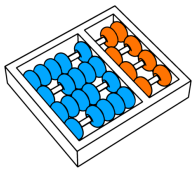
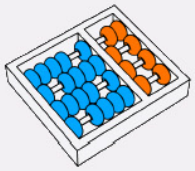


Raphael Moreira Zinsly

“Técnicas de formação de regiões para projetos de máquinas virtuais eficientes”

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

Raphael Moreira Zinsly

"Técnicas de formação de regiões para projetos de máquinas virtuais eficientes"

Orientador(a): **Prof. Dr. Sandro Rigo**
Co-Orientador(a): **Prof. Dr. Edson Borin**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Raphael Moreira Zinsly, sob orientação de Prof. Dr. Sandro Rigo.

Assinatura do Orientador(a)

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Z66t

Zinsly, Raphael Moreira, 1989-
Técnicas de formação de regiões para projetos de máquinas virtuais eficientes / Raphael Moreira Zinsly. – Campinas, SP : [s.n.], 2013.

Orientador: Sandro Rigo.
Coorientador: Edson Borin.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Compiladores (Programas de computador). 3. Máquinas virtuais. 4. Sistemas de computação. I. Rigo, Sandro,1975-. II. Borin, Edson,1979-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Region formation techniques for efficient virtual machines design
Palavras-chave em inglês:
Computer architecture
Compilers (Computer programs)
Virtual machines
Computing systems
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Sandro Rigo [Orientador]
Mauricio Breternitz Junior
Paulo César Centoducatte
Data de defesa: 30-10-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 30 de outubro de 2013, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Mauricio Breternitz Junior
AMD RESEARCH

Prof. Dr. Paulo Cesar Centoducatte
IC / UNICAMP

Prof. Dr. Sandro Rigo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Técnicas de formação de regiões para projetos de máquinas virtuais eficientes

Raphael Moreira Zinsly[1]

30 de outubro de 2013

Banca Examinadora:

- Prof. Dr. Sandro Rigo (*Orientador*)
- Prof. Dr. Mauricio Breternitz Junior - AMD Research
- Prof. Dr. Paulo César Centoducatte - IC / UNICAMP
- Prof. Dr. Guido Costa Souza de Araújo (Suplente) - IC / UNICAMP
- Prof. Dr. Alexandro Baldassin (Suplente) - UNESP Rio Claro

[1]Suporte financeiro: CAPES 2011–2013

Abstract

Virtual machines have several applications in computer systems. The majority of virtual machine implementations is based on emulation, which can be implemented in several ways. The main techniques are interpretation and binary translation. The interpretation consists in a cycle of operations: an instruction fetch, its decoding and execution. The dynamic binary translation translates a source instruction block to a target instruction block, saving the translated code for reuse in order to decrease the fetch and decode costs.

The shape of the code region to be executed is important to dynamic optimizations. This code region is a compilation unity and it is where optimizations are applied, the virtual machine is responsible for building this region in binary translation. The region's size, the code frequency, the code duplication and the optimization cost are some factors which affect the techniques' choice. There are several region formation techniques, with different motivations and features. Implementing one of these techniques in a virtual machine is a complex task. Up to now there was no work in the literature presenting a comparison study involving several techniques. In order to perform the comparison without the need of actually implementing all techniques in a virtual machine, we created an infrastructure called Region Appraise Infrastructure (RAIn). By using RAIn, it was possible to achieve precise information about the techniques dynamic profile.

The main goals of this work were to study and to compare the region formation techniques to get a fair evaluation among them. Besides that, we included the requisite of comparing the techniques applying representative benchmarks of user's applications simulating a real environment. As result, we created a new technique named Last Executing Function (LEF), with the goal of achieving a better performance in such benchmarks.

Resumo

Máquinas virtuais têm diversas aplicações relacionadas a sistemas de computação. A maior parte das implementações de máquinas virtuais são baseadas em emulação, que pode ser implementada de diversas maneiras. As principais técnicas são a interpretação e a tradução de binários. A interpretação envolve um ciclo de operações: busca de uma instrução, sua decodificação e, enfim, a execução da operação necessária. Já a tradução dinâmica de binários traduz um bloco de instruções fonte para um bloco de instruções alvo, guardando o código traduzido para reuso, a fim de diminuir os custos de busca e de decodificação de instruções.

O formato da região de código a ser executado é um aspecto importante para a otimização dinâmica. Essa região de código é uma unidade de compilação e é nela que serão aplicadas otimizações. Na tradução dinâmica de binários, essa região é formada pela máquina virtual. O tamanho das regiões criadas, a frequência de execução do trecho de código, a duplicação de código e o custo de otimização são alguns dos fatores que afetam a escolha de uma técnica de formação de regiões. Há várias técnicas de formação de regiões, com diversas motivações e características. O trabalho de implementação de uma técnica de formação de regiões em uma máquina virtual é complexo, por isso até o momento não encontramos na literatura um trabalho realizando um estudo comparativo entre diversas técnicas. Para poder realizar a comparação sem precisar implementar diretamente todas as técnicas em uma máquina virtual, desenvolvemos uma infraestrutura chamada *Region Appraise Infrastructure* (RAIn). Esta infraestrutura nos possibilitou obter informações precisas sobre o perfilamento dinâmico das técnicas de formação de regiões.

O objetivo deste trabalho foi investigar e comparar as técnicas de formação de regiões existentes para obter uma avaliação justa entre elas. Além disso, colocamos o requisito de comparar as técnicas usando-se *benchmarks* representativos de aplicações de usuários que contenham instruções do sistema simulando um ambiente real. Como resultado, propusemos uma nova técnica de formação de regiões chamada *Last Executing Function* (LEF), visando melhorar o desempenho nesse tipo de *benchmark*.

Agradecimentos

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio à pesquisa realizada.

Agradeço ao Professor Doutor Sandro Rigo, pela oportunidade e confiança depositada, pela paciência, pela ajuda prestada durante todo o trabalho e por me guiar durante todo o processo.

Agradeço ao Professor Doutor Edson Borin, pelas discussões e ideias que foram de grande importância a esta pesquisa, pela constante ajuda e também pela cobrança.

Agradeço aos meus pais por todo o apoio que me deram. Ao meu pai Antonio Carlos Zinsly, por tentar me ajudar o tempo todo com o que pôde e por sempre se preocupar comigo. E principalmente à minha mãe, Petronilha Moreira Zinsly, pela enorme ajuda e por toda a fé depositada em mim, que tornaram meu caminho bem mais fácil.

Agradeço aos meus amigos e familiares, pela amizade e boa vontade. Aos que estavam perto por tornarem meus dias mais agradáveis e felizes. Aos que estavam longe por no fim se manterem por perto e sempre à disposição.

Por fim, agradeço aos meus colegas do LSC pela convivência e pela grande ajuda durante o trabalho, por estarem sempre dispostos a ajudar até nas coisas mais simples.

Sumário

Abstract **ix**

Resumo **xi**

Agradecimentos **xiii**

1 Introdução **1**

2 Trabalhos Relacionados **4**

3 Comparação Entre Técnicas de Formação de Região **15**
3.1 Técnicas de formação de regiões 16
3.2 Técnica LEF 19

4 Infraestrutura **21**

5 Resultados Experimentais **25**
5.1 Métricas 25
5.2 SPEC CPU 2006 27
5.3 SYSmark 33
5.4 Segmentação de Usuário e Sistema 39

6 Conclusões **45**

Referências Bibliográficas **48**

Lista de Tabelas

2.1 Tradutores de binários. 14

5.1 Comparação das técnicas no SPEC CPU 2006. 31
5.2 *Benchmarks* do SYSmark utilizados. 34
5.3 Comparação das técnicas no SYSmark. 37

Lista de Figuras

3.1 Exemplo de formação de região. 16
3.2 Superblocos do NET. 17
3.3 Funcionamento da LEF. 20

4.1 Exemplo do RAIn. 23

5.1 Número de regiões criadas pelas técnicas nos *benchmarks* do SPEC. 27
5.2 Cobertura sobre as instruções dos *benchmarks* do SPEC pelas regiões formadas. 28
5.3 90% *Cover Set* obtido pelas técnicas no SPEC. 29
5.4 *Completion Ratio* das técnicas no SPEC. 30
5.5 Duplicação de código causada pela *Trace Tree*. 31
5.6 Duplicação de código causada pelas técnicas nos *benchmarks* do SPEC. . . 32
5.7 Tamanho dinâmico médio das regiões criadas pelas técnicas no SPEC. . . . 33
5.8 Número de regiões criadas pelas técnicas. 35
5.9 Cobertura sobre as instruções pelas regiões formadas. 36
5.10 90% *Cover Set* obtido pelas técnicas. 37
5.11 *Completion Ratio* das técnicas. 38
5.12 Duplicação de código causada pelas técnicas. 39
5.13 Duplicação de código causada pela *Trace Tree*. 40
5.14 Tamanho dinâmico médio das regiões criadas pelas técnicas. 41
5.15 *Completion Ratio* obtido pela NET com e sem segmentação. 41
5.16 *Completion Ratio* obtido pela MRET2 com e sem segmentação. 42
5.17 *Completion Ratio* obtido pela LEI com e sem segmentação. 42
5.18 Duplicação de código criado pela NET com e sem segmentação. 43
5.19 Duplicação de código criado pela MRET2 com e sem segmentação. 43
5.20 Duplicação de código criado pela LEI com e sem segmentação. 44
5.21 Duplicação de código criado pela *Trace Tree* com e sem segmentação. . . . 44

Capítulo 1

Introdução

A virtualização proporciona um meio de melhorar a flexibilidade de sistemas de computação. Quando um sistema (ou subsistema) é virtualizado, sua interface e todos os seus recursos são mapeados na interface e nos recursos do sistema real. A virtualização pode ser aplicada não só aos subsistemas, mas a toda máquina. Uma máquina virtual é implementada adicionando-se uma camada de *software* à máquina real para dar suporte à arquitetura desejada da máquina virtual [45]. Em geral, uma máquina virtual pode contornar restrições de compatibilidade da máquina real e restrições de recursos de *hardware* para permitir um maior grau de portabilidade e flexibilidade de *software*.

Máquinas virtuais têm sido aplicadas em vários contextos para a resolução de problemas relacionados a sistemas de computação. Em linguagens de programação, as máquinas virtuais permitem uma grande portabilidade para que os programas sejam executados em diversas plataformas. Através de máquinas virtuais, múltiplos sistemas operacionais podem ser executados em um mesmo sistema de um modo eficaz. Emuladores permitem que programas compilados para uma arquitetura executem em outra com um conjunto de instruções diferentes. Na área de arquitetura de computadores, por exemplo, máquinas virtuais são utilizadas para permitir otimizações dinâmicas através de tradução dinâmica de binários.

A maioria das implementações de máquinas virtuais é baseada em emulação. Emulação é o processo que implementa a interface e as funcionalidades de um sistema (ou subsistema) em um sistema com interface e funcionalidades diferentes. A emulação de um conjunto de instruções pode ser feita usando uma variedade de métodos que requerem diferentes quantidades de recursos computacionais e oferecem diferentes características de desempenho e portabilidade. As principais técnicas são a interpretação e a tradução de binários.

A interpretação envolve um ciclo de operações: busca de uma instrução, sua decodificação e, enfim, a execução das operações necessárias. Em seguida começa a busca por outra instrução, tudo em *software*. É uma técnica relativamente simples, o que permite a implementação em uma linguagem de alto nível gerando uma maior portabilidade. Porém, geralmente é um processo lento, pois a busca de cada instrução é acompanhada de uma decodificação, o que pode tornar inviável sua utilização para algumas aplicações. A tradução de binários tenta amortizar os custos da busca e da decodificação traduzindo um bloco de instruções fonte para um bloco de instruções alvo, guardando o código traduzido para um reuso futuro. Ao contrário da interpretação, a tradução de binários tem um alto custo inicial de tradução e um baixo custo de execução. O processo de tradução pode ser feito estática ou dinamicamente; porém, nem sempre é possível fazer estaticamente, pois certas aplicações geram parte do código dinamicamente. A tradução dinâmica de binários provê, na maioria dos casos, uma execução mais rápida que a interpretação.

O formato da região de código a ser executado é um aspecto importante na otimização dinâmica de código. A máquina virtual deve formar uma região de código para o otimizador analisar, essa região de código é uma unidade de compilação e é nela que serão aplicadas otimizações. Em máquinas virtuais para linguagens de alto nível o formato da região é muitas vezes definido pelo conjunto de instruções de uma rotina no programa que executa a máquina virtual. Para máquinas virtuais que executam uma arquitetura de processadores reais a tarefa de criação de regiões é mais complexa, pois o programa que executa a máquina virtual não tem informações sobre a estrutura das aplicações executadas.

Existem diversos fatores que afetam a escolha de políticas para a formação de regiões. O tamanho das regiões criadas, a frequência de execução do trecho de código, a duplicação de código e o custo de otimização são alguns desses fatores. Há várias técnicas de formação de regiões existentes, com diversas motivações e características. O objetivo deste trabalho foi levantar e avaliar as técnicas existentes, fazendo uma comparação justa entre elas. Implementar uma técnica de formação de regiões em uma máquina virtual é complexo, não se encontra na literatura nenhum estudo comparativo entre diversas técnicas, o que pode tornar a escolha de uma política de formação de regiões uma tarefa menos fundamentada do que deveria.

Para poder realizar a comparação sem precisar implementar diretamente todas as técnicas em uma máquina virtual, uma infraestrutura para simular o funcionamento das técnicas foi desenvolvida. Essa infraestrutura é chamada de RAIn (*Region Appraise In-*

frastructure), ela consegue prover informações precisas acerca do perfilamento dinâmico sem a necessidade de implementar as técnicas em um sistema real tornado possível compará-las para realizar esta pesquisa.

Os trabalhos relacionados apresentam as técnicas de formação de regiões avaliadas em *benchmarks* do tipo SPEC [26]. Esses programas são adequados para a avaliação de desempenho de processadores, porém não são representativos da real carga de trabalho de boa parte dos usuários de virtualização. Também foi um objetivo deste trabalho comparar as técnicas de formação de regiões existentes usando-se *benchmarks* representativos de aplicações de usuários que contenham instruções de sistema simulando um ambiente real. Por fim, também foi proposta uma nova técnica de formação de regiões para compilação dinâmica em máquinas virtuais, visando melhorar o desempenho nos *benchmarks* representativos.

As contribuições deste trabalho foram:

- Levantamento das políticas de formação de regiões existentes.
- Criação de uma infraestrutura para a simulação das técnicas.
- Comparação justa entre as principais técnicas existentes.
- Avaliação das técnicas em ambientes representativos para usuários de virtualização.
- Uma nova técnica de formação de regiões que apresenta um bom desempenho em *benchmarks* com aplicações representativas para usuários de virtualização.

O texto está organizado da seguinte forma: O Capítulo 2 apresenta os trabalhos relacionados a tradutores dinâmicos de binários e virtualização, o Capítulo 3 detalha o processo de formação de regiões e apresenta as técnicas existentes, incluindo a nova técnica proposta, o Capítulo 4 descreve a ferramenta RAIn e apresenta a infraestrutura deste trabalho, o Capítulo 5 traz os resultados dos experimentos realizados através de um estudo comparativo e o Capítulo 6 resume nossas conclusões.

Capítulo 2

Trabalhos Relacionados

Neste capítulo apresentamos os trabalhos relacionados a tradutores dinâmicos de binários e virtualização, trazendo uma visão geral do estado da arte. Algumas das técnicas de formação de regiões surgiram a partir de alguns dos trabalhos a seguir, as técnicas serão discutidas no Capítulo 3. Muitos autores usam o termo traço para descrever regiões que podem ser mais bem descritas com o termo superblocos. Usaremos o termo traços para sequências de blocos básicos (sequência de instruções que não possui desvios) que podem ter mais de um ponto de entrada, utilizaremos o nome superblocos para descrever traços com um único ponto entrada. Todo superbloco é um traço, mas nem todo traço é um superbloco [45].

Cmelik e Keppel [14] descrevem o Shade. O Shade é uma ferramenta que combina simulação eficiente de conjunto de instruções e geração de traços (sequência de instruções). Ele funciona como um tradutor dinâmico e utiliza *cache* de código para amortizar o custo da tradução. O Shade salva informações dos traços durante a execução e integra a simulação com a geração dos traços. A estratégia de criação de traços pode variar dinamicamente, isso é controlado por um analisador. As informações armazenadas são somente aquelas que o analisador precisa, caso pouca informação sobre os traços seja necessária a sobrecarga adicionada é menor.

Ebcioglu e Altman [19] apresentam o Daisy. O Daisy *(Dynamically Architected Instruction Set from Yorktown)* é um tradutor dinâmico voltado para arquiteturas VLIW. Ele emula outras arquiteturas fazendo a tradução para VLIW em *software*. O principal objetivo do Daisy é alcançar altos níveis de paralelismo em nível de instrução mantendo a sobrecarga de desempenho a menor possível. Durante a execução, as operações são convertidas em primitivas RISC, então o Daisy substitui as primitivas por uma instrução VLIW. A região utilizada pelo Daisy são páginas. O código é armazenado em páginas na

memória virtual e cada página se torna uma unidade de tradução. As páginas contêm pontos de entrada que dependem dos valores nos registradores de controle de fluxo da máquina emulada.

Hookway e Herdeg [28] e Chernoff *et al.* [12] descrevem como o *software* Digital FX!32 funciona. O FX!32 é um *software* que combina emulação com tradução de binários, isto permite que aplicações de 32 *bits* que executem na plataforma x86 usando o Windows NT 4.0 possam ser instaladas e executadas em plataformas Alpha também rodando Windows NT 4.0. Os objetivos principais do *software* são o de alcançar uma execução transparente de aplicações x86 de 32 *bits* e o de manter aproximadamente o mesmo desempenho das aplicações na plataforma x86 quando executadas na plataforma Alpha. A unidade de tradução utilizada pelo FX!32 são rotinas. Elas são formadas por coleções de traços e se aproximam de uma rotina real do programa.

Bala, Duesterwald e Banerjia [4] descrevem a implementação do Dynamo, que é um sistema de otimização dinâmica feito inteiramente em *software*. É um sistema transparente - não precisa de uma fase preparatória na compilação - não precisa de assistência de um programador, e até mesmo binários nativos estaticamente otimizados podem ser acelerados com o Dynamo. As otimizações não só melhoram a execução do programa como também compensam a sobrecarga das operações do próprio Dynamo. Como resultado, foi alcançado um ganho de desempenho médio de 9% em comparação à execução diretamente no processador, chegando a mais de 22% em alguns casos.

