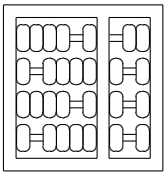

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Abortos Falsos em Sistemas de Memória Transacional em Software

Daniel Nicácio *Guido Araújo*

Technical Report - IC-09-32 - Relatório Técnico

September - 2009 - Setembro

Abortos Falsos em Sistemas de Memória Transacional em Software

Daniel Nicácio*
Unicamp

Guido Araújo
Unicamp

Resumo

Memória transacional (MT) continua a ser a abordagem mais promissora para substituir locks em programação concorrente, mas sistemas de MT em software (MTS) ainda não possuem desempenho satisfatório quando comparados a implementações utilizando *locks* bem refinados. Sabe-se que a operação crítica em sistemas de MT é garantir a atomicidade e o isolamento das *threads* que estejam executando concorrentemente. Essa tarefa é conhecida como a validação dos conjuntos de leitura/escrita. Na tentativa de realizar esse processo o mais rápido possível, sistemas de MTS costumam utilizar uma tabela de posse para realizar a detecção de conflitos, mas essa abordagem gera ocorrência de falsos positivos, os quais resultam em abortos falsos. Este trabalho mostra o verdadeiro impacto dos abortos falsos e como sua relevância aumenta à medida que o número de *threads* concorrentes também aumenta, mostrando que este fator é essencial para sistemas de MTS. Nós propomos duas diferentes técnicas para evitar abortos falsos e conseqüentemente melhorar o desempenho de MTS. A primeira é uma lista de colisão anexada à já existente tabela *hash*. A segunda è uma técnica completamente nova e eficiente para detectar conflitos entre transações. Esta nova técnica faz uso de tecnologia SSE para detectar conflitos e substitui a normalmente usada tabela *hash*. Devido ao seu esquema de mapeamento de memória, esta técnica consegue evitar a ocorrência de falsos positivos. Com uma detecção de colisão rápida e livre de falsos conflitos, nós conseguimos significativa melhora de desempenho em alguns programas do benchmark STAMP. O *speedup* obtido foi de ate 1,63x. Nós também mostramos que os *speedups* ficam cada vez maiores com o aumento do número de *threads* paralelas. Por fim, baseado nos resultados obtidos, nós caracterizamos os programas que foram mais beneficiados pelas duas técnicas.

1 Introdução

Chips com múltiplos núcleos já são uma realidade sólida, estando presentes em computadores pessoais, servidores e até mesmo em sistemas embarcados. Apesar de chips com vários núcleos terem se espalhado rapidamente, a programação paralela não teve o mesmo avanço. O uso de *locks* tem sido a ferramenta mais utilizada para controlar acesso a dados compartilhados, mas enquanto *locks* pouco refinados são fáceis de gerenciar, eles limitam o paralelismo

*This work is sponsored by CAPES

e possuem desempenho ruim. Por outro lado, *locks* bem refinados possuem ótimo desempenho, mas desenvolver tais sistemas é uma tarefa muito complexa, normalmente realizada apenas por especialistas. Cientistas da computação estão à procura de uma ferramenta que combine a baixa complexidade de *locks* pouco refinados com a contenção mínima gerada por *locks* bem refinados. O paradigma de programação de memória transacional proposto por Herlihy and Moss [16] é a resposta mais promissora e aceita atualmente.

Memória Transacional (MT) permite que o programador identifique segmentos de código como transações sem se preocupar com as interações entre operações concorrentes. Transações são automaticamente executadas como uma operação atômica e em paralelo com outras transações enquanto não haver um conflito. Um conflito ocorre se duas transações em execução tentam acessar um mesmo dado e pelo menos uma delas tenta modificá-lo. Existem várias propostas de sistemas de MT: baseados em hardware (MTH) [1, 12, 16, 20], puramente em software (MTS) [6, 14, 15, 17, 18, 21, 23], e técnicas hibridas (MTHi) que combinam hardware e software [5, 8, 19].

MT alcança seu objetivo ao diminuir a dificuldade de programação paralela, mas seu desempenho ainda não é o adequado. Algumas propostas de MTH conseguem ter desempenho adequado, mas possuem outras limitações; por exemplo, limitar o tamanho máximo de uma transação. Uma *thread* executando em um sistema de MTS pode ser de duas a sete vezes mais lenta do que seu respectivo código seqüencial. Alguns sistemas de MTHi melhoram o desempenho de MTS em até duas vezes ao fazer o gerenciamento dos conjuntos de leitura/escrita em hardware [19, 22].

Sabe-se que o a validação dos conjuntos de leitura/escrita é uma tarefa crítica em sistemas de MTS. Ela consiste em (1) acompanhar todas operações de leitura e escrita de cada uma das transações em execução, isso normalmente é feito armazenando conjuntos de leitura/escrita; (2) verificar para cada elemento no conjunto de escrita se ele está presente em algum outro conjunto de leitura ou de escrita de outra transação; se isto acontecer, um conflito é detectado e uma das transações deve ser abortada. Na tentativa de fazer esse processo de verificação o mais rápido possível, sistemas de MTS normalmente utilizam técnicas susceptíveis a respostas com falsos positivos (por exemplo, mapeamento de memória através de uma tabela *hash*), gerando abortos desnecessários, os quais nós denominamos abortos falsos.

Este trabalho afirma que abortos falsos são uma ameaça para sistemas de MTS, e que à medida que o número de *threads* concorrentes aumenta esse problema se torna cada vez mais crítico. A probabilidade de um conflito acontecer é pelo menos proporcional ao números de *threads* paralelas em execução. Para apoiar esta afirmação, fizemos uma abordagem quantitativa a respeito dos efeitos dos abortos falsos, medindo a taxa de abortos falsos e o tempo gasto em transações abortadas devido a abortos falsos. Em seguida, propomos duas soluções para este problema: (1) uma lista de colisão para a normalmente usada tabela *hash* e (2) uma rápida técnica de busca em listas utilizando tecnologia SSE. A primeira simplesmente evita a ocorrência de falsos positivos na função *hash*. A busca rápida em lista usando SSE adota um inovador esquema de mapeamento de memória e usa instruções SSE (*streaming SIMD extension*) para responder rapidamente se um endereço de memória está em uso por outra transação. Ambas as técnicas são livres de falsos abortos e normalmente têm seu custo computacional compensado pelos falsos abortos evitados. Em alguns casos,

conseguimos um significativo ganho de desempenho.

Na próxima sessão comentamos os trabalhos relacionados a falsos abortos, falsos conflitos e técnicas para detecção de conflitos. Na Sessão 3 nós explicamos o processo de detecção de conflitos e apresentamos dados quantitativos para apoiar a necessidade de uma técnica de detecção de conflitos livre de falsos positivos. Sessão 4 mostra em detalhes as duas técnicas mencionadas anteriormente, explicando o seu funcionamento e porque elas tendem a se tornarem cada vez mais poderosas com o aumento do número de núcleos em um único chip. Na Sessão 5 nós apresentamos os resultados experimentais das duas técnicas desenvolvidas. Baseado nesses resultados, caracterizamos os programas que foram mais beneficiados pelas duas técnicas. Na Sessão 6 concluímos o quão importante é evitar falsos abortos em sistemas de MTS e como essa importância cresce com o aumento do número de *threads* concorrentes.

2 Trabalhos Relacionados

Zilles and Rajwar [26] possuem um trabalho similar relacionado à taxa de falsos conflitos. Eles demonstraram através de um modelo analítico e validado por dados estatísticos que uma tabela *hash* sem rótulos (i.e. lista de colisão) resulta em taxas de falso conflito induzidas por *alias*. Essa taxa cresce aproximadamente ao quadrado da concorrência e da quantidade de memória utilizada por transação. Sua demonstração foi realizada em um ambiente simulado. Eles sugerem que uma tabela *hash* com rótulos seja usada para a eliminação de alias. Nossos trabalhos se diferem em vários aspectos: (1) enquanto eles usaram modelos analíticos e estatísticos, nós fizemos experimentos práticos em *benchmarks* normalmente utilizados e usando o sistema de MTS mais usado atualmente; (2) eles focaram em sistemas híbridos de MT, e nós usamos modelos de MTS; (3) apesar de eles terem sugerido o uso de uma tabela *hash* com rótulos, eles não chegaram a apresentar resultados (benefícios de tempo e custo computacional) dessa proposta; (4) e finalmente, nós também implementamos uma técnica completamente nova para evitar falsos conflitos.

