View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositorio da Producao Cientifica e Intelectual da Unicamp

Compressão de Código Baseada em Multi-Profile

Eduardo Bráulio Wanderley Netto

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Compressão de Código Baseada em Multi-Profile

Eduardo Bráulio Wanderley Netto[1]

21 de maio de 2004

Banca Examinadora:

- Paulo Cesar Centoducatte (Orientador)
- Manoel Eusébio de Lima
  Centro de Informática - UFPE
- Osamu Saotome
  Departamento de Eletrônica Aplicada - ITA
- Nelson Castro Machado
  Instituto de Computação - Unicamp
- Ricardo Pannain
  Instituto de Computação - Unicamp
- Edna Natividade da Silva Barros (Suplente)
  Centro de Informática - UFPE
- Mario Lúcio Côrtes (Suplente)
  Instituto de Computação - Unicamp

[1]Financiado em parte pelo CNPq, processo 140631/2001-1.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Wanderley Netto, Eduardo Bráulio

W183c Compressão de código baseada em multi-profile / Eduardo Bráulio Wanderley Netto -- Campinas, [S.P. :s.n.], 2004.

Orientador : Paulo Cesar Centoducatte

Co-orientador: Rodolfo Jardim de Azevedo

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Compressão de dados (Computação). 2. Arquitetura de computadores. 3. Sistemas embutidos de computadores. I. Centoducatte, Paulo Cesar. II. Azevedo, Rodolfo Jardim de. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Compressão de Código Baseada em Multi-Profile

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Eduardo Bráulio Wanderley Netto e aprovada pela Banca Examinadora.

Campinas, 02 de junho de 2004.

Paulo Cesar Centoducatte (Orientador)

Rodolfo Jardim de Azevedo (Co-orientador)

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

Substitua pela folha com a assinatura da banca

© Eduardo Bráulio Wanderley Netto, 2004.
Todos os direitos reservados.

Resumo

A compressão de códigos de programas representa uma alternativa para diminuição de área de silício usada na fabricação de chips para sistemas embarcados. Este requisito tem sido fortemente influenciado pela crescente funcionalidade, incluíndo aplicações em multimídia, exigida para os softwares que neles executam. Recentes estudos apresentam a compressão de código como alternativa também para melhorar o desempenho e reduzir o consumo de energia nestes sistemas. Este trabalho apresenta um novo método de compressão, o ComPacket, baseado em pequenos dicionários incompletos com um descompressor em hardware situado entre a cache e o processador (RISC), permitindo assim que a cache guarde o código comprimido e portanto possibilitando uma maior capacidade de armazenamento. Além disto, um novo paradigma de construção de dicionários é introduzido de tal forma a propiciar uma melhor exploração da tríade de requisitos área-desempenho-consumo de energia. Este paradigma baseia-se ao mesmo tempo em informações estatísticas obtidas de *profiles* dinâmico e estático do uso de instruções em um programa e por isto é denominado *Multi-Profile*. Foram realizados experimentos de uso de dicionários *Multi-Profile* em dois métodos de compressão de código: o *Instruction Based Compression* (IBC), desenvolvido anteriormente em nosso laboratório e o novo ComPacket. Para o IBC, a razão de compressão média varia entre 71% e 77% para um conjunto de aplicações retiradas das suites *Mediabench* e *MiBench*, enquanto o número de ciclos de clock usados para execução do código comprimido varia em média de 75% a 65% dos valores obtidos sem compressão. Usando o mesmo conjunto de aplicações e o ComPacket, a razão de compressão média varia entre 72% e 88%, o número de ciclos de clock chega a 52% do original para uma construção específica do dicionário e a redução no consumo de energia na cache de instruções chega a 46% do valor original (sem compressão).

Abstract

Code compression is an approach to reduce the silicon area requirement to produce embedded systems chips. This requirement is stronlgy affected by the increasing functionality, including multimedia applications, required by the embedded softwares. Recently, some researches point out the code compression as an alternative to increase performance and reduce energy consumption. This work introduces a new code compression method, the ComPacket, based on small, incomplete dictionary and a new decompressor hardware which is located between the cache and the processor (RISC), thus making the cache to keep compressed instructions augmenting its storage capacity. Moreover, a new paradigm to build dictionaries is introduced, such that a better exploration of area-performance-energy consumption trade-offs is achieved. This paradigm is based on both dynamic and static profiles informations at the same time, which led the name of Multi-Profile. We used this paradigm on two code compression scheme: the Instruction Based Compression (IBC), formerly developed in our Laboratory, and the new ComPacket. For the IBC, the average compression ratio varies from 71% to 77% for the benchmarks excerpted from Mediabench and MiBench suites, while a cycle count reduction of 75% to 65% were achieved (related to original uncompressed execution of the code). For the ComPacket, the average compression ratio varies from 72% to 88% and the cycle count reduction is as low as 52% for a special case of dicionary construction. The instruction cache energy reduction reaches 46% of the original.

Agradecimentos

Agradeço primeiro a Deus por ter me dado a chance de desafiar os meus próprios limites e me aventurar de forma definitiva na pesquisa científica.

Ao professor orientador Paulo Centoducatte, que nunca mediu esforços para vencer os desafios deste trabalho. Meu profundo e sincero agradecimento.

Ao professor Rodolfo Azevedo, por ter assumido junto com Paulo este trabalho de orientação e portanto vivido os mesmos desafios, as mesmas angústias, mas também, experimentado as mesmas vitórias.

Ao professor Guido Araújo, pela dedicação ao trabalho que tantas vezes me inspirou.

Aos colegas de sala, com os quais aprendi a viver em um salutar ambiente de pesquisa. Em especial a Nahri Moreano, por tanta paciência em ler, reler, e criticar sabiamente os manuscritos de artigos, que ao longo desta jornada foram preparados.

Aos tantos mestres do IC que não pecaram em ensinar em suas disciplinas que desafios são feitos para serem vencidos.

Ao CNPq, pela bolsa que me propiciou um aporte finaceiro adequado à minha estada e de minha família nesta terra longínqua de Campinas.

Ao CEFET/RN que acreditou e apostou em minha capacitação.

Finalmente, a minha esposa, que soube entender os tempos de ausência dolorosa, mas necessária, que compõem esta etapa da formação acadêmica. A minha filha, pequenina aprendiz, que me tirou tantas vezes de um estresse sem par, com apenas um sorriso... A minha mãe que a exemplo de Moisés nunca acomodou o braço de sua oração. E a todos, família de sangue, de credo, de convívio (Rivanaldo e Mônica em especial), de alegrias, ... Meus mais sinceros agradecimentos.

À Elker,
À Esther,
Às minhas tantas famílias:
de sangue, de fé, de trabalho...

Sumário

Resumo **vii**

Abstract **viii**

Agradecimentos **ix**

Acrônimos **xviii**

1 Introdução **1**

1.1 Introdução à compressão 3
1.1.1 Compressão de Código 5
1.1.2 Métricas de Avaliação de Compressão de Código 7
1.2 Contribuição 8
1.3 Organização 9

2 Trabalhos Relacionados **11**

2.1 Compactação de Código 11
2.2 Compressão CDM 13
2.3 Compressão PDC 20
2.3.1 Sistemas sem Cache 20
2.3.2 Sistemas com Cache 24
2.4 Conjunto de Instruções em 16/32 bits 28
2.5 Outros Trabalhos 30
2.5.1 Compressão para VLIW 30
2.5.2 *Wire Code* 31
2.5.3 Compressão Seletiva 32
2.6 Sumário e Conclusões 33

3 Dicionários Baseados em *Multi-Profile* **37**

3.1 Redundância de Instruções 37

3.2 Similaridade Entre os Dicionários 42
3.3 Influência dos Dados de Entrada 45
3.4 Estudo de Dicionários Incompletos Pequenos 45
3.4.1 Similaridade Entre Dicionários Pequenos 46
3.4.2 Influência dos Dados de Entrada em Dicionários Pequenos 49
3.4.3 Contribuição das Instruções de um Dicionário Pequeno na Construção de um Programa 49
3.5 Método *Multi-Profile* para Construção de Dicionários 51
3.5.1 Exemplo de Funcionamento do Algoritmo 52
3.6 Considerações Finais e Conclusões 55
3.6.1 Conclusões 58

4 O Método CDM-IBC *Multi-Profile* 59
4.1 Análise do Método IBC 59
4.1.1 Divisão em Classes 60
4.1.2 Tabela de Conversão de Endereços 62
4.1.3 Geração do Código Comprimido 64
4.2 Hardware Descompressor do IBC 65
4.3 IBC *Multi-Profile* 66
4.3.1 Análise da Razão de Compressão do IBC *Multi-Profile* 69
4.3.2 Análise de desempenho do IBC 71
4.3.3 Influência dos dados de entrada no desempenho do IBC 78
4.4 Considerações Finais e Conclusões 79

5 O Método PDC-ComPacket 83
5.1 Análise do Método PDC-ComPacket 83
5.2 Codificação das *Codewords*: ComPacket 86
5.3 O Método de Compressão 87
5.4 Hardware Descompressor do PDC-ComPacket 94
5.5 Considerações Finais e Conclusões 95

6 O PDC-ComPacket *Multi-Profile* 99
6.1 Dicionários *Multi-Profile* Revisitados 99
6.2 Infraestrutura Utilizada 100
6.3 Resultados Obtidos 102
6.3.1 Um Caso Selecionado de f 107
6.3.2 Influência dos dados de entrada 109
6.4 Análise Comparativa dos Resultados 112
6.5 Implementação do Hardware Descompressor 119

6.5.1 O processador Leon 119
6.5.2 O *kit* de desenvolvimento XESS XSV800 120
6.5.3 Descrição da implementação do hardware do PDC-ComPacket . . . 120
6.6 Considerações Finais e Conclusões 121

7 Conclusões e Trabalhos Futuros **125**
7.1 Trabalhos Futuros 126

Bibliografia **129**

Lista de Tabelas

2.1 Compêndio dos métodos de compactação 34
2.2 Compêndio dos métodos CDM 35
2.3 Compêndio dos métodos PCD 36
2.4 Compêndio dos ISAs mistos de 16/32 bits 36

3.1 *benchmarks* utilizados 38
3.2 Número e procentagem de instruções únicas para os *benchmarks* 39
3.3 Similiaridade entre os dicionários completos dos diversos *benchmarks* . . . 57
3.4 Similiaridade entre os dicionários pequenos (256 entradas) dos diversos *benchmarks* 57

4.1 Divisão em classes de um dicionário completo para o método IBC 61
4.2 Exemplo de Código a ser Comprimido 65
4.3 Taxa de acerto nas caches (em porcentagem) 71
4.4 Número de acessos à memória principal 73
4.5 Razão de Compressão usando diferentes dados de entrada 78

6.1 Taxa de acerto nas caches (em porcentagem) 103
6.2 Redução nas transições de bits nos barramentos de entrada da cache 105
6.3 Redução nas transições de bits nos barramentos de entrada da cache 107
6.4 Redução de energia na cache de instruções 109
6.5 Influência dos dados de entrada na razão de compressão do PDC-ComPacket 110
6.6 Influência dos dados de entrada no desempenho do sistema 110
6.7 Influência dos dados de entrada no desempenho do sistema 111
6.8 Comparação do PDC-ComPacket com métodos de compactação 113
6.9 Comparação do PDC-ComPacket com métodos CDM 114
6.10 Comparação do PDC-ComPacket com outros métodos PCD 117
6.11 Comparação do PDC-ComPacket com ISAs mistos de 16/32 bits 118

Lista de Figuras

1.1 Exemplo de Compressão Adaptativa 5
1.2 Arquiteturas para descompressão de código 6
1.3 Desalinhamento entre os endereços do código original e comprimido 7

2.1 Diagrama do CCRP 14
2.2 Árvores de prefixos usadas no CodePack da IBM 17
2.3 Abordagem de Lefurgy para execução de código comprimido 22
2.4 Abordagem de Lekatsas para execução de código comprimido 25
2.5 Esquema da linha de cache comprimida (Benini) 27
2.6 Mapeamento de campos de uma instrução Thumb para sua correspondente ARM 29
2.7 *Pipeline* do ARM/Thumb 29

3.1 Porcentagem do programa coberta pelas instruções únicas 40
3.2 Porcentagem da execução do programa coberta pelas instruções únicas . . 41
3.3 Grau de interseção entre os dicionários estático e dinâmico 43
3.4 Contribuição percentual das instruções dentro dos dicionários estático e dinâmico (normalizado pela maior contribuição) 44
3.5 Similaridade entre dicionários dinâmicos formados por dados de entrada distintos 46
3.6 Similaridade entre dicionários pequenos estáticos e dinâmicos 47
3.7 Distribuição da contribuição de cada instrução dentro do dicionário estático (normalizada) 48
3.8 Distribuição da contribuição de cada instrução dentro do dicionário dinâmico (normalizada) 48
3.9 Similaridade entre dicionários dinâmicos para dois conjuntos de dados de entrada 49
3.10 Contribuição acumulada das instruções (estáticas) de um pequeno dicionário na formação de um programa 50

3.11 Contribuição acumulada das instruções (dinâmicas) de um pequeno dicionário na formação de um programa 51
3.12 Algoritmo de construção de um dicionário unificado 52
3.13 Exemplo de funcionamento do algoritmo de unificação dos dicionários . . . 54

4.1 Implementações da IT 62
4.2 Diagrama de blocos da ATT 64
4.3 Conteúdo da memória para um trecho do código comprimido pelo IBC . . 65
4.4 *Pipeline* do descompressor IBC para o Leon 66
4.5 Família de curvas para distribuição das contribuições de um dicionário unificado (com f variando de 10% em 10% entre S e D) 68
4.6 Composição da razão de compressão do IBC 69
4.7 Razão de compressão em função de f 70
4.8 Taxa de acerto nas caches para o código original 72
4.9 Acessos à memória principal em função de f 73
4.10 Composição das falhas na cache 75
4.11 Ciclos de *clock* utilizados (em relação ao original) 76
4.12 Influência do fator dinâmico no desempenho do sistema (para memórias com respostas entre 1 e 64 ciclos) 77
4.13 Influência dos dados de entrada no desempenho do sistema 79

5.1 Exemplo de *patching* do código 85
5.2 Codificação usada nos ComPackets 87
5.3 Algoritmo de compressão do código 89
5.4 Exemplo do algoritmo de compressão: Passo 1 89
5.5 Exemplo do algoritmo de compressão: Passo 2 90
5.6 Exemplo do algoritmo de compressão: Passo 3 90
5.7 Exemplo do algoritmo de compressão: Passo 4 91
5.8 Exemplo do algoritmo de compressão: Passo 5 91
5.9 Exemplo do algoritmo de compressão: Passo 6 91
5.10 Exemplo do algoritmo de compressão: Passo 7 92
5.11 Exemplo do algoritmo de compressão: Passo 8 93
5.12 Exemplo do algoritmo de compressão: Final. 93
5.13 Diagrama de bloco do hardware descompressor do PDC-ComPacket 94
5.14 Lógica de descompressão para o PDC-ComPacket 95

6.1 Similaridade entre os dicionários dinâmicos (com restrições do PDC-ComPacket) 100
6.2 Infraestrutura utilizada para avaliar os algoritmos de compressão 101

6.3 Taxa de acerto nas caches para o PDC-ComPacket 104
6.4 Razão de compressão do PDC-ComPacket em função de f 105
6.5 Acessos à cache, em relação ao original, em função de f 106
6.6 Redução no número de ciclos devido a compressão do código 108
6.7 Influência dos dados de entrada no consumo de energia da cache de instruções 111
6.8 Diagrama de blocos do descompressor PDC-ComPacket para FPGA 121

Lista de Acrônimos

Relacionamos a seguir os principais acrônimos usados neste texto e que têm relação direta com compressão de código.

ATT	Address Translation Table
BRISC	Byte-coded RISC
CCRP	Compressed Code RISC Processor
CDM	Cache-Decompressor-Memory
CLB	Cache Line Address Lookaside Buffer
DCR	Decompression on Cache Refil
DD	Dynamic Dictionary
DF	Decompression of Fetch
EPM	External Pointer Macro
GOA	General Offset Assignement
IBC	Instruction Based Compression
IT	Instruction Table
LAT	Line Address Table
LZW	Lempel-Ziv-Welch
MTF	Move to Front
PBC	Pattern Based Compression
PDC	Processor-Decompressor-Cache
PPM	Prediction by Partial Matching
SADC	Semi-Adaptive Dictionary Compression
SAMC	Semi-Adaptive Markov Compression
SD	Static Dictionary
SOA	Single Offset Assignement
TBC	Tree Based Compression
UD	Unified Dictionary

Capítulo 1

Introdução

Na era dos submicrons, milhões de transistores podem ser integrados em uma única pastilha de silício. Hoje, se questionados sobre o destino destes transistores, a resposta mais enfática seria: "a maioria para memória"[1]. O uso eficaz das memórias é, portanto, um requisito de alta relevância no projeto de sistemas de computação. A afirmação se aplica de forma ainda mais veemente quando restringimos o escopo da sentença a sistemas embarcados.

Sistemas embarcados diferem dos sistemas de propósito geral sob muitos aspectos. De forma simplória, poderíamos citar como diferenças básicas: objetivos e métricas de qualidade. Sistemas de propósito geral são construídos para executar uma miríade de pacotes de software muito heterogênea, com requisitos de desempenho e comportamento dinâmico diferentes entre si, enquanto os sistemas embarcados desempenham uma tarefa única, ou um conjunto bem definido delas, durante toda sua existência. Segundo, enquanto o desempenho é o principal (e muitas vezes único) padrão de qualidade pelo qual um sistema de propósito geral é julgado, os projetos de sistemas embarcados costumam ser otimizados para compromissos (*trade-offs*) entre diversas métricas incluindo custos de desenvolvimento e fabricação (área do *die*[1], testabilidade etc.), consumo de energia e desempenho.

Como resultado destas diferenças, estratégias particulares de projetos são utilizadas para cada vertente. Sistemas de propósito geral são tipicamente projetados para executar eficientemente qualquer software que o usuário deseje, portanto são desenvolvidos para um desempenho máximo em um caso típico. Se fossem otimizados para uma aplicação

[1] pastilha de silício usada na fabricação de circuitos integrados

em particular tenderiam a ter um desempenho insatisfatório em outras (muito diferentes). No caso de sistemas embarcados a volatilidade das aplicações é muito menor (se houver alguma). Assim sendo, otimizações específicas podem ocorrer para a aplicação (ou conjunto de aplicações) embarcada.

Um dos recursos mais críticos em sistemas embarcados é a memória. Quando integrada à CPU e periféricos (circuitos de controle, Entrada/Saída etc.), formando um SoC (*System-on-Chip*) típico, este recurso usualmente representa um dos componentes mais caros, em especial porque uma vez dimensionado não pode mais ser alterado. Em boa parte dos SoCs a memória ocupa metade, ou mais, da área de silício e é uma das principais consumidoras de energia. Tudo isto justifica o esforço para otimizar seu uso.

A escolha do processador para compor o SoC também é importante. Por Exemplo: a simples escolha entre um processador RISC e um CISC pode aliviar o requisito de memória do sistema. Em textitkernels[2] de processamento digital de sinais, muito comuns nos softwares atuais para sistemas embarcados, um aumento no tamanho do código quando usamos arquiteturas RISC é bastante comum em relação aos seus pares implementados para uma arquitetura CISC [2]. Entretanto, o desempenho dos processadores RISC, o número de compiladores otimizadores para estas plataformas e a facilidade de encontrar núcleos em forma de IPs (*Intellectual Property*) acabam promovendo um maior uso de tais processadores. Não obstante, o problema da densidade de código precisa ser abordado, principalmente porque os softwares têm se tornado mais complexos, muitas vezes incluindo recursos de multimídia e, portanto, exigindo memórias cada vez maiores.

Na última década muito se estudou acerca do uso de técnicas de compressão de código para satisfazer a um volume de software maior e/ou uma menor área de silício dedicada à memória. Em 1992, Wolfe e Chanin [3] propuseram o *Compressed Code RISC Processor* (CCRP) e introduziram o problema de compressão de código para máquinas RISC. Muitos trabalhos se sucederam, mas, via de regra, todos se dedicaram, prioritariamente, à demanda de memória do sistema.

Recentemente, pesquisadores perceberam que a compressão de código também pode proporcionar uma redução no consumo de energia e mesmo melhorar o desempenho de um sistema computacional. A fundamentação para tais observações é que o número de acessos à memória principal diminui com a compressão e, portanto, o tempo de espera médio dos processadores também diminui. Além disto, gasta-se menos energia ao buscar menos palavras na memória. Infelizmente a descompressão tem um custo associado, seja

[2]Núcleo principal de um programa

em área, desempenho e/ou energia, mas este custo pode ser menor que as benesses obtidas pela compressão.

O trabalho aqui exposto avalia a interferência do modo de construção de dicionários para compressão de código, tendo em vista uma melhor exploração da tríade compressão-desempenho-consumo de energia. Esta avaliação nos levou a propor o primeiro método de construção de dicionários que considera ao mesmo tempo informações estatísticas advindas da contagem de instruções no código (*profile* estático) bem como da contagem de instruções executadas (*profile* dinâmico). Além disto, um novo método de compressão é proposto visando não somente a redução da área de memória ocupada por programas em sistemas embarcados, mas também a melhoria do desempenho e a redução no consumo de energia. Este método é direcionado às arquiteturas RISC dada sua crescente utilização em SoCs.

1.1 Introdução à compressão

Nestes últimos tempos temos testemunhado uma transformação (alguns chamam de revolução) nos processos de comunicação. A Internet, os sistemas de telefonia móveis e a crescente importância da comunicação por vídeo são exemplos desta transformação. A compressão de dados é uma das tecnologias que está por trás deste processo. Seria difícil armazenar e/ou transmitir um volume grande de imagens, áudio e vídeo em sítios *web* sem compressão. Os telefones celulares não teriam o prestígio de um meio de comunicação eficaz se não fosse a compressão. O advento da TV digital seria improvável sem compressão. Hoje, entendendo ou não de compressão, fazemos uso cotidiano dela. Seja num simples *fax*, no *modem*, ou na televisão via satélite, esta tecnologia está presente.

Sob uma perspectiva histórica, as idéias de compressão remontam a tempos antigos. O código Morse[4], desenvolvido pelo lendário Samuel F.B. Morse, é um exemplo. Nele, traços e pontos são usados para simbolizar letras. A atribuição destes signos seguiu um método de compressão, a saber: para letras que mais ocorrem (em inglês) menos traços e pontos são utilizados, diminuindo assim o tamanho das mensagens a serem transmitidas.

Quando nos referimos à compressão, estamos de fato falando de dois algoritmos, um de compressão e outro de descompressão. O algoritmo de compressão recebe uma informação χ e gera uma representação χ_c que utiliza menos espaço (em nosso caso, menos bits). O algoritmo de descompressão opera sobre χ_c para gerar uma reconstrução ξ. Dizemos que a compressão é sem perda (*lossless*) se a reconstrução ξ é exatamente igual à informação

original χ. Se ξ é uma aproximação de χ, então dizemos que a compressão é com perda (*lossy*).

A compressão de texto é um exemplo de compressão que exige um método sem perdas (também chamado de reversível), sob pena da mensagem original ser deturpada. Os métodos de compressão de texto de uma forma geral são classificados em duas categorias: estatístico e de dicionário.

Os métodos estatísticos [4] buscam encontrar as letras (símbolos) que mais ocorrem e atribuir representações (*codewords*) menores para elas. O método de Huffman[5] é um exemplo clássico. Nos métodos de dicionário são selecionadas frases (padrões de símbolos) para compor uma tabela (dicionário). A mensagem comprimida é composta de índices, que também chamamos de *codewords*, para estas frases. A compressão é possível desde que os índices escolhidos sejam, em média, menores que as frases que eles representam.

Formalmente um dicionário D = (M,ϕ) é um conjunto finito de padrões de símbolos M e uma função ϕ que mapeia M em um conjunto de códigos. Os padrões em M são gerados de um alfabeto A. Um dicionário é dito completo se para cada conjunto de dados a ser comprimido há um padrão correspondente em M, isto é, qualquer entrada pode ser construída a partir dos padrões em M. Caso contrário ele é dito incompleto.

Um dicionário é dito estático se ele é construído independentemente da entrada de dados que ele vai comprimir. Um exemplo de dicionário estático é o próprio código Morse. Os símbolos e respectivas *codewords* foram definidos a priori, e independentemente do texto particular que será comprimido (e/ou transmitido). O uso de um dicionário estático pode ser fortemente penalizado pela influência das idiossincrasias de linguagens. Por exemplo, por uma escolha extemporânea, os números em código Morse são representados com grande quantidade de pontos e traços, fazendo com que uma transmissão de uma planilha seja feita de forma ineficiente.

Para resolver este problema, uma abordagem semi-adaptativa pode ser utilizada. Um dicionário semi-adaptativo é aquele cujos padrões de símbolos e *codewords* são definidos para cada entrada particular. Nesta abordagem o resultado da compressão costuma ser um conjunto de dados menor que aquele gerado usando dicionários estáticos, mas o descompressor precisa sempre receber o dicionário e as respectivas *codewords* para realizar a descompressão. Portanto, o custo de armazenar o dicionário deve ser incluído no tamanho do objeto comprimido. Além disto, é preciso ser possível conhecer a entrada de dados em sua totalidade antes de iniciar o processo de compressão. A propósito, dada uma entrada particular, a tarefa de encontrar um dicionário ótimo para ela é um problema

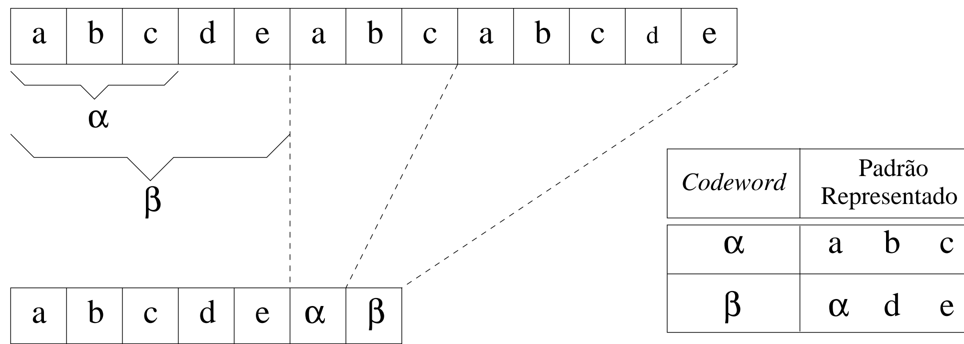


Figura 1.1: Exemplo de Compressão Adaptativa

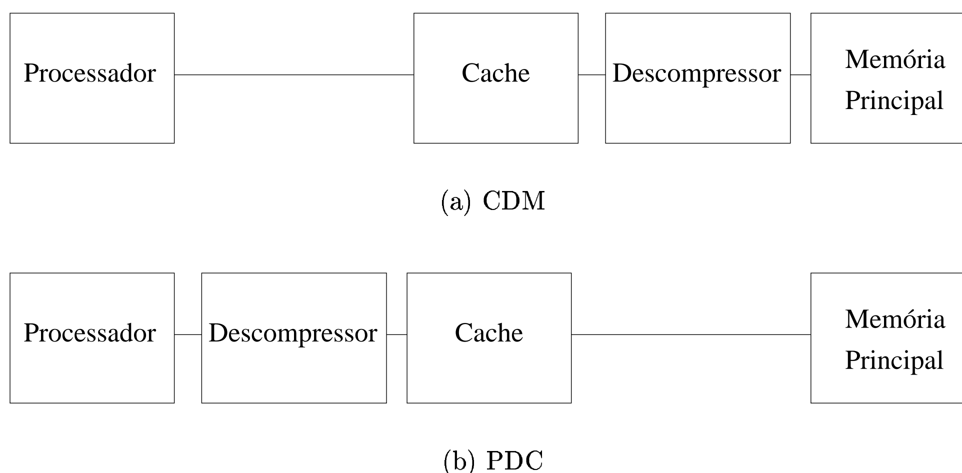
NP completo no tamanho da entrada [6].

A terceira possibilidade, quando não se quer usar um dicionário estático e/ou não se dispõem do conjunto de dados a ser comprimido antes do início da compressão, é usar uma abordagem denominada adaptativa. A idéia é que à medida que os dados vão chegando ao compressor eles formem padrões e quando os mesmos se repetirem apenas um apontador para aquele padrão é inserido. A Figura 1.1 mostra um exemplo clássico. À medida que o texto vai entrando no compressor, ele vai formando padrões e atribuindo *codewords* para os mesmos. Quando um padrão se repete, a *codeword* correspondente é usada para saída. Neste caso, o processo de descompressão requer uma leitura seqüencial de todos os dados para realizar sua tarefa.

1.1.1 Compressão de Código

O propósito da compressão de código é gerar uma imagem do programa original com o uso de menos bits. Limitamos, entretanto, o escopo deste trabalho ao segmento de código dos programas, não incluindo os dados na compressão. Para descomprimir o código é necessário usar uma rotina de descompressão (descompressão por software) ou um artefato de hardware que a implemente (descompressão por hardware).

Naturalmente, para compressão de código é preciso escolher um método de compressão sem perdas, sob pena de reconstruirmos o programa original de forma errada. Quando se deseja que a descompressão seja realizada em tempo de execução, instrução por instrução, também não é possível utilizar abordagens adaptativas, pois as mesmas requerem uma descompressão seqüencial e os programas possuem estruturas de controle de fluxo que exigem que se comece a descompressão em qualquer ponto do código comprimido (ou ao menos no início dos blocos básicos).



(a) CDM

(b) PDC

Figura 1.2: Arquiteturas para descompressão de código

A escolha de um método estatístico ou de dicionário é uma relação de custo/benefício. Todo método baseado em dicionário possui um método estatístico correspondente que produz um código comprimido menor ou igual em tamanho [4]. Por outro lado, em geral, os métodos estatísticos exigem maiores recursos para a descompressão o que a torna mais lenta.

Quando a descompressão de código é realizada por uma entidade de hardware dedicada, esta pode ser posicionada entre a cache e a memória principal ou entre o processador e a cache. Nós denominamos estes esquemas de CDM (*Cache-Decompressor-Memory*) e PDC (*Processor-Decompressor-Cache*) respectivamente. A Figura 1.2 mostra, de forma pictorial, estes dois esquemas de descompressão. Quando uma arquitetura CDM é usada, o descompressor só é invocado quando há uma falha na cache. Se a taxa de acerto é alta, então o tempo de descompressão pode não interferir muito no desempenho do sistema. Por outro lado, se uma arquitetura PDC é utilizada, o código fica comprimido na cache, elevando assim a taxa de acerto. Infelizmente, neste esquema, a descompressão ocorre a cada pedido de instrução do processador, o que exige um descompressor mais rápido (em relação às arquiteturas CDM).

Então, o uso de compressão estatística em um modelo CDM não é proibitivo. Por outro lado é preferencial, e quase mandatório, o uso de métodos baseados em dicionários para arquiteturas PDC.

Um detalhe importante da compressão de código, que via de regra não é tratado

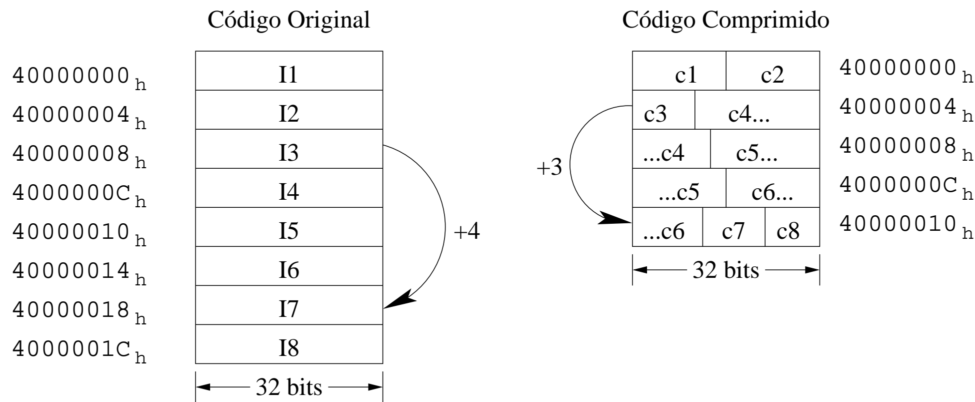


Figura 1.3: Desalinhamento entre os endereços do código original e comprimido

na clássica área de compressão de dados, é o desalinhamento de endereços. O fluxo de controle de um programa é fortemente orientado por endereços e, quando há compressão, os endereços originais se distinguem dos endereços 'comprimidos'. A Figura 1.3 ilustra esta situação. A instrução I3 é um salto relativo para I7. A codificação da instrução I3 inclui um deslocamento para a instrução que está 4 endereços à frente. Quando o código é comprimido, a *codeword* c3 (representação de I3) precisa apontar para *codeword* c7 (representação de I7) que está 3 endereços à frente. Além deste problema ainda existe o fato de que a *codeword* c7 não começa no primeiro bit da palavra de memória.

Para resolver estes problemas costuma-se, após a compressão, acertar os endereços alvos no código comprimido (*patching*) ou usar uma tabela de tradução de endereços. Os alvos de saltos podem ser forçosamente alinhados à palavras ou uma tabela de *offsets* precisa ser usada. Os detalhes destas soluções serão vistos nos próximos capítulos.

1.1.2 Métricas de Avaliação de Compressão de Código

Para avaliar os métodos de compressão, uma medida comumente encontrada na literatura é a razão de compressão[4, 7]. A razão entre o tamanho do código comprimido e o tamanho do código original (Equação 1.1) é uma métrica que implica em pensar que quanto menor melhor.

$$razão\ de\ compressão = \frac{tamanho\ do\ código\ comprimido}{tamanho\ do\ código\ original} \tag{1.1}$$

Infelizmente, o uso desta equação tem sido feito em duas vertentes diferentes,

dificultando o trabalho de comparação entre os métodos de compressão. A primeira vertente simplesmente desconsidera, nos métodos de dicionários, o tamanho do dicionário em si, apontando o código comprimido apenas como a seqüência de *codewords*. Isto seria plausível se fossem utilizados dicionários estáticos mas, em geral, os métodos usam dicionários semi-adaptativos. Em nossos resultados, quando não especificado, a razão de compressão sempre inclui o tamanho do dicionário (Equação 1.2). Para implementações de métodos estatísticos sempre existe um repositório onde ficam guardadas *codewords* e/ou símbolos, então os comentários acima também se aplicam.

$$\textit{razão de compressão} = \frac{\begin{array}{c}\textit{tamanho do código comprimido} + \\ \textit{tamanho do dicionário}\end{array}}{\textit{tamanho do código original}} \tag{1.2}$$

1.2 Contribuição

Neste trabalho é proposto um método de compressão PDC, denominado PDC-ComPacket, baseado em um dicionário pequeno e incompleto, com 256 entradas e tendo como símbolo a instrução. É admitido apenas um símbolo por entrada do dicionário, facilitando a descompressão.

O desalinhamento de endereços foi resolvido usando *patching*. Os índices para o dicionário são aglutinados em palavras de 32 bits chamadas ComPackets, minorando o problema do desalinhamento do início (*offset*) das *codewords*. Este novo esquema de codificação de índices também admite saltos para instruções que não necessariamente sejam a primeira apontada dentro do ComPacket. O objetivo de compactar um conjunto de índices dentro de uma palavra com tamanho regular é evitar múltiplos acessos à cache para descompactar apenas uma instrução.

Um novo método de construção de dicionários é usado. Até então, os dicionários eram construídos com base em informações estáticas ou dinâmicas do uso de instruções. Em nossa abordagem mostramos que usar apenas uma destas estatísticas torna alguns resultados ineficientes, por isto propomos o uso simultâneo destas informações para construção de um dicionário denominado dicionário unificado.

Mostramos também que este método de construção de dicionários pode ser usado nos casos de dicionários completos e incompletos. Para tanto adaptamos o método CDM-IBC [8], que usa dicionário completo, para experimentar esta nova forma de construção.

Aplicamos, finalmente, este mesmo método para o caso de dicionários pequenos em nosso algoritmo de compressão, que usa dicionário incompleto.

Em ambos os casos existe sempre uma construção que implica em melhores compromissos entre compressão e desempenho e/ou entre compressão e consumo de energia. Em suma, as contribuições deste trabalho são:

- Um método de construção de dicionários que apresenta os melhores *trade-offs* na tríade compressão-desempenho-consumo de energia.
- Um método de compressão PDC que apresenta um dos melhores resultados de desempenho e consumo de energia para o SPARC, sem se distanciar dos melhores resultados em razão de compressão.

1.3 Organização

Este trabalho está organizado da seguinte forma: o Capítulo 2 apresenta os principais métodos de compressão da literatura especializada, apontando seus méritos e deméritos; no Capítulo 3 são analisados os métodos de construção de dicionários e é proposta a construção do dicionário unificado; o Capítulo 4 apresenta como esta abordagem pode ser usada em um método de compressão CDM com dicionários completos; o Capítulo 5 introduz o novo método de compressão PDC-ComPacket. No Capítulo 6 este método PDC é avaliado com um dicionário unificado; finalmente, apresentamos as conclusões e possíveis extensões deste trabalho no Capítulo 7.

Capítulo 2

Trabalhos Relacionados

Já no início dos anos 70, os projetistas do Borroughs B1700 [9] utilizaram-se de campos menores para codificar instruções mais freqüentes, enquanto as pouco freqüentes ficavam com campos maiores. Mais tarde, esta mesma abordagem foi utilizada no projeto do conjunto de instruções do VAX [10]. Estes projetos ainda não tratavam de compressão de código, mas de organização eficiente do conjunto de instruções em relação ao uso da memória.

Neste Capítulo abordaremos os trabalhos relacionados à compressão de código divididos em técnicas para compactação (compressão para uma forma executável ou interpretável), compressão CDM, compressão PDC, compressão por Conjunto de Instruções mistos, outros trabalhos e apresentamos algumas conclusões.

2.1 Compactação de Código

Também na década de 70, muitas oportunidades de compressão de código foram propostas no nível de compiladores. Técnicas de otimizações específicas para redução do tamanho dos códigos surgiram. Um bom compêndio destas técnicas pode ser encontrado no trabalho de Debray *et al* [11]. Neste trabalho o termo compactação de código significa compressão para uma forma executável ou interpretável. A exploração principal a ser feita para redução do código é encontrar seus trechos redundantes (ou clonados). Muitas abordagens fazem uso de uma estrutura de dados representativa em forma de árvores de sufixo [12, 13, 14] para suportar seus métodos de compactação. Em outras abordagens, os Grafos de Fluxo de Controle (CFG, *Control Flow Graph*) com anotações específicas e

árvores de dominadores e pós-dominadores formam a infraestrutura para as otimizações. Classicamente, quando a métrica principal para compilação é o tamanho do código objeto, são evitadas transformações como *procedure inlining* e *loop unrolling* [15] e são evidenciadas otimizações para eliminação de código redundante, inalcançável e morto, além de alguns tipos de *strength reduction*.

A eliminação de código redundante é, decerto, a mais desafiadora das tarefas e onde reside boa parte dos esforços em compactação. Não somente a clássica e necessária eliminação de sub-expressões comuns (CSE, *Common Subexpression Elimination*), mas também abstrações de procedimento[16] e *cross-jumping*[17] formam um escopo mais agressivo para reduzir o tamanho do código. Em [11] todas estas técnicas foram usadas para criar o *squeeze* que é uma ferramenta para reescrita de binários com otimizações específicas para compactação de código. Como resultado uma redução de 30% no tamanho do código sobre a forma otimizada clássica foi alcançada. Esta última, por sua vez, já tem uma redução de 20% sobre o código não otimizado. Isto implica numa redução de 44% no tamanho final do código se compararmos os resultados do *squeeze* com um código não otimizado (razão de compressão de 56%).

Em 1995 Fraser e Hanson [18] adaptaram um compilador C (*lcc*) para, dado o código fonte, o mesmo produzir um código compacto e um interpretador para este código. Uma razão de compressão de 50% (para arquitetura SPARC) foi alcançada quando compilando o próprio *lcc*, mas a execução do código pelo interpretador é 20 vezes mais lenta que a execução do código nativo.

Ernst *et al* [19] uniram esforços a este grupo e propuseram o BRISC (*Byte-coded RISC*). No BRISC uma máquina virtual chamada OmniVM é usada para experimentação do método. Esta máquina possui um conjunto de instruções aumentado com macro-instruções que podem representar um conjunto de outras instruções. Depois do código ser gerado, o programa compactador realiza uma especialização de operandos (substituição de uma instrução genérica por uma instrução que já possui pré-fixado um dos seus operandos) e uma combinação de *opcodes* (duas instruções adjacentes podem vir a se tornar uma única macro-instrução) para reduzir o tamanho do código. O código BRISC pode ser interpretado com uma penalidade no tempo de 12 vezes e o tamanho do código comprimido chega a ser 40% menor que o código original (razão de compressão de 60%).

Trabalhos seguintes deste grupo [20, 21, 22] seguiram a mesma linha de pensamento, alterando detalhes que levaram a melhorias marginais seja para a razão de compressão,

seja para diminuir a penalidade na interpretação dos códigos (desempenho).

Para processadores de sinais digitais, algumas otimizações específicas também foram exploradas. Liao *et al* [23] formularam um modelo para alocação de dados na memória de tal forma a minimizar a tarefa da computação de endereços para alimentar os registros de endereçamento indireto. Os DSP possuem, via de regra, endereçamento indireto (por registradores dedicados) com auto-incremento e auto-decremento. Ou seja, quanto mais regular for o acesso aos dados, favorecendo o uso de auto-incremento e auto-decremento, menor será o número de instruções necessárias para alocar um endereço particular nos registradores de endereçamento. Portanto, uma distribuição de variáveis na memória que maximize o uso de auto-incremento e auto-decremento, reduz o tamanho do código final.

Para um processador com um único registrador de endereçamento, o problema de encontrar uma distribuição para as variáveis na memória (SOA, *Single Offset Assignement*) para minimizar o tamanho do código é NP-completo. A heurística proposta é modelar o problema como uma cobertura de caminhos de peso máximo (MWPC, *Maximum Weight Path Covering*) e resolvê-lo com um algoritmo similar ao de árvore de espalhamento máxima de Kruskal[24]. A generalização do problema para k registradores de endereçamento é chamada de GOA (*General Offset Assignement*). Os resultados em termos de compressão foram uma redução de 3% a 9% no tamanho do código original para SOA e entre 5% e 20% para GOA (razão de compressão entre 97% e 80%).

2.2 Compressão CDM

O CCRP [3] foi de fato o primeiro trabalho de compressão de código a usar uma arquitetura RISC (MIPS R2000) como processador e um hardware descompressor. A idéia dos autores foi utilizar código de Huffman [5] para comprimir o programa original. Um histograma de ocorrências de bytes no código serviu como base para geração dos signos a serem utilizados na codificação. Então, cada bloco de 32 bytes (tamanho de uma linha de cache nos experimentos realizados) foi comprimido utilizando estes signos. Quando ocorre uma falha na cache, um descompressor busca o bloco na memória principal (ou no próximo nível da hierarquia), descomprime-o e entrega a linha inteira à cache. Desta forma, as instruções na cache ficam descomprimidas e cada acerto (*hit*) não implica em tempo extra para descompressão. Se a taxa de acerto for alta, a penalidade devida ao tempo gasto na descompressão, que surge quando ocorre uma falha, não degrada consideravelmente o desempenho.

