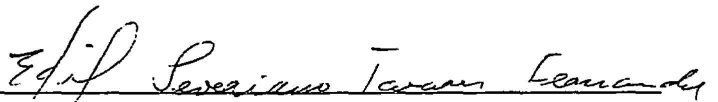
UMA NOVA TÉCNICA DE APRENDIZADO PARA OTIMIZAR O DESEMPENHO DE CACHES DE INSTRUÇÕES

Carlos Augusto Garcia Assis

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.



Prof. Edil Severiano Tavares Fernandes, Ph.D.

Prof. Cláudio Luís de Amorim, Ph.D.

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Siang Wun Song, Ph.D.

Prof. Alberto Ferreira de Souza, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
MAIO DE 2005

Assis, Carlos Augusto Garcia

Uma nova técnica de aprendizado para otimizar o desempenho de *caches* de instruções [Rio de Janeiro] 2005

XII, 127 p. 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2005)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Redes Bayesianas
2. Nova Técnica para Otimizar o Sistema de Memória de Computadores
3. *Cache* de Instruções

I. COPPE/UFRJ II. Título (série)

Agradecimento

Este trabalho é o resultado de uma teia de envolvimentos: dedicação, estudo, planejamento e objetivos definidos na procura do saber. Para realizá-lo, muitos caminhos foram percorridos e diversas pessoas foram envolvidas, participando de diferenciadas formas. Finalizando esta etapa, quero agradecer e expressar minha profunda gratidão.

Inicialmente, agradeço aos meus pais pela vida. Às minhas irmãs, Mônica e Susana, pelo compartilhamento de uma infância alegre, despreocupada e livre, base da minha formação ética, democrática e responsável.

Sentimento profundo, manifesto por Valmir Barbosa, professor e orientador no mestrado e no doutorado. Juntos, percorremos um longo caminho. Difícil expressar os limites da admiração pelo professor e a gratidão pelo incentivador. Sempre presente, dando-me apoio e estímulo nos momentos decisivos desta trajetória.

Para Edil Fernandes, professor e orientador no Doutorado, expresso o grande significado do seu saber que, como poucos, sabe compartilhar e ajudar nos momentos difíceis. Iluminou muitos momentos difíceis, sempre atuante, dando-me apoio e corrigindo meus erros. Seu humor tornou nossas reuniões mais prazerosas.

Agradeço aos professores Claudio Amorim e Alberto Ferreira que participaram do exame de qualificação e contribuiram com sugestões para a finalização da tese.

Às instituições educacionais onde convivi em diferentes situações, de aluno a professor como o Colégio Estadual Luis Reid, Colégio Princesa Isabel, à Universidade Federal Fluminense, Pontifícia Universidade Católica, Universidade Federal do Rio de Janeiro e Universidade Estácio de Sá.

Aos meus alunos da PUC e da Universidade Estácio de Sá que me apoiaram e incentivaram na construção de uma relação profícua, baseada na troca de experiências e do saber, meu eterno carinho, que estendo para todos os professores e funcionários da COPPE Sistemas.

A minha futura esposa Raquel Terra Agostinho que acompanha no dia a dia, todo esforço e dedicação. A todos que direta ou indiretamente contribuíram para a construção da minha tese, produto de esforço, determinação e dedicação, compartilho a grande alegria na realização deste sonho.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciência (D.Sc.).

UMA NOVA TÉCNICA DE APRENDIZADO PARA OTIMIZAR O DESEMPENHO DE CACHES DE INSTRUÇÕES

Carlos Augusto Garcia Assis

Maio/2005

Orientadores : Valmir Carneiro Barbosa
Edil Severiano Tavares Fernandes

Programa: Engenharia de Sistemas e Computação

A velocidade do processador sempre superou a da memória. Essa diferença de velocidade (denominada *memory-gap problem*) tem crescido paulatinamente, resultando em sistemas desbalanceados. Vários trabalhos, abrangendo o *hardware* e o *software*, têm sido desenvolvidos para abrandar os efeitos negativos provocados por esta diferença de velocidade. Vários níveis na hierarquia de memória, busca antecipada de instruções e dados (*instruction and data prefetch*) e *trace cache* são clássicos exemplos de mecanismos que foram incorporados no *hardware* para reduzir as penalidades impostas pelo *memory-gap problem*. Quanto ao *software*, somente na última década (1990) é que apareceram iniciativas para atacar o problema e o trabalho de Pettis&Hansen é pioneiro: instrumentando o programa de aplicação, os autores determinam os trechos freqüentemente executados. Em seguida, os trechos do programa são reorganizados segundo as freqüências obtidas e dessa forma, falhas na memória *cache* e tempo de execução do programa reorganizado é menor que o original.

Neste trabalho introduzimos a primeira abordagem probabilística ao problema. Através do uso de uma rede Bayesiana que modela a história de todas as entradas sobre as quais o programa já foi executado, criamos um modelo dinâmico que provê uma nova ordenação de blocos básicos para cada nova entrada. A nova técnica baseia-se no tempo de execução do programa sobre as diversas entradas, e por isso não depende diretamente dos muitos parâmetros arquiteturais. Resultados experimentais sobre o SPEC2000 revelam ganhos de até 15% sobre outras técnicas, incluindo a de Pettis e Hansen e também a otimização O3 do gcc.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the of Doctor of Science (D.Sc.).

A NEW LEARNING TECHNIQUE TO OPTIMIZE THE PERFORMANCE OF INSTRUCTION CACHES

Carlos Augusto Garcia Assis

May/2005

Advisors: Valmir Carneiro Barbosa
Edil Severiano Tavares Fernandes

Department: Systems Engineering and Computer Science

Processors are faster than their corresponding memory systems. This speed difference, called the memory-gap problem, is increasing and leads to imbalanced computer systems. In order to mitigate the negative effects provoked by the memory gap, many research projects involving both hardware and software have been developed. Classical examples of hardware mechanisms to tackle the problem are the inclusion of: several levels in the memory hierarchy; instruction and data prefetch; trace cache; and so on. Only in the last decade (1990) were the first software attempts undertaken to optimize the code. The Pettis&Hansen work is pioneer: by instrumenting application programs they determined the program profiles. Next, a reordering technique developed by the authors was applied and a new and more efficient code layout was generated.

Our work differs from the previous ones because it introduces a probabilistic approach to handle the problem. We employ a Bayesian network to model and accumulate previous program profiles (i.e., those produced by distinct input files). The model is dynamic and provides a new code layout after processing each input file, and it is this new (and more efficient) layout that will be used by the next program execution. Since the technique is based on the execution time of each input, it is not bound by architectural parameters. Performance gains reaching up to 15% over other techniques have been observed.

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 3

1.3 Metodologia 5

1.4 Organização do texto 8

2 Conceitos básicos **9**

2.1 Hierarquia de memória 9

2.2 Fases de execução 13

2.3 Características da arquitetura Intel 21

3 Trabalhos relacionados **26**

3.1 Pettis e Hansen 28

4 Modelagem **31**

4.1 Simulated annealing 31

4.2 Redes Bayesianas 33

4.3 Problema a ser resolvido 35

4.4 Método proposto 37

4.4.1 Instrumentação e coleta 37

4.4.2 Combinação 39

4.4.3 Otimização 46

4.5 Modelo histórico 47

5 Experimentos e resultados **53**

5.1 Características de hardware e software 53

5.2 Evolução dos programas 56

5.2.1 Programa gcc 58

5.2.2 Programa gzip 59
5.3 Resultados gerais 60
5.3.1 Conjuntos de entradas reference, train e test 62
5.3.2 SPEC2000 65
5.3.3 SPEC95 66
5.3.4 Algoritmos genéticos, equação linear e algoritmo recursivo . . 66
5.4 Análise de baixo nível 71
5.4.1 Resultados PAPI – programas SPEC2000 71

6 Conclusão 81

A Outros resultados 84
A.1 Evolução dos programas 84
A.1.1 Programa bzip2 84
A.1.2 Programa crafty 84
A.1.3 Programa gap 86
A.1.4 Programa mcf 88
A.1.5 Programa parser 90
A.1.6 Programa twolf 91
A.1.7 Programa vortex 92
A.1.8 Programa vpr 94
A.2 Resultados PAPI – programas SPEC2000 96
A.2.1 Programa bzip2 96
A.2.2 Programa crafty 98
A.2.3 Programa gap 101
A.2.4 Programa mcf 102
A.2.5 Programa parser 104
A.2.6 Programa twolf 104
A.2.7 Programa vortex 106
A.2.8 Programa vpr 107

Bibliografia 117

Lista de Figuras

1.1 Ambientes de execução . . . 4
1.2 Fase de instrumentação e coleta, combinação e otimização . . . 7

2.1 Hierarquia de memória . . . 10
2.2 Fases de execução . . . 14
2.3 Fases do compilador . . . 15
2.4 Quatro módulos objetos . . . 18
2.5 a) Programa executável antes da linkedição e relocação e b) Programa executável depois da linkedição e relocação . . . 19
2.6 Registradores do pentium . . . 23
2.7 Formato das instruções . . . 24
2.8 Modos de endereçamentos . . . 25

3.1 Grafo ponderado . . . 30

4.1 Rede *Bayesiana* . . . 33
4.2 Algoritmo de simulação estocástica . . . 35
4.3 Alocação do programa na memória *cache* antes da reordenação dos blocos básicos . . . 36
4.4 Alocação do programa na memória *cache* depois da reordenação dos blocos básicos . . . 37
4.5 Fase 1 - instrumentação e coleta . . . 38
4.6 Formato do arquivo de perfil . . . 39
4.7 Fase 2 - combinação . . . 40
4.8 Retorno da função . . . 41
4.9 Arquivo sloop . . . 42
4.10 Árvore binária de busca AVL . . . 43
4.11 Eliminação de loops no grafo . . . 44

4.12 Probabilidades iniciais da rede *Bayesiana* 45
4.13 Generalizando as probabilidades iniciais 46
4.14 Tabela do cálculo do Noisy-Or 47
4.15 Cálculo do Noisy-Or 48
4.16 Fase 3 - otimização 49
4.17 Caminho mais provável de ser executado 50
4.18 Gráfico do $\alpha = \left(1 + e^{T-t}(1-\alpha_0)/\alpha_0\right)^{-1}$ com $T = 6$ para $\alpha_0 = 0.2, 0.5, 0.8$. 51

5.1 Performance da evolução do programa gcc tabela 5.8. 60
5.2 Performance da evolução do programa gzip tabela 5.10. 62
5.3 Programas do SPEC2000 conjunto de entrada reference 63
5.4 Programas do SPEC2000 conjunto de entrada train 64
5.5 Programas do SPEC2000 conjunto de entrada test 65
5.6 Performance dos programas do SPEC2000 66
5.7 Performance dos programas extras 67
5.8 Redução nas falhas na *cache* L1 79
5.9 Redução nas falhas na *cache* L2 79
5.10 Redução nos desvios tomados 80
5.11 Redução nos desvios mal previstos 80

A.1 Performance da evolução do programa bzip2 tabela A.2. 86
A.2 Performance da evolução do programa crafty tabela A.4. 88
A.3 Performance da evolução do programa gap tabela A.6. 90
A.4 Performance da evolução do programa mcf tabela A.8. 91
A.5 Performance da evolução do programa parser tabela A.10. 93
A.6 Performance da evolução do programa twolf tabela A.12. 94
A.7 Performance da evolução do programa vortex tabela A.14. 96
A.8 Performance da evolução do programa vpr tabela A.16. 98

Lista de Tabelas

2.1 Tamanho e endereço inicial dos módulos 18

5.1 Descrição dos programas do SPEC2000 54
5.2 Entrada SPEC2000 55
5.3 Análise dos programas 55
5.4 Tempos (seg.) do programa gzip e entrada ref 56
5.5 Tempos (seg.) do programa vpr e entrada test 56
5.6 Tempos (seg.) do programa vortex e entrada test, train e ref 57
5.7 Tempo (seg.) de execução gcc 59
5.8 Evolução (seg.) do gcc 59
5.9 Tempo (seg.) de execução gzip 61
5.10 Evolução (seg.) do gzip 68
5.11 Tempo (seg.) de execução do conjunto de entrada reference 69
5.12 Tempo (seg.) de execução do conjunto de entrada train 69
5.13 Tempo (seg.) de execução do conjunto de entrada test 69
5.14 Tempo de execução (seg.) SPEC2000 70
5.15 Tempo de execução (seg.) dos programas extras 70
5.16 Redução nas falhas na *cache* L1 gcc 73
5.17 Redução nas falhas na *cache* L2 gcc 73
5.18 Redução nos desvios tomados gcc 73
5.19 Redução nos desvios mal previstos gcc 74
5.20 Redução nas falhas na *cache* L1 gzip 75
5.21 Redução nas falhas na *cache* L2 gzip 76
5.22 Redução nos desvios tomados gzip 77
5.23 Redução nos desvios mal previstos gzip 78

A.1 Tempo (seg.) de execução bzip2 85
A.2 Evolução (seg.) do bzip2 85

A.3 Tempo (seg.) de execução crafty 86
A.4 Evolução (seg.) do crafty 87
A.5 Tempo (seg.) de execução gap 87
A.6 Evolução (seg.) do gap 89
A.7 Tempo (seg.) de execução mcf 89
A.8 Evolução (seg.) do mcf 89
A.9 Tempo (seg.) de execução parser 91
A.10 Evolução (seg.) do parser 92
A.11 Tempo (seg.) de execução twolf 92
A.12 Evolução (seg.) do twolf 93
A.13 Tempo (seg.) de execução vortex 95
A.14 Evolução do vortex 95
A.15 Tempo (seg.) de execução vpr 97
A.16 Evolução (seg.) do vpr 97
A.17 Redução nas falhas na *cache* L1 bzip2 99
A.18 Redução nas falhas na *cache* L2 bzip2 99
A.19 Redução nos desvios tomados bzip2 99
A.20 Redução nos desvios mal previstos bzip2 100
A.21 Redução nas falhas na *cache* L1 crafty 100
A.22 Redução nas falhas na *cache* L2 crafty 100
A.23 Redução nos desvios tomados crafty 101
A.24 Redução nos desvios mal previstos crafty 101
A.25 Redução nas falha na *cache* L1 gap 102
A.26 Redução nas falhas na *cache* L2 gap 102
A.27 Redução nos desvios tomados gap 102
A.28 Redução nos desvios mal previstos gap 103
A.29 Redução nas falhas na *cache* L1 mcf 103
A.30 Redução nas falhas na *cache* L2 mcf 103
A.31 Redução nos desvios tomados mcf 104
A.32 Redução nos desvios mal previstos mcf 104
A.33 Redução nas falhas na *cache* L1 parser 105
A.34 Redução nas falhas na *cache* L2 parser 105
A.35 Redução nos desvios tomados parser 105
A.36 Redução nos desvios mal previstos parser 106

A.37 Redução nas falhas na *cache* L1 twolf 106
A.38 Redução nas falhas na *cache* L2 twolf 106
A.39 Redução nos desvios tomados twolf 107
A.40 Redução nos desvios mal previstos twolf 107
A.41 Redução nas falhas na *cache* L1 vortex 108
A.42 Redução nas falhas na *cache* L2 vortex 108
A.43 Redução nos desvios tomados vortex 108
A.44 Redução nos desvios mal previstos vortex 109
A.45 Redução nas falhas na *cache* L1 vpr 109
A.46 Redução nas falhas na *cache* L2 vpr 110
A.47 Redução nos desvios tomados vpr 110
A.48 Redução nos desvios mal previstos vpr 110

Capítulo 1

Introdução

Este capítulo apresenta a motivação da tese, a definição do problema tratado, os principais objetivos pretendidos e a organização do texto.

1.1 Motivação

Um dos problemas atuais de maior relevância no projeto de computadores refere-se à diferença de velocidade entre os processadores e seus sistemas de memória. Enquanto a velocidade do primeiro cresce cerca de 60% por ano, a do segundo cresce apenas 10% [73]. Esse descompasso prejudica sensivelmente o desempenho do sistema como um todo, e tem estimulado diversos projetos de pesquisa para reduzir o efeito no desempenho provocado por essa diferença. O que mais preocupa é o fato de o problema estar se agravando a cada ano. Na prática, isso significa que um processador deve esperar vários ciclos de *clock* até que a memória atenda uma requisição. A presente pesquisa visa diminuir o impacto desta discrepância [7, 17].

A propriedade relativa a um programa, que pode ser explorada regularmente, é o princípio da localidade [19], onde programas tendem a reutilizar dados e instruções que foram usados recentemente. Uma implicação do princípio da localidade é que pode predizer com razoável precisão quais as instruções e dados que um programa utilizará no futuro próximo, com base em seus acessos no passado recente. Dois tipos diferentes de localidade podem ser observados: temporal e espacial. A localidade temporal diz que posições de memória que foram acessadas há pouco tempo têm maior probabilidade de serem acessadas no futuro próximo enquanto a localidade espacial diz que posições de memória que estão próximas uma das outras tendem a ser referenciadas em um espaço de tempo próximo.

Em muitos programas, uma pequena porcentagem do código total é responsável

por grande parte do tempo gasto na execução. Vários pesquisadores afirmam que a maioria dos programas gasta 90% do seu tempo de execução em 10% de seu código. Isto ocorre também em outras áreas totalmente não relacionadas, por exemplo, a distribuição de terras, distribuição de renda, produção literária e científica e é conhecido como efeito Mateus [26].

Compiladores modernos são eficientes na otimização do código, porém as análises e otimizações que eles realizam são geralmente restritas às funções individuais. Mesmo aqueles que efetuam otimizações interprocedurais não possuem acesso às rotinas de bibliotecas pré-compiladas. Conseqüentemente, há um espaço considerável para a melhoria do desempenho do código gerado por compiladores convencionais, mesmo com o elevado grau de otimização do código que geram. Uma solução seria realizar uma fase adicional de otimização após os módulos objetos terem sido ligados. Neste estágio, todo o programa (agora disponível como código de máquina) e todo o código de biblioteca estaticamente ligado estão disponíveis para inspeção e modificação.

Um otimizador guiado pelo perfil (*profile*) de execução do programa poderia ser utilizado posteriormente para ajudar a otimizar o programa. Através do estabelecimento de um perfil do comportamento do programa em relação ao seu tempo de execução, usando-se amostras significativas de dados, será possível identificar as partes do código mais executadas, viabilizando assim a geração de um código mais eficiente [1, 2, 58, 66].

Os primeiros esforços relativos à otimização do *layout* do código concentraram em sistemas de memória virtual [22, 31, 40] e visavam produzir *layouts* de código que pudessem reduzir o número de páginas faltosas no tempo de execução. O surgimento do *translation lookside buffer* (TLB) e a introdução de vários níveis de *cache* em processadores recentes deslocaram o contexto das pesquisas para técnicas mais eficientes.

Algumas contribuições mais recentes neste contexto incluem algumas que visam à redução da taxa de erros da *cache* [38, 52, 60], ou a redução da "poluição" da *cache* e tráfego do barramento [30], para reduzir o tempo de execução do programa. A maioria dessas contribuições consiste em instrumentar e gerar um perfil para a otimização do *layout* do código [2, 58, 63].

Os resultados recentes obtidos por vários pesquisadores indicam que essa otimização de código executável na etapa de *linkedição* produz uma melhoria no de-

sempenho, mesmo se o programa já tiver sido sujeito a uma extensiva otimização em tempo de compilação. A maioria dessas investigações foi realizada com processadores da arquitetura RISC.

1.2 Objetivos

Estamos interessados em investigar a viabilidade da construção de um modelo probabilístico dinâmico de um programa. Especificamente, enquanto o programa executa com uma seqüência de entradas variadas, nosso modelo probabilístico é dinamicamente ajustado de modo que sempre reflita o histórico do comportamento do programa e possa ser usado para rearranjar o *layout* do código do programa para uma execução subseqüente. Esse modelo é construído através do perfil de execução do programa produzido por cada entrada de dados. O que estamos introduzindo é uma técnica baseada em um modelo que persiste ao longo de execuções do programa, levando em consideração o perfil da execução anterior.

Para ilustrar como nosso esquema de otimização funciona, vamos considerar dois ambientes: um ambiente de teste e um ambiente de produção (vide Figura 1.1). O programa do usuário sempre vai ser executado nesses dois ambientes. Na primeira execução, no ambiente de produção, o programa do usuário é o original. No ambiente de teste, será gerado o arquivo instrumentado (P.instrumentado), que quando executado vai gerar o arquivo de perfil (Arq.perfil). Na primeira execução se tornará o arquivo histórico (Arq.h1), nas demais execuções o arquivo de perfil será combinado através de uma média geométrica ponderada com o arquivo histórico (Arq.h2, Arq.h3). Esse arquivo histórico combinado servirá de entrada para um programa de otimização (P.otimização) gerar o arquivo otimizado (P.opt1, P.opt2, P.opt3).

Após a primeira execução, será realizada uma cópia do programa otimizado (gerado no ambiente de teste, P.opt1) para o ambiente de produção do usuário. Como pode ser visto na Figura 1.1, não necessariamente todas as entradas (I_1, I_2 I_3 etc...) executadas no ambiente de produção vão ter que ser instrumentadas e otimizadas no ambiente de teste, e sim um subconjunto de entradas (I_1, I_4 I_6), isto acontece porque o usuário pode desejar executar uma outra entrada e o processo de otimização, no ambiente de teste, ainda não ter terminado.

O principal objetivo dessa tese é reduzir o tempo de execução de um programa. O

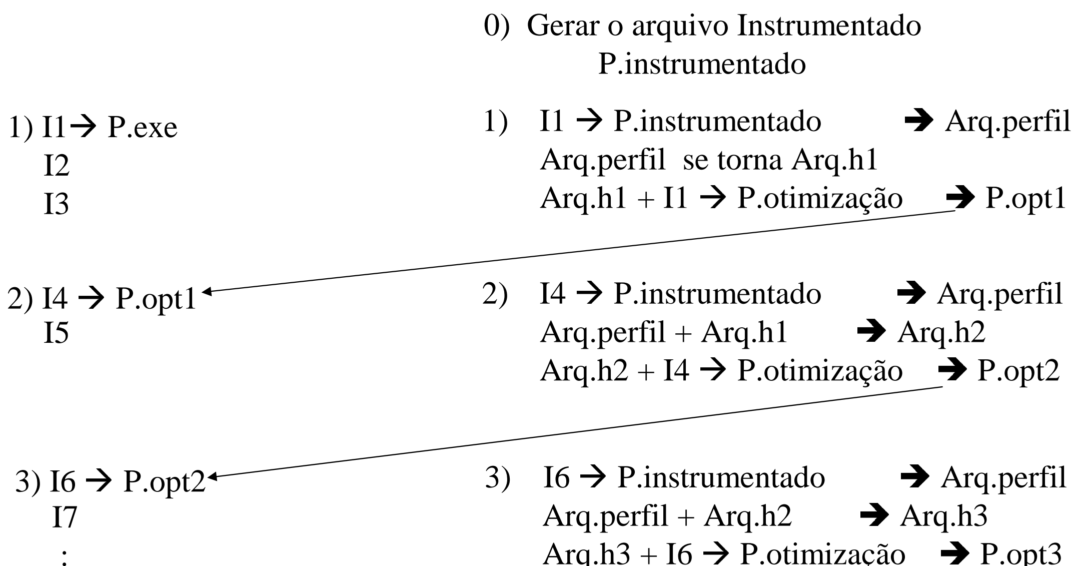


Figura 1.1: Ambientes de execução

foco do trabalho é fazer com que, a cada nova execução do programa, seja gerado um perfil com informações sobre o comportamento atualizado do programa. Este perfil será sempre atualizado com as informações da execução corrente. Serão armazenadas as informações de várias execuções em um arquivo histórico, tornando cada vez mais eficiente. A otimização será auxiliada com informações deste arquivo histórico.

Programas são formados por blocos básicos, sendo cada bloco básico uma seqüência linear de instruções com único ponto de entrada (no início do bloco) e somente uma saída no fim do bloco. Para otimizar o programa, se um bloco básico chama freqüentemente um outro, esses dois blocos básicos precisam ficar perto um do outro, aumentando a chance dos dois blocos básicos ficarem na mesma página. Conseqüentemente, teremos um número menor de falhas na memória *cache* (*cache misses*), e um reduzido número de desvios tomados e desvios mal previstos.

Utilizamos as técnicas redes *Bayesianas* e *Simulated Annealing* para achar o *hot path* (caminho mais provável de ser executado) do programa [4]. Utilizando essas técnicas foi possível realizar uma analise global do comportamento do programa e,

a cada nova execução, um novo *hot path* é encontrado. O *hot path* permite guiar o posicionamento dos blocos básicos na memória.

Várias características da arquitetura Intel IA32 (x86) tornam complexa a tarefa de instrumentação e otimização em tempo de *linkedição*. Entre elas podem ser destacadas: o grande número de tipos de instruções e de formas de endereçamentos do x86, que aumentam a complexidade da análise de um programa; instruções de tamanho variável, as quais dificultam o *disassembler* do código da máquina; e um reduzido número de registradores disponíveis no x86, que limitam algumas otimizações.

Nosso arquivo de perfil é gerado pela ferramenta PLTO (*Portable Link-Time Optimizer*) [18, 63] desenvolvida por Saumya Debray na University of Arizona. Essa feramenta apresenta algumas incompatibilidades com o projeto proposto pela tese. Já que o PLTO seria usado, então ele teve que ser modificado para ficar compatível e servir como entrada ao programa de otimização.

Nossa pesquisa está relacionada com otimização estática, isto é, o código que vai ser otimizado foi estaticamente compilado com suas bibliotecas, e as otimizações serão realizadas após a execução do código; a otimização dinâmica (em tempo de execução) não será considerada. Nossa meta é cumprir uma otimização completa em todo o programa, incluindo as bibliotecas ligadas estaticamente.

1.3 Metodologia

A seguir, apresentamos uma descrição em alto nível de nossa estratégia. Seja P o código binário de um programa de uma arquitetura qualquer, e dois arquivos de entrada I e I' de P. Suponha que temos meios de instrumentar P tal que, aplicando I em P, é gerado um modelo abstrato dessa execução particular que pode ser usado para estimar os blocos básicos de P que são prováveis de serem processados em futuras execuções. Se este for o caso, nós podemos reordenar os blocos básicos de P de tal modo que, quando entrarmos com I' para execução, os blocos básicos que irão ocupar os níveis mais altos na hierarquia de memória serão alguns daqueles anteriormente estimados.

Essa estimativa, de posicionamento dos blocos básicos, foi baseada unicamente na entrada I, então ela pode ser ineficiente com a nova entrada I'. No entanto a execução de P com I' pode ser instrumentada, e o módulo resultante desta segunda execução pode ser combinado com o anterior, podendo gerar um modelo geral e

menos dependente da execução, a ser usado em uma execução subseqüente de P.

Nossa premissa central neste trabalho é que esses modelos de execução de P podem realmente ser obtidos e usados com sucesso para um fluxo de entrada de P. O modelo que construímos a partir de uma determinada execução de P leva a uma rede *Bayesiana*. Combinando essa rede *Bayesiana* com outra que grave todo o "histórico" das execuções prévias de P e depois, resolvendo a rede *Bayesiana* resultante da combinação, temos a capacidade de previsão, o que permite execuções cada vez mais eficientes. Vamos discutir os detalhes de nosso modelo de construção e atualização do método no Capítulo 4.

Uma dificuldade imediata com essa abordagem é que ela pode levar a um esforço considerável para refinar o modelo a ser obtido de uma execução: não só porque instrumentar P o torna significativamente mais lento, mas também porque preparar uma rede *Bayesiana* e resolvê-la pode consumir algum tempo. Toda a estratégia então poderia ser inapropriada para um ambiente de execução "real", já que todo o ganho que poderia resultar seria totalmente ofuscado pelo custo para obtê-lo. Mas vemos uma dinâmica diferente para a bem sucedida aplicação de nossa abordagem, uma que só o aplica a um subfluxo do fluxo de entradas de P, de modo que versões rearranjadas de P somente estarão disponíveis com certa freqüência, ao contrário de estarem disponíveis após cada nova entrada processada, vide Figura 1.1.

Os problemas que encontramos e as correspondentes soluções do método para atingir os objetivos da tese podem ser resumidos em 3 fases: instrumentação do programa e coleta do perfil de execução, combinação das probabilidades e otimização do programa (vide Figura 1.2).

Instrumentação e coleta

A primeira fase é a instrumentação do código executável, isto é, um programa fonte escrito numa linguagem de alto nível (linguagem C na implementação corrente) é compilado pelo gcc com o parâmetro de relocação. Um arquivo executável que inclui dados de relocação e tabela de símbolos é produzido. Este arquivo relocável é a entrada de um programa de instrumentação (PLTO) que insere instruções no código binário, isto é, o programa de instrumentação altera o programa relocável de modo que o novo código (programa instrumentado), quando executado, gere um arquivo com o perfil de execução que contém o número de vezes que cada bloco básico chamou um outro.

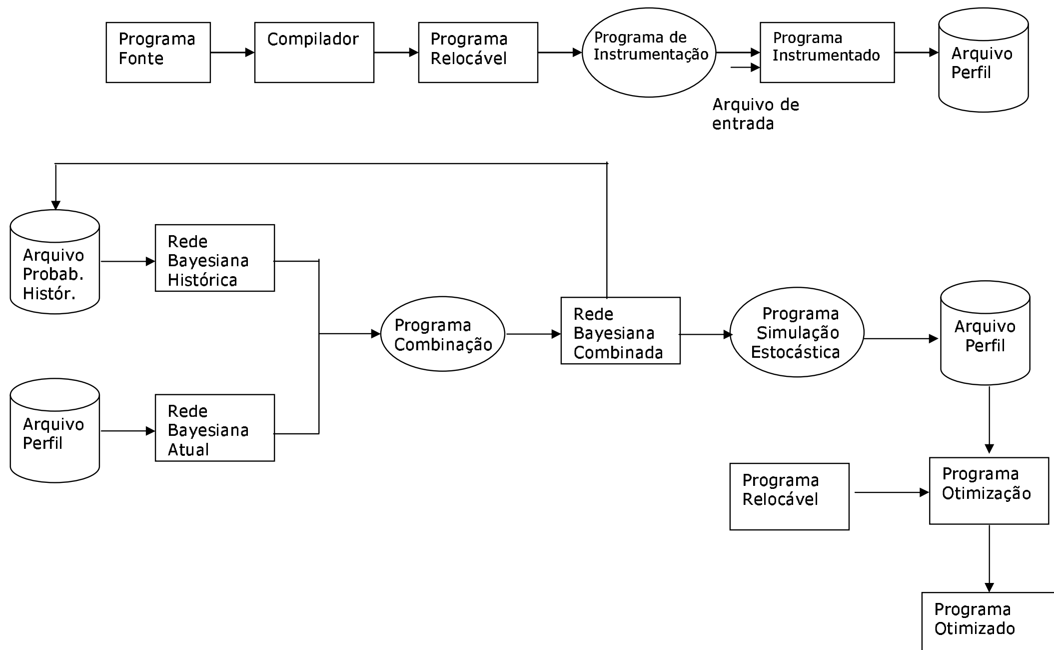


Figura 1.2: Fase de instrumentação e coleta, combinação e otimização

O usuário executa o programa instrumentado com um arquivo de entrada de dados (em um ambiente de teste), para gerar o arquivo de perfil atual. Esse arquivo contém o número de vezes que cada bloco básico chamou o seu sucessor.

Combinação

A segunda fase consiste em combinar dados do arquivo de perfil da execução atual com os dados armazenados em um arquivo histórico, vide Figura 1.2. A partir do arquivo perfil atual será gerado um grafo orientado e ponderado que será transformado numa rede *Bayesiana*, e todos os *loops* existentes neste grafo serão retirados porque redes Bayesianas manipulam um grafo acíclico.

As probabilidades da rede *Bayesiana* histórica serão armazenadas em um arquivo. Foi utilizada a técnica *Noisy-Or* [57], que permite guardar no arquivo de probabilidades uma quantidade de informações bem menor.

Após as duas redes *Bayesianas* serem construídas, será realizada uma média geométrica ponderada entre as probabilidades condicionais da rede *Bayesiana* atual e a rede *Bayesiana* histórica produzindo uma rede *Bayesiana* combinada. Os pesos utilizados nesta média geométrica ponderada serão ajustados de acordo com o tempo

de execução de uma determinada entrada do programa. Entradas executadas mais rapidamente terão peso menor, e entradas mais longas terão peso maior. Esses pesos serão ajustados de acordo com uma função sigmóide.

Otimização

A terceira fase é a otimização do código (vide Figura 1.2). Esta fase pode ser dividida em duas partes: na primeira parte, um programa de simulação estocástica usa a rede *Bayesiana* combinada para encontrar o *hot path* do programa; após ser encontrado o arquivo de perfil será atualizado. Na segunda, usamos o arquivo de perfil e o programa relocável (que contém informações de relocação do arquivo executável original) para posicionar os blocos básicos na memória e será auxiliada pelo PLTO. Será gerado um programa otimizado mais eficiente do que o programa original, e esse programa otimizado será copiado do ambiente de teste para o ambiente em produção, onde será executado gerando os resultados desejados.

Desenvolvemos um algoritmo de aprendizado, baseado em técnicas da inteligência artificial, para encontrar o *hot path* do programa, este problema é computacionalmente intratável, isto é, NP-difícil, daí o uso de uma técnica de simulação estocástica, que emprega redes *Bayesianas* e *Simulated Annealing* permitindo fazer uma análise global do perfil de execução do programa [4, 62].

1.4 Organização do texto

O restante deste trabalho está organizado da seguinte forma. O Capítulo 2 contém um resumo de todos os conceitos básicos necessários para os outros capítulos. São descritas as fases de execução de um programa, a hierarquia de memória e algumas características da arquitetura Intel.

No Capítulo 3 serão mostrados os principais trabalhos existentes na literatura, relacionados com os métodos de otimização, usando dados de um perfil, dando destaque ao trabalho de Pettis/Hansen [58].

No Capítulo 4 serão descritos conceitos básicos sobre redes *Bayesianas* e *Simulated Annealling*, o problema a ser resolvido, a modelagem utilizada, o algoritmo de otimização a ser utilizado, como atua o modelo de execução e o modelo histórico.

Os experimentos e resultados são mostrados no Capítulo 5. No Capítulo 6 é apresentada a conclusão do trabalho. No Apêndice A, apresentamos resultados adicionais (para que o Capítulo 5 não fique muito extenso).

Capítulo 2

Conceitos básicos

Este capítulo apresenta um resumo dos principais conceitos básicos que usamos na tese, como a hierarquia de memória, as fases de execução de um programa e características da arquitetura Intel.

2.1 Hierarquia de memória

Os pioneiros dos computadores previram corretamente que os programadores desejariam quantidades ilimitadas de memória. Uma solução econômica para atender a esse desejo foi o estabelecimento de uma hierarquia de memória, que tira proveito da localidade e da relação "custo x desempenho" das tecnologias de memória. O princípio da localidade estabelece que a maioria dos programas não acessa o código ou os dados de maneira uniforme.

