Experimentos com Gerenciamento de Contenção em uma Memória Transacional com Suporte em *Software*

Fernando Kronbauer, Sandro Rigo

Universidade Estadual de Campinas
Laboratório de Sistemas de Computação
Av. Albert Einstein, 1251, Campinas, Brasil
http://www.lsc.ic.unicamp.br

Resumo

Devido à grande disseminação recente de arquiteturas paralelas, mais e mais programadores são expostos aos problemas relacionados ao uso dos mecanismos tradicionais de controle de concorrência. Memórias transacionais têm sido propostas como um meio de aliviar as dificuldades encontradas ao escreverem-se programas paralelos. Neste trabalho exploramos um sistema de memória transacional em software (STM), apresentando uma abordagem nova para gerenciar a contenção entre transações, que leva em consideração os padrões de acesso aos diferentes dados de um programa ao escolher o gerenciador de contenção usado para o acesso a estes dados. Elaboramos uma modificação da plataforma de STM que nos permite realizar esta associação entre dados e gerenciamento de contenção, e realizamos uma caracterização baseada nos padrões de acesso aos dados de um programa executando em diferentes sistemas de computação.

1. Introdução

Em geral, a programação paralela é considerada mais difícil que a programação sequencial. Paralelismo e não determinismo aumentam em muito a quantidade de informações que um desenvolvedor de software deve manter em mente enquanto programa. Para que linhas de execução (threads) possam cooperar na realização de trabalho útil, é necessário que se comuniquem de forma segura, ou seja, dados compartilhados entre as linhas precisam ser acessados de forma ordeira, coordenada e síncrona. Atualmente esta coordenação entre linhas de execução é em grande parte de responsabilidade exclusiva do programador, que de uma forma geral possui à sua disposição somente mecanismos de baixo nível, como travas de exclusão mútua (*locks*) e semáforos, para prevenir que duas linhas de execução concorrentes interfiram umas com as outras. Sistemas de memória transacional foram propostos como um modelo de programação genérico e flexível, que permite que linhas de execução leiam e escrevam posições de memória em uma única operação e de maneira atômica através de transações, sem os detalhes complicados dos protocolos de sincronização convencionais. O desenvolvedor precisa apenas marcar as seções de código que devem ser executadas de forma atômica e isolada, e o sistema cuida dos detalhes de sincronização.

Neste trabalho apresentamos as modificações feitas a uma implementação de memória transacional baseada em *software*, sem suporte específico em *hardware*, com o intuito de possibilitar a experimentação com diferentes estratégias de gerenciamento de contenção entre transações. As modificações propostas permitem um maior controle por parte do programador sobre os gerenciadores de contenção a serem utilizados pelas transações. Permitem ao programador associar um gerenciador específico a cada transação, bem como amarrar a estratégia de gerenciamento de contenção aos dados de um aplicativo baseando-se nos padrões de acesso a estes dados. Conduzimos experimentos avaliando a implementação e apresentamos resultados para uma variedade de sistemas de computação.

Este artigo está organizado da seguinte forma: a seção 2 introduz o problema da detecção e gerenciamento de conflitos entre transações; a seção 3 discute trabalhos relacionados; a seção 4 apresenta as modificações realizadas na biblioteca de STM; a seção 5 apresenta os resultados dos experimentos; e a seção 6 mostra nossas conclusões.

2. Detecção e gerenciamento de conflitos

Transações lêem e escrevem objetos compartilhados. Duas transações conflitam entre si se acessam o mesmo ob-

jeto e pelo menos um dos acessos é uma escrita. Para que uma transação possa escrever em um objeto, primeiramente precisa adquiri-lo. A aquisição de um objeto é o "gancho" que permite a detecção de conflitos: torna as transações que escrevem visíveis umas às outras, bem como às transações que lêem tais objetos.

Aquisições podem ocorrer em qualquer momento, a partir do acesso inicial aos objetos até a confirmação final da transação. Aquisições realizadas quando do primeiro acesso para escrita a um objeto são chamadas de aquisições imediatas. Aquisições feitas somente durante o processo de confirmação da transação são chamadas de tardias. Aquisições tardias permitem maior especulação, e dão mais oportunidade para que transações conflitantes executem em paralelo. O paralelismo entre uma transação que escreve em um objeto e um grupo de transações que lêem este mesmo objeto pode ser 100% aproveitável se a transação que escreve completar por último. O paralelismo entre transações que escrevem a um mesmo objeto possui natureza mais puramente especulativa: apenas uma destas transações pode ser concluída, no entanto não há uma forma geral de saber qual delas **deve** completar [6].