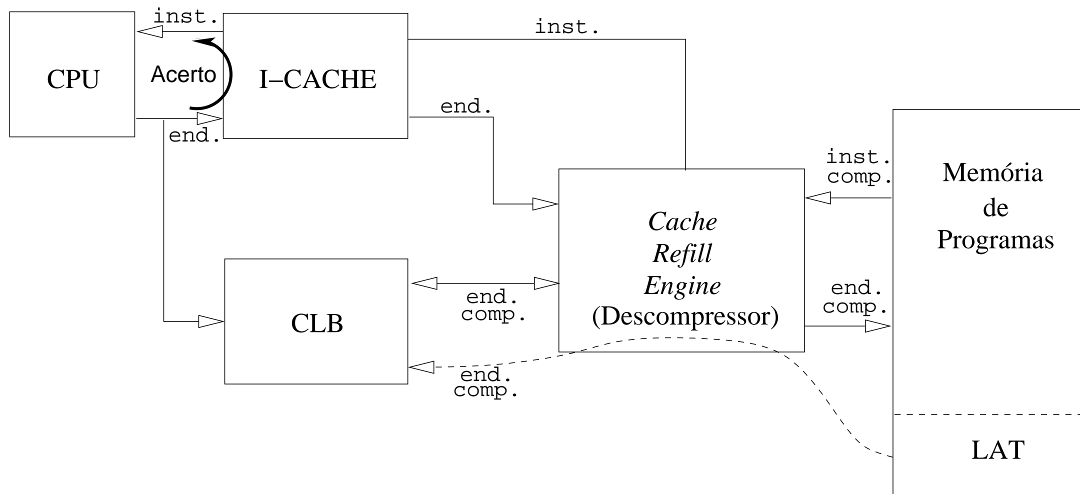


Figura 2.1: Diagrama do CCRP

No caso do CCRP, o problema de desalinhamento de endereços foi resolvido com uma tabela de tradução de endereços, ATT (*Address Translation Table*), chamada LAT (*Line Address Table*). O descompressor proposto trabalha com linhas de cache, portanto, somente os endereços das linhas são necessários nesta tabela. Um porém neste esquema é que o endereço do início de uma linha precisa ser reconhecido pelo sistema de memória. Por exemplo, se o endereçamento de um sistema de memória é alinhado a 32 bits, então cada endereço de início de uma linha comprimida precisa estar alinhado a 32 bits. Quando isto não ocorre é preciso usar bits de *padding*[1] para completar o alinhamento. Cada acesso à LAT produz um tempo extra no acesso à informação. Por isto, uma pequena cache de endereços foi utilizada para conversão - CLB (*Cache Line Address Lookaside Buffer*) simultânea ao acesso à cache.

A Figura 2.1 mostra os diversos componentes da proposta do CCRP. Veja que para um acerto na cache o descompressor (chamado de *Cache Refil Engine*) não é acionado. Por outro lado, uma falha na cache faz com que o endereço buscado seja repassado para a máquina de descompressão. Esta por sua vez, consulta a CLB. Se o endereço comprimido correspondente está disponível, o mesmo é usado para acionar a memória de programas, senão, é feita uma consulta a LAT (na memória prinicpal) e o endereço comprimido é repassado para a CLB e para o descompressor. A linha de cache comprimida é acessada na memória e repassada ao descompressor que a descomprime e entrega à cache.

Uma variação do Código de Huffman também foi averiguada porque algumas

[1]Bits de valor zero, usados para preencher espaços vazios de uma codificação

representações muito longas poderiam surgir na codificação de bytes pouco freqüentes. Um código de Huffman limitado a no máximo 16 bits foi utilizado (denominado Código de Huffman Limitado). Isto implica em diminuição na densidade de código que se poderia alcançar mas, como estes casos são raros, na prática se evidenciou apenas pequenas alterações nos tamanhos dos códigos comprimidos.

Um outro resultado interessante deste trabalho diz respeito ao impacto da mudança de programa no CCRP. Numa abordagem semi-adaptativa cada programa distinto possui seu próprio histograma de ocorrência de instruções e uma tabela de símbolos distinta. Como alternativas, ou se anexa ao código comprimido a tabela particular ou se encontra uma tabela genérica (modelagem estática). Para os experimentos apresentados criou-se uma tabela genérica, baseada na freqüência de ocorrência de todos os dez programas utilizados para medir o desempenho. Os resultados foram quase tão efetivos quanto com o Código de Huffman Limitado semi-adaptativo, dada a regularidade com que o código costuma ser gerado. Finalmente, em todos os experimentos, as linhas de cache que passaram a ter uma representação maior que a original correspondente foram seletivamente evitadas.

Uma razão de compressão de 73% foi alcançada, em média, no CCRP, mas os números não levaram em consideração o tamanho da LAT, da CLB nem do descompressor. Destacamos que o tamanho do símbolo utilizado é de um byte. Símbolos maiores implicam em máquinas de descompressão Huffman mais lentas. Kozuch e Wolfe [25] ampliaram os experimentos com o CCRP para 6 arquiteturas distintas (VAX 11/750; MIPS R4000; 68020; SPARC; RS6000; e MPC603) e apresentaram resultados interessantes como o impacto da expressividade do conjunto de instruções no tamanho do código (naturalmente está embutida a eficácia na geração de código do compilador). O mesmo código fonte produz um código de máquina 2,6 vezes maior num MIPS que num VAX (que produziu os menores códigos). Para as outras arquiteturas, 68020, SPARC, RS6000 e MPC603, os resultados foram códigos 2,8×, 3,2×, 4,3× e 4,8× maiores que o código do VAX, respectivamente. Ainda, uma medida da entropia [26] de cada programa foi calculada, usando um byte para símbolo, mostrando o limite teórico de compressão para as diversas arquiteturas. Os valores encontrados nos experimentos ficaram próximos deste limite, sugerindo que o método de compressão é bastante eficaz. De uma forma geral, a eficácia da compressão em relação à entropia fica, em média, em 95%, ou seja, quase no limite teórico para este tamanho de símbolo (8 bits).

Em 1997 Benes, Wolfe e Nowick [27] implementaram uma versão do CCRP original (para MIPS). O objetivo era investigar quão dispendioso seria um descompressor Huffman.

Outras implementações já haviam sido propostas para aplicações em vídeo digital. Os melhores resultados apontam uma taxa de 32bits/39ns em um processo MOSIS CMOSX 0,8mm com 3 camadas de metal. A área total do descompressor ocupa 0,75mm^2, o equivalente a aproximadamente 3Kbytes de ROM. Mais tarde, em 1998, os mesmos autores conseguiram melhorar o desempenho do descompressor para 32bits/25ns [28]. Sumarizando, 200ns adicionais seriam necessários para descomprimir uma linha de cache completa, além do tempo de busca da informação na ROM. Esta penalidade pode ser alta demais se a quantidade de falhas na cache também o for. Uma observação importante é que a área do descompressor não inclui a CLB nem a LAT.

Kirovsky *et al* [29] propuseram uma abordagem onde um procedimento inteiro é comprimido e uma cache especial, chamada *pcache*, é utilizada como repositório para descompressão. Quando um procedimento não se encontra descomprimido na *pcache*, ele passa por um descompressor e é armazenado nela. Este modelo traz um conjunto de implicações, a saber: mais de um procedimento pode estar armazenado na *pcache* (desde que a soma de seus tamanhos não ultrapasse a capacidade do repositório), isto significa que, em algumas situações, ao alocar um procedimento é necessário retirar outro(s). Ainda, é possível que haja espaço suficiente para alocação de um procedimento, mas devido ao algoritmo de substituição de código na *pcache*, este espaço não seja contíguo. Um compactador de *pcache* é utilizado fazendo com que os procedimentos não fiquem espaçados no repositório. Se o espaço livre total na *pcache* é menor que o necessário para alocar o procedimento chamado, então é necessário substituir algum(ns) procedimento(s). É preciso garantir também que a *pcache* tenha no mínimo o tamanho do maior procedimento.

Quanto ao endereçamento dos procedimentos, um serviço de diretório é utilizado ligando o identificador do procedimento ao endereço comprimido. O algoritmo utilizado para compactar os procedimentos foi uma versão [30] do clássico Ziv-Lempel [31, 32] que atinge 60% de compressão (razão de compressão de 40%) usando bytes como símbolos para arquitetura SPARC. Com uma *pcache* de 64kbytes, o impacto negativo no desempenho chega a 11% ou, se inclusos o gcc e o go no conjunto de *benchmarks* usado, 166%. Para uma *pcache* de 32kbytes, estes números saltam para 36% e 600% respectivamente. A razão de compressão não inclui o tamanho da *pcache* nem o hardware associado para as políticas de substituição de código.

A *International Business Machines Corporation* (IBM) lançou, em 1998, o CodePack [33, 34] que utiliza uma codificação baseada em ocorrências e implementada

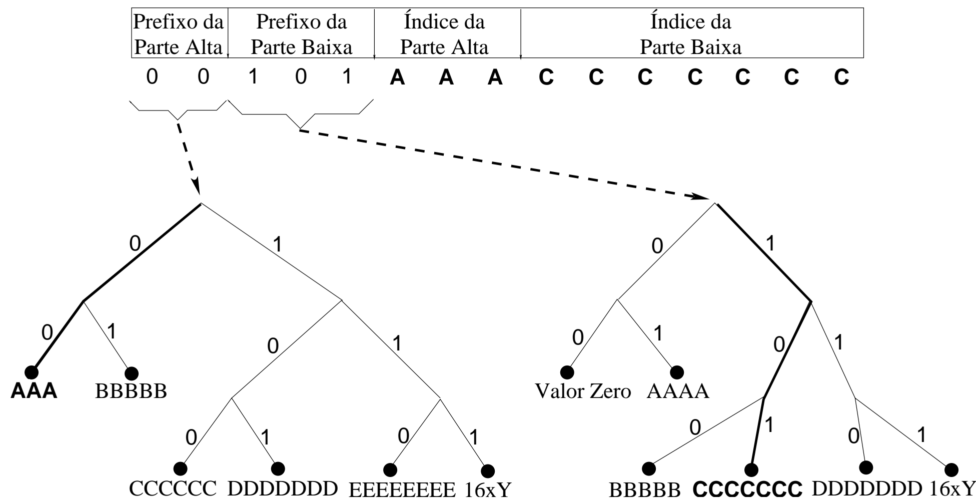


Figura 2.2: Árvores de prefixos usadas no CodePack da IBM

com dicionários para integrar a família de microprocessadores embarcados como o PowerPC 401 e 405. Cada instrução de 32 bits é dividida em duas partes de 16 bits (meia palavra) e um histograma de cada uma delas é montado. Com base no número de ocorrências de cada parte, a instrução é codificada. O esquema de codificação requer um prefixo para cada conjunto de 16 bits e a partir daí os bits seguintes são conhecidos. O prefixo não tem tamanho fixo, portanto a decodificação se torna mais complexa. As árvores na Figura 2.2 mostram os prefixos usados na codificação de cada uma das partes da instrução. Por exemplo, uma instrução que tenha sua codificação da parte alta como 00111_2 significa ter dois bits de prefixo (00_2) e três de índice (AAA=111_2) apontando para uma tabela de 8 posições contendo as mais freqüentes ocorrências destes 16 bits. Ao encontrar uma codificação 0100000_2 significa que estamos apontando para meia palavra de índice 00000_2 numa outra tabela que guarda as próximas 32 meia palavras mais freqüentes. O 16xY significa uma meia palavra original.

Neste esquema, a instrução de maior ocorrência na parte baixa e na alta pode ser codificada com 7 bits. A instrução comprimida é montada com a seguinte seqüência de bits: prefixo da parte alta, prefixo da parte baixa, índice da parte alta, índice da parte baixa. No pior caso, uma instrução pode ser codificada com 38 bits: 6 de prefixos e os 32 originais. É interessante notar que a distribuição de ocorrências da parte alta e da parte baixa das instruções é bem diferente, isto porque na parte alta ficam os *opcodes* e registradores e na parte baixa ficam, normalmente, imediatos. Assim, dois dicionários distintos são utilizados.

Para encontrar uma instrução comprimida na memória a solução foi usar tabelas, desta vez formando uma hierarquia, mas com o mesmo objetivo de tradução de endereços. A descompressão ocorre sempre que houver uma falha na cache, como no CCRP. A razão de compressão reportada pela IBM chega a 60%, incluindo a tabela de tradução de endereços, enquanto o desempenho varia 10% para mais ou menos. Um estudo interessante deste esquema foi feito por Lefurgy *et al* [35], variando a quantidade de falhas na cache, o processador (*single issue*, 4-*issue, out-of-order* e 8-*issue, out-of-order*) e o uso de diversas otimizações. A conclusão evidenciada é que muitas vezes a compressão de código propicia uma melhoria no desempenho da máquina, além da já esperada redução no tamanho do código.

Lekatsas e Wolf [36] usaram o mesmo modelo de hardware de descompressão do CCRP para propor dois algoritmos: *Semi-Adaptive Markov Compression* - SAMC e *Semi-Adaptive Dictionary Compression* - SADC. O SAMC particiona as instruções em conjuntos de bits, não necessariamente adjacentes, que podem ter tamanhos variados, mas têm posições fixas em todas as instruções. Estes conjuntos alinhados formam seqüências (*streams*) que são comprimidas independentemente usando um modelo de Markov de primeira ordem e codificação aritmética binária. Para o SPECint95 os autores reportam uma razão de compressão de 57% para código MIPS e 75% para o Pentium Pro.

O SADC usa um dicionário semi-adaptativo para comprimir *opcodes*, combinações de *opcodes* com registradores e combinações de *opcodes* com imediatos. A idéia das seqüências também se aplica aqui, mas agora os campos têm uma semântica definida, ou seja, uma seqüência de *opcodes*, uma seqüência de registradores, uma seqüência de imediatos, etc. Portanto, diferentemente do SAMC, o SADC é dependente do conjunto de instruções. A razão de compressão alcançada para o MIPS foi de 52%, mas em ambos os métodos o tamanho do descompressor não foi mencionado nem considerado no cálculo.

Em 1999, Lekatsas e Wolf [37] melhoraram a forma de decodificação dos bits para o SAMC e em [38] os experimentos foram estendidos para o ARM/Thumb. A velocidade de decodificação, que é o principal gargalo em uma codificação aritmética, passou a ser comparável a de um decodificador Huffman.

Em [39], Araujo *et al* averiguam três técnicas de compressão baseadas em dicionários completos. A idéia é dividir em classes o conjunto de instruções usado em um código. Para cada classe um tamanho de *codeword* e um prefixo são assinalados. A separação em classes segue a idéia de distribuir o número de ocorrências dos símbolos igualmente nas classes. Os símbolos são inicialmente classificados por ocorrências e então o montante

acumulado destas ocorrências é distribuído igualitariamente nas classes, o que equivale a criar uma pequena classe para os símbolos que mais ocorrem. Os prefixos possuem tamanhos fixos independentemente da classe a ele atribuída. O algoritmo de compressão é responsável por escolher uma quantidade de classes e tamanho das *codewords* de tal forma a maximizar a compressão. O mapeamento de endereços é feito com uma tabela de tradução de endereços (ATT) a as *codewords* e seus prefixos são justapostos no código objeto comprimido.

As três técnicas usam este mesmo paradigma para símbolos distintos em um código. O primeiro símbolo é uma árvore de expressão como definido em [15] e forma o método TBC (*Tree Based Compression*)[40]. A razão de compressão final, incluindo os dicionários de árvores e uma estimativa para a máquina de descompressão, fica em 60,7%. O segundo símbolo utilizado foi gerado a partir de um desmembramento de uma árvore de expressão em padrões. Isto é, em cada árvore as instruções foram separadas em operações e operandos e foram gerados um padrão de árvores de operações e um padrão de operandos. Este processo de separação foi denominado fatorização de operandos e o método de compressão associado é o PBC (*Pattern Based Compression*)[41]. Esta abordagem acaba levando a um mecanismo de descompressão mais complexo e a razão de compressão final fica em 61,3%. O terceiro e último símbolo utilizado foi a própria instrução, levando a uma simplificação do descompressor. A razão de compressão do método, denominado IBC (*Instruction Based Compression*), ficou em 53,6% para a arquitetura MIPS. Os detalhes destes métodos estão em [8, 40, 41] e o IBC será reapresentado no Capítulo 4.

O TBC também foi testado em uma arquitetura DSP. Especificamente, para o TMS320C25 a razão de compressão final ficou em 75% [42]. Algumas derivações do PBC foram propostas para melhorar sua razão de compressão que em alguns casos chegou a 46%, mas com outro conjunto de *benchmarks* [43, 44]. Uma variação do PBC, com três métodos de codificação (Huffman, VLC e um dicionário) foi utilizada em [45] para o TMS320C25 e a razão de compressão ficou em 67%.

Em 1999, Lefurgy e Mudge [46] propuseram um descompressor em software que atua como no CCRP. Foi usado um dicionário, indexado por *codewords* de 16 bits, contendo uma instrução por entrada. A idéia é que quando ocorrer uma falha na cache o descompressor seja invocado e preencha a linha correspondente com código descomprimido. Para realizar esta tarefa, dois requisitos são importantes: uma falha na cache deve gerar uma exceção cuja rotina de tratamento é o próprio código (software) descompressor; e deve existir no processador uma instrução que seja capaz de escrever

diretamente na cache. O conjunto de instruções do MIPS IV foi usado. A rotina de tratamento de exceção executa 74 instruções para descomprimir 32 bytes e alocá-los na cache. A razão de compressão ficou em 65,3% e o desempenho foi degradado entre 2% e 54% usando uma cache de 64KBytes (e taxa de falha menor que 1%). Estes resultados são extremamente sensíveis a falhas na cache. Uma taxa de falhas em torno de 5% produz uma queda de desempenho enorme, tornando a execução 3 vezes mais lenta.

Um trabalho interessante, derivado deste último, foi publicado em 2000 [47]. A A penalidade na execução do código foi aliviada usando duas técnicas: compressão seletiva por execução e por falha na cache. Na compressão seletiva por execução foram escolhidos para compressão apenas os procedimentos menos executados, isto é, aqueles com o menor número total de instruções executadas. Na compressão por falha de cache, foram escolhidos para não serem comprimidos os procedimentos que provocaram maior quantidade de falhas (no código original). A razão de compressão piorou para 72,8%. O desempenho "melhorou" para algo em torno de 2,8 vezes o tempo de execução em relação ao código original.

2.3 Compressão PDC

Alguns métodos de compressão foram avaliados em sistemas que não contêm caches. Os requisitos destes sistemas se assemelham aos de uma arquitetura PDC no que se refere à velocidade de descompressão e de tradução de endereços. Desta forma, esta seção inclui os trabalhos baseados em sistemas sem cache.

2.3.1 Sistemas sem Cache

Benini *et al* [48] propuseram um método de compressão para diminuir o consumo de energia. A idéia é construir uma tabela com as 255 instruções mais executadas (baseada em *profiling* dinâmico) e substituir no código original instruções por apontadores para esta tabela (no caso cada apontador tem um tamanho fixo de 8 bits). Um dos 256 apontadores endereçáveis é reservado para prefixar instruções não comprimidas. Para os *benchmarks* utilizados, as 255 instruções comprimidas perfazem 83% do número de instruções distintas executadas. O objetivo de diminuir a atividade do barramento, que implica na redução de energia, é alcançado porque um número maior de instruções comprimidas é executado em detrimento das instruções não comprimidas. A atividade do barramento é medida em

duas parcelas: total de acessos à memória de instruções e total de transições de bits. Para o DLX[49], o total de acessos ficou em 52% e o número de transições ficou em 61% em relação ao original.

Em 2001, Benini [50] propôs alterar o método de compressão acima para não gerar os prefixos nas instruções não comprimidas. Para tanto, a unidade de compressão passou a ser um bloco básico e apenas os limites de endereços dos blocos básicos não comprimidos passaram a ser armazenados. Isto implica que em cada *fetch*, o endereço é comparado com esta série de *ranges* e verificado se pertence a um bloco descomprimido. Para diminuir o tamanho desta tabela de endereços um algoritmo foi utilizado para levar em consideração também os blocos básicos adjacentes (ou seqüentes). Os resultados agora incluem uma medida da razão de compressão que ficou em 90% enquanto uma redução de energia de 46% a 52% foi alcançada (incluindo acessos à memória, descompressor e barramentos) .

Liao *et al* propuseram em 1995 [51] dois métodos de compressão. Uma técnica baseada no EPM (*External Pointer Macro*) [52] foi utilizada para encontrar seqüências de instruções comuns (chamadas mini sub-rotina) e inseri-las em um dicionário. Dois métodos diferentes foram propostos: um puramente em software e o outro em cooperação de hardware e software. No primeiro, uma instrução de chamada (CALL) é utilizada para chamar as mini sub-rotinas. Qualquer que seja a seqüência do dicionário invocada, ela sempre termina com uma instrução de retorno (RET). Adicionalmente, se uma seqüência é sufixo de uma entrada existente no dicionário, ela também pode ser substituída por uma instrução de chamada. Nesta abordagem um custo de duas instruções é inserido em cada mini sub-rotina[2]. Não se pode deixar de mencionar a semelhança com o trabalho de Fraser *et al* [53], no que se refere ao plano comum de encontrar trechos clonados do código e convertê-los em subrotinas/procedimentos.

No segundo método, uma nova instrução é inserida na arquitetura do conjunto de instruções: CALD (*Call Dictionary*). Ela tem como parâmetros o ponto de entrada no dicionário e a quantidade de instruções a serem executadas. Neste caso o dicionário é generalizado, sendo uma seqüência de instruções única onde qualquer subseqüência pode ser acessada. Os tamanhos dos códigos dos *benchmarks* utilizados foram diminuídos para 88% do código original (incluindo o tamanho do dicionário) utilizando-se o primeiro método e para 84% com o segundo.

Mais tarde [54], Liao formulou algumas considerações sobre o desempenho da arquitetura (TMS320C25) em que foram realizados os experimentos. Primeiramente

[2]Este é um método de compactação, apresentado aqui apenas para não dissocia-lo do seguinte

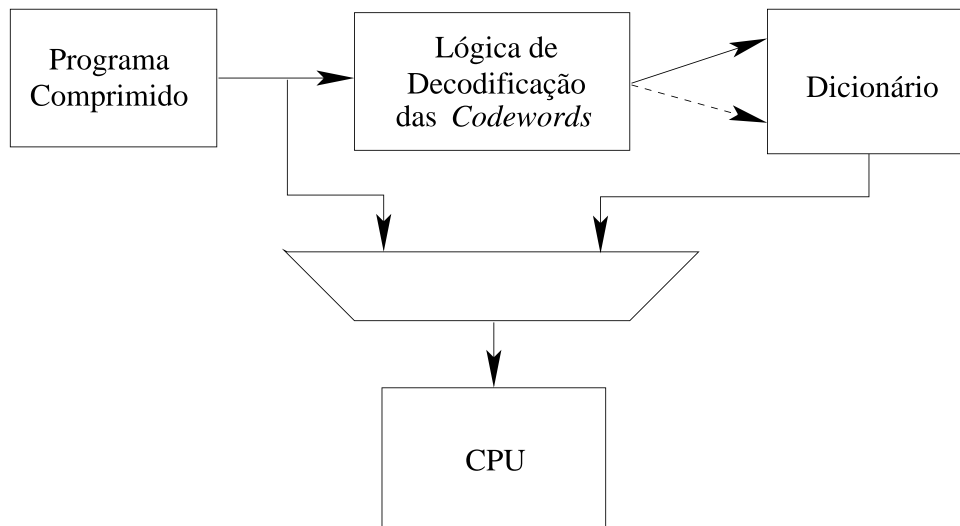


Figura 2.3: Abordagem de Lefurgy para execução de código comprimido

apenas o 'código frio' (menos executado) foi comprimido, com vistas a diminuir o impacto negativo da descompressão no desempenho da máquina. Embora uma degradação de 15% a 17% tenha sido observada na execução do código frio, o valor final da perda de desempenho total ficou entre 1% e 2%, enquanto a razão de compressão piorou de 2% a 3% (o código frio foi encontrado simplesmente fazendo *profiling* da execução do programa e encontrando as instruções menos freqüentes, limitadas a 90% do total). Neste trabalho, a abordagem da construção dos dicionários foi alterada, levando a razão de compressão para 85% e 82% respectivamente para os métodos enumerados acima. Vale ressaltar que o desempenho desta compressão ficou pior que os estudos anteriores, mas abordou o caso da compressão em máquinas tipicamente CISC como o TMS320C25[55].

Os métodos de Lefurgy [56, 57] também são baseados em dicionários. Seqüências de instruções comuns são encontradas e extraídas do código fonte. Estas seqüências são então colocadas em um dicionário e no código-fonte são substituídas por *codewords*. Com um dicionário incompleto, nem todas as seqüências são aproveitadas. O novo código fonte é composto então de um misto de instruções (originais) e *codewords*. Em tempo de execução, as *codewords* fornecem um índice para o dicionário onde estão as instruções originais correspondentes. Os padrões encontrados a serem alocados no dicionário têm tamanhos variados podendo, no máximo, ter uma seqüência inteira de um bloco básico. Os alvos dos desvios são alinhados a *nibbles* (4 bits) e, portanto, são necessárias alterações no endereçamento do processador (a unidade de controle tem de ser capaz de endereçar

nibbles). Instruções de desvio contendo *offsets* não são 'comprimidas' , facilitando as alterações *in loco* para os novos endereços (*patching*). Desvios indiretos (o alvo está em um registrador) podem ser comprimidos, mas é necessário alterar as tabelas de saltos (*Jump Tables*). A Figura 2.3 sumariza os componentes da proposta de Lefurgy. Cada instrução é observada pela Lógica de Decodificação de *codewords* e, se for uma instrução comprimida, o dicionário é acessado e as instruções correspondentes são entregues ao processador.

Três conjuntos de instruções foram usados nos experimentos: do PowerPC, do ARM e do i386. O primeiro método proposto usa um tamanho fixo de *codeword* (16 bits) e foi implementado apenas para o PowerPC. Os resultados para o SPECint95 apontam para uma razão de compressão de aproximadamente 68%, incluindo o tamanho do dicionário. Neste experimento o tamanho das entradas no dicionário estava limitado a quatro instruções. O tamanho máximo do dicionário também estava limitado a 128kbytes (devido a quantidade de bits disponível para endereçamento usando instruções ilegais do PowerPC). Variações destas condições limítrofes também foram feitas e daí derivadas duas importantes conclusões: para melhorar a razão de compressão é mais importante aumentar o número de entradas no dicionário que o tamanho delas; e mais de 6% da razão de compressão advém do uso de *codewords* que representam uma única instrução.

O segundo método proposto por Lefurgy *et al* foi uma variação do primeiro, mas com as *codewords* possuindo tamanhos variados (pré-fixados em 8, 12 e 16 bits). Limitando à 16 bytes o tamanho máximo de cada entrada do dicionário, as razões de compressão foram de 61%, 66% e 74% para o PowerPC, ARM e i386 respectivamente. Nenhuma informação sobre degradação do desempenho ou gasto de energia foi relatada.

Em 1998, Lefurgy e Mudge[57] propuseram aplicar seus métodos para um DSP. A escolha foi o SHARC[58] (com instruções de 48 bits). Limitaram, entretanto, a busca por instruções repetitivas a uma unidade, ou seja, cada instrução no código é representada por uma *codeword* de 16 bits. Para o SHARC foi necessário aumentar o *pipeline* em um estágio (incluir um estágio de pré-busca). A razão de compressão ficou em 48,3% e 54,6% para os códigos gerados sem e com otimizações (–O1) respectivamente. Interessante notar que os códigos não otimizados quando comprimidos ficaram com seus tamanhos menores que os códigos otimizados com –O1 e também comprimidos.

Lekatsas *et al* em [59], ainda usando o SAMC, investigaram o efeito da compressão de código no consumo de energia de um sistema. Quando se usa um codificador aritmético de precisão reduzida, que é o caso do método SAMC, existe uma certa liberdade na

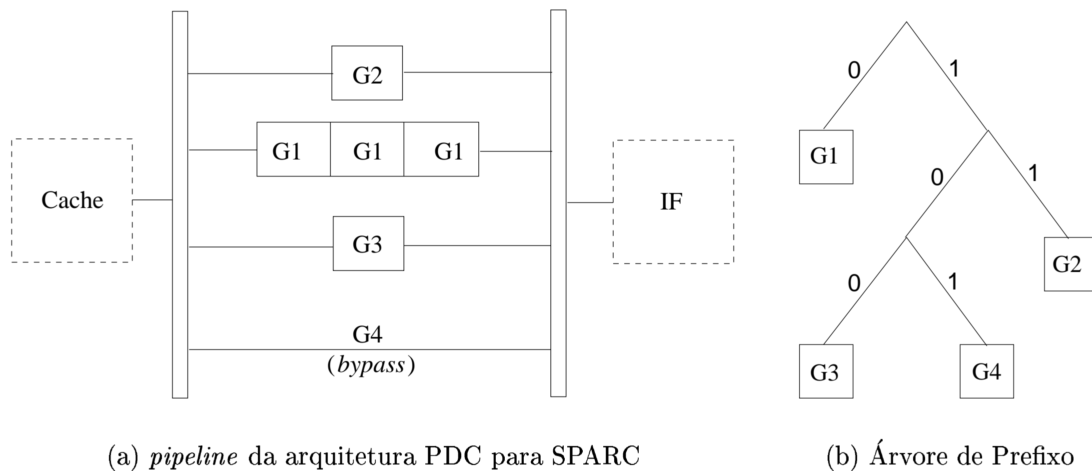
escolha dos códigos de cada símbolo e, portanto, é possível cuidar para que a distância de Hamming[3] seja a menor possível no código comprimido. Como medida de energia foi usada a quantidade de transições de bits nos barramentos entre a memória e o processador. O número de transições ficou 26% menor num sistema com código comprimido. Uma observação importante neste trabalho é que nem sempre melhores compressões levam a menores quantidades de transições. Isto pode ser verificado variando-se parâmetros de precisão do codificador aritmético. Outro detalhe importante é que o número de transições foi minorado apenas para seqüências de instruções estaticamente definidas, ou seja, nenhuma informação de *profiling* dinâmico foi utilizada.

2.3.2 Sistemas com Cache

Em [60] Lekatsas *et al* apresentaram duas possibilidades de posicionamento do descompressor e introduziram os termos pré-cache e pós-cache (os equivalentes a CDM e PDC respectivamente). Neste trabalho, as instruções do SPARC foram separadas em categorias: G1: instruções com imediatos; G2: *branches*; G3: instruções sem imediatos que não sejam *branches*; e G4: instruções não comprimidas. Para que o decodificador possa diferenciá-las um prefixo foi acrescentado em cada instrução (G1: 0_2; G2: 11_2; G3: 100_2; e G4: 101_2). As instruções com imediatos foram comprimidas usando o SAMC. Instruções sem imediatos, que não sejam *branches*, foram comprimidas usando um índice para uma tabela de 256 posições, chamada de dicionário rápido. Os branches são comprimidos usando um campo na instrução (já codificada) que indica a quantidade de bits usada no campo de deslocamento do salto. 54% das instruções para o conjunto de *benchmarks* utilizado pertencem ao G1. 26% ao G2, 20% ao G3 e 0,6% ao G4. A razão de compressão final ficou em torno de 65%, usando este método misto, mas sem considerar o tamanho do descompressor.

Em um modelo pós-cache as instruções na cache estão armazenadas comprimidas, então uma maior taxa de acerto é esperada. Por outro lado, a penalidade da descompressão fica no caminho crítico da execução. O tempo de execução dos experimentos foi medido em ciclos de *clock*, mas não está claro se o tempo necessário para descompressão foi considerado, especialmente nas instruções do tipo G1, que conhecidamente têm uma descompressão lenta e estão em maioria no histograma de ocorrências. Para cada *benchmark* foram escolhidos três tamanhos de caches próximos

[3]número de transições de bits entre duas palavra consecutivas



(a) *pipeline* da arquitetura PDC para SPARC (b) Árvore de Prefixo

Figura 2.4: Abordagem de Lekatsas para execução de código comprimido

da área de melhor compromisso entre tamanho e desempenho (para o código não comprimido). O desempenho foi melhorado em 25% em média (redução de 25% no número de ciclos de *clock*).

As medidas de energia, também apresentadas, são divididas no gasto na CPU, caches, memória principal e barramentos. Na CPU, a melhoria devido à compressão deve-se ao fato que o número de ciclos de espera causados por falhas na cache é diminuído, uma vez que sua taxa de acerto é aumentada. Na cache e na memória principal os gastos com energia são reduzidos porque a quantidade de acessos é menor. Nos barramentos o número total de transições é diminuído (porque o número de acessos à cache e à memória principal são diminuídos) e, portanto, a energia gasta é menor também. Por todos estes fatores a redução de energia total ficou em média em 28%. Nenhuma informação sobre penalidade por erro na cache está disponível. Este é um fator fundamental porque determina o impacto da compressão no desempenho do sistema.

Em 2001 Lekatsas *et al.* [61] apresentaram um modelo do descompressor para uso no trabalho anterior. Um conjunto de *pipelines*, um para cada tipo de instrução (G1; G2; G3; e G4), usados em paralelo, formam o descompressor. As instruções do tipo G1, críticas para descompressão, usam 3 estágios em seu *pipeline* particular, enquanto instruções G2 e G3 usam apenas 1 estágio e G4 usam um *bypass*. A Figura 2.4 apresenta o conjunto de *pipelines* e uma árvore de prefixos usadas na identificação de cada categoria de instrução. Naturalmente é necessário manter um controle do posicionamento de cada *codeword* dentro de um conjunto de 32 bits que chega do sistema de memórias. Além

disto, como cada *codeword* é prefixada e tem tamanhos diferentes, num conjunto de 32 bits podemos ter, por exemplo, três instruções do tipo G1 e mais uma do tipo G2 em qualquer seqüência e precisam ser inicialmente descobertas para o envio para seus respectivos *pipelines*. Este posicionamento e alocação podem levar um tempo não desprezível, mas não foram considerados no trabalho. Outro ponto é a quantidade de estágios para instruções G1. Mesmo com as melhorias providas no SAMC para descompressão simultânea de múltiplos bits (6 a 8 por ciclo de *clock*) seria necessário garantir que, no pior caso, apenas 3 estágios seriam suficientes para descomprimir 32 bits o que não é mencionado. Isto afeta diretamente os resultados porque instruções G1 são as que mais ocorrem no código.

Um outro problema é a descompressão fora de ordem: considerando que uma instrução G1 leva 3 ciclos para ser descomprimida, uma instrução do tipo G2 pode ficar pronta antes de G1. Uma unidade de controle, capaz de paralisar a descompressão da instrução seguinte, se faz necessária para não haver um despacho fora de ordem para unidade de *fetch* do processador. Ainda, a penalidade causada por um desvio no fluxo de controle (seja por *branches*, *calls* ou interrupções) agora passa a ser maior. Mesmo assim, a razão de compressão do código e o paralelismo do descompressor compensam os efeitos contrários. Neste artigo fica claro que a razão de compressão não inclui o tamanho do descompressor, que neste caso, com esta sofisticação, pode ser considerável. Ou seja, a garantia de menor área de silício devido à compressão não pode ser utilizada.

Em [62, 63] um descompressor com apenas um estágio de *pipeline* foi utilizado. O processador alvo, o Xtensa-1040[64], possui um barramento de 32 bits, mas suas instruções são codificadas em 24 e/ou 16 bits. O método de compressão usado foi baseado em instruções com um dicionário de 256 posições. As instruções de 24 bits de maior ocorrência no código foram comprimidas em *codewords* de 16 bits e 8 bits. Os quatro bits menos significativos, em qualquer que seja a codificação (original ou comprimida), indicam o tamanho da palavra. Esta é uma característica que já estava prevista para o processador sem compressão e as *codewords* utilizadas mantiveram esta premissa. O dicionário foi duplicado para permitir a descompactação de duas *codewords* ao mesmo tempo. As instruções de salto são codificadas separadamente, para ser possível recalcular os alvos e estes últimos são alinhados em palavras, trocando assim razão de compressão por simplicidade do descompressor.

Uma outra restrição que se aplica aqui é o tempo de descompressão que precisa ser menor ou igual a um ciclo de *clock*. Como as palavras têm tamanhos fixos (24 bits, 16 bits e 8 bits) um decodificador olha todas as possibilidades de junção em 32 bits

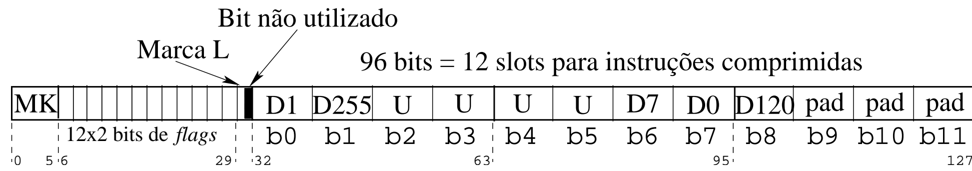


Figura 2.5: Esquema da linha de cache comprimida (Benini)

destas palavras e escolhe o dicionário a ser usado ou o *bypass*. Novamente, a melhoria de desempenho foi mais intensa quando usadas caches de menor tamanho que o ponto de saturação[4]. Em média, o número de ciclos de *clock* foi reduzido em 25% enquanto a compressão chegou a diminuir o tamanho do código original em 35%, sem considerar o tamanho do descompressor. Um detalhe importante tecnologicamente neste artigo é a implementação do sistema inteiro em uma FPGA. Nenhuma informação de *profiling* dinâmico foi utilizada para compressão nestes trabalhos de Lekatsas.

Benini, em [65], propôs um outro método de compressão com o objetivo de reduzir a energia gasta no sistema, mas agora considerando a existência de uma cache. Assim, os termos *Decompression on Fetch* (DF) e *Decompression on Cache Refil* (DCR) foram cunhados para indicar o local do descompressor. De fato, o momento da ação do descompressor leva ao nome, pois o descompressor está integrado à cache. O método de compressão, especializado para DF, mais uma vez usa uma tabela de 256 posições para guardar as instruções mais freqüentemente executadas. As instruções são comprimidas em grupos do tamanho de uma linha de cache (4x32 bits nos experimentos).

Em cada linha da cache ou existem 4 instruções originais ou existe um conjunto de instruções comprimidas, possivelmente intercaladas com outras não comprimidas, prefixado por uma palavra de 32 bits. Esta palavra tem um posicionamento fixo e serve para distinguir uma linha comprimida de uma linha 'regular'.

A Figura 2.5 mostra a estrutura de uma linha comprimida. Um conjunto de 6 bits identifica o tipo da linha. Um opcode inválido é usado para sinalizar uma linha comprimida. Segue-se um conjunto de 12 pares de bits que sinalizam se os bytes correspondentes (b0 a b11) contêm instruções comprimidas ou não. Isto possibilita intercalar em uma linha instruções adjacentes não pertencentes ao dicionário, desde que no total, uma linha comprimida possua mais instruções que uma linha original. Na ilustração vemos as instruções pertencentes ao dicionário (D1, D255, D7, D0, D120) intercaladas com

[4]Ponto onde um aumento do tamanho da cache não implica em melhoria significativa da sua taxa de acerto

uma instrução original (UUUU). A marca L indica se a última instrução na linha está comprimida.

O algoritmo de compressão percorre o código para encontrar um conjunto de instruções adjacentes, contendo algumas pertencentes ao dicionário de tal forma que uma linha da cache possa ser representada de forma comprimida. Os alvos dos saltos são alinhados a palavras, assim podem existir bits de *padding* entre as palavras comprimidas e alvos. Também não é permitido que uma palavra ultrapasse o limite de uma linha, sendo freqüentemente necessário *padding* no seu final, como visto na ilustração da Figura 2.5 (posições b9 a b11).

O descompressor, integrado à cache, entra em ação a cada acesso, identificando se a linha está comprimida e entregando a instrução apropriada ao processador. Por isto, o tempo de acesso à cache foi aumentado e também a energia por acesso. Mesmo assim, há uma redução do número de falhas na cache e, portanto do número de acesso à memória principal incorrendo no final em uma redução da energia total gasta.

Duas memórias externas foram utilizadas nos experimentos: uma SRAM *on-chip* de 512K bytes e uma Flash *off-chip* de 4M bits. A redução de energia observada foi de 30% usando a SRAM e 50% usando a Flash. A razão de compressão ficou em média em 75%, mas o tempo de execução do programa comprimido não foi comparado com o do código original. Mesmo diminuindo o número de acessos à memória principal, o que implica em ganho de desempenho, há uma perda de tempo por acesso à cache devido ao descompressor, então não é possível determinar o impacto que o método tem sobre o desempenho de forma direta. Além disto, embora escalonável, o descompressor depende do tamanho da linha da cache.

2.4 Conjunto de Instruções em 16/32 bits

A compressão por conjuntos de instruções reduzidos segue uma discussão do bom uso de instruções de 32 bits em detrimento de seus pares de 16 bits [66]. A iniciativa da ARM foi coabitar os seus processadores com ISAs mistos de 16 e 32 bits. O conjunto de instruções Thumb (16 bits) [67] é uma representação de 36 instruções de 16 bits, dos seus pares originais ARM (32 bits). Para tanto, os campos das instruções foram reduzidos. Por exemplo, os registradores originalmente especificados com 4 bits passam a ter apenas 3. Outra técnica usada foi a redução do número de registradores por instrução, quando aplicável, de 3 para dois, sendo um dos operadores fonte obrigatoriamente também

destino. Isto limita o número de registradores acessíveis ao thumb. A Figura 2.6 mostra o mapeamento entre os campos da instrução Thumb ADD Rd, Constant (soma uma constante (entre 0 e 255) ao valor de Rd e escreve o resultado em Rd) e os campos da instrução ARM correspondente.

