagem para um código independente, é de transformar as funções do código dependente em árvores.

GIMPLE: GIMPLE é uma representação de três endereços derivada da RI GENERIC, desmontando as expressões de GENERIC em tuplas com no máximo três operandos (com algumas exceções, como chamadas de função). Variáveis temporárias são introduzidas nesse passo para armazenar valores intermediários para computar expressões complexas. Além disso, é importante notar que todas as estruturas de controle usadas em GENERIC são transformadas em saltos condicionais, e os escopos léxicos são removidos.

A fase de compilação que converte GENERIC em GIMPLE é referida como *gimplifier*. O *gimplifier* trabalha recursivamente, gerando tuplas GIMPLE das expressões GENERIC.

Um detalhe importante da representação GIMPLE são como os seus operandos são tratados. Como praticamente toda sentença GIMPLE contém uma referência a uma variável ou posição de memória, e essas sentenças tem tamanhos e formas diferentes, seus operandos estarão localizados em vários pontos da árvore da sentença. Para facilitar o acesso aos operandos da sentença, eles são organizados em listas associadas com cada

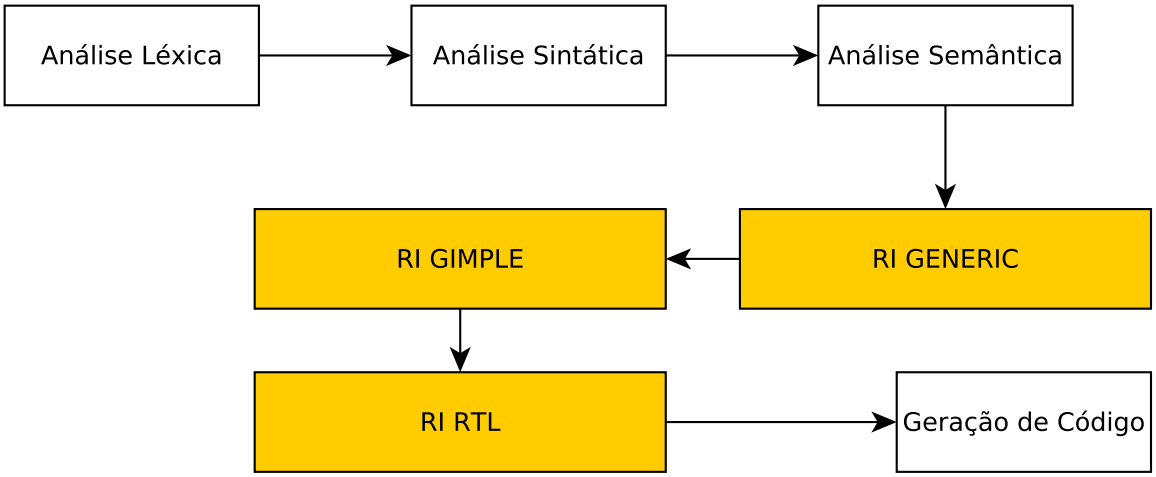

Figura 4.2: Processo de compilação do GCC

sentença. Cada elemento em uma lista de operandos é um ponteiro para uma declaração de variável, declaração de parâmetro ou nome SSA. O nome SSA é baseado na forma SSA (*single static assignment*), que parte da premissa que as variáveis de um programa são atribuídas em exatamente uma posição do programa. Novas atribuições da mesma variável criam novas versões dela.

RTL: A última parte da RI é feita em uma representação de baixo nível chamada *Register Transfer Language* (RTL). Nessa linguagem, as instruções são descritas, uma a uma, em forma algébrica, que descreve o que a instrução faz.

4.3 Localizando a função que cria barreiras de escrita e leitura em variáveis

No código-fonte do GCC (versão utilizada: 6.2.0), o arquivo `trans-mem.c` é responsável por realizar o processo de otimização das sentenças de memória transacional em tempo de compilação. Durante esse processo de otimização, é verificado se as variáveis usadas dentro de um bloco atômico devem ser instrumentadas ou não. Essa verificação é realizada em uma função chamada `requires_barrier`.

A Figura 4.3 mostra, de maneira abstraída, a função `requires_barrier`. Essa função recebe como parâmetro uma árvore (linha 1), uma estrutura de dados central usada pela representação interna do GCC. Através da macro TREE_OPERAND (linha 5), é possível acessar os operandos da árvore `x`. Esses operandos representam constantes, variáveis, expressões e vetores. Para determinar se um operando será instrumentado ou não, realiza-se um teste condicional através da estrutura switch case (linhas 7 a 12) utilizando como parâmetro de valor a macro TREE_CODE (linha 7). Esta macro retorna

```
1  static bool requires_barrier (tree x, //...)
2  {
3    tree orig = x;
4    while (handled_component_p (x))
5     x = TREE_OPERAND (x, 0);
6
7    switch(TREE_CODE(x))
8    {
9     //...
10    default:
11     return false;
12   }
13 }
```

Figura 4.3: Função do GCC que instrumenta barreiras no código transacional

o tipo do operando, como por exemplo variável declarada, retorno de função, entre outros. Dependendo do tipo do operando, um conjunto de operações são realizadas, retornando ao fim delas um valor falso ou verdadeiro. Se o valor retornado for falso, este operando não é instrumentado, caso contrário, ele é. Caso nenhuma condição seja satisfeita, o valor retornado sempre é falso (linha 11).

Sabendo como a função `requires_barrier` funciona para definir a instrumentação de um bloco atômico, através do retorno de valores verdadeiro e falso, o trabalho do pragma seria simplesmente retornar falso para as variáveis que forem especificadas que não devem possuir barreiras de escrita e leitura.

Como a função `requires_barrier` ocorre até a RI GIMPLE, não foi necessário acessar a RI RTL durante a implementação do pragma.

