Otimizações para Acesso à Memória em Tradução Binária Dinâmica

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Wesley Attrot e aprovada pela Banca Examinadora.

Campinas, 12 de Dezembro de 2008.

Prof. Dr. Guido Araújo (Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues CRB8a / 2116

Attrot, Wesley
At86o Otimizações para acesso à memória em tradução binária dinâmica / Wesley Attrot -- Campinas, [S.P. :s.n.], 2008.

Orientador : Guido Costa Souza de Araújo
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Otimização. 2. Compiladores (Computadores). 3. Alocação de recursos. 4. Arquitetura de computador. I. Araújo, Guido Costa Souza de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Optimization for memory access in dynamic binary translation.

Palavras-chave em inglês (Keywords): 1. Optimization. 2. Compilers (Computers).
3. Resource allocation. 4. Computer Architecture.

Área de concentração: Sistemas de Computação

Titulação: Doutor Ciência da Computação

Banca examinadora:

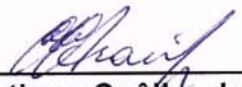
Prof. Dr. Guido Costa Souza de Araújo (IC-UNICAMP)
Prof. Dr. Cristiano Coelho de Araújo (UFPE)
Prof. Dr. Luíz Cláudio Villar dos Santos (UFSC)
Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP)
Prof. Dr. Sandro Rigo (IC-UNICAMP)
Prof. Dr. Manoel Eusébio de Lima - Suplente (UFPE)
Prof. Dr. Edmundo Roberto Mauro Madeira - Suplente (IC-UNICAMP)
Prof. Dr. Paulo Cesar Centoducatte - Suplente (IC-UNICAMP)

Data da defesa: 12-12-2008

Programa de pós-graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

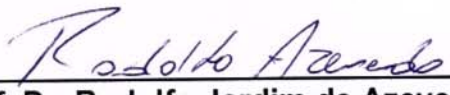
Tese Defendida e Aprovada em 12 de dezembro de 2008, pela Banca examinadora composta pelos Professores Doutores:



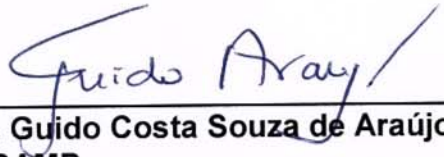
Prof. Dr. Cristiano Coêlho de Araújo
Centro de Informática / UFPE.

Prof. Dr. Luiz Cláudio Villar dos Santos
Departamento de Informática e Estatística / UFSC.

Prof. Dr. Sandro Rigo
IC / UNICAMP.



Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP.



Prof. Dr. Guido Costa Souza de Araújo
IC / UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Otimizações para Acesso à Memória em Tradução Binária Dinâmica

Wesley Attrot[1]

Dezembro de 2008

Banca Examinadora:

- Prof. Dr. Guido Araújo (Orientador)
- Prof. Dr. Cristiano Coêlho de Araújo
  Centro de Informática - UFPE
- Prof. Dr. Luíz Cláudio Villar dos Santos
  Departamento de Informática e Estatística - UFSC
- Prof. Dr. Rodolfo Jardim de Azevedo
  Instituto de Computação - Unicamp
- Prof. Dr. Sandro Rigo
  Instituto de Computação - Unicamp
- Prof. Dr. Manoel Eusébio de Lima (Suplente)
  Centro de Informática - UFPE
- Prof. Dr. Edmundo Roberto Mauro Madeira (Suplente)
  Instituto de Computação - Unicamp
- Prof. Dr. Paulo César Centoducatte (Suplente)
  Instituto de Computação - Unicamp

[1]Suporte financeiro do CNPq (processo 141703/2004-0) 2004–2008

"Almost anything can be accomplished with a strong sense of purpose."

Resumo

Tradutores binários dinâmicos ou **DBTs**[2], são programas projetados para executar, em uma arquitetura-alvo, programas binários de arquiteturas diferentes, realizando assim a tradução do programa binário em tempo de execução. Eles também podem ser utilizados para se melhorar o desempenho de programas nativos de uma dada arquitetura.

DBTs podem coletar informação de *profile* da aplicação em tempo de execução, habilidade essa impossível para um compilador estático. Este tipo de informação pode ser usada pelos DBTs para realizar novos tipos de otimizações, não possíveis em um compilador estático, seja por falta de informação do comportamento do programa, ou por não conhecer que regiões do código são mais importantes para otimizar, em detrimento de outras.

Como os DBTs gastam tempo para traduzir o código binário, é muito importante que os processos de tradução e otimização sejam extremamente rápidos, para que o impacto final no tempo total de execução seja o mínimo possível. Desta forma, para um tradutor binário dinâmico é essencial saber onde aplicar as otimizações, isto é, descobrir quais regiões do código traduzido são realmente importantes e que podem resultar em ganhos de desempenho. Uma vez que tais regiões tenham sido identificadas, os DBTs irão aplicar às mesmas, otimizações de código de forma a tentar compensar o tempo gasto na tradução do programa binário e mesmo melhorar o desempenho da aplicação traduzida.

Como o acesso à memória é algo custoso para um programa, evitá-lo em um ambiente dinâmico pode fazer com que o programa traduzido obtenha ganhos de desempenho, compensando assim parte do tempo gasto no processo de tradução

Com isso, neste trabalho investigou-se o ganho de desempenho que pode ser obtido em um ambiente de tradução dinâmica ao se tentar otimizar os acessos à memória que o programa traduzido realiza dentro das regiões de código selecionadas para otimização. O processo de otimização tenta, tanto quanto possível, evitar acessos à memória principal do computador, transformando-os em acessos à registradores da arquitetura alvo.

Como grande parte das otimizações de código necessita de informações de fluxo de dados para poder realizar transformações de código, este trabalho também investigou

[2]Do inglês, Dynamic Binary Translators

uma nova forma de se melhorar as análises de fluxo de dados que são executadas em trechos limitados de código pelo tradutor binário dinâmico.

Os resultados mostram que otimizar os acessos à memória produz ganhos pequenos, da ordem de 2%. No tocando a melhora da informação de fluxo de dados, descobriu-se que quando se busca por registradores disponíveis, pode-se descobrir que quase 25% do total dos registradores investigados estão de fato vazios e podem ser utilizados em otimizações.

Abstract

Dynamic binary translators or DBTs, are programs designed to execute, in a target architecture, binary programs from different architectures, performing the translation of the binary program during the execution time. They can also be used to improve the performance of native programs for a specific architecture.

DBTs can collect profile information from the application during runtime, this skill is impossible for a static compiler. This kind of information can be used by the DBTs to perform new kinds of optimizations, not possible in the static compiler, due to few information about the program's behavior, or does not know the regions of the code that are more important to optimize, in detriment of others.

DBTs spend time translating the binary code, so is very important that the translation and the optimization process, both be as fast as possible, to the impact in the overall execution time, be the minimum possible. In this way, for a dynamic binary translator, is essential to know where to apply the optimizations, that is, find out what regions of the translated code are really important and that can generate performance improvements. When these regions are identified, the DBTs apply code optimizations in these regions to compensate the time spend to translate the binary program and even improve the performance of the translated aplication.

Memory access is a expensive operation for programs, to avoid it in a dynamic environment may result in performance improvement in the translated program, compensating the time spend to translate the binary.

In this work, we investigate the performance improvement that can be achieved in a dynamic translation environment when we optimize the memory access that the translated program performs inside the regions selected for optimization. The optimization process tries, when possible, to avoid access to the main computer memory, transforming them into registers access of the target architecture.

Many code optimizations need data flow information to perform code transformations, in this work we also investigate a new way to improve the data flow analysis that are performed in constraint regions of code by the dynamic binary translator.

The results show that optimize the memory access produce small gains, about 2%.

When we try to improve the data flow information, we have discovered that when we are looking for available registers, we can find that almost 25% of the investigated registers are empty and can be used for optimizations.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Símbolos

$\mathtt{R}_\alpha$	Um registrador qualquer da arquitetura
$\mathtt{i}_n$	n-ésima instrução do programa
C	Custo
C_T	Custo Total
$C_{r,i}$	Custo do registrador r na instrução i
$t_{n,n-1}$	Custo de transferência do registrador da instrução $\mathtt{i}_{n-1}$ para o registrador na instrução $\mathtt{i}_n$
$t_{n,n-1}^{\mathtt{R}a,\mathtt{R}b}$	Custo de transferência do registrador $\mathtt{R}_b$ na instrução $\mathtt{i}_{n-1}$ para o registrador $\mathtt{R}_a$ na instrução $\mathtt{i}_n$
$C_T^{(\mathtt{R}_a,\mathtt{i}_n)}$	Custo total computado até o momento para o registrador $\mathtt{R}_a$ na instrução $\mathtt{i}_n$
$P_{\mathtt{R}_\alpha}$	Rótulo utilizado para diferenciar registradores das escolhas do algoritmo de programação dinâmica
P_f	Registrador que gera o menor custo ao final do algoritmo de programação dinâmica

Capítulo 1

Introdução

Uma Máquina de Turing é uma representação abstrata e formal de uma máquina universal capaz de realizar computações e que inclusive pode modelar qualquer computador digital [30], o que permite concluir que uma Máquina de Turing é capaz de executar qualquer programa da atualidade bem como simular qualquer computador atual.

Atualmente pode-se dizer que o conceito de uma máquina simular outra máquina não é mais apenas um conceito abstrato. Em tradução binária dinâmica faz-se justamente isso, onde existe um programa (tradutor binário) que executa em uma máquina-alvo e que emula uma máquina-origem para poder executar na máquina alvo programas escritos para a máquina origem, e que têm, portanto, uma "linguagem binária" diferente da máquina onde são executados.

Os tradutores binários, ao emularem uma máquina em outra, gastam tempo no processo de traduzir a "linguagem binária estrangeira" em linguagem binária nativa, então eles tentam compensar o tempo gasto na tradução realizando otimizações no programa binário traduzido. Otimizações de código têm por objetivo fazer um programa executar mais rapidamente, ao executar um mesmo trabalho em uma mesma máquina, a questão chave é identificar que porções do código é que precisam ser melhoradas [41].

Um compilador comum não é capaz de saber que regiões de um programa são mais relevantes e trata todas com a mesma importância, ao passo que um tradutor binário tem como saber quais os trechos mais relevantes, pois ele pode analisar o programa traduzido enquanto o mesmo executa. Com isso, ele pode aplicar as otimizações que desejar, somente nos trechos mais relevantes da aplicação, na esperança de compensar o tempo gasto na tradução e até mesmo conseguir um desempenho melhor do que se o programa "estrangeiro" executasse na máquina origem. Para exemplificar tal situação, considere a Figura 1.1, onde é apresentado o grafo de fluxo de controle para um programa fictício.

Na Figura 1.1 observa-se que o programa possui dois *loops* aninhados. Um compilador estático, ao otimizar tal programa, geralmente irá considerar o *loop* mais interno com um

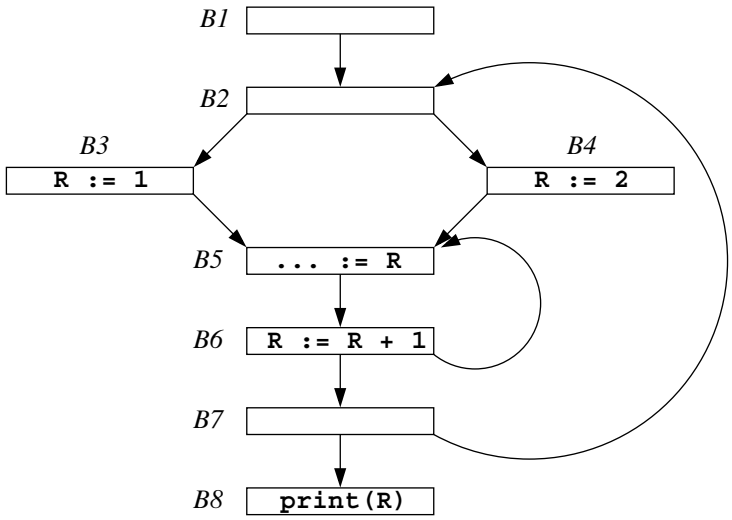


Figura 1.1: Grafo de fluxo de controle de um programa fictício

peso maior que o externo, pois espera-se que o *loop* mais interno seja o mais executado, mas será pouco provável que possa realizar alguma otimização no programa apresentado.

Suponha agora que tal programa esteja sendo executado por um tradutor binário e que este identificou que a seqüência de blocos básicos $B3$, $B5$ e $B6$ são muito executados consecutivamente. Esta seqüência de blocos é o que se chama de *trace* [37, 58], isto é, uma seqüência de blocos que é executada, sendo que o fluxo de execução entra somente no primeiro bloco do *trace*. A identificação do *trace* e sua construção são apresentadas na Figura 1.2. Observe que no grafo de fluxo de controle, existem arestas chegando ao bloco básico $B5$, mas no *trace* da Figura 1.2 o fluxo de execução sempre se iniciará por $B3$ e este será sempre o único bloco a preceder $B5$. Isto não quer dizer que as arestas de entrada em $B5$ deixaram de existir, o que ocorre de fato é que os blocos básicos do *trace* são duplicados e otimizados a parte, e depois inseridos novamente no grafo de fluxo de controle como se fossem um único super-bloco básico, como mostra a Figura 1.3, onde o *trace* ainda não foi otimizado. Essa duplicação corresponde a operação de inserir **código de compensação** [67].

Um fato interessante é que a formação do *trace* indica que, ao contrário do que um compilador comum possa pensar, o *loop* mais interno praticamente não é executado, mas somente o *loop mais externo*. Na Figura 1.1 o bloco básico $B2$ possui dois sucessores, mas a formação do *trace* indica que o sucessor representado pelo bloco básico $B3$ é tomado a maioria das vezes. Tais fatos não são acessíveis a um compilador, pois somente em tempo de execução é que se pode tomar conhecimento do real comportamento de um programa, comportamento este que pode variar muito dependendo do conjunto de dados de entrada. Obviamente, o *trace* apresentado na Figura 1.2 foi encontrado por uma técnia fictícia,

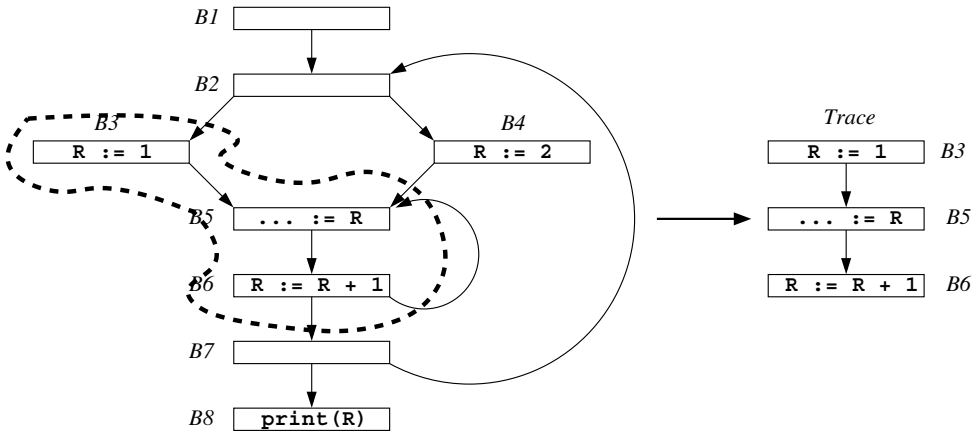


Figura 1.2: Identificando um *trace*

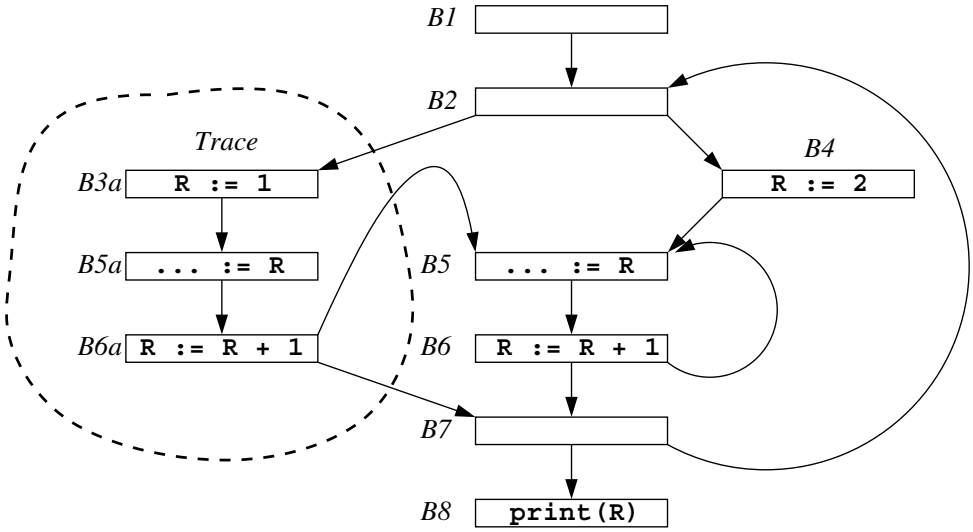


Figura 1.3: Posicionamento do *trace* dentro do programa

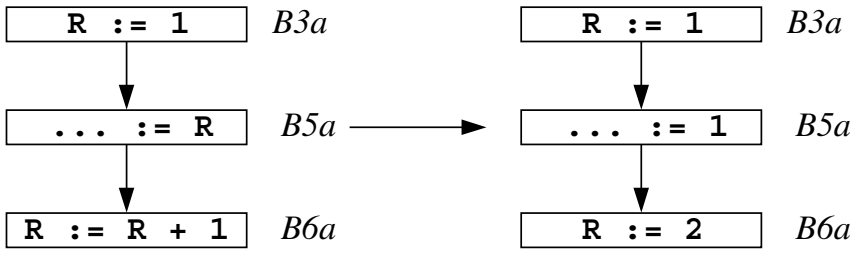


Figura 1.4: *Constant Propagation*

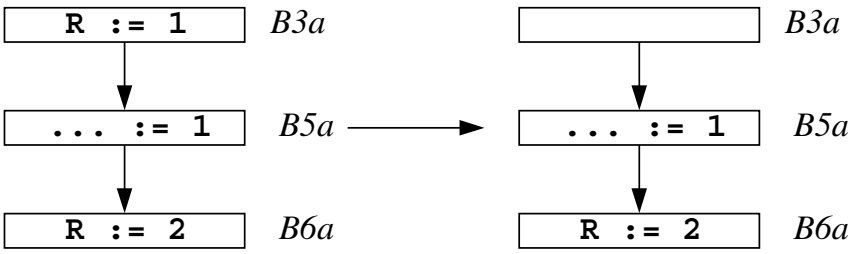


Figura 1.5: *Dead Code Elimination*

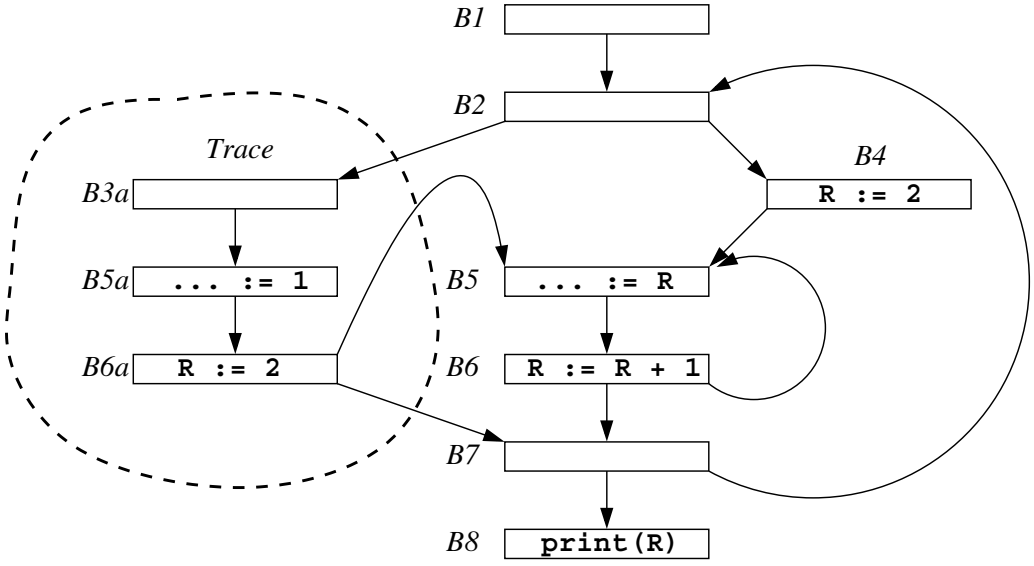


Figura 1.6: Resultado final da otimização do *trace*

onde ele foi escolhido para poder ilustrar o conceito de otimização em *traces*.

Na Figura 1.1 o bloco básico $B5$ possui três arestas de entrada, mas no bloco básico $B5a$, duplicado de $B5$ na Figura 1.3, só existe uma aresta de entrada. Tal configuração leva a novas oportunidades de otimização, como mostra a Figura 1.4, onde a aplicação de *Constant Propagation* [7,10,58] fez com que as instruções em $B5a$ e $B6a$ fossem otimizadas de forma a não mais lerem o valor da variável `R`. Em $B5a$, `R` foi substituído pelo valor 1 e em $B6a$ uma operação de soma foi transformada em uma atribuição de um valor constante.

As modificações criadas por *Constant Propagation* fizeram com que a definição de `R` no bloco básico $B3a$ se tornasse sem efeito, podendo assim ser removida por *Dead Code Elimination* [7, 10, 58] como mostra a Figura 1.5.

Com as otimizações realizadas dentro do *trace*, o total de instruções foi reduzido de 3 para 2, onde as instruções restantes são atribuições de valores constantes. O resultado final do processo de otimização é mostrado na Figura 1.6. Embora o programa-exemplo seja simples, ele serve para mostrar que é possível partir de um programa onde aparentemente nenhuma otimização é aplicável, e chegar a um outro onde diversas transformações foram realizadas.

As otimizações apresentadas só foram possíveis por terem sido aplicadas dentro de um *trace* de código, o qual é tratado como um super-bloco básico. Se o programa da Figura 1.6 fosse real, poderia se dizer que haveria uma melhora em seu tempo de execução, pois o seu comportamento é executar muitas vezes o *loop* mais externo, e poucas vezes ou nenhuma vez o *loop* mais interno, onde os blocos básicos $B3a$, $B5a$ e $B6a$ serão executados consecutivamente a maior parte do tempo.

Um processo semelhante de otimização poderia ter sido realizado a partir do bloco básico $B4$, mas como o fluxo de controle praticamente não passa por ele, seria um gasto inútil de esforço e tempo realizar tal processo de otimização.

1.1 Pesquisa Realizada

A bem conhecida diferença de velocidade do processador e da memória do computador e tema de pesquisa para diversas otimizações, como por exemplo alocação de registradores **??** e *preload* de dados, onde os dados são carregados para a *cache* do processador antes de serem necessários. Acessar a memória sempre se constitui em uma penalidade para o programa, tanto que os modernos processadores possuem uma quantidade considerável de memória *cache* (um Intel Core2 Duo pode ter até 4MB de memória *cache*) justamente para tentar compensar o tempo gasto nos acessos a memória principal do computador.

Em uma ambiente de tradução dinâmica, o problema de se acessar a memóra também se faz presente no código traduzido, desta forma se for possível, evitar muitos acessos à

memória nos *traces* muito executados da aplicação traduzida, pode-se obter algum ganho de desempenho no tempo da aplicação traduzida, compensando assim o tempo gasto pelo tradutor binário no processo de tradução.

Desta forma, neste trabalho, investigou-se o ganho de desempenho que pode ser obtido em um ambiente de tradução dinâmica ao se tentar otimizar os acessos à memória que o programa traduzido realiza dentro das regiões de código selecionadas para otimização. O processo de otimização tenta, tanto quanto possível, evitar acessos à memória principal do computador, transformando-os em acessos à registradores da arquitetura alvo. A investigação deu-se através da criação de uma nova otimização, chamada de *hole allocation*, a qual foi implementada no tradutor binário chamado StarDBT [72], sendo que os resultados foram obtidos utilizando-se o benchmark SPEC CPU 2000 [3].

Como grande parte das otimizações de código necessita de informações de fluxo de dados para poder realizar transformações de código, este trabalho também investigou uma nova forma de se melhorar as análises de fluxo de dados que são executadas em trechos limitados de código pelo tradutor binário dinâmico. Tal técnica é chamada de *Cold Code Analysis* e também foi implementada no StarDBT e avaliada utilizando-se o SPEC CPU 2000.

1.2 Organização da Tese

Este trabalho está dividido da seguinte maneira:

- **Capítulo 2:** Define o conceito de tradução binária dinâmica e suas aplicações, bem como apresenta alguns trabalhos relacionados com esta Tese.
- **Capítulo 3:** Descreve o ambiente utilizado para o desenvolvimento deste trabalho de pesquisa.
- **Capítulo 4:** Apresenta uma forma de se obter mais informações nas análises de fluxo de dados que um tradutor binário possa vir a precisar.
- **Capítulo 5:** Apresenta a técnica denominada de *hole allocation* e como utilizá-la como uma otimização em um tradutor binário.
- **Capítulo 6:** Por fim, são apresentadas as conclusões obtidas, as contribuições deste trabalho e os trabalhos futuros.

Capítulo 2

Tradutores Binários Dinâmicos

Tradutores binários dinâmicos ou **DBTs**[1], são programas projetados para executar, em uma arquitetura alvo, programas binários de arquiteturas diferentes, realizando assim a tradução do programa binário em tempo de execução. DBTs também podem ser utilizados para se melhorar o desempenho de programas nativos de uma dada arquitetura [14].

Nos últimos anos, **DBT**[2] se tornou um assunto interessante devido as suas vantagens, sendo tema de pesquisa de diversos trabalhos [14, 24, 35, 36, 56]. DBT provê portabilidade de código entre arquiteturas totalmente diferentes, oferecendo um novo patamar de liberdade em relação ao ambiente de computação em que se pretende executar a aplicação. Um DBT converte o código a ser executado na nova arquitetura, sem ser necessário recompilar o código fonte, tendo como outro benefício a possibilidade de se usar os novos recursos, da arquitetura alvo, não disponíveis na época em que o código foi escrito, como por exemplo, um número maior de registradores [8].

DBTs traduzem cada instrução a ser executada em uma, ou mais instrução equivalente da arquitetura alvo. Alguns DBTs possuem ainda uma *codecache* [14] para armazenar as instruções traduzidas e executá-las; desta forma a *codecache* é uma otimização que poupa tempo, na medida em que evita o custo extra de traduzir as mesmas instruções repetidamente [14].

DBTs podem também coletar informação de *profile* da aplicação em tempo de execução, habilidade essa impossível para um compilador estático. Este tipo de informação pode ser usada pelo DBT para realizar novos tipos de otimizações, não possíveis em um compilador estático, seja por falta de informação do comportamento do programa, ou por não conhecer que regiões do código são mais importantes para otimizar, em detrimento de outras.

Como um DBT é um programa que gasta tempo para traduzir o código binário, é

[1] Do inglês, Dynamic Binary Translators
[2] Dynamic Binary Translation

muito importante que os processos de tradução e otimização sejam extremamente rápidos, para que o impacto final no tempo total de execução seja o mínimo possível. Desta forma, para um DBT é essencial saber onde aplicar as otimizações, isto é, descobrir quais regiões do código traduzido são realmente importantes, e que podem resultar em ganhos de desempenho. Gastar tempo otimizando regiões de código que serão executadas uma ou duas vezes não é vantajoso, ao passo que gastar tempo otimizando regiões de código que executarão milhares de vezes, pode ser extremamente benéfico. Se, por exemplo, 90% do tempo de execução de um programa estiver em 10% do código, então vale muito a pena identificar o código correspondente a estes 10% e otimizá-lo, pois se reduzirmos o seu tempo de execução em 50%, o programa traduzido irá executar em um tempo equivalente a 55% do tempo do programa não otimizado.

Uma particularidade de se otimizar dinamicamente um programa em execução, é o fato de se poder analisar o comportamento do mesmo. Um DBT, por traduzir um programa em execução, pode instrumentar o mesmo, e analisar o código sendo executado descobrindo quais regiões são de fato importantes. Às vezes um programa pode se comportar de forma distinta para entradas distintas, mas para um DBT isso não é problema, pois ele sempre saberá qual a região relevante do código, bastando para isso instrumentar o código traduzido. Tais regiões altamente executadas são também chamadas de regiões "quentes".

2.1 Traces

Para identificar regiões quentes, um DBT constrói **traces**. *Traces* nada mais são que uma seqüência de instruções, incluindo saltos, mas não incluindo *loops*, que é executada para um conjunto de dados de entrada de um programa [37, 58].

Em geral, os *traces* são construídos instrumentando-se o programa traduzido, e os monitorando de forma a identificar quais *traces* são bons candidatos a serem otimizados, como por exemplo, contar quantas vezes cada um deles é executado, e então otimizar os *traces* que são executados um número de vezes maior que um determinado limiar.

Técnicas para seleção de *traces* são estudas há muito tempo. Chang e Hwu [27] usaram contadores nas arestas entre os blocos básicos, de forma que o bloco básico que possua a aresta de saída mais executada seja conhecido. Um *trace* é construído seguindo-se um caminho que acompanha sempre a aresta mais tomada. Ball e Larus [15], usam uma heurística onde um conjunto com todos os caminhos acíclicos para um programa são determinados e a cada um deles é atribuído um estado, sendo que existe um contador associado a cada caminho possível, que é incrementado cada vez que este caminho é executado.

A técnica *Next Executing Tail* (NET) [34] é muito popular entre os sistemas de otimização dinâmica. Nesta técnica, são mantidos contadores em cada bloco básico que

é alvo de um *backward branch*[3]. Cada caminho é divido em *path-head* e *path-tail*. Informação de *profile* é utilizada para predizer o *path-head* de um *hot path* e especulação é utilizada para predizer o *path-tail*. Um *path-head* indica que o programa está executando uma região quente e o caminho de execução a seguir pertence a essa região. A idéia desta técnica é focar todo o esforço de *profiling* somente na parte inicial de uma região quente, limitando assim o número de contadores a um valor máximo igual ao número de blocos básico de programa. Com tal heurística, NET constrói *traces* a partir de *loops*, sendo que quando o contador associado a um *path-head* ultrapassar um determinado limiar, se inicia o processo de identificar um *trace*. Hiser [49] e Hiniker [48] apontam que 40,2% dos *traces* gerados por NET consistem de somente um bloco básico, e que este bloco possui um número médio de 14,8 instruções.

