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

FGSCM: uma abordagem de omissão de lock transacional com granularidade fina na resolução de conflitos

Gustavo José de Sousa

Rio Claro – SP
2017

Gustavo José de Sousa

FGSCM: uma abordagem de omissão de lock transacional com granularidade fina na resolução de conflitos

Orientador: Prof. Dr. Alexandro José Baldassin

Dissertação de Mestrado elaborada junto ao Programa de Pós-Graduação em Ciência da Computação – Área de Concentração em Computação Aplicada, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Rio Claro – SP
2017

Sousa , Gustavo José de.
FGSCM: uma abordagem de omissão de lock transacional com granularidade fina na resolução de conflitos / Gustavo José de Sousa. -- São José do Rio Preto, 2017
57 f. : il.

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

1. Computação. 2. Programação (Computadores) 3. Sincronização. 4. Software. 5. Omissão de lock. 6. Memória transacional. I. Universidade Estadual Paulista "Júlio de Mesquita Filho". Instituto de Biociências, Letras 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

FGSCM: uma abordagem de omissão de lock transacional com granularidade fina na resolução de conflitos

Gustavo José de Sousa

30 de agosto de 2017

Banca Examinadora:

- Prof. Dr. Alexandro José Baldassin (Orientador)
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista – UNESP
- Prof. Dr. Orlando de Andrade Figueiredo
  Departamento de Estatística, Matemática Aplicada e Computação
  Universidade Estadual Paulista – UNESP
- Prof. Dr. Rodolfo Jardim de Azevedo
  Instituto de Computação
  Universidade Estadual de Campinas – UNICAMP

Rio Claro – SP
30 de agosto de 2017

Agradecimentos

A Deus, por renovar minhas forças a cada amanhecer e me permitir chegar até aqui.

A minha esposa, *Natália*. Muito obrigado por seu apoio e gestos de encorajamento, isso me fortaleceu de maneira crucial! *Você é minha inspiração.*

A meus pais, por todo o suporte provido durante esta jornada, desde o ensino básico infantil até a elaboração desta dissertação. Tenho certeza que não alcançaria isto sem vocês!

Ao meu mais estimado professor, Dr. Alexandro José Baldassin. Obrigado por sua influência em minha vida acadêmica e profissional, e por seu tempo, algo muito precioso, investido na orientação deste trabalho.

A todos os educadores por quem passei durante minha caminhada acadêmica. Eu reconheço a importância de seu trabalho em minha vida e de muitos outros.

A todos aqueles que, de alguma forma, me acompanharam durante o desenvolvimento deste trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

RESUMO

Omissão de lock é uma técnica onde operações de aquisição e liberação de lock são omitidas (especulação) de forma a permitir que regiões críticas compartilhando um mesmo lock possam executar concorrentemente, permitindo assim se explorar um nível maior de concorrência em programas que utilizam esse método popular de sincronização. Para se manter o princípio de atomicidade, as modificações no estado do programa realizadas pela região crítica são mantidas em um *buffer* interno e são efetivadas apenas ao fim da mesma. Em caso de inconsistências, diferentes políticas em como proceder são possíveis, o que diferencia as diversas abordagens de omissão de lock encontradas na literatura. Por exemplo, a abordagem original, *Speculative Lock Elision* (SLE), que é implementada no nível microarquitetural, recorre a adquirir o lock de forma tradicional quando uma especulação falha. Em algumas situações, esta política conservadora acaba por restringir o ganho em desempenho originalmente pretendido por impor um volume de sincronização desnecessário (*lemming effect*). Uma forma de superar tal limitação é o emprego de omissão de lock transacional (Transactional Lock Elision, em inglês), onde a especulação de regiões críticas se dá por meio de transações e o controle de execução é devolvido ao software em eventos de transações abortadas, o que permite que diferentes estratégias sejam empregadas com o objetivo de permitir execução concorrente mesmo em presença de falha de especulação. Neste contexto, uma das abordagens possíveis é o esquema chamado *Software-assisted Conflict Management* (SCM), onde um lock auxiliar é utilizado para sincronizar transações abortadas e, assim, manter o lock original livre, permitindo que outras transações prossigam sua execução. No presente trabalho, uma extensão ao SCM é proposta, o esquema *Fine-grained Software-assisted Conflict Management* (FGSCM), onde múltiplos locks são utilizados para permitir que transações abortadas por conflitos em diferentes regiões de memória possam ser executadas de forma concorrente. O algoritmo proposto foi implementado utilizando a interface RTM da extensão Intel® TSX e experimentos foram realizados em um máquina quadcore, para os quais, em casos com predominância de operações de leitura em memória, observou-se um ganho em desempenho médio de 11% e 36% com relação à abordagem SCM original e ao uso de um *spin lock* comum, respectivamente.

Palavras-chave: Programação Concorrente. Omissão de Lock. Memória Transacional.

ABSTRACT

Lock elision is a technique that omits acquire/release lock operations (speculation) so as to allow critical regions sharing the same lock to run concurrently, which yields a higher level of concurrency explored by programs that use such popular synchronization mechanism. In order to honor atomicity, modifications on the program's state made by the critical regions are kept in an internal buffer and only applied at the end of the speculation. If inconsistency is found, different policies on how to proceed are possible, which make up the several existing approaches found in the literature. As an example, the original one, namely Speculative Lock Elision (SLE), which is implemented at the level of microarchitecture, falls back to acquire the lock in a standard manner when there is speculation error. In some situations, such conservative policy ends up restricting the intended performance gains due to the unnecessary synchronization imposed (lemming effect). A way to address this issue is through Transactional Lock Elision (TLE) techniques, in which speculation of critical regions is done by means of transactions and execution control is passed back to software on abort events, which makes possible the use of different strategies to allow concurrent execution even in presence of speculation error. In this context, one possible approach is called Software-assisted Conflict Management (SCM), where an auxiliary lock is used to serialize aborted transactions and, as such, keep the original one free, so that others may proceed on their execution. The work presented in this document proposes an extension of SCM, called Fine-grained Software-assisted Conflict Management (FGSCM), where multiple auxiliary locks are applied in order to allow transactions aborted due to conflict on different regions of memory to be executed concurrently. The proposed algorithm was implemented by using the RTM interface from Intel®'s TSX extension and experiments were performed on a quadcore machine. On read-dominated workloads, an average performance gain of 11% and 36% was observed against the original SCM and a typical spin lock, respectively.

Keywords: Concurrent Programming. Lock Elision. Transactional Memory.

Capítulo 1

Introdução

Entre 1986 e 2002, a desempenho de microprocessadores aumentava em torno de 50% a cada ano. Entretanto, após 2002 esta taxa diminuiu para uma média de 20% ao ano. A principal razão para tal diminuição é o fato de que aumentar a frequência de processadores tem se tornado difícil devido a questões de consumo de energia e questões térmicas [14].

Com a estagnação da desempenho dos processadores com um único núcleo de processamento, em meados de 2005, as fabricantes de microprocessadores começaram a dar maior atenção ao paralelismo e começaram a desenvolver chips com múltiplos processadores (*multicore*). Esse tipo de processador utiliza memória compartilhada para realizar a comunicação entre as linhas de execução. Dada a possibilidade de mais de uma unidade de processamento acessar recursos compartilhados em intervalos de tempos sobrepostos, potencialmente causando inconsistência de dados, viu-se necessária a criação de mecanismos que sincronizassem o acesso a tais recursos. Nesse contexto, diferentes técnicas para sincronização de código paralelo foram propostas [26].

No modelo de memória compartilhada, uma técnica muito conhecida e adotada pela comunidade é o uso de estruturas popularmente conhecidas como *locks*. As duas operações básicas de aquisição e liberação de um lock definem, respectivamente, o início e o fim de uma região crítica. Resumidamente, regiões críticas que compartilham o lock não podem ser executadas em intervalos de tempos sobrepostos.

Um dos grandes problemas dessa técnica é a dificuldade de aplicar seu uso de forma a explorar um alto nível de paralelismo, o que geralmente é feito com locks de granularidade fina. Portanto, muitas vezes seu uso é realizado de forma conservadora, onde muitas sincronizações são feitas desnecessariamente, caso muito comum com abordagens de locks com granularidade grossa.

Por outro lado, um outro paradigma, que pode ser considerado ainda emergente, é o de memória transacional [13], onde o programador declara *regiões atômicas* do programa, cujas instâncias em execução são chamadas de *transações*. A sincronização das transações é realizada de forma transparente ao programador e idealmente apenas quando necessária. Uma vez que não é necessário controle explícito de como a sincronização é realizada, trata-

se de uma técnica significantemente mais simples que o uso de locks. No entanto, um dos fatores complicantes à adoção desse paradigma é a larga existência de aplicações paralelas baseadas em locks.

Uma outra abordagem que tem se tornado popular no meio acadêmico recente é a de *omissão de locks* [1], onde regiões críticas são executadas de forma especulativa, sem haver aquisição do lock, o que permite que regiões críticas possam ser executadas concorrentemente. Durante a execução especulativa, alterações em memória e do estado arquitetural são mantidas em um estado transacional e apenas efetivadas ao fim da execução da região crítica, caso não haja conflito de dados com outra região crítica em execução. Em presença de conflitos a especulação pode ser reiniciada ou a sincronização pode ser forçada. Tal abordagem parte da ideia de que, por eventualmente não acessarem recursos de forma conflitante, nem sempre é necessário haver serialização entre duas ou mais regiões críticas que compartilham um mesmo lock.

Omissão de locks pode ser implementada de tal forma que aplicações existentes baseadas em locks possam adotar seu uso com nenhuma ou mínima modificação do código fonte. Por isso, dada sua "natureza transacional", uma das vantagens providas por essa técnica é que tais aplicações podem herdar os benefícios inerentes ao paradigma de memória transacional de forma implícita, sem necessidade de adaptação do código fonte. Outra grande vantagem do esquema de omissão de locks é que ele permite que o programador se beneficie da simplicidade do uso de locks com granularidade grossa e do ganho de desempenho devido ao refinamento de sincronização e redução de outras penalidades impostas pelo uso de locks em sua forma tradicional.

Apesar disso, o modelo original de omissão de locks introduz um problema chamado *lemming effect* [17], que adiciona sincronização desnecessária, mesmo que em menor escala em comparação com locks tradicionais. Trata-se de um fenômeno onde, a partir do momento que uma *thread* desiste da execução especulativa e resolve adquirir o lock de maneira padrão, todas as regiões regiões críticas em execução compartilhando tal lock são contidas até que o mesmo esteja livre. Diferentes propostas de melhorias surgiram para amenizar tal problema, entre elas uma chamada de *Software-assisted Conflict Management* (SCM) [2], onde se utiliza um lock auxiliar para enfileirar transações abortadas, deixando o lock original livre.

1.1 Objetivos

Como é observado no Capítulo 4, existe a possibilidade de se explorar um nível maior de concorrência com SCM ao se modificar seu algoritmo original de forma a permitir que diferentes locks auxiliares sejam utilizados para a serialização de transações conflitantes. Esse novo algoritmo é proposto nesse trabalho como *Fine-grained Software-assisted Conflict Management* (FGSCM).

Um ponto crucial nesse novo algoritmo é como escolher qual lock auxiliar utilizar para uma transação conflitante a ser serializada. Tal decisão depende fortemente em *identificar a fonte do conflito de uma transação*, tarefa para a qual o hardware atual não provê o suporte arquitetural necessário. Por isso, se vê necessária uma maneira de simular tal funcionalidade, o que é feito neste trabalho através de um *microbenchmark*.

Portanto, em suma, o presente trabalho:

- propõe o esquema *Fine-grained Software-assisted Conflict Management* (FGSCM), onde múltiplos locks auxiliares são utilizados no caminho de serialização;
- implementa o algoritmo juntamente com um *microbenchmark* especializado, que provê as condições necessárias para testá-lo e permite simular diversas classes de programas paralelos;
- apresenta uma análise detalhada dos resultados experimentais obtidos de diversas execuções do *microbenchmark* utilizando diferentes variações de seus parâmetros. Um resultado interessante a se destacar em termos de desempenho é que a nova abordagem superou o esquema SCM e o uso de um *spin lock* tradicional em uma média de 11% e 36%, respectivamente, para aplicações com predominância de operações de leitura e tamanho total de seções críticas balanceado com o de seções não críticas.

As contribuições e resultados deste trabalho foram compilados em um artigo submetido e aceito pelo Simpósio Internacional de Arquitetura de Computadores e Computação de Alto Desempenho (SBAC-PAD), edição de 2017, com o título "FGSCM: A Fine-Grained Approach to Transactional Lock Elision".

O resto desta dissertação está organizado conforme o seguinte: o Capítulo 2 apresenta resumidamente os dois principais modelos arquiteturais de computação paralela, dando ênfase maior ao de memória compartilhada, bem como o paradigma de sincronização com locks e com memória transacional; o Capítulo 3 trata do assunto de omissão de locks, mostrando o modelo originalmente proposto e melhorias realizadas nos últimos anos; o Capítulo 4 apresenta o esquema FGSCM, proposto neste trabalho, bem como a metodologia utilizada para avaliá-lo e uma análise detalhada dos resultados experimentais; finalmente, o Capítulo 5 conclui este trabalho e aponta para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta, de forma introdutória, fundamentos necessários para o entendimento do assunto principal deste trabalho. A Seção 2.1 apresenta uma breve classificação de tipos de arquiteturas de computadores bem como uma descrição de modelos de computação paralela; a Seção 2.2 mostra como estruturas comumente chamadas de locks podem ser utilizadas para a sincronização de *threads* e apresenta memória transacional como um paradigma alternativo que supera algumas das limitações impostas pelo uso tradicional de locks.

2.1 Modelos arquiteturais

Arquiteturas de computadores podem ser classificadas de acordo com a quantidade de fontes de instruções que podem ser executadas e de dados que podem ser manipulados, ambos simultaneamente. A taxonomia usualmente usada para classificar esses modelos é devido a Flynn [12].

Arquiteturas de computadores SISD (*Single Instruction stream, Single Data stream*[1]) podem executar apenas uma fonte de instruções (em outras palavras, um programa[2]) e manipular um item lógico de dados por instrução em um dado momento. Este modelo é considerado serial na visão do programador. Há processadores SISD que utilizam pipelines para permitir que mais de uma instrução seja executada por ciclo de *clock* [14]. Embora isto possa ser considerado paralelismo, tal paralelismo se dá ao nível de instrução (ILP – *Instruction Level Parallelism*) de tal forma que o programador não o explora diretamente. Além disso, embora *pipelining* permita execução de múltiplas instruções por ciclo, tais instruções são provenientes de um único programa, isto é, uma única fonte de instruções.

Arquiteturas SIMD (*Single Instruction, Multiple Data*) permitem que uma instrução seja executada sobre vários conjuntos de dados simultaneamente. Tal recurso é útil quando

[1] É muito comum a omissão do termo "*stream*".