4.4 Implementação do Pragma

4.4.1 Metodologia

Visto que a função `requires_barrier` é uma função que retorna apenas verdadeiro ou falso, a estratégia adotada para a implementação do pragma é feita da seguinte forma: para toda variável especificada a ter suas barreiras removidas, se é adicionado um tipo de marcação à estrutura dessa variável. Dessa forma, quando a variável passar pela função `requires_barrier`, a primeira análise realizada será se essa marcação existe ou não na variável. Caso a marcação exista, toda a análise da função é ignorada e retorna-se falso para a variável, garantindo que nenhuma barreira seja adicionada para as operações de leitura e escrita da variável.

Como mencionado anteriormente, a estrutura de dados central usada pela RI é uma árvore, definida no arquivo `tree-core.h`. Para representar essa marcação, criou-se uma variável dentro da estrutura chamada `tree_base` (acessada por todos os tipos de nó da árvore) chamada `elidebar_flag`. Se ela estiver com o valor 1 armazenado, a variável não

```
1  cpp_register_deferred_pragma
2  (parse_in, "tm", "elidebar", PRAGMA_TM_ELIDEBAR, true,
       false);
```

Figura 4.4: Função que registra o pragma

receberá barreiras, caso contrário, passará pela análise da função `requires_barrier`.

4.4.2 Registrando o Pragma

Para que o pragma seja reconhecido pelo GCC, ele precisa ser registrado, caso contrário, qualquer pragma não reconhecido pelo compilador é ignorado. O pragma implementado foi registrado no arquivo `c-pragma.c`, que possui os pragmas registrados para a linguagem C. É possível que o pragma seja registrado de duas maneiras: que ele seja tratado pelo analisador sintático da linguagem C, ou por uma função gerenciadora dentro do arquivo `c-pragma.c`. O pragma implementado foi registrado para ser tratado pelo analisador sintático da linguagem C, já que nesse caso é possível criar um escopo para o pragma.

Para registrar o pragma, deve-se utilizar a função `cpp_register_deferred_pragma`, para que a função seja deferida para ser analisada pelo *front-end* do C. Essa função deve ser inserida dentro da função `init_pragma`. O pragma implementado foi registrado da forma como a Figura 4.4 apresenta.

O parâmetro `parse_in` é a variável usada que define a posição em que o pragma está localizado. O parâmetro `tm` é o espaço em que o pragma foi registrado, `elidebar` é o nome do pragma, PRAGMA_TM_ELIDEBAR é a identificação do pragma, os dois últimos parâmetros referem-se a permitir expansão de nome e de função.

4.4.3 Análise Sintática e Semântica do Pragma

Com o pragma registrado com sucesso, agora é possível realizar sua análise no arquivo c-parser.c, que é o parser front-end da linguagem C.

Uma função chamada `c_parser_pragma_tm_elidebar` foi criada para analisar o pragma. A análise consome o prefixo `#pragma tm elidebar`, e analisa as variáveis dentro dos parênteses, todas as variáveis chamam o marcador `elidebar_flag` e armazenam o valor 1 a ele. Além disso, é necessário criar o escopo do pragma. A Figura 4.5 apresenta o trecho do código que faz isso.

Das linhas 3 a 10 é realizado a análise de todos os elementos que estão dentro do escopo do pragma (entre as chaves "{}"). As linhas 12 a 16 criam a representação do pragma na linguagem GENERIC, criando um nó da árvore para o pragma.

```
1   location_t body_loc = c_parser_peek_token (parser) ->
       location;
2   block = c_begin_compound_stmt(true);
3   if(c_parser_next_token_is(parser, CPP_OPEN_BRACE))
4   {
5    tree stmt = c_parser_compound_statement(parser);
6    add_stmt(stmt);
7   }
8
9   block = c_end_compound_stmt(body_loc,block, true);
10
11  tree tmstmt = make_node(TM_ELIDEBAR);
12  TREE_TYPE(tmstmt) = void_type_node;
13  TM_ELIDEBAR_BODY(tmstmt) = block;
14  SET_EXPR_LOCATION(tmstmt,body_loc);
15  add_stmt(tmstmt);
```

Figura 4.5: Trecho de código que cria o escopo do pragma implementado

4.5 *Gimplificando* o Pragma

A representação de uma transação muda várias vezes durante o processo de otimização do GCC. No início, tem-se a transação representada por uma árvore GENERIC. Logo em seguida, o passo inicial de *gimplificação* ocorre (neologismo usado para indicar a transformação da representação GENERIC para GIMPLE) onde o nó GENERIC é substituído por um nó GIMPLE.

O arquivo fonte `trans-mem.c` é responsável por realizar as otimizações mais específicas do nó GIMPLE. Este arquivo também é responsável por "abaixar"o nível da representação das construções transacionais em formato GIMPLE para prepará-las para as otimizações da RI RTL que levará a geração de código final.

Durante essas fases de otimização no arquivo fonte `trans-mem.c`, a função `requires_barrier` é chamada. Em particular, esta função é chamada em duas fases de otimização: a fase `pass_lower_tm`, onde o corpo da transação é examinado em busca de `aborts`. E na fase `pass_tm_mark`, onde as sentenças de memória transacional são substituídas por chamadas ao builtin, e chamadas de função são substituídas por seus clones instrumentados (caso eles estejam disponíveis). Essas duas fases distintas exigem que as variáveis marcadas pelo pragma sejam tratadas de formas diferentes. Portanto, as próximas subseções mostram o que foi necessário ser feito para que a instrumentação de variáveis marcadas se comportassem da maneira esperada. Quando o GIMPLE ainda não está com o seu nível completamente "abaixado", ele é chamado de HIGH GIMPLE, onde o escopo léxico e expressões aninhadas ainda existem, enquanto que LOW GIMPLE expõe todos os saltos de controle e expressões de exceção diretamente na árvore assim como elimina o escopo léxico.