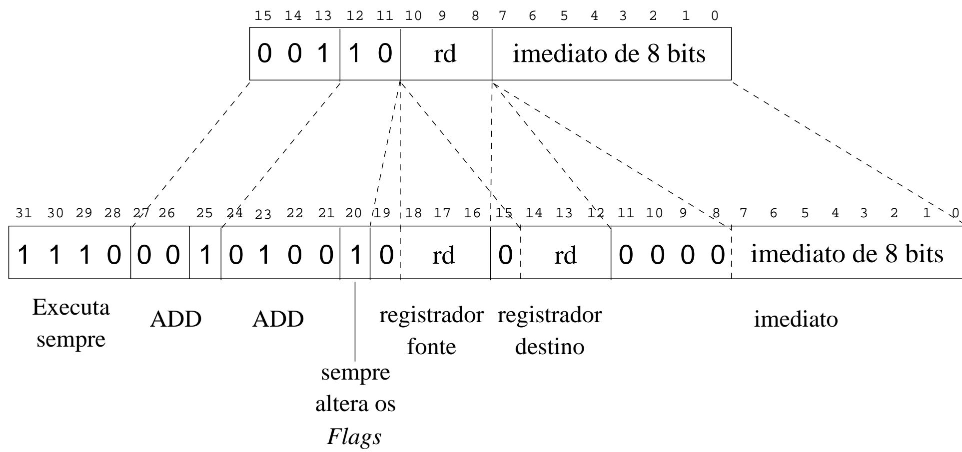


Figura 2.6: Mapeamento de campos de uma instrução Thumb para sua correspondente ARM

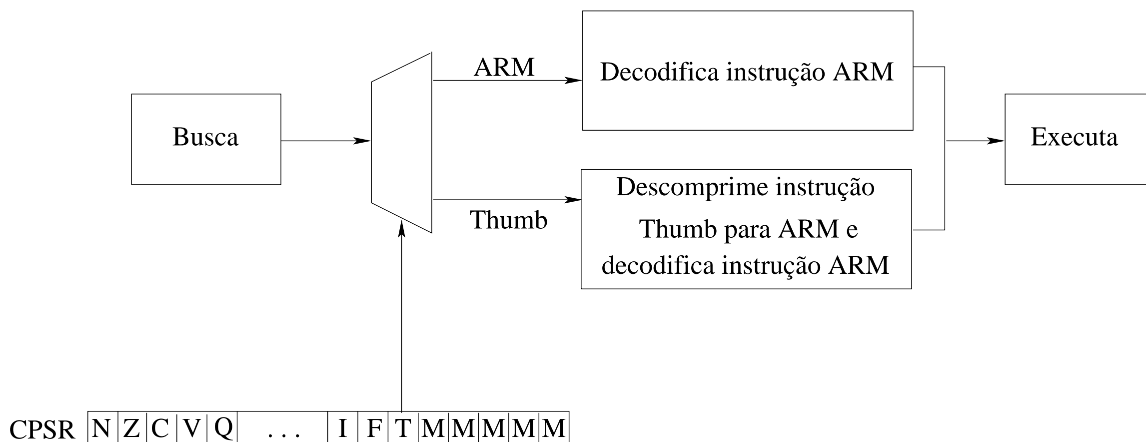


Figura 2.7: *Pipeline* do ARM/Thumb

O processador opera em dois modos distintos: ARM ou Thumb e o chaveamento entre estes dois estados ocorre através de instruções de Salto-Chaveamento (*Branch and Exchange* e *Branch-and-link and Exchange*). Estas instruções alteram o bit T do Registrador de *Status* do Programa Corrente, CPSR (*Current Program Status Register*) que indica qual *pipeline* deve ser utlizado. A Figura 2.7 mostra o *pipeline* do ARM/Thumb e o CPRS com o bit T selecionando o caminho da instrução. A escolha da granularidade do chaveamento depende do compilador e/ou de um programa associado. Um estudo interessante em [68] explora diversas alternativas, obtendo razões de compressão em torno de 70%. Neste estudo, o desempenho do sistema caiu cerca de 5%, ou seja, o número de ciclos de *clock* aumentou em 5%.

Uma outra iniciativa foi realizada pela MIPS, que implementou o MIPS16, codificando instruções do conjunto MIPSIII em 16 bits para coabitar com ISAs de 32 e 64 bits. A exemplo do Thumb, este novo conjunto de instruções também tem restrições no número de registradores acessíveis e no número de registradores por instrução. O chaveamento entre os conjuntos de instruções ocorre com uma instrução de *Jump and link and Exchange*. Kissel *et al* [69] reportam uma razão de compressão de 60%, mas não mencionam as aplicações usadas na medição nem o impacto no desempenho. Ambas as abordagens desprezam o tamanho do descompressor nos cálculos da razão de compressão.

Esta abordagem para compressão de código sofre com a redução da expressividade do conjunto de instruções, além de possuir um 'dicionário' estático de instruções comprimidas e assim não se adaptando às idiossincrasias de cada programa. Além disto, quando há necessidade de *padding*, 16 bits são utilizados, diminuindo a eficiência da compressão.

2.5 Outros Trabalhos

2.5.1 Compressão para VLIW

Nam *et al* [70] propuseram usar um método de dicionário, com fatoração de operandos para arquiteturas VLIW. Assumiram que os dicionários são ilimitados e cada entrada possui espaço para os operadores (operandos) de uma instrução VLIW inteira. Instruções com 4, 8 e 12 operações foram testadas e razões de compressão de 63%, 69% e 71% foram respectivamente alcançadas. Conte *et al* [71] propuseram uma codificação de instruções VLIW onde as operações de NOP, usadas como *padding*, não estão presentes. O artigo se limitou a apresentar diversos mecanismo de *fetch* para esta codificação, sem explicitar o

ganho em termos de compressão. De forma ilustrativa os autores citaram que em média apenas 2 operações dentre 8 são usadas em uma instrução. Xie *et al* [72, 73] propuseram utilizar um codificador aritmético para comprimir código para o TMS320C6x VLIW DSP. Dois modelos foram propostos: um estático e um semi-adaptativo baseado em Markov. Usando a modelagem estática, a razão de compressão ficou em 80% e para o modelo baseado em Markov, em 72%. A descompressão, apesar de ter sido melhorada por um algoritmo de conversão de *codewords* de tamanho variável para fixo, leva de 5 (estático) a 40 (Markov) ciclos por instrução.

2.5.2 *Wire Code*

Ernst *et al* em [19] também propuseram um compactador para código intermediário aperfeiçoado, baseado no trabalho do Fraser [74]. A seqüência de ações tomadas para construir o *wire code*[5] é a seguinte: compila o fonte para árvores de código virtual (usando a representação intermediária do lcc); separa os operandos dos operadores e forma *streams* distintas; aplica uma técnica de melhoria da localidade temporal em códigos comprimidos: MTF (*move-to-front*, [75, 76]); codifica os índices MTF usando Huffman; e finalmente comprime todas as *streams* com o gzip (baseado em LZ). O tamanho do código comprimido fica em torno de um quinto do código sem compressão. Neste caso o código inteiro precisa ser descomprimido antes de ser utilizado (interpretado ou compilado com um JIT, *Just-In-Time compiler*).

Em 1997 Franz e Kistler [77] apresentaram o *slim binaries*. Neste formato os arquivos objetos não contêm código executável, mas sim uma representação intermediária do programa, "altamente compacta"e independente de arquitetura. Um gerador de código é utilizado em diversas plataformas para gerar, a partir desta representação compacta, o código executável. O método de compressão está baseado em LZW, com as devidas alterações para lidar com árvores de sintaxe abstrata. A principal diferença entre este trabalho e o supracitado ([19]), além do método de compressão, é que este comprime dados e código enquanto aquele comprime apenas os segmentos de código. Isto pode mascarar os resultados que levam o *slim binaries* a ter uma razão de compressão de 33%, enquanto o método de Ernst apresenta um valor bem melhor, de 20%.

Drinic *et al* [78] usaram uma abordagem inversa, ou seja, os autores partem do

[5] Código que não é interpretado diretamente, mas que pode ser parcialemente descomprimido para uma forma interpretável ou mesmo compilado (incluíndo a descompressão) para uma forma executável.

binário para comprimir o código. A idéia é melhorar o desempenho de uma variante do conhecido algoritmo de Predição por Casamento Parcial (PPM, *Prediciton by Partial Matching*), o PPMD [79], explorando a sintaxe e a semântica do código. Dois passos são realizados antes da compressão: a separação em *streams* e um re-escalonamento de instruções para intensificar a compressão (torná-la mais densa). As *streams* resultantes são, então, comprimidas separadamente usando o PPMD. A razão de compressão final para o conjunto do Microsoft OfficeXP, o MSWord2000 e um compilador C ficou em torno de 40%, o que representa uma melhoria considerável sobre a aplicação do PPMD diretamente no executável (razão de compressão de 50%).

2.5.3 Compressão Seletiva

Embora os primeiros trabalhos em compressão usando hardwares descompressores já tenham apresentado alguma seletividade, dois em especial se dedicaram a esta tarefa.

Debray e Evans [80] propuseram um método de compressão seletivo baseado em informações de *profiling* dinâmico, seguindo a mesma idéia de Lefurgy [57] de comprimir apenas os trechos menos executados. Ambas as abordagens usam um descompressor em software, diferindo em especial na granularidade do código descomprimido que aqui é feita por função (no primeiro usa-se uma linha da cache). Uma outra diferença é que Lefurgy necessita de uma instrução capaz de escrever na cache, enquanto nenhum suporte de hardware é requerido neste trabalho. A proposta é comprimir as funções que são menos executadas. No código original estas funções são substituídas por *stubs* (pequeno trecho de código) que invocam o descompressor passando um argumento que é um índice para uma tabela de endereços de funções comprimidas. O descompressor recebe este argumento, consulta o endereço da função comprimida, descomprime-a para um *buffer* especial e transfere a execução para esta localidade.

A partição do código em frio (menos executado) e quente (mais executado) é feita a partir de um limiar (*threshold*) que indica a porcentagem de código frio presente (com granularidade de funções). Por exemplo, um limiar de 0,25 (25%) indica que todo código identificado como frio não pode ultrapassar 25% do total de instruções executadas. É notório observar que quando frações não muito grandes de código frio são executadas o impacto no desempenho é enorme. Para se ter uma idéia, quando o limiar é de apenas 0,005% (das funções) alguns códigos já requerem o dobro de tempo para serem executados. Para obtenção da razão de compressão os experimentos foram primeiro comprimidos com

o *squeeze*[6] (já reduz 30%) e depois submetidos ao *squash* (ferramenta para este modelo de compactação). Uma redução adicional no tamanho do código entre 14% e 19% foi observada, já considerando o tamanho do *buffer*, dos *stubs* e da tabela de endereços.

Da mesma forma, Xie [81] tornou seu método de compressão anterior seletivo. Isto significa que a perda de desempenho devido ao uso de uma compressão aritmética, normalmente lenta em decodificação, fica minimizada. A razão de compressão final para o único *benchmark* explorado (ADPCM) ficou entre 72% e 88%, dependendo do valor do limiar usado para separar o código em 'código quente' e 'frio'. O impacto da descompressão variou entre 1,1 e 8 vezes o tempo necessário para execução do código original. Mais uma vez é interessante perceber que a inclusão de poucas instruções do código quente na compressão deteriora substancialmente o desempenho. Se apenas 10% do código quente for comprimido o número de ciclos para execução do código cresce 50% para o algoritmo de melhor razão de compressão.

2.6 Sumário e Conclusões

Um grande desafio na vasta literatura de compressão de código é apresentar comparativos entre os diversos métodos. Não somente por sua diversidade, mas muitas vezes por sua ortogonalidade. Classificar os métodos já é tarefa árdua e comparar, usando plataformas tão distintas como visto acima, pode ser extremamente injusto. Um recente esforço Húngaro-Finlandes[82] apresenta 6 classificações possíveis e 8 métricas de avaliação para 12 métodos de compressão e conclui corroborando nossa observação de que não há como fazer uma comparação justa dos métodos.

Mesmo assim, apresentamos de forma compendiada os resultados, para uma melhor visualização de algumas métricas. Tentamos apresentar os resultados mais expressivos, sem nos atermos a melhorias pontuais ou métodos derivados diretamente de outros. As tabelas a seguir (Tabela 2.1, Tabela 2.2, Tabela 2.3 e Tabela 2.4) mostram os resultados dos principais métodos estudados.

Alguns cuidados básicos devem ser tomados ao avaliarmos estas tabelas. Primeiro: a arquitetura base pode ser um diferencial. Uma extensão do CCRP para diversas arquiteturas mostrou que o tamanho do código gerado pelo compilador pode variar consideravelmente [25]. A compressão também depende desta arquitetura: quanto mais expressivo for o conjunto de instruções maior é a maleabilidade do compilador em gerar

[6] Veja seção sobre compactação

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução
Debray	squeeze	Alpha	Mediabench SPEC95	54%	-16%
Fraser	lcc	SPARC	lcc	50%	+2000%
Ernest	BRISC	SPARC	lcc, ggg, word, agrep, xlisp, espresso	60%	+1260%
Liao	SOA/GOA	TMS320C25	chendct, chenidct, leedct, ileedct, jrev, readgif, autocrop, smooth, hufftree, gnucrypt	80% / 97%	–

Tabela 2.1: Compêndio dos métodos de compactação

código e por isto menores as chances para compressão. Outro fator é o conjunto de otimizações usadas pelo compilador. Quanto mais otimizado o código menos chances para compressão surgem. Os *benchmarks* também são importantes: programas menores parecem mais difíceis de comprimir porque há menos chances de redundância.

Para as arquiteturas SPARC, os melhores resultados vêm de uma compactação de código [18], mas apenas 1 *benchmark* foi avaliado. Os demais métodos apresentam razão de compressão em torno de 60%. Para arquiteturas MIPS a maioria dos trabalhos usa o SPEC95 e por isto é mais fácil arriscar dizer que o método que apresenta melhores resultados é o SADC, porém, este método não computa em sua razão de compressão o tamanho das tabelas. Sem computá-las o TBC e o IBC passam a ser melhores, com uma razão de compressão em torno de 30%. Para o TMS320C25 o TBC se apresenta com melhor razão de compressão em relação ao método de Liao, usando praticamente os mesmos aplicativos.

Quanto à energia consumida na cache de instruções, o trabalho de Benini apresenta os melhores resultados, mas infelizmente, neste trabalho, o ciclo de leitura da cache aumenta de tamanho então, potencialmente interfere negativamente no desempenho do sistema (cujos dados não foram apresentados).

O melhor desempenho apresentado na literatura é do Thumb/ARM, mas não há informações sobre o conjunto de aplicações que foi utilizado para obter este valor. Em um trabalho derivado, usando o Mediabench o desempenho caiu 5%.

Na literatura podemos encontrar muitas alternativas de uso em artefatos reais.

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução
Wolf e Chanin	CCRP	MIPS	lex, pswarp, yacc, eightq, matrix25 lloop01, xlisp, espresso e spim	73%	–
kirovski	*pcache*	SPARC	Mediabench SPEC95	40%	+11%
IBM	CodePack	PowerPC	Mediabench SPEC95	60%	±10%
Lekatsas	SAMC	MIPS	SPEC95	57%	–
Lekatsas	SADC	MIPS	SPEC95	52%	–
Lekatsas	SADC	PentiumPro	SPEC95	75%	–
Lefurgy	por Software	MIPS	Mediabench SPEC95	65%	+2% ∼ +54%
Pannain	PBC	MIPS	SPECint95	61,3%	–
Centoducatte	TBC	MIPS	SPECint95	60,7%	–
Centoducatte	TBC	TMS320C25	aipint, bench, gnucrypt, gzip, hill, jpeg,rx, set	75%	–
Azevedo	IBC	MIPS	SPECint95	53,6%	–
Azevedo	IBC	SPARC	SPECint95	63,8%	-12% ∼ +84%

Tabela 2.2: Compêndio dos métodos CDM

Os melhores resultados em compressão são para compactação em nível de código intermediário. O principal problema com esta abordagem é a engenharia reversa dos códigos que se torna muito facilitada. Industrias de softwares provavelmente não adotariam tais modelos por expor os seus códigos fontes. Outro problema é a baixa velocidade de descompressão que degrada muito o desempenho.

Os métodos CDM podem explorar algoritmos de compressão mais complexos embora, via de regra, sejam mais lentos na descompressão. A razão de compressão costuma ser melhor que os PDC, mas o ganho em desempenho e consumo de energia é pior.

Os métodos PDC, em sistemas com cache, ainda são pouco explorados. Principalmente a possibilidade de alteração do processador para suportar um método destes sem causar a desnecessária replicação da unidade de cálculo do PC. Pode-se avaliar, por exemplo, como uma arquitetura destas, com modelagem semi-adaptativa, se encaixaria em um conjunto de instruções de 16 ou mesmo 8 bits.

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução	Energia†
Liao	mini-sub rotina	TMS320C25	aipint, bench gnucrypt gzip, hill jpeg, rx, set, cache, compress	84% ∼ 88%	+115%	–
Lefurgy	dicio-nários	PowerPC	SPECint95	61%	–	–
Lefurgy	dicio-nários	ARM	SPECint95	66%	–	–
Lefurgy	dicio-nários	i386	SPECint95	74%	–	–
Benini	sem cache	MIPS	Ptolomy	90%	–	-46%‡ ∼ -52%‡
Benini	com cache	MIPS	Ptolomy	72%	–	-50%
Lekatsas	com cache	SPARC	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	-28%
Lekatsas	com cache	Xtensa 1040	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	

†Quando não especificado a energia se refere àquela consumida na cache
‡Energia total no sistema

Tabela 2.3: Compêndio dos métodos PCD

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução	Energia†
ARM	Thumb	ARM	–	55% ∼ 70%	-30%	–
MIPS	MIPS16	MIPS	–	60%	–	–

Tabela 2.4: Compêndio dos ISAs mistos de 16/32 bits

Capítulo 3

Dicionários Baseados em *Multi-Profile*

O conjunto 'reduzido' de instruções das arquiteturas RISC e a regularidade do código gerado pelos compiladores produzem uma grande quantidade de instruções e/ou padrões de instruções que se repetem com freqüência nos programas. Seguindo esta premissa, neste capítulo abordaremos o processo de construção de dicionários de instruções para utilização em compressão de código. Na grande maioria dos métodos de compressão que utilizam dicionários, publicados na literatura, o processo de sua formação está baseado em informações estatísticas obtidas do código de forma estática (*profile* Estático). Recentemente uma abordagem dinâmica, usando a contagem de instruções executadas (*profile* Dinâmico), foi experimentada por alguns pesquisadores [65, 72].

Em geral, métodos que utilizam a abordagem estática para formação do dicionário obtêm melhores razões de compressão, enquanto aqueles que se baseiam em informações dinâmicas apresentam maior ganho de desempenho e/ou menor consumo de energia. A abordagem desenvolvida neste trabalho procura associar os melhores resultados dos dois procedimentos, produzindo um novo método de construção de dicionários denominado *Multi-Profile*.

3.1 Redundância de Instruções

O princípio da localidade estabelece que em um programa 90% do tempo de execução é gasto em apenas 10% das suas instruções [49]. Também foi verificado que, estaticamente,

Categoria	Aplicações	D1 †	D2 †
Automotiva & Controle Industrial	*susan*◊	Imagem em branco e preto de um retângulo	Imagem complexa de um ambiente de uma biblioteca
Bens de Consumo	*cjpeg*△	Imagem de uma rosa	Monalisa
	djpeg△	Imagem de uma rosa	Monalisa
Automação de Escritório	stringsearch◊	Busca de 1332 palavras em 1332 frases (em inglês)	Busca de 1332 palavras em 1332 frases (em português)
Redes	*dijkstra*◊	Grafo de 100 vértices para encontrar o s10 menores caminhos	Outro grafo de 100 vértices
Segurança	*pegwit*△	Texto com listagem de instruções SPARC	Texto bíblico
Telecomunicações	*adpcm_e*△	Clinton	S_16_44
	adpcm_d△	Clinton	S_16_44

†D1 e D2 foram dimensionados de tal forma a possuirem o mesmo tamanho
△Mediabench
◊MiBench

Tabela 3.1: *benchmarks* utilizados

cerca de 20% das instruções de um código são suficientes para cobrir 80% de seu todo [8] (quando utilizando o processador MIPS e a suíte de *benchmarks* SPECint95). Não obstante, estas regras precisam ser verificas *in loco*, dado que cada conjunto de programas de testes para avaliação de desempenho (*benchmark*) possui características próprias que podem indicar um melhor modelo a ser aplicado na compressão.

Os experimentos relatados a seguir usam como base um conjunto de aplicações retirados do *Mediabench* [83] e do *MiBench* [84]. Dentro do escopo deste trabalho procurou-se avaliar o desempenho de compressores e descompressores de código para sistemas embarcados, especialmente capazes de executar códigos multimídia. Assim sendo, uma grande variedade de aplicações se apresentam como possíveis candidatos a testes de desempenho. Neste trabalho foram utilizados como *benchmarks* programas representativos das seis categorias propostas no *MiBench* mesclando códigos das duas suítes. A Tabela 3.1 mostra as seis categorias e os *benchmarks* implementados. Além disto, foram avaliados dois conjuntos de dados de entradas (D1 e D2) para cada programa com a finalidade de determinar qual a influência destes na construção dos dicionários.

O *benchmark susan* é um pacote de reconhecimento de imagens. Ele foi desenvolvido para reconhecer cantos (*corners*) e bordas (*edges*) em imagens de ressonância magnética do cérebro. O JPEG converte imagens de um formato para outro. Usamos entradas no formato PPM para o codificador JPEG (*cjpeg*), bem como para as saídas do decodificador (*djpeg*). O *stringsearch* (*search*) busca palavras em frases com um algoritmo insensível ao caso (maiúsculas ou minúsculas). O *dijkstra* busca os menores caminhos entre os vértices de um grafo. O *pegwit* é usado para criptografar um texto a partir de uma combinação de chaves pública e privada. Finalmente o ADPCM converte sons no formato *pcm* para *adpcm* (*adpcm_e*) e vice-versa (*adpcm_d*). Todos os *benchmarks* foram compilados para a arquitetura SPARCv8[85] com o compilador LECCS[86][1] usando opção –O2 para evitar otimizações que tipicamente aumentam o tamanho do código.

A contagem de instruções únicas nos programas pode ser vista na Tabela 3.2. Em média, apenas 35% das instruções de um código são únicas, ou seja, todo escopo restante do código é formado de repetições. Além disto, o valor acumulado da contribuição percentual de cada uma destas instruções na construção do código (Figura 3.1) revela que com apenas 30% de instruções únicas já é possível expressar 70% do código. Isto significa que a premissa usada na introdução deste capítulo é aplicável aos *benchmarks* utilizados. Além disto, a regra geral de execução de instruções também pode ser averiguada. Na Figura 3.2 vemos a distribuição das instruções na execução do código. Verifica-se que 10% das instruções são responsáveis por mais de 90% da execução do programa.

O que se espera de um algoritmo de compressão é que ele possa fazer uso destas redundâncias intrínsecas de instruções nos programas para diminuir o tamanho do código.

Programa	Tamanho (em instruções)	Instruções Únicas (%)
susan	18.604	6.671 (36%)
cjpeg	25.316	8.231 (33%)
djpeg	28.692	9.616 (34%)
search	8.760	3.530 (40%)
dijkstra	7.944	3.163 (40%)
pegwit	18.804	6.699 (36%)
adpcm_e	2.284	683 (30%)
adpcm_d	2.284	683 (30%)

Tabela 3.2: Número e procentagem de instruções únicas para os *benchmarks*

[1] Compilador cruzado baseado no GCC 2.94

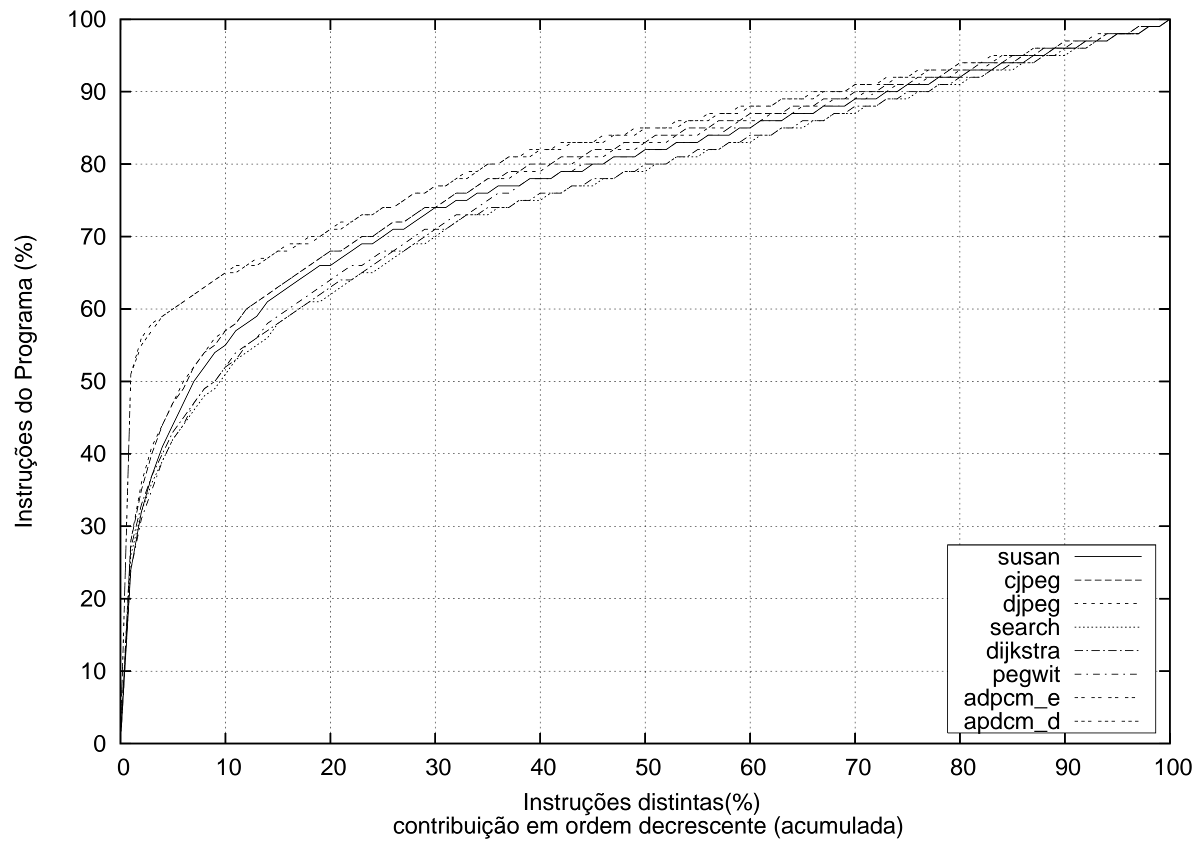


Figura 3.1: Porcentagem do programa coberta pelas instruções únicas

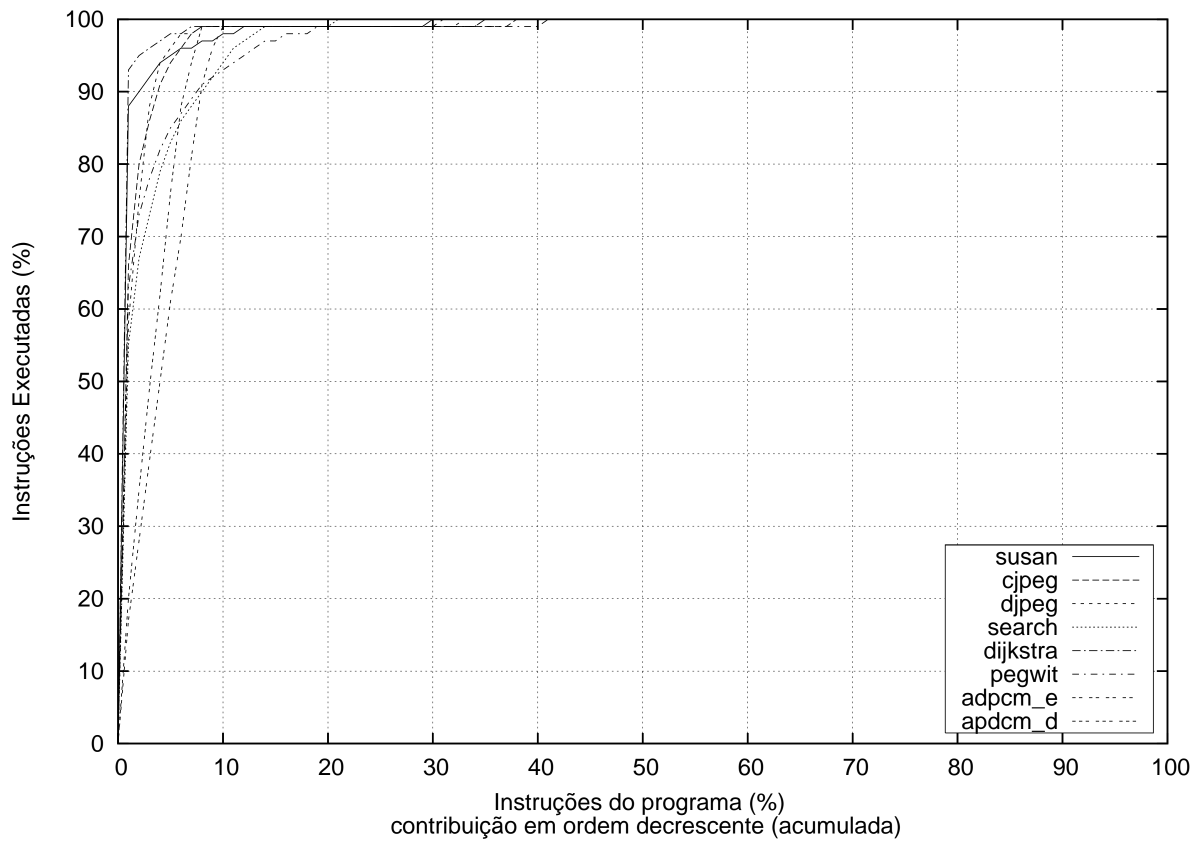


Figura 3.2: Porcentagem da execução do programa coberta pelas instruções únicas

3.2 Similaridade Entre os Dicionários

Na construção de dicionários de instruções, normalmente são feitas uma contagem e ordenação das instruções do código por seu número de ocorrências. Algumas técnicas utilizam esta ordenação para atribuir *codewords* menores às instruções que mais ocorrem, dentro de um mesmo dicionário [8], ou para decidir que instruções devem compor um dicionário incompleto [87]. Esta ordem, portanto, passa a ter uma certa relevância dependendo do método de compressão que a utilizará.

Neste trabalho, chamamos de dicionário estático[2], SD (*Static Dictionary*), aquele formado com base em informações estáticas de *profiling* do código e de dicionário dinâmico, DD (*Dynamic Dictionary*) aquele formado com base na contagem do número de vezes que cada instrução é executada.

A investigação qualitativa da formação destes dicionários, usando como base informações estáticas e dinâmicas, revelou que as primeiras instruções que compõem o dicionário estático são consideravelmente distintas das primeiras instruções do dicionário dinâmico. A Figura 3.3 mostra o grau de interseção entre os dicionários estáticos e dinâmicos. Podemos verificar, por exemplo, que até cerca de 25% do tamanho do dicionário menos de 10% das instruções são comuns aos dois. Isto é, notadamente, um dado a ser considerado em métodos que utilizam dicionários ordenados.

Além desta observação, dentro de um dicionário, a contribuição de cada instrução (na formação do código ou na execução do mesmo) forma uma curva com declive bastante acentuado (mais ainda que uma hipérbole típica), ou seja, poucas instruções são de fato muito redundantes. A Figura 3.4 mostra a contribuição de cada instrução dentro do dicionário ordenado normalizada pela instrução que mais ocorre (a) estática e (b) dinamicamente.

A escala logarítmica ajuda na visualização das contribuições de até 10% das instruções do dicionário. Daí em diante a contribuição individual de cada instrução é pífia com relação às instruções que mais ocorrem (estática ou dinamicamente).

Estes gráficos nos levam a perceber que há uma necessidade de considerar tanto informações estáticas como dinâmicas das instruções na ordenação de um dicionário.

---

[2] O termo dicionário estático é, de fato, um abuso de notação pois o mesmo possui outra semântica na teoria clássica de compressão

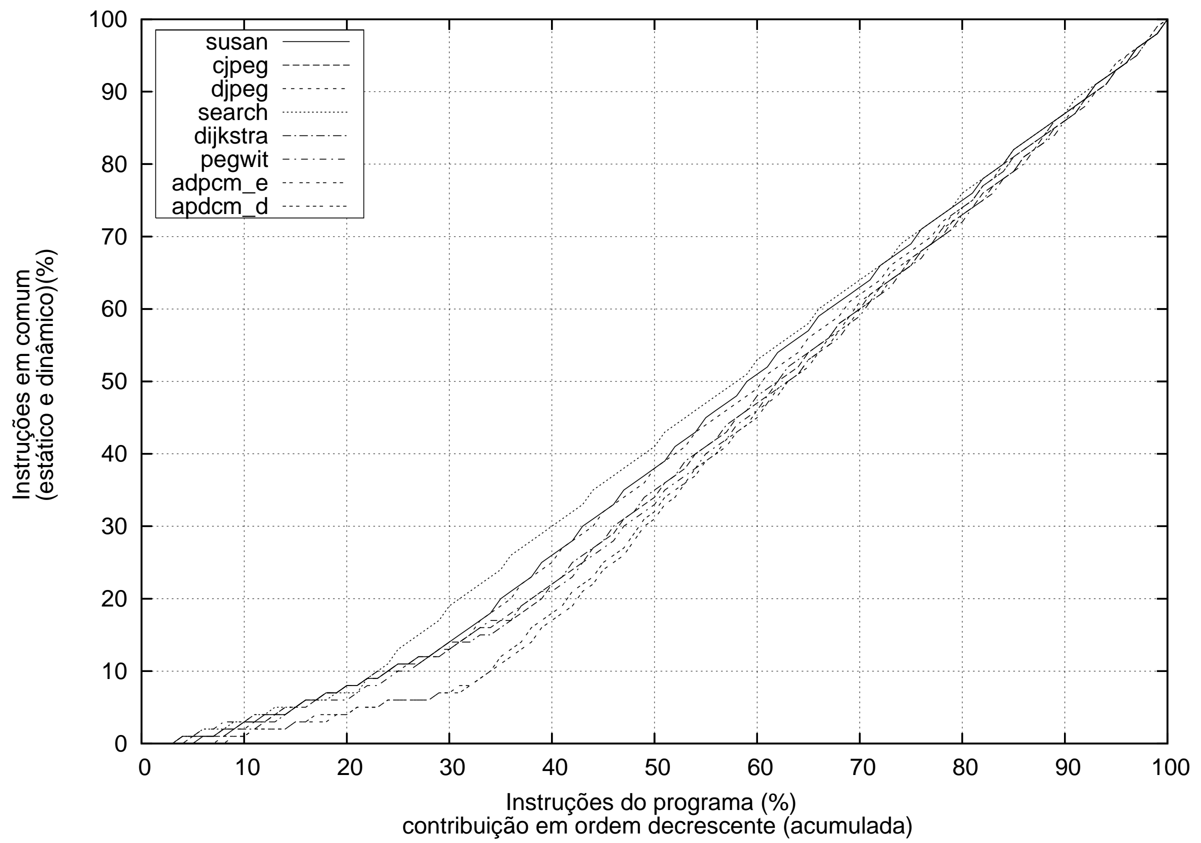
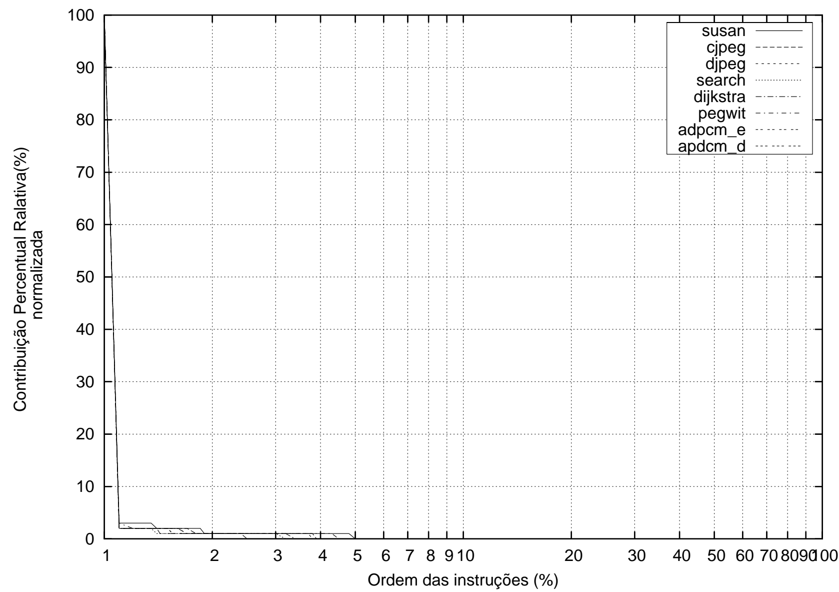
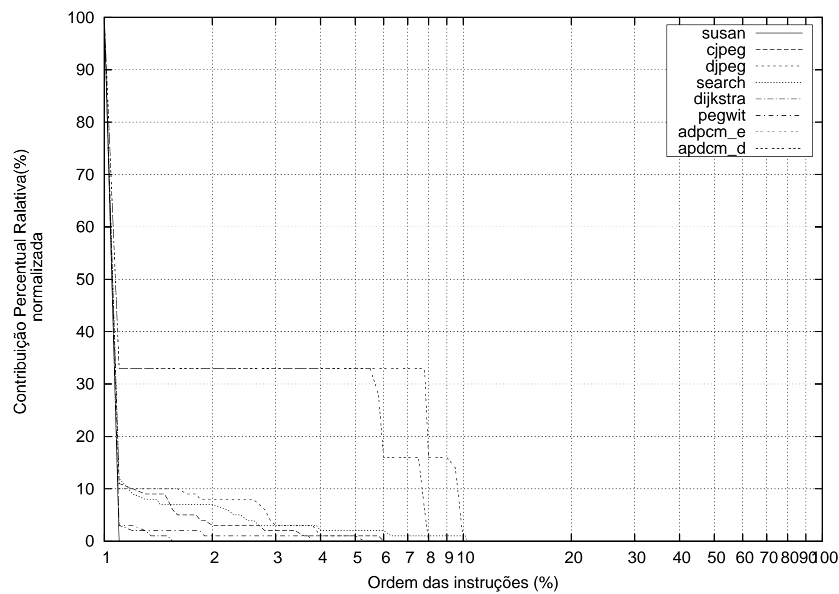


Figura 3.3: Grau de interseção entre os dicionários estático e dinâmico



(a) Dicionário Estático



(b) Dicionário Dinâmico

Figura 3.4: Contribuição percentual das instruções dentro dos dicionários estático e dinâmico (normalizado pela maior contribuição)

3.3 Influência dos Dados de Entrada

Dicionários estáticos não sofrem alterações no seu processo construtivo devido a variações na massa de dados de entrada, uma vez que o número de ocorrência de cada instrução no programa não depende dos dados. Por outro lado, a execução de um programa pode ser consideravelmente influenciada pelos dados que por ele são processados. Isto pode se tornar mais crítico quando usamos um conjunto de *benchmarks* e a qualidade dos dados de entrada utilizados não é representativa da massa típica de dados. Então, dois fatores são determinantes para a construção de um dicionário dinâmico: quão orientados por dados são as aplicações e quão representativo é o conjunto de entrada de dados fornecido na suíte de *benchmarks*.

O fato de escolher um novo conjunto de dados de entrada vem a responder sobre a orientação da aplicação pelos dados. Entradas com as mesmas características (duas imagens JPEG ou dois arquivos de som ADPCM) também são importantes para determinar esta influência. Escolher entradas de formatos distintos implicaria em tomar outros caminhos dentro do código, não necessariamente influenciados pelos dados que compõem a imagem (ou som, ou texto), mas também pelo seu tipo. Nosso foco, entretanto, foi alterar apenas os dados que compõem o arquivo de entrada, permitindo uma análise de granularidade mais fina quanto à orientação à dados.

Mas afinal, quão diferentes são os dicionários formados usando dois conjuntos de dados do mesmo tipo? A Figura 3.5 mostra o grau de similaridade entre os dicionários dinâmicos de cada aplicação formados pelos dois conjuntos de entradas mostrados na Tabela 3.1. Via de regra, para os *benchmarks* utilizados, os dicionários mantém grande similaridade, isto é, não sofrem influência dos dados de entrada de um mesmo formato e/ou a aplicação não é influenciada decisivamente pelos dados.

3.4 Estudo de Dicionários Incompletos Pequenos

O termo "dicionários pequenos", em especial a qualificação "pequenos"requer um esforço para justificar o vocábulo. De fato, um limite entre pequeno e grande não é mesmo motivo de análise na comunidade científica, mesmo porque há uma forte componente temporal em jogo, a saber: o grande de hoje pode ser o pequeno de amanhã. Esquivando-se desta discussão mais filosófica, a terminologia "pequeno"neste trabalho diz respeito a dicionários de até 256 entradas. Estes dicionários têm sido usados com freqüência nos últimos

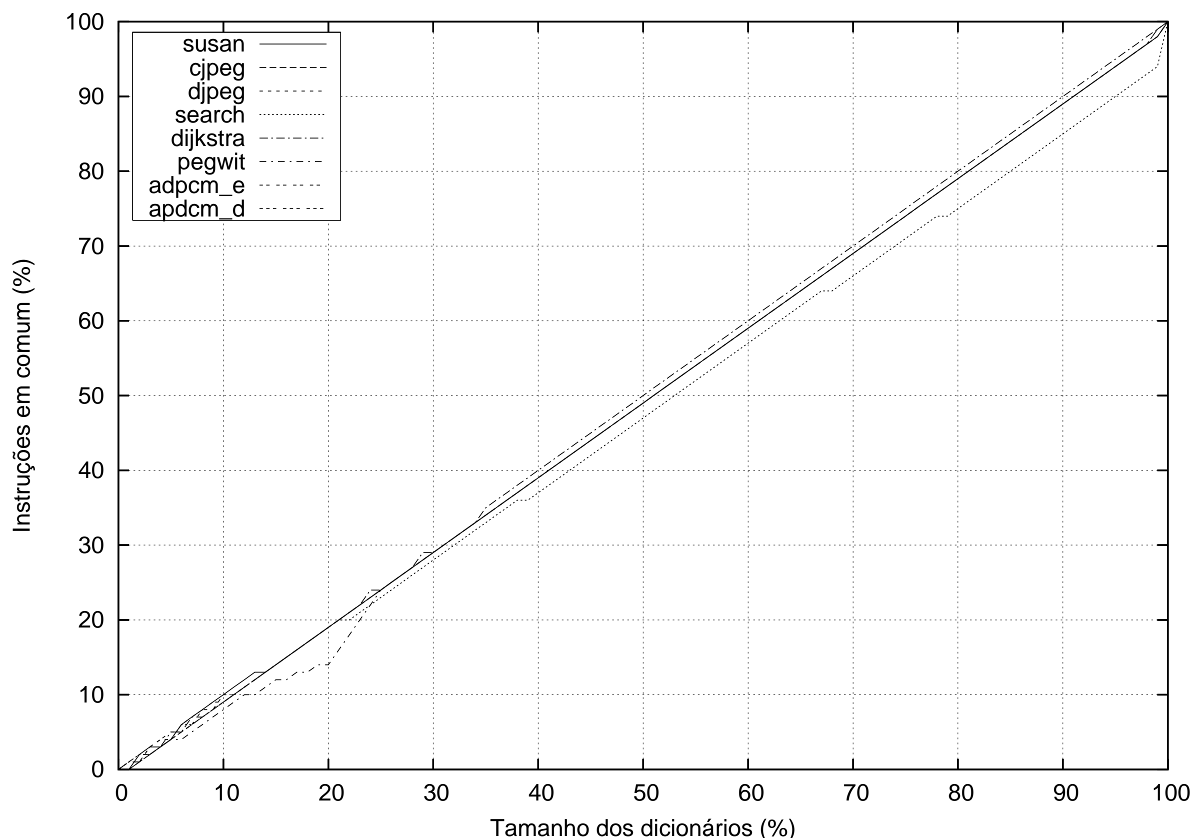


Figura 3.5: Similaridade entre dicionários dinâmicos formados por dados de entrada distintos

trabalhos sobre compressão de código apresentados na literatura, por isto merecem uma atenção especial. Em um modelo rudimentar de nosso compressor, descrito no Capítulo 5, chegamos mesmo a equacionar o uso de outros tamanhos de dicionários, entretanto, o que melhor se adequou ao algoritmo de compressão (produziu melhores razões de compressão) foi o de 256 instruções.