A hierarquia de memória é organizada em vários níveis cada um deles com menor capacidade de armazenamento, mais rápido e mais dispendioso do que o nível seguinte. O objetivo é fornecer um sistema de memória com custo quase tão baixo quanto o nível de memória mais econômico e com velocidade quase tão grande quanto o nível mais rápido. Em geral, os níveis da hierarquia são subconjuntos uns dos outros, todos os dados em um nível também são encontrados no nível abaixo dele e todos os dados nesse nível também são encontrados no nível abaixo do segundo e assim por diante, até ser alcançado o nível inferior da hierarquia, isto é, a base da pirâmide na Figura 2.1.

No topo da hierarquia estão os registradores do processador, que podem ser acessados pelo processador executando à sua velocidade máxima. Em seguida vem a memória *cache*, cuja capacidade está atualmente na faixa que vai de 32 KB até poucos MB (em 2005). Logo a seguir vem a memória principal, com tamanhos que

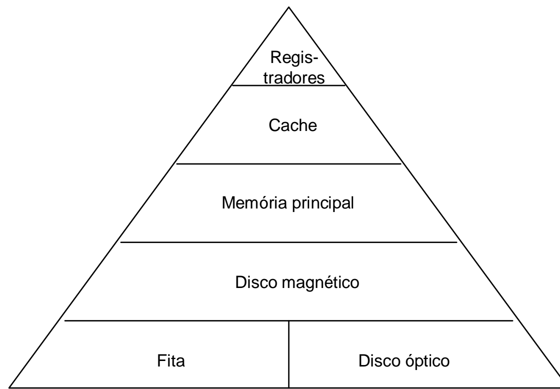


Figura 2.1: Hierarquia de memória

vão de 16 MB para sistemas pequenos a dezenas de GB para máquinas de alto desempenho. Depois vem os discos magnéticos, a tecnologia atual para armazenamento permanente de informações. Finalmente, na parte mais baixa da hierarquia, encontramos a fita magnética e os discos ópticos. Na medida em que a hierarquia desce, pode-se verificar um aumento no tempo de acesso e na capacidade de armazenamento [56, 72].

Considerando a enorme importância que a hierarquia de memória tem no projeto de um sistema computacional e nesta tese, será descrito um breve resumo sobre cada um dos tipos de memória existentes.

Memória principal e secundária

A memória é formada por um conjunto de células (ou posições), cada uma capaz de armazenar uma informação. Cada célula tem um número associado, denominado endereço da célula. É por meio desse número que os programas podem referenciar a célula. Se a memória tiver n células, elas terão endereços de 0 a $n-1$. Todas as células de uma memória são formados por um mesmo número de *bits* [56].

Por maior que seja a capacidade da memória principal, ela sempre será considerada muito pequena. Os usuários sempre querem armazenar mais informações do que a memória pode guardar, principalmente porque, na medida em que a tecnologia evolui, as pessoas começam a pensar em armazenar um volume de informações não imaginável, utilizando assim a memória secundária que são os discos magnéticos (HD).

As pesquisas envolvendo o Sistema Operacional Atlas na Universidade de Manchester no início da década de 60 [35, 36, 37, 42] resultaram na criação dos conceitos

de Memória Virtual e de Hierarquia de Memória. D. E. Knuth apresentou um estudo empírico de programas FORTRAN, onde verificou que apenas 4% das instruções eram responsáveis por 90% do tempo de execução de um programa [43]. O princípio da localidade e o modelo de Conjunto Mínimo de Trabalho foram apresentados em 1968 [19].

Memória *c*ache

Cache é o nome dado ao primeiro nível da hierarquia de memória encontrado assim que o endereço é gerado pela CPU. Quando a CPU encontra na *cache* o item de dados solicitado, diz-se que ocorreu um acerto na memória *cache* (*cache hit*). Quando a CPU não encontra o item requerido, ocorre uma falha na memória *cache* (*cache miss*). Caso ocorra uma falha, uma coleção de tamanho fixo de dados contendo a palavra referenciada, chamada bloco, é recuperada da memória principal e inserida na *cache*. A localidade temporal estabelece que provavelmente essa palavra será necessária de novo em um futuro próximo, assim é útil inserí-la na *cache*, para ser acessada com rapidez. A penalidade imposta por uma falha na *cache* depende da latência e da largura de banda da memória. A latência é o tempo requerido para recuperar a primeira palavra do bloco, e a largura de banda determina o tempo para recuperar o restante desse bloco. Uma falha da *cache* é tratada pelo *hardware* e provoca uma parada no processo em execução até que os dados estejam disponíveis.

O uso de várias *caches* tem-se revelado uma das técnicas mais eficientes para aumentar a banda passante e reduzir a latência de um sistema de memória. Uma técnica que funciona muito bem é introduzir duas *caches*, uma para instruções e outra para dados, técnica conhecida como divisão da *cache*. Muitas vantagens decorrem do uso de duas *caches* separadas para instruções e dados. Em primeiro lugar, as operações com a memória podem ser iniciadas de maneira independente em cada uma das *caches*, dobrando efetivamente a banda passante do sistema de memória.

Hoje em dia, a maioria dos sistemas de memória é mais sofisticado do que descrito, e tem uma *cache* adicional, conhecida como *cache* de nível 2, que fica entre as *caches* de instruções e dados e a memória principal. Na verdade, pode haver três ou mais níveis de *cache*, dependendo da sofisticação e da necessidade do sistema de memória. As *caches* funcionam de modo inclusivo, ou seja, todo o conteúdo da *cache* de nível 1 está na *cache* de nível 2 e todo o conteúdo da *cache* de nível 2 está na *cache* de nível 3.

Quatro aspectos importantes envolvendo os níveis hierárquicos do sistema de memória devem ser discutidos.

1. Onde um bloco pode ser inserido na memória *cache* ?

   Existem três categorias de organização de *caches*:

   - Se cada bloco só tiver um lugar em que ele pode ser armazenado na *cache*, diz-se que a *cache* é de mapeamento direto. Normalmente o mapeamento é : (Endereço do bloco) mod (Número de blocos na *cache*).
   - Se um bloco puder ser inserido em qualquer lugar na *cache*, dizemos que a *cache* é completamente associativa.
   - Se um bloco for inserido em um conjunto restrito de linhas na *cache*, dizemos que a *cache* é associativa de conjunto. Um conjunto é um grupo de blocos na *cache*. Um bloco é mapeado primeiro em um conjunto e depois o bloco pode ser inserido em qualquer lugar dentro desse conjunto. Movendo n blocos em um conjunto, o mapeamento da *cache* é chamado associativa de conjunto de n vias [34, 56].

2. Como um bloco é recuperado se estiver na *cache* ?

   As *cache*s têm um campo *tag* que armazena parte do endereço de cada *frame* bloco. Há também um *bit* de validade indicando se a entrada contém ou não um endereço válido. Se este for o caso, o endereço da CPU é fragmentado em duas partes: endereço do bloco e *offset* do bloco. O endereço do *frame* de bloco pode ser dividido ainda no campo de *tag* e no campo de índice. O campo de *offset* do bloco seleciona os dados desejados do bloco, o campo de índice seleciona o conjunto e o campo de *tag* é comparado com ele em busca de um acerto.

3. Que bloco deve ser substituído ao ocorrer uma falha de *cache* ?

   Quando ocorre uma falha, o controlador da *cache* deve selecionar um bloco para ser substituído pelos dados desejados. Uma vantagem da organização de mapeamento direto é que as decisões do *hardware* são simplificadas, isto é, não há nenhuma escolha, apenas um bloco é verificado em busca de um acerto, e somente esse bloco pode ser substituído. No caso da organização completamente associativa ou associativa de conjunto, há mais de um bloco

a escolher no caso de uma falha. Existem três estratégias principais que são empregadas para selecionar o bloco a substituir:

- Aleatória - Para distribuir uniformemente a alocação, os blocos candidatos são selecionados ao acaso.
- Menos recentemente usado (*LRU - Least recently used*) - Para reduzir a chance de descartar informações que serão necessárias em breve, os acessos a blocos são registrados. Baseando-se no passado para prever o futuro, o bloco substituído é aquele que não é utilizado por um tempo mais longo. A estratégia LRU se baseia em um corolário da localidade que diz que, se os blocos recentemente usados têm maior probabilidade de serem utilizados outra vez, então um bom candidato a ser descartado é o bloco menos recentemente usado.
- Primeiro a entrar, primeiro a sair (*FIFO - First In, First Out*) - Como pode ser complicado usar a LRU em cálculos, essa opção se aproxima da LRU definindo o bloco mais antigo, em lugar do bloco menos recentemente usado.

4. O que acontece em uma gravação?

   As normas de gravação freqüentemente distinguem projetos de *cache*. Existem duas opções básicas quando se grava na *cache*:

   - *Write-through* - As informações são gravadas no bloco na *cache* e no bloco da memória de nível inferior.
   - *Write-back* - As informações são gravadas apenas no bloco da *cache*. O bloco da *cache* modificado será gravado na memória principal somente quando for substituído.

2.2 Fases de execução

Como pode ser visto na Figura 2.2, um programa em linguagem de alto nível é primeiro compilado, gerando um programa equivalente, expresso em linguagem simbólica (*assembly language*), e depois transformado em linguagem de máquina que é o programa objeto. O *linkeditor* combina os diversos módulos objetos com as rotinas da biblioteca, resolve todas as referências entre eles, gerando um programa

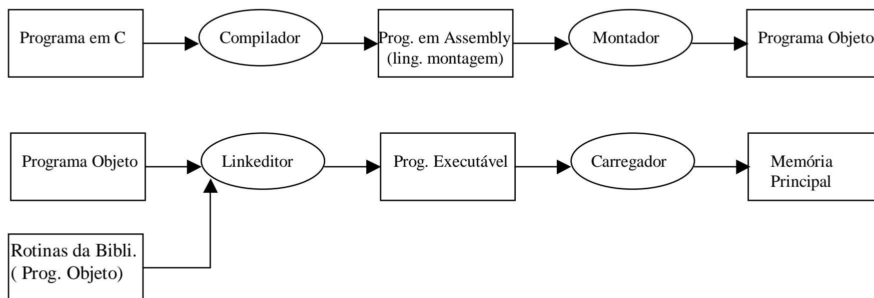


Figura 2.2: Fases de execução

executável. O carregador (*loader*) tem a responsabilidade de colocar o programa executável (código de máquina) nos endereços corretos da memória, permitindo que tal código seja então executado pelo processador.

Compilador

Em geral, os tradutores de linguagens de programação (compiladores, interpretadores e montadores) são programas complexos. Porém, devido à experiência acumulada ao longo dos anos e, principalmente, ao desenvolvimento de teorias relacionadas às tarefas de análise e síntese de programas, existe um consenso sobre a estrutura básica desses tradutores independentemente da linguagem fonte ou do programa objeto a ser gerado. A tarefa de análise engloba o analisador léxico, sintático e semântico, e a tarefa de síntese engloba o gerador de código intermediário, o otimizador de código e o gerador de código objeto. Os tradutores, de um modo geral, incluem funções padronizadas, que compreendem a análise do programa fonte e a posterior síntese para a derivação do código objeto, como pode ser visto na Figura 2.3 [1, 26]. As principais fases do compilador são: análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização de código e geração do código objeto.

Análise léxica e sintática

O objetivo principal da análise léxica é identificar seqüências de caracteres que constituem unidades léxicas (*tokens*). O analisador léxico lê, caractere a caractere, o texto fonte, verifica se os caracteres lidos pertencem ao alfabeto da linguagem, identifica *tokens*, e despreza comentários e brancos desnecessários. Os *tokens* representam classes de símbolos tais como palavras reservadas, delimitadores, identificadores, etc.

A fase de análise sintática verifica se a estrutura gramatical do programa está

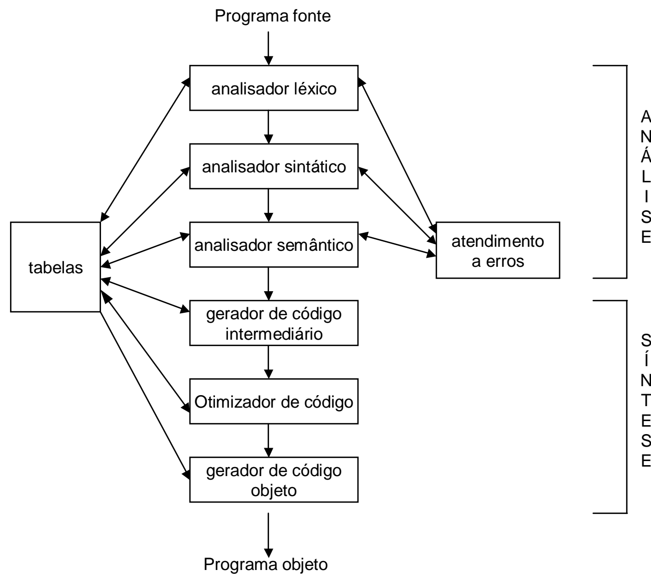


Figura 2.3: Fases do compilador

correta (isto é, se a estrutura foi formada conforme estabelecido pelas regras gramaticais da linguagem). Sua função é verificar se as estruturas do programa irão fazer sentido durante a execução.

Código intermediário

A fase de geração do código intermediário utiliza a representação interna produzida pelo analisador sintático e gera como saída uma seqüência de código. A geração de código intermediário é a transformação da árvore de derivação em um segmento de código. Esse código pode, eventualmente, ser o código objeto final, mas, na maioria das vezes, é um código intermediário, pois a tradução de código fonte para objeto em mais de um passo apresenta algumas vantagens:

- Permite otimizar o código intermediário, para obter um código objeto final mais eficiente;
- Simplifica a implementação do compilador, resolvendo, gradativamente, as dificuldades da transformação do código fonte para objeto (alto-nível para baixo-nível), já que o código fonte pode ser visto como um texto condensado que "explode" em inúmeras instruções elementares de baixo nível;

A desvantagem de gerar código intermediário é que o compilador requer um passo adicional. A tradução direta do código fonte para objeto é um procedimento mais

rápido. A diferença marcante entre código intermediário e código objeto final é que o intermediário não especifica detalhes da máquina alvo, tais como os registradores que serão usados, que endereços de memória serão referenciados etc.

Otimização de código

Uma vez gerado o código intermediário, inicia a fase de otimização que tem o objetivo de tornar o código intermediário mais apropriado para a produção do código objeto (código de máquina) eficiente, tanto em relação ao tamanho como ao tempo de execução. A fase de otimização consiste em identificar segmentos seqüenciais do programa, chamados blocos básicos, representá-los através de grafos orientados e submetê-los a um processo de otimização.

Um programa de computador é formado por blocos básicos, sendo cada bloco básico composto de uma seqüência linear de instruções com um único ponto de entrada (no início do bloco) e somente um ponto de saída (a última instrução do bloco). A ligação entre blocos básicos não adjacentes é feita por comandos de controle. Normalmente, o processo de otimização desenvolve-se em duas fases. A primeira inclui técnicas para eliminar atribuições redundantes, suprimir sub-expressões comuns, trocar instruções de lugar, de modo a obter um código intermediário menor. A otimização de código objeto é realizada através da troca de instruções de máquina por instruções mais rápidas e da melhor utilização de registradores.

Tabela de símbolos

Esta estrutura compreende um conjunto de tabelas e rotinas associadas que são utilizadas por quase todas as fases do tradutor.

Para produzir a versão binária de cada instrução pertencente ao programa em linguagem simbólica, o montador precisa saber os endereços que correspondem a cada um dos *labels* do programa. Os montadores obtém os *labels* usados em instruções de desvio e de transferência de dados examinando uma tabela, chamada tabela de símbolos. Como seria de se esperar, esta tabela armazena pares do tipo símbolo/endereço. Dado um símbolo, a memória informa o seu valor. A Tabela de Símbolos deve ser estruturada de uma forma tal que permita rápida inserção e extração de informações, porém deve ser tão compacta quanto possível.

Montador

Os montadores (*assembler*) como pode ser visto na Figura 2.2, são aqueles tradutores que traduzem programas escritos em linguagem de montagem (*assembly*) para programas em linguagem de máquina (objeto), geralmente, numa relação unívoca, isto é, uma instrução de linguagem simbólica para uma instrução de máquina.

Linkeditor

A maioria dos programas possui mais de um procedimento. Compiladores e montadores traduzem um procedimento de cada vez colocando a saída em disco. Antes que o programa possa executar, todos os seus procedimentos precisam ser localizados e ligados uns aos outros formando um único texto de código.

O *linkeditor* produz um arquivo executável. Tipicamente, este arquivo gerado tem o mesmo formato do programa-objeto, exceto pelo fato de não conter referências não resolvidas, nem informações de relocação. É possível haver arquivos ligados apenas parcialmente, como no caso das rotinas de biblioteca, que ainda possuem endereços não-resolvidos. O *linkeditor* combina os espaços de endereçamento independentes dos módulos objeto em um único espaço de endereçamento linear, por meio da execução dos seguintes passos:

1. O *linkeditor* constrói uma tabela contendo todos os módulos-objeto e seus respectivos tamanhos.

2. Com base nessa tabela, o *linkeditor* atribui um endereço inicial a cada módulo-objeto, conforme Tabela 2.1.

3. O *linkeditor* descobre todas as instruções que fazem referência à memória e soma a cada um dos endereços que elas referenciam uma constante de relocação cujo valor é igual ao valor do endereço inicial do módulo à qual a instrução pertence.

4. O *linkeditor* encontra todas as instruções que fazem referência a um outro procedimento e insere o endereço desse procedimento no lugar adequado da instrução [69].

Módulo	Tamanho	Endereço Inicial
A	400	100
B	600	500
C	500	1100
D	300	1600

Tabela 2.1: Tamanho e endereço inicial dos módulos

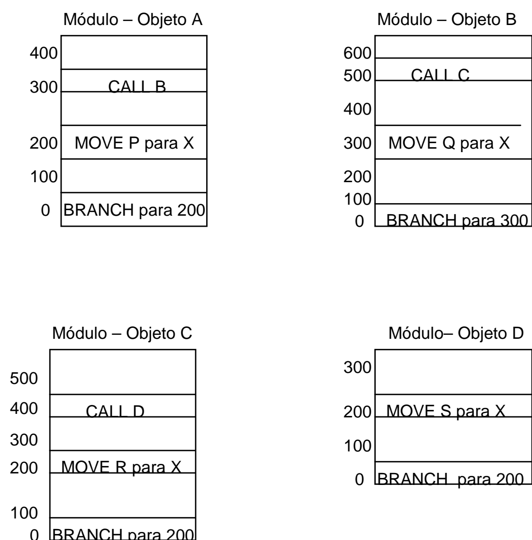


Figura 2.4: Quatro módulos objetos

Problema da relocação

A Figura 2.4 mostra quatro módulos objetos que serão executados em uma máquina genérica. Nesse exemplo, cada um dos módulos começa com uma instrução *branch* (no endereço zero), que desvia para uma instrução *move* dentro do módulo [72].

Para possibilitar a execução do programa, o *linkeditor* coloca na memória principal os módulos-objeto que o compõem, para formar a imagem do programa binário executável, conforme mostra a Figura 2.5 (a). A idéia é formar uma imagem exata dentro do *linkeditor* do espaço de endereçamento virtual do programa e posicionar todos os seus objetos nas posições corretas. Em geral, uma pequena parte da memória, a partir do endereço zero, é usada para acomodar os vetores de interrupção, para implementar a comunicação com o sistema operacional, para guardar os ponteiros, além de outros propósitos menos importantes, de maneira que o programa começa efetivamente em um endereço acima do endereço zero. No exemplo,

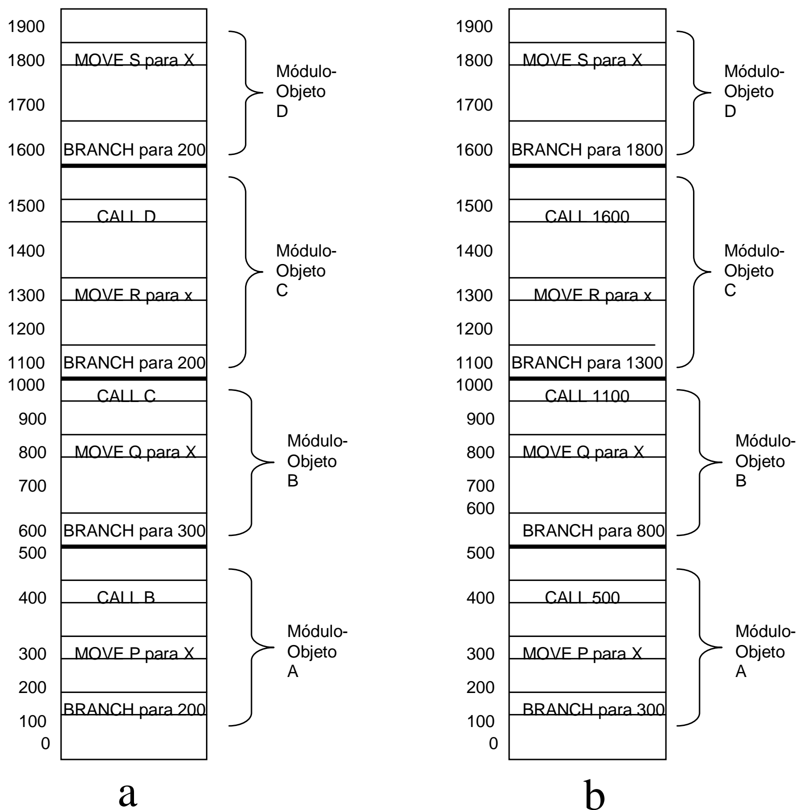


Figura 2.5: a) Programa executável antes da linkedição e relocação e b) Programa executável depois da linkedição e relocação

ilustrado na Figura 2.5 a, foi escolhido arbitrariamente o endereço 100 para endereço inicial do programa.

O programa da Figura 2.5 (a), apesar de ter sido carregada a imagem do binário executável, ainda não está pronto para execução. Considere o que aconteceria se a execução começasse com a instrução situada no início do módulo A. O programa não iria desviar para a instrução *move*, como seria de se esperar, pois essa instrução ocupa o endereço 300. De fato, as referências à memória feitas pelo programa vão falhar pela mesma razão. Esse problema, conhecido como problema da relocação, ocorre em função de cada módulo objeto da Figura 2.4 representar um espaço de endereçamento separado.

O *linkeditor* descobre todas as instruções que fazem referência à memória e soma cada um dos endereços que elas referenciam uma constante de relocação cujo valor é igual ao valor do endereço inicial do módulo à qual a instrução pertence.

Problema com as referências externas

A tabela de símbolos que inclui os módulos é mostrada na Tabela 2.1. Ela fornece o nome, tamanho e endereço inicial de cada módulo.

Além do mais, as instruções de chamada de procedimento (vide objeto A na Figura 2.5 (a)) também não vão funcionar. No endereço 400, o programador tinha a intenção de chamar o módulo-objeto B, mas, como cada procedimento é traduzido isoladamente, o montador não tem como saber que endereço inserir na instrução *call* B. O endereço do módulo-objeto B só será conhecido quando o *linkeditor* efetuar seu trabalho. Este problema é um problema de referência externa. Ambos os problemas (o da relocação e o da referência externa) são resolvidos pelo *linkeditor*.

A Figura 2.5 b mostra os mesmos módulos-objetos após a ligação e a relocação terem ido realizadas. Juntos esses módulos formam um programa binário executável, pronto para ser executado.

Módulo-objeto

O arquivo-objeto típico para o sistema operacional Linux, formato ELF - *Executable and Linking Format* (Formato de Execução e Linkedição), contém seis partes distintas:

- O cabeçalho do arquivo-objeto, que descreve o tamanho e a posição das demais partes componentes do arquivo.
- O segmento de texto que contém o código em linguagem de máquina.
- O segmento de dados que contém os dados que forem necessários à execução do programa, sejam eles estáticos (alocados pelo próprio programa), ou dinâmicos que podem crescer ou diminuir segundo as necessidades do programa, durante sua execução.
- As informações sobre relocação, que visam a identificar as palavras de instrução e de dados que dependem de endereços absolutos, por ocasião da carga do programa na memória.
- A tabela de símbolos, que contém os *labels* remanescentes não-definidos, como por exemplo as referências externas.

- As informações para análise de erros, que contêm uma descrição concisa de como os módulos foram compilados, de modo que o *debbuger* (depurador) possa associar instruções de máquina com arquivos-fonte em C, tornando mais legíveis as estruturas de dados [47, 69].

Loader

Agora que já existe o arquivo executável contendo o programa armazenado em disco, o sistema operacional deve preparar a máquina para executar o programa, e o carregador deve transferir o arquivo para a memória. No Linux são realizados os seguintes passos:

1. Leitura do cabeçalho do arquivo executável para determinar o tamanho dos segmentos de código e de dados.

2. Criação de espaço na memória o suficiente para armazenar código e dados.

3. Copiar as instruções e os dados para a memória.

4. Copiar os parâmetros (se houver) para a pilha do programa principal.

5. Inicializar os registradores da máquina e colocar no *stack pointer* o valor do primeiro endereço livre na memória.

6. Desviar para uma rotina de inicialização que copia os parâmetros nos registradores de argumento e chama a rotina principal do programa. Quando a rotina principal termina, a rotina de inicialização termina o programa, executando a saída do sistema Linux [47, 56, 69].

2.3 Características da arquitetura Intel

Nesta seção serão apresentadas as principais características da arquitetura IA32 da Intel, como os tipos de registradores, formato das instruções e modos de endereçamentos.

Tipos de registradores

A Figura 2.6 mostra os registradores do Pentium. Os primeiros quatro registradores, EAX, EBX, ECX e EDX, são de 32 *bits* e de propósito geral. Cada um deles tem suas próprias peculiaridades: EAX é o principal registrador aritmético; EBX serve para

armazenar ponteiros (endereços de memória); ECX é usado no controle da execução de *loops*; EDX é utilizado em operações de multiplicação e de divisão, durante as quais, junto com o EAX, armazenam os 64 *bits* do produto e do dividendo.

Cada um desses registradores é implementado por meio de um registrador físico de 16 *bits* para armazenar os 16 *bits* mais significativos e um outro registrador físico de 8 *bits* para armazenar os 8 *bits* menos significativos e os 8 *bits* mais significativos. O uso desses dois registradores facilita a manipulação de valores representados em 16 e 8 *bits*, respectivamente. Esta manipulação é muito comum na arquitetura IA32 para manter a compatibilidade com modelos anteriores. Tanto o 8088 quanto o 80286 têm registradores de 8 e de 16 *bits*. Os registradores de 32 *bits* só apareceram no 80386, junto com o prefixo E, que significa extensão.

Os próximos três registradores da arquitetura IA-32 também são considerados de propósito geral, embora tenham outras peculiaridades. Os registradores ESI e EDI armazenam ponteiros para a memória, são usados na execução de instruções de manipulação de cadeia de caracteres, com o registrador ESI apontando para cadeia de caracteres fonte e o registrador EDI para a cadeia de caracteres destino. O registrador EBP aponta para a base da pilha local, razão pela qual é conhecido como apontador de quadro. O registrador ESP aponta para o topo da pilha. O próximo grupo de registradores, formado pelos registradores CS até GS, são os registradores de segmento. Em seguida há o registrador EIP (*Extended Instruction Pointer*), que é o *program counter*. Finalmente, há o registrador EFLAGS, que é o PSW (*program status word*) da arquitetura IA-32 [56, 72].

Formato das instruções

O formato das instruções da arquitetura IA-32 é complexo e irregular, com até seis campos de tamanhos variáveis, cinco dos quais são opcionais. O aspecto geral das instruções é exibido na Figura 2.7. Este formato decorre da evolução da arquitetura durante muitas gerações e em função de algumas escolhas não muito adequadas que ocorreram ao longo do tempo.

Nas primeiras arquiteturas Intel, os códigos de operação tinham 1 *byte*, daí a necessidade do emprego de um *byte* de prefixo para modificar algumas instruções. Um *byte* de prefixo é um código de operação extra, colocado à frente de uma instrução que modifica sua ação.

Os *bits* individuais dos códigos de operação do IA-32 não dão muita informação

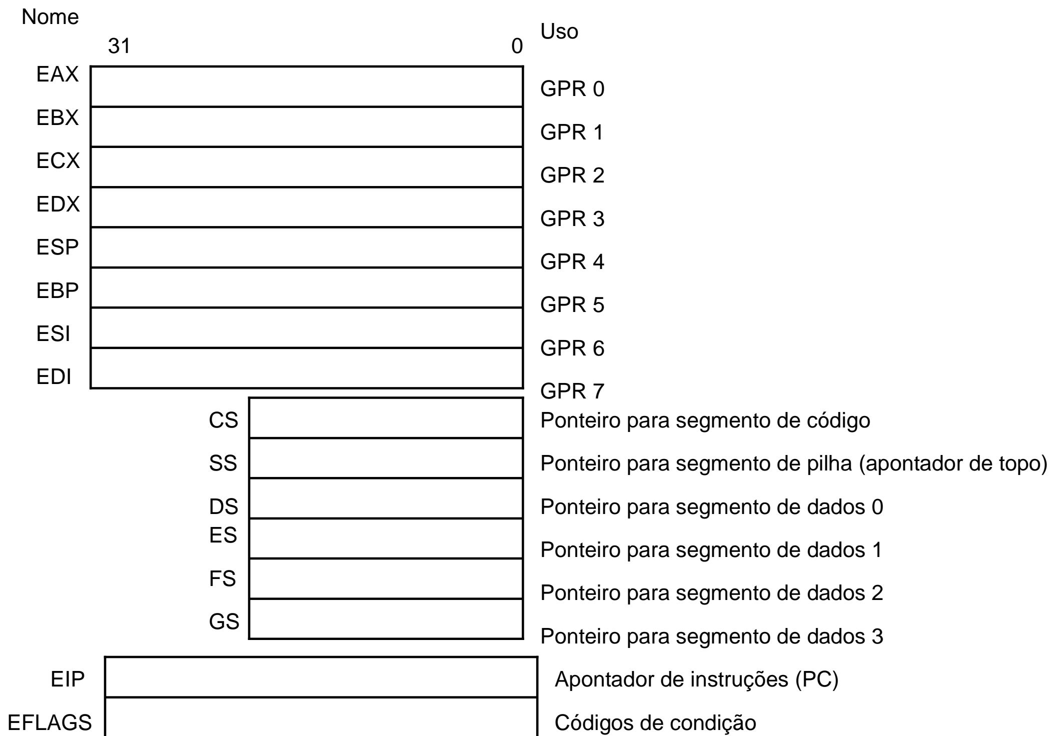


Figura 2.6: Registradores do pentium

a respeito da instrução. A estrutura do campo contendo o código de operação permite o uso dos *bits* de mais baixa ordem de algumas instruções para determinar o uso de um *byte* ou de uma palavra, e o uso de um outro *bit* indica se o endereço de memória (quando houver) é fonte ou destino. Portanto, em geral o código de operação precisa ser totalmente decodificado para determinar a classe da operação a ser realizada e, portanto, para determinar o tamanho da instrução. Isso dificulta a implementação de modelos de alto desempenho, pois é necessária a realização de toda a decodificação antes que se possa conhecer onde a próxima instrução começa.

Após o *byte* do código de operação, a maioria das instruções que referenciam um operando na memória tem um segundo byte com informações sobre esse operando. Nestes oito *bits* há um campo de dois *bits* chamado MOD e dois campos de três *bits*, REC e R/M. Em alguns casos, os três primeiros *bits* desse *byte* são usados como extensão do código de operação, permitindo códigos de operação com onze *bits*. Os dois *bits* do campo de MOD indicam que só há quatro maneiras de se endereçar operandos, e que um deles está em um registrador. Portanto, qualquer dos registradores EAX, EBX, ECX, EDX, ESI, EDI, EBP e ESP pode ser especificado como operando de uma instrução, mas as regras de codificação proíbem algumas

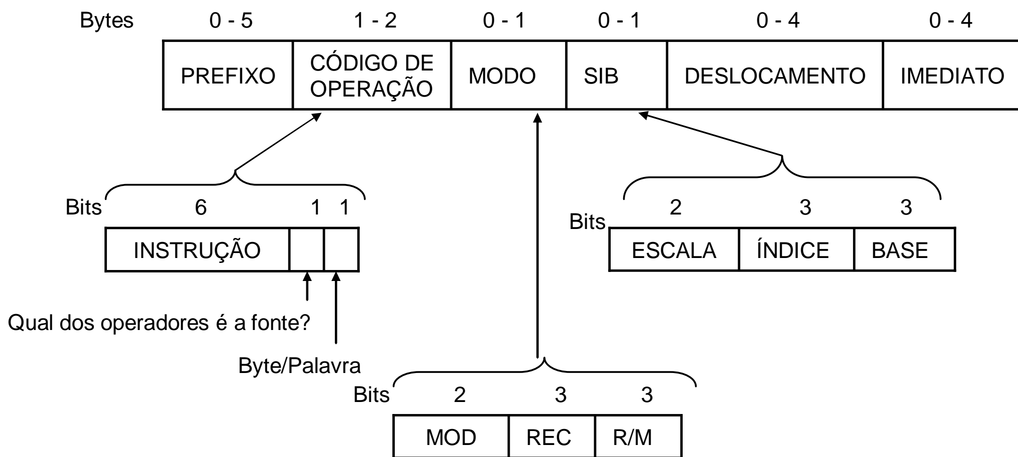


Figura 2.7: Formato das instruções

combinações usando-os em casos especiais. Alguns modos precisam de um byte adicional, conhecido como SIB - Escala, Índice, Base *(Scale, Index, Base)*, que contém mais algumas especificações sobre a instrução. Esse esquema está longe do ideal, mas representa um compromisso entre a compatibilidade com modelos anteriores e a necessidade de incorporar inovações que não foram visualizadas no início do projeto da arquitetura.

Adicionalmente, algumas instruções da arquitetura IA-32 têm mais mais *bytes* para especificar um endereço de memória (deslocamento) e possivelmente outros 1, 2 ou 4 *byte* contendo uma constante (operando imediato) [72].

Modos de endereçamentos

Os modos de endereçamentos da arquitetura IA-32 são também irregulares, apresentando muitas diferenças entre instruções no modo 16 *bits* e instruções no modo 32 *bits*. Aqui abordaremos somente o modo 32 *bits*. Os modos suportados incluem o imediato, o direto, o modo via registrador, o indireto via registrador, o indexado, e um modo especial para endereçamento de elementos de *arrays*. O problema é que nem todos os modos podem ser aplicados a todas as instruções e nem todos os registradores podem ser usados em todos os modos de endereçamento. Isso dificulta a tarefa do compilador para o IA-32.

O *byte* MOD (vide Figura 2.8) controla os modos de endereçamentos do IA-

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + DESLOCAMENTO8]	M[EAX + DESLOCAMENTO32]	EAX ou AL
001	M[ECX]	M[ECX + DESLOCAMENTO8]	M[EAX + DESLOCAMENTO32]	ECX ou CL
010	M[EDX]	M[EDX + DESLOCAMENTO8]	M[EAX + DESLOCAMENTO32]	EDX ou DL
011	M[EBX]	M[EBX + DESLOCAMENTO8]	M[EAX + DESLOCAMENTO32]	EBX ou BL
100	SIB	SIB com DESLOCAMENTO8	M[EAX + DESLOCAMENTO32]	ESP ou AH
101	Direto	M[EBP + DESLOCAMENTO8]	M[EAX + DESLOCAMENTO32]	EBP ou CH
110	M[ESI]	M[ESI + DESLOCAMENTO8]	M[EAX + DESLOCAMENTO32]	ESI ou DH
111	M[EDI]	M[EDI + DESLOCAMENTO8]	M[EAX + DESLOCAMENTO32]	EDI ou BH

Figura 2.8: Modos de endereçamentos

32. Um dos operandos é especificado pela combinação dos campos MOD e R/M. O outro operando é sempre um registrador, determinado pelo campo REC. As 32 combinações que podem ser especificadas pelos campos MOD de 2 *bits* e R/M de 3 *bits*. Se ambos os campos forem zero, por exemplo, o operando é lido do endereço de memória armazenado no registrador EAX.