O GCC possui um arquivo fonte chamado `gimplify.c`. Nele ocorre a *gimplificação* de todos os nós GENERIC. A Figura 4.6 apresenta a função `gimplify_tm_elidebar`, que

```
1  static void
2  gimplify_tm_elidebar (tree *expr_p, gimple_seq *pre_p)
3  {
4   tree expr = *expr_p;
5   tree tbody = TM_ELIDEBAR_BODY (expr);
6
7   gimple *g;
8   gimple_seq body = NULL;
9   gtm_elidebar *tm_stmt;
10
11  push_gimplify_context();
12  g = gimplify_and_return_first (TM_ELIDEBAR_BODY (expr),
       &body);
13  if (gimple_code (g) == GIMPLE_BIND)
14    pop_gimplify_context (g);
15  else
16    pop_gimplify_context (NULL);
17
18  tm_stmt = gimple_build_tm_elidebar (body);
19
20  gimplify_seq_add_stmt (pre_p, tm_stmt);
21  *expr_p = NULL_TREE;
22 }
```

Figura 4.6: Função que realiza a gimplificação do corpo do pragama

realiza a *gimplificação* do corpo do pragma e adiciona o novo nó GIMPLE à sequência de sentenças. A linha 11 chama a função `push_gimplify_context`, que avisa ao compilador que estamos tratando de um novo nível de escopo. A função na linha 12, `gimplify_and_return_first` realiza a *gimplificação* do corpo do pragma. As Linhas 14 a 16 finalizam o novo escopo, voltando para o nível de escopo anterior. A linha 18 trivialmente transforma a árvore `TM_ELIDEBAR_BODY` em uma sentença GIMPLE chamada `GIMPLE_TM_ELIDEBAR`. Essa nova sequência é adicionada ao fim da sequência de sentenças na linha 20.

Durante a *gimplificação* do corpo do pragma na linha 12, variáveis temporárias são introduzidas. Esses temporários são usados para armazenar valores intermediários necessários para computar expressões complexas. Dessa forma, foi necessário herdar o valor da *flag* `elidebar_flag` da variável original para todas as variáveis temporárias que podem ser criadas. Há três funções principais no arquivo fonte `gimplify.c` que são responsáveis por criar variáveis temporárias: `get_initialized_tmp_var`, `create_tmp_var` e `get_formal_tmp_var`.

Com o pragma transformado em uma sentença GIMPLE, agora é possível manipular a função `requires_barrier` para que ela retorne falso para as variáveis com a *flag* `elidebar_flag` ativada.

4.6 Elidindo Barreiras com o Pragma

Essa Seção será dividida em duas partes: Como foi feita a manipulação da função `requires_barrier` na fase de otimização `pass_lower_tm` e então na fase `pass_tm_mark`.

4.6.1 Fase pass_lower_tm

A fase `pass_lower_tm` examina o corpo de uma transação em busca de `aborts`, marca as variáveis que precisam ser instrumentadas, e registra todos os dados necessários para construir um grafo de fluxo de controle apropriado. A Figura 4.7 apresenta um fluxo de execução das funções que a fase passa desde o seu início até chegar à função `requires_barrier`, abstraindo outros passos irrelevantes para a discussão a seguir. Setas coloridas em azul e vermelho podem ser executadas múltiplas vezes.

A classe `pass_lower_tm` é a declaração dessa fase (definida no arquivo fonte passes.def). Ela chama a função de ponto de entrada `execute_lower_tm`. O corpo da função corrente (referindo-se ao código a ser compilado) é analisado através da função `walk_gimple_seq_mod`, que tem como parâmetros o corpo da função a ser caminhada, a forma de como ela será tratada, geralmente através de uma sub-rotina, se será retornado o valor da função, e quais sentenças do corpo serão caminhadas (geralmente indicadas como WI, *walk_gimple_stmt*) . No caso, a subrotina é `lower_sequence_no_tm`, que é o próximo passo de execução denotado na Figura 4.7.

`lower_sequence_no_tm` verifica se há transações dentro do corpo. Se tiver, chama a função `lower_transaction`, avançando para a próxima fase. Caso contrário essa função transforma de maneira apropriada todas as sentenças GIMPLE da sequência que estão fora da transação.

A função `lower_transaction`, como o nome já implica, realiza a transformação de uma sentença GIMPLE_TRANSACTION, ou seja, o corpo da transação é transformado (*lowered*). Assim como em `execute_lower_tm`, `walk_gimple_seq_mod` é invocada para caminhar agora sobre o corpo da transação, usando a função `lower_sequence_tm` para tratá-lo. Além disso a função omite transações aninhadas, cria dois caminhos (um instrumentado e outro não) do código e registra as alterações realizadas para serem usadas depois, como por exemplo as variáveis que foram instrumentadas.

A função `lower_sequence_tm`, como a Figura 4.7 apresenta, pode ser chamada várias vezes, assim como chamar novamente a função `lower_transaction` caso sejam encontradas transações aninhadas. A função `lower_sequence_tm` pode invocar a função `lower_tm_elidebar`, que trata o corpo do pragma (transformando uma sentença `GIMPLE_TM_ELIDEBAR`, e pode ser chamada mais vezes caso tenha mais de um pragma, seja aninhado ou não, dentro do corpo da transação. E pode também chamar a função `examine_assign_tm`, que chama a função `requires_barrier` para o lado esquerdo e direito de uma sentença GIMPLE de atribuição. Para o lado direito (rhs - *right handed side*), marca

