Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Sistemas Integráveis e Fotônica

Compressão de Código de Programa Usando Fatoração de Operandos

Ricardo Pannain

Tese de Doutorado

Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-orientador: Prof. Dr. Furio Damiani.

Junho de 1999

Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Compressão de Código de Programa Usando Fatoração de Operandos

Ricardo Pannain

Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-orientador: Prof. Dr. Furio Damiani.

Tese apresentada à Faculdade de Engenharia Elétrica e Computação da Universidade Estadual de Campinas, para obtenção do título de Doutor em Engenharia Elétrica.

Campinas
Junho de 1999

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

P195c

Pannain, Ricardo
Compressão de código de programa usando fatoração de operandos. / Ricardo Pannain.--Campinas, SP: [s.n.], 1999.

Orientadores: Guido Costa Souza de Araújo, Furio Damiani.
Tese (doutorado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Compressão de arquivos. 2. Sistemas embutidos de computador. 3. Circuitos integrados – Integração em escala muito ampla. I. Araújo, Guido Costa Souza de. II. Damiani, Furio. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Banca Examinadora

Prof. Dr. Guido Costa Souza de Araújo – Orientador (IC/UNICAMP)

Prof. Dr. Claudionor José Nunes Junior – (DCC/UFMG)

Prof. Dr. Mário Lúcio Côrtes – (IC/UNICAMP)

Prof. Dr. Carlos Alberto dos Reis Filho – (FEEC/UNICAMP)

Prof. Dr. José Raimundo de Oliveira – (FEEC/UNICAMP)

Agradecimentos

Agradeço a minha querida esposa Luciana e a meus queridos filhos Paulo, Roberto e Eduardo pelo apoio espiritual, pela compreensão nos momentos mais difíceis, pelo incentivo e por suportarem os momentos em que minha ausência se tornou uma constante.

Agradeço também ao meu orientador, Guido Costa Souza de Araújo, pela sua garra, profissionalismo e amizade; ao meu co-orientador, Furio Damiani, pela experiência e idéias passadas nas várias conversas; aos meus colegas Israel Geraldi, Paulo Centoducatte, Mário Lúcio Côrtes, Luís Felipe da Costa Antoniosi e Rogério Sigrist Silva, pelas idéias, pela colaboração e pela amizade; aos meus amigos e colegas do Instituto de Informática da PUC Campinas e do Instituto de Computação da UNICAMP, em especial à Angela de Mendonça Engelbrecht, Carlos Miguel Tobar Toledo e Frank Herman Behrens, pela ajuda na etapa final do trabalho.

Dedico este trabalho ao meu inesquecível pai que, onde quer esteja, sempre me acompanhou e torceu para que tudo isto acontecesse; à minha mãe, porque ser mãe já basta; à minha esposa e aos meus filhos.

A todos minha eterna gratidão.

SUMÁRIO

Resumo

O crescente uso de sistemas embutidos é uma evidência nos mercados de telecomunicações, multimídia e produtos eletro-eletrônicos em geral. Por serem sistemas utilizados em um mercado com grande volume de produção, as reduções de custo de projeto têm um impacto considerável no preço final do produto. Como resultado da necessidade de redução de custos, estes sistemas são muitas vezes implementados integrando um núcleo de um processador, um circuito de aplicação específica (*ASIC – Application Specific Integrated Circuits*) e uma memória de programa/dados em um único *chip* (*SOC - System-On-a-Chip*). Como os sistemas embutidos estão se tornando cada vez mais complexos, o tamanho dos seus programas vem crescendo de maneira considerável. O resultado é o aparecimento de sistemas nos quais a memória de programa ocupa uma grande área de silício, mais do que os outros módulos. Portanto, minimizar o tamanho do programa torna-se uma parte importante dentro dos esforços de projeto destes sistemas. Uma maneira de reduzir o tamanho do programa é projetar sistemas que possam executar código comprimido. Nós propomos uma técnica de compressão de código de programa chamada de fatoração de operandos. A idéia principal desta técnica é a separação das árvores de expressão do programa em seqüências de operadores representadas por um conjunto de instruções e de operandos representados pelo conjunto de registradores e imediatos das instruções. Com esta idéia, mostramos que tanto os padrões de árvores como de operandos têm distribuição de freqüência exponencial. Implementamos alguns experimentos para determinar a melhor técnica de codificação que explora esta característica. Os resultados experimentais mostraram, em média, uma razão de compressão de 43% para os programas pertencentes ao SPEC CINT95 com o processador MIPS R2000 e 67% para um conjunto de programas específicos executados no processador TMS320C25.

As máquinas de descompressão propostas montam os operadores e operandos em instruções não comprimidas utilizando uma combinação de dicionários e máquinas de estados.

Palavras-chave: Compressão de Código de Programa, Sistemas Embutidos, VLSI, DSP

CAPÍTULO 1

Introdução

1.1 Sistemas Embutidos

Os processadores podem ser divididos em duas grandes classes de acordo com a aplicação a que se destinam: (a) processadores para computação de propósito geral; (b) processadores para uso em sistemas dedicaodos. As aplicações na computação incluem computadores pessoais, *notebooks*, estações de trabalho e servidores. Suas principais características se concentram na possibilidade do usuário final poder progamá-los e na variedade das aplicações que eles podem executar. Sistemas embutidos são mais específicos. Eles são sistemas computacionais projetados para um domínio de aplicação específica e formam um segmento importante no mercado de telecomunicações, multimídia e equipamentos eletro-eletrônicos em geral, tornando os equipamentos destes nichos mais baratos e flexíveis.

Os processadores utilizados nestes sistemas podem ser dos seguintes tipos: microcontroladores (*MCU – Microcontroller Unit*), processadores utilizados em processamento digital de sinal (*DSPs – Digital Signal Processors*) e microprocessadores (*MPUs – Microprocessor Units*). Os microprocessadores são, geralmente divididos em computadores com conjunto de instruções complexas (*CISC – Complex Instruction-Set Computer*) e computadores com conjunto de instruções reduzidas (*RISC – Reduced Instruction-Set Computer*). Outra classe de processadores encontrados em sistemas embutidos é o processador com conjunto de instruções específicas projetadas para uma aplicação específica (*ASIP – Application Specific Instruction-set Processor*). Ele é um processador programável projetado para uma classe de aplicações específica e bem

definida, além de ser caracterizado pelo seu tamanho pequeno e pelo conjunto de instruções definido em concordância com a aplicação. Os *ASIPs* são geralmente encontrados em processamento de sinais em tempo real, em processamentos de imagens e em aplicações microcontroladas.

O mercado de sistemas embutidos, segundo dados da *International Data Corporation*, cresce 40% ao ano (circa 1998) [2]. Esta projeção torna este mercado altamente atraente, razão pela qual os esforços para torná-los mais competitivos, baratos e flexíveis, são plenamente justificáveis. Estes esforços se concentram na otimização da relação *custo x desempenho*. Uma das maneiras usadas, hoje em dia, para diminuir o custo decorrente de área, é implementar suas unidades: o circuito integrado de aplicação específica (*ASIC – Application Specific Integrated Circuits*), a memória para armazenamento de programa e dados e o núcleo de um processador, todos integrados em um único *chip* (*SOC – System-On-a-Chip*). Isto faz com que os custos caiam devido ao tamanho da produção, e que o desempenho seja melhor, pois todos os módulos estão integrados em um mesmo *chip*, além de melhorar a confiabilidade do sistema. A figura 1.1 esquematiza um sistema embutido implementado em um *SOC*. Para termos alta flexibilidade e competitividade de mercado, a estratégia de projeto tem sido a reutilização de sistemas (*hardware*) com alteração dos programas de controle (*software*).

O crescimento do tamanho das aplicações de sistemas embutidos tem, cada vez mais, resultado em programas complexos, com mais funções e, portanto, maiores, a despeito dos códigos otimizados gerados pelos compiladores atuais. Isto resulta no aumento da área de memória ocupada pelo programa influenciando o seu custo final.

A alta flexibilidade e competitividade necessitam de programas grandes implicando em uma memória maior e, conseqüentemente, aumentando a área de silício (custo). Portanto, a implementação destes sistemas em um *SOC* não é o bastante, é necessário diminuir a área final de silício (custo) através, por exemplo, da diminuição da área de memória que, pelas características dos programas, chega a ocupar mais da metade da área final do *chip*. O custo final de um *die* é proporcional ao cubo de sua área [3]. A diminuição do tamanho da memória de programa permite não só a diminuição do custo final do produto como a possibilidade de inserir mais funções ao sistema dentro de uma mesma área de silício.

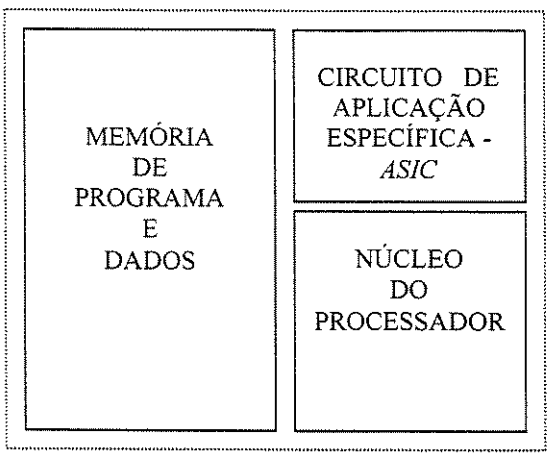


Figura 1.1 – Esquema de um sistema embutido implementado em um *SOC*

Duas abordagens podem ser usadas para reduzir o tamanho da memória. A primeira é através da utilização de alguns processadores *RISC* especiais como, por exemplo, o *Thumb* e *MIPS16* [4], e processadores especializados como, por exemplo, o *TMS320C25* [5]. Estes processadores têm instruções de pequeno tamanho e portanto produzem códigos com tamanho de 60% a 70% do tamanho do código para um *RISC* padrão, fazendo com que o tamanho da memória necessária para armazenar o programa seja 40% a 30% menor. A desvantagem desta abordagem é que, para estes processadores *RISCs*, não se conseguem *razões de compressão* (razão entre o tamanho do código comprimido e o tamanho do código original) menores que 50% e, para os processadores especializados, a desvantagem é que os seus respectivos compiladores não conseguem gerar códigos eficientes. A outra abordagem é projetar processadores que permitam a execução de código binário de programa comprimido. Para que isto seja viável, é necessário um circuito capaz de fazer a descompressão do código em tempo real. É esta abordagem que usamos neste trabalho.

O enfoque deste trabalho é compactar o código binário de forma a obter uma *razão de compressão* (razão entre o tamanho do código comprimido e o tamanho do código original) menor possível e, com isto, diminuir o tamanho da memória necessária para armazenar o programa de controle do sistema. As duas principais características que distinguem este tipo de compressão do problema de compressão tradicional são a necessidade de descompressão em tempo real, utilizando pouco recurso de memória e a possibilidade de

descompressão aleatória de palavras codificadas. A primeira característica é essencial para permitir que o processador execute as funções estabelecidas pelo programa e a segunda é necessária pois, como os programas têm instruções de desvios, a descompressão nem sempre é feita de forma seqüencial. Nós propomos uma técnica de compressão de código baseada no conceito de fatoração de operandos. A idéia principal está na fatoração dos padrões de operandos (*operand-pattern*) das árvores de expressão de um programa. As expressões fatoradas são chamadas de padrões de árvores (*tree-pattern*). Os padrões de árvores e os padrões de operandos são codificados separadamente. Esta idéia de fatoração é baseada no conceito de *padronização (patternization)* utilizada por Fraser et al [6] para compressão de *byte-code* para redes de máquinas virtuais.

A contribuição deste trabalho está na proposta de um algoritmo eficiente para codificação de programa, com características superiores às descritas na literatura, que permita a implementação de máquinas de descompressão eficientes, que trabalhem em tempo real.

1.2 Organização da Tese

Uma das maneiras de diminuir os custos ou aumentar as funções de um sistema embutido é comprimir o código de programa. Neste Capítulo, situamos o problema e mostramos as razões que nos levaram a seguir esta linha de trabalho. No próximo Capítulo, damos uma visão geral das arquiteturas dos processadores utilizados para validação do nosso método de compressão. No Capítulo 3, mostramos os métodos de compressão de dados, dentre os quais, os estatísticos, como o método de *Huffman*, e os de codificação por fatores, como por exemplo o de *Lempel-Ziv*, dentre outros. No Capítulo 4, apresentamos os mais importantes trabalhos publicados sobre compressão de código de programa, discutindo as suas contribuições, vantagens e desvantagens até o momento (circa 1998). No Capítulo 5, apresentamos o método de compressão de código usando fatoração de operandos assim como os resultados experimentais conseguidos com os processadores MIPS R2000 e TMS320C25. As máquinas de descompressão, para ambos os processadores são propostas no Capítulo 6 e, no Capítulo 7 mostramos os resultados experimentais. Finalmente, no Capítulo 8, apresentamos as conclusões e os possíveis trabalhos futuros.

CAPÍTULO 2

Processadores e Conjunto de Instruções

O objetivo de um processador é executar um conjunto de instruções que chamamos de programa. O processador busca as instruções na memória e as executa obedecendo a alguma seqüência. A execução de uma instrução é dividida em etapas: busca da instrução na memória (*fetch*), cálculo dos operandos e execução das operações especificadas pela instrução e posterior armazenamento dos resultados. A figura 2.1 ilustra a execução seqüencial de um programa mostrando as etapas de execução de uma instrução [7].

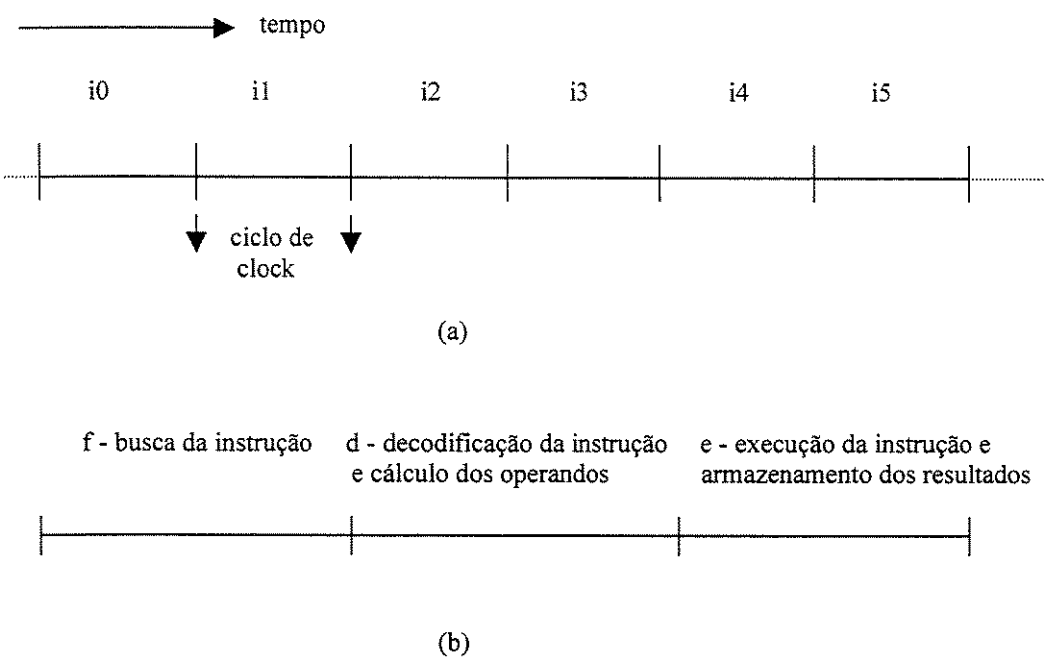


Figura 2.1 (a) Seqüência de execução de um programa ; (b) Etapas da execução de uma instrução

O tempo de execução de um programa, pelo processador, é determinado pelo tamanho do programa, pelo número médio de ciclos necessários para executar uma instrução e pelo tamanho do ciclo de um processador [7]. Entendemos por ciclo de um processador o período de um sinal (*clock*) que sincroniza as operações deste processador. O desempenho de um processador pode então ser otimizado melhorando um ou mais dos fatores vistos acima. Em 1981, Kogge [8]utilizou a técnica de *pipelining* para melhorar o desempenho de um processador. Esta técnica constitui em sobrepor as etapas de execução de uma instrução. O *pipeline* de instruções é a execução das diversas etapas do ciclo de execução de uma instrução por unidades independentes e que se comunicam entre si através de registradores sincronizados pelo *clock*. Estas unidades representam os estágios do *pipeline*. A figura 2.2 ilustra o conceito de *pipeline*.

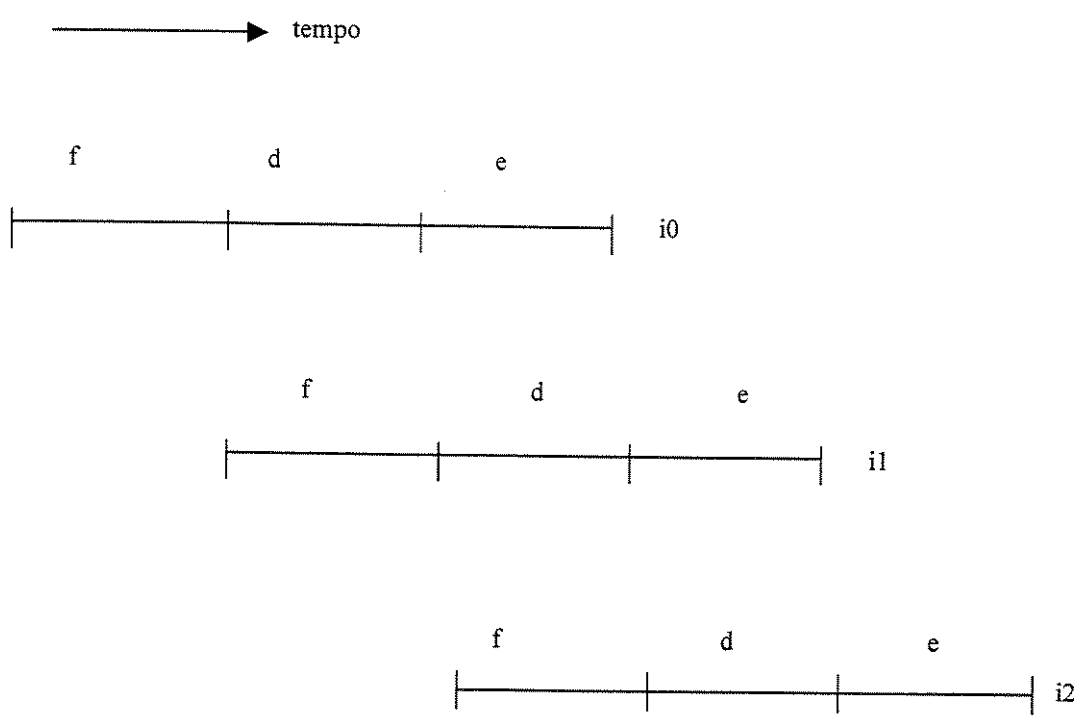


Figura 2.2 Execução de uma instrução em um processador com *pipeline*

A execução de instruções através de processadores com *pipeline* resulta numa diminuição do número médio de ciclos para a execução de uma instrução, embora o tempo de execução de uma determinada instrução seja o mesmo. A razão do pipeline apresentar um número médio de ciclos menor é que ele permite a manipulação simultânea de várias instruções pelo processador. O pipeline é uma tecnologia utilizada por quase todos os tipos de processadores atuais. Segundo Kogge [8], quando os primeiros microprocessadores foram desenvolvidos, a etapa de busca da instrução na memória levava muito mais tempo que as outras etapas, devido principalmente à tecnologia de memória existente na época e a sua interconexão com o processador. Esta foi a principal razão do aparecimento das arquiteturas com conjunto de instruções complexas (*CISC – Complex Instruction Set Computers*). O objetivo da arquitetura CISC foi aproveitar o tempo gasto na busca da instrução, que era grande, e projetar instruções que conseguissem executar mais operações. Isto aumentou o tempo das etapas de decodificação e de execução da instrução, fazendo com que a sobreposição destas etapas com a etapa de busca diminuisse o número médio de ciclos necessários para a execução de uma instrução. A figura 2.3 mostra a execução de um grupo de instruções por um processador CISC [7].