3.4.1 Similaridade Entre Dicionários Pequenos

Já foi mencionado que as primeiras instruções dos dicionários estáticos formam uma pequena interseção com seus pares dos dicionários dinâmicos. Quando os dicionários são pequenos, esta interseção diminuta pode influenciar substancialmente na razão de compressão do código e nas características de desempenho e energia envolvidas no uso de código comprimido. O gráfico na Figura 3.6 mostra o quão similar são as instruções

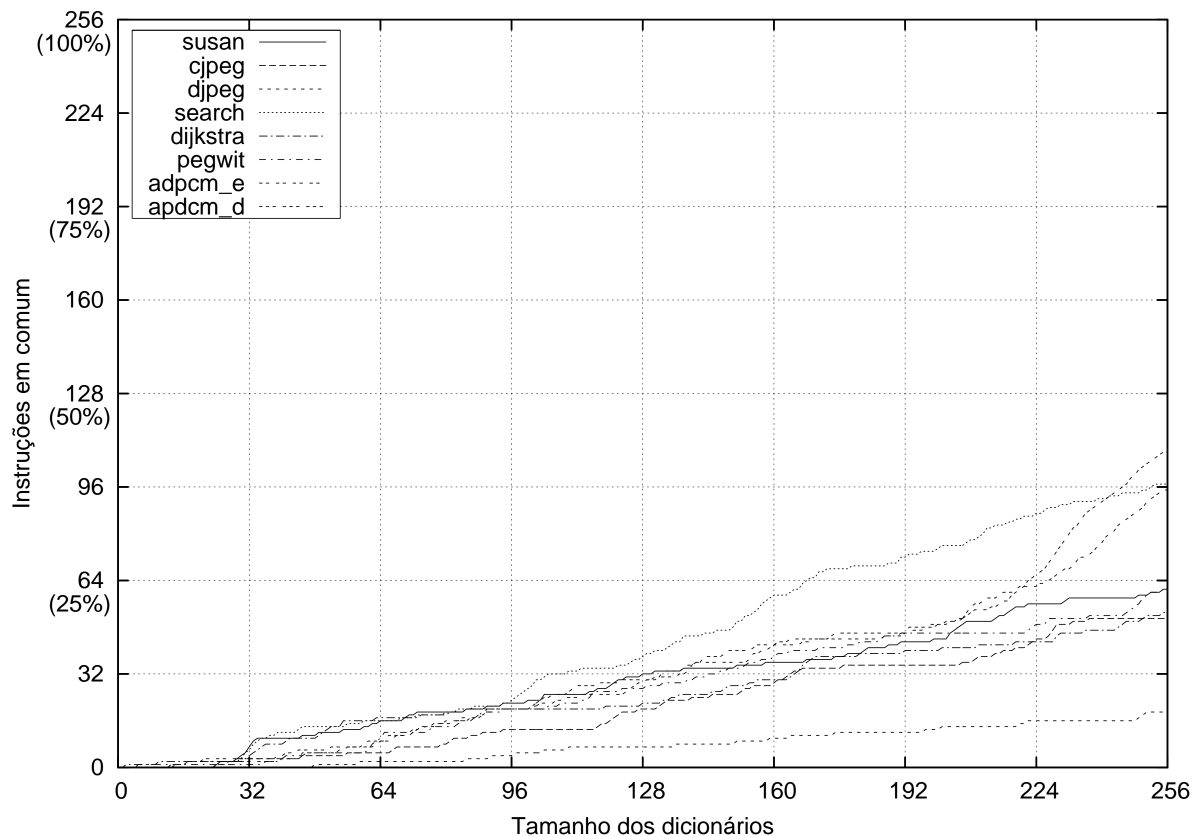


Figura 3.6: Similaridade entre dicionários pequenos estáticos e dinâmicos

dos dicionários estáticos e dinâmicos para os diversos tamanhos de dicionários pequenos. Para o caso de 256 entradas, apenas 26% (em média) das instruções são comuns a ambos os dicionários. Para dicionários de 32 entradas, apenas 10% das instruções são comuns.

Também a contribuição das instruções dentro destes dicionários pequenos forma uma curva de declive bastante acentuado. A Figura 3.7 e a Figura 3.8 mostram a contribuição de cada instrução, normalizada pela maior contribuição. Mais uma vez percebe-se que as primeiras instruções de um dicionário são fundamentais na exploração da redundância.

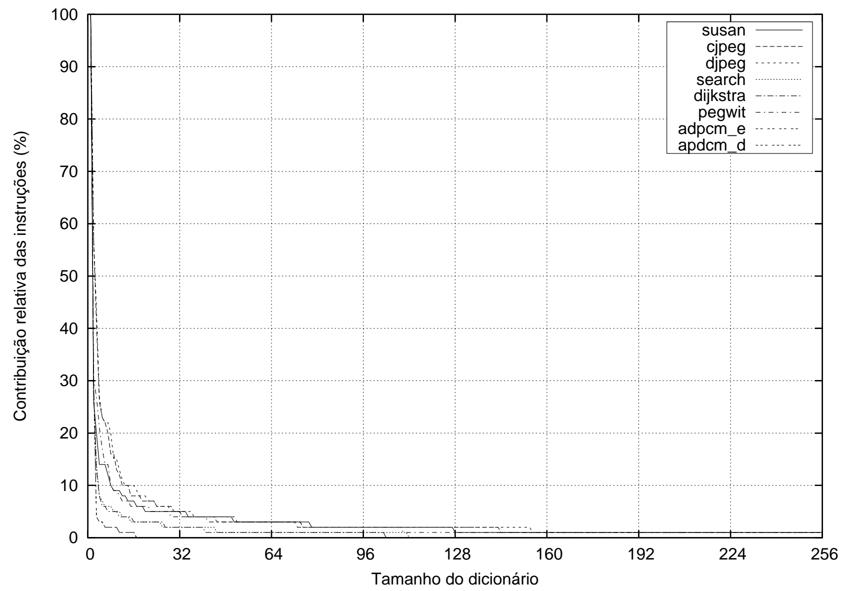


Figura 3.7: Distribuição da contribuição de cada instrução dentro do dicionário estático (normalizada)

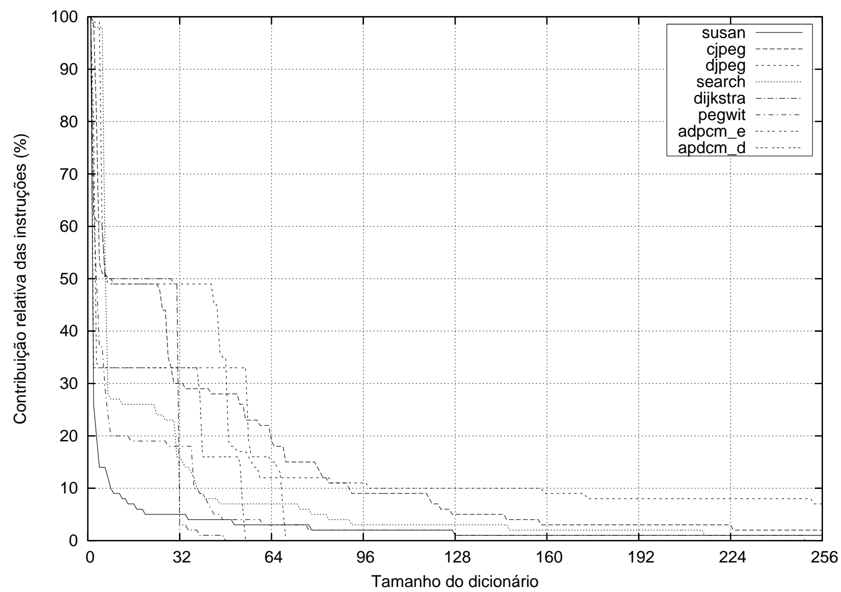


Figura 3.8: Distribuição da contribuição de cada instrução dentro do dicionário dinâmico (normalizada)

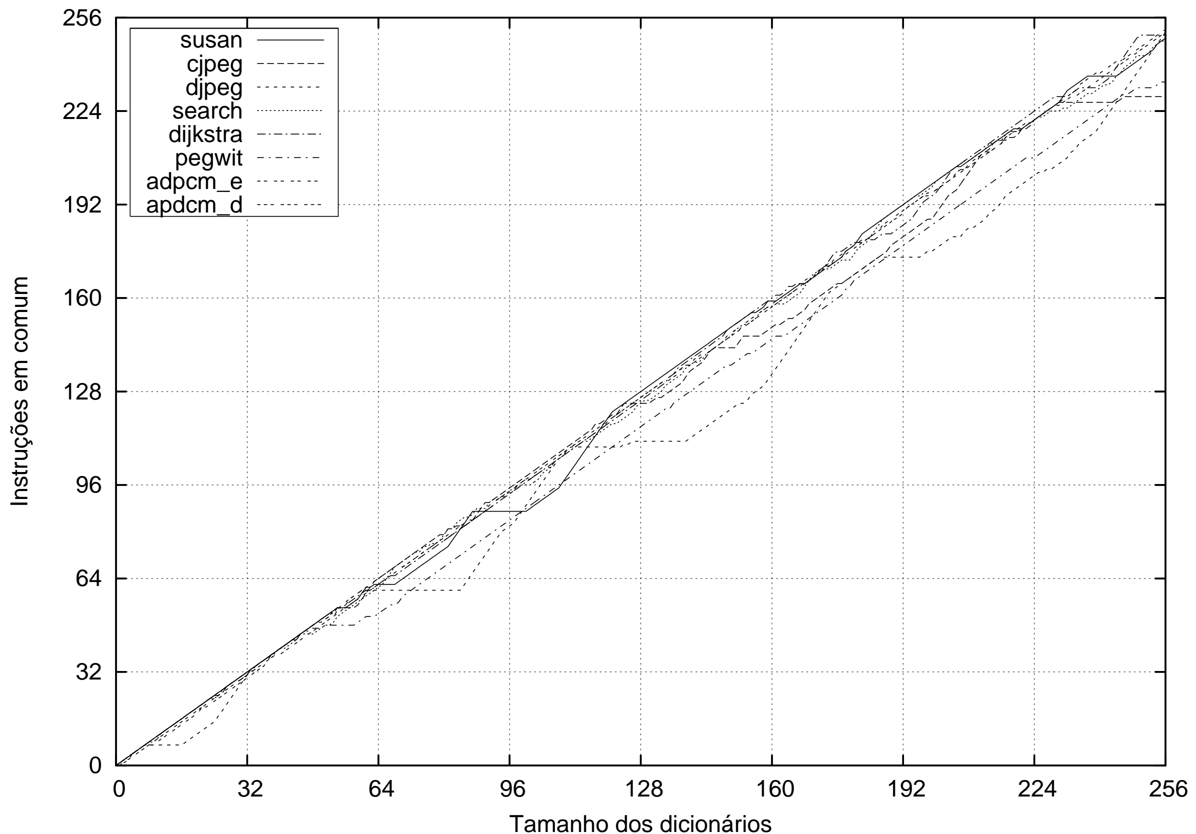


Figura 3.9: Similaridade entre dicionários dinâmicos para dois conjuntos de dados de entrada

3.4.2 Influência dos Dados de Entrada em Dicionários Pequenos

De forma similar ao que foi apresentado para os dicionários completos, agora vamos mostrar como os dicionários pequenos são influenciados pelos dados de entrada. A Figura 3.9 mostra o grau de interseção dos dicionários dinâmicos construídos a partir da execução dos programas com dois conjuntos de dados de entrada (vide Tabela 3.1). A tendência de similaridade se mantém, embora se exacerbem algumas diferenças.

3.4.3 Contribuição das Instruções de um Dicionário Pequeno na Construção de um Programa

Finalmente apresentamos o quanto cada dicionário pequeno contribui na formação de um programa. De fato, esta é uma reedição das figuras 3.1 e 3.2 para o caso de dicionários pequenos. Para dicionários estáticos, mostrados na Figura 3.10, podemos perceber que o

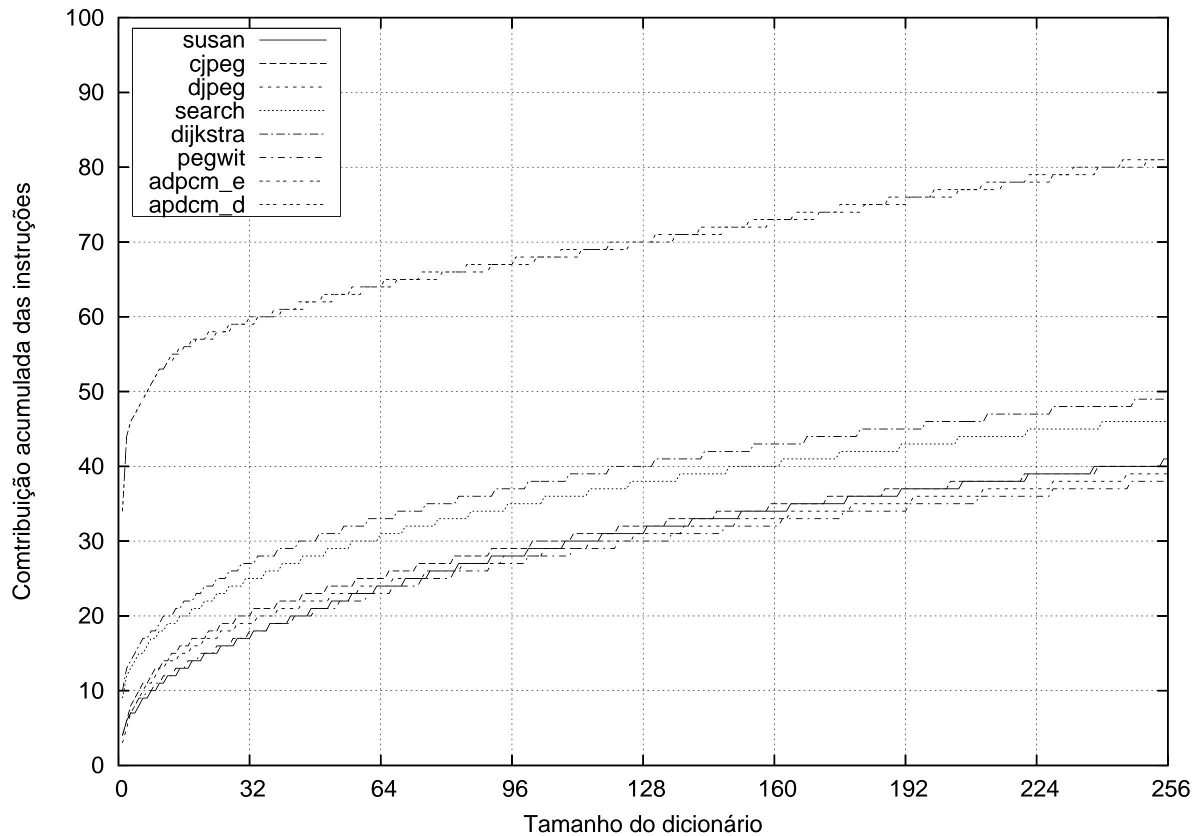


Figura 3.10: Contribuição acumulada das instruções (estáticas) de um pequeno dicionário na formação de um programa

ADPCM se destaca por produzir a maior influência na construção do código. Isto deve-se ao pequeno número de instruções únicas da aplicação (683). Um dicionário de 256 posições representa então 38% do dicionário completo. De todos os *benchmarks* utilizados este é o que apresenta maior tamanho relativo para um dicionário pequeno. No outro extremo está o *djpeg*. Um dicionário de 256 posições representa, para esta aplicação, apenas 2% do dicionário completo. De forma similar analisamos dicionários dinâmicos mostrados na Figura 3.11. Existem casos (*adpcm_e*, *adpcm_d*, *pegwit*) onde um dicionário pequeno contém quase a totalidade das instruções que são efetivamente executadas no programa.

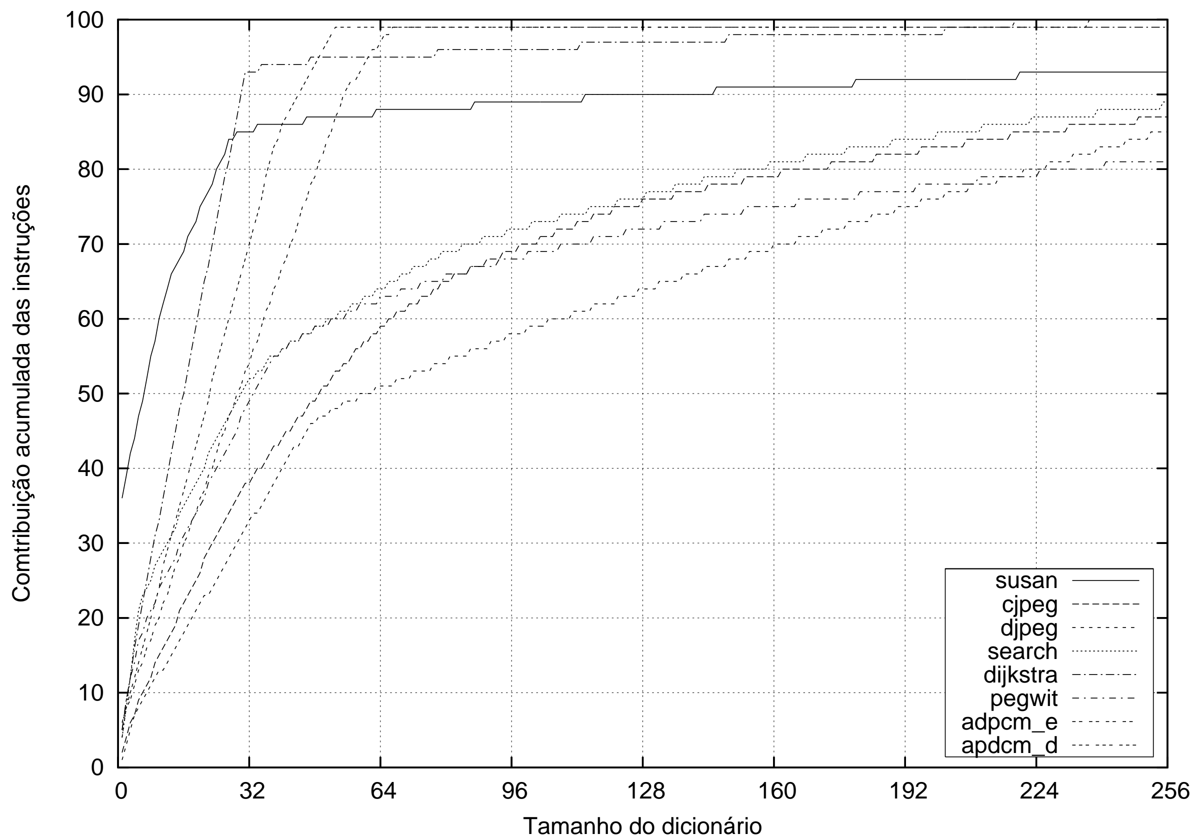


Figura 3.11: Contribuição acumulada das instruções (dinâmicas) de um pequeno dicionário na formação de um programa

3.5 Método *Multi-Profile* para Construção de Dicionários

Com base nas evidências acima expostas foi proposto um algoritmo para construir dicionários que levem em consideração informações estáticas e dinâmicas simultaneamente. O algoritmo procura incluir no dicionário em formação as instruções mais relevantes do dicionário estático, bem como as mais relevantes do dicionário dinâmico. A abordagem mais simples para o caso é intercalar as instruções dos dois dicionários (ordenados). O algoritmo, em pseudocódigo, está apresentado na Figura 3.12. O dicionário resultante da intercalação é chamado Dicionário Unificado, UD (*Unified Dictionary*). Este método de construção de dicionários foi publicado em [87].

Uma sofisticação deste algoritmo implica em determinar a contribuição de instruções dinâmicas dentro do dicionário unificado, isto é, poder dar mais ênfase às características

```
UD MERGE_DICT (SD, DD) {
  1 UD ← ∅
  2 Enquanto UD não estiver completo
    2.1 Verifica se a primeira instrução de SD está em UD
        - SIM: SD  ← SD - {primeira instrução de SD};
               Executa Passo 2.1
        - NÃO: UD  ← UD ⋃{primeira instrução de SD};
               SD  ← SD - {primeira instrução de SD};
    2.2 Verifica se UD está completo
        - SIM: break;  //sai do enquanto
    2.3 Verifica se a primeira instrução de DD está em UD
        - SIM: DD  ← DD - {primeira instrução de DD};
               Executa Passo 2.3
        - NÃO: UD  ← UD ⋃{primeira instrução de SD};
               DD  ← DD - {primeira instrução de DD};
    2.4 Verifica se UD está completo
        - SIM: break; //sai do enquanto
    2.5 Executa Passo 2.1
  3 Fim do enquanto
  4 Retorna UD
}
```

Figura 3.12: Algoritmo de construção de um dicionário unificado

dinâmicas ou estáticas e assim explorar completamente o espaço de construção de dicionários unificados. Para tanto uma nova versão deste algoritmo foi proposta. Nele está incluída uma métrica definida como fator dinâmico, *f*. O fator dinâmico é uma porcentagem de instruções que efetivamente vêm do dicionário dinâmico. A Equação 3.1 define *f*.

$$f = \frac{\text{Instruções } de\ UD\ provenientes\ de\ DD}{Tamanho\ de\ DD} \times 100\% \tag{3.1}$$

3.5.1 Exemplo de Funcionamento do Algoritmo

Um valor de $f = 20\%$ significa que, no mínimo, 20% das instruções do dicionário dinâmico serão incluídas no dicionário unificado. A forma de intercalação das instruções, porém, continua a mesma. Isto porque as figuras 3.4, 3.7 e 3.8 mostram que as primeiras instruções de um dicionário praticamente definem a redundância de instruções em um código. Então, no algoritmo de unificação é apenas inserido um limiar para a quantidade de instruções que efetivamente é retirada do dicionário dinâmico para compor o dicionário unificado.

Um detalhe adicional neste algoritmo diz respeito à quantidade de instruções dinâmicas

presentes em UD. Dado que existe uma interseção, mesmo que pequena, entre SD e DD o f pode apenas garantir um piso para esta quantidade. Entretanto, para dicionários pequenos, a redundância de instruções também é muito pequena, isto implica que f define praticamente o quão orientado para aspectos dinâmicos está o dicionário unificado.

Vamos mostrar aqui um exemplo de funcionamento do algoritmo estendido de unificação dos dicionários. Tomemos dois dicionários de 16 entradas cada (um estático e outro dinâmico) e suponhamos um valor de $f = 20\%$. Isto implica que 3 ($\lfloor 16 \times 20\% \rfloor$) instruções serão inseridas em UD a partir de DD. A Figura 3.13 mostra os passos do algoritmo: Inicialmente A de SD é colocado em UD (a). A primeira instrução de DD também é A e, portanto, não é inserida em UD. A segunda instrução é B e pode ser inserida em UD (b). Conta-se uma instrução advinda de DD. M de SD é colocada em UD (c) bem como C de DD (d). Conta-se duas instruções de DD. A partir de (f), quando 3 instruções de DD já foram usadas, somente instruções de SD vão para UD.

Algumas condições de contorno também são necessárias: no caso onde $f = 100\%$ nenhuma instrução de SD é inserida voluntariamente em UD; para um valor de $f > 50\%$ o limitador se aplica ao SD e não ao DD; e no caso das instruções de SD ou DD forem insuficientes para preencher UD o dicionário que ainda tem possibilidades de inserir alguma instrução volta a ser utilizado[3]. É importante notar que para dicionários completos o uso do método *Multi-Profile* só faz sentido se houver uma necessidade de ordenação para fins de compressão.

[3]Este caso só ocorre se o tamanho de SD ou DD for menor que UD, o que não é um caso comum

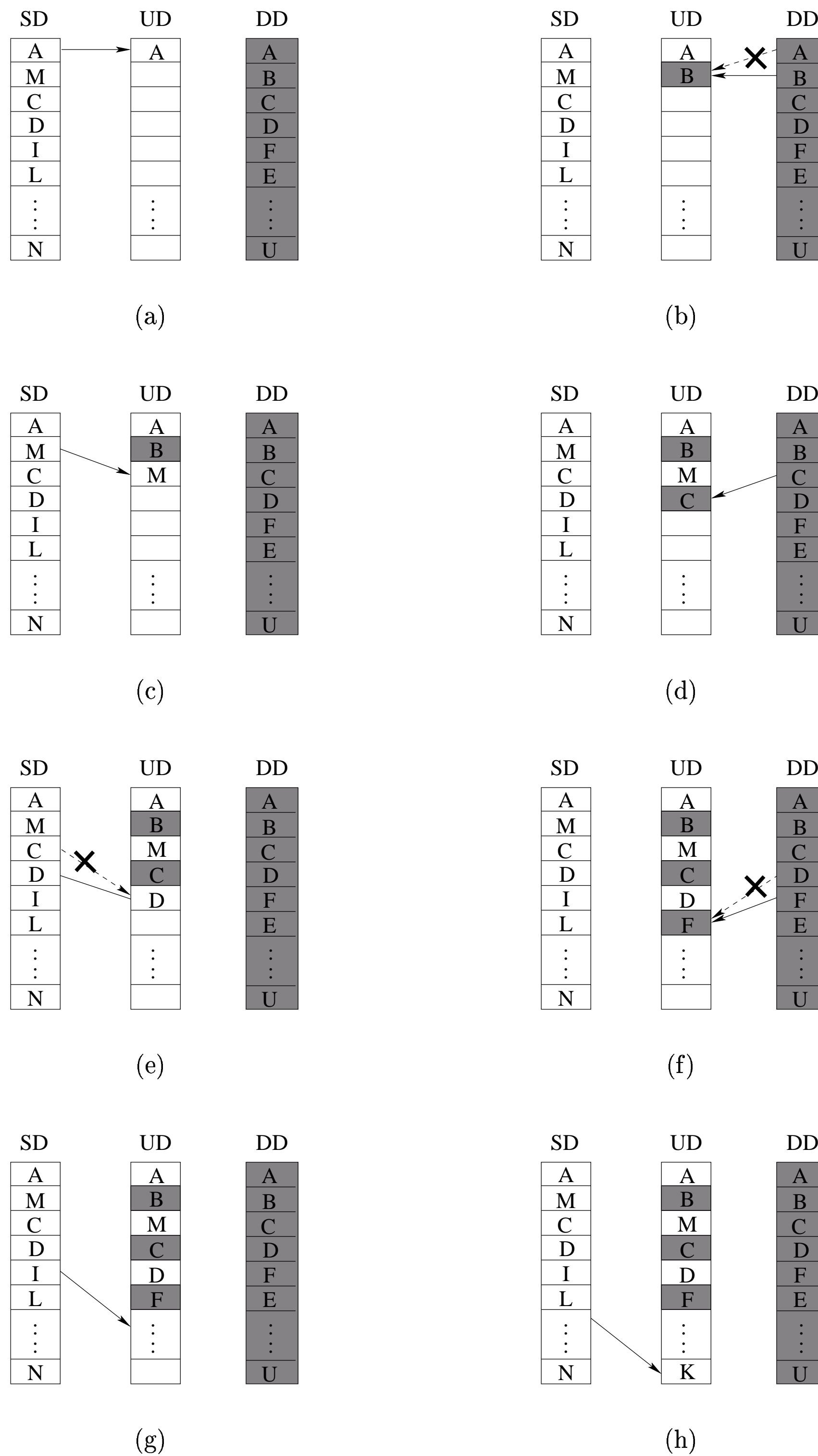


Figura 3.13: Exemplo de funcionamento do algoritmo de unificação dos dicionários

3.6 Considerações Finais e Conclusões

Algumas questões ainda permeiam a construção dos dicionários. A primeira delas é sobre o uso de uma instrução como símbolo. De fato, muitos algoritmos usam símbolos maiores como duplas, triplas, quádruplas de instruções [57, 46, 51] ou árvores de expressão [40, 41]. Neste caso, os dicionários teriam de apresentar uma estrutura capaz de determinar o fim de uma seqüência de entrada. Um outro fator a considerar é o tamanho médio destas seqüências: no caso de árvores de expressão, o tamanho médio de uma árvore é de 1,2 instruções [8] (sem manipular a ordem das instruções no código original) e no caso de dicionários que admitem entradas com mais de uma instrução, entre 50% e 80% das entradas são formadas por apenas uma instrução [88]. Isto implica em admitir um hardware descompressor mais complexo, mas com uso minoritário. Além disto, pode haver redundância dentro do próprio dicionário, já que uma mesma instrução pode participar de diversas seqüências.

No anverso desta discussão há métodos que buscam utilizar símbolos menores como bytes, halfwords ou operandos das instruções [38, 3, 33]. Neste caso a implicação premente está na complexidade do hardware descompressor. Para descomprimir uma única instrução é preciso acesso a múltiplos dicionários e uma forma de anexar as partes para formar um todo.

A simplicidade do descompressor e o seu uso eficaz nos levaram a escolher uma instrução como símbolo. Esta escolha justifica o uso indiscriminado dos termos "entradas" ou "número de instruções" ou "tamanho do dicionário" nas seções anteriores.

Quanto à formação dos dicionários dinâmicos também cabem algumas ressalvas. A primeira delas é sobre o método de construção de dicionários para entradas heterogêneas. Embora não explorado neste trabalho (e em nenhum outro da literatura), para determinar as instruções que mais são executadas em um código sujeito a diversos formatos de entrada é preciso ponderar as contagens individuais, executando o programa diversas vezes, submetido a diferentes cargas de dados. Com isto seria possível determinar a influência dos dados na construção de um dicionário em um nível diferente do apresentado aqui.

Um outro aspecto a ser ponderado é a questão do tamanho dos dados de entrada. A variação dos tamanhos implica na exacerbação da contagem das instruções que mais são executadas. Isto não tem implicação direta na construção de um dicionário, embora possa haver ligeiras alterações nas instruções de ordem intermediária. Em conformidade

com a Figura 3.4 estas instruções não costumam representar uma grande diferença para os métodos de compressão, dado que são relativamente menos preponderantes que as primeiras instruções dos dicionários.

A terceira ponderação sobre a construção de dicionários dinâmicos está restrita ao caso de processadores que podem anular uma instrução dentro do *pipeline*. Este é mesmo o nosso caso particular, onde instruções de salto podem simplesmente ignorar a instrução no seu *delay-slot*. Entretanto, o efeito desta nulidade se aplica somente à unidade de execução e as seguintes. Do ponto de vista da memória, a instrução é lida regularmente. então, quando nos referimos a instruções executadas, para efeito de compressão, queremos dizer instruções enviadas ao *pipeline*. De qualquer forma, o número de instruções que são anuladas em média é de 1,9% das instruções que são retiradas da memória, para o nosso conjunto de aplicações (valor medido em nosso simulador).

A última consideração geral deste Capítulo refere-se ao uso de dicionários semi-adaptativos para compressão de código. Intuitivamente podemos alegar que um método que obtém dados estatísticos de uma aplicação em particular para comprimi-la apresenta melhores resultados que um método que busca um conjunto de instruções genérico que pode ser aplicado a qualquer programa indiscriminadamente. Por outro lado, o argumento de que os compiladores geram estruturas muito padronizadas levaram a construção de algumas alternativas para a indústria [67, 69] com dicionários estáticos (no sentido clássico da nomenclatura).

Para justificar o uso de dicionários semi-adaptativos neste trabalho mostramos duas tabelas que apontam a similaridade dos dicionários entre as aplicações. Cada matriz deve ser lida como contendo duas partes: os dicionários estáticos sao apresentados na porção triangular superior das matrizes e os dinâmicos na porção triangular inferior. Naturalmente, para dicionários completos a matriz é simetrica. A porcentagem de instruções em comum foi calculada com relação ao maior dicionário. A Tabela 3.3 mostra que para dicionários completos, sejam estáticos ou dinâmicos, salvos raros casos, menos de 50% das instruções são comuns. Por exemplo, para um dicionário estático, 33% das instruções são comuns entre o *cjpeg* e o *pegwit*. Quando os dicionários são pequenos, aqui apresentados os valores para um dicionário de 256 instruções (Tabela 3.4), as similaridades, via de regra, continuam a ser menor que 50%. Interessante perceber que para dicionários dinâmicos pequenos estas diferenças se acentuam consideravelmente.

Por causa desta similaridade tão minoritária, o ideal é produzir um dicionário para cada programa (dicionário semi-adaptativo).

	susan	*cjpeg*	*djpeg*	*search*	*dijkstra*	*pegwit*	*adpcm_e*	*adpcm_d*	
susan	–	30%	26%	36%	35%	40%	7%	7%	E
cjpeg	30%	–	48%	28%	26%	33%	6%	6%	s
djpeg	26%	48%	–	24%	23%	28%	5%	5%	t
search	36%	28%	24%	–	76%	41%	14%	14%	á
dijkstra	35%	26%	23%	76%	–	40%	16%	16%	t
pegwit	40%	33%	28%	41%	40%	–	7%	7%	i
adpcm_e	7%	6%	5%	14%	16%	7%	–	100%	c
adpcm_d	7%	6%	5%	14%	16%	7%	100%	–	os
	Dinâmicos								

Tabela 3.3: Similiaridade entre os dicionários completos dos diversos *benchmarks*

	susan	*cjpeg*	*djpeg*	*search*	*dijkstra*	*pegwit*	*adpcm_e*	*adpcm_d*	
susan	–	42%	42%	44%	46%	48%	16%	16%	E
cjpeg	8%	–	66%	48%	46%	48%	19%	19%	s
djpeg	2%	8%	–	44%	42%	43%	17%	17%	t
search	34%	10%	2%	–	79%	51%	21%	21%	á
dijkstra	8%	8%	4%	12%	–	52%	23%	23%	t
pegwit	7%	6%	5%	7%	7%	–	17%	17%	i
adpcm_e	8%	6%	1%	8%	7%	4%	–	100%	c
adpcm_d	10%	5%	1%	11%	7%	4%	57%	–	os
	Dinâmicos								

Tabela 3.4: Similiaridade entre os dicionários pequenos (256 entradas) dos diversos *benchmarks*

A título de curiosidade, alguns métodos de compressão também foram testados em arquiteturas CISC. Notadamente, as características de redundância de instruções, estudas neste Capítulo para construção de dicionários, acontecem também para esta classe de processadores. Não obstante, mais instruções são necessárias para cobrir a mesma fração dos programas em uma comparação direta com arquiteturas RISC [40]. Isto sugere que é mais difícil comprimir código para as arquiteturas CISC que para as RISC. Os resultados apresentados nestes trabalhos corroboram a afirmação acima. Também é sabido que arquiteturas RISC costumam possuir códigos maiores que os seus pares CISC [2], por isto carecem mais de métodos de compressão, sendo este o nosso enfoque.

3.6.1 Conclusões

Neste capítulo apresentamos um estudo sobre a construção de dicionários de instruções para uso em compressão de código. Percebemos que as instruções que compõem um dicionário formado a partir de informações estatísticas de *profiling* estático são consideravelmente distintas das instruções que compõem um dicionário formado a partir de informações de *profiling* dinâmico (para o caso de pequenos dicionários de até 256 entradas). Ainda, a contribuição percentual, na construção do código ou na execução do mesmo, das instruções que compõem um dicionário é muito distinta, sendo poucas instruções responsáveis pelo alto grau de redundância nos programas.

Finalmente, propomos um novo método de construção de dicionários baseado em informações de *profiling* dinâmico e estático ao mesmo tempo. Nos capítulos seguintes usaremos este método, com as devidas adaptações, para comprimir código usando dois algoritmos de compressão desenvolvidos em nossa base de pesquisa.

Capítulo 4

O Método CDM-IBC *Multi-Profile*

O método de construção de dicionários *multi-profile* representa uma forma de aproveitar as vantagens da compressão baseada em *profile* estático e da compressão baseada em *profile* dinâmico ao mesmo tempo.

O método CDM-IBC, ou simplesmente IBC [8], foi projetado seguindo a abordagem dos experimentos com árvores de expressão[40, 41]. Ele usa dicionários completos, mas separados em classes de tal forma a atribuir menores *codewords* a instruções que mais ocorrem. Neste Capítulo vamos mostrar detalhes do funcionamento do IBC e como adaptar a construção de um dicionário unificado para utilização neste método de compressão.

4.1 Análise do Método IBC

A idéia no método IBC é dividir em classes as instruções do dicionário (completo ordenado) de tal forma a atribuir *codewords* de tamanhos distintos para cada uma das classes. A divisão em classes é a tarefa mais delicada do método. Além dela há a construção de uma tabela de tradução de endereços para converter os endereços convencionais para os respectivos endereços das instruções comprimidas e finalmente a geração do código comprimido em si. Nas seções seguintes vamos detalhar cada uma destas fases da compressão usadas no IBC.

4.1.1 Divisão em Classes

A divisão em classes tem inspiração nos algoritmos de Huffman para compressão. Cada classe tem igual chance de ocorrer no código, mas como as instruções têm uma diferença considerável de contribuição para formação de um programa (vide Figura 3.4) o número de instruções em cada classe é bastante distinto. Assim sendo, os tamanhos das *codewords* atribuídas às classes são bem diferentes, sendo as menores usadas para endereçar as instruções que mais ocorrem.

O número de classes a serem usadas também pode variar, não obstante, para as medidas realizadas e apresentadas neste Capítulo usamos sempre a opção padrão de 4 classes. Esta opção é justificada porque a melhor razão de compressão (em média) ocorre exatamente quando 4 classes são usadas[8].

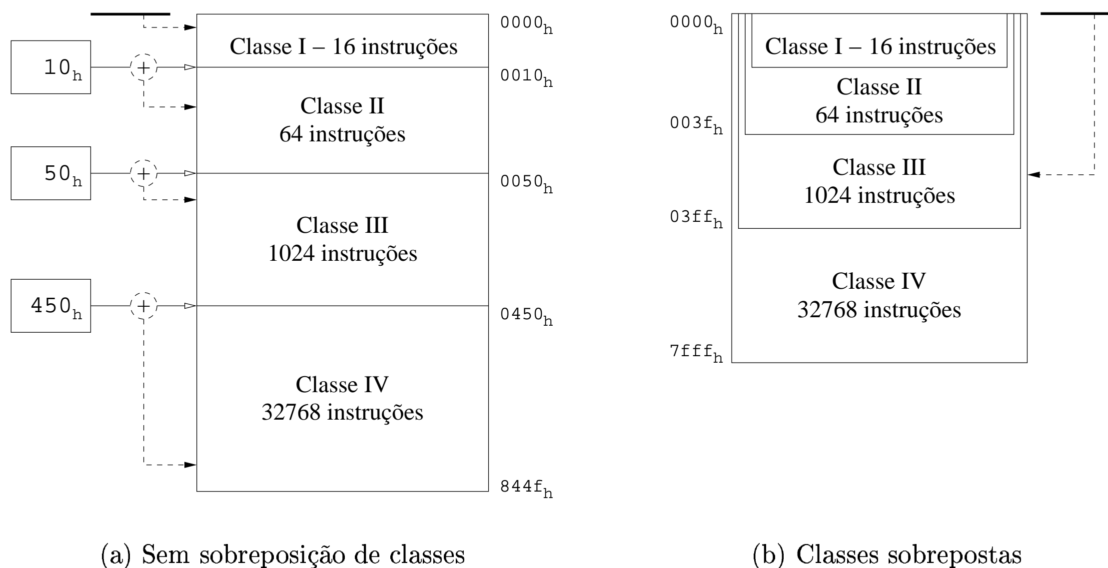
Para elucidar o processo vamos apresentar um exemplo da divisão em 4 classes de um dicionário completo de 30 instruções para um código de 1024 bytes (256 instruções). A Tabela 4.1 mostra o exemplo. As classe devem ter n_i (para i = 1 até 4) instruções que no total contribuam igualitariamente para formação do código. Então, a divisão inicial é que cada classe contribua com 64 ocorrências de instruções do programa. Ora, as instruções A e B são suficientes para totalizar as 64 ocorrências requeridas. A e B são, portanto, atribuídas à classe I. De forma semelhante, C, D, E e F contribuem juntas com 64 ocorrências no código e são atribuídas à classe II. O processo de formação é o mesmo para o restante das classes.

As *codewords* para endereçamento dentro de cada uma das classes (I, II, III, IV) possuem então 1, 2, 3 e 4 bits respectivamente. As instruções que mais contribuem na formação do código são então mapeadas para serem representadas com as menores *codewords*.

Naturalmente não se espera encontrar sempre programas com o número de instruções múltiplo de 4, então a divisão inicial é apenas um limiar a ser almejado. Além disto, nem sempre temos contribuições acumuladas nas classes com um número de instruções que seja potência de 2. Por isto, a escolha do tamanho da *codeword* para cada classe é feita de forma exaustiva e o resultado que apresentar melhor razão de compressão é adotado para ser usado no código comprimido.

Instrução	Ocorrências	Total	Classe
A	38	64	I
B	26		
C	18	64	II
D	16		
E	16		
F	14		
G	10	64	III
H	10		
I	8		
J	8		
K	8		
L	8		
M	6		
N	6		
O	6	64	IV
P	6		
Q	6		
R	6		
S	6		
T	6		
U	6		
V	5		
W	5		
X	4		
Y	3		
Z	1		
α	1		
β	1		
γ	1		
δ	1		

Tabela 4.1: Divisão em classes de um dicionário completo para o método IBC



(a) Sem sobreposição de classes (b) Classes sobrepostas

Figura 4.1: Implementações da IT

O dicionário de instruções do IBC é chamado IT (*Instruction Table*). Na implementação da IT (Figura 4.1(a)) é usado um módulo de memória e para calcular os endereços de cada *codeword* é preciso armazenar os três endereços iniciais de cada classe (II, III e IV) e utilizar um somador. Para evitar estes registradores e o somador foi usado um esquema de sobreposição de classes, como mostra a Figura 4.1(b). Embora a capacidade de armazenamento de cada classe seja reduzida, há um somador a menos no caminho crítico do descompressor o que torna mais eficiente a descompressão e, como todos os tamanhos de classes são testados, quase sempre se consegue reduzir o impacto da diminuição da capacidade das classes na razão de compressão do código.

4.1.2 Tabela de Conversão de Endereços

A tabela de conversão de endereços (ATT) tem como objetivo receber um endereço normal da execução do programa e encontrar um endereço correspondente dentro do código comprimido. Quando ocorrer a compressão do código, as *codewords* são justapostas para formar o novo código[1], portanto uma instrução comprimida pode começar em qualquer dos 32 bits da palavra de memória. Isto requer que a saída da ATT disponha de um *offset*

[1]Entre as *codewords* existe um prefixo para indicar a classe a que ela pertence. Isto será abordado na próxima seção

para encontrar o bit onde se inicia a instrução comprimida correspondente.

Uma consulta à ATT ocorre sempre que há uma falha na cache, então um mapa inicial de todos os endereços de início de linha de cache é necessário. Além disto existe uma parcela do endereçamento original que nunca é alterada (os bits mais altos). Esta parcela pode ser ignorada na entrada e o código comprimido correspondente pode ser alocado em outro espaço. Por exemplo: um código original ocupa os endereços $\texttt{40000000}_h$ até $\texttt{400023ac}_h$, portanto os 18 primeiros bits do endereço nunca são alterados. O código comprimido pode ser então remanejado para uma outra área ($\texttt{60000000}_h$ até $\texttt{60000100}_h$). Isto é usado quando há um código parcialmente comprimido e o descompressor observa o endereço para saber se deve interpretar a palavra da memória como código regular ou comprimido.

Um diagrama simplificado da ATT é mostrado na Figura 4.2. Quando um endereço é solicitado ele passa por uma máscara de bits contendo:

- n_u: bits que não serão utilizados
- n_i: bits usados para indexar a tabela de endereços interna à ATT
- n_{cl}: bits que são usados para indexar palavras dentro de uma linha de cache. No exemplo uma linha contém 32 bytes (8 palavras, indexadas por 3 bits).
- n_b: bits utilizados para endereçamento de bytes dentro de uma palavra.
- n_e: bits extras utilizados para diminuir o tamanho da tabela da ATT. Cada bit extra utilizado divide o tamanho da tabela ATT pela metade. No exemplo, onde $n_e = 1$ bit, para cada 2 linhas da cache apenas 1 endereço correspondente é mapeado na ATT. A linha de cache que tem um endereço inicial mapeado na ATT é chamada de linha de cache base. Quando um acesso à memória busca um endereço inicial de uma linha de cache não base, o descompressor busca seqüencialmente na memória os endereços a partir da última linha de cache base. Naturalmente há um compromisso entre a razão de compressão e o impacto no desempenho do sistema quando escolhemos a quantidade de bits para compor n_e.