As colunas 01 e 10 da Figura 2.8 tratam dos modos nos quais um registrador é somado a um deslocamento de 8 ou de 32 *bits* que sucede o código de operação na instrução. Se o deslocamento selecionado for de 8 *bits*, ele precisa ser transformado em um valor de 32 *bits*, antes de ser adicionado ao valor do registrador.

A coluna com MOD = 11 permite a escolha de um entre dois registradores. Para instruções que manipulam palavras, utiliza-se a primeira escolha; para instruções com *bytes*, utiliza-se a segunda escolha. Observe que a tabela não é inteiramente regular. Por exemplo, não há como implementar o modo de endereçamento indireto usando EBP, nem empregar deslocamento a partir de ESP.

Capítulo 3

Trabalhos relacionados

Nas últimas décadas, diversos trabalhos foram realizados para reduzir o tempo de execução de um programa usando o perfil de execução. O trabalho de Pettis e Hansen [58] em 1990 foi pioneiro e vamos relatar com mais detalhes na Seção 3.1.

Reestruturar programas visando um melhor desempenho do sistema de memória foi estudado por vários investigadores, os principais foram: Ferrari em [22, 23] e Hartley em [31]. Mais recentemente, técnicas para posicionamento dos blocos básicos foram apresentados por McFarling [51, 52] (que usa dados de um arquivo de perfil para guiar o posicionamento) e Hwu e Chang [9, 38] também usaram esta técnica. Existe também o algoritmo sobre eficiente técnicas de tracing e *profile* (perfil) [2, 33, 76]. R. Gupta trabalhou na eliminação de código redundante e na eliminação de desvios condicionais [28, 29, 27, 6, 5]. Outros trabalhos se destacaram como o do Calder e do Young [8, 24, 75].

O trabalho realizado por Larus e Ball em 1996 [3] faz a descrição de um novo algoritmo para encontrar o *hot path* do programa que é o caminho mais provável de ser executado. É um algoritmo simples, rápido, realiza a instrumentação do programa e seu objetivo principal é minimizar o *overhead* do tempo de execução da instrumentação. O algoritmo faz uma instrumentação que calcula a freqüência de execução dos caminhos do grafo de fluxo de controle (GFC). A instrumentação não é apenas simples e de baixo custo, mas é calculada de uma maneira que minimiza o seu *overhead*. Notavelmente, embora o perfil de caminho colete muito mais informação do que o perfil de aresta, o seu *overhead* pode ser menor, outros trabalhos podem ser vistos em [2, 44].

A otimização em tempo de *linkedição*, reescrita binária, instrumentação binária e otimização do programa foram exploradas e diversas ferramentas foram implemen-

tadas com esse objetivo.

O Alto é um otimizador em tempo de *linkedição* para as máquinas Alfa da Compaq tendo como principal pesquisador o S. Debray [55]. Para a arquitetura Intel pode-se destacar o PLTO [18, 63], que é a versão do Alto para arquitetura Intel. O PLTO foi o *software* utilizado nesta tese por ser uma excelente ferramenta para instrumentar aplicações em ambiente Intel/Linux , e seu o código é aberto e gratuito.

O Etch [61] é um sistema focado em instrumentação de arquivos executáveis IA-32, esse *software* existe versão para ambientes Windows e Linux, mas não foi disponibilizado pela Universidade de Harvard.

O Vtune [39] da Intel não é gratuito, não possui o código aberto e só existe para ambiente Windows. Outros pacotes podem ser citados: NT-Atom e Hi-Prof [15] para Windows NT, que são especializados em instrumentação e análise de arquivos executáveis IA-32, somente disponíveis para ambiente Windows.

A maioria das ferramentas disponíveis para realizar a instrumentação de binário é focada para arquitetura RISC como Alto [55], Atom [16, 67], Qpt2 [45, 50], EEL [46], OM [68] e o Spike [13], esses dois últimos são direcionados ao Compaq Alfa.

Também existe o UQBT (*the University of Queensland Binary Translator*), que é capaz de estaticamente, traduzir arquivos executáveis em diferentes arquiteturas. Ele é direcionado, entre outras coisas, para a arquitetura IA-32 [12, 11].

Vários desses pacotes como o: etch, alto, qpt2, nt-atom, vtune foram estudados e/ou instalados mas foram descartados por diferentes razões. No início do trabalho foi tentado desenvolver uma ferramenta de instrumentação, utilizando as ferramentas como objdump [59], gprof, lcov e gcov [25] contudo o PLTO surgiu como uma excelente opção.

Hoje em dia, uma boa razão para a relevância das otimizações guiadas pelo perfil é que muitas oportunidades para as otimizações que não podiam ser exploradas em processadores antigos, agora podem devido aos aspectos avançados do *hardware* presente em processadores modernos. A assistência para a execução especulativa e a prevenção de intruções cuja especulação depende de predicados pode ser útil na execução de otimizações guiadas pelo perfil [20, 64].

O *trace* do fluxo do programa (TFP) pode ter um tamanho extremamente grande e, portanto, as representações que mantêm as TFP's na forma comprimida foram consideradas em [44], Larus utilizou o algoritmo Sequitur para a compressão en-

quanto que em [76], Zhang e Gupta propuseram técnicas alternativas de remoção de redundância para a compressão. Vários trabalhos visando otimizar a utilização da memória *cache* aparecem em [10, 30, 32, 38, 41, 74].

Diversos trabalhos de otimização, inclusive esse, utilizam um arquivo de perfil para fazer a otimização. Existem vários tipos de arquivo perfil diferentes: perfil de vértice, perfil de arestas [50], *profile* de duas arestas [53], perfil de caminho [3] e outros.

3.1 Pettis e Hansen

Este estudo de Karl Pettis e Robert C. Hansen de 1990 [58] foi o marco inicial para a investigação realizada na década passada sobre técnicas de posicionamento de código usando dados de um perfil. Foram desenvolvidos os protótipos de compilador e *linkeditor* que usam informações de um perfil como *feedback* para guiar o posicionamento do código a fim de reduzir o *overhead* na hierarquia de memória. Especificamente, eles tentaram melhorar o desempenho da *cache* de instruções, embora outros benefícios terem sido encontrados.

Naquele trabalho, três protótipos para a arquitetura HP (PA-RISC) foram implementados. O primeiro (construído no *linker*) realiza o posicionamento do código baseado nos procedimentos (*procedures*). Este protótipo tem a habilidade de mover um procedimento em uma ordem que é determinada pela estratégia *close is best* (mais próximo é melhor).

O segundo protótipo (construído em um otimizador existente) que posiciona o código segundo os blocos básicos, será descrito com mais detalhes porque foi utilizado pelo PLTO.

O terceiro protótipo, pega os códigos que nunca são executados e os separam do código principal de um procedimento por uma técnica chamada *splitting*.

Posicionamento por bloco básico

O segundo protótipo utiliza informações do perfil de execução para guiar o posicionamento dos blocos básicos. Na maioria das aplicações existem caminhos que raramente ou nunca são percorridos durante uma execução normal. O objetivo é identificar o código executado raramente de modo que o fluxo normal de controle seja uma seqüência em linha reta. O sentido do desvio condicional será trocado para refletir uma nova ordem do código. Benefícios esperados:

- Longas seqüências do código são executadas antes de tomado o desvio.

- Na média o número de instruções executadas e que já estavam na *cache* aumenta.

- Redução no número de *cache misses* ou (falhas na cache).

- Penalidades impostas pelos desvios são reduzidas.

- Em alguns casos (*if then else* com a cláusula *else* raramente executada), pode-se mover o código executado com pouca freqüência.

A idéia deste algoritmo é formar cadeias de blocos básicos que deveriam ser colocadas em linha reta. A Figura 3.1 mostra o grafo ponderado onde os valores das arestas indicam quantas vezes um bloco básico chama um outro e uma descrição rápida da técnica. O algoritmo procura a aresta mais pesada (maior valor) que é aresta que possui o peso igual a 6964 (essa aresta liga os blocos básicos B e C) e une esses 2 blocos básicos numa cadeia. Depois a aresta de peso 6950 que liga o bloco básico C e D. Colocamos C no fim da cadeia e assim sucessivamente, conforme a Figura 3.1. No final ficamos com apenas 6 cadeias individuais de blocos básicos.

- A

- E-N-B-C-D-F-H

- I-J-L

- G-O

- K

- M

Depois que as cadeias são formadas, foi estabelecida então uma relação de precedência entre elas baseada no desejo de se ter desvios condicionais não tomados para frente. É necessário decidir sobre os desvios condicionais em C, F , I e J (vide Figura 3.1).

- cadeia 2 (C) antes da cadeia 4 (G)

- cadeia 2 (F) antes de cadeia 3 (I)

- cadeia 3 (I) antes de cadeia 6 (M)
- cadeia 3 (J) antes de cadeia 5 (K)

O único desvio condicional que saiu do critério foi B que ficou depois de E, deveria ficar antes. Como eles fazem parte da mesma cadeia, o desvio vai ser para trás. A ordem final dos blocos básicos é: A, E-N-B-C-D-F-H, I-J-L, G-O, K, M. A principal idéia que está por atrás do algoritmo é tentar reduzir o número de desvios tomados.

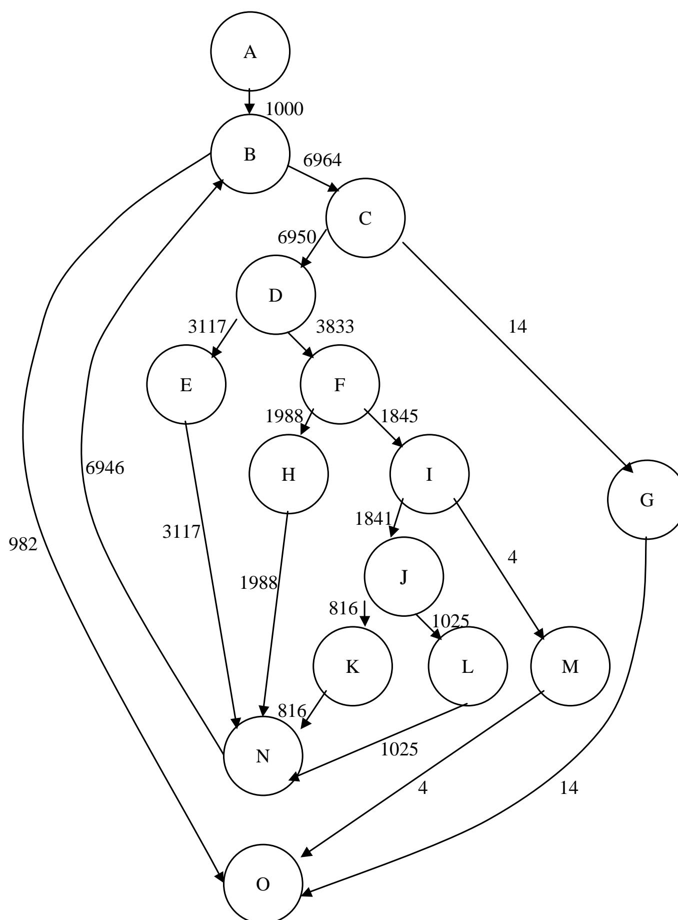


Figura 3.1: Grafo ponderado

Capítulo 4

Modelagem

Este capítulo caracteriza o problema, apresenta com detalhes a heurística que será usada, relata os problemas que foram resolvidos para implementação, explica o funcionamento do modelo de execução, o modelo histórico e faz um resumo da estratégia. Antes vamos descrever alguns conceitos básicos de redes *Bayesianas* e *simulated annealing* para um melhor entendimento da modelagem.

4.1 Simulated annealing

Simulated annealing é um método seqüencial para tentar encontrar o mínimo global das funções ao permitir movimentos crescentes ocasionais, por exemplo, movimentos nos quais os valores da função aumentam durante o processo de otimização.

Considere uma função f em D^n , seja $y \in D^n$ o ponto corrente e seja $y' \in D^n$ o "vizinho" de y, que está sendo considerado como o próximo ponto na seqüência. Se $f(y') \leq f(y)$, então y' será aceito e considerado como o novo ponto corrente. Se, por outro lado, $f(y') > f(y)$, então y' será aceito com uma certa probabilidade. Se y' não for aceito, então y continuará a ser o ponto corrente e um outro de seus "vizinhos" será selecionado.

Na física, *annealing* (recozimento) é um processo térmico para obter estados de baixa energia em um banho quente. O processo possui os seguintes passos: incrementar a temperatura do banho quente para um valor máximo no qual o sólido derrete e reduzir vagarosamente a temperatura do banho quente até que as partículas se encontrem no estado de baixa energia do sólido. Na fase líquida, todas as partículas do sólido arranjam-se aleatoriamente. No estado de baixa energia, as partículas formam um arranjo altamente estruturado conhecido como cristal, onde a energia do sistema é mínima. O estado fundamental do sólido é obtido somente se a tem-

peratura máxima for suficientemente alta e o resfriamento for suficientemente lento. Caso contrário o sólido é congelado em um estado meta-estável ao invés do estado fundamental.

Computacionalmente, o recozimento pode ser visto como um processo estocástico de determinação de uma organização dos átomos de um sólido que apresente energia mínima. Em temperatura alta, os átomos se movem livremente e, com grande probabilidade, podem mover-se para posições que incrementarão a energia total do sistema. Quando se baixa a temperatura, os átomos gradualmente se movem em direção a uma estrutura regular e, somente com pequena probabilidade, incrementarão suas energias.

O algoritmo Metropolis [54] é um algoritmo seqüencial que gera uma seqüência de pontos $x_0, x_1, ..., x_{n-1}$ em D^n com o valor de T (temperatura) fixo. Este procedimento quando modificado pela variação de T com o tempo, é o método de minimização conhecido como *simulated annealing*, o qual é iniciado com um valor alto para T e o reduz gradualmente. Com valores altos de T (por exemplo, no início da simulação), praticamente todo novo ponto gerado é aceito. Em baixas temperaturas de T (por exemplo, próximo ao fim da simulação), praticamente nenhum aumento no valor da função é mais aceito [4, 62].

Para viabilizar a implementação do algoritmo é necessário tomar algumas decisões, tais como:

- Valor inicial do parâmetro de controle (T_0). O valor inicial de T deve ser suficientemente grande para que a maioria dos movimentos seja aceito.

- Redução geométrica da temperatura (β). Como nos processos de resfriamento de metais este é um dos fatores mais importantes, o valor que a temperatura será decrementada. Na prática, o que é feito é o uso de uma redução geométrica para reduzir o valor de T, realizada por um parâmetro β ($T \leftarrow \beta\ T$), onde $0 < \beta < 1$; esse valor deve ficar bem perto de 1 (geralmente entre 0.8 e 0.99). O valor de β usado no algoritmo afeta o tempo de execução e a qualidade desejada para a solução. Pode-se observar, que o número de vezes que a temperatura é decrementada é igual a $1 + \log_{\frac{1}{\beta}}(T_0/T_f)$, onde T_f é a temperatura final.

- Valor final da temperatura (T_f). Na teoria, o processo deve continuar até que a temperatura final seja próxima de zero. Na prática, o processo pode ser encerrado quando a razão de aceitação de novas soluções se tornar insignificante.

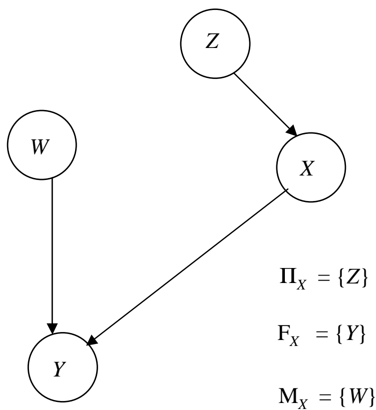


Tabela de probabilidades condicionais

Y	W	X	P(Y | W,X)
0	0	0	0.95
0	0	1	0.6
0	1	0	0.5
0	1	1	0.3
1	0	0	0.05
1	0	1	0.4
1	1	0	0.5
1	1	1	0.7

Figura 4.1: Rede *Bayesiana*

4.2 Redes Bayesianas

Redes *Bayesianas* são modelos gráficos que permitem expressar a relação causa-efeito entre variáveis aleatórias de um sistema, através do uso de probabilidades condicionais e métodos *Bayesianos.* Uma rede *Bayesiana* é um grafo acíclico, cujos vértices são variáveis aleatórias e cujas arestas orientadas indicam relações causais diretas, isto é, se X e Y são vértices, então a existência de uma aresta $(X \to Y)$ indica que o valor de X influência diretamente o valor de Y. Cada variável (vértice do grafo) possui um valor binário 0 ou 1.

Vamos usar Π_X para denotar o conjunto de variáveis que possuem arestas orientadas em direção a X (são chamados pais de X numa rede *Bayesiana*), F_X será o conjunto de variáveis que possuem arestas que partem de X (são chamados filhos de X numa rede *Bayesiana*) e M_X será o conjunto de "mates" de X que são os outros pais dos filhos de X. Podemos verificar na Figura 4.1 que existe uma aresta orientada de X para Y e de W para Y para algum Y, logo W faz parte do conjunto M_X.

Cada variável tem associado a si um valor 0 ou 1 e uma tabela de probabilidades condicionais que quantificam os efeitos que os pais exercem sobre um vértice (probabilidade do vértice estar num estado específico (0 ou 1) dado o estado dos seus pais), logo cada vértice está associado a uma tabela com $2^{1+|\Pi_X|}$ probabilidades condicionais. Como pode ser visto na Figura 4.1, cada vértice possui uma tabela

que armazena a probabilidade condicional da variável ter valor 0 ou 1 dado os valores de seus pais. Podemos observar que basta guardar na tabela as probabilidades condicionais em que um vértice possui valor 0 ou 1 porque a outra probabilidade é complementar, isto é, como na tabela da Figura 4.1, a probabilidadeda da variável Y ter valor 1 dado que seus pais tenham valores 1 é 0.7, logo a probabilidade da variável Y ter valor 0 dado que seus pais tenham valores 1 é 0.3. Concluímos que é necessário que cada vértice tenha associado a ele somente uma tabela com $2^{|\Pi_X|}$ probabilidades condicionais. Mais adiante vamos mostrar como essas probabilidades são calculadas.

Existem algumas variáveis que são especiais. Essas variáveis constituem a entrada ou a evidência da rede *Bayesiana* e são variáveis que possuem valores conhecidos. O problema fundamental que se deseja resolver, uma vez que a rede *Bayesiana* esteja criada, é calcular a probabilidade conjunta de todas as variáveis que não são evidências dados os valores dessas variáveis em evidência. Nosso objetivo é maximizar essa probabilidade conjunta, o que é computacionalmente intratável, no sentido de ser NP-difícil. Na prática, será realizada uma aproximação por um algoritmo de simulação estocástica.

O mecanismo básico da simulação estocástica diz que o algoritmo converge se todas as variáveis são atualizadas infinitamente freqüentemente, então, um mínimo global vai ser alcançado. O algoritmo de simulação estocástica, descrito na Figura 4.2, consiste em atualizar os valores de todas as variáveis que não sejam variáveis evidências em função de uma probabilidade que depende do parâmetro T de *simulated annealing*. Indiretamente, o que ela faz é otimizar globalmente uma função cujo ótimo fornece a atribuição conjunta de valores mais provável. Ao atualizar a variável X com valor x, por exemplo, a probabilidade proporcional (por uma constante de normalização) a ser usada é,

$$\prod_{Y \in \mathbf{N_X}} p(y \mid \Pi_Y)^{1/T}, \tag{4.1}$$

Na Equação 4.1, $\mathbf{N_X}$ compreende o próprio X e seus filhos; indiretamente, então, a distribuição de probabilidade depende somente da variável X, seus pais, filhos e seus "mates" [4].

O algoritmo de simulação estocástica da Figura 4.2 determina aleatoriamente valores iniciais (0 ou 1) para todas as variáveis da rede *Bayesiana* (n variáveis). Inicialmente é determinado um valor alto para a temperatura inicial (T_0), T é reduzido

geometricamente e a temperatura final (T_f) é 1. O algoritmo processa cada vértice que não seja um vértice evidência, calculando uma probabilidade conforme (4.1) e atualizando seu valor correspondentemente. No final do processamento do algoritmo, os vértices são examinados e todos os que possuem valor 1 são os vértices mais prováveis de serem executados. Os vértices que possuem valor 1 são os vértcies que são executados e os que possuem valores 0 são os vértices que não foram executados.

```
Início
 Para i de 1 até n
   Cada vértice recebe aleatoriamente o valor 0 ou 1
 Fim-Para
 T = T_0;
 Enquanto (T ≥ T_f)
   Para i de 1 até n
     Se (X não é um vértice evidência)
       Calcular a Prob.Condicional conforme Equação 4.1
       Atualizar X
     Fim-Se
   Fim-Para
   T = β. T;
 Fim-Enquanto
Fim
```

Figura 4.2: Algoritmo de simulação estocástica

4.3 Problema a ser resolvido

Um dos problemas que têm desafiado os projetistas de computadores ao longo da história é o desenvolvimento de sistemas de memória capazes de suprir o processador com instruções e dados com a mesma velocidade que ele processa essas informações. O descompasso entre a velocidade do processador e a da memória tem resistido por décadas e o problema está se agravando a cada ano. Considerando a enorme importância que a memória principal tem no projeto de um sistema computacional, essa situação é um fator negativo no desenvolvimento de sistemas, e tem estimulado diversas pesquisas para tratar o problema.

O principal objetivo desse trabalho é reduzir o tempo de execução de um programa P. O problema que estamos querendo resolver é reordenar o programa executável de forma que melhore o desempenho da memória, isto é, será realizada uma

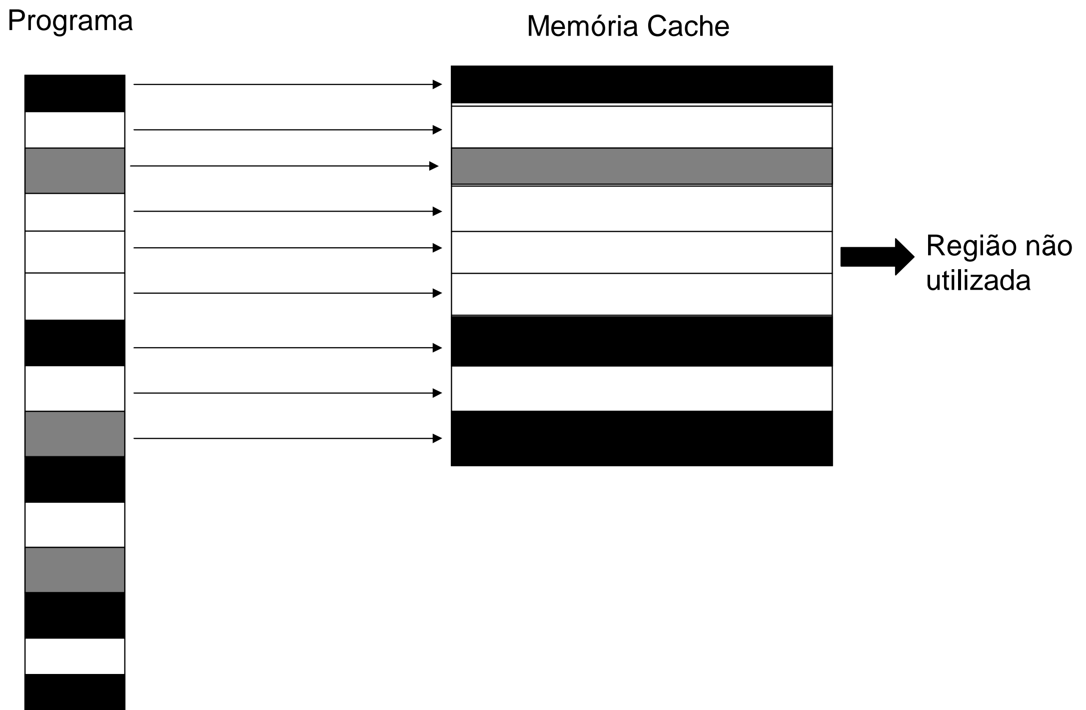


Figura 4.3: Alocação do programa na memória *cache* antes da reordenação dos blocos básicos

reordenação dos blocos básicos do programa executável. Podemos ver nas Figuras 4.3 e 4.4 que um dos nossos objetivos com essa reordenação é utilizar melhor a memória *cache* de instruções.

O problema que queremos resolver no contexto das redes *Bayesianas*, já foi explicado na Seção 4.1.2. Vamos resolver o problema em três fases como será descrito posteriormente: instrumentação e coleta, combinação e otimização.

Em cada uma dessas fases existem alguns problemas que devem ser superados. Na fase de instrumentação, são inseridos contadores para determinar com que freqüência cada bloco básico chamou um outro durante a execução do programa. Após a coleta de informações é gerado um arquivo que contém o perfil de execução do programa.

Na fase de combinação, o arquivo de perfil pode possuir *loops*, o que não é aceitável já que as redes *Bayesianas* utilizam grafos acíclicos. Todos os problemas são tratados e o arquivo é transformado num grafo e depois numa rede *Bayesiana*.

O arquivo histórico é armazenado e guarda as probabilidades da rede *Bayesiana* histórica como será descrito mais adiante. Outro problema significante é a necessidade de armazenar em disco um arquivo histórico de probabilidades muito grande.

Essas duas redes *Bayesianas* são combinadas e é gerada uma rede *Bayesiana*

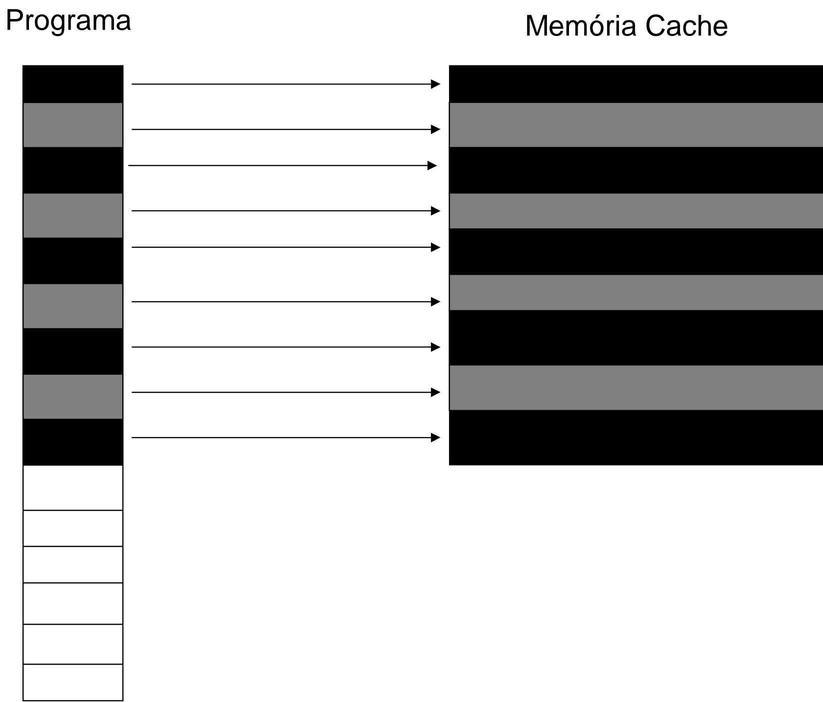


Figura 4.4: Alocação do programa na memória *cache* depois da reordenação dos blocos básicos

resultante que servirá de entrada para o programa de simulação estocástica.

Na fase de otimização o primeiro objetivo é encontrar um *hot path* do programa, logo após atualizar o arquivo de perfil com o *hot path* encontrado. O segundo objetivo é reordenar na memória esses blocos básicos de acordo com arquivo de perfil atualizado.

4.4 Método proposto

Vamos explicar a modelagem para resolver o problema em 3 fases: instrumentação e coleta, combinação e otimização.

4.4.1 Instrumentação e coleta

Esta fase será auxiliada pela ferramenta PLTO [18, 63] e pode ser visualizada na Figura 4.5. Para compilar um programa fonte na linguagem C, usando o compilador gcc, informações de relocação usadas pelo PLTO devem ser retidas através do *flag* “r”para o *linkeditor*. Isso pode ser feito especificando a diretiva “-Wl, -r”. Desse modo, para compilar um programa p.c com o otimizador de nível 3 (O3), deve ser executado o seguinte comando:

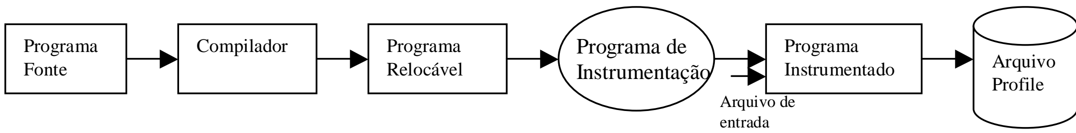


Figura 4.5: Fase 1 - instrumentação e coleta

$ gcc -O3 p.c -Wl, -r -o p.reloc

Este programa relocável (p.reloc), gerado pelo compilador, serve de entrada para um programa de instrumentação (PLTO) que insere instruções extras no código executável, isto é, o programa de instrumentação altera o programa relocável gerando um programa instrumentado (p.prof). Uma opção na linha de comando (opção -p) é usada para instruir o PLTO a adicionar o código de instrumentação necessário para gerar o programa instrumentado [18].

$ plto -l +p p.reloc

O programa de instrumentação realiza as seguintes tarefas:

- É realizado um *disassembler* de todos os segmentos contendo código; é gerada uma seqüência de instruções simples, fragmentando o programa em blocos básicos.
- É gerado um grafo orientado onde os vértices são os blocos básicos do programa executável.
- É inserido um contador em cada aresta do grafo que acumula o número de vezes que foi executada. A área de dados do programa será expandida para acomodar esses contadores.
- O grafo é analisado e todo vértice não atingível é eliminado.
- No final da execução do programa instrumentado, é gerado um arquivo de perfil.

Quando o programa instrumentado for executado com seu respectivo arquivo de entrada (em um ambiente de teste), será gerado um arquivo de perfil que tem o formato mostrado na Figura 4.6. Cada linha possui o endereço de identificação do

```
0x080481e0 -> 0x0804d0f0[0] [1]
0x080481f3 -> 0x080487c8[0] [1]
0x080481fb -> 0x08048210[0] [1]
0x08048210 -> 0x08048c20[0] [2500]
0x08048215 -> 0x08048224[0] [1250]
0x08048215 -> 0x0804821c[0] [1249]
0x0804821c -> 0x0804822a[0] [1249]
0x08048224 -> 0x0804822a[0] [1250]
0x0804822a -> 0x0804823c[0] [2000]
0x0804822a -> 0x08048238[0] [4998]
0x08048238 -> 0x0804823d[0] [4998]
0x0804823c -> 0x0804823d[0] [2000]
0x0804823d -> 0x08048c20[0] [2500]
0x08048242 -> 0x08048254[0] [1249]
0x08048242 -> 0x0804824a[0] [1250]
0x0804824a -> 0x0804825a[0] [1250]
0x08048254 -> 0x0804825a[0] [1249]
```

Figura 4.6: Formato do arquivo de perfil

bloco básico origem e destino, o último número dentro de colchetes é a freqüência de desvio entre os dois blocos básicos, isto é, a quantidade de vezes que o bloco básico origem chamou o bloco básico destino. O terceiro campo é o *jumple table* que é o desvio em relação à base, muito usado no comando *switch*.

4.4.2 Combinação

Na segunda fase da modelagem (vide Figura 4.7), o arquivo de perfil gerado na fase anterior é transformado numa rede *Bayesiana*. Na primeira execução, como não existe ainda arquivo de probabilidade histórica, será realizado uma cópia da rede *Bayesiana* atual para rede *Bayesiana* histórica.

Nas demais execuções, esse arquivo de probabilidade histórica gera uma rede *Bayesiana* histórica que será combinada com a rede *Bayesiana* da execução atual.

Após as duas redes *Bayesianas* serem construídas, será calculada a média geométrica ponderada entre as probabilidades condicionais da rede *Bayesiana* atual com a rede *Bayesiana* histórica produzindo uma rede *Bayesiana* combinada. Os pesos utilizados nesta média geométrica ponderada serão ajustados de acordo com o tempo de execução da entrada do programa. Entradas executadas mais rapidamente terão peso menor, e entradas mais demoradas terão peso maior. Esses pesos serão ajustados de acordo com a função sigmóide que será explicada mais adiante. Programas com tempos de execução muito abaixo da média histórica devem influenciar menos o histórico e programas acima devem influenciar mais, vamos tratar com mais detalhes na Seção 4.4.

Os dados históricos armazenados são as probabilidades de cada vértice dados os valores dos vértices dos seus pais. Em cada vértice será armazenado somente $|\Pi_X|$

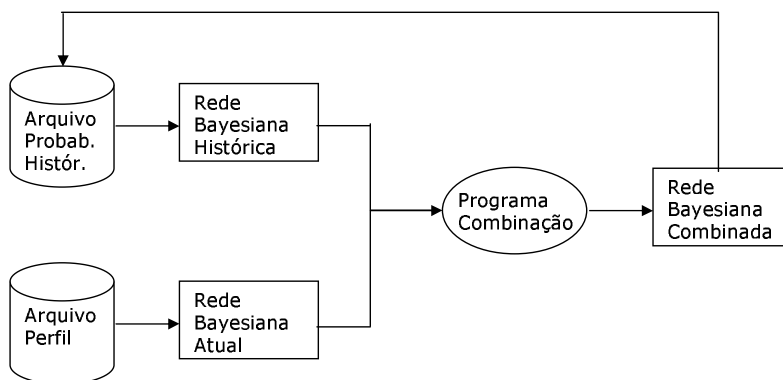


Figura 4.7: Fase 2 - combinação

(número de pais de X) probabilidades, ao invés de $2^{|\Pi_X|}$ probabilidades, por causa da técnica Noisy-Or relatada mais adiante.

Após o arquivo de perfil ser gerado na fase de instrumentação, são realizadas algumas etapas para esse arquivo ser transformado numa rede *Bayesiana*. O arquivo possui alguns problemas que devem ser corrigidos. Vamos relatar alguns problemas encontrados para transformar o arquivo de perfil e o histórico em redes *Bayesianas*.

Problemas no arquivo de perfil

Problema 1: O arquivo de perfil não contém retorno da função.

O arquivo perfil não possui uma aresta indicando o retorno de uma função. Como pode ser visto na Figura 4.8, se o programa principal chama uma função B, não existe uma linha (aresta) no arquivo perfil onde a função B retorna para o programa principal. Foi inserida uma aresta de retorno após o bloco básico do programa principal que chamou a função B e foi retirada a aresta entre os 2 blocos básicos contíguos do programa principal.