Xiaoqiang, Lin, e Lunguo [25] fizeram um trabalho sobre altas taxas de aborto de transações, fator que leva à degradação do desempenho de sistemas com muita contenção. Seus resultados experimentais (obtidos através do STMBench7 em uma máquina de oito núcleos) mostram que um sistema de MTS não possui boa escalabilidade. O número de transações abortadas por segundo cresce para cada núcleo adicionado. De acordo com os autores, *threads* concorrentes com alta contenção pioram o desempenho devido ao aumento do número de conflitos, o que gera desperdício de recursos com transações abortadas. Para superar este problema, eles tentaram reduzir substancialmente o número de conflitos entre transações, usando um modelo de consistência de memória causal com semânticas mais fracas. Nós concordamos plenamente em como o número de conflitos aumenta com a adição de novos núcleos, mas neste trabalho abordamos o problema por outro ângulo, reduzindo o número de conflitos gerados por falsos positivos.

Existem alguns trabalhos relacionados a técnicas de detecção de conflitos. Agrawal, Fineman e Sukha [3] desenvolveram uma técnica para detecção de conflitos chamada XConflict, utilizada em um sistema de MT projetado para transações paralelas aninhadas. O algoritmo divide a computação em *traces* dinamicamente, onde cada *trace* consiste em

operações de memória, e então o XConflict gerencia conflitos apenas entre *traces*. Infelizmente, a análise de desempenho do modelo não inclui checagem de conflitos com múltiplos leitores e possíveis abortos, o que poderia aumentar o tempo de execução significativamente.

Shriraman e Dwarkadas [24] analisaram a correlação entre resolução de conflitos e políticas de gerenciamento de contenção em sistemas de MT com suporte de hardware. Apesar da maneira como a detecção de conflitos é realizada não ter sido alterada, eles afirmam que políticas *lazy* provem melhor desempenho do que políticas *eager*. Eles também avaliaram um modo de detecção de conflitos misto, que detecta conflitos escrita-escrita de maneira *eager* e conflitos leitura-escrita de maneira *lazy*. De acordo com eles, este modo possui bom comprometimento entre flexibilidade e complexidade de implementação. Atoofian, Baniasadi e Coady [4] também discutem a eficiência das políticas *lazy* e *eager*, propondo uma política de validação que se adapta de acordo com as características do programa que esta sendo executado.

Gupta et al. [11] evidenciam o fato de que técnicas de detecção de conflitos em MT podem ser utilizadas para realizar eficiente detecção de corridas a memória compartilhada dinamicamente; aumentando ainda mais a importância de detecção de conflitos.

Esses trabalhos recentes indicam a grande importância dada à detecção de conflitos pelos cientistas da computação e sua importância para o desenvolvimento de sistemas de MT eficientes. Na iminência do lançamento de chips com 16 núcleos, isso se torna crucial para a escalabilidade de MT.

3 A questão dos abortos falsos

Esta Sessão provê dados relacionando o aumento de falsos abortos ao aumento do número de núcleos em um chip. Alem disso, mostramos o tempo desperdiçado em transações que terminaram em falsos abortos, deixando clara a importância de uma técnica de detecção de conflitos livre de falsos positivos.

O trabalho aqui apresentado foi implementado no sistema de MTS TL2 [10], considerado o estado da arte em implementações de MTS. TL2 armazena um conjunto-leitura e um conjunto-escrita para cada transação em execução. Nestes conjuntos são armazenados todos os endereços lidos e/ou escritos pela sua respectiva transação. De forma a sempre manter a memória consistente, e por conseqüência não produzir transações zumbis, o sistema checa se cada operação de leitura e escrita é valida; essa checagem é chamada de detecção de conflitos. Detecção de conflitos também é realizada imediatamente antes de uma transação ser finalizada (para assegurar que nada passou a ser inconsistente durante a execução da transação). Essa tarefa é feita da seguinte forma: (1) para cada operação de leitura, o sistema verifica o contador associado ao endereço sendo lido e compara este contador com o contador da própria transação; (2) antes da transação ser finalizada, o contador de cada leitura do conjunto-leitura é novamente verificado. Se qualquer um deles apresentar um estado inválido, isto significa que uma operação de escrita foi executada, alterando o contador do endereço em questão. Este caso caracteriza um conflito detectado e, por conseqüência, a transação precisa ser abortada.

Na tentativa de deixar esse processo o mais rápido possível, TL2 usa uma função *hash*

para mapear os endereços de memória para seus respectivos contadores. Mas note que, quando uma função *hash* é utilizada, dois diferentes endereços podem ser mapeados para uma mesma posição da tabela *hash*. Suponha que a função *hash* mapeie os endereços X e Y para a mesma entrada da tabela *hash*. Como conseqüência, se a transação T1 lê o endereço X, em seguida a transação T2 escreve no endereço Y; quando T1 tentar finalizar, ela detectara um conflito no endereço X e será abortada. Mas este conflito é um falso conflito, já que X e Y são duas posições de memória completamente distintas. Quando uma transação é abortada devido a um falso conflito, nós chamamos isso de um falso aborto. A maioria dos sistemas de MTS [2, 13, 21] usam técnicas suscetíveis a falsos conflitos.

Nós realizamos modificações na TL2 para que ela pudesse medir a quantidade de tempo em casa transação. Em seguida, classificamos as transações em três grupos: finalizadas, que terminaram com sucesso; abortos verdadeiros, que abortaram devido a um conflito verdadeiro; e abortos falsos, que abortaram devido a um conflito falso. A quantidade de ocorrências e o tempo gasto em cada um dos tipos foram medidos.

O gráfico da Figura 1 mostra, para sete programas do *benchmark* STAMP, o percentual de transações iniciadas que terminaram em um aborto falso e a porcentagem de tempo gasto neste tipo de transação. Executando com oito *threads* paralelas, cerca de um quarto das transações abortadas no Kmeans High (26%) e Kmeans Low (18%) terminaram em abortos falsos. Este número continua alto nos programas Labyrinth (8%) e Vacation High (6%). O restante dos programas teve, em media, 0,6% de suas transações abortadas devido a falsos conflitos. Todos os programas apresentaram um padrão crescente à medida que o número de *threads* paralelas aumentou.

O tempo mostrado neste gráfico corresponde exclusivamente ao tempo gasto em código transacional (incluindo o custo de gerenciamento do sistema de MTS), excluindo tempo gasto em código não transacional. Executando com oito *threads* paralelas, Labyrinth gastou 22% de seu tempo em transações que terminaram em um aborto falso; Kmeans High gastou 11%; Genome, Bayes e Vacation High gastaram entre 5% e 10%; o restante dos programas gastaram 1,7%. Novamente, notamos um padrão crescente nas curvas do gráfico.

O programa SSCA2 não apresentou abortos de quaisquer tipos, portanto ele não foi apresentado no gráfico.

O gráfico deixa claro que abortos falsos existem em uma quantidade substancial em alguns programas. Note que, em todos os programas, a curva continua subindo à medida que o número de núcleos aumenta. Como já mencionado, o tempo gasto em transações que terminaram em abortos falsos foi bem alto no programa Labyrinth; se evitarmos abortos falsos neste programa, provavelmente conseguiríamos um *speedup* em torno de 22%. Além disso, quando um aborto falso é eliminado, uma série de conseqüências positivas é gerada: (1) o tempo gasto na transação não é desperdiçado, (2) a computação desperdiçada nao precisa ser refeita, (3) o processo de *roll-back* para restaurar o estado prévio do processador é eliminado, e (4) à medida que diminuímos o tempo dentro de transações, a chance de conflitos reais ocorrerem também diminui.

É importante notar que o número de transações que terminam em abortos falsos não é proporcional ao tempo gasto no mesmo tipo de transações. A razão principal disso é o tamanho das transações em cada programa. O tamanho da transação tem influência direta no tempo necessário para fazer o *roll-back*, o que aumenta o custo de um aborto.