Chen *et al.* [11] apresentam o Mojo. O Mojo é um sistema de otimização dinâmica em *software* para a arquitetura x86. Ele tem suporte a aplicações *multi-thread*, não precisa de suporte do sistema operacional e utiliza tratamento de exceções. Ele não se mostrou muito eficaz na maioria das aplicações testadas. Como região o Mojo utiliza traços, chamados de caminhos pelos autores. Ao executar, o Mojo procura o apontador da instrução na *cache* de traços, se existe uma entrada na *cache* correspondente ao ponteiro, este caminho é executado diretamente. Quando o sistema encontra blocos básicos que formam um caminho quente (um caminho executado frequentemente), o Mojo constrói um traço com eles, otimiza-o e o armazena na *cache* de traços.

Gschwind *et al.* [23] apresentam o BOA. O BOA *(Binary-translation Optimized Architecture)* é um tradutor dinâmico de binários focado em otimização dinâmica. Ele é inspirado no Daisy, mas sua prioridade é aumentar a frequência do processador e não diminuir os ciclos por instrução. O BOA faz uma tradução de binários transparente e utiliza traços como região. Ele usa informações obtidas dinamicamente para gerar traços

nos melhores caminhos do programa. O BOA realiza otimizações nos traços e faz o escalonamento do código para melhorar o desempenho.

Merten *et al.* [36] apresentam um mecanismo em *hardware* para coletar e aplicar código otimizado em tempo de execução. Ele utiliza o Hot Spot Detector, um mecanismo que determina, durante a execução, as instruções de desvios mais frequentemente executadas enquanto faz os perfis delas. São utilizados traços como região, eles são formados por uma unidade de geração de traços nos pontos quentes do código. O *hardware* desenvolvido faz *code straightening*, *loop unrolling*, *partial function inlining* e *branch promotion* [2] para melhorar o desempenho da busca de instruções.

Ung e Cifuentes [50] [51] apresentam o UQDBT. O UQDBT *(University of Queensland Dynamic Binary Translator)* é um tradutor adaptável à maquina. Ele é baseado no *framework* UQBT e tem suporte para diferentes máquinas fonte e destino. As otimizações feitas pelo UQDBT são genéricas e podem ser aplicadas a vários tipos de máquinas. A região utilizada pelo UQDBT são traços. Os caminhos quentes são encontrados através de *edge weight profiling*, são transformados em traços e armazenados na *hot cache*.

Patel e Lunetta [40] descrevem o rePLay. O rePLay é um *framework* em *hardware* com suporte a otimizações dinâmicas. Ele é uma microarquitetura que combina um mecanismo de execução de alto desempenho com um mecanismo de otimização programável. Isto permite que as otimizações ocorram simultaneamente com a execução. A região utilizada pelo rePLay é chamada de *frame* (Seção 3.1). O rePLay também utiliza um mecanismo de recuperação, caso seja detectado que dentro de algum *frame* um desvio deveria ter sido tomado, a execução daquele *frame* é anulada.

Slechta *et al.* [44] avaliam o desempenho de otimizações em regiões de microoperações x86, onde as regiões são maiores que blocos básicos. Eles utilizaram a infraestrutura rePLay que contêm suporte em *hardware* para realização de otimizações e suporte em *hardware* para especulação. Para examinar os efeitos da otimização dinâmica em nível de microoperações foi usado um conjunto de apenas sete otimizações. Eles estudaram os efeitos da otimização usando simulação dirigida a traços através de *benchmarks* SPECint 2000 e aplicações x86 comerciais. Os resultados indicaram que, em média, a otimização com o rePLay reduziu o número de microoperações em 21%. Os dados contribuíram com os trabalhos sobre o rePLay avaliando o impacto da otimização em nível de microoperações no contexto do conjunto de instruções x86.

Cifuentes, Lewis e Ung [13] descrevem o Walkabout. O Walkabout é um *framework* de

tradução dinâmica de binários inspirado no UQBT. Ele tem como foco ser direcionável: poder trocar as máquinas fonte e destino por outras da mesma família. Como região, o Walkabout utiliza traços, chamados pelos autores de fragmentos. Os traços são gerados com os caminhos quentes e guardados na *cache* de fragmentos. O Walkabout tem uma ferramenta chamada *PathFinder* que faz otimizações no código, como por exemplo: *layout* de código, *inlining caching* [17], ligação de blocos básicos e de traços.

Baraz *et al.* [6] descrevem os aspectos do IA-32 Execution Layer (IA-32 EL), que executa aplicações IA-32 em sistemas Intel Itanium. O IA-32 EL é um *software* de tradução dinâmica de binários de duas fases, a primeira traduz as instruções IA-32 para instruções Itanium e a segunda retraduz as partes quentes e faz otimizações. O *software* roda tanto em Windows quanto em Linux. O IA-32 EL tem como principais características a grande quantidade de informações coletadas dinamicamente durante a primeira fase de tradução, o uso dessas informações para a segunda fase, um sistema operacional binário independente para traduzir aplicações em múltiplos sistemas operacionais e um mecanismo para uma manipulação precisa de exceções. Como resultado o IA-32 EL alcançou um nível de desempenho de 65% do desempenho nativo dos *benchmarks*. Como região o IA-32 EL utiliza traços, chamados de hiperblocos, que contém em média 20 instruções.

Bruening, Garnett e Amarasinghe [9] apresentam um *framework* para a implementação de análises e otimizações dinâmicas baseado no DynamoRIO. O DynamoRIO é um otimizador dinâmico que tem como objetivo observar e potencialmente manipular cada instrução antes de sua execução. Como região, o DynamoRIO utiliza tanto blocos básicos como superblocos, que são traços que contém uma única entrada. Ele copia blocos básicos para uma *cache* de códigos e os executa nativamente, ele junta dois blocos básicos subsequentes caso eles estejam ligados por um desvio direto. No caso de desvios indiretos, não é possível ligá-los da mesma forma, o DynamoRIO utiliza então uma tabela *hash* de *lookup*. Para melhorar a eficiência dos desvios indiretos os blocos básicos são agrupados em um traço. As *caches* de código do DynamoRIO consistem de uma *cache* de blocos básicos, de uma tabela de *lookup* para desvios indiretos e de uma *cache* de traços.

Chen *et al.* [10] apresentam um sistema de obtenção de informações necessárias para detectar oportunidades de otimizações dinâmicas e implantá-las. Como região eles utilizam traços, eles examinaram quanto tempo de execução pode ser reduzido pelos traços gerados dinamicamente. Utilizaram *hardware* para coletar dados de desempenho, mas para a detecção de código quente utilizaram *software*. Como resultado conseguiram capturar em média 60% do tempo de execução e o sistema de detecção chegou a uma sobrecarga média de 2% a 4%. Além disso, o sistema consegue proporcionar informações sobre

problemas de desempenho para ajudar na otimização dos traços selecionados.

Dehnert *et al.* [16] mostram a estrutura do *Code Morphing software* (CMS). O CMS é a camada de *software* do microprocessador Crusoe, que implementa a arquitetura x86. O sistema tem uma abordagem única: ele utiliza um conjunto de instruções interno independente do externo. Isso permite uma implementação simples e compacta, com liberdade para modificar o conjunto de instruções interno entre gerações. Especialmente, o CMS expõe várias questões que haviam recebido pouca atenção na literatura, como interrupções e exceções, entrada e saída, código auto modificável e acesso direto à memória. O CMS visa implementar fielmente toda a arquitetura x86; também visa prover um bom desempenho para uma grande variedade de sistemas e aplicações, desde jogos e multimídia até aplicações de servidores. Um dos objetivos do *software* é fazer com que o CMS não dependa de informações ou outra assistência do sistema. Krewell [31] descreve o CMS no processador Efficeon. Os autores não dão detalhes sobre o formato da região, apenas sobre a ordem de grandeza (aproximadamente 100 instruções x86). Borin *et al.* [8] estendem a infraestrutura do CMS no Efficeon e mencionam que o formato das regiões é um CFG (*Control Flow Graph*) genérico, que pode incluir laços e chamadas de funções.

Kim e Smith [30] pesquisam tradução dinâmica de binários para uma máquina virtual com projeto integrado de *hardware* e *software* que utiliza um processador superescalar distribuído. Eles focam na simplicidade para a tradução de binários, a única otimização que fazem é *code straightening*, em que os blocos básicos são alocados estaticamente de acordo com a ordem em que são mais comumente executados dinamicamente.

Lu *et al.* [34] [33] descrevem o ADORE. O ADORE *(ADaptive Object code REoptimization)* é um sistema de otimização dinâmica que utiliza o monitor de desempenho de *hardware (hardware performance monitor - HPM)* do Itanium para criar perfis e selecionar as regiões. As regiões utilizadas pelo ADORE são traços, a seleção de traços é baseada nas amostras de traços de desvios coletadas pelo HPM. O ADORE é implementado como uma biblioteca compartilhada do Linux para IA64 e pode ser ligada automaticamente a aplicação durante a inicialização. Durante a execução, existem duas *threads* rodando no ADORE, uma executando o código sem modificações e outra encarregada das otimizações dinâmicas e da seleção dos traços. Como resultado o ADORE alcançou um ganho de desempenho de 57% sobre compiladores que não fazem *prefetching* e de 20% sobre os que fazem.

Scott *et al.* [43] apresentam o Strata. O Strata é um sistema de tradução dinâmica em *software*. Ele provê um *framework* para o desenvolvimento de tradutores dinâmicos

para uma variedade de arquiteturas. Ele também permite o reuso de código através de composição, ou seja, pesquisadores podem usar o trabalho de outros para melhorar seus tradutores dinâmicos. A região utilizada pelo Strata é chamada de fragmento. Um fragmento é uma sequência de código em que desvios condicionais podem aparecer somente no fim. Durante a execução, o Strata vai preenchendo os fragmentos buscando, decodificando e traduzindo as instruções uma a uma até alcançar uma condição de término de fragmento, esta condição depende do tradutor a ser implementado. Usualmente a condição é alcançada quando um desvio é encontrado, mas os tradutores podem, por exemplo, formar fragmentos em que uma única instrução é emulada.

Almog *et al.* [3] estudam otimizações dinâmicas baseadas em *hardware* dentro de microarquiteturas de alto desempenho. Eles usaram a infraestrutura PARROT, que é uma microarquitetura baseada em *cache* de traços e com um mecanismo dissociado para execução de códigos frequentes. O estudo é focado no desempenho e no potencial para poupar energia da otimização dinâmica feita nos traços mais frequentes na execução do programa. O sistema PARROT emprega otimização dinâmica dissociada. Dissociar a otimização da execução permite executar vários ciclos sem afetar o progresso da execução. Os traços utilizados no PARROT têm somente um fluxo de controle, semelhante ao *frame* do rePLay, o que permite uma execução atômica em conjunto a um mecanismo de recuperação.

Fahs *et al.* [20] estudam o potencial da otimização dinâmica, mas especificamente procuram dizer qual o desempenho em potencial de um otimizador dinâmico baseado em traço de execução. Eles fizeram uma análise entre os conjuntos de instruções das arquiteturas SPARC e x86 em um ambiente livre de sobrecarga de desempenho. Eles estudam os processos de: formação de regiões, otimização, armazenamento de regiões otimizadas e envio de regiões otimizadas em *cache*, durante a execução. Estes são processos comuns a várias abordagens de otimizações dinâmicas tanto em *software* quanto em *hardware*. O artigo é dividido em dois estudos, o primeiro é examinar o potencial da aceleração em função do tamanho dos traços utilizados. Nesse aspecto os resultados mostraram que traços executados atomicamente permitem uma otimização mais profunda que traços não atômicos, mas na melhor das hipóteses oferecem uma aceleração de duas vezes, mesmo em uma situação ideal. O segundo estudo avalia o impacto da seleção de traços no desempenho, cobertura e tamanho da *cache* de código. Como resultado foi concluído que a seleção parcial de traços é um ingrediente importante para o desempenho do sistema.

Bellard [7] descreve o QEMU. O QEMU é uma máquina virtual que utiliza uma tradução dinâmica com grande portabilidade. Ele é usado principalmente para rodar sistemas operacionais emulando todo o sistema, mas ele também tem um modo de usuário

Linux específico para rodar o Linux sem precisar iniciar uma máquina virtual completa. O QEMU usa código gerado pelo *GNU C Compiler* (GCC) e trabalha com microoperações. O artigo descreve a tradução utilizada pelo QEMU e os detalhes da implementação. Como resultado, o QEMU se mostrou 30 vezes mais rápido que o Bochs. Como região, o QEMU utiliza blocos de tradução (*Translated Block - TB*), essa região acaba quando um desvio ou outra instrução que modifica o estado estático da CPU é encontrado, de forma que não é possível ser deduzido em tempo de tradução. O QEMU guarda os TBs em uma *cache* de 16 MB chamada de *Translation Cache*.

Luk *et al.* [35] descrevem o Pin, um sistema em *software* que realiza instrumentação de binários em tempo de execução em aplicações Linux. O objetivo do Pin é prover uma plataforma de instrumentação para construir uma grande variedade de ferramentas de análise para múltiplas arquiteturas. Enfatizando facilidade de uso, portabilidade, transparência, eficiência e robustez. O Pin utiliza uma API de ferramentas de instrumentação escrita em C/C++ e utiliza um compilador JIT *(Just-In-Time)* para realizar compilação dinâmica e como região utiliza traços. Os traços no Pin terminam em alguma das seguintes condições: um desvio incondicional, após um pré-determinado número de desvios condicionais ou após um pré-determinado número de instruções. Os autores argumentam que o Pin se mostrou 3.3 vezes mais rápido que o Valgrind e duas vezes mais rápido que o DynamoRIO quando utilizam instrumentação. Wallace e Hazelwood [52] descrevem o projeto e implementação do SuperPin. O SuperPin é a versão paralelizável do Pin. Para diminuir a sobrecarga da instrumentação o SuperPin divide o código em grandes porções que são executadas paralelamente. Dependendo da quantidade de núcleos do processador, do tamanho da memória e do tamanho de cada porção é possível alcançar uma velocidade de execução próxima ao alcançado pelo código sem instrumentação.

Zhang, Calder e Tullsen [56] descrevem o Trident, um *framework* de otimização dinâmica, ele apresenta mais suporte em *hardware* para o monitoramento dinâmico da execução e a habilidade de executar várias *threads*. O sistema é voltado para otimizações dinâmicas para processadores *multithreaded*. A otimização é dirigida a eventos, o desempenho e o monitoramento de eventos podem ser feitos sem causar sobrecarga de *software* no sistema. Como resultado, o ganho de desempenho alcançado pelas otimizações usando o Trident chegou a 20% em média, as demonstrações foram baseadas em *software*. Como região, o Trident utiliza traços, ele utiliza uma *thread* para detectar código quente e forma os traços quando uma sequência de blocos básicos é quente. Caso haja um desvio incondicional, nem todos os blocos farão parte do traço.

Adams e Agesen [1] descrevem o VMM *(Virtual Machine Monitor)* da VMware Works-

tation's feito para a arquitetura x86. Eles também descrevem um VMM que utiliza suporte em *hardware* e comparam ambos. O VMM utiliza blocos básicos como unidades de tradução, a menos que o bloco tenha mais de 12 instruções. Ele não realiza otimizações, pois assume que os desenvolvedores do sistema operacional já otimizaram o código e que uma simples tradução de binários encontraria poucas novas oportunidades. O VMM em *software* se saiu melhor do que o que utiliza *hardware*, exceto nas aplicações que utilizam muitas chamadas de sistema.

Gal, Probst e Franz [22] apresentam o Hotpath VM, um compilador JIT (Just in Time) que é complemento do Jam VM, uma máquina virtual para dispositivos embarcados. O compilador roda em apenas 150 kB e para códigos normais alcança um ganho de desempenho similar ao alcançado por compiladores mais pesados, ou seja, que precisam de mais espaço para rodar. O Hotpath VM utiliza compilação baseada em traços usando *Static Single Assignment Form*(SSA)[2].

Hu e Smith [29] estudam e desenvolvem sistemas de tradução de binários utilizando projeto integrado de *hardware* e *software* para tentar reduzir o tempo de inicialização em máquinas virtuais. Eles propõem dois mecanismos de assistência em *hardware* visando primeiramente a tradução de blocos básicos. Os mecanismos são um decodificador *dual mode* no *frontend* do *pipeline* e uma unidade funcional de propósito específico adicionada ao *backend* do processador.

Sridhar *et al.* [47] [46] apresentam o HDTrans. O HDTrans é um tradutor dinâmico *open source* para a arquitetura IA-32. Ele é focado em simplicidade e tenta amortizar ao máximo o custo de otimizações evitando muita complexidade. O HDTrans não utiliza geração de código e utiliza uma tabela de tradução, o tradutor é dirigido a esta tabela. Os autores argumentam que ele obteve um desempenho melhor que o DynamoRIO e o Pin. Como região o HDTrans utiliza superblocos, de forma semelhante ao Dynamo. Ele também utiliza linearização de traços de forma lenta utilizando uma heurística.

Neelakantam *et al.* [38] demonstram os benefícios da atomicidade de *hardware* no contexto da Java VM. Execução atômica é executar uma região do código completamente como se todas as operações desta região acontecessem em um instante. É este o princípio do artigo para melhorar a eficácia das otimizações existentes em compilação e para simplificar a implementação de otimizações adicionais. O trabalho usa a atomicidade em *hardware* para otimizações especulativas. Basicamente a execução atômica simplifica a implementação de otimizações especulativas de três modos: o *hardware* mantém o estado necessário para a recuperação a partir de otimizações especulativas, a atomicidade per-

mite análises e otimizações para ignorar código frio (código pouco executado) enquanto otimiza código quente, e o *hardware* isola a execução das outras *threads*. O artigo apresenta um trabalho semelhante ao rePLay mas em *software*. A região utilizada é chamada de região atômica. As regiões atômicas formam um subgrafo com uma única entrada e múltiplas saídas, contendo um fluxo de controle interprocedural e arbitrário. Os resultados mostraram uma média de aceleração de 12%, além de uma redução no número de microoperações executadas pelo processador.

