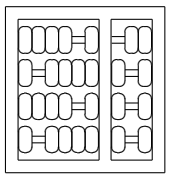


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Avaliação do Impacto de Técnicas de Formação de Regiões no Gerenciamento de *Code Cache*

G. Vieira, A. Carvalho,
G. Piccoli, G. Valente, J. de Lucca

Technical Report - IC-13-99 - Relatório Técnico

December - 2013 - Dezembro

Avaliação do Impacto de Técnicas de Formação de Regiões no Gerenciamento de *Code Cache*

Gilvan Vieira* Alisson Linhares* Guilherme Piccoli*
Gilberto Valente† Jonatas de Lucca‡

2013-12-12

1 Introdução

Uma máquina virtual, como definida originalmente por Popek e Goldberg [1], é uma duplicata, isolada e eficiente, de uma máquina real. Atualmente o termo é usado de forma mais genérica e não está necessariamente associado à duplicata de máquinas implementadas em *hardware*. As máquinas virtuais são sempre implementadas como camadas de *software*, de modo que podem estar "acima" do sistema operacional, como máquinas de processo, ou mais próximas do *hardware*, como *co-designed virtual machines*. Independente do tipo de máquina virtual, sua implementação em geral recai em duas técnicas para emular o comportamento de sua entidade virtualizada (que doravante será denominada *guest*): interpretação ou tradução.

Interpretação é uma técnica relativamente elementar, que busca executar as instruções da arquitetura *guest* uma-a-uma no *host*(ambiente real em que a máquina virtual é executada), de modo que, embora seja precisa e mais simples de implementar, a técnica peca no desempenho. Já a tradução, que pode ser estática ou dinâmica, consiste em traduzir blocos de código da arquitetura *guest* para código nativo da arquitetura *host*, de tal modo que o desempenho é melhor que a interpretação, mas a dificuldade em se implementar tradutores de código é maior.

Tradutores dinâmicos são muito mais usados, pois a tradução estática é muito limitada (devido à problemas como localização de código e avaliação de alvos de saltos indiretos) - por outro lado, a tradução dinâmica ocorre concomitantemente com a execução do programa *guest*, portanto o desempenho do tradutor e do código tra-

*Instituto de Computação - Universidade Estadual de Campinas (UNICAMP)
†Faculdade de Engenharia Mecânica - Universidade Estadual de Campinas (UNICAMP)
‡Instituto Eldorado

duzido são cruciais para que a máquina virtual possa executar numa escala temporal tolerável e condizente com a execução do código *guest* em seu ambiente nativo.

1.1 Motivação

Os tradutores dinâmicos de binários se utilizam de informações sobre o comportamento em tempo de execução de um programa *guest* para melhorar o desempenho deste. Uma abordagem para se atingir esse objetivo é o perfilamento contínuo, para se obter informações dos trechos ditos quentes de um programa - pode-se então efetuar otimização nesses fragmentos de código, que são responsáveis pela maior parte do tempo de execução do programa [2]. Em geral, o processo de detecção de código quente, otimização e execução é subdividido em quatro etapas: primeiramente, analisa-se o conjunto de instruções executadas buscando determinar o fluxo de execução da aplicação; em seguida, é feita a tradução (em geral com otimização) dos fragmentos quentes de código. Então os blocos de código traduzidos são armazenados em uma estrutura denominada *code cache* - ou simplesmente *cache* -, de forma que seja possível executá-los; finalmente, o bloco de código traduzido é executado diretamente da *code cache* pelo resto da execução da aplicação, ou até que seja removido da *cache* para fornecer espaço para novas traduções [3].

Para garantir que não seja desperdiçado tempo na otimização de código pouco executado, o tradutor dinâmico agrupa instruções por regiões de código, que são unidades de execução supostamente quentes - ou seja, é esperado que regiões contenham trechos do código executados múltiplas vezes, como laços. As otimizações são aplicadas no escopo de tais regiões. A formação de regiões é importante principalmente no que toca a qualidade das otimizações realizadas no código *guest*, pois espera-se que tais otimizações, por serem custosas, sejam aplicadas em trechos de significativo impacto no tempo total de execução da aplicação. Além disso, a formação de regiões permite que sejam aplicadas transformações de código mais agressivas, pois o escopo das otimizações aumenta.

Um fator por vezes limitador do desempenho de máquinas virtuais, e complicador de sua implementação, é a *code cache*. Tal estrutura - que na seção 4.2 será explicada em mais detalhes - consiste numa coleção de códigos já traduzidos e prontos para a execução; de fato, a execução ocorre na própria *code cache*. O grande problema aqui é que tal *cache* possui tamanho limitado e se faz necessária uma política de gerenciamento da mesma, pois conforme novas regiões são criadas, mais espaço da *code cache* é usado, de modo que é preciso remover regiões da estrutura. É fato também que programas possuem fases [4], logo manter regiões *ad eternum* na *code cache* é desnecessário, pois as regiões são quentes apenas em determinadas fases da aplicação. Assim, o uso de perfilamento e uma técnica efetiva de formação de regiões permite aumentar a eficiência da *code cache* através da criação de regiões realmente quentes.

Nosso trabalho busca especialmente avaliar a relação entre técnicas de formação de regiões, modos de perfilamento e políticas de gerenciamento de *code cache*. Almejamos com isso apresentar um resultado que seja um facilitador na escolha de uma técnica de formação de regiões em conjunto com uma política de gerenciamento de *code cache* para um projeto de máquina virtual.

2 Objetivos

Nosso objetivo consiste em apresentar estatísticas sobre a relação de técnicas de formação de regiões com políticas de gerenciamento de *code cache*. Buscamos embasar uma escolha do par técnica de formação/política de gerenciamento que permita obter o máximo desempenho em implementações de máquinas virtuais; para tanto, apresentaremos avaliações qualitativas e quantitativas acerca do desempenho de *benchmarks* variando as técnicas de formação de regiões bem como as políticas de gerenciamento da *code cache*. Além disso, uma análise sobre o método de perfilamento usado e suas implicações no desempenho da técnica de formação de regiões é apresentada.

3 Trabalhos relacionados

Zinsly [5] apresenta, em sua dissertação de mestrado, um estudo comparativo entre técnicas de formação de regiões. Tal estudo pode ser considerado inovador, visto que a implementação de múltiplas técnicas de formação de regiões é muito complicada. Na dissertação foi usada uma abordagem que busca simular o comportamento da execução de uma aplicação usando um autômato, do mesmo modo que no presente trabalho (veja seção 4). Foram levantadas várias estatísticas sobre o desempenho das aplicações ao se usar variadas técnicas de formação de regiões; análises dessas estatísticas levaram à conclusão de que para aplicativos do *benchmark* SPEC [6], as técnicas NET e MRET2 (ambas descritas na seção 4.1) se mostram mais eficientes, ao passo que para o *benchmark* SYSMark [7] a técnica LEF - que consiste em agrupar funções por regiões, e foi proposta na dissertação - é mais interessante.

Hazelwood e Smith [8] apresentam um estudo sobre técnicas de gerenciamento de *code cache*. São várias as políticas avaliadas, como *Flush When Full*, *Least-Recently Accessed* (comumente chamada de *Least-Recently Used*), *Fine Grained FIFO* (denominada no artigo como LRC, ou *Least-Recently Created*), *Largest Element*, entre outras. Levando em conta critérios como fragmentação da *code cache*, complexidade de implementação e taxa de *miss*, os resultados apontam a política *Fine Grained FIFO* como sendo a de melhor custo/benefício, pois consegue reduzir a taxa de *miss* para um valor próximo da metade do obtido ao se usar a técnica *Flush When Full*.