Uma vez que leitores e escritores se tornam visíveis, escolher as circunstâncias nas quais "roubar" um objeto (e desta forma abortar a transação que o adquiriu previamente) ou esperar até que este recurso seja liberado é um problema a ser resolvido pelo sistema de gerenciamento de contenção. O gerenciamento de contenção não afeta a corretude da implementação de um sistema de memória transacional, apenas o seu desempenho. Algumas propostas de sistemas de TM possuem um método fixo para gerenciamento de contenção, ao passo que outras propostas tratam o problema como um aspecto modular do sistema, podendo ser alterado para melhorar o desempenho de um programa sob determinada carga de trabalho.

3. Trabalhos Relacionados

3.1 A biblioteca de Memória Transacional RSTM

RSTM é uma implementação de memória transacional baseada em *software* (STM) e implementada como uma biblioteca C++. Em nosso trabalho exploramos a versão 3 da biblioteca RSTM. RSTM possui uma interface de programação baseada em *smart pointers* e *templates*, que tem como objetivo reduzir a complexidade de programação e capturar vários erros de programação comuns [1]. RSTM possui duas implementações internas. A primeira é não-bloqueante e utiliza uma única indireção para acessar dados transacionais. A segunda é bloqueante, sem níveis de indireção e baseada em registros de gravação (*redo-logs*).

A implementação não-bloqueante de RSTM utiliza um nível de indireção para acessar dados agregando informações adicionais a cada objeto transacional. Mais especificamente, dois novos campos são adicionados a cada objeto transacional: um apontandor para o descritor da transação ao qual o objeto pertence e o outro para a versão antiga do objeto. Um objeto está sob propriedade de uma transação se o descritor de transação para o qual aponta possuir o status "ativo". Se o status for "abortado", então a versão atual do objeto é aquela apontada como sendo a versão "antiga". Caso o status for "confirmado", então o objeto já está em sua versão correta. Um objeto transacional é acessado através de um cabeçalho especial, desta forma implicando em apenas um nível de indireção para acessos. Uma vez que uma transação tenha lido um objeto, podemos ter certeza de que a versão lida não irá mudar. Quando uma transação realiza escritas em um objeto, altera uma cópia privada. Se a implementação utiliza leituras invisíveis, a validação de todos os objetos lidos ou escritos desde o início de uma transação precisa ser realizada incrementalmente cada vez que um objeto é aberto para leitura ou escrita. Também há suporte a leitores visíveis.

A implementação interna baseada em registros de gravação (*redo-logs*) também adiciona dois campos a cada objeto transacional. Um objeto não pertence a nenhuma transação se o primeiro campo se comportar como um número de versão ímpar e se o segundo campo for nulo. De outra forma, o objeto foi adquirido, e o primeiro campo aponta para o descritor da transação que adquiriu o objeto e o segundo campo aponta para o registro de gravação. Um caso especial ocorre quando o primeiro campo possui o valor 2, e o dono do objeto está presumivelmente copiando o registro de gravação para o objeto. Objetos ficam inacessíveis durante a aplicação do registro de gravação, e uma transação que deseja acessá-los deve esperar. Diferentemente da implementação não-bloqueante, objetos transacionais são acessados diretamente, sem precisar de um objeto de cabeçalho. O leitor deve notar que um objeto aberto para leitura não é imutável, uma vez que a aplicação do registro de gravação ocorre *in loco*, e portanto uma transação precisa validar incrementalmente os objetos por ela abertos a cada acesso, e não somente quando da sua abertura. Além do método *clone*, objetos transacionais precisam implementar também uma operação de aplicação do registro de gravação (*redo*).

Ambas as implementações bloqueante e não-bloqueante suportam aquisições tardias ou imediatas. Se uma transação tenta adquirir um objeto que foi adquirido por outra, um gerenciador de contenção é invocado para tratar o conflito.

3.2 Outros trabalhos sobre gerenciamento de contenção

O gerenciamento de contenção entre transações em memória foi extensivamente estudado em outros trabalhos publicados [8, 4, 3]. Dentre estes trabalhos, o de maior interesse em nosso contexto é um sobre gerenciamento polimórfico de contenção [3]. No trabalho referido, diferentes gerenciadores de contenção podem ser associados a diferentes transações de uma forma parecida com a apresentada em nosso trabalho, apesar de os autores não explorarem a noção de associação de diferentes gerenciadores de contenção aos dados baseando-se nos padrões de acesso a estes. Em vez disso, os autores propõem a adaptação da estratégia de gerenciamento de contenção baseando-se na variação da carga de trabalho — mais precisamente, escolhendo o gerenciador de contenção baseando-se no número de linhas de execução ativas no programa.