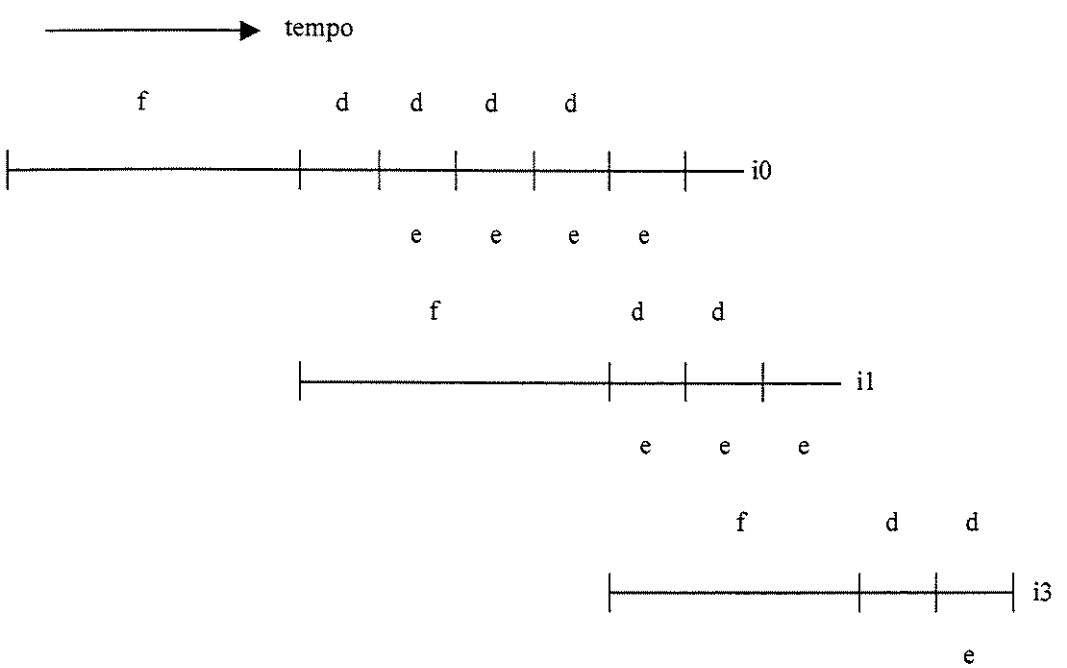


Figura 2.3 Pipeline em um processador CISC com limitação de tempo de acesso de memória

No fim dos anos 70 e início dos anos 80, as memórias e a tecnologia de encapsulamento mudaram rapidamente. Encapsulamento com grande número de pinos permitiram o projeto de interfaces de memória com tempos de acesso menores e memórias com maior densidade de integração [7]. As mudanças no encapsulamento fizeram com que Smith [8] propusesse a utilização de memórias *cache*, memória local de alta velocidade integrada com o processador. O resultado foi a diminuição drástica do tempo de busca da instrução na memória. A figura 2.4 mostra como ficou a execução de instruções em um processador *CISC* que utilizavam memórias *cache*. Nesta figura, podemos observar que o tempo limitante deixou de ser o tempo de busca e passou a ser o tempo de decodificação e execução.

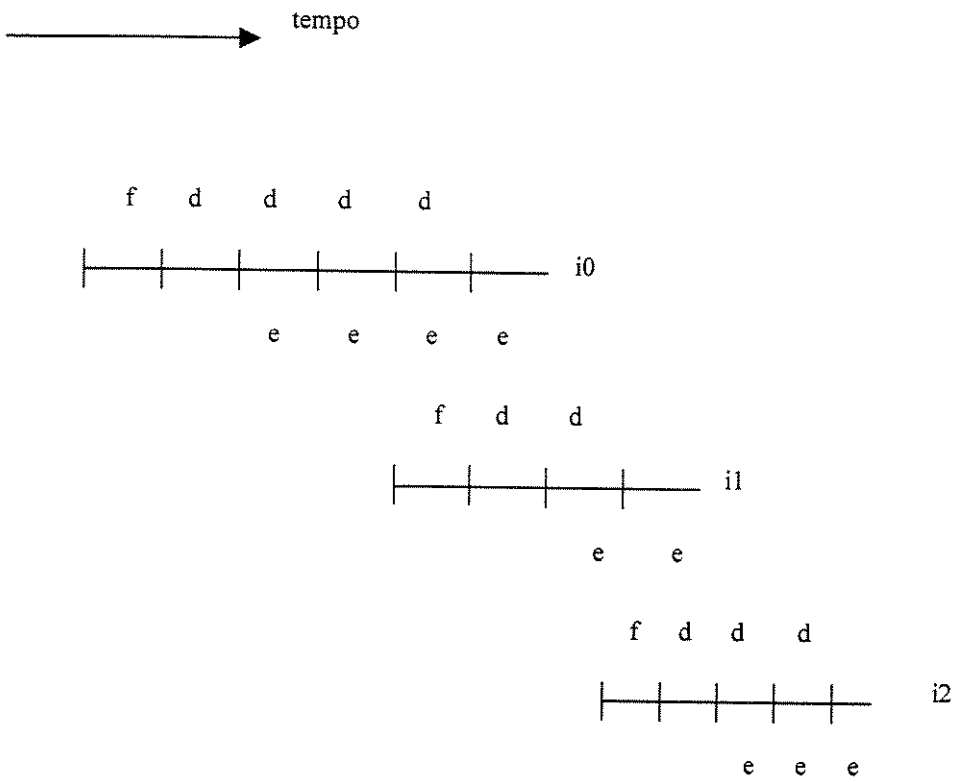


Figura 2.4 Execução de um grupo de instruções por um processador CISC sem limitações de memória

O avanço das tecnologias de memória e de encapsulamento motivou o aparecimento das arquiteturas com conjunto de instruções reduzidas (*RISC – Reduced Instruction Set Computers*). O objetivo deste tipo de arquitetura é a redução do número médio de ciclos para execução de uma instrução, em detrimento do tamanho do programa [3]. Em comparação com os processadores *CISC*, os processadores *RISC* reduzem o número de ciclos por instrução de um fator que pode variar de 3 a 5, enquanto propiciam um aumento no tamanho do código de 30% a 50% [7]. Além disso, os *RISCs* têm características auxiliares importantes, tais como, um número grande de registradores de propósito geral, memória cache de dados e programas que auxiliam o compilador a reduzir o número total de instruções ou que permitem reduzir o número de ciclos por instrução. Estes processadores dependem muito do grau de desenvolvimento das tecnologias de memória, de encapsulamento e de compiladores.

2.1 Processador *MIPS R2000*

Um exemplo de uma arquitetura *RISC* é o processador *MIPS R2000* [9]. Este processador foi um dos utilizados nos testes de validação deste trabalho pois, embora seja um processador antigo, ele é uma arquitetura *RISC* clássica que tem muitas das carcterísticas dos processadores *RISCs* modernos. Muitos dos processadores *RISCs* especiais utilizados em sistemas embutidos são derivados da arquitetura do *R2000*. Por exemplo, a empresa *MIPS Technologies Inc.* oferece, para uso em sistemas embutidos, um núcleo de processador chamado *JADE* que é um processador de 32 *bits* padrão *MIPS RISC* compatível com o processador *R3000* e com o sistema de gerenciamento de memória (*MMU – Memory Manegment Unit*) do R4000 [10]. O *JADE* é um núcleo que apresenta alto desempenho e baixo consumo de potência. Este núcleo é muito utilizado em sistemas embutidos nas áreas de automação de escritório, comunicação, gerenciamento de rede e equipamentos eletrônicos de consumo.

O processador *R2000* possui uma arquitetura *load–store* que trabalha com três tipos de instruções: tipo R para instruções operacionais; tipo J para instruções de desvios; e tipo I

para instruções com imediato e instruções *load-store*. Estes formatos podem ser vistos na figura 2.5 [9].

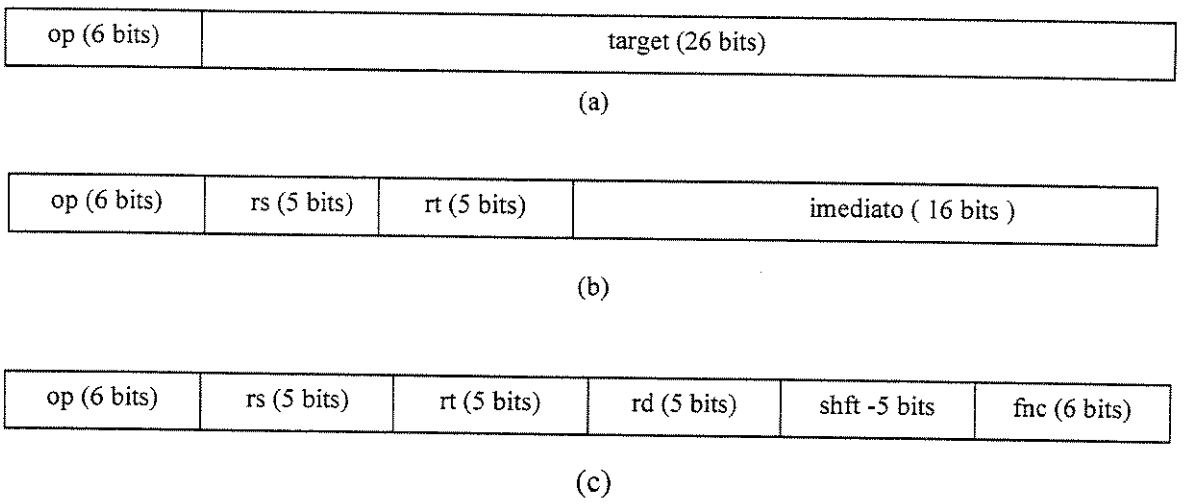


Figura 2.5 Formato de Instruções MIPS: (a) tipo J; (b) tipo I e (c) tipo R (RS = registrador fonte, RT = registrador alvo, RD = registrador destino, Shft = número de deslocamento e Fnc = especializa o tipo de instrução)

O conjunto de instruções MIPS pode ser dividido nos seguintes grupos:

- *Load/Store*: são instruções que movem dados entre memória e registradores. São do tipo I e o modo de endereçamento é formado pelo registrador base mais imediato (*offset*) de 16 *bits*
- *Operacional*: são instruções que executam operações lógicas, aritméticas e de deslocamento com valores armazenados nos registradores. Podem ser do tipo R, se operandos e resultado são os registradores e do tipo I, se um dos operandos for um imediato de 16 *bits*;
- *Jump* e *Branch*: são instruções que alteram a seqüência de execução de um programa. As instruções de *jump*, tipo J, são saltos para um endereço absoluto formado pela combinação de um *target* (26 *bits*) com 4 *bits* do PC (*Program Counter*). As de tipo R são

endereços de 32 *bits* armazenados nos registradores. As instruções de *branch* têm um *offset* de 16 bits relativo ao PC e são do tipo I;

- *Co-processador*: são instruções de ponto flutuante que usam o co-processador de ponto flutuante. Utilizam formatos diferentes dos formatos R, J ou I.
- *Special*: são instruções do tipo *syscall* – chamadas de rotinas do sistema e *rfe* – retorno de exceção. São geralmente do tipo R.

2.2 Processador *TMS320C25*

Processadores utilizados em processamento digital de sinal (*DSPs – Digital Signal Processors*) [11,12], são utilizados em um grande número de áreas, tais como: telecomunicações, computação gráfica, instrumentação, eletrônica de consumo e processamento de sinais. O crescimento do mercado de *DSPs* [14,15] tem demonstrado que estes processadores já são um nicho importante na indústria eletrônica atual.

Este tipo de arquitetura tem como característica a necessidade de executar uma operação de soma e uma de multiplicação em um único ciclo de máquina. A razão disto são as restrições que encontramos nas aplicações que envolvem processamento digital de sinal. O processador é projetado para que o tempo do ciclo de instrução seja igual ao tempo do ciclo de *hardware* para a maioria das instruções. O resultado disto é que todas etapas do ciclo de execução de uma instrução: busca da instrução (*instruction fetch*), decodificação, cálculo do endereço dos operandos, busca dos operandos e execução da instrução, são feitas em um único ciclo de máquina.

Estes processadores são do tipo *CISC* com a característica de ter instruções, de certa forma, já compactadas, ou seja, o número de tarefas executadas por instrução é bem maior se comparado, por exemplo, com uma instrução de um processador *RISC*. A desvantagem destes processadores é que os seus respectivos compiladores não geram códigos eficientes.

Como as unidades de ponto flutuante ocupam uma área de silício grande e necessitam mais ciclos de relógio para executarem um cálculo, consomem mais potência e são mais lentas que as unidades de ponto fixo. A maioria dos sistemas baseados em *DSPs* utilizam somente unidades de ponto fixo [11]. Pela necessidade de se executarem as instruções de

maneira rápida neste tipo de aplicação é necessário termos nas arquiteturas *DSPs*, um sistema de memória com desempenho que permita a execução de uma instrução em um ciclo de máquina. A unidade de memória é dividida em duas partes, uma para armazenar programas e a outra para dados que podem ser acessadas simultaneamente através de barramentos separados. A razão disto é permitir a busca simultânea de uma instrução e dos operandos de outra instrução que está executando no *pipeline*. As arquiteturas que têm a característica de armazenar dados e programas em memórias distintas são conhecidas como arquitetura *Harvard*. Utilizando esta abordagem, os *DSPs* são projetados com uma memória de dados RAM (fast static *RAM - Random Access Memory*) e uma memória de programa *ROM (Read Only Memory)*, ambas aptas a trabalhar com ciclo de acesso igual a um ciclo de máquina. Geralmente os processadores *DSPs* são baseados em arquiteturas memória-registrador como, por exemplo, a família TMS320C1x/C2x/C5x/C54x da Texas Instruments [9]. Nestas arquiteturas, os códigos binários das instruções utilizam um operando na memória e outro de um registrador qualquer armazenando sempre o resultado em um acumulador, como mostra a figura 2.6 [11].

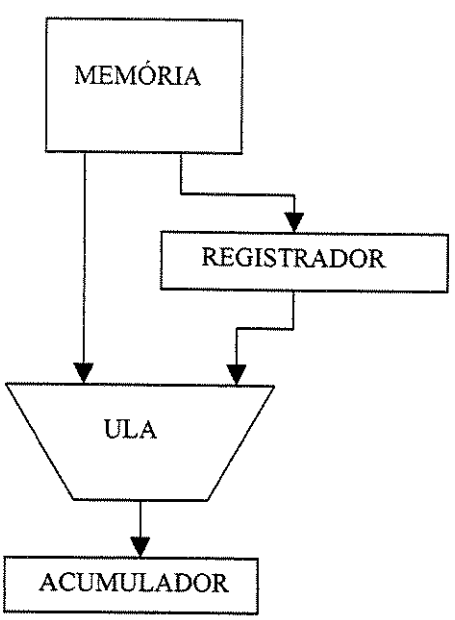


Figura 2.6 Arquitetura memória-registrador de um *DSP*

A maioria dos *DSPs* tem um conjunto de registradores (*register file*) especializados. A especialização é decorrente da necessidade de alto desempenho, devido aos cálculos de somatórias de fatores, e do baixo custo destas arquiteturas. Elas utilizam *pipeline* de instrução, podendo ter 3 a 4 estágios. Durante a busca da instrução (*instruction fetch* – IF), uma palavra de instrução é lida da memória de programa. No estágio de decodificação (*instruction decode* – ID), ela é decodificada e seus operandos são lidos da memória de dados e/ou registradores. Finalmente, no estágio de execução, a operação é computada. O *DSP* TMS320C2x [5] é um dos mais utilizados pela indústria de aplicações de processamento digital de sinal. Seu conjunto de instruções tem 3 modos de endereçamento: modo de endereçamento direto, modo de endereçamento indireto e modo de endereçamento imediato. Os dois primeiros modos são usados para acesso de dados na memória. Estes modos têm a vantagem reduzir o número de *bits* usados para codificar uma instrução através da diminuição do número de *bits* necessários para representar um endereço. No modo direto, os 7 *bits* menos significativos da palavra de instrução são concatenados com os 9 *bits* do ponteiro de página da memória de dados (DP) formando os 16 *bits* da palavra de endereço de memória. A figura 2.7 ilustra este processo. Todas as instruções podem utilizar este modo de endereçamento, exceto as instruções *CALL, as* instruções de desvio, as instruções que operam com imediatos e as instruções sem operandos. Este modo permite que as instruções tenham o tamanho de uma única palavra.

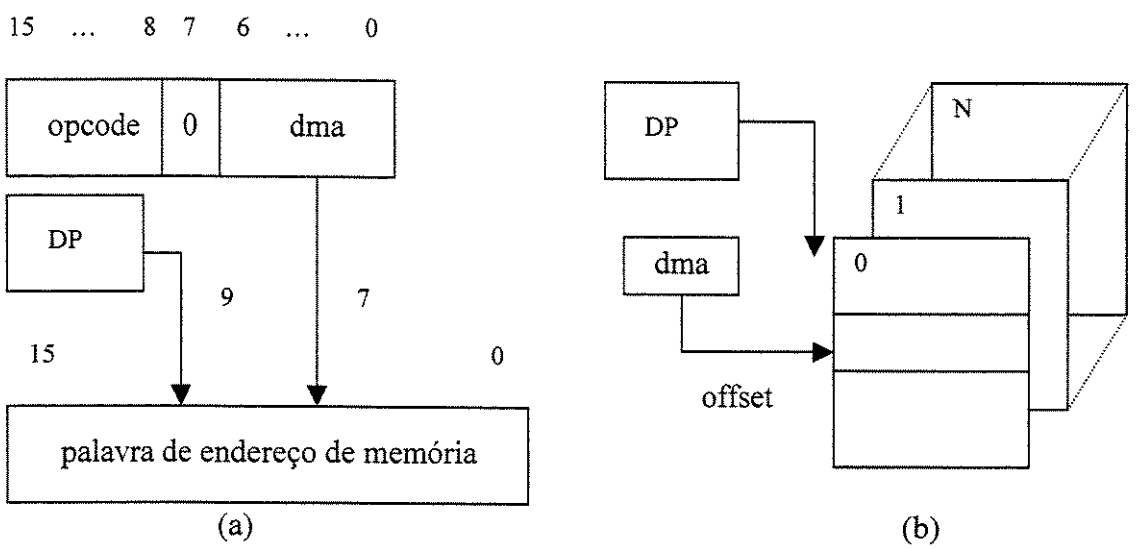


Figura 2.7 Diagrama do modo de endereçamento direto: (a) formação do endereço físico, (b) acesso a uma localização em uma página.

No modo de endereçamento indireto, são usados registradores auxiliares (AR0 – AR7) de 16 *bits* para fornecer a palavra de endereço do dado. Para selecionar um dos oito registradores auxiliares, utiliza-se um registrador (ARP) com o endereço do registrador auxiliar (AR). O ARP necessita 3 *bits*, pois tem que endereçar 8 registradores. O conteúdo dos registradores auxiliares pode ser modificado através de uma unidade aritmética de registradores auxiliares (ARAU), que executa operações aritméticas de 16 *bits*. As operações são executadas no mesmo ciclo da execução da instrução. A operação a ser executada em um AR só é feita após o uso deste mesmo AR pela instrução. A figura 2.8 mostra este esquema. Este método permite aumentar a velocidade de acesso a localizações vizinhas, características de matrizes de dados, estruturas muito utilizadas em processamento digital de sinal.

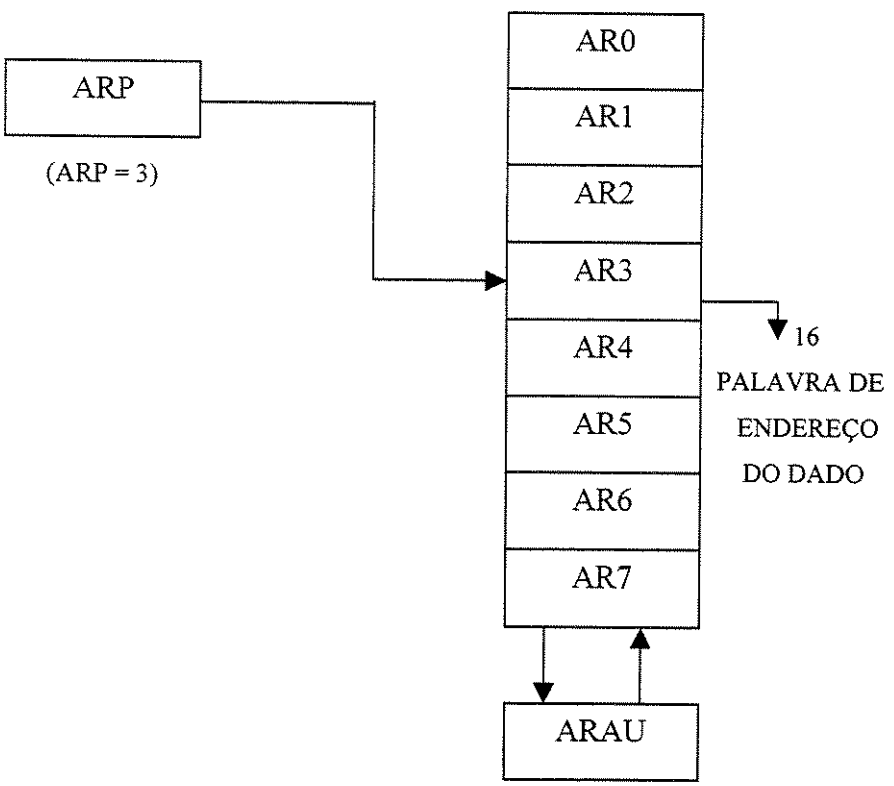


Figura 2.8 Diagrama do modo de endereçamento indireto

Todas as instruções podem utilizar este método de endereçamento, exceto as instruções que operam com imediatos e as instruções sem operandos. O formato das instruções com este modo de endereçamento pode ser visto na figura 2.9.