No trabalho de Hazelwood e Smith [3] são exploradas políticas de gerenciamento da *code cache* do tipo *Coarse Grained FIFO*, onde as regiões são agrupadas, e a remoção é por agrupamento, e não por região; busca-se assim reduzir o custo de se iniciar o processo de remoção para apenas uma região. Os autores utilizaram o tradutor dinâmico de binários DynamoRIO [9] para gerar as regiões e alimentar o simulador de *code cache*, que por sua vez implementa as várias técnicas analisadas. Também neste trabalho foi construído um modelo analítico de custo, obtido através do perfilamento do DynamoRIO, fornecendo informações sobre o custo da realização de tarefas como remoção de regiões, remoção de encadeamento e de tradução. Seus resultados mostraram que o uso de uma granularidade média para um política FIFO de gerenciamento resulta em um bom balanceamento entre faltas na *cache* e custo total.

4 Metodologia

A unidade de código que vamos analisar são os traços dinâmicos. Traços dinâmicos são registros das instruções executadas por um programa dentro de um ambiente virtualizado que são armazenados em um arquivo para posterior análise - para gerar grande parte dos traços usados nesse trabalho, foi utilizado o interpretador de código x86 denominado *bochs* [10]. Os traços possuem estruturas de dados que descrevem as instruções executadas, como seus *opcodes*, operadores, etc., além do tamanho das instruções e de uma *flag* que indica se a estrutura representa uma instrução executada ou um endereço de memória que foi acessado.

Avaliar técnicas de formação de regiões em ambientes concretos de virtualização é uma tarefa muito complicada - pequenas modificações teóricas nas técnicas poderiam acarretar em profundas modificações na implementação dessas. Assim, uma solução bastante satisfatória para esse problema foi proposta por Porto et. al. [2] - o dito trabalho apresenta um autômato finito determinístico que simula a execução das instruções. Tal autômato, denominado TEA (*Trace Execution Automata*), lê instruções de traços dinâmicos e simula sua execução com transições entre estados - essa estrutura se assemelha visualmente a um grafo dirigido em que os nós representam instruções e as arestas indicam caminhos possíveis entre elas.

A ferramenta RAIn, proposta por Zinsly [5], implementa o TEA e será usada no presente trabalho para analisar tais traços. Essa ferramenta é capaz de varrer esses traços e realizar qualquer tipo de operação sobre eles, desde exibir na tela as instruções/endereços até imprimir estatísticas desejadas. A principal vantagem em se utilizar o RAIn é justamente a possibilidade de se avaliar técnicas sofisticadas sem ter que de fato implementá-las. Um pormenor de tal ferramenta é a ausência de um decodificador de instruções x86 que permita exibir as instruções numa maneira de mais alto-nível, como mnemônicos de linguagem *assembly*, por exemplo. Uma biblioteca

decodificadora denominada Udis86 [11] foi agregada à ferramenta para tanto.

4.1 Técnicas de formação de regiões

A unidade básica de tradução de código é o bloco básico dinâmico, que é definido como um conjunto de instruções iniciado imediatamente após uma instrução de salto, e que contém uma sequência de instruções até o próximo salto. Regiões são agrupamentos de blocos básicos dinâmicos em estruturas maiores, que em geral possuem apenas uma entrada, mas podem conter mais de uma saída - tais estruturas são denominadas superblocos. A formação de regiões eficientes - isto é, que representam trechos quentes do programa - é um mecanismo chave para que se possa realizar otimizações adequadas no programa *guest*, de modo que seu tempo de execução seja o mais próximo possível do que seria num ambiente real. Nesse trabalho, avaliamos as seguintes técnicas de formação de regiões: NET e MRET2.

A técnica NET (*Next-Executing Tail*) [12] - previamente denominada MRET (*Most Recently Executed Tail*) - consiste em agrupar blocos básicos dinâmicos em superblocos seguindo o seguinte critério:

1. Através de perfilamento, determinam-se bons potenciais inícios de regiões - para tanto, adicionam-se contadores para instruções que são alvos de saltos "para trás" ou saídas de regiões já formadas;

2. Uma vez que um determinado contador tenha atingido um certo limiar, passa-se a incluir as instruções subsequentes na região, até que seja encontrado um critério de parada, como por exemplo, instruções que já pertençam a uma região (seja ela diferente ou a mesma que está sendo formada), instruções de salto "para trás" ou até que um número máximo previamente determinado de instruções tenham sido incluídas na região.

A idéia da técnica NET é que se um caminho no programa é executado com frequência (código quente), é provável que a partir do momento de início do processo de formação de uma região tal caminho seja seguido, bastando portanto adicionar as instruções sequencialmente na região. Há o risco de erro, pois mesmo um caminho sendo quente ele não é sempre tomado; no caso de erro, a região não será composta por um caminho frequentemente usado e logo deverá ser desfeita/substituída na *code cache*.

Para atenuar a chance de formar uma região fria, foi proposta uma variação da técnica NET, denominada MRET2 [13]. Tal técnica segue passos parecidos com os da NET, contudo a partir de um ponto de início de uma região, a técnica avalia dois caminhos, e não apenas um. Quando um ponto de início de superbloco é atingido, a técnica passa a gravar as próximas instruções até que um critério de parada seja satisfeito, mas uma região não é formada ainda. Se esse mesmo ponto de início atingir novamente o *threshold*, as instruções em sequência passam a ser gravadas - assim, há

dois caminhos partindo do mesmo ponto. Então, a técnica seleciona as instruções comuns entre os caminhos para formar a região, diminuindo assim a probabilidade de formar um região fria.

4.2 Políticas de gerenciamento de *code cache*

A *code cache* é um dos componentes centrais em um sistema de tradução dinâmica de binários, e o seu bom gerenciamento é uma das tarefas mais fundamentais no desenvolvimento de uma máquina virtual. Assim, uma implementação adequada deve manter em seu espaço de memória os blocos de instruções que são frequentemente utilizados pelo tempo em que de fato são executados, reduzindo o número de blocos que são repetidamente traduzidos [14].

Ainda que sejam similares, a *cache* usual (implementada em *hardware*) e a *code cache* possuem diferenças fundamentais [15], tais como:

1. O tamanho dos blocos é variável na *code cache*;
2. A localização dos blocos é relevante na *code cache*, visto que esses são dependentes entre si, por causa do encadeamento de blocos (*chaining*);
3. Não há cópia do conteúdo da *code cache* - blocos removidos devem ser novamente traduzidos.

Na prática, as implementações de *code cache* possuem um tamanho limitado - dependendo da carga de trabalho da aplicação, rapidamente pode-se esgotar o espaço na estrutura, sendo então necessário remover blocos para fornecer espaço para novas traduções [3]. A política mais simples possível de gerenciamento é não remover as traduções - contudo, tal abordagem é inviável, devido ao limite físico de memória existente. Assim se faz necessário utilizar uma política de gerenciamento de exclusão de dados da *code cache*; nesse trabalho, avaliamos as seguintes políticas: *Flush When Full*, *Fine Grained FIFO* e *Coarse Grained FIFO*.

A política *Flush When Full* consiste em deixar a *code cache* ser preenchida por traduções até que o limite de sua capacidade seja atingido - quando isso ocorre, todos os blocos traduzidos são removidos de uma única vez. Uma vantagem dessa técnica é que não há necessidade de se gerenciar o encadeamento entre os blocos, além de sua fácil implementação; todavia a técnica incorre no risco de remoção de blocos ainda quentes, forçando retraduções dos mesmos.