[2] O termo "fonte de instruções" aqui é preferível, uma vez que pode-se confundir "programa" com um processo em execução (que pode ter várias *threads*).

se trabalha com vetores de dados onde é realizada uma mesma operação para todos os itens ou uma região contígua do vetor. Em um programa, essas instruções geralmente residem dentro de um laço e compiladores o otimizam ("vetorizam") de forma a explorar o paralelismo provido por este tipo de arquitetura (programadores podem também utilizar o recurso explicitamente). Os processadores para computadores *desktop* mais recentes possuem suporte a SIMD. Processadores gráficos (GPUs – *Graphics Processing Unit*) são outro exemplo onde a vetorização é bastante utilizada.

Arquiteturas MIMD (*Multiple Instruction, Multiple Data*) podem ser vistas como uma extensão de SIMD, de forma que mais de uma fonte de instruções podem ser executadas ao mesmo tempo. Enquanto pode-se imaginar sistemas SIMD como tendo uma única unidade de controle de instruções e várias unidades lógico-aritméticas, sistemas MIMD possuem unidades de processamento independentes que funcionam de forma assíncrona e cada uma executa diferentes fontes de instruções em um dado ponto no tempo. O fluxo do programa em sistemas SIMD é sequencial. Já em sistemas MIMD, um programa pode ter trechos de código que são executados paralelamente em mais de uma unidade de processamento. Virtualmente todos os processadores comerciais hoje (*multicore*) utilizam essa arquitetura.

Uma outra possível arquitetura, MISD (*Multiple Instruction, Single Data*), não possui fins práticos, sendo *arrays* sistólicas (*Systolic Array*) [18] o melhor exemplo desse tipo de computação.

Este trabalho foca basicamente em arquiteturas MIMD. Neste contexto, existem dois modelos principais: os de memória compartilhada e os de memória distribuída.

2.1.1 Memória distribuída

Os sistemas de memória distribuída, comumente chamados de *clusters*, consistem de uma rede de computadores, onde cada nó desta rede é considerado uma unidade de processamento que possui sua própria memória (privada).

Neste modelo, a comunicação entre os nós é feita geralmente por troca de mensagens ou uso de funções específicas para acesso de memória alheia. Outra alternativa é o uso de linguagens que provêm recursos de particionamento global de endereços, onde *arrays* são particionados entre os nós, de forma que cada um armazena parte dos dados e o acesso aos dados é feito como se a memória do sistema fosse contígua.

Sistemas de memória distribuída são bem mais baratos para se escalar eficientemente que no caso dos de memória compartilhada. Existem sistemas hoje com milhares de unidades de processamento[3]. Por isso, este tipo de sistema é a melhor opção para se processar uma quantidade de dados massivamente grande.

Mais detalhes sobre este tipo de sistema, como topologias de rede e paradigmas utili-

[3]Bons exemplos são os *clusters* do NERSC (*National Energy Research Scientific Computing Center*).

zados, podem ser encontrados na obra de Pacheco [22]. Eles não são cobertos aqui uma vez que o presente trabalho foca em um paradigma utilizado em sistemas de memória compartilhada.

2.1.2 Memória Compartilhada

Os sistemas de memória compartilhada possuem um ou mais processadores de múltiplos núcleos e um espaço de endereçamento comum a todos os núcleos. A comunicação entre os núcleos é feita através de instruções de leitura e escrita em um espaço de memória compartilhada.

Quando o sistema tem mais de um processador [4], dois tipos de arquiteturas são possíveis: i) existe um bloco de memória que é acessada por todos os processadores; ii) cada processador tem seu bloco de memória e existe hardware adicional que permite a outros processadores acessarem seu bloco. O primeiro tipo de arquitetura é chamado UMA (*Uniform Memory Access*) enquanto o último, NUMA (*Nonuniform Memory Access*) [14]. A principal diferença entre essas arquiteturas é a latência de acesso à memória: enquanto em UMA o acesso a uma variável em memória é o mesmo para todos os processadores, em NUMA essa latência depende de quão próximo à memória o processador realizando o acesso está. Virtualmente todos os sistemas atuais são NUMA.

No modelo com memória compartilhada, uma ou mais rotinas chamadas de *threads* são executadas em paralelo e assincronamente. As implementações podem utilizar *threads* dinâmicas ou estáticas. *Threads* dinâmicas são aquelas que são inicializadas, quando necessário, pela *thread* principal para execução de tarefas e são finalizadas quando terminam. Este método traz a vantagem de economia de recursos do sistema, uma vez que as *threads* são criadas apenas quando necessário e os recursos alocados para elas são liberados quando terminam.

No paradigma de *threads* estáticas, um número de *threads* é inicializado no início do programa e essas são finalizadas ao final do programa. Neste caso, geralmente a *thread* principal atribui tarefas a serem realizadas pelas *threads* criadas. Muitas vezes o trabalho principal do programa é realizado por essas *threads* e a *thread* principal é responsável apenas por coordená-las.

A escolha de qual paradigma utilizar depende muito do problema tratado. Por exemplo, um servidor de aplicação possui períodos de tempo de ociosidade, e manter *threads* ativas significa recursos alocados não sendo utilizados. Outra questão é que o número de requisições sendo processadas em um determinado instante variam. Assim, utilizar *threads* estáticas não parece uma boa ideia para este tipo de aplicação, sendo o uso de *threads* dinâmicas uma melhor abordagem. Entretanto, a criação e finalização de *threads* são

[4] É importante notar que o termo "processador"pode se referir a um conjunto de núcleos de processamento. Por exemplo, um processador *multicore* possui mais de um núcleo.

procedimentos custosos, por isso há problemas em que o uso de *threads* estáticas é uma melhor alternativa. Geralmente, tais problemas mantêm as *threads* ocupadas durante a maior parte do tempo de execução do programa.

Em alguns casos, uma solução híbrida também pode ser utilizada. Ainda tomando o exemplo de um servidor de aplicações, supondo que o mesmo receba uma quantidade média mínima de requisições constantemente, um número equivalente de *threads* pode ser mantido ativo de forma a evitar o *overhead* de criar novas *threads* para grande parte das requisições. Caso haja mais requisições que o número de *threads* estáticas, então novas *threads* podem ser criadas para processar tais requisições e finalizadas quando o trabalho estiver pronto.

As variáveis utilizadas pelos programas em memória compartilhada podem ter uma das duas classificações no contexto de *threads*: variáveis privadas e variáveis compartilhadas. Uma variável privada, como o nome sugere, pode apenas ser acessada pela *thread* a qual ela pertence. Por outro lado, uma variável compartilhada é visível por todas as *threads*[5] e é o principal mecanismo de comunicação utilizado.

Uma vez que variáveis compartilhadas são acessíveis a mais de uma *thread*, são possíveis casos em que mais de uma *thread* acessem uma variável compartilhada "ao mesmo tempo", onde pelo menos uma tente modificar seu valor. Tal tipo de situação, também conhecida como *condição de corrida* na literatura [22], potencialmente causa inconsistência e comportamentos inesperados do programa. São, portanto, necessários mecanismos de sincronização entre as *threads* que possam evitar tais eventos. A próxima seção trata especificamente desse assunto.

2.2 Sincronização

Dada a natureza não-determinística do modelo paralelo, mecanismos de sincronização para evitar condições de corrida são necessários. Como dito anteriormente, condições de corrida ocorrem quando mais de uma *thread* entram em seções de códigos que manipulam um recurso compartilhado em intervalos de tempo sobrepostos, o que pode levar o programa a comportar-se de maneiras inesperadas, além de produzir resultados inesperados e possivelmente incorretos. Estas seções do programa são denominadas *seções críticas*. O termo *recurso* aqui pode referir-se a qualquer coisa utilizada pelo programa. Mesmo não se tratando de valores de variáveis (*data streams*, por exemplo), eles são referenciados por variáveis de alguma forma, que neste contexto são compartilhadas.

[5]Em sistemas operacionais típicos, variáveis compartilhadas são somente visíveis por *threads* dentro de um mesmo processo; ou por processos "bifurcados".

2.2.1 Locks

Um dos mecanismos de sincronização é o de *bloqueio de exclusão mútua* pelo uso de um tipo de variável comumente chamado de *mutex* [22]. Trata-se de uma variável compartilhada, geralmente relacionada a um recurso compartilhado, que é utilizada para definir início e fim de regiões críticas do programa. Antes de entrar em uma região crítica, uma *thread* adquire exclusividade para o recurso a ser manipulado chamando uma função *lock* no respectivo *mutex*; ao sair da região crítica, uma função *unlock* libera o *mutex*.

É comum encontrarmos o simples uso do termo *lock* para se referir ao *mutex*. Assim, o ato de chamar as funções lock e unlock no *mutex* são simplesmente ditas como adquirir e liberar o lock, respectivamente. Esta notação será preferencialmente utilizada aqui por se tratar de uma forma mais simples e geral de se referir a este tipo de mecanismo.

Quando uma *thread* tenta adquirir um lock, se o mesmo já tiver sido adquirido por outra *thread* e ainda não liberado, aquela entra em estado de bloqueio, esperando ser "sua vez" de entrar na região crítica. Quando uma *thread* libera um lock, se há *threads* em estado de bloqueio esperando a liberação do *mutex*, uma delas é desbloqueada e recebe exclusividade naquele lock, entrando então na região crítica.

É importante notar que no código do programa podem existir mais de uma região crítica relacionada a um determinado recurso compartilhado. Também pode haver mais de um recurso compartilhado, de forma que muitas vezes é necessário o uso de mais de um lock.

Em alguns casos, *threads* só podem continuar sua execução quando uma determinada variável compartilhada x recebe um valor específico através de uma outra *thread* B. Uma forma de conseguir tal comportamento é ter um laço onde uma *thread* A constantemente adquire o lock relacionado a x, verifica seu valor e então o libera. Isto é repetido até que o valor de x seja o esperado pela *thread* A. Embora tal abordagem seja válida, ela possui a desvantagem de repetidamente realizar operações de lock e unlock, relativamente custosas, além de gastar tempo de execução do processador que poderia estar sendo utilizado por outras *threads*. Uma alternativa mais eficiente é o uso de *variáveis de condição* (*condition variables* [22]).

Variáveis de condição é um outro método de sincronização que é utilizado em conjunto com locks. Uma variável de condição nada mais é que um objeto utilizado pelas *threads* para se comunicar através de *sinais*. Uma variável de condição sempre está associada a um lock. Em resumo, seu funcionamento se dá conforme o seguinte. Uma *thread* A, após ter adquirido o lock associado, pode esperar por uma sinalização na variável de condição v. Quando isto acontece, a execução de A é suspensa, ou seja, a *thread* "dorme" até que ela seja "acordada" por um sinal em v. Uma outra *thread* B em execução, com o lock associado adquirido, sinaliza a variável de condição v, e então libera o lock. Assim que B libera o lock, se houver *threads* "dormindo" e esperando por um sinal em v, uma delas é

"acordada" (não necessariamente A) e a posse do lock associado a v é entregue à mesma, que eventualmente o libera. Uma outra operação possível é a *difusão* (*broadcast*) de sinal em v, de forma que todas as *threads* esperando pelo sinal o recebem em vez de apenas uma.

Aplicando este método no caso apresentado anteriormente, a *thread* A não precisa mais verificar o valor de x repetidamente em conjunto com as operações de lock e unlock. Em vez disso, ainda usando um laço, mas com um única operação de lock e unlock por fora do laço, a *thread* A espera em uma variável de condição v. Quando A é acordada pelo sinal em v, ela verifica o valor de x e sai do laço caso seu valor seja o esperado. No caso da *thread* B, esta mandaria (ou difundiria) um sinal em v após a alteração de x.

Essa abordagem é mais eficiente, uma vez que as operações de lock e unlock são realizadas apenas uma vez e o processador fica livre para outras *threads* enquanto a *thread* A espera pelo sinal em v. Um exemplo onde o uso de variáveis de condição faz-se necessário é no problema do *buffer* limitado (*bounded buffer*) [11]. Nesse caso, dois tipos de *threads* existem: as consumidoras, que retiram elementos do *buffer*, e as produtoras, que inserem elementos no *buffer*. *Threads* consumidoras e produtoras executando concorrentemente precisam articular o acesso ao *buffer*, o que pode envolver espera (por exemplo, uma *thread* consumidora não pode executar enquanto o *buffer* estiver vazio).

Existem outros métodos de sincronização interessantes que não são discutidos neste trabalho, como semáforos, barreiras e monitores [22]. Para alguns problemas, seu uso pode ser mais conveniente que utilizando apenas *mutexes*. No entanto, *mutexes* provêm maior flexibilidade, dando a habilidade de codificar comportamentos que não seriam possíveis utilizando os métodos alternativos citados.

Um programa deve ter o menor número de regiões críticas possível e uma região crítica deve conter o mínimo de código possível, uma vez que adquirir e liberar locks é um processo custoso e regiões críticas "serializam" o programa em relação aos recursos manipulados nelas. O impacto da serialização no desempenho é ditado pela Lei de Amdahl [5], sendo representado pela seguinte equação:

$$S(p) = \frac{1}{f_s + \frac{1-f_s}{p}}$$

$S(p)$ é o *speedup*, f_s a fração sequencial do código e p o número de processadores. Como exemplo, suponha que 50% do código seja serial. Independente do número de processadores utilizados, o máximo *speedup* conseguido nesse caso é 2.

Além disso, é importante que sua implementação de programas utilizando locks seja livre de *deadlocks*, um erro muito comum enfrentado por programadores que será descrito dentro da próxima subseção.

2.2.2 Limitações relacionadas ao uso de locks

Enquanto definir regiões críticas com o uso de locks é uma das mais populares formas de sincronização utilizadas pelos programadores, ela possui certas desvantagens. Embora algumas delas sejam fortemente ligadas ao paradigma, como, por exemplo, a inabilidade de composição de rotinas utilizando locks, grande parte das limitações impostas pelo uso de locks é solucionada ou amenizada por técnicas de omissão de locks. As questões mais relevantes são apresentadas a seguir.

2.2.2.1 Paralelismo *versus* programabilidade

A utilização de locks pode ser classificada de duas formas com respeito a granularidade: (i) *granularidade grossa* e (ii) *granularidade fina*. O uso de locks com granularidade grossa se dá quando o lock está vinculado a uma estrutura de dados como um todo, de forma que este protege o acesso a todos os componentes da estrutura uma vez que é adquirido. Já locks de granularidade fina são responsáveis por proteger cada um dos componentes relevantes de uma estrutura de dados isoladamente, de forma que existe um lock para cada componente.

Uma forma fácil de ilustrar a diferença entre as duas abordagens é pensando na tarefa de proteger uma lista de tamanho fixo que permite alterações concorrentes dos valores em cada índice. Com granularidade grossa, aloca-se apenas um lock para a lista inteira, de forma que operações de edição em qualquer índice da lista são serializadas. Já no caso de granularidade fina, existe um lock alocado para cada índice da lista, de maneira que, para a edição em determinado índice, apenas o respectivo lock é necessário.