Problema 2: O endereço inicial da função main.

Outro problema foi identificar, na rede *Bayesiana*, qual o bloco básico inicial da função *main* para que o *hot path* seja identificado. Existe uma opção no PLTO (opção -s) que gera um arquivo chamado *sloop* contendo informações do programa executável, nome das funções, nome das instruções, endereços dos blocos básicos, freqüência das arestas, vide Figura 4.9. A partir desse arquivo é encontrado o endereço inicial da função *main*, nesse exemplo é o 0x080481e0.

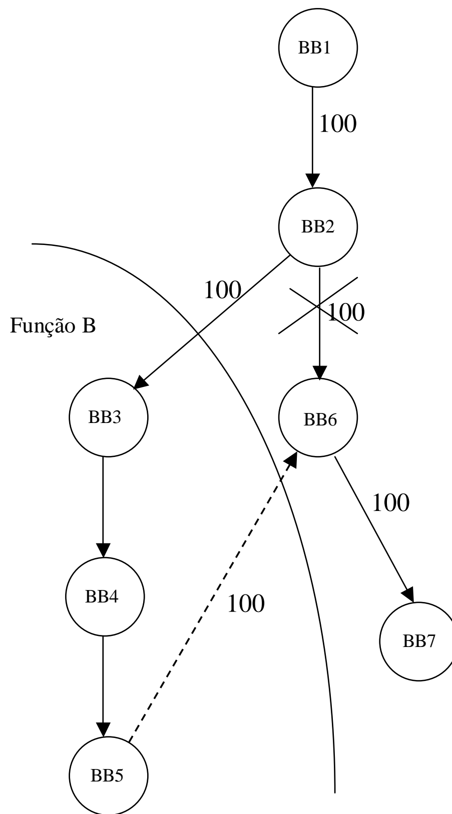


Figura 4.8: Retorno da função

Problema 3: Transformar o arquivo de perfil em um grafo ponderado e orientado.

A partir das informações do arquivo de perfil é construída uma representação do grafo de fluxo do controle (GFC), onde cada bloco básico corresponde a um vértice, cujo rótulo é o endereço inicial do respectivo bloco básico, e a cada possível caminho do fluxo de controle corresponde uma aresta, cujo rótulo (peso da aresta) é a freqüência com que o respectivo caminho foi tomado.

Para facilitar algumas análises, que devem ser feitas na ordem crescente de endereços, os vértices do GFC são indexados pelos seus rótulos (endereços dos blocos básicos). Para tanto, a estrutura de dados escolhida foi uma árvore binária de busca AVL que possui esse nome graças as iniciais dos seus inventores Adelson, Velskii and Landis. A árvore AVL permite a inserção e a busca de nós em tempo logarítmico, bem como a visita de forma simples a todos os nós, em ordem crescente dos rótulos, através de um percurso simétrico como pode ser visto na Figura 4.10, existe uma árvore AVL onde cada rótulo da árvore possui um ponteiro para um nó do grafo que

```
%FUNCTION{main}{8}{
      %CALLSITES{
      }
      %BBL{32}{
            %COUNT{0}
            %I{ 128  0x080481e0  pushl %ebp }%I
            %I{ 129  0x080481e1  movl %ebp <- %esp }%I
            %I{ 130  0x080481e3  pushl %esi }%I
            %I{ 131  0x080481e4  pushl %ebx }%I
            %I{ 132  0x080481e5  subl %esp <- $8 }%I

      }
      %BBL{37}{
            %COUNT{0}
            %I{ 156  0x08048227  addl %esp <- $16 }%I
            %I{ 157  0x0804822a  cmpl %ebx,$3 }%I
            %I{ 158  0x0804822d  jle 0x0804823d }%I
      }

%EDGES{{32 -> 33 : 0}{33 -> 34 : 0}{34 -> 35 : 0}{35 -> 36 : 0}
{36 -> 37 : 0}{37 -> 29582 : 0}{37 -> 38 : 0}{38 -> 39 : 0}
{39 -> 29582 : 0}{29582 -> 40 : 0}{40 -> 41 : 0}
}
%FUNCTION{readfile}{9}{
      %CALLSITES{
            %CALLER{main{8},38,48,0}
      }
      %CHECKED{no}
      %BBL{27540}{
            %COUNT{-99999}
      }
      %BBL{27541}{
            %COUNT{-99999}
      }
```

Figura 4.9: Arquivo sloop

possui o mesmo rótulo da árvore. Uma árvore binária é denominada AVL quando, para qualquer nó dessa árvore, as alturas de suas subárvores, esquerda e direita, diferem em módulo de até uma unidade. O percurso em ordem simétrica é muito utilizado para árvores binárias de busca, primeiro percorre a subárvore esquerda, em ordem simétrica, depois visita a raiz e por último percorre a subárvore direita, em ordem simétrica [70, 71].

Problema 4: Eliminar loops no grafo.

É muito comum que programas tenham *loops*, que se refletem em ciclos no GFC. Isso exige modificações na estrutura do GFC, pois é necessário que o grafo seja acíclico para ser tratado como uma rede *Bayesiana*. Para eliminar ciclos, foi utilizada a pesquisa por profundidade (DFS *depth first search*), associada à exclusão das arestas de retorno. Esse algoritmo associa cada vértice a um dos 3 estados: não-visitado, descoberto e terminado. Sempre que visitarmos um vértice e seu estado é descoberto

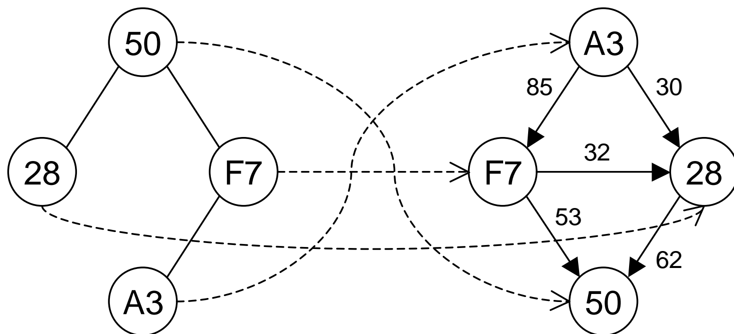


Figura 4.10: Árvore binária de busca AVL

é porque existe uma aresta de retorno e essa aresta é eliminada [14].

Um problema na eliminação de *loops* ocorre quando uma aresta for eliminada. Nesta situação serão criados dois vértices, um *source* e um *sink* para que a soma dos pesos das arestas de entrada de um vértice seja igual a soma das arestas de saída, como acontecia antes da eliminação da aresta. Um exemplo pode ser visto na Figura 4.11, onde um *loop* envolvendo o bloco básico D e o bloco básico B. Quando a aresta for eliminada, vão ser criados dois blocos básicos D' e B' com o mesmo peso da aresta que faz o retorno do *loop*.

Uma precaução especial é tomada quando os vértices *sink-source* precisarem ser atualizados no algoritmo de simulação estocástica. O vértice *source* nunca é atualizado e sim copiado do seu par *sink* sempre que esse for atualizado. Isso visa a assegurar a consistência semântica que a criação das duas variáveis sugerem.

Gerar rede Bayesiana atual

Após a obtenção do grafo acíclico ponderado, será calculada a probabilidade condicional inicial de todos os vértices do grafo que foram gerados a partir do arquivo perfil. A probabilidade condicional de um vértice é calculada através do valor da aresta do seu pai dividido pela soma dos valores das arestas de todos os filhos do seu pai, conforme mostrado nas Figuras 4.12 e 4.13. Todos os vértices de uma rede *Bayesiana* possuem aleatoriamente um valor binário 0 ou 1. Quando o valor do vértice for 1 significa que o bloco básico correspondente é executado e quando for 0 ele não foi executado. Cada vértice (Y) da rede *Bayesiana* armazena uma tabela semelhante a da Figura 4.1, que contém as probabilidades do vértice Y (possuir valor 0 ou 1) dado todas as combinações possíveis dos valores (0 ou 1) de seus pais.

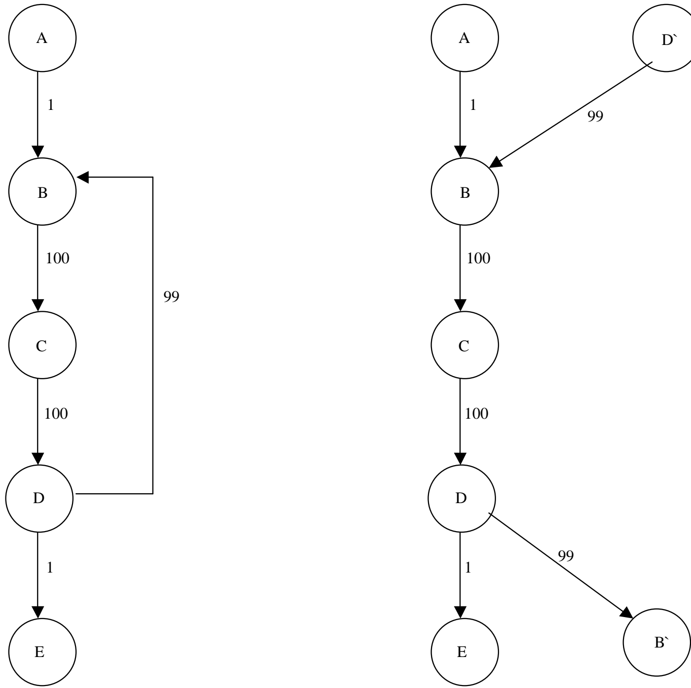


Figura 4.11: Eliminação de loops no grafo

Gerar rede Bayesiana histórica

Nosso problema agora é armazenar as probabilidades condicionais dos vértices que possuem centenas de pais, ja que teríamos que armazenar para cada vértice da rede *Bayesiana* uma tabela com $2^{|\Pi_X|}$ probabilidades. Para resolver esse problema vamos utilizar um dos modelos mais comuns que é a interação disjuntiva (*Noisy-OR gate*), e é largamente utilizada em redes *Bayesianas*. Tal interação ocorre quando qualquer membro de um conjunto de condições é causa provável de um certo evento, e esta probabilidade não diminui quando muitas dessas condições prevalecem simultaneamente.

Assim sendo, os *noisy-or gates* são usados para descrever a interação entre n causas $C_1, C_2, \ldots, C_n$ e seu efeito comum E, como pode ser visto na Figura 4.14, assumindo-se que cada uma das causas C_i é suficiente para causar E na ausência das demais, e que a habilidade que cada uma delas tem de causar E é independente da presença das demais.

Vamos supor que p_i representa a probabilidade de que o efeito E seja verdadeiro se a causa C_i estiver presente e todas as outras causas $C_j, i \neq j$, estiverem ausentes. Em outras palavras, $p_i = P(E = 1|\bar{C}_1, \bar{C}_2, \ldots, C_i = 1, \ldots, \bar{C}_{n-1}, \bar{C}_n)$. É fácil verificar que a probabilidade de E, dado um subconjunto dos C_i's que estão presentes, é

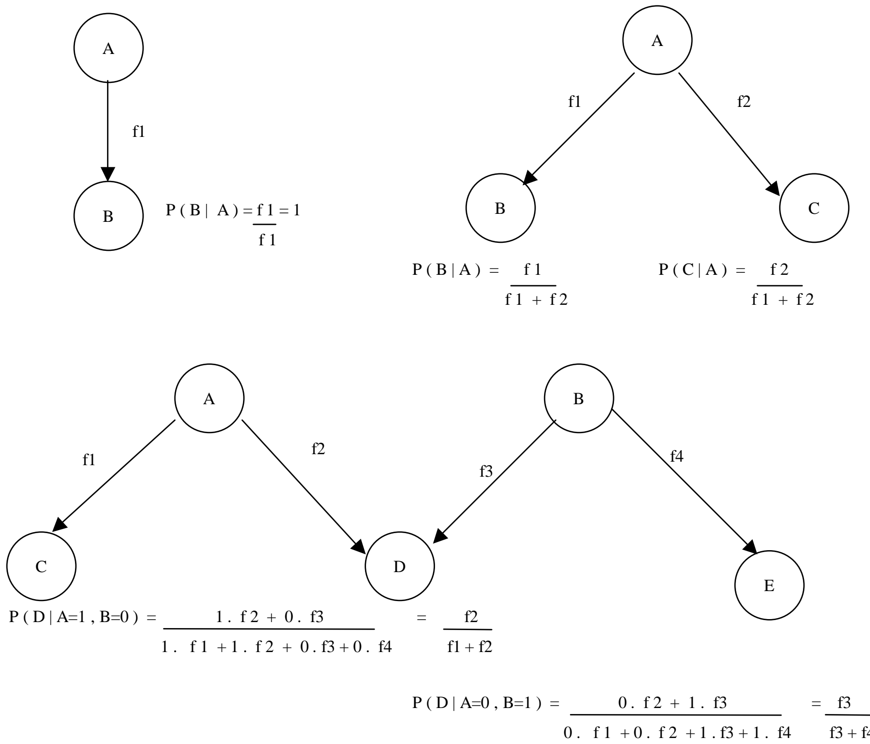


Figura 4.12: Probabilidades iniciais da rede *Bayesiana*

dada pela seguinte fórmula:

$$P(E = 1 \mid C_1 = 1, \ldots, C_n = 1) = 1 - \prod_{i=1}^{n}(1 - p_i) \tag{4.2}$$

A tabela na Figura 4.14 mostra claramente como foi calculada a Fórmula 4.2 do Noisy-OR. A probabilidade de $E = 1$ dado que somente $C_3 = 1$ é $\mathbf{a}$, caso $E = 0$ vai ser o complementar que é $\mathbf{1} - \mathbf{a}$ e assim sucessivamente. A probabilidade de $E = 0$ quando $C_2 = 1$ e $C_3 = 1$ é $(\mathbf{1} - \mathbf{a})(\mathbf{1} - \mathbf{b})$, a de $E = 1$ é o complementar $\mathbf{1} - [(\mathbf{1} - \mathbf{a})(\mathbf{1} - \mathbf{b})]$. Generalizando, a Fórmula 4.2 é obtida. Esta fórmula é suficiente para derivar a tabela de probabilidades condicionais de E condicionado aos seus pais $C_1, C_2, \ldots, C_n$.

Um exemplo numérico do cálculo do *Noisy-OR* pode ser visto na Figura 4.15, onde as probabilidades iniciais condicionais do vértice são calculadas de acordo com os valores de seus pais. Primeiramente é calculada a probabilidade condicional do vértice D ser executado, dado que o vértice A foi executado e o vértice B não. Depois é calculada a probabilidade condicional do vértice D ser executado, dado que o vértice B foi executado e o vértice A não. Esses valores são armazenados em um vetor com apenas duas posições (o vértice possui exatamente 2 pais). A probabilidade do

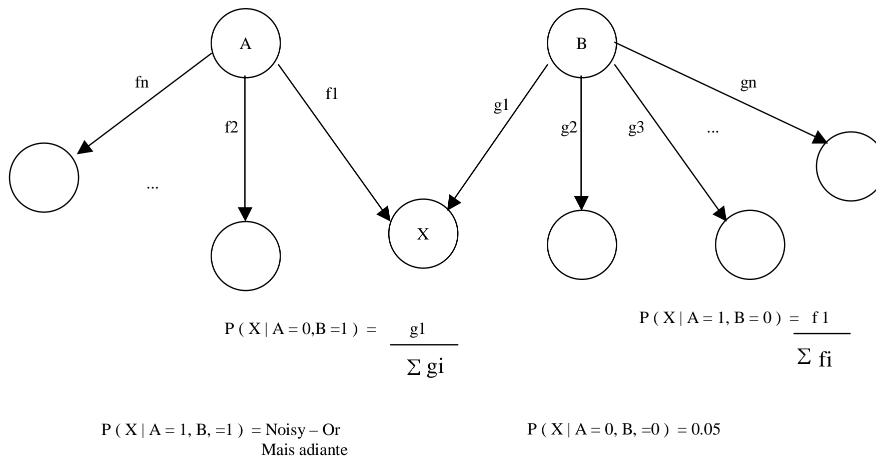


Figura 4.13: Generalizando as probabilidades iniciais

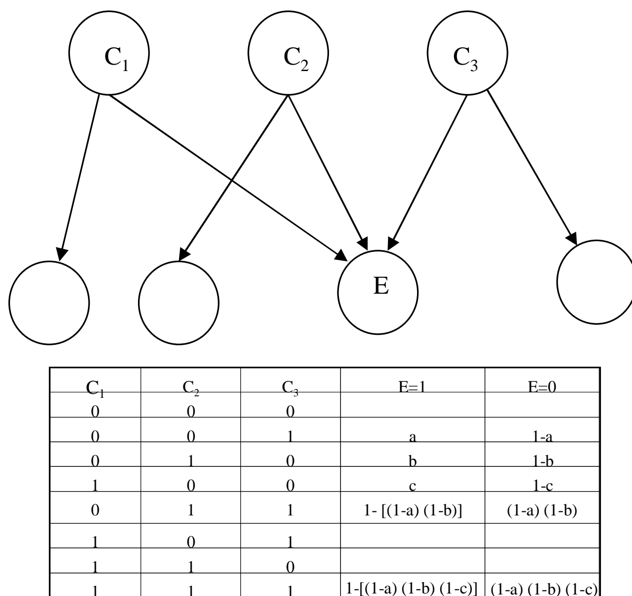
vértice D ser executado dado que os vértices A e B foram executados, é calculada durante o processamento pela Fórmula 4.2 e não será necessário guardar esse valores em arquivos. Por isso só será necessário guardar em arquivos Π_X probabilidades como mostra a Figura 4.15.

4.4.3 Otimização

A fase de otimização pode ser dividida em duas partes (vide Figura 4.16), a primeira para encontrar o *hot path* e atualizar o arquivo de perfil e a segunda para reordenar os blocos básicos (que será auxiliada pelo PLTO) e gerar o arquivo otimizado. Esta ferramenta terá como entrada o arquivo de perfil atualizado e o programa relocável.

Na primeira fase, o objetivo é descobrir o caminho mais provável que o programa foi executado (*hot path* do programa). No exemplo da Figura 4.17, os vértices marcados em preto mostram os blocos básicos que devem ficar adjacentes na memória, ou seja , é gerada uma seqüência em linha reta do código de acordo com o *hot path* do programa, aumentando a chance dos blocos básicos ficarem na mesma página. Esse *hot path* foi gerado pelo algoritmo de simulação estocástica relatado na Seção 4.1.2.

Após determinar o *hot path* do programa, o arquivo de perfil será atualizado. O que será atualizado é o valor da freqüência das arestas que fazem parte do *hot path*. Verificamos o valor da aresta com maior freqüência (MAX) em todo arquivo de perfil e multiplicamos esse valor por 10. Esse valor será dado a primeira aresta do *hot*



C_1	C_2	C_3	E=1	E=0
0	0	0		
0	0	1	a	1-a
0	1	0	b	1-b
1	0	0	c	1-c
0	1	1	1- [(1-a) (1-b)]	(1-a) (1-b)
1	0	1		
1	1	0		
1	1	1	1-[(1-a) (1-b) (1-c)]	(1-a) (1-b) (1-c)

Figura 4.14: Tabela do cálculo do Noisy-Or

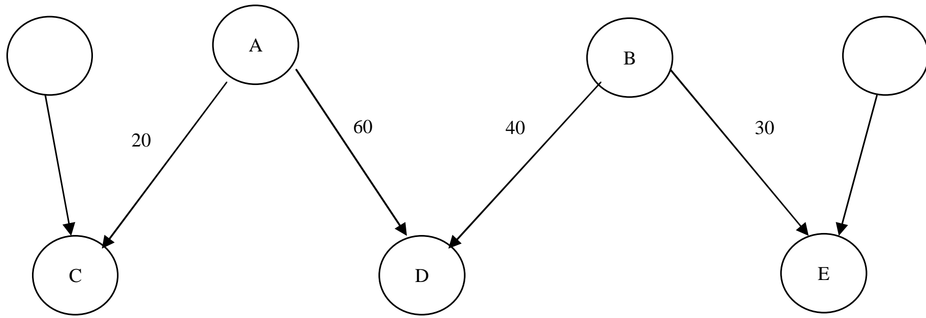
path encontrado e decrementando esse valor numa PA que possui o primeiro termo 10 MAX e último termo $MAX + 1$. Cada freqüência das arestas que fazem parte do *hot path* do programa receberá um valor dessa PA e as outras arestas continuam com o mesmo valor.

A segunda parte do processo de otimização realiza a reordenação dos blocos básicos que vão ficar adjacentes de acordo com o *hot path* encontrado na fase anterior. Esse arquivo de perfil atualizado juntamente com o arquivo relocável do programa vai servir de entrada para o PLTO reordenar os blocos básicos na memória. O PLTO usa a filosofia de Pettis e Hansen para a reordenação de blocos básicos na memória, como já foi explicado na Seção 3.1, do Capítulo 3.

4.5 Modelo histórico

Quando estamos falando de uma seqüência de entradas $I_1, I_2, \ldots$ sendo executadas em um determinado programa P, cujos tempos de execução são $t_1, t_2, \ldots$, não podemos aplicar um peso (α) igual a todos eles como se estivessem sendo examinados isoladamente, já que programas com tempos de execução muito abaixo da média histórica devem influenciar menos o histórico e programas acima devem influenciar mais.

Seja T_{i-1} uma média ponderada das $i-1$ primeiras execuções. Se α_i é o peso com que o i-ésimo programa (com seu respectivo arquivo de entrada de dados) influenciará o histórico, então, com $\gamma > 0$, fazemos



$P(D\,|\,A=1, B=0) = \frac{60}{20+60} = \frac{60}{80} = 0.75$ $\qquad$ $P(D\,|\,A=0, B=1) = \frac{40}{40+.30} = \frac{40}{70} = 0.57$

Vértice D = | 0.57 | 0.75 | Tamanho do vetor = Nº de pais

Noisy - or

$P(D\,|\,A=1, B=1) = 1-[(1-\frac{4}{7})(1-\frac{3}{4})] = 1-[(\frac{3}{7})(\frac{1}{4})] = 1-[\frac{3}{28}] = \frac{25}{28} = 0.89$

Figura 4.15: Cálculo do Noisy-Or

$$T_{i-1} = \frac{\alpha_1 t_1 + \cdots + \alpha_{i-1} t_{i-1}}{\alpha_1 + \cdots + \alpha_{i-1}}, \tag{4.3}$$

e

$$\alpha_i = \frac{1}{1 + (\frac{1-\alpha_0}{\alpha_0})e^{-\gamma(t_i - T_{i-1})}}. \tag{4.4}$$

Cada α_i funciona com o peso de t_i no cálculo de T_i conforme a Fórmula 4.3. Note que a primeira entrada necessariamente influencia o tempo histórico com peso 1 (para ele, tudo ocorre como se estivesse isolado), fornecendo $T_1 = t_1$. A partir daí os pesos são obtidos segundo a Equação 4.4, essa função sigmóide assume um intervalo contínuo de valores entre 0 e 1 e o valor de γ serve para controlar a inclinação da sigmóide, ilustrações com $\gamma = 1$ são apresentadas na Figura 4.18.

A função sigmóide sempre garante valor α_0 para os programas cujos tempos de execução sejam iguais à média ponderada corrente, isto é, $\alpha_i = \alpha_0$ quando $t_i = T_{i-1}$, ou seja quando P roda a i-ésima entrada tão rápido quanto sua média; os valores menores de t_i traz α_i para mais perto de 0, e valores maiores o trazem para mais perto de 1.

Para $i > 1$ (ou seja, depois de P ter sido rodado ao menos uma vez), o modelo

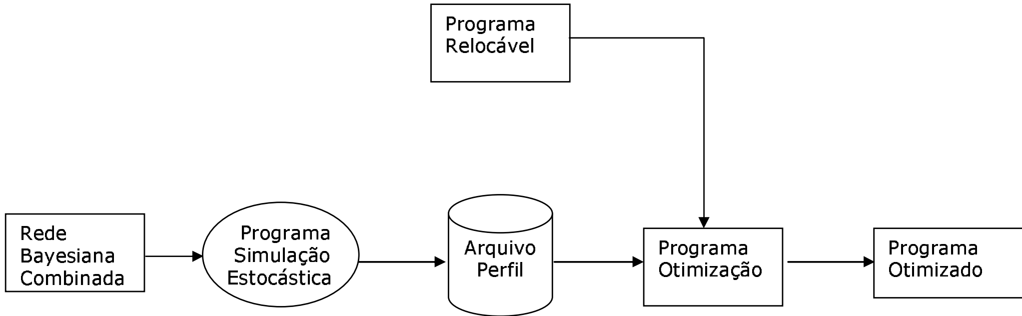


Figura 4.16: Fase 3 - otimização

histórico das primeiras $i-1$ execuções é uma rede *Bayesiana* histórica, denotado por H_{i-1}, que incorpora conhecimentos probabilísticos das ocorrências dos blocos básicos de P conforme é executado. Agora descreveremos como incorporar o conhecimento probabilístico que B_i (rede *Bayesiana* atual) incorpora sobre a i-ésima execução de P em H_{i-1}, de modo que H_i (rede *Bayesiana* combinada), agora com informações das primeiras i execuções incorporadas, possa ser obtida.

Para alcançar a combinação de H_{i-1} e B_i em H_i, primeiro devemos nos assegurar de que as duas redes *Bayesianas* com as quais iremos trabalhar têm o mesmo conjunto de vértices e arestas. Isto pode ser alcançado determinando primeiramente a união dos dois conjuntos de vértices e da união dos dois conjuntos de arestas, e depois ampliar cada conjunto de vértices para torná-lo igual à união dos dois conjuntos, depois similarmente para o conjunto de arestas.

O único problema disso é que deixa alguns rótulos de vértices incompletos. Por exemplo, em H_{i-1} e B_i pode haver uma variável X com menos de Π_X (número de pais de X) probabilidades condicionais especificadas para ele. Cada probabilidade que falta é uma probabilidade em que $X = 0$, dado que criou um novo pai com valor 1 e deixou todos os outros com valor 0. O que fazemos nesses casos é atribuir a todas as probabilidades que faltam um valor bem pequeno $\epsilon \in (0, 1)$, foi determinado o valor de 0.05. [1]

Podemos então assumir, por causa dos argumentos desta seção, que H_{i-1} e B_i têm os mesmos vértices e o mesmo conjunto de arestas, e também que ele têm valores nos vértice completamente especificados dentro da técnica *Noisy-OR*. O conjunto de

[1]É importante que ϵ seja um valor estritamente positivo. Definir essas probabilidades em 0 romperia a natureza fundamental de uma rede *Bayesiana*, e nesse caso toda a teoria em que se baseia o processo de otimização se perderia.

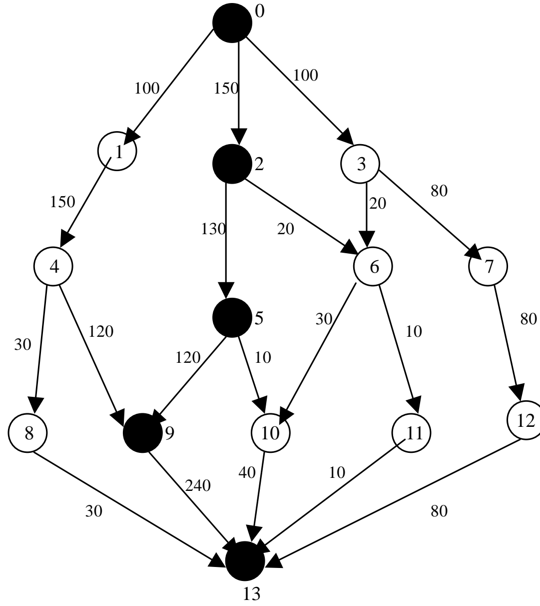


Figura 4.17: Caminho mais provável de ser executado

arestas e vértices são também os vértices e arestas da rede *Bayesiana* combinada, H_i.

Toda nossa discussão é relacionada à evolução do modelo do histórico para $i > 1$. Para $i = 1$, nenhum modelo histórico existe e B_1 simplesmente se torna H_1 enquanto deixamos que $T_1 = t_1$. Além disso, determinando α_1 para uso posterior requer que T_0 (tempo médio inicial) e α_0 (peso inicial) sejam conhecidos. Descobrimos experimentalmente que o melhor valor para o peso inicial é 0.8 e o tempo médio inicial foi inicializado com 0 (esse valor não altera o desempenho da otimização).

Agora para $i > 1$, nós gostaríamos de obter H_i como sendo a média geométrica das probabilidades de cada vértice de H_{i-1} e B_i. Cada variável (vértice do grafo, X) da rede *Bayesiana* possui um valor binário 0 ou 1 que vamos denotar por x e π_X é o conjunto de valores (0 ou 1) dos seus pais. Cada vértice de uma rede *Bayesiana* contém uma tabela de probabilidades condicionais, como já foi descrito anteriormente.

Seja p_i a distribuição de probabilidade para B_i e q_i a distribuição de probabilidade para H_i, e seja $\alpha_i \in (0, 1)$. Então

$$q_i(x \mid \pi_X) = \frac{q_{i-1}(x \mid \pi_X)^{1-\alpha_i} p_i(x \mid \pi_X)^{\alpha_i}}{Z}, \tag{4.5}$$

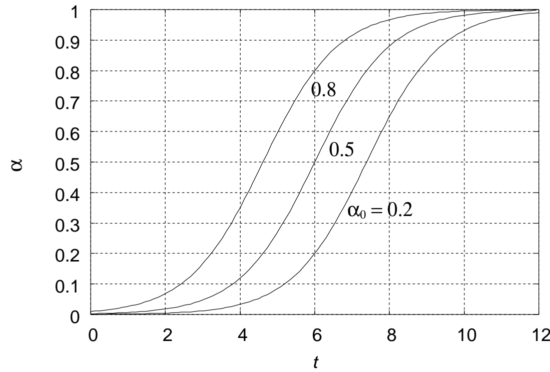


Figura 4.18: Gráfico do $\alpha = \left(1 + e^{T-t}(1-\alpha_0)/\alpha_0\right)^{-1}$ com $T = 6$ para $\alpha_0 = 0.2, 0.5, 0.8$.

onde Z é uma constante de normalização.

Nossa estratégia é resumida a seguir:

1. Executar o programa P com a entrada I_1:
   (a) Determinar α_1 de (4.4).
   (b) $T_1 = t_1$.
   (c) Duplicar a rede *Bayesiana* B_1 para produzir a rede Bayesiana histórica H_1.
   (d) Resolver H_1 identificando os blocos básicos que são mais executados e reordenando os blocos básicos de P de acordo.
2. Para $i > 1$, rodar P com a entrada I_i:
   (a) Determinar α_i de (4.4).
   (b) Fazer
   $$T_i = \frac{\alpha_1 t_1 + \cdots + \alpha_i t_i}{\alpha_1 + \cdots + \alpha_i}$$
   (c) Obter o conjunto de vértices e arestas de H_i como a união, respectivamente, do conjunto de vértices e arestas de H_{i-1} e B_i.
   (d) Calcular as média geométricas conforme Equação 4.5 para todos os vértices da rede *Bayesiana* combinada.
   (e) Resolver H_i identificando os blocos básicos que são mais executados e reordenando os blocos básicos de P de acordo.

Resolver o modelo histórico nos passos 1(d) e 2(e) significa executar a variação da simulação estocástica mencionada anteriormente, e examinar todas as variáveis que têm valor 1 ao final da solução do modelo histórico.

Partimos da premissa que existem dois ambientes, como foi citado anteriormente, e pode ser visto na Figura 1.1: um ambiente de produção e outro ambiente de teste. No ambiente de teste é executada uma versão instrumentada do programa para encontrar e gravar o *hot path* do modelo histórico. São procedimentos caros, então um dos objetivos é quanto à praticidade desta técnica numa situação real. Nossa visão aqui é que a estratégia, resumida anteriormente, não está sendo aplicada à seqüência de todas as entradas que vêm junto com a execução do programa P no ambiente de produção, mas sim numa subseqüência daquela seqüência, como o exemplo que segue.

Quando a entrada $I1$ chega, duas instâncias de P são executadas, uma em cada ambiente. A primeira instância no ambiente de produção não é instrumentada e retorna o resultado da execução assim que ele se torna disponível. A segunda instância no ambiente de teste, por sua vez, é instrumentada e rende um modelo de execução a ser incorporado no modelo histórico de P. Novas entradas que aparecem no meio tempo só são executadas no ambiente de produção. Uma vez que o modelo histórico de P foi atualizado e resolvido, um código reordenado correspondente é obtido, uma nova entrada de P pode novamente disparar duas execuções de P (uma em cada ambiente), mas agora empregando o novo código reordenado. Nesta visão, um sistema de *background* pode ser dedicado para manter modelos históricos e de tempos em tempos atualizar com versões atualizadas do programa.

Capítulo 5

Experimentos e resultados

Realizamos experimentos para avaliar o desempenho da estratégia relatada no Capítulo 4. Nosso objetivo foi duplo: primeiro, verificar a habilidade de fornecer melhores tempos de execução enquanto o programa é repetidamente executado com a mesma entrada, possivelmente com a intervenção de outras entradas; segundo, comparar os tempos de execução com aqueles obtidos com a otimização gcc nível 3 (sem nenhuma reordenação) e com a reordenação de Pettis-Hansen (conforme implementado no PLTO).

No restante do trabalho, nos referimos às duas últimas estratégias pelos títulos O3 e PH, respectivamente. A estratégia PH já foi explicada no Capítulo 3. Concentramos os experimentos nos programas do SPEC2000. Foi utilizado o PAPI (*performance application programming interface*) [49] para fazer análise de baixo nível dos programas.

5.1 Características de hardware e software

Os experimentos foram realizados em um processador AMD Athlon XP 2000, 2.0 GHZ com 256 MB de memória principal, memória *cache* L1 separada de dados e instruções de 128 KB, *cache* L2 de 256 KB e 1 GB de espaço em disco para *swap*. O sistema operacional usado foi RedHat Linux 7.3, kernel 2.4.18-3. Os programas foram compilados com o gcc versão 2.96 e nível de otimização O3.

Os tempos de execução para todos os programas foram obtidos executando cada programa 3 vezes e descartando o maior e o menor tempo. O tempo de execução foi medido usando o comando *time* do Linux que fornece o tempo de execução do programa. Todos os programas foram medidos com esse tempo que é muito utilizado no meio acadêmico.

O benchmark SPEC2000 *(Standard Performance Evaluation Corporation)* [65] avalia o desempenho do sistema, principalmente do processador, arquitetura de memória e compilador. O SPEC2000 possui 12 programas para inteiros conforme Tabela 5.1 (omitimos o eon e o perl de nossos experimentos porque existe uma incompatibilidade com o PLTO), cada programa possui vários conjuntos de entrada. Cada entrada dos nossos experimentos faz parte de um dos 4 conjuntos, conforme Tabela 5.2: o conjunto *reference*, *train*, *test* e *reduced.* O conjunto de entrada *reference* é o que faz o teste mais completo e demorado. Em seguida vem o conjunto de entrada *train* que também é um conjunto de teste bem aceito no meio acadêmico e o conjunto *test e reduced* são testes bem mais rápidos. Cada um desses conjuntos possui um determinado número de entradas numeradas conforme Tabela 5.2.