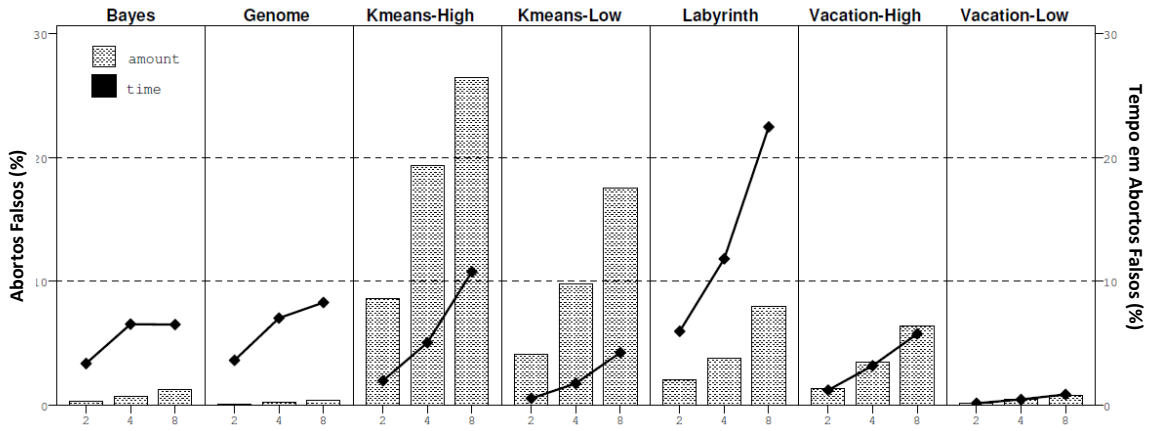

Figura 1: O percentual de transações que terminaram em abortos falsos e o percentual de tempo gasto nesse tipo de transação para sete programas do *benchmark* STAMP

Por exemplo, Labyrinth possui transações longas, enquanto Kmeans High possui transações bem curtas. Isso explica porque o Kmenas High, apesar do alto número de transações que terminam em abortos falsos, não tem uma alta porcentagem de tempo gasto nessas transações. O número de transação terminadas em abortos falsos e o tempo gasto nelas será discutido mais a fundo na Sessão 5.

4 Detecção de Conflitos Livre de Falsos Abortos

Na última sessão percebemos que uma certa quantidade de tempo em sistemas de MTS é desperdiçado devido a abortos falsos. Esse fato cria uma oportunidade de otimização pra sistemas de MTS. O desempenho de MTS pode ser melhorado em alguns programas se eliminarmos os abortos falsos. Para atingir esse objetivo, os abortos falsos precisam ser evitados, mas a um custo computacional bem baixo para que possa valer a pena. Nas próximas duas sub-sessões apresentaremos duas diferentes maneiras de se fazer isso. A primeira foca da desambiguação de conflitos na tabela *hash*, e a segunda é um novo esquema de mapeamento de memória livre de ambigüidades.

4.1 Lista de Colisão para Tabela *Hash*

Zilles e Rajwar [26] propuseram uma tabela *hash* encadeada pra eliminar falsos conflitos, mas eles não chegaram a implementá-la para avaliar seus benéficos, e eles focaram em sistemas híbridos. Nossa implementação segue a mesma idéia proposta por eles, mas algumas modificações se fizeram necessárias devido a restrições técnicas e ao fato de tentarmos modificar o algoritmo da TL2 o mínimo possível.

TL2 utiliza uma tabela *hash* de 4MB capaz de armazenar 1000 entradas. Cada entrada é uma palavra de 32 bits. Dependendo do valor do bit menos significativo, os 30 bits

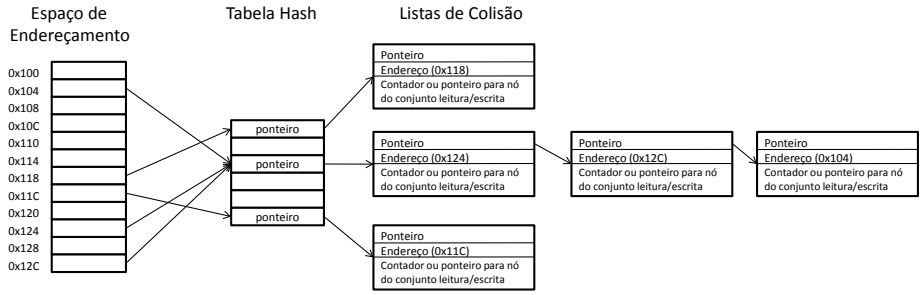

Figura 2: A lista de colisão para a tabela *hash* em um sistema de MTS

mais significativos armazenam um contador ou um ponteiro para um elemento do conjunto leitura/escrita de uma transação.

A principio, pensamos em usar o segundo bit menos significativo para guardar a informação se aquela entrada tem uma colisão ou não (um bit de colisão), dessa maneira, um nível de indireção seria necessário apenas nas entradas onde havia uma colisão. Infelizmente, neste contexto, essa abordagem parece não ser possível. Se existe um contador na entrada em questão, não é possível determinar o endereço associado a esse contador, e portanto, não há informação disponível para definir o bit de colisão corretamente.

A implementação usada neste trabalho guarda um ponteiro para uma lista de colisão em cada entrada da tabela *hash*. Um nó da lista de colisão é uma estrutura contendo: (1) um ponteiro para um possível nó seguinte, (2) o endereço de memória associado ao nó e (3) a palavra de 32 bits que estaria presente na tabela *hash* original. O custo adicional desta implementação é um nível de indireção adicionado e a memória gasta para construir os nós da lista de colisão (cada nó requer 12 bytes). A Figura 2 mostra a idéia da lista de colisão.

Os resultados obtidos com essa técnica será mostrado e analisado na Sessão 5.

4.2 Técnica rápida de busca em listas usando SSE

A tarefa de detecção de conflitos em sistemas de MT é basicamente o mesmo problema de análise de *alias*. Dada duas operações, queremos saber se elas operam na mesma posição de memória ou não. Esse é um problema de suma importância em várias áreas da ciência da computação, principalmente em compiladores e projetos de hardware. Com isso em mente, apresentamos uma técnica inovadora para responder rapidamente à questão do *alias* sem falsos positivos. Aqui, ela será usada para resolver detecção de conflitos em sistema MTS, mas ela pode vir a ser usada em outros campos da computação.

Basicamente, dado um endereço de memória X , precisamos responder se X esta em uso por outra *thread*/transação. Em outras palavras, precisamos procurar X em uma lista e verificar seu atual dono.

Nosso algoritmo divide X em duas partes: highAddr e lowAddr. O highAddr corresponde aos oito bits mais significativos de X; e lowAddr aos 24 bits menos significativos de X. Dito isso, o algoritmo trabalha com dois tipos de listas: uma lista de highAddr e uma lista de lowAddr para cada highAddr existente. Os elementos da lista highAddr são agrupa-