ASTM explora outros aspectos da adaptação do sistema de memória transacional baseando-se na carga de trabalho do programa [5]. Explora quatro dimensões diferentes do espaço de projeto de um sistema de TM: aquisições imediatas versus aquisições tardias, o método para aquisição de objetos, a estrutura de meta-dados, e diferentes semânticas não-bloqueantes para transações. O sistema de TM adapta-se ao longo destas quatro dimensões em tempo de execução para atingir as necessidades da aplicação. ASTM não explora o uso ou adaptação de diferentes gerenciadores de contenção de acordo com os padrões de acesso ao dados de um programa.

4 Trabalho realizado

4.1 Gerenciamento de contenção em RSTM

Como em DSTM [7], ASTM [5] e SXM [3], o gerenciamento de contenção é tratado como um aspecto modular do sistema. Cada transação é associada a um objeto que representa a sua estratégia de gerenciamento de contenção corrente. Este objeto possui métodos que casam o ciclo de vida de uma transação (**onBeginTransaction**, **onTryCommitTransaction**, **onTransactionCommitted**, e **onTransactionAborted**), métodos que casam os diferentes eventos que ocorrem devido a interações com objetos transacionais (**onContention**, **onOpenRead**, **onOpenWrite** e **onReOpen**), e um método para decidir se deve ou não abortar uma transação conflitante (**shouldAbort**). A estratégia de gerenciamento de contenção não pode ser mudada no decorrer de uma transação, uma vez que os métodos invocados em função do ciclo de vida da transação são em geral utilizados para inicializar e atualizar os dados internos do objeto que representa a estratégia de gerenciamento.

Dentre as estratégias de gerenciamento de contenção propostas na literatura [8, 4], escolhemos apresentar algumas que trouxeram desempenho razoável na execução do *benchmark* proposto nesse trabalho. O gerenciador de contenção *Aggressive* é o mais simples de todos: ele prontamente aborta qualquer transação conflitante. O gerenciador *Greedy* usa um marcador de tempo (*timestamp*), adquirido pela transação quando de sua primeira tentativa de execução, para determinar sua "idade". Se duas transações estão em conflito, em geral a mais velha prevalece. Mas se a transação mais jovem está bloqueada pela mais velha e detecta que esta também está bloqueada, esperando por um recurso adquirido por outra transação, então a transação mais jovem aborta a mais velha e toma-lhe o recurso. O gerenciador *Killblocked* marca a transação como bloqueada quando esperando por algum recurso transacional. Se em uma tentativa subseqüente de abrir o mesmo objeto a transação adversária também está bloqueada, a adversária é abortada. De outra forma a transação continua esperando por um certo número de intervalos de tempo fixos, tentando acessar o mesmo objeto, até que desiste de esperar e aborta a transação conflitante.

O gerenciador de contenção *Karma* dá prioridade a uma transação baseando-se no número de objetos acessados por ela desde sua primeira tentativa de execução. A contagem de objetos acessados é portanto reiniciada quando a transação termina de forma bem sucedida. Uma transação de prioridade menor espera por um certo número de intervalos de tempo fixos ao tentar acessar um objeto adquirido por outra transação de prioridade mais alta, mas se o número de tentativas para acessar um recurso excede a diferença entre as prioridades das duas transações, a transação de prioridade mais baixa aborta a de prioridade mais alta e toma-lhe o recurso. O gerenciador *Polka* é muito parecido com o *Karma*, a diferença sendo que a transação bloqueada espera por intervalos de tempo exponencialmente crescentes, com um componente randômico. O gerenciador *Eruption* também deriva de *Karma*, com a diferença de que a transação bloqueada adiciona sua prioridade a da transação conflitante para que esta tenha a possibilidade de vencer conflitos com outras adversárias, terminar sua execução e liberar o recurso o quanto antes. *Polkaruption* combina os princípios de *Polka* e *Eruption*: como *Polka*, *Polkaruption* usa o número de objetos acessados para determinar a prioridade de uma transação e faz esperas exponencialmente maiores (com um componente randômico) quando a transação está bloqueada. Como *Eruption*, adiciona sua prioridade à prioridade da transação conflitante, para que esta termine o quanto antes e libere o recurso sob conflito. *Highlander* também se baseia nos princípios de *Polka*, mas quando uma transação aborta sua adversária, adiciona a prioridade da adversária a sua. Outra estratégia de gerenciamento de contenção baseada em *Polka* é a *Whpolka*, na qual

objetos abertos para escrita tem peso maior ao incrementar a prioridade da transação.