Hiniker et. al. [48] propuseram uma técnica chamada *Last Executed Iteration* (LEI). Tal técnica é um pouco semelhante a técnica NET. Em LEI, existe um *history buffer* que armazena os desvios mais tomados recentemente. Se o alvo de um desvio está presente no *history buffer*, então um *loop* foi executado e o seu caminho se encontra no *buffer*, e este *loop* recém-executado passa a ser analisado para se construir um *trace*. De forma semelhante a NET, o processo de construção do *trace* terá início quando o *loop* executar um número de vezes maior que um determinado limiar, iniciando-se a construção do *trace* pelo *loop header* que em NET equivale ao *path-head*. Dados os endereços e o alvo, de cada desvio que é realizado, o caminho completo é reconstruído adicionando repetidamente as instruções que se encontram entre o alvo da instrução de desvio corrente e o endereço do próximo desvio. O *trace* estará concluído quando um ciclo for completado ou a próxima instrução for o início de algum *trace* existente. Segundo Hiniker et. al. [48], a segunda condição de parada para a construção do *trace*, faz com que não haja muita duplicação de código, como em NET, fazendo com que LEI seja mais eficiente para capturar ciclos do que a técnica NET. Outra afirmação de Hiniker et. al, é que os *traces* gerados por LEI possuem em média 18,3 instruções, cobrindo aproximadamente 90% do tempo de execução do programa.

Gal e Franz [39] criaram uma técnica chamada *Trace Tree*. Ela basicamente procura por regiões contendo *loops* altamente executados (regiões quentes), e os representa em uma estrutura de árvore. Para construir a árvore, os *loop headers* são monitorados, sendo que quando o fluxo de execução passar por um *loop header* um número de vezes maior que determinado limiar, inicia-se a construção do *trace*, o qual irá ter o *loop header* como início. Na construção do *trace*, são seguidos os blocos básicos subseqüentes até que um *backward branch* para o *loop header* seja encontrado. Neste momento tem-se um *trace*. Até este momento o *trace* gerado é uma seqüência de blocos, como em outras técnicas. Se existir algum outro *loop* no programa que tenha o mesmo *loop header* que o *trace* gerado,

[3]Desvio para trás

então ao invés de se construir um novo *trace*, os seus blocos serão adicionados ao *trace* já existente. A adição do novo conjunto de *traces* se dá criando-se um novo ramo no *trace*, isto é, em algum momento um bloco básico do *trace* irá possuir mais de um sucessor, criando assim uma estrutura em árvore. A técnica pretende com isso, aglomerar um *loop* e todos os seus *inner loops*[4] em um único *trace*, uma vez que muitos blocos básico são compartilhados pelos diversos *loops*. Em outras técnicas, cada *loop* daria origem a um único *trace*, havendo assim duplicação de código. Na construção de uma *trace tree* também pode haver duplicação de código, levando a construção de *traces* relativamente grandes; para evitar tal situação a técnica utiliza algumas condições para limitar crescimento e o tamanho da *trace tree*.

Bala et. al. criaram a técnica chamada *Most Recent Executed Tail* (MRET) [14], usada no DBT chamado Dynamo [14]. A técnica consiste em associar um contador a *hot spots* (uma instrução) do programa, onde tais *hot spots* são, geralmente, alvos de *backward branchs*. O objetivo é capturar os *loops* do programa. Quando um contador associado a um *hot spot* ultrapassa um determinado limiar, o *trace* começa a ser construído, marcando-se os blocos básicos que são percorridos após o *hot spot*. A construção continua até que uma condição de parada seja alcançada, como por exemplo, alcançar algum bloco já marcado. A técnica se baseia na idéia de que uma vez que uma instrução é definida como um *hot spot*, as instruções subseqüentes também serão quentes.

Uma variação da técnica MRET foi criada e é chamada de *Two Pass MRET* ou simplesmente MRET2[5], sendo que tal técnica, no momento da escrita deste texto, se encontrava sob processo de patente, não havendo assim outra forma de documentação além daquela apresentada ao escritório de patentes [5]. Em MRET2 um *trace* potencial é construído inicialmente da mesma forma que em MRET, então o contador associado ao *hot spot* que deu origem ao potencial *trace* é reinicializado e um novo *trace* potencial é encontrado a partir do mesmo, novamente utilizando o método MRET. Depois que dois *traces* potenciais construídos com MRET estão associados a um mesmo *hot spot*, MRET2 realiza a interseção entre os mesmos, construindo assim o *trace* final contendo os blocos que são comuns ao dois *traces* potenciais.

Uma vez que um *trace* tenha sido construído e identificado como um bom candidato a otimização, um DBT pode aplicar transformações de código bem conhecidas, como *copy propagation* [7,58], *CSE* [7,58], *dead code elimination* [7,58]; e outras que podem ser mais específicas e dependentes do ambiente de tradução e execução. Alocação de registradores, em particular, é uma otimização que demanda mais esforço, particularmente quando as arquiteturas alvo e origem possuem uma quantidade de registradores diferentes.

A qualidade dos *traces* onde a otimização é aplicada, tem um grande impacto no resul-

[4]Laços internos

[5]USPTO Application #: 20070079293

tado final, podendo maximizar os ganhos ou resultar em perda de desempenho. As técnicas de detecção de *traces* são fundamentais para o sucesso de otimizações [48]. Um *trace*, para proporcionar bons resultados quando otimizado deve possuir três características [60]: um elevado *trace length*, uma *Execution Coverage* (EC) elevada e uma alta *Completion Rate* (CR), as quais são definidas a seguir:

- **Trace Length**

  Corresponde a quantidade de instruções presentes em um *trace*. No caso do *trace* ser formado por apenas um bloco básico, será muito difícil encontrar oportunidades de otimização no mesmo.

- **Execution Coverage**

  Execution Coverage (EC) corresponde a porcentagem do tempo de execução do programa gasto pelo *trace*, isto é, a relevância do *trace* para o tempo total de execução. Se um programa executa em 100 segundos e 10 segundos são gastos dentro do *trace* Y, estão diz-se que Y tem EC = 10%.

  Pela Lei de Amdahl [9], o ganho em desempenho de qualquer otimização em um *trace*, será limitado por sua EC. Se for possível obter um ganho de desempenho de 50% em um *trace* que possua EC = 10%, o efeito no programa como um todo será uma melhora de apenas 5%. Desta forma, EC é um importante fator a se considerar quando se seleciona um *trace* a ser otimizado.

- **Completion Rate**

  Um *trace* é formado por muitos blocos básicos e existem instruções de desvio no meio de um *trace*, que podem fazer com que o fluxo de execução deixe o *trace* antes de atingir o seu final. A execução de um *trace* é dita completa quando todas as instruções do *trace* são executadas, da primeira a última, sem que nenhuma instrução de desvio seja tomada em direção para fora do *trace*. A *completion rate* (CR) é uma medida para indicar a quantidade de execuções completas de um dado *trace*, em relação número total de vezes que o *trace* é executado, mesmo que de forma não completa. Desta forma CR é a porcentagem de execuções completas de um *trace*.

Técnicas de construção de *traces* e análise de *traces* são ferramentas para identificar que porções de código são propícias a serem otimizadas, ao passo que o foco deste trabalho é a aplicação de otimizações em *traces* e ferramentas para auxiliar tais otimizações. Uma análise completa de tais características aplicadas a *traces* gerados pela técnica MRET2 a todos os programas do benchmark SPEC CPU 2000 [3] no DBT StarDBT [72], pode ser

encontrada em [60]. Em Hiniker et. al [48], pode ser encontrada uma comparação entre NET e LEI.

2.2 Trabalhos Relacionados

Tradutores binários dinâmicos utilizam otimizações de código, entre outras técnicas, para tentar compensar o custo da tradução do código binário. A seguir, serão apresentados e descritos, de forma breve, alguns DBTs.

2.2.1 FX!32

O FX!32 [50] realiza a tradução e execução de programas x86 em uma arquitetura Alpha executando WindowsNT. O FX!32 é uma DLL que intercepta a chamada de sistema *CreateProcess* do Windows. Se a imagem especificada na chamada *CreateProcess* é um código x86, o FX!32 invoca o módulo de *Run-Time* para executar a imagem. O código é emulado e informações sobre sua execução são coletadas. Assim que a imagem x86 não for mais executada, o módulo tradutor traduz o código da imagem que foi emulada. As informações coletadas são utilizadas para fazer as devidas otimizações. O código traduzido é então armazenado em um banco de dados, o qual será executado da próxima vez que a imagem for requisitada para execução.

2.2.2 Dynamo

Dynamo [14] opera com um sistema de interpretação de instruções PA-8000 executando sobre um processador PA-8000 [53]. O código original é interpretado até que um *trace* muito executado seja encontrado. Quando tal *trace* é identificado, ele é otimizado e armazenado em uma *cache*, para ser executado assim que o programa passar novamente pelo *trace* original.

A principal otimização do Dynamo é a transformação de *branchs*, onde os mesmos são modificados para que o caminho de *fall-through* seja direcionado para dentro do *trace*. Outras otimizações incluem *copy propagation*, *constant propagation*, *strength reduction*, *loop invariant code motion* e *loop unrolling* [7].

2.2.3 IA-32EL

O IA-32EL [16] é um DBT para executar código IA32 em processadores Itanium utilizando interpretação e tradução binária dinâmica. Ele foi projetado para trabalhar com sistemas operacionais baseados no Itanium, o que é obtido separando-se os mecanismos de

tradução em uma biblioteca independente do sistema que se comunica com um biblioteca dependente do sistema. Há dois modos de tradução, o *Cold Code* que simplesmente interpreta o código IA32 e o *Hot Code* que traduz um conjunto de blocos (um super-bloco ou *trace*) se o número de execuções do trecho de código ultrapassar um determinado limiar. Os blocos traduzidos são armazenados em uma *cache* e executados sempre que requeridos. Para o tratamento de exceções o IA-32EL fornece um mecanismo no qual, quando uma exceção é gerada, o sistema operacional a transmite para o IA-32 EL, ao invés de transmiti-la para a aplicação. Assim o IA-32EL se encarrega de converter a exceção de uma arquitetura de 64 bits para um exceção no padrão IA32.

2.2.4 CMS

O CMS (*Code Morphing Software*) [32] está presente no processador Crusoe da Transmeta, uma implementação completa de uma arquitetura x86. O Crusoe consiste em um processador VLIW (*Very Long Instruction Word*) nativo com uma camada de software (CMS). Ao contrário dos outros tradutores apresentados, que funcionam em nível de aplicação, o CMS funciona em nível de sistema. O CMS se baseia em um paradigma de especulação agressiva e em um mecanismo de recuperação *rollback-commit* com suporte de hardware, através da existência de registradores *shadow*. Para cada registrador x86, existe um outro registrador *shadow*. Na ocorrência de alguma exceção ou tomada de algum *branch* indevidamente, uma operação de *rollback* substitui os valores dos registradores x86 pelo conteúdo dos registradores *shadow*. O conteúdo dos registradores *shadow* é definido pelo último *commit* feito. Em uma operação de *commit* o conteúdo dos registradores x86 é copiado para os registradores *shadow*. Assim como a maioria dos tradutores binários dinâmicos, o CMS inicialmente interpreta o código x86 e assim que o número de execuções de uma determinada região do código alcança um limiar, o tradutor é invocado. O tradutor traduz e otimiza um determinado trecho de código e o guarda em uma *cache*, para que o mesmo possa ser executado quando requisitado. O CMS é um sistema disponível comercialmente, que provê alto desempenho em uma tradução de um conjunto de instruções da arquitetura x86 em um conjunto de instruções de uma arquitetura diferente, no caso VLIW.

2.2.5 DAISY

DAISY [36] é um sistema de emulação que permite executar aplicações de diversas arquiteturas em uma arquitetura VLIW. O processo de tradução é feito com a decodificação das instruções origem em instruções RISC (se a arquitetura origem for CISC). A cada instrução RISC produzida é encontrada uma instrução VLIW na qual a instrução RISC possa ser alocada. Ao mesmo tempo, é feito o escalonamento global de instruções VLIW. O DAISY

usa um sistema agressivo de reordenação de instruções, através de especulações usando suporte de hardware, onde existem registradores *non-architected*[6] que são usados nas especulações. Se o resultado de uma especulação não é o previsto, nenhuma exceção será gerada. O mapeamento de memória é feito dividindo-se a memória do sistema VLIW em três partes: uma sendo a memória utilizada pelo programa-base, outra sendo a memória do *Virtual Machine Monitor*, responsável pela tradução do código. E por último, a memória destinada a *cache* de código traduzido. Os resultados demonstram a obtenção de um grau de paralelismo por volta de 2 instruções RISC em cada instrução VLIW (com instruções de largura 4: 2 ALU, 2 Load/Store, 1 Branch).

Considerações sobre DBT

Os sistemas de tradução binária dinâmica aqui apresentados diferem muito na estratégia de tradução e nas arquiteturas alvo e destino utilizadas. As otimizações presentes em tais sistemas se constituem em mecanismos que auxiliam a tradução de uma determinada arquitetura-origem para uma arquitetura-alvo, sendo mais de cunho particular das arquiteturas envolvidas (com excessão talvez do Dynamo), do que otimizações a plamente conhecidas na literatura, desta forma uma comparação mais profunda entre os diversos sistemas não é algo simples, bem como comparar o trabalho desenvolvido nesta Tese com as otimizações específicas de cada um destes ambientes. Deve-se resaltar que o presente trabalho pode ser adaptado para qualquer ambiente de tradução binária independentemente das arquiteturas alvo e origem.

[6] Não nativos

Capítulo 3

Ambiente de Trabalho

3.1 StarDBT

O ambiente de trabalho selecionado para o desenvolvimento dos experimentos a serem apresentados nesta Tese, foi o StarDBT [72] da Intel, desenvolvido no laboratório de nome PSL[1] o qual faz parte de tal empresa, como ferramenta de pesquisas em DBT. A estrutura do StarDBT é apresentada na Figura 3.1. Por ser uma ferramenta fechada, toda a descrição do StarDBT aqui apresentada é baseada em publicações realizadas pela Intel.

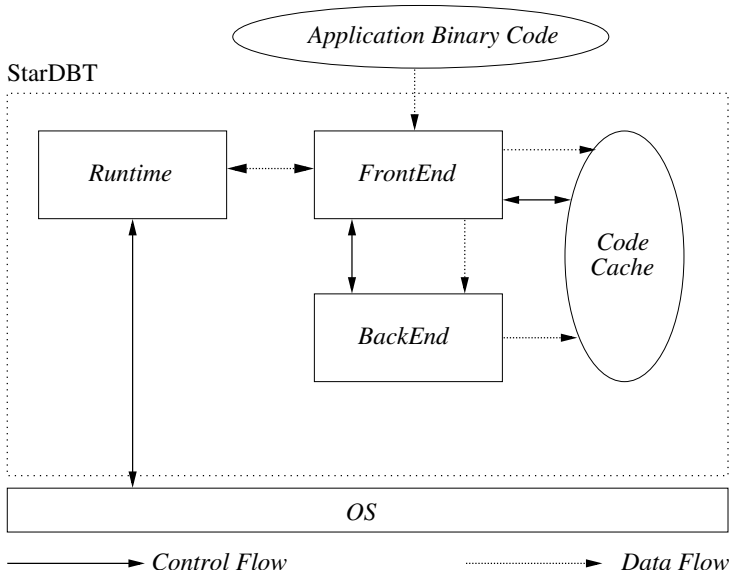


Figura 3.1: Estrutura do StarDBT

O StarDBT é um DBT multi-plataforma que funciona sob Windows x64 e Linux,

[1]Programming Systems Lab.

sendo capaz de traduzir código de IA32 para IA32 (Linux) e de IA32 para IA64 (Windows x64). Neste texto IA32 representa a arquitetura Intel x86 de 32 bits e IA64 representa a arquitetura Intel x86 de 64 bits. Abstração e modularização é a chave do StarDBT para suportar várias plataformas. Toda a parte genérica que forma um DBT é separada, modularizada e compartilhada entre as diversas plataformas. A parte que é dependente da plataforma é separada do restante do DBT e possui uma API (*Application Programming Interface*) interna para se comunicar com o mesmo.

O StarDBT possui três módulos principais: *frontend*, *runtime* e *backend*. O módulo *frontend* traduz as instruções da aplicação a ser executada, e armazena as instruções geradas em uma **code cache**, bem como controla a execução do programa traduzido que está nesta *cache*. O módulo *runtime* realiza a comunicação entre o DBT e o Sistema Operacional e entre a aplicação traduzida e o Sistema Operacional, provendo *interfaces* de I/O (entrada e saída), chamadas de sistema, manipulação de sinais, carga dinâmica de objetos compartilhados (arquivos .dll em Windows e .so em Linux) e código auto-modificável. Os módulos *frontend* e *runtime* interagem entre si para gerenciar características dependentes de plataforma, sendo que o *frontend* também é responsável por selecionar regiões de código a serem otimizadas pelo módulo *backend*.

Processo de Tradução

Tradução binária dinâmica, implica em um *overhead* extra na execução do programa traduzido; especialmente em uma arquitetura tão complexa como a x86, que possui um extenso conjunto de instruções. Como muitos sistemas, o StarDBT usa uma estratégia simples de tradução do código e quando uma região quente é detectada, ela é otimizada.

O módulo tradutor presente no *frontend*, tenta gerar instruções com o mínimo de *overhead* possível. Para muitas instruções IA32 (operações aritméticas e de movimentação de dados), o StarDBT simplesmente decodifica a instrução e as reconhece, sendo que as instruções são simplesmente copiadas na geração de código IA64 (Windows) e IA32 (Linux). Algumas instruções IA32 que não estão mais disponíveis em IA64, bem como instruções de salto, chamadas de função e **returns** precisam ser reescritas para poderem funcionar no código traduzido.

Tradução IA32 para IA64

Como dito anteriormente, algumas instruções IA32 não possuem equivalente em IA64, desta forma elas devem ser emuladas, como por exemplo as instruções **pop** e **push**, que em IA64 suportam somente operandos de 64 bits. Por exemplo, a instrução **push32 eax** é traduzida pelo StarDBT na seguinte seqüência de instruções:

```
lea esp, [rsp - 4]
```

```
mov [rsp], eax
```

A Tabela 3.1 sumariza o processo de conversão de algumas instruções de IA32 para IA64 que não podem ser simplesmente copiadas durante a tradução.

Instrução IA32 Original	**Instruções IA64 geradas**
PUSH ESP	REX. MOV R8/32,ESP LEA ESP,[RSP-4] REX. MOV [RSP],R8/32
PUSH imm/32(imm/8)	LEA ESP,[RSP-4] MOV [RSP],imm/32(imm/32*) //sign extend imm/8 to imm/32
PUSH r/32	LEA ESP, [RSP-4] MOV [RSP],r/32
PUSH m/32	REX. MOV R8/32,M/32 LEA ESP,[RSP-4] REX. MOV [RSP],R8/32
POP ESP	MOV ESP,[RSP]
POP r/32	MOV r/32,[RSP] LEA ESP, [RSP+4]
POP m/32	REX. MOV R8/32,[RSP] REX. MOV m/32,R8/32 LEA ESP,[RSP+4]
ENTER imm16,imm8	LEA ESP,[RSP-4] MOV [RSP],EBP LEA ESP, [RSP-imm16](imm16 > 0) REX. MOV R8/32,[EBP-4] REX. MOV [RSP+imm16-4],R8/32 ... REX. MOV R8/32,[EBP-4*imm8+4] (imm8 > 1) REX. MOV [RSP+imm16-4*imm8+4],R8/32 (imm8 > 1) LEA EBP,[RSP+imm16] MOV [RSP+imm16-4*imm8],EBP (imm8 > 0)

LEAVE	MOV ESP,EBP
	MOV EBP,[RSP]
	LEA ESP,[RSP + 4]

Tabela 3.1: Tradução de IA32 para IA64

Ao se analisar a Tabela 3.1 conclui-se que a tradução do código IA32 para IA64 claramente causa expansão de código, pelo fato de nos casos apresentados ser necessário gerar mais de uma instrução, bem como pela adição do prefixo REX em algumas instruções, aumentando assim o seu tamanho.

Tradução de Instruções de Controle

A tradução das instruções que controlam o fluxo de controle da aplicação (instruções de salto, chamadas de função, *syscalls* e **returns**), são um outro ponto de *overhead* para o StarDBT. Uma instrução de salto não pode ser simplesmente copiada para o código traduzido, pois o endereço alvo do desvio não é mais o mesmo, ou seja, o código para onde ele aponta está localizado em outro endereço, uma vez que foi traduzido pelo StarDBT e armazenado na *codecache*. Uma outra possibilidade, é o endereço-alvo do salto ainda não ter sido traduzido.

Desta forma, desvios condicionais/incondicionais e chamadas de função são inicialmente traduzidos de forma que o controle volte ao StarDBT para que o mesmo realize uma busca em uma tabela de endereços traduzidos, chamada **lookup table**, a fim de verificar se o endereço, para o qual a instrução deseja desviar o controle, já foi traduzido. Caso o endereço não tenha sido traduzido, o StarDBT irá iniciar a tradução do trecho em questão e depois iniciar a execução do mesmo. Caso a busca na *lookup table* indique que o endereço já tenha sido traduzido, então o StarDBT irá reescrever a instrução de controle, de forma que ela chame diretamente o novo endereço, evitando assim o *overhead* de se realizar uma busca na *lookup table* quando a instrução for executada novamente, o que tornaria a execução do código traduzido ainda mais lenta. Os saltos são realizados através da criação de *stubs* que são adicionados no fim do bloco básico. No StarDBT cada *stub* tem um tamanho em bytes fixo e é responsável por salvar o estado da aplicação e depois devolver o controle da execução de volta ao módulo *runtime*. Desta forma uma instrução de salto é transformada em um salto para o *stub* o qual se encarregará de realizar o salto real. Os diversos casos a serem tratados são os seguintes:

- **Branch Condicional:** Apresentado na Figura 3.2. A ligação entre um o bloco básico A e seus sucessores por salto condicional, é feita adicionando-se um *stub* em A para verificar e realizar os testes do salto e em seguida são adicionados outros dois *stubs* para saltar para os sucessores. Os *stubs* de teste retornam o controle da execução para o módulo *runtime*, que irá verificar se o endereço de desvio já foi traduzido. Em caso positivo devolve o controle de execução para este endereço, caso contrário, traduz o código neste endereço e depois desvia a execução para ele. Uma vez que o endereço alvo já tenha sido traduzido, o StarDBT pode reescrever o *stub* para que o mesmo desvie diretamente para o endereço alvo ao invés de passar pelo módulo *runtime*.

- **Branch Indireto:** Apresentado na Figura 3.3. Neste tipo de salto é adicionado no bloco A um *stub* para verificar e realizar os testes de salto. Em seguida é adicionado um outro *stub* que realiza uma busca pelo bloco básico B na *lookup table*. Tal busca é necessária pois o endereço de salto só vai ser conhecido após a execução da

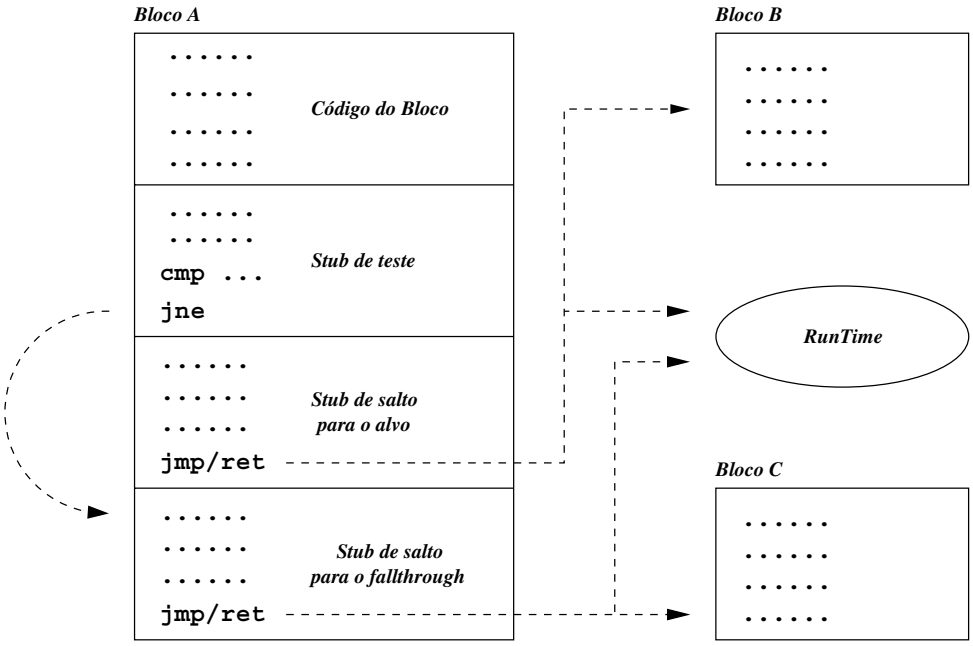


Figura 3.2: Ligação de blocos para *branchs* condicional

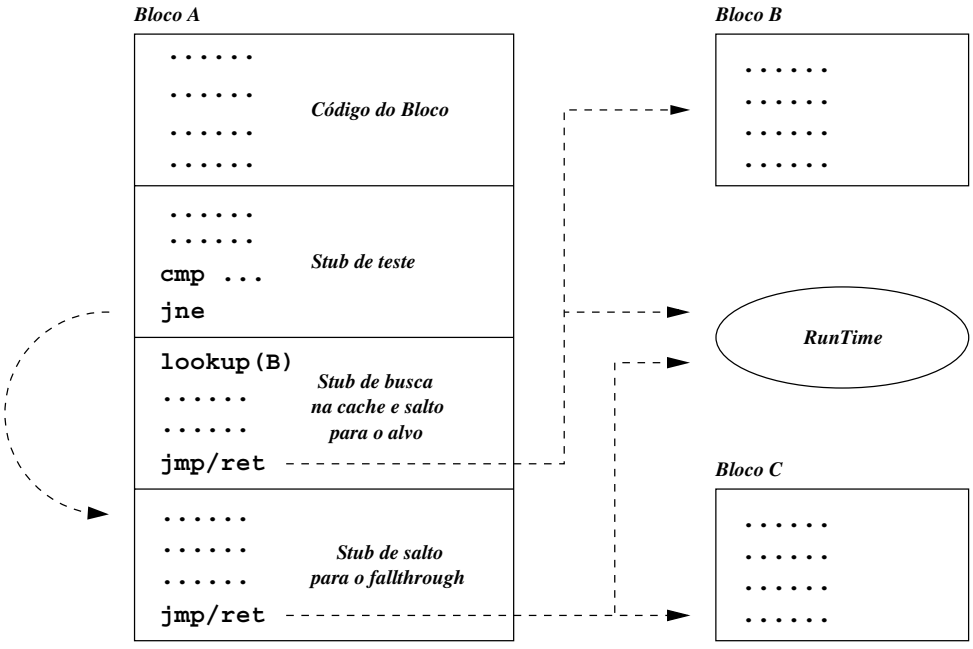


Figura 3.3: Ligação de blocos para *branchs* indireto

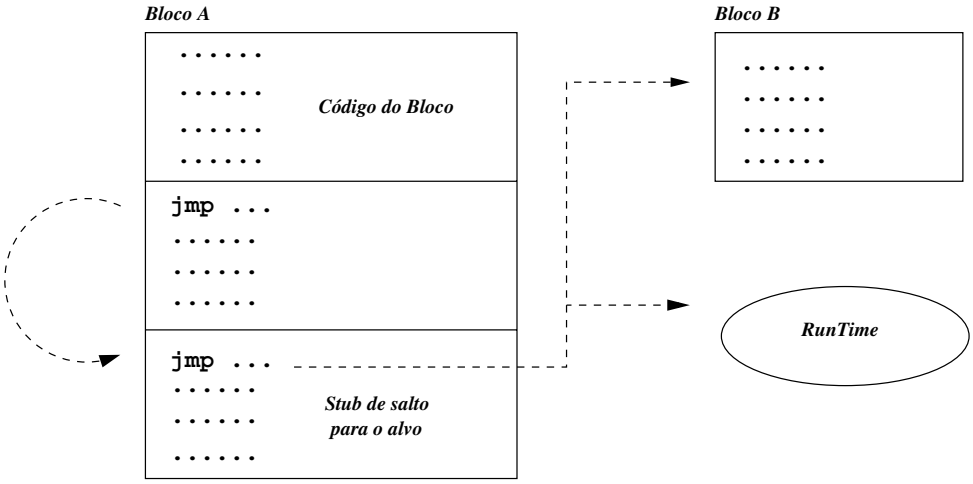


Figura 3.4: Ligação de blocos para *branchs* incondicionais

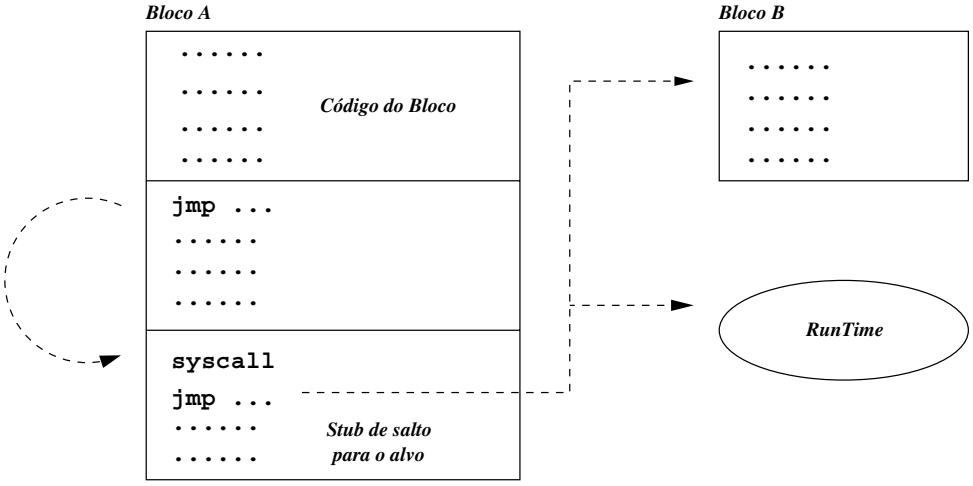


Figura 3.5: Ligação de blocos para *system calls*

última instrução antes do *branch*. Neste tipo de caso, o StarDBT sempre tem que retornar o controle da execução para o módulo *runtime* para que o mesmo busque por B e realize o salto. Para o bloco *fallthrough* é adicionado um *stub* que também devolve o controle ao *runtime*, o qual faz busca na *lookup table* por C. Tal *stub* pode posteriormente ser substituído por uma instrução que realiza o desvio sem passar pelo *runtime*, uma vez que o endereço traduzido nunca muda.