Programa	Descrição
bzip2	Compressão de arquivos (ordenação de blocos)
crafty	Jogo de xadrez
gap	Um pacote de aplicações de teoria de grupo
gcc	Compilador GNU C
gzip	Compressão de arquivos
mcf	Otimização combinatória de controle de trânsito
parser	Analisador sintático do idioma inglês
twolf	Simulador de posição e rota VLSI
vortex	Banco de dados orientados a objetos
vpr	Posicionamento e roteamento de circuito FPGA

Tabela 5.1: Descrição dos programas do SPEC2000

Seja ne o número de entradas distintas de um programa do SPEC2000, nas próximas tabelas, as entradas serão numeradas de 0 até $ne - 1$. Conforme Tabela 5.7, o gcc tem 9 arquivos de entrada, numerados de 0 até 8. Nossa metodologia para experimentação foi aplicar à rede *Bayesiana* a seqüência $I_0, \ldots, I_{ne-1}$ de entradas geradas aleatoriamente, de modo que cada uma das ne entradas aparece exatamente 6 vezes na seqüência, totalizando 6 ne execuções do programa.

A Tabela 5.3 apresenta as características de cada programa do SPEC2000 em relação ao número de instruções, número de blocos básicos e número de arestas.

Quando utilizamos o algoritmo de simulação estocástica para resolver a nossa modelagem, foi utilizado o *Simulated Annealing* (SA). Os parâmetros utilizados no SA foram: temperatura inicial igual a 10^4, temperatura final igual a 1, o valor de

Programa	Conjunto de Entradas			
	REF	TRAIN	TEST	RED
bzip2	0,1,2	3	4	5,6,7
crafty	0	1	2	3,4,5
gap	0	1	2	3,4,5
gcc	0,1,2,3,4	5	6	7,8
gzip	0,1,2,3,4	5	6	7 a 21
mcf	0	1	2	3,4
parser	0	1	2	3,4,5
twolf	0	1	2	3,4
vortex	0,1,2	3	4	5,6,7
vpr	0,1	2,3	4,5	6,7,8,9

Tabela 5.2: Entrada SPEC2000

Programa	N.Instrução	N.BB	N.Arestas
bzip2	100.964	24.758	34.094
crafty	135.691	32.249	45.097
gap	242.666	48.168	65.500
gcc	423.765	120.348	189.327
gzip	107.515	24.990	34.434
mcf	89.665	21.962	30.364
parser	112.403	30.130	41.546
twolf	144.573	38.974	56.094
vortex	213.637	58.750	87.017
vpr	125.610	28.885	40.350

Tabela 5.3: Análise dos programas

β foi de 0.98 que é a razão da redução geométrica. Esses valores foram descobertos experimentalmente.

Podemos verificar que T recebe $\lceil 1 - \ln 10^4 / \ln 0.98 \rceil = 457$ valores continuamente decrescentes. Para cada valor, cada variável do modelo pode ser atualizada de acordo com o algoritmo de simulação estocástica do Capítulo 4.

Nossa metodologia para os experimentos foi aplicar a rede *Bayesiana* na seqüência de entradas dos programas do SPEC2000; primeiro escolhemos os valores de α_0 (peso inicial). Quando os programas são olhados isoladamente, podemos observar nas Tabelas 5.4, 5.5 e 5.6, que o melhor peso (α_0) encontrado para a execução do programa atual é 0.8, na maioria do experimentos, foram os melhores resultados

obtidos, isto é, os menores tempos (em segundos).

Peso		Entradas				
Hist.	Atual	0	1	2	3	4
0.0	1.0	41.13	21.55	51.04	42.47	72.07
0.1	0.9	40.76	20.35	50.67	41.15	70.91
0.2	0.8	39.85	19.80	49.79	40.83	70.70
0.3	0.7	41.03	20.00	51.23	42.42	72.73
0.4	0.6	39.04	19.98	49.98	41.04	71.36
0.5	0.5	39.99	19.82	49.79	41.10	71.39
0.6	0.4	40.92	19.93	49.94	41.19	70.97
0.7	0.3	41.34	19.92	50.18	41.25	70.48
0.8	0.2	40.11	20.01	50.04	41.48	71.06
0.9	0.1	40.12	20.10	50.12	41.49	71.07

Tabela 5.4: Tempos (seg.) do programa gzip e entrada ref

Peso		Entradas	
Hist.	Atual	0	1
0.0	1.0	1.23	0.75
0.1	0.9	1.21	0.74
0.2	0.8	1.18	0.74
0.3	0.7	1.21	0.74
0.4	0.6	1.18	0.74
0.5	0.5	1.21	0.74
0.6	0.4	1.21	0.74
0.7	0.3	1.21	0.75
0.8	0.2	1.22	0.75
0.9	0.1	1.22	0.75

Tabela 5.5: Tempos (seg.) do programa vpr e entrada test

5.2 Evolução dos programas

Vamos separar nossos resultados por programa do SPEC2000. Cada programa terá duas tabelas e um gráfico com os dados da segunda tabela.

A primeira tabela de cada programa vai informar o tempo de execução (em segundos) do programa em cada uma de suas entradas, isto é, o tempo de execução do algoritmo da tese em sua última execução ($t_{último}$), primeira execução (t_{prim}), tempo de execução de O3 (t_{O3}) e PH (t_{PH}).

Peso		Entradas				
Hist.	Atual	0	1	2	3	4
0.0	1.0	114.10	95.91	140.01	9.84	5.59
0.1	0.9	115.64	95.99	141.73	9.84	5.59
0.2	0.8	114.15	95.61	140.23	9.74	5.51
0.3	0.7	114.98	96.08	141.59	9.85	5.58
0.4	0.6	115.46	96.28	141.30	9.87	5.58
0.5	0.5	116.08	96.32	141.76	9.88	5.58
0.6	0.4	115.08	96.11	141.46	9.83	5.57
0.7	0.3	11543	97.01	141.78	9.86	5.60
0.8	0.2	114.96	96.08	140.08	9.85	5.58
0.9	0.1	115.43	97.76	142.54	9.86	5.60

Tabela 5.6: Tempos (seg.) do programa vortex e entrada test, train e ref

Na primeira tabela de cada programa, a primeira coluna contém a entrada que estamos analisando, variando de 0 até $ne - 1$ (número de entradas); a segunda e terceira coluna são o tempo de execução (em segundos) da última execução e da primeira execução do algoritmo da tese, respectivamente. A quarta coluna é o percentual de melhora ou piora (negativo) da segunda coluna em relação à terceira coluna, isto é, como foi o ganho na evolução do histórico de cada programa. A quinta coluna contém o tempo de execução (em segundos) da otimização O3; a sexta coluna o percentual de melhora ou piora (negativo) da última execução da tese (segunda coluna) em relação a O3. A penúltima coluna contém o tempo de execução (em segundos) da otimização PH, e a última coluna o percentual de melhora ou piora (negativo) da última execução do algoritmo da tese (segunda coluna) em relação à otimização de PH.

Calculamos os percentuais nas tabelas das colunas 4, 6 e 8 através das fórmulas:

$$g_{prim} = 100(t_{prim} - t_{\acute{u}ltimo})/t_{prim},$$

$$g_{O3} = 100(t_{O3} - t_{\acute{u}ltimo})/t_{O3}$$

e

$$g_{PH} = 100(t_{PH} - t_{\acute{u}ltimo})/t_{PH}$$

Com raríssimas exceções, ganhos positivos dominam as tabelas e indicam um desempenho superior de até **12.61%** em relação a primeira execução, **14.50%** em relação ao O3 e **12.96%** em relação ao PH. Ganhos sobre O3 tendem a ser maiores que aqueles em relação ao PH, com raras exceções.

Na segunda tabela de cada programa vamos ver exatamente como funciona a evolução do algoritmo da tese em relação ao histórico. Essa tabela vai servir também para interpretar os gráficos. Para cada entrada de um determinado programa vamos especificar o momento em que essa entrada foi utilizada na seqüência de evolução (*num*), isto é, a posição que uma determinada entrada será executada e seu tempo de execução (em segundos). Lembrando que cada entrada foi executada exatamente 6 vezes e a ordem foi escolhida aleatoriamente.

Cada um dos programas do SPEC2000 terá, além da tabela de execução, 2 gráficos para uma melhor visualização. Dividimos os gráficos conforme as entradas, isto é, colocamos no primeiro gráfico aquelas entradas (para os quais o tempo de execução é maior, geralmente o conjunto REF) que tendem melhorar notadamente o tempo de execução dos programas enquanto a seqüência de entradas é dada. Para as entradas do segundo gráfico de cada programa podemos notar que o tempo de execução teve uma melhora menor e em alguns casos é quase inalterado.

O gráfico possui na abscissa um número seqüencial variando de 1 até 6 ne, onde ne é o número de entradas do programa.

Vamos mostrar neste capítulo apenas os resultados de dois programas, os resultados dos outros oito programas serão colocados no Apêndice A, para este capítulo não se tornar extenso.

5.2.1 Programa gcc

Como podemos ver na Tabela 5.7, o algoritmo da tese teve uma melhora em algumas entradas de quase **15%**. Em média o programa teve um desempenho **7.75%** melhor do que o do compilador gcc e **4.26%** melhor que o algoritmo de Petis e Hansen.

Conforme mostram as Tabelas 5.7 e 5.8 e o Gráfico 5.1, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de até **12%** em algumas entradas. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **5.04%**.

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	80.15	83.53	4.05	85.89	6.68	83.31	3.79
1	86.43	89.98	3.95	92.82	6.89	89.78	3.73
2	8.78	9.98	12.02	10.27	14.50	8.95	1.90
3	14.73	15.90	7.35	15.92	7.47	14.97	1.60
4	46.98	50.20	6.41	52.02	9.69	50.82	7.56
5	3.01	3.29	8.51	3.23	6.81	3.14	4.14
6	1.16	1.21	4.13	1.31	11.45	1.16	0.00
7	0.32	0.32	0.00	0.35	8.57	0.32	0.00
8	0.06	0.06	0.00	0.06	0.00	0.06	0.00
Soma	241.62	254.47	5.04	261.93	7.75	252.38	4.26

Tabela 5.7: Tempo (seg.) de execução gcc

Entrada	gcc						
0	num	6	8	15	21	27	39
	tempo	83.53	82.49	81.51	80.73	80.01	80.15
1	num	4	5	9	16	19	20
	tempo	89.98	88.64	88.11	88.83	87.17	86.43
2	num	2	3	28	40	44	49
	tempo	9.98	9.23	8.70	9.20	8.37	8.78
3	num	11	12	14	18	22	42
	tempo	15.90	15.47	14.73	15.03	14.52	14.73
4	num	30	37	43	47	48	51
	tempo	50.20	49.48	49.01	48.83	48.11	46.98
5	num	1	13	25	33	38	50
	tempo	3.29	3.01	3.02	3.02	3.01	3.01
6	num	17	23	24	31	34	35
	tempo	1.21	1.18	1.18	1.16	1.16	1.16
7	num	7	10	29	45	46	54
	tempo	0.32	0.32	0.33	0.32	0.32	0.32
8	num	26	32	36	41	52	53
	tempo	0.06	0.06	0.06	0.06	0.07	0.06

Tabela 5.8: Evolução (seg.) do gcc

5.2.2 Programa gzip

Como podemos observar na Tabela 5.9, o algoritmo da tese teve uma melhora em algumas entradas de quase **11%**. Em média o programa teve um desempenho **9.44%** melhor do que o do compilador gcc e **7.19%** melhor que o algoritmo de Petis e

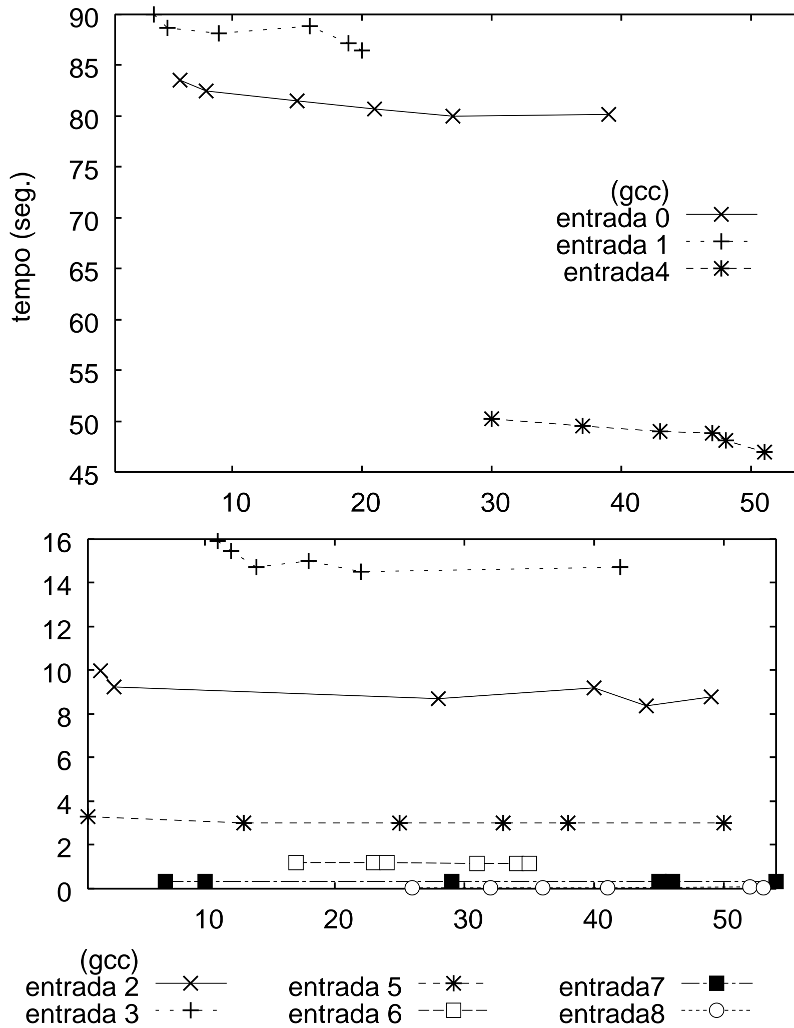


Figura 5.1: Performance da evolução do programa gcc tabela 5.8.

Hansen.

Conforme mostram as Tabelas 5.9 e 5.10 e no Gráfico 5.2, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de quase **9%** em algumas entradas.

Podemos verificar no gráfico que as entradas com tempo de execução muito baixo tiveram uma pequena melhora. Em média a melhora na evolução foi na ordem de **5.79%**.

No conjunto de entrada *ref e train* (entradas 0, 1, 2, 3, 4 e 5) tiveram uma melhora de **10%** em relação ao O3.

5.3 Resultados gerais

A Tabela 5.2 mostra o número de arquivos de entrada de cada um dos 4 conjuntos de entrada do SPEC2000 (conjuntos de entrada *ref, train, test e reduced*). Como

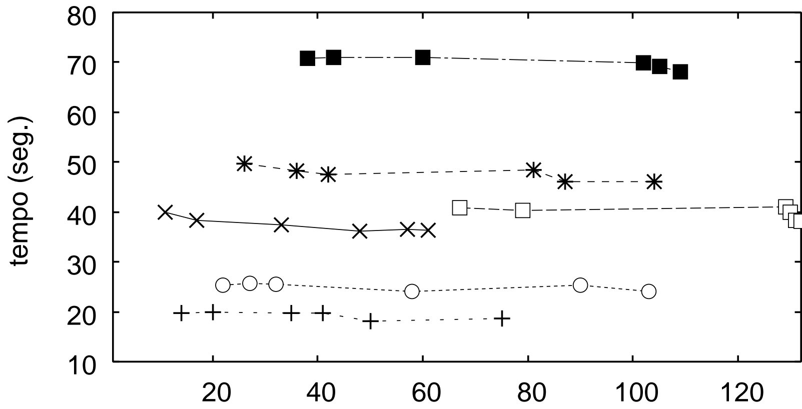
Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	36.38	39.87	8.75	40.77	10.77	40.13	9.34
1	18.65	19.83	5.95	20.72	9.99	19.89	6.23
2	46.11	49.76	7.34	51.14	9.84	50.06	7.89
3	38.11	40.87	6.75	42.52	10.37	41.34	7.81
4	68.08	70.76	3.79	74.26	8.32	72.23	5.74
5	24.14	25.30	4.58	26.96	10.46	26.56	9.11
6	0.00	0.00	0.00	0.00	0.00	0.00	0.00
7	0.74	0.74	0.00	0.76	2.63	0.74	0.00
8	0.27	0.27	0.00	0.29	6.90	0.27	0.00
9	0.82	0.83	1.20	0.85	3.53	0.84	2.38
10	0.68	0.68	0.00	0.70	2.86	0.68	0.00
11	1.23	1.23	0.00	1.27	3.15	1.25	1.60
12	0.70	0.73	4.11	0.73	4.11	0.71	1.41
13	0.28	0.28	0.00	0.30	6.67	0.28	0.00
14	0.54	0.54	0.00	0.57	5.26	0.54	0.00
15	0.68	0.69	1.45	0.72	5.55	0.69	1.44
16	1.01	1.01	0.00	1.05	3.80	1.02	0.98
17	0.69	0.69	0.00	0.71	2.82	0.70	1.43
18	0.29	0.29	0.00	0.30	3.33	0.29	0.00
19	2.23	2.24	0.45	2.32	3.88	2.27	1.76
20	0.68	0.68	0.00	0.71	4.23	0.69	1.44
21	1.41	1.42	0.71	1.49	5.37	1.44	2.08
Soma	243.72	258.71	5.79	269.14	9.44	262.62	7.19

Tabela 5.9: Tempo (seg.) de execução gzip

o conjunto *reduced* possui tempos muito pequenos (quase 0) não vamos analisar separadamente como faremos para os outros conjuntos.

Vamos analisar também dois programas do SPEC95 (compress e go). E também outros programas de usuários.

Nas Tabelas a seguir comparamos o tempo de execução do algoritmo da tese com as otimizações do gcc nível 3 (O3) e da otimização Petis-Hansen (PH). Outra análise que faremos é sobre a melhora utilizando o histórico, isto é, comparando a primeira execução com a última execução.



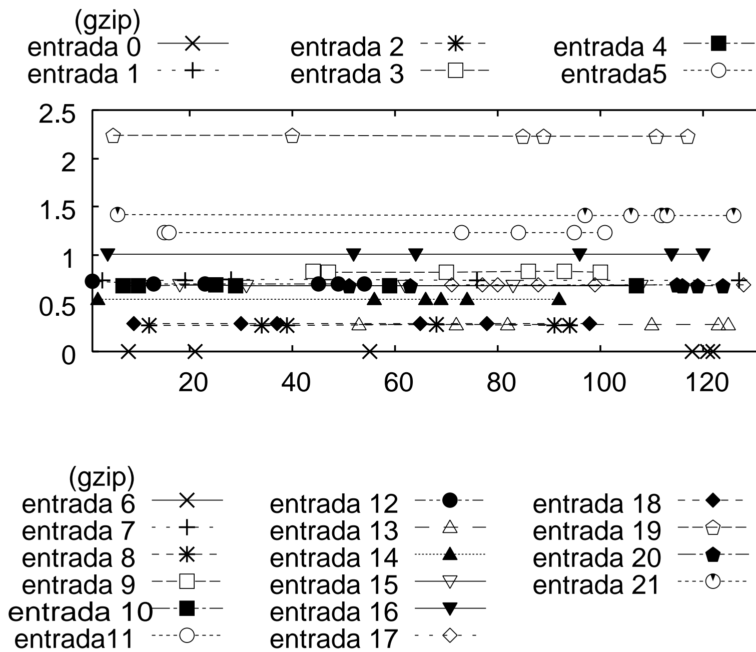


Figura 5.2: Performance da evolução do programa gzip tabela 5.10.

5.3.1 Conjuntos de entradas reference, train e test

Reference

Na Tabela 5.11 pode-se observar o tempo de execução de todos os programas do *benchmark* SPEC2000 com o conjunto de entrada *ref*, o tempo de execução do programa compilado com o otimizador O3, o tempo utilizando o algoritmo de Pettis-Hansen e o tempo de execução do algoritmo da tese. Na Figura 5.3 podemos ver o percentual de melhora do tempo da última execução do programa em relação as 3 otimizações (O3, PH e evolução histórico).

Podemos ver uma melhora em média de **5.45%** do algoritmo da tese em relação ao compilador gcc com otimização O3 e uma melhora em média de **3.26%** do algo-

ritmo da tese em relação ao algoritmo de Pettis e Hansen. Podemos observar uma melhora de **3.53%** na evolução dos programas, isto é, a relação entre a primeira e a última execução de um determinado programa.

Conseguimos melhores desempenhos nos programas gzip, gcc e vpr e esse ganho foi de quase **10%**.

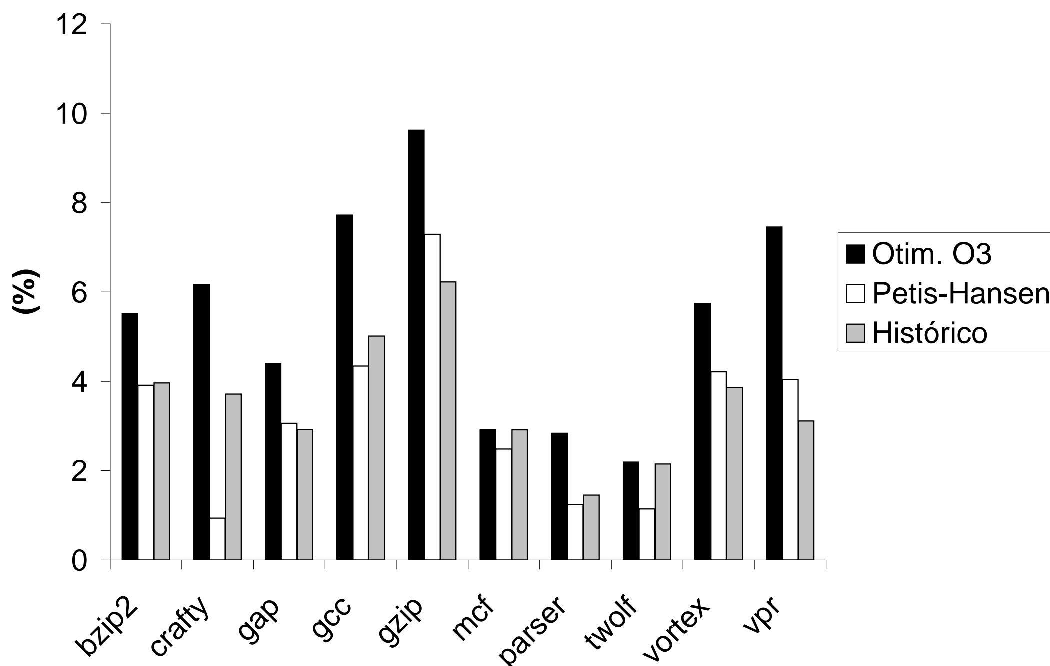


Figura 5.3: Programas do SPEC2000 conjunto de entrada reference

Train

Na Tabela 5.12 pode-se observar o tempo de execução de todos os programas do *benchmark* SPEC2000 com o conjunto de entrada *train*, o tempo de execução do programa compilado com o otimizador O3, o tempo utilizando o algoritmo de Pettis-Hansen e o tempo de execução do algoritmo da tese. Na Figura 5.4 podemos ver o percentual de melhora do tempo da última execução do programa em relação as 3 otimizações (O3, PH e evolução histórico).

Podemos ver uma melhora em média de **8.18%** do algoritmo da tese em relação ao compilador gcc com otimização O3, uma melhora em média de **5.79%** do algo-

ritmo da tese em relação ao algoritmo de Pettis e Hansen e uma melhora de **6.69%** na evolução dos programas, isto é, a relação entre a primeira e última execução de um determinado programa.

O conjunto train foi o que obtivemos os melhores resultados. Conseguimos melhores desempenhos novamente nos programas gzip e vpr, e também o programa bzip2 e crafty, e esse ganho foi de quase **13%**.

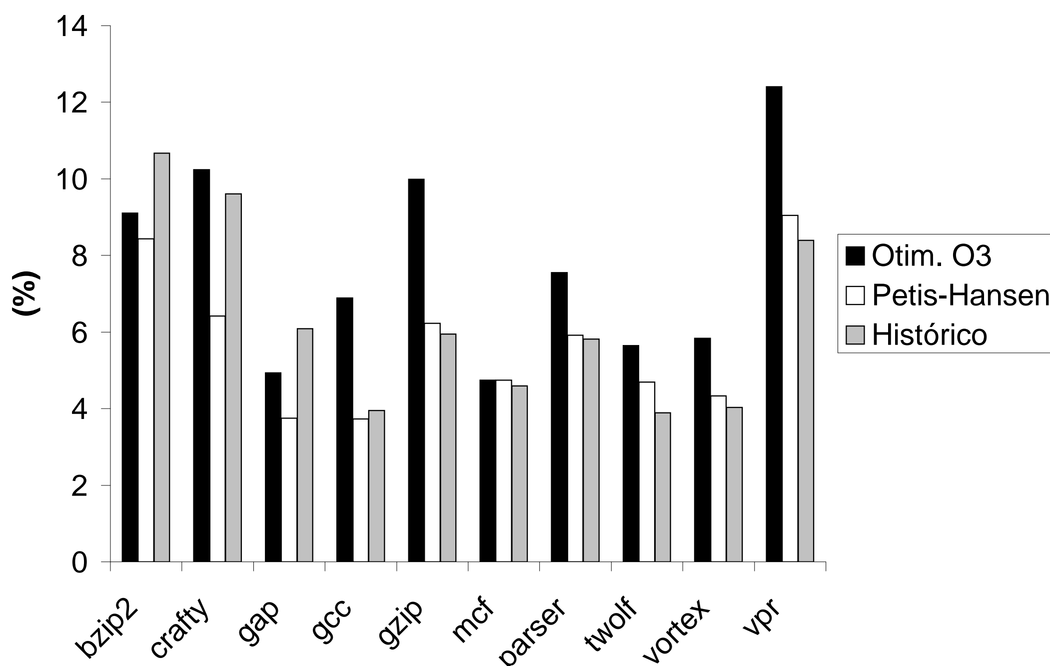


Figura 5.4: Programas do SPEC2000 conjunto de entrada train

Test

Na Tabela 5.13 pode-se observar o tempo de execução de todos os programas do *benchmark* SPEC2000 com o conjunto de entrada *test*, o tempo de execução do programa compilado com o otimizador O3, o tempo utilizando o algoritmo de Pettis-Hansen e o tempo de execução do algoritmo da tese. Na Figura 5.5 podemos ver o percentual de melhora do tempo da última execução do programa em relação as 3 otimizações (O3, PH e evolução histórico).

Podemos ver uma melhora em média de **5.9%** do algoritmo da tese em relação ao

compilador gcc com otimização O3, uma melhora em média de **1.97%** do algoritmo da tese em relação ao algoritmo de Pettis e Hansen e uma melhora de **2.53%** na evolução dos programas, isto é, a relação entre a primeira e última execução de um determinado programa.

Conseguimos melhores desempenhos nos programas bzip2 e gcc, e esse ganho foi de quase **12%**.

O programa parser teve uma piora quando analisado em relação ao seu histórico (-0.34%).

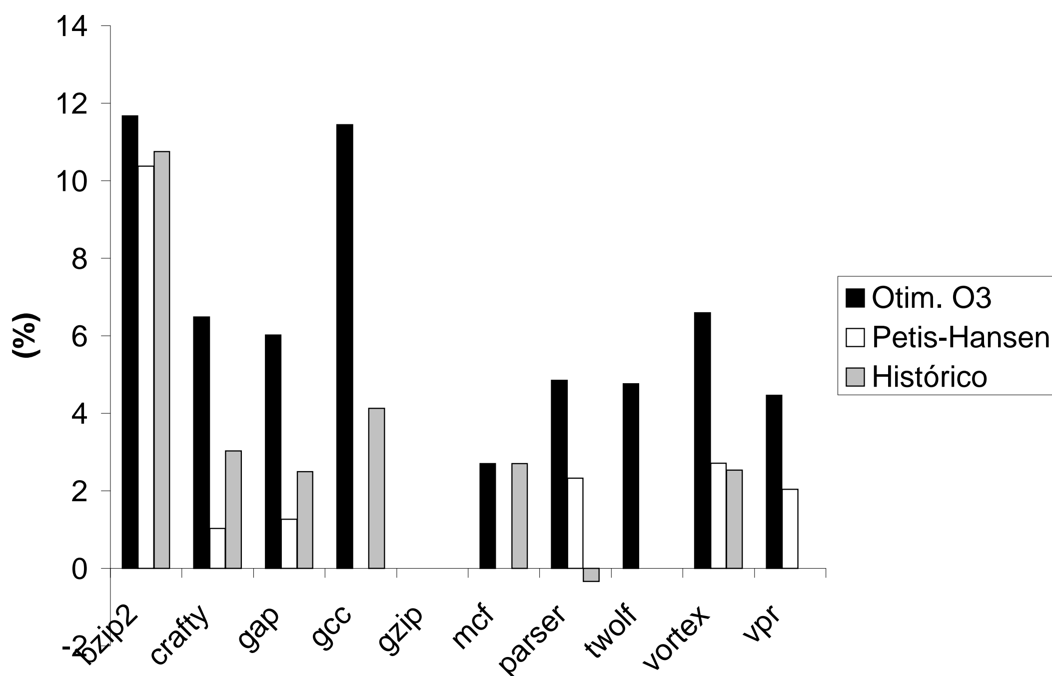


Figura 5.5: Programas do SPEC2000 conjunto de entrada test

5.3.2 SPEC2000

Foram realizados testes com 10 programas do SPEC2000 e o algoritmo proposto na tese teve um desempenho melhor do que o compilador gcc utilizando o nível de otimização O3, melhor que o algoritmo de Pettis e Hansen e teve uma melhora na evolução dos programas. Na Tabela 5.14 podemos ver um resumo de cada programa, os ganhos (ou perdas) em relação as outras 3 otimizações (O3, PH e Histórico). Na

Figura 5.6 podemos ver o percentual de melhora do tempo da última execução do programa em relação as 3 otimizações (O3, PH e evolução histórico).

Tivemos uma melhora em média de **6.50%** do algoritmo da tese em relação ao compilador gcc com otimização O3 e uma melhora em média de **3.66%** do algoritmo da tese em relação ao algoritmo de Pettis e Hansen. Podemos observar uma melhora de **4.22%** na evolução dos programas, isto é, a relação entre a primeira e a última execução de um determinado programa.

Os melhores resultados foram obtidos nos programas gzip, gcc, vpr e gap.

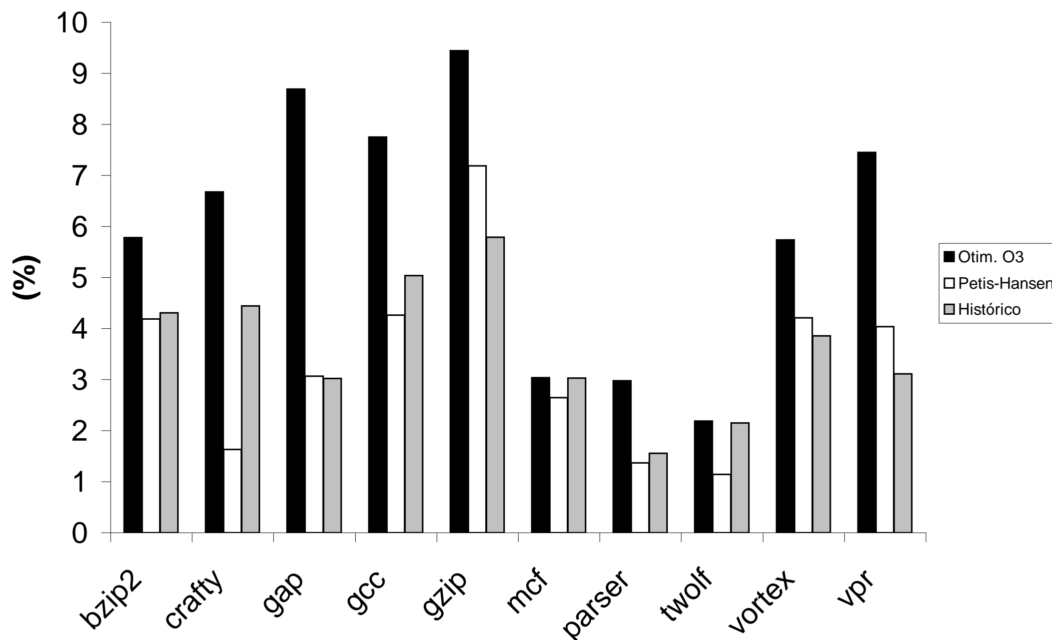


Figura 5.6: Performance dos programas do SPEC2000

5.3.3 SPEC95

Foram realizados testes com dois programas do SPEC95 (compress e go) e o algoritmo proposto na tese teve um desempenho **8.4%** melhor do que o compilador gcc utilizando o nível de otimização O3, **4.35%** em relação ao algoritmo de Petis e Hansen e teve uma melhora no histórico na faixa de **5.30%**.

5.3.4 Algoritmos genéticos, equação linear e algoritmo recursivo

Foram testados outros tipos de programas, um programa utilizando algoritmos genéticos, um outro programa para o cálculo de componentes fortemente conexos

de um grafo e um programa para solucionar equações lineares. Como podemos ver na Tabela 5.15 e no Gráfico 5.7, ocorreu uma grande melhora no tempo de execução em relação ao otimizador do compilador (O3), uma melhora em média **24.4%**. Os outros percentuais estão na Tabela 5.15.

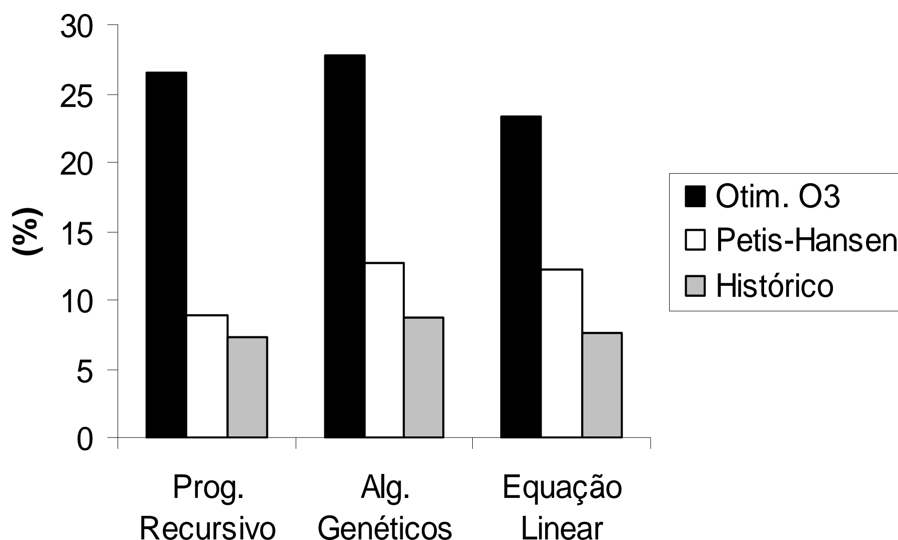


Figura 5.7: Performance dos programas extras

Foram realizados diversos testes para analisar como funciona o histórico. Um dos testes foi deixar executando n vezes (150) um determinado programa (bzip2) somente com a otimização O3 e com a otimização Petis e Hansen, com entradas variando aleatoriamente e verificamos que os resultados não são alterados.