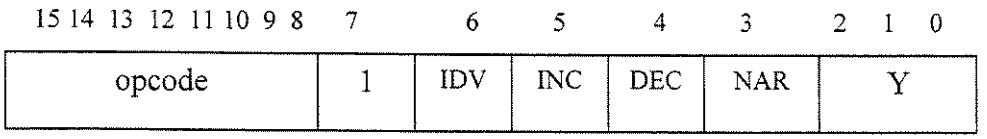


Figura 2.9 Formato de instrução com modo indireto de endereçamento

Na figura acima, os *bits* 8 a 15 representam o código da instrução. O *bit* 7 define o modo de endereçamento. Para o modo indireto seu valor tem que ser sempre 1. O conjunto de *bits* formado pelos *bits* 0 a 6 representa os *bits* de controle do modo indireto. O *bit* 6 indica quando o AR0 será usado para incrementar ou decrementar o registrador auxiliar corrente. Se ele for igual a 1, AR0 deverá ser somado ou subtraído do registrador auxiliar corrente. A soma ou subtração é controlada pelos *bits* 5 e 4 respectivamente. Os *bits* 3 a 0 são utilizados para atualizar o ARP. Se o *bit* 3 for igual a 1, o valor Y (*bits* 2 a 0) será o próximo valor de ARP; caso contrário, ele manterá seu valor. No modo de endereçamento imediato, a instrução contém o valor do operando imediato. As instruções podem ser de 16 *bits*, com constantes de 8 *bits* ou 13 *bits*, ou de 32 *bits*, com constante de 16 *bits*. O tamanho da constante depende da instrução. A figura 2.10 mostra estes formatos.

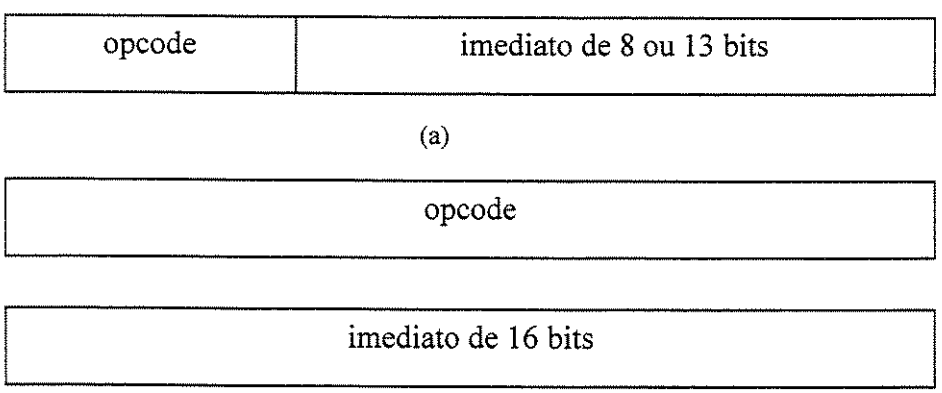


Figura 2.10 Formato de instrução no modo imediato de endereçamento:
(a) instrução com imediato curto; (b) instrução com imediato longo

CAPÍTULO 3

Métodos de Compressão de Dados

Neste capítulo, mostraremos os métodos mais utilizados em compressão de textos, que apesar de não serem próprios para compressão de código de programa, seus algoritmos são utilizados na codificação dos símbolos neste tipo de compressão.

O processo de compressão transforma um texto fonte em um outro de comprimento menor que o original. Isto é possível devido à redundância de informações que os mesmos apresentam. A redundância está presente na freqüência de símbolos, na existência de símbolos repetidos, nos padrões e na localidade de referência exibida por estes [16,17]. Entendemos por símbolo um caracter ou uma seqüência de caracteres que formam um texto a ser comprimido. A freqüência de símbolos pode ser explicada como a ocorrência de determinadas letras em um texto como, por exemplo, em um texto em português, a letra *a* ocorre mais vezes que a letra *z*. Na compressão, esta característica pode ser aproveitada atribuindo à letra *a* um código com menor número de *bits* que o código da letra *z*.

Em um texto, geralmente existem palavras ou cadeias de caracteres que se repetem. As palavras podem ser construções típicas como preposição, pronomes, e outras, ou mesmo palavras referentes ao assunto que se trata no texto. As cadeias de caracteres são geralmente construções próprias da linguagem como, por exemplo, na língua portuguesa, *qu*, *ção*, *ch*, *ss*, etc. Estas palavras e cadeias são chamados de *padrões*. Por *repetição de símbolos*, entende-se, por exemplo, uma cadeia de brancos ou zeros em textos comerciais, ou uma cadeia de *pixels* de mesmo valor em imagens. O último aspecto que determina o aparecimento da redundância é a *localidade de referência*, ou seja, a freqüência da ocorrência de padrões em partes específicas do texto.

O objetivo da compressão é usar a existência de redundância para associar um código a um padrão, ou combinação de padrões, de tal maneira que quanto maior sua freqüência, menor seja o tamanho deste código.

3.1 Entropia

A capacidade de um texto ser comprimido pode ser medida pela *entropia*. Podemos definir *entropia* como a menor quantidade de *bits* por *símbolos* necessária para guardar o conteúdo de informação da fonte e, portanto, para representar textos recuperáveis gerados por ela. Ela pode ser considerada um limite para a compressão e é usada como uma medida de eficiência para os métodos de compressão.

3.2 Modelos de Fonte de Dados

Fontes de dados são fontes que permitem gerar símbolos baseados em um *alfabeto de símbolos*. Um alfabeto de símbolos é o conjunto de símbolos válidos para uma determinada fonte. As fontes de dados podem ser as fontes de linguagens naturais, as de dados numéricos, as de imagens e outras. Elas, geralmente, podem ser classificadas como fontes independentes ou como fontes de *Markov*. Em fontes independentes, a probabilidade associada a um símbolo é independente daquelas de outros símbolos. As fontes *Markovianas* são aquelas em que a geração de um símbolo depende da geração dos símbolos anteriores. Uma fonte de *Markov* de *n-ésima* ordem é aquela em que a ocorrência do símbolo depende da ocorrência dos n símbolos anteriores [16].

3.3 Tipos de Modelagem de Dados

A modelagem dos símbolos de um texto pode ser classificada como *estática* ou *adaptativa*. Na modelagem estática, as probabilidades são fixas, isto é, a associação entre os símbolos e os códigos não muda durante a codificação. Aqui, tanto o compressor como o descompressor devem ter conhecimento do modelo de dados ou da tabela de códigos antes de iniciada a codificação. Este tipo de modelagem é utilizado em aplicações onde a

associação de códigos não pode variar. A modelagem adaptativa, ou dinâmica, utiliza uma tabela de símbolos inicial, geralmente construída considerando probabilidades iguais ou nulas para todos os símbolos. A partir daí, o compressor constrói novamente a tabela de códigos, a fim de acompanhar as mudanças de probabilidades apresentadas pelos símbolos. Este tipo de modelagem é utilizado em aplicações *on-line*, como comunicação de dados. Todos os métodos que utilizam esta última abordagem apresentam resultados iguais ou superiores aos que utilizam a modelagem estática [16].

3.4 Métodos Estatísticos de Compressão

Os métodos estatísticos consideram a fonte de dados bem caracterizada com seus códigos associados aos respectivos símbolos com base nas suas probabilidades. Estes métodos permitem a separação entre a modelagem e a codificação dos dados. Mostramos a seguir alguns métodos que utilizam esta abordagem.

3.4.1 Método Shannon-Fano

O método Shannon-Fano [51,52] foi apresentado por C. E. Shannon e por R. M. Fano em 1949. O objetivo deste método é associar códigos menores a símbolos mais prováveis e códigos maiores aos menos prováveis. Os códigos têm comprimentos variáveis, e os símbolos podem ter comprimentos fixos ou variáveis. A codificação parte da construção de uma árvore ponderada considerando o peso de cada símbolo, ou seja, a probabilidade de ocorrência do mesmo em um texto. As regras para construção da árvore são:

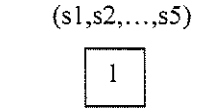
1. Montar uma lista com os símbolos em ordem decrescente de peso. Esta lista é associada à raiz da árvore;

2. Dividir a lista em duas sublistas de tal maneira que as somas dos pesos de cada uma delas seja mínima, isto é, a soma de uma delas deve ser aproximadamente igual à soma da outra;

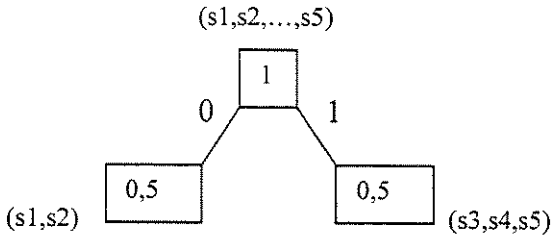
3. As sublistas serão então os filhos do nó anterior;

4. Os passos 2,3 e 4 acima são repetidos até que as sublistas sejam unitárias.

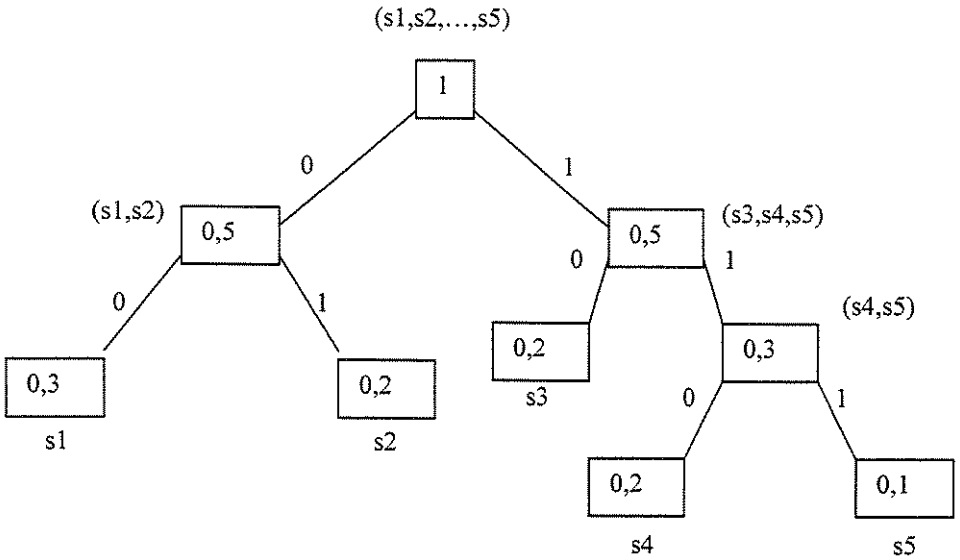
Por exemplo, seja a seguinte lista L = {s1, s2, s3, s4, s5}, com respectivos pesos P = {0,3; 0,2; 0,2; 0,2; 0,1}. A figura 3.1 ilustra o método de Shannon-Fano para este exemplo [16].



a) Árvore Inicial.



b) Primeira divisão da lista



c) Árvore final de Shannon-Fano

Figura 3.1 Árvores de Shannon-Fano para o exemplo dado.

Portanto, teríamos a seguinte codificação para o exemplo acima:

s1 – 00 s2 – 01 s3 – 10 s4 – 110 s5 – 111

Na construção do código de Shannon-Fano, não temos a garantia de construir uma árvore de menor altura. Portanto, não se garante uma codificação ótima para um conjunto de símbolos com pesos definidos, apesar de, na teoria, a codificação alcançar a entropia quando o comprimento do alfabeto tende a infinito [16].

3.4.2 Método de Huffman

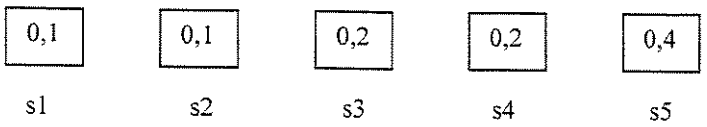
D.A. Huffman [18] propôs, em 1952, um método com o objetivo de obter a redundância mínima desejada do texto comprimido. Isto é possível através de uma árvore de menor altura ponderada. Como no método anterior, aqui também é considerado a existência de um alfabeto fonte, onde cada símbolo tem seu respectivo peso. Como o objetivo é criar um código de prefixo mínimo, associamos códigos menores a símbolos mais prováveis e códigos maiores a símbolos menos prováveis. Para a formação do código de Huffman temos o seguinte algoritmo:

1. Considerar uma floresta em que cada árvore tenha sua raiz associada a um símbolo do alfabeto com seu respectivo peso;

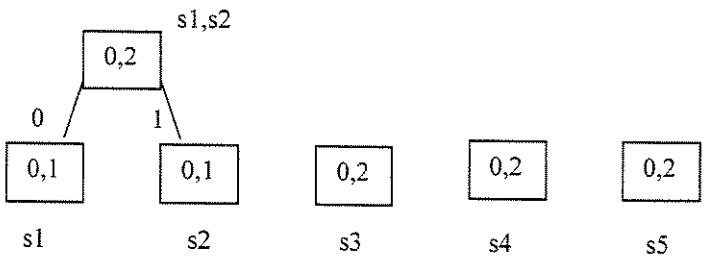
2. Remover quaisquer duas árvores cujas raízes tenham menor peso. Acrescente uma nova árvore que tenha uma raiz cujos filhos sejam árvores anteriores e cujo peso seja a soma dos pesos das raízes dessas árvores;

3. Repetir o passo anterior até que exista somente um árvore na floresta.

Para ilustrar este método, consideremos o exemplo, com a lista de símbolos L = {s1, s2, s3, s4, s5} e os respectivos pesos P = {0,1; 0,1; 0,2; 0,2; 0,4}. As figura 3.2, 3.3 e 3.4 mostram o processo descrito acima.



a) Floresta inicial



b) Floresta após o primeiro passo

Figura 3.2 Construção inicial da árvore de Huffman

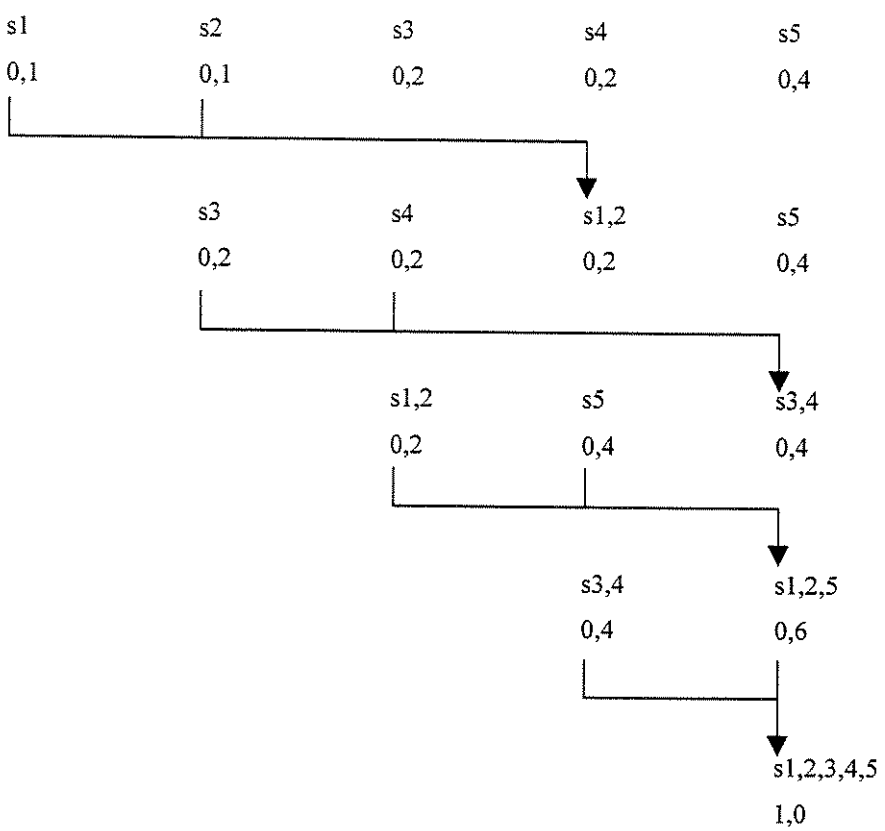


Figura 3.3 Atualização da lista de símbolos

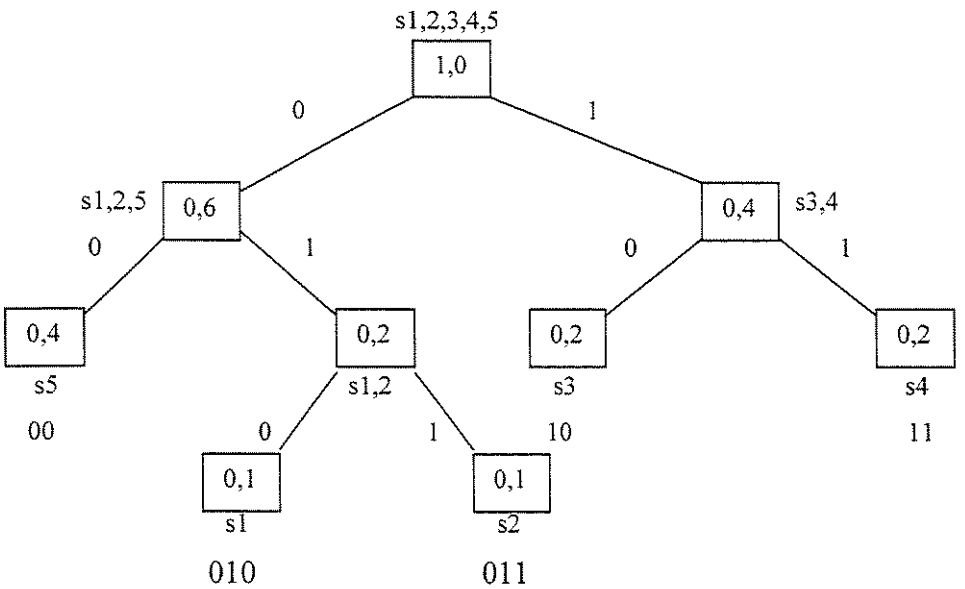


Figura 3.4 Árvore de Huffman.

A árvore de Huffman não é única, mas garante códigos de redundância mínima. Além disto, é o método que mais se aproxima da entropia calculada, isto é, o que mais se aproxima do mínimo teórico [16]. Este método necessita de duas leituras sobre o texto fonte, fato que o torna deficiente em alguns casos, como por exemplo na transmissão de dados. Neste caso, alguns autores procuraram construir os códigos de Huffman dinamicamente, ou seja, a árvore de Huffman é reconstruída conforme as mudanças de pesos apresentadas pelos símbolos fonte [16].

3.4.3 Método Aritmético

O princípio do método aritmético foi proposto por C. E. Shannon [51] e aperfeiçoado por Abramson [53]. A idéia deste método é aplicar um processo recursivo ao código de Shannon, para encontrar o peso acumulativo para um texto fonte de *N* símbolos. Para isto, o processo parte dos pesos individuais dos símbolos e do peso do texto fonte de *N-1* símbolos. O resultado será um número real no intervalo [0,1) que representará a

compressão do texto. O problema aqui é a precisão necessária do número que é diretamente proporcional ao tamanho do texto. Portanto, este método tem por objetivo encontrar um número no intervalo [0,1) com a menor quantidade de dígitos possíveis que represente um texto fonte. Para isto, supomos uma fonte de dados onde os símbolos tenham um determinado peso. Na codificação, inicialmente consideramos o intervalo [0,1) para a representação do texto. À medida que codificamos o texto, este intervalo vai diminuindo e a quantidade de dígitos para representá-lo vai aumentando. A leitura de um novo símbolo reduz o intervalo a um valor que depende de seu peso. Quanto maior o peso, maior será o intervalo obtido e menor será a quantidade de dígitos para representar o texto. Se o símbolo tem peso pequeno, o intervalo obtido será pequeno e a quantidade de dígitos para representar o texto será maior.