- **Branch Incondicional:** Apresentado na Figura 3.4, também se aplica a chamadas de função. Neste caso, a última instrução do bloco salta diretamente para o *stub*, de modo que o mesmo devolva o controle ao módulo *runtime* e este por sua vez faça a busca na *lookup table* pelo endereço de desvio, realizando em seguida o salto para o referido endereço. No StarDBT, chamadas de função são traduzidas desta forma, sendo tratadas como desvios incondicionais. Este tipo de desvio também pode ser reescrito posteriormente para que o controle não retorne ao *runtime*.

- **SystemCall:** Apresentado na Figura 3.5. Quando houver uma *syscall* o controle sempre irá retornar ao *runtime*, o qual irá realizar a *syscall* e devolver o resultado ao programa.

3.2 Offline Profile

O módulo *backend* é o responsável por realizar a otimização dos *traces* gerados no StarDBT. Este módulo possui uma habilidade especial: ele pode realizar o **dump** dos *traces* para um arquivo texto. Uma outra habilidade deste módulo é poder realizar a carga de um conjunto de *traces* presentes em um arquivo e usar tais *traces* no programa a ser traduzido. Obviamente esses *traces* devem ter sido gerados pelo programa que se pretende executar.

O processo *dump*/carga dos *traces* para/de arquivos texto é controlado por um módulo especial do *backend* chamado *Persistence Layer*. Quando os *traces* são gravados em arquivo, primeiro o StarDBT executa o programa traduzido e os *traces* são gerados de acordo com a técnica MRET2. Quando o programa termina a sua execução, o StarDBT terá então uma coleção com todos os traces que foram identificados. Com este conjunto de *traces*, o *Persistence Layer* os converte para um formato que permite que os mesmos possam ser carregados novamente em uma futura execução do mesmo programa.

A Figura 3.6 mostra a forma como os *traces* são representados em arquivo. O *trace* apresentado foi extraído do programa 164.gzip que integra o SPEC CPU 2000. A representação do *trace* possui diversos campos. O primeiro deles é o *trace identifier* para numerar o *trace*. Na Figura 3.6, o identificador possui número 2. Um cabeçalho chamado `ORIGINAL_ADDRESS` contém o endereço no código que foi traduzido, que gerou o *trace* em

```
Trace 2 {

  ORIGINAL_ADDRESS: 0x0804883a

  CHECKSUM: 0x9e29860b

  Predecessors 2 {
    ORG 0x080487f8
    ORG 0x08048acb
  }

  Stats {
    ExecutionFrequency: 1841987194
    NumberOfBlocks: 3
    HeadBlockSize: 40
    RealHeadBlockSz: 40
  }

  Code {
    exec_info    1841987194
    bb_org_addr    0x0804883a
    blk {
      8b 45 08                      ; mov      eax, DWORD PTR [ebp+08h]
      8d 80 40 21 11 08             ; lea      eax, DWORD PTR [eax+08112140h]
      89 45 f4                      ; mov      DWORD PTR [ebp-12], eax
      8b 45 f4                      ; mov      eax, DWORD PTR [ebp-12]
      8b 55 f0                      ; mov      edx, DWORD PTR [ebp-16]
      0f b6 04 02                   ; movzx    eax, BYTE PTR [edx+eax]
      0f b6 c0                      ; movzx    eax, al
      0f b6 55 fc                   ; movzx    edx, BYTE PTR [ebp-4]
      0f b6 d2                      ; movzx    edx, dl
      3b c2                         ; cmp      eax, edx
    }
    je           ORG 0x08048862
    exec_info    1676374850
    bb_org_addr    0x08048aae
    blk {
      8b 45 08                      ; mov      eax, DWORD PTR [ebp+08h]
      25 ff 7f 00 00                ; and      eax, 0x7fffh
      0f b7 04 45 00 d5 0e 08       ; movzx    eax, WORD PTR [eax*2+080ed500h]
      0f b7 c0                      ; movzx    eax, ax
      89 45 08                      ; mov      DWORD PTR [ebp+08h], eax
      8b 55 ec                      ; mov      edx, DWORD PTR [ebp-20]
      3b c2                         ; cmp      eax, edx
    }
    jbe          ORG 0x08048ad8
    exec_info    1643523311
    bb_org_addr    0x08048acb
    blk {
      8b 45 dc                      ; mov      eax, DWORD PTR [ebp-36]
      48                            ; dec      eax
      89 45 dc                      ; mov      DWORD PTR [ebp-36], eax
    }
    jnz          ORG 0x0804883a
    exec_info    10629421
    jmp          ORG 0x08048ad8
  }

}
```

Figura 3.6: Representação de um *trace* em arquivo

questão; esse cabeçalho é utilizado no momento de realizar a carga do *trace*, pois o *Persistence Layer* precisa dessa informação para fazer **code patching**, isto é, para poder inserir o *trace* carregado no programa traduzido. Um campo chamado `checksum` é utilizado para verificar a consistência do *trace* ao carregar o mesmo. O campo `Predecessors` contém uma lista com os endereços dos *traces* que são predecessores do *trace* representado. O campo `Stats` possui diversos contadores internos que fornecem algumas informações sobre o *trace*: `ExecutionFrequency` - informa quantas vezes o *trace* foi executado; `NumberOfBlocks` - informa quantos blocos básicos o *trace* possui; e `HeadBlockSize` e `RealHeadBlockSz` que são usados para se saber quanto espaço está disponível no início do *trace* para se realizar alguma instrumentação de código.

A seção `Code` é composta por `blk`'s (Blocos Básicos) e instruções de salto. Cada `blk` é formado por uma seqüência de instruções, onde são mostrados o opcode e mnemônico de cada uma. Cada subseção `blk` ainda contém dois atributos; o primeiro atributo é `bb_org_addr`, que contém o endereço original do bloco básico; e o segundo atributo é `exec_info`, que é usado para saber quantas vezes cada **Side Exit** (SE) foi tomada, isto é, quantas vezes uma instrução de salto no fim do bloco básico foi tomada, fazendo com que o fluxo de execução não atingisse o fim do *trace*. As instruções de salto no meio de um *trace* são chamadas de *side exits* porque elas quebram o fluxo de execução contínuo do *trace*, isto é, o *trace* não é executado até a última instrução.

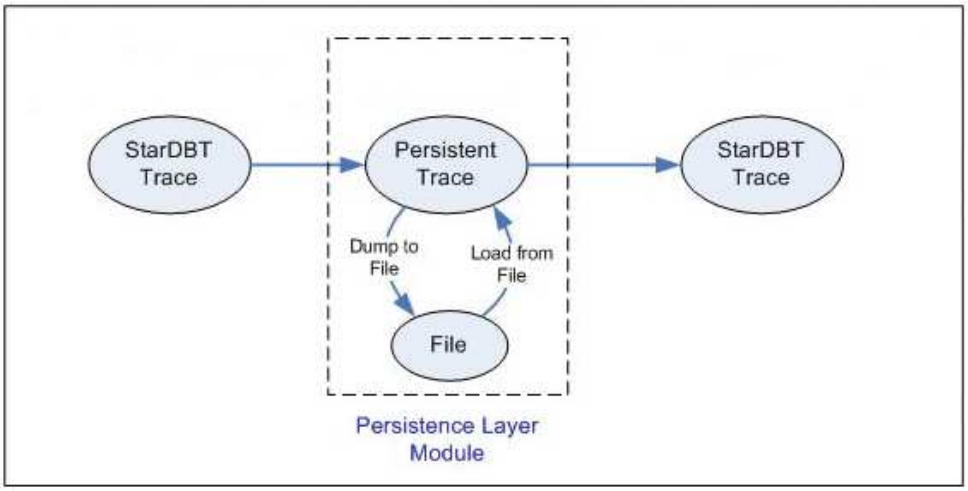


Figura 3.7: Módulo *Persistence Layer*

A Figura 3.7 mostra o esquema usado para gerar/carregar os *traces* para/de um arquivo texto. Um *trace*, como apresentado na Figura 3.6 é chamado de *Persistent trace* (também é chamado de *PTrace*). Tal mecanismo é muito útil para se fazer análises *offline* dos *traces*, pois permite que sejam analisados *traces* sem otimização, *traces* já otimizados,

bem como é possível realizar otimizações e/ou análises nos *traces* sem que os mesmos estejam executando no StarDBT.

Desta forma, ao se ter em mãos os trechos de código quentes para uma aplicação, pode-se fazer uma análise dos mesmos para saber quais otimizações podem resultar em ganhos, ou mesmo realizar modificações manualmente (diretamente no arquivo texto) de forma a se analisar o impacto no tempo de execução da aplicação.

O mecanismo de gerar/carregar *traces* permite que se analise o impacto que uma otimização pode causar no tempo de execução de uma aplicação, desconsiderando-se o tempo gasto para se realizar a otimização. Para isso cada aplicação deve ser executada duas vezes segundo o roteiro abaixo:

Primeira Execução:

1. Execução da aplicação
2. StarDBT encontra os *hot traces*
3. O *backend* otimiza os *traces*
4. O *Persistence Layer* realiza o *dump* dos *traces* otimizados para um arquivo

Segunda Execução:

1. O StarDBT carrega os *traces* otimizados de um arquivo de entrada, através do *Persistence Layer*
2. O *Persistence Layer* realiza o *link* dos *traces* carregados ao código traduzido
3. O código é então executado com os *traces* otimizados e não há a geração de nenhuma informação de *profile*, nem construção/otimização de *traces*

Tal esquema de geração e carga de *traces* foi criado [60] para permitir uma análise dos programas traduzidos, sem levar em conta o *overhead* de se construir e otimizar os *traces* durante a execução da aplicação, visto que na segunda execução do programa, o StarDBT não constrói e portanto não otimiza nenhum *trace*.

Uma análise detalhada do *overhead* do mecanismo de geração/carga dos *traces* pode ser obtida em [20].

3.3 Laços em Traces

Uma outra característica do módulo *backend* do StarDBT é que ele pode construir *loops* com os *traces* gerados. Devido às características da técnica MRET2, um *loop* de um programa é quebrado em diversos *traces*, podendo fazer com que sejam gerados muitos *traces* com uma baixa *completion rate* e uma baixa *execution coverage*. A reconstrução dos *loops* em forma de *trace* pode fazer com que sejam criados *traces* com alta *execution coverage* e *completion rate*.

Nos *traces* que foram criados a partir de um *loop* do programa, em geral, uma *side exit* irá desviar para um endereço que gerou um *trace*, como mostra a Figura 3.8. Nestes casos pode ocorrer que as *side exits* sejam muito tomadas.

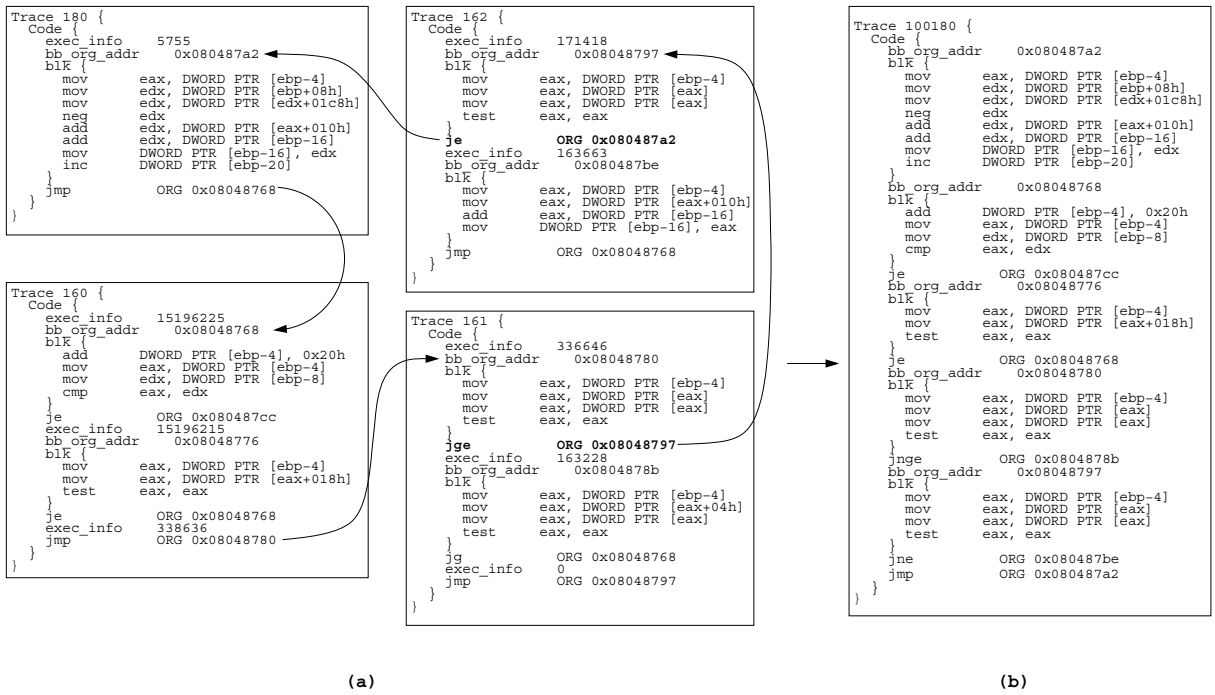


Figura 3.8: Um *loop* contruído a partir de *traces*, seguindo cada possível *side exit*

Considerando o `Trace 180` da Figura 3.8 como *loop header*, o *tail* do *trace* salta para o `Trace 160`, que depois desvia para o `Trace 161`, o qual possui uma *side exit* que salta para o `Trace 162`, que por sua vez possui uma *side exit* que desvia para o *loop header* representado pelo `Trace 180`. Observe que na Figura 3.8 (b), o *trace* construído, `Trace 100180`, teve todas as *side exits* que deram origem ao *loop* negadas. Por exemplo, a *side exit* `jge ORG 0x08048797` no `Trace 161` tornou-se `jnge ORG 0x0804878b` no `Trace 100180`, pois agora o fluxo de execução irá continuar dentro do *trace*.

Quando os *traces* são unidos em um ciclo como na Figura 3.8, não se sabe ao certo qual

trace deve ser o *loop header*, o `Trace 180` foi usado como exemplo, mas qualquer um dos outros *traces* poderia ser utilizado como *loop header*, pois cada um deles funciona como uma entrada do *loop*. Desta forma, o StarDBT replica o ciclo, criando 4 novos *traces*, usando cada *trace* como um *loop header* diferente: `Trace 160`, `Trace 161`, `Trace 162` e `Trace 180`. A replicação ocorre porque o ciclo não pode ser mantido, uma vez que por definição um *trace* não possui ciclos. A idéia de reconstruir o ciclo e depois quebrá-lo em *traces* é criar vários *traces* que possivelmente irão ter alta *execution coverage* e *completion rate*, uma vez que espera-se, as *side exits* não serão muito tomadas.

3.4 Modificações Implementadas no StarDBT

O StarDBT, como já dito, possui toda uma infra-estrutura de detecção e construção de *traces*, bem como o módulo *backend* que em conjunto com o módulo *Persistence Layer* permite gravar os *traces* gerados em arquivo e os carregar posteriormente, em uma nova execução do programa que os gerou, para fins de análise das otimizações efetuadas. Toda esta infra-estrutura, existente no StarDBT, bem como o conhecimento que o autor desta Tese adquiriu sobre o mesmo durante um período de estágio na Intel Corporation [2], fizeram com que o mesmo fosse a escolha óbvia para desenvolver seu trabalho de pesquisa de otimização em tradutores binários dinâmicos.

3.4.1 TraceOpt

Nesta seção será descrito um novo módulo do StarDBT, o **TraceOpt**, que foi desenvolvido pelo autor, como infra-estrutura deste trabalho.

Apesar do StarDBT possuir toda uma infra-estrutura para permitir a realização de otimizações, ele não possui nenhuma otimização de código implementada. Embora o formato *Persistent Trace* (ou *PTrace*) seja propício para o armazenamento de *traces* em arquivos texto; ele não é prático para a realização de otimizações. Para contornar essa questão, foi desenvolvido um módulo chamado *TraceOpt*, cuja ligação com o StarDBT pode ser vista na Figura 3.9.

Como se pode observar, o *TraceOpt* está ligado ao módulo *Persistence Layer*, pois é este último módulo que concentra os *traces* gerados no StarDBT. O *TraceOpt* converte os *traces* do formato *Persistent Trace*, para o formado *TraceOpt Trace*. O *TraceOpt* realiza então todas as otimizações que julgar necessárias e converte os *traces* novamente para o formato *Persistent Trace*.

O formato *Persistent Trace*, embora contenha diversas informações sobre um *trace*, não é muito adequado para a realização de otimizações, porque as instruções que formam o *trace* estão aglutinadas na seção `blk`, as quais são intercaladas por instruções de salto. In-

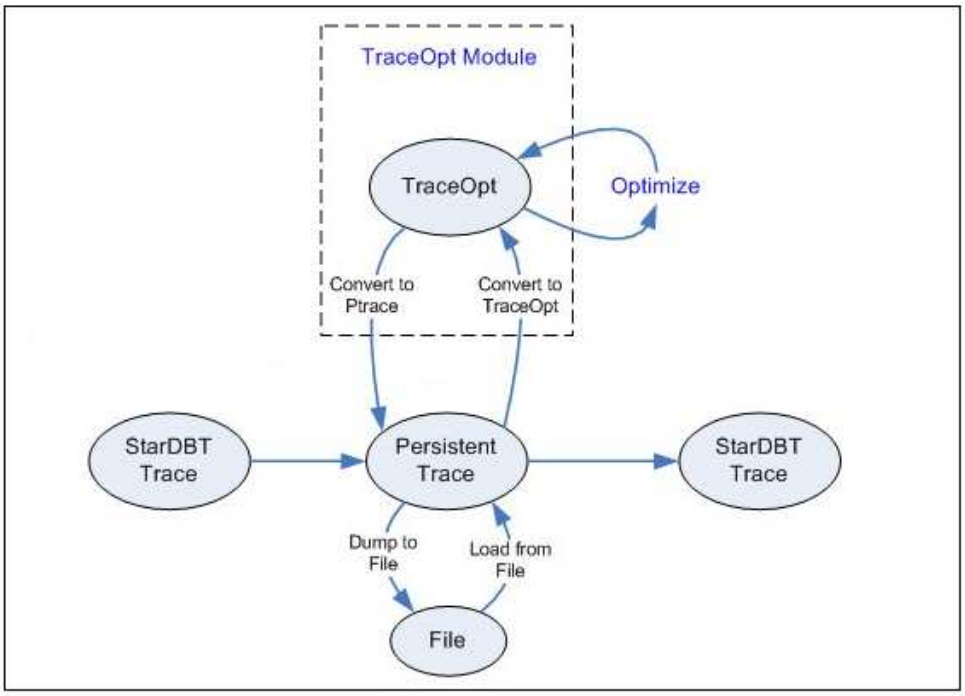


Figura 3.9: Módulo *TraceOpt*

ternamente as instruções que compõem um bloco básico (um seção `blk`), são representadas como uma simples seqüência de bytes, sem qualquer distinção ou separação entre elas. Tal formato torna complicado saber onde uma instrução termina e outra começa. Desta forma, a conversão de *Persistent Trace* para *TraceOpt Trace*, decodifica cada seqüência de bytes que forma um seção `blk`, recuperando quais instruções estão ali armazenadas, e as representa de forma individual.

A Figura 3.10 fornece uma visão do processo de conversão do formato utilizado pelo *Ptrace* para o formato utilizado por *TraceOpt*. Cada seção `blk` é quebrada nas diversas instruções que compõe a mesma, sendo que o *TraceOpt* constrói uma lista ligada com todas as instruções que formam o *trace*.

Depois de construir a lista ligada com todas as instruções do *trace*, o *TraceOpt* percorre as instruções geradas e computa quais registradores são definidos e usados por cada uma delas, o mesmo acontecendo com as **flags** que são definidas na arquitetura x86. Isso permite que o *TraceOpt* realize Análises de Fluxo de Dados [7,58] nas instruções **assembly** de um programa binário x86. Análises de fluxo de dados são uma ferramenta básica para muitas otimizações de código, sendo quase impossível otimizar um código sem recorrer a esse tipo de informação. O formato *Persistent Trace* torna impossível realizar qualquer tipo de análise de fluxo de dados, devido a forma como ele encapsula as instruções do *trace*. Com isso, o módulo *TraceOpt* torna simples analisar, modificar e otimizar um *trace*

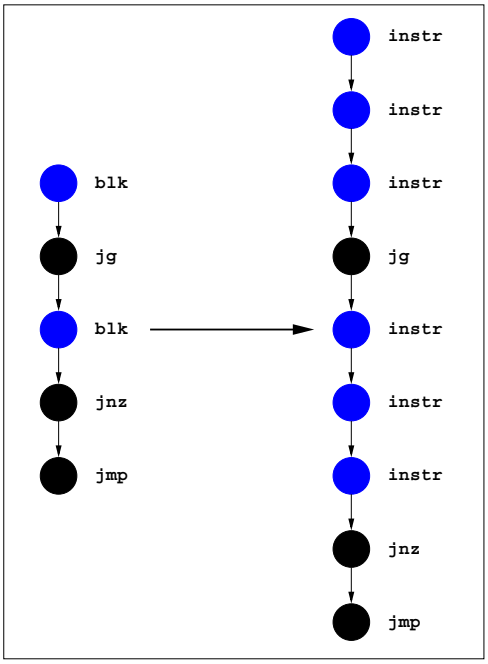


Figura 3.10: Conversão de *PTrace* para *TraceOpt*

qualquer.

Depois que as otimizações são finalizadas, o *TraceOpt* recria a estrutura de *Persistent Trace*, re-empacotando novamente as instruções, com exceção das instruções de controle, em seqüências de bytes e também as estrutura que formam um *Persistent Trace*, tornando assim o *trace* otimizado apto a ser gravado de volta em arquivo.

Capítulo 4

Cold Code Analysis

Otimizações de código realizam análises de fluxo de dados a fim de obter informações sobre o programa a ser otimizado. Tais informações são utilizadas para que a otimização não realize nenhuma transformação de código que introduza erros no programa.

Em DBT, as análises de fluxo de dados são realizadas somente dentro dos *traces* a serem otimizados, limitando assim a quantidade de código analisada para se extrair informações. Considere por exemplo o *trace* apresentado na Figura 4.1. Nesta figura os blocos básicos tracejados não fazem parte do *trace*. No bloco básico $B1$ do *trace* existe um uso do registrador `R` e no bloco básico $B3$ existe uma definição do mesmo.

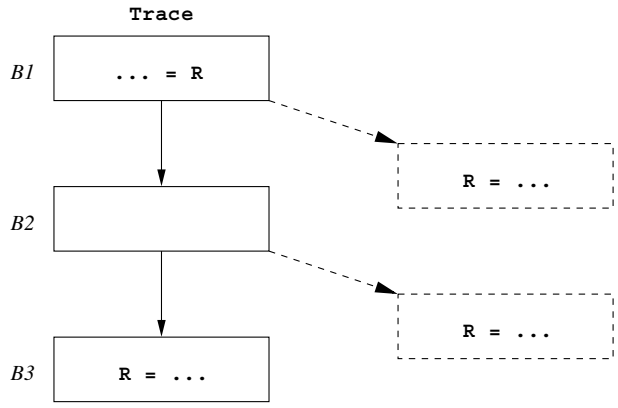


Figura 4.1: Análise de blocos básicos fora do *trace*

Se a análise de fluxo de dados *Liveness Analysis* for realizada no *trace*, não se pode afirmar com certeza se no bloco básico $B2$ o registrador `R` está vivo ou não, isto porque a análise se limitou ao blocos do *trace*, e como se observa na Figura 4.1 existem definições de `R` nos blocos fora do *trace*, significando que em $B2$ o registrador `R` não está vivo. Tal pergunta só pôde ser respondida ao se analisar os blocos básicos que não pertencem ao *trace*. Não saber com certeza se `R` está vivo ou não dentro do *trace* pode fazer com que

as otimizações se comportem de forma menos agressiva, perdendo assim oportunidades de gerar ganhos de desempenho no código, por terem de agir de forma conservativa, uma vez que em DBT as otimizações se limitam aos *traces*.

4.1 Side Exits e Traces

A Figura 4.2 contém um *trace* extraído do programa 164.gzip do benchmark SPEC. Como este *trace* é um *hot trace*, espera-se que o seu fluxo de execução entre na primeira instrução `shl esi,cl` e que deixe o *trace* somente na última instrução `jmp ...` grande parte do tempo. O fluxo de execução também pode sair do *trace* em alguma das 3 instruções de salto que estão no meio do *trace*, mas espera-se que tal comportamento ocorra poucas vezes. Essas instruções de salto são as *side exits* do *trace* porque quebram o fluxo de execução contínua do mesmo.

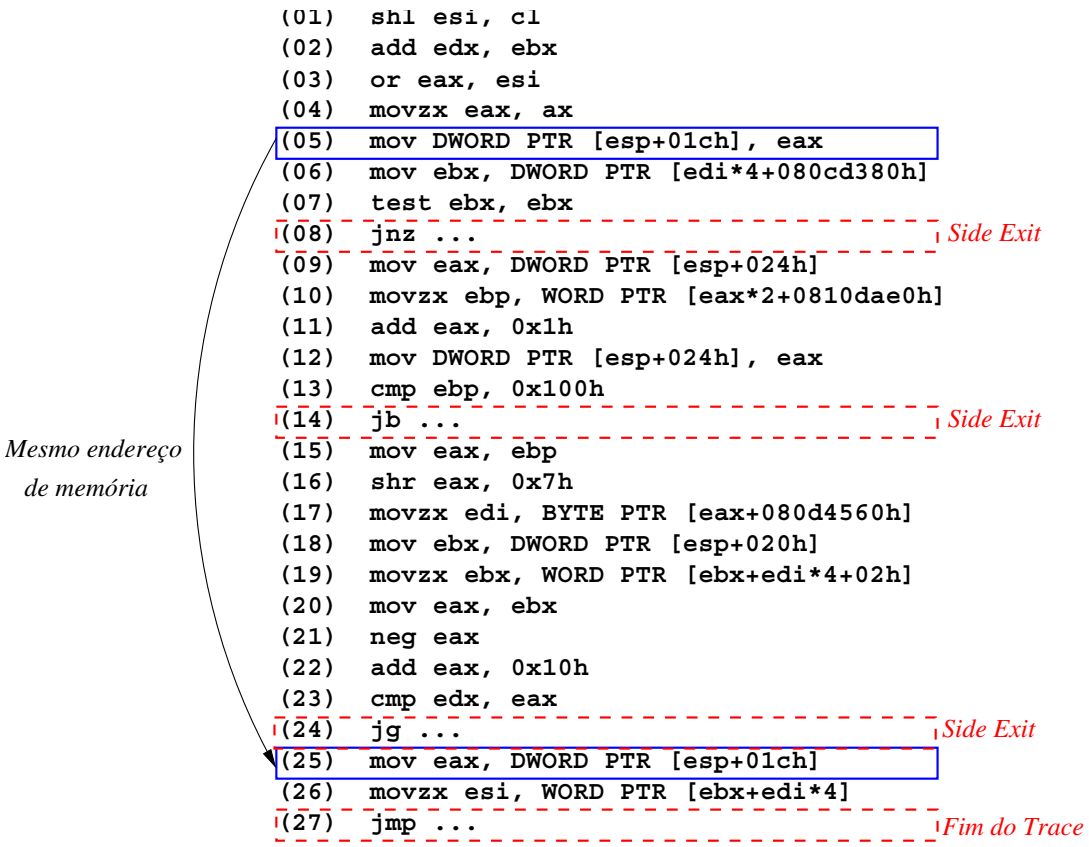


Figura 4.2: *Trace* do programa 164.gzip do benchmark SPEC

Uma análise simples do código da Figura 4.2 revela que a 5ª instrução do *trace* armazena o valor de `EAX` na posição de memória apontada por `DWORD PTR [esp+01ch]` e a

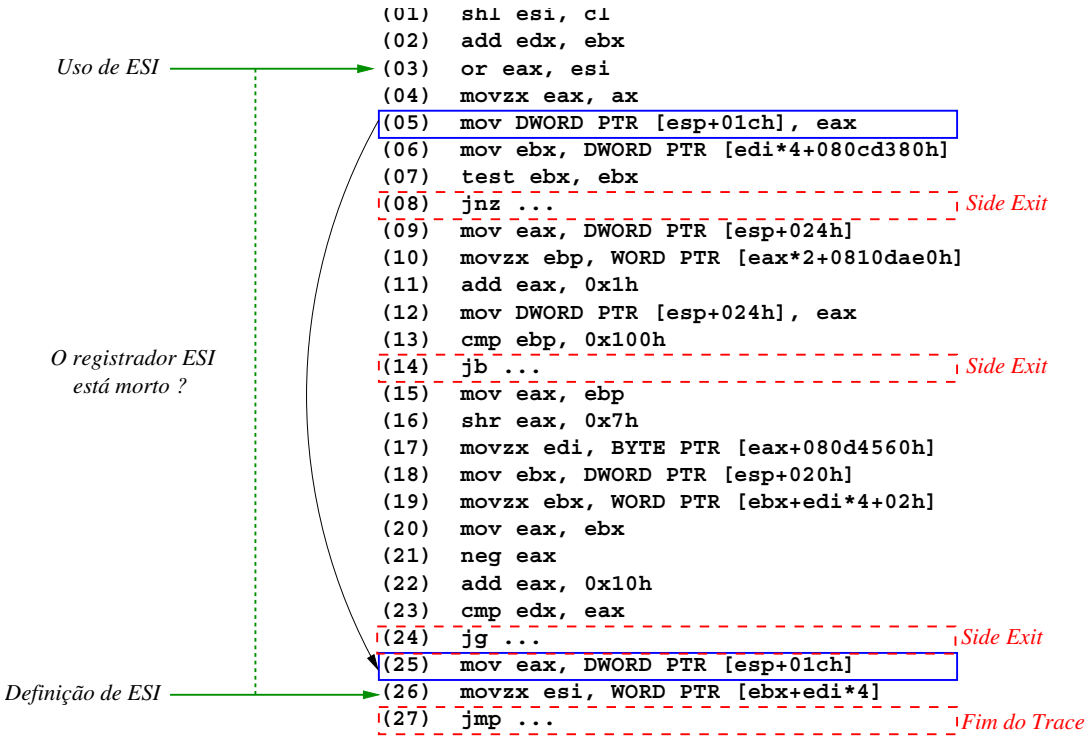


Figura 4.3: *Trace* do programa 164.gzip do benchmark SPEC

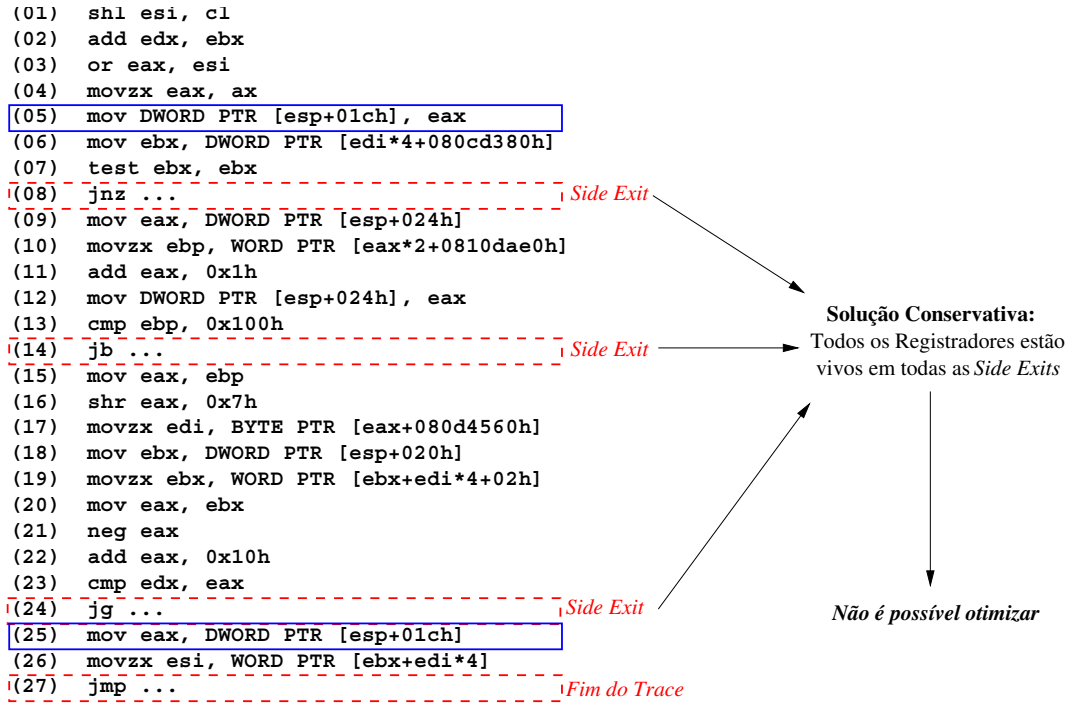


Figura 4.4: *Trace* do programa 164.gzip do benchmark SPEC

25ª instrução lê esta mesma posição de memória. Se for possível encontrar um registrador vazio suficientemente grande que possa conter estas duas instruções, então pode-se copiar o valor que a 5ª instrução armazena na memória para o registrador e modificar a 25ª de forma a não mais acessar a memória, mas sim o registrador.