Outro tipo de análise que fizemos foi executar um determinado programa (bzip2 com diversas entradas) centenas de vezes (400) e verificar até quando o programa otimizado da tese melhora. Verificamos que chega um certo momento (dependendo do tempo de execução do programa) que essa melhora é ínfima. Conseguimos uma melhora de quase 15% em algumas entradas; no final do processo a entrada *reference* estava tendo ainda pequenas melhoras.

Entrada	gzip						
0	num	11	17	33	48	57	61
	tempo	39.87	38.30	37.34	36.15	36.52	36.38
1	num	14	20	35	41	50	75
	tempo	19.83	19.87	19.81	19.68	18.12	18.65
2	num	26	36	42	81	87	104
	tempo	49.76	48.17	47.54	48.38	46.06	46.11
3	num	67	79	129	130	131	132
	tempo	40.87	40.29	41.08	39.89	38.38	38.11
4	num	38	43	60	102	105	109
	tempo	70.76	70.96	70.97	69.95	69.12	68.08
5	num	22	27	32	58	90	103
	tempo	25.30	25.77	25.46	24.16	25.31	24.14
6	num	8	21	55	118	121	122
	tempo	0.00	0.00	0.00	0.00	0.00	0.00
7	num	3	19	28	46	76	127
	tempo	0.74	0.74	0.75	0.75	0.74	0.74
8	num	12	34	39	68	91	94
	tempo	0.27	0.27	0.27	0.28	0.27	0.27
9	num	44	47	70	86	93	100
	tempo	0.83	0.82	0.82	0.83	0.83	0.82
10	num	7	10	25	29	59	107
	tempo	0.68	0.68	0.69	0.68	0.68	0.68
11	num	15	16	73	84	95	101
	tempo	1.23	1.23	1.23	1.23	1.23	1.23
12	num	1	13	23	45	49	54
	tempo	0.73	0.70	0.70	0.70	0.70	0.70
13	num	53	72	82	110	123	125
	tempo	0.28	0.28	0.28	0.28	0.28	0.28
14	num	2	56	66	69	74	92
	tempo	0.54	0.54	0.54	0.54	0.54	0.54
15	num	18	24	31	62	83	108
	tempo	0.69	0.69	0.68	0.68	0.68	0.68
16	num	4	52	64	96	114	120
	tempo	1.01	1.01	1.01	1.01	1.01	1.01
17	num	71	77	80	88	99	128
	tempo	0.69	0.69	0.69	0.69	0.69	0.69
18	num	9	30	37	65	78	98
	tempo	0.29	0.29	0.29	0.29	0.29	0.29
19	num	5	40	85	89	111	117
	tempo	2.24	2.24	2.23	2.23	2.23	2.23
20	num	51	63	115	116	119	124
	tempo	0.68	0.68	0.69	0.68	0.68	0.68
21	num	6	97	106	112	113	126
	tempo	1.42	1.41	1.41	1.41	1.41	1.41

Tabela 5.10: Evolução (seg.) do gzip

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
bzip2	370.50	385.81	3.96	392.18	5.52	385.61	3.91
crafty	125.03	129.85	3.71	133.25	6.16	126.21	0.94
gap	194.83	200.70	2.92	203.78	4.39	200.98	3.06
gcc	237.07	249.59	5.01	256.92	7.72	247.83	4.34
gzip	207.33	221.09	6.22	229.41	9.62	223.65	7.29
mcf	767.03	789.98	2.91	789.98	2.91	786.53	2.48
parser	403.08	408.93	1.45	414.80	2.84	408.07	1.24
twolf	887.48	906.99	2.15	907.41	2.19	897.73	1.14
vortex	352.16	366.30	3.86	373.61	5.74	367.67	4.21
vpr	464.57	479.52	3.11	501.98	7.45	484.13	4.04

Tabela 5.11: Tempo (seg.) de execução do conjunto de entrada reference

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
bzip2	10.97	12.28	10.67	12.07	9.11	11.92	8.43
crafty	18.06	19.98	9.61	20.12	10.24	19.30	6.42
gap	6.17	6.57	6.09	6.49	4.93	6.41	3.75
gcc	86.43	89.98	3.95	92.82	6.89	89.78	3.73
gzip	18.65	19.83	5.95	20.72	9.99	19.89	6.23
mcf	66.05	69.23	4.59	69.34	4.74	69.34	4.74
parser	9.06	9.62	5.82	9.80	7.55	9.63	5.92
twolf	20.53	21.36	3.89	21.76	5.65	21.54	4.69
vortex	91.81	95.67	4.03	97.50	5.84	95.97	4.33
vpr	29.34	32.03	8.39	33.50	12.41	32.26	9.05

Tabela 5.12: Tempo (seg.) de execução do conjunto de entrada train

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
bzip2	8.63	9.67	10.75	9.77	11.67	9.63	10.38
crafty	2.88	2.97	3.03	3.08	6.49	2.91	1.03
gap	0.78	0.80	2.50	0.83	6.02	0.79	1.27
gcc	1.16	1.21	4.13	1.31	11.45	1.16	0.00
gzip	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mcf	0.36	0.37	2.70	0.37	2.70	0.36	0.00
parser	2.94	2.93	−0.34	3.09	4.85	3.01	2.33
twolf	0.20	0.20	0.00	0.21	4.76	0.20	0.00
vortex	5.39	5.53	2.53	5.77	6.59	5.54	2.71
vpr	1.92	1.92	0.00	2.01	4.47	1.96	2.04

Tabela 5.13: Tempo (seg.) de execução do conjunto de entrada test

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
bzip2	396.44	414.32	4.31	420.78	5.78	413.80	4.19
crafty	146.81	153.64	4.44	157.33	6.68	149.25	1.63
gap	202.58	208.90	3.02	221.87	8.69	209.01	3.07
gcc	241.62	254.47	5.04	261.93	7.75	252.38	4.26
gzip	243.72	258.71	5.79	269.14	9.44	262.62	7.19
mcf	835.42	861.56	3.03	861.68	3.04	858.22	2.65
parser	418.92	425.57	1.56	431.82	2.98	424.78	1.37
twolf	865.97	884.64	2.11	884.64	2.11	875.21	1.06
vortex	336.61	350.13	3.86	356.79	5.65	351.41	4.21
vpr	433.12	445.38	2.75	466.28	7.11	449.73	3.69

Tabela 5.14: Tempo de execução (seg.) SPEC2000

Prog.	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
Alg.Recursivo	754	814	7.37	1025	26.43	827	8.82
Alg.Gen.	6876	7540	8.80	9528	27.80	7880	12.74
Equacao Linear	36	39	7.69	47	23.40	41	12.19

Tabela 5.15: Tempo de execução (seg.) dos programas extras

5.4 Análise de baixo nível

Ao longo da última década, a velocidade dos processadores cresceu muito enquanto a velocidade da memória cresceu pouco. Conseqüentemente, as penalidades de *cache* miss têm aumentado de alguns ciclos para mais de 100 ciclos [21]. Os *caches misses* reduzem bastante a habilidade de um processador moderno de explorar efetivamente o paralelismo a nível de instrução. De um modo geral, há duas classes de técnicas de otimização que visam a melhoria do desempenho do *cache*: aquelas que reduzem o número de *caches misses* e outras para reduzir as penalidades impostas pelas falhas na *cache*. O foco deste trabalho é sobre as otimizações do primeiro tipo.

Nós usamos a ferramenta PAPI [48, 49] para fazer medidas de baixo nível dos programas do SPEC2000. PAPI foi implementado para facilitar o uso de contadores de hardware que existem na maioria das plataformas de processadores. Esses contadores podem oferecer valiosas informações sobre o desempenho de partes críticas de uma aplicação e apontar caminhos para melhorar o desempenho. Eles permitem que o desempenho do processador seja monitorado e calculado.

PAPI oferece duas interfaces para o contador de hardware, uma simples de alto nível para a aquisição de simples medidas e outra que foi utilizada neste trabalho, completamente programável direcionada a usuários com propósitos mais sofisticados, para gerenciar os eventos de hardware. A implementação do PAPI para o Linux IA-32 utiliza um *patche* (perfctr) para habilitar o acesso aos contadores de monitoração de desempenho. O processador AMD usado no trabalho possui 4 contadores.

Com os contadores podemos fazer uma análise da aplicação; em função de diversos eventos, escolhemos 4 que podem ser melhorados com a otimização proposta. A memória *cache* L1 e L2, desvios mal previstos e desvios tomados, eventos que podem ser afetados com a reordenação do código. Fizemos uma comparação envolvendo os executáveis gerados pelo compilador gcc a nível O3, os executáveis gerados por Petis e Hansen e os executáveis gerados pela otimização da tese.

5.4.1 Resultados PAPI – programas SPEC2000

Serão relatados todos os resultados de baixo nível nos dez programas do SPEC2000 em quatro tabelas. A primeira tabela vai conter o número de instruções que ocorreram falhas na memória *cache* L1, a segunda tabela vai conter o número de instruções que ocoreram falhas na memória *cache* L2, a terceira tabela vai conter o número de

instruções que ocorreram desvios tomados e a quarta tabela o número de instruções que ocorreram desvios mal previstos. Podemos verificar que não tivemos nenhuma melhora nos desvios tomados e desvios mal previstos em relação a Petis e Hansen e ao histórico mas tivemos grandes melhoras em relação ao otimizador O3 principalmente em relação aos desvios tomados. Não existe uma fórmula ou uma regra que podemos seguir para verificar o quanto cada um desses parâmetros está relacionado com a redução no tempo de execução de cada um dos programas.

Alguns percentuais negativos mostram que ocorreu uma piora em relação a última execução do algoritmo da tese.

Vamos relatar somente dois programas neste capítulo para não se tornar muito extenso, os outros resultados dos oito programas serão colocados no Apêndice A.

Programa gcc

Nas tabelas a seguir, o NI é o número de instruções que foram executadas em cada uma das 3 otimizações, e será calculado o ganho ou a perda do número de instruções executadas na última execução em relação as 3 otimizações (O3, PH e evolução histórico).

Como podemos ver na Tabela 5.16, as falhas na *cache* L1 tiveram uma redução em média **13.06%** melhor do que o otimizador O3, e somente **3.42%** melhor em relação ao histórico, a evolução das execuções e **5.17%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela 5.17, as falhas na *cache* L2 tiveram uma redução, em média **18.55%** melhor do que o otimizador O3, **10.25%** melhor em relação ao histórico, a evolução das execuções e **14.46%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela 5.18 e na Tabela 5.19, somente ocorreu uma melhora em relação ao otimizador O3, **22.97%** melhor no caso dos desvios tomados e **14.86%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Petis e Hansen.

Existem alguns casos que ocorreu uma piora em relação as outras otimizações mas foram raríssimos os casos.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	11524742	12956762	11.05	14932508	22.82	13001456	11.35
1	32719107	28712567	−13.95	30258905	−8.13	30098765	−8.70
2	5504955	6105429	9.83	6734560	18.25	6309529	12.75
3	3138943	3792138	17.22	4109876	23.62	3905429	19.62
4	24902539	28436097	12.42	33098631	24.76	27970319	10.96
5	4028414	4623901	12.87	4986212	19.20	4789301	15.88
6	3332561	3711216	10.20	4011314	16.92	3890116	14.33
7	1089352	999012	−9.04	1130395	3.63	1023989	−6.38
8	274692	243450	−12.83	259876	−5.70	245671	−11.81

Tabela 5.16: Redução nas falhas na *cache* L1 gcc

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	2083294	2402719	13.29	2890381	27.92	2609210	20.15
1	3871968	4208998	8.00	4509282	14.13	4308978	10.14
2	615895	786012	21.64	820621	24.94	810895	24.04
3	628837	661207	4.89	723207	13.04	690236	8.89
4	3287139	3603790	8.78	3920298	16.15	3807682	13.67
5	520349	590901	11.93	621591	16.28	610473	14.76
6	570235	630121	9.50	698104	18.31	680205	16.16
7	235376	276771	14.95	320835	26.63	290932	19.09
8	62666	72274	13.29	77834	19.48	75390	16.87

Tabela 5.17: Redução nas falhas na *cache* L2 gcc

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	1923261911	1923262987	0.00	2918920560	34.11	1923261959	0.00
1	6036377472	6036377472	0.00	7291740245	17.21	6036374209	0.00
2	603725101	603725131	0.00	982209820	38.53	603725202	0.00
3	606083275	606083375	0.00	943081743	35.73	606083752	0.00
4	3243510091	3243510504	0.00	3931083201	17.49	3243510401	0.00
5	277913228	277913331	0.00	389021385	28.56	277913335	0.00
6	120546672	120546832	0.00	180710467	33.29	120546643	0.00
7	32140749	32140432	0.00	39018036	17.62	32140421	0.00
8	5881344	5881476	0.00	6398260	8.07	5881353	0.00

Tabela 5.18: Redução nos desvios tomados gcc

Programa gzip

Como podemos ver na Tabela 5.20, as falhas na *cache* L1 tiveram uma excelente redução, em média **22.66%** melhor do que o otimizador O3, **11.81%** melhor em

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	133728681	133728777	0.00	158104578	15.41	133728693	0.00
1	779921088	779921329	0.00	910301856	14.32	779926723	0.00
2	87451531	87451531	0.00	98432719	11.15	87451571	0.00
3	58491952	58494572	0.00	65053149	10.08	58498264	0.00
4	525321567	525321627	0.00	632954129	17.00	525321671	0.00
5	56206505	56206632	0.00	64917435	13.41	56206790	0.00
6	29716712	29716850	0.00	33851131	12.21	29716888	0.00
7	8655416	8655719	0.00	9210728	6.02	8655421	0.00
8	1451287	1453218	0.00	1590252	8.73	1452452	0.00

Tabela 5.19: Redução nos desvios mal previstos gcc

relação ao histórico, a evolução das execuções e **2.16%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela 5.21, as falhas na *cache* L2 tiveram uma redução um pouco menor, em média **15.73%** melhor do que o otimizador O3, **7.47%** melhor em relação ao histórico, a evolução das execuções e **14.09%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela 5.22 e 5.23, somente ocorreu uma excelente melhora em relação ao otimizador O3, **37.52%** melhor no caso dos desvios tomados e somente **13.52%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Petis e Hansen.

Vamos mostrar nos Gráficos 5.8 e 5.9, a redução nas falhas na *cache* L1 e L2 respectivamente de todos os programas do SPEC2000. Os programas tiveram reduções no número de falhas na *cache* L1 e L2, tanto em relação ao gcc (O3), quanto a Pettis e Hansen e a evolução do histórico. Programas como bzip2, gap e mcf tiveram uma redução no número de falhas na memória *cache* L1 superior a **20%**.

A redução no número de desvios tomados em todos os programas analisados do SPEC2000 em relação ao otimizador O3, pode ser vista no Gráfico 5.10. Não existem melhoras na quantidade de desvios tomados entre a última execução do programa da tese (histórico) e a primeira execução, e também em relação ao algoritmo de Pettis e Hansen.

A redução no número de desvios tomados foi em média de **29.53%**. A maioria dos programas (sete) teve uma redução nos desvios tomados de mais de **25%** e

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	92906	104673	11.24	103606	10.32	100648	7.69
1	47958	51341	6.58	68548	30.03	59135	18.90
2	100836	118772	15.10	127541	20.93	127822	21.11
3	68456	90238	24.13	99742	31.36	99742	31.36
4	155237	174505	11.04	226887	31.57	183710	15.49
5	53441	67118	20.37	72688	26.47	67706	21.06
6	62120	57370	−8.27	61118	−1.63	57706	−7.64
7	3161	3291	3.95	3801	16.83	3479	9.14
8	1495	1712	12.67	1967	23.99	1787	16.34
9	3303	3566	7.37	3971	16.82	3619	8.73
10	2513	2824	11.01	3057	17.79	2726	7.81
11	4266	4519	5.59	4592	7.09	4530	5.82
12	2909	3046	4.49	3374	13.78	3057	4.84
13	2482	2434	−1.97	2130	−16.52	2202	−12.71
14	2616	2838	7.82	3250	19.50	3049	14.20
15	2379	2512	5.29	2750	13.49	2583	7.89
16	3100	3355	7.60	3556	12.82	3450	10.14
17	3014	3058	1.43	3416	11.76	3103	2.86
18	1979	1759	−12.50	2176	9.05	1865	−6.11
19	5889	5636	−4.48	5819	−1.20	5736	−2.66
20	2899	2497	−16.09	2838	−2.14	2535	−14.35
21	4709	4716	0.14	4838	2.66	4813	2.16

Tabela 5.20: Redução nas falhas na *cache* L1 gzip

alguns como o crafty e o gap em mais de **40%**. Somente o gcc, twolf e o vortex é que tiveram uma redução inferior a **25%** mas superior a **14%**.

A redução no número de desvios mal previstos foi em média de **8.02%** e seus resultados serão exibidos no Gráfico 5.11. Os ganhos são modestos e variam de **2%** o twolf até **15%** o gcc. Não existem melhoras na quantidade de desvios mal previstos entre a última execução do programa da tese e a primeira execução (evolução do histórico), e também em relação ao algoritmo de Pettis e Hansen.

Como podemos ver nos Gráficos 5.6, 5.8, 5.9, 5.10 e 5.11, os melhores ganhos em tempo de execução foram em relação aos programas gzip, gap e gcc e foram exatamente esses que tiveram os maiores ganhos em relação a redução nas falhas na *cache* e na redução dos desvios tomados e mal previstos. Os programas mcf, parser e twolf foram os que tiveram os piores ganhos em relação ao tempo de execução e foram justamente os que tiveram também as menores reduções nas perfomance de

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	52568	53860	2.39	57559	8.67	55927	6.00
1	27485	27684	0.71	38544	28.69	32598	15.68
2	50775	57231	11.28	59615	14.82	61589	17.55
3	47098	51047	7.73	52620	10.49	51939	9.32
4	75454	86336	12.60	91311	17.36	98374	23.29
5	33004	34972	5.62	37574	12.16	36532	9.65
6	34871	37512	7.04	42841	18.60	40001	12.82
7	2279	2444	6.75	2862	20.37	2481	8.14
8	1212	1311	7.55	1624	25.36	1396	13.18
9	2343	2472	5.21	2955	20.71	2566	8.69
10	2193	1992	−10.09	2247	2.40	1908	−14.93
11	2701	2736	1.27	3263	17.22	2930	7.81
12	2079	2171	4.23	2584	19.54	2208	5.84
13	1299	1368	5.04	1699	23.54	1623	19.96
14	1806	1985	9.01	2360	23.47	2009	10.10
15	1601	1720	6.91	2124	24.62	1631	1.83
16	2128	2183	2.51	2661	20.03	2220	4.14
17	2058	2195	6.24	2610	21.14	2276	9.57
18	1341	1421	5.62	1743	23.06	1452	7.64
19	3222	3281	1.79	3600	10.5	3432	6.11
20	1604	1633	1.77	2053	21.87	1662	3.48
21	2908	2933	0.85	3300	11.87	3029	3.99

Tabela 5.21: Redução nas falhas na *cache* L2 gzip

baixo nível analisadas.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	5187750255	5187750235	0.00	7878685895	34.15	5187750234	0.00
1	2632722270	2632727285	0.00	3764504283	30.06	2632728431	0.00
2	6109061638	6109061645	0.00	9820473215	37.79	6109061616	0.00
3	5005468889	5005468775	0.00	8881230063	43.63	5005468770	0.00
4	9657891352	9657895517	0.00	15824323309	38.96	9657895524	0.00
5	3577220059	3577220063	0.00	5562160115	35.68	3577220055	0.00
6	97637132	97637132	0.00	147903579	33.98	97637141	0.00
7	97634441	97634841	0.00	114432174	14.67	97634811	0.00
8	42141976	42141977	0.00	47903579	12.02	42141976	0.00
9	109318393	109318394	0.00	165768935	34.05	109318395	0.00
10	82675820	82675820	0.00	146984745	43.75	82675819	0.00
11	174253090	174253089	0.00	267977254	34.97	174253089	0.00
12	93925807	93925807	0.00	144007261	34.77	93925807	0.00
13	42462557	42462558	0.00	54954361	22.73	42462558	0.00
14	79082746	79082745	0.00	112703557	29.83	79082745	0.00
15	82674980	82674998	0.00	146982983	43.75	82674981	0.00
16	185681724	185681724	0.00	255824865	27.41	185681724	0.00
17	89788118	89788117	0.00	137200699	34.55	89788117	0.00
18	43351858	43351858	0.00	57333920	24.38	43351858	0.00
19	311123404	311123404	0.00	450678940	30.96	311123406	0.00
20	82663576	82663577	0.00	146977830	43.75	82663576	0.00
21	268861992	268861991	0.00	374802941	28.26	268861991	0.00

Tabela 5.22: Redução nos desvios tomados gzip

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	621323095	621323424	0.00	782963612	20.64	621323832	0.00
1	213814308	213811609	0.00	226207751	5.47	213812158	0.00
2	881872802	881873576	0.00	910701011	3.16	881874384	0.00
3	651990589	651995562	0.00	791687132	17.64	651995663	0.00
4	1094770435	1094771024	0.00	1278127542	14.34	1094771254	0.00
5	406850302	406850963	0.00	498633274	18.40	406850509	0.00
6	12008431	12009716	0.00	14371214	16.44	12009134	0.00
7	11437001	11437837	0.00	12874317	11.16	11437127	0.00
8	2364373	2364907	0.00	2754274	14.15	2364436	0.00
9	15343453	15343786	0.00	16976887	9.62	15343656	0.00
10	10606140	10606289	0.00	12505483	15.18	10606451	0.00
11	19288885	19288007	0.00	19458112	0.87	19288992	0.00
12	11544044	11548549	0.00	13148657	12.20	11549150	0.00
13	2464664	2464716	0.00	2814330	12.42	2464703	0.00
14	9182675	9182345	0.00	9437113	2.69	9182844	0.00
15	10622461	10622102	0.00	11487113	7.53	10622714	0.00
16	15103046	15103611	0.00	16228699	6.93	15103773	0.00
17	11553618	11553815	0.00	13189916	12.40	11553953	0.00
18	2720633	2720829	0.00	2892228	5.93	2720879	0.00
19	66025561	66026898	0.00	68438603	3.52	66029503	0.00
20	10613550	10611691	0.00	12984469	18.27	10613000	0.00
21	21402250	21402891	0.00	26578187	19.47	21402684	0.00

Tabela 5.23: Redução nos desvios mal previstos gzip

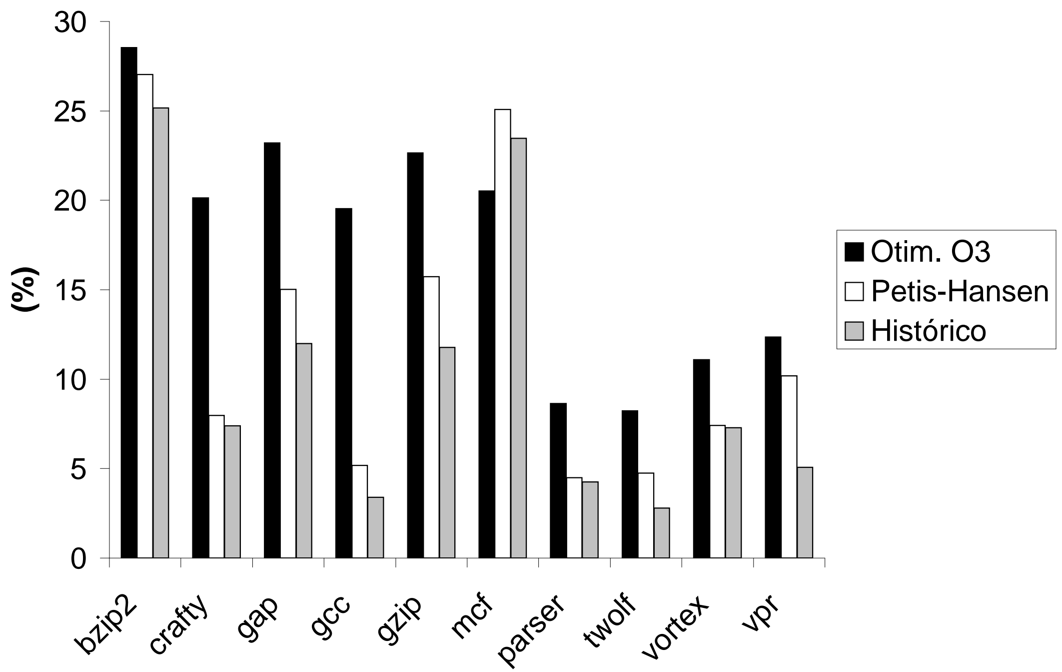


Figura 5.8: Redução nas falhas na *cache* L1

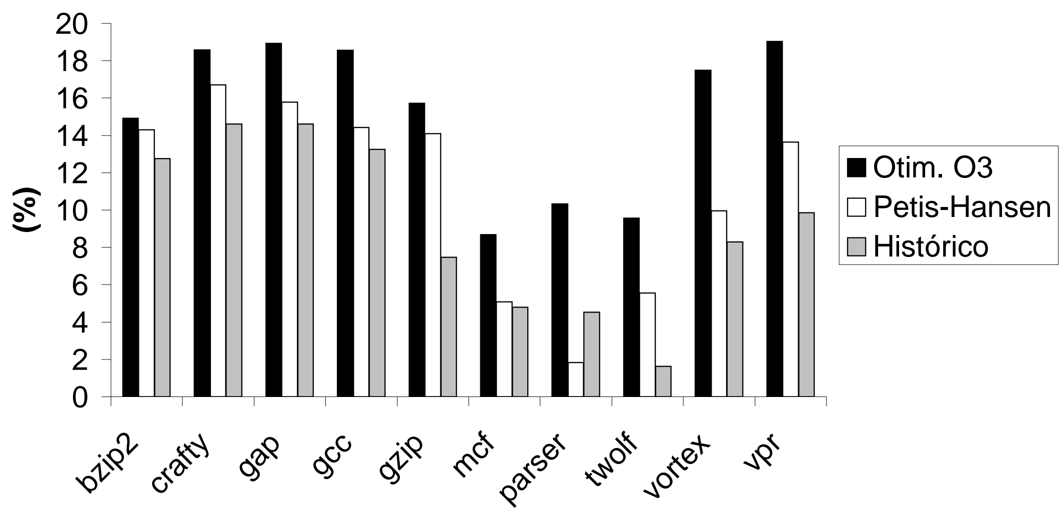


Figura 5.9: Redução nas falhas na *cache* L2

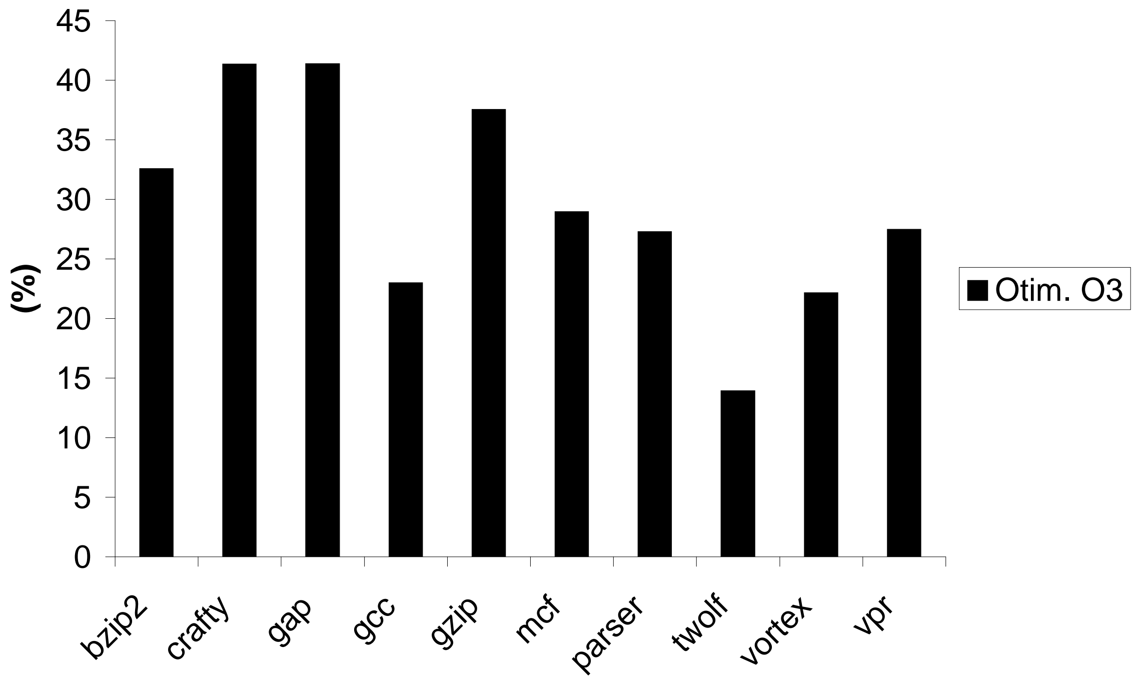


Figura 5.10: Redução nos desvios tomados

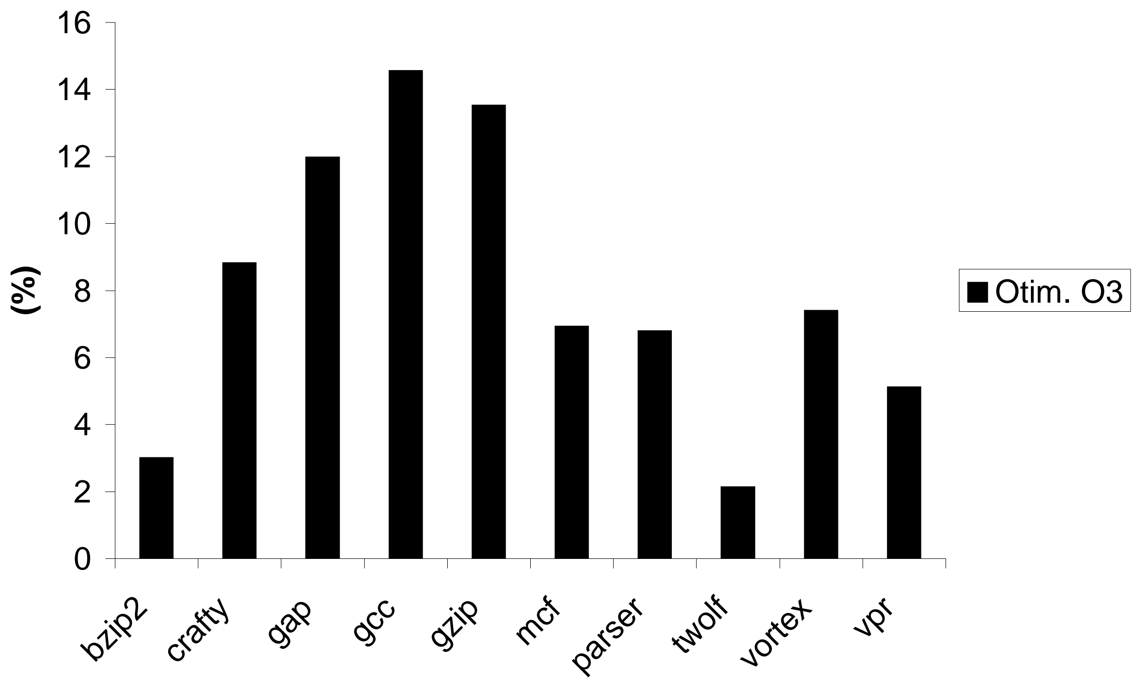


Figura 5.11: Redução nos desvios mal previstos

Capítulo 6

Conclusão

Neste trabalho introduzimos uma técnica probabilística e dinâmica para melhorar o desempenho da memória e conseqüentemente o tempo de execução de programas em máquinas da arquitetura x86.

Nossa técnica inclui dois modelos: o primeiro, denominado modelo de execução, processa o perfil de uma execução "programa + arquivo de entrada"e gera a correspondente rede *Bayesiana* cujos vértices são obtidos deste arquivo de perfil (gerado enquanto o programa processa o arquivo de entrada).

O segundo modelo, denominado modelo histórico, também é uma rede *Bayesiana*, relacionada ao programa, mas que processa os perfis gerados pela execução de uma variada seqüência de arquivos de entradas de dados.

O modelo de execução atual é incorporado ao modelo do histórico por uma técnica que atualiza os rótulos dos vértices por uma média geométrica ponderada entre dois rótulos correspondentes, uma do modelo histórico, e a outra do modelo da execução atual.

O efeito desta atualização contínua do modelo histórico, quando o programa é executado com novos *input files*, é que, para cada nova entrada, o código atual a executar pode levar em conta todo o conhecimento armazenado no modelo histórico. Para que isto seja alcançado, é necessário reordenar os blocos básicos. Essa reordenação é realizada de modo que os blocos básicos do programa mais prováveis de serem executados, ocupem o nível mais alto na hierarquia de memória.

Incorporar um modelo de execução atual dentro do modelo histórico pode ser realizado de várias maneiras, mesmo se nos restringirmos a usar o critério de média geométrica. A maneira escolhida neste trabalho foi atribuir pesos para as médias geométricas. O peso de execuções mais longas (tempo maior) influenciam mais o

modelo enquanto execuções menos longas, pouco influenciam o modelo.

Neste trabalho, nosso objetivo foi duplo: primeiro, verificar se a nossa técnica é capaz de fornecer melhores tempos de execução quando o programa for repetidamente executado com a mesma entrada, possivelmente com a intervenção de outras entradas; segundo, comparar os tempos de execução com aqueles obtidos com a otimização do compilador gcc nível 3 (i.e., sem reordenação dos blocos básicos) e com a reordenação Pettis-Hansen (conforme implementado no PLTO).

Os resultados obtidos com programas do conjunto SPEC2000 mostramm a eficiência do nosso modelo. Observamos ganhos nos tempos de execução, principalmente quando os arquivos de entradas requerem mais tempo de processamento. Os resultados também mostram, que na maioria dos casos investigados, os tempos de execução tendem a melhorar significativamente do que aqueles obtidos com a otimização O3 e PH.

Os trabalhos anteriores nessa área conseguiram em média uma melhora de 2% a 4% [63, 77]. Em alguns programas conseguimos ganhos superiores de até 14.50% em relação ao compilador O3. Nossa pesquisa conseguiu uma melhora em média de 6.5% em relação ao SPEC2000 e 8.4% em relação ao SPEC95 que são programas altamente otimizados. Com programas de usuários, conseguimos uma melhora de desempenho acima de 20% em relação ao compilador, com nível de otimização O3.

Nossa técnica apresenta uma melhora em relação à técnica de Pettis-Hansen, pois leva em consideração o caminho mais provável de ser seguido pelo fluxo de controle do programa, de forma que sua decisão inicial de alinhamento e agrupamento é global, em contrapartida às decisões gulosas e locais de Pettis-Hansen. Nosso modelo dinâmico chega a um determinado momento em que vai ter um arquivo de perfil histórico que funciona bem independente da entrada, já o Pettis e Hansen usa sempre o perfil da entrada corrente.