Nethercote e Seward [39] descrevem o Valgrind, uma infraestrutura de instrumentação dinâmica de binários. Ele constrói ferramentas de análise dinâmica de binários com foco em ferramentas *shadow values*, essas ferramentas sombreiam (*shadow*) cada valor de memória ou registrador que diga algo sobre a análise desejada. O artigo apresenta as características das ferramentas *shadow values*, mostra como ter suporte a *shadow values* na instrumentação dinâmica de binários. Superblocos são utilizados como unidade de tradução.

Wang *et al.* [53] descrevem o StarDBT. O StarDBT é um sistema de tradução dinâmica de binários multiplataforma voltado para a arquitetura Intel. Ele roda em Windows e Linux, além de rodar outros sistemas operacionais comerciais no *virtual machine monitor* em nível de sistema. Para ter suporte a multiplataforma, o StarDBT tem um módulo genérico e um módulo dependente da plataforma utilizada. Uma API interna faz a interface entre estes módulos. Como região o StarDBT utiliza traços. Os traços são criados somente em partes quentes do código, o resto é traduzido sem otimização e rapidamente. São feitas otimizações simples nos traços, tais como *layout* de código e eliminação parcial de redundância. O StarDBT se saiu bem em comparação aos outros DBTs. Ele também foi testado utilizando aplicações comuns do Windows e obteve resultados satisfatórios.

Watson [55] apresenta o VirtualBox. O VirtualBox é um *software* de virtualização que visa criar ambientes para instalação e utilização de sistemas operacionais distintos. Ele suporta as arquiteturas x86 e IA64 e é distribuído pela Oracle. Tem duas versões: uma delas é *open source* e não tem todas as funcionalidades, a outra é grátis para uso pessoal e pago para uso empresarial. O autor não descreve as técnicas de formação de regiões utilizadas.

Guan *et al.* [24] apresentam o CoDBT. O CoDBT *(hardware-software Collaborative Dynamic Binary Translator)* é um sistema de tradução dinâmica de binários genérica com projeto integrado de *hardware* e *software*. Ele utiliza um tradutor dinâmico completo em *software* em conjunto com um acelerador em *hardware*. Os dois podem executar

as mesmas funcionalidades alternadamente ou em paralelo. O CoDBT ainda possui um controlador de memória e um barramento no *chip* para fazer a comunicação entre os três componentes. Como região o CoDBT utiliza blocos básicos.

A Tabela 2.1 sumariza os tradutores de binários indicando quais são as máquinas traduzidas por cada trabalho, para quais arquiteturas elas são traduzidas e qual a região formada por eles. Não estavam disponíveis informações sobre para quais arquiteturas são utilizados os trabalhos rePLay, Kim e Smith, Trident e Hotpath VM.

Tabela 2.1: Tradutores de binários.

Trabalho	Traduz de	Para	Região
Shade	SPARC v8/v9, MIPS I	SPARC v8	Traços
Daisy	PowerPC	Arq. VLIW	Páginas
FX!32	x86	Alpha	Rotinas
Dynamo	HP PA-8000	HP PA-8000	Superblocos
Mojo	x86	x86	Traços
Boa	PowerPC	PowerPC	Traços
UQDBT	Adaptável	Adaptável	Traços
rePLay			*Frames*
Walkabout	Adaptável	Adaptável	Traços
IA-32 EL	x86	Itanium	Traços
DynamoRIO	x86	x86	Superblocos
CMS	x86	Crusoe	
Kim e Smith			Blocos Básicos
Adore	Itanium	Itanium	Traços
Strata	Adaptável	Adaptável	Fragmentos
PARROT	x86	x86	*Frames*
Fahs *et al.*	SPARC, x86	SPARC, x86	Traços
QEMU	x86, PowerPC, ARM, SPARC	x86, PowerPC, ARM, SPARC, Alpha, MIPS	*Translated Blocks*
Pin	x86, EM64T, Itanium, ARM	x86, EM64T, Itanium, ARM	Traços
Trident			Traços
Vmware	x86	x86	Blocos Básicos
Hotpath VM			Traços
HDTrans	x86	x86	Superblocos
Valgrind	x86, AMD64, PowerPC	A mesma arquiteruta	Superblocos
StarDBT	IA	IA	Traços
VirtualBox	IA	IA	
CoDBT	Várias	Várias	Blocos Básicos

Capítulo 3

Comparação Entre Técnicas de Formação de Região

Neste capítulo apresentamos o conceito de regiões e o que são políticas de formação de regiões. Discutimos as técnicas existentes atualmente e explicamos a técnica LEF, que foi proposta neste trabalho.

Normalmente, a unidade fundamental de compilação é uma função ou método; o processo de formação de regiões consiste em particionar o programa em um novo conjunto de unidades de compilação, que são chamadas de regiões [25]. São essas unidades de compilação que sofrerão otimizações.

São várias as vantagens de usar a abordagem de formação de regiões. O compilador tem total controle sobre o tamanho e o conteúdo das unidades de compilação, o que não acontece quando as unidades são funções. O uso de perfis de informação para a formação de regiões permite que o compilador selecione unidades de compilação que melhor reflitam o comportamento dinâmico do programa, possibilitando ao compilador a produção de um código mais otimizado.

A Figura 3.1 mostra um exemplo de formação de regiões em uma função: à esquerda está a função original e à direita está a função após aplicada uma técnica de formação de regiões. O trecho foi dividido em duas regiões (A e B); então foi retirado da região A o código x = a + b, que não tem relevância para esta região, e foi adicionado como código de compensação na transição da região A para a região B.

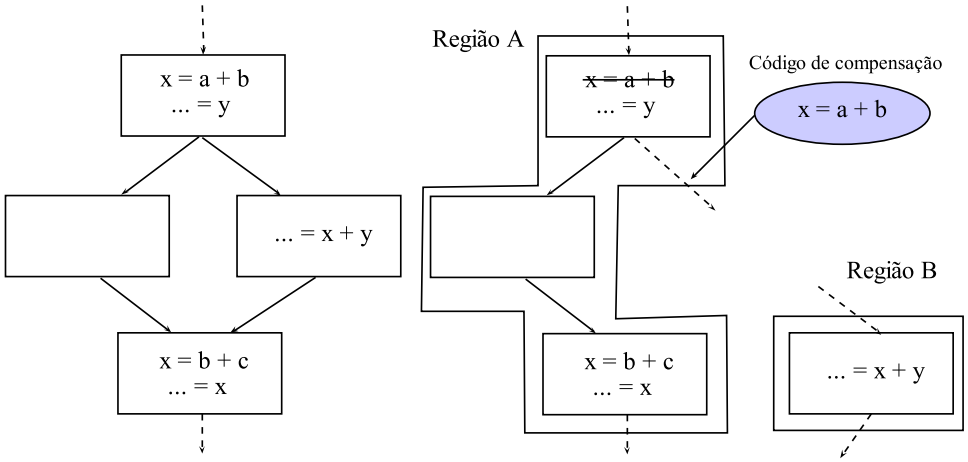


Figura 3.1: Programa original e após formação de regiões e compilação separada [48]

3.1 Técnicas de formação de regiões

A NET [18] *(Next-Executing Tail)* foi apresentada primeiramente pelos autores do Dynamo [4]. Inicialmente chamada de MRET (*Most Recently Executed Tail*), a técnica cria regiões no formato de superblocos. Ela consiste em associar um contador a potenciais inícios de superblocos, são eles alvos de desvios para trás ou alvos de saídas de superblocos já formados. O contador é incrementado sempre que a instrução associada a ele for executada. Após o contador passar um certo limite, a sequência de instruções a partir daquele ponto começa a ser gravada como um superbloco, que termina após encontrar outro salto para trás ou a região passar de um número de instruções estipulado. Já a MRET2 [54] foi implementada no StarDBT [53]. Ela consiste em executar a NET em duas fases. A MRET2 seleciona duas sequências de códigos a partir do mesmo início usando a técnica NET duas vezes, o caminho em comum entre as duas sequências forma o superbloco.

A Figura 3.2 (a) apresenta um código de exemplo para demonstrar a criação de um superbloco. Na Figura 3.2 (b) vemos o grafo de fluxo de controle *(Control Flow Graph - CFG)* daquele código. Finalmente na Figura 3.2 (c) temos a representação deste código em dois superblocos, T1 e T2, gravados pelo NET.

A LEI *(Last-Executed Iteration)* foi proposta por Hiniker, Hazelwood e Smith [27]. Ela seleciona superblocos cíclicos baseado em um *buffer* de histórico contendo os desvios tomados mais recentes. O superbloco só é formado depois de executado pré-determinadas vezes, a técnica se foca em evitar duplicação de laços aninhados nos superblocos. No

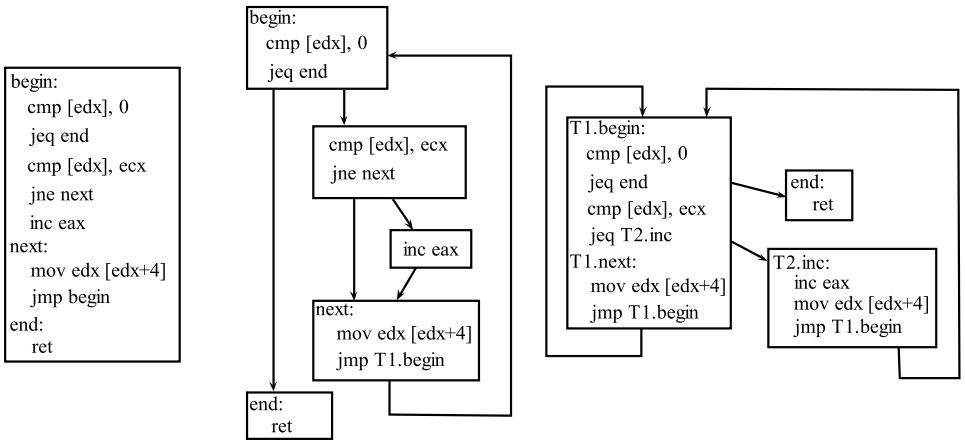


Figura 3.2: (a) Código de exemplo. (b) CFG do código. (c) Superblocos do NET.

mesmo artigo os autores também descrevem um algoritmo para combinar regiões que tenham caminhos sobrepostos e sejam executadas frequentemente em uma região maior. Este algoritmo não depende de como a região é formada, podendo ser aplicado ao próprio LEI, ao NET, entre outros. O algoritmo reduz o problema da separação de regiões e resulta em uma *cache* de código menor. Porém, o *buffer* de histórico necessário para a implementação do LEI causa uma grande sobrecarga no perfilamento que ofusca os potenciais benefícios [15].

A *Trace Tree* [1] foi descrita por Gal e Franz [21]. Uma árvore de traços é um grafo dirigido representando um conjunto de traços do programa. A construção das árvores de traço é feita sob demanda à medida que novos traços são formados, cada vez que ela é estendida, o grafo inteiro é recompilado. As árvores representam os laços do programa, a raiz é formada pela cabeça(ponto de entrada) de um laço. Instruções de alocação de memória ou o crescimento exponencial da região podem cancelar a construção de uma árvore. Caso uma saída lateral seja executada com frequência, a técnica expande a região incorporando um traço criado a partir dali na árvore.

Banerjia, Havanki e Conte [5] descrevem a técnica de formação de região chamada Treegion. As Treegions são formadas por árvores de decisão e servem para escalonar as instruções em tempo de compilação. Ela é um subgrafo do CFG do programa, o tamanho e o número de Treegions em um CFG é dado por sua topologia, não pelo seu perfil. A formação da região começa em cada nó de entrada do CFG, os nós encontrados são adi-

[1] árvore de traços em inglês

cionados a uma Treegion até que pontos de união sejam encontrados, estes se tornam as raízes de novas Treegions.

A região utilizada pelo rePLay [40] e o PARROT [3] é chamada de *frame*. O *frame* é uma região de código em que todos os desvios internos são eliminados de forma a deixar a região atômica, ou seja, *frames* têm somente um fluxo de controle.

Suganuma, Yasue e Nakatani [48] descrevem o projeto e implementação de uma técnica de compilação dinâmica baseada em formação de regiões. A região utilizada consiste em métodos onde todos os blocos básicos raramente executados são removidos. A técnica de formação consiste em marcar os blocos como raros ou não raros, isto é feito de acordo com heurísticas combinadas com perfis gerados dinamicamente, e propagar as marcações pelos outros blocos básicos do método. No artigo também são descritas outras duas otimizações relacionadas, que se aproveitam das regiões selecionadas. Como resultado, a técnica empregada teve um desempenho de 4% a 8% melhor comparado às técnicas tradicionais, chegando ao pico de 24%. A técnica alcançou em média uma redução de quase 20% no tempo de compilação, e entre 10% e 20% no tempo para as aplicações de inicialização.

Borin *et al.* [8] apresentam o LAR-CC *(Large Atomic Region with Conditional Commit)*. O LAR-CC é uma técnica que permite a sistemas com projeto integrado de *hardware* e *software* formar e otimizar grandes regiões atômicas por meio de *commits* condicionais. Ele dinamicamente expande as regiões atômicas para se ajustar aos recursos do *hardware* que estão, por especulação, disponíveis. A técnica consiste em desvios condicionais para pular operações de *commit* e em transformações de código que substituem as operações de *commit* por *commits* condicionais, que permitem fazer otimizações em grandes regiões atômicas. O LAR-CC foi implementado dentro do CMS no Efficeon e mostrou que pode aumentar o tamanho das regiões atômicas significativamente. Ele chegou a regiões com mais de 1000 instruções.

Porto *et al.* [42] descrevem uma técnica de otimização de árvores de traço e propõem a ideia de árvores de traço compactas. A técnica de otimização que utiliza árvores de traço sofre com muita duplicação de código em ambientes onde é feita a tradução dinâmica de binários. Isto faz com que caminhos muito longos sejam descartados. As árvores de traços sempre terminam com um desvio para a cabeça do laço, as árvores de traço compactas permitem que os desvios apontem para outras regiões internas das árvores, o que proporciona menos duplicação de código. Os resultados dos experimentos mostraram que a técnica de árvores de traço é lenta em ambientes de tradução dinâmica de binários, enquanto a técnica de árvores de traço compacta se mostrou eficiente e com boa con-

vergência, também se observou que mesmo quando o tamanho da árvore selecionado é maior que o necessário, a técnica de árvores de traço compacta é bem efetiva.

A NETPlus foi uma técnica proposta por Davis e Hazelwood [15]. Inicialmente ela começa a formar superblocos iguais ao da NET, ao alcançar o limite para terminar o superbloco, a técnica o forma normalmente até alcançar a condição de parada. Então, ao invés de parar a formação, o algoritmo varre estaticamente todos os alvos de desvio a frente procurando um salto para a cabeça do superbloco. A técnica adiciona todas as instruções nesse caminho dentro do superbloco. Nos experimentos realizados no artigo os autores concluíram que o NETPlus seleciona os caminhos quentes tão bem quanto o NET e captura muito mais laços em superblocos individuais, além de introduzir pouca sobrecarga no processo de seleção de superblocos.

Implementamos e comparamos as técnicas NET, MRET2, LEI e *Trace Tree*. Escolhemos técnicas que constroem as regiões dinamicamente e que sejam importantes na literatura. Não implementamos a Treegion porque como o objetivo dela é o escalonamento ela não utiliza nenhuma política de detecção de código quente, criando treegions em todo código, o que causa uma grande sobrecarga na simulação.

3.2 Técnica LEF

Após compararmos quatro das técnicas de formação de regiões e observar os resultados, aproveitamos a facilidade da ferramenta RAIn (Capítulo 4) e implementamos uma nova técnica de formação de regiões, a técnica LEF (*Last Executing Function*), que pode ser vista como um complemento à técnica NET. Ela cria superblocos e regiões baseadas em funções dinamicamente. A motivação por trás da técnica é a de buscar regiões que representem a porção mais frequentemente executada das funções visando otimizar o código da função inteira. Regiões grandes podem possibilitar mais oportunidades de otimização e melhorar o desempenho significativamente [8].

A técnica foi proposta com o intuito de obter um desempenho melhor nos benchmarks representativos do uso da virtualização. Pois os *benchmarks* do tipo SPEC contém aplicações com um comportamento bem definido, o que facilita a formação das regiões. Buscamos com a LEF obter resultados parecidos com a NET onde ela se saiu bem e melhorar o tamanho dinâmico, que indica a localidade da execução, sem aumentar a duplicação de código, o que causaria um aumento na carga da *cache* de código. Como a NET apresentou bons resultados para o SPEC, não esperamos que a LEF supere-a neste tipo de *benchmark*.

A técnica LEF consiste em criar superblocos da mesma forma que a NET. Caso uma instrução de retorno seja encontrada ao construir uma região, a técnica fecha o superbloco e entra em uma fase de junção de superblocos. O algoritmo faz uma busca em largura por todos os superblocos que chegam àquele superbloco que possui o retorno, incluindo todos os superblocos em uma única região. A busca segue até que não haja mais arestas ou caso uma chamada de função seja encontrada, assim o algoritmo termina a busca e fecha a nova região que substituirá as outras.

A Figura 3.3 (a) representa regiões criadas seguindo a forma que a NET usa. Supondo que a região R1 tenha sido alcançada a partir de uma chamada de função e a região R6 termine com um retorno de função, a Figura 3.3 (b) mostra a junção que a técnica LEF realiza. Todas as regiões que estão dentro do retângulo farão parte de uma região só agora. A Figura 3.3 (c) representa essa junção caso a região R2 não seja quente o suficiente para virar uma região quando a execução chegar a criação da região R6. Apesar de não ter chamada de função na região R3 a junção das regiões terminaria nela, pois o algoritmo não alcançou outra região.