A abordagem de granularidade grossa é considerada conservadora e possui suas vantagens: a facilidade e agilidade de desenvolvimento do software, além da facilidade de verificação da corretude do código. No entanto, tal conservadorismo limita o desempenho do programa, pois, em muitos casos, ocasiona um grande volume desnecessário de serialização. Esta sincronização desnecessária ocorre entre regiões críticas que acessam componentes diferentes da mesma estrutura de dados e que, portanto, não geram conflito de dados.

Tal problema é solucionado com locks de granularidade fina: por se tratar de um lock diferente para cada componente da mesma estrutura, a sincronização desnecessária observada no caso anterior não ocorre nesta abordagem. Porém, enquanto o paradigma anterior reduz a complexidade do código, granularidade fina aumenta significantemente a complexidade do código, tornando a tarefa de programar mais difícil e suscetível a erros, sendo *deadlock* o mais comum deles.

2.2.2.2 Contenção *versus overhead*

Contenção ocorre quando uma *thread* tenta adquirir um lock que já está em posse de outra, o que faz com que a primeira seja suspensa. O uso de locks resulta em um *overhead* de recursos utilizados: alocação de memória e tempo extra de CPU para as operações. Como visto na subseção anterior, quanto maior a granularidade dos locks, menos contenção ocorrerá, porém, por outro lado, maior será o *overhead* resultante do uso de locks.

2.2.2.3 Regiões críticas de apenas leitura

Um outro problema relacionado ao método de sincronização com locks é a serialização quando há apenas operações de leitura em um determinado recurso. Obviamente, tal sincronização é desnecessária, e nem mesmo o uso de locks de granularidade fina resolve este caso. Basta o código conter pelo menos uma operação de escrita em um recurso compartilhado para que seja necessário que todas as operações de leituras no mesmo sejam codificadas dentro de regiões críticas, o que é muito comum.

Tal situação pode ser ilustrada considerando um código em que existe um recurso compartilhado x para o qual há no código *apenas uma* operação de escrita e uma ou mais regiões onde é realizada leitura de x. Por causa daquela única operação de escrita, todos os acessos a x devem ser colocados em regiões críticas protegidas pelo lock vinculado ao recurso [6]. Caso dinamicamente haja muitas operações de leituras em x e poucas de escrita, em vários momentos operações de leituras concorrentes sem a presença de escrita concorrente serão desnecessariamente serializadas por estarem definidas dentro de regiões críticas, o que limita significativamente a desempenho do programa.

Vale mencionar que, para situações desse tipo, existem os *readers-writer* locks [22], que permitem operações de leituras concorrentes mas bloqueiam para operações de escrita. Para isso, rotinas de aquisição e liberação específicas para cada tipo de acesso são utilizadas. Embora esse tipo de lock permita leituras concorrentes, as implementações existentes ainda precisam sincronizar acessos a contadores ou *flags* internos e, por isso, há um certo *overhead* envolvido no uso desse tipo de lock.

2.2.2.4 *Deadlocks*, o pesadelo dos programadores

Deadlocks ocorrem quando *threads* entram em estado de dependência cíclica, de forma que uma *thread* aguarda por um evento (liberação de lock, sinal de variáveis de condição etc) a ser produzido por uma outra *thread*, que, por sua vez, aguarda diretamente ou indiretamente por um evento da primeira. Uma vez que *deadlock* é estabelecido, nenhuma das *threads* pode prosseguir.

[6] Pode haver casos em que é possível saber que apenas uma *thread* acessará o recurso compartilhado em um dado momento. Tal situação, porém, não é muito comum e está sendo descartada no exemplo.

Um exemplo simples de situação eventual de *deadlock* é um programa, com duas *threads* A e B e dois locks a e b, em que a *thread* A adquire a e b, nesta ordem, e a *thread* B o faz na ordem inversa. Eventualmente, em alguma execução arbitrária do programa, A e B terão sucesso ao adquirir a e b, respectivamente. Isto retrata uma situação de *deadlock*, pois a *thread* A tentará em seguida adquirir b, que está em posse de B, que por sua vez tentará adquirir a, que está em posse de A – ou seja, ambas as *threads* ficarão esperando uma pela outra "para sempre".

Embora seja fácil de encontrar o que causa o *deadlock* no exemplo acima– a ordem em que os locks são adquiridos –, muitas vezes pontos causadores de *deadlocks* no código são bem sutis. Isso faz situações de *deadlocks* serem difíceis de ser antecipadas durante a escrita do programa, e, quando ocorrem, são difíceis de se depuradar.

2.2.2.5 Outras questões

Além das questões apresentadas, existem outros problemas relacionados ao uso de locks. Um deles é a inversão de prioridade de *threads* em um sistema operacional [20]. Isso se dá quando uma *thread* com maior prioridade é bloqueada quando tenta adquirir um lock em posse de uma de menor prioridade, o que faz com que uma *thread* de menor prioridade esteja sendo executada quando uma de maior prioridade poderia/deveria estar em execução.

Também há o problema de *threads* esperarem muito tempo por uma *thread* que tenha sido suspensa por interrupções de diferentes tipos enquanto estava na região crítica.

2.2.3 Memória transacional

Memória transacional é um paradigma de programação paralela com implementação recente e em constante pesquisa [13]. Trata-se de um modelo de execução que trabalha com conceito de transações em memória e serve como alternativa a modelos de sincronização baseados em bloqueios, como locks. Nessa abordagem, programadores marcam trechos de código onde há acesso a variáveis compartilhadas como *blocos atômicos*, não precisando se preocupar em codificar como tais trechos devem ser sincronizados.

Uma transação, no contexto de memória transacional, é uma execução de um bloco atômico. Uma transação deve garantir que: (i) alterações no estado do programa não sejam visíveis para outras *threads* até que a transação complete com sucesso; (ii) em presença de falhas por falta de recursos ou conflitos de dados, a transação descarte os resultados parciais e reinicie sua execução; e (iii) o estado do programa seja consistente antes e depois da execução da transação.

As três principais vantagens em relação ao uso de locks são:

1. *Programabilidade.* O programador não precisa se preocupar em como garantir que

regiões críticas sejam sincronizadas *corretamente*. Basta declarar blocos de código como atômicos, de forma que a sincronização fique sob responsabilidade do sistema;

2. *Desempenho.* Como visto anteriormente, o uso de locks implica sincronização desnecessária, mesmo quando se aplica granularidade fina. No caso de memória transacional, blocos atômicos em execução concorrente são verificados *dinamicamente* por necessidade de sincronização, de acordo com os acessos realizados por cada um à memória. Assim, trechos de programa que seriam necessariamente serializados na abordagem com locks são executados concorrentemente no modelo de memória transacional, dado que não haja conflito de dados;

3. *Composição.* Composição de blocos atômicos pode ser feita simplesmente aninhando-os em um outro bloco atômico.

O suporte a memória transacional pode ser implementado em software (*Software Transactional Memory*) [25] ou em hardware (*Hardware Transactional Memory*) [15]. No caso do primeiro, acessos à memória originados do bloco atômico de uma transação em execução são encapsuladas por uma rotina pertencente à biblioteca que provê a implementação de memória transacional. No caso de memória transacional em hardware, já existe suporte em alguns processadores, incluindo processadores Intel® através da extensão TSX, que é apresentada a seguir.

2.2.4 Intel® *Transactional Synchronization Extensions* (TSX)

Intel® TSX [16] foi pioneiramente implementada no processador Intel® Haswell. Trata-se de uma extensão que provê duas interfaces de software que podem ser utilizadas pelo programador para definir blocos transacionais.

Uma delas, HLE (*Hardware Lock Elision*), provê dois prefixos de instruções que servem para indicar aquisição e liberação de locks, respectivamente, XACQUIRE e XRELEASE. O uso correto de tais prefixos possibilita a execução de regiões críticas utilizando omissão de lock, que será explorado com detalhes no Capítulo 3.

A outra, RTM (*Restricted Transactional Memory*), provê uma interface básica para execução de transações. Três instruções são providas nessa interface: (i) XBEGIN, para dizer que uma transação está sendo iniciada, recebe um operando representando o *offset* local para o endereço da instrução para onde o processador deve mudar o fluxo de controle caso a transação aborte; (ii) XEND, para encerrar e efetivar a transação; e (iii) XABORT, para explicitamente abortar a transação, recebe um operando imediato de 8 bits que é carregado no registrador EAX.

Capítulo 3

Omissão de Locks e Trabalhos Existentes

Basicamente, técnicas de omissão de lock omitem operações de aquisição e liberação de locks com o objetivo de permitir que regiões críticas que compartilham o lock executem concorrentemente. Inicialmente, este capítulo discute as vantagens de tal abordagem em relação ao uso tradicional de locks e apresenta uma visão geral do mecanismo. Em seguida, diferentes abordagens encontradas na literatura são descritas. Em particular, a abordagem chamada *Software-assisted Conflict Management* (SCM) é deixada para ser apresentada no Capítulo 4, cujo conteúdo é um estudo detalhado de uma extensão à mesma.

3.1 Omissão de lock e suas vantagens

Um dos grandes problemas relacionados ao uso de locks apresentados no capítulo anterior é o volume de sincronização desnecessária. Isso se dá pelo fato de o programador utilizar apenas informação estática quando codificando um programa, de forma que acessos a recursos compartilhados são estaticamente serializados (i.e., no código fonte) diante da menor possibilidade de conflitos de dados no contexto dinâmico do programa. A principal motivação de técnicas de omissão de locks é resolver tal problema, removendo serialização dinamicamente desnecessária de regiões críticas protegidas por locks.

Além da diminuição de serialização desnecessária, outra vantagem da abordagem de omissão de locks é que se remove a tensão entre desempenho e programabilidade imposta pelo uso tradicional de locks. Uma vez que se utiliza o contexto dinâmico para detectar conflitos e serializar as instruções, o programador pode se beneficiar da facilidade de codificar com locks com granularidade grossa e ainda assim contar com desempenho semelhante ou melhor que a obtida com locks de granularidade fina. Esta vantagem é fortemente associada a um dos benefícios do uso de memória transacional apresentados

na Seção 2.2.3 (item *Programabilidade*) justamente devido à natureza transacional do esquema de omissão de lock.

Além dos principais benefícios apresentados acima, omissão de locks, por sua natureza, acabam por resolver ou amenizar outras questões também, como problemas de contenção, *overhead* do uso de locks e inversão de prioridade.

3.2 Visão geral do mecanismo

Regiões críticas devem satisfazer duas condições para repeitar o princípio de atomicidade: (i) dados lidos não podem ser modificados por outras *threads* até que a região crítica tenha terminado e (ii) dados modificados não podem ser modificados ou lidos por outras *threads* até que a região crítica tenha terminado. A ideia por trás das técnicas de omissão de locks é que *nem sempre* é necessário o lock ser adquirido para que regiões críticas satisfaçam tal princípio.

De forma geral, o mecanismo de omissão de lock executa regiões críticas sem realmente adquirir o lock, permitindo que regiões críticas que compartilham o mesmo lock executem de forma concorrente e, assim, aumentando o nível de paralelismo explorado ao nível de *threads*. Para dar a aparência de atomicidade, o mecanismo mantém as modificações realizadas em uma região crítica sendo executada especulativamente em um *buffer* interno, fazendo tais modificações visíveis a outras *threads* apenas quando o fim da região crítica é alcançado com sucesso, isto é, sem conflitos de dados.

Dado que a aquisição do lock é omitida, conflitos de dados são claramente possíveis e são detectados pelo mecanismo. A forma como falhas na especulação são tratadas é o que diferencia as diferentes abordagens de omissão de lock existentes na literatura.

3.2.1 SLE e TLE

A primeira proposta de omissão de lock foi a chamada "Omissão de Lock Especulativa" [23] (*Speculative Lock Elision (SLE)*, em inglês), onde operações de aquisição e liberação de lock são detectadas no nível de microarquitetura e especulação é realizada para as instruções da região detectada. Em presença de falha de especulação (por conflito de dados ou falta de recursos), a especulação é descartada e o lock é adquirido efetivamente. Alguns processadores Intel® possuem suporte à interface HLE da extensão Intel® TSX [16], que pode ser considerada uma versão em hardware da funcionalidade proposta por SLE, com a diferença que instruções de aquisição e liberação de locks são explicitamente marcadas com os prefixos XACQUIRE e XRELEASE, respectivamente.

Abordagens posteriores, cujo pioneirismo em hardware se atribui ao grupo de pesquisa da antiga Sun Microsystems (hoje Oracle) no contexto do processador Rock [6, 10], utilizam memória transacional para executar as especulações e, por isso, podem ser classi-

ficadas como técnicas de omissão de lock transacional (*Transacional Lock Elision* (TLE), em inglês). Resumidamente, técnicas de TLE omitem as operações de aquisição e liberação do lock e encapsulam regiões críticas em transações. Uma grande vantagem de TLE sobre SLE é que TLE é mais flexível quando uma especulação falha: com TLE, o controle de execução é devolvido ao programa quando a transação encapsulando a região crítica é abortada, enquanto SLE simplesmente decide adquirir o lock de maneira tradicional. Tal flexibilidade desempenha um papel fundamental para se lidar com um fenômeno chamado *Lemming Effect*, discutido na Seção 3.4.

Antes do mecanismo de memória transacional estar disponível em hardware, já havia algumas pesquisas sobre a possibilidade de implementar TLE em software. Neste contexto, um sistema de *runtime* para permitir TLE em software é apresentado por [24], cuja implementação é similar a de sistemas de memória transacional em software, onde cada acesso à memória é instrumentado. Ainda no contexto de implementação em software, outra abordagem foi a chamada *Pessimistic Lock-Elision* [4] (PLE), que se baseava em um sistema pessimista de memória transacional em software.

Uma abordagem híbrida adaptativa utilizando transações em hardware, execução otimista em software ou a versão original do lock é apresentada em [9] com o nome de *Adaptative Lock Elision* (ALE). A biblioteca de *runtime* implementada coleta estatísticas de execução para usá-las na decisão de qual combinação dos três modos mencionados utilizar para cada região crítica encontrada.

Na mesma época em que ALE apareceu, duas diferentes abordagens foram propostas em [2]: *Software-assisted Lock Removal* (SLR) e *Software-assisted Conflict Management* (SCM). SRL faz a leitura do lock apenas no final da transação para evitar que especulações falhem de forma prematura devido a regiões críticas em execução com o lock adquirido. O esquema SCM serializa transações abortadas utilizando um lock auxiliar com o objetivo de permitir transações livres de conflito a continuarem executando de forma concorrente.

Trabalhos recentes utilizando TLE incluem as abordagens Amalgamated Lock-Elision [3] (AmLE); e RW-TLE e FG-TLE [8]. Todas empregam instrumentação do programa para permitir que as transações em hardware executem concorrentemente com as regiões críticas serializadas. Em AmLE, o caminho serializado é executado utilizando múltiplas pequenas transações em hardware. No caso de RW-TLE e FG-TLE, acessos à memória no caminho serializado são monitoradas de forma a permitir notificação de inconsistências ao hardware executando as transações. FG-TLE se diferencia de RW-TLE por ser menos restritivo: enquanto o lock está em posse de alguma thread, RW-TLE permite concorrência apenas para regiões críticas que não realizem escrita em memória, enquanto FG-TLE não impõe tal restrição.