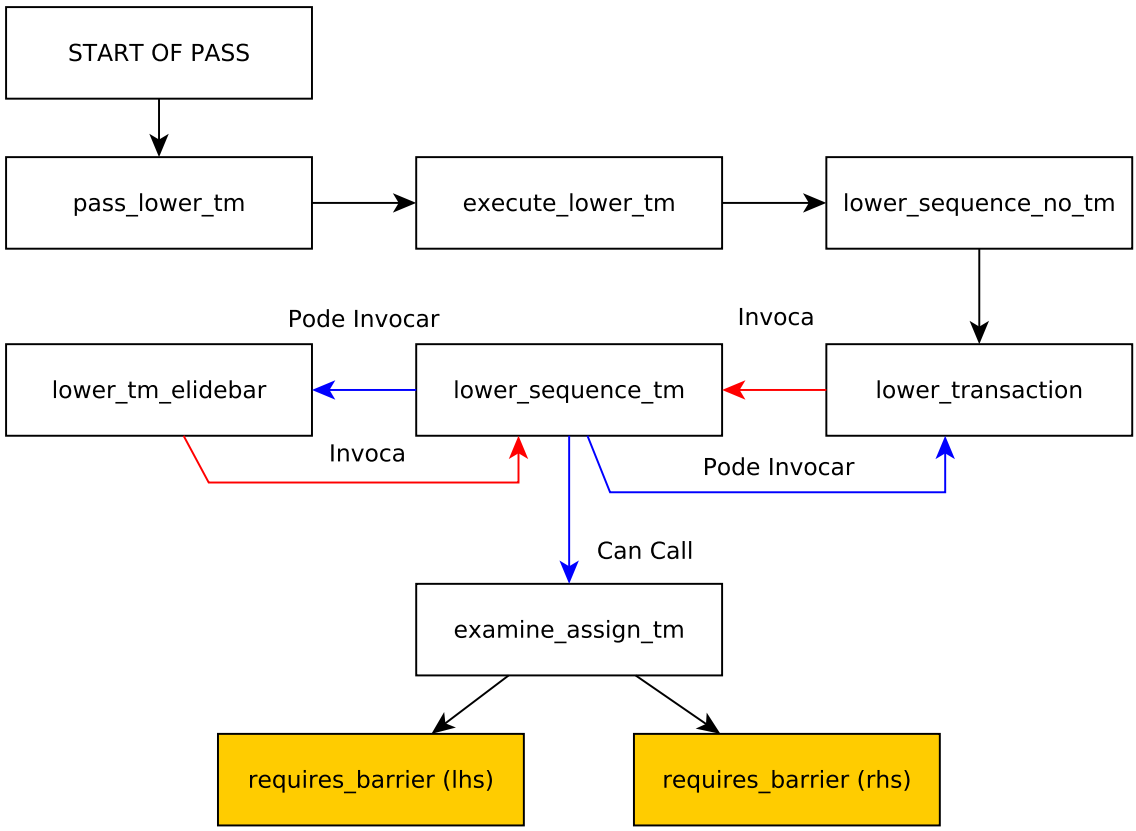

Figura 4.7: Fluxo de execução da fase pass_lower_tm para chegar à função requires_-barrier

se a variável será instrumentada com barreiras de leitura, e barreiras de escrita para o lado esquerdo (lhs - *left handed side*).

A Figura 4.8 mostra, de maneira abstraída, a função `lower_tm_elidebar`. As linhas 4 a 15 são importante para essa fase, enquanto que as linhas 18 a 32 são importantes para a fase `pass_tm_mark`. Dessa forma a discussão dessa Subseção se concentrará na primeira parte da função. Uma outra observação importante é que certos mecanismos implementados para que o pragma funcione em situações mais complexas (como pragmas aninhados) foram abstraídos deste texto para facilitar a compreensão da implementação.

As linhas 4 a 9 são declarações gerais para preparação da caminhada sobre o corpo do pragma. Quando as variáveis são marcadas no pragma, a *flag* que determina se elas serão instrumentadas ou não pode ser vista por todo o escopo da função a quem ela pertence, mas como é desejado que isso só seja válido para o escopo do pragma que as marcou, utilizou-se a *flag* auxiliar das linhas 11 (ativada) e 15 (desativada) para determinar que este é o escopo em que a análise do pragma é válida.

A linha 13 é a invocação da função `walk_gimple_seq_mod` para caminhar sobre o corpo do pragma através da função `lower_sequence_tm`. Dessa forma, como a Figura 4.7 mostra, a função `lower_sequence_tm` pode chamar a função `examine_assign_tm` que então invoca a função `requires_barrier`, onde nela, dado que a variável esteja com a

```
1   static tree
2   lower_tm_elidebar (gimple_stmt_iterator *gsi, struct walk_stmt_info *wi)
3   {
4    gimple_stmt_iterator *g;
5    gtm_elidebar *stmt = as_a <gtm_elidebar *> (gsi_stmt (*gsi));
6    struct walk_stmt_info this_wi;
7
8    memset(&this_wi, 0, sizeof(this_wi));
9    this_wi.info = (void *) &state_tm_elidebar;
10
11   flag_tm_elidebar = 1;
12
13   walk_gimple_seq_mod(gimple_tm_elidebar_body_ptr(stmt), lower_sequence_tm, NULL, &
         this_wi);
14
15   flag_tm_elidebar = 0;
16
17   /*Importante para pass_tm_mark*/
18   tree block = make_node (BLOCK);
19   gbind *bind;
20   bind = gimple_build_bind (NULL, NULL, block);
21
22   gsi_replace(gsi, bind, true);
23
24   gimple_bind_add_seq (bind, gimple_tm_elidebar_body (stmt));
25
26   tree avar = create_tmp_var_raw (void_type_node, "TMELIDEBAR");
27   tree bvar = create_tmp_var_raw (void_type_node, "ELIDEBARBEGIN");
28   tree cvar = create_tmp_var_raw (void_type_node, "ELIDEBAREND");
29
30   gsi_insert_before(gsi, gimple_build_assign(avar, bvar), GSI_CONTINUE_LINKING);
31   gsi_next(gsi);
32   gsi_insert_after(gsi, gimple_build_assign(avar, cvar), GSI_CONTINUE_LINKING);
33
34  }
```

Figura 4.8: Função lower_tm_elidebar abstraída

flag `elidebar_flag`, a função retorna o valor FALSO, sem precisar passar pelo resto da análise que a função faz.