Para ilustrar este método, consideremos o seguinte texto fonte.

Texto fonte : *bacb#*.

A cada símbolo do alfabeto fonte, aqui o próprio alfabeto romano, é atribuída uma faixa do intervalo inicial [0,1) estabelecida com base nos pesos acumulativos dos símbolos, como mostra a figura 3.5. A ordem dos símbolos não é relevante.

SÍMBOLO	PESO	PESO ACUMULATIVO	FAIXA
a	0,2	0,0	[0,0;0,2)
b	0,4	0,2	[0,2;0,6)
c	0,2	0,6	[0,6;0,8)
#	0,2	0,8	[0,8;1,0)

Figura 3.5 Os símbolos e seus respectivos pesos e faixas.

A figura 3.5 mostra os símbolos, seus respectivos pesos e faixas. As colunas 3 e 4 são calculadas levando em consideração os pesos dos símbolos (coluna 2). A coluna 4 indica as faixas atribuídas aos símbolos. Por exemplo, o símbolo *a* é associado aos primeiros 20%, o *b*, à faixa de 20% a 60%, e assim sucessivamente. A figura 3.6 ilustra a divisão em faixas.

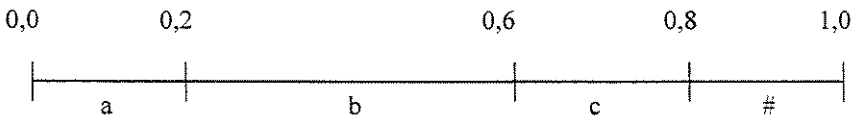


Figura 3.6 Divisão do intervalo [0,1) em faixas de acordo com os pesos dos símbolos

O início da codificação é representado pelo intervalo [0,1). Após a leitura do primeiro símbolo *b*, de seu respectivo peso *(0,4)*, recalculamos o intervalo que tem sua amplitude reduzida de 40%. De acordo com a respectiva faixa de *b*, os novos limites do intervalo deverão estar entre 20% e 60% do intervalo anterior, portanto teremos o intervalo [0,2;0,6), como mostra a figura 3.7.

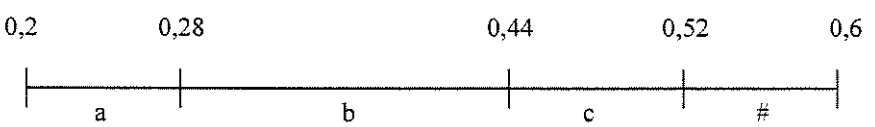


Figura 3.7 Novo intervalo após a leitura do símbolo *b*

Calculamos os novos limites dos símbolos da seguinte forma:

- símbolo *a*: intervalo antigo:[0,0;0,2)
  intervalo novo: extremidade esquerda = 0,2+0,0*(0,6-0,2) = 0,2
  extremidade direita = 0,2+0,2*(0,6-0,2) = 0,28
- símbolo *b*: intervalo antigo:[0,2;0,6)
  intervalo novo: extremidade esquerda = 0,2+0,2*(0,6-0,2) = 0,28
  extremidade direita = 0,2+0,6*(0,6-0,2) = 0,44
- símbolo *c*: intervalo antigo:[0,6;0,8)
  intervalo novo: extremidade esquerda = 0,2+0,6*(0,6-0,2) = 0,44
  extremidade direita = 0,2+0,8*(0,6-0,2) = 0,52
- símbolo #: intervalo antigo:[0,8;1,0)
  intervalo novo: extremidade esquerda = 0,2+0,8*(0,6-0,2) = 0,52
  extremidade direita = 0,2+1,0*(0,6-0,2) = 0,6

Com o próximo símbolo *a*, reduzimos o intervalo em 20% pois, de acordo com a coluna 4 da figura 3.5, o intervalo que corresponde a este símbolo é de 0% a 20%. A faixa resultante ficou igual a [0,2;0,28). A figura 3.8 mostra esta nova divisão.

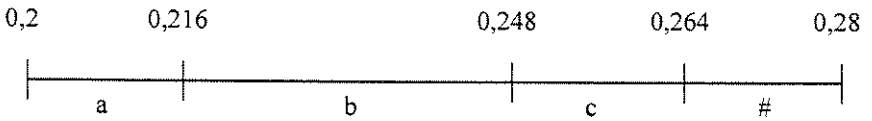


Figura 3.8 Novo intervalo após a leitura do símbolo *a*

Calculamos os novos limites dos símbolos da seguinte forma:

- símbolo *a*: intervalo antigo:[0,2;0,28)
  intervalo novo: extremidade esquerda = 0,2+0,0*(0,28-0,2) = 0,2
  extremidade direita = 0,2+0,2*(0,28-0,2) = 0,216
- símbolo *b*: intervalo antigo:[0,28;0,44)
  intervalo novo: extremidade esquerda = 0,2+0,2*(0,28-0,2) = 0,216
  extremidade direita = 0,2+0,6*(0,28-0,2) = 0,248
- símbolo *c*: intervalo antigo:[0,44;0,52)
  intervalo novo: extremidade esquerda = 0,2+0,6*(0,28-0,2) = 0,248
  extremidade direita = 0,2+0,8*(0,28-0,2) = 0,264
- símbolo #: intervalo antigo:[0,52;0,6)
  intervalo novo: extremidade esquerda = 0,2+0,8*(0,28-0,2) = 0,264
  extremidade direita = 0,2+1,0*(0,28-0,2) = 0,28

Com a letra *c*, reduzimos o intervalo anterior para 20%, nos limites de 60% e 80% (coluna 4 da figura 3.5), ficando [0,248;0,264). A figura 3.9 mostra esta divisão.

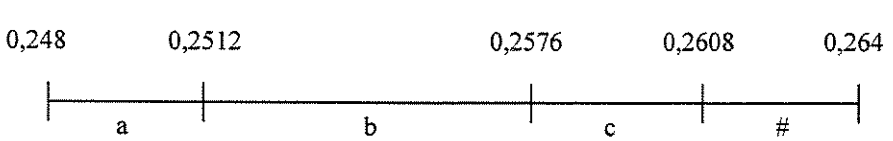


Figura 3.9 Novo intervalo após a leitura do símbolo *c*

Calculamos os novos limites dos símbolos da seguinte forma:

- símbolo *a*: intervalo antigo:[0,2;0,216)

  intervalo novo: extremidade esquerda = 0,248+0,0*(0,264-0,248) = 0,248

  extremidade direita = 0,248+0,2*(0,264-0,248) = 0,2512

- símbolo *b*: intervalo antigo:[0,216;0,248)

  intervalo novo: extremidade esquerda = 0,248+0,2*(0,264-0,248) = 0,2512

  extremidade direita = 0,248+0,6*(0,264-0,248) = 0,2576

- símbolo *c*: intervalo antigo:[0,248;0,264)

  intervalo novo: extremidade esquerda = 0,248+0,6*(0,264-0,248) = 0,2576

  extremidade direita = 0,248+0,8*(0,264-0,248) = 0,2608

- símbolo #: intervalo antigo:[0,264;0,28)

  intervalo novo: extremidade esquerda = 0,2+0,8*(0,264-0,248) = 0,2608

  extremidade direita = 0,248+1,0*(0,264-0,248) = 0,264

Com a nova ocorrência do símbolo *b*, reduzimos o intervalo anterior para [0,2512;0,2576). A figura 3.10 mostra o novo intervalo.

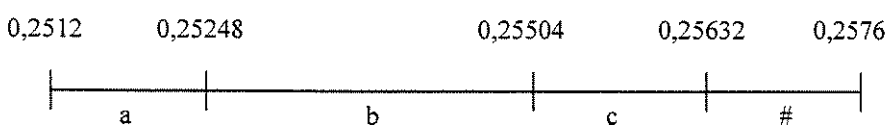


Figura 3.10 Novo intervalo após a leitura da nova ocorrência do símbolo *b*

Calculamos os novos limites dos símbolos da seguinte forma:

- símbolo *a*: intervalo antigo:[0,248;0,2512)

  intervalo novo: extremidade esquerda = 0,2512+0,0*(0,2576-0,2512) = 0,2512

  extremidade direita = 0,2512+0,2*(0,2576-0,2512) = 0,25248

- símbolo *b*: intervalo antigo:[0,2512;0,25248)

  intervalo novo: extremidade esquerda = 0,2512+0,2*(0,2576-0,2512) = 0,25248

  extremidade direita = 0,248+0,6*(0,2576-0,2512) = 0,255504

- símbolo *c*: intervalo antigo:[0,2576;0,2608)

  intervalo novo: extremidade esquerda = 0,2512+0,6*(0,2576-0,2512) = 0,25504

  extremidade direita = 0,2512+0,8*(0,2576-0,2512) = 0,25632

- símbolo #: intervalo antigo:[0,2608;0,264)

  intervalo novo: extremidade esquerda = 0,2512+0,8*(0,2576-0,2512) = 0,25632

  extremidade direita = 0,2512+1,0*(0,2576-0,2512) = 0,2576

Finalmente com símbolo #, reduzimos o intervalo para os últimos 20%, resultando os valores [0,25632;0,2576). A figura 3.11 mostra esta última divisão.

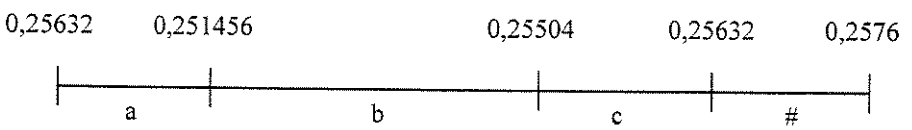


Figura 3.10 Novo intervalo após a leitura da nova ocorrência do símbolo *b*

Calculamos os novos limites dos símbolos da seguinte forma:

- símbolo *a*: intervalo antigo:[0,2512;0,25248)

  intervalo novo: extremidade esquerda = 0,25632+0,0*(0,2576-0,25632) = 0,25632

  extremidade direita = 0,25632+0,2*(0,2576-0,25632) = 0,256576

- símbolo *b*: intervalo antigo:[0,25248;0,25504)

  intervalo novo: extremidade esquerda = 0,25632+0,2*(0,2576-0,25632) = 0,256576

  extremidade direita = 0,25632+0,6*(0,2576-0,25632) = 0,257088

- símbolo *c*: intervalo antigo:[0,25504;0,25632)

  intervalo novo: extremidade esquerda = 0,25632+0,6*(0,2576-0,25632) = 0,257088

  extremidade direita = 0,25632+0,8*(0,2576-0,25632) = 0,257344

- símbolo #: intervalo antigo:[0,25632;0,2576)

  intervalo novo: extremidade esquerda = 0,25632+0,8*(0,2576-0,25632) = 0,257344

  extremidade direita = 0,25632+1,0*(0,2576-0,25632) = 0,2576

Portanto, número que pode representar o texto fonte é um número qualquer dentro do intervalo [0,25632;0,2576). Para a decodificação temos que conhecer os valores vistos na figura 3.5 e o valor numérico resultante da codificação.

3.5 Métodos de Codificação por Fatores

O método de codificação por fatores tem um processo conjunto de modelagem e codificação de dados com o objetivo de tentar identificar e remover redundâncias. Eles tentam substituir cadeias de símbolos consecutivos que ocorrem mais de uma vez no texto, chamados de *fatores*, por índices que apontam para uma única cópia dos fatores. Os fatores fazem parte do dicionário de dados. Para que a compressão tenha sucesso, é importante que os índices tenham seus tamanhos menores que o comprimento dos fatores que eles apontam. A seguir serão apresentados alguns exemplos destes métodos.

3.5.1 Método de Elias-Bentley

Elias [54] e Bentley [55] propuseram em 1986, de forma independente, um método que tem como objetivo a eliminação de redundâncias devido à localidade de referência. Ele é um método adaptativo e se baseia no reconhecimento de palavras no texto. As palavras são mantidas em uma lista, onde as mais freqüentes estão sempre no início da lista. O código é resultado da posição e, conseqüentemente, do índice da palavra na lista. Por exemplo, seja o seguinte texto:

texto fonte : *viva maria iá iá*
viva a bahia iá iá iá iá

A codificação para o texto pode ser vista na figura 3.11

PASSO	LISTA DE PALAVRAS	PALAVRA LIDA	CÓDIGO GERADO
1	*vazio*	viva	1 viva
2	viva	maria	2 maria
3	maria viva	iá	3 iá
4	iá maria viva	iá	1
5	iá maria viva	viva	3
6	viva iá maria	a	4 a
7	a viva iá maria	bahia	5 bahia
8	bahia a viva iá maria	iá	4
9	iá bahia a viva maria	iá	1
10	iá bahia a viva maria	iá	1
11	iá bahia a viva maria	iá	1

Figura 3.11 – Compressão pelo Método Elias-Bentley.

A lista de palavras deverá sempre estar limitada no seu tamanho. Isto significa que a lista terá, em um determinado momento, somente as palavras que mais ocorrem no texto. Como a inserção é feita no final da lista, se a lista estiver cheia, a palavra a ser removida será uma das menos freqüentes.

3.5.2 Método de Lempel-Ziv

J. Ziv e A. Lempel [19] propuseram em 1977 um método, também adaptativo, baseado em uma *janela deslizante* sobre um texto fonte contendo os últimos símbolos reconhecidos do texto. O objetivo é identificar cadeias consecutivas de fatores cada vez maiores. A compressão está na substituição das ocorrências destes fatores por um conjunto de índices que mostram sua posição e tamanho na janela deslizante. Esta janela é uma porção contínua e, à medida que os símbolos vão sendo reconhecidos, eles vão sendo acrescentados no seu final, ao mesmo tempo que os do início são excluídos. A janela e o fator a ser reconhecido têm seus tamanhos limitados. Seja N o tamanho máximo da janela, e F o comprimento máximo do fator. A janela será formada por duas partes: uma chamada de *janela reconhecida*, com comprimento N-F que dá os últimos símbolos do texto fonte que já foram reconhecidos e a outra de *janela não-codificada*, que tem outros F símbolos que

ainda não foram comprimidos. O código resultante será composto por triplas *(d, l ,s)*, onde *d* é o deslocamento do fator para a borda da janela reconhecida, *l* é o comprimento do fator, e *s* é o próximo símbolo do texto original que não pôde ser reconhecido. A figura 3.12 mostra a divisão da janela.

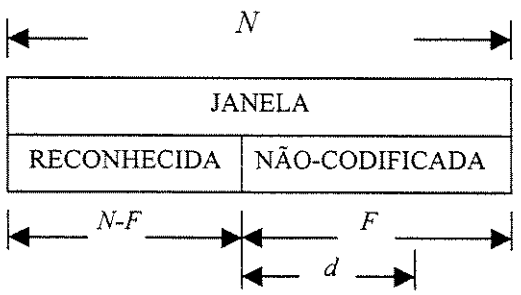


Figura 3.12 Janela deslizante

Usaremos o seguinte exemplo para ilustrar este método:

Texto Fonte: *aababbaababbaaa...*

Consideremos o maior comprimento do fator sendo igual a 8 e a janela com tamanho suficientemente grande. As etapas para a compressão podem ser vistas na figura 3.13. A figura 3.14 ilustra a descompressão.

PASSO	JANELA		TEXTO FONTE	MAIOR FATOR	CÓDIGO GERADO
	RECONHECIDA	NÃO CODIFICADA			
1		aababbaa	babbaaa	*vazio*	(0,0,a)
2	a	ababbaab	abbaaa	a	(1,1,b)
3	aab	abbaabab	baaa	ab	(2,2,b)
4	aababb	aababbaa	a	aababbaa	(6,8,a)

Figura 3.13 Compressão Lempel-Ziv

PASSO	JANELA	TEXTO COMPRIMIDO	FATOR
1		(0,0,a)(1,1,b)(2,2,b)(6,8,a)	a
2	a	(1,1,b)(2,2,b)(6,8,a)	ab
3	aab	(2,2,b)(6,8,a)	abb
4	aababb	)(6,8,a)	aababbaaa

Figura 3.14 Descompressão segundo Lempel-Ziv

O resultado da codificação será:

Texto original:	a	ab	abb	aababbaaa
Texto comprimido:	(0,0,a)	(1,1,b)	(2,2,b)	(6,8,a)

A eficiência deste método está ligado ao tamanho da janela. Se a janela for muito grande, seu desempenho está próximo do método visto na seção 3.5.1 .

3.5.3 Método de Lempel-Ziv (1978)

O método Lempel-Ziv (1978) [16] difere do anterior porque mantém um dicionário de fatores fixos em vez da janela deslizante. A conseqüência disto é que, enquanto o método Lempel-Ziv pode referenciar qualquer fator do texto já lido, este pode referenciar apenas os fatores constantes no dicionário. Este método funciona de maneira adaptativa, mantendo um dicionário de fatores. O objetivo é reconhecer fatores longos no texto fonte que se encontram no dicionário e substituí-los pelos índices referentes às respectivas entradas do dicionário. Cada entrada do dicionário será formada pelo fator reconhecido no texto juntamente com o próximo símbolo que não pôde ser reconhecido. O texto codificado será composto de duplas na forma *(i, s)*, onde *i* é o índice do fator no dicionário e *s* é o próximo símbolo que não foi reconhecido. O tamanho de *i* é variável e cresce com o tamanho do dicionário. Esta nova abordagem permite o uso ilimitado da memória e simplifica o processo de codificação, facilitando a busca dos fatores no dicionário.

Para ilustrar este método, consideremos o seguinte texto:

Texto Fonte: *aababbaababbaaa...*

A figura 3.15 mostra os passos necessários para a codificação

PASSO	TEXTO FONTE	MAIOR FATOR	DICIONÁRIO			CÓDIGO GERADO
			I	FATOR	PREFIXO+S	
1	aababababaaaab	*vazio*	1	a	0a	(0,a)
2	abababaaaab	a	2	ab	1b	(1,b)
3	ababaaaab	ab	3	aba	2a	(2,a)
4	baaaab	*vazio*	4	b	0b	(0,b)
5	aaaab	a	5	aa	1a	(1,a)
6	aab	aa	6	aab	5b	(5,b)

Figura 3.15 Compressão pelo método Lempel-Ziv (1978)

Com esta codificação geramos o seguinte texto comprimido:

Texto original: a ab aba b aa aab

Texto comprimido: (0,a) (1,b) (2,a) (0,b) (1,a) (5,b)

Para efetuar a descompressão, o descompressor segue o mesmo caminho do compressor para reconstruir o dicionário. Quando o descompressor lê o índice, ele percorre o dicionário de acordo com o prefixo das entradas, gerando o fator em ordem reversa. A figura 3.16 ilustra este processo.

PASSO	TEXTO COMPRIMIDO	FATOR GERADO	DICIONÁRIO		
			I	FATOR	PREFIXO+S
1	(0,a)(1,b)(2,a)(0,b)(1,a)(5,b)	a	1	a	0a
2	(1,b)(2,a)(0,b)(1,a)(5,b)	ab	2	ab	1b
3	(2,a)(0,b)(1,a)(5,b)	aba	3	aba	2a
4	(0,b)(1,a)(5,b)	b	4	b	0b
5	(1,a)(5,b)	aa	5	aa	1a
6	(5,b)	aab	6	aab	5b

Figura 3.16 Descompressão segundo Lempel-Ziv (1978)

No método Lempel-Ziv (1978), quanto maior o tamanho do texto mais próximo este estará da entropia, isto é, para textos que tendem ao infinito, a compressão será dita ótima. Isto faz com que este método seja um pouco restrito pois, geralmente, os textos são curtos e, mesmo que os textos sejam grandes, não se dispõe de memória física infinita, de tal maneira a ser suficiente para a realização da compressão.