4.2 Modificações Propostas

Foi necessário realizar algumas modificações na biblioteca RSTM de modo a permitir que diferentes gerenciadores de contenção possam ser associados a diferentes transações ou a diferentes objetos transacionais. Primeiramente, tentamos inserir um campo adicional em cada objeto transacional para denotar o gerenciador de contenção a ele associado, e fazer com que a transação detectasse qual gerenciador de contenção estava associado ao primeiro objeto aberto pela transação e utilizar este gerenciador pelo restante de sua execução. Este campo adicional teria semânticas de acesso semelhantes aos demais campos do objeto definidos pelo programador, e portanto os métodos **clone** e **redo** precisariam levá-lo em consideração ao criar clones e aplicar registros de gravação. Esta abordagem demonstrou-se inviável em função do trabalho adicional inserido nos métodos **clone** e **redo**. Também tentamos utilizar herança para introduzir o campo citado apenas em objetos transacionais específicos, mas então foi preciso utilizar conversão de tipos dinâmica ao consultar o objeto a respeito de qual gerenciador de contenção estava associado a ele, o que pareceu ser outra fonte significante de trabalho adicional. E de qualquer forma, um teste adicional ainda deveria ser realizado a cada abertura de objeto pela transação, para determinar se o mesmo era o primeiro sendo acessado.

Decidimos então usar uma abordagem mais simples e direta. Permitimos ao programador associar uma estratégia de gerenciamento de contenção a uma transação, modificando a *macro* que delimita seu início (**BEGIN_TRANSACTION**), fazendo com que recebesse como parâmetro um valor enumerado capaz de identificar o gerenciador de contenção a ser usado. O gerenciador pode desta forma ser associado ao objeto que encapsula a estrutura de dados transacional, e diferentes gerenciadores de contenção podem ser associados a diferentes estruturas de dados ou mesmo a diferentes operações em uma mesma estrutura de dados. Cada linha de execução possui um vetor com gerenciadores de contenção pré-alocados, e o parâmetro passado a **BEGIN_TRANSACTION** é usado para configurar o gerenciador de contenção a ser utilizado pela transação ao começo de sua execução. Na figura 1 mostramos o idioma de programação descrito no contexto da implementação de uma árvore binária balanceada. Note que na linha 4 definimos um campo que armazena o tipo do gerenciador de contenção associado à estrutura de dados. Este campo é inicializado no construtor (linha 9) e usado para especificar o gerenciador de contenção em operações de busca na estrutura (linha 14). Note que cada instância de árvore balanceada no programa pode ser associada com

```
1.  class RBTree
2.   {
3.   private:
4.     stm::cm::CMEnum m_cm;
5.     stm::sh_ptr<RBNode> sentinel;
6.
7.   public:
8.     RBTree(stm::cm::CMEnum cm)
9.         : m_cm(cm),
10.          sentinel(new RBNode()) { }
11.
12.    virtual bool lookup(int val) const
13.      {
14.        BEGIN_TRANSACTION_CM(m_cm);
15.
16.        // ...
17.
18.        END_TRANSACTION;
19.      }
20.
21.    // ...
22.  };
```

Figura 1. Seg. de código da árvore vermelha-e-preta.

diferentes gerenciadores de contenção.

O trabalho adicional inserido é bastante pequeno e pago somente uma vez, no início da transação. Também precisamos realizar pequenas limpezas no código dos gerenciadores de contenção, para permitir a interação entre diferentes implementações de gerenciadores (lembremo-nos de que **shouldAbort** pode receber como parâmetro gerenciadores de tipos arbitrários). Estas limpezas de fato trouxeram pequena melhora ao desempenho da biblioteca, se comparado com a implementação original de RSTM.

Para compilar e executar a versão baseada em registros de gravação em arquiteturas x86 de 32 bits, precisamos portar a biblioteca para funcionar nesta plataforma, uma vez que o time de desenvolvimento da versão 3 de RSTM dá suporte a esta implementação interna somente para a plataforma SPARC. Notamos no entanto que a versão 4 da biblioteca, recentemente disponibilizada, possui suporte a ambas as implementações internas (bloqueante e não-bloqueante) tanto em arquiteturas SPARC quanto x86.

4.3 Experimentos

Para avaliar as modificações feitas à biblioteca de STM, concebemos um *benchmark* consistindo em estruturas de dados com diferentes padrões de acessos. Para isolar componentes capazes de interferir na avaliação, as estruturas de dados devem ser do mesmo tamanho e tipo, a única diferença sendo o nível de contenção encontrado pelas

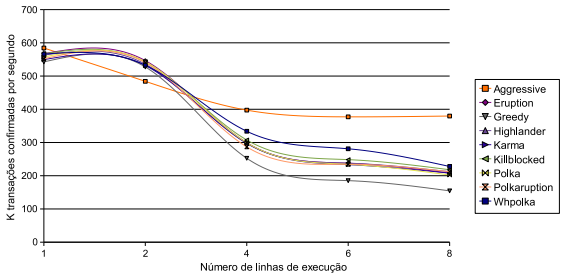

Figura 2. Core 2 Duo, não-bloq., alta cont.