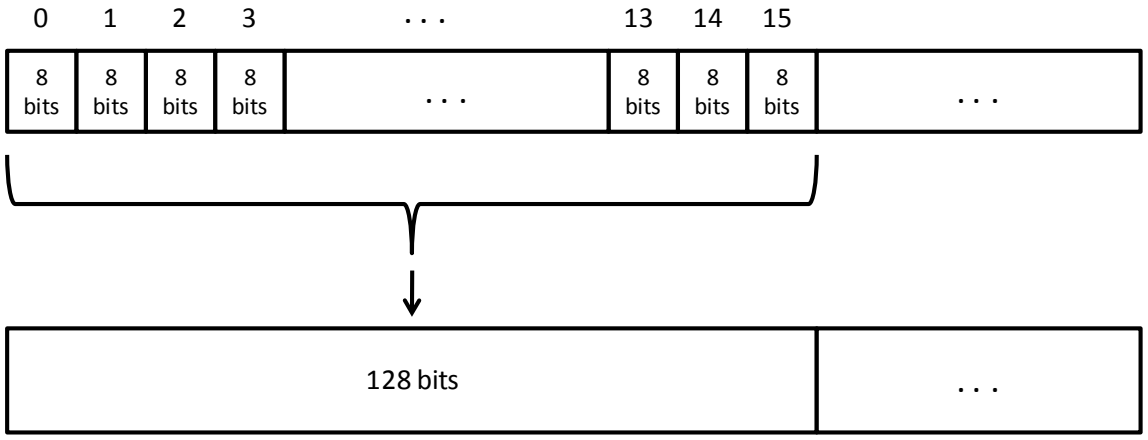

Figura 3: Os elementos da lista highAddr agrupados em elementos de 128 bits cada

dos em conjuntos de 16 elementos cada, formando elementos de 128 bits como mostrado na Figura 3. Cada elemento da lista highAddr aponta para sua correspondente lista lowAddr, cada lista lowAddr possui 2^{22} entradas e cada entrada guarda o contador ou o ponteiro para um elemento do conjunto de leitura/escrita usado pelo TL2. A lista lowAddr possui apenas 2^{22} elementos, e não 2^{24}, porque a API do TL2 define a granularidade mínima dos dados como uma palavra. O esquema completo é mostrado na Figura 4. Dessa maneira, cada endereço de memória é mapeado para uma única entrada, eliminando a possibilidade de falsos positivos.

Para procurar se um dado endereço X está presente na lista, os seguintes passos devem ser tomados: (1) X é dividido em duas partes (X.high e X.low) como descrito acima; (2) em seguida, replicamos X.high 16 vezes para formar um elemento de 128 bit chamado pattern. O processo de replicação é feito com poucas instruções. Primeiro, definimos as quatro primeiras posições de um vetor com o valor de highAddr, então, usando o vetor inteiro como um elemento de 128 bits, usamos a instrução MOVDDUP para replicas os 64 bits menos significantes nos 64 bits mais significativos, o passo final é usar a instrução MOVSLDUP para replicar os bits 0-31 e 64-95 nos bits 32-63 e 96-127. Ao final desse processo, o elemento de 128 bits será composto por 16 highAddrs. Figura 5 ilustra o processo de replicação; (3) em seguida comparamos cada elemento de 128 bits da lista highAddr com o pattern. A comparação é feita pela instrução PCMPEQB (*Compare Packed Data for Equal*), ela compara byte a byte os valores de dois operandos. Se um par de elementos for igual, o correspondente byte no operando destino tem todos os seus bits definidos como 1s, se forem diferente, os bits são definidos como 0s. Esse processo é graficamente descrito na Figura 6.

A divisão do endereço em uma parte de 8 bits e outra de 24 bits permite a comparação de até 16 highAddr com uma única instrução e mantém a lista highAddr com poucos elementos, já que o principio da localidade nos diz que a parte alta do endereço deve alterar poucas vezes durante os acessos à memória de um programa. Outra possibilidade seria dividir o endereço pela metade, com duas partes de 16 bits, mas o número de comparações por instrução cairia pela metade e a lista highAddr pode vir a ficar muito extensa. Para realizar

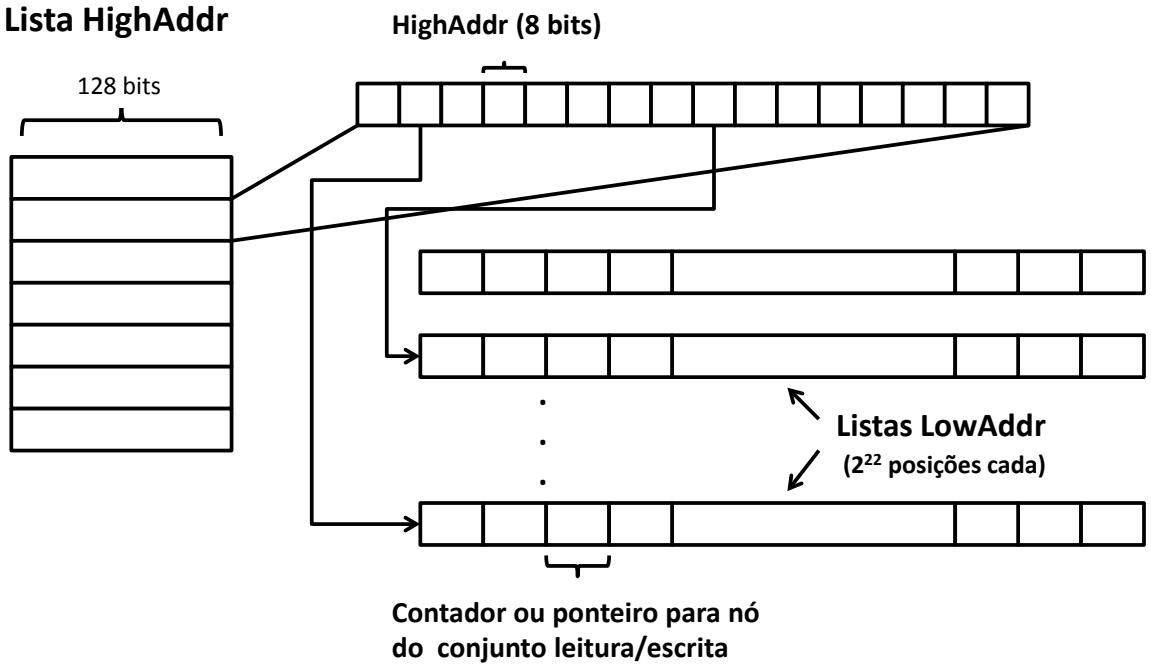

Figura 4: A organização e os papéis das listas highAddr e lowAddr

esse esquema, algumas otimizações seriam necessárias, elas serão discutidas na Sessão 4.2.1.

O resultado da comparação é então comprimido, criando uma máscara de 32 bits, onde apenas um único bit estará definido como 1. Isso também é feito com apenas uma instrução chamada PMOVMSKB (*Move Byte Mask*), ela cria uma máscara utilizando o bit mais significativo de cada byte do operando fonte (128 bits) e o resultado é um palavras de 16 bits. Novamente, a funcionalidade da instrução é mostrada na Figura 7.

A instrução BSR (*Bit Scan Reverse*) varre a máscara a partir do bit mais significativo até o bit menos significativo à procura de um bit definido como 1 e então retorna sua posição. Por exemplo, na Figura 7 ela retorna 11 (o 12º bit). Essa posição nos diz qual lista lowAddr corresponde ao endereço X que nós estamos procurando. Se mais de uma comparação com o pattern for necessária, nós adicionamos 16 a este resultado para cada comparação adicional realizada. Apenas uma comparação costuma ser necessária, porque, como o principio da localidade afirma [9], referencias à memória tendem a acessar apenas uma pequena região da memória total. Portanto, quando dividimos o endereço de memória em duas partes, esperamos que o highAddr permaneça o mesmo durante boa parte do programa, fazendo com que a lista highAddr permaneça bem pequena. Em nosso experimentos, ela normalmente teve apenas 4 a 6 elementos (8 bits cada).

Finalmente, X.low indica qual elemento da lista lowAddr deve ser verificado. O lowAddr[X.low] contém a informação que era previamente guardada na tabela *hash*: um contador ou um ponteiro para o conjunto leitura/escrita.