3.6 Compressores de Textos

A maioria dos compressores existentes se baseiam, principalmente no método Lempel-Ziv [19]. Alguns compressores, aplicam uma compressão estátistica, após o texto comprimido, utilizando principalmente o método de Huffman [18] ou o de Shannon-Fano [51,52]. Segue algusn compressores mais conhecidos e os seus respecyivos métodos de compressão. O compressor *compress (* da *Free Softwre Foundation)*, utiliza o método de Lempel-Ziv [19]; o *compact* e o *pack* utilizam o de Huffman [18]; o *PKZIP* utiliza o método de Lempel-Ziv [19] e o de Shannon-Fano [51,52] e o *ARJ* utiliza o método Lempel-Ziv [19] e o de Huffman [16].

CAPÍTULO 4

Métodos de Compressão de Programas

Os algoritmos vistos no Capítulo 3 não são adequados à compressão de código de programas principalmente por dois motivos. O primeiro é que não permite a descompressão aleatória de palavras codificadas. Isto é necessário pois a descompressão nem sempre pode ser feita de maneira seqüencial devido às instruções de desvios existentes no programa. Por exemplo, seja o trecho de programa da figura 4.1.

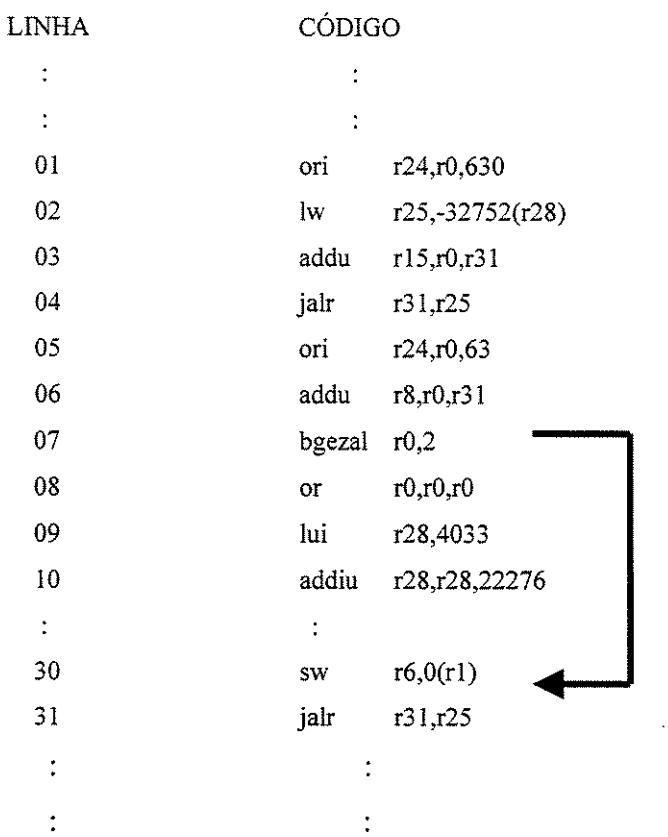


Figura 4.1 Trecho de um programa em linguagem de montagem *MIPS R2000*

Supondo que a instrução de desvio da linha 07 tenha que ser executada e que a instrução alvo deste desvio seja a da linha 30, na descompressão, para que a execução do programa seja feita corretamente, será necessário enviar ao processador as instruções a partir da linha 30. Se o trecho de programa da figura 4.1 fosse comprimido segundo quaisquer dos métodos vistos no Capítulo 3, a próxima linha a ser descomprimida seria a linha subseqüente à linha 07. Portanto as instruções que estariam sendo passadas ao processador não seriam as informações corretas. O segundo motivo está relacionado com a quantidade de recursos necessários à sua descompressão. Por exemplo, mesmo para um programa sem desvios, as baixas razões de compressão encontradas por estes algoritmos dependem da utilização de janelas muito grandes, implicando em manter uma memória ou *buffer* de descompressão grande, ocupando uma grande área de silício. Isto implica em uma redução considerável da razão de descompressão final. Por estas razões, compressão de código de programa tem que ser tratada de maneira especial.

No restante deste capítulo, procuraremos apresentar alguns trabalhos importantes referentes à compressão de código objeto. A seguir discutiremos várias idéias apresentadas para este tipo de compressão.

4.1 *Compress Code RISC Processor (CCRP)*

Andrew Wolfe e Alex Chanin [20], propõem um sistema de descompressão chamado de *Compressed Code RISC Processor – CCRP* [21,22], que permite que os processadores RISC executem programas comprimidos. O objetivo é a utilização de processadores RISC em sistemas embutidos. Para isto, o tamanho do código gerado para este tipo de processador tem que ser diminuído através de técnicas de compressão. A descompressão é feita em tempo real e deve ser transparente ao processador. O CCRP é um sistema composto de um processador RISC, uma memória cache de instruções, uma memória de instruções, uma tabela de mapeamento de endereços e um *buffer* auxiliar. Este sistema pode ser visto na figura 4.2

O código comprimido é armazenado na memória de instruções onde uma máquina de descompressão, chamada *máquina de preenchimento da cache* (*Cache Refill Engine*), lê

este código da memória, descomprime-o e armazena-o na memória *cache* de instrução. A descompressão é dita transparente ao processador pois o código buscado por ele na cache já está descomprimido. A compressão tem que ser feita levando-se em conta o tamanho da linha da memória *cache* para que na descompressão, uma linha da *cache* seja totalmente preenchida.

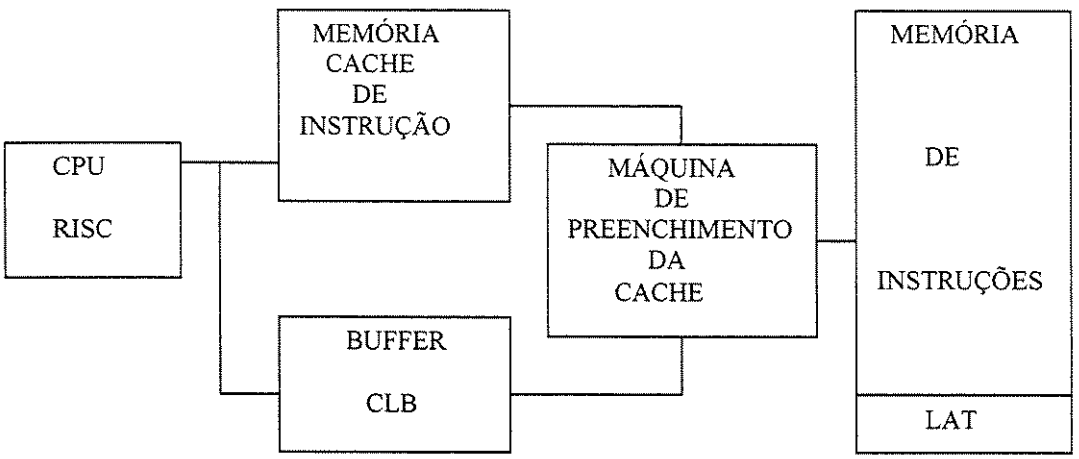


Figura 4.2 – Organização do Sistema CCRP

Para mapear os endereços dos blocos da memória de instruções (endereços de códigos comprimidos) em endereços de blocos da memória cache (endereços de códigos não comprimidos), o sistema usa uma tabela chamada *Tabela de Endereços de Linha* (*Line Address Table – LAT*). Esta tabela pode ser implementada em *hardware* ou é armazenada na memória de instrução. Esta é a opção utilizada por Wolfe e Chanin [20].

O *buffer* auxiliar *CLB* (*Cache Line Address Lookaside Buffer*) é utilizado para armazenar as referências à *cache* mais recentemente utilizadas. A função deste *buffer* é diminuir o número de acessos à *LAT* quando uma linha desejada não estiver na cache. Quando isto ocorrer, se o endereço desta linha estiver armazenado no *buffer CLB*, então o mapeamento de endereços comprimidos para não comprimidos é feito de imediato no *CLB*,

sem que a *LAT* precise ser acessada. O *CLB* tem a mesma função que o *TLB (Translator Lookaside Buffer)* em sistemas de memória paginada. O aumento na razão de compressão, devido à tabela *LAT*, é da ordem de 3% do tamanho do programa original.

4.1.1 Técnicas de Compressão Utilizadas

As técnicas de compressão utilizadas no *CCRP* são baseadas no programa *compress* do sistema operacional *Unix* e em três técnicas de compressão baseadas no método de codificação de *Huffman*. O *compress*, apesar de ser o mais eficiente pois resultou em razão de compressão média de 40%, só funciona bem para blocos de dados maiores que uma linha de cache; portanto não é adequado para ser usado pelo *CCRP*. A razão disto é que o *CCRP* busca na memória de instrução uma linha de memória que deve ter pelo menos um bloco de dados codificado. A segunda alternativa adotada no *CCRP* é o método de codificação de *Huffman* no qual são comprimidos blocos de 32 *bits*. A desvantagem deste método está na descompressão pois, geralmente, a codificação *Huffman* gera códigos com palavras grandes e com tamanho variável, fazendo com que o circuito responsável pela decodificação se torne caro e complexo. As palavras de código grandes fazem com que, em alguns casos, não possam ser armazenadas em uma única palavra de memória. Isto acontecendo, a máquina de descompressão teria que ir à memória mais de uma vez para poder descomprimir uma instrução. O tamanho variável da palavra de código aumenta a complexidade do decodificador, pois ele tem que detectar o tamanho da palavra de código antes de decodificá-la. A razão de compressão conseguida neste caso foi em média 73%. Como os códigos com tamanhos grandes são pouco freqüentes, o *CCRP* implementa um esquema onde o tamanho da palavra de código não pode ultrapassar os 16 bits, com o objetivo de diminuir o aspecto da complexidade da máquina de descompressão. Este método foi chamado de *Bounded Huffman*, que é menos eficiente na compressão, mas torna a descompressão menos complexa. Apesar disto, este método resultou uma razão de compressão média próxima da encontrada na abordagem anterior. O inconveniente do *Bounded Huffman* é que, como no método *Huffman* tradicional, as palavras de código que formam o dicionário são armazenadas junto com o programa comprimido para possibilitar a

descompressão. Para a solução deste problema, Wolfe e Chanin [20], baseados em dados estatísticos levantados com os programas testes, utilizaram um mesmo conjunto de palavras de código *Bounded Huffman* para comprimir todos os programas. Com isto, as informações da associação do código com as palavras do programa foram implementadas na máquina de descompressão. Este método é chamado de *Preselect Bounded Huffman* e sua razão de compressão varia muito pouco quando comparada com as duas abordagens anteriores.

4.1.2 Aspectos da Implementação

A implementação do *CCRP* é baseada em um processador *RISC* de 32 *bits* que executa o conjunto de instruções do *MIPS R2000*. O *CCRP*, através da máquina de preenchimento da *cache*, é responsável pelo controle do preenchimento da memória cache de instruções. Se existir uma memória cache de dados, ela deve ser separada da cache de instruções.

As linhas da tabela *LAT* tem como conteúdo uma palavra de 8 *bytes* e são organizadas como mostra a figura 4.3.

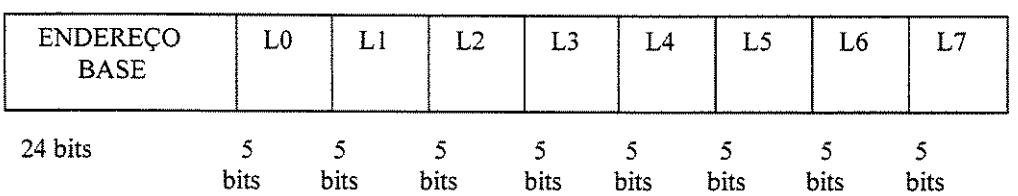


Figura 4.3 Conteúdo das entradas da *Line Address Table – LAT*

Os primeiros 3 *bytes* desta palavra contêm um ponteiro para o primeiro bloco comprimido de um conjunto de 8 blocos. Estes 3 *bytes* são seguidos de oito conjuntos de 5 *bits*, um para cada bloco, que representam o endereço de cada bloco comprimido. Com isto, temos a informação de relocação sobre oito blocos contíguos.

O *CLB* armazena o endereço físico para o cálculo do endereço do bloco comprimido armazenado na memória de instruções quando o bloco referenciado não está na cache. A

organização do CLB pode ser vista na figura 4.4. O decodificador implementado na figura 4.4 usa o método *Preselected Bounded Huffman* e é baseado no circuito apresentado por Benès, Wolfe e Nowick [23,24].

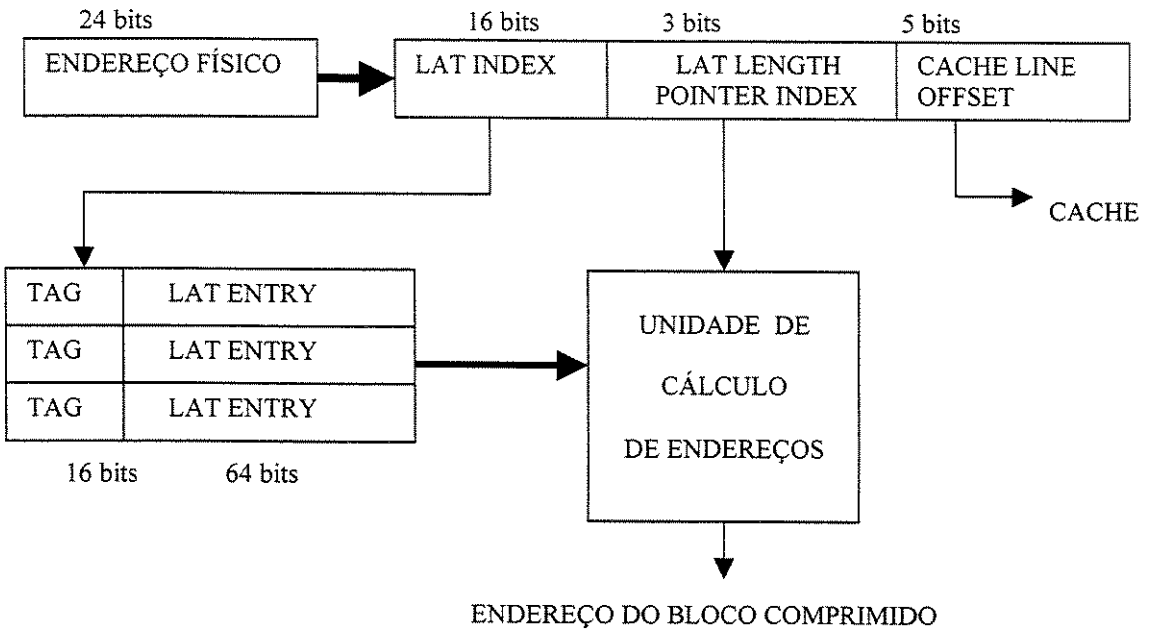


Figura 4.4 – Organização do *Cache Line Address Lookaside Buffer – CLB*

Neste circuito, podemos ver que o primeiro campo do endereço físico (*LAT Index*), de 16 *bits*, é comparado com os *tags* mantidos pelo *CLB*. Se ele for igual a um *tag* qualquer e existir um *cache miss*, a entrada correspondente da *LAT* é usada como uma das entradas da *Unidade de Cálculo de Endereços*. Se a entrada da *LAT* não está no *CLB*, então ela é lida da memória usando o campo *LAT Index* e o registrador base da *LAT*. A *Unidade de Cálculo de Endereços* utiliza, além da entrada da *LAT*, um campo de 3 bits do endereço físico (*LAT Length Pointer Index*) para calcular o endereço do bloco comprimido.

4.1.3 Comentários

O *CCRP* foi o primeiro sistema de compressão/descompressão proposto para um processador *RISC*. Os resultados apresentados na compressão não são muitos expressivos, pois o problema é tratado como um simples problema de compressão de dados, isto é, o programa é visto como uma seqüência de símbolos sem nenhuma relação entre eles. Além disso, a área de silício relativa ao sistema de descompressão não está computada na razão de compressão, o que nos dá uma visão distorcida sobre o resultado final da compressão. Outro ponto desfavorável no CCRP é a complexidade das alterações de *hardware* no processador requeridas para implementar a máquina de descompressão, tais como a *Line Address Table*, *Cache Line Address Lookaside Buffer*, e a máquina de preenchimento da *cache*.

4.2 Método de Lefurgy et al

Lefurgy et al [25] propõem uma técnica de compressão de código baseado na codificação do programa usando um dicionário de códigos. A figura 4.5 mostra o esquema geral desta abordagem.

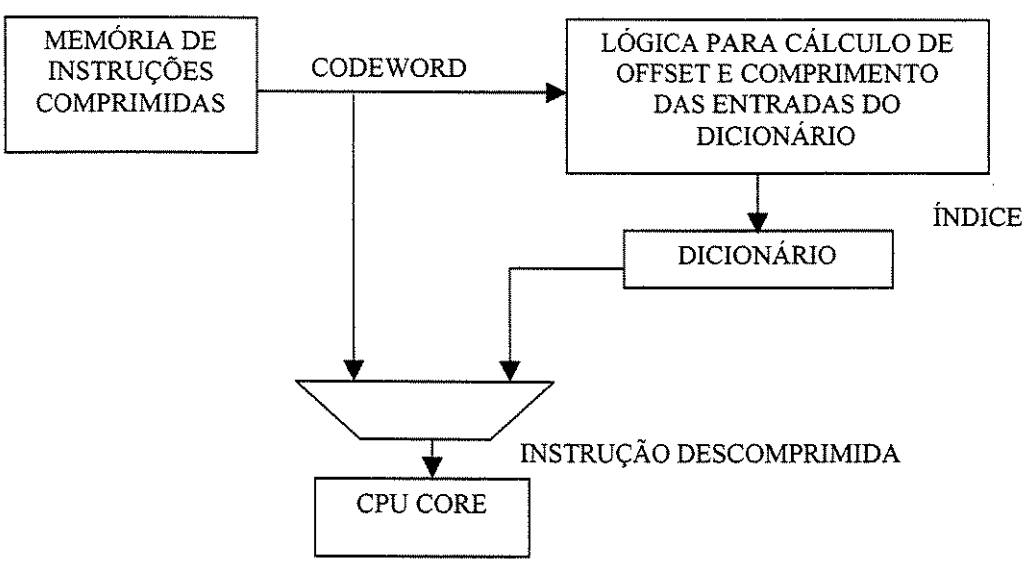


Figura 4.5 Esquema de compressão e descompressão de Lefurgy et al

Nesta técnica, o código objeto é analisado e as seqüências comuns de instruções são substituídas por uma palavra codificada, como na compressão de texto. Apenas seqüências freqüentes são comprimidas. Um *bit* (*escape bits*) é utilizado para distinguir uma palavra comprimida (codificada) de uma instrução não comprimida. As instruções correspondentes às palavras codificadas são armazenadas em um dicionário na máquina de descompressão. As palavras codificadas são usadas para indexar as entradas do dicionário. A figura 4.6 mostra o mecanismo de codificação e indexação no dicionário

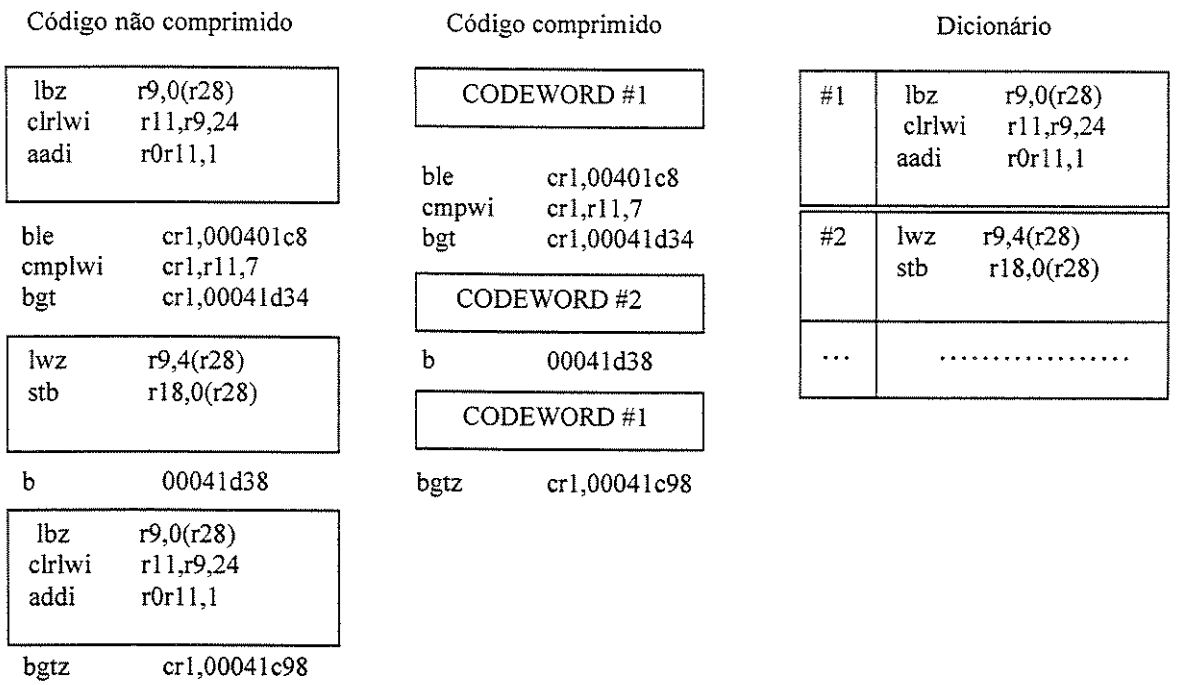


Figura 4.6 Exemplo do esquema de compressão

Esta abordagem é sustentada por um levantamento estatístico que revelou que apenas um pequeno número de instruções tem padrões de bits que não se repetem em um programa. Por exemplo, a compilação do programa *go* (*SPEC CINT95* [26]), para o *PowerPC* [27] utilizando o *gcc 2.7.2* [28] com a opção de otimização *O2*, mostrou que 1% das seqüências de instruções mais freqüentes são responsáveis por 30% do tamanho do programa e 10% das seqüências mais freqüentes é responsável por 66%. Em média, o levantamento mostrou que mais de 80% das instruções aparecem várias vezes no programa.