A ferramenta PAPI utilizada para fazer a análise de baixo nível nos programas, oferece contagem por processo, o que garante a precisão dos resultados. Porém as mudanças na *cache* não dependem apenas do processo que estamos rodando (*benchmark*), mas de todos os outros processos que entram no processador (inclusive os do Sistema Operacional).

É impossível fazer uma contagem dos *caches misses* apenas do processo do *benchmark*, porque durante sua execução, os processos do Sistema Operacional certamente serão alocados no processador. A única coisa que podemos garantir é que,

durante o experimento, o único processo modo usuário (não *kernel*) que havia era o nosso. Quando isso ocorre, os resultados dos testes possuem valores próximos, porque a única interferência externa vem do sistema operacional, que tem um comportamento mais ou menos constante. Por outro lado, o número de desvios tomados e mal previstos é exato porque não sofre nenhuma interferência do sistema operacional.

Alguns programas como bzip2, gap e mcf tiveram uma redução no número de *cache miss* superior a 20%. Em relação aos desvios tomados, observamos programas com uma redução de mais de 40%. Os ganhos em relação a redução dos desvios mal previstos são mais modestos e chegam em alguns programas a 15%.

Os ganhos no tempo de execução, quando comparado com O3 e Petis e Hansen, foram alcançados pelos programas gzip, gap e gcc e foram exatamente esses que apresentaram as maiores reduções no número de falhas na *cache* L1 e L2, desvios tomados e desvios mal previstos. Os programas mcf, parser e twolf tiveram moderadas reduções no tempo de execução e foram justamente os que tiveram também as menores reduções nas características de baixo nível analisadas.

Não conseguimos nenhuma melhora quanto aos desvios tomados e desvios mal previstos em relação a otimização Petis e Hansen e ao histórico, somente obtivemos melhoras em relação ao otimizador O3.

O que procuramos com este trabalho foi a maior generalidade possível, isso inclui por exemplo, que nossa modelagem não depende da configuração do hardawre, como tamanho da memória *cache*. Nossa abordagem também não dependente diretamente de nenhuma configuração arquitetural. Contudo, os resultados finais podem variar dependendo do ambiente de execução.

Apêndice A

Outros resultados

Vamos relatar os demais resultados dos programas do SPEC2000, em relação ao tempo de execução e as análises de baixo nível.

A.1 Evolução dos programas

Serão relatados todos os tempos de execução dos programas e a evolução do histórico.

A.1.1 Programa bzip2

Como podemos ver na Tabela A.1, o algoritmo da tese teve uma melhora em algumas entradas de quase **12%**. Em média o programa teve um desempenho **5.78%** melhor do que o do compilador gcc e **4.19%** melhor que o algoritmo de Pettis e Hansen.

Conforme mostra a Tabela A.2 e o Gráfico A.1, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de mais de 10% em algumas entradas. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **4.31%**.

Um fato diferente no programa bzip2 é que no conjunto de entrada train (entrada 3) o ganho foi maior no histórico do que em relação ao programa gcc (O3).

A.1.2 Programa crafty

Como podemos ver na Tabela A.3, o algoritmo da tese teve uma melhora em algumas entradas de quase **11%**. Em média o programa teve um desempenho **6.68%** melhor do que o do compilador gcc e **1.63%** melhor que o algoritmo de Pettis e Hansen.

Conforme mostra a Tabela A.4 e no Gráfico A.2, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de até **9%** em algumas entradas. Podemos

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	141.87	148.13	4.23	148.91	4.73	148.11	4.21
1	115.76	119.34	3.00	122.51	5.51	119.47	3.11
2	112.87	118.34	4.62	120.76	6.53	118.03	4.37
3	10.97	12.28	10.67	12.07	9.11	11.98	8.43
4	8.63	9.67	10.75	9.77	11.67	9.63	10.38
5	2.38	2.54	6.30	2.65	10.18	2.59	8.11
6	2.02	2.07	2.42	2.07	2.42	2.02	0.00
7	1.94	1.95	0.51	2.04	4.90	1.97	1.52
Soma	396.44	414.32	4.31	420.78	5.78	413.80	4.19

Tabela A.1: Tempo (seg.) de execução bzip2

Entrada	bzip2						
0	num	12	30	35	36	39	40
	tempo	148.13	147.56	145.45	144.12	143.54	141.87
1	num	14	19	24	37	43	45
	tempo	119.34	119.67	117.23	117.45	116.21	115.76
2	num	10	18	44	46	47	48
	tempo	118.34	117.23	118.65	116.43	114.12	112.87
3	num	15	23	26	32	34	38
	tempo	12.28	11.75	10.56	11.38	10.90	10.97
4	num	2	3	4	11	21	33
	tempo	9.67	9.12	8.82	8.76	9.30	8.63
5	num	6	13	22	25	27	29
	tempo	2.54	2.58	2.59	2.66	2.42	2.38
6	num	1	5	8	9	17	28
	tempo	2.07	2.02	2.02	2.01	2.02	2.02
7	num	7	16	20	31	41	42
	tempo	1.95	1.96	1.95	1.96	1.95	1.94

Tabela A.2: Evolução (seg.) do bzip2

verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **4.44%**.

Um fato diferente no programa crafty é que no conjunto de entrada reduced (entrada 3, 4 e 5) não tiveram quase nenhum ganho e teve uma piora em relação ao Pettis e Hansen.

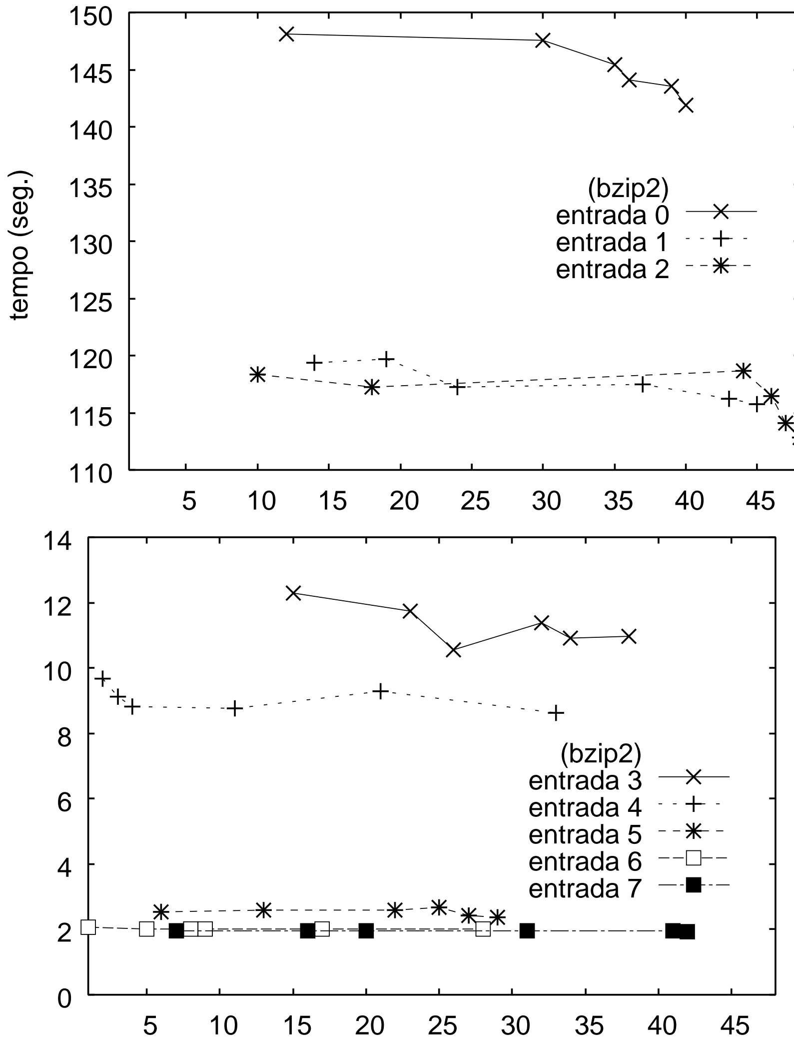


Figura A.1: Performance da evolução do programa bzip2 tabela A.2.

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	125.03	129.85	3.71	133.25	6.16	126.21	0.94
1	18.06	19.98	9.61	20.12	10.24	19.30	6.42
2	2.88	2.97	3.03	3.08	6.49	2.91	1.03
3	0.56	0.56	0.00	0.58	3.45	0.55	−1.82
4	0.21	0.21	0.00	0.23	8.70	0.21	0.00
5	0.07	0.07	0.00	0.07	0.00	0.07	0.00
Soma	146.81	153.64	4.44	157.33	6.68	149.25	1.63

Tabela A.3: Tempo (seg.) de execução crafty

A.1.3 Programa gap

Como podemos ver na Tabela A.5, o algoritmo da tese teve uma melhora em algumas entradas de quase **8%**. Em média o programa teve um desempenho **8.69%** melhor

Entrada	crafty						
0	num	6	7	9	10	19	22
	tempo	129.85	128.22	128.01	127.32	127.38	125.03
1	num	3	11	20	21	23	26
	tempo	19.98	19.43	19.99	19.81	19.43	18.06
2	num	14	24	25	30	32	34
	tempo	2.97	3.01	2.95	3.10	2.95	2.88
3	num	2	8	13	15	27	29
	tempo	0.56	0.55	0.56	0.56	0.56	0.56
4	num	4	5	16	31	35	36
	tempo	0.21	0.21	0.22	0.21	0.22	0.21
5	num	1	12	17	18	28	33
	tempo	0.07	0.07	0.07	0.07	0.07	0.07

Tabela A.4: Evolução (seg.) do crafty

do que o do compilador gcc e **3.07%** melhor que o algoritmo de Pettis e Hansen.

Conforme mostra a Tabela A.6 e no Gráfico A.3, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de quase **7%** em algumas entradas. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **3.02%**.

No conjunto de entrada train (entrada 1) a melhora em relação a Pettis e Hansen foi melhor do que a da otimização O3.

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	194.83	200.70	2.92	203.78	4.39	200.98	3.06
1	6.17	6.57	6.09	6.49	4.93	6.41	3.75
2	0.78	0.80	2.50	0.83	6.02	0.79	1.27
3	0.52	0.55	5.45	0.56	7.14	0.53	1.89
4	0.22	0.22	0.00	0.22	0.00	0.24	8.33
5	0.06	0.06	0.00	0.06	0.00	0.06	0.00
Soma	202.58	208.90	3.02	211.94	4.42	209.01	3.07

Tabela A.5: Tempo (seg.) de execução gap

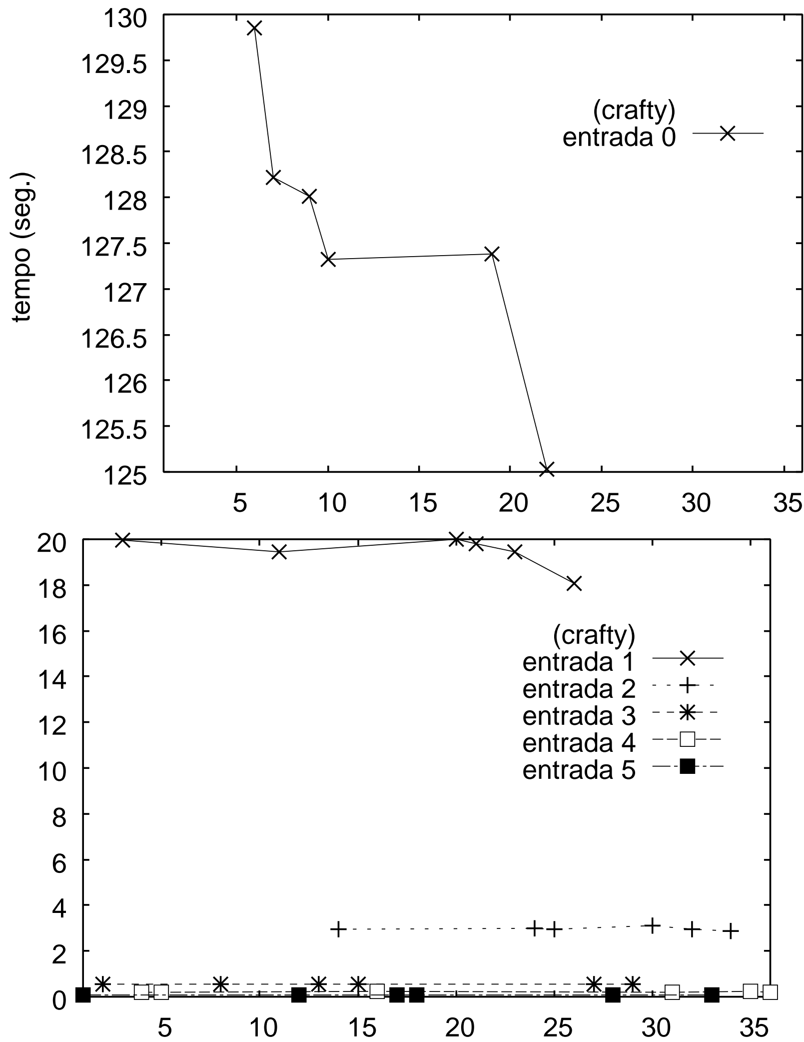


Figura A.2: Performance da evolução do programa crafty tabela A.4.

A.1.4 Programa mcf

Como podemos ver na Tabela A.7, o algoritmo da tese teve uma melhora em algumas entradas de quase **5%**. Em média o programa teve um desempenho **3.03%** melhor do que o do compilador gcc e **2.65%** melhor que o algoritmo de Pettis e Hansen.

Conforme mostra a Tabela A.8 e no Gráfico A.4, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de até **5%** em algumas entradas. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **3.03%**.

A entrada 0 teve uma melhora de mais de 20 segundos em relação ao histórico.

Entrada	gap						
0	num	6	7	9	10	19	22
	tempo	200.70	198.51	198.18	195.21	196.56	194.83
1	num	3	11	20	21	23	26
	tempo	6.57	6.53	6.32	6.29	6.09	6.17
2	num	14	24	25	30	32	34
	tempo	0.80	0.81	0.79	0.78	0.78	0.78
3	num	2	8	13	15	27	29
	tempo	0.55	0.55	0.54	0.52	0.53	0.52
4	num	4	5	16	31	35	36
	tempo	0.22	0.22	0.22	0.22	0.22	0.22
5	num	1	12	17	18	28	33
	tempo	0.06	0.07	0.08	0.07	0.06	0.06

Tabela A.6: Evolução (seg.) do gap

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	767.03	789.98	2.91	789.98	2.91	786.53	2.48
1	66.05	69.23	4.59	69.34	4.74	69.34	4.74
2	0.36	0.37	2.70	0.37	2.70	0.36	0.00
3	1.85	1.85	0.00	1.86	0.54	1.86	0.54
4	0.13	0.13	0.00	0.13	0.00	0.13	0.00
Soma	835.42	861.56	3.03	861.68	3.04	858.22	2.65

Tabela A.7: Tempo (seg.) de execução mcf

Entrada	mcf						
0	num	1	7	11	14	15	16
	tempo	789.98	777.33	775.06	771.36	768.29	767.03
1	num	2	3	5	6	8	20
	tempo	69.23	68.53	67.22	65.01	64.81	66.05
2	num	4	9	12	18	22	23
	tempo	0.37	0.37	0.37	0.36	0.36	0.36
3	num	10	17	19	24	26	28
	tempo	1.85	1.84	1.83	1.84	1.83	1.85
4	num	13	21	25	27	29	30
	tempo	0.13	0.13	0.13	0.13	0.13	0.13

Tabela A.8: Evolução (seg.) do mcf

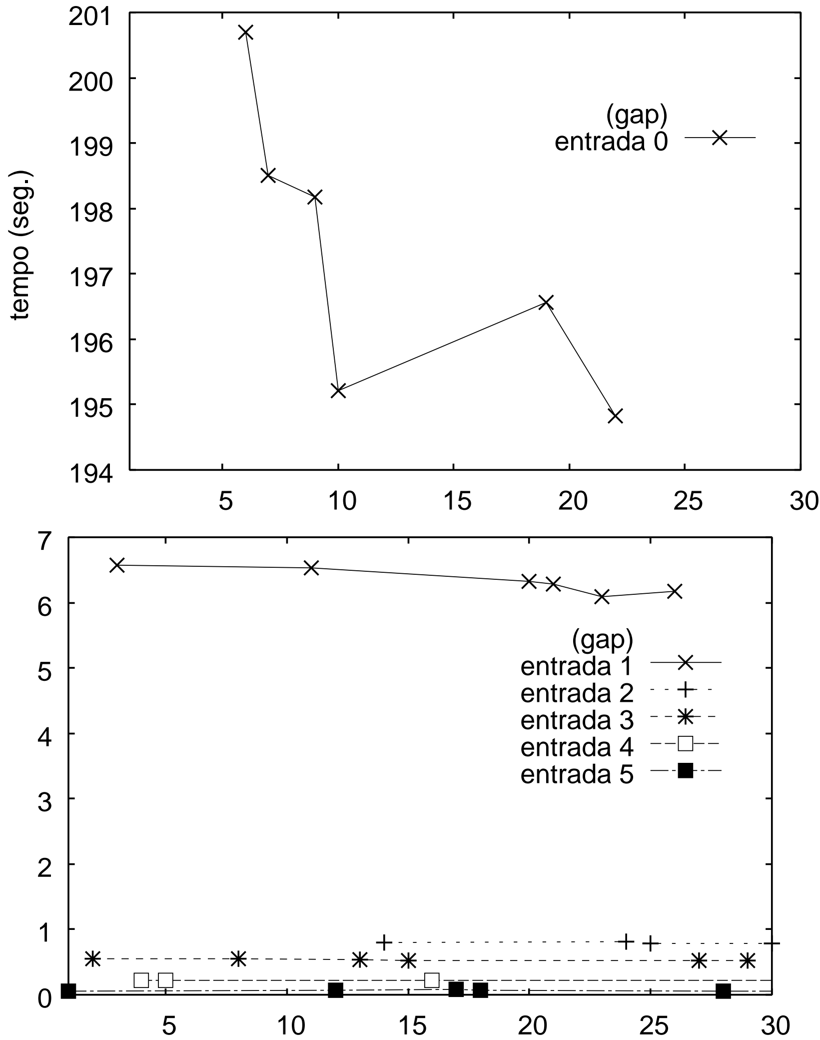


Figura A.3: Performance da evolução do programa gap tabela A.6.

A.1.5 Programa parser

Como podemos ver na Tabela A.9, o algoritmo da tese teve uma melhora em algumas entradas de quase **8%**. Em média o programa teve um desempenho **2.98%** melhor do que o do compilador gcc e **1.37%** melhor que o algoritmo de Pettis e Hansen.

Conforme mostra a Tabela A.10 e no Gráfico A.5, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de até **6%** em algumas entradas. Tivemos uma piora em relação ao histórico com a entrada 2. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **1.56%**.

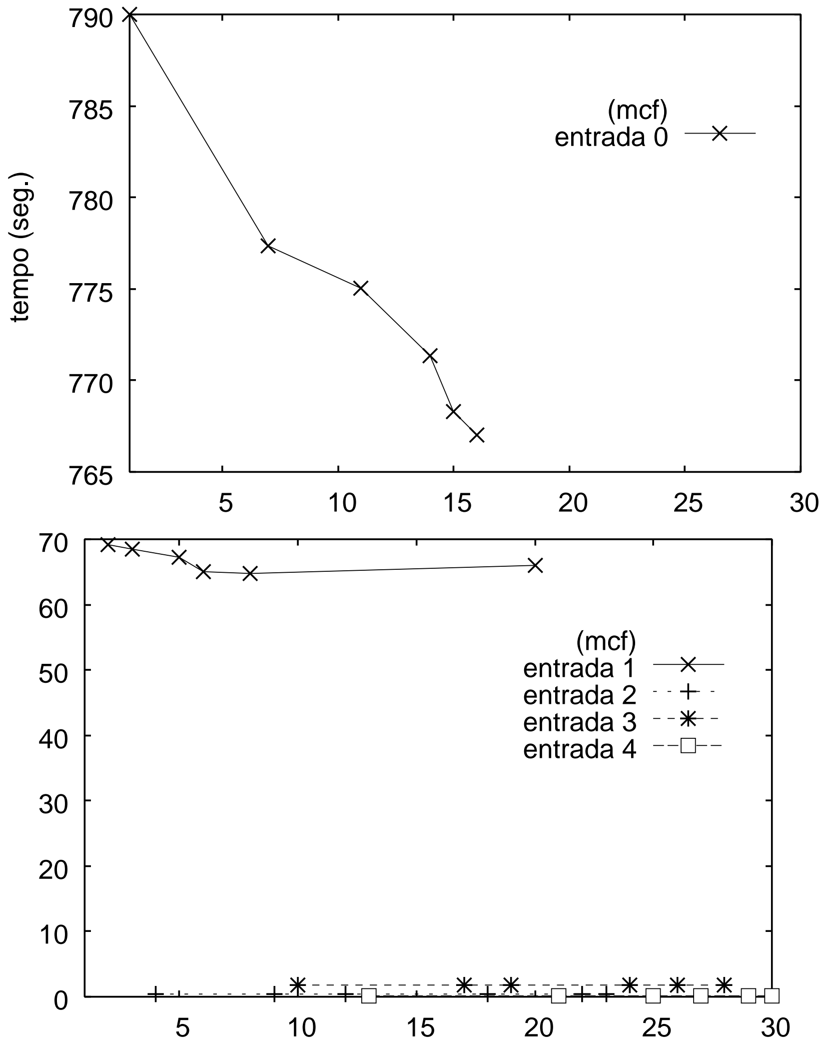


Figura A.4: Performance da evolução do programa mcf tabela A.8.

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	403.08	408.93	1.43	414.80	2.82	408.07	1.22
1	9.06	9.62	5.82	9.80	7.55	9.63	5.92
2	2.94	2.93	−0.34	3.09	4.85	3.01	2.33
3	3.38	3.41	0.88	3.45	2.03	3.41	0.88
4	0.46	0.48	4.17	0.48	4.17	0.46	0.00
5	0.20	0.20	0.00	0.20	0.00	0.20	0.00
Soma	419.12	425.57	1.52	431.82	2.94	424.78	1.33

Tabela A.9: Tempo (seg.) de execução parser

A.1.6 Programa twolf

Como podemos ver na Tabela A.11, o algoritmo da tese teve uma melhora em algumas entradas de quase **13%**. Em média o programa teve um desempenho **2.15%**

Entrada	parser						
0	num	6	7	9	10	19	22
	tempo	408.93	407.78	405.02	405.11	406.18	403.08
1	num	3	11	20	21	23	26
	tempo	9.62	9.69	9.54	9.49	9.98	9.06
2	num	14	24	25	30	32	34
	tempo	2.93	3.02	2.95	3.01	2.94	2.94
3	num	2	8	13	15	27	29
	tempo	3.41	3.45	3.40	3.38	3.43	3.38
4	num	4	5	16	31	35	36
	tempo	0.48	0.48	0.46	0.46	0.47	0.46
5	num	1	12	17	18	28	33
	tempo	0.20	0.19	0.20	0.21	0.21	0.20

Tabela A.10: Evolução (seg.) do parser

melhor do que o do compilador gcc e **1.14%** melhor que o algoritmo de Pettis e Hansen.

Conforme mostra a Tabela A.12 e no Gráfico A.6, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de até **4%** em algumas entradas. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **2.15%**.

A entrada 0 teve uma melhora de mais de 20 segundos em relação ao histórico.

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	865.97	884.64	2.11	884.64	2.11	875.21	1.06
1	20.53	21.36	3.89	21.76	5.65	21.54	4.69
2	0.20	0.20	0.00	0.21	4.76	0.20	0.00
3	0.71	0.71	0.00	0.72	1.39	0.70	−1.43
4	0.07	0.08	12.50	0.08	12.50	0.08	12.50
Soma	887.48	906.99	2.15	907.41	2.19	897.73	1.14

Tabela A.11: Tempo (seg.) de execução twolf

A.1.7 Programa vortex

Como podemos ver na Tabela A.13, o algoritmo da tese teve uma melhora em algumas entradas de quase **9%**. Em média o programa teve um desempenho **5.74%**

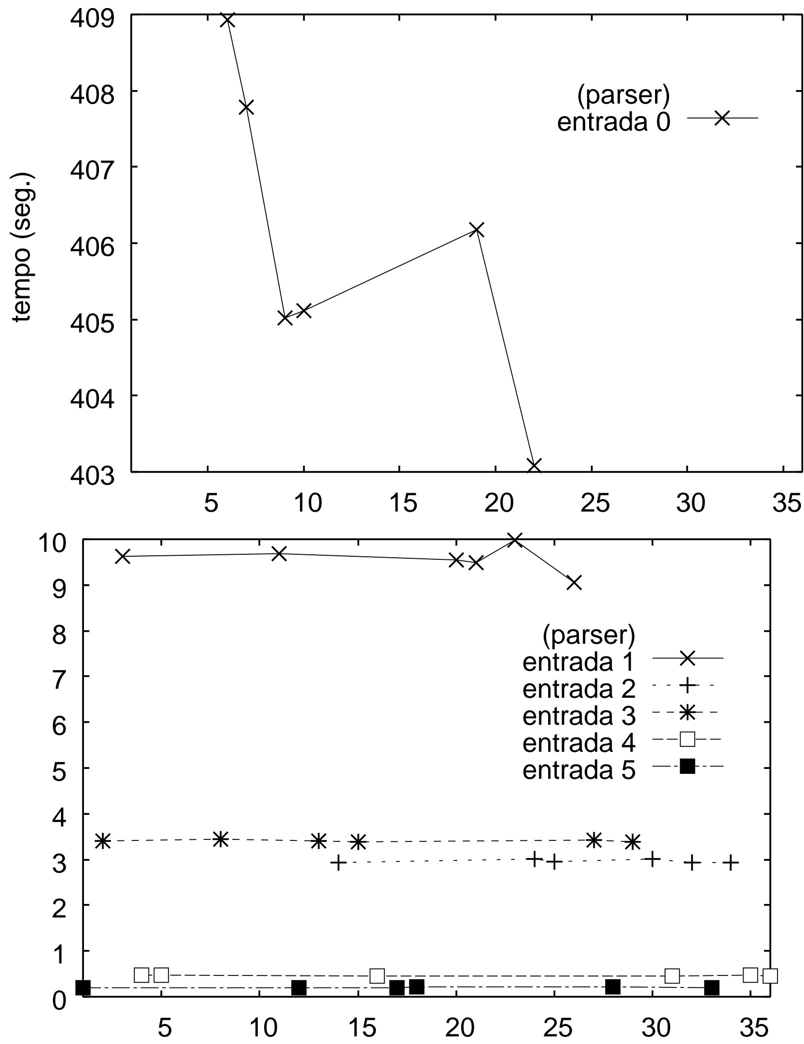


Figura A.5: Performance da evolução do programa parser tabela A.10.

Entrada	twolf						
0	num	1	7	11	14	15	16
	tempo	884.64	875.61	872.43	871.87	868.10	865.97
1	num	2	3	5	6	8	20
	tempo	21.36	21.28	21.01	20.03	20.08	20.53
2	num	4	9	12	18	22	23
	tempo	0.20	0.21	0.20	0.20	0.20	0.20
3	num	10	17	19	24	26	28
	tempo	0.71	0.71	0.71	0.72	0.71	0.71
4	num	13	21	25	27	29	30
	tempo	0.08	0.08	0.07	0.07	0.07	0.07

Tabela A.12: Evolução (seg.) do twolf

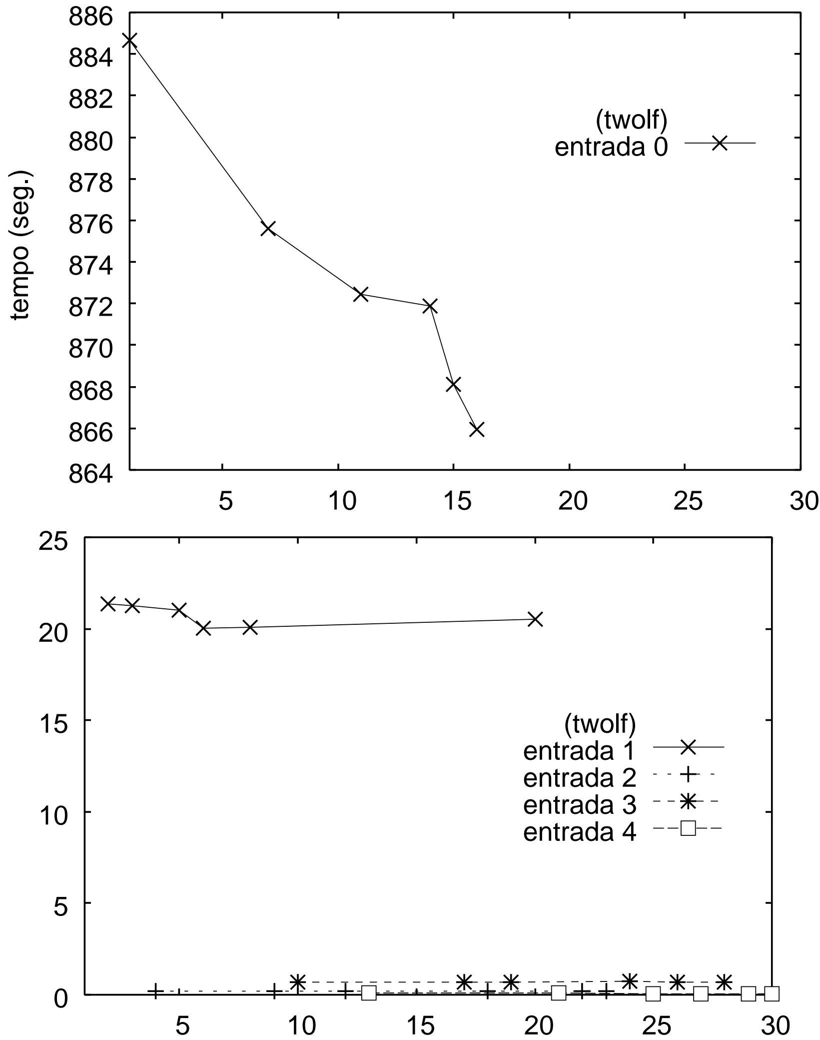


Figura A.6: Performance da evolução do programa twolf tabela A.12.

melhor do que o do compilador gcc e **4.21%** melhor que o algoritmo de Pettis e Hansen.

Conforme mostra a Tabela A.14 e no Gráfico A.7, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de até **4%** em algumas entradas. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **3.86%**.

A.1.8 Programa vpr

Como podemos ver na Tabela A.15, o algoritmo da tese teve uma melhora em algumas entradas de quase **13%**. Em média o programa teve um desempenho **7.45%** melhor do que o do compilador gcc e **4.04%** melhor que o algoritmo de Pettis e

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	108.78	114.21	4.75	116.00	6.22	114.32	4.85
1	91.81	95.67	4.03	97.50	5.84	95.97	4.33
2	136.02	140.25	3.02	143.29	5.07	141.12	3.61
3	9.29	9.76	4.82	10.13	8.29	9.87	5.88
4	5.39	5.53	2.53	5.77	6.59	5.54	2.71
5	0.61	0.61	0.00	0.65	6.15	0.59	−3.39
6	0.21	0.22	4.55	0.22	4.55	0.21	0.00
7	0.05	0.05	0.00	0.05	0.00	0.05	0.00
Soma	352.16	366.30	3.86	373.61	5.74	367.67	4.21

Tabela A.13: Tempo (seg.) de execução vortex

Entrada	vortex						
0	num	12	30	35	36	39	40
	tempo	114.21	113.73	111.12	109.33	109.12	108.78
1	num	14	19	24	37	43	45
	tempo	95.67	95.99	95.00	94.32	93.54	91.81
2	num	10	18	44	46	47	48
	tempo	140.25	140.12	140.36	138.02	137.16	136.02
3	num	15	23	26	32	34	38
	tempo	9.76	9.65	9.34	9.31	9.27	9.29
4	num	2	3	4	11	21	33
	tempo	5.53	5.53	5.47	5.42	5.83	5.39
5	num	6	13	22	25	27	29
	tempo	0.61	0.61	0.62	0.60	0.61	0.61
6	num	1	5	8	9	17	28
	tempo	0.22	0.21	0.20	0.20	0.21	0.21
7	num	7	16	20	31	41	42
	tempo	0.05	0.05	0.06	0.05	0.06	0.05

Tabela A.14: Evolução do vortex

Hansen.

Conforme mostra a Tabela A.16 e no Gráfico A.8, o algoritmo da tese teve uma melhora na evolução (usando o histórico) de até **3%** em algumas entradas. Podemos verificar no gráfico que as entradas com tempo de execução muito baixo apresentaram uma pequena melhora. Em média a melhora na evolução foi na ordem de **3.11%**.

O conjunto de entrada *train* (entrada 2 e 3) teve uma melhora de mais de **12%**

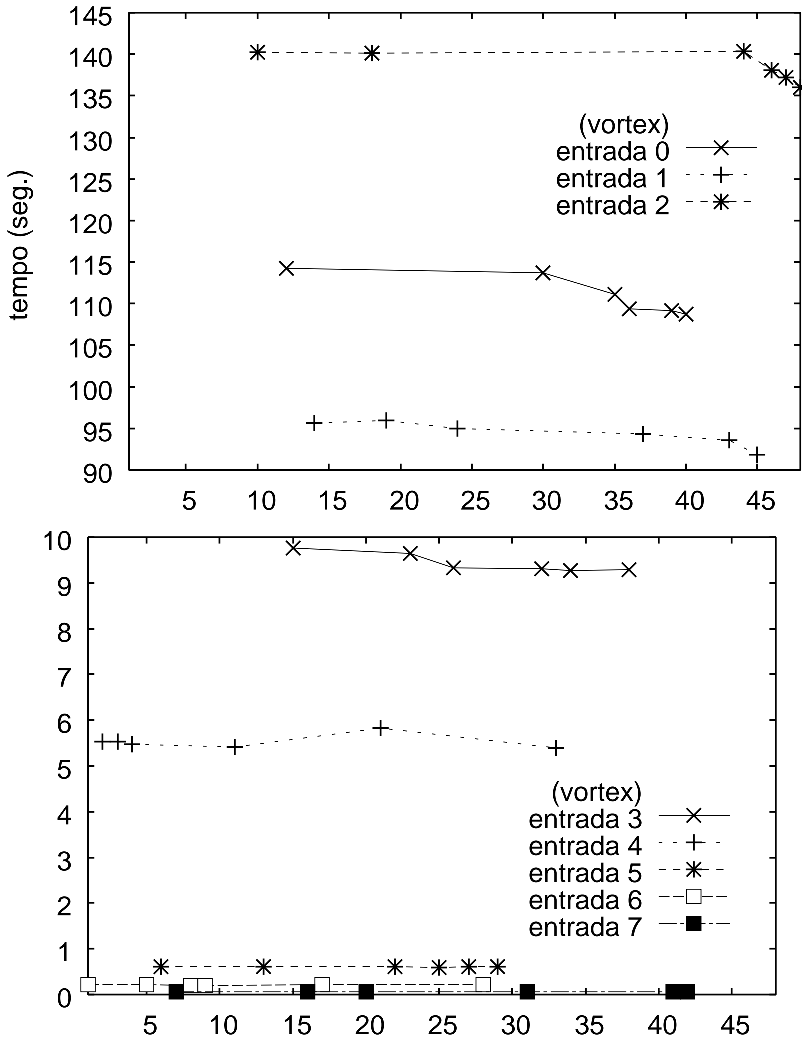


Figura A.7: Performance da evolução do programa vortex tabela A.14.

em relação ao gcc (O3).