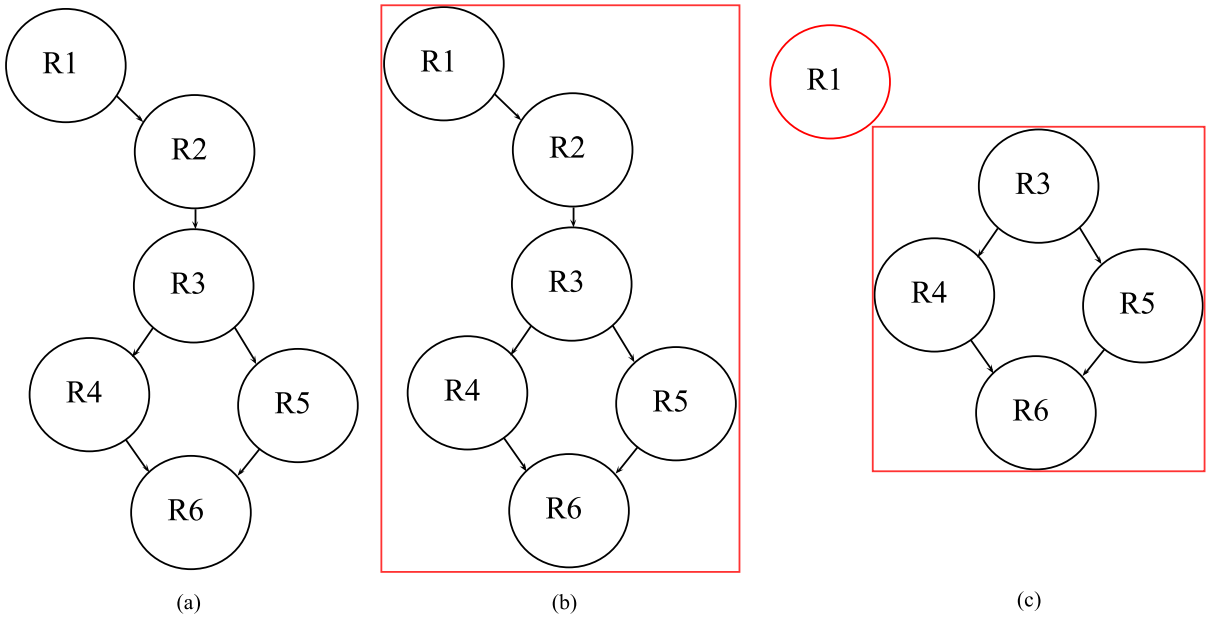


Figura 3.3: (a) Superblocos criados seguindo a política da NET. (b) Junção de superblocos pela LEF. (c) Junção de superblocos pela LEF se R2 não for quente.

Capítulo 4

Infraestrutura

Este capítulo apresenta o RAIn, a infraestrutura desenvolvida para implementar e avaliar as técnicas de formação de regiões.

Implementar diretamente uma técnica de formação de região em um sistema real é desafiante, principalmente porque depurar códigos gerados dinamicamente é difícil. Implementar várias técnicas e compará-las é bem complexo e extremamente custoso em termos de tempo. Por isso ao invés de implementar realmente as técnicas nós as simulamos através da Infraestrutura de Avaliação de Regiões (RAIn[1]). Como a tradução dinâmica de binários depende fortemente de informações acerca do perfilamento dinâmico, o RAIn é uma ferramenta ideal pois ela é capaz de coletar informações precisas de perfilamento antes de precisarmos implementar uma técnica em um sistema real. O RAIn é diretamente baseado no TEA [41], que é um autômato de execução de traços.

O TEA (*Trace Execution Automata*) utiliza um Autômato Finito Determinístico (AFD) para mapear as instruções ou blocos básicos executados em traços pré-estabelecidos. Ele contém duas fases, a de gravação em que o TEA constrói o AFD para um traço e a fase de execução, quando um traço já gravado é executado. O TEA pode ser usado para construir traços em um sistema, como por exemplo, em um tradutor dinâmico de binários, para gravar formas de traços com informações de perfilamento para reuso em execuções futuras e também para construir e coletar informações de traços sem realmente implementá-los em um sistema real. Esta última possibilidade foi o que motivou o uso do TEA como base para o RAIn.

O RAIn também utiliza um AFD, ele mapeia as instruções executadas em regiões pré-estabelecidas. Para essa pesquisa, implementamos a ferramenta como um Grafo de

[1]do inglês *Region Appraise Infrastructure*

Fluxo de Controle (CFG[2]) de regiões, as regiões por sua vez foram representadas como um CFG de instruções. A entrada do RAIn é um traço completo de execução, retirado de um sistema previamente executado. O RAIn percorre este traço simulando a execução do sistema, a cada instrução lida é tratada como se estivesse sendo executada naquele momento, desta forma é possível simular a execução e a criação dinâmica das regiões. Dentro do RAIn nós implementamos as técnicas NET, MRET2, LEI e *Trace Tree*.

A ferramenta funciona da seguinte forma: primeiro o programa cria uma região chamada NTE (*No Trace being Executed*), toda vez que uma instrução executa fora de uma região sua execução é marcada em NTE. O RAIn monitora as instruções a procura de uma região de acordo com o algoritmo de formação de regiões implementado, por exemplo, contando os alvos de desvios para trás na técnica NET. Inicialmente todas as instruções são executadas em NTE até que o RAIn entre em modo de gravação que é ativado pela técnica (na NET isso ocorre quando o contador de frequência de um candidato passa do limite estipulado). O modo de gravação consiste em gravar as instruções em uma região até que sua condição de parada ocorra, um salto para trás no caso da NET, então o RAIn encerra a região e efetivamente a incluí no autômato ligando-a às regiões que chegaram e vieram dela. Fora do modo de gravação o RAIn atualiza suas regiões e as transições entre regiões à medida que as instruções são executadas. Ao fim, o autômato nos fornece várias informações relevantes sobre as regiões, como a frequência dinâmica das instruções contidas nelas, o número de entradas, saídas laterais (quando a região termina antes do fim), frequência das transições, etc.

A Figura 4.1 (a) mostra a representação de dois superblocos, T1 e T2, formados através da política do NET. Na Figura 4.1 (b) vemos esses superblocos em um AFD, cada nó do AFD representa uma instrução e cada transição entre eles representa o fluxo de controle dentro das regiões. A instrução `ret` não aparece no autômato porque ela não faz parte de nenhuma região. Por fim vemos a aplicação completa no RAIn na Figura 4.1 (c), supondo que os superblocos gerados pelo NET sejam toda a parte quente do programa. Temos todas as transições entre regiões e também para trechos frios do programa, representados pelo estado NTE.

A implementação é dividida em várias classes: as classes *RegionManager*, *RegionRecorder*, a que representa uma região e a classe do RAIn. Uma região é representada por um CFG em que cada nó é uma instrução, tem uma lista de entradas e saídas, a região pode ser moldada de acordo com a técnica. A classe *RegionManager* gerencia a criação das regiões no RAIn, ela verifica quando ativar ou abortar uma gravação de uma região.

[2]do inglês *Control Flow Graph*

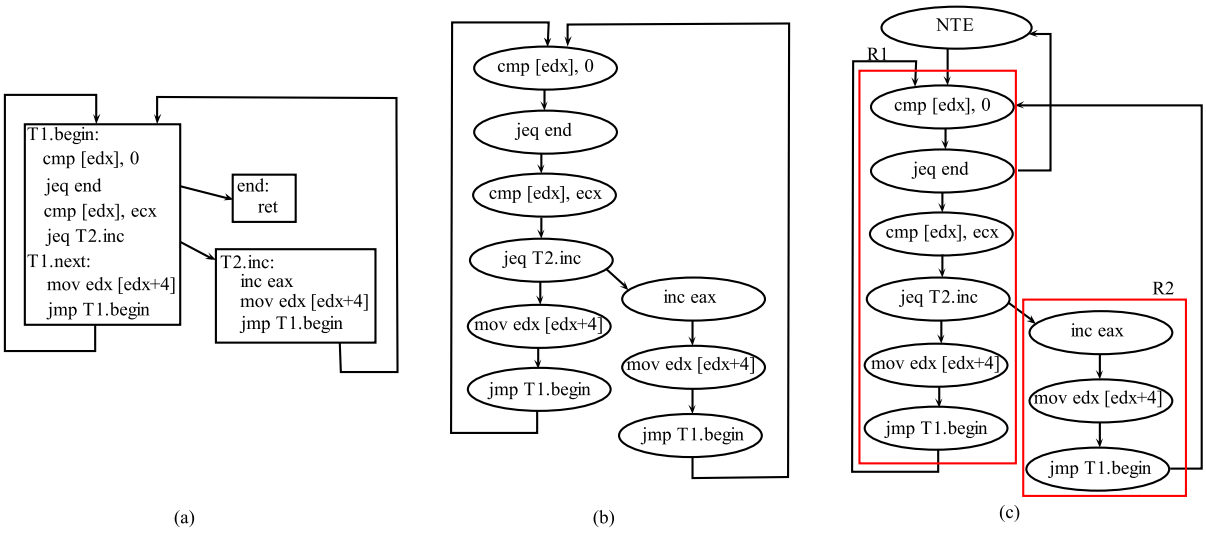


Figura 4.1: (a) Superblocos do NET. (b) AFD para os superblocos do NET. (c) RAIn do programa.

A classe *RegionRecorder* é quem grava a região quando a gravação está ativa. Por fim a classe do RAIn é a classe principal, mantém a representação do RAIn além de gerenciar a atualização das regiões, o andamento da execução e organizar as informações e estatísticas. Há também um conjunto de funções para a técnica a ser executada no RAIn, esse conjunto dita o formato das regiões e a forma como a técnica as constrói. Executar uma nova técnica no RAIn é fácil, só é necessário incluir o arquivo com as funções da técnica à função *Main* do programa e escrever no *RegionManager* a função que determina quando a gravação deve ser ativada.

Implementamos as técnicas no RAIn utilizando traços que continham tanto instruções de usuário quanto de sistema, pois o objetivo do trabalho era avaliar as técnicas em um cenário real com virtualização de sistemas, e não apenas de processos. Percebemos que o código de sistema faz com que as técnicas percam desempenho, pois as regiões criadas não tomam a forma que deveriam tomar se as instruções de usuário e sistema não se misturassem. Por exemplo, se acontecer uma falta de página durante a criação de uma região, o código que trata a falta de página será inserido na região, mas nas próximas vezes que a região for executada ela não chegará ao fim a menos que haja outra falta de página no mesmo ponto. Técnicas como a NET, por exemplo, verificam se um salto para trás foi tomado para iniciar e encerrar a gravação de uma região, como as instruções de sistema estão em outra região da memória a transição delas para a região de usuário pode ser vista erroneamente como um salto para trás, transformando o comportamento

do algoritmo. Por isso criamos uma modificação no RAIn para que ele segmente código de usuário separadamente de código de sistema. Dessa forma instruções de sistema são ignoradas enquanto uma região de usuário está sendo criada e instruções de usuário são ignoradas quando região criada for de sistema. Transições entre regiões de usuário e sistema ainda podem ocorrer sem problemas. A modificação se mostrou valiosa e necessária para poder reproduzir resultados condizentes aos obtidos por outros trabalhos.

Utilizamos o Bochs [32] para coletar os traços de execução. O Bochs é um programa que emula completamente a arquitetura IA-32(x86), é escrito na linguagem C++ e foi projetado para rodar nas mais diferentes plataformas. O Bochs simula todo o ambiente do PC, interpreta cada instrução e tem um modelo de dispositivo para cada periférico. O *software* que está executando na simulação "acredita" que está rodando em uma máquina real. O Bochs é uma máquina virtual de sistema integral, onde um sistema completo em *software* é executado em um sistema hospedeiro que roda em um conjunto de instruções e um sistema operacional diferentes.

Bochs é um *software* livre, escrito em C++, permite a aplicação das modificações e técnicas necessárias e tem uma boa documentação para usuários e desenvolvedores. Além disso, ele facilita a possibilidade de instrumentação. Em nossos experimentos, o Bochs tomou, em média, 50 a 250 ciclos da máquina hospedeira para emular cada instrução da máquina hóspede. Apesar de lento, o Bochs nos forneceu os traços de execução adequados contendo as instruções de sistema sem que fosse necessário grande modificação. Os traços poderiam ser obtidos com outras ferramentas, o formato da entrada do RAIn não tem qualquer relação com o Bochs.

Geramos traços de 10 bilhões de instruções para cada aplicação dos *benchmarks* utilizados, que já ocupam um grande espaço de armazenamento. 10 bilhões de instruções já é o suficiente para analisar o comportamento das aplicações e das técnicas de formação de regiões. O trecho das aplicações coletado para os traços de entrada da região foi o segundo trecho de 10 bilhões de instruções após o início, ou seja, iniciamos a aplicação, ignoramos 10 bilhões de instruções e coletamos os próximos 10 bilhões. Todas as técnicas usaram o mesmo traço de entrada, desta forma a comparação entre elas foi justa. Para validar cada técnica de formação de região implementada no RAIn utilizamos a seguinte metodologia: para cada uma imprimimos um CFG de algum *benchmark* grande e marcamos as duas regiões mais quentes, duas mornas e uma fria. Então comparamos com como o algoritmo da técnica deveria reagir.

Capítulo 5

Resultados Experimentais

Neste capítulo são apresentados os resultados experimentais do trabalho. Apresentamos as métricas utilizadas para avaliar o desempenho das técnicas em um ambiente de uma máquina virtual de sistema. Os experimentos foram realizados com os *benchmarks* do SPEC CPU 2006 e no SYSmark 2012. Utilizamos o SPEC, pois ele é tradicionalmente usado na avaliação das técnicas de formação de regiões do estado da arte, como nosso objetivo foi avaliar as técnicas em um ambiente que represente as aplicações utilizadas por usuários reais de virtualização utilizamos também o SYSmark. O SYSmark é um *benchmark* baseado em aplicações que reflete os padrões de uso dos usuários empresariais em diversas áreas, de produtividade de escritório até gerenciamento de sistemas. Usualmente os experimentos realizados nas técnicas de formação de regiões são feitos somente em ambiente com código de usuário, sem a interferência de código do sistema. Como o trabalho tem como objetivo estudar ambientes reais de virtualização, fizemos os testes incluindo as instruções de sistema e implementamos as técnicas segmentando código de usuário e código de sistema como visto no Capítulo 4.

5.1 Métricas

A comparação entre as técnicas foi feita observando-se uma série de métricas encontradas na literatura. Estas métricas são descritas a seguir:

Número de Regiões O número de regiões indica quantas regiões a técnica formou no total. Esta métrica mede a sobrecarga de compilação, quanto mais regiões são formadas mais compilação é necessária, ou seja, quanto menor o número de regiões menor é a sobrecarga criada pela técnica.

Cobertura do Código A cobertura do código indica qual a porcentagem do código coberto por regiões. Diz o quanto a política de detecção de código quente e formação de regiões está acertando, quanto mais execuções fora de regiões, maior a chance de haver código quente que não se transformou em região. É importante comparar essa métrica em conjunto com o número de regiões formadas, pois há técnicas que formam poucas regiões mantendo uma baixa cobertura, o que não é desejável.

90% Cover Set A métrica *90% cover set* indica qual o número mínimo de regiões necessárias para cobrir 90% da execução do código. Serve para medir o desempenho geral de otimizações dinâmicas. Quanto menor o *cover set*, menos regiões são necessárias para cobrir a parte mais quente do código, são essas regiões que devem sofrer a maior otimização.

Completion Ratio Porcentagem de vezes em que as regiões foram executadas inteiramente, ou seja, todas as instruções contidas naquela região foram executadas. Um bom *completion ratio* é importante para garantir que a técnica forme regiões que realmente serão utilizadas completamente, pois otimizações aplicadas em regiões com uma taxa baixa de *completion ratio* são desperdiçadas, além de inserir mais compilação ou precisar reverter o estado da arquitetura para o início da região, dependendo do sistema.

Duplicação de Código A duplicação de código é uma métrica que mede a quantidade de vezes que instruções aparecem repetidas, ou seja, em mais de uma região. A duplicação implica a carga da *cache* de código, quanto maior a duplicação, menos otimizado é seu uso. Além de exigir mais compilação pois a mesma instrução é compilada diversas vezes nas várias regiões em que ela aparece. É necessária uma baixa duplicação de código para tornar a implementação real de uma técnica viável.

Tamanho Dinâmico Médio O tamanho dinâmico é o número total de instruções executadas pela região dividido pelo número de vezes em que a região foi executada. O tamanho dinâmico de regiões com baixa taxa de completude (*completion ratio*) pode ser menor do que o tamanho estático. Por outro lado, regiões que contém laços podem apresentar um tamanho dinâmico muito maior do que o tamanho estático. A métrica indica a localidade da execução, quanto maior o tamanho dinâmico maior é o trecho de código que executa junto em uma região e menor é a transição entre as regiões, mostrando que

o algoritmo escolheu bem ao formar aquela região. Uma técnica que obtém um tamanho dinâmico alto provavelmente proporcionará regiões que melhorarão o desempenho da aplicação após serem otimizadas, principalmente se obtiverem um bom *completion ratio*.

5.2 SPEC CPU 2006

Foram testados traços de execução de 10 *benchmarks* SPECFP e 4 SPECINT. Os traços gerados continham 10 bilhões de instruções, como visto no Capítulo 4. Os resultados desta seção correspondem às técnicas NET, MRET2, LEI, *Trace Tree* e a LEF, todas com segmentação de código.

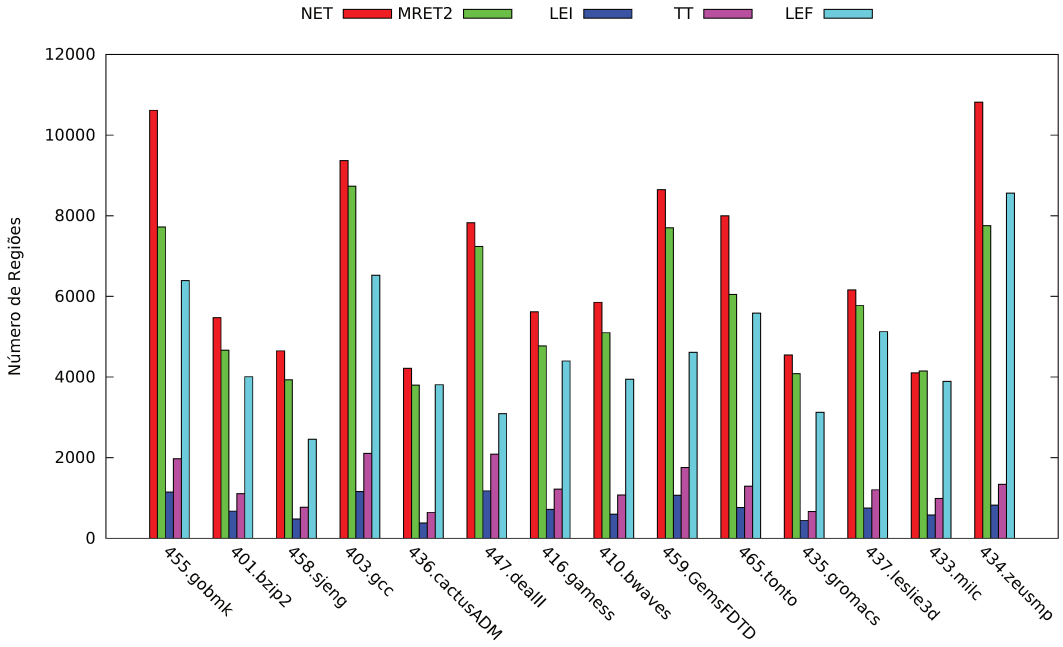


Figura 5.1: Número de regiões criadas pelas técnicas nos *benchmarks* do SPEC.