A Figura 3.1 exibe uma linha do tempo com as abordagens citadas acima.

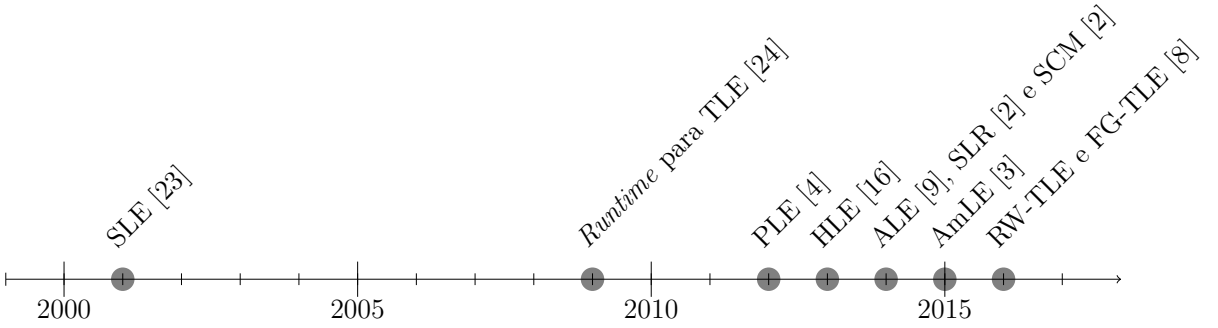

Figura 3.1: Linha do tempo das diferentes abordagens de omissão de lock

3.3 Omissão especulativa de locks (SLE)

SLE (*Speculative Lock Elision*) [23] foi a primeira proposta de omissão de locks. Trata-se de uma implementação no nível microarquitetural, onde instruções características de aquisição e liberação de lock são detectadas no programa em execução e as instruções encapsuladas por elas são executadas especulativamente. O mecanismo foi implementado e avaliado através de um simulador modelando um multiprocessador com tal capacidade.

A execução especulativa de uma região crítica é apresentada como um processo de quatro etapas:

1. *Quando uma operação de lock é detectada*, o processador prediz que as operações de memória ocorrerão atomicamente, ignorando, portanto, a operação que ratifica a aquisição do lock;

2. A seção crítica é executada e resultados são *armazenados em um buffer*;

3. *Se atomicidade não pode ser garantida pelo hardware*, considera-se um erro de especulação, recupera-se o estado arquitetural do início da seção crítica e, caso o número máximo de novas tentativas de especulação for alcançado, o lock é adquirido (i.e., região crítica não é executada especulativamente), senão, volta-se à etapa 1;

4. *Se uma operação de liberação do lock é detectada*, significa que não houve erro de especulação e, consequentemente, o princípio de atomicidade não foi violado. Portanto, a operação de liberação do lock é ignorada e as alterações parciais dos dados são efetivadas e feitas visíveis para as outras *threads*.

A Figura 3.2 mostra um exemplo de um código que define uma região crítica com aquisição e liberação de lock, além das respectivas instruções de máquina. O grafo à esquerda representa os possíveis fluxos de controle na região. O grafo à direita representa o fluxo modificado pelo SLE para uma dada execução. A parte em cinza não é executada. As instruções 6 e 16, respectivamente para aquisição e liberação do lock, são omitidas, isto é, removidas do fluxo pelo sistema.

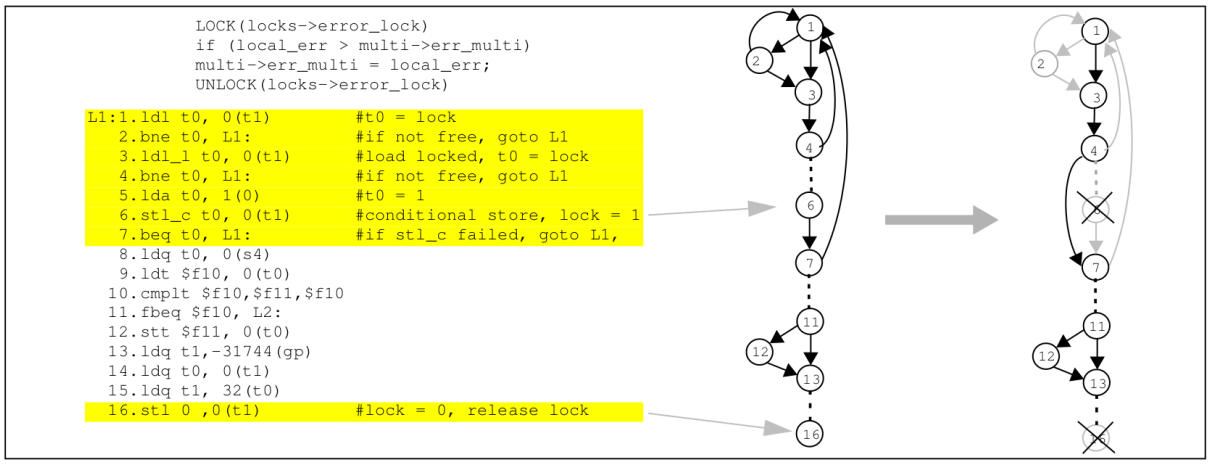

Figura 3.2: Exemplo de omissão de lock com SLE (fonte: [23])

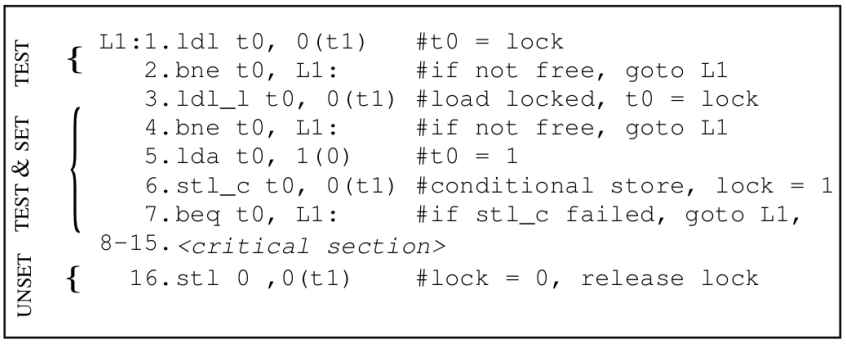

Figura 3.3: Código típico para adquirição e liberação de lock (fonte: [23])

Questões importantes relacionadas à microarquitetura com respeito às etapas apresentadas acima são: como operações de aquisição e liberação de locks são detectadas; como modificações de dados são mantidos como um estado interno durante a especulação; e como um erro de especulação é detectado.

3.3.1 Detecção de aquisição e liberação de locks

O processador apenas enxerga uma série de instruções, não sendo, portanto, informado explicitamente quando um lock está sendo adquirido. No entanto existe um padrão de operações para garantir exclusão mútua comumente adotado chamado TEST&TEST&SET, onde o valor do lock é lido e testado (TEST) e, caso esteja livre, seu valor é alterado para ocupado atomicamente (TEST&SET). No final da região crítica o lock recebe o valor anterior, a saber, livre.

A Figura 3.3 mostra um exemplo de um trecho típico em *assembly* de uma região crítica protegida por um lock. As instruções 1-7 adquirem o lock, sendo que 1-2 equivalem ao primeiro TEST e 3-7 a TEST&SET. As instruções 8-15 referem-se à região crítica e a instrução 16 é onde o lock é liberado.

Dessa forma, não é necessário acesso à informação semântica de alto nível do programa sendo executado para especular que um lock está sendo adquirido. Caso o padrão da

sequência de instruções característica à aquisição de lock seja encontrado, o sistema prediz que o lock em questão estaria sendo adquirido e que, *em um número curto de instruções, o valor da variável voltaria ao de antes do início da especulação*, o que significa a liberação do lock.

3.3.2 Modificações mantidas em estado especulativo

Durante a especulação, alterações de registradores e de memória são armazenados em *buffers*.

No caso de registradores, duas opções são citadas: o uso do *Reorder Buffer* (ROB) ou de *Register Checkpoint* [14]. A última permite regiões críticas maiores em relação à primeira.

Para o caso da memória, propõe-se a modificação dos *buffers* de escrita à cache de forma que eles permitam armazenar escritas especulativas. Caso a seção crítica especulativa termine com sucesso, as alterações especulativas são efetivadas na cache de nível mais próximo ao processador. Caso haja erro de especulação, tais alterações são invalidadas no *buffer*.

3.3.3 Erro de especulação

Um erro de especulação significa que atomicidade não pode ser garantida pelo hardware. Pode acontecer em dois cenários antes que a operação de liberação do lock seja detectada:

Conflitos de dados A detecção de conflitos de dados em memória com outros processadores já é disponibilizada por implementações de protocolos de coerência de cache em processadores modernos [14].

Para o caso da escolha da abordagem usando *Register Checkpoint* para alterações especulativas de registradores, dados podem sair da ROB para serem escritos na cache. Nesse caso, é necessária a modificação da cache de maneira a permitir informar que blocos estão sendo acessados especulativamente, de forma que acessos externos suficientes para invalidar a atomicidade das operações provoquem erro de especulação;

Falta de recursos A execução especulativa da região crítica pode também ser invalidada devido a limitação de recursos, que pode acontecer de duas formas: (i) limitações no tamanho da ROB, da memória cache ou do *buffer* de escrita; ou (ii) eventos que não podem ser revertidos (e.g., chamadas de sistema).

Quando uma *thread* detecta erro de especulação e não há mais novas tentativas restantes, o lock é adquirido. A aquisição do lock provoca uma escrita na variável correspondente, o que invalida outras regiões críticas especulativas para o mesmo lock, fazendo

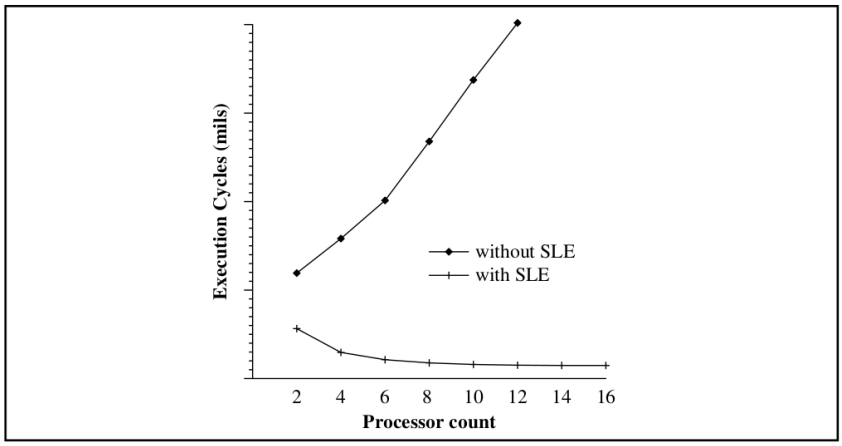

Figura 3.4: SLE Microbenchmark: tempo de execução com diferentes números de *threads* (fonte: [23])

com que elas tentem adquiri-lo. Dessa forma, a sincronização é estabelecida por causa ou de conflitos de dados (o que não se refere ao lock) ou de falta de recursos.

3.3.4 Resultados de SLE

Testes de *benchmark* foram executados em simuladores de diferentes configurações de sistemas de múltiplos processadores. Os testes são compostos por um *microbenchmark* e seis aplicações com padrões de uso de locks, acesso à memória e comportamento das regiões críticas variados.

O *microbenchmark* utilizado cria várias *threads*, onde cada uma incrementa um contador próprio, mas que compartilham um único lock para os contadores. Trata-se de um programa propositalmente escrito para demonstrar o potencial do modelo e, por isso, com nenhuma possibilidade de conflito de dados. Os testes demonstraram que, enquanto a desempenho do programa degradava conforme o número de *threads* era aumentado para uma execução sem SLE, a execução com SLE alcançou escalabilidade perfeita. A Figura 3.4 contém o gráfico com o tempo de execução em ciclos para variados números de processadores para uma execução sem e outra com SLE.

Os *benchmarks* com as seis aplicações constataram altas porcentagens de omissão de locks com sucesso para a maior parte delas e das configurações, sendo quase 100% em alguns casos. Os testes foram executados com o máximo de uma nova tentativa de omissão de locks em caso de erro de especulação - testes com tal parâmetro como zero resultaram entre 10% a 30% *a menos* de omissões com sucesso.

Os testes com as aplicações também apresentaram ganho de uma média aproximada de 10% em desempenho em relação à execução sem SLE. Para uma aplicação em específico, que continha alta frequência de sincronização, o ganho de desempenho chegou em até 46%, o que mostrou que SLE é bem efetivo para aplicações do tipo.

De acordo com os autores, as razões principais para o ganho em desempenho nos

benchmarks foram:

Execução concorrente de regiões críticas : Isso já é proposto naturalmente pelo modelo;

Reduzidas latências por acesso à memória : Geralmente a aquisição do lock causa um *miss* na cache e requer uma escrita ao sistema de memória. Uma vez que, com SLE, o lock é lido, mas as instruções de escrita no mesmo são omitidas, a variável tem condições favoráveis para permanecer na cache;

Tráfego de memória reduzido : Um lock adquirido e mantido na cache do processador em estado exclusivo gera tráfico de mensagens de barramento quando outros processadores requerem o lock. As omissões das aquisições e liberações de locks contribuem para a redução do tráfego.

3.4 *Lemming effect*

Um dos grandes problemas relacionados a técnicas de omissão de lock é o chamado *lemming effect* [17], que ocorre da seguinte maneira. Quando uma região crítica sendo executada especulativamente desiste de especular e decide adquirir o lock, as outras regiões críticas especulativas que compartilham o mesmo lock acabam detectando erro de especulação, uma vez que aquela primeira faz uma escrita no lock. Dessa forma, aquelas regiões críticas acabam por também tentar adquirir o lock, o que causará serialização. Além disso, novas execuções de regiões críticas podem aparecer *enquanto* a serialização está ocorrendo. Uma vez que o lock já estará tomado, essas novas execuções acabarão por também entrar no conjunto de sincronização. Isso ocorrerá enquanto o conjunto de sincronização formado não se esvaziar. Apenas quando todas aquelas regiões críticas terminarem suas execuções (uma após a outra) é que será possível execução especulativa novamente para o dado lock.

O ponto chave a se observar no problema do *lemming effect* é que muitas regiões críticas que não causariam conflitos de dados são serializadas durante o fenômeno. Uma maneira de ilustrar isso é um programa que realiza leituras e escritas em uma lista, onde os acessos possuem um alto grau de concorrência e dos quais pouca parte é escrita e grande parte é de leitura. Um programador que implementa tal programa utilizando granularidade grossa de locks (por considerar os benefícios do modelo de omissão de locks) pode observar ganhos de desempenho não tão altos como esperado: as poucas operações de escrita ocasionarão o enfileiramento de acessos de leituras de índices não relacionados ao conflito que causou a primeira aquisição do lock.