Para resumir as etapas da técnica, seu algoritmo é apresentado na Figura 8. Primeiro, dividimos o endereço (addr) em highAddr (8 bits mais significativos de addr) e lowAddr

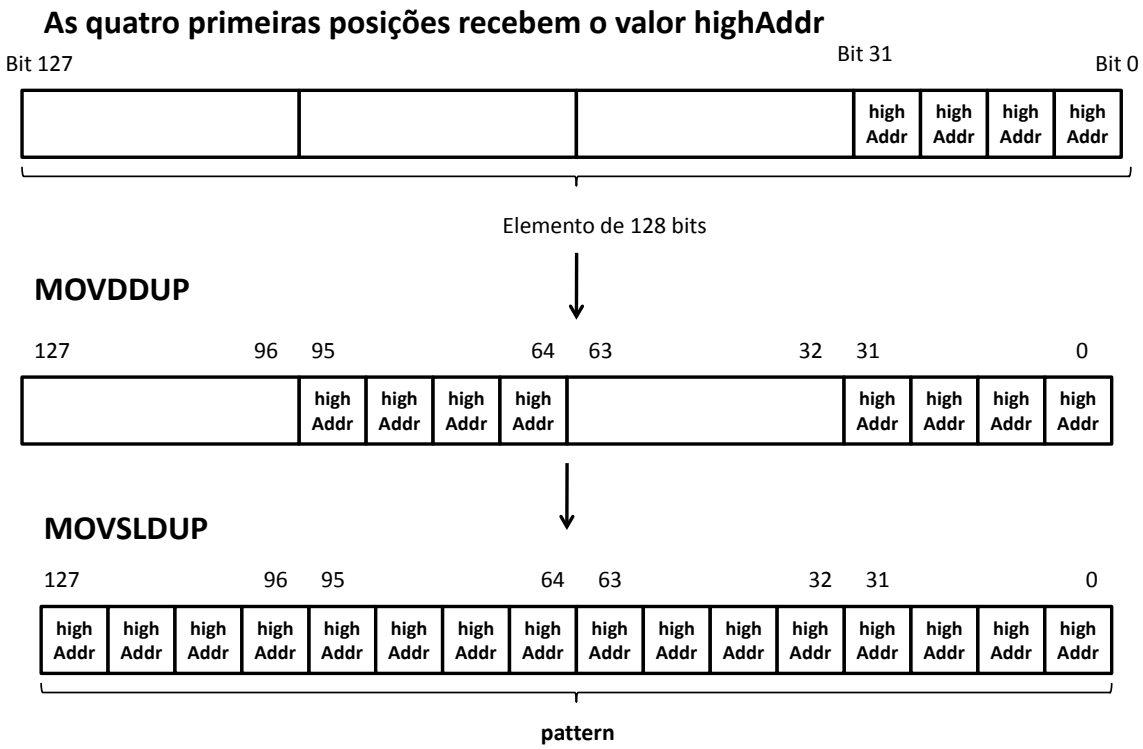

Figura 5: O processo de criação da variável pattern

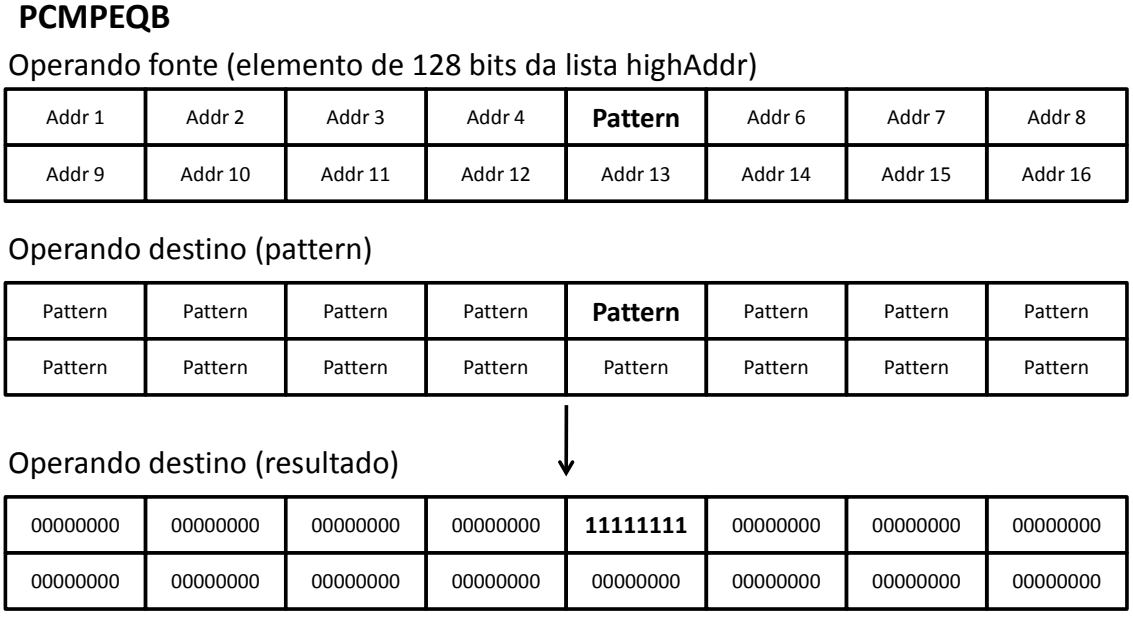

Figura 6: A instrução PCMPEQB e como ela funciona

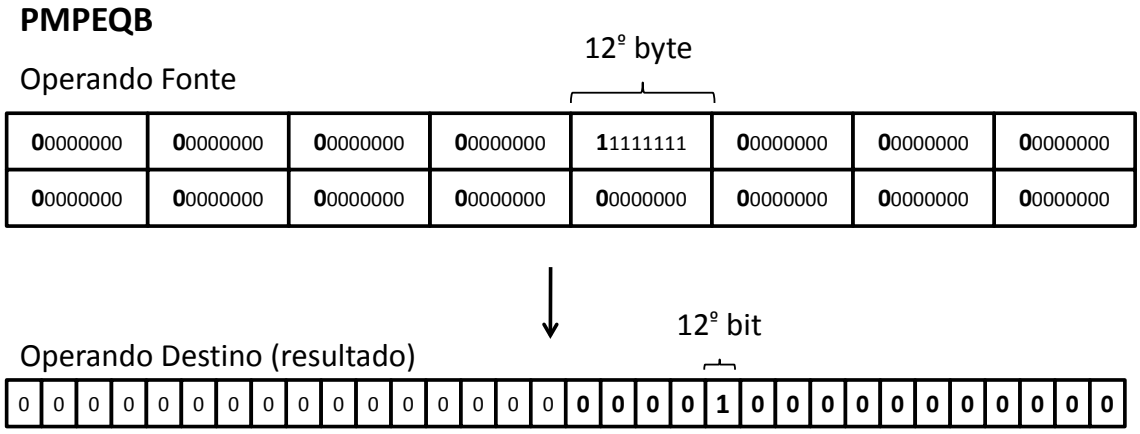

Figura 7: A instrução PMOVMSKB e como ela funciona

(24 bits menos significativos de addr). Segundo, o elemento de 128 bits chamado pattern é criado usando um pequeno laço seguido por duas instruções de replicação em palavras de 128 bits. Terceiro, pattern é comparado com cada elemento de 128 bits da lista highAddr. Essa comparação é feita pela instrução pcmpeqb, que usa palavras de 128 bits como parâmetro e compara 16 bytes em paralelo. O resultado da comparação é compactado em uma palavra de 23 bits pela instrução pmovmskp. A comparação continua até que ela retorne verdadeiro ou que a lista highAddr chegue ao seu final. Quarto, verifica se o highAddr foi encontrado, se este não for o caso, ele é inserido na lista highAddr e sua posição de memória é retornada. Caso contrário, sua posição de memória é calculada através da quantidade de comparações realizadas e do valor de lowAddr, a posição de memória é então retornada. A posição de memória aqui mencionada contem a informação requisitada pela TL2.

Como foi visto, com poucas instruções conseguimos procurar por um endereço de memória e indicar seu dono. A principal vantagem dessa técnica é sua escalabilidade; não importa quantas transações estão trabalhando em paralelo, ela terá um tempo constante de resposta. O único fator que influencia é a quantidade de dados a ser usada pela aplicação, mas mesmo assim, o tempo será afetado apenas se a quantidade de dados ultrapassar 256MB (a capacidade de representação de 16 elementos highAddr). Essa característica faz com que a técnica de busca rápida em listas usando SSE seja uma ferramenta muito promissora para sistemas de alta contenção.