De fato a Figura 4.3 indica que na 3ª instrução do *trace* existe um uso do registrador `ESI` o qual é definido na 26ª instrução. No intervalo que vai da 4ª até a 25ª instrução do *trace* não existe nenhuma outra definição do registrador `ESI`. Inicialmente conclui-se que `ESI` está vazio entre 4ª e a 25ª instrução, mas isso não pode ser afirmado com certeza.

Existem 3 *side exists* no intervalo sob análise. O valor armazenado em `ESI`, que é usado na 3ª instrução, pode estar vivo fora do *trace* em qualquer uma das *side exists*. Se isso for verdade, então a otimização proposta não é possível. A aplicação da análise de fluxo de dados *Liveness Analysis* no *trace* não é suficiente para poder responder se `ESI` está vivo ou não. Por razões conservativas, deve-se assumir que todos os registradores estão vivos em uma *side exit* e também ao fim do *trace*. Tal suposição torna impossível realizar a otimização proposta, embora seja o correto a se fazer, pois um compilador ou otimização deve sempre ser conservativo para não realizar uma transformação indevida no programa [10].

A otimização do acesso à memória só pode utilizar registradores que estão comprovadamente vazios. Se existir dúvida quanto ao estado de um registrador, como na Figura 4.4, então nenhuma transformação no código deve ser realizada. Neste tipo de situação, a otimização do acesso à memória não pode se aplicada, e nenhuma transformação pode ser realizada no *trace*, isto a princípio, limita a aplicação de otimização proposta (como talvez outras otimizações) a regiões do *trace* onde não existe nenhuma *side exit*, isto é, a princípio só poderiam ser realizadas otimizações dinâmicas ao nível de bloco básico, pois uma *side exit* é justamente uma instrução de salto, que além de quebrar o fluxo de execução do *trace*, marca o fim de um dado bloco básico e o início de outro.

4.2 Cold Code Analysis

Situações como a apresentada na Figura 4.2 limitam a agressividade e a região de atuação das otimizações, permitindo somente atuação local, dentro de blocos básico. Se para toda as otimizações sempre forem adotadas medidas conservativas, muitas oportunidades de otimização podem ser perdidas.

O registrador `ESI` não pode ser utilizado para remover o acesso à memória na 25ª instrução, com suposições conservativas sob o tempo de vida dos registradores. A análise de algumas instruções fora do *trace*, em busca de mais informações, poderá fazer com que se descubra que o registrador `ESI` está morto em todas as *side exists*, permitindo assim a realização da otimização.

O código assembly que fica fora do *trace* é chamado de código frio[1] porque, em geral, ele não é executado tantas vezes quanto um *trace*, desta forma não é vantajoso gastar tempo otimizando tal código. Por outro lado, este código contém informações que podem ser úteis aos *hot traces*, como o uso de registradores e *flags* do processador.

Cold Code Analysis consiste em se realizar análise de fluxo dados em trechos do código fora do *trace* e obter as informações desejadas de forma a aumentar a precisão e quantidade de informação da análise de fluxo de dados realizada no *hot trace*. Como DBTs não geram um grafo de fluxo de controle completo, algumas heurísticas devem ser empregadas para que se possa **caminhar no código frio**.

Mergulhar[2] no código frio envolve diversas questões: Quão fundo deve-se mergulhar no código frio? Quanto tempo gastar analisando código frio? Não existe nenhuma garantia de que o tempo extra que é gasto em tal análise irá aumentar o nível de informação de uma dada análise de fluxo de dados ou mesmo que esse aumento do nível de informação irá se traduzir em mais transformações realizadas pelas otimizações.

4.3 Mergulhando no Código Frio

Um DBT traduz somente os trechos de código da aplicação que são executados, desta forma, os blocos básicos traduzidos e executados não formam um grafo de fluxo de controle completo. A Figura 4.5 apresenta uma visão do que é construído durante o processo de tradução de um programa.

Depois que o código traduzido é instrumentado, um DBT pode identificar os *hot traces* como mostra a Figura 4.6 e em tal conjunto de blocos realizar diversas otimizações. Para a realização de *cold code analysis*, são considerados apenas o blocos básicos não pertencentes ao *trace* que podem ser alcançados através das *side exists*, bem como os seus sucessores, como mostra a Figura 4.7.

A Figura 4.8 mostra que podem existir diversos blocos básicos no código frio que são alcançáveis através de uma *side exit*. Analisar todos estes blocos básicos pode ser extremamente custoso em termos de tempo e recursos computacionais, sendo assim proibitivo em um ambiente de tradução dinâmica.

Para não incorrer em um custo computacional alto, a técnica de *cold code analysis* mergulha no código frio seguindo um sistema de níveis de profundidade de mergulho. Neste sistema, realizar análise de fluxo de dados somente nos blocos básicos do *trace* corresponde ao **nível 0 de mergulho**, onde nenhum código frio é analisado e desta forma assume-se que todos os registradores estão vivos em cada *side exit*.

[1] *Do inglês, Cold Code*

[2] O termo mergulhar será empregado com o mesmo sentido de caminhar

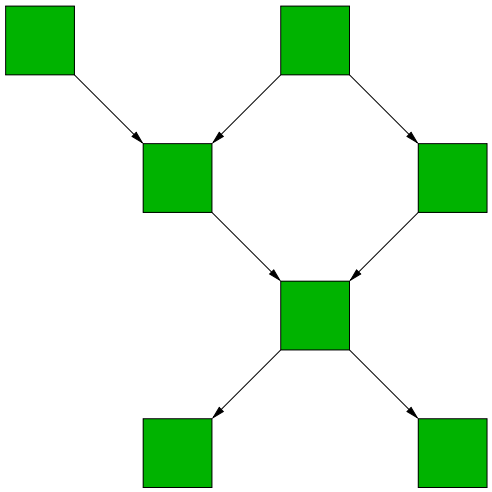


Figura 4.5: Porção de um *CFG* construída por um DBT

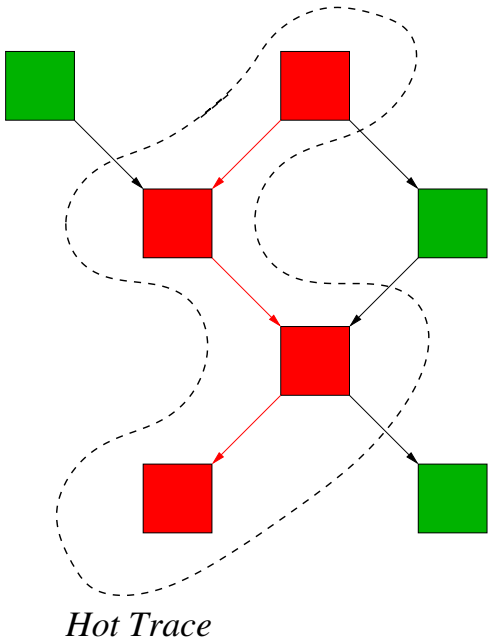


Figura 4.6: *Hot Trace* identificado pelo DBT

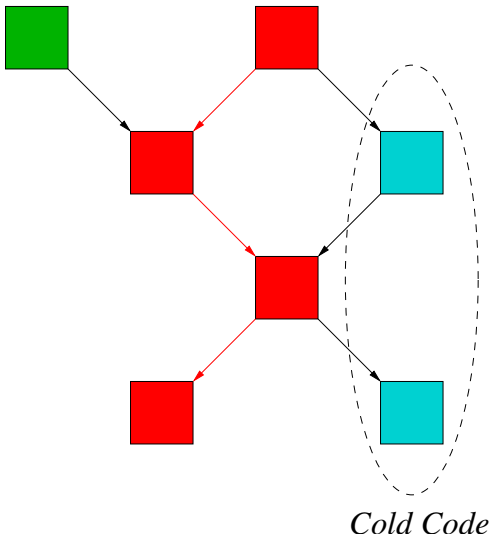


Figura 4.7: Código Frio

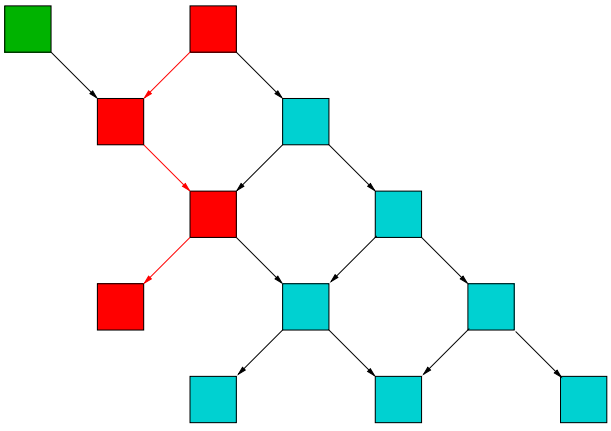


Figura 4.8: Vários níveis de código frio

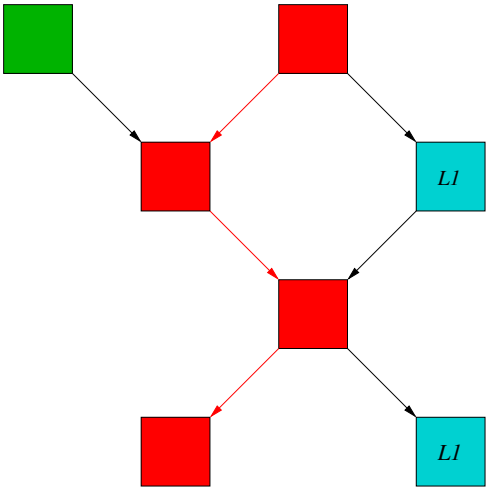


Figura 4.9: Analisando 1 nível

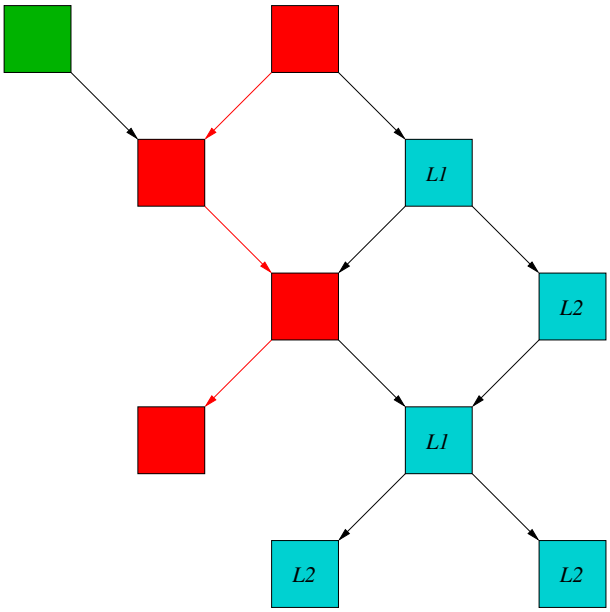


Figura 4.10: Analisando 2 níveis

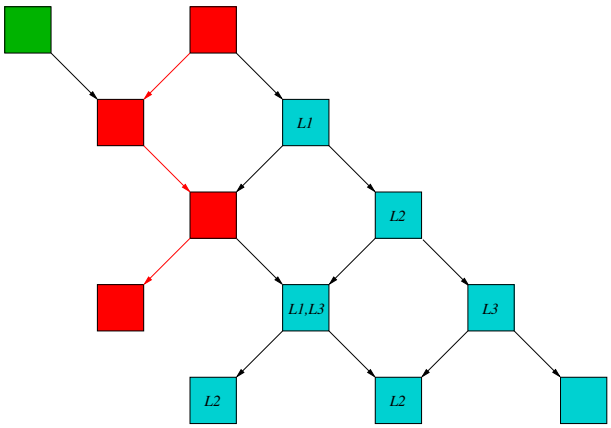


Figura 4.11: Analisando 3 níveis

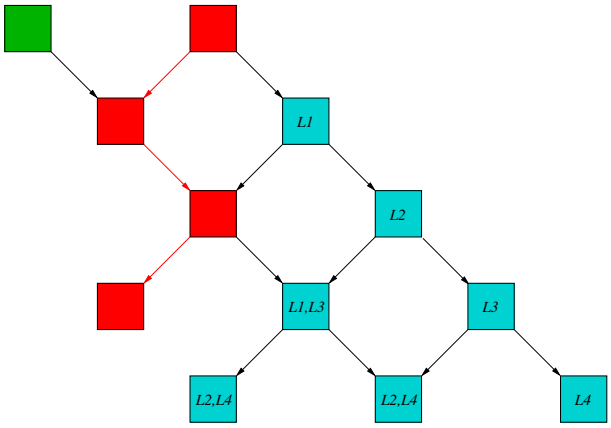


Figura 4.12: Analisando 4 níveis

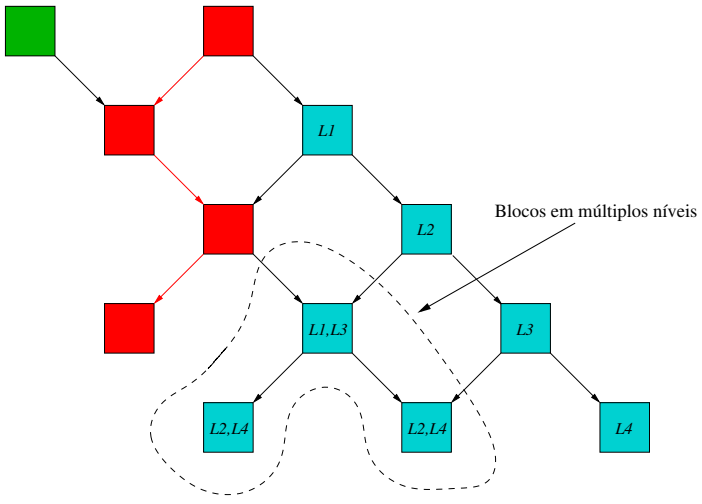


Figura 4.13: Bloco Básicos pertencentes a múltiplos níveis

A Figura 4.9 mostra o mergulho de 1 nível no código frio. Neste nível somente um bloco básico em cada *side exit* é analisado. O fim do bloco básico no código frio é delimitado por uma instrução de salto, uma chamada de função, um retorno de função ou um desvio para um endereço de memória que ainda não foi traduzido. Quando uma destas instruções é encontrada, assume-se que todos os registradores, para os quais não foi possível determinar o estado de vivo/morto, estão vivos.

Se for decidido por mergulhar mais um nível (**nível 2**), estão deve-se analisar os blocos básicos que estão no nível 1, bem como os blocos básicos que são seus sucessores, quando possível, como mostrado na Figura 4.10. Ali vê-se que cada bloco básico do nível 1 possui seus sucessores adicionados à análise. Para mergulhar até o nível 3, deve-se analisar os blocos básicos que estão no nível 1, no nível 2 e adiciona-se à análise os sucessores dos blocos básicos que estão no nível 2, como mostra a Figura 4.11.

Tal processo pode ser repetido indefinidamente, para se mergulhar tão fundo quanto desejado ou necessário. Em geral, níveis muito profundos não serão atingidos ou possuirão poucos blocos básicos devido as restrições já citadas, impossibilitando assim determinar o sucessor de um dado bloco básico.

Embora níveis profundos possam ter poucos blocos básicos, eles são mais caros de se analisar, pois quanto mais profundo se mergulhar, maior será a quantidade de blocos básicos a serem analisados.

Uma situação que pode ocorrer durante a análise de código frio, é o fato de um determinado bloco básico pertencer a mais de um nível, como mostra a Figura 4.13. Existem blocos básicos que pertencem simultaneamente aos níveis 1 e 3, bem como aos níveis 2 e 4. Tal fato é uma fonte de *overhead* para *cold code analysis*, pois alguns blocos acabam sendo analisados mais de uma vez, o que é desnecessário, pois eles sempre geram a mesma informação. Um mecanismo que controle quais blocos foram analisados pode evitar a reanálise de tais blocos, diminuindo o tempo total de análise do código frio.

4.4 Resultados Experimentais

A técnica de *Cold Code Analysis* foi implementada no StarDBT e aplicada em de forma a tentar descobrir o maior número possível de *available registers*, de forma a verificar se a otimização de acessos à memória pode ser aplicada no *trace* da Figura 4.2.

O método foi avaliado na versão do StarDBT que executa sobre Linux traduzindo de IA32 para IA32. Tal escolha é uma forma de verificar a eficiência da análise, uma vez que não haverão os registradores extras da arquitetura IA64, tornando assim o ambiente mais restrito e a pressão por registradores ainda maior.

Todos os programas que compõem o benchmark SPEC CPU 2000 foram utilizados para se colher resultados. Para cada um dos programas o StarDBT identificou os *hot traces*

e depois cada uma das *side exits* dos mesmos foram analisadas em busca de *available registers*.

Uma particularidade da arquitetura x86 é que no conjunto de registradores existem algumas relações de *alias*, onde o uso ou definição de um registrador implica necessariamente no uso ou definição de um outro. Por exemplo, uma definição do registrador `EAX` também irá definir os registradores `AX`, `AH` e `AL`. De forma semelhante, um uso do registrador `BH` implica no uso dos registradores `BX` e `EBX`. Para gerenciar tais questões, os registradores são tratados como diferentes recursos em *cold code analysis*, desta forma `EAX`, `AX` bem como os demais registradores e seus *alias* são tratados de forma distinta, como se fossem recursos distintos. Obviamente a relação de implicação entre usos/definições dos *alias* é respeitada. Os registradores utilizados nos experimentos são apresentados na Tabela 4.1.

Registradores x86 Analisados
EAX AX AH AL
EBX BX BH BL
ECX CX CH CL
EDX DX DH DL
EBP BP
ESI SI
EDI DI
ESP SP
MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7
XMM0 XMM1 XMM2 XMM3 XMM4 XMM5
XMM6 XMM7 XMM8 XMM9 XMM10 XMM11
XMM12 XMM13 XMM14 XMM15

Tabela 4.1: Registradores da arquitetura x86 utilizados para análise

Os experimentos foram realizados utilizando-se um máximo de 8 níveis de profundidade. Para uma primeira execução do SPEC pelo StarDBT, somente um nível foi analisado, em uma segundo execução dois níveis foram analisados e assim sucessivamente até o total de 8 níveis. Como mostrado na Tabela 4.1, em cada *side exit* a análise irá tentar determinar o estado de disponível para 48 registradores. O conjunto de *traces* gerado pelo StarDBT possui um total de 240.612 *side exits*. Desta forma, se todos os registradores estiverem disponíveis em todas as *side exits*, tem-se um total de 11.549.376 registradores para o qual se tenta determinar se os mesmos estão mortos.

Na implementação de *cold code analysis* realizada no StarDBT, um bloco básico pode ser analisado mais de uma vez se ocorrerem situações como a da Figura 4.13, onde um bloco pertence a múltiplos níveis, pois nenhum mecanismo de controle dos blocos já

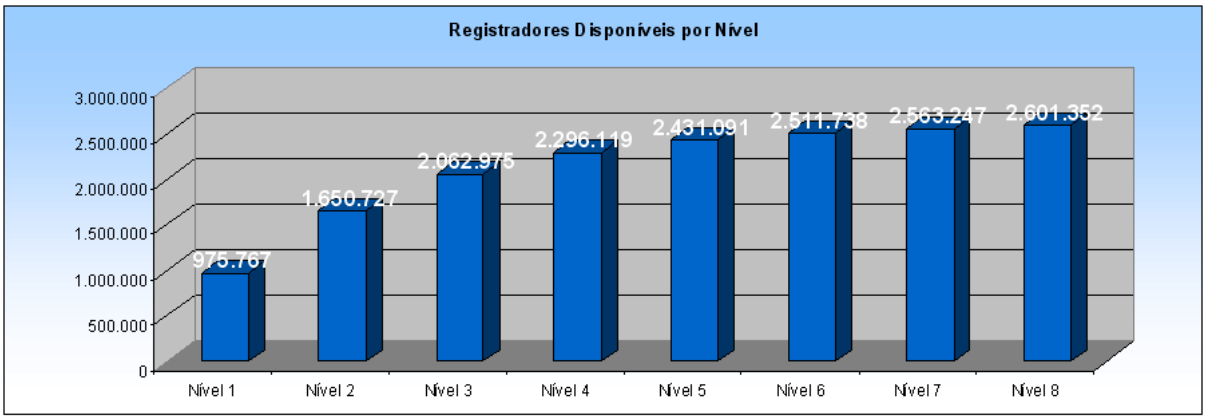


Figura 4.14: Registradores disponíveis em cada nível

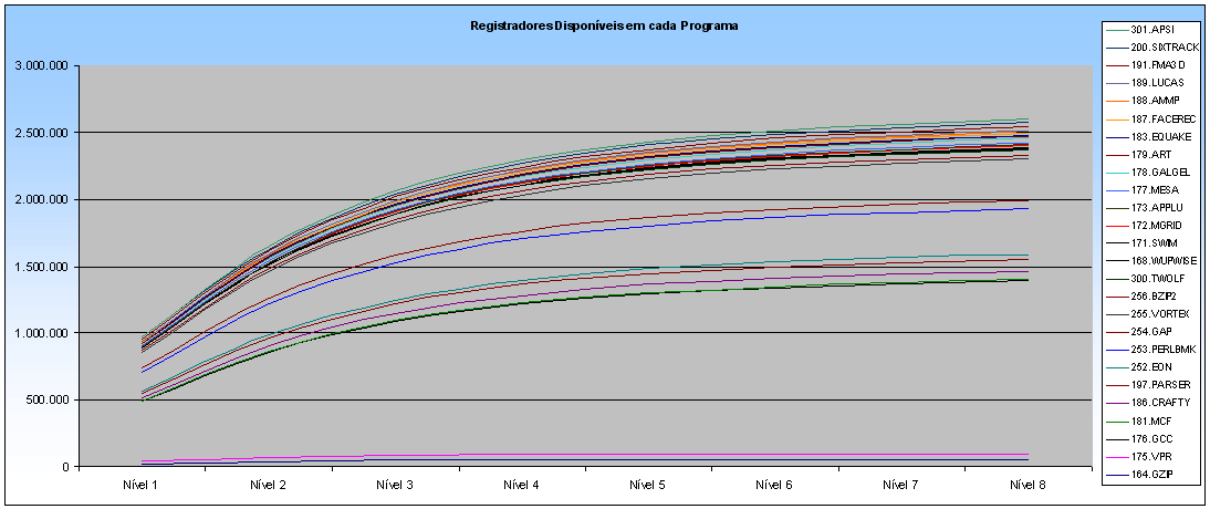


Figura 4.15: Registradores disponíveis em cada nível de cada programa

visitados foi implementado.

A Tabela 4.2 contém o total de registradores que são descobertos como disponíveis a cada nível de mergulho no código frio para cada programa. De forma semelhante o Figura 4.15 mostra de forma progressiva o total de registradores disponíveis, cada vez que a análise mergulha um nível. Os resultados são apresentados de forma sumarizada no gráfico que se encontra na Figura 4.14, onde o total de registradores disponíveis em cada de nível de cada programa é somado de forma a se obter uma visão geral. Uma análise de tais resultados permite concluir que mergulhar 8 níveis permite descobrir que aproximadamente 22,52% dos registradores analisados estão disponíveis, isto é mortos, e portanto podem ser utilizados, isto é quase 25% do total de 11.549.376 registradores inicialmente analisados. Para uma arquitetura com poucos registradores, tal resultado é

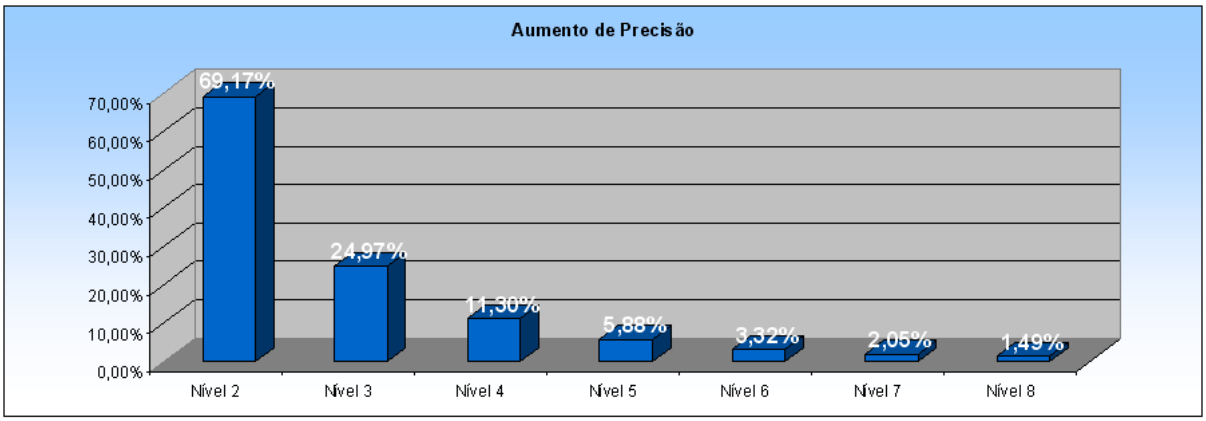


Figura 4.16: Aumento da Precisão da informação sobre registradores disponíveis

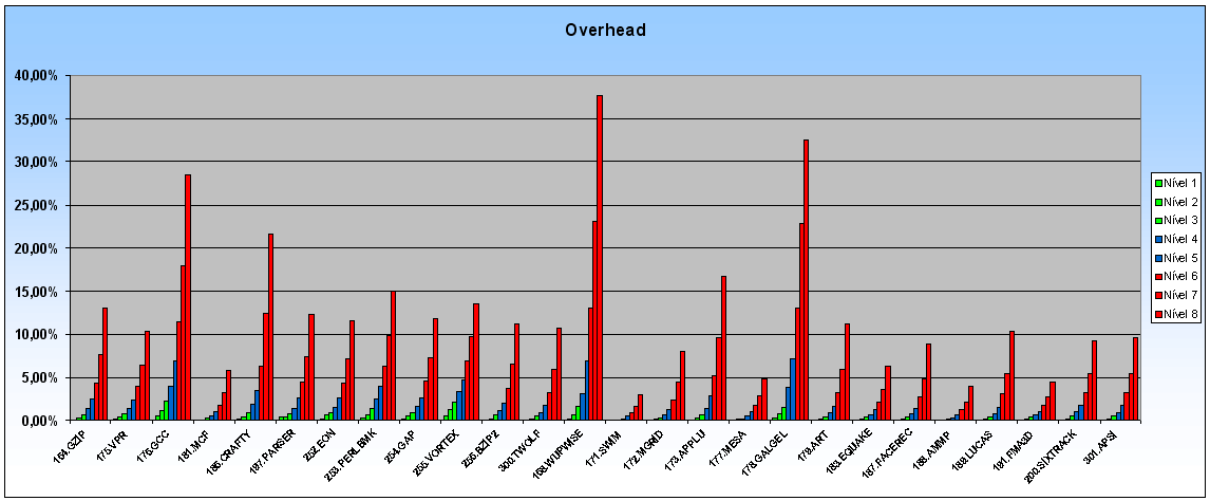


Figura 4.17: *Overhead* de se realizar *Cold Code Analysis*

um recurso interessante a ser explorado por otimizações dinâmicas, uma vez que a técnica de *cold code analysis* pode ser aplicada a outras análises de fluxo de dados. Se a análise de fluxo de dados sob análise fosse *liveness analysis*, provavelmente seria possível determinar que a maioria dos registradores estão vivos, pois este tende a ser um resultado complementar de *available registers*, só não sendo possível resolver situações onde o registrador se encontre em um trecho de código não alcançável por *cold code analysis*.