A técnica *Fine Grained FIFO* é utilizada para tirar proveito da localidade temporal no acesso aos blocos traduzidos. Para isso a *code cache* é gerenciada na forma de um *buffer* circular que funciona como uma fila - a primeira região adicionada será a primeira a ser excluída, quando houver necessidade. Visto que os programas possuem fases, uma clara vantagem desse modo de gerenciamento é a expectativa de

que regiões removidas sejam aquelas usadas em fases mais "antigas" do programa, ou seja, espera-se que o impacto da remoção de um superbloco seja menor que no caso de um *flush* completo, por exemplo. Como desvantagem, essa técnica exige o monitoramento das ligações de encadeamento entre superblocos, adicionando o custo de remoção dessas quando uma região é removida - além da implementação mais sofisticada.

Por fim, a política *Coarse Grained FIFO* busca reduzir justamente o custo de remoção das ligações de encadeamento entre regiões. Tal política funciona de maneira similar à *Fine Grained FIFO*, ou seja, também funciona em forma de *buffer* circular - no entanto, é feito um agrupamento de superblocos em unidades maiores dentro da *cache*, que podem ser enxergadas como "contêineres" de regiões. No interior dessas estruturas existem superblocos encadeados, mas quando há a necessidade de remoção na *code cache*, contêineres inteiros são removidos, de maneira que apenas é necessária a remoção das ligações de encadeamento de regiões que estão em contêineres diferentes. Uma desvantagem dessa técnica é que ao remover um contêiner, algumas regiões contidas nele podem ainda ser quentes - é interessante encontrar um balanço entre o número dessas estruturas e o impacto da remoção dos encadeamentos. Note que o caso de 1 contêiner determina uma política *Flush When Full* e conforme o número desses aumenta, temos um comportamento cada vez mais próximo da *Fine Grained FIFO*- nossos testes foram realizados com 2, 4 e 8 contêineres.

4.3 *Benchmarks*

A escolha de *benchmarks* pertinentes para os testes é uma tarefa fundamental para que a relevância e aplicabilidade das conclusões sejam efetivas. Nossa escolha recai sobre a conhecida suíte de testes denominada SPEC CPU 2006 [6], que contém variados *benchmarks* de computação numérica, compressão de dados, compilação de código, etc. - optamos por avaliar especialmente quatro desses *benchmarks*; três fazem parte do pacote de computação de ponto-flutuante, e um do pacote de computação inteira. São eles:

- *milc* [16]: Acrônimo para *MIMD Lattice Computation*, *milc* é uma ferramenta de simulações numéricas cujo objetivo é avaliar campos de força na teoria de física quântica. Escrito em C, faz parte do pacote de computação de ponto-flutuante.

- *bwaves* [17]: Aplicativo de simulação numérica de ondas de choque - faz uso de um algoritmo iterativo para resolver um sistema de equações diferenciais do tipo Navier-Stokes. Escrito em Fortran 77, efetua apenas computações de ponto-flutuante.

- *dealII* [18]: Aplicação para a resolução numérica de equações diferenciais parciais que utiliza, para tanto, o método dos elementos finitos adaptativo. Escrito em C++ moderno, faz parte do pacote de computação de ponto-flutuante.

- *bzip2* [19]: Implementação de um algoritmo de compressão de dados, realiza uma série de compressões - em memória - de imagens, binários e arquivos de texto. Escrito em C, faz parte do pacote de computação inteira do SPEC CPU.

4.4 Métricas

Usamos as seguinte métricas para coletar estatísticas sobre as regiões geradas:

- Número de regiões: tal métrica avalia o número total de regiões formadas por uma determinada técnica de formação de regiões. Equivale ao número total de traduções realizadas, portanto quanto maior o número de regiões, maior o tempo gasto na tradução de código. Essa métrica pode ser boa tanto se for alta quanto se for baixa - isso depende da taxa de completude das regiões e da cobertura do código, explicadas abaixo.

- Cobertura do código: tal medida descreve o percentual do número total de instruções que são executadas dentro de regiões, ou seja, a cobertura determina o quanto da execução se dá em regiões. Assim, essa métrica está relacionada à qualidade das regiões, que se forem quentes de fato, tendem a fazer a cobertura subir.

- Taxa de completude: medida bastante importante para se determinar a qualidade das regiões formadas, a completude indica a porcentagem de vezes que a região foi executada completamente, ou seja, do início ao fim. Se a completude for baixa, isso nos permite inferir que a técnica de formação de regiões está gerando regiões não-eficientes, que não estão sendo utilizadas por completo e portanto desperdiçam tempo de otimização e espaço na *code cache*.

- Duplicação de código: tal métrica determina a quantidade de vezes que instruções aparecem em mais de uma região, isto é, uma alta duplicação indica que estão havendo muitas retraduções de código.

Boas técnicas de formação de regiões tendem a apresentar resultados que acompanham um baixo número de regiões com alta completude e cobertura, e de preferência sem duplicação excessiva de código.

4.5 Validação dos resultados

Para verificar a corretude das técnicas de formação de regiões e gerenciamento de *code cache*, optou-se por utilizar traços de execução sintéticos. Tais traços são gerados artificialmente e de maneira controlada. Seu conteúdo é totalmente conhecido e seu comportamento é definido visando testar os algoritmos de formação de regiões e gerenciamento de *cache* em condições específicas.

Os traços gerados possuem apenas instruções de um mesmo tamanho fixo, o que permite validar facilmente o tamanho das regiões geradas e monitorar o uso da *code cache*. Os traços não possuem registros de acessos à memória pois essa informação não é relevante para os algoritmos testados nesse trabalho. Apenas o fluxo de execução das instruções é levado em consideração para a formação de regiões. O conteúdo das instruções simuladas (valores do *opcode*) também não é utilizado.

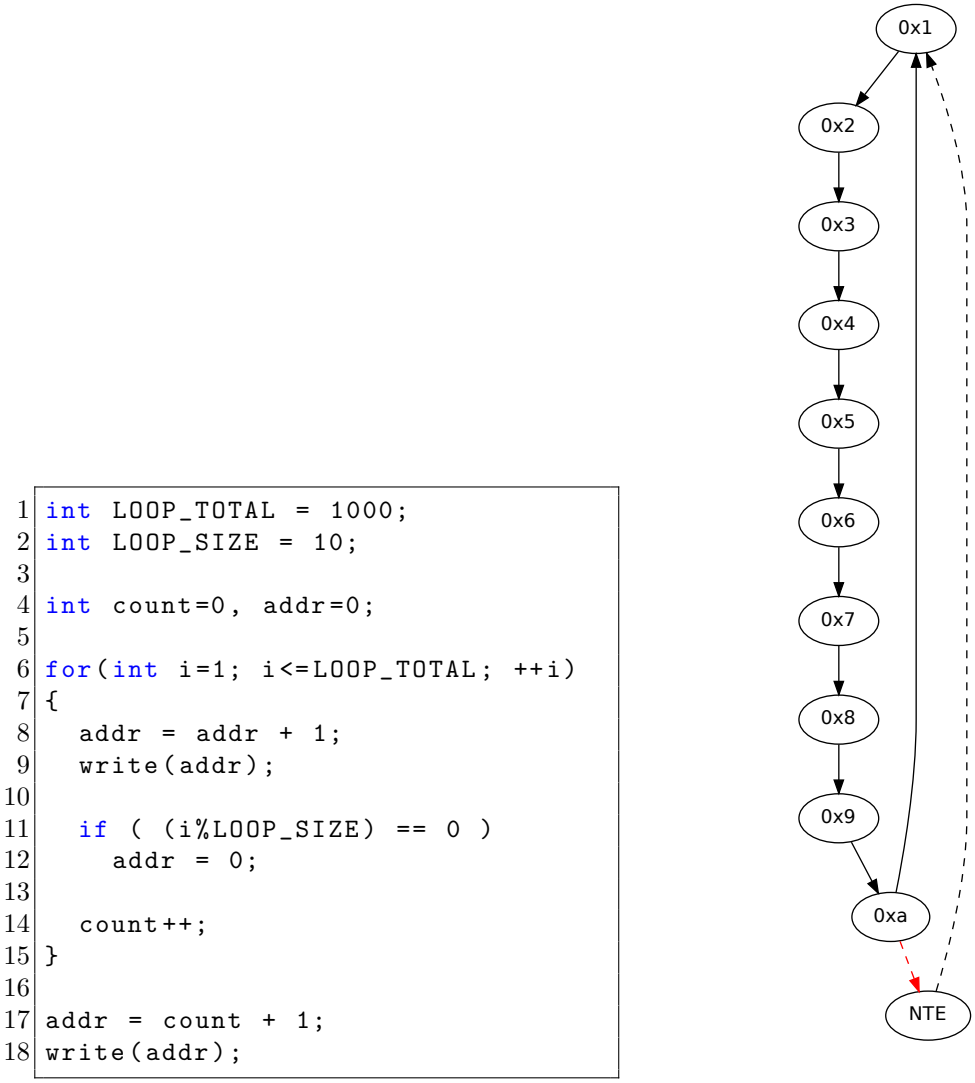
A geração do traço é feita utilizando endereços sequenciais arbitrários. Isso facilita a geração dos *traces* e a visualização das regiões geradas. Os testes podem ser feitos variando-se o número de instruções executadas em cada traço gerado. O número de regiões formadas pode ser controlado variando o número de laços de repetição presentes no traço, o número de iterações de cada laço e o número de execuções necessários para se formar uma região (*threshold* do algoritmo de formação de regiões). O uso da *cache* pode ser analisado variando-se o número de regiões geradas pelo traço, o tamanho de cada região e a capacidade da *cache*.