Experimentos mostraram que, enquanto essa técnica é capaz de verificar a igualdade de 16 elementos, uma simples busca em uma lista ligada é capaz de verificar apenas três elementos na mesma quantidade de tempo. Esse resultado mostra que a técnica de busca rápida em listas usando SSE certamente irá superar a lista de colisão em uma tabela *hash* se o número de colisões exceder três.

```
procedure SearchAddress(addr)

    highAddr ← 8MSBof(addr);
    lowAddr ← 24LSBof(addr);
    for i ← 0 until 3 do
        pattern[i] ← highAddr;
    pattern ← movddup(pattern);
    pattern ← movddup(pattern);
    x ← 0;
    times ← numberOfElements(highAddrList);
    place ← 0;
    while x <= times do
        result128bits ← pcmpeqb(highAddrList[x], pattern);
        place ← pmovmskb(result128bits)
        if (place)
            break;
        x++;
    if (x > times)
        position ← highAddrList.insert(highAddr);
        return position.lowAddrList[lowAddr];
    index ← bsr(place);
    index ← (16*x) + (index);
    return index.lowAddrList[lowAddr];
```

Figura 8: O algoritmo da busca rápida em lista usando SSE

4.2.1 Questões e Soluções

Uma questão que vem à tona é a portabilidade da técnica. Algumas instruções usadas (pcmpeqb and pmovmskb), só estão disponíveis em processadores Intel. Apesar de essas instruções serem específicas do conjunto de instruções da arquitetura Intel, vários processadores suportam mecanismos de múltiplos dados em uma única instrução (SIMD). As funcionalidades das instruções usadas neste trabalho já podem estar presentes em outros processadores, e se ainda não estão, elas poderiam facilmente serem incluídas no suporte à MT de futuros processadores.

A quantidade de memória utilizada pela técnica também pode ser um problema. Cada lista lowAddr possui 2^{22} elementos, cada elemento é uma palavra de 32 bits. Sendo assim, cada lista lowAddr utiliza 16MB de memória. Na média, cada programa do STAMP utiliza de 4 a 6 listas lowAddr, Fazendo com que a memória total necessária seja de 64MB a 96MB. Com essa quantidade de memória sendo constantemente utilizada em um programa, é possível que a memória cache do processador esteja sempre cheia de listas lowAddr, deixando nenhum espaço para outros dados do programa. Acreditamos que boa parte do custo computacional de nossa técnica provém desse fato.

Já temos em mente uma otimização promissora para resolver esta questão. É de conhecimento comum que, enquanto alguns dados são usados durante toda a execução de um programa, a outra grande maioria é usada apenas em uma fração de tempo. Portanto, uma variável alocada em uma lista lowAddr não precisa permanecer lá até que o programa termine. Uma heurística pode ser usada para verificar se a variável pode ser removida desta lista. Em outras palavras, é possível desenvolver um coletor de lixo. As listas lowAddr devem guardar apenas as variáveis que estão sendo usadas por alguma transação, se nenhuma transação estiver utilizando a variável, ela pode ser removida pelo coletor de lixo. Dessa forma, se uma transação trabalha em Z bytes, a memória máxima utilizada seria $Z * numeroDeThreadsCorrentes$ bytes. Tal número pode ser facilmente armazenado em uma única lista lowAddr. É até possível que a lista lowAddr fique com espaços vazios. Neste caso, o endereço X utilizado na explicação acima poderia ser divido pela metade, duas partes de 16 bits, ao invés de uma parte de 8 bits e outra de 24 bits. Dessa maneira, a lista lowAddr teria apenas 2^{14} posições, usando apenas 64KB de memória. Essa otimização esta em desenvolvimento e seus resultados devem ser apresentados em um futuro próximo.

5 Resultados Experimentais

Nesta sessão apresentamos os *speedups* obtidos com as duas técnicas: lista de colisão para a tabela *hash* e a busca rápida em listas usando SSE. Os resultados estão divididos em dois grupos: o primeiro utiliza a política de detecção *eager*, e o segundo a política *lazy*. Baseado nesses resultados, também caracterizamos os programas que foram mais beneficiados pelas técnicas. Nossos experimentos e resultados foram medidos em uma máquina Intel dual-Xeon (oito núcleos de 2,0 GHz), 1MB L2 *cache* para cada núcleo. Todos os resultados aqui apresentados correspondem à média de 20 execuções e seus desvios padrão foram medidos e analisados.

Essa avaliação foi feita com os programas do *benchmark* STAMP. Os oito programas

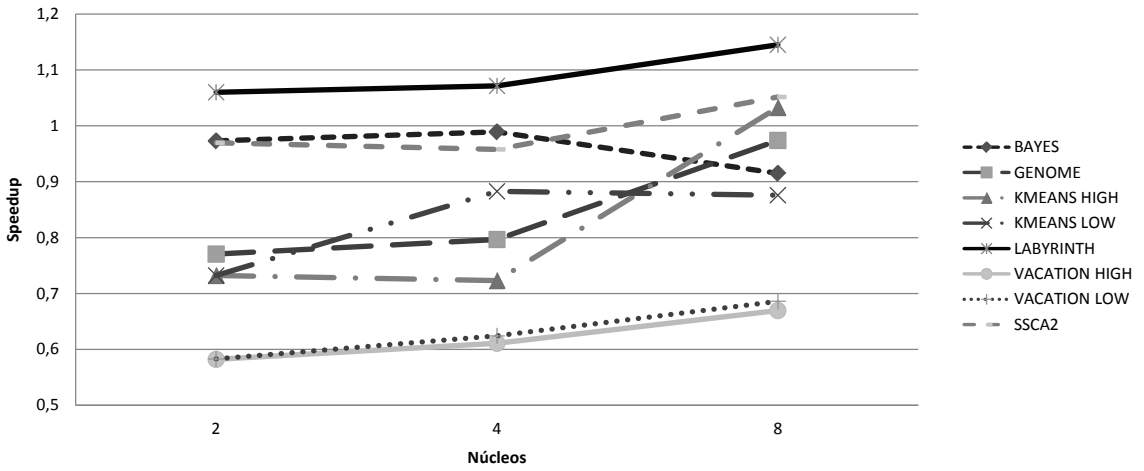

Figura 9: Os *speedups* da busca rápida em listas usando SSE no *benchmark* STAMP, usando a política de detecção *eager*

incluídos no STAMP são um seqüenciador de genes (Genome), uma rede de aprendizado bayesiana (Bayes), um algoritmo de detecção de intrusos em uma rede (Intruder), um algoritmo de cluster para k-medias (KMeans), um algoritmo para rotas em um labirinto (Labyrinth), um conjuntos de grafos de kernel (SSCA2), um sistema de reservas servidor-cliente (Vacation) e um algoritmo de refinamento de malha Delaunay (Yada). Desses, Yada e Intruder não executaram corretamente com o pacote da TL2 disponível com os mesmos. Portanto, não disponibilizamos os resultados destes dois programas. Para o restante dos programas relatamos o *speedup* obtido.

5.1 Detecção de Conflito *Eager*

As políticas de detecção de conflito diferenciam no momento em que os conjunto de leitura e escrita são validados. Em sistemas *eager* (pessimistas), o sistema de MT detecta e resolve conflitos toda vez que a transação acessa uma posição de memória. Em sistemas *lazy* (otimistas), o conflito só é analisado quando a transação tenta finalizar com sucesso.

Nosso primeiro gráfico, na Figura 9, mostra os *speedups* da busca rápida em listas usando SSE no *benchmark* STAMP usando a política de detecção *eager*. A maioria dos programas começa apresentando *slowdown* quando executando apenas duas *threads* concorrentes, mas quando o número de *threads* aumenta para quatro e oito, o *slowdown* diminui em alguns casos (Labyrinth, SSCA2 e Kmenas High) o *speedup* fica evidente. O *speedup* máximo alcançado neste conjunto foi 1,15x no Labyrinth. A principal informação a ser extraída desse resultado é a clara tendência de *speedup* em praticamente todos os programas, fazendo com que futuros testes em máquinas de 16 núcleos sejam muito promissores. A única exceção é o Bayes, mas seu algoritmo apresenta um setor aleatório, o que faz com que seu desvio padrão seja elevado.