4.6.2 Fase pass_tm_mark

Com a primeira fase que passa por `requires_barrier` tratada, agora é necessário preparar a fase `pass_tm_mark`. Essa preparação já começa durante a função `lower_tm_elidebar`, como mencionado na Subseção anterior.

Enquanto na primeira fase o GIMPLE está na forma HIGH GIMPLE, a fase `pass_tm_mark` já está na forma LOW GIMPLE, significando que características como o escopo léxico já não existem mais. Por conta disso a estratégia usada na primeira fase para indicar que o escopo do pragma foi encontrado e portanto permitir que as variáveis marcadas pelo pragma tenham suas barreiras elididas não funciona.

Após a linha 15 da Figura 4.8, a sentença GIMPLE_TM_ELIDEBAR não terá mais relevância para a execução. Porém, para que o corpo do pragma continue sendo processador pelo compilador, é necessário transformar essa sentença. A estratégia adotada foi transformar o corpo em uma sentença GIMPLE_BIND (representação de um escopo na forma GIMPLE). Visto que a única característica de GIMPLE_TM_ELIDEBAR é o corpo do pragma, essa transformação é bastante trivial e feita dentro das linhas 18 a 24. Nessa fase, o escopo léxico é eliminado em favor dos *builtins* apropriados, no caso da transação, o seu corpo é marcado com *builtins* de início e fim, por exemplo. Como o pragma criado não possui *builtins*, a estratégia foi simulá-los com variáveis temporárias. As linhas 26 a 28 criam as variáveis temporárias. Para que elas sejam analisadas pelo

compilador, criou-se uma atribuição para a variável que determina o início do pragma (linha 30) e a variável que determina o fim do pragma (linha 32).

Durante essa fase, é necessário prestar atenção somente na função `expand_assign_tm`, que expande as sentenças de atribuição em *builtins* de transação. Aqui é onde as variáveis temporárias criadas são lidas, e quando identificadas, a *flag* `flag_tm_elidebar` é ativada quando encontra a variável ELIDEBARBEGIN, que indica o início do corpo do pragma, e desativada pela variável ELIDEBAREND, que indica o fim do corpo do pragma. A forma como o `requires_barrier` é manipulado é idêntica à primeira fase.

Dessa maneira, o pragma é implementado completamente, atuando da maneira esperada, afetando as variáveis marcadas somente quando encontradas em seu escopo e manipulando a função `requires_barrier` para estas.

4.7 Discussões Sobre a Implementação do Pragma

Em dissertações acadêmicas, um compilador muito utilizado é o LLVM [20]. A sua principal razão é como ele foi desenhado desde o início como uma API, permitindo que ele possa ser reutilizado por ferramentas de análise de código fonte, IDEs, refatoração de código, entre outras. Sua árvore abstrata também foi projetada de forma a ser facilmente compreensível para programadores com experiência em como um compilador funciona. A decisão de escolher o GCC sobre um compilador que oferece tantas vantagens para um programador foi bem simples: o LLVM ainda não oferece suporte à memória transacional.

O GCC não possui uma documentação robusta, dificultando a compreensão do seu funcionamento completo, o que levou a múltiplas situações em que a solução foi encontrada através de testes de tentativa e erro. Um aspecto bastante laborioso foi propagar o valor da *flag* da variável original para as variáveis temporárias criadas baseada nela. Devido à forma como o GCC foi implementado, foi necessário adicionar esse valor manualmente para todas as instâncias possíveis de uma variável temporária. O mesmo ocorreu para os nomes SSA criados.

As otimizações feitas pelo GCC também atrapalharam na análise do código, já que o GCC simplifica o código durante a sua análise. Por exemplo, se o código-fonte analisado possuir uma operação como "x-x", a árvore abstrata do GCC conterá o valor 0, sem nenhuma referência a x. Dessa forma, durante a depuração de certos problemas, foi muito difícil encontrar a razão pela qual determinado erro estava acontecendo. Várias limitações durante o desenvolvimento, como a impossibilidade de gerar o binário a partir das representações intermediárias, causaram bastante frustrações no início do projeto.

Apesar dessas dificuldades, foi possível implementar o pragma com sucesso, e o GCC, apesar de suas desvantagens para pesquisa, ofereceu uma vasta fonte de novos conhecimentos durante o desenvolvimento desse projeto. Um objetivo para trabalhos futuros é utilizar o que foi implementado e o conhecimento adquirido sobre o GCC para oferecer

suporte de memória transacional a compiladores melhor projetados para pesquisa, como o LLVM.

Capítulo 5

Resultados Experimentais

Esse capítulo mostra o ganho de desempenho que um programador pode conquistar ao elidir barreiras desnecessárias através do pragma criado, como a instrumentação em excesso ocorre, e as dificuldades apresentadas para obter os resultados experimentais.

5.1 Exemplo Sintético

Para mostrar o potencial do pragma proposto, considere o código da Figura 5.1. Ela mostra uma função que retorna a comparação entre duas *strings*: `p1` e `p2`. A anotação da linha 1 diz ao GCC que essa função pode ser chamada por transações e portanto o compilador irá gerar uma versão com nenhuma instrumentação e um clone, chamado por transações, com instrumentação. Como não é conhecido em tempo de compilação se as duas *strings* comparadas podem ser modificadas por transações concorrentes, o compilador é pessimista e gera barreiras para os acessos às variáveis `s1` e `s2` (linhas 8 e 9).