Por ultrapassar os limites de um *trace*, a técnica de *cold code analysis* pode determinar o estado de vivo/morto para quase todos os registradores em cada *side exit*, permitindo assim que as otimizações realizem escolhas melhores do que deve ser otimizado ou não. Vale observar que um registrador só é considerado como disponível (morto) quando for possível afirmar com certeza que existe uma definição do mesmo na *side exit* sendo anal-

Programa	Nível 1	Nível 2	Nível 3	Nível 4	Nível 5	Nível 6	Nível 7	Nível 8
164.GZIP	22400	37814	45750	49860	51865	53202	54214	55072
175.VPR	17284	29500	35962	39765	41754	42724	43348	43832
176.GCC	449857	791290	1006804	1130114	1200160	1241944	1269702	1290139
181.MCF	3292	5673	7128	7965	8376	8650	8836	8959
186.CRAFTY	25413	41025	50369	55525	59101	61326	63363	64743
197.PARSER	32207	56242	70443	78724	83000	85867	87288	88147
252.EON	14382	24644	29789	33089	35661	37175	37979	38487
253.PERLBMK	145071	231982	281460	309090	324502	331243	336208	339382
254.GAP	25102	42528	51865	57286	59576	60539	61079	61378
255.VORTEX	124361	198125	247507	273877	292178	304234	308621	312736
256.BZIP2	12490	21009	25159	27245	27987	28571	28891	29291
300.TWOLF	15838	27352	34184	37427	38994	39983	40636	41189
168.WUPWISE	2807	4007	4953	5695	6209	6662	7024	7394
171.SWIM	2099	3664	4628	5323	5729	6021	6253	6432
172.MGRID	4836	7787	9193	10213	10937	11655	12064	12349
173.APPLU	2976	5054	6221	7049	7511	7811	8063	8193
177.MESA	5018	8555	10447	11366	11886	12249	12569	12760
178.GALGEL	12931	22259	28404	31536	33815	35522	37301	38700
179.ART	3483	5803	7453	8449	9003	9502	9826	10090
183.EQUAKE	4030	6673	8454	9196	9574	9802	10042	10194
187.FACEREC	5067	8693	10914	12066	12745	13238	13630	13926
188.AMMP	5683	9956	12903	15060	16290	17007	17434	17731
189.LUCAS	1238	2055	2197	2294	2340	2369	2389	2399
191.FMA3D	13388	20205	24415	26219	27242	27711	28159	28377
200.SIXTRACK	12836	19983	24172	27119	28848	30055	31083	31782
301.APSI	11678	18849	22201	24567	25808	26676	27245	27670

Tabela 4.2: Registradores disponíveis descobertos a cada nível

isada. Se um dado registrador não é usado em um programa, ele nunca será considerado como disponível, pois não será encontrada nenhuma instrução definindo o mesmo.

O gráfico que se encontra na Figura 4.16 apresenta uma evolução, em termos percentuais, da quantidade de registradores que são descobertos como disponíveis, cada vez que se mergulha mais um nível. Os valores são em relação ao nível anterior, desta forma ao se passar do nível 1 para o nível 2, a quantidade de novos registradores que foram descobertos como disponíveis (no nível 2), equivale a 69,17% do total de registradores descobertos somente ao se analisar até o nível 1. De forma análoga, ao se passar do nível 2 para o nível 3, a quantidade de novos registradores que foram descobertos como disponíveis (no nível 3), equivale a 24,97% do total de registradores descobertos somente ao se analisar

Programa	Nível 1	Nível 2	Nível 3	Nível 4	Nível 5	Nível 6	Nível 7	Nível 8
164.GZIP	0,11%	0,32%	0,75%	1,39%	2,48%	4,38%	7,63%	13,01%
175.VPR	0,17%	0,41%	0,86%	1,47%	2,34%	3,93%	6,42%	10,28%
176.GCC	0,49%	1,10%	2,31%	4,06%	6,91%	11,46%	17,96%	28,42%
181.MCF	0,09%	0,12%	0,27%	0,50%	1,00%	1,73%	3,23%	5,77%
186.CRAFTY	0,20%	0,41%	0,95%	1,88%	3,54%	6,33%	12,40%	21,62%
197.PARSER	0,40%	0,42%	0,83%	1,45%	2,57%	4,44%	7,45%	12,23%
252.EON	0,23%	0,58%	0,97%	1,59%	2,68%	4,30%	7,08%	11,51%
253.PERLBMK	0,34%	0,73%	1,43%	2,43%	3,94%	6,28%	9,76%	14,96%
254.GAP	0,17%	0,48%	0,93%	1,61%	2,66%	4,55%	7,27%	11,90%
255.VORTEX	0,51%	1,21%	2,17%	3,36%	4,69%	6,95%	9,70%	13,51%
256.BZIP2	0,14%	0,26%	0,61%	1,11%	2,02%	3,72%	6,58%	11,21%
300.TWOLF	0,11%	0,22%	0,47%	0,94%	1,78%	3,28%	5,97%	10,69%
168.WUPWISE	0,21%	0,76%	1,66%	3,05%	6,94%	13,00%	23,07%	37,59%
171.SWIM	0,01%	0,06%	0,14%	0,19%	0,47%	0,93%	1,67%	3,03%
172.MGRID	0,05%	0,16%	0,34%	0,69%	1,25%	2,40%	4,45%	8,05%
173.APPLU	0,13%	0,32%	0,69%	1,41%	2,84%	5,28%	9,57%	16,70%
177.MESA	0,05%	0,17%	0,24%	0,51%	1,07%	1,74%	2,91%	4,85%
178.GALGEL	0,29%	0,78%	1,58%	3,81%	7,15%	13,03%	22,76%	32,45%
179.ART	0,08%	0,16%	0,45%	0,92%	1,69%	3,19%	6,00%	11,23%
183.EQUAKE	0,07%	0,16%	0,44%	0,71%	1,26%	2,15%	3,65%	6,26%
187.FACEREC	0,04%	0,21%	0,42%	0,83%	1,48%	2,74%	4,86%	8,88%
188.AMMP	0,05%	0,10%	0,20%	0,36%	0,65%	1,22%	2,17%	3,88%
189.LUCAS	0,06%	0,19%	0,37%	0,83%	1,50%	3,09%	5,47%	10,27%
191.FMA3D	0,08%	0,19%	0,39%	0,67%	1,07%	1,73%	2,78%	4,47%
200.SIXTRACK	0,10%	0,23%	0,55%	1,01%	1,76%	3,16%	5,42%	9,33%
301.APSI	0,10%	0,24%	0,50%	0,94%	1,80%	3,18%	5,51%	9,57%

Tabela 4.3: *Overhead* de se realizar *Cold Code Analysis*

até o nível 2. E assim sucessivamente até o nível 8. Observa-se que até atingir-se o nível 8, tal aumento na quantidade de novos registradores disponíveis vai diminuindo. Depois de se analisar 4 ou 5 níveis, a quantidade de novos registradores que são descobertos como disponíveis já não é significativa, podendo indicar que, para muitos propósitos de otimização, mergulhar 4 ou 5 níveis pode ser o suficiente para se determinar o estado de vivo/morto de uma grande maioria dos registradores.

O gráfico que se encontra na Figura 4.17 apresenta o *overhead* que a técnica *cold code analysis* causa no tempo execução do StarDBT e da aplicação traduzida, sendo a relação entre o tempo gasto pela análise e o tempo de execução do StarDBT, da aplicação traduzida e da técnica em si somados. Como se pode observar, até o nível 3 o *overhead*

é menor que **1%**, em 21 dos 26 programas analisados. Os valores detalhados sobre o *overhead* causado por *cold code analysis* são apresentados na Tabela 4.3, onde para cada programa é apresentado o *overhead* causado no StarDBT cada vez que se analisa um nível mais profundo. Como esperado, o *overhead* apresenta um comportamento exponencial, fazendo com que níveis muito profundos tenham altos custos de tempo. Isso provavelmente se deve ao fato de haver muitos blocos básicos sendo analisados e também reanalisados, como no caso de um *loop*.

Capítulo 5

Hole Allocation

5.1 Alocação de Registradores

Alocação de registradores é uma otimização muito importante em qualquer compilador moderno. Uma alocação mal feita pode fazer com que um programa altamente otimizado, tenha um baixo desempenho. Uma heurística de alocação de registradores inadequada pode acarretar em muitos acessos à memória e em um programa final maior.

Existem diversas técnicas para mapear registradores a variáveis de um programa, como coloração de grafos [26] e *Linear Scan* [63]. Em geral, todas as técnicas tentam, tanto quanto possível, evitar o uso da memória principal do computador, devido à diferença de velocidade do processador e da memória. Muitos acessos à memória, podem tornar um programa consideravelmente lento, desta forma, fazer um uso correto dos registradores do processador se constitui em uma forma de minimizar tal penalidade.

A quantidade de registradores disponíveis quase nunca é suficiente e desta forma, o algoritmo de alocação não conseguirá mapear cada variável em um registrador. Quando essa situação ocorre, o algoritmo deve reservar uma espaço de memória na pilha de execução do programa para armazenar as variáveis não mapeadas em registrador. Tal processo recebe o nome de geração de **spill code** [7, 10, 17, 26, 28, 40, 52, 58].

Em DBT, a alocação de registradores também se constitui em uma otimização importante, isto porque, na tradução do código binário, um DBT vai utilizar os registradores da arquitetura alvo, e se a arquitetura origem contiver mais registradores que a arquitetura alvo, então a tradução do código binário irá gerar *spill code* para os registradores que não conseguiram ser mapeados em um registrador na arquitetura alvo. Se a arquitetura alvo possuir mais registradores que a arquitetura origem, então uma nova alocação de registradores e análise dos acessos à memória no programa binário, pode resultar em melhora no desempenho do código traduzido, uma vez que pode ser possível evitar determinados acessos à memória.

Chaitin [26] foi o primeiro a visualizar que o problema de alocação de registradores poderia ser resolvido como um problema de coloração de grafos, dando uma solução poderosa para o problema de alocação de registradores, embora sua metodologia para se gerar *spill code* fosse bem simples, consistindo em inserir uma instrução de *store* depois de cada definição da variável e uma instrução de *load* antes de cada uso da mesma, quebrando assim a *live range* [7, 10, 17, 28, 40, 52, 58] da variável em diversas outras menores. Embora seja poderosa, a alocação baseada em coloração de grafos tem, no pior caso, um custo computacional quadrático no número de *live ranges*. George e Appel [40] criaram uma variação da alocação baseada em coloração de grafos, onde eles conseguem eliminar mais instruções de *move* do programa pois, conseguem realizar mais *coalescing* [58].

Poletto e Sarkar [63] desenvolveram um método de alocação de registradores cujo objetivo é ter complexidade linear, para poder ser utilizado em sistemas *Just-In-Time* (JIT) [6, 33] onde tempo é muito importante. Este método, chamado de *Linear Scan*, não é baseado em coloração de grafos, mas sim nas *live ranges* das variáveis. Neste método as *live ranges* são ordenadas em uma lista de acordo com a instrução do programa onde elas começam e terminam. A lista com as *live ranges* ordenadas é então percorrida para se realizar a alocação.

Como gerar *spill code* é parte de qualquer alocador de registradores, várias pesquisas foram realizadas com o intuito de minimizar o custo do *spill code* no programa. Bergner et. al. [17] criaram uma técnica chamada de *Interference Region Spilling*. Neste método de se gerar *spill code*, quando uma *live range* S é selecionada para *spill*, procura-se entre as *live ranges* que interferem com S, aquela que tenha uma região de interferência pequena, isto é, uma outra *live range* L cuja interseção com S seja bem pequena. Isto faz com que a porção de S que não interfere com L, possa usar o mesmo registrador de L. A porção de S que interfere com L é então quebrada em diversas *live ranges* menores da mesma maneira que o algoritmo de Chaitin gera *spill code*, isto é, inserir uma instrução *store* depois de cada definição e uma instrução *load* antes de cada uso. Koseki et. al. [52] aperfeiçoaram o método de Bergner, fazendo com que a porção de S que interfere com L seja associada, se possível, a um registrador, com a qual ela não interfira, sendo que *load's* e *store's* são inseridos para se realizar a troca de registrador. Cooper e Simpson [28] criaram um método para a geração de *spill code* chamado de *Live Range Splitting*. Neste método, quando uma *live range* S é selecionada para *spill*, o algoritmo procura por uma *live range* L já alocada de forma que as duas *live ranges* possam compartilhar o mesmo registrador, ao custo de se inserir *load's* e *store's* e fazer com que uma *live range* já alocada seja quebrada. Antes de fazer com que as *live ranges* L e S compartilhem o registrador inicialmente atribuído a L, o algoritmo irá computar o custo deste compartilhamento e, caso ele seja maior que o custo de se fazer o *spill* de S, então o compartilhamento não é realizado e será gerado *spill code* para S sem que nenhuma alteração seja feita em L.

5.2 Hole Allocation

Hole Allocation [13] é uma técnica inicialmente desenvolvida, pelo autor, como um algoritmo de geração de *spill code*, que se baseia nas lacunas que existem nas *live ranges* das variáveis de um programa.

Se analisarmos um programa depois de realizada a alocação de registradores, nós iremos encontrar diversos **"buracos"** nas *live ranges* dos registradores da arquitetura alvo. Haverá lugares onde um registrador não está sendo usado e nenhum valor útil está armazenado no mesmo. Na Figura 5.1, tem-se um exemplo de tal situação. Se a instrução (2) não realizar qualquer referência direta ou indireta (implícita) ao registrador $R1$, então tal registrador não está vivo na instrução (2). Se nenhuma outra referência a $R1$ ocorrer até a instrução (6), então existirá um **dead hole**, que é o nome que a técnica de *hole allocation* usa para este trecho na *live range* do registrador $R1$. Observe que existe literalmente um buraco no tempo de vida do registrador $R1$, criando assim, duas *live ranges* associadas ao mesmo, no exemplo da Figura 5.1. O *dead hole* irá existir entre o fim e o início das duas *live ranges* associadas ao registrador, independentemente do número de instruções entre as mesmas, bastando apenas que as *live ranges* em questão sejam independentes.

```
(1) ... := R1
(2) instr_a
(3) instr_b
(4) instr_c
(5) instr_d
(6) instr_e
(7) R1 := ...
```

Figura 5.1: *Dead Hole*

Outro tipo de situação que pode ocorrer é apresentada na Figura 5.2, na qual observa-se que existe uma *live range* associada ao registrador $R1$. A instrução (1) realiza uma definição de $R1$ e a instrução (7) realiza um uso de $R1$. Desta forma o registrador $R1$ está vivo entre as instruções (1) e (7). Se não houver nenhuma referência direta/indireta ao registrador $R1$ entre as instruções (2) e (6) (inclusive), então pode-se observar que existe um buraco na *live range* de $R1$, nas instruções onde ele não é referenciado. Ao contrário do buraco da Figura 5.1, no exemplo da Figura 5.2, o registrador $R1$ contém um valor, mas o mesmo não está sendo usado por 5 instruções. Este buraco na *live range* de $R1$ é denominado de **live hole** pela técnica *hole allocation*. Isto porque o buraco contém um valor vivo que não está sendo usado naquele trecho da *live range*.

(1) R1 := ...
(2) instr_a
(3) instr_b
(4) instr_c
(5) instr_d
(6) instr_e
(7) ... := R1

Figura 5.2: *Live Hole*

Técnicas tradicionais de geração de *spill code* [26], inserem um **store** depois de cada definição e um **load** antes de cada uso da mesma. Desta forma, muitas instruções de *load/store* são inseridas no programa.

Existem diversas técnicas para a geração de *spill code* [11,17,18,28,46,51,52,54,59] que tentam minimizar a quantidade de *load's* e *store's* que são inseridos no programa, sendo que a técnica de Cooper e Simpson [28] usa um conceito semelhante ao de *live hole*, ao passo que as técnicas de Bergner et. al. [17] e Akira Koseki et. al. [52] usam um conceito semelhante ao de *dead hole*.

A técnica *hole allocation*, é diferente das demais, na medida em que utiliza *dead holes* e *live holes* de forma simultânea e sistematizada para gerar *spill code*, tentando assim tirar proveito destas duas formas de *buracos* que aparecem na *live range* de um registrador durante um programa. A geração de *spill code* via *hole allocation* usa programação dinâmica, e assim como as demais técnicas, gera *load's* e *store's*, mas também pode gerar instruções de **move**, que simplesmente realizam movimentação de dados, movendo o conteúdo de um registrador para outro. Pode-se imaginar que as instruções de *move* são geradas para mover o conteúdo de um determinado burado para outro buraco.

Por exemplo, considere a situação apresentada na Figura 5.3 onde duas *live ranges* possuem registrador e uma terceira, $l3$, foi selecionada para gerar *spill code*. Nesta Figura, cada bolinha preta representa uma referência ao registrador, representando uma leitura ou escrita. Os traços em $l3$ também indicam leitura ou escrita, mas da variável da *live range*, uma vez que esta ainda não recebeu nenhum registrador, sendo assim utilizados para diferenciar uma *live range* selecionada para *spill* das demais. A geração de *spill code* proposta por Chaitin [26], quebraria a *live range* $l3$ em várias *live ranges* menores, inserindo um *load* antes de cada uso e um *store* depois de cada definição. Se $l3$ possuir muitas instruções, então muito *spill code* será gerado.

A *live range* $l2$ possui um *dead hole*, mas ele cobre apenas uma porção de $l3$. Observe que após várias referências ao registrador $R1$, a *live range* $l1$ possui um *live hole*, onde

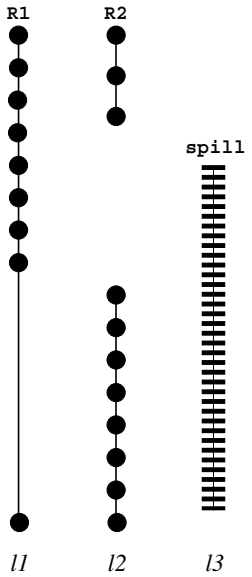


Figura 5.3: *Live Ranges* para serem alocadas

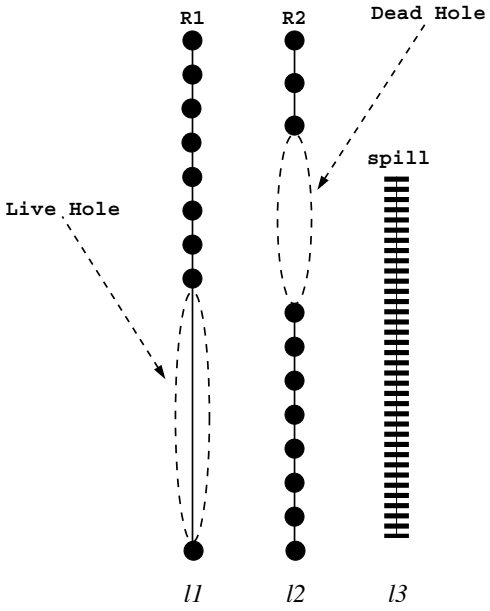


Figura 5.4: *Live Ranges* para serem alocadas

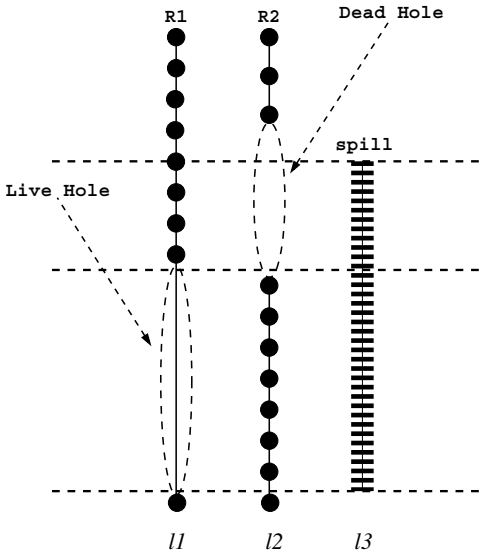


Figura 5.5: Associando $l3$ a *live/dead holes* de outras *live ranges*

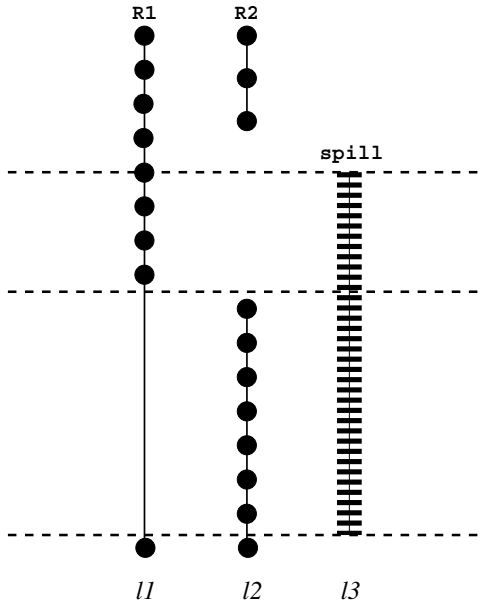


Figura 5.6: Associando $l3$ a *live/dead holes* de outras *live ranges*

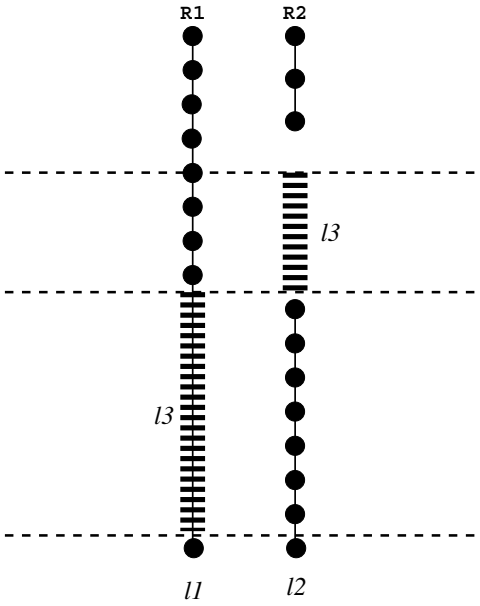


Figura 5.7: Quebrando a *live range*

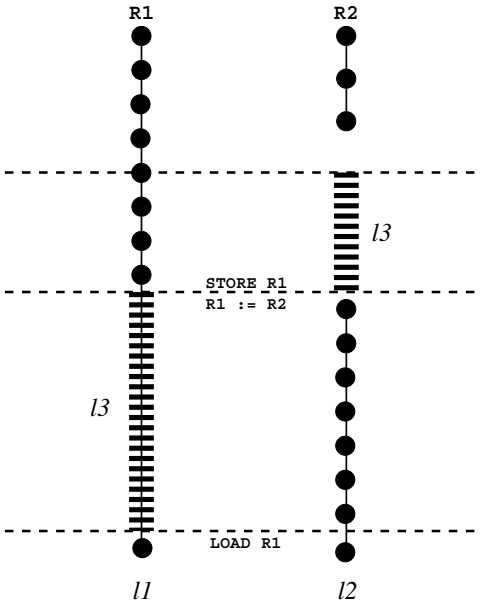


Figura 5.8: Inserindo instruções de ajuste

nenhum acesso ao registrador $R1$ é realizado. Desta forma pode-se usar simultaneamente os buracos de $l1$ e $l2$ para absorver a *live range* $l3$, como se pode observar na Figura 5.6. A primeira porção de $l3$ pode ser absorvida pelo *dead hole* de $l2$, e a segunda porção de $l3$ pode ser absorvida pelo *live hole* em $l1$.

A Figura 5.7 mostra como cada porção de $l3$ se encaixa em $l1$ e $l2$, depois de $l3$ ser quebrada. Em um primeiro momento a *live range* $l3$ irá usar o registrador $R2$, até ele ser usado novamente pela *live range* $l2$, sendo que a partir deste ponto, a *live range* $l3$ começará a utilizar o registrador $R1$.

Em *hole allocation*, as instruções inseridas para gerar o *spill code*, são chamadas de **instruções de ajuste**, isto porque elas irão "ajustar os encaixes" da *live range* selecionada para *spill* nas demais *live ranges*. Na Figura 5.8, podemos ver a configuração final da quebra de $l3$. Como o primeiro buraco usado é um *dead hole*, nenhuma **instrução de ajuste** é necessária para fazer com que $l3$ use o registrador $R2$. Quando $l3$ muda de buraco, trocando $R2$ por $R1$, então é necessário inserir uma instrução *move*, para realizar a troca de buraco. Pode-se observar que o buraco em $l1$ é um *live hole*, desta forma, antes de realizar a troca de $l3$ do registrador $R2$ para $R1$, é necessário armazenar o conteúdo de $R1$ (*live range* $l1$) na memória, para preservar o seu conteúdo. Quando a *live range* de $l3$ termina, o conteúdo de $R1$ é restaurado inserindo-se uma instrução *load*, fazendo com que ele contenha novamente o valor de $l1$.

Pode-se observar que não importa a quantidade de instruções que $l3$ contenha, ou quantas referências ela realiza ao registrador que lhe é associada; ao se quebrar $l3$ entre os buracos de $l1$ e $l2$, a quantidade de instruções de ajuste a ser inserida será sempre 3 instruções: um *store*, um *move* e um *load*. Em outras técnicas de geração de *spill code* a quantidade de instruções inseridas, em geral, depende da quantidade de usos/definições presentes na *live range* selecionada para *spill*, como por exemplo a técnica de Chaitin [26]. Outro fato a se observar, é que instruções *move* são mais baratas computacionalmente que instruções *load* e *store*, uma vez que *move*s não acessam a memória principal (assume-se que os operandos destino/fonte são sempre registradores).

5.2.1 Algoritmo Hole Allocation

O objetivo da otimização *hole allocation* é transformar acessos à memória em usos de registradores, utilizando os *dead holes* e *live holes* que existem no código assembly de um programa. Como visto na Figura 5.8, às vezes pode ser necessário inserir instruções de ajuste para fazer com que as *live ranges* utilizem os buracos de forma correta e também a não gerar erro algum no programa. Desta forma, a otimização deve ser capaz de escolher que buracos utilizar de forma a minimizar a quantidade de instruções de ajuste a serem inseridas.

A Figura 5.9 será utilizada para explicar como *hole allocation* funciona. Para simplificar o modelo, supõe-se que o programa é formado por apenas um bloco básico e que somente 3 registradores (`R1`, `R2` e `R3`) estão disponíveis para alocação.

Na Figura 5.9 existem 4 *live ranges*, sendo que 3 delas foram alocadas a registradores, e uma quarta *live range* que não possui registrador. Cada um dos 3 registradores possui *dead holes* em sua *live range*, onde na porção direita da figura, tem-se os *dead holes* em cada instrução. Os *dead holes* são representados pelas porções hachuradas da *live range* de cada registrador. A porção em preto das *live ranges* indicam que os registradores ou estão sendo referenciados (leitura/escrita) ou correspondem a um *live hole*, significando que não estão disponíveis para serem utilizados na otimização.

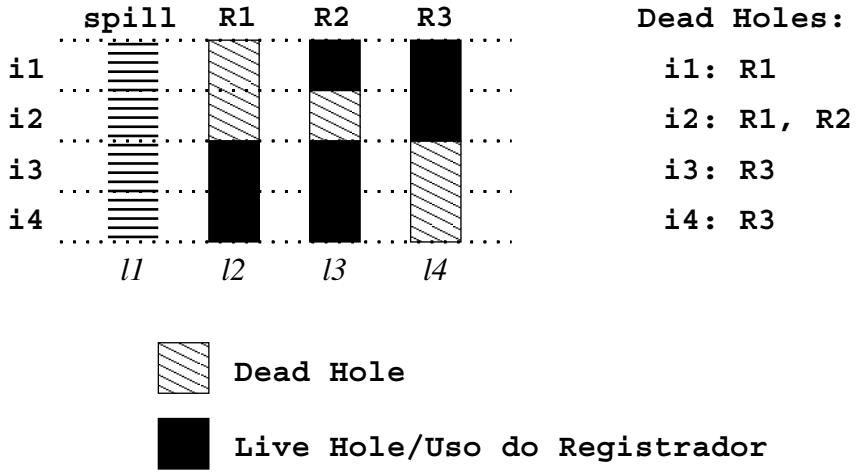


Figura 5.9: *Dead holes* em um bloco básico

De forma a encontrar a melhor maneira de utilizar os *dead holes* para alocar a *live range* $l1$ da Figura 5.9, um algoritmo de programação dinâmica [30] é empregado. O mesmo tenta todas as possibilidades de combinações para encontrar a combinação de *dead holes* que irá resultar no menor custo possível, isto é, na menor quantidade de instruções de ajuste a serem inseridas no programa.

As combinações que o algoritmo de programação dinâmica realiza, são mostradas na Figura 5.10, onde pode-se observar que é possível associar qualquer registrado a cada uma das instruções. A construção de tal malha de caminhos deve ser feita de forma a haver uma correspondência entre cada registrador e instrução do programa, isto é, um grafo dirigido onde o número de vértices será o total de registradores multiplicado pelo total de instruções, onde pode-se associar cada vértice a um par da forma (*registrador*,*instrução*), que daqui por diante será representado pela forma ($\mathtt{R}_\alpha$,$\mathtt{i}_n$), indicando um registrador $\mathtt{R}_\alpha$ associado a n-ésima instrução $\mathtt{i}_n$.

As arestas da Figura 5.10 indicam todos os caminhos que são possíveis nesta malha,

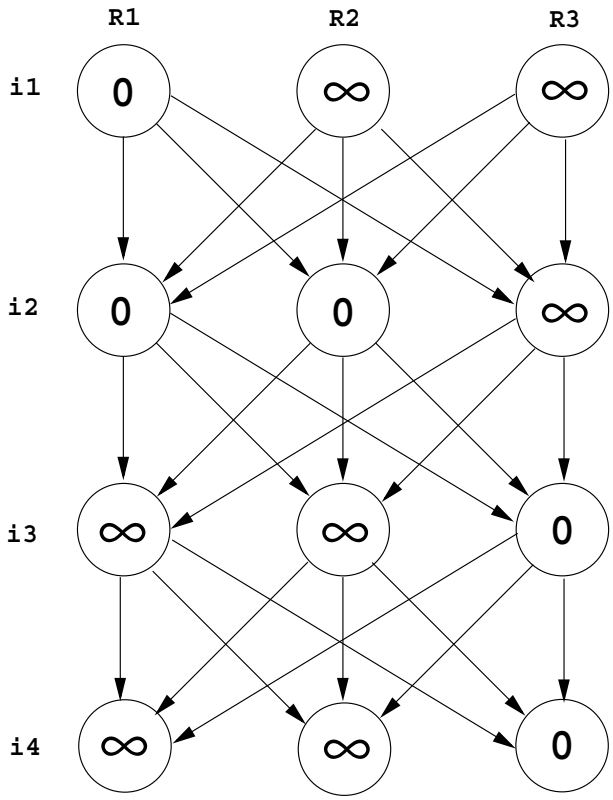


Figura 5.10: Caminhos possíveis em um bloco básico

onde cada caminho tem um custo associado. Um vértice rotulado com 0 indica que aquela combinação específica ($\mathtt{R}_\alpha$,$\mathtt{i}_n$) corresponde a um *dead hole*. Por exemplo, o vértice (R3,i3) possui o valor 0, e tal como indicado na Figura 5.9, na instrução i3 o registrador R3 corresponde a um *dead hole*. O vértice (R1,i4) está rotulado com o valor ∞, indicando que para a instrução i4 o registrador R1 não está disponível, isto é, não é um *dead hole*, como se confirma novamente na Figura 5.9.