O gráfico na Figura 10 mostra os *speedups* da lista de colisão da tabela *hash* no *benchmark*

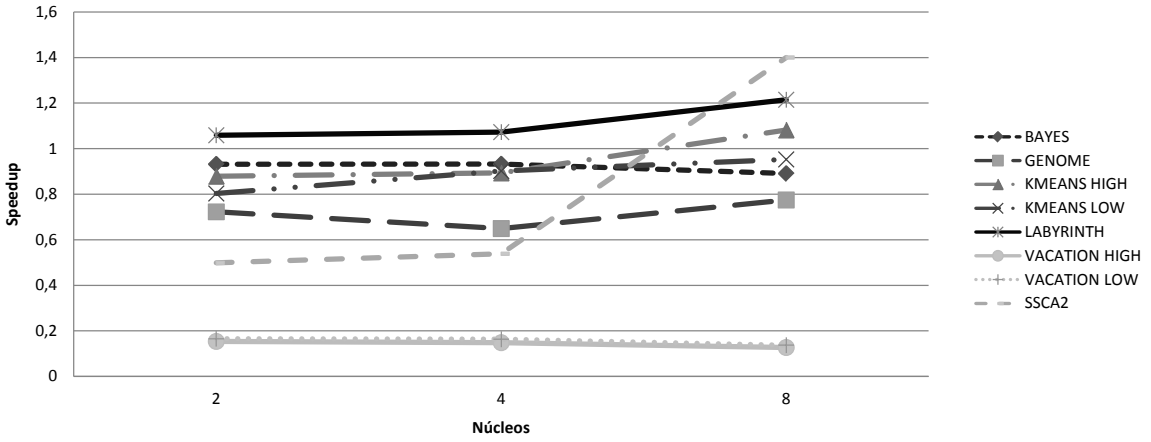

Figura 10: Os *speedups* da lista de colisão anexada à tabela *hash* no *benchmark* STAMP, usando a política de detecção *eager*

STAMP usando a política de detecção de conflitos *eager*. Novamente, com apenas dois núcleos, a maioria dos programas apresentou *slowdown*; assim como na busca rápida em lista usando SSE, esse *slowdown* é proveniente do custo computacional inserido pelas técnicas. A tendência de *speedup* também esta presente aqui, à medida que o número de núcleos em execução aumenta, o número de conflitos falsos também aumenta. SSCA2, Labyrinth e Kmeans High apresentaram *speedup*, o *speedup* máximo obtido foi 1,29x no SSCA2, seguido por 1,21x no Labyrinth e 1,08x no Kmeans High. Vacation High e Vacation Low tiveram um forte *slowdown*; eles ficaram até seis vezes mais lento que o código original. Isso se deve ao baixo número de conflitos (quase nulo) nesses programas. Sem espaço para melhoras, o custo computacional da técnica piora o desempenho dos programas.

Comparando as duas técnicas, notamos que a lista de colisão é mais leve que a busca rápida em lista usando SSE, tendo melhores resultados com poucos núcleos. Mas a escalabilidade da busca rápida em lista usando SSE é uma vantagem sobre a lista de colisão em sistemas com alta contenção.

5.2 Detecção de Conflito *Lazy*

O gráfico na Figura 11 mostra os *speedups* da busca rápida em listas usando SSE no *benchmark* STAMP usando a política de detecção *lazy*. Seguindo o mesmo padrão da política de detecção *eager*, apenas alguns programas começaram com *speedups* quando usando apenas duas *threads*. Quando o número de *threads* sobe de para 4 e depois para 8, o desempenho melhora na maioria dos programas. O *speedup* máximo alcançado nesse conjunto foi 1,33x no Labyrinth, seguido de 1,07 no Bayes. Vacation High e Vacation Low permaneceram com um valor constante, deixando clara a baixa taxa de conflitos em ambos os programas.

O gráfico na Figura 12 mostra os *speedups* da lista de colisão da tabela *hash* no *benchmark* STAMP usando a política de detecção de conflitos *lazy*. Vacation High e Vacation Low apresentando forte *slowdown* com a técnica da lista de colisão. Os melhores *speedups* foram

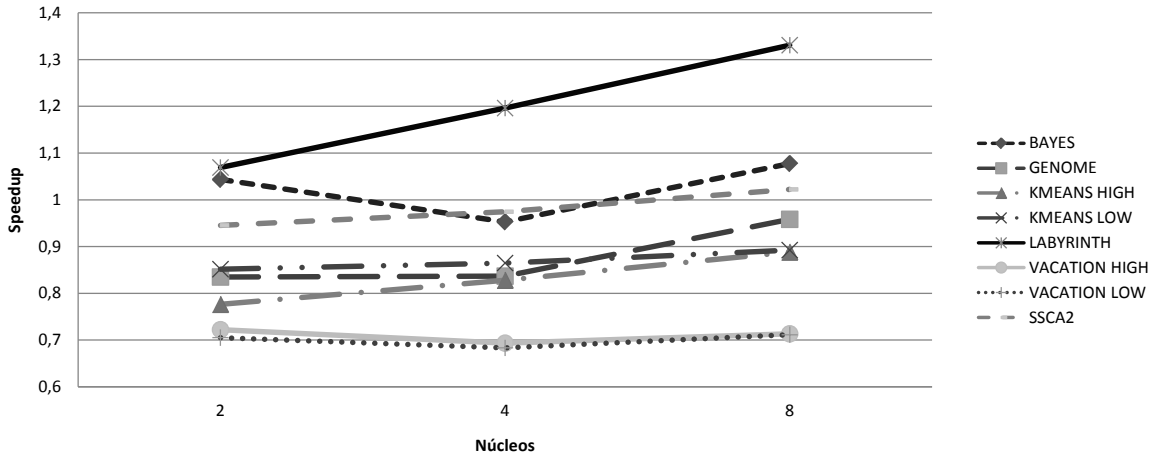

Figura 11: Os *speedups* da busca rápida em listas usando SSE no *benchmark* STAMP, usando a política de detecção *lazy*

alcançados no Labyrinth (1,63x) e SSCA2 (1,33x). Esse é o único gráfico onde a algumas curvas apresentam uma tendência de *slowdown* (Kmeans High, Kmeans Low e Bayes). Bayes provavelmente apresenta esse resultado devido a seu comportamento aleatório. Ambos Kmeans High e Kmeans Low são afetados pela política *lazy* e não tem a mesma tendência de *speedup* apresentada na política *eager*.

Não há diferença substancial entre os resultados quando comparando as duas políticas de detecção de conflito. A tendência de spedup é mais clara na política *eager*. A política *lazy* obteve os *speedups* mais altos. O resultado geral mostra que quanto maior é a contenção do sistema, mais beneficiado ele é por uma técnica de detecção de conflitos livre de falsos conflitos. Se a tendência de *speedup* mantiver este rítmo, o q provavelmente acontecerá, a maioria dos programas apresentará *speedups* quando executarem 16 *threads* em paralelo.

5.3 Caracterização dos Programas

Na Sessão 3 vimos que programas desperdiçam tempo em transações que são abortadas devido a falsos conflitos, e resultados obtidos através das duas técnicas foram de acordo com o esperado. Os programas onde os *speedups* foram maiores são os mesmos programas que apresentavam a maior quantidade de tempo desperdiçado: Labyrinth, Bayes e Genome.

Agora iremos identificar o que esses programas têm em comum para apresentar esse comportamento. Minh et al. [7] forneceu uma caracterização detalhada das aplicações incluídas no *benchmark* STAMP. Eles usaram um ambiente simulado com 16 núcleos para medir informações úteis dos programas. A análise deles se refere às entradas de dado 'básica' e '+'. Apesar do trabalho aqui desenvolvido utilizar a entrada de dados '++' e ter sido executado em um ambiente real de oito núcleos, pelo menos dois de seus resultados continuam sendo válidos para nossos propósitos: número de instruções por transação (média) e tempo dentro de transações (%). Esses valores, juntamente com outros resultados são mostrados na tabela da Figura 13. Essa tabela será detalhada posteriormente.