Em alguns casos as *strings* a serem comparadas são apenas leitura, o que significa que a instrumentação é desnecessária. Uma aplicação que tem essa função e sofre da instrumentação em excesso é a `genome`, do pacote STAMP. Adicionando o pragma criado como mostrado na linha 4 força o compilador a omitir as barreiras de acesso às variáveis `s1` e `s2`.

Utilizando essa função, um simples programa foi criado onde múltiplas *threads* invocam a função `tm_strcmp` repetidas vezes para comparar duas *strings* aleatórias. Com 16000000 comparações usando *strings* de 64 caracteres, a Figura 5.2 mostra o speedup-1 (para facilitar a visualização) obtido sobre o código sequencial para execuções com e sem o pragma proposto. Como pode ser visto, a versão gerada pelo GCC não apresenta speedup em nenhum momento, enquanto que a versão que utiliza o pragma proposto conseguiu um speedup de até 3.5x com 8 threads.

```
1  __attribute__((transaction_safe))
2  int tm_strcmp(char* s1, char* s2)
3  {
4    #pragma tm elidebar(s1, s2)
5    {
6      char c1, c2;
7      do {
8        c1 = *s1++;
9        c2 = *s2++;
10       if (c1 == '\0')
11         return c1 - c2;
12     } while (c1 == c2);
13     return c1 - c2;
14   }
15 }
```

Figura 5.1: Exemplo Sintético.

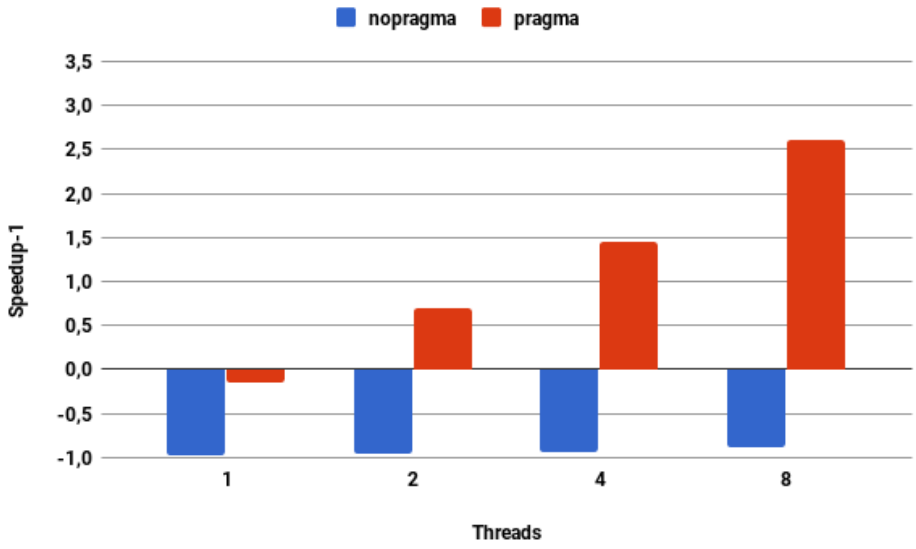

Figura 5.2: Speedup relativo à versão sequencial para o exemplo sintético

5.2 Configuração dos Experimentos

Para validar o novo suporte de linguagem implementado, utilizou-se o conjunto de aplicações STAMP [22]. As aplicações presentes no pacote STAMP são todas aplicações científicas de diversos campos de estudo, como bioinformática (`Genome`), segurança (`Intruder`) e computação geométrica (`Yada`). Essas aplicações também fazem uso de uma biblioteca com diversas estruturas de dados para atender as necessidades de cada aplicação. Por exemplo, a aplicação `Genome` faz uso da estrutura de dados *hashtable* para reconstruir uma sequência de nucleotídeos. Essa *hashtable*, por sua vez, utiliza uma estrutura de lista ligada e suas operações de inserção, remoção e busca.

As aplicações do pacote STAMP escolhidas para os experimentos foram `Genome` e `Vacation`, baseado nos resultados da Figura 3.5. Além dessas duas aplicações, apenas

`Yada` e `Intruder` poderiam mostrar melhorias. As aplicações `K-means` e `SSCA2` apresentam speedups muito pequenos em software e devem oferecer melhores resultados em hardware (HTM). A aplicação `Labyrinth` apresenta uma diferença muito pequena comparada com a versão manual, e portanto, não seria interessante analisá-la.

Em contraste às aplicações `Genome` e `Vacation`, `Yada` e `Intruder` se mostraram bem mais complexas de se analisar e aplicar o pragma criado corretamente, necessitando de um conhecimento maior sobre as duas aplicações e ferramentas mais específicas.

As seguintes implementações de TM foram avaliadas:

- **NOrec:** Esse código foi desenvolvido pelo Grupo de Sincronização de Rochester (*Rochester Synchronization Group*) e lançado como parte do pacote RSTM [21]. `Norec` aplica um *lock* de sequência global, usa versionamento deferido e detecção de conflitos adiantada. Essa implementação será referida como instrumentação manual.

- **libitm:** Biblioteca de tempo de execução usada pelo GCC para gerar código transacionais. Implementação semelhante à `NOrec` para os métodos de *start, abort, commit* e barreiras de leitura/escrita encapsuladas pelas implementações das funções da ABI da Intel[15].

- **libitm+elidebar:** Essa é a implementação `libitm` anotadas com o pragma implementado.

Os resultados apresentados neste capítulo representam a média de 20 execuções. Todas as aplicações e implementações de TM foram compiladas pelo GCC (GNU C/C++) versão 6.2, modificado com a implementação do mecanismo de omissão de barreiras, e usando otimização nível três (-O3). Os experimentos foram conduzidos em uma máquina com as seguintes configurações: processador Intel Core i7-2600K 3.40GHZ com 8GB de RAM. O processador i7-2600k possui 4 núcleos físicos com 2 threads por núcleo (Total de 8 threads). A máquina utiliza CentOS 6.9 e Linux kernel 3.14. A análise faz uso da benchmark STAMP [22], em particular das aplicações genome e vacation. O binário das aplicações com barreiras inseridas manualmente (`NOrec`) e automática (`libitm`) foram obtidas compilando o mesmo código fonte.