Uma vez que SLE é implementado no nível microarquitetural, de forma transparente ao programador, aplicações utilizando tal mecanismo acabam se tornando "reféns" desse

tipo de problema, podendo no máximo desativar o mecanismo [1] quando muitas falhas de especulação são detectadas.

Neste cenário, diversas abordagens de TLE foram propostas para solucionar ou amenizar o *lemming effect*, muitas delas foram citadas na Seção 3.2.1. Como mencionado anteriormente, elas executam regiões críticas dentro de transações sem adquirir o lock e aplicam diferentes estratégias para evitar que erros de especulação bloqueiem futuras transações e aquelas já em execução. Tais estratégias são geralmente aplicadas quando transações abortam, momento em que o sistema de memória transacional retorna o controle de execução à função responsável por adquirir o lock. As próximas seções apresentam algumas dessas abordagens.

3.5 *Software assisted lock removal* (SLR)

Como visto anteriormente, a serialização induzida por SLE se dá justamente pelo fato de que as regiões críticas sendo executadas especulativamente, apesar de não escreverem no lock, ainda fazem operação de leitura no mesmo ao início da transação. SLR [2] é uma solução que executa as regiões críticas especulativamente como transações em hardware (HTM – *Hardware Transactional Memory*) e que, ao invés de ler o lock no início da região crítica, o faz no fim da mesma antes de efetivar a transação. Caso o lock lido já esteja tomado, a transação é abortada. Após um número limite de tentativas, o lock é adquirido. A Figura 3.5 mostra o algoritmo originalmente proposto para as funções para adquirir o lock (lock) e para liberá-lo (unlock).

A ideia por trás da abordagem SLR é que há transações que são abortadas apenas pelo fato de a leitura do lock estar no início, de forma que não acontecem de fato conflitos de dados com outras regiões em execução. Fazendo a leitura do lock apenas no momento de efetivar a transação garante que a mesma somente seja invalidada caso haja o risco de inconsistência devido a regiões críticas não especulativas (i.e., com lock) *ainda* em execução.

Uma ideia tentadora para esse tipo de implementação seria fazer com que transações não lessem o lock de forma alguma e adquirissem o lock somente quando abortassem, porém não é realmente uma boa ideia. A razão pela qual não se deve simplesmente deixar de ler o lock em transações é que há potencial possibilidade de inconsistência. Considere duas regiões críticas, C_1 e C_2. C_1 está executando especulativamente e não lê o lock durante toda a transação. Já C_2 está executando de forma não especulativa, o que significa que C_2 adquiriu o lock. A situação de inconsistência pode acontecer quando C_1 termina a transação e efetiva enquanto C_2 ainda não terminou a execução não especulativa. Uma vez que C_1 chegou ao fim da transação, não houve conflitos de dados, e por isso a

[1] Por exemplo, com a interface HLE da extensão Intel® TSX, isso pode ser feito através de um caminho de execução alternativo, onde os prefixos XACQUIRE e XRELEASE não são utilizados

```
1  shared variables:
2    lock : speculative lock
3
4  thread local variables:
5    retries : int
6
7  lock() {
8    retries := 0
9    // speculative path
10   XBEGIN (Line 14) // jump to Line 14 on abort
11   return
12
13   // fallback path
14   retries ++
15   if ( retries < MAX_RETRIES)
16     goto Line 10
17   else
18     lock.lock() // standard lock acquire
19 }
20
21 unlock() {
22   if (XTEST()) { // returns TRUE if the run is speculative
23     if (lock is locked)
24       XABORT // aborts the speculative run
25     else
26       XEND
27   } else {
28     lock.unlock() // standard lock release
29   }
30 }
```

Figura 3.5: SLR: algoritmo para funções lock e unlock (fonte: [2])

C_1	C_2

lock(L)
load(X)
load(Y)
unlock(L)
	lock(L)
	store(X, 1)
	# C1 termina
	store(Y, 1)
	unlock(L)

Figura 3.6: O programa desse exemplo apresentaria inconsistência se não houvesse leitura do lock na transação. C_1 executa especulativamente e C_2 não.

transação é efetivada. No entanto, uma vez que C_2 ainda não terminou, é possível que os acessos subsequentes à memória por C_2 causem conflito, que não será detectado, com os acessos realizados por C_1.

A Figura 3.6 apresenta um exemplo de programa para a situação anteriormente descrita. X e Y são ambos zero antes do início de C_1 e C_2, que executam, respectivamente, especulativamente e de forma não especulativa. No caso, C_1 termina antes da instrução store(Y, 1) de C_2. Assim, ao fim de C_1, tem-se $X = 1$ e $Y = 0$, o que define uma situação de inconsistência.

Experimentos foram feitos usando um processador Intel® Haswell com extensões de sincronização transacional (TSX) [16]. Experimentos com programas de *benchmark* do conjunto STAMP [21] mostraram que o uso de SLR, em alguns casos, proveu *speedup* médio próximo de 2x em relação a HLE (implementação da Intel para omissão de locks em hardware). Em outros casos os ganhos de desempenho foram insignificantes.

3.6 RW-TLE

RW-TLE [8][2] propõe prover o comportamento de omissão de locks utilizando memória transacional controlada por software, porém com melhorias em relação ao modelo original. Nesse contexto, uma região crítica sendo executada especulativamente significa ela sendo executada dentro de uma transação em hardware. A ideia é tentar deixar apenas uma *thread* utilizando o lock enquanto outras executam especulativamente.

O sistema cria dois possíveis caminhos de fluxo para uma dada região crítica. Um caminho chamado "rápido", onde não há modificação da região crítica, e outro chamado "lento", onde as operações de escritas são instrumentadas para que as transações em hardware sejam impossibilitadas de realizar escritas quando executadas no caminho lento. As execuções especulativas podem ser feitas tanto no caminho rápido quanto lento. Já as execuções com lock adquirido sempre utilizam o caminho lento.

Quando uma *thread* vai executar uma região crítica, ela verifica se o lock está adquirido.

[2]RW é uma sigla para *Reader/Writer*

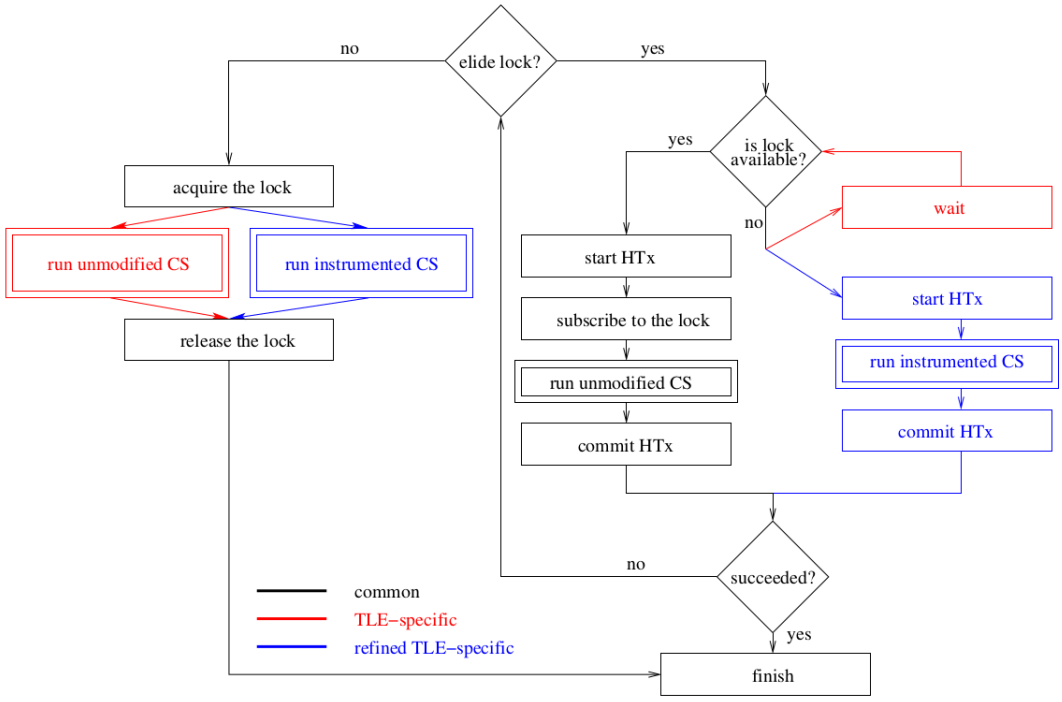

Figura 3.7: RW-TLE: esquema de execução de uma região crítica (fonte: [8])

Em caso negativo, a região crítica é executada especulativamente no caminho rápido – inicia-se a transação, o lock é lido, a região crítica sem instrumentação é executada, e, por fim, a transação é efetivada quando há sucesso. Já em caso afirmativo, diferentemente do modelo original, onde a *thread* esperaria até que ela encontrasse o lock disponível, a região crítica é executada especulativamente e concorrentemente com a execução não especulativa, porém no caminho lento e sem haver leitura do lock.

Em ambos casos, se a transação é abortada, a *thread* pode ou tentar novamente a etapa descrita no parágrafo anterior, ou tentar realmente adquirir o lock, onde ocorre serialização caso o lock já esteja tomado. Como dito anteriormente, a execução é feita no caminho lento nesse caso, isto é, a região crítica é executada com instrumentação.

A Figura 3.7 mostra o fluxograma para a execução de uma região crítica dentro do esquema proposto. Os itens em vermelho referem-se ao que foi removido do modelo original, enquanto os itens em azul ao que é adicionado com a nova proposta.

O objetivo da instrumentação das operações de escrita é, durante uma execução não especulativa, permitir concorrência entre regiões críticas apenas enquanto somente operações de leitura são realizadas pelas transações. Em outras palavras:

1. Se uma transação instrumentada tenta escrever, ela é abortada;

2. Se a *thread* com o lock realiza escrita, todas as transações são abortadas. Note que a *thread* com o lock ainda consegue realizar a escrita com sucesso.

O caso (1) é simples de resolver: na função (i.e., na instrumentação) que realiza escrita,

basta verificar se a *thread* está executando uma transação e abortar em caso afirmativo.

O caso (2) pode ser realizado através do uso de uma *flag* compartilhada que diz que uma *thread* com lock realizou uma escrita. Tal *flag* tem valor inicial como falso, é definida como verdadeiro quando a *thread* com lock realiza escrita e é redefinida como falso quando o lock é liberado. Assim, basta que a *thread* executando a transação no caminho lento leia tal *flag* no início e aborte caso esta tenha valor verdadeiro. Isso também cobre o caso de quando a *thread* com o lock realiza escrita mesmo após a *flag* ser verificada pela transação, pois, a *flag* ter mudado seu valor para verdadeiro implica uma escrita na mesma, o que naturalmente faz as transações no caminho lento abortarem por terem lido a *flag* anteriormente.

Embora tal abordagem pareça muito restritiva, ela faz diversos programas terem melhor desempenho em relação à tradicional omissão de locks por dois motivos:

1. Existem diversos programas que possuem regiões críticas que só realizam leituras (dinamicamente ou estaticamente);

2. Devido ao mecanismo de *prefetching*, regiões críticas no caminho lento que realizam escrita e são, consequentemente, abortadas poderão já ter parte ou todos os dados necessários na cache na próxima vez que forem executadas.

3.7 FG-TLE

FG-TLE [8][3] é uma proposta que se diferencia de RW-TLE em como as regiões críticas são instrumentadas. Na verdade, ambos são implementações de uma mesma classe de esquema de omissão de locks chamada pelos autores de omissão transacional de locks refinada (*Refined Transactional Lock Elision*). No caso de FG-TLE, operações de leitura e escritas são instrumentadas no caminho lento de forma a permitir que transações possam ler e escrever na memória e efetivar (dado que não haja conflitos com outras regiões críticas) mesmo quando há uma região em execução não especulativa, assumindo, portanto, um caráter menos restritivo que RW-TLE.

FG-TLE utiliza o conceito de registros de posse (*orecs*) de memória transacional com certas modificações [7]. De forma resumida, em memória transacional, um registro de posse é utilizado para dizer que um certo endereço está em posse de uma determinada transação ou para escrita ou para leitura. Assim, esses registros são utilizados para verificação de conflitos de dados entre transações.

Em memória transacional os registros de posse são lidos e atualizados por todas as transações envolvidas. No caso de FG-TLE, uma região crítica sendo executada com o lock é garantida de vencer conflitos por definição. Portanto, não é necessário que regiões

[3]FG é uma sigla para *Fine Grained*

críticas não especulativas verifiquem por conflitos quando acessando memória. Além disso, regiões críticas especulativas não precisam explicitamente verificar por conflitos com outras transações, pois tal verificação já é naturalmente realizada pelo mecanismo transacional do hardware. Assim, tais transações não necessitam escrever nos registros de posse, apenas ler, de modo a verificar por conflitos com a *thread* não especulativa. Portanto, a diferença no uso de registros de posse entre essa abordagem e a de memória transacional é que, em FG-TLE, apenas a *thread* com o lock escreve nos registros de posse e apenas as transações no caminho lento realizam leitura nos mesmos.

O funcionamento de FG-TLE é descrito a seguir. Uma região crítica com lock sempre registra as operações de leitura e escrita em endereços de memória na lista de registros de posse. A escolha do índice do registro de posse é feita através de uma função de *hash* no endereço de memória.

As transações que estão executando no caminho lento, para cada operação de escrita ou leitura, sempre verificam, através dos registros de posse, por conflito de dados com a execução não especulativa. Isto é, verificam se o endereço sendo acessado já foi registrado para escrita ou se o endereço já foi registrado para leitura e o acesso a ser realizado é de escrita. Em caso de conflito, a transação aborta explicitamente.

Por fim, quando a região crítica não especulativa termina, todos os registros de posse são marcados como livre, e, então, o lock é liberado.

Esta e as abordagens discutidas anteriormente são alguns exemplos recentes de como memória transacional pode ser utilizada para implementar modelos de omissão de lock mais flexíveis. O capítulo seguinte descreve uma outra abordagem, chamada SCM, e discute como FGSCM, esquema proposto por este trabalho, a estende de maneira a aumentar o nível de concorrência explorado.

Capítulo 4

Fine-grained SCM

Este capítulo se inicia descrevendo a abordagem original SCM na Seção 4.1. Em seguida, sua extensão, FGSCM, proposta neste trabalho, é apresentada na Seção 4.2 e a metodologia de avaliação utilizada bem como resultados experimentais são discutidos na Seção 4.3.

4.1 *Software-assisted conflict management (SCM)*

O esquema SCM [2] propõe uma maneira de serializar as regiões críticas que estão envolvidas em conflitos sem adquirir o lock originalmente utilizado, de forma que a serialização não afete regiões críticas não conflitantes. Isso é feito através de um lock auxiliar.

Em resumo, o algoritmo encapsula uma técnica de omissão de lock em uma transação e age quando tal técnica falha adquirindo o lock auxiliar e tentando a execução especulativa novamente. A execução especulativa com o lock auxiliar tomado é reiniciada até que a especulação complete com sucesso ou um limite de tentativas é alcançado. No último caso, como uma última medida, o lock original é adquirido de forma não especulativa, o que garante progresso do sistema.