Para encontrar qual caminho na Figura 5.10 irá resultar na menor quantidade de instruções de ajuste (menor custo), deve-se caracterizar o que seria uma solução ótima. Inicialmente, qualquer um dos registradores poderia ser usado para começar o caminho, mas como R2 e R3 possuem um custo $C_{r,i}$ (custo do registrador r na instrução i) de uso igual a ∞, isto é $C_{\mathtt{R2,i1}} = \infty$ e $C_{\mathtt{R3,i1}} = \infty$, a melhor opção é iniciar o caminho por R1, pois $C_{\mathtt{R1,i1}} = 0$, tendo então, custo total igual a 0. Desta forma se temos apenas uma instrução, escolher o menor custo é simples, uma vez que basta verificar o custo de se usar cada registrador e escolher aquele que resultar no menor custo. Se a arquitetura possuísse apenas um registrador, computar o custo total seria simples, pois bastaria verificar a disponibilidade de tal registrador em cada instrução da *live range* selecionada para *spill* e verificar a quantidade de instruções a serem inseridas, mas como mostra a Figura 5.10

em cada instrução `i2`, `i3`, ... `i`n, pode-se escolher entre 3 registradores para ser utilizado pela instrução.

Suponha que na instrução `i2` foi determinado que se deve utilizar o registrador `R2` para armazenar o valor da *live range* de *spill*. A princípio, utilizar a combinação (`R2`,`i2`) tem custo zero porque tal registrador é um *dead hole* em `i2`. Na instrução `i1`, qualquer um dos três registradores pode ter sido utilizado, e deseja-se saber qual combinação irá resultar no menor custo. Os custos a serem considerados são o custo total na instrução anterior e o custo $t_{n,n-1}$ de transferência (peso da aresta), isto é, de transferir o valor de um registrador em uma instrução $\texttt{i}_{n-1}$ para outro registrador em uma instrução $\texttt{i}_n$. Tal custo também pode ser representado por $t^{\texttt{R}a,\texttt{R}b}_{n,n-1}$, para indicar que a transferência ocorre do registrador `R`b na instrução $\texttt{i}_{n-1}$ para o registrador `R`a na instrução $\texttt{i}_n$.

	`(R1,i1)`		`(R2,i1)`		`(R3,i1)`	
	$C_{\texttt{R1,i1}}$	$t_{\texttt{i2,i1}}$	$C_{\texttt{R2,i1}}$	$t_{\texttt{i2,i1}}$	$C_{\texttt{R3,i1}}$	$t_{\texttt{i2,i1}}$
	0	1	∞	0	∞	1
C_T para `(R2,i2)`	1		∞		∞	

Tabela 5.1: Custos de se trocar de (`R`i,`i1`) para (`R2`,`i2`)

A Tabela 5.1 sumariza os custos envolvidos e o custo total de cada uma das combinações possíveis, sendo que a mesma foi construída assumindo-se que na instrução `i2` o registrador a ser usado seria `R2`. Suponha que na instrução `i1` a *live range* esteja no registrador `R1`. O custo do par (`R1`,`i1`) é 0 como se observa pela Figura 5.10, sendo representado por $C_{\texttt{R1,i1}} = 0$. O custo de se trocar de registrador de `i1` para `i2` é $t_{\texttt{i2,i1}} = 1$, isto porque deve-se inserir uma instrução *move* para copiar o valor que se encontra em `R1` para `R2`. Como o par (`R2`,`i2`) tem um custo $C_{\texttt{R2,i2}} = 0$, então podemos dizer que o custo total (representado por C_T) para esta troca é $C_T = C_{\texttt{R2,i2}} + t_{\texttt{i2,i1}} + C_{\texttt{R1,i1}}$, ou de outra forma: $C_T = 0 + 1 + 0 = 1$. Com isso temos o seguinte conjunto de trocas possíveis:

$$C_T = C_{\texttt{R2,i2}} + t_{\texttt{i2,i1}} + C_{\texttt{R1,i1}} = 0 + 1 + 0 = 1 \tag{5.1}$$

$$C_T = C_{\texttt{R2,i2}} + t_{\texttt{i2,i1}} + C_{\texttt{R2,i1}} = 0 + 0 + \infty = \infty \tag{5.2}$$

$$C_T = C_{\texttt{R2,i2}} + t_{\texttt{i2,i1}} + C_{\texttt{R3,i1}} = 0 + 1 + \infty = \infty \tag{5.3}$$

Observe que para a Equação 5.2 (mover de (`R2`,`i1`) para (`R2`,`i2`)) o custo de se trocar de registrador $t_{\texttt{i2,i1}} = 0$, isto porque o registrador origem e destino é o mesmo (`R2`),

fazendo com que não seja necessário inserir um instrução *move* para trocar o valor de registrador.

Pelo que se observa da Tabela 5.1, bem como da resolução das trocas possíveis, tem-se que o menor custo de se manter um valor em `R2` em `i2` será obtido ao se manter tal valor em `R1` em `i1`, sendo apenas necessário inserir uma instrução *move* entre `i1` e `i2`: `mov R2, R1`.

A Tabela 5.1 foi construída assumindo-se que na instrução `i2` o registrador a ser usado seria `R2`. No entanto, o que se deseja de fato, é saber qual a combinação ($\mathtt{R}i, \mathtt{i}_n$) para cada instrução de forma a se obter um custo total C_T mínimo, considerando todos os caminhos possíveis na malha da Figura 5.10. Pelo exemplo da Tabela 5.1, pode-se generalizar o custo de uma troca de registradores $\mathtt{R}_a$ e $\mathtt{R}_b$ entre duas instruções $\mathtt{i}_{n-1}$ e $\mathtt{i}_n$ para $n > 2$ como:

$$C = C_{\mathtt{R}_b,\mathtt{i}_n} + t_{n,n-1} + C_{\mathtt{R}_a,\mathtt{i}_{n-1}} \tag{5.4}$$

Pelo fato de existirem 3 registradores possíveis (correspondem ao $\mathtt{R}_a$ do passo anterior) para que um dado $\mathtt{R}_b$ possa escolher de forma a obter o menor custo, então a formulação completa do menor custo para troca de registradores entre duas instruções $\mathtt{i}_{n-1}$ e $\mathtt{i}_n$ para $n > 2$ é dada pela Equação 5.5:

$$C = C_{\mathtt{R}_b,\mathtt{i}_n} + min(t^{\mathtt{R}_b,\mathtt{R1}}_{n,n-1} + C_{\mathtt{R1},\mathtt{i}_{n-1}}, t^{\mathtt{R}_b,\mathtt{R2}}_{n,n-1} + C_{\mathtt{R2},\mathtt{i}_{n-1}}, t^{\mathtt{R}_b,\mathtt{R3}}_{n,n-1} + C_{\mathtt{R3},\mathtt{i}_{n-1}}) \tag{5.5}$$

Como dito anteriormente, quando se possui apenas uma instrução, o menor custo corresponderá ao registrador que possui o menor $C_{r,i}$. Com isso pode-se agora definir de forma recursiva a fórmula que dará o menor custo para um número arbitrário n de instruções e quantidade arbitrária α de registradores como:

$$C_T = \begin{cases} C_{r,i} & \text{se } n = 1 \\ C_{\mathtt{R}_b,\mathtt{i}_n} + min(\begin{array}{l} t^{\mathtt{R}_b,\mathtt{R1}}_{n,n-1} + C_{\mathtt{R1},\mathtt{i}_{n-1}}, \\ t^{\mathtt{R}_b,\mathtt{R2}}_{n,n-1} + C_{\mathtt{R2},\mathtt{i}_{n-1}}, \\ t^{\mathtt{R}_b,\mathtt{R3}}_{n,n-1} + C_{\mathtt{R3},\mathtt{i}_{n-1}}, \\ \vdots \\ t^{\mathtt{R}_b,\mathtt{R}\alpha}_{n,n-1} + C_{\mathtt{R}\alpha,\mathtt{i}_{n-1}}) \end{array} & \text{se } n \geq 2 \end{cases} \tag{5.6}$$

A Equação 5.6 fornece soluções para subproblemas, sendo que uma solução completa é definida por ela de forma recursiva, pois cada custo depende do valor do menor custo

anterior. Desta forma, o algoritmo utiliza uma tabela para armazenar os valores dos custos anteriores. A Tabela 5.2 contém os cálculos realizados pelo algoritmo para resolver a Figura 5.10. Embora a Equação 5.6 seja definida de forma recursiva, a solução é realizada em etapas, sendo considerada apenas uma instrução por vez.

	i1	i2	i3	i4
R1	0	0	∞	∞
R2	∞	1	∞	∞
R3	∞	∞	1	1

Tabela 5.2: Custos computados

A Tabela 5.2 contém uma coluna para cada instrução $\mathtt{i}_n$ do programa e uma linha para cada registrador $\mathtt{R}_\alpha$. A tabela é preenchida analisando-se primeiro a instrução i1 para cada registrador, ou seja, qual o custo de se realizar as combinações (R1,i1), (R2,i1) e (R3,i1). Neste caso, sempre a primeira opção ($n = 1$) da Equação 5.6 será executada, e cada valor obtido será armazenado na Tabela 5.2. A partir da instrução i2 somente a segunda parte da Equação 5.6 será executada ($n \geq 2$). A equação será executada para cada combinação possível em i2: (R1,i2), (R2,i2) e (R3,i2); sendo que desta vez cada uma das combinações já irá tentar obter o menor custo possível em relação ao nível anterior. Assim a seqüência de operações para a instrução i2 é:

$$\begin{aligned} C_T^{(\mathtt{R1,i2})} &= C_{\mathtt{R1,i2}} + min(t_{\mathtt{i2,i1}}^{\mathtt{R1,R1}} + C_{\mathtt{R1,i1}}, t_{\mathtt{i2,i1}}^{\mathtt{R1,R2}} + C_{\mathtt{R2,i1}}, t_{\mathtt{i2,i1}}^{\mathtt{R1,R3}} + C_{\mathtt{R3,i1}}) \\ &= 0 + min(0+0, 1+\infty, 1+\infty) \\ &= 0 + min(0, \infty, \infty) \\ &= 0 + 0 = 0 \end{aligned}$$

$$\begin{aligned} C_T^{(\mathtt{R2,i2})} &= C_{\mathtt{R2,i2}} + min(t_{\mathtt{i2,i1}}^{\mathtt{R2,R1}} + C_{\mathtt{R1,i1}}, t_{\mathtt{i2,i1}}^{\mathtt{R2,R2}} + C_{\mathtt{R2,i1}}, t_{\mathtt{i2,i1}}^{\mathtt{R2,R3}} + C_{\mathtt{R3,i1}}) \\ &= 0 + min(1+0, 0+\infty, 1+\infty) \\ &= 0 + min(1, \infty, \infty) \\ &= 0 + 1 = 1 \end{aligned}$$

$$\begin{aligned} C_T^{(\texttt{R3,i2})} &= C_{\texttt{R3,i2}} + min(t_{\texttt{i2,i1}}^{\texttt{R3,R1}} + C_{\texttt{R1,i1}}, t_{\texttt{i2,i1}}^{\texttt{R3,R2}} + C_{\texttt{R2,i1}}, t_{\texttt{i2,i1}}^{\texttt{R3,R3}} + C_{\texttt{R3,i1}}) \\ &= \infty + min(1+0, 1+\infty, 0+\infty) \\ &= \infty + min(1, \infty, \infty) \\ &= \infty + 1 = \infty \end{aligned}$$

Com isso obtêm-se os valores que preenchem a coluna `i2` da Tabela 5.2, e o processo é repetido para as demais instruções.

Além de computar o menor custo possível, o algoritmo deve ser capaz de lembrar o caminho percorrido, de forma a posteriormente poder associar os devidos registradores para cada uma das instruções.

	i2	i3	i4
$P_{\texttt{R1}}$	R1	R1	R1
$P_{\texttt{R2}}$	R1	R1	R1
$P_{\texttt{R3}}$	R1	R1	R3

$P_f = \texttt{R3}$

Tabela 5.3: Escolhas feitas pelo algoritmo

A Tabela 5.3 é utilizada para armazenar cada uma das escolhas que o algoritmo faz. Ela possui colunas rotuladas de `i2` até `i`n e cada uma das linhas é rotulada $P_{\texttt{R}_\alpha}$ para cada um dos registradores da arquitetura. As colunas iniciam-se na instrução `i2` pois é a partir desta instrução que a segunda parte da Equação 5.6 começa a ser utilizada. A Tabela 5.3 armazena o registrador escolhido na função *min* da equação. No exemplo apresentado de cômputo dos custos para a instrução `i2`, a função *min* escolheu sempre a seqüência $t_{n,n-1} + C_{\texttt{R}_a,\texttt{i}_{n-1}}$ que correspondia ao menor custo para um dado $\texttt{R}_a$, desta forma este $\texttt{R}_a$ é que deve ser armazenado na Tabela 5.3. Assim para os cômputos dos custos de `i2`, as seguintes escolhas foram feitas:

- Para $C_T^{(\texttt{R1,i2})}$ escolheu-se o registrador `R1` fazendo $[P_{\texttt{R1}},\texttt{i2}] = \texttt{R1}$
- Para $C_T^{(\texttt{R2,i2})}$ escolheu-se o registrador `R1` fazendo $[P_{\texttt{R2}},\texttt{i2}] = \texttt{R1}$
- Para $C_T^{(\texttt{R3,i2})}$ escolheu-se o registrador `R1` fazendo $[P_{\texttt{R3}},\texttt{i2}] = \texttt{R1}$

Um processo semelhante ocorre para as demais instruções, com as escolhas realizadas em cada etapa armazenadas na Tabela 5.3. Uma vez que as Tabelas 5.2 e 5.3 estejam preenchidas, o algoritmo terá determinado qual o menor custo total C_T e também qual

caminho resultou neste custo. Na Tabela 5.3 existe ainda um termo denominado P_f. Este termo é utilizado para se saber qual o registrador que foi escolhido quando se determinou o C_T mínimo. Pela Tabela 5.2 tem-se que para a instrução `i4` o menor custo foi obtido no par (`R3`,`i4`), assim $C_T = 1$. Com isso o registrador associado ao menor C_T é `R3`. Assim sendo P_f = `R3`, e a partir dele, pode ser determinado qual registrador cada uma das instruções anteriores a `i4` deve utilizar.

Algoritmo 1 Imprimir Registradores Escolhidos

```
n ← total of instructions in the program
r ← P_f
print(i_n, r)
for j ← n downto 2 do
  r ← P_r[i_j]
  print(i_{j-1}, r)
end for
```

Para determinar os registradores que cada instrução vai utilizar, utiliza-se o Algoritmo 1, que a partir do valor de P_f percorre a Tabela 5.3 e associa cada instrução $\texttt{i}_n$ do programa a um registrador $\texttt{R}_\alpha$. Para o exemplo da Figura 5.9, o algoritmo de *Hole Allocation* irá determinar a seqüência de registradores apresentada na Figura 5.11.

$$\begin{array}{ll} i_4 & R3 \\ i_3 & R3 \\ i_2 & R1 \\ i_1 & R1 \end{array}$$

Figura 5.11: Associação registrador/instrução feita pelo *Hole Allocation*

O caminho correspondente à associação registrador/instrução da Figura 5.11 é mostrado na Figura 5.12, onde se observa que o caminho na malha de trocas se inicia pelo registrador `R1` em `i1`, continuando em `R1` em `i2`. De `i2` para `i3`, o valor muda de `R1` para `R3`, tal troca corresponde ao valor 1 de C_T, ou seja, a inserção de uma instrução *move* que realiza a operação `mov R3, R1`. Em `i4` o valor continua em `R3`, finalizando assim a *live range* da variável inicialmente seleciona para *spill*.

Determinação de *Live/Dead Holes*

A Figura 5.9 serviu de guia para exemplificar o funcionamento do algoritmo *Hole Allocation*, mas ela parte do princípio de que *Live* e *Dead Holes* já estão computados.

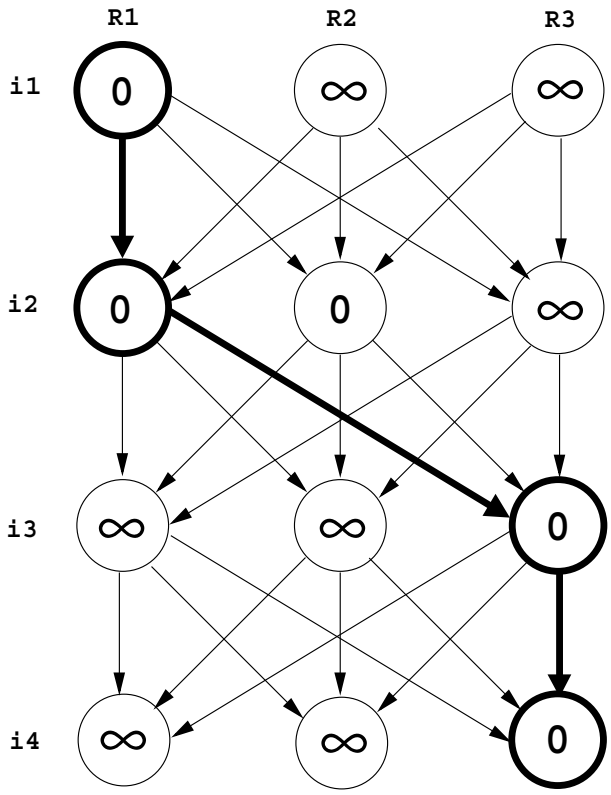


Figura 5.12: Caminho escolhido pelo algoritmo

O processo de se determinar os *Live/Dead Holes* de um programa é bem simples. O primeiro passo é realizar a análise de fluxo de dados, que no *Hole Allocation* é chamada de *Available Registers*[1]. Tal análise de fluxo de dados fornece informações que são exatamente o oposto da análise de fluxo de dados *Liveness Analysis*. Enquanto *Liveness Analysis* diz quais registradores estão vivos depois de cada instrução, *Available Registers* diz quais registradores estão mortos depois de cada instrução, isto é, os registradores que estão vazios depois de cada instrução. No Algoritmo 2 o primeiro *for* inicializa os conjuntos *gen* e *kill* para cada bloco básico, onde o cojunto *gen* contém os registradores disponíveis gerados pelo bloco e o cojunto *kill* contém os registradores que o bloco utiliza, ou seja, que não estão mais disponíveis. O segundo *for* inicializa os conjuntos *avail_in* e *avail_out* para cada bloco básico, tais conjuntos irão conter respectivamente os registradores disponíveis na entrada e saída de cada bloco básico. O *while* irá resolver as equações de fluxo de dados até que os conjuntos *avail_in* e *avail_out* de cada bloco básico se estabilizem, isto é, não mudem mais. O terceiro *for* irá então computar os registradores disponíveis depois de cada instrução.

[1]Registradores Disponíveis

Algoritmo 2 Computar *Available Registers*

```
for each basic block B do
  gen[B] = ∅
  kill[B] = ∅
  for each instruction i in B from last to first do
    gen[B] = gen[B] ∪ defs[i]
    gen[B] = gen[B] − uses[i]
    kill[B] = kill[B] − defs[i]
    kill[B] = kill[B] ∪ uses[i]
  end for
end for
for each basic block B do
  avail_in[B] = all registers
  avail_out[B] = all registers
end for
while any avail_in change do
  avail_out[B] = ∩ avail_in[P]
  avail_in[B] = gen[B] ∪ (avail_out[B] − kill[B])
end while
for each basic block B do
  avail = avail_out[B]
  for each instruction i in B from last to first do
    avail = avail ∪ defs[i]
    avail = avail − uses[i]
    avail[i] = avail
  end for
end for
```

Depois de que os *available registers* estão computados, todas as instruções do programa são percorridas uma única vez, onde os *Live/Dead Holes* são computados de acordo com o Algoritmo 3. Para cada instrução, os seus *dead holes* são obtidos removendo-se de seu conjunto *avail* os registradores que são usados e definidos na instrução, pois tais registradores podem pertencer ao conjunto *avail* da instrução sem necessariamente serem um *dead hole* na instrução em si. Os *live holes* são obtidos de forma semelhante, onde para cada instrução determina-se o conjunto-complemento. Inicialmente o conjunto *LiveHole* de cada instrução é preenchido com todos os registradores e em seguida remove-se os *dead holes* da instrução e os registradores que a mesma define e usa, restando apenas os *live holes*

Algoritmo 3 Computar *Live/Dead Holes*

```
for each basic block B do
  for each instruction i in B do
    DeadHoles[i] = avail[i]
    DeadHoles[i] = DeadHoles[i] − defs[i]
    DeadHoles[i] = DeadHoles[i] − uses[i]
    LiveHoles[i] = all registers
    LiveHoles[i] = LiveHoles[i] − DeadHoles[i]
    LiveHoles[i] = LiveHoles[i] − defs[i]
    LiveHoles[i] = LiveHoles[i] − uses[i]
  end for
end for
```

Algoritmo Principal

Algoritmo 4 *Hole Allocation*

```
Compute live/dead holes
Build and solve the graph that associates the registers to instructions
Determine the register assigned to each instruction
for each basic block B do
  R_c = ∅
  R_a = ∅
  for each instruction i in B do
    R_c = the register assigned to instruction i
    if i is not the first instruction then
      if R_c ≠ R_a then
        Before i insert the instruction: move R_c, R_a
      end if
    end if
    Rewrite i assigning R_c to it
    R_a = R_c
  end for
end for
```

O Algoritmo 4 contém os passos do algoritmo principal de *hole allocation*, concentrando tudo o que foi apresentado até o momento. Ele considera que apenas os *dead holes* serão utilizados para otimização e que a mesma será realizada somente dentro de blocos básicos, isto é, otimização local.

A primeira parte do algoritmo consiste em se computar os *live/dead holes* para cada instrução, o que pode ser feito utilizando-se os algoritmos 2 e 3. Uma vez que os buracos

tenham sido determinados, o próximo passo é construir a malha de caminhos, como no exemplo da Figura 5.10. Uma vez que tal malha esteja construída, ela é resolvida pela aplicação sucessiva da Equação 5.6, até que o caminho de menor custo tenho sido determinado para o bloco básico. Em seguida a aplicação do Algoritmo 1 irá determinar qual registrador cada instrução deverá utilizar.

A etapa final do algoritmo *hole allocation* consiste em reescrever o código e inserir as instruções de ajuste quando necessário. O *for* mais externo presente no Algoritmo 4 faz justamente isso. A variável R_c contém o registrador que foi associado à instrução corrente. A variável R_a contém o registrador que foi associado à instrução anterior. Se o registrador da instrução corrente é diferente do registrador da instrução anterior, então um *move* de R_a para R_c deve ser inserido no código antes da instrução corrente. O último passo é reescrever a instrução corrente para utilizar o registrador armazenado em R_c.

Embora não tenha sido apresentado aqui, em [13] pode ser encontrada a forma de se utilizar os *live holes* em conjunto com os *dead holes* para se gerar *spill code* em um grafo de fluxo de controle inteiro e não somente em um bloco básico.

Complexidade do Método

Embora a príncípio possa parecer que a complexidade de *hole allocation* seja exponencial, na verdade a sua complexidade é $O(n)$, onde n é o número de instruções do programa. O método de *hole allocation* envolve computar os *live/dead holes* e depois as instruções de ajuste, onde cada passo possui a seguinte complexidade:

- *Available Registers*: Possui a mesma complexidade de *Liveness Analysis* [7, 10, 58], sendo de $O(n^2)$ no pior caso.
- *Live/Dead Holes*: A complexidade é da ordem de $O(n)$, pois deve-ser percorrer cada uma das instruções do programa uma única vez.
- Instruções de Ajuste: A complexidade de se computar as instruções de ajuste também é da ordem de $O(n)$, pois basta percorrer também cada uma das instruções do programa uma única vez.

A estapa principal do algoritmo tem complexidade $O(n)$, que corresponde a computar as instruções de ajuste. No caso de o algoritmo ser repetido no mesmo programa várias vezes, o cômputo dos *available registers* e dos *live/dead holes* é realizado uma vez, pois em uma segunda execução do método, basta atualziar a informação sobre *live/dead holes* o que pode ser feito percorrendo-se as instruções do programa uma vez. Desta forma o passo mais caro, computar *available registers*, só é realizado uma vez.

5.3 Hole Allocation no StarDBT

Em DBT, otimizações são uma forma de compensar o *overhead* da tradução do programa binário e também em uma oportunidade de melhorar o desempenho do código traduzido. Com este intuito, a técnica *hole allocation* foi adaptada para funcionar como uma otimização dinâmica no StarDBT que, ao invés de gerar *spill code*, irá transformar acessos a memória em acessos a registradores, pois utilizar registradores é bem mais rápido do que acessar a memória.

Um bom compilador tentará fazer o melhor uso possível dos registradores disponíveis, de forma a gerar um código que seja o mais rápido possível. De modo a entender como um código já altamente otimizado pode ser melhorado, a Figura 5.13 contém um trecho de código assembly que foi retirado do programa 256.bzip2 que faz parte do benchmark SPEC. Tal código assembly foi gerado pelo compilador ICC da Intel Versão 9.1 [2], utilizando-se o nível mais alto de otimização e informação de *profile*, isto é, compilou-se o programa uma vez, realizou-se uma execução do mesmo, onde dados do seu comportamento foram gerados, e uma nova compilação do programa foi realizada com base nesses dados.

A Figura 5.13 contém apenas um trecho de código retirado de um *trace* gerado pelo StarDBT. A análise deste trecho tem por finalidade encontrar *dead/live holes* para verificar se os mesmos podem ser utilizados em otimizações. Em código assembly, as *live ranges* são os registradores da arquitetura alvo, neste caso a arquitetura x86. Os registradores analisados foram: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI` e `EBP`.

A Figura 5.13 contém uma tabela em sua porção direita, contendo uma análise de cada registrador, sendo que cada instrução é analisada individualmente para cada registrador. Um espaço vazio (`| |`) na tabela significa que naquele ponto o registrador está livre, isto é, não contém nenhum valor útil ao programa e portanto é um *dead hole*. Um espaço preenchido (`|#|`) indica que naquele ponto o registrador está vivo, isto é contém um valor que será referenciado no futuro, mas que naquele ponto não está sendo utilizado, constituindo assim um *live hole*. Um espaço marcado com um traço (`|-|`), indica que naquele ponto o registrador está sendo acessado pela instrução, seja para leitura ou escrita; tais pontos não são nem *dead holes* nem *live holes*. Um espaço preenchido com (`|?|`) indica que não foi possível determinar se o registrador é um *dead hole* ou *live hole* neste ponto.

Mesmo para programas altamente otimizados, ainda é possível encontrar buracos nas *live ranges* dos registradores. Presume-se que *live holes* irão de fato ocorrer, pois uma variável pode estar viva ao longo de todo um procedimento, sem ser necessariamente referenciada todo o tempo. *Dead holes* por sua vez, deveriam ser poucos e pequenos, principalmente em uma arquitetura com poucos registradores como a x86. O que se observa é que com exceção do registrador `EAX`, todos os demais registradores analisados

```
(bzip2)                                                          +-+-+-+-+-+-+-+-+
                                                                 |E|E|E|E|E|E|E|E|
                                                                 |A|B|C|D|S|D|B|
                                                                 |X|X|X|X|I|I|P|
                                                                 +-+-+-+-+-+-+-+-+
0ab3fd821h:          mov     DWORD PTR [esp+094h], ebx           |#|-| |#| | |?|
0ab3fd828h:          xor     eax, eax                            |-| | |#| | |?|
0ab3fd82ah:          xor     edx, edx                            |#| | |-| | |?|
0ab3fd82ch:          mov     DWORD PTR [esp+060h], 00h           |#| | | | | |?|
0ab3fd830h:          mov     DWORD PTR [esp+038h], 00h           |#| | | | | |?|
0ab3fd834h:          mov     esi, 00h                            |#| | | |-| |?|
0ab3fd836h:          mov     DWORD PTR [esp+06ch], 00h           |#| | | |#| |?|
0ab3fd83ah:          mov     edi, DWORD PTR [esp+078h]           |#| | | |#|-|?|
0ab3fd83eh:          mov     DWORD PTR [esp+064h], 00h           |#| | | |#|#|?|
0ab3fd842h:          mov     ecx, 00h                            |#| |-| |#|#|?|
0ab3fd844h:          mov     DWORD PTR [esp+05ch], 00h           |#| |#| |#|#|?|
0ab3fd848h:          mov     edx, DWORD PTR [esp+098h]           |#| |#|-|#|#|?|
0ab3fd84fh:          lea     edx, DWORD PTR [edx+edi*2]          |#| |#|-|#|-|?|
0ab3fd852h:          mov     edi, DWORD PTR [esp+084h]           |#| |#|#|#|-|?|
0ab3fd859h:          mov     DWORD PTR [esp+068h], edx           |#| |#|-|#|#|?|
0ab3fd86ch:          mov     ebx, edx                            |#|-|#|-|#|#|?|
0ab3fd85dh:          lea     edx, DWORD PTR [edx+edi*4-4]        |#|#|#|-|#|-|?|
0ab3fd861h:          mov     DWORD PTR [esp+0c0h], edx           |#|#|#|-|#| |?|
0ab3fd868h:          mov     edx, DWORD PTR [esp+05ch]           |#|#|#|-|#| |?|
0ab3fd870h:          movzx   edi, WORD PTR [ebx]                 |#|-|#|#|#|-|?|
0ab3fd873h:          movzx   ebx, BYTE PTR [edi+080cffa0h]       |#|-|#|#|#|-|?|
0ab3fd87ah:          add     edx, ebx                            |#|-|#|-|#|#|?|
                                                                 +-+-+-+-+-+-+-+-+
```

```
|-| Uso ou definição do registrador
|#| Registrador está vivo     (Live Hole)
| | Registrador não está vivo (Dead Hole)
|?| Indeterminado
```

Figura 5.13: *Dead/Live Holes* em trecho de código assembly

possuem *dead holes* em suas *live ranges*. O registrador `EBX` possui um *dead hole* que se estende por 14 instruções de um total de 22, implicando que tal registrador fica morto durante aproximadamente 63,6% das instruções do *trace*, um número bastante elevado se considerarmos que um *trace* é escolhido justamente por ser uma região de código que é muito executada. O registrador `EBP` nunca foi referenciado durante o trecho de código da Figura 5.13, indicando que toda a sua *live range* pode ser um *dead hole* ou um *live hole*, mas para responder a tal questão é necessário análise adicional do *trace*, através de *Cold Code Analysis*.