Em seguida os n_i bits significativos apontam para a posição da tabela da ATT onde está o endereço comprimido correspondente. Apenas os bits que não são fixos são armazenados nesta tabela. No exemplo anterior, quando o código comprimido ocupava as posições $\texttt{60000000}_h$ a $\texttt{60000100}_h$ apenas 10 bits (os 12 menos significativos, menos os 2 bits de

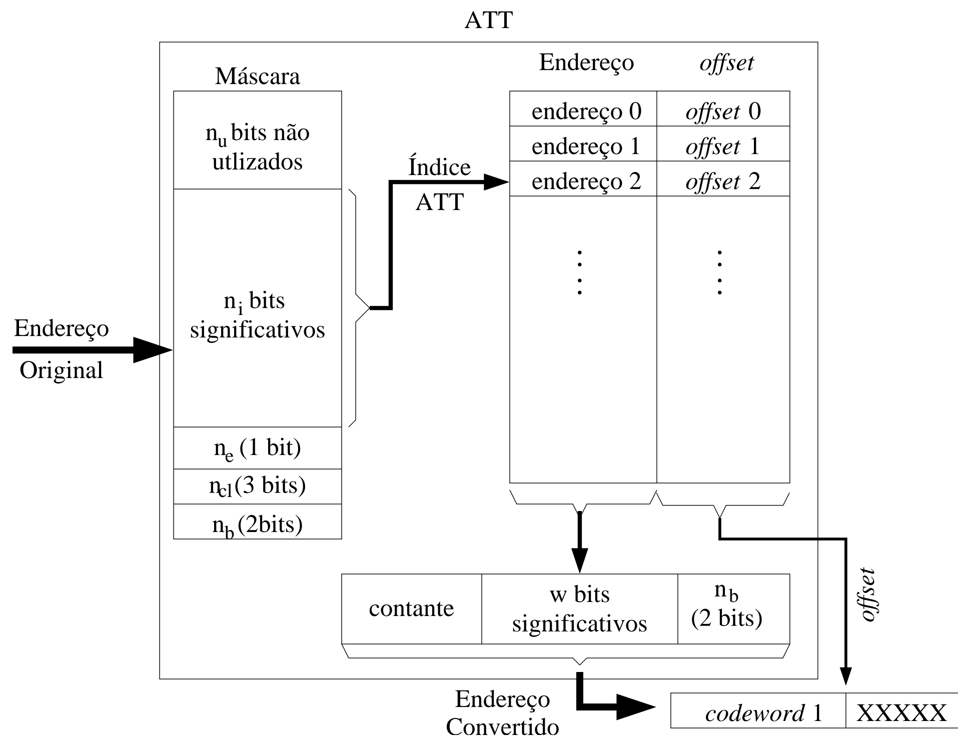


Figura 4.2: Diagrama de blocos da ATT

endereçamento de bytes) são necessário para armazenagem no campo Endereço da tabela. O campo *offset* possui 5 bits e serve para endereçar um dos 32 bits onde se inicia a *codeword* buscada. Finalmente são anexados a constante inicial e os dois bits de endereçamento de bytes (00_2 por *default*).

4.1.3 Geração do Código Comprimido

Uma vez escolhidos os tamanhos das *codewords* é preciso identificar a classe a que ela pertence. No método usa-se um prefixo de tamanho fixo para esta identificação. No caso de 4 classes, dois bits são utilizados. A atribuição dos bits da *codeword* é seqüencial e indiferente para o método. Um exemplo da geração do código comprimido especificado na Tabela 4.2 pode ser visto na Figura 4.3.

Os pares prefixo/*codewords* são inseridos na memória a partir do bit 0 do primeiro endereço comprimido e justapostos de tal forma a obter a maior compressão possível.

Endereço	Instrução	*codeword*	Prefixo
40000000h	sethi 281, %o0	001011101110_2	10_2
40000004h	or %o0, 184, %o0	010111110101_2	01_2
40000008h	call 8	$01000000000000000000000000100_2$	11_2
4000000ch	nop	001110_2	00_2

Tabela 4.2: Exemplo de Código a ser Comprimido

... Prefixo 2 *codeword* 1 Prefixo 1 *codeword* 0 Prefixo 0
60000000 h: 1 0 0 1 1 0 1 0 1 1 1 0 1 1 0 1 0 1 0 0 1 0 1 1 1 0 1 1 1 0 1 0
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

...*codeword* 3 Prefixo 3 *codeword* 2...
60000004 h: 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

60000008 h: X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X X 0
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Figura 4.3: Conteúdo da memória para um trecho do código comprimido pelo IBC

4.2 Hardware Descompressor do IBC

O hardware do descompressor do IBC foi projetado e prototipado em um FPGA Virtex (XCV800) para uma arquitetura SPARCv8[2]. O descompressor possui 4 estágios de *pipeline* descritos a seguir:

- *Input*: estágio responsável por decodificar os endereços solicitados pelo processador e efetuar a conversão entre endereço não comprimido e comprimido.
- *Fetch*: estágio responsável pela leitura das instruções comprimidas na memória.
- *Decode*: neste estágio as palavras lidas na memória são agrupadas e as *codewords* são decodificadas.
- *Output*: finalmente, o estágio de saída é o responsável por fornecer ao processador (à cache, na verdade) a instrução descomprimida.

Um diagrama simplificado do *pipeline* do IBC é mostrado na Figura 4.4. Quando o processador (a cache especificamente) pede uma instrução no endereço *addr*, a ATT é acionada para encontrar o endereço da linha da cache base e o *offset* da primeira instrução

[2] Também há um descompressor para MIPS, mas este não foi prototipado, apenas simulado em VHDL

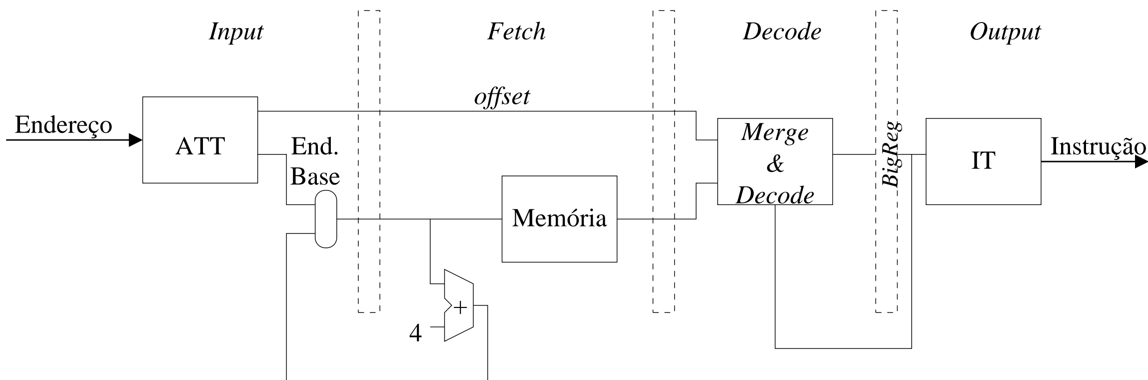


Figura 4.4: *Pipeline* do descompressor IBC para o Leon

comprimida daquela linha (*Input*). O endereço base é enviado para memória e o próximo endereço é calculado (*Fetch*). O *offset* é usado para fazer um deslocamento para direita da palavra carregada da memória, de tal forma que os dois bits menos significativos sejam sempre um prefixo (*Decode*). A palavra é armazenada em *BigReg*. A *codeword* é usada para endereçar a IT e fornecer a palavra comprimida (*Output*). Ao mesmo tempo o estágio *Decode* já desloca a palavra de memória lida anteriormente para que o próximo par prefixo/*codeword* se inicie novamente no bit 0 do *BigReg*. O processador pede então a próxima instrução. Como ela já está no *BigReg*, o descompressor faz mais um acesso à IT e entrega imediatamente a instrução descomprimida. Quando for detectada a necessidade de mais um acesso à memória para completar a descompressão de uma linha, o próximo endereço, calculado no estágio *Fetch*, é utilizado. Se restaram bits da palavra anterior estes são anexados (*merged*) à nova palavra comprimida e entregues ao *BigReg*.

A versão detalhada do funcionamento do descompressor pode ser encontrada em [8].

4.3 IBC *Multi-Profile*

O IBC é um método que usa dicionários completos[3], mas que exige uma ordenação prévia para divisão em classes das instruções. Por este motivo e pela disponibilidade imediata da plataforma completa nós utilizamos o IBC para mostrar como o método de construção de dicionários *multi-profile* pode ser aplicado.

O método IBC produz ainda a melhor razão de compressão de código para MIPS e uma

[3] O compressor do IBC admite especificar o tamanho da IT, assim limitando o dicionário, mas não o utilizamos

das melhores para SPARC (*circa* maio de 2004), não obstante, não foi exaustivamente testado quanto às características de desempenho. A priori, o número de ciclos para obter uma nova linha de cache da memória aumenta consideravelmente, dado que antes de iniciar a entrega da primeira instrução à cache, 4 ciclos são usados para preencher o *pipeline* do descompressor. Esta perda, porém, tem um contrapeso: o número de acessos à memória principal pode ser reduzido porque cada palavra da memória pode conter um conjunto de mais de uma instrução comprimida. No caso de memórias lentas, onde vários ciclos são necessários para leitura de uma palavra, as perdas devido ao transiente inicial do *pipeline* do descompressor podem ser completamente eliminadas.

Para melhorar o desempenho, ou diminuir as perdas devido à presença do descompressor, é conveniente atribuir as menores *codewords* não às instruções que mais ocorrem no código, mas àquelas que efetivamente são mais executadas (lidas da memória). O algoritmo de divisão das classes do dicionário precisa das informações dinâmicas do uso de instruções para encontrar a melhor distribuição. Naturalmente espera-se que haja um preço a ser pago por esta abordagem, principalmente porque já sabemos que as primeiras instruções do dicionário dinâmico não são muito parecidas com as primeiras do dicionário estático e, portanto interferem decisivamente na construção das primeiras classes.

Um outro detalhe sobre o desempenho do IBC, em termo de ciclos, refere-se ao tamanho das caches utilizadas, ou melhor, à sua taxa de falha. Quanto melhor o desempenho da cache, menos o descompressor será utilizado e menor é a interferência no desempenho do sistema original.

A abordagem para adaptação do IBC ao dicionário *multi-profile* é certamente mais complexa no que se refere à divisão em classes. Uma vez escolhida a nova seqüência das instruções no dicionário unificado, é preciso cuidar da análise quantitativa que o compressor usa. Se simplesmente mantivéssemos a idéia original de contar as ocorrências de cada instrução para divisão das classes, criaríamos classes maiores no inicio do dicionário (porque algumas instruções que são muito executadas não ocorrem com muita freqüência no código) penalizando a razão de compressão. Se por outro lado, usássemos a freqüência de execução das instruções para divisão das classes, estaríamos atribuindo *codewords* menores a algumas instruções advindas do dicionário estático, que não são executadas com tanta freqüência, embora sejam alocadas no início do dicionário unificado por causa de sua ocorrência no código, desta forma penalizando a execução do código.

Para resolver este impasse, a solução adotada foi dissociar a ordem do dicionário e a contribuição de cada instrução. Para tanto foi aplicada uma média ponderada

da contribuição estática e dinâmica usando o valor de f para multiplicar a contagem dinâmica e o seu complemento (100%–f) para multiplicar a contagem estática. O exemplo hipotético a seguir (Figura 4.5) mostra a nova distribuição do dicionário unificado para ser usado no IBC. Em destaque, quando usamos $f = 0\%$ (baseado em contagem estática (S)) o método segue a premissa original. Por outro lado quando usamos $f = 100\%$, a divisão em classes é feita totalmente baseada na contagem dinâmica das instruções (D). Isto permite atribuir classes pequenas e, portanto, menores *codewords*, a instruções que são mais executadas. Por fim, o valor atribuído a f dita a forma da contribuição de cada instrução no novo dicionário unificado. Forma-se então uma família de curvas para os valores de f ($0\% < f < 100\%$) usados na construção do dicionário unificado. Quanto maior o valor de f, mais parecida com a curva D ($f = 100\%$) fica a nova curva formada. Por outro lado, quando f se aproxima de 0% mais parecida com S ($f = 0\%$) fica a nova curva.

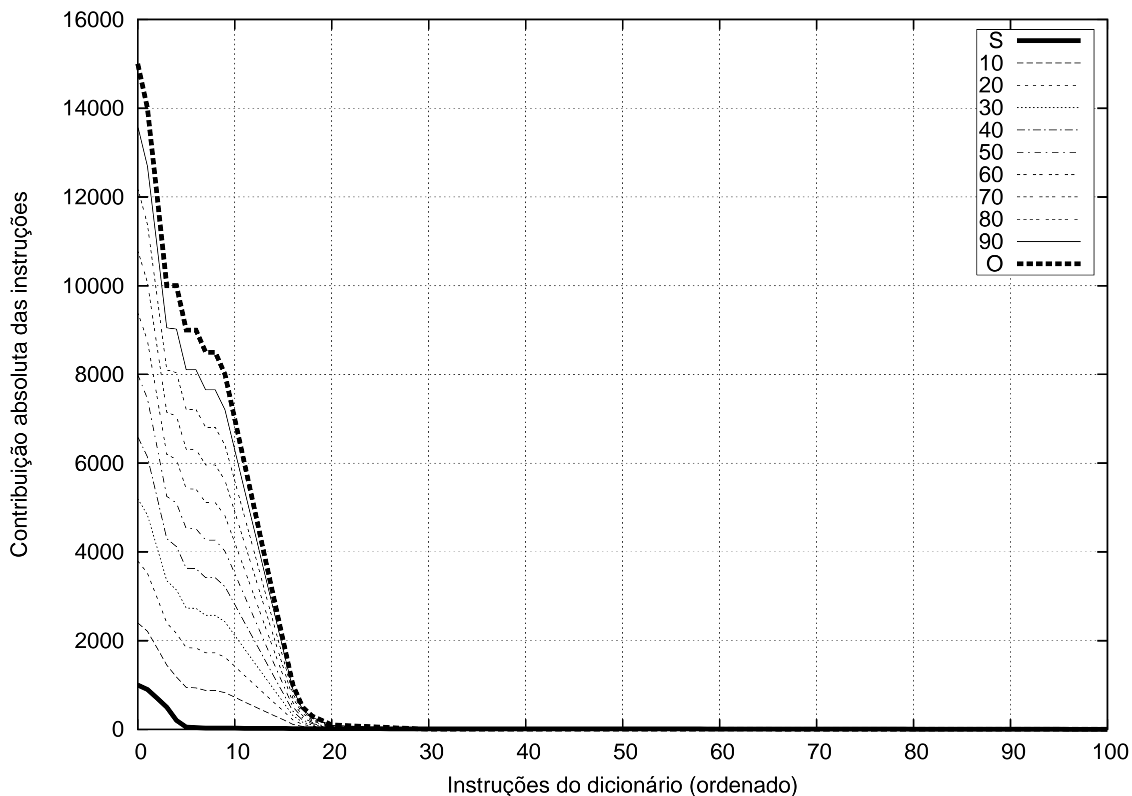


Figura 4.5: Família de curvas para distribuição das contribuições de um dicionário unificado (com f variando de 10% em 10% entre S e D)

4.3.1 Análise da Razão de Compressão do IBC *Multi-Profile*

Uma vez construídos o dicionário unificado e sua distribuição de contribuição a serem usados no IBC fizemos a análise da razão de compressão do método. A razão de compressão total deve incluir não somente o novo tamanho do código comprimido, mas os tamanhos das tabelas usadas no descompressor, a saber, a ATT e a IT. A Figura 4.6 mostra a contribuição das tabelas e do código comprimido na construção da razão de compressão para os casos extremos de $f = 0\%$ (S) e $f = 100\%$(D). A IT tem uma influência considerável (em média 33%) no tamanho final do programa, especialmente nos menores códigos. A ATT por sua vez, representa uma porcentagem fixa de apenas 3% do código comprimido.

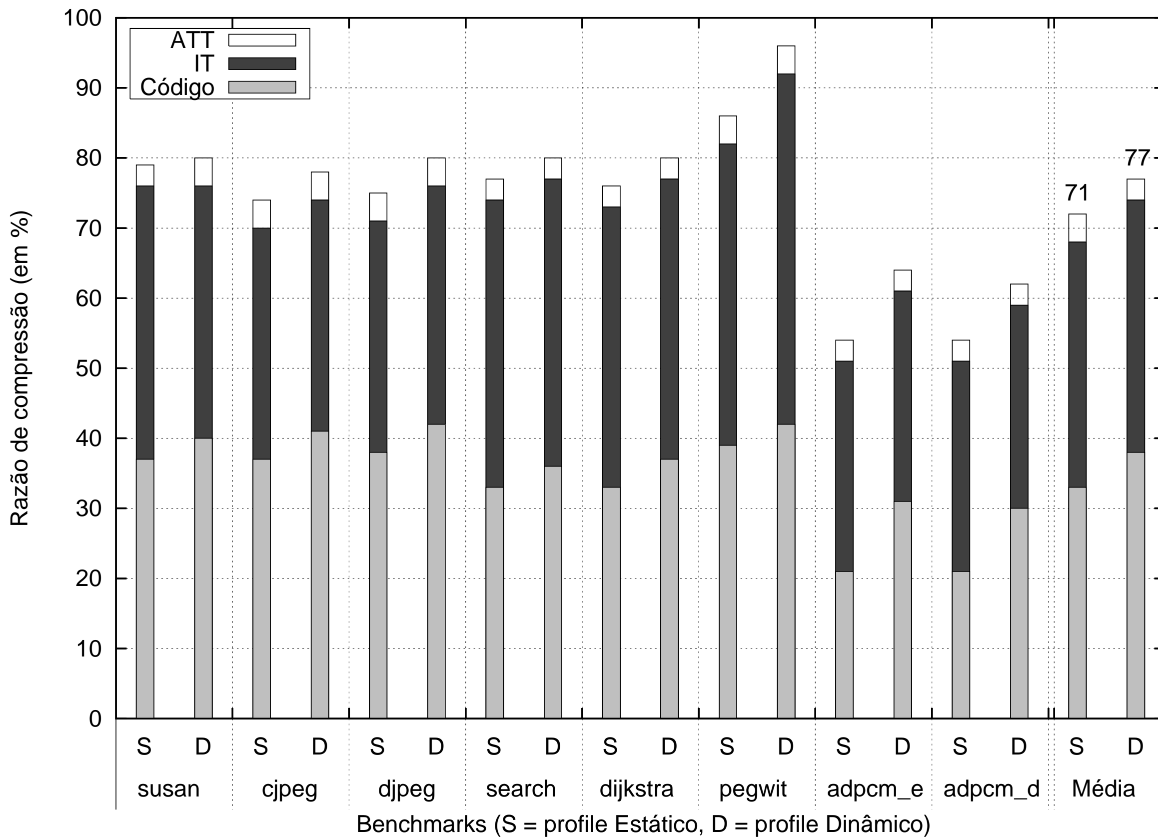


Figura 4.6: Composição da razão de compressão do IBC

Finalmente, a razão de compressão média difere em 6% entre a compressão usando informações estáticas (S) e a compressão usando informações dinâmicas (D). Esta diferença se aplica ao código comprimido, já que a IT e ATT se mantém praticamente do mesmo tamanho (em média).

Já a Figura 4.7 apresenta a exploração da razão de compressão (incluindo as tabelas) completa para todos os valores de f. As alterações na razão de compressão são perceptíveis quando inserimos algumas instruções do dicionário estático no início do dicionário unificado (1% em média de aumento entre $f = 0\%$ e $f = 10\%$) e principalmente, no outro extremo, quando apenas poucas instruções do dicionário estático são usadas no início do dicionário unificado (4% de aumento entre $f = 90\%$ e $f = 100\%$).

Concluímos dos experimentos que usando um valor de f abaixo de 90% não há um prejuízo considerável para razão de compressão.

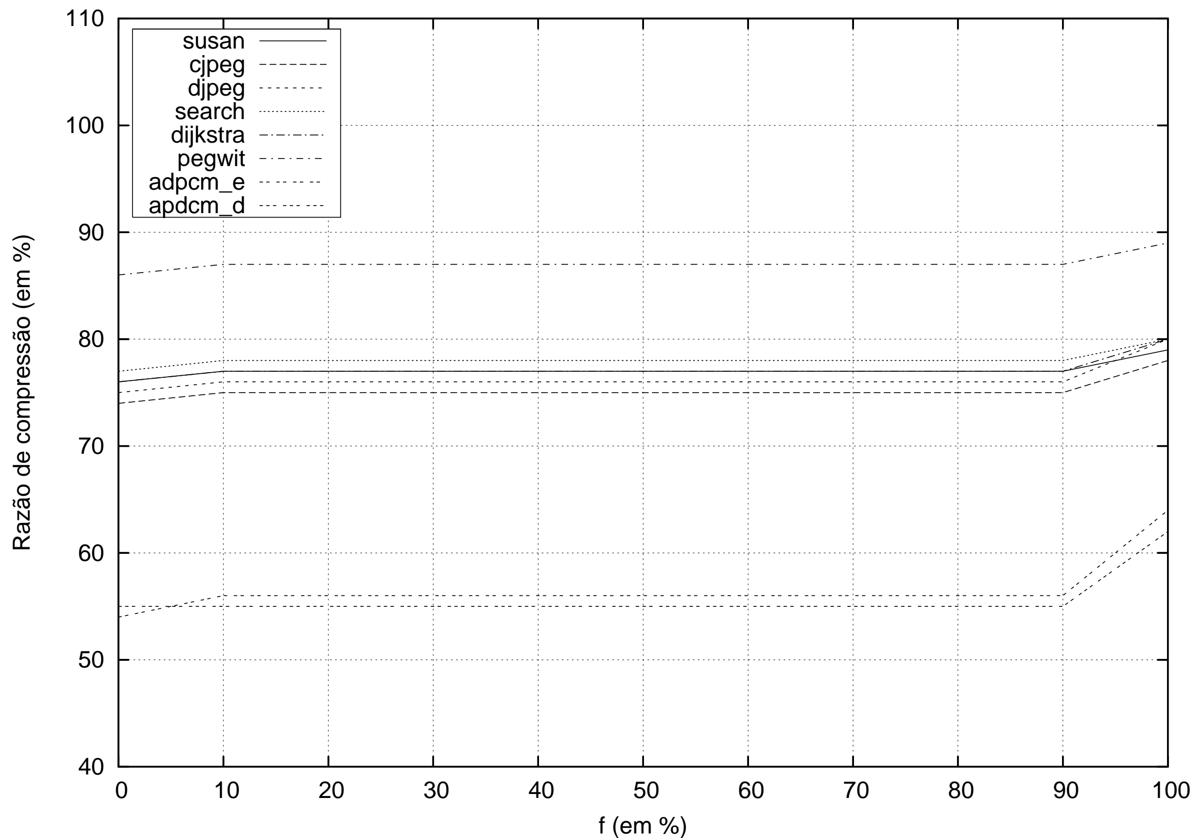


Figura 4.7: Razão de compressão em função de f

4.3.2 Análise de desempenho do IBC

Como já dissemos, a análise de desempenho do IBC foi pouco explorada em [8]. Nesta seção vamos mostrar como o desempenho do IBC é afetado pela presença do descompressor e pela velocidade da memória principal.

A primeira escolha desta análise refere-se ao tamanho da cache a ser utilizada. De fato uma cache que apresente uma taxa de falhas elevada pode comprometer profundamente o desempenho do IBC, já que seu descompressor seria invocado com mais freqüência. O método possui, por *default*, um tamanho de linha de cache de 32 bytes (8 instruções) e por isto variamos apenas os tamanhos das caches de instruções entre 64 bytes e 16Kbytes. Todas são mapeadas diretamente, *write-through* e *no write-allocate*. Embora pequenas, estas caches se mantém representativas dados os tamanhos dos *benchmarks* usados.

A Tabela 4.3 e o gráfico correspondente na Figura 4.8 mostram a taxa de acerto para os diversos tamanhos de caches estudados. A escolha de um tamanho para cada aplicação depende do projeto, mas normalmente se escolhe um valor onde a taxa de acerto seja grande, notadamente maior que 90%, e uma variação para o dobro do tamanho da cache anterior não influencie muito nesta taxa. Escolhemos um limiar em 5% para esta variação.

Duas exceções às regras estipuladas acima foram feitas para o ADPCM e para o *dijkstra*. As caches escolhidas (em destaque na Tabela 4.3) não satisfazem a regra dos 5% e, no caso do ADPCM, nem a regra dos 90%, mas o fato é que depois do tamanho escolhido a próxima taxa de acerto equivale a computar apenas as falhas compulsórias, o que significa praticamente descartar a presença da cache (e do descompressor) durante a execução do código.

	Tamanho das Caches								
	64	128	256	512	1024	2048	4096	8192	16384
susan	78	81	**96**	96	96	97	99	99	99
cjpeg	83	89	**92**	93	94	98	99	99	99
djpeg	83	**92**	93	96	98	99	99	99	99
search	84	**92**	92	92	92	94	97	98	99
dijkstra	84	**92**	99	99	99	99	99	99	99
pegwit	84	**94**	95	96	96	98	99	99	99
adpcm_e	82	**86**	99	99	99	99	99	99	99
adpcm_d	85	**87**	99	99	99	99	99	99	99

Tabela 4.3: Taxa de acerto nas caches (em porcentagem)

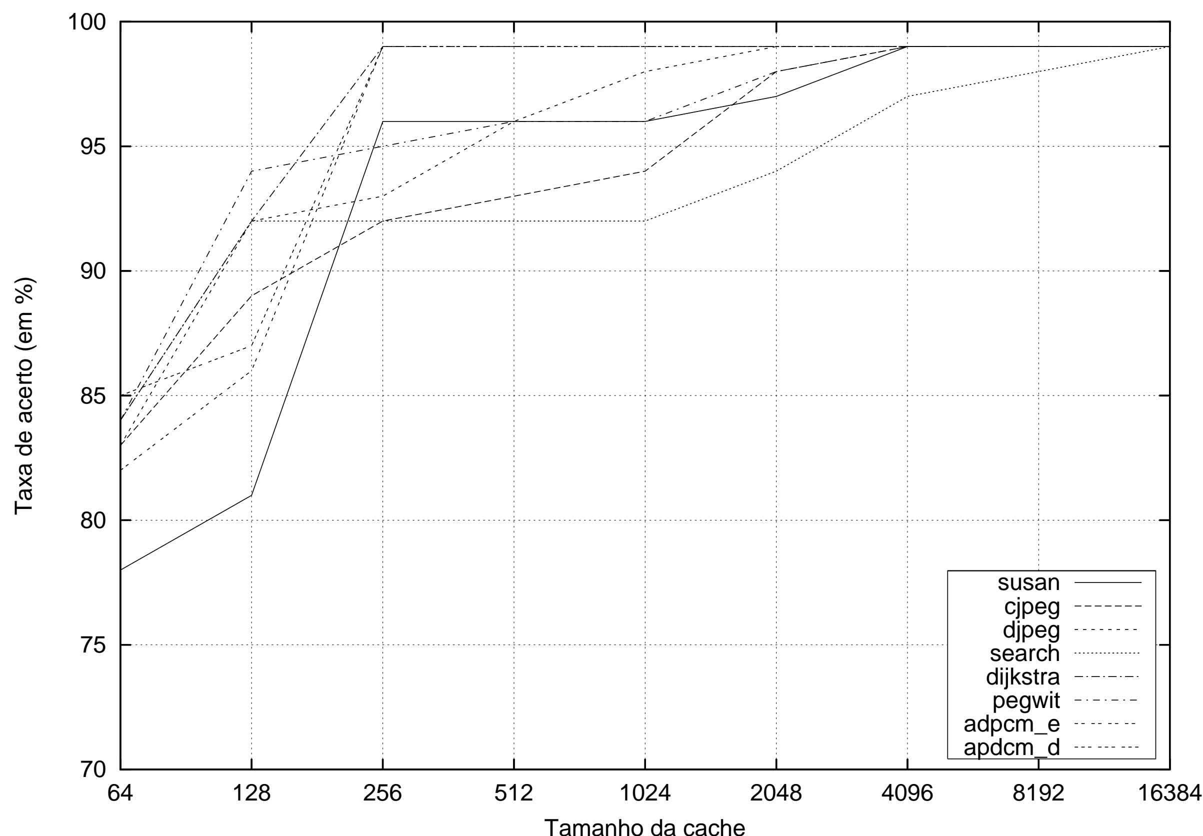


Figura 4.8: Taxa de acerto nas caches para o código original

Também se espera que a quantidade de acessos à memória principal caia com o aumento do valor de f. Naturalmente, quando $f = 100\%$, as menores *codewords* são aquelas que mais se aproximam dos principais caminhos de execução do código. Isto pode ser comprovado na Tabela 4.4 mostrada a seguir. O número de acessos à memória principal decaiu (em média) para 72% do valor original quando o dicionário estático (S) é usado na compressão. Quando o dicionário é baseado somente em informações dinâmicas (D) já se verifica uma queda um pouco mais considerável (para 62% do original em média) no número de acessos à memória. A variação dos percentuais devida ao uso das abordagens estática ou dinâmica também chega a 10% em média.

O número de acessos à memória em função de f é mostrado na Figura 4.9. Percebe-se uma queda acentuada entre $f = 0\%$ e $f = 20\%$ (em média, 7%). Depois, entre $f = 80\%$ e $f = 100\%$ outra queda é observada (agora de 2% em média). Assim sendo, para $f > 20\%$ já se obtém uma redução considerável no número de acessos à memória devido a compressão.

	#Acessos originais	#Acessos Comp. S		#Acessos Comp. D		Variação dos percentuais
		abs	%	abs	%	
susan	40475144	27070474	68,88	22749061	56,21	–10,68%
cjpeg	8546768	6377667	74,62	5542787	64,85	–9,77%
djpeg	2346880	1890574	80,56	1549324	66,02	–14,54%
search	6684720	4341426	64,95	4060580	60,74	–4,20%
dijkstra	31042544	25757673	82,98	20721180	66,75	–16,22%
pegwit	14864744	10552745	70,99	9765736	65,70	–5,29%
adpcm_e	10634688	6794195	63,89	6055402	56,94	–6,95%
apdmc_d	7093016	4874949	68,73	3989679	56,25	–12,48%
		Média	71,10		61,68	–10.02%

Tabela 4.4: Número de acessos à memória principal

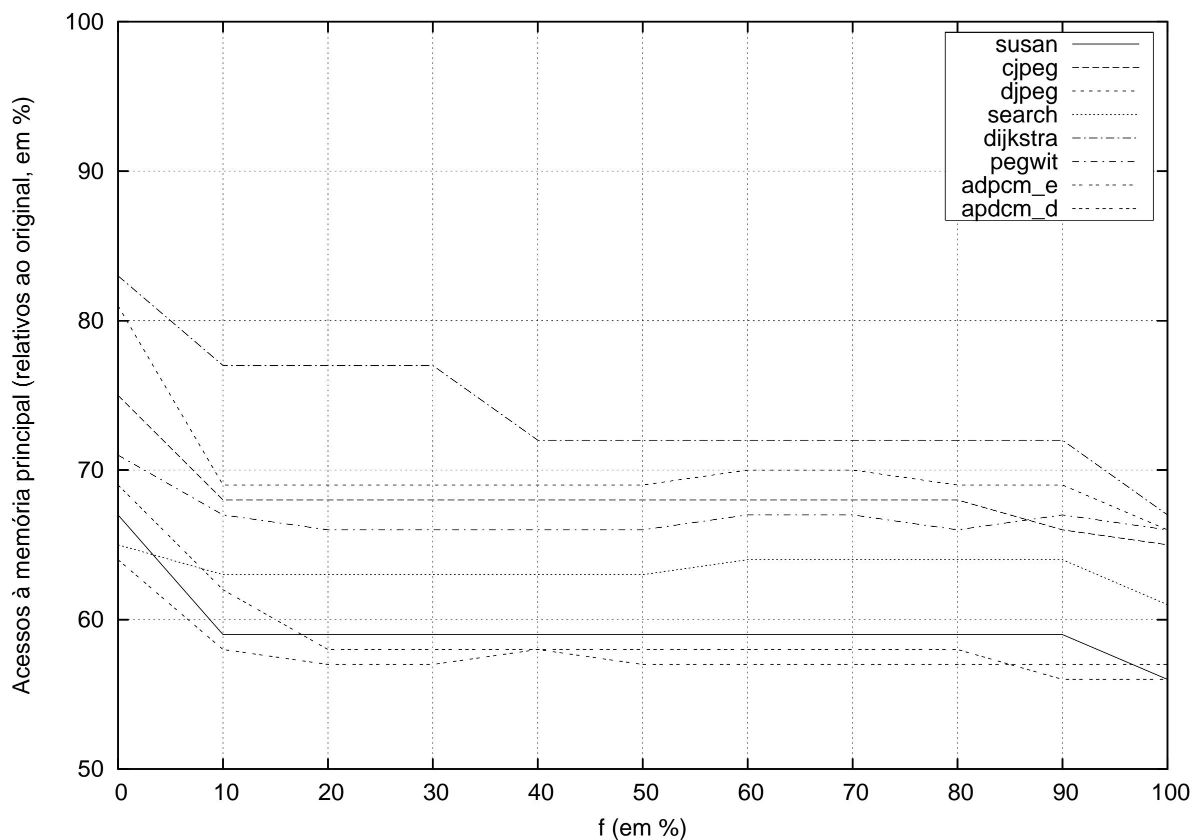


Figura 4.9: Acessos à memória principal em função de f

Esta é uma importante métrica para análise de consumo de energia, já que há um gasto considerável de energia por acesso à memória principal. A energia consumida na memória decairá com a redução no número de acessos.

Finalmente chegamos às medidas de desempenho. A presença do descompressor altera substancialmente o desempenho, permitindo ganhos em alguns casos e perdas em outros. As variáveis envolvidas na contagem de ciclos são:

- o número de acessos à memória principal: quanto menor o número de acessos e menores as *codewords* encontradas durante a execução, melhor será o desempenho;
- a quantidade de linhas de caches não bases, solicitadas durante a execução do código: quanto maior for esta quantidade pior será o desempenho; e
- a velocidade da memória principal: para memórias rápidas a vantagem da redução do número de acessos é diminuída.

O comportamento dos acessos à memória principal já foi mencionado anteriormente. A composição (em linhas de cache base e não base) das falhas da cache está mostrada na Figura 4.10. Há um desbalanceamento considerável nas aplicações *dijkstra* e *adpcm_d* onde muitas linhas de cache não base são acessadas pelo descompressor em relação ao número de linhas base. Todas as outras variam dentro de uma margem de 10% para mais ou para menos.

Por fim abordamos a influência da velocidade da resposta da memória principal. Nossos experimentos buscaram cobrir uma família de memórias com o número de ciclos de resposta variando entre 1 (resposta imediata) e 64. A Figura 4.11 mostra o número de ciclos gastos em cada aplicação relativos ao valor original para 3 destas memórias em função de f.

Para uma memória com resposta imediata (a) a vantagem da diminuição do número de acessos é praticamente descartada pela quantidade de ciclos extras gerados pelo descompressor para preencher a linha da cache, mas para uma memória tão rápida até o uso da cache deve ser questionado. Para as mais lentas ((b) e (c)) o número de ciclos começa a ser fortemente influenciado pela redução no número de acessos.

A influência de f também pode ser percebida na Figura 4.12. À medida que f aumenta, o desempenho do sistema também aumenta (o número de ciclos de *clock* diminui). Para se ter uma idéia geral desta influência, fizemos uma média da redução de ciclos de todas as aplicações para cada velocidade de memória em função de f. À medida que a velocidade

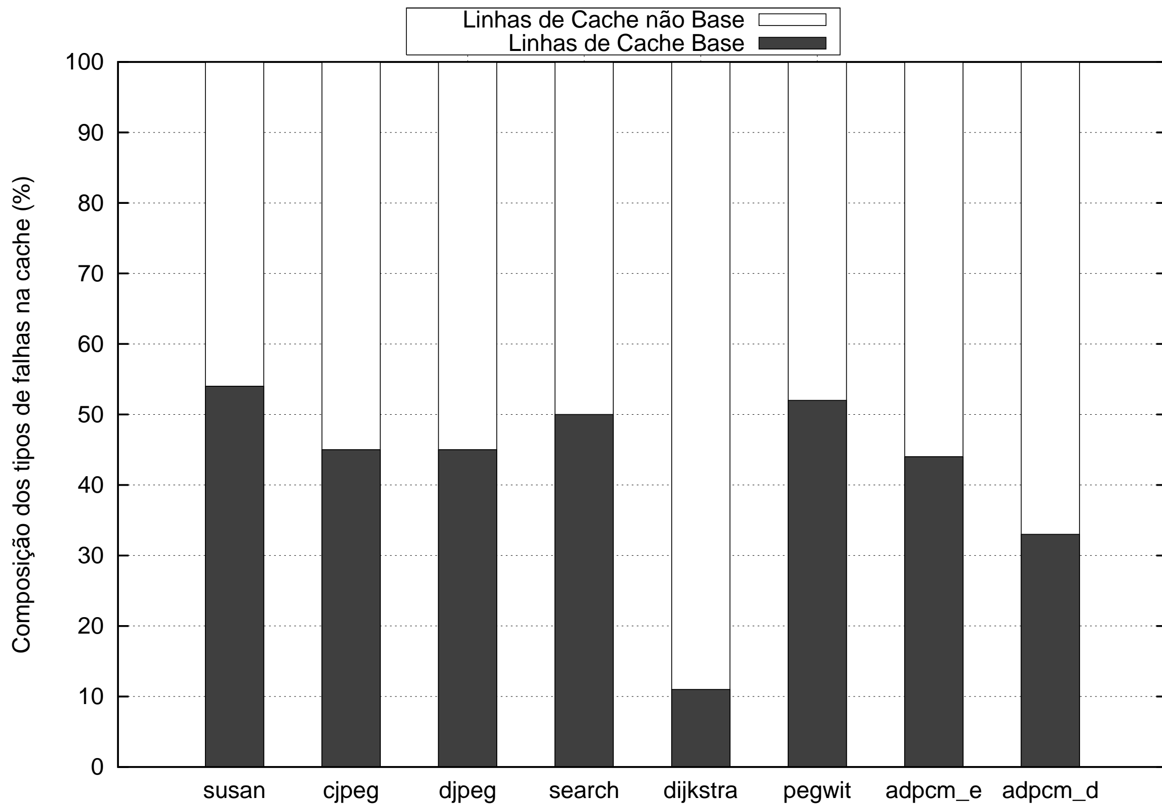
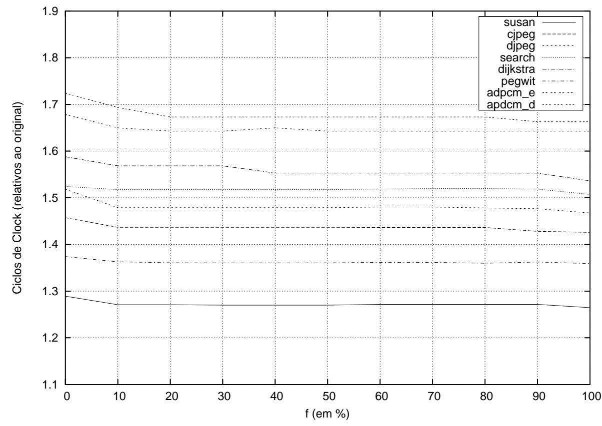


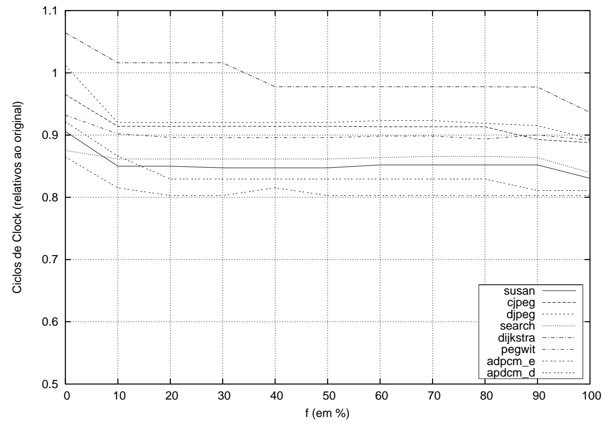
Figura 4.10: Composição das falhas na cache

de resposta da memória principal cai, em média, o desempenho aumenta e mais influente é o valor de f. As curvas vão tomando um formato mais parecido com a da redução de acessos mostrando que este fator é determinante no desempenho quando se usa memórias mais lentas.

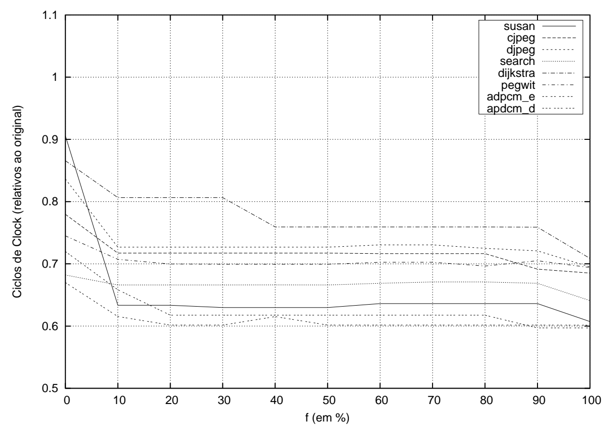
Para a memória mais rápida, o simples fato de escolher $f = 0\%$ ou $f = 100\%$, já produz uma redução de 152% para 148%, ou seja, 4 pontos percentuais, dos quais 3 se observa entre $f = 0\%$ e $f = 20\%$. No caso da memória mais lenta, entre $f = 0\%$ e $f = 100\%$, há uma queda de 75% para 65%, e mais uma vez, entre $f = 0\%$ e $f = 20\%$, 7% desta diferença é consumida. Em suma, para $f = 20\%$ a redução no número de ciclos de *clock* atinge cerca de 70% de seu total e se mantém praticamente inalterada até $f = 90\%$ (variação de 10%). Depois deste valor há mais uma discreta queda de 20%.