Veja por exemplo o trecho de código apresentado na Figura 1a. Tal código representa um laço de tamanho 10 que será executado 100 vezes, seguido de uma única instrução, totalizando 1001 instruções executadas. O traço gerado para esse código é o da Figura 1b, em que 0x1 à 0x8 são instruções diversas, 0x9 é uma comparação, 0xA é um desvio condicional, a linha vermelha indica a saída do laço de repetição e a linha tracejada indica a entrada da região. Se o valor da variável `LOOP_TOTAL` for muito baixo, a região não será formada. Caso o número de execuções do laço seja maior que o *threshold*, a região será formada. Como os traços são arbitrários e podem começar e acabar em qualquer instrução, a região poderia não retornar ao interpretador (transição representada pelo nó NTE do grafo).

5 Resultados

Experimentos foram realizados com as técnicas de formação de regiões NET e MRET2, explorando as políticas de gerenciamento de *code cache* : *Flush When Full*, *Fine Grained FIFO* e *Coarse Grained FIFO* com granularidades 2, 4 e 8.

Uma vez que todas as traduções realizadas para uma determinada aplicação caibam na *code cache*, sem que seja necessário remover regiões por falta de espaço, a política de gerenciamento não faz diferença. No presente trabalho nosso interesse recai em descobrir qual o comportamento do tradutor dinâmico de acordo a política



```
1  int LOOP_TOTAL = 1000;
2  int LOOP_SIZE = 10;
3
4  int count=0, addr=0;
5
6  for(int i=1; i<=LOOP_TOTAL; ++i)
7  {
8    addr = addr + 1;
9    write(addr);
10
11   if ( (i%LOOP_SIZE) == 0 )
12     addr = 0;
13
14   count++;
15 }
16
17 addr = count + 1;
18 write(addr);
```


(a) (b)

Figura 1: Código de validação e traço gerado.

de gerenciamento; assim, para todos os resultados neste artigo é assegurado que o tamanho da *cache* é menor do que o necessário para armazenar todas as traduções da aplicação. Para isto, foram executadas simulações onde o tamanho máximo da *cache* é ilimitado - obtivemos assim a Figura 2, que mostra na primeira coluna o tamanho em *bytes* necessários para armazenar todas as regiões de cada aplicação e na segunda coluna o número de instruções traduzidas.

Para assegurar o aumento da pressão na *cache*, os tamanhos da *code cache* usados

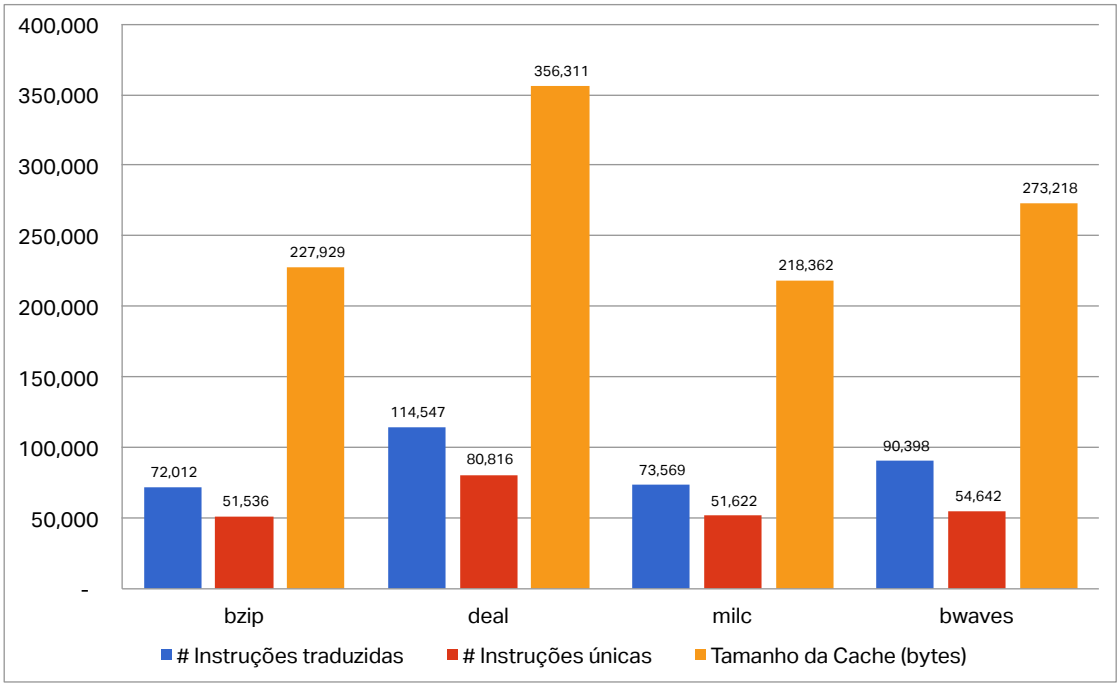


Figura 2: Tamanho da cache sem política de gerenciamento.

nos experimentos foram calculados na forma $MaxCache/n$, onde $MaxCache$ é o tamanho em *bytes* obtido ao executar a aplicação no simulador com um tamanho de *cache* ilimitado e n é um fator de pressão utilizado para assegurar que a política de gerenciamento está realmente em uso.

5.1 Encadeamento de traduções

Sabemos que o ato de traduzir uma região para o conjunto de instruções do *host* é uma tarefa relativamente custosa. Por este motivo, para compensar o custo de tradução, é desejável maximizar o tempo de execução dentro das regiões traduzidas, recorrendo o mínimo possível ao gerenciador de emulação.

Uma técnica bastante usada em máquinas virtuais, que reduz o impacto gerado pelos saltos entre as regiões traduzidas, é o *chaining* - conhecido também como encadeamento. Esta técnica é caracterizada por substituir as chamadas para o gerenciador de emulação por instruções de saltos para as regiões-alvo. O princípio consiste em verificar se o endereço alvo do salto foi previamente traduzido, e em seguida, inserir um trecho de código responsável por efetuar o salto de forma direta e com o menor custo possível. Desta forma, as regiões são encadeadas dentro da *cache*, evitando o desvio do fluxo de execução para o gerenciador de emulação.