5.3 Resultado Obtidos

A Figura 5.3 mostra o speedup relativo à versão sequencial para as aplicações genome e vacation, com a execução das implementações manualmente instrumentada (`NOrec`), automaticamente instrumentada com pragma (`libitm+elidebar`) e sem pragma (`libitm`).

Como a Figura 5.3 mostra, ambas as aplicações escalam bem com `NOrec`. Porém, nenhum tipo de speedup foi observado com a versão gerada pelo GCC sem pragmas (`libitm`). Para a aplicação genome, a maior parte do seu tempo de execução é gasto em uma fase

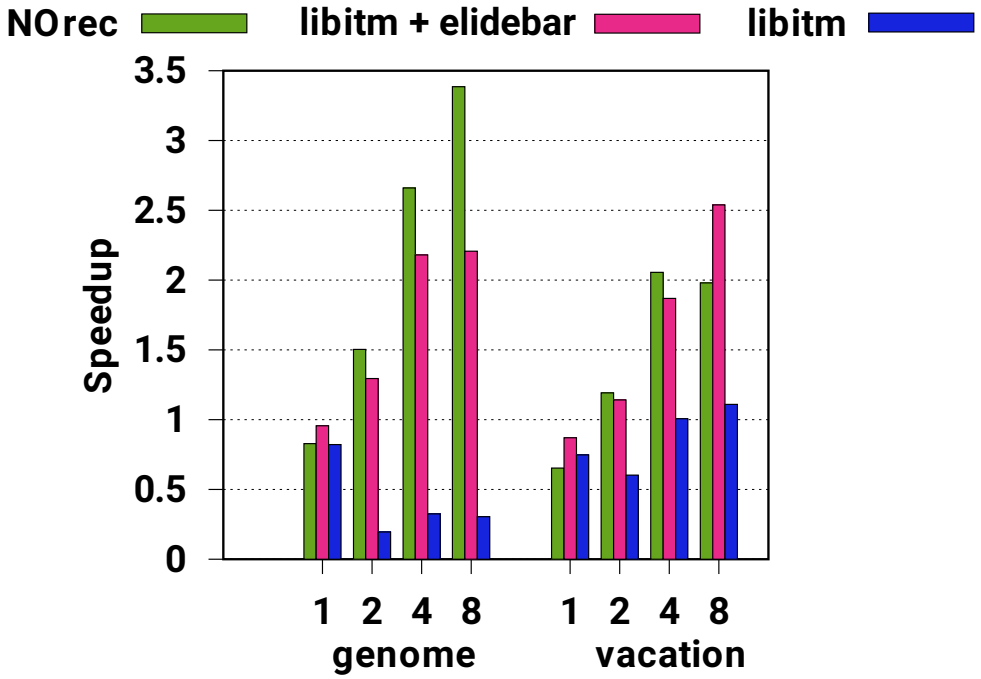

Figura 5.3: Speedup relativo à versão sequencial para as aplicações genome e vacation.

onde a maioria das barreiras estão acessando dados que são do tipo somente leitura, e portanto, podem ser omitidas. Como essas barreiras são existem na versão manual, é possível entender o porquê a aplicação escala tão bem nesse tipo de implementação. Quando as barreiras desnecessárias são elididas com o pragma criado, um significante ganho de desempenho é observado, como os resultados de `libitm+elidebar` mostram. A implementação `libitm+elidebar` é 7.2x mais rápida que a implementação `libitm` com 8 threads para a aplicação genome. Além disso, a implementação `libitm+elidebar` perde para `NOrec` apenas por 54%. Esses resultados foram obtidos, principalmente, por usar somente um pragma elidebar na função `tm_strcmp` encontrada no arquivo fonte `sequencer.c` (Figura 5.1).

No caso da aplicação vacation, ela gasta a maior parte de seu tempo realizando operações de pesquisa (lookup) em um mapa estruturado. Este mapa é parte das bibliotecas STAMP e é implementado como uma árvore rubro-negra por padrão. Apesar da maioria das barreiras inseridas pelo compilador, ou manualmente, são, de fato, requeridas para questões de corretude, foi descoberta uma oportunidade de otimização. Dado que a semântica de uma pesquisa apenas retorna `true` apenas se, e somente apenas, a estrutura de dados possuir o elemento procurado, conclui-se que a única barreira requeridas nessas operações são aquelas que garantem que nenhum tipo de violação ocorra. Como consequência, ao adicionar uma flag de validação ao item sendo manipulado pelo mapa, é possível elidir todas as barreiras anteriormente inseridas através do pragma implementado, já que todas as operações do mapa checam agora essa flag de validação através de barreiras. É suficiente abortar uma pesquisa caso um elemento inválido seja encontrado durante sua busca. Operações de remoção limpam a flag de validação, e operações de

inserção a ativa. Com essa otimização, notou-se um significante ganho de desempenho como podemos ver na Figura 5.3. Com a maiorias das barreiras elididas graças ao pragma, `libitm+elidebar` é 2.2x mais rápida que `libitm`.

Durante a realização desses experimentos ficou evidente que apesar da nova cláusula pragma ser uma ferramenta muito poderosa na extração de maior paralelismo em aplicações transacionais, o seu uso não é trivial. Em particular, a aplicação `Yada` apresentou centenas de barreiras de leitura e de escrita, apresentando enormes dificuldades para obter um ganho no desempenho. Dessa forma, espera-se que a cláusula apresentada nessa dissertação seja usada por programadores experientes e que tenham um bom conhecimento das aplicações que eles estão utilizando.