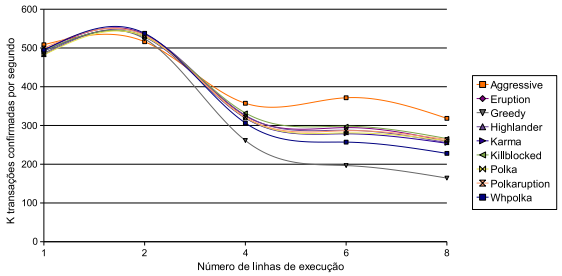

Figura 3. Core 2 Duo, não-bloq., baixa cont.

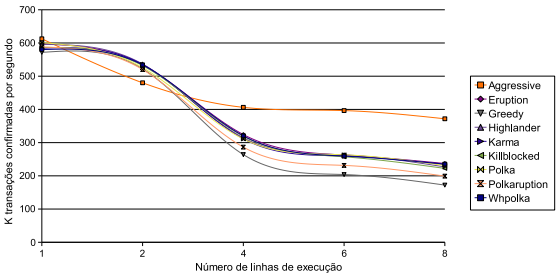

Figura 4. Core 2 Duo, bloqueante, alta cont.

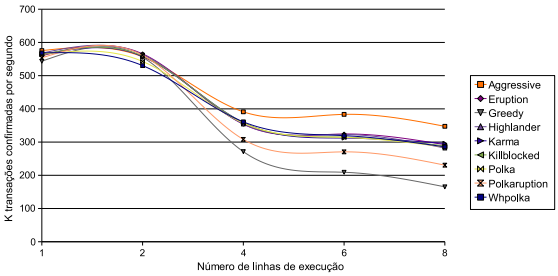

Figura 5. Core 2 Duo, bloqueante, baixa cont.

transações ao acessá-las.

Escolhemos aproveitar a implementação de árvores balanceadas do tipo vermelha-e-preta encontrada junto aos *benchmarks* disponíveis na implementação original da biblioteca RSTM. Árvores balanceadas exibem grande potencial de desempenho para acessos paralelos a diferentes partes de uma mesma árvore, ao mesmo tempo que provêm a oportunidade de conflitos entre transações quando atualizações precisam rebalancear as subárvores, propagando modificações na estrutura de dados desde os nós-folha até a raiz. Criamos uma tabela de dispersão de tamanho fixo contendo árvores balanceadas. Operações sobre a estrutura de dados composta precisam primeiramente encontrar a árvore apropriada para o operando. Isto é feito dividindo o operando pelo número de árvores, e usando o resultado para indexar a tabela de dispersão.

Para assegurar diferentes padrões de acesso a diferentes estruturas de dados, geramos os operandos com diferentes probabilidades. Há uma chance aproximada de 50% de que o número gerado seja mapeado à primeira árvore, enquanto os 50% restantes são igualmente distribuídos entre as demais árvores binárias. Portanto, a primeira árvore possui um padrão de acesso de alta contenção, enquanto as demais são acessadas sob baixa contenção. Conduzimos experimentos com 2 até 6 árvores de baixa contenção, e como os resultados se mostraram consistentes ao longo deste espectro, escolhemos mostrar apenas os resultados para 1 árvore binária de alta contenção e 4 árvores de baixa.

Todos as variações do *benchmark* foram executadas três vezes por períodos de 60 segundos, e as médias aritméticas foram tiradas. A estratégia de alocação de memória utilizada foi a pilha de memória com coleta automática de lixo, as heurísticas de contador global de finalização de transações foram desligadas e a privatização de dados feita através de barreiras unidirecionais transacionais (*transactional fences*). A estratégia de validações utilizada foi a de leitores invisíveis, com aquisições imediatas. Mapeamos um total de 256 elementos a cada árvore, isto é, para uma árvore de alta contenção e 4 de baixa, os operandos encontram-se na amplitude de 0 até 1279. Os tipos de operações foram particionados igualmente, sendo um terço buscas, um terço inserções e um terço remoções. Executamos os *benchmarks* em ambas as implementações internas da biblioteca RSTM, a não-bloqueante e a bloqueante. Três sistemas de computação diferentes foram utilizados para as execuções. O primeiro, um sistema baseado no processador Intel Core 2 Duo a 2.8 GHz, com 2 GB de memória RAM. O segundo, um sistema baseado no processador Intel Core 2 Quad, a 2.4 GHz e com 4 GB de RAM. O terceiro, um sistema com dois processadores Intel Core 2 Quad a 2GHz e 4 GB de RAM. Todos os sistema utilizaram como sistema operacional uma distribuição Linux padrão (*kernel* 2.6).