Apesar da vantagem em encadear as regiões traduzidas, o *chaining* afeta direta-

mente o desempenho da política de gerenciamento de *cache*. Isso pois sempre que existe a necessidade de se remover uma região, será necessário desfazer todos os encadeamentos previamente construídos, cuja região a ser removida faz parte. Por este motivo, este trabalho buscou realizar experimentos com diferentes *benchmarks*, na tentativa de verificar a quantidade de *chains* gerados pelas técnicas de formação de regiões, quando combinados com diferentes políticas de gerenciamento de *cache*.

Na Figura 3, podemos verificar o número de *chains* removidos quando a pressão da *cache* sobe. É interessante notar que a medida que o número de agrupamentos da *Coarse Grained FIFO* é aumentado, o desempenho passa a se aproximar da *Fine Grained FIFO* - conforme explicado na seção 4.2 -, removendo uma quantidade substancial de *chains*. Devido ao fato de que a *Coarse Grained FIFO* divide a *cache* em grupos, é natural que o aumento da granularidade faça com que os algoritmos apresentem um desempenho similar.

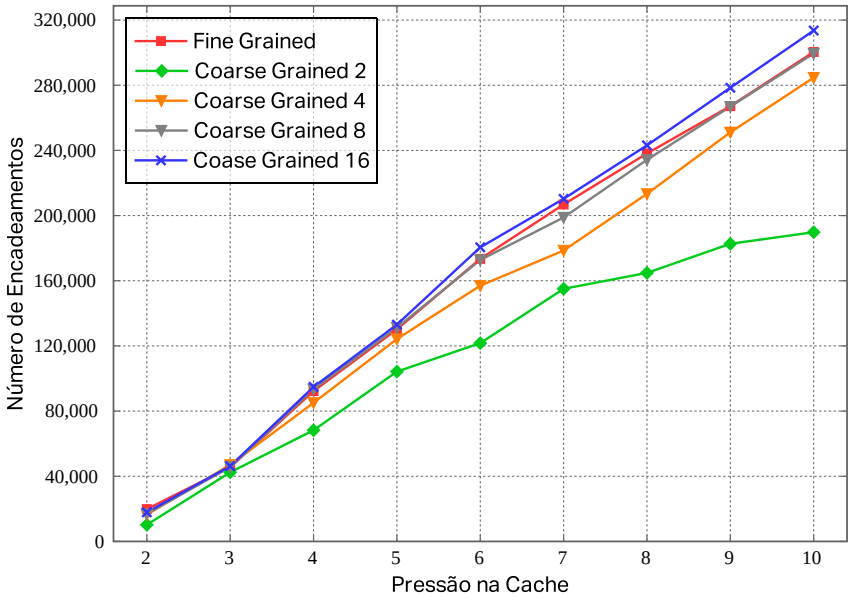


Figura 3: Número de *chains* removidos para aplicação *milc* e a técnica NET.

Foi verificado ainda que, quando o número de agrupamentos chega a uma determinada proporção da *cache*, seu desempenho começa a degradar, pelo fato da quantidade de elementos presente no tal agrupamento formar mais *chains* entre as regiões fora dele do que nas regiões internas a ele - pudemos verificar uma perda de desempenho bastante acentuada quando o agrupamento ultrapassou a granularidade 8.

5.2 Influência do *profiler*

Um dos grandes desafios na formação de regiões é a detecção de código quente, isto é, trechos em programas que são executados com grande frequência. O papel do *profiler* é o de auxiliar o algoritmo de formação de regiões, armazenando estatísticas de execução de pontos específicos em um determinado programa. Esses pontos perfilados são escolhidos com base em uma condição previamente estabelecida, podendo por exemplo, ser imposta pelo algoritmo de formação de regiões.

Como os programas de computador tendem a respeitar a localidade espacial e temporal, um perfilamento extremamente simples que pode ser aplicado é a contagem de vezes que um determinado endereço é executado. Contudo, é possível encontrar técnicas mais elaboradas, que fazem uso de propriedades estáticas dos binários, ou até mesmo, do comportamento das execuções anteriores de um dado programa.

Um dos problemas da abordagem de perfilamento adotada originalmente pelo RAIn, é que todos os endereços alvos de saltos são perfilados sem que se respeite as mudanças de fases dos programas. Isto é, a técnica de perfilamento usada pelo RAIn não prevê que em estágios de execução diferentes um mesmo endereço pode ser executado com uma frequência menor que a apresentada em outras fases da aplicação. Essa característica, aliada a um perfilamento incremental, pode gerar regiões que não são realmente quentes em uma determinada fase da aplicação, aumentando o número total de regiões formadas e prejudicando a taxa de completude.

Além de apresentar um aquecimento incremental, a política de perfilamento original do RAIn não apresenta um resfriamento das instruções - o *profiler* é cumulativo. Dessa forma, após a remoção, a região removida continuará quente e quando for revisitada será retraduzida. Com isso, o número de regiões aumenta drasticamente, pois em determinadas fases do programa a região pode ser acessada com uma frequência muito menor que em fases anteriores, não sendo necessário uma retradução.

Buscando melhorar tal abordagem de perfilamento, desenvolvemos duas técnicas de resfriamento do *profiler*, o *full reset* e o *partial reset*. A técnica *full reset* consiste em resetar todos os endereços perfilados sempre que uma região é criada. Isso garante que somente as regiões que são utilizadas muito frequentemente, em fases específicas do programa, consigam atingir o limiar mínimo necessário para serem consideradas quentes. Já a técnica de *partial reset* reinicializa somente os contadores dos endereços que são alvos de salto e fazem parte das regiões. Isto garante que após a remoção de uma região, essa, por sua vez, só poderá ser inserida novamente na *cache* quando atingir o limiar de aquecimento imposto pelo *profiler*.

Com base nos experimentos realizados, verificamos que o total de traduções geradas pela técnica NET - na ausência de *cache* e fazendo uso da política original de perfilamento - é superior ao número de traduções quando aplicamos algumas das técnicas de resfriamento sugeridas. Entretanto, na presença de *cache*, a formação de regiões passa a ser afetada pela política de gerenciamento de *code cache* sempre

que há necessidade de se liberar espaço para novas traduções. Assim, quanto maior a pressão sobre a *cache*, mais regiões serão removidas, e consequentemente, mais instruções serão retraduzidas. Este efeito pode ser visualizado na Figura 4, onde à medida que a pressão sobre a *cache* aumenta, o número total de instruções traduzidas aumenta. Contudo, este aumento é muito menor quando o perfilamento utiliza algumas das políticas de resfriamento sugeridas.