Idealmente, o uso do lock auxiliar faz com que apenas as transações conflitantes sejam serializadas, uma vez que as transações não conflitantes enxergarão o lock original como livre. Por isso, a etapa de aquisição (e também liberação) do lock auxiliar é chamada *caminho de serialização*.

Após o lock auxiliar ser adquirido, a região crítica associada volta a ser executada, mas *ainda de forma especulativa e transacional*, pois potencialmente haverá regiões críticas especulativas associadas ao mesmo lock original executando de forma concorrente. Nesse ponto, é possível haver conflito de dados com outra transação (y), e então, dado que x refere-se à transação em posse do lock auxiliar, uma das duas situações pode ocorrer: (i) y aborta, tentando também adquirir o lock auxiliar, tendo, portanto, que esperar por "sua vez" de executar; ou (ii) x aborta e y é efetivada.

```
 1: procedure SCM-LOCK(lock)
 2:     aux_tries ← 0
 3:
 4:     Start transaction, go to line 9 if aborted
 5:     Acquire lock speculatively
 6:     return
 7:
 8:     if thread does not own aux_lock then
 9:         Acquire aux_lock
10:     end if
11:     aux_tries ← aux_tries + 1
12:     if aux_tries > max_aux_tries then
13:         Acquire lock non-speculatively
14:     else
15:         Go to line 5
16:     end if
17: end procedure
18: procedure SCM-UNLOCK(lock)
19:     if thread is in active transaction then
20:         Release lock speculatively
21:         Finish transaction
22:     end if
23:
24:     if thread does not own aux_lock then
25:         return
26:     end if
27:
28:     if thread owns lock then
29:         Release lock non-speculatively
30:     end if
31:     Release aux_lock
32: end procedure
```

Figura 4.1: SCM: algoritmo para as funções de lock e unlock.

Dado que o número de aparições de execuções especulativas de novas regiões críticas é indefinido do ponto de vista do algoritmo, o número de vezes que uma transação pode abortar enquanto com o lock auxiliar adquirido também o é. Para evitar que tal transação fique constantemente abortando com o lock auxiliar adquirido, fazendo com que as outras regiões críticas fiquem em constante espera no caminho de serialização, se define um número máximo de tentativas de execução especulativa para uma transação no caminho de serialização. Quando esse limite é alcançado, finalmente o lock original é adquirido, de forma que a região crítica é executada com exclusividade e de forma não especulativa, tendo, portanto, garantia de sucesso.

Finalmente, quando uma transação termina sua execução os locks são liberados na ordem correta, isto é, primeiro o lock original caso o mesmo tenha sido adquirido, e então o lock auxiliar. A Figura 4.1 mostra o algoritmo do esquema SCM para as funções lock e unlock.

A Figura 4.2 ilustra um caso de execução das regiões críticas no esquema SCM. Cada linha representa o fluxo de controle de uma região crítica. As caixas com as operações de lock e unlock prefixadas por "*optimistic*" e "*standard*" representam, respectivamente, as operações com omissão de locks e aquelas onde o lock auxiliar é adquirido. Cada símbolo em formato de "x" representa o momento em que uma transação aborta.

Vale notar que a execução especulativa das regiões críticas, isto é, omissão da aquisição de lock, pode ter diferentes implementações desde que seja possível que SCM saiba quando a execução especulativa "aborta definitivamente", ou seja, o momento em que o lock *seria* de fato tomado. Nesse momento, SCM passa a ter o controle da execução e o lock original não é tomado como seria no modelo tradicional de omissão de lock.

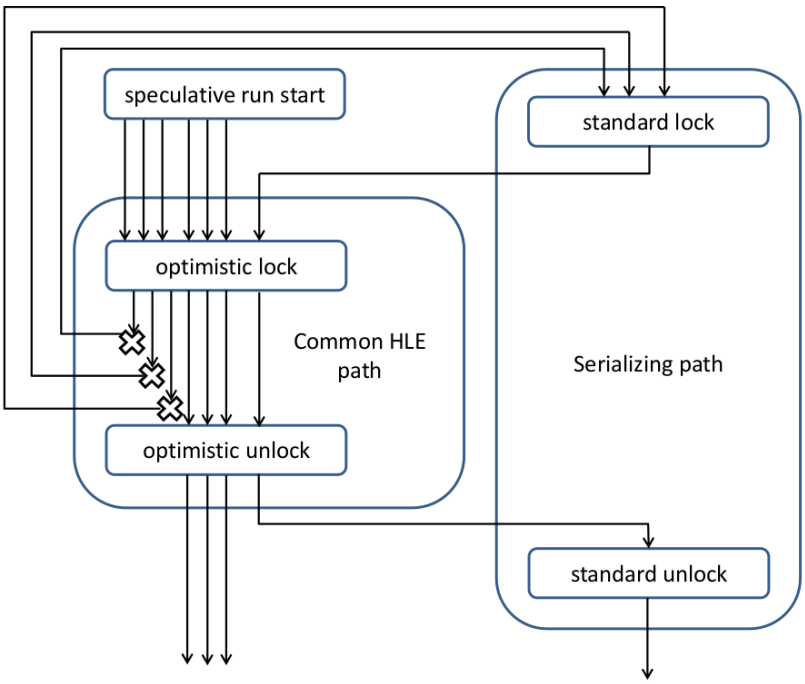

Figura 4.2: SCM: fluxo de controle das regiões críticas (fonte: [2])

4.2 *Fine-grained software-assisted conflict management (FGSCM)*

O esquema FGSCM, proposto neste trabalho, é uma extensão do modelo SCM, onde mais de um lock auxiliar é utilizado no caminho de serialização. A motivação por trás dessa abordagem pode ser ilustrada a seguir. Considere um grupo de transações onde, em um determinado pequeno período de tempo p, seis transações, x_0 até x_5, abortem devido a conflito de dados; e que o tempo de execução de cada uma seja maior que p. Considere também que x_0 e x_1 conflitem entre si, mas não com as outras transações em discussão, e que o mesmo ocorre com os pares (x_2, x_3) e (x_4, x_5). Em tais circunstâncias, SCM forçaria todas essas transações a serem sincronizadas mesmo diante do fato de que algumas transações poderiam potencialmente ser executadas em paralelo sem causar conflitos (por exemplo, x_0 com x_3 e x_1 com x_4). Neste cenário, uma abordagem de granularidade fina pode ser aplicada permitindo o uso de mais de um lock auxiliar no caminho de serialização – um para cada par (x_{2i}, x_{2i+1}), para $i \in 0, 1, 2$ – e serializando cada transação usando o lock auxiliar associado.

Antes de se formalizar esta abordagem, vale definir o seguinte:

Transações prontas para serialização: no algoritmo do SCM, uma transação é serializada quando a técnica de omissão de lock encapsulada falha (ou seja, quando ela decide adquirir o lock de forma não especulativa). Neste contexto, tal transação é dita estar *pronta para serialização*. O algoritmo FGSCM (apresentado mais adiante) permite que a especulação sem uso do lock auxiliar seja tentada um número

```
1: procedure FGSCM-LOCK(lock)
2:     retries ← 0
3:     aux_tries ← 0
4:
5:     Start transaction, go to line 9 if aborted
6:     Acquire lock speculatively
7:     return
8:
9:     if retries < max_retries then
10:        retries = retries + 1
11:        Go to line 5
12:    end if
13:
14:    aux_lock ← get_aux_lock()
15:    if thread does not own aux_lock then
16:        Acquire aux_lock
17:    end if
18:    aux_tries ← aux_tries + 1
19:    if aux_tries > max_aux_tries then
20:        Acquire lock non-speculatively
21:    else
22:        Go to line 5
23:    end if
24: end procedure
25: procedure FGSCM-UNLOCK(lock)
26:    if thread is in active transaction then
27:        Release lock speculatively
28:        Finish transaction
29:    end if
30:
31:    aux_lock ← get_aux_lock()
32:    if thread does not own aux_lock then
33:        return
34:    end if
35:
36:    if thread owns lock then
37:        Release lock non-speculatively
38:    end if
39:    Release aux_lock
40: end procedure
```

Figura 4.3: Algoritmo para o esquema FGSCM

de vezes antes que a transação se torne pronta para serialização[1];

Fonte de conflito: dado um evento de conflito entre as transações x e y, onde x é abortada, a *fonte de conflito* de x é definida como o objeto que representa o recurso que causou o conflito.

O esquema pode ser definido a seguir. Seja C, em um certo ponto no tempo, um conjunto não vazio de transações prontas para serialização *abortadas devido a conflito de dados*. Para cada transação x em C, seja $\gamma(x)$ a fonte de conflito de x para o último evento de conflito antes de x se tornar pronta para serialização (a outra transação conflitante com x não pertence necessariamente a C).

C pode ser particionado em uma família de conjuntos $G(C) = G_{c_0}, ..., G_{c_{n-1}}$, onde $c_0, ..., c_{n-1}$ são fontes de conflitos diferentes pertencentes a $\{\gamma(x) : x \in C\}$. Cada conjunto G_c dessa partição é chamado de *grupo de conflito* e definido como $G_c = \{x \in C : \gamma(x) = c\}$. Em outras palavras, um grupo de conflito agrupa transações que foram abortadas devido a conflito de dados com a mesma fonte de conflito justo antes de se tornarem prontas para serialização.

Os pressupostos que guiam a abordagem de granularidade fina são que (i) existe a possibilidade de que duas transações de um mesmo grupo de conflito entrem em conflito

[1]Em [2], algumas variantes da implementação do esquema também apresentam tal característica, porém a mesma é apresentada como um aspecto pertinente à técnica de omissão de lock encapsulada e não ao algoritmo SCM.

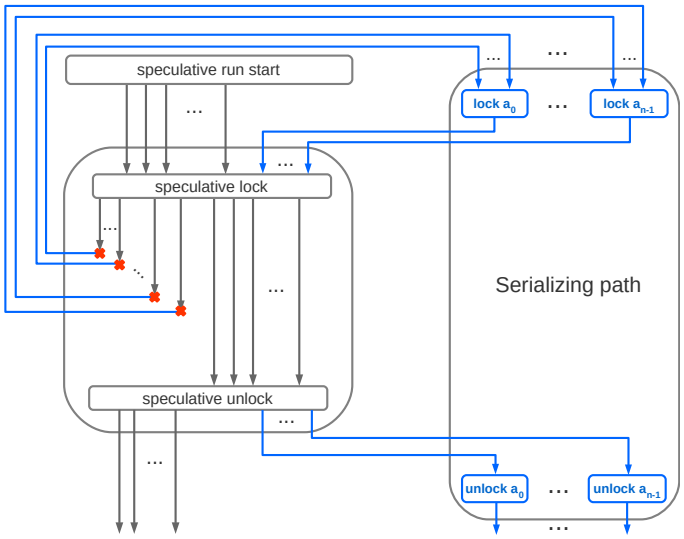

Figura 4.4: Um diagrama de bloco ilustrando o mecanismo do esquema FGSCM. Os destaques em azul expressão a extensão realizada com respeito ao SCM.

se executadas concorrentemente, uma vez que elas foram abortadas devido a uma mesma fonte de conflito no passado; enquanto que, (ii) para duas transações de grupos de conflito diferentes, a mesma possibilidade é indeterminada dada a informação provida por $G(C)$.

Dessa maneira, o lock FGSCM é pessimista com respeito a (i), mas otimista em relação a (ii). Em outras palavras, o algoritmo procura serializar transações que pertencem ao mesmo grupo de conflito, mas permite que transações pertencentes a grupos de conflitos diferentes executem concorrentemente. Formalmente, as transações $x_0, ..., x_{k-1}$ de C são permitidas a executar concorrentemente se, e somente se, $\forall x_i \forall x_j (i \neq j \rightarrow \gamma(x_i) \neq \gamma(x_j))$. Isso pode ser realizado através do uso de *um lock auxiliar diferente para cada grupo de conflito* em vez de usar apenas um, como é feito no algoritmo original.

A Figura 4.3 mostra o algoritmo para o novo esquema com as extensões ao original em destaque, sendo a parte em azul onde o aspecto de granularidade fina é introduzido, e a parte em verde, a adição de um novo parâmetro ao algoritmo ($max\_retries$). Em vez de usar um único lock auxiliar, o algoritmo utiliza a função $get\_aux\_lock()$, a qual retorna o lock auxiliar apropriado de acordo com a fonte de conflito que fez a transação abortar. As variáveis $max\_retries$ e max_aux_tries são parâmetros que definem quantas vezes FGSCM tentará a especulação antes e depois de entrar no caminho de serialização, respectivamente. A Seção 4.3.3.2 descreve os melhores valores encontrados para esses parâmetros de acordo com os resultados dos experimentos.

Uma ilustração do mecanismo proposto pode ser vista na Figura 4.4, que foi baseada na ilustração originalmente utilizada por [2]. Os elementos em azul representam o que mudou em relação ao algoritmo original – no bloco representando o caminho de serialização, n locks auxiliares são utilizados em vez de apenas um, o que pode potencialmente prover um nível maior de concorrência.

As rotinas *FGSCM-lock()* e *FGSCM-unlock()* podem ser implementadas como parte

de uma biblioteca que substitua ou estenda bibliotecas de *locking* existentes, de forma que aplicações que já utilizam locks como mecanismos de sincronização não precisem ter o código fonte modificado para usar o novo esquema.

A implementação da função *get_aux_lock*() depende diretamente de como a fonte de conflito é representada e como ela pode ser identificada no sistema computacional utilizado. Idealmente, o valor da fonte de conflito seria passado pelo processador no momento em que a transação abortasse e a função *get_aux_lock*() mapearia esse valor para o lock auxiliar a ser utilizado.

Por exemplo, em processadores Intel® com suporte a memória transacional, onde, portanto, conflito de dados é detectado através de seu sistema de coerência de cache, a fonte de conflito de uma transação pode ser representada pelo índice do *cache set* de onde se originou o conflito. Em termos de implementação, a função *get_aux_lock*() utilizaria este índice para retornar o lock auxiliar respectivo, obtido de um vetor contendo um lock para cada *cache set* possível.

Os processadores que suportam TLE ainda não proveem informação necessária para identificar a fonte de conflito de uma transação abortada, portanto a implementação de *get_aux_lock*() da maneira como é idealizada aqui ainda não é possível. Assim, uma abordagem possível seria utilizar suporte de compilador para instrumentar regiões críticas para que se tenha uma estimativa dos endereços de memórias acessados pela mesma e utilizá-los como fonte de informação para a escolha do lock auxiliar. Um fator complicador nessa estratégia é que qualquer alteração realizada no estado do programa durante a execução da transação é perdida quando a mesma é abortada, o que implica que os endereços a serem acessados pela transação precisam ser "descobertos" antes dela ser iniciada e isso pode introduzir *overhead* ao programa. Nesse sentido, o processador Power8, da IBM, possibilita suspender e resumir transações para permitir modificações não transacionais [19], o que pode ser utilizado como alternativa ao pré-processamento da transação.