Avaliamos o melhor gerenciador de contenção para uso sob alta e baixa contenções, e também em um cenário de

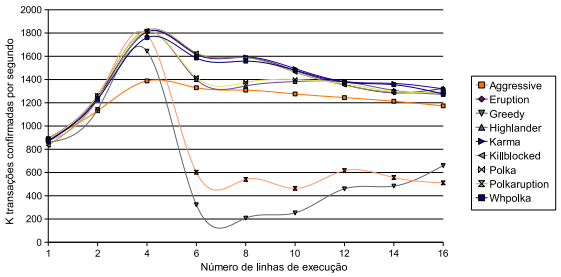

Figura 6. Core 2 Quad, não-bloq., alta cont.

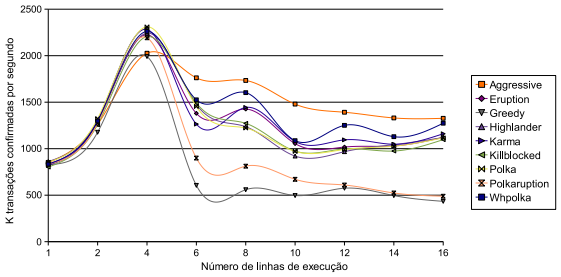

Figura 7. Core 2 Quad, não-bloq., baixa cont.

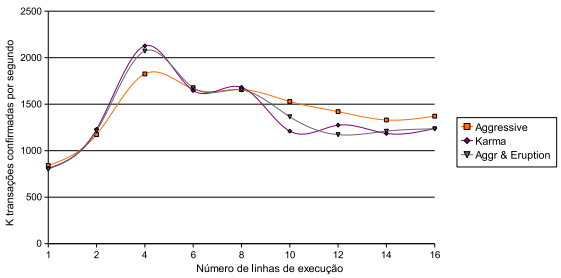

Figura 8. Core 2 Quad, não-bloq., cont. mista.

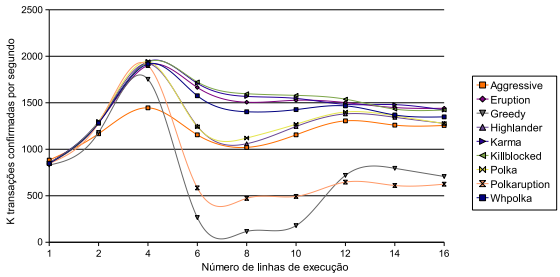

Figura 9. Core 2 Quad, bloqueante, alta cont.

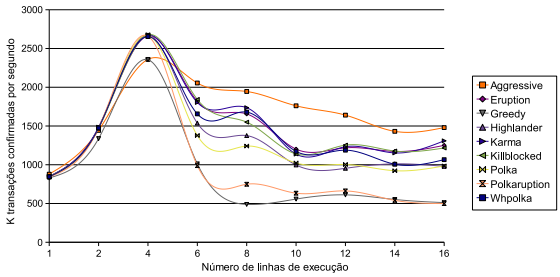

Figura 10. Core 2 Quad, bloqueante, baixa cont.

contenção mista. Primeiramente executamos experimentos para apenas uma árvore balanceada, sob alta contenção, e apresentamos os resultados nas figuras 2, 4, 6, 9, 12 e 14. Então executamos os experimentos com as árvores de baixa contenção, e mostramos os resultados para as execuções com quatro estruturas de dados nas figuras 3, 5, 7, 10, 13 e 15. O próximo passo seria identificar os melhores gerenciadores de contenção para alta e baixa contenção e então associar os melhores gerenciadores às estruturas de dados baseando-se nos padrões de acesso a estas estruturas de dados.

Fizemos com que o número de linhas de execução variasse ao longo das execuções do *benchmark*, de uma linha até o número de linhas de execução igual a quatro vezes o número de núcleos de processamento no sistema. Como Dragojevic argumenta [2], não podemos esperar que um sistema de TM aumente seu desempenho quando o número de linhas de execução excede o número de núcleos de processamento, mas seu desempenho deve ao menos degradar de forma amena sob estas circunstâncias, uma vez que usuários de um programa escrito com memória transacional na maioria das vezes não poderão fazer um ajuste fino da aplicação de acordo com o número de núcleos de processamento disponíveis. Também argumentamos que o número de núcleos do sistema é apenas um limite máximo do número de núcleos que um programa terá disponível a

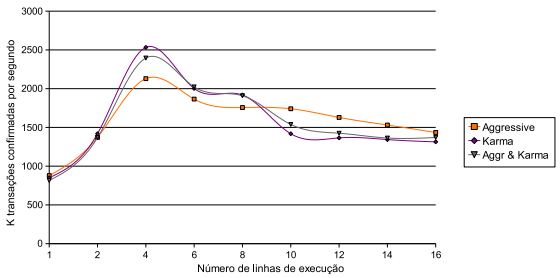

Figura 11. Core 2 Quad, bloqueante, cont. mista.