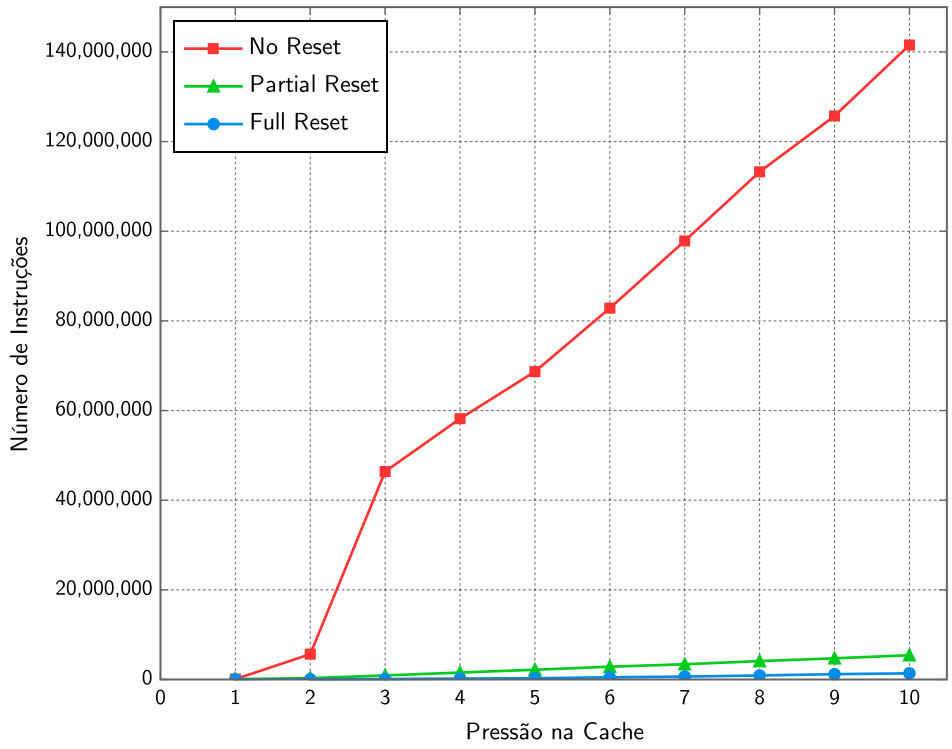
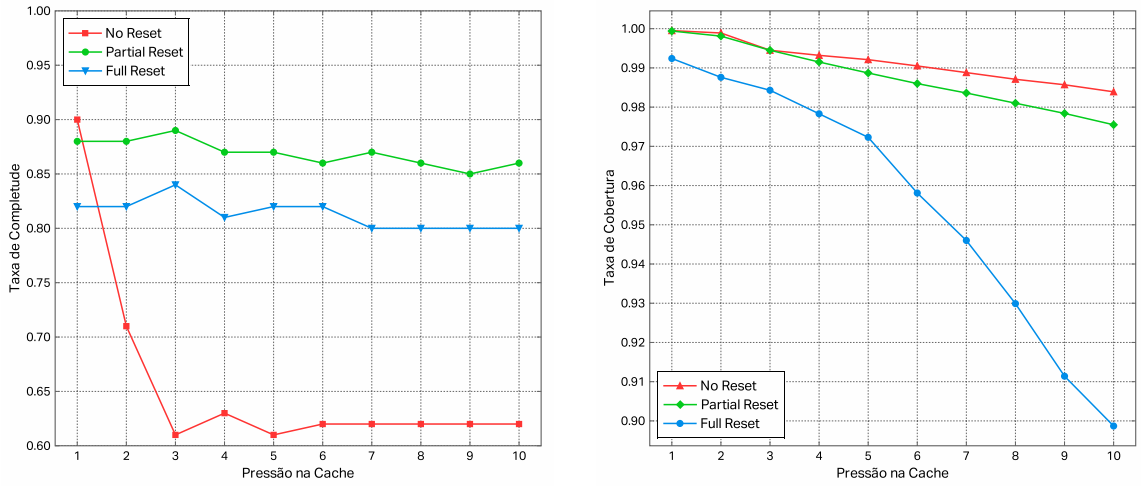


Figura 4: Número de instruções traduzidas para a aplicação *milc*.

É interessante notar que, apesar da redução no número total de traduções, a taxa de cobertura com o uso de um *partial reset* é bem próxima da obtida pela política sem *reset*. Isso se deve ao fato de que o código gerado passa a apresentar uma taxa de completude muito superior, à medida que a pressão da *cache* aumenta, conforme visualizado nas Figuras 5a e 5b. Isso significa que com o resfriamento do *profiler*, as regiões são executadas completamente com uma maior frequência.

5.3 Impacto das técnicas de formação de regiões no gerenciamento de *code cache*.

As análises apresentadas nessa seção se concentram em 2 *benchmarks*: *milc* e *bwaves*. Essa escolha se deu pois em geral os resultados para todos os *benchmarks* acom-



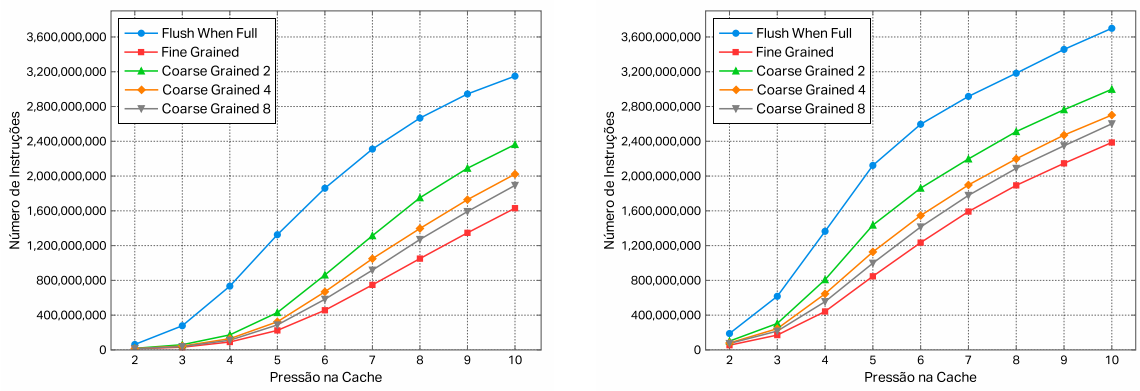
(a) Completude (b) Cobertura

Figura 5: Avaliação do *profiler* utilizando a técnica NET e a aplicação *milc*.

panham o mesmo padrão, portanto esses 2 aplicativos são representativos de todos. Por vezes uma característica avaliada é mais perceptível em um ou outro. A seguir, discorremos sobre nossas avaliações e hipóteses sobre os principais resultados experimentais obtidos. Todas os experimentos foram efetuados utilizando-se a técnica de perfilamento *partial reset*.

A Figura 6 exibe o número de instruções interpretadas em relação ao aumento da pressão na *cache*. A quantidade de instruções interpretadas está diretamente ligada à política de gerenciamento de *code cache* escolhida. Quanto maior for o número de regiões eliminadas, maior será o número de instruções interpretadas. Com base nos resultados obtidos, podemos verificar que a *Flush When Full* é a política que interpreta mais instruções, seguida pela *Coarse Grained FIFO*. Isso se deve ao fato desses dois algoritmos descartarem uma grande quantidade de regiões por vez, quando comparadso à *Fine Grained FIFO*, que apresenta melhor resultado.

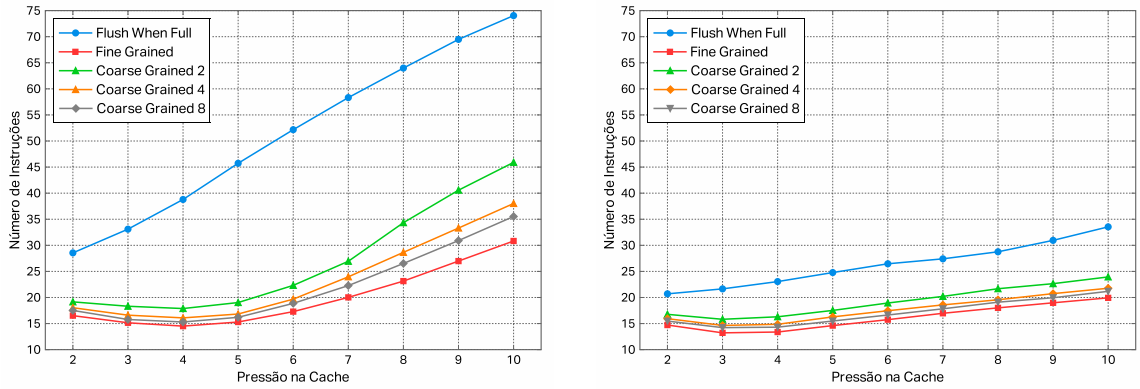
Diferentemente do resultado obtido na interpretação de instruções, o tamanho estático das regiões cresce de forma inversa ao número de regiões eliminadas. Um dos critérios de parada da técnica NET de formação de regiões é que, ao encontrar uma instrução já pertencente à outra região, a formação do superbloco deve parar. Assim, no caso da *Flush When Full* por exemplo, como todas as regiões são removidas de uma vez, esse critério de parada da NET é dificilmente verificado após a exclusão total das regiões da *code cache*- logo, regiões com maior número de instruções são formadas. Ou seja, políticas que eliminam um maior número de regiões tendem a possuir um tamanho estático médio maior. Como podemos visualizar na Figura 7, o tamanho estático para a política *Flush When Full* é até duas vezes maior que o tamanho estático para a política *Fine Grained FIFO*. A técnica MRET2 faz uso dos mesmos