Com o objetivo de verificar o potencial do esquema *FGSCM*, decidiu-se por avaliá-lo através de um *microbenchmark* (discutido nas seções seguintes) e deixar a abordagem discutida acima para ser explorada em trabalhos futuros.

4.3 Avaliação da nova abordagem

FGSCM foi avaliado utilizando um *microbenchmark* com diversos parâmetros. Este *microbenchmark*, cuidadosamente implementado, foi preciso devido à falta de suporte necessário para implementar a função *get_aux_lock*() (como idealizada na seção anterior) em processadores atuais que permitem a aplicação de omissão de lock transacional. Por isso, decidiu-se manter o foco em analisar detalhadamente o *potencial* dessa nova abordagem utilizando um *microbenchmark*, embora a abordagem ideal seria testar FGSCM em aplicações reais, o que pode ser parte de um trabalho futuro.

4.3.1 O *microbenchmark*

O *microbenchmark*, referido simplesmente por *benchmark* deste ponto adiante, emula um programa de computador que realiza operações concorrentes de escrita e leitura em uma região de memória compartilhada, protegida logicamente por um único lock. O programa realiza os seguintes passos:

1. *Alocação de memória.* Esta etapa foi definida cuidadosamente de tal forma que *false sharing* não interferisse nas operações de escrita e leitura realizadas durante a *execução concorrente*. A memória é alocada de maneira que dados de estatística, locks auxiliares e os dados utilizados para as operações de escrita e leitura mapeiem conjuntos diferentes da cache;

2. *Inicialização dos dados de operação.* Nesta etapa, os dados para a *execução concorrente* são inicializados. Cada operação é definida por: o índice do *byte* que será acessado na região de memória compartilhada; o tipo de operação, o que diz se a operação é de escrita ou leitura; e um *byte* contendo o valor usado para operações de escrita. Os valores do índice e do *byte* são selecionados aleatoriamente e o tipo de operação é selecionado de acordo com a *taxa de escrita* (parâmetro descrito mais adiante);

3. *Execução concorrente.* Uma quantidade de *threads* é criada e executada. Cada *thread* itera sobre sua porção dos dados de operação, que são divididos igualmente de acordo com o número de *threads*. Para cada operação, a *thread* (i) adquire o lock, (ii) executa a operação e (iii) libera o lock.

Para a análise, três modos de execução são considerados com respeito a sincronização: (i)*Normal*, onde um *spin lock* padrão é utilizado; (ii) *SCM*, a abordagem inicial; e (iii) *FGSCM*, o modelo proposto neste trabalho.

Para os últimos dois modos, o algoritmo da Figura 4.3 foi implementado de forma que o valor de *get_aux_lock*() é passado explicitamente como argumento para as rotinas de aquisição e liberação do lock. Para o modo SCM, um lock auxiliar global é utilizado enquanto que, para o modo FGSCM, os locks auxiliares são informados explicitamente de acordo com região de memória sendo acessada.

Diversos parâmetros foram definidos para que o *benchmark* pudesse simular diferentes classes de programas. Eles seguem abaixo:

- *Número de threads*: para a execução concorrente;

- *Número de operações*: número de escritas ou leituras na memória compartilhada, que é repartido igualmente entre as threads;

- *Tamanho compartilhado* (s): número de *bytes* a serem alocados para a região de memória acessada pelas operações de leitura e escrita. O espaço para os dados de execução é alocado como um bloco contíguo de s *bytes* e alinhado com o tamanho da linha de cache do processador sendo utilizado;

- *Taxa de escrita*: um número de 0 a 1 representado como um valor de ponto flutuante. Este parâmetro representa a probabilidade de uma operação ser de escrita. A probabilidade de uma operação ser de leitura é o complemento da primeira;

- *Modo de lock*: que tipo de lock utilizar (*Normal*, *SCM* ou *FGSCM*);

- *Tamanho de overhead* (x): este parâmetro permite que o tamanho da região crítica seja ajustada através do valor de x: o número de divisões inteiras (para gastar ciclos do processador) realizadas antes e depois que a região de memória compartilhada é acessada;

- *Tamanho de overhead externo*: similar ao *tamanho de overhead*, porém com o objetivo de controlar o tempo gasto antes de entrar na região crítica em cada iteração da execução concorrente.

- *Máximo de reexecuções*: número máximo de tentativas de reexecuções da especulação sem o lock auxiliar ter sido tomado ainda. Este parâmetro se refere ao valor de $max\_retries$ na Figura 4.3 e se aplica apenas para os modos SCM e FGSCM;

- *Máximo de tentativas auxiliares*: número máximo de tentativas de executar a transação depois de se entrar no caminho de serialização, isto é, após o lock auxiliar ser adquirido. Este parâmetro se refere ao valor de max_aux_tries na Figura 4.3 e se aplica apenas para os modos SCM e FGSCM.

4.3.2 Detalhes e considerações sobre a implementação

O *benchmark* foi implementado para ser executado em um processador Intel® Haswell quadcore com suporte a 2 threads simultâneas por núcleo através de sua tecnologia de *hyper-threading*. Assim, o algoritmo FGSCM foi implementado utilizando a API provida por *Transactional Synchronization Extensions* (TSX) [16], especificamente a interface provida por *Restricted Transactional Memory* (RTM). Uma vez que o mecanismo por trás de RTM usa o protocolo de coerência da cache do nível L1 para identificar conflito de dados em transações, a fonte de conflito foi representada pelo índice do conjunto de linhas da cache L1 mapeado pelo endereço de memória sendo acessada pela transação abortada.

A implementação do FGSCM no *benchmark* estende a variante da implementação original do SCM que utiliza um lock do tipo TEST&TEST&SET [2]. Além de substituir

[2]O código fonte foi obtido diretamente dos autores.

o caminho de serialização original por um de granularidade fina, outra modificação realizada foi que agora também existe um parâmetro que define um quantidade máxima permitida de reexecuções especulativas antes de fazer a transação passar pelo caminho de serialização, cujo nome é *máximo de reexecuções*.

Depois que uma transação se torna pronta para serialização, a implementação atual do FGSCM no *benchmark* utiliza o lock auxiliar correspondente (isto é, aquele correspondente ao endereço de memória acessado) independentemente do motivo pelo qual a transação foi abortada. Uma possível melhoria futura para o algoritmo atual seria levar em consideração a causa da falha de forma que apenas transações em conflito passem pelo caminho de serialização e outras causas sejam tratadas diferentemente.

4.3.3 Resultados

Para os experimentos cujos resultados são apresentados nesta seção, o *benchmark* foi executado 20 vezes para cada combinação de parâmetros e os tempos de execução foram registrados. As seguintes combinações de parâmetros foram utilizadas:

- *Número de threads*: 2, 4, 6 e 8 threads;
- *Número de operações*: fixado em 20000 operações, um número de operações suficiente para permitir um tempo de execução suficientemente longo para análise posterior;
- *Tamanho compartilhado*: 256 *bytes* e 512 *bytes*, isto é, 4 e 8 conjuntos de linhas de cache em uso, respectivamente;
- *Taxa de escrita*: 0,2, 0,5 e 0,8;
- *Modo de lock*: *Normal*, *SCM* e *FGSCM*;
- *Tamanhos de overhead interno e externo*: Os valores para estes parâmetros foram selecionados de maneira a se manter um "tamanho de execução" constante para diferentes instâncias do experimento. Experimentos iniciais mostraram que um *overhead* interno de 60000 divisões inteiras sempre fizeram as transações abortarem devido a falta de recursos, então um valor máximo de 50000 divisões foi estabelecido. Portanto, a seguinte combinação de valores na forma *overhead interno*:*overhead externo* foram utilizados:
  - 0:50000 (0,0:1,0)
  - 10000:40000 (0,2:0,8)
  - 20000:30000 (0,4:0,6)
  - 25000:25000 (0,5:0,5)

  - 30000:20000 (0,6:0,4)
  - 40000:10000 (0,8:0,2)
  - 50000:0 (1,0:0,0)

  Os valores em parênteses representam as respectivas "proporções de tamanho de execução". Por exemplo, a proporção (0,4:0,6) para combinação 20000:30000 significa que cerca de 40% da execução do programa acontece dentro da região crítica e 60% fora dela;

- *Máximo de reexecuções*: 2, 5 e 10;
- *Máximo de tentativas auxiliares*: 2, 5 e 10.

Dado o número de combinações de parâmetros, os experimentos somaram mais de 50000 execuções do *benchmark*, o que levou cerca de 6 dias para terminar. Os resultados mais interessantes são apresentados nas seções que se seguem. A Seção 4.3.3.1 apresenta as principais vantagens da nova abordagem em relação aos modos de lock Normal e SCM; a Seções 4.3.3.2 e 4.3.3.3 proveem uma análise da configuração de parâmetros específicos do FGSCM que proveram os melhores resultados; e, por fim, a Seção 4.3.3.4 aponta situações onde o esquema FGSCM não obteve resultados satisfatórios.

4.3.3.1 Ganho em desempenho

Como a Seção 4.3.3.2 mostra, os melhores valores para os parâmetros ($max\_retries$, max_aux_tries) encontrados para os esquemas FGSCM e SCM nos experimentos foram (2, 5) e (5, 5), respectivamente. Além disso, a Seção 4.3.3.3 mostra que, na maioria dos casos onde FGSCM tem melhor desempenho que ambos modos SCM e Normal, as porções pertencentes a regiões críticas e regiões não críticas (*overheads* interno e externo) possuem tamanhos balanceados. Por último, os experimentos também revelaram um comportamento já esperado, o que é explicado com detalhes na Seção 4.3.3.4: FGSCM geralmente possui ganho de desempenho apenas quando o número de threads é maior que dois.

Levando em conta os parâmetros descritos no parágrafo anterior, a Tabela 4.1 mostra o ganho médio de desempenho do esquema FGSCM quando comparado com os outros modos de lock do *benchmark*. Pode-se observar que FGSCM provê os melhores ganhos de desempenho para aplicações com uma predominância de operações de leitura – uma média de 36% de ganho comparado ao modo Normal e 11% comparado ao SCM.

A Figura 4.5 mostra, para cada modo de lock, o *speedup* com relação à execução do *benchmark* no modo Normal utilizando apenas uma thread. Os casos cobertos pela imagem são execuções com baixa taxa de escrita (0,2) e tamanho da região crítica balanceado com a região não crítica. Pode-se notar que, quando são utilizadas 4 ou mais threads, SCM e FGSCM sempre possuem desempenho melhor que o modo Normal. Também é

Tabela 4.1: Ganho médio de desempenho do esquema FGSCM

Taxa de escrita	Comparado ao modo Normal	Comparado ao modo SCM
.2	36,44%	11,17%
.5	1,85%	7,80%
.8	-10,30%	5,09%

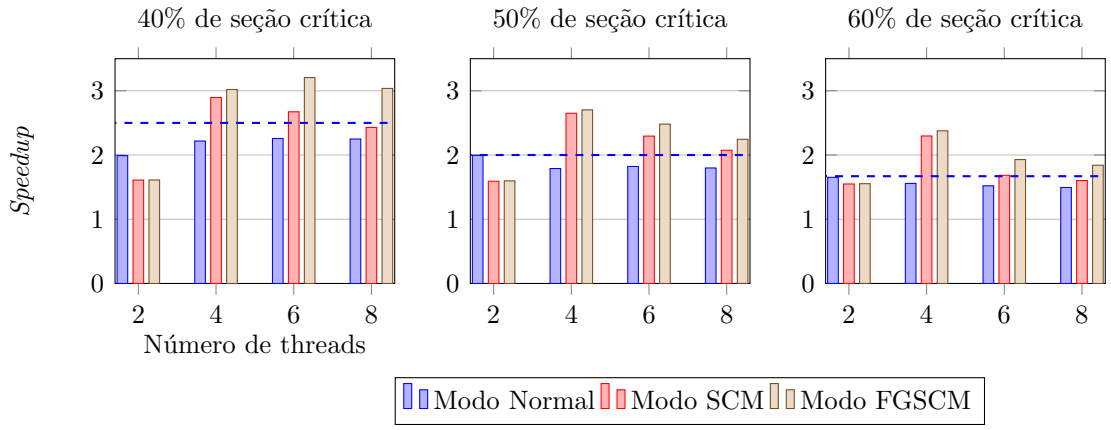

Figura 4.5: *Speedup* para experimentos com taxa de escrita de 0,2

possível observar que FGSCM apresenta um ganho evidente de desempenho em relação SCM quando *hyper-threads* (casos com 6 ou 8 threads) são utilizadas [3].

A linha tracejada na Figura 4.5 indica uma estimativa teórica do valor máximo de *speedup*, baseado na Lei de Amdahl [5], de acordo com a proporção de execução paralela explorada em cada gráfico, o que é dado pelo complemento da proporção de região crítica. Isso significa que é esperado que valores de *speedup* de execuções paralelas no modo Normal não ultrapassem significativamente tal linha. Como os gráficos apontam, os modos SCM e FGSCM geralmente a ultrapassam, o que significa que, efetivamente, ambos exploram um nível de concorrência maior do que seria *possivelmente* explorado utilizando um lock padrão. Ainda mais, é possível observar que, quando 8 threads são utilizadas, apenas o esquema FGSCM consegue ultrapassar esse limite.

4.3.3.2 Melhores parâmetros de lock

Como visto na Seção 4.3.1, o *benchmark* possui dois parâmetros para controlar limites de tentativas de especulações para ambos modos FGSCM e SCM, a saber, os parâmetros *máximo de reexecuções* ($max\_retries$) e *máximo de tentativas auxiliares* (max_aux_tries). O par ($max\_retries$, max_aux_tries) e o modo de lock podem ser chamados de *parâmetros de lock*; e os outros parâmetros (número de threads, taxa de escrita, valores de *overhead* etc) podem ser referidos como *parâmetros independentes*, uma vez que não são relacionados ao lock utilizado pelo programa. Dessa forma, cada caso de execução do *benchmark*

[3] Experimentos em processadores com mais núcleos, o que não foi feito devido a indisponibilidade desse tipo de sistema ao autor durante a realização deste trabalho, poderiam ajudar a responder se tal ganho expressivo é em realmente função do uso de *hyper-threads* ou se trata-se de apenas uma coincidência.

Tabela 4.2: Frequência com que cada combinação (*max_retries*, *max_aux_tries*) foi ranqueada entre as melhores para cada modo de lock.

max_retries	*max_aux_tries*	FGSCM (%)	SCM (%)
2	2	65,91	52,27
2	5	75,76	51,52
2	10	**87,12**	55,30
5	2	47,73	61,36
5	5	50,76	**81,82**
5	10	51,52	73,48
10	2	40,15	53,79
10	5	40,91	58,33
10	10	37,88	51,52

pode ser visto como uma combinação de parâmetros de lock e parâmetros independentes.

Para que as melhores combinações de parâmetros de lock fossem encontradas para cada um dos modos FGSCM e SCM, foram contadas quantas vezes cada par (*max_retries*, *max_aux_tries*) ficou ranqueado entre as melhores execuções para cada combinação de parâmetros independentes [4]. Assim, para cada combinação de parâmetros independentes e cada um dos modos FGSCM e SCM, um par (*max_retries*, *max_aux_tries*) foi considerado entre os melhores do grupo de execuções contemplado se a diferença de tempos de execução do caso contemplado e o melhor do grupo não passasse de 2% do tempo de execução do modo Normal equivalente.