(a) Tempo de resposta da memória principal: 1 ciclo



(b) Tempo de resposta da memória principal: 8 ciclos



(c) Tempo de resposta da memória principal: 64 ciclos

Figura 4.11: Ciclos de *clock* utilizados (em relação ao original)

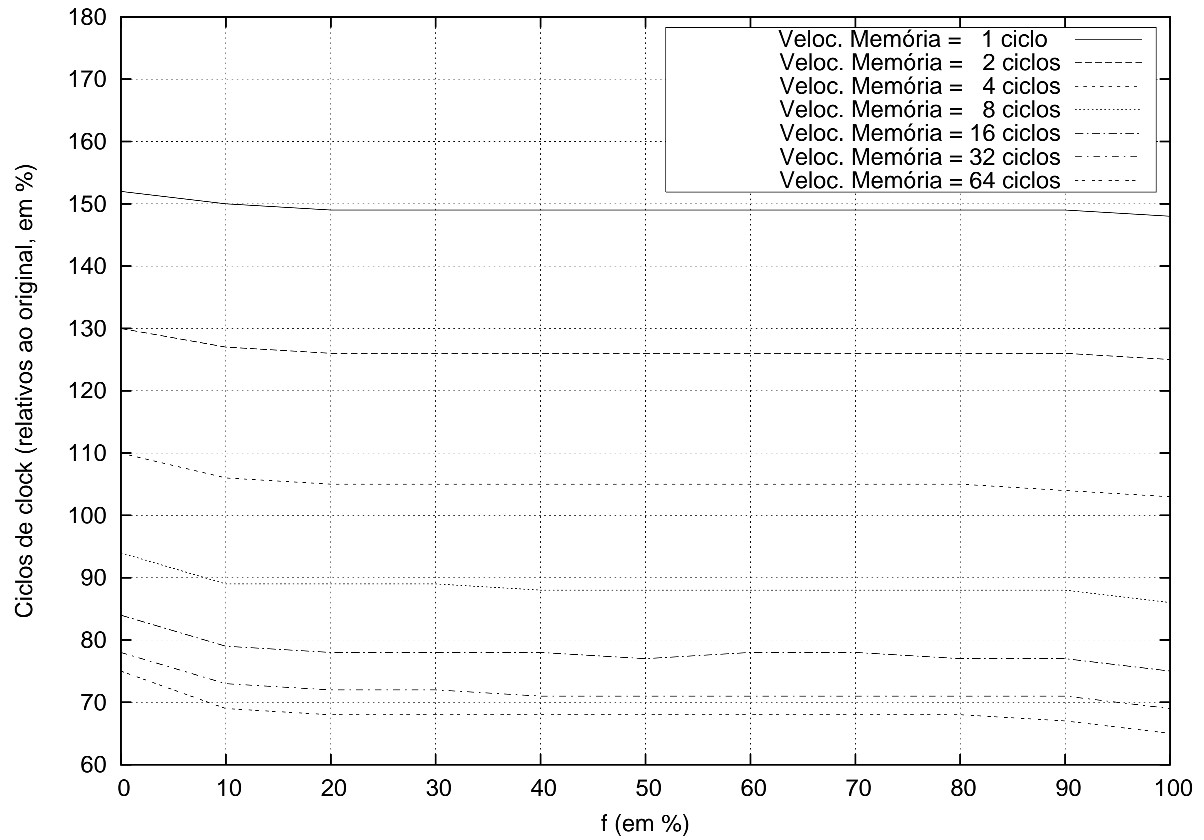


Figura 4.12: Influência do fator dinâmico no desempenho do sistema (para memórias com respostas entre 1 e 64 ciclos)

4.3.3 Influência dos dados de entrada no desempenho do IBC

Como já mencionamos na Seção 3.3, a influência dos dados de entrada na formação do dicionário é pequena. Nós testamos pares de dicionários estáticos, dinâmicos e de um dos dicionários unificados ($f = 50\%$, por ser um valor intermediário) para demonstrar o efeito dos dados de entrada no desempenho como um todo do sistema.

Iniciamos com a razão de compressão. A Tabela 4.5 mostra como as diferentes entradas influenciam este fator. O código comprimido foi gerado a partir das estatísticas obtidas com o conjunto de dados D1, depois experimentamos usar as estatísticas obtidas com D2 para comprimir o mesmo código. A formação dos dicionários influenciou minimamente na razão de compressão final para as diversas aplicações. No pior caso (*pegwit*) a diferença foi de apenas 0,14%, portanto insignificante neste item.

Para avaliar a influência dos dados no desempenho (Figura 4.13) escolhemos uma memória de velocidade intermediária (8 ciclos por acesso a uma palavra) e os mesmo valores de f mencionados anteriormente. Os programas foram comprimidos com base nos dados retirados das estatísticas de execução dos códigos quando submetidos ao conjunto D1 de dados. Para a execução do código comprimido foi utilizado o conjunto D1 e o conjunto D2 como entrada. Para f igual a 0%, 50% e 100%, as variações médias na redução dos ciclos de *clock* foram de 5,3%, 4,9% e 4,5% respectivamente. Isto é, quanto menor a influência de f mais propenso à interferência dos dados de entrada é a compressão, isto porque os dicionário dinâmicos são bastante parecidos (vide Figura 3.5). Para outras velocidades de memória as variações médias totais (média das médias) variam entre 2% (1 ciclo por acesso) e 6% (64 ciclos por acesso).

Neste item chegamos a conclusão que é importante manter nos dicionários algum grau

	f=0%		f=50%		f=100%	
	D1	D2	D1	D2	D1	D2
susan	76,03%	76,03%	77,01%	77,02%	79,52%	79,51%
cjpeg	73,64%	73,64%	74,63%	74,65%	77,52%	77,55%
djpeg	75,10%	75,10%	76,26%	76,28%	79,55%	79,55%
search	77,10%	77,10%	77,97%	78,07%	80,08%	80,05%
dijkstra	75,96%	75,96%	76,76%	76,74%	79,86%	79,85%
pegwit	86,15%	86,15%	86,76%	86,69%	89,08%	89,22%
adpcm_e	53,56%	53,56%	55,59%	55,59%	63,72%	63,73%
adpcm_d	53,56%	53,56%	54,89%	54,91%	62,02%	62,02%

Tabela 4.5: Razão de Compressão usando diferentes dados de entrada

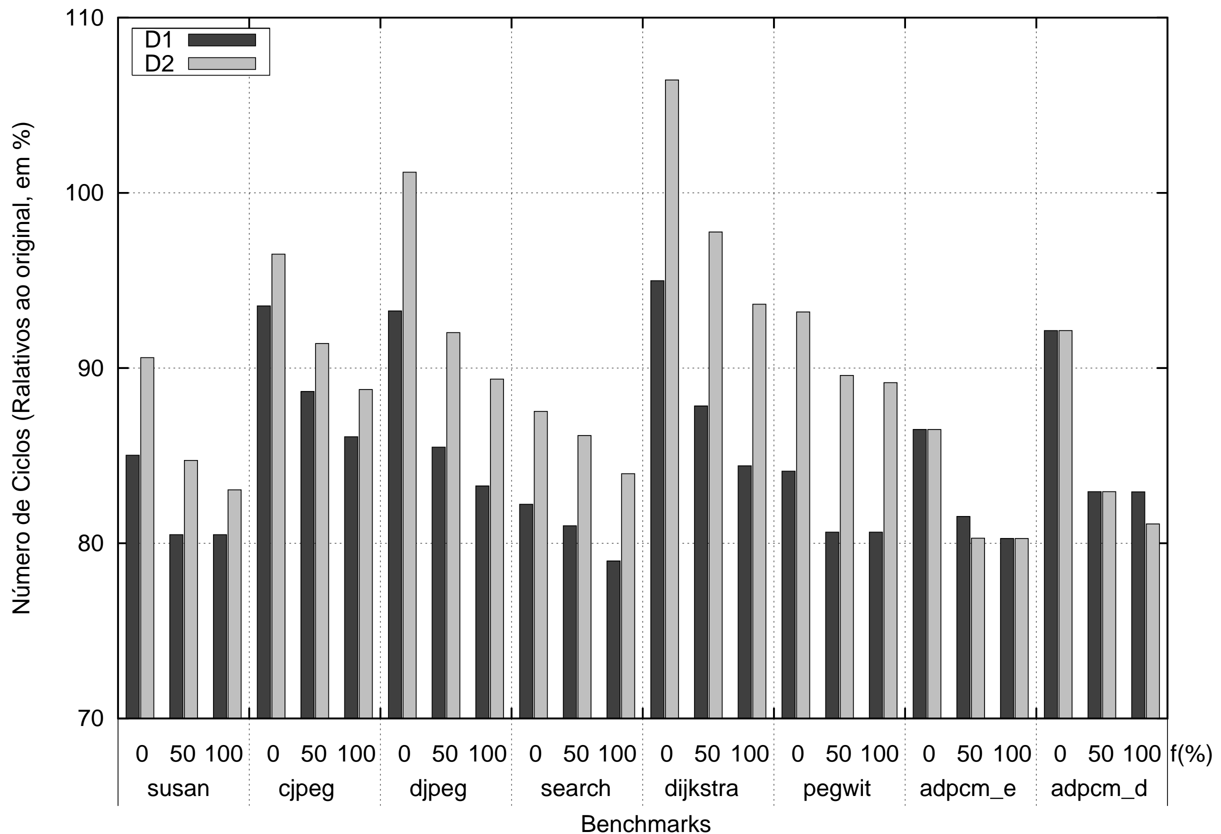


Figura 4.13: Influência dos dados de entrada no desempenho do sistema

de informação dinâmica, desde que os dicionários dinâmicos sejam similares, para reduzir o efeito dos dados de entrada no desempenho do sistema.

4.4 Considerações Finais e Conclusões

Aqui cabem alguns comentários sobre o desempenho do IBC. Em termos de razão de compressão o método foi exaustivamente confrontado com seus pares [8] e apresentou um dos melhores resultados neste item para a arquitetura SPARCv8 e a suíte de *benchmarks* SPECint95. Sobre o aspecto de desempenho, ele havia sido experimentado com um conjunto de aplicações de teste de um modelo VHDL do SPARCv8 (programas *c-irq*, *dhry*, *hello*, *paranoia* e *stanford*), apresentando uma sobrecarga de tempo de 5% (para uma memória rápida com resposta em apenas 1 ciclo). Os experimentos aqui apresentados mostram que para aplicações maiores este desempenho ligeiramente inalterado não se

mantém. A influência do descompressor, na melhor das situações, aumenta em cerca de 50% o tempo de execução do programa. Esta enorme diferença se explica pelo uso de memórias caches grandes (com relação ao tamanho dos códigos) no protótipo, produzindo praticamente apenas falhas compulsórias. Aqui a escolha foi seletiva, uma cache para cada programa e além disto foi escolhida uma taxa de falhas que envolve mais que as falhas compulsórias da cache.

Para diminuir o impacto do descompressor, quando da análise do seu desempenho, algumas alternativas poderiam ser implementadas. A primeira refere-se ao uso do bit extra na ATT. Este bit influencia a razão de compressão em cerca de 3%, mas o desempenho pode ser bastante afetado. Por isto, não havendo uma restrição considerável para o aumento do código comprimido vale a pena descartar este bit extra. A segunda, caso resolva-se manter o bit extra da ATT, é deixar armazenada a linha de cache não utilizada de tal forma que, quando necessária, ela não precise mais de acessos à memória externa. Naturalmente, isto diminui o impacto da descompressão de uma linha não base se na seqüência ela for solicitada.

Quanto à energia, nós mencionamos na seção 4.3.2 que a energia consumida na memória decairá proporcionalmente ao número de acessos. Isto ocorre se o número de transições de bits de endereço durante a execução do código comprimido for semelhante ao original. Em média, em ambos os casos, 2 bits mudam de estado por endereço de acesso, então a premissa é verdadeira e pode ser usada. Não obstante, a redução do gasto de energia na memória principal não implica em redução total no sistema porque há um gasto também no descompressor. Sob este ponto de vista, aumentar a taxa de acerto na cache pode fazer a diferença, mas este é um *trade-off* a ser avaliado porque a redução de energia na memória principal é maior se ela for mais solicitada.

Uma observação também importante refere-se à forma de obtenção dos dados dinâmicos para construção dos dicionários. No estudo apresentado estes dados foram coletados no estágio de *fetch* do processador, mas como o descompressor só é acionado quando há uma falta na cache, poderia ser feita uma análise dos dados lidos na memória principal. Há uma abordagem na literatura (Lefurgy *et al*[47]) que tenta usar esta contagem, mas os resultados não são muito distintos dos obtidos com o método direto aqui exposto. Guardadas as diferenças entre o IBC e o método de Lefurgy, seria necessário uma averiguação para saber se o que ocorreu em [47] também ocorre aqui.

Na literatura uma única abordagem de construção de dicionários que usa múltiplas contagens se limita a encontrar médias de ocorrências de instruções no código para

construir um dicionário estático (no sentido original da palavra) [3]. Nunca havia sido feita uma abordagem para mesclar informações dinâmicas e estáticas, por isto não há como comparar os resultados do presente trabalho com qualquer outro da literatura especializada em termos de melhores abordagens para mixagem de instruções. Nós acreditamos que uma média simples ou mesmo ponderada seria menos eficiente porque poderia retirar das primeiras posições dos dicionários instruções importantes para compressão ou para os aspectos dinâmicos aqui relatados.

Por último, chegamos à seguinte conclusão: para não aumentar consideravelmente o tamanho do código comprimido é necessário usar um valor de $f<90\%$. Por outro lado, para não impactar contra-producentemente na contagem de ciclos usados pelo processador é necessário usar um valor de $f>20\%$. Assim sendo um valor intermediário de f deve ser usado. Além disto, medir o desempenho de um método de compressão baseado em apenas uma estatística de contagem de instruções pode vilipendiar a totalidade dos aspectos envolvidos na construção de um projeto de sistemas embarcados.

Os resultados das arquiteturas CDM podem ser influenciados pelo valor de f utilizado na construção do dicionário. A seguir apresentaremos a arquitetura PDC que também será avaliada com a abordagem *multi-profile* de construção de dicionários

Capítulo 5

O Método PDC-ComPacket

O método IBC foi avaliado neste trabalho usando uma abordagem *multi-profile*. Neste Capítulo apresentamos mais uma contribuição deste trabalho.

O método PDC-ComPacket [89] foi elaborado com o objetivo de diminuir o número de acessos à cache e por conseguinte à memória principal. Por residir no caminho crítico do processador, ele precisa de uma descompressão rápida e eficaz. O método (e também seus pares [60, 62, 65]) utiliza dicionários incompletos, sem reordenação do código original. Além disso, uma nova codificação foi proposta para as *codewords* tendo em vista evitar que elas sejam escritas no programa comprimido em mais de uma palavra de memória, e por conseguinte o método elimina acessos múltiplos à cache para descompressão de uma única instrução. Neste Capítulo vamos mostrar a construção do método com ênfase no algoritmo de compressão e uma visão geral do projeto do hardware do descompressor.

5.1 Análise do Método PDC-ComPacket

O método PDC-ComPacket inclui um hardware descompressor que reside entre o processador e a cache. Para compressão escolhemos um método baseado em dicionários para evitar uma latência maior que um ciclo de *clock* para a tarefa de descompressão. Além de escolher um método de dicionário, o dicionário escolhido é incompleto de 256 entradas tendo uma instrução por entrada. A principal motivação desta escolha, além de uma investigação inicial, com uma das versões básicas do método, foi poder comparar o PDC-ComPacket de forma menos passiva, com os outros métodos PDCs da literatura, que fazem uso da mesma estrutura de dicionários.

Um dos grandes sorvedouros de energia em um sistema embarcado é o sistema de memória. Basicamente, a energia gasta está diretamente relacionada ao número de acessos ao sistema de memórias e ao número de transições de bits nos barramentos durante os acessos. Naturalmente existe a parcela de energia de *leakage*[1], mas ela se mantém relativamente constante independentemente de haver ou não compressão de código. O método desenhado busca então diminuir o número de acessos ao primeiro nível da hierarquia de memória. Para isto, não permitimos que hajam *codewords* que perpassem o limite de 32 bits, ou seja, nunca é necessário dois acessos à cache para reconstruir uma instrução para entrega ao processador.

A escolha entre utilizar uma ATT ou fazer *patching* nos endereços é uma questão de tamanho da ATT. Ora, estando o descompressor entre a cache e o processador, seria necessário usar uma tabela contendo uma tradução de endereços de 1 para 1, ou seja, todos os endereços comprimidos precisavam ocupar uma entrada da ATT. Isto significa que a ATT teria o tamanho do código (em número de entradas). Mais importante ainda é a questão do tempo de descompressão. Além da consulta ao dicionário para converter as *codewords* em instruções, seria necessária uma consulta à ATT para identificar os endereços comprimidos. Embora estes problemas com a ATT possam ser diminuídos, mas não evitados, fizemos a opção pelo uso de *patches*.

O *patching* introduzido em [88], requer a reconstrução de todas as instruções que são usadas para cálculos de endereçamentos do programa. Os saltos diretos relativos a PC são os primeiros a serem corrigidos pelo *patching*. Saltos indiretos também são importantes, mas neste caso as instruções que constroem os endereços que serão armazenados nos registradores de "indireção"é que devem ser alteradas. Finalmente, as tabelas de saltos (*jump tables*) precisam também ser corrigidas. Um exemplo comum de *patching* pode ser visto na Figura 5.1. Do código original é extraído um dicionário (incompleto). As *codewords* [b..i] têm tamanhos fixos de 8 bits. Elas são usadas para construir o novo código. Depois de comprimido o código, os endereços de saltos relativos (BR) precisam ser corrigidos (*patching*). Esta codificação em 8 bits das *codewords* é meramente didática. O método PDC-ComPacket usa uma outra abordagem que será apresentada a seguir.

Existe uma exceção onde não se pode usar *patching*. Se o usuário fornecer explicitamente um endereço e uma instrução de salto indireto receber este valor como operando. O grau de interferência do usuário na execução do código torna incapaz qualquer abordagem usando *patching*. Felizmente, esta situação é de fato uma grande

[1]energia gasta por correntes de fuga

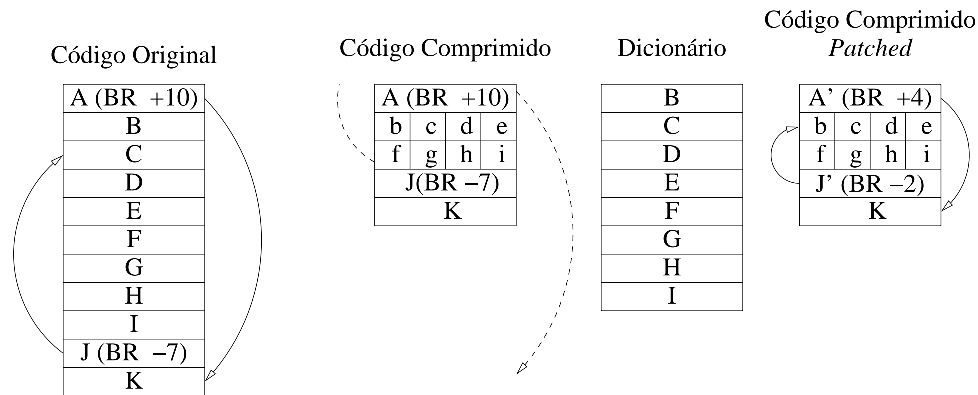


Figura 5.1: Exemplo de *patching* do código

exceção para o nosso domínio de aplicações. Nenhum de nossos códigos apresentou problemas do gênero. Mesmo assim, no caso de tal software (inclui documentação), seria possível indicar na documentação os possíveis alvos desejados pelo usuário.

Os métodos PDCs conhecidos na literatura evitam colocar no dicionário instruções de saltos relativos porque se eles fossem permitidos, depois do *patching*, os saltos dentro do dicionário teriam outro valor de deslocamento. Seria necessário corrigir as *codewords* no código comprimido para apontar para outras instruções no dicionário que tivessem este valor. O problema é que se as *codewords* tiverem tamanhos diferentes e/ou o dicionário for incompleto, é preciso alterar, no código comprimido, os tamanhos das *codewords* afetadas, gerando outros valores de deslocamento e sendo necessário um novo *patch*. Este problema é, mais uma vez, NP completo.

O fato de não permitir que instruções de *branch* pertençam ao dicionário, piora a razão de compressão em cerca de 5% para os nossos *benchmarks*, então optamos por uma estratégia de alocar os saltos no dicionário. Apenas os bits semanticamente significativos dos saltos são guardados no dicionário e o *offset* é alocado junto à *codeword*, no código comprimido. Mais adiante faremos mais uma restrição às instruções de salto que podem compor o dicionário. Em nenhum dos trabalhos PDCs usa-se comprimir instruções de chamadas de funções (CALL) o que facilita o retorno do fluxo de controle para a posição correta.

Uma outra restrição encontrada nos métodos PDC é a questão do alinhamento dos alvos. No exemplo da Figura 5.1, a instrução J', no código comprimido, salta para a *codeword* c, que representa a instrução C. Neste caso, o descompressor precisa saber a partir de qual bit de uma palavra comprimida ele deve começar a descomprimir a

instrução. Uma solução imediata para o problema é usar bits de *padding* para alinhar todos os alvos do programa. Considerando que um bloco básico tem em média 5 instruções a quantidade de *padding* necessária para alinhar todos os alvos pode ser não desprezível, interferindo na razão de compressão. Para diminuir este efeito, uma possível solução é permitir que o processador salte para *nibbles*, por exemplo. Neste caso, o limite dos saltos do conjunto de instruções original é encurtado, porque é necessário usar mais bits para contemplar o novo alinhamento. Embora não tenhamos medido o grau desta interferência nos nossos *benchmarks*, nós optamos por permitir que alvos sejam desalinhados e usamos um conjunto de bits na codificação das *codewords*, para indicar onde o descompressor deve começar a atuar. Esta é mais uma alternativa que estamos introduzindo.

5.2 Codificação das *Codewords*: ComPacket

Seguindo as premissas acima, nós propomos usar palavras comprimidas de tamanho fixo contendo n índices para um dicionário. Como o dicionário é incompleto, é preciso incluir também uma seqüência de escape para que o descompressor diferencie uma instrução normal de um pacote comprimido. Como resultado apresentamos o ComPacket, um pacote de índices e uma seqüência de escape, contida nesta palavra de tamanho fixo. Além disto, também alocamos dentro de um ComPacket o alvo de um possível salto de entrada e o deslocamento, se houver instruções de salto apontadas por um dos índices do ComPacket. A quantidade de índices também varia, entre 2 e 4. Para que tudo isto caiba dentro de uma palavra de 32 bits foi preciso fazer algumas restrições. Nem todos os saltos podem então participar do dicionário, mas somente aqueles que têm *offsets* pequenos, que possam ser representados com até 8 bits (85% dos saltos dos nossos *benchmarks* atendem a estes limites).

A Figura 5.2 mostra as quatro variações de ComPacket usadas em nosso método de compressão. A seqüência de escape ESC contém 8 bits sendo 4 usados para identificar uma instrução inválida do conjunto do SPARCv8 (op = $\mathsf{00}_2$; op2 = $\mathsf{00X}_2$, onde X = *don´t care*).

Este terceiro bit de op2 foi usado para índices em nossa codificação). Os demais bits são usados para informações de identificação do ComPacket: o par TT indica que há uma instrução, apontada por um dos índices, que é alvo. Não permitimos que mais de uma instrução que seja alvo seja apontada dentro de um mesmo ComPacket. Assim, os dois bits TT são suficientes para indicar qualquer alvo dentro de um ComPacket. O bit S é

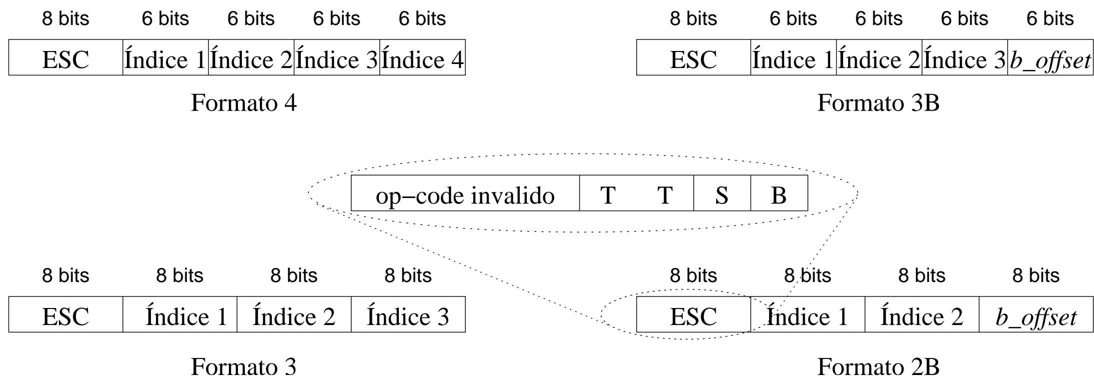


Figura 5.2: Codificação usada nos ComPackets

responsável por indicar o tamanho dos índices. Para $\mathsf{S} = 0$ os índices têm 6 bits, para $\mathsf{S} = 1$, são 8 bits. O bit B indica se uma das instruções apontada por algum dos índices é um salto. Neste caso usa-se o último *slot* para conter o *offset* do salto.

Os ComPackets mostrados na Figura 5.2 são nomeados pela quantidade de índices que possuem e pela presença de um salto. Assim, um ComPacket 3B possui três índices de 6 bits e um *slot*, também de 6 bits, para conter um *offset*. Ora, como já dissemos acima, usamos dicionários de 256 posições, então este formato de ComPacket só pode usar as primeiras 64 instruções do dicionário e o tamanho do *offset* precisa ser menor que 6 bits. Os Formato 3 e 2B fazem uso de todo o dicionário. O formato 2B, apesar do nome, pode ou não conter um salto. No caso de não conter, o *slot* do *offset* não é usado. Por causa da restrição do número de bits no ComPacket, nós não suportamos mais de uma instrução de salto, bem como mais de um possível alvo, no mesmo pacote.

5.3 O Método de Compressão

O método de compressão se inicia com a construção do dicionário, entretanto trataremos deste assunto no Capítulo 6 à frente onde usaremos a abordagem *multi-profile*. Construído o dicionário, o código é percorrido e marcado com as instruções que pertencem ao mesmo. Além disto todos os alvos de saltos são marcados e as instruções que necessitam de *patch* também. Este último passo, embora pareça simples, envolve a reconstrução do fluxo de dados onde são gerados os valores dos registradores para serem usados nos saltos indiretos o que não é uma tarefa trivial.

De posse deste código anotado, o compressor realiza uma busca para construção

dos ComPackets. A idéia é aproveitar ao máximo as instruções que formam o topo do dicionário, por dois motivos: primeiro, elas contribuem mais para formação ou execução do código; segundo, elas podem ser representadas com menos bits, facilitando a criação de ComPackets formatos 4 e 3B que são mais densos de índices e portanto geram maior compressão. Desta forma, ao contrário de percorrer o código em uma abordagem gulosa (*greedy*), marcando os ComPackets que fossem possíveis no código, o que seria mais intuitivo, nós optamos por fazer uma busca de cada instrução do dicionário no código e marcar os ComPackets com uma janela de 8 instruções.

Tendo marcado todos os ComPackets começa a fase de compactação em si. Uma única passada no código original e já estará montado o novo código comprimido e uma tabela contendo todos os endereços convencionais e seus pares comprimidos.

Finalmente, o algoritmo busca todas as instruções do código que precisam de *patching* e, usando a tabela de tradução de endereços montada no passo anterior, faz os ajustes. A propósito, instruções que necessitam de *patching* não são comprimidas, embora possam pertencer ao dicionário. A exceção fica por conta dos saltos (especificamente os *branches*) que podem pertencer ao dicionário e onde o *patching* é aplicado na própria *codeword*. Ainda, uma instrução de ADD pode aparecer em diversos endereços do código e pode ser comprimida em qualquer um deles, exceto nas posições onde é usada para cálculo de endereçamento indireto. O algoritmo relatado pode ser visto na Figura 5.3.

Um exemplo de funcionamento deste algoritmo é mostrado nas figuras 5.4 a 5.12. Depois da construção do dicionário, as instruções do código original são marcadas (D: pertencentes ao dicionário; T: alvos de saltos; e P: necessitam de Patching) (Figura 5.4).

A primeira instrução do dicionário é procurada no código. Sua primeira ocorrência está no endereço 04_h. O algoritmo tenta formar o ComPacket 4 usando a instrução corrente e as 3 instruções anteriores ou 2 anteriores e 1 posterior ou 1 anterior e 2 posteriores ou 3 posteriores. Na Figura 5.5, apesar de haver uma seqüência de 4 instruções que pertencem ao dicionário, não é possível usar um formato 4 porque duas das instruções são alvos, e apenas 1 alvo é permitido por ComPacket.

A segunda tentativa é formar um ComPacket 3. Ainda assim não é possível pelo mesmo motivo citado anteriormente. A próxima tentativa é para o Formato 3B, e mais uma vez não é possível formar um ComPaket.

Agora a tentativa é para formar um ComPacket 2B. Neste caso é possível formar um par com as instruções a e b. Elas são marcadas para um ComPacket 2B (Figura 5.6).

O algoritmo continua a percorrer o código original em busca da instrução b. Ela é

```
CodigoComprimido  COMPRESS(CodigoOriginal) {
  1  Constrói o dicionario DICT  // Capítulo 3
  2  Marca no CodigoOriginal
     2.1 Instruções que pertencem ao DICT
     2.2 Alvos dos saltos
     2.3 Instruções que precisam de  patching
  3  Enquanto DICT não estiver vazio
     3.1 Intrução Atual (IA) = Primeira instrução de DICT
         Encontra no código cada ocorrência de IA
           - Tenta marcar um ComPacket4
           - Tenta marcar um ComPacket3
           - Tenta marcar um ComPacket3B
           - Tenta marcar um ComPacket2B
     3.2 DICT = DICT - {IA}
  4  Fim do Enquanto
  5  Gera a Tabela de Tradução de Endereços
  6  Comprime o código  //CodigoComprimido
  7  Realiza o patching em todas as instruções que precisam
  8  Retorna CodigoComprimido
}
```

Figura 5.3: Algoritmo de compressão do código

encontrada no endereço $0c_h$ (Figura 5.7). A primeira tentativa agora é um formato 4. Não há possibilidade de formação para um ComPacket 4 porque não há 4 instruções na circunvizinhança de *b* que ainda não estejam marcadas para um ComPacket e pertençam ao dicionário. O formato 3 pode ser utilizado pois *c*, *b* e *a* pertencem ao dicionário. Elas são marcadas para um ComPacket 3 (Figura 5.8)

Não há mais instruções *b* no restante do código. A instrução *a* do dicionário passa a ser procurada no código. Sua primeira ocorrência em 00_h, já está marcada para um ComPacket, bem como a segunda em 10_h. Não há mais instruções *a* no código. O

Figura 5.4: Exemplo do algoritmo de compressão: Passo 1

Figura 5.5: Exemplo do algoritmo de compressão: Passo 2

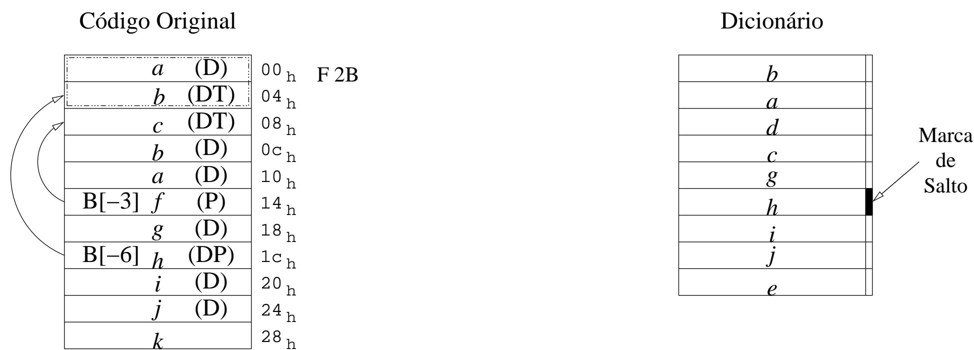


Figura 5.6: Exemplo do algoritmo de compressão: Passo 3

passo se repete para c e d do dicionário e nenhum novo ComPacket é formado. Agora a instrução g é averiguada. Sua ocorrência em 18_h pode gerar um ComPacket 3B dado que g, h e i pertencem ao dicionário, apenas uma delas é um salto, apenas uma delas é um alvo e todas elas se encontram nas primeiras 64 instruções do dicionário (podem ser representadas com 6 bits). Elas são então marcadas para formar um ComPacket 3B (Figura 5.9). As demais instruções do dicionário são averiguadas no código, mas não formam nenhum novo ComPacket.

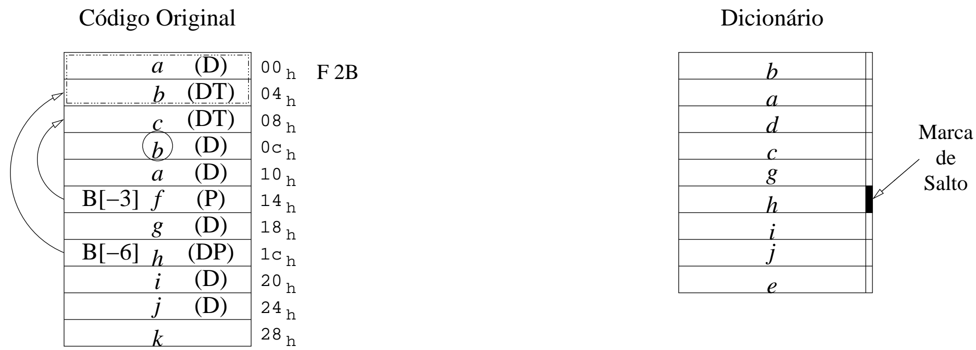


Figura 5.7: Exemplo do algoritmo de compressão: Passo 4

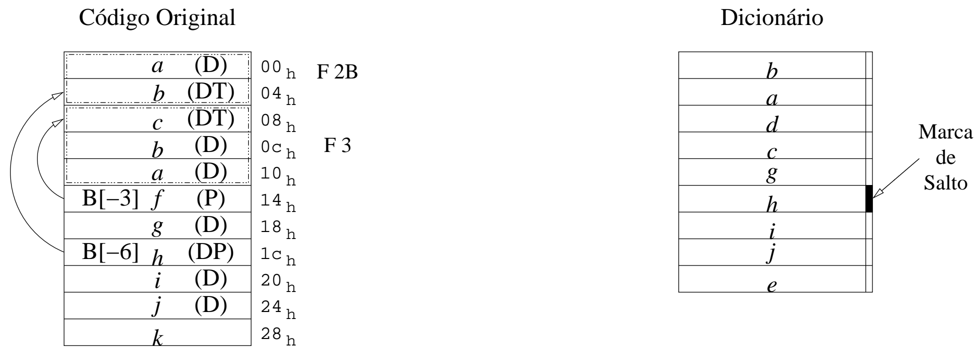


Figura 5.8: Exemplo do algoritmo de compressão: Passo 5

Figura 5.9: Exemplo do algoritmo de compressão: Passo 6

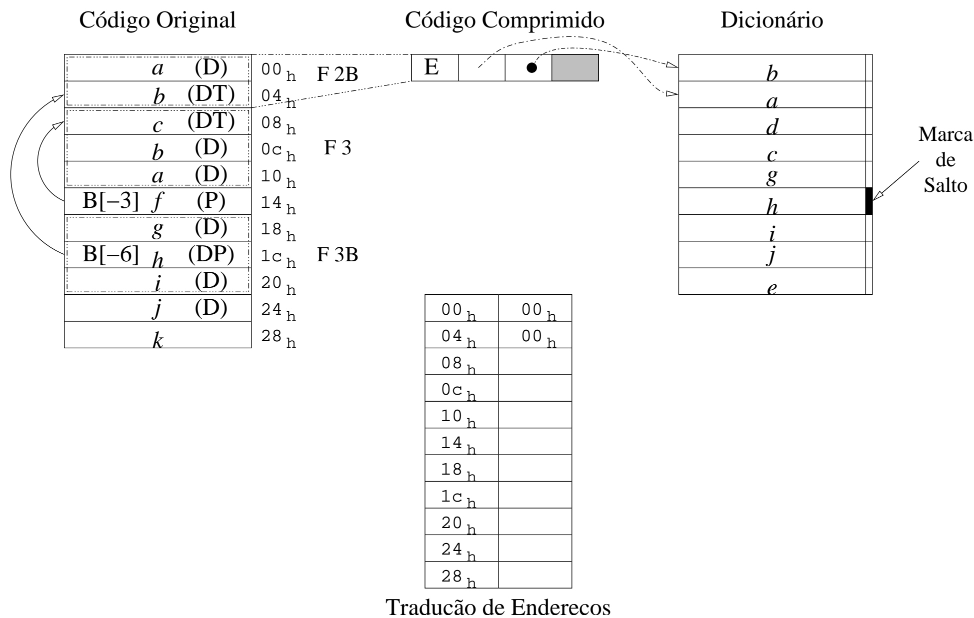


Figura 5.10: Exemplo do algoritmo de compressão: Passo 7

Terminada esta fase de marcação dos ComPacket, resta montá-los e criar uma tabela de tradução de endereços. O código original é percorrido. Inicialmente o ComPacket 2B é achado. A seqüência de escape é montada (Figura 5.10) levando em consideração que a instrução *b* é um alvo, que não há saltos e que o ComPacket é 2B. Neste caso o *slot* do *offset* do salto é inutilizado (preenchido com zeros). O ponto no índice da instrução *b* é a indicação do alvo na figura (indicado pelos bits TT). A tabela de tradução de endereços é alimentada com as primeiras duas instruções do código.

O algoritmo continua até montar todos os ComPackets do Código (Figura 5.11). Montada a tabela de tradução de endereços e formado o código comprimido, agora é necessário atualizar os saltos (*patching*). O endereço 08_h comprimido contém um salto que originalmente tinha *offset* de -3. Depois de comprimido, ele passará a ter *offset* de -1, portanto será necessária uma intervenção no código comprimido. Outra instrução que necessita de *patching* é a *h*. De fato, por pertencer ao dicionário, sequer o *offset* original foi marcado no ComPacket correspondente. O novo valor do deslocamento é -3 (antes era -6). O resultado final da compressão pode ser visto na Figura 5.12 .

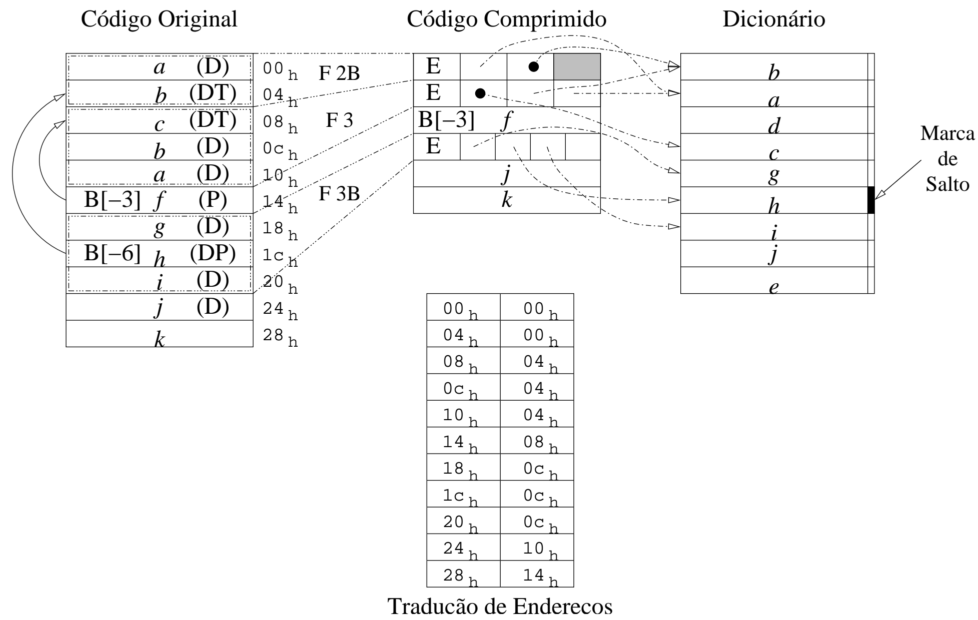


Figura 5.11: Exemplo do algoritmo de compressão: Passo 8

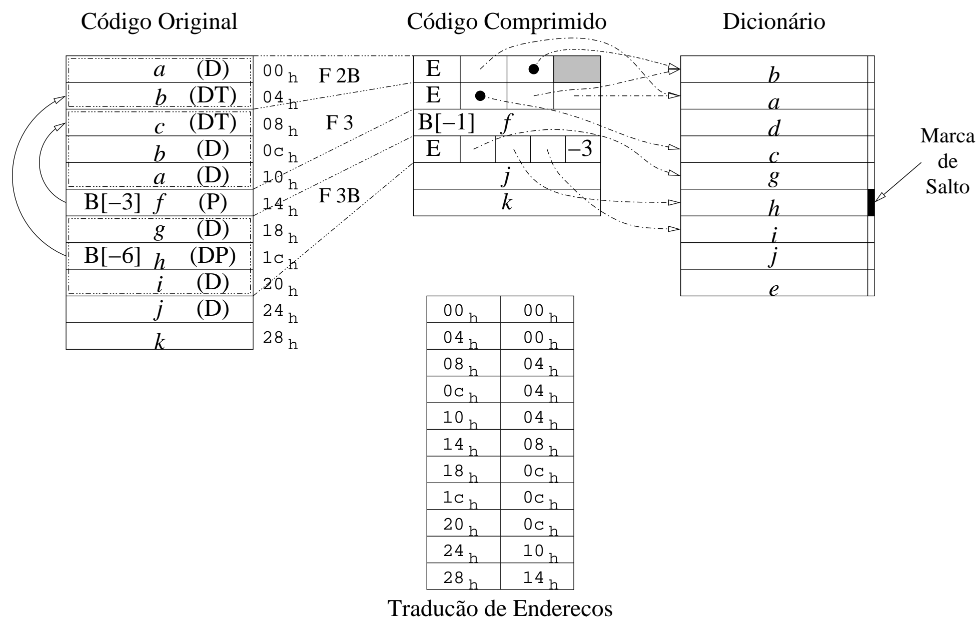


Figura 5.12: Exemplo do algoritmo de compressão: Final.

5.4 Hardware Descompressor do PDC-ComPacket

O Hardware do descompressor foi projetado para atuar como um bloco independente do processador e da cache [90]. Sob o prisma do processador, ele pode ser considerado como um estágio de pré-busca. Os principais componentes deste bloco descompressor podem ser vistos na Figura 5.13. Além da lógica de descompressão, que contém o dicionário utilizado, um bloco de cálculo de endereços comprimido também é utilizado. Este bloco tem como objetivo gerar, a partir da solicitação do processador, um endereço comprimido correspondente. Nele está replicada a lógica de PC do processador, e uma nova lógica paralela de cálculo de PC que é usada para gerar os endereços comprimidos (cPC).

Finalmente, há um *buffer* de entrada que armazena a última instrução que transitou pelo descompressor. Este *buffer* é um componente importante na construção do descompressor porque ele pode evitar acessos desnecessários à cache. Quando um ComPacket é encontrado no código comprimido, somente após a execução de todas as instruções previstas dentro dele é que um novo acesso à cache precisa ser feito (salvo no caso de desvio de fluxo). Aqui, é possível visualizar que quanto mais ComPacket forem encontrados durante a execução do código, menor será a necessidade de acessos à cache.

Por atuar como um estágio de pré-busca do processador, quando ocorre um salto tomado, uma bolha precisa ser inserida no *pipeline*. Assim, o *buffer* de saída mantém uma instrução NOP e a instrução corrente, prontas para entrega. A unidade de cálculo de endereço comprimido é, também, responsável por avaliar se um salto foi tomado.