A Figura 5.1 indica o número de regiões que cada técnica cria para os *benchmarks* testados, a métrica reflete parte da sobrecarga da compilação, buscamos técnicas que tenham um número baixo de regiões. A LEI é a técnica com o menor número de regiões, sendo quase a metade da *Trace Tree*, que é a segunda menor. Como ambas as técnicas usam uma forma mais específica de completar as regiões (A LEI fecha com um ciclo e a *Trace Tree* com um salto para a raiz(Seção 3.1)) era esperado que houvessem menos regiões formadas por elas. Já a NET e a MRET2 formam muito mais regiões que as outras, sendo

a NET a pior técnica. A LEF cria um grande número de regiões, em alguns casos ela possui um número tão grande quanto a NET e em outros a metade desta. Ela não tem um número tão bom quanto a LEI e a *Trace Tree* porém é melhor que a NET e a MRET2.

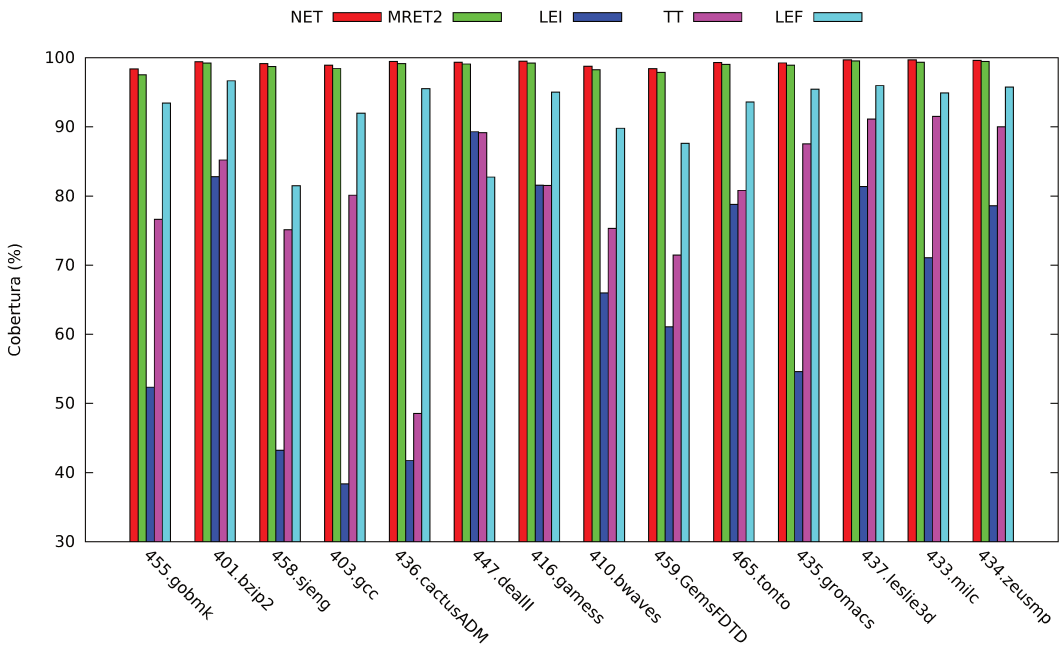


Figura 5.2: Cobertura sobre as instruções dos *benchmarks* do SPEC pelas regiões formadas.

A Figura 5.2 revela a porcentagem das instruções executadas cobertas por regiões produzidas por cada técnica de formação de regiões. Entre as quatro técnicas tradicionais, a NET e a MRET2 cobrem quase todo o código e são claramente as melhores. A *Trace Tree* tem uma cobertura menor que a NET e a MRET2, ela se mantém perto dos 80%. A LEI varia muito com a aplicação e tem uma cobertura baixa, a baixa cobertura é condizente com o baixo número de regiões criadas por ela. Conclui-se que a NET e a MRET2 são as que melhor conseguem construir o código em regiões. A LEF tem uma cobertura semelhante à da NET e da MRET2, caindo um pouco em alguns *benchmarks* como o `458.sjeng` e o `447.dealII`, que contém retornos no meio de regiões, impedindo que a mesma forme regiões maiores. O retorno faz com que o algoritmo pare a construção dessas regiões.

A Figura 5.3 mostra o 90% *cover set* obtido pelas técnicas no SPEC, desejamos obter um baixo 90% *cover set* pois assim conseguimos cobrir a parte mais quente do código

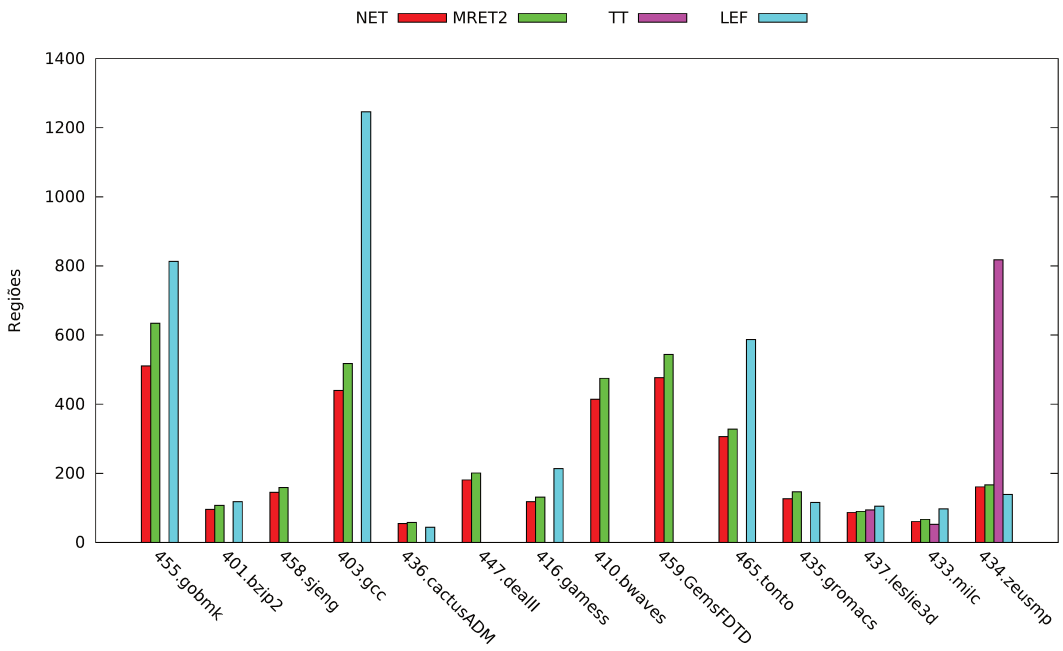


Figura 5.3: 90% *Cover Set* obtido pelas técnicas no SPEC.

com poucas regiões. Claramente a NET e a MRET2 obtêm os melhores resultados. A LEI não alcança 90% de cobertura, como já foi visto na Figura 5.2. A *Trace Tree* raramente chega a cobrir mais de 90% do código, só o fez nos *benchmarks* `437.leslie3d`, `433.milc` e `434.zeusmp`. Isso implica que as regiões criadas pela NET exploram melhor a localidade de execução melhorando a oportunidade de otimização. Já para a LEF o 90% *cover set* varia, em alguns casos ela se equipara à NET, como esperado, porém, para o `403.gcc`, o `465.tonto` e o `455.gobmk` ela é bem pior, pois as regiões com maior frequência de execução estão em mais de uma função na NET e são dividas na LEF. Para os *benchmarks* `458.sjeng`, `447.dealII`, `410.bwaves` e `459.GemsFDTD` a LEF nem chega a 90% de cobertura, como visto na Figura 5.2.

A Figura 5.4 indica o *completion ratio* das técnicas de formação de região nos *benchmarks*, é desejável que as técnicas alcancem uma taxa alta. A MRET2 é a técnica que obtém a melhor taxa entre todas elas, seguida pela NET. A LEI varia conforme a aplicação e tem uma taxa baixa em muitos *benchmarks*. Para *Trace Tree* não faz sentido calcular o *completion ratio*, pois uma região formada por ela não tem só uma saída principal, além de crescer conforme a execução vá seguindo. Portanto a MRET2 é a técnica que torna as regiões mais seguras para otimizações, pois as executa completamente na grande maioria das vezes. A LEF tem um bom *completion ratio* ficando entre as taxas da NET e da

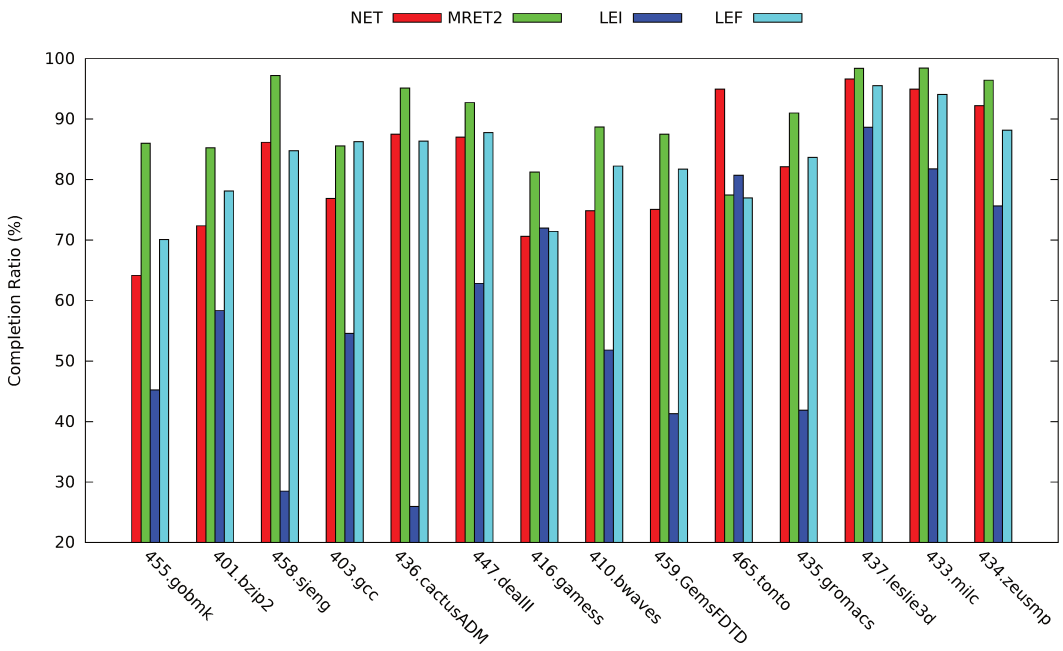


Figura 5.4: *Completion Ratio* das técnicas no SPEC.

MRET2.

As Figuras 5.5 e 5.6 mostram a duplicação de código gerada pela *Trace Tree* e pelas outras técnicas respectivamente. A duplicação gerada pela *Trace Tree* é muito superior como se pode notar pela diferença na escala entre os dois gráficos. As outras técnicas têm um valor de duplicação semelhante entre elas. A LEF acompanha o comportamento das outras técnicas, com um pouco mais de duplicação. Esse resultado demonstra que a *Trace Tree* não é uma boa técnica a ser implementada pois a duplicação excessiva pode causar uma grande sobrecarga no desempenho durante a compilação das regiões.

A Figura 5.7 revela o tamanho dinâmico médio das regiões formadas pelas técnicas no SPEC CPU 2006. A *Trace Tree* tem um dos maiores tamanho dinâmico médio entre as técnicas, isso é esperado já que o tamanho estático das regiões formadas por ela é grande, porém devido à grande taxa de duplicação a técnica não se torna viável. Outra técnica com um ótimo tamanho dinâmico é a LEI, pois ela captura os laços da aplicação que são as regiões com maior potencial de execução sem transição para outras regiões, isto faz com que as otimizações feitas nas regiões da LEI tenham uma grande capacidade de causarem um bom impacto na execução. A NET e a MRET2 tem um tamanho dinâmico médio menor que as outras e tamanho comparável entre elas, no geral a NET é melhor,

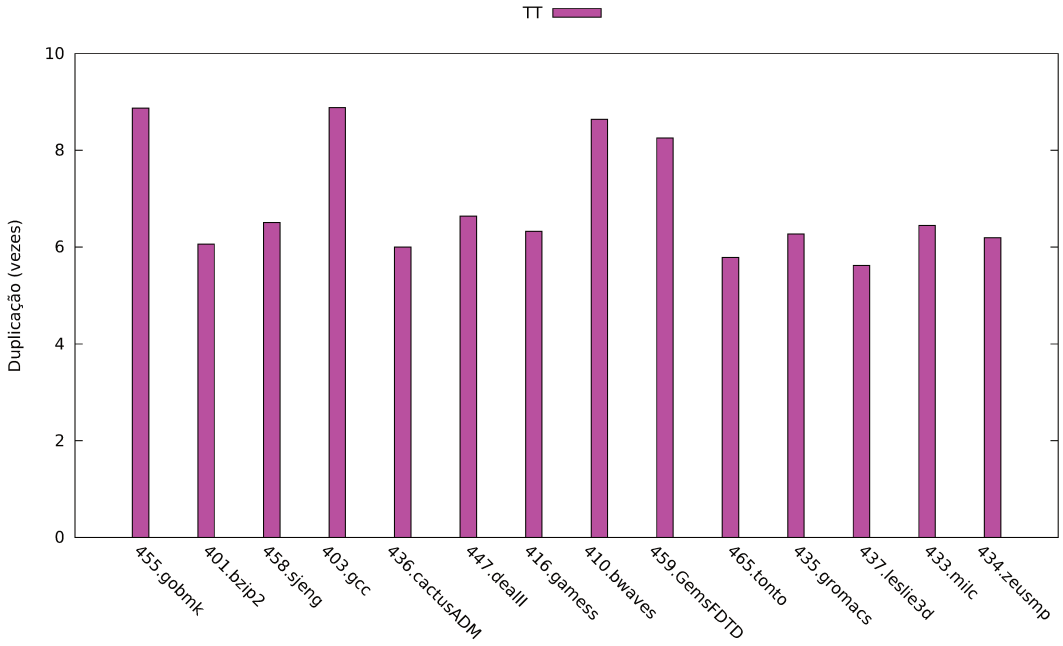


Figura 5.5: Duplicação de código causada pela *Trace Tree*.

sendo muito melhor no *benchmark* `447.dealII` onde a MRET2 dividiu regiões com um alto tamanho dinâmico. A LEF tem um tamanho dinâmico um pouco maior que a NET e a MRET2 na maioria dos casos, exceto no `447.dealII` em que ela é menor que a NET e a MRET2, pois uma chamada de função em um laço desfez uma região que teria um bom tamanho dinâmico. Já nos *benchmarks* `458.sjeng` e `459.GemsFDTD` a LEF se sobressai a todas as outras, em ambos os casos ela forma regiões com funções contendo dois trechos quentes que estão em regiões separadas na outras técnicas.

Tabela 5.1: Técnicas Avaliadas no SPEC CPU 2006.

Técnica	# Regiões	Cobertura	*90% CS*	*Comp. Ratio*	Duplicação	Tam. Din.
NET	médio	bom	bom	bom	bom	ruim
MRET2	médio	bom	bom	bom	bom	ruim
LEI	bom	ruim	ruim	médio	bom	bom
Trace Tree	bom	médio	ruim	nenhum	ruim	bom
LEF	médio	bom	médio	bom	médio	médio

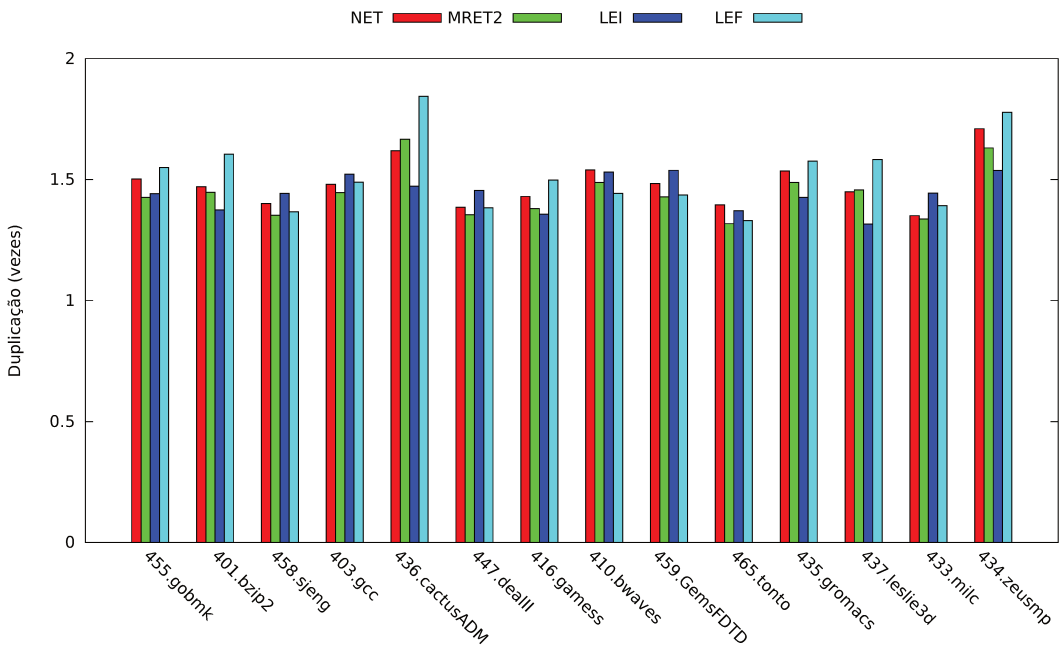


Figura 5.6: Duplicação de código causada pelas técnicas nos *benchmarks* do SPEC.

A Tabela 5.1 classifica as técnicas para as aplicações do SPEC de acordo com as métricas apresentadas de forma resumida.