O algoritmo de Lefurgy et al é dividido em 3 partes: construção do dicionário, substituição das seqüências de instruções por *palavras de código* (*codewords*) e a codificação das palavras de código. Na construção do dicionário (um problema NP-completo), é usado um algoritmo de busca rápida para determinar as seqüências mais freqüentes. O critério de parada para busca das seqüências de instruções é o tamanho da palavra de código que, por sua vez, é especificado pelo método de codificação. Outra limitação deste método é que cada entrada do dicionário também está limitada pela seqüência de instruções de um bloco básico.

A substituição das seqüências é feita pelo algoritmo de busca junto com a construção do dicionário. Todas as instâncias de uma entrada do dicionário são substituídas por uma marca. Estas marcas são depois codificadas segundo o algoritmo de compressão utilizado, gerando as palavras de códigos. A codificação usa códigos de comprimento fixo o que facilita a decodificação. Códigos de comprimento variável (método de *Huffman*) são também estudados. Neste caso, de modo a facilitar o alinhamento, o tamanho do código é limitado a um valor que seja múltiplo de uma unidade básica. Por exemplo, tamanhos de 8, 12 e 16 bits que são múltiplos de 4 bits, que é o valor usado para alinhamento das palavras de códigos.

4.2.1 Os Problemas Decorrentes das Instruções de Desvio

Um dos problemas mais comuns encontrados na compressão de programas se refere à determinação do endereços alvo das instruções de desvio. Normalmente este tipo de instrução não é codificado para evitar a necessidade de reescrever as palavras de códigos que representam estas instruções [25]. Se isto fosse feito, como os endereços alvos seriam alterados, seria necessário reescrever as palavras de códigos que por sua vez, fariam com que os endereços alvos tivessem que ser novamente alterados e, assim, sucessivamente, tornando esta tarefa um problema NP-completo [25]. Os desvios indiretos podem ser codificados normalmente, pois, como seus endereços alvos estão armazenados em registradores, apenas as palavras de códigos necessitam ser rescritas. Neste caso, é

necessário apenas uma tabela para mapear os endereços originais armazenados no registrador para os novos endereços comprimidos.

O problema dos endereços alvo está no alinhamento das instruções. Existem duas soluções para isto. A primeira é alinhar todas as instruções alvo, como definida no conjunto de instruções da arquitetura (*Instruction Set Arqchitecture – ISA*), ou seja, sempre no início da palavra de memória. Apesar de ser uma solução simples, ela diminui a taxa de compressão. A segunda solução é mais complexa, pois requer alterações na unidade de controle do processador para tratar *offsets* com alinhamento segundo o tamanho da palavra de código. Aqui, o endereço não comprimido tem que ser mapaedo em dois endereços comprimidos. O primeiro para indicar em qual palavra de memória está a instrução alvo codificada e o segundo (*offset*) para indicar dentro da palavra, onde está a instrução. Esta solução foi a empregada por Lefurgy et al [25].

4.2.2 Comentários

Com a técnica de códigos com comprimento variável Lefurgy et al [25] encontraram as taxas de compressão 39%, 34% e 26% para os processadores PowerPC [27], ARM [29] e i386 [30], respectivamente. Estes números na verdade não espelham a realidade, pois aqui também não estão computados os aumentos de área devido ao dicionário e às mudanças no *hardware* do processador.

4.3 Método de Liao et al

Liao et al [31,32,33] apresentam dois métodos para minimização do tamanho de código. O primeiro método envolve somente software, ou seja, nenhuma modificação de *hardware* é necessária, enquanto o segundo utiliza alterações no *hardware* do sistema para executar a descompressão. Na abordagem adotada os autores utilizam dois conceitos importantes que são: a compressão de dados utilizando o modelo EPM (*external pointer macro*) e a compressão baseada em *set covering*, discutidos a seguir.

4.3.1 *External Pointer Macro*

No modelo EPM, Storer e Szymanski [34] tratam o dado original como uma cadeia finita de caracteres e o código comprimido como um dicionário e um esqueleto. O dicionário é uma cadeia de caracteres e o esqueleto é uma seqüência de símbolos do alfabeto intercaladas com ponteiros para o dicionário. Cada ponteiro representa uma subcadeia do texto original. Um ponteiro consiste de um par de números inteiros (a,l) onde a indica a posição no dicionário e l o comprimento da subcadeia. A figura 4.7 mostra um exemplo deste modelo. O ponto "." separa o dicionário do esqueleto.

Cadeia x = bbcdabbefabffbef dicionário z = abbef
Comprimindo x, utilizando z temos: y = abbef . (2,2)cd(1,5)(1,2)ff(3,3)

Figura 4.7 – Exemplo de codificação utilizando o modelo EPM

4.3.2 *Set Covering*

O problema de *set covering* [56,57] é fundamental em computação. Soluções exatas para este problema, baseadas em busca *branch-and-bound* [36], foram apresentadas por Grasselli e Luccio [37], Gimpel [38] e Rudell [39]. Coudert [40] propôs alterações que melhoraram a eficiência dos métodos anteriores. O problema de *set covering* é descrito como se segue. Seja X um conjunto de variáveis e Y um subconjunto de 2^X. Um elemento y de Y cobre um elemento x de X se x pertence a Y. A cada elemento y de Y é associado um custo não negativo cost(y). O problema *set covering* (ou *unate covering*) consiste em selecionar um subconjunto Z de Y que tenha a menor somatória dos custos dos elementos de cada subconjunto possível de Y, tal que cada elemento de X seja coberto por pelo menos um elemento de Z.

Por exemplo, seja: $X = \{x_1, x_2, x_3, x_4, x_5\}$ e $Y = \{y_1, y_2, y_3, y_4\}$, onde: $y_1 = \{x_1,x_2,x_3\}$, $y_2=\{x_1,x_3,x_4\}$, $y_3=\{x_2,x_4,x_5\}$ e $y_4=\{x_2,x_3,x_4\}$. Com os custos associados iguais a: $\text{cost}(y_1) = 2$, $\text{cost}(y_2) = 3$, $\text{cost}(y_3) = 1$ e $\text{cost}(y_4) = 3$. As duas soluções para este problema de cobertura são: $\{y_1,y_4\}$ e $\{y_2,y_3\}$, com respectivos custos: $\text{cost}(y_1)+\text{cost}(y_4) = 5$ e $\text{cost}(y_2)+\text{cost}(y_3) = 4$. Portanto, como o custo da segunda solução é menor, ela é a solução escolhida.

Uma outra maneira de tratar o problema de cobertura é utilizar um conjunto de expressões booleanas para indicar como cada x é coberto. Por exemplo, se y_i é uma variável booleana e desejamos uma solução para a cobertura de x_1, y_1 ou y_2 deve ser selecionada, ou seja, a solução pode ser representada pela equação booleana $y_1 + y_2$. Então, para cada elemento de X temos uma equação deste tipo e podemos montar a matriz de cobertura mostrada na figura 4.8.

	y_1	y_2	y_3	y_4	
$y_1 + y_2$	1	1			x_1
$y_1 + y_3 + y_4$	1		1	1	x_2
$y_2 + y_4$		1		1	x_3
$y_2 + y_3 + y_4$		1	1	1	x_4
$y_1 + y_3$	1		1		x_5

Figura 4.8 – Matriz de cobertura dos elementos do conjunto X

O preenchimento desta matriz é feito colocando-se o dígito 1 na posição (i,j) . O objetivo do problema de cobertura é achar um conjunto de colunas de menor custo. Isto é feito selecionando todas as colunas que tenham pelo menos um dígito 1 [38,39]. Uma coluna j é dita *essencial* se apenas ela cobre alguma linha i. Uma linha i é dita *dominada* por outra linha i' se a cobertura de i' necessariamente cobre também a linha i; por exemplo, na figura 4.8, a linha $(y_2 + y_4)$ é dominada pela linha $(y_2 + y_3 + y_4)$. Uma coluna j é dita *dominada* por uma coluna j' se j cobre um subconjunto de linhas cobertas pela coluna j' e o

cost(j') é menor ou igual ao *cost(j)*. Podemos repetir a remoção de colunas essenciais, linhas dominadas e colunas dominadas para chegar à menor matriz chamada de núcleo cíclico da matriz de cobertura [35]. O problema de cobertura pode tanto ser resolvido exatamente usando algoritmos *branch-and-bound* [36] ou aproximadamente utilizando algoritmos heurísticos, como *non-backtracking* [36] ou *limited-backtracking* [36] .

4.3.3 Métodos de Compressão Propostos

Ambos os métodos são baseados no modelo de compressão EPM ("External Pointer Macro") e diferem apenas em como o dicionário e os ponteiros são definidos. O algoritmo de compressão utilizado é baseado na formulação do problema de *set covering*. O algoritmo apresenta duas fases, sendo a primeira a geração das entradas do dicionário e a segunda a substituição das instruções por código comprimido e geração do dicionário. Durante a primeira fase pode ocorrer a não cobertura de subcadeias freqüentes em um conjunto de *bits* nas instruções. Isto então passa a ser um problema de *set covering* no qual as variáveis correspondem às seleções das substituições e às seleções das entradas do dicionário. Na geração das potenciais entradas no dicionário, uma cadeia de instruções é dividida em blocos básicos [41,42,43]. Cada bloco básico é comparado com todos os outros blocos básicos, inclusive com ele mesmo, a fim de detectar subcadeias de caracteres comuns. A substituição e geração do dicionário envolve a decisão de qual ocorrência das potenciais entradas do dicionário encontradas será substituída por ponteiros para o dicionário e também qual das potenciais entradas fará parte deste dicionário. Este problema é tratado como um problema de cobertura (*set-covering*). Na formulação da cobertura, é criada uma matriz de cobertura onde as colunas são variáveis binárias e as linhas são cláusulas distintas de diferentes subconjuntos de variáveis [35]. São encontrados então os valores para as variáveis de tal maneira a satisfazer cada uma das cláusulas e de minimizar o custo da função, segundo a formulação do problema de *set-covering* [35,40]. Um exemplo de compressão usando *set-covering* pode ser visto a seguir. Suponha os seguinte conjunto de instruções mostrado na figura 4.9.

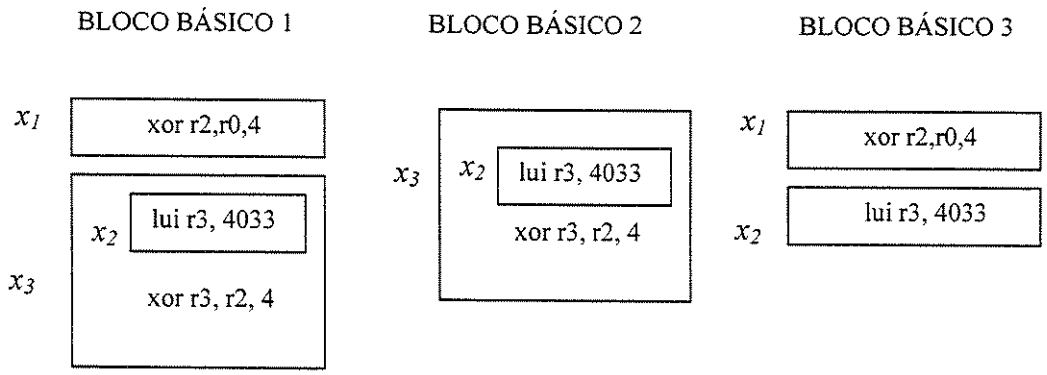


Figura 4.9 Blocos básicos a serem analisados para formação do dicionário utilizando *set-covering*

Na figura 4.9, podemos ver associado a cada conjunto comum de instruções dos blocos básicos da figura 4.9, uma variável x_i, resultando em $X = \{ x_1, x_2, x_3 \}$ onde:

$x_1 = xor\ r2,r4$; $x_2 = lui\ r3,4033$ e $x_3 = lui\ r3,4033$
$xor\ r3,r2,4.$

Para cada possível entrada no dicionário, associamos uma variável y_i. Portanto teremos $Y=\{y_1, y_2, y_3\}$ onde: $y_1 = \{ x_1 \}$, $y_2 = \{ x_3 \}$ e $y_3=\{ x_2, x_3 \}$. Para compressão do código de programa, isto resultaria na seguinte matriz de cobertura:

	y_1	y_2	y_3
y_1	1		
y_2			1
$y_2 + y_3$		1	1

x_1
x_2
x_3

Figura 4.10 – Matriz de cobertura dos elementos do conjunto X

Removendo as colunas essenciais e as colunas e linhas dominadas, temos a seguinte matriz reduzida mostrada na figura 4.11. Esta matriz minimiza o código de programa, mas a solução completa tem que levar em conta a minimização do tamanho do dicionário.

	y_1	y_3
y_1	1	
$y_2 + y_3$		1
y_3		1

x_1
x_2
x_3

Figura 4.11 – Matriz de cobertura reduzida dos elementos do conjunto X

A seguir discutiremos os dois métodos propostos por Liao et al.

a) **Método I**

Neste método são extraídas seqüências comuns para formar as entradas de um dicionário. Estas seqüências são substituídas por chamadas ao dicionário, como se fossem chamadas de sub-rotinas sem a passagem de parâmetros. Mesmo que uma seqüência seja formada só por uma parte de uma entrada ao dicionário, ela pode ser substituída por uma chamada informando o início da respectiva subseqüência. A determinação das seqüências não está limitada somente aos *blocos básicos* [41].

b) Método II

Este método usa o dicionário de modo mais flexível. Ele pode ser visto como uma implementação em *hardware* do modelo EPM. Neste caso, o dicionário é visto como uma grande entrada e não um conjunto de entradas. Com isso, qualquer subcadeia do dicionário pode ser substituída por um ponteiro apropriado. Aqui a seqüência de instruções pertencentes ao dicionário é tratada como uma sub-rotina, em que uma instrução especial de chamada de sub-rotina *CALD (addr,len)* é utilizada. Esta instrução não necessita de uma instrução de retorno, pois seus parâmetros são o endereço da sub-rotina no dicionário (*addr*) e o número de instruções a serem executadas (*len*). Para a determinação das

seqüências, devem ser observados os limites dos blocos básicos, pois, como os caminhos de um desvio condicional nem sempre têm o mesmo tamanho e como o ponto de retorno está implícito no parâmetro que indica o número de instruções (*len*), não se pode generalizar quando o retorno irá acontecer. Para este método é necessário uma lógica adicional, composta por: um *latch* para indicar o modo de funcionamento do processador, modo dicionário ou normal; um contador responsável pelo controle do número de instruções da sub-rotina, que serão executadas e uma pilha de registradores (*link register stack*) que armazena o endereço de retorno da chamada da sub-rotina. A utilização desta lógica permite que a instrução *CALD* inicialize o processador no modo dicionário (carrega 1 no *latch*), carrega o valor *len* no contador; empilha o endereço de retorno na pilha de *link register* e carrega o *program counter* (*PC*) com o valor *addr*. A figura 4.12 mostra esta arquitetura.

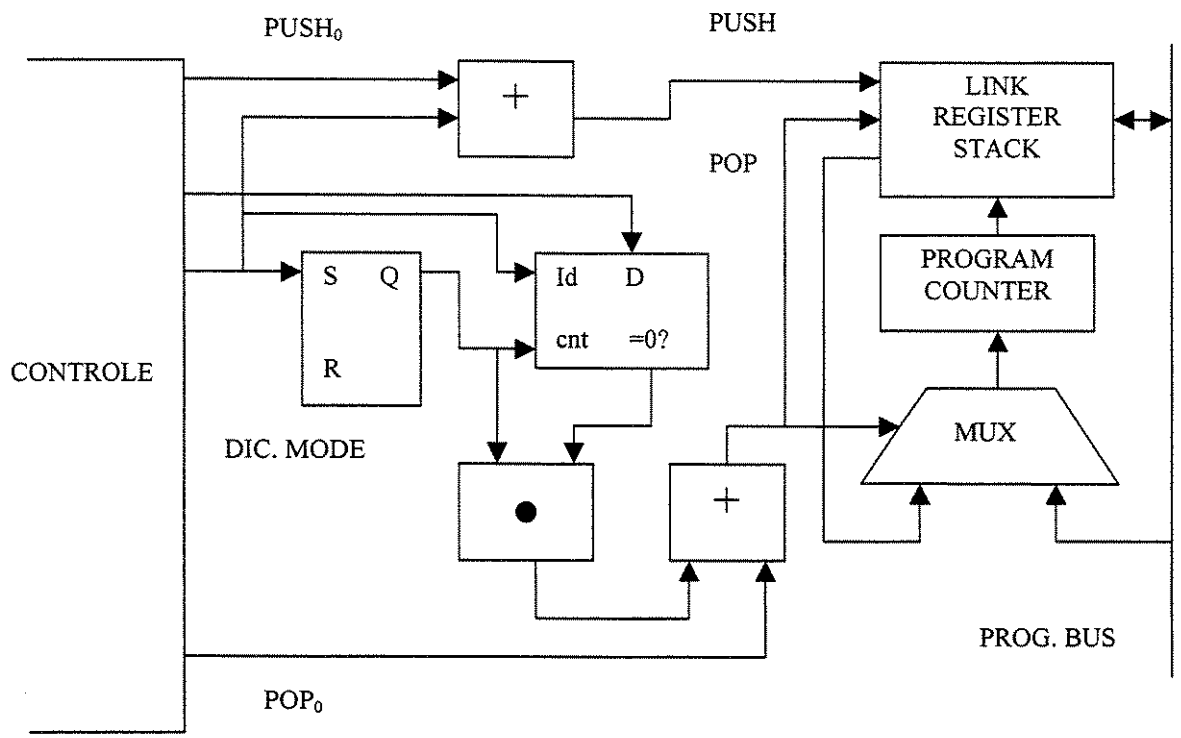


Figura 4.12 Arquitetura do Método II de Liao et al

4.3.4 Resultados Experimentais e Comentários

Para os experimentos, Liao et al [31,32,33] utilizaram o processador TMS320C25. Os resultados apresentados mostraram uma razão de compressão média de 85% no método I e 80% no método II. Estes valores, diferentes dos apresentados nos itens 4.1 e 4.2, estão próximos de uma implementação real, pois levam em conta o tamanho do dicionário.

4.4 Método de Lekatsas et al

Lekatsas et al [44] propuseram dois algoritmos de compressão, um independente e outro dependente do conjunto de instruções do processador. Os algoritmos são apresentados a seguir.

4.4.1 *Semiadaptative Markov Compression (SAMC)*

Este método utiliza um codificador aritmético binário [45] e o modelo de *Markov* [16,46], utilizando como símbolos a serem codificados os campos de uma instrução. O objetivo de utlizar os campos de uma instrução é aproveitar a correlação existente entre eles que não acontece quando estes métodos são utilizados em compressão de textos. O campos são obtidos dividindo a instrução em subconjuntos de *bits* (*stream*). Por exemplo, na figura 4.13, temos a divisão para uma instrução do processador *MIPS R2000*. Após a divisão da instrução, é construído o modelo de *Markov* para cada um dos campos. Para o modelo de *Markov* é gerada uma árvore binária com 2^{ki+1} estados, onde k_i é o número de *bits* da cadeia. O estado inicial (S0), corresponde a nenhum *bit* de entrada. Os filhos da esquerda correspondem ao *bit* 0 e os da direita ao *bit* 1. Cada transição de estado tem sua probabilidade obtida pela análise total do programa. A figura 4.14 mostra um exemplo da árvore de *Markov*, para $k_i = 2$.