A Figura 5.14 mostra o bloco LOGIC que é efetivamente a lógica usada para descompressão dos ComPackets. Os principais componentes são os MUXes responsáveis pela escolha do índice que será utilizado para endereçar o dicionário; e o extensor de sinais

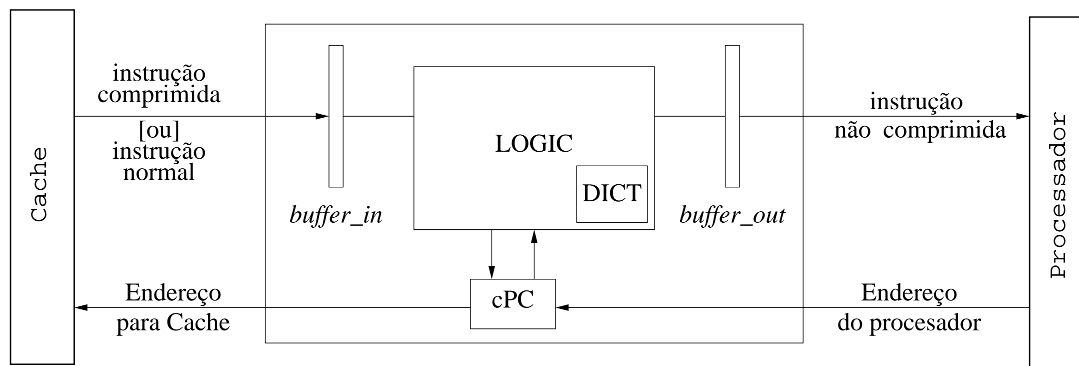


Figura 5.13: Diagrama de bloco do hardware descompressor do PDC-ComPacket

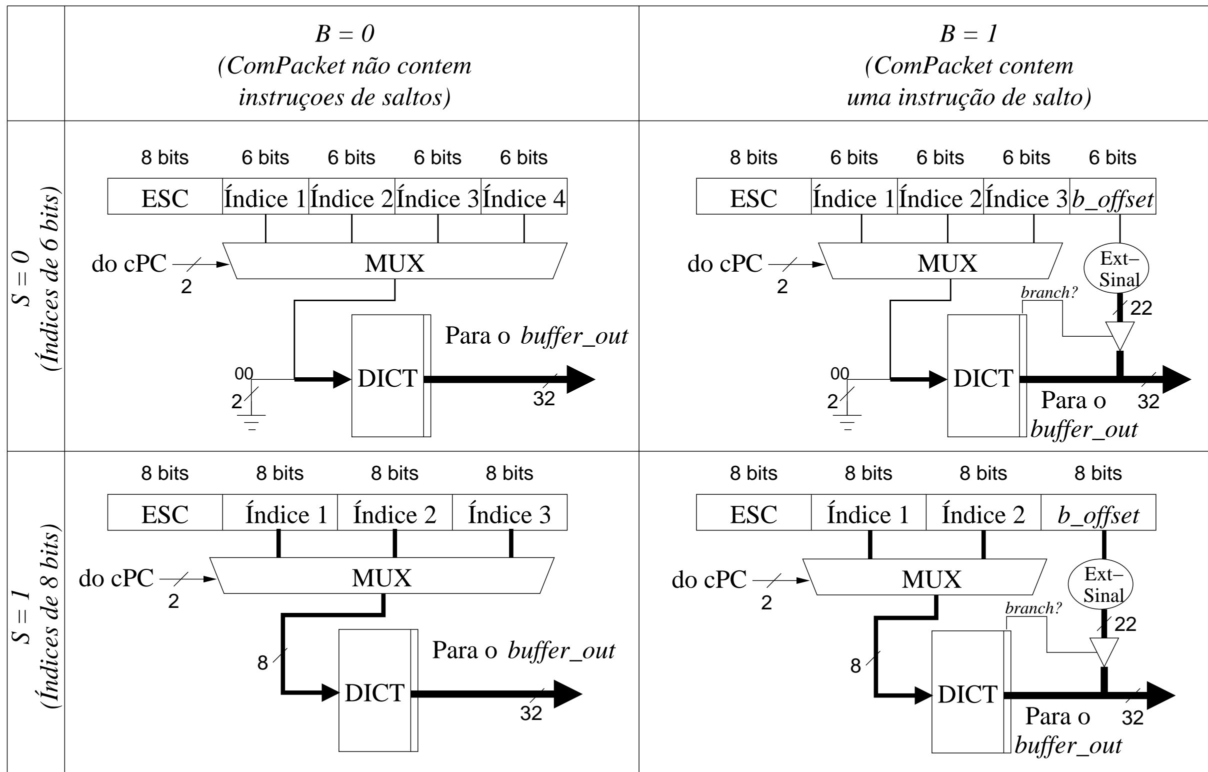


Figura 5.14: Lógica de descompressão para o PDC-ComPacket

usado para reconstruir o *offset* das instruções de salto que estejam presentes no dicionário. A propósito, a marca de salto (*Branch Mark*) mostrada tanto no exemplo do algoritmo de compressão quanto na Figura 5.14 serve para controlar a saída do extensor de sinais que depois será anexada (com um *wire-OR*) aos bits mais significativos da instrução de salto. O valor de S é usado para determinar o tamanho dos *slots* para índices dentro do ComPacket ($S = 0 \Rightarrow 6$ bits; $S = 1 \Rightarrow 8$ bits) e o valor de B, para indicar a presença de um *offset* para instruções de salto.

5.5 Considerações Finais e Conclusões

O método de compressão ora apresentado sofre basicamente de uma restrição: nem todas as instruções que pertencem ao dicionários podem ser aproveitadas na compressão. De fato, do exemplo na Figura 5.12, podemos perceber que a instrução j, embora pertença ao dicionário, não pode ser comprimida porque nenhuma outra instrução na vizinhança estava disponível para formar um ComPacket. O impacto destas instruções

não aproveitadas na compressão do código é de 5% na razão de compressão para os nossos *benchmarks*. Mesmo assim, consideramos importante a idéia de não desalinhar o código para facilitar a descompressão.

Algumas instruções que pertencem ao dicionário contribuem pouco com a razão de compressão do código. Nos casos em que estas instruções não são determinantes para as medidas dinâmicas de desempenho e energia, elas poderiam ainda serem substituídas por instruções que produzam mais ComPackets, numa abordagem bem mais complexa que a utilizada neste trabalho.

Embora pictorialmente as seqüências de escape tenham sido apresentadas com seus 8 bits contíguos, na prática eles são disjuntos porque os campos *opcode* e o *opcode2* do conjunto de instruções do SPARCv8, usados para identificar uma instrução comprimida, não permitem que os bits TT, S e B fiquem juntos e formem com eles dois um conjunto de 8 bits contíguo. De qualquer forma este não é um empecilho, dado que as posições de cada um se mantêm fixas independentemente do tipo de ComPacket a ser formado.

Um outro detalhe, que aparece em algumas figuras, é uma marca de instrução de salto usada junto ao dicionário. Ela ocupa 1 bit na frente de cada instrução do dicionário, portanto, para um dicionário de 256 instruções, perfazendo um total de 8192 bits, estes 256 bits representam um impacto de 3% de acréscimo no tamanho do dicionário.

O método de compressão pode ainda ser melhorado, usando um algoritmo de programação dinâmica para marcar os ComPacket. De fato, no algoritmo de compressão apresentado, não é feita nenhuma avaliação de custo/benefício da formação dos ComPackets, no caso de um conjunto de instruções pertencentes ao dicionário. O algoritmo tem complexidade O(n.m) onde n é o tamanho do código original e m o tamanho do dicionário. No caso de um dicionário pequeno como o nosso não houve uma sobrecarga computacional considerável que justificasse a busca de um algoritmo de complexidade menor.

Uma outra melhoria que poderia ser implementada é o reescalonamento das instruções, em um passo logo posterior à marcação do dicionário no código original. O objetivo seria juntar as instruções que pertencem ao dicionário, mas que se encontram isoladas no código e portanto não conseguem fazer parte de um ComPacket. A implicação para esta reordenação é a dependência de dados. É preciso fazer o novo escalonamento seguindo as regras básicas de otimizações de compiladores.

Quanto ao hardware descompressor, uma possibilidade de diminuir ainda mais os acessos à cache é manter um *buffer* maior que uma palavra. Neste caso ele atuaria

praticamente como uma *filter* cache. Nós experimentamos aumentar o tamanho do *buffer* para 2, 4, e 8 instruções, mas apenas 2 pontos percentuais de diferença na redução de acessos à caches sobre os valores alcançados com um *buffer* simples (mostrados a seguir no Capítulo 6) foi observada.

Cuidados adicionais no hardware devem ser tomados, principalmente quando se trata de instruções de chamadas de funções (CALL). Neste caso o endereço de retorno, armazenado dentro do processador, fica com um valor aleatório. Para cada instrução CALL executada uma instrução extra de ajustes do endereço de retorno é inserida pelo descompressor após o *delay slot*.

No caso de interrupções, o contexto do descompressor precisa ser guardado, porque pode ocorrer que uma instrução que faz parte de um ComPacket seja a causadora da interrupção e neste caso o endereço de retorno precisa ser preservado, inclusive o valor do cPC.

Este descompressor foi projetado e poderia ser utilizado caso não houvesse possibilidade de alterações no processador. Entretanto, existe uma alternativa mais eficaz que será apresentada no próximo Capítulo.

Embora ainda passivo de otimizações, o método de compressão é bastante simples, o que implica em uma descompressão também bastante simples. No próximo Capítulo apresentaremos a integração do método com o dicionário *multi-profile* e apresentaremos os resultados obtidos com o conjunto de *benchmarks* que estamos utilizando.

Capítulo 6

O PDC-ComPacket *Multi-Profile*

No Capítulo 4 mostramos a adaptação do método IBC para o uso de dicionários *multi-profile* e no Capítulo 5 introduzimos o método PDC-ComPacket. Neste Capítulo apresentamos a adaptação do método PDC-ComPacket para uso com dicionários *multi-profile*. O ambiente de simulação (LeonSim) usado foi desenvolvido em nosso laboratório (LSC[1]), como parte deste trabalho, e contém um conjunto de aplicações que perfazem mais de 15.000 linhas de código C++. A descrição em VHDL do descompressor e a integração com o processador Leon[91] (SPARCv8) demonstra a aplicabilidade do método em um ambiente real, numa FPGA. Finalizamos este capítulo com os resultados obtidos para o desempenho, a energia consumida no sistema de memórias e a razão de compressão, incluindo uma análise comparativa com outros métodos da literatura.

6.1 Dicionários *Multi-Profile* Revisitados

O PDC-ComPacket usa dicionários pequenos incompletos, mas limita a presença de algumas instruções neles (CALLs e BRANCHs com *offsets* maiores que 8 bits). O método de construção de dicionários *multi-profile*, apresentado no Capítulo 3, não impõe restrições à presença de instruções, por isto é preciso estudar os efeitos que as restrições do PDC-ComPacket têm sobre os dicionários. Inicialmente verificamos se o fato de não admitir algumas instruções interfere na similaridade entre as medidas estáticas e dinâmicas.

O gráfico da Figura 6.1 mostra que, em uma comparação direta com as curvas de

[1]Laboratório de Sistemas de Computação - IC/UNICAMP

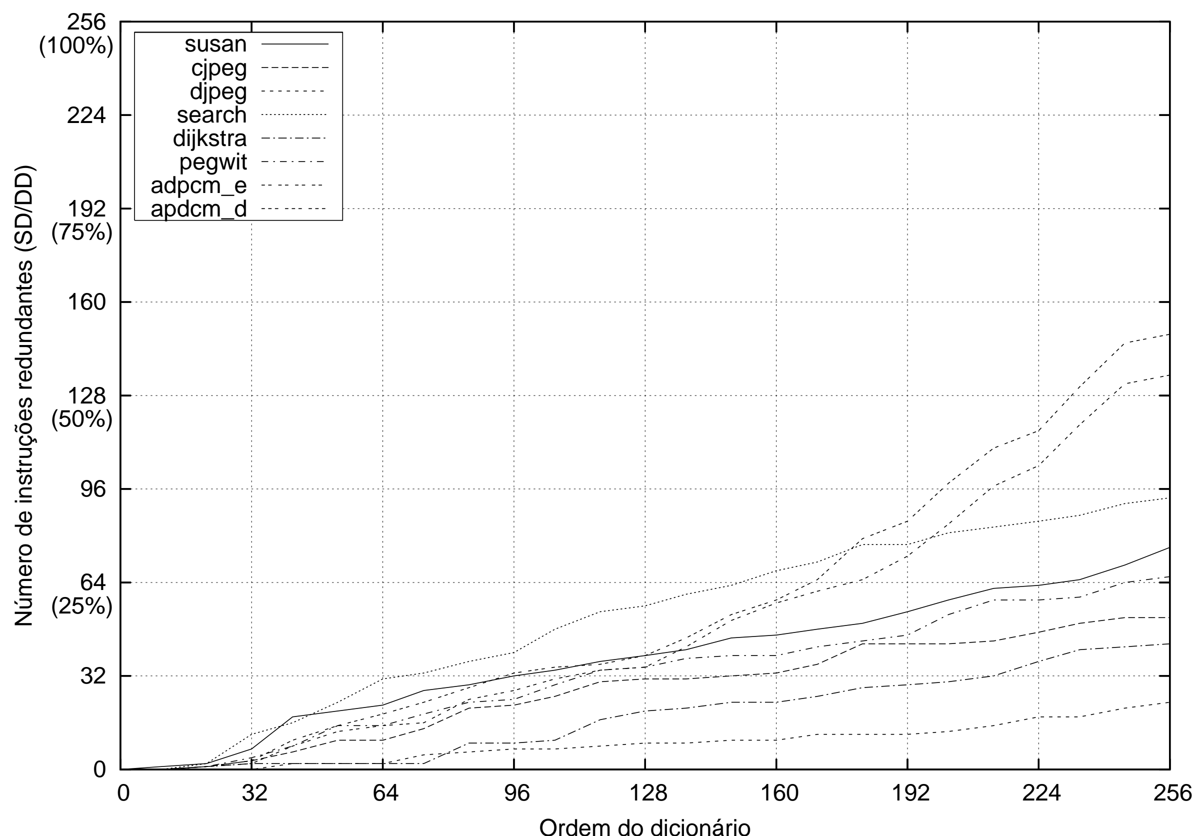


Figura 6.1: Similaridade entre os dicionários dinâmicos (com restrições do PDC-ComPacket)

similaridade dos pequenos dicionários sem restrições (Figura 3.6), há uma maior interseção entre as instruções que compõem os dicionários estáticos e dinâmicos. De qualquer forma, ainda assim, o valor médio da quantidade de instruções em comum não ultrapassa os 30%, além disto, para as primeiras 32 instruções este valor chega a menos de 13%.

A propósito, a contribuição de cada instrução pertencente ao dicionário para formação do código, ou para sua execução, segue uma curva ainda mais acentuada que aquelas mostradas nas figuras 3.7 e 3.8, o que é intuitivo. Enfim, as premissas básicas que suportam a construção de dicionários *multi-profile* permanecem válidas.

6.2 Infraestrutura Utilizada

Todos os experimentos apresentados neste trabalho foram montados sobre uma infraestrutura de software e hardware que envolve mais de 30 mil linhas de código C++,

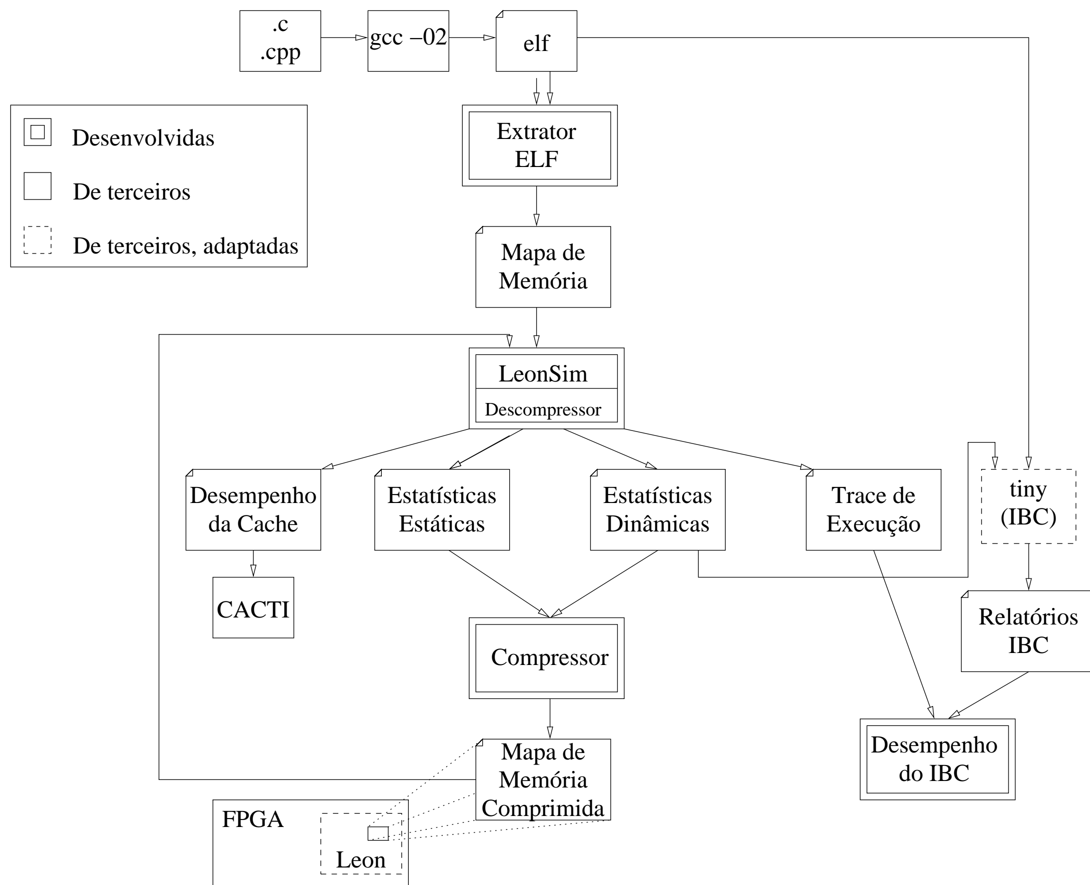


Figura 6.2: Infraestrutura utilizada para avaliar os algoritmos de compressão

uma FPGA e um computador hospedeiro. A Figura 6.2 mostra os principais componentes desta estrutura.

A partir dos códigos em C dos *benchmarks*, usamos o LECCS[86] (GCC para o Leon) para gerar o código executável no formato ELF. A primeira de nossas ferramentas é um extrator que busca todas as seções do ELF que alocam memória, incluindo código (.ini, .fini, .text) e dados (.data, .bss etc) e gera um mapa de memória contendo endereços e conteúdos. Este mapa de memória é utilizado pelo simulador da arquitetura do Leon (LeonSim), também desenvolvido por nós neste trabalho.

O LeonSim é um simulador que gera diversas estatísticas básicas para uso em compressão de código. Além das contagens das ocorrências estáticas e dinâmicas das instruções no código, ele apresenta o desempenho da cache e da memória principal (quantidade de acessos, falhas na cache e as transições de bits na entrada de cada um dos componentes da hierarquia de memória). Por fim, um *trace* da execução do código

também é gerado pelo simulador.

De posse das estatísticas dinâmicas e estáticas, o compressor PDC-ComPacket é usado para gerar um mapa de memória contendo o código comprimido. Este novo mapa realimenta o simulador que executa então este código e mede o desempenho do processador e do sistema de memórias.

O relatório de desempenho das caches é usado também para estimativas do consumo de energia. Utilizamos a ferramenta CACTI [92] que é um estimador de consumo de energia em caches. Ela apresenta os gastos em diversos componentes que constituem as caches (decodificador, *sense amplifiers*, comparador, *drivers* de sinais, *bitlines* e *wordlines*). A ferramenta usa um conjunto de equações que estimam a energia gasta por acesso, sendo os valores apresentados em Joules.

Para realizar os experimentos com o IBC, nós fizemos um porte da ferramenta de compressão (*tiny*) usada por aquele método, para admitir a entrada de um dicionário externo. O objetivo é poder usar um dicionário baseado em contagem dinâmica ou *multi-profile* para compressão. O *tiny* gera um conjunto de relatórios sobre a compressão que nós utilizamos em mais uma ferramenta desenvolvida por nós, para medir o desempenho do IBC.

Finalmente, o mapa de código comprimido é usado para alimentar o sistema prototipado na FPGA contendo um protótipo do descompressor em hardware. O código das ferramentas desenvolvidas perfaz um total de 15.000 linhas de programação em C++, enquanto o *tiny*, que originalmente possuía 15.000 linhas de código C++, passou para perto de 16.000 para suportar a entrada de um dicionário externo.

6.3 Resultados Obtidos

A exemplo do que aconteceu com as medidas de desempenho do IBC, iniciamos nossos experimentos escolhendo uma configuração de cache. Uma das evidentes vantagens de usar uma cache com código comprimido está no fato de que nós podemos ter, de imediato, um aumento na quantidade de instruções por linha. Escolhemos então um tamanho de linha de cache de 16 bytes, contrastando com os 32 bytes usados para o IBC.

A escolha do tamanho da cache seguiu as mesmas premissas usadas para o IBC: taxa de acerto maior que 90% e uma variação menor que 5% quando a cache dobrar de tamanho. Os resultados dos experimentos estão mostrado na Tabela 6.1 e na Figura 6.3. As exceções à regra mais uma vez ficaram com os ADPCM, pois acima do tamanho escolhido as falhas

	Tamanho das Caches								
	64	128	256	512	1024	2048	4096	8192	16384
susan	72	84	94	94	95	96	99	99	99
cjpeg	79	85	87	93	97	98	98	98	99
djpeg	75	86	87	93	97	98	98	98	99
search	78	84	86	87	89	94	96	98	99
dijkstra	77	91	95	98	98	99	99	99	99
pegwit	75	80	85	89	94	96	98	99	99
adpcm_e	75	75	99	99	99	99	99	99	99
adpcm_d	73	81	99	99	99	99	99	99	99

Tabela 6.1: Taxa de acerto nas caches (em porcentagem)

são compulsórias.

É esperado que a razão de compressão do PDC-ComPacket seja um pouco pior que a do IBC, dado que o método usa dicionário incompleto e tem como objetivo a simplicidade na descompressão. Mesmo assim, a razão de compressão do PDC-ComPacket varia entre 71,6% e 88,2% em média dependendo do valor de f usado para a construção do dicionário unificado. Aqui a influência de f é bem mais decisória que no IBC. A variação média entre $f = 0\%$ e $f = 100\%$, chega a 15,6%, em contraste com os 6% do IBC.

A Figura 6.4 mostra a razão de compressão para os *benchmarks* escolhidos. É interessante perceber que entre $f = 90\%$ e $f = 100\%$ há um acentuado aumento no valor da razão de compressão (piora a compressão). Mais da metade de toda diferença entre $f = 0\%$ e $f = 100\%$ se concentra neste trecho das curvas.

Já a Figura 6.5 mostra a redução no número de acessos à cache devido a utilização da compressão (especificamente por causa do *buffer* de entrada). Em média, a diferença na redução no número de acessos para $f = 0\%$ e $f = 100\%$ é de 30 pontos percentuais. Destes, 22 são obtidos entre $f = 0\%$ e $f = 20\%$.

Esta redução no número de acessos à cache, combinado a uma possível redução no número de transições de estado de bits, implica diretamente na redução do consumo de energia no sistema. As transições de bits ocorridas nos barramentos entre o descompressor e a cache podem ser verificadas na Tabela 6.2. As transições de bits seguem a mesma curva da redução de acessos à cache, assim apresentamos apenas os valores de f que têm maior significância. Mais uma vez, entre $f = 0\%$ e $f = 20\%$ a redução das transições é extremamente significativa, enquanto para $f > 20\%$ já se observa um declínio menos acentuado na redução do número de transições.

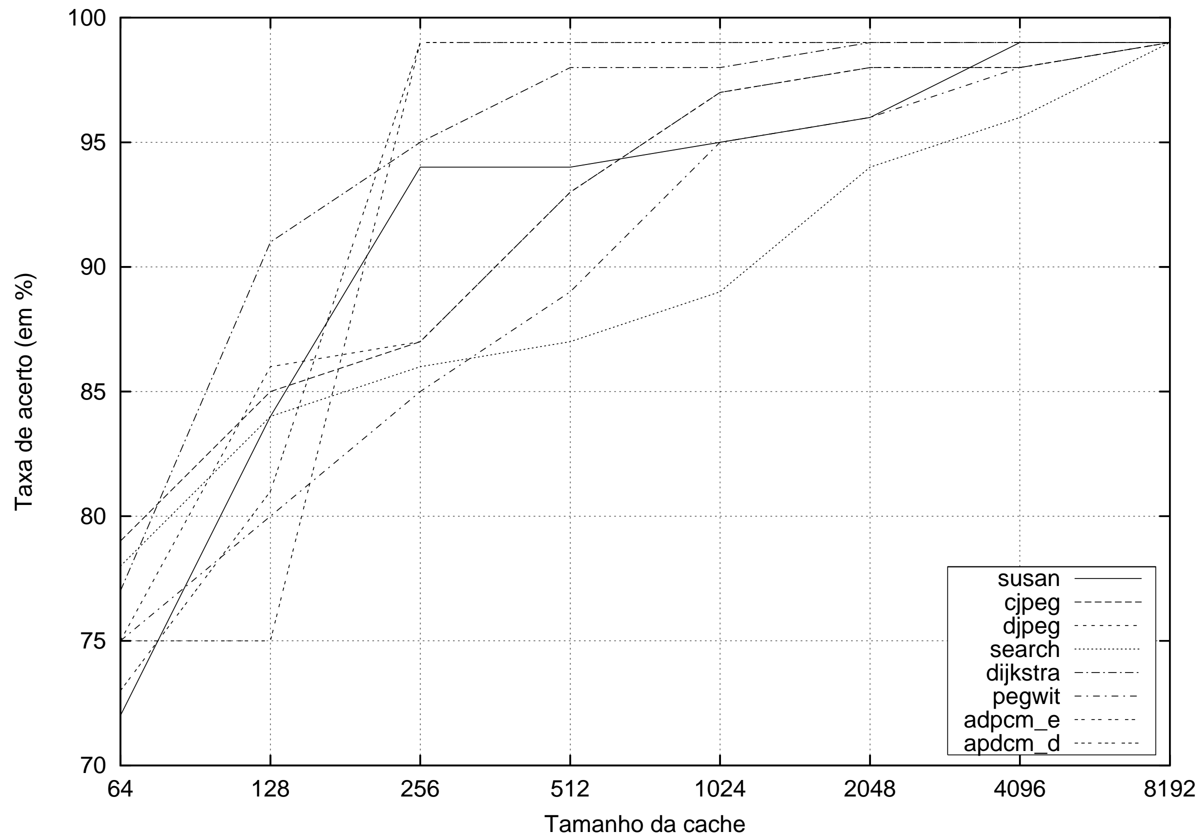


Figura 6.3: Taxa de acerto nas caches para o PDC-ComPacket

Concluímos assim que para não afetar fortemente a razão de compressão é preciso usar um valor de $f < 90\%$ e para diminuir ao máximo o número de acessos à cache é preciso usar um $f > 20\%$. Montando um sistema trivial de inequações construímos uma faixa de valores onde há pouca interferência em compressão e em redução de acessos à cache.

	Original (uncomp)	*f*			
		0%	20%	50%	100%
susan	1131068300	99%	65%	61%	52%
cjpeg	194398718	96%	78%	73%	63%
djpeg	45878397	97%	78%	74%	54%
search	112028630	90%	65%	60%	54%
pegwit	409946579	94%	70%	65%	56%
dijkstra	665188662	92%	56%	52%	49%
adpcm_e	126567600	63%	58%	56%	45%
adpcm_d	96667537	53%	48%	48%	42%
Média		85%	65%	61%	52%

Tabela 6.2: Redução nas transições de bits nos barramentos de entrada da cache

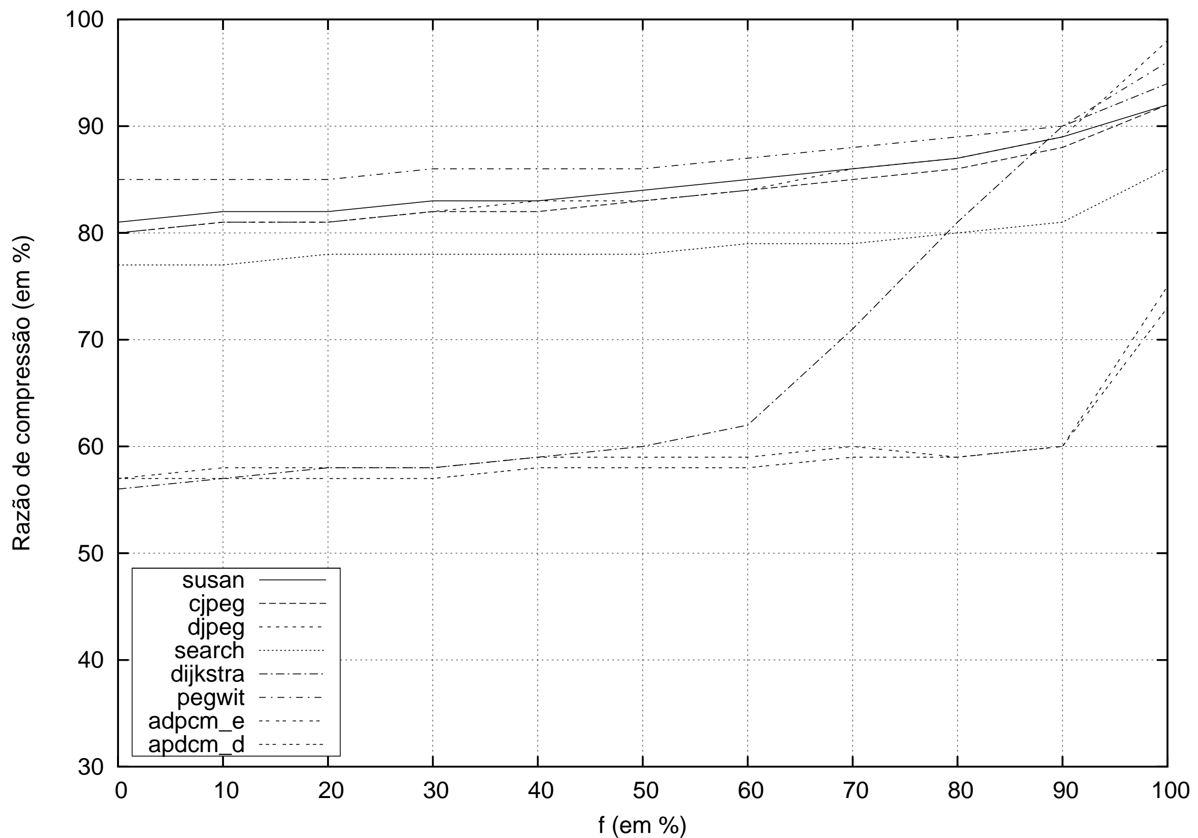


Figura 6.4: Razão de compressão do PDC-ComPacket em função de f

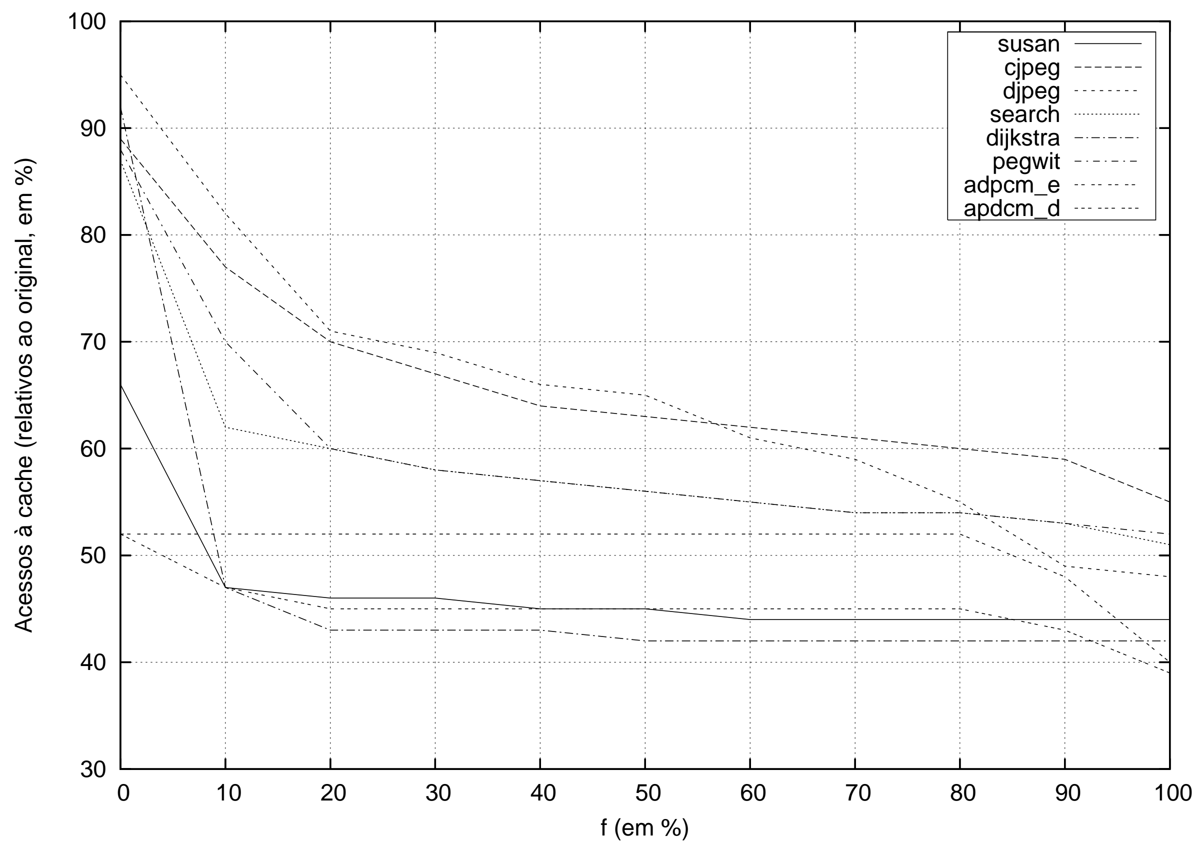


Figura 6.5: Acessos à cache, em relação ao original, em função de f

	Razão de Compressão					Redução de acessos à cache				
	Δ do melhor	Δ do pior	Melhor	Pior	f=50%		Pior	Melhor	Δ do pior	Δ do melhor	
susan	2	9	81	92	83	45	66	44	21	1	*susan*
cjpeg	3	9	80	92	83	63	89	55	26	8	*cjpeg*
djpeg	3	15	80	98	83	65	95	48	30	17	*djpeg*
search	1	9	78	86	77	56	87	51	31	5	*search*
dijkstra	4	34	56	94	60	42	92	42	50	0	*dijkstra*
pegwit	1	10	85	96	86	56	88	52	32	4	*pegwit*
adpcm_e	2	16	57	75	59	52	52	40	0	8	*adpcm_e*
adpcm_d	1	15	57	73	58	45	52	39	7	6	*adpcm_d*
Média	2	15							25	6	Média

Tabela 6.3: Redução nas transições de bits nos barramentos de entrada da cache

6.3.1 Um Caso Selecionado de f

Doravante apresentaremos resultados a partir de um valor de f escolhido de tal forma a interferir minimamente em compressão ou transições de bits ou acessos à cache. Este valor escolhido é um ponto intermediário nas curvas: $f = 50\%$. Para este valor podemos perceber na Tabela 6.3 que os melhores resultados de compressão distam dele em apenas 2 pontos percentuais, enquanto os piores distam 15, em média. Para redução de acessos à cache estes valores são de 6% e 25% respectivamente. Este valor de f, portanto, aproxima, ao mesmo tempo, os melhores resultados em compressão e os melhores resultados em redução de acessos à cache.

Não há duvidas que adicionar algo ao caminho crítico de processador causa uma certa apreensão. Apesar do descompressor agir como em um estágio de pré-busca, ele provoca um ônus na execução do código: sempre que há um salto tomado, um ciclo extra é utilizado para descartar a instrução que já está no descompressor. Por outro lado, a bonificação em ciclos está na quantidade de acessos à memória principal, que decai devido à compressão do código. Quanto mais lenta for a memória, mais ciclos serão evitados.

Nós montamos um experimento variando a quantidade de ciclos usados por falha na cache. Escolhemos um hipotético valor de 0 ciclos até 100 ciclos baseados nos números apresentados em [93]. A Figura 6.6 mostra a redução no número final de ciclos de *clock* relativa à execução do código original. Mesmo para uma memória rápida, cuja penalidade seja de cerca de 10 ciclos por falha na cache, uma redução de 25% em média no número

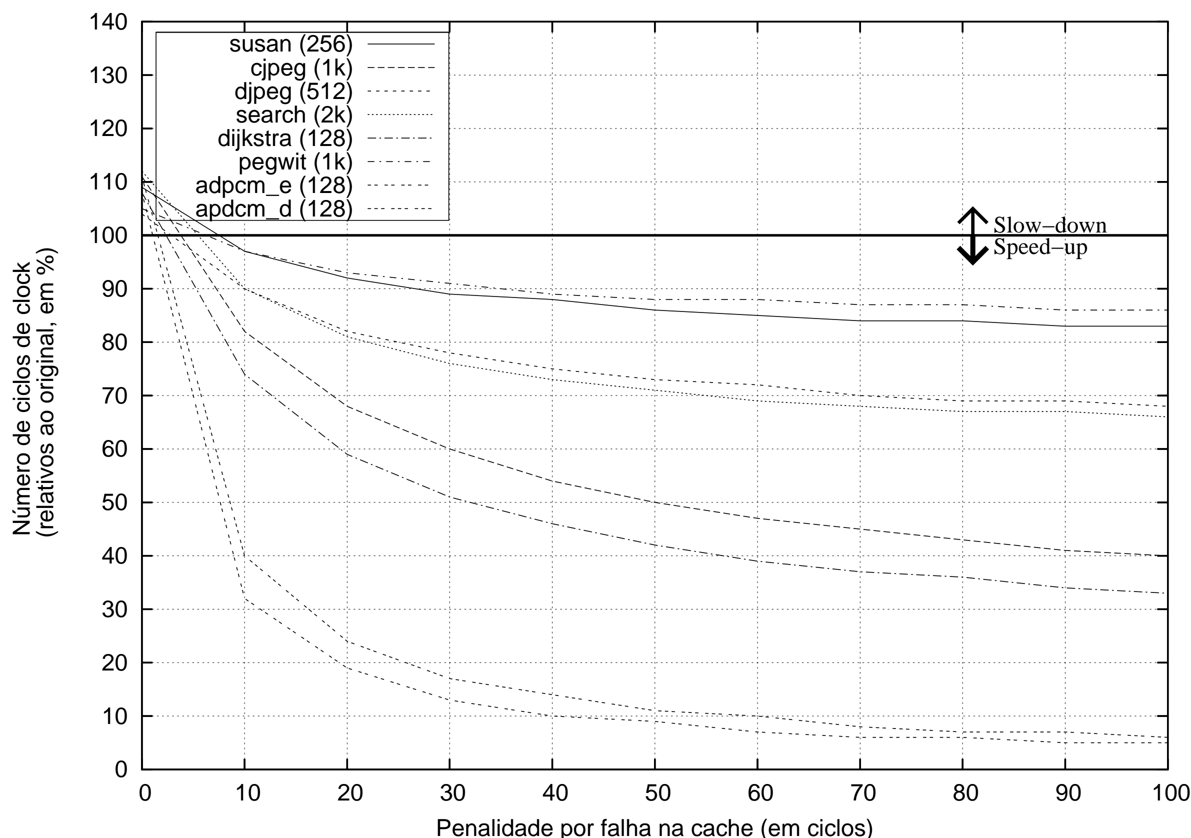


Figura 6.6: Redução no número de ciclos devido a compressão do código

de ciclos já é alcançada. Mais ainda, mesmo para as aplicações que apresentam os piores resultados alguma redução no número de ciclos acontece para esta memória.

Para uma memória muito lenta (100 ciclos/falha na cache), a redução chega a um impressionante valor de 52%. Isto é, com menos da metade do tempo que o programa sem compressão gasta, o seu equivalente comprimido chega ao fim da execução.

A energia usada na cache de instruções foi estimada usando o CACTI para uma tecnologia CMOS 0,8mm. Apesar desta tecnologia não representar o estado-da-arte do processo fabril ela não é de todo irrazoável. Além disto, escolhemos apenas um valor básico que pode ser escalonado para tecnologias mais atuais. Embora o CACTI seja um estimador analítico de consumo de energia ele é comumente utilizado em trabalhos com caches, pois apresenta aproximações muito razoáveis. A propósito, uma de suas equações usa uma taxa de transições de bits de entrada (endereço) fixa. O fato é que, tanto no código comprimido, como no código não comprimido, esta taxa (por acesso) se mantém, fazendo com que não seja efetivamente necessária alguma alteração na equação.

	Código Normal (Joule x 10^{-3})	Código comprimido (Joule x 10^{-3})	Redução (%)
susan	307,98	139,59	45
cjpeg	47,68	30,26	37
djpeg	9,82	6,38	35
search	30,34	17,10	44
dijkstra	110,66	47,47	57
pegwit	104,34	58,93	44
adpcm_e	19,94	10,38	48
adpcm_d	14,84	6,80	54
		Média	46

Tabela 6.4: Redução de energia na cache de instruções

A Tabela 6.4 apresenta os valores do consumo de energia para os *benchmarks* utilizados. Em média o consumo cai para 46% do seu valor original. Este é outro número importante para atestar a eficácia do método de compressão.

6.3.2 Influência dos dados de entrada

Ainda, seguindo este caso selecionado de f, nós abordamos a influência dos dados de entrada na compressão do código. A Tabela 6.5 mostra os tamanhos dos códigos comprimidos quando a compressão foi baseada no conjunto de dados de entrada D1 e também para os dados de entrada D2 (já citados na Tabela 3.1). A variação na razão de compressão fica em apenas 0,2% em média, indicando que os dados de entrada influenciam pouco este parâmetro.

Também averiguamos a interferência dos dados de entrada no desempenho do sistema. Para tanto, escolhemos 4 velocidades de resposta da memória principal, a saber: 10 ciclos, 20 ciclos, 40 ciclos e 100 ciclos, necessários para preenchimento de uma linha da cache quando ocorre um falha. A Tabela 6.6 e a Tabela 6.7 mostram os resultados obtidos. O código original foi comprimido com o dicionário gerado a partir de D1 e depois, para execução do código comprimido, foram utilizados os dois conjuntos de dados de entrada D1 e D2. A diferença na redução de ciclos não chega a 2% entre um caso e outro, assim, concluímos que na métrica de desempenho, os dados de entrada têm pouca interferência, como já ocorrera na razão de compressão.

Por fim a influência dos dados de entrada no consumo de energia na cache também foi avaliada. Como anteriormente, o código foi comprimido com base em informações

retiradas quando da execução do código submetido ao conjunto de dados D1 e a seguir ele foi avaliado com a entrada de dados D2. A Figura 6.7 mostra que a maioria dos programas não sofre uma influência considerável dos dados de entrada.