Analisando os resultados do SPEC vemos que a NET e a MRET2 são técnicas que possuem uma alta cobertura, elas criam várias regiões para isso. Alcançam um bom 90% *cover set* e um bom *completion ratio*. O que as torna uma boa escolha como política de criação de regiões caso o alvo seja evitar compilações excessivas ou precisar reverter o estado da arquitetura para o início da região. Entre ambas, a NET é mais simples e possui um desempenho melhor para construir as regiões, enquanto a MRET2 possui um melhor *completion ratio* as custas de uma segunda fase na construção, a melhor escolha dependerá de qual o objetivo da implementação.

A LEI contém um tamanho dinâmico maior que as outras sem perder muito na duplicação de código. Ela não tem uma cobertura de instruções boa e seu *completion ratio* é menor do que as outras, porém ela cria poucas regiões. Se o objetivo for otimizar o uso da *cache* de código e otimizar as porções mais quentes da aplicação a LEI é uma boa escolha. A *Trace Tree*, apesar de seu tamanho dinâmico alto, não tem uma boa cobertura e possui muita duplicação de código, portanto acabou sendo a pior entre as técnicas testadas. A LEF não teve resultados ruins, mas apesar de próxima em algumas métricas ela é pior

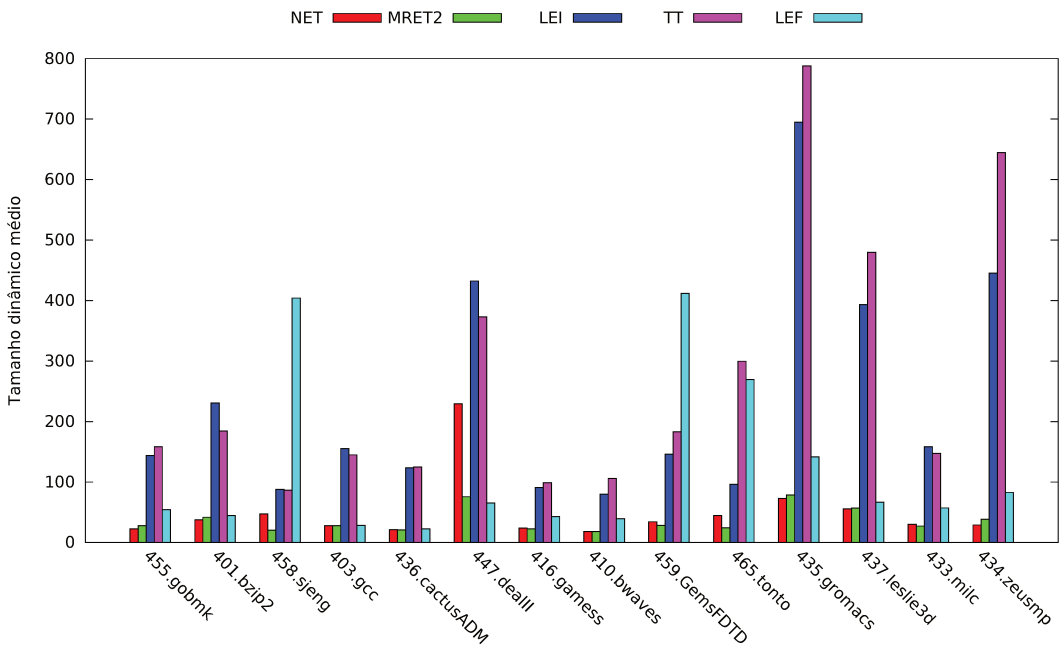


Figura 5.7: Tamanho dinâmico médio das regiões criadas pelas técnicas no SPEC.

que a NET e a MRET2 nos quesitos em que ela é melhor que a LEI. A LEF ainda tem um custo maior para construir regiões, porém, como visto na Seção 3.2, não era o intuito da técnica ser a melhor no SPEC e sim no SYSmark, pois a NET e a MRET2 já se saem bem no SPEC.

5.3 SYSmark

Os traços gerados do SYSmark também contém 10 bilhões de instruções cada. O resultado de todas as técnicas foi comparado com os resultados obtidos com os traços do SPEC. O cenário de produtividade *Office* do *benchmark* SYSmark 2012 foi executado e os resultados a seguir correspondem às técnicas com a modificação de segmentação usuário e sistema. As aplicações do *benchmark* podem ser vistas na Tabela 5.2.

A Figura 5.8 apresenta o número de regiões criadas pelas técnicas. Todas as técnicas criam mais regiões no SYSmark do que no SPEC. A NET continua criando mais regiões que as outras técnicas tradicionais, e a diferença é bem maior no SYSmark. A MRET2 é a segunda técnica a criar mais regiões, no SYSmark ela cria bem menos que a NET. A LEI e a *Trace Tree* criam poucas regiões, sendo a LEI a que cria menos, mostrando

Tabela 5.2: *Benchmarks* do SYSmark.

Aplicação	Descrição
Adobe Acrobat Pro 9	Leitor, criador e conversor de PDFs.
Microsoft Excel 2010	Planilhas de cálculo.
ABBYY FineReader pro 10.0	Reconhecimento ótico de caracteres (OCR).
Mozilla Firefox 14.0.1	Navegador de internet.
Microsoft Outlook 2010	E-mails, calendários e agendas.
Winzip Pro 14.5	Compressor de arquivos.
Microsoft Internet Explorer 8	Navegador de internet.
Microsoft PowerPoint 2010	Apresentação de *slides*.
Microsoft Word 2010	Documentos de texto.

que definitivamente a LEI é a técnica que produz menos sobrecarga de compilação pelo número de regiões formadas e a NET é a mais prejudicada. A LEF tem um número de regiões um pouco maior do que a NET em todos os casos no SYSmark, se tornando a pior técnica para esta métrica.

A Figura 5.9 mostra a cobertura de instruções pelas regiões formadas. O comportamento entre as técnicas no SYSmark se mantêm muito semelhante ao comportamento no SPEC. A NET continua sendo a técnica que melhor cobre o código, no SYSmark a cobertura é um pouco menor, ainda assim se mantém próximo ao 100%. A MRET2 também apresenta uma cobertura menor do que para os *benchmarks* do SPEC, as regiões se mantêm mais perto de 90% de cobertura do que de 100%. De qualquer forma, entre as técnicas tradicionais, a NET e a MRET2 ainda são as técnicas que melhor constroem o código em regiões, apesar da NET se sair melhor na cobertura a MRET2 tem um equilíbrio melhor entre a cobertura e o número de regiões. A *Trace Tree* e a LEI variam de acordo com a aplicação semelhante ao observado no SPEC, elas não obtém uma boa cobertura de código. A cobertura da LEF se mantém parecida com a da NET também no SYSmark, o que é um bom resultado.

A Figura 5.10 indica o 90% *cover set* obtido pelas técnicas no SYSmark. Graças ao segundo passo da MRET2, em que a técnica executa o passo da NET pela segunda vez (Seção 3.1), a técnica forma regiões menores e quando se trata de trechos quentes isso faz com que o 90% *cover set* aumente, o que é ruim. Pelo gráfico é possível observar que a NET é melhor que a MRET2, exceto para o `Internet Explorer` em que a MRET2 conseguiu formar muito bem as regiões mais frequentes. O número de regiões para alcançar 90% de cobertura é muito maior no SYSmark do que no SPEC. Tanto a LEI quanto a

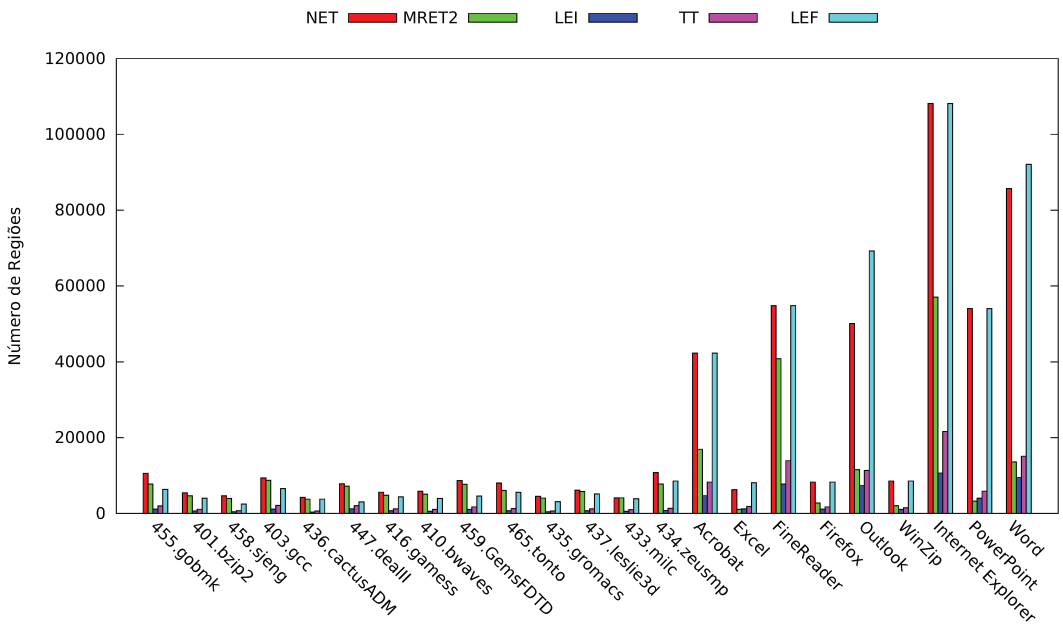


Figura 5.8: Número de regiões criadas pelas técnicas.

Trace Tree não alcançaram 90% de cobertura no SYSmark. Para a LEF o 90% *cover set* no SYSmark é bem semelhante ao da NET em todos os casos, não varia da forma como varia no SPEC, mostrando que a técnica se sai melhor no SYSmark.

A Figura 5.11 apresenta o *completion ratio* das técnicas de formação de regiões. A variação do *completion ratio* para uma mesma técnica é menor no SYSmark, a MRET2 continua sendo a técnica que alcança um melhor *completion ratio* mesmo não chegando a valores tão altos quanto os do SPEC, mostrando que mesmo que a técnica não tenha uma localidade de execução tão boa ela cria regiões seguras para serem otimizadas, pois o algoritmo perde em *cover set* mas ganha em *completion ratio*. A NET, no SYSmark assim como no SPEC, obtém o segundo melhor valor. A LEI continua com o pior valor, se mantendo por volta dos 70% em média. A LEF apresenta um *completion ratio* um pouco menor que a NET, com exceção do `Internet Explorer` em que a taxa é 30% menor, pois o algoritmo une regiões de funções incompletas (a junção não chegou até uma chamada de função) que são executadas inteiramente poucas vezes, por haver caminhos até o retorno de função mais frequentes que o na região. Como visto na Seção 5.2 não faz sentido calcular o *completion ratio* para a *Trace Tree*.

As Figuras 5.13 e 5.12 mostram a duplicação de código gerada pela *Trace Tree* e pelas

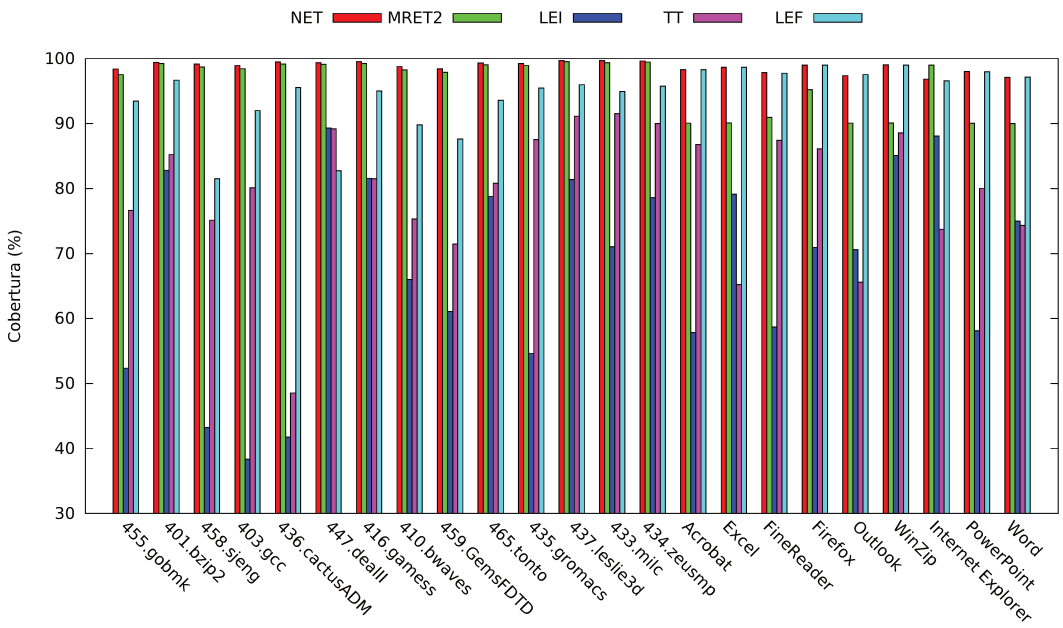


Figura 5.9: Cobertura sobre as instruções pelas regiões formadas.

outras técnicas respectivamente. A NET e a MRET2 continuam com a menor duplicação entre as técnicas, para o SYSmark a duplicação é até menor do que para o SPEC. Para a LEI a duplicação varia um pouco com a aplicação, mas no geral tem valores parecidos com os obtidos para o SPEC. A *Trace Tree* continua tão ruim no SYSmark quanto visto para o SPEC. A duplicação da LEF no SYSmark é boa, chega a ser até menor que a da NET e da MRET2, exceto para o `Internet Explorer` e o `PowerPoint` em que a duplicação é alta. Apesar da LEF na realidade ter mais instruções únicas duplicadas, há algumas instruções de sistema que se repetem muito nas regiões criadas pelas outras técnicas, ao juntar várias regiões em uma só, a LEF não duplica as instruções que estão na intersecção entre elas. Isso faz com que a duplicação total seja menor, o que não ocorreu no `Internet Explorer` e no `PowerPoint` em que algumas dessas instruções que se repetem não se juntaram.

A Figura 5.14 apresenta o tamanho dinâmico médio das regiões formadas. A *Trace Tree* obtém um tamanho dinâmico maior para o SYSmark, ainda assim a enorme duplicação da técnica a torna inviável. Em geral a LEI chega a ser melhor no SYSmark do que no SPEC, se mantendo melhor do que as outras duas técnicas, o que a torna uma boa candidata a otimizações pesadas já que há pouca transição entre regiões. Porém, há o problema de seu baixo *completion ratio* que pode fazer com que otimizações se percam em boa parte da execução e fazer com que uma sobrecarga de compilação seja adicionada. A

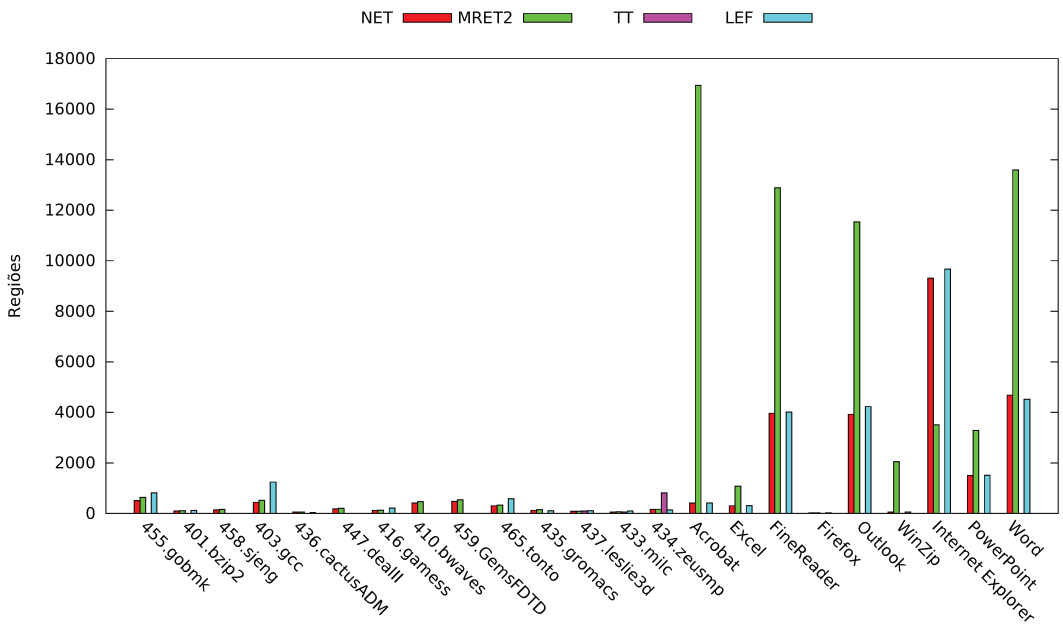


Figura 5.10: 90% *Cover Set* obtido pelas técnicas.

NET e a MRET2 alcançam um tamanho dinâmico semelhante. A LEF tem um tamanho dinâmico alto no SYSmark, ainda não fica próximo ao tamanho da LEI, mas é bem maior que o da NET e o da MRET2 o que é um ótimo resultado, pois o *completion ratio* dela é maior que o da LEI e sua duplicação de código é baixa.

Tabela 5.3: Técnicas Avaliadas no SYSmark.

Técnica	# Regiões	Cobertura	*90% CS*	*Comp. Ratio*	Duplicação	Tam. Din.
NET	ruim	bom	bom	bom	bom	ruim
MRET2	médio	bom	médio	bom	bom	ruim
LEI	bom	ruim	ruim	médio	bom	bom
Trace Tree	bom	ruim	ruim	nenhum	ruim	bom
LEF	ruim	bom	bom	médio	bom	bom

A Tabela 5.3 classifica as técnicas para as aplicações do SYSmark de acordo com as métricas apresentadas de forma resumida.