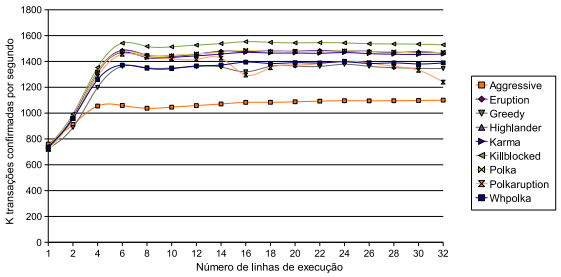

Figura 12. Dual Core 2 Quad, não-bloq., alta cont.

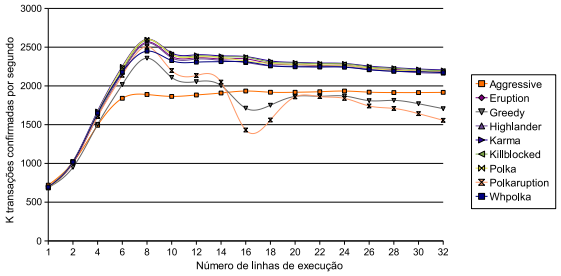

Figura 13. Dual Core 2 Quad, não-bloq., baixa cont.

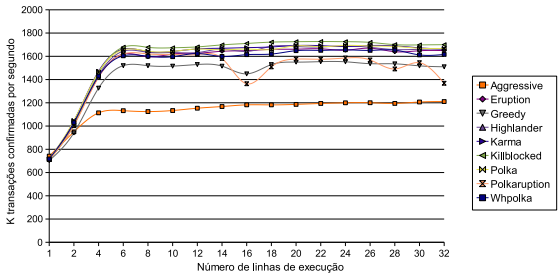

Figura 14. Dual Core 2 Quad, bloqueante, alta cont.

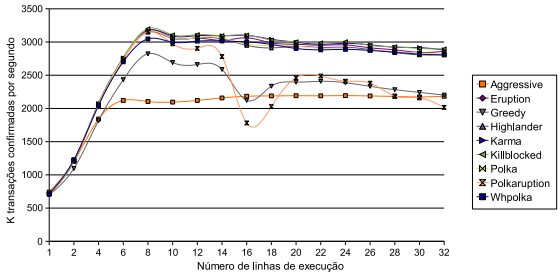

Figura 15. Dual Core 2 Quad, bloq., baixa cont.

qualquer momento, e que em um ambiente multitarefa é bem provável que um aplicativo escrito com memória transacional terá de competir com outros aplicativos (possivelmente também paralelos) pelos núcleos de processamento do sistema.

As figuras 2 até 5 mostram os resultados para as execuções do *benchmark* no sistema Intel Core 2 Duo. As primeiras duas figuras são os resultados para a implementação interna não bloqueante, enquanto que as duas seguintes são para a implementação bloqueante baseada em registros de gravação. Como o melhor gerenciador de contenção foi o mesmo (*Aggressive*) tanto no cenário de alta quanto no cenário de baixa contenção, não executamos os testes em um cenário de contenção mista.

Resultados semelhantes são apresentados para o sistema com dois processadores Intel Core 2 Quad (com um total de oito núcleos de processamento), nas figuras 12 até 15. Podemos ver que tanto sob alta quanto sob baixa contenção um mesmo gerenciador apresentou o melhor desempenho, desta vez o *Killblocked*. Nossa expectativa era encontrar um gerenciador que tivesse desempenho melhor sob baixa contenção e outro que apresentasse desempenho superior sob alta contenção. Novamente, não executamos o *benchmark* em um cenário de contenção mista.

Os resultados para o sistema com um processador Core 2 Quad (com um total de 4 núcleos de processamento) são mostrados nas figuras 6 até 11. Sob alta contenção (figuras 6 e 9), quatro gerenciadores apresentaram o melhor desempenho. Estes foram *Killblocked*, *Karma*, *Eruption* e *Whpolka*. Sob baixa contenção (figuras 7 e 10), o gerenciador *Aggressive* ofereceu o melhor desempenho de uma forma geral, especialmente quando o número de linhas de execução excede o número de núcleos de processamento. No cenário de contenção mista (figuras 8 e 11), as diferentes árvores balanceadas foram associadas aos melhores gerenciadores de contenção de acordo com os respectivos níveis de contenção. Nas figuras que representam o cenário de contenção mista, apresentamos apenas os melhores resultados para torná-las mais fáceis de interpretar, sem as poluir visualmente. Mais precisamente, apresentamos os melhores resultados para quatro linhas de execução, para quatro até oito linhas, e para oito até dezesseis linhas de execução.