A Tabela 4.2 sumariza o resultado deste processo em termos de frequência relativa ao número de combinações possíveis de parâmetros independentes. Ela mostra que, em geral, os melhores valores para (*max_retries*, *max_aux_tries*) para os modos FGSCM e SCM são (2, 10) e (5, 5), respectivamente.

Em particular, no caso do modo FGSCM, o valor baixo para o parâmetro *max_retries* é facilmente justificável. Em princípio, uma vez que uma transação aborta por conflito, o lock auxiliar associado à fonte de conflito já é conhecido. Portanto, é geralmente melhor que o mesmo seja rapidamente adquirido em vez de repetidamente tentar a especulação sem passar pelo caminho de serialização ao risco de gerar conflito com (i) transações que estejam na mesma situação, (ii) transações futuras, ou (iii) transações que já estejam no caminho de serialização. No entanto, vale lembrar que foi utilizada uma distribuição uniforme para endereços de memória acessados pelas threads do *benchmark* – aplicações reais podem apresentar distribuições diferentes, e, por isso, existiria a possibilidade do esquema FGSCM desempenhar melhor com valores maiores de *max_retries* para tais aplicações.

[4] Vale notar que cada combinação de parâmetros independentes agrupa casos para diversas combinações de parâmetros de lock – apenas um caso de execução no modo Normal e casos de execução para os modos FGSCM e SCM, onde se variam os valores de (*max_retries*, *max_aux_tries*)

4.3.3.3 Classificando casos de execução

Uma outra maneira de olhar para os resultados dos experimentos é classificar cada caso de execução do *benchmark* em alguns grupos de interesse:

FGSCM-win* Casos onde o FGSCM é melhor que ambos modos SCM e Normal;

FGSCM-SCM-win* Casos onde FGSCM e SCM empatam em desempenho e são ambos melhores que o modo Normal;

FGSCM-other* Casos que não se encaixam com os dois primeiros grupos, mas que ainda são favoráveis ao modo FGSCM [5], isto é, casos em que FGSCM nunca perde para os outros modos e, às vezes, até chega a ser melhor que um deles;

Normal-win Casos onde o modo Normal é melhor que ambos FGSCM e SCM;

FGSCM-lose Casos que não se encaixam o grupo Normal-win e onde FGSCM é pior que pelo menos um dos outros modos.

Os grupos decorados (com o sufixo "*") refletem casos onde o esquema FGSCM supera ou empata com a desempenho de outros modos. Em particular, o grupo FGSCM-win coleta os melhores casos para a nova abordagem proposta com este trabalho, onde *uma melhoria real sobre o algoritmo original SCM deve ser observada*; seguido pelo grupo FGSCM-SCM-win, onde *o esquema FGSCM deve manter as melhorias do algoritmo SCM com relação ao modo Normal.*

A classificação foi realizada comparando o tempo de execução dos casos de execução com a mesma combinação de parâmetros independentes. Dado um conjunto de parâmetros de lock para uma determinada combinação de parâmetros independentes, foi selecionada a melhor combinação de parâmetros de lock para cada um dos modos Normal, SCM e FGSCM, resultando em três casos (*conjunto dos melhores*). Entre esses três casos, um modo de lock foi considerado melhor que outro se a diferença entre os tempos de execução fosse no mínimo 5% do tempo de execução do caso com o modo Normal.

Um olhar detalhado nos grupos **FGSCM-win** e **FGSCM-SCM-win**, que são os de mais interesse desta seção, releva que:

- a maioria dos casos em FGSCM-SCM-win usam valores de taxa de escrita baixos e moderados enquanto a distribuição de taxas de escrita no grupo FGSCM-win é de certa forma mais balanceada entre valores baixos, moderados e altos (embora taxas mais baixas ainda sejam predominantes). A Tabela 4.3 mostra que *a abordagem de granularidade fina é mais tolerante a valores de taxa de escrita mais altos que*

[5] Este grupo poderia ser separado em outros grupos utilizando critérios de classificação mais refinados. Tal detalhamento, entretanto, não é de interesse da análise atual e, portanto, não é explorado neste trabalho.

Tabela 4.3: Distribuição em porcentagens dos casos nos grupos FGSCM-win e FGSCM-SCM-win para cada taxa de escrita.

Grupo / Taxa de escrita	Baixa (0,2)	Moderada (0,5)	Alta (0,8)
FGSCM-win	45,8	29,2	25
FGSCM-SCM-win	60,7	25	14,3

o algoritmo original SCM, o que já era de se esperar, dado que taxas de escritas mais altas são propícias a causar mais transações a abortarem por conflito. Neste sentido, algoritmo FGSCM permite que uma quantidade dessas transações ainda executem concorrentemente enquanto SCM serializa todas elas;

- a maioria dos casos em FGSCM-SCM-win simula aplicações que gastam a maioria de seu tempo de execução em regiões críticas, cerca de 80% a 100% do tamanho da execução. Já para o grupo FGSCM-win, este intervalo cai para cerca de 40% a 60%. Isto mostra que, potencialmente, FGSCM poderia apresentar uma melhora sobre SCM para programas com regiões críticas e não críticas de tamanhos balanceados. Entretanto, os experimentos realizados foram baseados em um valor máximo de 10^5 divisões inteiras (soma dos parâmetros de *overhead*) para cada iteração de cada thread. Por isso, mais experimentos variando este valor, enquanto preservando a proporção de regiões críticas e não críticas, seriam necessários para validar tal proposição.

Mais adiante, a Seção 4.3.3.4 descreve casos onde geralmente não há melhorias significantes do algoritmo FGSCM sobre os outros modos de lock. Com tais casos filtrados, a distribuição resultante da classificação dos resultados pode ser vista na figura Figure 4.6. O gráfico mostra que, em 31,48% dos casos, o esquema FGSCM mantém as melhorias de desempenho que o esquema SCM já provia sobre o esquema padrão de lock; e que o esquema FGSCM apresenta uma melhoria sobre o SCM em 33,33% dos casos. Ainda mais, para uma parte significante dos casos (77,77%), FGSCM pode ser utilizado sem perda e com possibilidade de ganho em desempenho (isto é, união dos casos decorados).

4.3.3.4 Limitações

Alguns dos resultados experimentais foram úteis para confirmar algumas características do *benchmark* que já eram esperadas de serem desfavoráveis ou não muito úteis para a abordagem proposta:

i) *regiões críticas muito pequenas*: isto implica em quase nenhuma contenção, o que faz com que todos os modos de lock escalem bem o tempo de execução em geral. De fato, para todos os casos de execução com regiões críticas muito pequenas, embora o modo FGSCM *não* seja pior que os outros modos, ele também não apresenta melhorias expressivas;

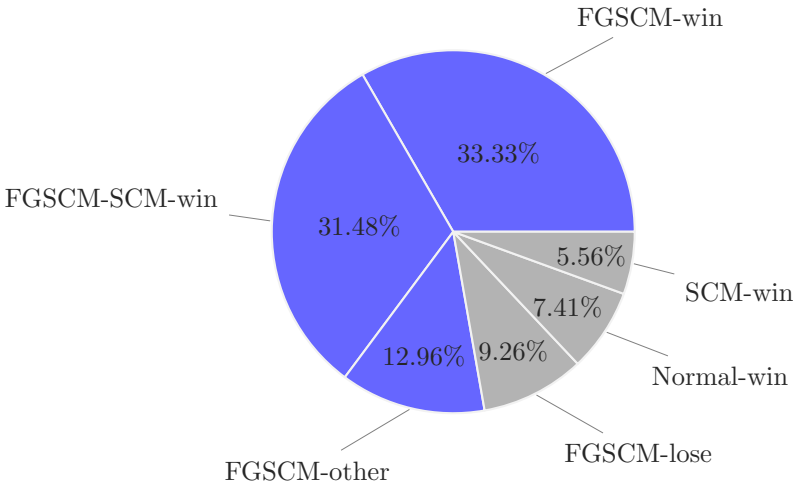

Figura 4.6: Gráfico exibindo a distribuição dos grupos formados

ii) *alta taxa de escrita*: como o *benchmark* utiliza uma distribuição uniforme para definir os índices do bytes a serem acessados pelas operações de escrita e leitura realizadas pelas threads, quanto mais alta a taxa de escrita, maior a probabilidade de transações futuras entrarem em conflito com outras em execução;

iii) *uso de apenas 2 threads*: por definição, em teoria, o esquema FGSCM não provê qualquer melhoria em relação ao SCM quando apenas duas threads são consideradas. Quando há um conflito entre estas duas threads [6], haverá apenas um lock auxiliar a ser tomado, o que faz a abordagem de granularidade fina equivalente à abordagem original.

4.3.4 Resumo dos principais resultados

Os experimentos com o esquema FGSCM através do *benchmark* mostraram que a nova abordagem apresenta ganhos de desempenho em relação ao SCM e a um *spin lock* típico quando regiões críticas apresentam baixa taxa de escrita e possuem tamanhos balanceados com o resto do programa. Também foi possível observar que FGSCM bem como SCM conseguiram aumentar o nível de paralelismo explorado a um ponto onde não seria teoricamente possível utilizando o modo *Normal* do *benchmark*, o que significa que ambas abordagens de TLE efetivamente *conseguiram paralelizar a parte considerada serial* pela abordagem tradicional de exclusão mútua. Por último, vale notar que FGSCM é mais tolerante a taxas de escrita mais altas quando comparado ao esquema SCM.

Pelos resultados experimentais, também foi possível observar que o esquema FGSCM não apresenta ganhos de desempenho quando regiões críticas são muito pequenas, existe alta taxa de escrita ou apenas duas *threads* são utilizadas.

[6] Isto não leva em consideração a possibilidade de outro processo do sistema operacional fazer as threads serem abortadas devido a conflitos em diferentes regiões de memória não controladas pelo *benchmark*. Para todos os efeitos, tal cenário não é explorado por este trabalho.

Capítulo 5

Conclusão

A estagnação do desempenho de processadores de único núcleo e a demanda por mais poder computacional acabou por levar a indústria a explorar o desenvolvimento de processadores com múltiplos núcleos de execução. A possibilidade de escrever programas paralelos provocou a necessidade da adoção de mecanismos de sincronização que tornassem possível acesso a recursos compartilhados sem causar inconsistência ao programa, sendo estruturas de *exclusão mútua* (locks) um dos mais populares. Um outro paradigma de programação paralela emergente é o de *memória transacional*, que possui suas vantagens em relação ao uso de locks, sendo programabilidade e redução de sincronização as mais marcantes.

Uma solução proposta pela academia para reduzir a quantidade de sincronização imposta pelo uso tradicional de locks são as técnicas de omissão de lock, que incorporam aspectos provenientes do paradigma de memória transacional e podem ser facilmente integradas a aplicações concorrentes legadas que utilizam locks como meio de de sincronização. Neste contexto, diferentes abordagens utilizando memória transacional (TLE) têm sido propostas para amenizar um fenômeno popularmente conhecido por *lemming effect*. Uma delas é chamada de *Software-assisted Conflict Management (SCM)*, onde se utiliza um lock auxiliar para serializar transações abortadas e permitir outras transações a executarem concorrentemente.

O trabalho apresentado nesta dissertação propôs o esquema *Fine-grained Software-assisted Conflict Management* (FGSCM), uma abordagem de TLE que estende o esquema SCM através do uso de diferentes locks auxiliares no caminho de serialização de acordo com a fonte de conflito da transação a ser serializada. Um *microbenchmark* específico foi implementado para conduzir a avaliação deste novo método. Resultados experimentais indicaram que, para programas com predominância de operações de leituras (80% de taxa de leitura), FGSCM apresenta ganhos em desempenho de 11% e 36% em relação ao esquema SCM original e ao uso de um *spin lock* comum, respectivamente. Os experimentos ainda mostraram que, em casos típicos, ambos FGSCM e SCM superam o limite máximo teórico de *speedup* calculado considerando regiões críticas com exclusão mútua, o que sig-

nifica que, efetivamente, ambas abordagens aumentam a parcela dinamicamente paralela do programa executado.

Trabalhos futuros com respeito ao FGSCM envolvem:

- criar um ambiente de simulação capaz de emular um processador com suporte a memória transacional que provê meios necessários para a definição da fonte de conflito de uma transação abortada, o que permitiria explorar o uso de FGSCM em aplicações reais com um maior nível de confiabilidade;
- como mencionado antes, explorar uma versão do algoritmo que utilize suporte de compilador para instrumentar regiões críticas de forma a estimar possíveis fontes de conflito de uma região crítica a ser executada, o que permitiria o uso de FGSCM utilizando a tecnologia atualmente disponível.

Referências Bibliográficas

[1] Y. Afek, A. Levy e A. Morrison. Programming with hardware lock elision. Em *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 295–296, 2013.

[2] Y. Afek, A. Levy e A. Morrison. Software-improved hardware lock elision. Em *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pp. 212–221, 2014.

[3] Y. Afek, A. Matveev, O. R. Moll e N. Shavit. Amalgamated lock-elision. pp. 309–324, outubro de 2015.

[4] Y. Afek, A. Matveev e N. Shavit. Pessimistic software lock-elision. pp. 297–311, outubro de 2012.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. Em *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 483–485, abril de 1967.

[6] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer e M. Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, março/abril de 2009.

[7] 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.

[8] D. Dice, A. Kogan e Y. Lev. Refined transactional lock elision. pp. 1–12, março de 2016.

[9] D. Dice, A. Kogan, Y. Lev, T. Merrifield e M. Moir. Adaptive integration of hardware and software lock elision techniques. pp. 188–197, junho de 2014.

[10] D. Dice, Y. Lev, M. Moir e D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. Em *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 157–168, março de 2009.

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

[12] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept de 1972.

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

[14] J. L. Hennessy e D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edição, 2011.

[15] 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.

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

[17] A. Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, março de 2014.

[18] S. Y. Kung. *VLSI Array Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[19] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira e T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[20] D. Locke, L. Sha, R. Rajikumar, J. Lehoczky e G. Burns. Priority inversion and its control: An experimental investigation. *Ada Lett.*, VIII(7):39–42, junho de 1988.

[21] 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.

[22] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edição, 2011.

[23] R. Rajwar e J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. Em *Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on*, pp. 294–305, Dec de 2001.

[24] A. Roy, S. Hand e T. Harris. A runtime system for software lock elision. Em *Proceedings of the 4th European Conference on Computer Systems*, pp. 261–274, abril de 2009.

[25] N. Shavit e D. Touitou. Software transactional memory. Em *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pp. 204–213, agosto de 1995.

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

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

TERMO DE REPRODUÇÃO XEROGRÁFICA

Autorizo a reprodução xerográfica do presente Trabalho de Conclusão, na íntegra ou em partes, para fins de pesquisa.

São José do Rio Preto, 17/10/2017

Gustavo José de Sousa
Assinatura do autor