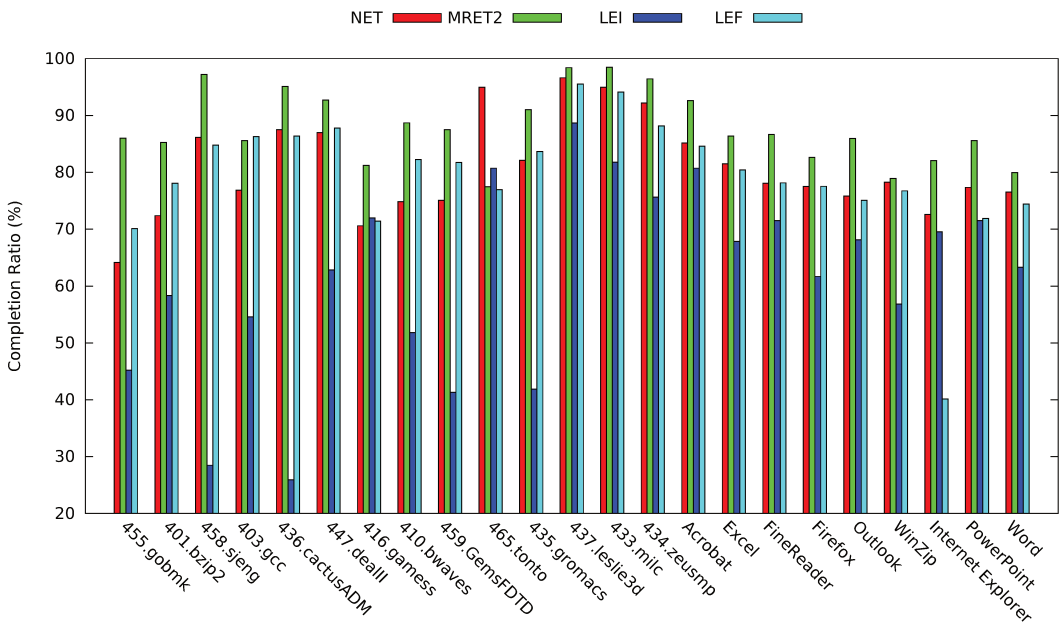


Figura 5.11: *Completion Ratio* das técnicas.

Considerando as análises apresentadas, podemos ver que os resultados mudam de perfil para o SYSmark. A NET perde para a MRET2 em número de regiões e apesar de alcançar uma maior cobertura a diferença não é grande. O *completion ratio* do MRET2 continua maior e ela só perde da NET no 90% *cover set*, tornando a MRET2 uma escolha melhor que a NET. A *Trace Tree* é ainda pior para o SYSmark, sem dúvida é a pior escolha. A LEI ainda é a técnica que alcança o melhor tamanho dinâmico, porém a LEF também alcança um bom resultado nesta métrica.

A LEF para o SYSmark teve ótimos resultados em comparação às outras técnicas, transformando-se em uma boa escolha como política de criação de regiões, a LEI possui um tamanho dinâmico maior que ela, porém perde nas outras métricas e ainda precisa de um *buffer* de histórico que causa uma grande sobrecarga para realizar o perfilamento (como visto na Seção 3.1). A LEF só não se sai bem no número de regiões criadas, o que pode criar uma sobrecarga de compilação.

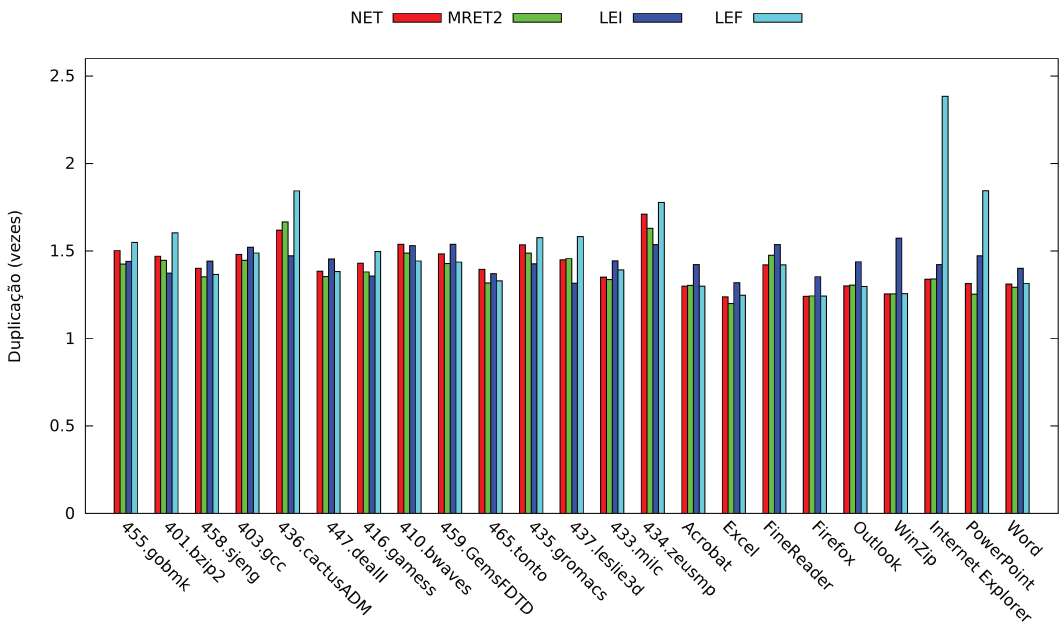


Figura 5.12: Duplicação de código causada pelas técnicas.

5.4 Segmentação de Usuário e Sistema

Analisamos também os mesmos resultados sem a modificação para segmentar as instruções entre regiões exclusivas de usuário e exclusivas de sistema, implementando as técnicas diretamente no sistema como está. Fizemos os experimentos com todos os *benchmarks*, porém a diferença para o SPEC entre os resultados com e sem segmentação não é tão significativa quanto para o SYSmark. Há diferenças mas como as aplicações do SPEC tem um comportamento mais padronizado as técnicas conseguem um desempenho melhor mesmo sem segmentação. Portanto analisaremos somente o SYSmark nesta seção.

As Figuras 5.15, 5.16 e 5.17 apresentam a comparação do *completion ratio* entre as técnicas com e sem segmentação. É possível perceber claramente como as técnicas falham em criar boas regiões quando não há segmentação, a diferença é maior para a LEI e para a NET, que não alcançam nem 10% de *completion ratio* no `PowerPoint` e no `Excel`. A MRET2 não se sai tão mal, isso porque ao passar por sua segunda fase consegue eliminar parte do código de sistema que invade regiões de código de usuário, isso é garantido quando o código de sistema não está na intersecção entre as duas regiões criadas pelas duas fases.

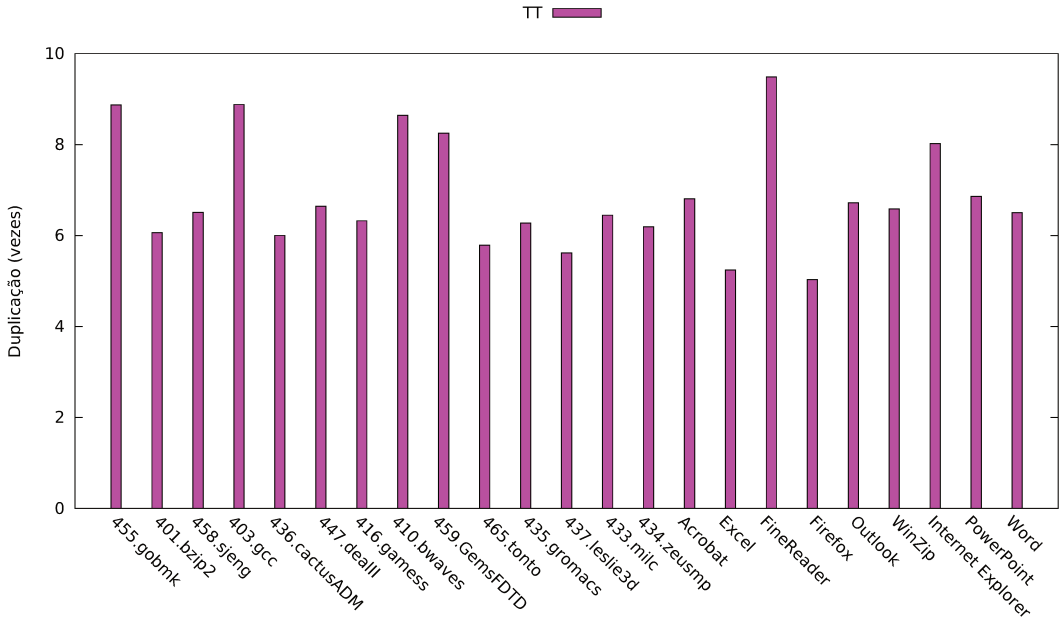


Figura 5.13: Duplicação de código causada pela *Trace Tree*.

As Figuras 5.18, 5.19, 5.20 e 5.21 mostram a duplicação de código das técnicas com e sem segmentação. Em todos os casos a duplicação é maior sem o uso de segmentação de código entre usuário e sistema. Novamente para a MRET2 o impacto é menor do que para as outras técnicas, sendo assim, ela é a que melhor lida com o código de sistema sem segmentação. A LEI e a NET se tornam inviáveis para algumas aplicações, pois a duplicação chega a 8 e 11 vezes respectivamente. A *Trace Tree* que já era péssima chega a ficar até 6 vezes pior no `PowerPoint`.

O comportamento da LEF é semelhante ao das outras técnicas. Para as demais métricas os resultados não foram tão destoantes, salvo alguns casos específicos. Analisando a duplicação de código e o *completion ratio* já é suficiente para entender claramente porque a segmentação é o melhor caminho ao usar uma técnica de formação de regiões. Se ainda assim, uma técnica for implementada diretamente no sistema sem segmentar o código a melhor opção é a MRET2, pois ela não recebe um impacto tão negativo quanto as outras técnicas.

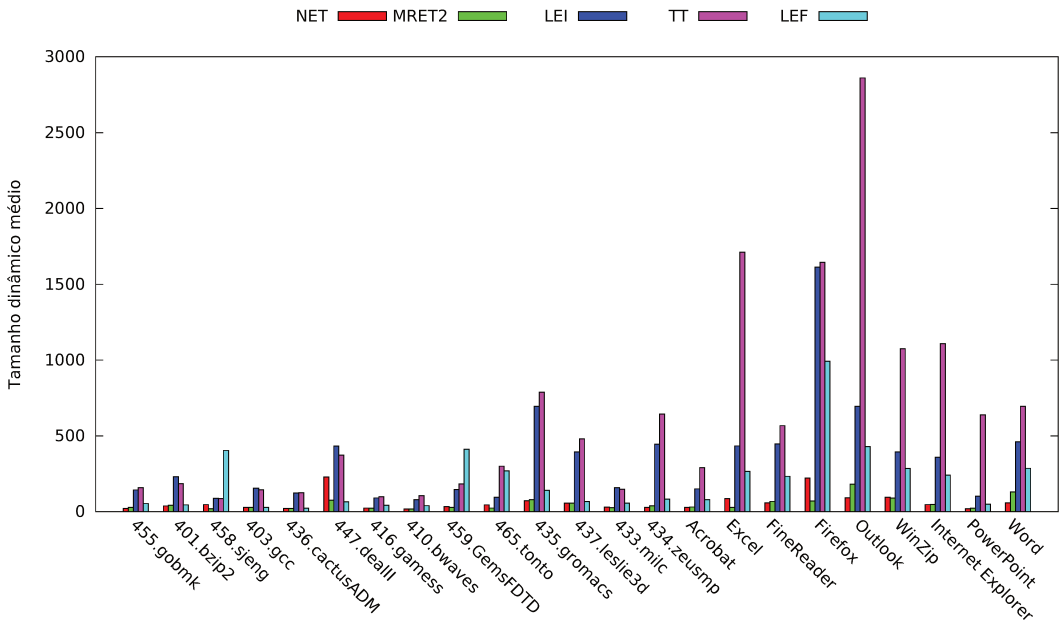


Figura 5.14: Tamanho dinâmico médio das regiões criadas pelas técnicas.

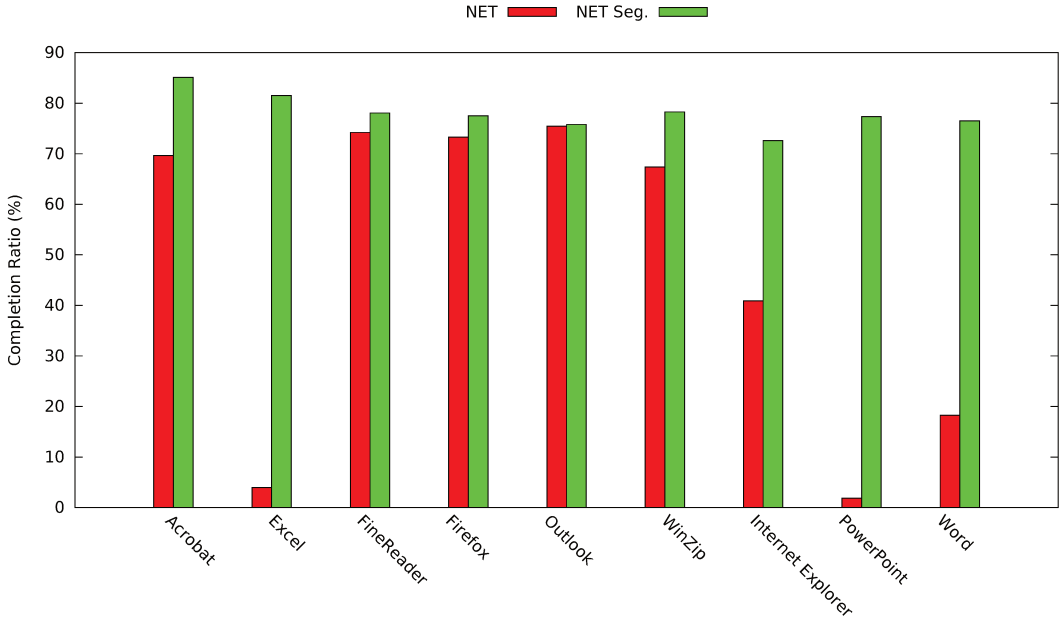


Figura 5.15: *Completion Ratio* obtido pela NET com e sem segmentação.

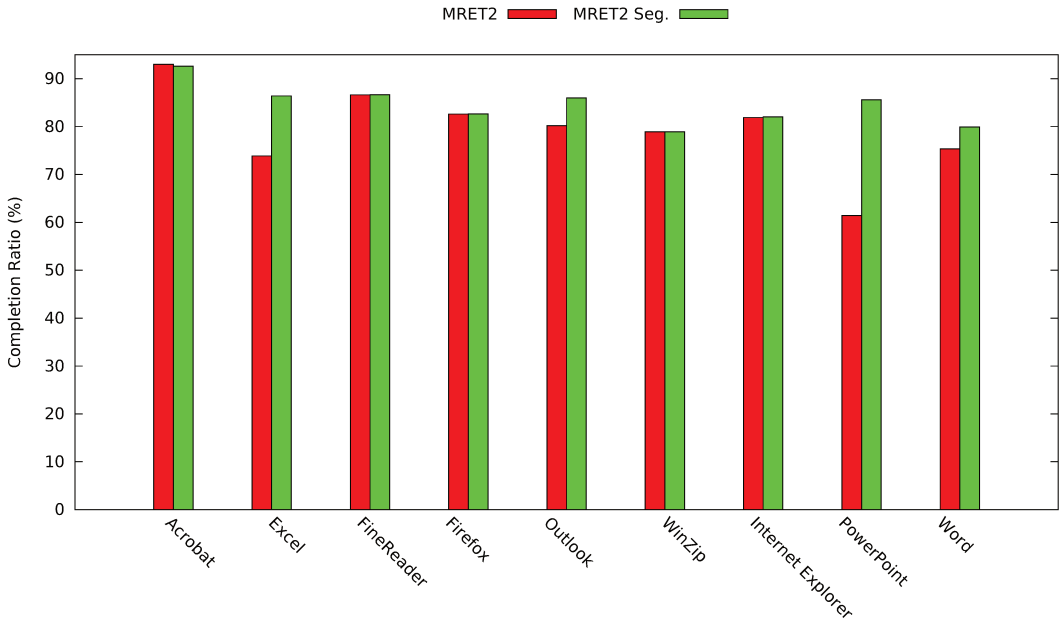


Figura 5.16: *Completion Ratio* obtido pela MRET2 com e sem segmentação.

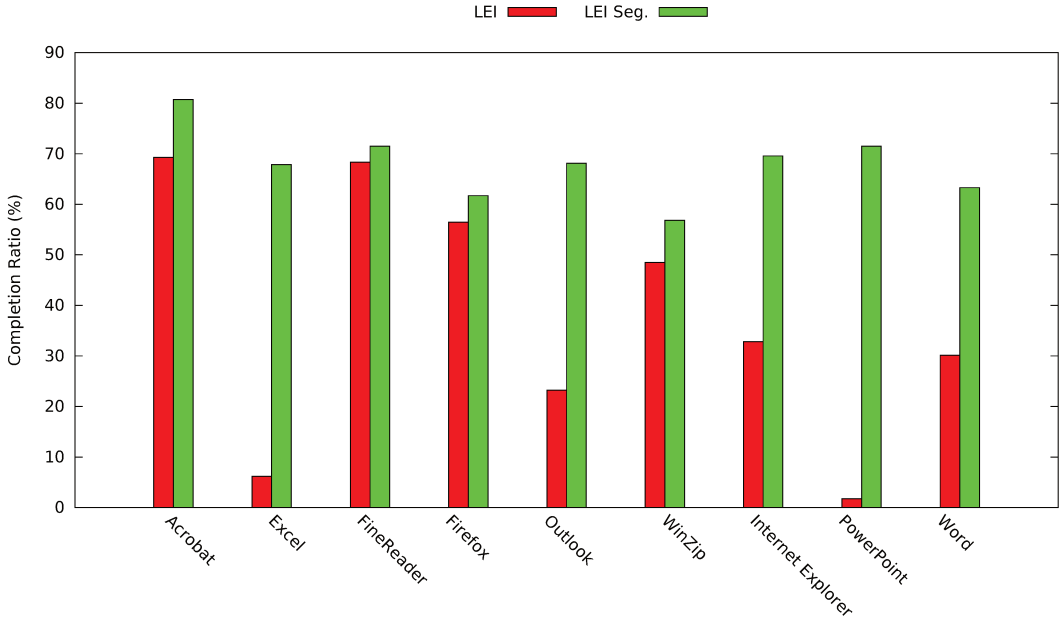


Figura 5.17: *Completion Ratio* obtido pela LEI com e sem segmentação.