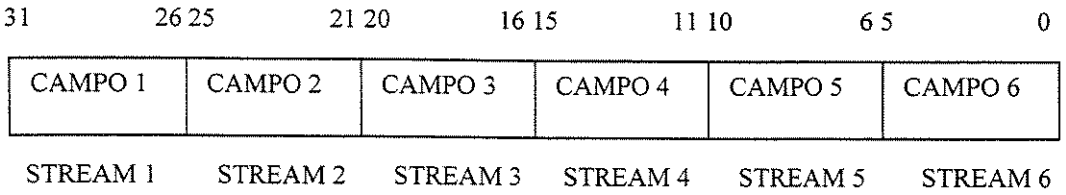


Figura 4.13 Exemplo de divisão de campos de uma instrução do *MIPS*

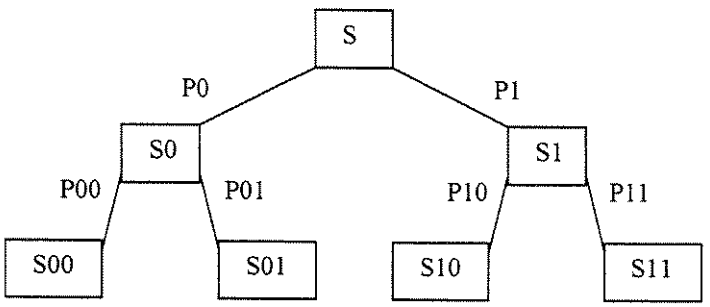


Figura 4.14 Árvore binária de *Markov*

Após o cálculo das probabilidades, o compressor as usa para a codificação aritmética binária. Esta abordagem é independente do formato da instrução, pois os campos de *bits* não necessitam ter nenhuma correlação entre si.

4.4.2 *Semiadaptative Dictionary Compression (SADC)*

Outro modo de compressão proposto por Lekatsas et al [44] utiliza um dicionário para comprimir *opcodes*, ou combinação de *opcodes* e registradores ou *opcodes* e imediatos. Aqui também as instruções são divididas em *streams*, mas de forma que haja dependência entre eles. Por exemplo, para uma instrução *MIPS* a divisão é feita em 4 diferentes *streams*: *opcode*, registradores, imediatos e imediatos longos. O dicionário mapeia índices em *opcodes* ou combinações de *opcodes*. A formação do dicionário leva em conta duas características. A primeira diz respeito à relação entre pares de *opcodes* adjacentes e à possível correlação entre eles. A segunda é relativa à geração de novos *opcodes* para

instruções que aparecem freqüentemente com registradores específicos como, por exemplo, a instrução *jr r31*. Para formar este dicionário, o gerador percorre o programa e cria uma árvore com todos os *opcodes* e suas freqüências, todos os grupos de 2 *opcodes* e suas freqüências e todos os grupos de 3 *opcodes* e suas freqüências. Após isto, o gerador insere no dicionário todos os *opcodes* , codificando o grupo de instruções adjacentes ou codificando o *opcode* com um registrador ou imediato específico.

4.4.3 Resultados Experimentais e Comentários

Utilizando o conjunto de programas SPEC95, Lekatsas et al obtiveram uma razão de compressão média de aproximadamente 57% para o *MIPS* e 75% para o PentiumPro. Estes valores se referem ao segundo método, sem considerar o tamanho da máquina de compressão. Este trabalho é o que mais se aproxima do nosso por não tratar os programas com uma simples seqüência de bits.

4.5 Outros Trabalhos

Ernst et al. [47] utilizam uma técnica, chamada de *patternization*, que propõe a codificação de cadeias de caracteres. A idéia básica é extrair todos os padrões de instruções possíveis de uma árvore do programa. Estes padrões de árvores são compilados em um gerador de código baseado no gerador proposto por Fraser et al. [6]. É feita então, uma análise da cobertura de árvores de expressão para determinar qual o melhor conjunto de padrões que cobrem o programa. As árvores de expressão originais do programa são substituídas pelos seus padrões e a seqüência de padrões é comprimida utilizando o utilitário de compressão *gzip* [28]. O formato comprimido é chamado *wire-format*, e resulta em uma razão de compressão próxima dos 30%. O principal objetivo de *patternization* é a síntese em uma máquina virtual eficiente que possa executar o código comprimido através de uma rede. O uso do *gzip* [28] torna esta técnica não apropriada para descompressão em tempo real porque ele requer muita memória auxiliar na hora da descompressão.

Frank e Thomas [48] propõem uma técnica de compressão independente do formato, denominada *slim binaries*. Nesta técnica, as árvores sintáticas abstratas são comprimidas em uma seqüência de símbolos em um dicionário adaptativo. As árvores sintáticas similares são codificadas usando a mesma seqüência. O esquema de compressão é baseado no algoritmo LZ [19], e os resultados da compressão são maiores que 50%. A técnica *slim binaries* é voltada para compressão de programas fontes e tem sido usada, desde 1993, para comprimir programas objetos no processador *MC68020*, que equipava os computadores Macintosh.

CAPÍTULO 5

Codificação Utilizando Fatoração de Operandos

Neste capítulo apresentaremos uma nova proposta para compressão de código objeto, que consiste em aplicar a idéia fatoração de operandos à compressão de código de programas.

A *fatoração de operandos* consiste na separação de cada *árvore de expressão* [41] (*expression-tree*) do programa em dois componentes: um chamado de *padrão de árvores (tree-pattern)*, que é formado pelo conjunto de instruções que forma a árvore de expressão, sem os operandos; e o outro chamado de *padrão de operandos (operand- pattern)*, que contém o conjunto de registradores e imediatos usados pelas instruções da referida árvore. Árvore de expressão é uma seqüência de instruções interligadas através da dependência de entre seus operandos. O objetivo da utilização da fatoração de operandos é aproveitar a repetição dos padrões resultantes da geração do código binário de um programa pelos compiladores. Esta característica vem da semelhança existente entre sentenças em uma linguagem de alto nível [11,12]. A separação dos operandos das instruções de uma árvore de expressão, resulta em dois conjuntos: um formado por padrões de árvores e outro por padrões de operandos. Cada um destes conjuntos é processado com o objetivo de encontrarmos um conjunto mínimo de padrões que cobrem todas as árvores de expressão de um programa [1]. A fatoração de operandos é baseada no conceito de padronização proposto por Proebsting [49] e utilizada por Fraser [6] como técnica de compressão para *byte-code* em máquinas virtuais de uma rede. Mostraremos a seguir as etapas para chegarmos ao conjunto de padrões de árvores e de operandos.

5.1 Compressão em Arquiteturas RISC.

Um dos processadores utilizados para verificarmos a eficiência da fatoração de operandos na compressão de programas foi o *MIPS R2000* [3,9]. Este processador, embora antigo, é uma arquitetura *RISC* clássica e possui muitas das características de um processador *RISC* moderno. Como programas de testes foi utilizado um subconjunto de programas pertencentes ao *SPEC CINT95* [26]. Estes programas foram compilados usando *gcc* versão 2.8.1, com a opção de otimização *O2*.

5.1.1 Determinação dos Blocos Básicos, Árvores de Expressão, Padrões de Árvores e Padrões de Operandos.

Para chegarmos ao conjunto de padrões de árvores e operandos, partimos inicialmente do programa codificado em linguagem de máquina. Primeiro o código é analisado a fim de localizar o ponto inicial do programa. A partir deste ponto, o programa é dividido em *blocos básicos* [41]. Um bloco básico é definido como uma seqüência de instruções consecutivas na qual o fluxo de controle entra no início da seqüência e deixa pelo seu fim, sem que haja qualquer desvio exceto no seu fim. A figura 5.1 ilustra este processo. Depois, cada bloco é dividido em árvores de expressão. Uma instrução é raiz de uma árvore se for uma instrução que armazena um valor na memória ou se o seu operando é a fonte de uma ou mais instruções dentro de um mesmo bloco básico. Em seguida os componentes referentes aos códigos das instruções e suas respectivas listas de operandos são, então, separados. A figura 5.2, utiliza o Bloco Básico #n+1 da figura 5.1 e mostra este processo.

Considerando o bloco básico mostrado na figura 5.2(a), podemos ver a floresta de árvores de expressão em 5.2(b). As árvores de expressão são formadas obedecendo à seqüência das instruções de um bloco básico e às respectivas dependências existentes entre os operandos destas instruções. Isto pode ser visto na figura 5.2(b), onde a árvore da esquerda tem o registrador *r28* como a ligação entre os dois nós.

```
            .....

            ori     r24,r0,630
            lw      r25,-32752(r28)
            addu    r15,r0,r31
            jalr    r31,r25
            ori     r24,r0,630
            addu    r8,r0,r31
            bgezal  r0,2
label:      or      r0,r0,r0
            lui     r28,4033
            addiu   r28,r28,22276
            sw      r6,0(r1)
            jalr    r31,r25
            .....
```

(a)

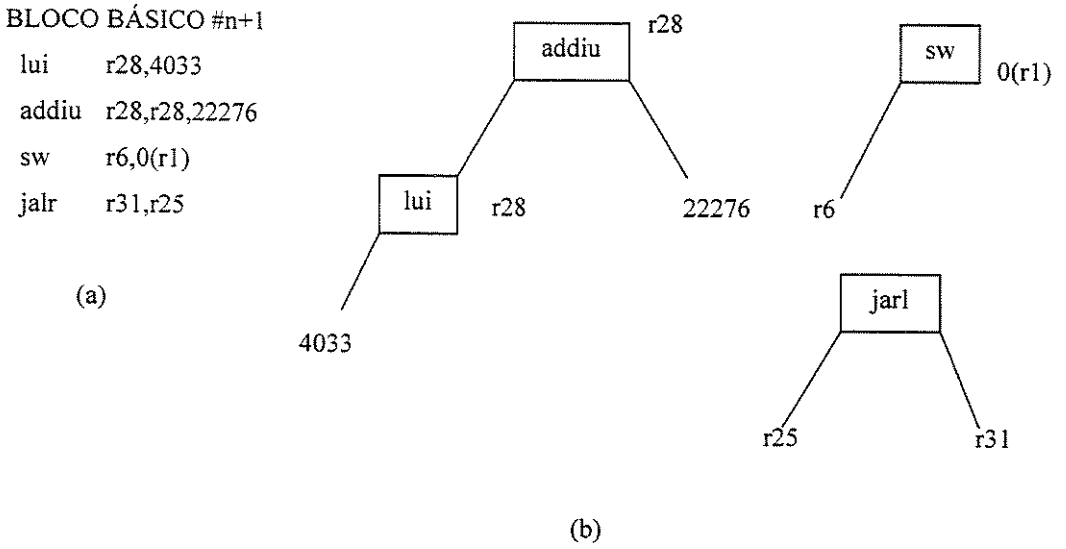
```
BLOCO BÁSICO #n-2

            ori     r24,r0,630
            lw      r25,-32752(r28)
            addu    r15,r0,r31
            jalr    r31,r2

BLOCO BÁSICO #n-1

            ori     r24,r0,630
            addu    r8,r0,r31
            bgezal  r0,2

BLOCO BÁSICO #n

            or      r0,r0,r0

BLOCO BÁSICO #n+1

            lui     r28,4033
            addiu   r28,r28,22276
            sw      r6,0(r1)
            jalr    r31,r25
```

(b)

Figura 5.1 a) Código original; b) Divisão em blocos básicos

A figura 5.2(c) mostra os padrões de árvore resultante após os operandos terem sidos retirados das árvores de expressão originais. Os asteriscos são usados no lugar dos operandos originais. Um padrão de operandos é formado pesquisando a seqüência de instruções da árvore e listando os operandos quando encontrados. A figura 5.2(d) mostra os padrões de operandos determinados após a fatoração das árvores de expressão. Eles consistem numa lista formada pelas folhas das árvores de expressão e os registradores de

vértices intermediários, quando estes são percorridos da direita para a esquerda e de baixo para cima.



PADRÃO DE ÁRVORE #1

addiu *,*,*

lui *,*

PADRÃO DE OPERANDOS #1

4033, r28, 22276, r28

PADRÃO DE ÁRVORE #2

sw *,*

PADRÃO DE OPERANDOS #2

r6, 0(r1)

PADRÃO DE ÁRVORE #3

jarl *,*

PADRÃO DE OPERANDOS #3

r25,r31

(c)

(d)

Figura 5.2 (a) Bloco básico; (b) Floresta de árvores de expressão (c) Padrões de árvores (d) Padrões de operandos

5.1.2 Identificação de Padrões Distintos

Os programas utilizados para testes tiveram seus blocos básicos determinados, assim como as árvores de expressão e os seus respectivos padrões de árvores e de operandos. Chamamos de padrões de árvores ou de operandos distintos o conjunto de padrões únicos, ou seja, o conjunto resultante da eliminação dos padrões repetidos. A tabela 5.1 mostra o número de árvores expressão, padrões de árvores distintos e padrões de operandos distintos achados para o conjunto de programas de teste.

PROGRAMA	ÁRVORES DE EXPRESSÃO	PADRÕES DE ÁRVORES - (%)	PADRÕES DE OPERANDOS – (%)
go	54651	578 - (1,1)	12561 - (23,0)
li	15722	158 - (1,0)	3048 - (19,4)
compress	1444	125 - (9,0)	731 - (51,0)
perl	62915	648 - (1,0)	11209 - (18,0)
gcc	311488	1547 - (0,5)	41486 - (13,3)
vortex	128104	471 - (0,4)	16143 - (13,0)
ljpeg	38426	767 - (2,0)	9839 - (26,0)

Tabela 5.1 Número de padrões de árvores e operandos em um programa

Na tabela, notamos que o número de padrões de árvores e de operandos distintos em um programa não é somente pequeno, mas também muito menor que o número total de árvores de expressão em cada programa. Como podemos ver na tabela 5.1, o programa *gcc* tem 311488 árvores de expressão que podem ser representadas por 1547 padrões de árvores distintas, isto é, apenas 0,5% de todas as árvores. Este pequeno número é decorrente de alguns fatores, tais como: o pequeno número de instruções no conjunto de instruções de um processador *RISC*; o tamanho da maioria das árvores de expressão e, portanto, das poucas combinações possíveis de suas instruções; e da maneira determinística com que os

compiladores geram os códigos durante a construção de *árvores sintáticas abstratas*, tais como as que representam os comandos *if-then-else* e laços *for* [41].

Com relação aos padrões de operandos, encontramos um comportamento semelhante, como podemos ver na tabela 5.1. Por exemplo, o programa *gcc* tem 311488 seqüências de operandos e estas podem ser representadas por 41486 padrões de operandos, isto é, 13,3% de todas as seqüências. Estes padrões têm maior uniformidade na distribuição quando comparados aos padrões de árvores. Isto acontece porque é grande o número de combinações entre os registradores e imediatos e as instruções em um padrão de operandos.

Com os dados da tabela 5.1, podemos concluir que os compiladores têm a tendência de gerar programas com muita redundância de código. Observamos também que programas pequenos apresentam menor redundância; por exemplo, no programa *compress* (o menor programa analisado), os padrões de árvores correspondem a 9% de todas as possíveis árvores no programa, enquanto que os padrões de operandos são 51% de todas as seqüências de operandos. Dois fatores explicam esta característica. Primeiro é o próprio tamanho, ou seja, programas grandes utilizam mais combinações das instruções e dos operandos disponíveis no conjunto de instruções de uma arquitetura fazendo com que ocorram mais repetições de padrões. Segundo, as estruturas típicas de linguagens de alto nível, tais como como *if-then-else* e *for*, são compiladas utilizando conjuntos de padrões similares. Quanto maior o programa maior a repetição destas estruturas e portanto maior a redundância. Com este experimento, podemos concluir que qualquer código programa gerado pelo mesmo compilador tem um conjunto comum de padrões que são utilizados muitas vezes (poucas) quando os programas são grandes (pequenos).

5.1.3 Freqüência e Tamanhos de Padrões em Programas

Uma outra questão que estudamos é como cada padrão aparece em um programa, ou seja, qual a freqüência de ocorrência deste padrão no programa e qual é a sua influência no tamanho deste programa, em número de *bits*. Para isto, realizamos dois experimentos, discutidos a seguir.

- **Freqüência de Padrões**

Neste primeiro experimento calculamos as freqüências individuais de cada padrão de árvores, com relação à sua ocorrência e as ordenamos decrescentemente. Para cada padrão, calculamos, então, a porcentagem acumulativa das árvores de expressão cobertas por ele. Podemos ver os resultados na figura 5.3. Analisando os gráficos desta figura, concluímos que a freqüência dos padrões de árvores decrescem quase exponencialmente, à medida que eles se tornam menos freqüentes, ou seja, padrões de árvores freqüentes cobrem mais árvores de expressão que os não tão freqüentes. Esta figura mostra ainda que, em média, 20% dos padrões de árvores correspondem a quase todas as árvores no programa. A exceção é o programa *compress*, pois, como é um programa relativamente pequeno, a distribuição das suas árvores de expressão é mais uniforme, conforme mencionado anteriormente.

Encontramos um comportamento semelhante para os padrões de operandos. A figura 5.4 mostra o número acumulativo de seqüências de operandos de um programa, que são cobertos por padrões de operandos distintos. Como visto nesta figura, 20% dos padrões de operandos cobrem em média 80% das seqüências de operandos de um programa. Novamente, isto não vale para o programa *compress*, pelo motivo discutido acima.

- **Freqüência relacionada ao tamanho do padrão**

O segundo experimento se concentrou em analisar a influência dos padrões no tamanho do programa, expresso em *bits*. Padrões freqüentes não necessariamente irão contribuir para a maioria dos *bits* do programa. Alguns padrões não muito comuns podem contribuir mais que outros mais freqüentes por terem um número grande de *bits*, compensando as poucas vezes que eles aparecem.