A análise da Figura 5.13 revela que *dead holes* e *live holes* de fato existem em código altamente otimizado. A próxima questão é se eles podem ser utilizados para se otimizar código e como utilizá-los. A Figura 5.14 pode fornecer uma pista para responder à questão

```
mov       edi, edx
mov       DWORD PTR [esp+034h], eax
lea       edx, DWORD PTR [eax-1]
mov       esi, ebx
xor       ecx, ecx
mov       DWORD PTR [esp+02ch], edx
mov       edx, 0x1h
lea       edi, DWORD PTR [edi+edi-1]
mov       DWORD PTR [esp+030h], edi
mov       edi, DWORD PTR [esp+02ch]
movzx     eax, BYTE PTR [esi]
cmp       eax, edi
```

Figura 5.14: Código assembly de um *trace*

formulada. Ela também contém código assembly de um *trace* do programa 256.bzip2 compilado pelo ICC da Intel.

Ao se analisar os acessos à memória, descobre-se que duas instruções, `mov DWORD PTR [esp+02ch], edx` e `mov edi, DWORD PTR [esp+02ch]`, estão referenciando a mesma posição de memória (`DWORD PTR [esp+02ch]`), como indicado na Figura 5.15, sendo que a primeira referência escreve na memória e a segunda referência lê a memória.

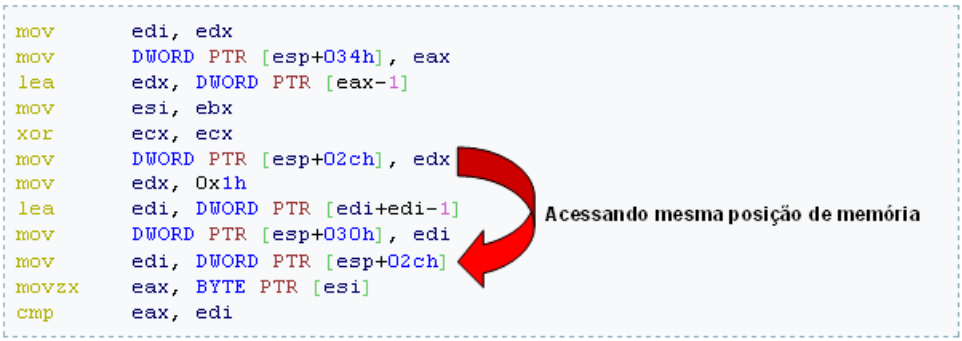


Figura 5.15: Acessos de memória iguais

Uma análise das *live ranges* dos registradores no trecho de código da Figura 5.14 irá mostrar que o registrador `EAX` possui um *dead hole* cobrindo 7 instruções do *trace* como mostrado na Figura 5.16. O retângulo que delimita o *dead hole* do registrador `EAX` também engloba as duas instruções que acessam a posição de memória `DWORD PTR [esp+02ch]`.

A idéia por traz da otimização é tentar remover um dos acessos à memória, fazendo com que uma das instruções acesse o registrador `EAX`. A instrução a ser otimizada deve ser aquela que lê a memória, pois se a instrução que escreve na memória for modificada, de modo a não mais armazenar o valor na memória, problemas de consistência podem

ocorrer, pois instruções fora do *trace* podem ler tal posição de memória e obter um valor errado; com isso, ao se otimizar tal padrão de instruções, opta-se por manter a memória consistente.

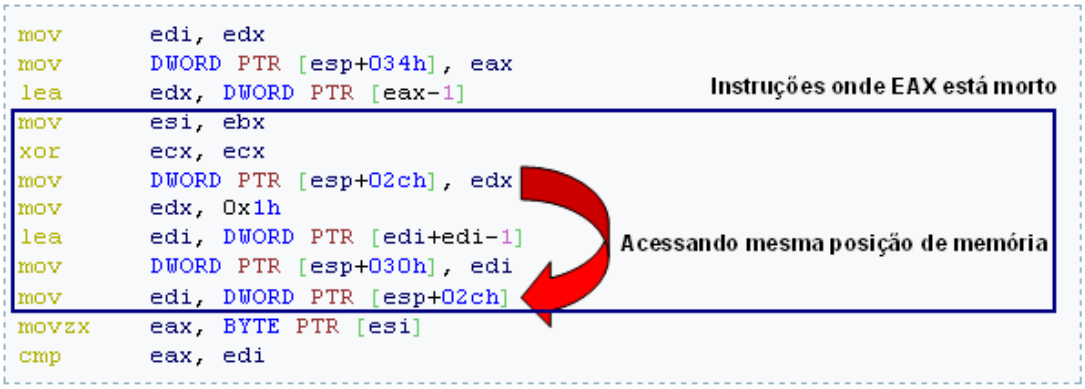


Figura 5.16: *Dead hole* do registrador `EAX`

A Figura 5.17 mostra todo o processo de otimização do trecho de código da Figura 5.14, sendo que à esquerda se encontra o código antes da otimização e à direita o código já otimizado. A otimização consiste em, adicionar após a instrução `mov DWORD PTR [esp+02ch], edx`, uma nova instrução: `mov eax, edx`. Essa nova instrução irá copiar o valor do registrador `edx` para o registrador `eax`; observe que agora tanto a posição de memória `DWORD PTR [esp+02ch]`, quanto `eax` contém o mesmo valor. O próximo passo é modificar a instrução `mov edi, DWORD PTR [esp+02ch]`, transformando-a em `mov edi, eax`. Tal transformação é possível, pois como já dito, `eax` contém o mesmo valor da posição de memória que seria lida e também pelo fato de `eax` ser um *dead hole* no trecho em questão.

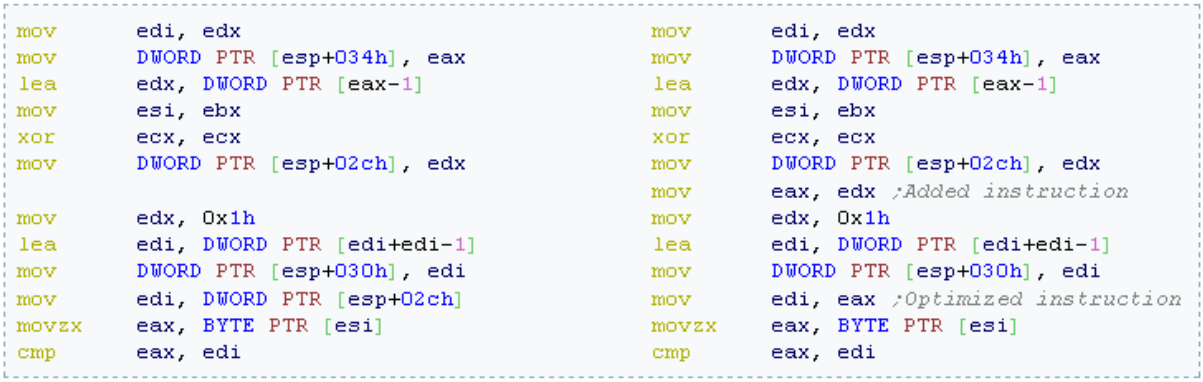


Figura 5.17: Otimização do código assembly

Na Figura 5.17 somente um acesso à memória foi transformado, mas se o *dead hole* de

`eax` fosse maior e houvesse outras leituras da posição de memória `DWORD PTR [esp+02ch]`, todas poderiam ser transformadas em leituras de `eax`, desde que não ocorresse nenhuma outra modificação do endereço de memória sendo otimizado.

5.3.1 Particularidades do Ambiente de Tradução Dinâmica

No exemplo da Figura 5.14 a otimização encontrou dois endereços da forma `[esp + xxxxh]` para otimizar, mas a otimização não precisa ser limitada somente a este tipo de padrão de endereço. Quaisquer endereços iguais podem ser otimizados usando-se a mesma idéia apresentada. Observe que se a arquitetura destino contém mais registradores que a arquitetura origem (tradução de IA32 para IA64 por exemplo), inicialmente não é necessário buscar por *dead holes*, pois os registradores extras não usados estarão completamente livres e poderão ser utilizados como grandes *dead holes*. Na implementação da otimização, *live holes* não são utilizados por envolver a inserção de instruções *load* e *store* o que inviabilizaria o objetivo de remover *load*'s e *store*'s.

Na Figura 5.14 a otimização procurou por endereços baseados no padrão `[esp + xxxxh]`, que em geral correspondem a *spill code* gerado pelo compilador estático em tempo de compilação, desta forma a otimização está revertendo *spill code* em usos de registradores, fazendo com que pedaços da *live range* da variável sejam alocados em *dead holes* de forma semelhante à Figura 5.7. Embora qualquer acesso à memória possa ser otimizado, existe o problema de *alias* [7, 44, 58], isto é, onde uma mesma posição de memória pode ser acessada de duas formas diferentes. Um simples exemplo que pode criar um *alias* de memória é:

```
lea eax, DWORD PTR [esp+02ch]
```

Essa simples instrução faz com que o registrador `eax` aponte para a mesma posição de memória que `esp+02ch`, fazendo com que `eax` seja um *alias* de tal endereço de memória. Se um *store* for otimizado, sem levar em conta o registrador `eax`, a otimização pode criar um erro no programa. Outro tipo de situação que pode ocorrer é mostrada no trecho de código a seguir:

```
mov DWORD PTR [esp+014h], eax
push ebx
mov ebx, DWORD PTR [esp+014h]
```

Em um primeiro momento, pode-se pensar que as duas referências à memória `[esp + 014h]` apontam para a mesma posição de memória, mas a instrução `push ebx` modifica o registrador `esp`, fazendo com que a segunda referência para `[esp + 014h]` corresponda

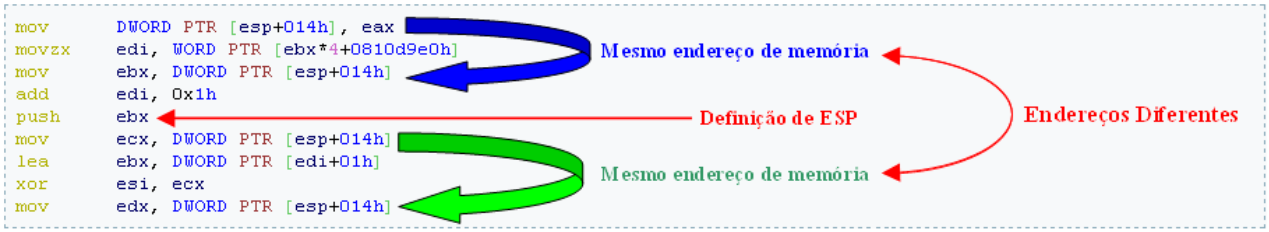


Figura 5.18: Desambigüação de Memória

a um endereço diferente da primeira referência, fazendo com que a otimização não possa ser aplicada neste caso.

A Figura 5.18 apresenta a situação de uma forma mais clara. Ela contém quatro referências à memória seguindo o padrão `[esp+014h]`. Pode-se observar que a primeira referência escreve na memória e as demais realizam leitura da memória. Se existir um *dead hole* que englobe as 4 referências, então inicialmente as três instruções que realizam a leitura da memória poderiam ser transformadas em usos de registrador, mas isto não é possível, pois como está indicado na Figura 5.18 pela seta em vermelho, existe a instrução `push ebx` que modifica o valor do registrador `esp`, fazendo com que as duas últimas referências apontem para um endereço de memória diferente das duas primeiras. Com isso tem-se que as instruções `mov DWORD PTR [esp+014h], eax` e `mov ebx, DWORD PTR [esp+014h]` apontam para uma posição de memória; ao passo que as instruções `mov ecx, DWORD PTR [esp+014h]` e `mov edx, DWORD PTR [esp+014h]` apontam para outra posição de memória. Obviamente os dois últimos acessos à memória podem ser otimizados se houver um *dead hole* englobando tais instruções, sendo que neste caso somente o segundo acesso à memória seria transformado em uso de registrador.

Embora quaisquer duas instruções que acessem a mesma posição de memória possam ser otimizadas, situações como as apresentadas aqui são fatores complicadores para a otimização. Um problema maior é a desambigüação de memória [44], isto é de acessos à memória que não seguem um padrão como as instruções da forma `[esp + xxxxh]`. Nestes casos é muito mais complicado determinar se as posições de memória acessadas são as mesmas, como mostram as instruções abaixo:

```
mov DWORD PTR[edi*4+080cd38h], ebx
mov edi, DWORD PTR[eax+080d4560h]
```

O endereço de memória que é acessado em cada uma das instruções vai depender do valor dos registradores `edi` na primeira instrução e `eax` na segunda, os quais somente são conhecidos em tempo de execução. Desta forma, a implementação de *hole allocation* do StarDBT somente analisa os acessos à memória que seguem o padrão `[esp + xxxxh]`, em que a simples análise da instrução já permite identificar acessos semelhantes.

5.3.2 Análise de Traces Otimizados

O *Hole Allocation* no StarDBT como uma otimização dinâmica irá realizar transformações no código assembly traduzido da aplicação em execução. Como uma otimização solitária, espera-se que ela gere melhoras no código traduzido bem como crie novas oportunidades de otimização para outras transformações de código que eventualmente venham a ser executadas no ambiente de tradução dinâmica.

Esta seção apresenta alguns *traces* otimizados por *Hole Allocation* no StarDBT. Todos os *traces* foram removidos do programa 256.bzip2 do benchmark SPEC, o qual foi compilado pelo compilador da Intel com informação de *profile*.

A Figura 5.19 apresenta um *trace* que possui número de identificação igual 281. Na porção esquerda da figura encontra-se o código assembly antes de otimização e na porção direita encontra-se o código assembly já otimizado. Neste *trace* o registrador `EDI` foi identificado como sendo um *dead hole* e foi utilizado para remover um acesso à memória. Observe que após a otimização a seguinte seqüência de instruções acabou sendo criada:

```
mov edi, esi
mov esi, edi
```

Claramente tal seqüência de instruções pode ser removida por aplicações subseqüentes de outras otimizações, como *copy propagation* [7,58] e *dead code elimination* [7,58], resultando em um código final menor. Um caso idêntico ocorre na Figura 5.21, onde novamente a mesma seqüência de instruções aparece após a otimização do *trace*.

A Figura 5.20 contém um *trace* onde foram utilizados dois registradores diferentes como *dead holes*. Em um primeiro momento utilizou-se o registrador `ESI` e posteriormente utilizou-se o registrador `EDI`, sendo que foi necessário inserir a instrução `mov edi, esi` para trocar o valor de buraco. Observe que o acesso otimizado à memória ocorre logo após a instrução que realiza a troca de buracos, criando novamente uma seqüência de instruções que pode ser removida por outras otimizações (*copy propagation*, *dead code elimination*) , diminuindo assim o tamanho do código traduzido.

A Figura 5.22 contém um caso diferente dos demais pois, ao invés de causar melhora no código traduzido, a otimização gera mais instruções, sem remover nenhum acesso à memória. Nota-se que, ao contrário dos demais *traces* apresentados, na Figura 5.22 ocorre primeiro uma instrução que lê uma posição da memória e uma instrução depois, aparece uma escrita nesta posição de memória. Como escritas na memória não podem ser removidas, sob pena de se inserir um erro no programa, as instruções que copiam o valor utilizado para o *dead hole* são inseridas, ao passo que nenhum acesso à memória é transformado em uso de registrador. Tal situação cria não uma otimização, mas sim uma situação onde mais tempo está sendo gasto pelo código assembly traduzido, por conta de duas instruções extras.

A Figura 5.23 contém um caso onde a otimização foi bem sucedida sem que efeitos colaterais fossem gerados, fazendo com que a aplicação de outras otimizações não consiga realizar nenhuma outra transformação de código.

A Figura 5.24 por sua vez gerou, como em outros exemplos, uma situação interessante. O registrador escolhido como *dead hole* é o mesmo que contém o valor escrito na memória, gerando assim a instrução:

```
mov ecx, ecx
```

Tal instrução é claramente um **código morto**[2] e como tal pode ser removida pela aplicação de outras otimizações, como *dead code elimination*, por exemplo.

A análise dos *traces* apresentados mostrou como *Hole Allocation* pode modificar o código de forma a remover acessos à memória, bem como criar oportunidades para outras otimizações agirem no código traduzido, pois a aplicação das otimizações de código bem conhecidas [7] em um programa altamente otimizado por tais otimizações, dificilmente irá resultar em alguma nova transformação ou melhoria, sendo necessário que outras transformações criem as condições necessárias para a aplicações de outras otimizações, sempre com a esperança de criar novas oportunidades para as demais otimizações que serão aplicadas.

[2]Instrução sem efeito para o programa

Trace 281

```
;Original Code                                  ;Optimized Code
mov      eax, DWORD PTR [esp+0134h]             mov      eax, DWORD PTR [esp+0134h]
movzx    ebx, BYTE PTR [eax+080d0100h]          movzx    ebx, BYTE PTR [eax+080d0100h]
mov      eax, DWORD PTR [ebx*4+080df6f0h]       mov      eax, DWORD PTR [ebx*4+080df6f0h]
mov      edx, DWORD PTR [0x80df728h]            mov      edx, DWORD PTR [0x80df728h]
add      DWORD PTR [esp+0138h], -0x1            add      DWORD PTR [esp+0138h], -0x1
cmp      edx, eax                               cmp      edx, eax
jnge     ORG 0x0804dc21                         jnge     ORG 0x0804dc21
mov      esi, DWORD PTR [0x80df724h]            mov      esi, DWORD PTR [0x80df724h]
mov      DWORD PTR [esp+014ch], esi             mov      DWORD PTR [esp+014ch], esi
                                                mov      edi, esi ;Added instruction
mov      esi, DWORD PTR [esp+014ch]             mov      esi, edi ;Optimized instruction
sub      edx, eax                               sub      edx, eax
mov      ecx, edx                               mov      ecx, edx
shr      esi, cl                                shr      esi, cl
mov      ecx, eax                               mov      ecx, eax
mov      edi, 0x1h                              mov      edi, 0x1h
shl      edi, cl                                shl      edi, cl
add      edi, -0x1                              add      edi, -0x1
and      esi, edi                               and      esi, edi
mov      edi, ebx                               mov      edi, ebx
shl      edi, 0xah                              shl      edi, 0xah
lea      ebx, DWORD PTR [edi+ebx*8]             lea      ebx, DWORD PTR [edi+ebx*8]
cmp      esi, DWORD PTR [ebx+eax*4+080da600h]   cmp      esi, DWORD PTR [ebx+eax*4+080da600h]
mov      DWORD PTR [esp+013ch], ebx             mov      DWORD PTR [esp+013ch], ebx
jg       ORG 0x0804da3d                         jg       ORG 0x0804da3d
jmp      ORG 0x0804da35                         jmp      ORG 0x0804da35
```

Figura 5.19: *Trace* 281 do programa 256.bzip2 do benchmark SPEC

Trace 282

```
;Original Code                                  ;Optimized Code
mov     eax, DWORD PTR [esp+0134h]              mov     eax, DWORD PTR [esp+0134h]
movzx   esi, BYTE PTR [eax+080d0100h]           movzx   esi, BYTE PTR [eax+080d0100h]
mov     edi, DWORD PTR [esi*4+080df6f0h]        mov     edi, DWORD PTR [esi*4+080df6f0h]
mov     DWORD PTR [esp+011ch], esi              mov     DWORD PTR [esp+011ch], esi
mov     DWORD PTR [esp+013ch], edi              mov     DWORD PTR [esp+013ch], edi
add     DWORD PTR [esp+0138h], -0x1             add     DWORD PTR [esp+0138h], -0x1
cmp     ecx, edi                                cmp     ecx, edi
jnge    ORG 0x0804df9f                          jnge    ORG 0x0804df9f
mov     eax, DWORD PTR [0x80df724h]             mov     eax, DWORD PTR [0x80df724h]
mov     DWORD PTR [esp+0120h], eax              mov     DWORD PTR [esp+0120h], eax
                                                mov     esi, eax ;Added instruction
mov     eax, edi                                mov     eax, edi
                                                mov     edi, esi ;Changing hole
mov     esi, DWORD PTR [esp+0120h]              mov     esi, edi ;Optimized instruction
sub     ecx, eax                                sub     ecx, eax
mov     DWORD PTR [esp+0124h], ecx              mov     DWORD PTR [esp+0124h], ecx
shr     esi, cl                                 shr     esi, cl
mov     ecx, eax                                mov     ecx, eax
mov     edi, 0x1h                               mov     edi, 0x1h
shl     edi, cl                                 shl     edi, cl
mov     ecx, DWORD PTR [esp+011ch]              mov     ecx, DWORD PTR [esp+011ch]
add     edi, -0x1                               add     edi, -0x1
and     esi, edi                                and     esi, edi
mov     edi, ecx                                mov     edi, ecx
shl     edi, 0xah                               shl     edi, 0xah
lea     edi, DWORD PTR [edi+ecx*8]              lea     edi, DWORD PTR [edi+ecx*8]
cmp     esi, DWORD PTR [edi+eax*4+080da600h]     cmp     esi, DWORD PTR [edi+eax*4+080da600h]
mov     ecx, DWORD PTR [esp+0124h]              mov     ecx, DWORD PTR [esp+0124h]
jg      ORG 0x0804db38                          jg      ORG 0x0804db38
mov     eax, DWORD PTR [esp+0130h]              mov     eax, DWORD PTR [esp+0130h]
mov     DWORD PTR [0x80df728h], ecx             mov     DWORD PTR [0x80df728h], ecx
mov     DWORD PTR [esp+0140h], ebx              mov     DWORD PTR [esp+0140h], ebx
mov     ebx, DWORD PTR [esp+013ch]              mov     ebx, DWORD PTR [esp+013ch]
sub     esi, DWORD PTR [edi+ebx*4+080dbe40h]    sub     esi, DWORD PTR [edi+ebx*4+080dbe40h]
mov     ebx, DWORD PTR [edi+esi*4+080dd680h]    mov     ebx, DWORD PTR [edi+esi*4+080dd680h]
test    ebx, ebx                                test    ebx, ebx
je      ORG 0x0804decb                          je      ORG 0x0804decb
cmp     ebx, 0x1h                               cmp     ebx, 0x1h
je      ORG 0x0804dd0a                          je      ORG 0x0804dd0a
jmp     ORG 0x0804dbbc                          jmp     ORG 0x0804dbbc
```

Figura 5.20: *Trace* 282 do programa 256.bzip2 do benchmark SPEC

Trace 285

```
;Original Code                                  ;Optimized Code
mov     esi, DWORD PTR [0x80df724h]             mov     esi, DWORD PTR [0x80df724h]
mov     DWORD PTR [esp+014ch], esi              mov     DWORD PTR [esp+014ch], esi
                                                mov     edi, esi ;Added instruction
mov     esi, DWORD PTR [esp+014ch]              mov     esi, edi ;Optimized instruction
sub     edx, eax                                sub     edx, eax
mov     ecx, edx                                mov     ecx, edx
shr     esi, cl                                 shr     esi, cl
mov     ecx, eax                                mov     ecx, eax
mov     edi, 0x1h                               mov     edi, 0x1h
shl     edi, cl                                 shl     edi, cl
add     edi, -0x1                               add     edi, -0x1
and     esi, edi                                and     esi, edi
mov     edi, ebx                                mov     edi, ebx
shl     edi, 0xah                               shl     edi, 0xah
lea     ebx, DWORD PTR [edi+ebx*8]              lea     ebx, DWORD PTR [edi+ebx*8]
cmp     esi, DWORD PTR [ebx+eax*4+080da600h]    cmp     esi, DWORD PTR [ebx+eax*4+080da600h]
mov     DWORD PTR [esp+013ch], ebx              mov     DWORD PTR [esp+013ch], ebx
jg      ORG 0x0804da3d                          jg      ORG 0x0804da3d
jmp     ORG 0x0804da35                          jmp     ORG 0x0804da35
```

Figura 5.21: *Trace* 285 do programa 256.bzip2 do benchmark SPEC

Trace 311

```
;Original Code                                  ;Optimized Code
mov     eax, DWORD PTR [esp+0134h]              mov     edx, DWORD PTR [esp+0134h]
                                                mov     eax, edx ;Added instruction
add     eax, 0x1h                               add     eax, 0x1h
mov     DWORD PTR [esp+0134h], eax              mov     DWORD PTR [esp+0134h], eax
                                                mov     edx, eax ;Added instruction
mov     edx, 0x32h                              mov     edx, 0x32h
mov     DWORD PTR [esp+0138h], edx              mov     DWORD PTR [esp+0138h], edx
jmp     ORG 0x0804d9ca                          jmp     ORG 0x0804d9ca
```

Figura 5.22: *Trace* 311 do programa 256.bzip2 do benchmark SPEC

Trace 346

```
;Original Code                          ;Optimized Code
mov     edx, DWORD PTR [esp+020h]       mov     edx, DWORD PTR [esp+020h]
test    edx, edx                        test    edx, edx
jle     ORG 0x0804ee60                  jle     ORG 0x0804ee60
mov     edi, edx                        mov     edi, edx
mov     DWORD PTR [esp+034h], eax       mov     DWORD PTR [esp+034h], eax
lea     edx, DWORD PTR [eax-1]          lea     edx, DWORD PTR [eax-1]
mov     esi, ebx                        mov     esi, ebx
xor     ecx, ecx                        xor     ecx, ecx
mov     DWORD PTR [esp+02ch], edx       mov     DWORD PTR [esp+02ch], edx
                                        mov     eax, edx ;Added instruction
mov     edx, 0x1h                       mov     edx, 0x1h
lea     edi, DWORD PTR [edi+edi-1]      lea     edi, DWORD PTR [edi+edi-1]
mov     DWORD PTR [esp+030h], edi       mov     DWORD PTR [esp+030h], edi
mov     edi, DWORD PTR [esp+02ch]       mov     edi, eax ;Optimized instruction
movzx   eax, BYTE PTR [esi]             movzx   eax, BYTE PTR [esi]
cmp     eax, edi                        cmp     eax, edi
je      ORG 0x0804eddf                  je      ORG 0x0804eddf
jmp     ORG 0x0804eb32                  jmp     ORG 0x0804eb32
```

Figura 5.23: *Trace* 346 do programa 256.bzip2 do benchmark SPEC

Trace 365

```
;Original Code                          ;Optimized Code
movzx   eax, BYTE PTR [esp+edx+028h]    movzx   eax, BYTE PTR [esp+edx+028h]
test    eax, eax                        test    eax, eax
je      ORG 0x0804a4a1                  je      ORG 0x0804a4a1
mov     DWORD PTR [esp+0ach], edx       mov     DWORD PTR [esp+0ach], edx
mov     ecx, edx                        mov     ecx, edx
shl     ecx, 0x4h                       shl     ecx, 0x4h
mov     DWORD PTR [esp+0c0h], ecx       mov     DWORD PTR [esp+0c0h], ecx
                                        mov     ecx, ecx ;Added instruction (Dead Code)
mov     eax, -0x10                      mov     eax, -0x10
mov     edx, DWORD PTR [esp+0c0h]       mov     edx, ecx ;Optimized instruction
movzx   ecx, BYTE PTR [edx+eax+080cdb90h]  movzx   ecx, BYTE PTR [edx+eax+080cdb90h]
test    ecx, ecx                        test    ecx, ecx
je      ORG 0x0804af64                  je      ORG 0x0804af64
jmp     ORG 0x0804a3f6                  jmp     ORG 0x0804a3f6
```

Figura 5.24: *Trace* 365 do programa 256.bzip2 do benchmark SPEC

5.4 Resultados Experimentais

5.4.1 Ambiente Dinâmico

Para se avaliar o desempenho de *hole allocation* no StarDBT utilizou-se todo o benchmark SPEC, sendo que o mesmo foi compilado com o compilador da Intel Versão 9.1 com informação de *profile* e nível de otimização $-O3$. O computador utilizado foi um Intel Core 2 Duo de 2.66 GHz com 1GB de memória RAM executando o Sistema Operacional Linux, sendo que a distribuição utilizada foi o Fedora 6.

O processo de avaliação consistiu em um primeiro momento executar cada programa do SPEC e gerar os seus *traces*, os quais foram otimizados por *hole allocation*, **utilizando informações providas por** *Cold Code Analysis*, e então gravados em arquivo. Três conjuntos de *traces* foram gerados para cada programa do SPEC: `traces`, `loops` e `loops+traces`. O conjunto `traces` contém todos os *traces* que o StarDBT constrói utilizando a técnica MRET2. O conjunto `loops` contém todos os *loops* que foram identificados nos *traces*, sendo que tais *loops* são transformados em *traces* da maneira apresentada na Figura 3.8. O conjunto `loops+traces` contém todos os *traces* dos dois primeiros conjuntos. Tal etapa corresponde à **Primeira Execução** do mecanismo que foi apresentado na Seção 3.2 (*Offline Profile*), onde é descrito o processo de se avaliar uma otimização no StarDBT.

Uma vez gerados os três conjuntos para os programas do SPEC, cada um deles foi executado 10 vezes para cada conjunto de *traces*, resultando assim em um total de 30 execuções de cada programa. Na execução de cada programa, os *traces* otimizados foram carregados dos arquivos gerados, sendo que o StarDBT foi configurado então para utilizar os *traces* dos arquivos otimizados ao invés de construir *traces*, correspondendo esta parte à **Segunda Execução** do mecanismo na Seção 3.2.

Uma segunda rodada de execução foi realizada, onde novamente os conjuntos `traces`, `loops` e `loops+traces` foram gerados, mas nenhuma otimização foi realizada em qualquer um dos *traces*. Novamente cada programa foi executado 10 vezes para cada conjunto de *traces* num total de 30 execuções de cada programa, onde os *traces* não otimizados eram carregados dos arquivos gerados.

Depois de terminadas as execuções das duas rodadas, tem-se que cada programa do SPEC foi executado um total de 60 vezes, sendo que em 30 execuções utilizou-se *traces* otimizados e em 30 execuções utilizou-se *traces* não otimizados.

Tendo-se os tempos de execução em mãos, computou-se a média dos tempos de cada programa, de forma a se obter a diferença nos tempos de cada um deles, para se avaliar o ganho ou perda de desempenho. Computou-se também o desvio padrão para cada programa e partir deste determinou-se o ganho mínimo e máximo de cada aplicação utilizando-se um intervalo de confiança de 95%. Tal metodologia permite afirmar com certeza em que casos houve ganhos por parte da otimização, uma vez que para que isso

aconteça, tanto o ganho mínimo, quanto o ganho máximo devem ser positivos.

Conjunto Traces

A Tabela 5.5 e o gráfico na Figura 5.25 contém os resultados obtidos para o conjunto `traces`, onde somente os *traces* gerados por MRET2 são otimizados e analisados.