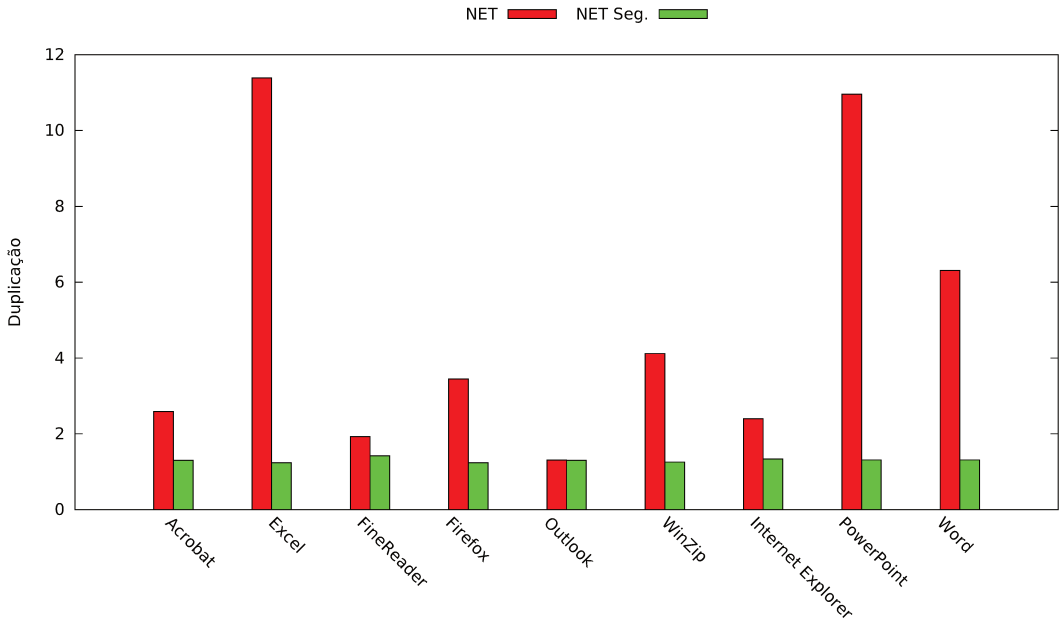


Figura 5.18: Duplicação de código criado pela NET com e sem segmentação.

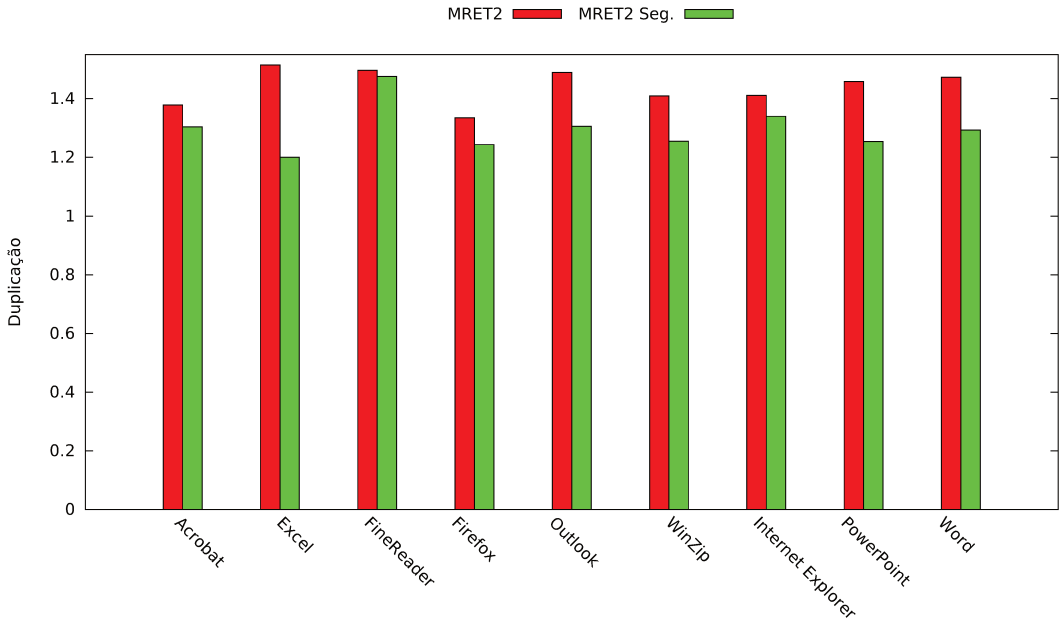


Figura 5.19: Duplicação de código criado pela MRET2 com e sem segmentação.

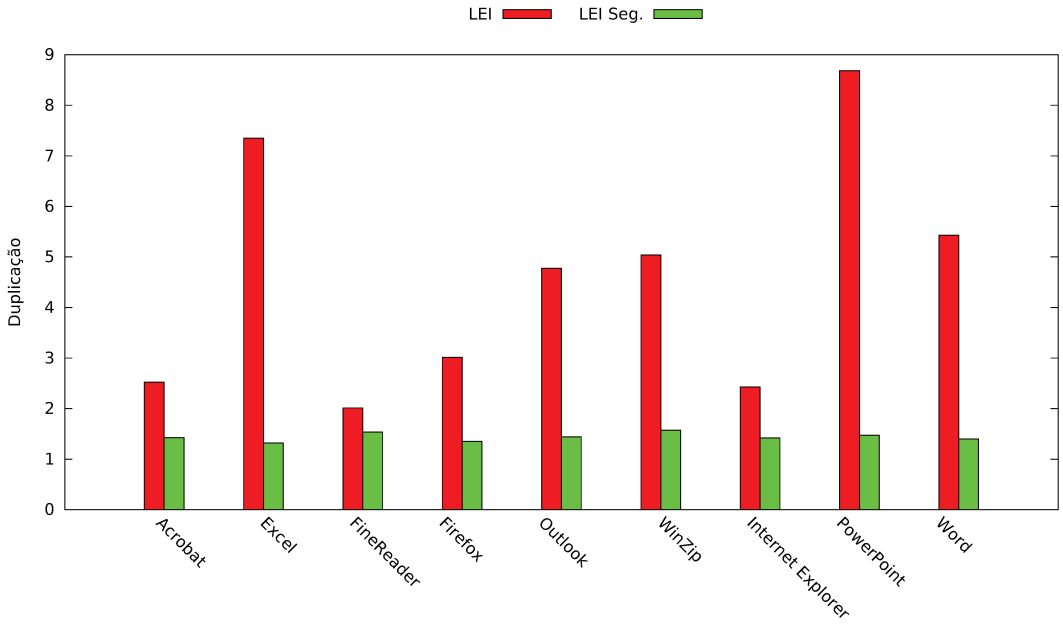


Figura 5.20: Duplicação de código criado pela LEI com e sem segmentação.

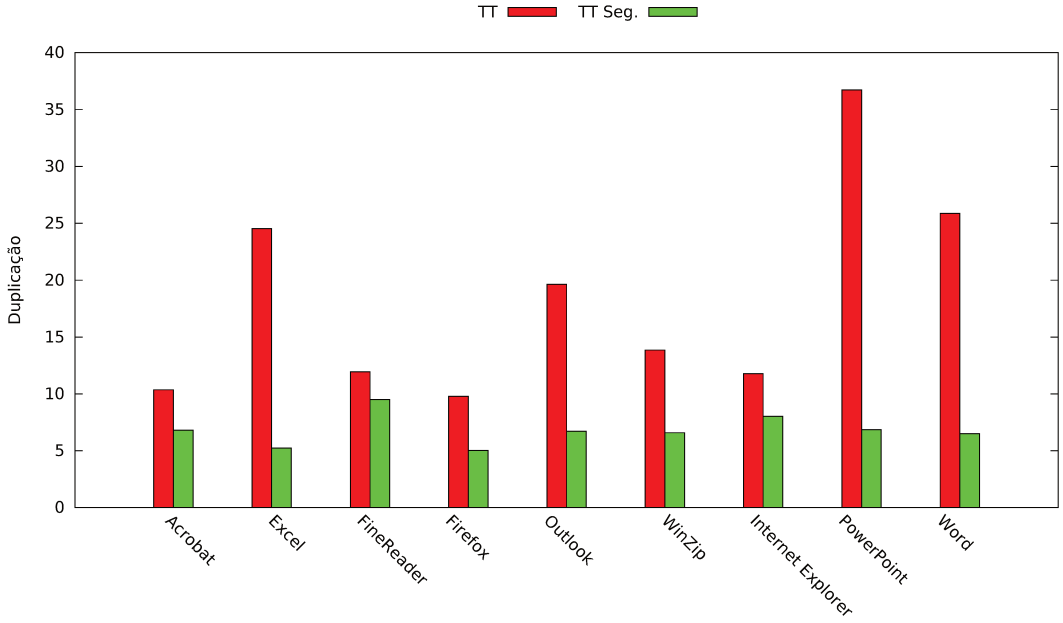


Figura 5.21: Duplicação de código criado pela *Trace Tree* com e sem segmentação.

Capítulo 6

Conclusões

Formação de regiões é um fator importante na otimização dinâmica de código em tradutores dinâmicos de binários. O formato da região que a máquina virtual deve formar influenciará em como o otimizador irá analisar aquela região.

Não há na literatura uma comparação entre as técnicas de formação de região existentes e os trabalhos que apresentam as técnicas normalmente as avaliam com *benchmarks* do tipo SPEC. Isto não é apropriado pois estes *benchmarks* não são representativos da real carga de trabalho dos usuários de virtualização.

Neste trabalho foi apresentada uma comparação justa entre as principais técnicas de formação de regiões existentes, utilizando *benchmarks* que representam a carga real de um usuário de virtualização. Uma nova técnica foi proposta para melhorar o desempenho nos ambientes representativos e também foi apresentada neste trabalho.

Os experimentos realizados demonstraram que dentre as técnicas comparadas, a técnica NET e a técnica MRET2 são as melhores escolhas como políticas de formação de região para *benchmarks* do tipo SPEC. Elas possuem uma alta cobertura, um bom 90% *cover set* e um bom *completion ratio*. Já para formação de regiões em aplicações do *benchmark* SYSmark, a técnica sugerida neste trabalho é uma escolha melhor. Ela tem um bom tamanho dinâmico sem causar muita duplicação de código e ainda tem uma boa cobertura e um bom 90% *cover set*. Também avaliamos que para um bom desempenho é necessário o uso de segmentação entre o código de sistema e o código de usuário ao utilizar as técnicas de formação de regiões.

Também apresentamos a infraestrutura RAIn. Ela foi criada para simular a execução das técnicas sem precisar implementá-las em um sistema real, além de prover a possibili-

dade de implementar novas técnicas de forma simples. O RAIn utiliza um autômato para mapear instruções em regiões pré-estabelecidas. Como entrada o RAIn recebe um traço de execução completa de um sistema previamente executado, neste trabalho utilizamos o Bochs para gerar este traço. A infraestrutura se provou muito útil, ela fornece informações precisas acerca do perfilamento dinâmico das aplicações.

Como efeito colateral desta pesquisa implementamos um compressor de traços. Percebemos que era um problema armazenar traços contendo a execução completa de um programa, já que estes chegavam frequentemente a trilhões de instruções executadas o que consome muitos *gigabytes*. Isso já é um problema conhecido por projetistas de sistemas que utilizam simulação dirigida a traços [49], por isso já existem trabalhos que utilizam compressores para poder armazenar estes traços, como o SBC (*Stream-Based Compression*) [37]. O SBC se mostrou muito eficiente comparado às outras ferramentas de compressão, mas ele não suporta instruções e operações de memória de tamanho variado, assim não consegue comprimir traços gerados a partir de código x86. Baseando-se no SBC criamos o VITC (*Variable-length Instruction Trace Compressor*) que comprime traços com instruções e operações de memória de tamanho variado e suporta traços gerados de código x86, com ele é possível adicionar mais informações relevantes aos traços comprimidos sem aumentar o tamanho do arquivo. Não utilizamos traços completos de execução para avaliar as técnicas de formação de região, utilizamos somente traços de 10 bilhões de instruções, pois o tempo para coletar traços completos e comprimi-los foi proibitivo. Portanto não precisamos do compressor para este trabalho. O VITC foi publicado no WSCAD-SCC em 2012 [57].

A partir dos estudos deste trabalho é possível imaginar alguns trabalhos futuros. Comparamos técnicas que constroem regiões de forma dinâmica, há técnicas que constroem regiões de forma estática e técnicas que combinam algoritmos estáticos com dinâmicos. É interessante avaliar como estas técnicas se comportam e compará-las com as técnicas de construção dinâmica de regiões. Há também trabalhos que apresentam algoritmos que melhoram a formação de regiões ou evitam certos erros comuns das políticas de formação, são trabalhos que podem ser aplicados a diversas técnicas de formação de região. Um possível trabalho futuro seria investigar esses algoritmos para entender para quais técnicas eles trazem mais vantagens e se a utilização deles compensa alguma eventual sobrecarga de compilação ou execução.

As técnicas de formação de região tem como objetivo transformar em região somente a parte quente do código, assim o custo da criação e armazenamento das regiões é compensado pelas otimizações aplicadas naquelas regiões e também por não precisar interpretar

as instruções ao reutilizar as regiões. Um estudo importante é pesquisar as técnicas de detecção de código quente e entender como elas se comportam com cada técnica de formação de regiões. Outro estudo relevante a esta pesquisa é investigar quais otimizações dinâmicas possuem um desempenho melhor com cada técnica.

Referências Bibliográficas

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2-13, 2006.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education Inc., 2006.

[3] Y. Almog, R. Rosner, N. Schwartz, and A. Schmorak. Specialized dynamic optimizations for high-performance energy-efficient microarchitecture. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO' 04)*, page 137, 2004.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[5] S. Banerjia, W. A. Havanki, and T. M. Conte. Treegion scheduling for highly parallel processors. In *Proceedings of the 3rd International Euro-Par Conference on Parallel Processing*, pages 1074–1078, 1997.

[6] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itaniumr-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, page 191, 2003.

[7] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 41, 2005.

[8] E. Borin, Y. Wu, M. Breternitz Jr., and C. Wang. Lar-cc: Large atomic regions with conditional commits. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO' 11)*, pages 54–63, 2011.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 265–276, 2003.

[10] H. Chen, W. Hsu, J. Lu, P. Yew, and D. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the international symposium on Code generation and optimization (CGO' 03)*, pages 79–90, 2003.

[11] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, 2000.

[12] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. Fx!32: A profile-directed binary translator. *IEEE Micro*, pages 56–64, 1998.

[13] C. Cifuentes, B. Lewis, and D. Ung. Walkabout - a retargetable dynamic binary translation framework. In *Technical Report, Sun Microsystems Laboratories*, page 106, 2002.

[14] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution prqfiling. *Technical Report*, pages 128–137, 1993.

[15] D. Davis and K. Hazelwood. Improving region selection through loop completion. In *ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*, 2011.

[16] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: using speculation, recovery and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO' 03)*, pages 15–24, 2003.

[17] P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, 1984.

[18] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, 2000.

[19] K. Ebcioglu and E. R. Altman. Daisy: dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 26–37, 1997.

[20] B. Fahs, A. Mahesri, F. Spadini, S. J. Patel, and S. S. Lumetta. The performance potential of trace-based dynamic optimization. *Technical report*, pages 1–22, 2004.

[21] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. *Technical Report*, pages 06–16, 2006.

[22] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE' 06)*, pages 144–153, 2006.

[23] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *Computer*, 33(3):54–59, 2000.

[24] H. Guan, B. Liu, Z. Qi, Y. Yang, H. Yang, and A. Liang. Codbt: A multi-source dynamic binary translator using hardware–software collaborative techniques. *Journal of Systems Architecture*, pages 500–508, 2010.

[25] R. E. Hank, W. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 158–168, 1995.

[26] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, pages 28–35, 2000.

[27] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–154, 2005.

[28] R. J. Hookway and M. A. Herdeg. Digital fx!32: combining emulation and binary translation. *Digital Tech*, pages 3–12, 1997.

[29] S. Hu and J. E. Smith. Reducing startup time in co-designed virtual machines. *Computer Architecture News*, pages 277–288, 2006.

[30] H. Kim and J. E. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO' 03)*, pages 25–35, 2003.

[31] K. Krewell. Transmeta gets more efficeon. *Microprocessor report*, 2003.

[32] K. P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996.

[33] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 180, 2003.

[34] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, pages 1–24, 2004.

[35] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation (PLDI' 05)*, pages 190–200, 2005.

[36] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hmu. A hardware mechanism for dynamic extraction and relayout of program hot spots. *SIGARCH Computer Architecture News*, pages 59–70, 2000.

[37] A. Milenković and M. Milenković. Exploiting streams in instruction and data address trace compression. In *Proceedings of IEEE 6th Annual Workshop on Workload Characterization*, pages 99–107, 2003.

[38] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA' 07)*, pages 174–185, 2007.

[39] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices*, pages 89–100, 2007.

[40] S. J. Patel and S. S. Lumetta. replay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[41] J. P. Porto, G. Araujo, E. Borin, and Y. Wu. Trace execution automata in dynamic binary translation. In *3rd Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2010.

[42] J. P. Porto, G. Araujo, Y. Wu, E. Borin, and C. Wang. Compact trace trees in dynamic binary translators. In *Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT' 08)*, 2008.

[43] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 36–47, 2003.

[44] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. J. Patel, and S. S. Lumetta. Dynamic optimization of microoperations. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA' 03)*, page 165, 2003.

[45] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., 2005.

[46] S. Sridhar, J. S. Shapiro, and P. P. Bungale. Hdtrans: A low-overhead dynamic translator. pages 135–140, 2007.

[47] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. Hdtrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 175–185, 2006.

[48] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 134–174, 2006.

[49] R. A. Uhlig. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29:128–170, 2004.

[50] D. Ung and C. Cifuentes. Dynamic re-engineering of binary code with run-time feedbacks. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, page 2, 2000.

[51] D. Ung and C. Cifuentes. Dynamic binary translation using run-time feedbacks. *Science of Computer Programming*, pages 189–204, 2006.

[52] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 209–220, 2007.

[53] C. Wang, S. Hu, H.-S. Kim, S. R. Nair, M. Breternitz Jr., Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Asia-Pacific Computer Systems Architecture Conference*, pages 4–15, 2007.

[54] C. Wang, B. Zheng, H. Kim, M. Breternitz Jr., and Y. Wu. Two-pass mret trace selection for dynamic optimization, 2007.

[55] J. Watson. Virtualbox: Bits and bytes masquerading as machines. *Linux Journal*, page 166, 2008.

[56] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT' 05)*, pages 87-98, 2005.

[57] R. M. Zinsly, S. Rigo, and E. Borin. Compressing variable-length instruction traces. *Simpósio em Sistemas Computacionais*, 0:103–109, 2012.