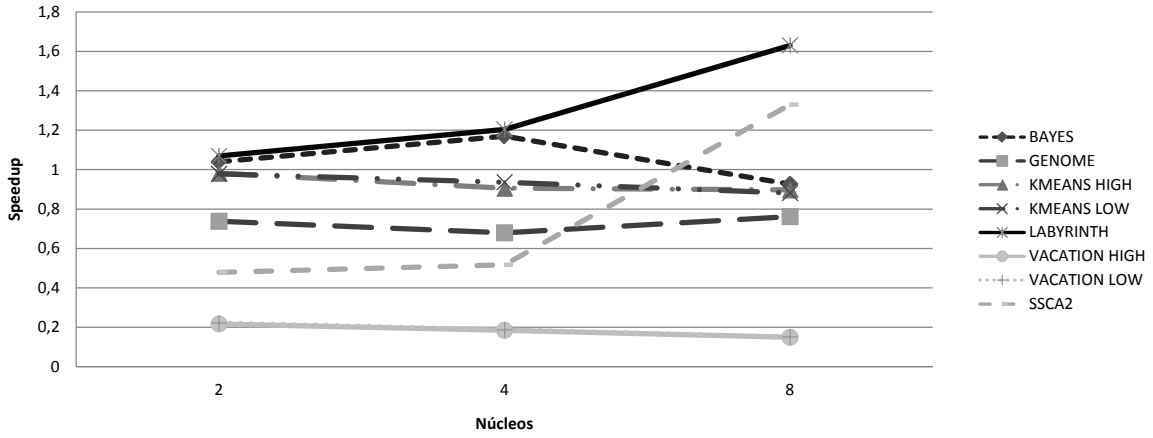

Figura 12: Os *speedups* da lista de colisão anexada à tabela *hash* no *benchmark* STAMP, usando a política de detecção *lazy*

A primeira informação que vamos olhar é o tempo gasto em transações, ele corresponde à porcentagem de tempo total do programa gasto em código transacional. Os benefícios de nossas técnicas são diretamente limitados por essa informação. Labyrinth, Bayes, e Genome permanecem em código transacional a maior parte do tempo. Kmeans High, que apresentou um alto tempo dentro de transações abortadas, permanece apenas 6% do seu tempo em código transacional, fazendo com que seus benefícios sejam cortados pelo mesmo fator. Outra característica é o tamanho da transação em cada programa, medido pelo número de instruções por transação (média); Labyrinth, Bayes, e Genome possuem mais de 1500 instruções por transação. A última característica é a taxa de abortos, o número aqui usado foi medido juntamente com as informações apresentadas na Sessão 3. Kmeans High, Kmeans Low e Labyrinth apresentam altas taxas de aborto. Vacation Low é um exemplo de programa que possui transações longas e alto percentual de tempo gasto em transações, mas a sua taxa de abortos é praticamente nula, fazendo com que ele seja um candidato ruim para os nosso propósitos.

Esses números indicam que podemos gerar uma heurística para ligar/desligar as técnicas baseadas em algumas características do programa alvo. Para ilustrar isso, sugerimos uma fórmula que pode indicar qual seria os benefícios de uma técnica livre de falsos conflitos em determinado programa. Para os números mostrados na tabela da Figura 13, a fórmula usada é mostrada abaixo

$$\sqrt{(tamanhoDaTransacao) * (tempoGastoEmTransacoes) * (taxaDeAbortos)}$$

Essa fórmula foi criada baseada em testes empíricos; fórmulas mais precisas são possíveis.

Na Figura 13, apresentamos para cada programa: o tamanho da transação, o percentual de tempo gasto em código transacional, a taxa de abortos, o resultado da fórmula, o qual chamamos de FAI (*False Abort Index*) e o *speedup* alcançado com a técnica de busca rápida em lista usando SSE com a política *eager*. Labyrinth apresenta o maior FAI (3247), sendo

Aplicação	Instruções por Transação	Tempo em Transações (%)	Taxa de Abortos (%)	FAI	Speedup
BAYES	57130	0.83	4.36	454.95	7%
GENOME	1709	0.97	0.87	37.88	-4%
KMEANS HIGH	153	0.06	45.89	20.53	-13%
KMEANS LOW	153	0.03	24.36	10.58	-12%
LABYRINTH	687809	1	15.34	3247.91	25%
SSCA2	50	0.16	0.00	0.04	2%
VACATION HIGH	4193	0.92	6.55	158.90	-40%
VACATION LOW	3161	0.92	0.85	49.75	-41%

Figura 13: A caracterização, o índice de falsos abortos e o *speedup* para cada programa do *benchmark* STAMP

também o maior *speedup* (1,25x). Bayes tem o FAI igual a 454 e seu *speedup* é de 1,07x. Programas que apresentaram *slowdown* tiveram o FAI abaixo de 50. SSCA2 é uma exceção, sua taxa de aborts próxima de zero vai contra o *speedup* obtido, mas seu tempo gasto em código transacional é pequeno, o que indica que a técnica de busca rápida em listas usando SSE tem pouco impacto sobre essa aplicação.

Não é possível saber dinamicamente o percentual de tempo gasto dentro de transações, mas o tamanho das transações e a atual taxa de abortos podem ser medidos dinamicamente. Portanto, é possível criar uma otimização para sistemas de MTS que ligue/desligue essas técnicas baseando em informações colhidas em tempo de execução.

6 Conclusao

Com o trabalho realizado até o momento, ficou claro que abortos falsos é um problema para sistemas MTS; nos fazendo afirmar que ele é uma séria ameaça a sistemas com alta contenção. Nós medimos o número e o tempo gasto em transações que terminaram em um aborto falso. Até 26% das transações iniciadas terminam sendo abortadas devido à um falso conflito. O percentual de tempo gasto nesse tipo de transação atingiu 22%.

Para superar este obstáculo, nós propusemos duas soluções: ma lista de colisão anexada à normalmente usada tabela *hash* e uma inovadora técnica para substituir a tabela *hash*. Essa

técnica usa um esquema de mapeamento de memória totalmente associativo que faz buscas rápidas de endereços em listas usando instruções SIMD; essa técnica evita a ocorrência de falsos conflitos e tem como maior vantagem sua escalabilidade. As duas técnicas foram implementadas no sistema de MTS TL2 e avaliadas através do *benchmark* STAMP.

Resultados mostraram a utilidade de ambas as técnicas e como elas se tornam cada vez mais importantes à medida que o número de *threads* paralelas aumentam. O *speedup* máximo obtido foi 1,63x quando executando oito *threads* em paralelo. Alguns programas tiveram ganhos significativos de desempenho, mas outros não obtiveram os mesmo resultados. Programas que ainda não foram beneficiados pela técnica tendem a apresentar *speedup* quando processadores de 16 cores ou mais estiverem disponíveis.

Finalmente, nós caracterizamos os programas nos quais os benefícios foram maiores. Encontramos que programas com transações longas e que permanecem a maior parte do tempo em código transacional são os melhores alvos de uma técnica de detecção de conflitos livre de abortos falsos. Com esse conjunto de características definido, sugerimos uma promissora otimização para sistemas de MTS.

Referências

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, New York, NY, USA, 2009. ACM.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[3] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 163–174, New York, NY, USA, 2008. ACM.

[4] E. Atoofian, A. Baniasadi, and Y. Coady. Adaptive read validation in time-based software transactional memory. pages 152–162, 2009.

[5] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 115–126, Washington, DC, USA, 2008. IEEE Computer Society.

[6] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 247–258, New York, NY, USA, 2008. ACM.

[7] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[9] P. J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[11] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Roetteler. Racetm: detecting data races using transactional memory. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 104–106, New York, NY, USA, 2008. ACM.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[13] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[14] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[17] V. J. Marathe, W. N. S. Iii, and M. L. Scott. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

[18] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[19] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[20] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[21] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[22] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[23] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[24] A. Shriraman and S. Dwarkadas. Refereeing conflicts in hardware transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 136–146, New York, NY, USA, 2009. ACM.

[25] Z. Xiaoqiang, P. Lin, and X. Lunguo. Lowering conflicts of high contention software transactional memory. In *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, pages 307–310, Washington, DC, USA, 2008. IEEE Computer Society.

[26] C. Zilles and R. Rajwar. Implications of false conflict rate trends for robust software transactional memory. In *IISWC '07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, pages 15–24, Washington, DC, USA, 2007. IEEE Computer Society.