Capítulo 6

Conclusão

Esta dissertação detalhou o grande potencial que a abordagem baseada em Memória Transacional tem em simplificar como aplicações paralelas são desenvolvidas. Dito isso, conduziu-se um estudo mostrando as deficiências atuais que compiladores do estado da arte, como o GCC, têm apresentado para geração de código transacional. Apresentou-se a principal causa de tais deficiências: o fenômeno chamado instrumentação excessiva (over-instrumentation).

A principal contribuição do trabalho foi o desenvolvimento de um novo suporte de linguagem ao GCC através de uma cláusula pragma com o intuito de resolver a instrumentação excessiva. A nova cláusula pragma permite que o programador instrua o compilador a elidir barreiras para variáveis que forem selecionadas pelo pragma. O texto detalhou como a implementação da cláusula foi feita. Vale notar que o desenvolvimento apresentou bastante obstáculos ao longo de todo o seu processo. O GCC, apesar de um robusto compilador, oferece uma documentação bastante escassa, além de exigir do programador amplo conhecimento de sua implementação para conseguir realizar as adições desenvolvidas neste trabalho. O GCC foi escolhido por ser um compilador bastante popular, além de que o compilador geralmente usado pela academia, o compilador LLVM, não oferece suporte à memória transacional.

Para evidenciar o potencial da nova cláusula pragma, foram realizados múltiplos experimentos. Os resultados apresentados usando o pragma proposto foram muito promissores, mostrando ganhos de até 7.2x para aplicações utilizando o pragma quando comparadas com o código gerado pelo GCC. Dessa forma, é evidente a grande oportunidade de ganho de desempenho ao resolver os problemas de instrumentação excessivas que estão presentes na geração de código transacional.

Apesar do uso do pragma ser simples, saber onde usá-lo mostrou-se um grande desafio, especialmente para aplicações não criadas pelo próprio programador. O maior problema reside na escassez de ferramentas que ajudem a realizar uma análise mais profunda de aplicações transacionais. É importante notar, porém, que o pragma criado é uma poderosa ferramenta para programadores experientes usarem em suas aplicações e melhorarem

significativamente o desempenho das mesmas.

Considerando as dificuldades levantadas, trabalhos futuros podem envolver a criação de uma ferramenta de perfilamento de aplicações transacionais. Nela, as variáveis instrumentadas mais acessadas durante a execução, e o tempo de execução associado a elas seriam exibidos. Dessa forma, o programador conseguiria atacar as variáveis com instrumentação em excesso de maneira muito mais fácil, visto que o número de variáveis que ele deve analisar seria severamente reduzido.

Referências Bibliográficas

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha e T. Shpeisman. Compiler and runtime support for efficient software transactional memory. Em *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 26–37, junho de 2006.

[2] F. M. Carvalho e J. Cachopo. Runtime elision of transactional barriers for captured memory. Em *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming*, pp. 303–304, fevereiro de 2013.

[3] L. Dalessandro, M. F. Spear e M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. Em *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*, pp. 67–78, janeiro de 2010.

[4] E. W. Dijkstra. Cooperating sequential processes. Relatório Técnico EWD-123, Technological University, 1965.

[5] A. Dragojevic, Y. Ni e A.-R. Adl-Tabatabai. Optimizing transactions for captured memory. Em *Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 214–222, agosto de 2009.

[6] P. Felber, C. Fetzer, P. Marlier e T. Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, dezembro de 2010.

[7] P. Felber, C. Fetzer e T. Riegel. Dynamic performance tuning of word-based software transactional memory. Em *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, pp. 237–246, fevereiro de 2008.

[8] J. Gray e A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[9] T. Harris, J. Larus e R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2 edição, junho de 2010.

[10] T. Harris, M. Plesko, A. Shinnar e D. Tarditi. Optimizing memory transactions. Em *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 14–25, junho de 2006.

[11] M. Herlihy, V. Luchangco, M. Moir e W. N. Scherer. Software transactional memory for dynamic-sized data structures. Em *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pp. 92–101, julho de 2003.

[12] M. Herlihy e J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. Em *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 289–300, junho de 1993.

[13] M. Herlihy e N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[14] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, outubro de 1974.

[15] Intel Corporation. *Intel Transactional Memory Compiler and Runtime Application Binary Interface*, 1.1 edição, 2009.

[16] Intel Corporation. *Intel® Architecture Instruction Set Extensions Programming Reference*, fevereiro de 2012.

[17] C. Jacobi, T. Slegel e D. Greiner. Transactional memory architecture and implementation for IBM system z. Em *Proceedings of the 45th ACM/IEEE International Symposium on Microarchitecture*, pp. 25–36, dezembro de 2012.

[18] R. Kalla, B. Sinharoy e J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, março/abril de 2004.

[19] J. R. Larus e R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[20] C. Lattner e V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Em *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar de 2004.

[21] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer e M. L. Scott. Lowering the overhead of nonblocking software transactional memory. Em *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, junho de 2006.

[22] C. C. Minh, J. Chung, C. Kozyrakis e K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. Em *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 35–46, setembro de 2008.

[23] W. Ruan, Y. Liu, C. Wang e M. Spear. On the platform specificity of STM instrumentation mechanisms. Em *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 1–10, fevereiro de 2013.

[24] H. Sutter e J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, setembro de 2005.

[25] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera e M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. Em *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pp. 127–136, setembro de 2012.

[26] C. Wang, W.-Y. Chen, Y. Wu, B. Saha e A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. Em *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 34–48, março de 2007.

[27] R. M. Yoo, C. J. Hughes, K. Lai e R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. Em *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pp. 19:1–19:11, New York, NY, USA, 2013. ACM, Denver, Colorado.

[28] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai e H.-H. S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. Em *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 265–274, junho de 2008.