(a) NET (b) MRET

Figura 6: Número de instruções emuladas por interpretação na aplicação *bwaves*.

critérios de parada adotados pela NET, de modo que apresenta um comportamento parecido.

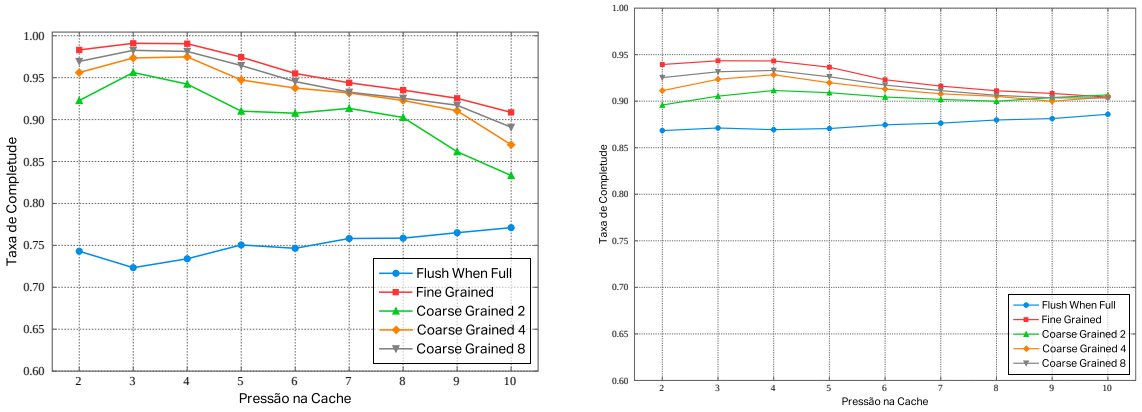


(a) NET (b) MRET

Figura 7: Tamanho estático médio das regiões para a aplicação *bwaves*.

Note que a taxa de completude - indicador da qualidade das regiões - é superior no caso da MRET2, como pode ser visto na Figura 8. Em especial, a política *Flush When Full* apresenta os piores resultados, por motivos já justificados no parágrafo anterior. A técnica MRET2 pois um critério de formação de regiões mais estrito - usa para tanto 2 rodadas da NET e forma regiões que possuem caminhos comuns nessas rodadas. Portanto, é de se esperar que possua completude maior, pois as regiões formadas são mais eficientes, e tendem a ser executadas completamente. A pressão da *code cache* influencia negativamente na completude, pois se o tamanho da *cache* for diminuto, a expectativa é de que regiões sejam criadas a todo momento e a cada nova região, novas estatísticas são geradas; as regiões não têm tempo de ficar

executando na *cache* pois rapidamente são apagadas de lá. Além disso, no caso da NET, o decréscimo é maior conforme aumenta a pressão - nossa hipótese é que isso também se dá por influência do critério de parada da técnica, pois uma vez que a pressão é muito grande, uma maior quantidade de traduções ocorre, gerando regiões de maior tamanho estático mas menor completude.

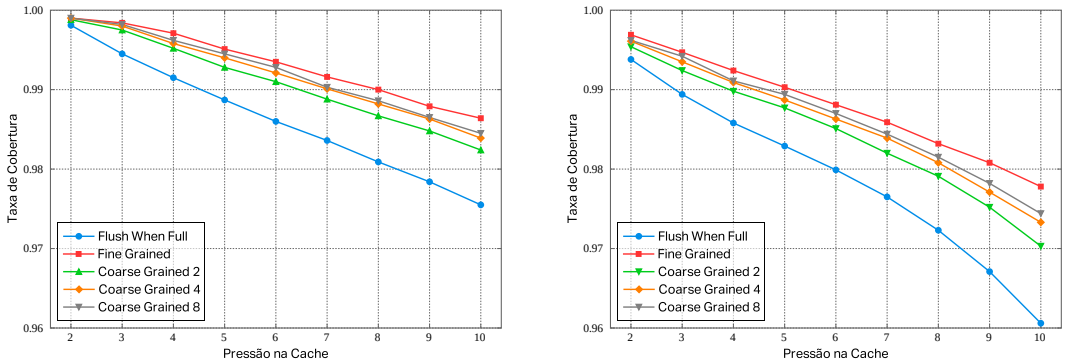


(a) NET (b) MRET

Figura 8: Taxa de completude das regiões para a aplicação *bwaves*.

A taxa de cobertura das regiões, medida bastante importante para avaliar o grau de eficácia da técnica de formação, pode ser vista na Figura 9. Tanto a técnica NET quanto MRET2 apresentam bons resultados, que naturalmente decrescem conforme a pressão na *cache* aumenta. Isso ocorre pois, quanto menor o tamanho da *code cache*, maior o número de traduções que serão necessárias ao longo da execução do programa; assim, maior é o tempo gasto executando instruções via interpretação até que o *threshold* seja atingido para que se efetuem as traduções. E uma vez que ocorram as tais traduções, pouca é a sobrevida delas na *code cache* devido ao grande número de *flushes* que tende a ocorrer, por causa do diminuto tamanho da estrutura. Note que mais uma vez o pior resultado foi obtido pela política *Flush When Full*, devido ao seu alto número de regiões excluídas por *flush*.

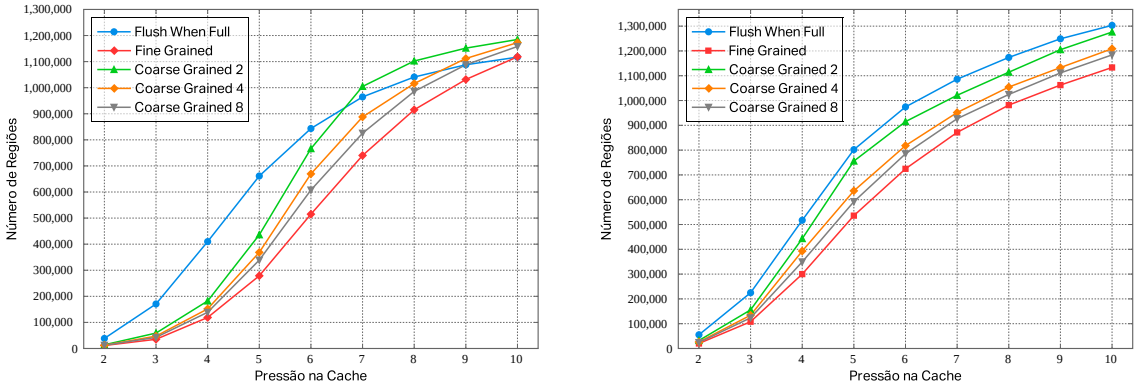
A respeito do número total de regiões, na Figura 10 podemos ver que tal medida é similar para MRET2 e NET, sendo que a primeira apresenta um número levemente maior - nossa hipótese para isso é que, por seu critério estrito, a técnica MRET2 gera mais regiões de menor tamanho estático e com maior completude. Por exemplo, numa situação como a do código da Figura 11, em que há um laço com uma estrutura condicional em seu interior, imagine que o `if` é tomado metade das vezes, contudo de maneira intermitente com o `else`, ou seja, a cada iteração um deles é tomado. A técnica NET geraria uma grande região com todo o laço, de modo que tal região teria baixa completude. Por outro lado, a técnica MRET2 poderia criar 3 regiões, sendo uma com o código comum do laço, uma para o `if` e outra para o `else`, todas