	Tamanho original (em instruções)	D1		D2	
		Tamanho comprimido (em instruções)	Razão de compressão	Tamanho comprimido (em instruções)	Razão de compressão
susan	18.604	15.503	84,7%	15.505	84,7%
cjpeg	25.316	20.855	83,4%	20.871	83,5%
djpeg	28.816	23.816	83,5%	23.801	83,5%
search	8.760	6.662	79,0%	6.673	79,1%
dijkstra	7.944	5.984	78,6%	5.993	78,7%
pegwit	18.804	15.699	84,8%	15.709	84,9%
adpcm_e	2.284	1.093	59,1%	1.108	59,7%
adpcm_d	2.284	1.072	58,1%	1.085	58,7%

Tabela 6.5: Influência dos dados de entrada na razão de compressão do PDC-ComPacket

	10 ciclos de penalidade			20 ciclos de penalidade		
	ciclos originais	D1	D2	ciclos originais	D1	D2
susan	210.001.854	88,2%	86,8%	285.205.414	77,9%	75,9%
cjpeg	27.046.122	82,5%	82,5%	36.052.832	68,3%	68,3%
djpeg	6.568.799	89,6%	89,6%	8.796.659	82,3%	82,3%
search	16.306.384	90,4%	103,1%	23.011.724	81,3%	97,6%
dijkstra	109.926.131	73,9%	73,2%	153.505.751	59,1%	58,1%
pegwit	57.454.708	97,1%	97,6%	77.499.958	93,2%	94,0%
adpcm_e	33.679.526	32,3%	32,3%	57.314.276	19,0%	19,0%
adpcm_d	20.690.604	39,7%	39,7%	33.993.824	24,2%	24,2%
	Média	74,2%	75,6%	Média	63,2%	64,9 %

Tabela 6.6: Influência dos dados de entrada no desempenho do sistema

	40 ciclos de penalidade			100 ciclos de penalidade		
	ciclos originais	D1	D2	ciclos originais	D1	D2
susan	435.612.534	68,0%	65,3%	886.833.894	58,4%	55,1%
cjpeg	54.066.252	54,2%	54,2%	108.106.512	40,0%	40,0%
djpeg	13.252.379	75,1%	75,1%	26.619.539	68,0%	68,0%
search	36.422.404	73,2%	91,3%	76.654.444	65,9%	83,8%
dijkstra	240.664.991	45,7%	44,4%	502.142.711	33,3%	31,7%
pegwit	117.590.458	89,5%	90,5%	237.861.958	85,8%	87,1%
adpcm_e	104.583.776	10,5%	10,5%	246.392.276	4,5%	4,5%
adpcm_d	60.600.264	13,6%	13,6%	140.419.584	6,0%	6,0%
	Média	53,7%	55,6%	Média	45,2%	47,0%

Tabela 6.7: Influência dos dados de entrada no desempenho do sistema

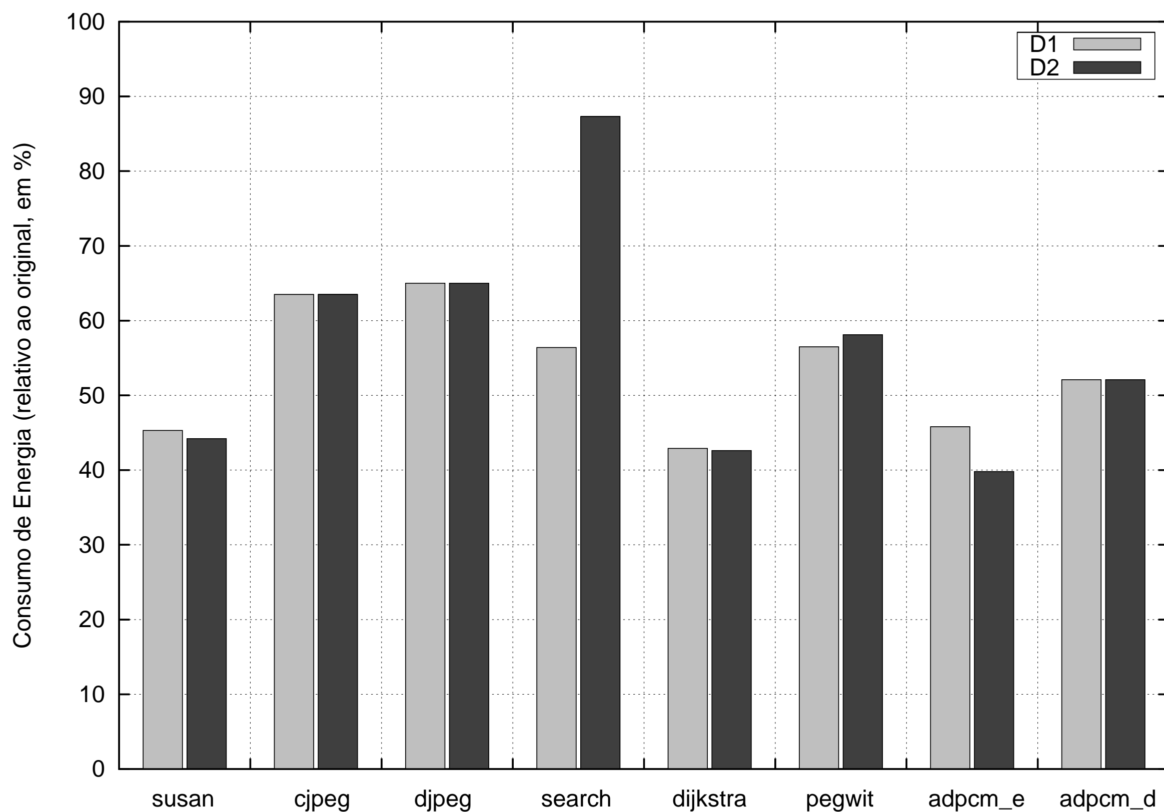


Figura 6.7: Influência dos dados de entrada no consumo de energia da cache de instruções

6.4 Análise Comparativa dos Resultados

A tarefa mais ousada quando propomos avaliar um método de compressão é a comparação com seus pares. A inexistência de uma plataforma comum leva a análises que são apenas indícios de uma realidade, isto quando não se faz um arrazoado cético com números de tabelas que compendiam os resultados. Apesar disto, às vezes não há condições suficientes para abalizar um raciocínio completo, seja por falta de dados, seja pela própria inovação que o trabalho apresenta, ou mesmo pelo objetivo do método, assim restando uma avaliação *per summa capita* dos benefícios da compressão.

Escolhemos uma configuração intermediária para análise comparativa. Uma memória principal que exige 30 ciclos para preencher uma linha de cache no PDC-ComPacket e para o caso do IBC, uma memória que requer 8 ciclos por leitura. As configurações de cache são exatamente as mesmas descritas nos Capítulos anteriores, e o valor de f também segue o raciocínio neste Capítulo ($f = 50\%$)

Iniciamos nossa abordagem comparativa a partir dos compêndios extraídos do Capítulo 2. Dos métodos de compactação podemos apreciar apenas as medidas de razão de compressão e tempo de execução. O PDC-ComPacket apresenta uma razão de compressão pior que a maioria dos métodos (eles utilizam *benchmarks* maiores e por isto podem apresentar uma razão de compressão melhor), mas o grande problema destes métodos de compactação é a execução do código comprimido que pode ser muito lenta quando se gera de código interpretável (trabalhos de Fraser e Ernest). Para métodos que geram código executável (Debray e Liao) os resultados de desempenho chegam a apresentar ganhos, mas neste caso há uma simples manipulação do código original, exatamente como são feitas as otimizações no nível de compiladores. Isto significa que a compressão pode ser aplicada diretamente após a compactação, pois os métodos se tornam ortogonais. Os resultados estão compendiados na Tabela 6.8.

A conclusão que se pode tirar destes números é que há possibilidades de melhorias na densidade de código ainda antes da aplicação de um método de compressão.

Para os principais métodos CDM apresentados podemos perceber que a maioria se baseia nas arquiteturas MIPS o que torna o trabalho de comparação mais difícil com o nosso. A Tabela 6.9 mostra os resultados dos métodos CDM, inclusive o IBC com a nossa contribuição da abordagem *multi-profile* para construção de dicionários. Mas, a partir desta tabela, podemos chegar a algumas conclusões. Primeiro, para um mesmo método de compressão aplicado em uma arquitetura MIPS e em uma arquitetura

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução
Debray	squeeze	Alpha	Mediabench SPEC95	54%	-16%
Fraser	lcc	SPARC	lcc	50%	+2000%
Ernest	BRISC	SPARC	lcc, ggg, word, agrep, xlisp, espresso	60%	+1260%
Liao	SOA/GOA	TMS320C25	chendct, chenidct, leedct, ileedct, jrev, readgif, autocrop, smooth, hufftree, gnucrypt	80% / 97%	–
Wanderley Netto	PDC-ComPacket	SPARC	Mediabench & MiBench	72% ∼ 88%	-45%♣

♣Resultado obtido para um valor de $f = 50\%$ e uma memória pricipal que requer 30 ciclos de *clock* para o preenchimento de uma linha de cache.

Tabela 6.8: Comparação do PDC-ComPacket com métodos de compactação

SPARC, as melhores razões de compressão ficam para MIPS. Assim sendo, pode-se esperar que o PDC-ComPacket apresente resultados de compressão melhores se aplicado nesta arquitetura alvo.

Segundo, para o método IBC, quando aplicado aos *benchmarks* SPECint95 e o conjunto retirado do *Mediabench* e *MiBench*, há uma diferença considerável na razão de compressão. O conjunto do SPEC comprime muito mais. Basicamente esta diferença acontece devido ao tamanho dos códigos originais, que no SPEC é bem maior, abrindo mais oportunidades de compressão.

Terceiro, o PDC-ComPacket e o IBC quando submetidos aos mesmos *benchmarks* apresentam razões de compressão parecidas, mas o PDC-ComPacket é muito mais sensível ao valor de f (a variação na razão de compressão é devido ao valor de f utilizado).

Por fim, em termos de desempenho, podemos avaliar que tanto o IBC como o PDC-ComPacket, quando usando uma abordagem *multi-profile* com valor de $f = 50\%$ apresentam os melhores resultados para arquiteturas SPARC. A propósito, à medida que novos métodos de compressão vêem surgindo, a inclusão de uma análise de desempenho tem se tornado praxe.

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução
Wolfe & Chanin‡	CCRP	MIPS	lex, pswarp, yacc, eightq, matrix25 lloop01, xlisp, espresso e spim	73%	–
Lekatsas‡	SAMC	MIPS	SPEC95	57%	–
Lekatsas‡	SADC	MIPS	SPEC95	52%	–
Lefurgy	por Software	MIPS	Mediabench SPEC95	65%	+2% ∼ +54%
Pannain	PBC	MIPS	SPECint95	61,3%	–
Centoducatte	TBC	MIPS	SPECint95	60,7%	–
Azevedo	IBC	MIPS	SPECint95	53,6%	–
IBM	CodePack	PowerPC	Mediabench SPEC95	60%	±10%
Lekatsas‡	SADC	PentiumPro	SPEC95	75%	–
Centoducatte	TBC	TMS320C25	aipint, bench, gnucrypt, gzip, hill, jpeg,rx, set	75%	–
kirovski‡	pcache	SPARC	Mediabench SPEC95	40%	+11%
Azevedo	IBC	SPARC	SPECint95	63,8%	-12% ∼ +84%
Wanderley Netto	PDC-ComPacket	SPARC	Mediabench & MiBench	72% ∼ 88%	-45%♣
Wanderley Netto	IBC	SPARC	Mediabench & MiBench	71% ∼ 88%	-22%♡

‡ Não considera os tamanhos das tabelas das tabelas na razão de compressão

♣ Resultado obtido para um valor de $f = 50\%$ e uma memória pricipal que requer 30 ciclos de *clock* para o preenchimento de uma linha de cache.

♡ Resultado obtido para um valor de $f = 50\%$ e uma memória pricipal que requer 8 ciclos de *clock* para entregar uma palavra de 32 bits para cache.

Tabela 6.9: Comparação do PDC-ComPacket com métodos CDM

Talvez o comparativo mais apropriado e necessário do método PDC-ComPacket seja com seus pares mais próximos. Os métodos PDC na literatura são basicamente três: o de Lekatsas, o de Benini e o nosso. Os outros métodos, que nós também consideramos PDC por causa dos requisitos, não prevêem o uso de uma cache.

Para estes métodos começaram a surgir medidas de energia que envolvem o uso de código comprimido. Mesmo assim estas medidas são difíceis de serem confrontadas, pela forma e ou local onde são observadas. Dentro do sumário dos métodos PDC na Tabela 6.10 podemos perceber que o primeiro trabalho de Benini, para um MIPS sem cache, apresenta o resultado do ganho de energia em todo o sistema. Já em seu segundo trabalho, ele apresenta de forma isolada o consumo na cache de instruções. Lekatsas, por sua vez, não dissocia as caches (dados e instruções). Estas condições dificultam a análise dos resultados de energia, mas poderíamos arrazoar que o PDC-ComPacket apresenta uma redução no consumo de energia no mínimo comparável aos resultados de Benini. O motivo é que, como a arquitetura MIPS possui uma melhor razão de compressão que uma SPARC, é esperado que a variação de energia também siga a compressão. Assim podemos inferir que a redução de energia do PDC-ComPacket em um MIPS é esperada ser maior que no SPARC. Ora, se os valores de redução já não são tão distintos assim, é razoável a afirmação acima.

No caso do trabalho de Lekatsas com o SPARC, o PDC-ComPacket se mostrou muito mais eficaz na redução do consumo de energia. O problema que está por trás desta comparação direta é que a energia apresentada por Lekatsas compreende ambas as caches de dados e instrução. Então, mais uma vez, é preciso uma análise cautelosa. Ora, em alguns estudos sobre consumo de energia em memórias cache, podemos averiguar que a cache de instruções consome próximo a 65% do total de energia [94]. Isto significa que na medida de Lekatsas há uma parcela que é fixa (já que somente o código é comprimido) do consumo de energia e há outra variável (a do consumo na cache de instruções). Então, para perfazer 28% de redução de energia nas caches é preciso um valor maior que este de redução na cache de instruções. Mas a cache de dados contribui minoritariamente com este consumo, então o valor não precisaria ser muito diferente de 32% se pudéssemos aplicar a mesma regra de divisão de consumo entre as caches. Assim sendo podemos afirmar que para arquiteturas SPARC o PDC-ComPacket promove a melhor redução de energia na cache de instruções, guardadas as diferenças entre os *benchmarks*.

Para o desempenho, o PDC-ComPacket se mostrou mais eficaz que o método de Lekatsas, mas não há como fazer uma comparação justa porque não há informações

naquele trabalho sobre a penalidade em ciclos que uma falha na cache promove.

Quanto à razão de compressão, cabe ressalvar que os trabalhos de Lekatsas nunca computam as tabelas, os dicionários e o hardware descompressor usados na descompressão como parte desta métrica. Em nossas aplicações, o *overhead* do dicionário perfaz em média 5 pontos percentuais na razão de compressão. Então, se descartarmos o dicionário usado teríamos um valor de 67% para nossa razão de compressão, muito próximo à melhor marca de 65%.

Lefurgy usa dicionários maiores e *benchmarks* também. Como já vimos, para *benchmarks* maiores é factível encontrar razões de compressão melhores, assim o PDC-ComPacket não estaria tão distante da marca de Lefurgy em compressão.

Para uma comparação com o método de Liao, vale a pena ressaltar que arquiteturas DSP comprimem menos que as RISC, mas não dá para inferir se aquele método suplantaria o PDC-ComPacket em razão de compressão. De qualquer forma o desempenho obtido por Liao traz sempre uma piora nos valores originais.

Por último, apresentamos uma comparação entre o PDC-ComPacket e os modelos comerciais que implementam um conjunto de instruções nativo em 16 bits. A Tabela 6.11 mostra os resultados obtidos. Dificilmente uma comparação eficaz poderia ser feita. Primeiro porque não há uma suíte de *benchmarks* usada para avaliar os métodos comerciais, então não se sabe o quão extensivo ou quais características foram exploradas na obtenção dos dados. De qualquer forma, estes métodos pecam por reduzir a expressividade do conjunto de instruções original. E, além disto, eles têm um conjunto fixo de instruções de 16 bits, desta forma operam como se possuísse um dicionário estático (no sentido original da palavra), não se adaptando às idiossincrasias dos programas. De qualquer forma o PDC-ComPacket suplanta os resultados de desempenho obtidos nos dois casos (ARM/Thumb e MIPS16).

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução	Energia†
Liao	mini-sub rotina	TMS320C25	aipint, bench gnucrypt gzip, hill jpeg, rx, set, cache, compress	84% ∼ 88%	+115%	–
Lefurgy	dicio-nários	PowerPC	SPECint95	61%	–	–
Lefurgy	dicio-nários	ARM	SPECint95	66%	–	–
Lefurgy	dicio-nários	i386	SPECint95	75%	–	–
Benini	sem cache	MIPS	Ptolomy	90%	–	-46%◇ ∼ -52%◇
Benini	com cache	MIPS	Ptolomy	72%	–	-50%
Lekatsas‡	com cache	Xtensa 1040	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	
Lekatsas‡	com cache	SPARC	compress, diesel, i3d, key, mpeg smo, trick	65%	-25%	-28%♠
Wanderley Netto	PDC-Com Packet	SPARC	Mediabench MiBench	72% ∼ 88%	-45%♣	-46%♣

† Quando não especificado a energia se refere àquela consumida na cache
‡ Não considera os tamanhos das tabelas na razão de compressão
♣Resultado obtido para um valor de $f = 50\%$ e uma memória pricipal que requer 30 ciclos de *clock* para o preenchimento de uma linha de cache.
♠Energia no sistema de caches (inclusive dados)
◇Energia total no sistema

Tabela 6.10: Comparação do PDC-ComPacket com outros métodos PCD

Autor Principal	Projeto	Arquitetura	Suite de *benchmarks*	Razão de Compressão	Tempo de Execução
ARM	Thumb	ARM	–	55% ∼ 70%	-30%
MIPS	MIPS16	MIPS	–	60%	–
Wanderley Netto	PDC-ComPacket	SPARC	Mediabench & MiBench	72% ∼ 88%	-45%♣

♣Resultado obtido para um valor de $f = 50\%$ e uma memória pricipal que requer 30 ciclos de *clock* para o preenchimento de uma linha de cache.

Tabela 6.11: Comparação do PDC-ComPacket com ISAs mistos de 16/32 bits

6.5 Implementação do Hardware Descompressor

A implementação do descompressor em hardware do PDC-ComPacket foi feita em um *kit* de desenvolvimento XESS XSV800 [95]. O processador sintetizado é o Leon [91], um *core SPARCv8 compliant*. O Modelo VHDL do processador é configurável e particularmente adequado para *System-on-Chip*. O código fonte está disponível na forma de licença GNU-LGPL, permitindo uso gratuito para pesquisas e/ou aplicações comerciais.

6.5.1 O processador Leon

O Leon foi desenvolvido originalmente para a Agencia Espacial Européia (ESA), e posteriormente foi disponibilizado para domínio público. O processador possui as seguintes características:

- *Pipeline* de 5 estágios implementando o conjunto de instruções do SPARCv8
- Unidades de multiplicação, divisão e MAC em hardware;
- Cache de instruções e dados separadas;
- Módulo de *snooping* da cache de dados;
- Barramentos AHB e APB AMBA-2.0;
- Controlador de memória externa programável com 8/16/32 bits;
- Periféricos *on-chip*, como temporizadores, controladores de interrupções e porta de entrada/saída de 16 bits;
- Unidade de depuração *on-chip*;
- Modo de operação de baixo consumo de energia; e
- Interface com unidade de ponto flutuante Meiko e co-processador definido pelo usuário.

6.5.2 O *kit* de desenvolvimento XESS XSV800

O kit de desenvolvimento usado para prototipagem foi o XESS SXV800 que possui as seguintes características:

- FPGA Virtex XCV800 da Xilinx;
- CPLD XC95108, usada para controle e programação da FPGA;
- Dois bancos de memória SRAM de 512k x 16;
- Oscilador programável para até 100MHz; e
- Conectores para interface paralela e serial

A ferramenta de síntese utilizada foi a ISE específica do fabricante para FPGA Xilinx.

6.5.3 Descrição da implementação do hardware do PDC-ComPacket

Embora a implementação do hardware descompressor no simulador tenha seguido o modelo no Capítulo 5, a implementação em FPGA é uma evolução natural daquela idéia. Como já comentamos, o descompressor possui uma unidade de cálculo de PC bastante complexa que precisa fornecer os endereços para cache a partir dos endereços do processador. Além disto, ele precisa saber quando acontece uma interrupção a fim de preservar seu estado após a execução da rotina de tratamento associada. Ainda, é preciso usar instruções especiais para correção de endereços de retorno de chamadas de funções. O pior de tudo é que um ciclo de *clock* é perdido quando um salto é tomado.

Por todos estes motivos, um novo projeto foi utilizado na implementação em FPGA. Este projeto integra o descompressor dentro da *datapath* do processador, assim evitando ciclos extras devido aos saltos. O desafio então é como integrar o descompressor e não alterar o tempo de ciclo do processador. De fato, o descompressor necessita de um tempo menor que um ciclo para realizar uma descompressão, mas se somado ao tempo gasto no estágio de *fetch* ou ao gasto no *decode*, o ciclo do processador precisaria aumentar. Para evitar esta interferência, a opção foi quebrar as tarefas do descompressor em dois estágios e distribuí-las nos estágios de *fetch* e *decode* do Leon.

A maior parte das tarefas é executada ainda no *fetch*. No *decode* apenas instruções de salto pertencentes ao dicionário são montadas e entregues para decodificação. A Figura 6.8

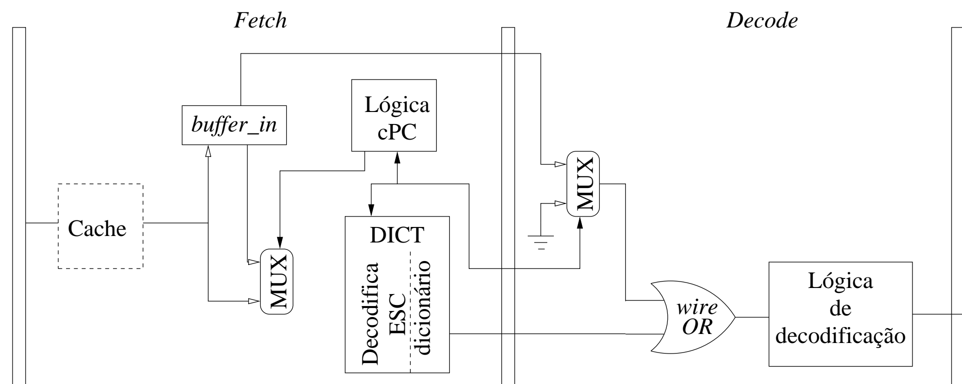


Figura 6.8: Diagrama de blocos do descompressor PDC-ComPacket para FPGA

mostra um diagrama de blocos do descompressor indicando onde cada tarefa é realizada. No estágio de *fetch* a instrução recebida da cache é enviada ao bloco DICT e ao mesmo tempo ao *buffer* de entrada (Buffer_in). O bloco DICT reconhece a instrução. Sendo um ComPacket ele habilita a escrita em Buffer_in. O bloco "Lógica cPC"tem um contador para instruções comprimidas que faz as vezes de um PC. A propósito, este bloco segura o sinal de *hold* do registrador PC do processador, impedindo que o mesmo evolua no caso da próxima instrução a ser executada ainda esteja no ComPacket.

A próxima instrução a ser descomprimida vem direto do Buffer_in, evitando acessos à cache.

No estágio de *decode* do processador, apenas uma tarefa é realizada quando há compressão. Os bits do *offset* de uma possível instrução de desvio são anexados com um *wire-OR* aos bits significativos (de uma instrução de salto) retirados do dicionário.

De certa forma, este diagrama de blocos contém os diagramas mostrados na Figura 5.14. A implementação do Hardware descompressor está sendo feita de forma colaborativa com um dos alunos de Mestrado do Laboratório de Sistemas de Computação.

6.6 Considerações Finais e Conclusões

As medidas de desempenho do PDC-ComPacket foram obtidas através de simulação intensiva dos programas. O simulador, embora não otimizado para um nível comercial, apresenta desempenho bom em comparação com o tsim-leon (versão comercial disponível no *site* do Leon). Em uma estação simples (Pentium III/0.8GHz, 128M de memória) ele processa em média 1 milhão de instruções por segundo, enquanto o tsim chega a 3

milhões. Considerando que o número de relatórios gerados é muito maior no LeonSim, o desempenho do simulador foi suficiente para este trabalho (no pior caso, menos de uma dezena de minutos foi utilizada para executar uma aplicação).

Algumas otimizações também podem ser feitas no descompressor. Uma delas é a implementação de uma política de predição de saltos evitando ciclos extra no descompressor para todos os saltos tomados (caso o mesmo seja implementado fora do processador).

Um detalhe também interessante é que a taxa de falhas na cache não cai com este método. Embora o número de falhas observado seja menor que no código sem compressão, o fato do uso do *buffer* de entrada diminuir também o número de acessos à cache impede uma redução na taxa de falhas (dado que a redução de acessos foi proporcional a redução de falhas).

O modelo de energia utilizado se restringiu à memória cache de instruções. Outras possíveis reduções de energia podem ser observadas na diminuição dos ciclos de espera do processador, além é claro, da redução de uso da memória principal e nos barramentos.

Um outro problema com as medidas de energia se referem às perdas no descompressor. Usando código comprimido, estamos diminuindo acessos à cache e fazendo uso contínuo do dicionário. É intuitivo que o consumo por acesso na cache seja maior que o consumo por acesso no dicionário, mas estamos trocando acessos à cache, por acessos ao dicionário sem evitar completamente o primeiro. Os acessos ao dicionário consomem parte do que foi economizado nas caches para os nossos experimentos (se consideramos os acessos ao dicionário o consumo de energia vai em média para 94% do valor original, para um valor de $f = 50\%$).

Apesar deste efeito negativo, consideramos que os resultados não são de todo ineficazes, ou mesmo pífios. A razão para esta consideração é que estamos trabalhando com uma escala de tamanhos de programas muito pequena e, portanto, as caches escolhidas também são pequenas. Ora, para uma cache de 128 bytes, o consumo de energia por acesso é de 2mJ enquanto o dicionário consome 1mJ, ou seja, 50% do primeiro valor. Se escalonarmos o tamanho da cache até chegarmos a um valor mais próximo de uma implementação real, como por exemplo, 256k bytes, a energia consumida por acesso ao dicionário fica em apenas 3% daquela consumida na cache. Assim, a troca de acessos na cache por acessos ao dicionário não teria um impacto forte nas medidas de redução de energia.

O método PDC-ComPacket se aproxima dos melhores resultados de compressão apontados na literatura, bem como se mostra mais eficiente na redução de consumo de

energia e aumento do desempenho que seus pares. Esta frase se baseia naturalmente em um olhar cético sobre os resultados finais obtidos.

Para uma comparação mais justa, seria necessário aproximar as plataformas de hardware e software onde foram baseados os experimentos. De qualquer forma, pelas razões apresentadas em nossa análise de resultados, não seria de todo injusto afirmar que o PDC-ComPacket apresenta o melhor compromisso entre compressão, desempenho e consumo de energia, se comparado com seus pares.

Concluímos ainda, corroborando o Capítulo 4, que se basear apenas em um tipo de estatística para construção de um dicionário pode comprometer o conjunto de aspectos envolvidos na construção de um sistema embarcado completo. Nossa abordagem, a primeira do gênero a mesclar informações de *profile* estático e dinâmico, aproveita melhor a tríade de requisitos (área-consumo de energia-desempenho) mais comumente encontrada nos projetos de sistemas embarcados.

Capítulo 7

Conclusões e Trabalhos Futuros

Neste trabalho apresentamos um novo método de construção de dicionários baseado em informações estatísticas retiradas dos programas com *profile* estático e dinâmico. Esta nova abordagem leva ao aproveitamento dos principais resultados da compressão de código, seja para redução da área de memória utilizada pelos programas, seja para redução do consumo de energia na cache de instruções, ou seja, para melhoria do desempenho, simultaneamente.

Ainda, apresentamos um novo método de compressão de código, o PDC-ComPacket, e projetamos um descompressor associado. Este método apresentou uma razão de compressão variando entre 71,6% e 88,2% em média para a arquitetura SPARCv8, usando um conjunto de *benchmarks* retirados de duas suites, *Mediabench* e *MiBench*. Além disto, a redução do consumo de energia na cache de instruções foi estimada e obtivemos um percentual de 46% de redução. Finalmente, para um conjunto de memórias com diferentes características de respostas à cache, o método foi capaz de melhorar o desempenho do sistema em até 52% em média.

Apesar do método ter sido testado para uma combinação de plataformas de hardware e software específicas, o PDC-ComPacket apresentou os melhores resultados de redução de energia na cache de instruções e aumento de desempenho da literatura. A comparação direta, não é de todo irrazoável se acompanhada de elementos que indiquem sua validade, como a outrora estabelecida entre os seus pares PDC.

Uma plataforma de software foi desenvolvida, para suportar os experimentos bem como para gerar as estatísticas para o método de compressão. A partir dela uma ferramenta também desenvolvida por nós realiza a compressão do código. Todo este conjunto de ferramentas perfaz 15.000 linhas de código C++. Foi feito um porte da ferramenta de

compressão para o IBC-CDM, o programa tiny, também desenvolvido no LSC-IC, para que o mesmo pudesse comprimir código a partir do uso de estatísticas dinâmicas ou *multi-profile*. O *patch* para o tiny tem menos de 1.000 linhas de código C++, perfazendo o novo tiny, praticamente 16.000 linhas de código.

Conforme comprovado pelos experimentos aqui expostos, o uso de compressão de código não apenas pode aliviar os requisitos de memória, mas também promover uma redução no consumo de energia e aumentar o desempenho de um sistema. Se usada com base em uma abordagem *multi-profile*, os melhores *trade-offs* podem ser obtidos para tríade desempenho-consumo de energia-compressão.

7.1 Trabalhos Futuros

Embora o método de compressão se apresente bastante eficaz, um conjunto de otimizações pode ser usado para melhorias puntuais:

- do ponto de vista do desempenho do IBC-CDM, um aumento no tamanho da ATT pode levar a uma redução no *overhead* de ciclos de execução. Isto ocorre porque o número de acessos à memória principal decai se evitarmos a presença de linhas de cache não base.

- embora razoável, a abordagem de adaptação do uso de um dicionário *multi-profile* para o IBC-CDM, pode ser reavaliada. O fato da escolha de uma média ponderada para determinar a contribuição das instruções ser plausível, não descarta possibilidades como manter a separação em classes baseada somente na contagem estática, ou mesmo somente na contagem dinâmica do uso de instruções.

- para o PDC-ComPacket um algoritmo baseado em programação dinâmica poderia melhorar a escolha dos formatos dos ComPackets assim promovendo uma redução na razão de compressão e possivelmente nos aspectos dinâmicos derivados do uso da compressão de código.

- uma outra melhoria no PDC-ComPacket poderia surgir a partir do reescalonamento de instruções dentro do código, com o objetivo de agrupar mais instruções pertencentes ao dicionário.

Numa outra vertente de trabalho, o compilador poderia ser adaptado de tal forma a promover uma melhor compressão de código. Isto seria possível, por exemplo, com

uma nova estratégia de alocação de registradores, para tornar o número de instruções redundantes maior.

Um estudo envolvendo também outros pontos para medida do consumo de energia, e/ou a integração de um estimador dentro do simulador ajudaria na obtenção de uma visão mais global desta variável.

Neste trabalho, estudamos o comportamento do IBC em programas menores. Investigar o comportamento do PDC-ComPacket no SPECint95 e em outra arquitetura alvo seria agora interessante.

Pela primeira vez foi apresentado um estudo do impacto dos dados no desempenho de uma arquitetura PDC (neste trabalho). Porém seria interessante averiguar o comportamento do método com uma massa de dados maior, inclusive variando os tamanhos dos dados de entrada.

Por fim, uma análise de compatibilidade entre os métodos poderia informar se seria possível utilizar o IBC e o PDC-ComPacket ao mesmo tempo. O objetivo seria aproveitar a redução de acessos à cache do PDC-ComPacket e ao mesmo tempo a redução de acessos à memória principal do IBC.

Referências Bibliográficas

[1] L. Benini, A. Macii, and M. Poncino. Energy-aware design of embedded memories: a survey of tecnologies, architecture and optimization techniques. *ACM Transactions on Embedded Computing Systems*, 2(1):5–32, 2003.

[2] E. Wanderley Netto, R. Oliveira, and O. Saotome. RISC processor for digital signal processing. In *Proc. Int'l Conference on Digital Signal Processing Application and Tecnology*, pages 805–809, October 1995.

[3] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. Int'l Symp. on Microarchitecture*, pages 81–91, November 1992.

[4] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[5] D. Huffman. A method for construction of minimum redundancy codes. In *Proc. of the IEEE*, volume 40, pages 1098–1101, 1952.

[6] J. Storer. NP-completeness results concerning data compression. Technical Report TR-234, EECS Department, Princeton University, 1977.

[7] K. Sayood. *Introduction to Data Compression.* Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[8] R. Azevedo. *Uma arquitetura para Código Comprimido em Sistemas Dedicados.* PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, June 2002.

[9] W. Wilner. Burroughs B1700 memory utilization. In *Fall Joint Computer Conference*, pages 579–586, 1972.

[10] Digital Equipment Corp. *VAX Architecture Handbook.* Digital Equipment Corp., 1979.

[11] S. Debray, W. Evans, R. Muth, and B. Sutter. Compiler techniques for code compression. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[12] P. Weiner. Linear pattern matching algorithms. In *Proc. Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[13] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[14] E. Ukkonen. Constructing suffix-trees on-line in linear time. *Information Processing*, I(92):484–492, 1992.

[15] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools.* Addison Wesley, 1988.

[16] T. Standish, D. Harriman, D. Kibler, and J. Neighbors. The irvine program transformation catalogue. Technical report, University of California, Irvine, 1976.

[17] W. Wulf, R. Johnsson, C. Weinstock, C. Hobbs, and C. Geschke. *Design of an Optimizing Compiler.* Elsevier North-Holland, 1975.

[18] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, 1995.

[19] J. Ernst, C. Fraser, W. Evans, S. Lucco, and T. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.

[20] W. Evans and C. Fraser. Bytecode compression via profiled grammar rewriting. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 148–155, June 2001.

[21] S. Lucco. Split-stream dictionary program compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 27–34, June 2000.

[22] C. Fraser. Automatic inference of models for statistical code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 242–246, May 1999.

[23] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18(3):235–253, May 1996.

[24] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT press, 1990.

[25] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proc. Int'l Conf. on Computer Design*, pages 270–277, October 1994.

[26] C. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, Jul. and Oct. 1948.

[27] M. Beneš, A. Wolfe, and S. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proc. Conf. on Advanced Research in VLSI*, pages 219–236, September 1997.

[28] M. Beneš, S. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 43–56, September 1998.

[29] D. Kirovski, J. Kin, and W. Mangione-Smith. Procedure based program compression. In *Proc. Int'l Symp. on Microarchitecture*, pages 194–203, December 1997.

[30] R. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Proc. Data Compression Conference*, pages 362–371, April 1991.

[31] J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, January 1976.

[32] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[33] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 4.1.* International Business Machines (IBM) Corporation, March 2001.

[34] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6):807–812, September 1998.

[35] C. Lefurgy, E. Piccininni, and T. Mudge. Evaluation of a high-performance code compression method. In *Proc. Int'l Symp. on Microarchitecture*, pages 93–102, November 1999.

[36] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *Proc. ACM/IEEE Design Automation Conference*, pages 516–521, June 1998.

[37] H. Lekatsas and W. Wolf. Random access decompression using binary arithmetic coding. In *Proc. Data Compression Conference*, pages 306–315, March 1999.

[38] H. Lekatsas and Wayne Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on CAD*, 18(12):1689–1701, December 1999.

[39] G. Araújo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE Transactions on VLSI Systems*, 8(5):530–533, March 2000.

[40] P. Centoducatte. *Compressão de Programas Usando Árvores de Expressão*. PhD thesis, Instituto de Computação, Universidade Estadual de Campinas, 1999.

[41] R. Pannain. *Compressão de Código de Programa Usando Fatoração de Operandos*. PhD thesis, Faculdade de Engenharia Elétrica e Computação, Universidade Estadual de Campinas, 1999.

[42] P. Centoducatte, R. Pannain, and G. Araújo. Compressed code execution on DSP architectures. In *Proc. ACM/IEEE Int'l Symp. on System Synthesis*, pages 56–61, November 1999.

[43] K. Lin, J. Shann, and C-P. Chung. Code compression by register operand dependency. In *Proc. of the Workshop on Interaction Between Compilers and Computer Architecture*, pages 91–101, February 2002.

[44] M. Hampton and M. Zhang. Cool code for hot RISC. [OnLine], October 2003. Available: http://cag.lsc.mit.edu/6.893-f2000/project/hampton_final.pdf.

[45] R. Pannain, P. Centoducatte, and G. Araújo. Using operand factorization to compress dsp programs. In *Proc. of the Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, pages 223–229, November 1999.

[46] C. Lefurgy and T. Mudge. Fast software-managed code decompression. In *Proc. of the 2nd International Workshop on Compiler and Architecture Support for Embedded Systems.*, pages 139–143, October 1999.

[47] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *Proc. of the International Symposium on High-Performance Computer Architecture*, pages 218–227, January 2000.

[48] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. In *Proc. Int'l. Symp. on Low-Power Electronics and Design*, pages 206–211, 1999.

[49] D. Patterson and J. Hennessy. *Computer Organization and Design: the Hardware and Software Interface.* Morgan Kaufmann Publishers, second edition, March 1998. San Francisco/CA.

[50] L. Benini, A. Macii, and E. Macii. Exact and heuristic algorithm for low-energy code compression in performance and memory constrained embedded systems. In *Proc. of the Midwest Symposium on Circuits nad Systems*, pages 552–555, August 2001.

[51] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. In *Proc. Conf. on Advanced Research in VLSI*, pages 272–285, March 1995.

[52] J. Storer and T. szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, October 1982.

[53] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, pages 117–121, June 1984.

[54] S. Liao, K. Keutzer, and S. Devadas. A text compression based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, January 1999.

[55] E. Wanderley Netto. Uma arquitetura RISC para o processamento digital de sinais. Master's thesis, Instituto Tecnológico de Aeronáutica, October 1995.

[56] C. Lefurgy, P. Bird, I-C. Chen, and T. Mudge. Improving code density using compression techniques. In *Proc. Int'l Symp. on Microarchitecture*, pages 194–203, December 1997.

[57] C. Lefurgy and T. Mudge. Code compression for DSP. Technical Report CSE-TR-380-98, EECS Department, University of Michigan, November 1998.

[58] ADI. *SHARC User's Manual.* Analog Devices, 1998.

[59] H. Lekatsas, Joerg Henkel, and Wayne Wolf. Arithmetic coding for low power embedded system design. In *Proc. Data Compression Conference*, pages 430–439, March 2000.

[60] H. Lekatsas, Joerg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proc. ACM/IEEE Design Automation Conference*, pages 294–299, 2000.

[61] H. Lekatsas, J. Henkel, and W. Wolf. Design and simulation of a pipelined decompression architecture for embedded systems. In *Proc. ACM/IEEE Int'l Symp. on System Synthesis*, pages 63–68, October 2001. (To appear).

[62] H. Lekatsas, J. Henkel, and W. Wolf. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *Proc. ACM/IEEE Design Automation Conference*, pages 34–39, June 2002.

[63] H. Lekatsas, J. Henkel, and W. Wolf. 1-cycle code decompression cicuitry for performance increase of xtensa-1040-based embedded systems. In *Proc. of the IEEE Custom Integrated Circuits Conference*, pages 9–12, May 2002.

[64] E. Killian and F. Warthman. *Xtensa Instruction Set Architecture Reference Manual.* Tensilica, 2001.

[65] L. Benini, A. Macii, and A. Nannarelli. Code compression for cache energy minimization in embedded systems. *IEE Proceedings on Computers and Digital Techniques*, 149(4):157–163, July 2002.

[66] J. Bunda, D. Fussell, R. Jenevein, and W. Athas. 16-bit vs. 32-bit instructions for pipelined microprocessors. Technical Report TR-92-39, The University of Texas at Austin, Department of Computer Sciences, 1992.

[67] ARM. *An Introduction to Thumb.* Advanced RISC Machines Ltd., March 1995.

[68] A. Krishnaswamy and R. Gupta. Profile guided selection of ARM and thumb instructions. In *Proc. of the joint Conf. on Languages, Compilers and Tools for Embedded Systems: software and Compilers for Embedded Systems*, pages 56–64, June 2002.

[69] K. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proc. of Real Time Systems*, pages 559–571, 1997.

[70] S-J. Nam, I-C. Park, and C-M. Kyung. Improving dictionary-based code compression in VLIW architectures. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A(11):2318–2324, November 1999.

[71] T. Conte, S. Banerjia, S. Larin, K. Menezes, and A. Sathaye. Instruction fetch mechanisms for vliw architectures with compressed encodings. In *Proc. Int'l Symp. on Computer Architecture*, pages 201–211, November 1996.

[72] Y. Xie, W. Wolf, and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. In *Proc. ACM/IEEE Int'l Symp. on System Synthesis*, pages 138–143, October 2002.

[73] Y. Xie, H. Lekatsas, and W. Wolf. A code decompression architecture for VLIW processors. In *Proc. Int'l Symp. on Microarchitecture*, pages 66–75, December 2001.

[74] C. Fraser and T. Proebsting. Custom instruction sets for code compression. [OnLine], October 2003. http://research.microsoft.com/˜toddpro/papers/pldi2.ps.

[75] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):520–540, April 1986.

[76] P. Elias. Interval and recency rank source coding: Two on-line adaptive variable-length schemes. *IEEE Transactions on Information Theory*, 33(1):3–10, January 1987.

[77] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[78] M. Drinic, D. Kirovski, and H. Vo. Code optimization for code compression. In *Proc. Int'l. Symp. on Code Generation and Optimization*, pages 315–324, March 2003.

[79] P. Howard and J. Vitter. Design and analysis of fast text compression based on quasi-arithmetic coding. In *Proc. Data Compression Conference*, pages 98–107, 2003.

[80] S. Debray and W. Evans. Profile-guided code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 95–105, June 2002.

[81] Y. Xie, W. Wolf, and H. Lekatsas. Profile-driven selective code compression. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, pages 462–467, March 2003.

[82] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karisto. Survey of code-size reduction method. *ACM Computing Surveys*, 35(3):223–267, September 2003.

[83] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia communication systems. In *Proc. Int'l Symp. on Microarchitecture*, pages 330–337, December 1997.

[84] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, and T. Mudge. Mibench: a free, commercially representative embedded benchmark suite. In *Proc. of the IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, December 2001.

[85] SPARC International Inc. *SPARC V8 Architecture Manual*.

[86] G. Gaisler. Leccs. [OnLine], October 2003. Available: http://www.gaisler.com.

[87] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Mixed static/dynamic profiling for dictionary based code compression. In *Proc. of the International Symposium on System-on-Chip*, pages 159–163, November 2003.

[88] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, June 2000.

[89] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In *Proc. ACM/IEEE Design Automation Conference*, June 2004. (To appear).

[90] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Code compression to reduce cache accesses. Technical Report IC-03-23, Instituto de Computação - UNICAMP, November 2003.

[91] G. Gaisler. Leon. [OnLine], October 2003. Available: http://www.gaisler.com.

[92] S. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solic-State Circuit*, 31(5):677–688, May 1996.

[93] D. Patterson and J. Hennessy. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003. San Francisco/CA.

[94] J. Montanaro, R. Witek, K. Anne, A.Black, E. Cooper, D. Dobberpuhl, P. Donahue, J. Eno, G. Hoeppner, D. Kruckemyer, T. Lee, P. Lin, L. Madden, D. Murray, M. Pearce, S. Santhanam, K. Snyder, R. Stephany, and S. Thierauf. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *IEEE Journal of Solic-State Circuit*, 31(11):1703–1714, November 1996.

[95] XESS Corporation. Xsv800. [OnLine], May 2004. Available: http://www.xess.com.