A.2 Resultados PAPI – programas SPEC2000

Vamos relatar a análise de baixo nível dos programas do SPEC2000.

A.2.1 Programa bzip2

Como podemos ver na Tabela A.17, as falhas na *cache* L1 tiveram uma excelente redução em média **28.55%** melhor do que o otimizador O3, **25.14%** melhor em relação ao histórico, a evolução das execuções e **27.04%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.18, as falhas na *cache* L2 tiveram uma redução

Entrada	$t_{\text{último}}$	$t_{\text{prim.}}$	$g_{\text{prim.}}$	t_{O3}	g_{O3}	t_{PH}	g_{PH}
0	226.91	233.47	2.81	240.89	5.80	237.21	4.34
1	206.21	211.91	2.69	225.39	8.51	212.52	2.97
2	10.88	12.45	12.61	12.45	12.61	12.50	12.96
3	18.46	19.58	5.72	21.05	12.30	19.76	6.58
4	1.19	1.18	−0.85	1.24	4.03	1.22	2.46
5	0.73	0.74	1.35	0.77	5.19	0.74	1.35
6	0.17	0.17	0.00	0.18	5.56	0.17	0.00
7	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.02	0.02	0.00	0.01	−100.00	0.01	−100.00
9	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Soma	464.57	479.52	3.11	501.98	7.45	484.13	4.04

Tabela A.15: Tempo (seg.) de execução vpr

Entrada	vpr						
0	num	19	48	51	52	59	60
	tempo	233.47	233.85	231.99	229.54	227.98	226.91
1	num	18	39	43	47	55	58
	tempo	211.91	209.23	208.44	208.87	207.81	206.21
2	num	1	8	21	25	28	33
	tempo	12.45	11.80	11.12	11.06	10.31	10.88
3	num	6	13	14	23	24	34
	tempo	19.58	19.67	18.19	18.94	18.34	18.46
4	num	10	12	17	20	42	56
	tempo	1.18	1.18	1.19	1.20	1.20	1.19
5	num	26	30	45	53	54	57
	tempo	0.74	0.73	0.74	0.74	0.73	0.73
6	num	11	27	32	35	37	49
	tempo	0.17	0.17	0.17	0.17	0.17	0.17
7	num	2	4	15	36	40	46
	tempo	0.00	0.00	0.00	0.00	0.00	0.00
8	num	7	9	29	41	44	50
	tempo	0.02	0.01	0.02	0.02	0.01	0.02
9	num	3	5	16	22	31	38
	tempo	0.00	0.00	0.00	0.00	0.00	0.00

Tabela A.16: Evolução (seg.) do vpr

um pouco menor, em média **14.91%** melhor do que o otimizador O3, **12.75%** melhor em relação ao histórico, a evolução das execuções e **14.30%** melhor que o algoritmo de Pettis e Hansen.

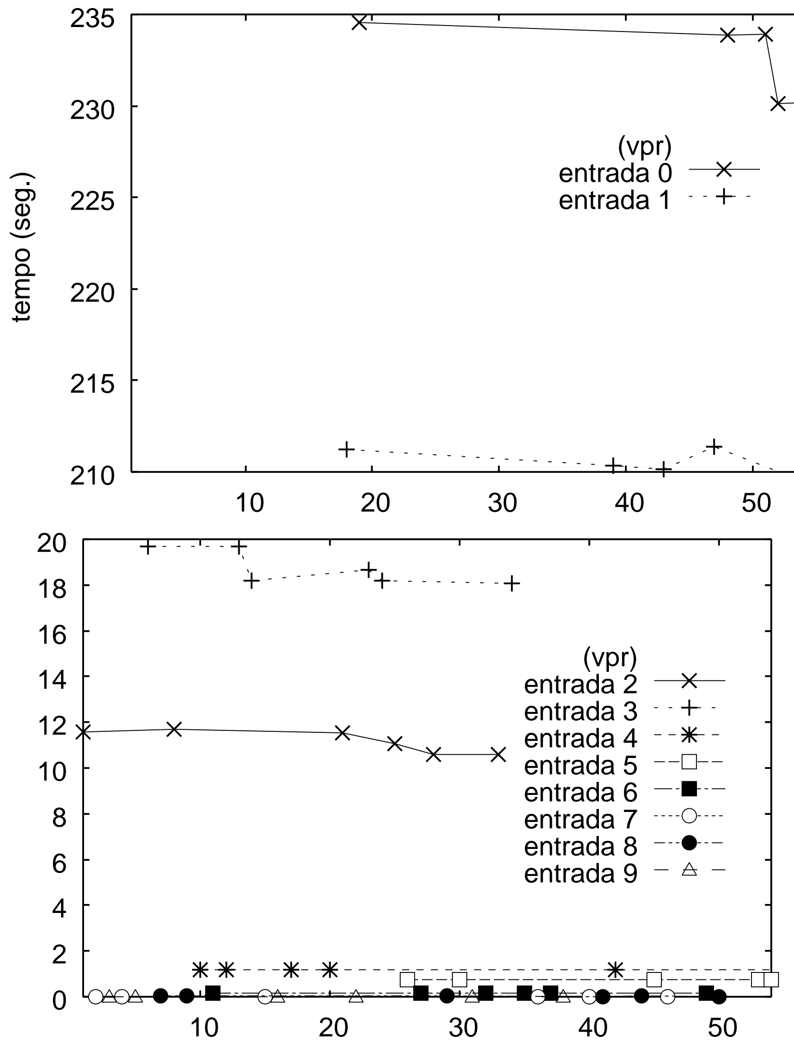


Figura A.8: Performance da evolução do programa vpr tabela A.16.

Como podemos ver na Tabela A.19 e na Tabela A.20, somente ocorreu uma ótima melhora em relação ao otimizador O3, **32.54%** melhor no caso dos desvios tomados e somente **2.52%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

A.2.2 Programa crafty

Como podemos ver na Tabela A.21, as falhas na *cache* L1 tiveram uma boa redução, em média **20.13%** melhor do que o otimizador O3, **7.39%** melhor em relação ao histórico, a evolução das execuções e **7.98%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.22, as falhas na *cache* L2 tiveram uma redução

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	103159	127827	19.29	130355	20.86	128295	19.59
1	91868	128363	28.43	131157	29.95	130018	29.34
2	96153	134860	28.70	145255	33.80	143127	32.81
3	7452	11218	33.57	11094	32.82	11274	33.90
4	5162	7019	26.40	7463	30.83	7156	27.86
5	3029	3117	2.82	4614	34.35	3212	5.69
6	3461	3202	-8.08	4469	22.55	3279	-5.55
7	3308	3458	4.33	4425	25.24	3476	4.83

Tabela A.17: Redução nas falhas na *cache* L1 bzip2

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	69250	75675	8.49	77076	10.15	76795	9.82
1	64427	73704	12.58	74929	14.01	73757	12.64
2	70006	84703	17.35	87617	20.09	87634	20.11
3	5347	6480	17.48	6679	19.94	6542	18.26
4	2963	3895	23.92	4103	27.78	4090	27.55
5	2108	2141	1.54	2366	10.90	2199	4.13
6	2114	2189	3.42	2310	8.48	2203	4.03
7	2896	2360	-22.71	2454	-18.01	2472	-17.15

Tabela A.18: Redução nas falhas na *cache* L2 bzip2

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	9220241068	9220241026	0.00	13765635212	33.02	9220241034	0.00
1	8555159524	8555159542	0.00	12124159661	29.43	8555159552	0.00
2	6178142465	6178142508	0.00	9666821046	36.08	6178142598	0.00
3	759086288	759086293	0.00	1123028281	32.40	759086297	0.00
4	846205064	846205065	0.00	1212387411	30.20	846205065	0.00
5	179274081	179274084	0.00	258341771	30.60	179274079	0.00
6	142319549	142319545	0.00	205634664	30.79	142319549	0.00
7	101540102	101540101	0.00	161504882	37.12	101540110	0.00

Tabela A.19: Redução nos desvios tomados bzip2

parecida com as falhas na cache L1, em média **18.58%** melhor do que o otimizador O3, **14.61%** melhor em relação ao histórico, a evolução das execuções e **16.70%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.23 e na Tabela A.24, somente ocorreu uma

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	879595335	879595345	0.00	899858496	2.25	879595980	0.00
1	858890211	858887233	0.00	890529200	3.55	858887276	0.00
2	846109458	846109994	0.00	874922556	3.29	846109498	0.00
3	28707971	28707998	0.00	29773814	3.57	28708996	0.00
4	20987532	20987991	0.00	21117473	0.61	20987904	0.00
5	15353063	15353973	0.00	15595527	1.54	15353124	0.00
6	15191343	151915672	0.00	15604061	2.65	15191537	0.00
7	14759134	14759146	0.00	15193211	2.85	14759179	0.00

Tabela A.20: Redução nos desvios mal previstos bzip2

excelente melhora em relação ao otimizador O3, **41.31%** melhor no caso dos desvios tomados e somente **8.82%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	389879770	419888303	7.14	491612312	20.69	420001636	7.17
1	59983725	65002666	7.72	70865382	15.35	68079065	11.89
2	4104336	5093678	19.42	5895002	30.37	5091668	19.39
3	1029671	1330614	22.61	1353989	23.95	1329213	22.53
4	475362	532448	10.72	580753	18.14	490523	3.09
5	135383	165684	18.28	178780	24.27	175962	23.06

Tabela A.21: Redução nas falhas na *cache* L1 crafty

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	689151	814882	15.42	840461	18.00	827105	16.67
1	123838	140489	11.85	154090	19.63	149973	17.42
2	30333	33027	8.15	40462	25.03	36192	16.18
3	7173	9391	23.61	10768	33.38	9442	24.03
4	6551	5699	−14.94	6121	−7.02	5740	−14.12
5	2956	3695	20.00	4397	32.77	4054	27.08

Tabela A.22: Redução nas falhas na *cache* L2 crafty

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	10816141532	10816141523	0.00	18349306888	41.05	10816141533	0.00
1	1605546134	1605546129	0.00	2830341876	43.27	1605546135	0.00
2	261042895	261042902	0.00	432930083	39.70	261042902	0.00
3	45023487	45023487	0.00	77593247	41.97	45023484	0.00
4	17905870	17905944	0.00	29350641	38.99	17905943	0.00
5	6156001	6156898	0.00	8384162	26.57	6156897	0.00

Tabela A.23: Redução nos desvios tomados crafty

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	2357516342	2357656728	0.00	2552543953	7.63	2357439711	0.00
1	359690863	359698603	0.00	421763923	14.71	359698859	0.00
2	61681410	61811503	0.00	72719162	14.99	61816821	0.00
3	10870800	10878419	0.00	12114320	10.26	10872883	0.00
4	4183926	4185752	0.00	5235136	20.07	4184064	0.00
5	1374539	1378976	0.00	1663627	17.37	1375365	0.00

Tabela A.24: Redução nos desvios mal previstos crafty

A.2.3 Programa gap

Como podemos ver na Tabela A.25, as falhas na *cache* L1 tiveram uma excelente redução, em média **23.21%** melhor do que o otimizador O3, **11.99%** melhor em relação ao histórico, a evolução das execuções e **15.03%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.26, as falhas na *cache* L2 tiveram uma redução um pouco menor, em média **18.92%** melhor do que o otimizador O3, **14.60%** melhor em relação ao histórico, a evolução das execuções e **15.79%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.27 e na Tabela A.28, somente ocorreu uma excelente melhora em relação ao otimizador O3, **41.34%** melhor no caso dos desvios tomados e somente **11.97%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	3369066	3810424	11.58	4330794	22.20	3951359	14.73
1	1425392	1654370	13.84	1953214	27.02	1732831	17.74
2	186825	187257	0.23	197523	5.41	180017	−3.787
3	197405	249627	20.92	286062	30.99	250520	21.20
4	69920	63198	−10.63	68466	−2.12	63589	−9.95
5	6075	6122	0.76	6955	12.65	6377	4.73

Tabela A.25: Redução nas falha na *cache* L1 gap

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	310698	344821	18.59	350597	19.93	354262	20.76
1	58503	59278	1.30	69049	15.27	57942	−0.96
2	18578	18691	0.60	22503	17.44	18370	−1.13
3	14407	14566	1.09	17667	18.45	13883	−3.77
4	6459	6341	−1.86	7382	12.50	5615	−15.03
5	1702	1684	−1.06	1931	11.85	1601	−6.30

Tabela A.26: Redução nas falhas na *cache* L2 gap

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	15130561432	15130561654	0.00	25890022556	41.55	15130561817	0.00
1	646889011	646889010	0.00	1024108714	36.83	646889010	0.00
2	74600826	74600926	0.00	119105815	37.36	74600854	0.00
3	48068136	48068234	0.00	78142120	38.48	48068136	0.00
4	22771296	22771298	0.00	38405758	40.70	22771845	0.00
5	3865908	3865956	0.00	5604638	31.02	3865932	0.00

Tabela A.27: Redução nos desvios tomados gap

A.2.4 Programa mcf

Como podemos ver na Tabela A.29, as falhas na *cache* L1 tiveram uma ótima redução, em média **20.53%** melhor do que o otimizador O3,**23.46%** melhor em relação ao histórico, a evolução das execuções e **25.07%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.30, as falhas na *cache* L2 tiveram uma redução um pouco menor, em média **8.69%** melhor do que o otimizador O3, **4.79%** melhor em relação ao histórico, a evolução das execuções e **5.09%** melhor que o algoritmo

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	1851313714	1851313724	0.00	2108551537	12.19	1851313814	0.00
1	73373904	73373978	0.00	77996627	5.92	73373908	0.00
2	10506693	10506797	0.00	12021764	12.60	10506698	0.00
3	7776036	7778038	0.00	8526405	8.80	7776095	0.00
4	3683544	3683751	0.00	4398924	16.25	3683730	0.00
5	371922	371972	0.00	435686	14.63	371982	0.00

Tabela A.28: Redução nos desvios mal previstos gap

de Pettis e Hansen.

Como podemos ver na Tabela A.31 e na Tabela A.32, somente ocorreu uma boa melhora em relação ao otimizador O3, **28.93%** melhor no caso dos desvios tomados e somente **8.93%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	345396	449803	23.21	426330	18.98	460339	24.96
1	38988	54260	28.14	57334	31.99	54444	28.38
2	1268	1398	9.29	1644	22.87	1338	5.23
3	14660	18129	19.13	18804	22.03	18702	21.61
4	1392	1304	−6.74	1383	−0.65	1350	−3.11

Tabela A.29: Redução nas falhas na *cache* L1 mcf

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	283042	303073	6.60	314868	10.10	304272	6.97
1	37872	34408	−10.06	36744	−3.06	34452	−9.92
2	864	727	−18.84	839	−2.97	733	−17.87
3	1968	1840	−6.95	1862	−5.69	1631	−20.66
4	552	583	5.31	862	35.96	631	12.51

Tabela A.30: Redução nas falhas na *cache* L2 mcf

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	5353808879	5353802978	0.00	7485579970	28.47	5353808639	0.00
1	722525114	722525152	0.00	1051535662	31.28	722525634	0.00
2	12963738	12963739	0.00	18165468	28.63	12963738	0.00
3	52534686	52534682	0.00	84644214	37.93	52534692	0.00
4	6678066	6678066	0.00	12622882	47.09	6678066	0.00

Tabela A.31: Redução nos desvios tomados mcf

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	556680493	556680820	0.00	609436188	8.65	556688607	0.00
1	93036141	93036183	0.00	103810046	10.37	93039051	0.00
2	2639596	2639969	0.00	2863378	7.81	2639632	0.00
3	6072250	6072250	0.00	6943266	12.54	6072250	0.00
4	916617	916659	0.00	968449	5.35	916648	0.00

Tabela A.32: Redução nos desvios mal previstos mcf

A.2.5 Programa parser

Como podemos ver na Tabela A.33, as falhas na *cache* L1 tiveram uma pequena redução, em média **8.64%** melhor do que o otimizador O3, **4.25%** melhor em relação ao histórico, a evolução das execuções e **4.49%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.34, as falhas na *cache* L2 tiveram uma redução um pouco melhor, em média **10.32%** melhor do que o otimizador O3, **4.53%** melhor em relação ao histórico, a evolução das execuções e **1.83%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.35 e na Tabela A.36, somente ocorreu uma melhora em relação ao otimizador O3, **27.25%** melhor no caso dos desvios tomados e somente **6.79%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

A.2.6 Programa twolf

Como podemos ver na Tabela A.37, as falhas na *cache* L1 tiveram uma pequena redução, em média **8.23%** melhor do que o otimizador O3, **2.80%** melhor em relação

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	6355465	6604772	3.77	6870517	7.49	6618928	3.98
1	213238	233824	8.80	265239	19.60	236077	9.67
2	300861	332615	9.54	357392	15.81	334088	9.94
3	70060	73205	4.29	90268	22.38	73395	4.54
4	24863	28980	14.20	36785	32.40	29034	14.36
5	7765	9087	14.54	11719	33.74	9102	14.68

Tabela A.33: Redução nas falhas na *cache* L1 parser

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	4776362	4998634	4.44	5312742	10.09	4860303	1.72
1	187980	210510	10.70	215004	12.56	190317	1.22
2	172187	175807	2.05	197230	12.69	183440	6.13
3	56890	54916	−3.59	63596	10.54	55250	−2.96
4	14548	15379	5.40	18032	19.32	15780	7.80
5	2720	2780	2.15	3833	29.03	2902	6.27

Tabela A.34: Redução nas falhas na *cache* L2 parser

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	35416539360	35416539362	0.00	49049229153	27.79	35416539275	0.00
1	914924181	914924181	0.00	1049741505	12.84	914924026	0.00
2	439381871	439381869	0.00	509965133	13.84	439381720	0.00
3	311293661	311293663	0.00	369310909	15.70	311293512	0.00
4	56638769	56638770	0.00	83079610	31.82	56638620	0.00
5	24200208	24200106	0.00	27971023	13.48	24200406	0.00

Tabela A.35: Redução nos desvios tomados parser

ao histórico, a evolução das execuções e **4.74%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.38, as falhas na *cache* L2 tiveram uma redução bem parecida com a anterior, em média **9.58%** melhor do que o otimizador O3, **1.63%** melhor em relação ao histórico, a evolução das execuções e **5.55%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.39 e na Tabela A.40, somente ocorreu uma melhora em relação ao otimizador O3, **13.91%** melhor no caso dos desvios tomados

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	3262038742	3262045378	0.00	3484370935	6.38	3262039011	0.00
1	86878068	86879038	0.00	105010748	17.27	86871304	0.00
2	38980040	38984958	0.00	43513876	10.41	38980698	0.00
3	30615121	30618306	0.00	35405511	13.53	30618366	0.00
4	6900465	6906045	0.00	7029822	1.84	690098	0.00
5	4036228	4036354	0.00	4163884	3.06	4036623	0.00

Tabela A.36: Redução nos desvios mal previstos parser

e somente **2.11%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	2780347	2852777	2.53	3003269	7.42	2914320	4.59
1	124047	134542	7.80	154918	19.92	136564	9.16
2	165650	176731	6.26	194018	14.62	166797	0.68
3	36345	29317	−23.97	30425	−19.45	29333	−23.90
4	36951	40781	9.39	42887	13.84	52784	29.99

Tabela A.37: Redução nas falhas na *cache* L1 twolf

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	409060	425949	3.96	450183	9.13	445846	8.25
1	63659	54350	−17.12	71612	11.10	54607	−16.57
2	1952	2362	17.35	2548	23.39	2099	7.00
3	3267	3362	2.82	4019	18.71	3364	2.88
4	1920	1824	−5.26	2374	19.12	2148	10.61

Tabela A.38: Redução nas falhas na *cache* L2 twolf

A.2.7 Programa vortex

Como podemos ver na Tabela A.41, as falhas na *cache* L1 tiveram uma redução em média **11.08%** melhor do que o otimizador O3, **7.28%** melhor em relação ao histórico, a evolução das execuções e **7.42%** melhor que o algoritmo de Pettis e Hansen.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	28388451811	28388453961	0.00	32779012374	13.39	28388458361	0.00
1	1084047590	1084047652	0.00	1452874557	25.38	1084047612	0.00
2	18155374	18155265	0.00	21874557	17.00	18155814	0.00
3	62207806	62207805	0.00	75743857	17.87	62207931	0.00
4	5661033	5661032	0.00	7455409	24.06	5661451	0.00

Tabela A.39: Redução nos desvios tomados twolf

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	6084002652	6084093247	0.00	6213981601	2.09	6084064123	0.00
1	253065873	253084347	0.00	258874541	2.24	253068474	0.00
2	4548485	4548767	0.00	4969593	8.47	4548597	0.00
3	15910061	15944033	0.00	17126363	7.10	15951105	0.00
4	1473037	1473567	0.00	1483329	0.69	1473927	0.00

Tabela A.40: Redução nos desvios mal previstos twolf

Como podemos ver na Tabela A.42, as falhas na *cache* L2 tiveram uma redução um pouco melhor, em média **17.50%** melhor do que o otimizador O3, **8.29%** melhor em relação ao histórico, a evolução das execuções e **9.98%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.43 e na Tabela A.44, somente ocorreu uma melhora em relação ao otimizador O3, **22.13%** melhor no caso dos desvios tomados e somente **7.40%** melhor nos desvios mal previstos. Os desvios tomados e os desvios mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

A.2.8 Programa vpr

Como podemos ver na Tabela A.45, as falhas na *cache* L1 tiveram uma redução em média **12.36%** melhor do que o otimizador O3, **5.07%** melhor em relação ao histórico, a evolução das execuções e **10.23%** melhor que o algoritmo de Pettis e Hansen.

Como podemos ver na Tabela A.46, as falhas na *cache* L2 tiveram uma redução um pouco melhor, em média **19.04%** melhor do que o otimizador O3, **9.85%** melhor em relação ao histórico, a evolução das execuções e **13.65%** melhor que o algoritmo

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	142389697	144538506	1.48	151574799	6.05	144578475	1.51
1	185888349	216189606	14.01	218316007	14.85	216733587	14.23
2	158019235	160193628	1.35	173329075	8.83	160459870	1.52
3	23226980	28664004	18.96	28728434	19.14	28717316	19.11
4	13424521	14171513	5.27	15668467	14.32	14139734	5.05
5	2092617	2521689	17.01	2825968	25.95	2517185	16.86
6	607343	650889	6.69	708111	14.23	656656	7.50
7	161545	173275	6.76	205157	21.25	174531	7.44

Tabela A.41: Redução nas falhas na *cache* L1 vortex

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	8159479	8557778	4.65	9885765	17.46	8640033	5.56
1	10218590	11267138	9.30	12334643	17.15	11359252	10.04
2	8948687	10001286	10.52	10871779	17.68	10403491	13.98
3	1002303	1046526	4.22	1239376	19.12	1051734	4.69
4	253991	294858	13.85	316843	19.83	297643	14.66
5	29796	32743	9.00	36008	17.25	33843	11.95
6	16864	19414	13.13	22180	23.96	18742	10.02
7	10725	12801	16.21	12500	14.2	11055	2.98

Tabela A.42: Redução nas falhas na *cache* L2 vortex

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	11495107212	11495107463	0.00	16361496465	29.74	11495107505	0.00
1	11553151752	11553151735	0.00	14779348256	21.82	11553151653	0.00
2	12905204682	12905204705	0.00	15009760091	14.02	12905204622	0.00
3	1362414613	1362414613	0.00	1842890289	26.07	1362414613	0.00
4	774038291	774038304	0.00	933734013	17.10	774038310	0.00
5	86126474	86126476	0.00	104651425	17.70	86126466	0.00
6	31455419	31455420	0.00	38272012	17.81	31455404	0.00
7	6275445	6275445	0.00	7653618	18.00	6275421	0.00

Tabela A.43: Redução nos desvios tomados vortex

de Pettis e Hansen.

Como podemos ver na Tabela A.47 e na Tabela A.48, somente ocorreu uma melhora em relação ao otimizador O3, **27.45%** melhor no caso dos desvios tomados e somente **5.11%** melhor nos desvios mal previstos. Os desvios tomados e os desvios

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	214958496	214995481	0.00	224012401	4.04	214990752	0.00
1	292626249	292604828	0.00	322651338	9.31	292626210	0.00
2	23987113	23984051	0.00	24943541	3.84	23983489	0.00
3	39933100	39939626	0.00	46333877	13.81	39935897	0.00
4	21671543	21670512	0.00	22533728	3.82	21672340	0.00
5	3560844	3569932	0.00	3854918	7.62	3564447	0.00
6	1050289	1058395	0.00	1233705	14.86	1054686	0.00
7	326508	326934	0.00	336774	3.04	326890	0.00

Tabela A.44: Redução nos desvios mal previstos vortex

mal previstos não tiveram ganhos e nem perdas significativas (0%) em relação ao histórico e em relação ao algoritmo de Pettis e Hansen.

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	587152	613842	4.34	670255	12.39	657043	10.63
1	159274	176586	9.80	172484	7.65	179386	11.21
2	47862	48063	0.41	61313	21.93	48674	1.66
3	20183	20346	0.80	25297	20.21	23129	12.73
4	9344	9511	1.75	10525	11.22	10065	7.16
5	4340	4653	6.72	5239	17.15	4825	10.05
6	3165	2696	−17.39	3104	−1.92	2909	−8.08
7	2131	2496	14.62	2868	25.69	2679	20.45
8	2144	2003	−7.03	2087	−2.73	1934	−10.85
9	1045	1144	8.65	1532	31.78	1352	22.70

Tabela A.45: Redução nas falhas na *cache* L1 vpr

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	93366	103115	9.45	115021	18.82	115236	18.97
1	71815	80955	11.29	89629	19.87	71815	0.00
2	70714	78873	10.34	86734	18.47	88674	20.25
3	11168	11407	2.09	14872	24.90	12231	8.69
4	1596	1614	1.11	2318	31.14	1721	7.26
5	2258	2321	2.71	3050	25.96	2488	9.24
6	949	992	4.33	1130	16.01	1072	11.47
7	918	949	3.26	1186	22.59	1072	14.36
8	988	940	−5.10	1570	37.07	1213	18.54
9	1327	960	−38.22	1084	−22.41	1178	−12.64

Tabela A.46: Redução nas falhas na *cache* L2 vpr

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	4815972147	4815973574	0.00	6900088975	30.20	4815979046	0.00
1	5471899396	5471899987	0.00	7177556287	23.76	5471899717	0.00
2	714050189	714050971	0.00	1115662902	35.99	714050997	0.00
3	683958909	683958718	0.00	929749016	26.43	683958982	0.00
4	83337133	83337126	0.00	97396889	14.43	83337447	0.00
5	43101896	43101896	0.00	62398552	30.92	43101898	0.00
6	10901357	10907836	0.00	15572897	29.99	10902146	0.00
7	11932548	11932549	0.00	15783124	24.39	11932694	0.00
8	850482	850704	0.00	1100234	22.69	850747	0.00
9	900846	900948	0.00	1357174	33.62	900889	0.00

Tabela A.47: Redução nos desvios tomados vpr

Entrada	$NI_{\text{último}}$	$NI_{\text{prim.}}$	$g_{\text{prim.}}$	NI_{O3}	g_{O3}	NI_{PH}	g_{PH}
0	1585134173	1585177547	0.00	1653853431	4.15	1585199615	0.00
1	959548232	959579057	0.00	1013858673	5.35	959569734	0.00
2	192203340	192221713	0.00	217580261	11.66	1922803212	0.00
3	120432872	120432074	0.00	126313951	4.65	120479132	0.00
4	27438830	27474005	0.00	28857631	4.75	27485494	0.00
5	7750101	7757636	0.00	7864671	1.45	7751740	0.00
6	3767408	3761814	0.00	4017239	6.35	3762001	0.00
7	343101	343791	0.00	378876	9.44	343001	0.00
8	327290	327370	0.00	348522	6.09	327390	0.00
9	304258	304233	0.00	348522	12.70	304278	0.00

Tabela A.48: Redução nos desvios mal previstos vpr

Referências Bibliográficas

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, pages 1319–1360, 1994.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[4] V. C. Barbosa. *Massively Parallel Models of Computation*. Ellis Horwood, Chichester, UK, 1993.

[5] R. Bodk and R. Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 159–170. ACM Press, 1997.

[6] R. Bodk, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 146–158. ACM Press, 1997.

[7] D. Burger, J. R. Goodman, and A. Kgi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 78–89. ACM Press, 1996.

[8] B. Calder, D. Grunwald, D. Lindsayi, J. Martin, M. Mozer, and B. Zorn. Corpus-based static branch prediction. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 79–92, 1995.

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

[10] Y. Chou and J. P. Shen. Instruction path coprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 270–281, 2000.

[11] C. Cifuentes and M. V. Emmerik. UQBT: adaptable binary translation at low cost. *Computer*, pages 60–66, 2000.

[12] C. Cifuentes, M. V. Emmerik, and N. Ramsey. The design of a resourceable and retargetable binary translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, page 280. IEEE Computer Society, 1999.

[13] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing alpha executables on windows NT with spike. *Digital Tech. J.*, pages 3–20, 1998.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* The MIT Press, 2001.

[15] Compaq Corporation. Release notes for NT-ATOM 1.53.

[16] Digital Equipment Corporation. ATOM user manual, 1994.

[17] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 222–233. IEEE Computer Society, 1999.

[18] S. Debray. The PLTO binary rewriting system - a user manual, 2002.

[19] P. J. Denning. The working set model for program behavior. *Commun. ACM*, pages 323–333, 1968.

[20] C. Dulong. The IA-64 architecture at work. *Computer*, pages 24–32, 1998.

[21] E. S. T. Fernandes. *Paralelismo a nível de instrução e o custo de desvios.* 11 Escola de Computação, 1998.

[22] D. Ferrari. Improving locality by critical working sets. *Communications of the ACM*, pages 614–620, 1974.

[23] D. Ferrari. The improvement of program behavior. *Computer*, pages 39–47, 1976.

[24] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, pages 977–1027, 1999.

[25] S. L. Graham, P. B. Kessler, and M. K. Mckusick. GPROF: a call graph execution profiler. *SIGPLAN Not.*, pages 49–57, 2004.

[26] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Modern compiler design.* John Wiley Sons, Ltd., 2001.

[27] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, page 102. IEEE Computer Society, 1997.

[28] R. Gupta, D. A. Berson, and J. Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 358–368. IEEE Computer Society, 1997.

[29] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference on Computer Languages*, page 230. IEEE Computer Society, 1998.

[30] R. Gupta and C. H. Chi. Improving instruction cache behavior by reducing cache pollution. In *Proceedings of the 1990 Conference on Supercomputing*, pages 82–91, 1990.

[31] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Transactions on Software Engineering*, pages 1640–1644, 1988.

[32] A. Hashemi, D. Kaeli, and B. Calder. Procedure mapping using static call graph estimation, 1997.

[33] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 171–182, 1997.

[34] J. L. Hennessy and D. A. Patterson. *Computer architecture.* Elsevier Science, 2003.

[35] D. J. Howarth. Experience with the atlas scheduling system. In *Proceedings of the AFIPS SJCC*, pages 59–67, 1963.

[36] D. J. Howarth, P. D. Jones, and M. T. Wyld. The atlas scheduling system. *Computer Journal*, pages 238–244, 1962.

[37] D. J. Howarth, R. B. Payne, and F. H. Sumner. The manchester university atlas operating system. *Computer Journal*, pages 226–229, 1961.

[38] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th annual international symposium on Computer architecture*, pages 242–251. ACM Press, 1989.

[39] Intel. Intel-vtune, www.intel.com/software/products/vtune/.

[40] J. Johnson. Program restructuring for virtual systems. Technical report, 1975.

[41] J. Kalamatianos and D. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the The Fourth International Symposium on High-Performance Computer Architecture*, page 244. IEEE Computer Society, 1998.

[42] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner. The manchester university atlas operating system. *Computer Journal*, pages 222–225, 1961.

[43] D. E. Knuth. An empirical study of fortran programs. *In Software, Practice and Experience*, pages 105–133, 1971.

[44] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 259–269. ACM Press, 1999.

[45] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, pages 197–218, 1994.

[46] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. *SIGPLAN Not.*, pages 291–300, 1995.

[47] J. R. Levine and Morgan. *Linkers and loaders.* Kauffman, 2000.

[48] K. London. PAPI, http://icl.cs.utk.edu/papi/.

[49] K. London, J. Dongarra, Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on linux systems.

[50] C. G. N. Manning and I. H. Witten. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the Conference on Data Compression*, page 3. IEEE Computer Society, 1997.

[51] S. McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

[52] S. McFarling. Procedure merging with instruction caches. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 71–79. ACM Press, 1991.

[53] E. Mehofer and B. Scholz. A novel probabilistic data flow framework. *Lecture Notes in Computer Science*, pages 37–54, 2001.

[54] Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, pages 1087–1091, 1953.

[55] R. Muth, S. Debray, S. Watterson, and K. Bosschere. ALTO: A link-time optimizer for the DEC alpha. Technical Report TR98-14, Wednesday, 9 1998.

[56] D. A. Patterson and J. L. Hennessy. *Computer organization & design: the hardware/software interface.* Morgan Kaufmann Publishers Inc., 1998.

[57] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, 1988.

[58] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–27, 1990.

[59] GNU project free software foundation. Gnu manual online, 1997.

[60] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 155–164, 2001.

[61] T. Romer, G. V. D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. pages 1–8.

[62] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, NJ, second edition, 2003.

[63] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the intel IA-32 architecture. In *Proceedings of the Workshop on Binary Rewriting*, 2001.

[64] N. Snavely, S. K. Debray, and G. Andrews. Predicate analysis and if-conversion in an itanium. In *Proc. Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques (EPIC-2)*, 2002.

[65] SPEC CINT2000. `http://www.spec.org/cpu2000/CINT2000`.

[66] Y. N. Srikant and P. Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation.* CRC Press, 2002.

[67] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205. ACM Press, 1994.

[68] A. Srivastava and D. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, pages 1–18, December 1992.

[69] W. Stallings. *Computer organization and architecture: designing for performance.* Prentice Hall PTR, 2002.

[70] J. L. Szwarzcfiter. *Grafos e algoritmos computacionais.* Editora Campus, 1984.

[71] J. L. Szwarzcfiter and L. Markezon. *Estrutura de dados e seus algoritmos.* Editora LTC, 1994.

[72] A. Tanenbaum. *Structured computer organization; (4nd ed.).* Prentice-Hall, Inc., 1999.

[73] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, pages 2–7, 2001.

[74] C. Young, D. S. Johnson, M. D. Smith, and D. R. Karger. Near-optimal intraprocedural branch alignment. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 183–193. ACM Press, 1997.

[75] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 232–241. ACM Press, 1994.

[76] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 180–190. ACM Press, 2001.

[77] M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*, pages 317–327. IEEE Computer Society, 2005.