(a) NET (b) MRET

Figura 9: Taxa de cobertura das regiões para aplicação *milc*.

encadeadas na *code cache*. Assim, o número de regiões da MRET2 tende a ser maior que da NET. A pressão da *cache* naturalmente influencia o número de regiões - quanto maior for o número de *flushes*, maior a quantidade necessária de retraduções, e portanto maior o número de regiões geradas. Note que a sobrevida das regiões cai conforme a pressão aumenta.



(a) NET (b) MRET

Figura 10: Número total de regiões para a aplicação *bwaves*.

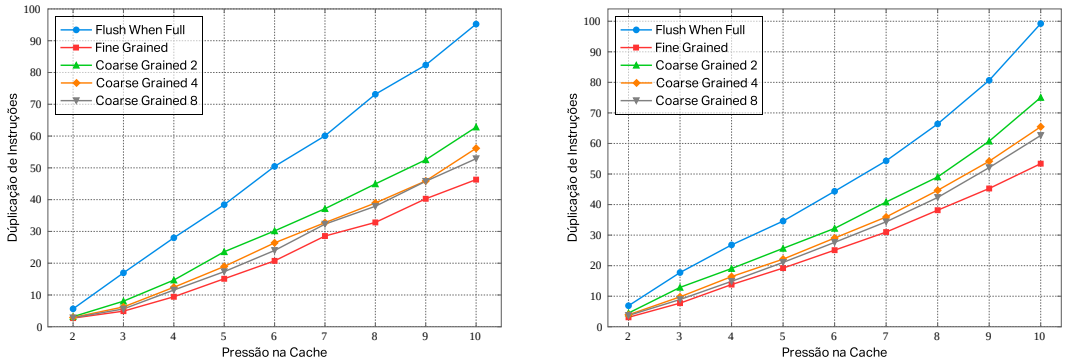
Um ponto interessante sobre a Figura 10a é que, entre as pressões 5 e 7, é possível ver que a política *Flush When Full* apresenta um aumento decrescente do número de regiões geradas, sendo ultrapassada pela *Coarse Grained FIFO* com granularidade 2, e no caso da pressão 10, até mesmo empatando com a *Fine Grained FIFO*. Nossa hipótese para justificar esse fato é a de que o tamanho estático das regiões, que - conforme já foi discutido - aumenta no uso da NET com a política *Flush When Full*, esteja ocasionando esse comportamento; note que maior tamanho estático implica

```
1 while(cond1) {
2   ...
3   if (cond2) {...}
4
5   else {...}
6 }
```

Figura 11: Código que geraria 1 região na NET e 3 na MRET2.

em menor número de regiões - é como se a região contemplasse uma porção maior do código da aplicação (ainda que isso implique numa região menos eficiente e com menor completude).

Por fim, a Figura 12 apresenta as medições de duplicidade de código. Tal medida se mostra bastante coerente com o esperado - podemos ver um crescimento linear na duplicidade, pois quanto maior a pressão na *cache*, maior o número de retraduções, ou seja, maior o número de instruções idênticas em regiões diferentes - ainda que tais instruções estejam em regiões temporalmente dispersas, a métrica nos permite concluir que um aumento exagerado na pressão da *cache*, em especial no caso da política *Flush When Full*, acarreta grande quantidade de retraduções de código.



(a) NET (b) MRET

Figura 12: Duplicação de código para a aplicação *milc*.

6 Conclusões

Com base na análise das estatísticas coletadas, podemos concluir que o uso de uma técnica de resfriamento parcial reduz consideravelmente o número de regiões traduzidas, principalmente quando usada em conjunto com uma política de gerenciamento de *cache*. Verificamos que com o uso de uma técnica de resfriamento, as taxas de cobertura e completude permanecem bem próximas dos valores obtidos pela

política original do RAIn, mesmo que o número de instruções traduzidas seja muito menor. Desta forma, podemos considerar que, na presença de *code cache*, o uso de uma política de resfriamento pode contribuir para a melhora de desempenho geral de uma máquina virtual. Além do impacto causado pelo resfriamento, verificamos que o número total de *chains* removidos cresce de maneira comportada entre os diferentes *benchmarks*. Contudo, foi possível verificar que a granulação da política *Coarse Grained FIFO* pode tornar o algoritmo pior que uma política *Fine Grained FIFO*, caso a quantidade de agrupamentos seja muito alta.

Em se tratando da relação de técnicas de formação de regiões com políticas de gerenciamento de *code cache*, notamos que ambas as técnicas são satisfatórias, com uma leve vantagem para a MRET2 devido à formação de regiões mais eficientes, com maior taxa de completude. É claramente visível pelos resultados que a política *Flush When Full* não é adequada para uma máquina virtual que almeja bom desempenho - tal política se mostrou a de menor desempenho em todos os experimentos. Também um fator importante é a pressão da *code cache* - quanto menor o tamanho da *cache*, maior a perda de desempenho. Entre as políticas de gerenciamento, tanto as políticas *Fine Grained FIFO* quanto *Coarse Grained FIFO* de granularidade baixa (2 ou 4) se mostraram boas opções, sendo que a última leva vantagem pois atenua o impacto da remoção de ligações de encadeamento[3].

Referências

[1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Comm-ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[2] J. Porto, G. Araujo, E. Borin, and Y. Wu, "Trace execution automata in dynamic binary translation," in *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pp. 99–116, Springer-Verlag, 2012.

[3] K. Hazelwood and J. E. Smith, "Exploring code cache eviction granularities in dynamic optimization systems," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pp. 89–, IEEE Computer Society, 2004.

[4] T. Sherwood and B. Calder, "Time varying behavior of programs," 1999.

[5] R. M. Zinsly, "Técnicas de formação de regiões para projetos de máquinas virtuais eficientes," Master's thesis, Unicamp, 2013.

[6] http://www.spec.org.

[7] http://bapco.com/products/sysmark-2012.

[8] K. M. Hazelwood and M. D. Smith, "Code cache management schemes for dynamic optimizers," in *Interaction between Compilers and Computer Architectures*, pp. 102–110, 2002.

[9] http://dynamorio.org.

[10] http://bochs.sourceforge.net.

[11] http://udis86.sourceforge.net.

[12] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more," in *ACM SIGOPS Operating Systems Review*, vol. 34, pp. 202–211, ACM, 2000.

[13] C. Wang, B. Zheng, H. Kim, M. Breternitz Jr, Y. Wu, *et al.*, "Two-pass MRET trace selection for dynamic optimization," 2010. US Patent 7,694,281.

[14] K. Hazelwood, *Code cache management in dynamic optimization systems*. PhD thesis, 2004.

[15] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann Publishers, 2005.

[16] http://www.spec.org/auto/cpu2006/Docs/433.milc.html.

[17] http://www.spec.org/auto/cpu2006/Docs/410.bwaves.html.

[18] http://www.spec.org/auto/cpu2006/Docs/447.dealII.html.

[19] http://www.spec.org/auto/cpu2006/Docs/401.bzip2.html.