Podemos ver que, no caso do sistema com um processador Core 2 Quad, misturar diferentes gerenciadores de contenção produz um bom impacto no desempenho do *benchmark* quando este executa com quatro até oito linhas de execução, isto é, quando o número de linhas excede em pouco o número de núcleos de processamento. Com oito até dezesseis linhas, o esquema de gerenciadores mistos tem desempenho melhor que o esquema utilizando apenas um daqueles gerenciadores melhores para baixa contenção (*Killblocked*, *Karma*, *Eruption* e *Whpolka*), uma vez que o

desempenho degrada de forma mais amena, mas o gerenciador *Aggressive* executa melhor que todos estes.

Notamos que conhecer o padrão de acesso às estruturas de dados não foi o suficiente para determinar os melhores gerenciadores de contenção para uso na aplicação, uma vez que os resultados mostraram alta variação nos diferentes sistemas de computação testados. Também, conhecer as diferentes configurações de *hardware* não foi suficiente porque, como podemos ver, os padrões de acesso aos dados de um programa possuem um papel importante ao decidir a melhor estratégia de gerenciamento de contenção a ser usada. Isto serve para reafirmar resultados encontrados anteriormente, que indicam que não há um gerenciador de contenção ideal a ser utilizado como uma escolha padrão [3]. Como os experimentos demonstraram, nossa implementação introduz custos adicionais de execução baixos, o que permite que seja utilizada naqueles casos em que diferentes gerenciadores de contenção executam melhor quando associados a diferentes dados ou transações. Aparentemente, nenhuma análise estática será suficiente para determinar o melhor gerenciador de contenção a ser usado em um programa, e encontrar o perfil de execução da aplicação em apenas uma configuração de *hardware* não irá funcionar para determinar o melhor gerenciador a ser utilizado se o sistema de computação mudar mesmo que apenas um pouco. A determinação dinâmica do perfil de execução do programa, bem como o ajuste dinâmico dos gerenciadores de contenção, parecem ser a resposta para o problema.

4.4 Conclusões

Diferentes trabalhos têm sido realizados buscando bons algoritmos de propósito geral para o problema de gerenciamento de contenção, mas parece pouco provável que programas escritos com memória transacional possuam um caso geral do problema. Pelo contrário, parece mais provável que o gerenciamento de contenção, bem como outros aspectos do sistema, devão receber um ajuste fino para alcançar bons níveis de desempenho, ao menos nos cenários que demandam níveis de desempenho bastante altos, como sistemas de processamento transacional *on-line* e bancos de dados. Em alguns casos permitiremos que o programador faça o ajuste fino de forma manual, e em outros desejaremos que o sistema o faça de forma automática.

O gerenciamento de contenção é uma área interessante para a busca pela melhoria de desempenho, uma vez que não traz impacto sobre a corretude do sistema, e existe uma grande quantidade de estratégias dentre as quais escolher. Podemos não somente querer escolher o melhor gerenciador padrão para todo o programa, mas podemos escolher gerenciadores individuais que agreguem o melhor desempenho a cada uma das transações da aplicação e a cada dado sendo acessado sob diferentes cargas de trabalho. O quão flexível podemos ser para a escolha de estratégias de gerenciamento depende do preço que queremos pagar por isto.

Neste trabalho mostramos uma forma de fazer o ajuste fino do gerenciamento de contenção em programas escritos com TM, associando o gerenciador aos dados acessados no aplicativo. Mostramos que a implementação demanda custos de execução adicionais bastante baixos. Como demonstramos, o nível de concorrência no programa e os padrões de acesso aos seus dados podem ter um impacto significativo no melhor gerenciador de contenção a ser usado no aplicativo, um aspecto não investigado em trabalhos anteriores. Podemos permitir que o programador confie em um gerenciador padrão para o programa todo ou dar-lhe condições para escolher programaticamente alguma das várias estratégias, baseando-se nos padrões de acesso e variações de carga de trabalho.

Referências

[1] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In *Second ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, USA, August 2007. In conjunction with PPoPP'07.

[2] A. Dragojevic, R. Guerraoui, and M. Kapalka. Dividing Transactional Memories by Zero. In *Third ACM SIGPLAN Workshop on Transactional Computing*. Salt Lake City, UT, USA, February 2008. In conjunction with PPoPP'08.

[3] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 303–323, New York, NY, USA, September 2005. LNCS, Springer.

[4] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, pages 258–264, New York, NY, USA, July 2005. ACM Press.

[5] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 354–368, New York, NY, USA, September 2005. LNCS, Springer.

[6] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. June 2006. In conjunction with PLDI'06.

[7] Maurice Herlihy and Victor Luchangco and Mark Moir and William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[8] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 240–248, New York, NY, USA, July 2005. ACM Press.