Em 7 programas pode-se afirmar que houve ganho com certeza, pois pela Figura 5.25 observa-se que a barra do gráfico está toda acima do eixo x. O intervalo de confiança de 95% nos permite dizer que em 95% das execuções do programa o valor do ganho obtido estará dentro da barra apresentada no gráfico. Embora pode-se dizer que em 95% das vezes o ganho no tempo de execução estará dentro da barra apresentada no gráfico, não se pode dizer em que ponto da barra o ganho se encontra, isto é, qual a porcentagem do ganho. Com isso um programa cuja barra no gráfico da Figura 5.25 cruza o eixo x, não permite concluir se houve ou não ganho em seu tempo de execução, caso do programa 181.MCF.

Pela Tabela 5.5 observa-se que o ganho mínimo foi de -0,90% e o ganho máximo foi de 2,21%. Embora em 95% das execuções um programa pode ter um ganho de até 2,21% ou uma perda de desempenho de até -0,90%, executando mais lentante; não se sabe ao certo se a maioria das execuções se encontra mais próxima do valor negativo ou do valor positivo, não permitindo assim dizer se para este programa em particular a otimização *hole allocation* trouxe benefícios ou piorou o tempo de execução do programa.

Para programa em que os ganhos máximo e mínimo são negativos, pode-se com certeza afirmar que a otimização piorou o seu tempo de execução, como se observa no programa 164.GZIP, que teve ganho mínimo de -0,18% e máximo de -0,07%. Neste caso, de fato a otimização acabou piorando o tempo de execução do programa.

Para o conjunto `traces`, a otimização piorou o tempo de execução de dois programas, como mostram a Figura 5.25 e a Tabela 5.5. Do total de 26 programas, em 17 não se pode concluir se houve ou não melhora no tempo de execução, pois possuem ganho mínimo negativo e ganho máximo positivo. Nos programas onde houve ganho, este foi de no máximo 2,21%.

Se ao invés do intervalo de confiança, a análise dos resultados se basear pela diferença da média dos tempos de execução, constata-se que houve melhora de desempenho em 11 programas, 4 a mais do que a análise baseada no intervalo de confiança, sendo que o ganho de tempo está limitado à faixa de 1 segundo.

Conjunto Loops

O conjunto `loops` foi construído com a esperança de se obter *traces* maiores e com uma *completion rate* e *execution coverage* maiores que os *traces* do conjunto `traces`. Além

disso, em *traces* maiores, pode-se ter mais oportunidade de se realizar otimizações.

Os resultados obtidos no conjunto `loops` são mostrados na Tabela 5.6 e no gráfico da Figura 5.26. Como se observa na Tabela 5.6, dois programas do SPEC (181.MCF e 186.CRAFTY) apresentaram problema, onde um deles não gera o resultado correto quando executado tendo com entrada o seu conjunto `loops` e o outro programa quebra durante a sua execução ao ter também o conjunto `loops` como entrada. Desta forma não sendo possível obter resultados para os mesmos.

No conjunto `loops`, obteve-se ganho de desempenho em dois programas (252.EON e 177.MESA), sendo que para os demais não se pode concluir se houve ou não ganho de desempenho, uma vez que suas barras de ganho cruzam o eixo x no gráfico.

No caso do programa 252.EON o ganho máximo foi de 0,51% e para o programa 177.MESA o ganho máximo foi de 1,21%, sendo assim menores que o ganho máximo de aproximadamente 2% registrado em dois programas do conjunto `traces`.

A análise da diferença de tempo da média das execuções revela ganho de desempenho em 10 programas, um valor bem diferente do intervalo de confiança, sendo que o programa 191.FMA3D teve uma melhora de aproximadamente 3,7 segundos em seu tempo de execução.

Tal diferença na quantidade de programas que obtém melhora, conforme o método escolhido, se deve ao fato de que em geral a melhora no tempo de execução, que foi obtida nos programas, é relativamente baixa quando comparada com o tempo total de execução de cada programa, fazendo com que a metodologia utilizando intervalos de confiança não seja favorável à maioria dos programas.

Conjunto Loops+Traces

Este conjunto tentou avaliar o possível ganho que todas as formas de *traces* gerados pelo StarDBT poderiam ter se estivessem combinadas em um único arquivo, uma vez que o conjunto `loops` não contém os *traces* que não puderam ser transformados em *loops*.

A Tabela 5.7 e o gráfico na Figura 5.27 mostram os resultados obtidos. Pela Tabela 5.7, observa-se que os programas 186.CRAFTY, 171.SWIM e 173.APPLU apresentaram problema e não foi possível avaliar seu possível desempenho. Os problemas são semelhantes ao que aconteceram no conjunto anterior, onde o programa gera um resultado errado ou quebra durante a execução, indicando talvez que a contrução de *loops* a partir de *traces* não seja um processo completamente consolidado no StarDBT.

Ao contrário do que se poderia imaginar, os ganhos não foram maiores que os obtidos nos conjuntos anteriores. Somente em 4 programas houve ganho de desempenho, sendo que o maior ganho foi de 2,42%, o melhor até aqui. Em 4 programas houve piora no desempenho, dois a mais de que o conjunto `traces`. Para os demais programas, não se

pode concluir se houve ou não ganho de desempenho, uma vez que todos têm a barra de ganho cruzando o eixo x.

A análise da diferença de tempo da média das execuções revela ganho de desempenho em 11 programas ao invés de 4, um valor consideravelmente maior de programas, sendo que em 2 programas, a melhora foi de mais de 1 segundo. Novamente tal diferença na quantidade de programas que são considerados como melhores se deve ao fato de o ganho de tempo obtido ser relativamente baixo em relação ao tempo total de execução de cada programa.

5.4.2 Ambiente Estático

O *Hole Allocation* também foi implementado no GCC 3.3 [1] e utilizado para se gerar *spill code* depois que um algoritmo de coloração de grafos [40] realizou a alocação de registradores. Alguns programas do SPEC foram utilizados para se medir a quantidade de *spill code* que é gerado pelo GCC.

Através da Tabela 5.4 pode-se observar que *hole allocation* gera menos instruções de *load* e *store* do que a técnica [17] empregada pelo GCC para gerar *spill code*. Em 3 programas a redução na quantidade de *load's* inseridos foi de mais de 90% e em 4 programas tal redução ficou acima de 50%. Quando comparadas as instruções de *store* inseridas, a redução foi maior que 80% em 2 programas e superior a 30% nos demais. Por outro lado, a quantidade de instruções *move* que foram inseridas é bem maior que a quantidade de código de rematerialização [22, 66] inserido pelo GCC. Como instruções *move* são baratas computacionalmente, a grande redução na quantidade de *load's* e *store's* compensa a inserção de diversas instruções *move*, pois elas evitam o acesso a memória principal, diminuindo assim a penalidade do *spill code* no desempenho do programa.

	Hole Allocation			**GCC**			**Ganhos**	
	LOAD	**STORE**	**MOVE**	**LOAD**	**STORE**	**REMAT**	**LOAD**	**STORE**
164.GZIP	15	12	43	53	50	0	71,70%	76,00%
175.VPR	18	38	93	227	188	84	92,07%	79,79%
177.MESA	47	109	281	803	506	14	94,15%	78,46%
181.MCF	7	6	11	16	9	4	56,25%	33,33%
186.CRAFTY	106	101	397	264	170	54	59,85%	40,59%
197.PARSER	11	13	70	68	76	3	83,82%	82,89%
254.GAP	19	21	158	244	160	0	92,21%	86,88%

Tabela 5.4: Comparação entre o *spill code* gerado pelo *Hole Allocation* e pelo GCC

5.4.3 Considerações

Os resultados obtidos pela aplicação de *hole allocation* no StarDBT permitem concluir que o conjunto de *traces* utilizado tem uma grande influência no resultado obtido. O conjunto `trace` foi o que apresentou a maior quantidade de programas com ganhos de desempenho, 7 no total, embora o maior ganho percentual tenha sido obtido no programa 177.MESA no conjunto `loops+traces`.

Outro fator a ser observado é o ganho percentual máximo, que para todos os conjuntos nunca chegou a 3%, um valor muito baixo em termos de desempenho. Vários fatores podem ter contribuído para não se obter ganhos maiores. O primeiro, como já dito, é o conjunto de *traces* utilizado. Talvez a técnica MRET2 não encontre *traces* suficientemente bons para serem otimizados.

Um futuro passo na avaliação de *hole allocation* é na otimização de código gerado para **embedded processors**[3] em ambiente estático, onde o ambiente de computação costuma ser mais restrito e não existem tantas otimizações em hardware para os acessos à memória. Desta forma, os ganhos que podem ser obtidos em tais arquiteturas, podem ser consideravelmente maiores do que na arquitetura x86.

Em relação ao ambiente dinâmico, talvez o emprego de *hole allocation* na tradução de IA32 para IA64 possa produzir resultados melhores do que na tradução de IA32 para IA32.

Outro caminho para a melhoria de *hole allocation* é tentar otimizar qualquer tipo de acesso à memória e não somente instruções que sigam o padrão `[esp + xxxxh]`, mas para isso é necessário todo um estudo do problema de desambigüação de memória [44], que atualmente é resolvido inserindo-se código de verificação dentro da aplicação traduzida, tendo a penalidade de tornar a execução mais lenta.

Embora no ambiente dinâmico os resultados não tenham sido muito promissores, os resultados iniciais obtidos no ambiente estático, parecem indicar que *hole allocation* seja melhor para se gerar *spill code* do que otimizar programas em ambientes dinâmicos.

[3] processadores dedicados

Programa	Diferença de Tempo	Intervalo de Confiança	
	Δ da Média (segundos)	Ganho Mínimo (%)	Ganho Máximo (%)
164.GZIP	-0,111	-0,18%	-0,07%
175.VPR	0,884	0,49%	2,10%
176.GCC	0,059	0,07%	0,20%
181.MCF	0,281	-0,90%	2,21%
186.CRAFTY	0,050	-0,09%	0,30%
197.PARSER	0,352	0,21%	0,63%
252.EON	0,151	0,16%	0,55%
253.PERLBMK	-0,471	-1,62%	0,20%
254.GAP	-0,022	-0,35%	0,25%
255.VORTEX	0,125	0,07%	0,42%
256.BZIP2	0,174	0,03%	0,45%
300.TWOLF	-0,001	-0,13%	0,12%
168.WUPWISE	-1,104	-2,66%	-2,01%
171.SWIM	-0,039	-0,09%	0,04%
172.MGRID	0,032	-0,01%	0,07%
173.APPLU	0,090	-0,02%	0,16%
177.MESA	-0,196	-1,46%	0,89%
178.GALGEL	-0,010	-0,66%	0,62%
179.ART	-0,251	-1,80%	0,20%
183.EQUAKE	-0,064	-0,25%	0,00%
187.FACEREC	-0,244	-0,68%	0,00%
188.AMMP	-0,056	-0,12%	0,05%
189.LUCAS	0,112	0,09%	0,16%
191.FMA3D	-0,561	-0,96%	0,01%
200.SIXTRACK	-0,340	-1,26%	0,60%
301.APSI	-0,622	-1,57%	0,71%
TOTAL	-1,782	-0,19%	0,08%

Tabela 5.5: Traces

Programa	Diferença de Tempo	Intervalo de Confiança	
	Δ da Média (segundos)	Ganho Mínimo (%)	Ganho Máximo (%)
164.GZIP	-0,036	-0,28%	0,20%
175.VPR	-0,549	-1,65%	0,08%
176.GCC	-0,012	-0,08%	0,02%
181.MCF	–	–	–
186.CRAFTY	–	–	–
197.PARSER	-0,238	-1,97%	1,43%
252.EON	0,119	0,07%	0,51%
253.PERLBMK	0,091	-0,66%	0,97%
254.GAP	-0,014	-0,20%	0,14%
255.VORTEX	-0,029	-0,25%	0,12%
256.BZIP2	0,467	-0,56%	1,85%
300.TWOLF	-0,898	-2,69%	0,93%
168.WUPWISE	0,218	-0,39%	1,36%
171.SWIM	-2,731	-5,78%	1,78%
172.MGRID	-0,412	-1,09%	0,36%
173.APPLU	-2,946	-7,19%	2,46%
177.MESA	0,433	0,09%	1,21%
178.GALGEL	-0,389	-4,24%	2,57%
179.ART	0,651	-0,09%	3,98%
183.EQUAKE	-0,038	-0,29%	0,15%
187.FACEREC	-0,724	-2,56%	0,59%
188.AMMP	-0,489	0,65%	0,02%
189.LUCAS	1,649	-1,19%	4,79%
191.FMA3D	3,765	-0,39%	6,65%
200.SIXTRACK	0,542	-0,81%	1,85%
301.APSI	0,077	-1,17%	1,28%
TOTAL	-1,493	-0,51%	0,63%

Tabela 5.6: Loops

Programa	Diferença de Tempo	Intervalo de Confiança	
	Δ da Média (segundos)	Ganho Mínimo (%)	Ganho Máximo (%)
164.GZIP	0,007	-0,10%	0,12%
175.VPR	0,771	0,46%	1,76%
176.GCC	0,068	-0,24%	0,55%
181.MCF	-0,316	-1,43%	-0,05%
186.CRAFTY	–	–	–
197.PARSER	0,131	-0,11%	0,42%
252.EON	0,195	-0,22%	1,13%
253.PERLBMK	-0,677	-1,63%	-0,41%
254.GAP	0,009	-0,35%	0,39%
255.VORTEX	-0,016	-1,92%	1,86%
256.BZIP2	-0,532	-2,19%	0,71%
300.TWOLF	1,074	0,31%	1,79%
168.WUPWISE	-0,192	-2,07%	1,25%
171.SWIM	–	–	–
172.MGRID	0,080	-0,04%	0,18%
173.APPLU	–	–	–
177.MESA	1,079	0,68%	2,42%
178.GALGEL	-0,399	-1,60%	-0,10%
179.ART	-0,918	-5,50%	-0,12%
183.EQUAKE	-0,049	-0,28%	0,10%
187.FACEREC	0,361	0,23%	0,77%
188.AMMP	-0,686	-1,54%	0,66%
189.LUCAS	-0,001	-0,07%	0,06%
191.FMA3D	0,240	-0,44%	0,85%
200.SIXTRACK	-0,578	-1,29%	0,19%
301.APSI	-1,403	-3,22%	1,30%
TOTAL	-1,752	-0,33%	0,26%

Tabela 5.7: Traces+Loops

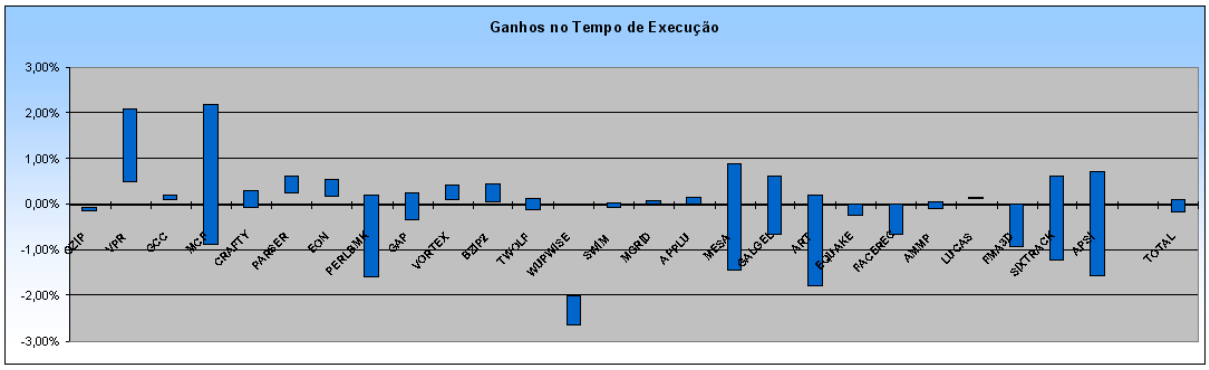


Figura 5.25: Intervalo de confiança do conjunto `traces`

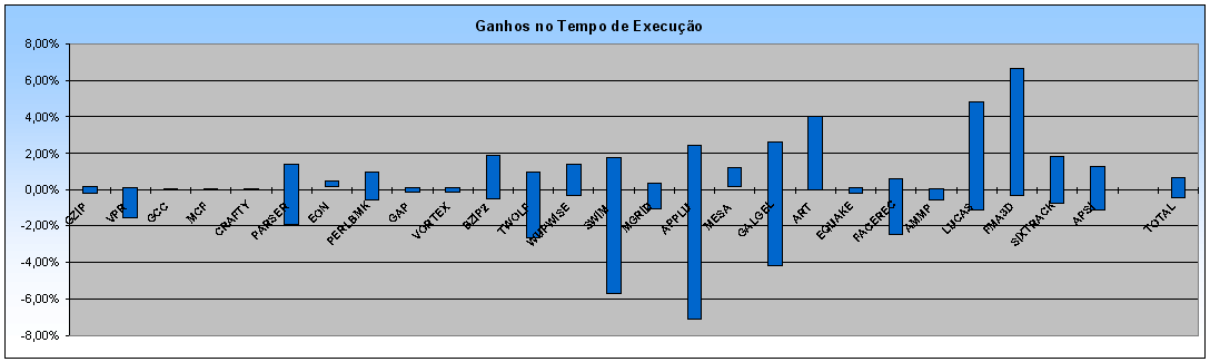


Figura 5.26: Intervalo de confiança do conjunto `loops`

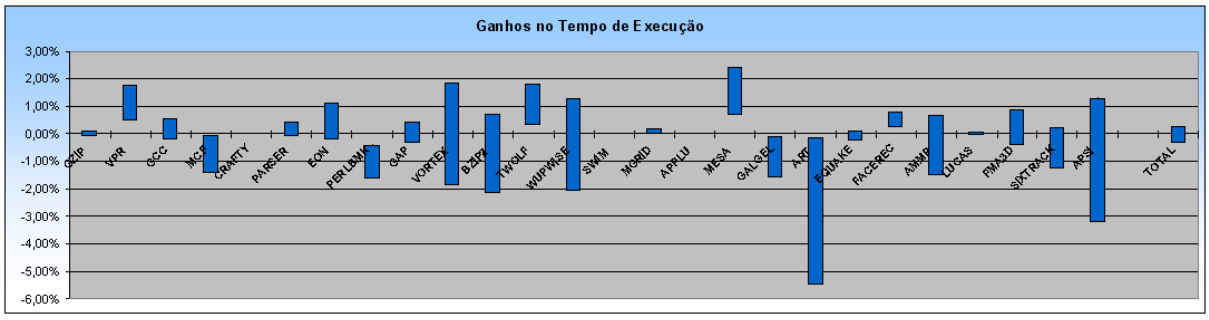


Figura 5.27: Intervalo de confiança do conjunto `loops+traces`

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho teve o objetivo de investigar a otimização de acessos à memória em tradução binária dinâmica, através de uma nova otimização chamada *hole allocation*, que foi implementada no tradutor binário StarDBT.

Os resultados obtidos mostram que a forma como se seleciona e constrói *traces* tem um grande impacto no resultado final da otimização, pois em uma dada configuração, praticamente não houve ganho e vários programas tiveram um desempenho inferior a versão não otimizada.

Como a memória *cache* dos processadores se constitui em uma forma de se compensar o custo de se acessar a memória principal, talvez a realização de otimizações que tentem melhoram o desempenho do programa evitando acessos à memória, como *hole allocation*, seja algo que dificilmente irá produzir altos ganhos de desempenho, como constatou-se neste trabalho com o *hole allocation*.

Em *Cold Code Analysis* os resultados são particularmente interessantes, pois com um pequeno *overhead* pode-se obter uma grande quantidade de informação extra. Com menos de 1% de *overhead* no tempo total de execução, consegue-se obter uma grande quantidade de informação extra. Até onde se sabe, nenhum DBT emprega nada parecido à *cold code analysis* para tentar dar oportunidades de transformação para as otimizações implementadas.

6.1 Contribuições

Como contribuição deste trabalho pode-se destacar a apresentação de uma nova otimização dinâmica que pode ser empregada para se transformar acessos em memória em acessos a registrador.

Outra contribuição consistiu em uma forma sistematizada de se melhorar os resultados de qualquer análise de fluxo de dados realizadas em ambientes de tradução dinâmica que

limitam suas otimizações somente a *traces*.

6.1.1 Publicações

As seguintes publicações estão associadas a este trabalho:

- *Cold Code Analysis,*
  Wesley Attrot and Guido Araújo,
  Proceedings of 1^{st} Workshop on Architectural and Microarchitectural Support for Binary Translation, pp 14-21,
  Junho de 2008,
  Beijing, China.

- *Dynamic Optimization Effects on DBT,*
  Daniel Nicácio, Wesley Attrot, Guido Araújo e Edson Borin,
  Relatório Técnico número 15/2008, Instituto de Computação - UNICAMP, 2008.

- *Hole Allocation in Spill Code Generation,*
  Wesley Attrot e Guido Araújo,
  Relatório Técnico número 27/2008, Instituto de Computação - UNICAMP, 2008.

Embora não estejam ligadas diretamente ao trabalho desenvolvido nesta tese, durante o seu doutorado o autor ainda teve a seguinte produção científica:

- *Efficient bloom filter,*
  Maurício Breternitz Jr., Youfeng Wu, Peter Sassone, Jeffrey P. Rupley, Wesley Attrot and Bryan Black,
  USPTO Applicaton #: 20080147714 - Class: 707102 (USPTO).

- *A Segmented Bloom Filter Algorithm for Efficient Predictors,*
  Maurício Breternitz Jr., Gabriel Loh, Bryan Black, Jeff Rupley, Peter Sassone, Wesley Attrot and Youfeng Wu,
  20^{th} IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 123-130,
  Outubro de 2008,
  Campo Grande, Brazil.

6.2 Trabalhos Futuros

Como continuação do trabalho desta Tese, pretende-se avaliar o desempenho que pode ser obtido por *hole allocation* um ambiente de tradução binária, onde a tradução seja feita de IA32 para IA64, de forma a investigar se a maior quantidade de registradores disponíveis pode criar novas oportunidades de otimização e, se essas otimizações extras, caso existam, podem causar algum ganho de desempenho.

Outra coisa que se pretende avaliar, é a otimização de endereços de memória que não sigam somente a forma `[esp + xxxxh]`. Pretende-se expandir a análise dos endereços de memória para qualquer tipo de acesso, ampliando assim a gama de instruções que podem ser otimizadas.

Em ambientes estáticos, pretende-se comparar *hole allocation* com outras técnicas de geração de *spill code* e também utilizá-la como uma técnica completa de alocação de registradores e realizar as devidas comparações com técnicas bem conhecidas e difundidas no meio acadêmico.

Em relação a *cold code analysis* pretende-se implementar um mecanismo que possa dizer quais blocos básicos já foram analisados e qual resultado foi obtido, para possivelmente diminuir o *overhead* obtido na análise. Se for possível fazer com que a análise de 4 ou 5 níveis de profundidade fique em torno de 1%, pode-se obter quase toda a informação extra que tal análise pode fornecer a um custo extremamente baixo.

Referências Bibliográficas

[1] *GNU Compiler Collection*. http://gcc.gnu.org.

[2] *Intel Corporation*. http://www.intel.com.

[3] *Standard Performance Evaluation Corporation (SPEC)*. http://www.spec.org.

[4] *SYSMARK 2004*. http://www.bapco.com/products/sysmark2004/.

[5] *United States Patent and Trademark office*. http://www.uspto.gov.

[6] Ole Agesen. Design and implementation of pep, a java just-in-time translator. *Theor. Pract. Object Syst.*, 3(2):127–155, 1997.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[8] E.R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *IEEE Computer*, 33(3):40–45, Mar 2000.

[9] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

[10] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, United Kingdom, 1998.

[11] Andrew W. Appel and Lal George. Optimal spilling for cisc machines with few registers. *SIGPLAN Not.*, 36(5):243–253, 2001.

[12] Wesley Attrot and Guido Araújo. Cold Code Analysis. In 1^{st} *Workshop on Architectural and Microarchitectural Support for Binary Translation*, pages 14–21, June 2008.

[13] Wesley Attrot and Guido Araújo. Hole Allocation in Spill Code Generation. Technical Report IC-08-27, Institute of Computing, University of Campinas, september 2008.

[14] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[15] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[16] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, and Yun Wang. IA-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *In 36th International Symposium on Microarchitecture*, pages 191–201, 2003.

[17] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 287–295, New York, NY, USA, 1997. ACM Press.

[18] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263, New York, NY, USA, 1989. ACM.

[19] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Edson Borin and Youfeng Wu. Characterization of dynamic binary translator overhead. In 1^{th} *Workshop on Architectural and Microarchitectural Support for Binary Translation*, pages 4–13, June 2008.

[21] Preston Briggs. Register allocation via graph coloring. Technical Report CRPC-TR92218, Ctr. for Research on Parallel Computation, Rice Univ., April 1992.

[22] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Not.*, 27(7):311–321, 1992.

[23] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[24] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Chaitin, Gregory, March Auslander, Ashok Chandra, John Cocke, Martin Hopkins, and Peter Markstein. Register Allocations Via Coloring. In *Computer Languages*, volume 6, pages 47–57, 1981.

[26] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, New York, NY, USA, 1982. ACM Press.

[27] P. P. Chang and W. W. Hwu. Trace selection for compiling large C application programs to microcode. In *MICRO 21: Proceedings of the 21st annual workshop on Microprogramming and microarchitecture*, pages 21–29, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[28] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag.

[29] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 174–187, London, UK, 1998. Springer-Verlag.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[31] Maxwell Monteiro Andrade de Souza. Tradução binária dinâmica orientada a mapeamento de instruções. Master's thesis, Institute of Computing, University of Campinas, March 10 2008.

[32] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing/spl trade/ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 15–24, 23-26 March 2003.

[33] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[34] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[35] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, 2001.

[36] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[37] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, 1981.

[38] Changqing Fu and Kent Wilken. A faster optimal register allocator. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 245–256, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[39] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. *Technical Report No. 06-16, Donald Bren School of Information and Computer Science*, November 2006.

[40] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[41] Richard Gerber. *The Software Optimization Cookbook: High-performance Recipes for the Intel Architecture*. Intel-Press, USA, 2002.

[42] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8):929–965, 1996.

[43] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. Practical and accurate low-level pointer analysis. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[44] Bolei Guo, Youfeng Wu, Cheng Wang, Matthew J. Bridges, Guilherme Ottoni, Neil Vachharajani, Jonathan Chang, and David I. August. Selective runtime memory disambiguation in a dynamic binary translator. In Alan Mycroft and Andreas Zeller, editors, *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2006.

[45] Yajun Ha, Bingfeng Mei, Patrick Schaumont, Serge Vernalde, Rudy Lauwereins, and Hugo De Man. Development of a design framework for platform-independent networked reconfiguration of software and hardware. In *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pages 264–274, London, UK, 2001. Springer-Verlag.

[46] Timothy J. Harvey. Reducing the impact of spill code. Technical Report TR98-319, 24 1998.

[47] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 176–191, London, UK, 1992. Springer-Verlag.

[48] David Hiniker, Kim Hazelwood, and Michael 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, Washington, DC, USA, 2005. IEEE Computer Society.

[49] J.D. Hiser, N. Kumar, Min Zhao, Shukang Zhou, B.R. Childers, J.W. Davidson, and M.L. Soffa. Techniques and tools for dynamic optimization. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, 25-29 April 2006.

[50] R. Hookway. Digital FX!32 running 32-bit x86 applications on Alpha NT. *Compcon '97. Proceedings, IEEE*, pages 37–42, 23-26 Feb 1997.

[51] Priyadarshan Kolte and Mary Jean Harrold. Load/store range analysis for global register allocation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 268–277, New York, NY, USA, 1993. ACM.

[52] Akira Koseki, Hideaki Komatsu, and Toshio Nakatani. Spill code minimization by spill code motion. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 125, Washington, DC, USA, 2003. IEEE Computer Society.

[53] A. Kumar. The HP PA-8000 RISC CPU: a high performance out-of-order processor. In *Proceedings of Hot Chips VIII*, Palo Alto, CA.

[54] Steven M. Kurlander and Charles N. Fischer. Zero-cost range splitting. *SIGPLAN Not.*, 29(6):257–265, 1994.

[55] Bich C. Le. An out-of-order execution technique for runtime binary translators. *SIGOPS Oper. Syst. Rev.*, 32(5):151–158, 1998.

[56] Jiwei Lu, Howard Chen, Pen chung Yew, and Wei chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:2004, 2004.

[57] John Lu and Keith D. Cooper. Register promotion in C programs. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 1997. ACM.

[58] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1997.

[59] Takuya Nakaike, Tatsushi Inagaki, Hideaki Komatsu, and Toshio Nakatani. Profile-based global live-range splitting. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 216–227, New York, NY, USA, 2006. ACM.

[60] Daniel Nicácio, Wesley Attrot, Guido Araújo, and Edson Borin. Dynamic optimization effects on DBT. Technical Report IC-08-15, Institute of Computing, University of Campinas, july 2008.

[61] Trek Palmer, Dino Dai, and Zovi Darko Stefanovic. SIND: A framework for binary translation. Technical Report Tech. Rep. TR-CS-2001-38, Department of Computer Science, University of New Mexico, December 2001.

[62] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. In *CATS '07: Proceedings of the thirteenth Australasian symposium on Theory of computing*, pages 3–3, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[63] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[64] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[65] Matthew Postiff, David Greene, and Trevor Mudge. The store-load address table and speculative register promotion. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 235–244, New York, NY, USA, 2000. ACM.

[66] Mukta Punjani. Rematerialization register in GCC. *GCC Developer's Summit 2004*, pages 131–140.

[67] Luiz C. V. Dos Santos. A method to control compensation code during global scheduling. *Workshop on Circuits, Systems and Signal Processing*, 1997.

[68] A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on SSA form. *SIGPLAN Not.*, 33(5):15–25, 1998.

[69] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288, New York, NY, USA, 2004. ACM.

[70] Omri Traub, Glenn H. Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–151, 1998.

[71] David Ung and Cristina Cifuentes. Optimising hot paths in a dynamic binary translator. In *In Workshop on Binary Translation*, page 2001, 2000.

[72] Cheng Wang, Shiliang Hu, Ho seop Kim, Sreekuma R. Nair, Mauricio Breternitz Jr., Zhiwei Ying, and Youfeng Wu. StarDBT: An efficient multi-platform dynamic binary translation system. *Lecture Notes in Computer Science*, 4697/2007:4–15, 2007.

[73] Christian Wimmer and Hanspeter Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 132–141, New York, NY, USA, 2005. ACM.