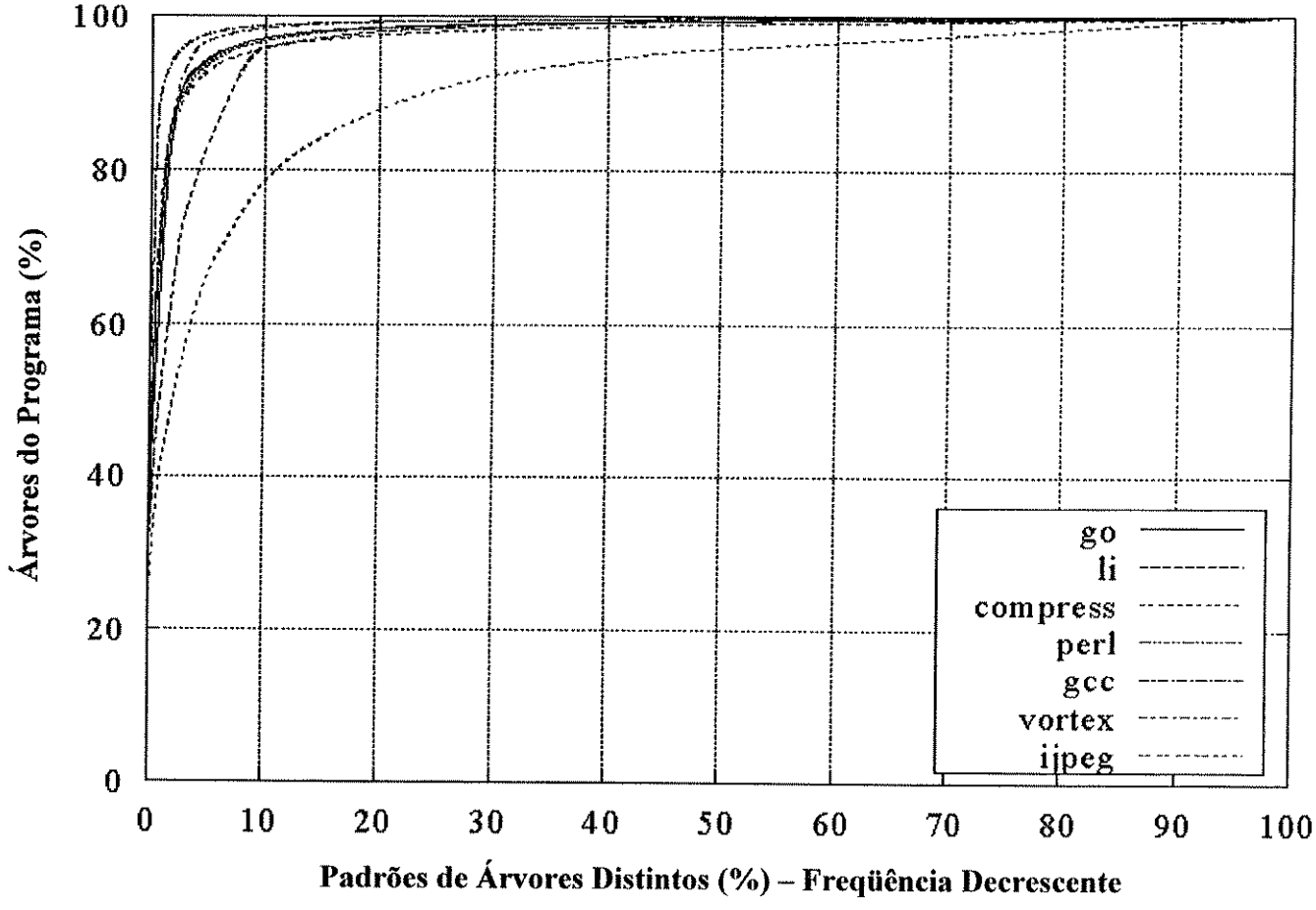


Figura 5.3 Porcentagem de cobertura das árvores de expressão pelos padrões de árvores distintos

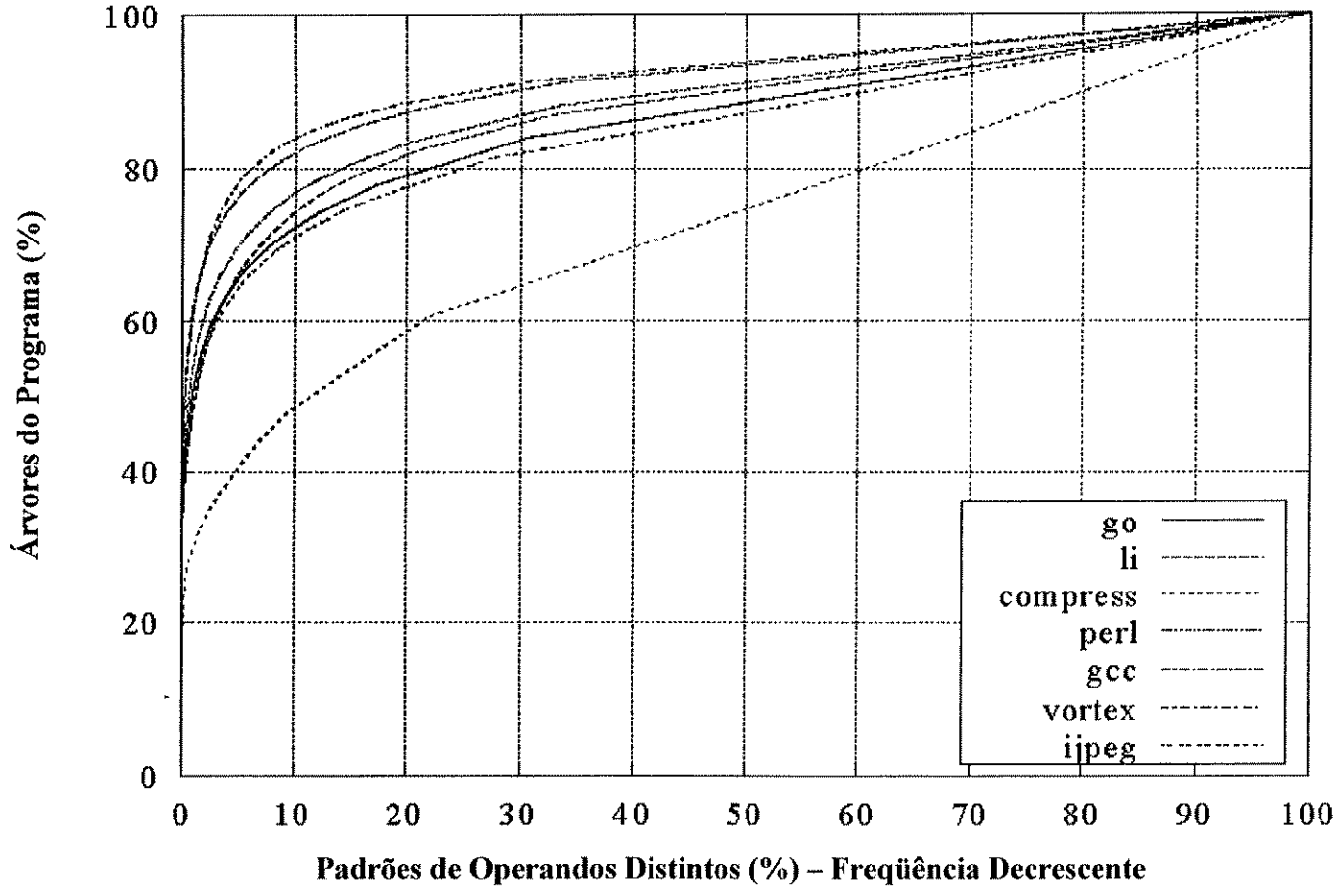


Figura 5.4 Porcentagem de cobertura das seqüências de operandos pelos padrões de operandos distintos

A figura 5.5 mostra o gráfico, interpolado pelo método de Bezier, da distribuição de tamanho dos padrões de árvores. Para a maioria dos casos, padrões de árvores com freqüência média têm mais *bits* que padrões comuns ou não comuns. Encontramos o mesmo comportamento para os padrões de operandos. A contribuição de cada padrão, em termos de *bits* do programa, é calculada multiplicando-se o número de ocorrências pelo tamanho do padrão e dividindo o resultado pelo tamanho total do programa expresso em número de *bits*.

A figura 5.6 mostra a porcentagem acumulada dos *bits* do programa devido aos padrões de árvores. Estes padrões estão ordenados decrescentemente, baseados na sua contribuição para o tamanho do programa. Eles contribuem com pelo menos 35% do total de *bits* do programa. Isto porque os padrões de árvores correspondem apenas aos *bits* referentes aos *opcodes* e na arquitetura *MIPS R2000* pelo menos 11 *bits* são usados para o *opcode*, isto é, 35,2% de uma instrução [9]. O gráfico desta figura revela ainda, que 20% dos padrões de árvores correspondem a aproximadamente 32% de todos os *bits* do programa. Na figura 5.7 podemos ver o gráfico correspondente para os padrões de operandos. Neste caso, a contribuição de padrões de operandos é mais dispersa para os diferentes programas, devido aos diferentes tamanhos tanto dos imediatos como dos registradores nas instruções do processador *MIPS R2000*. Entretanto, é possível dizer que 20% de padrões de operandos correspondem a 50% +/- 5% de todos os *bits* do programa.

5.2 Compressão em Arquiteturas *CISC*

O objetivo deste experimento é verificarmos a validade da fatoração de operandos para um processador *CISC*. O processador utilizado aqui foi o *TMS320C25* [5] da *Texas Instruments Inc.*. Ele é um processador com conjunto de instruções extremamente codificado, utilizado em Processamento Digital de Sinais – *DSP*. Suas instruções têm um tamanho de 16 *bits* e formatos que podem ser codificados em 16 tipos diferentes, utilizando números de bits diferentes para representar o *opcode*. São 3 os modos de endereçamentos, como visto na seção 2.2 [5]:

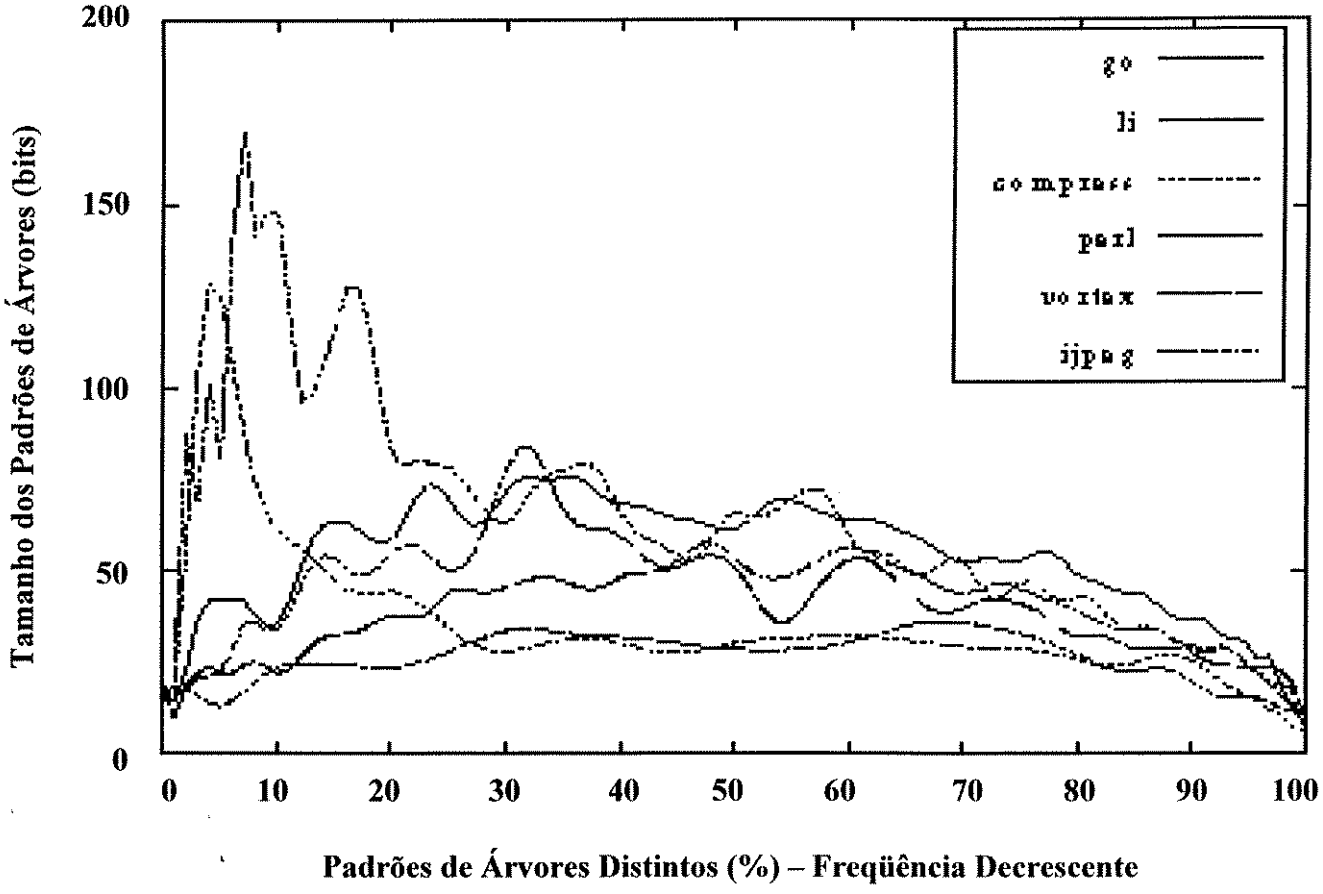


Figura 5.5 Distribuição da média do tamanho dos padrões de árvores distintos

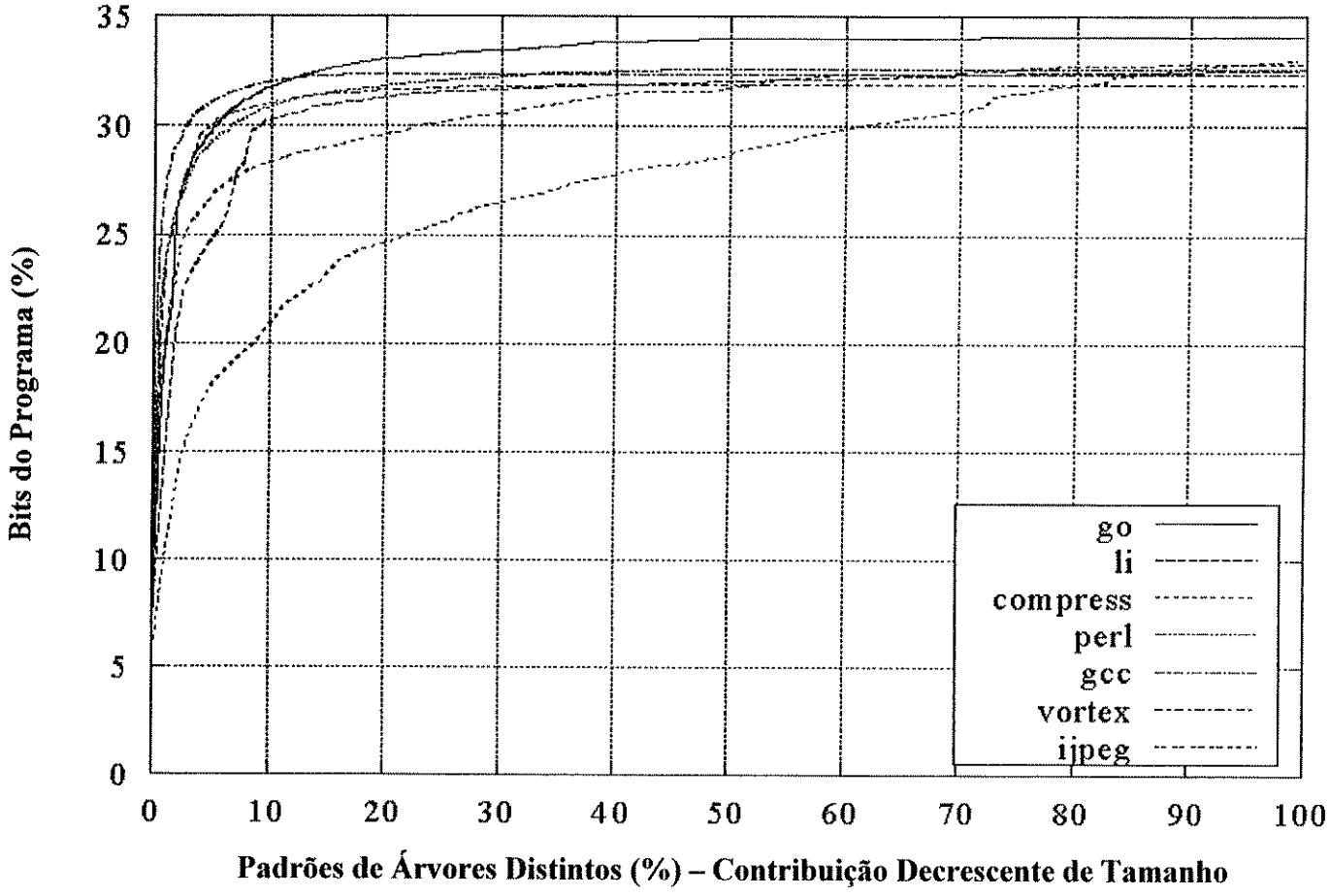


Figura 5.6 Porcentagem de *bits* do programa devido aos padrões de árvores distintos

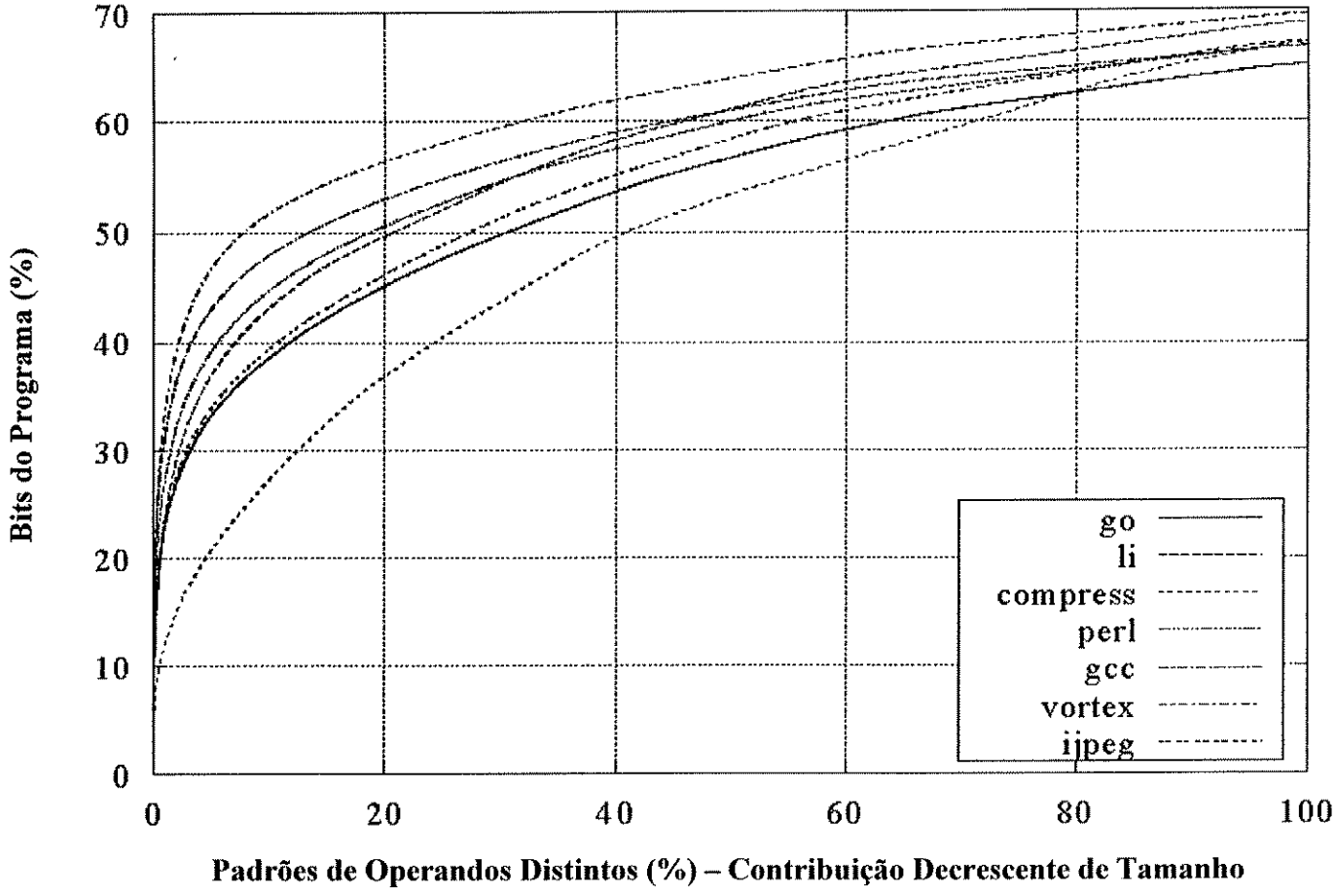


Figura 5.7 Porcentagem dos *bits* do programa devido aos padrões de operandos distintos

- direto: em que o endereço do operando é codificado no campo da instrução;
- indireto: em que o formato do operando é dado por *<ind,next>*, sendo que *ind* representa uma operação de um auto-incremento (+ ou -) com o registrador de endereços corrente *AR*, e *next* representa o próximo registrador de endereços corrente;
- imediato: em que o valor do operando é codificado no campo da instrução.

Para estudar a fatoração de operandos em uma arquitetura *CISC*, utilizamos um conjunto de testes representativo de aplicações de *DSPs*, formados pelos programas descritos a seguir. O programa *jpeg* é uma implementação do algoritmo de compressão de imagens *JPEG*; o programa *bench* é um controlador de *cache* de disco, o *gzip* é um algoritmo de compressão; *set* é um conjunto de rotinas de manipulação de *bits* para aplicações usando *DSP*; os programas *hill* e *gnucrypt* são programas para criptografia de dados e, por fim, o programa *rx* que é uma rotina de controle para máquinas de estados embutidas. Os códigos foram compilados usando o compilador *TI TMS320C25* com a opção de otimização *O2*.

5.2.1 Determinação dos Blocos Básicos, Árvores de Expressão, Padrões de Árvores e Padrões de Operandos.

Como visto anteriormente, a operação básica na fatoração é a remoção dos operandos de uma instrução. Para isto então, é necessário dividir o programa em blocos básicos, construir as árvores de expressão e extrair os operandos. O algoritmo para determinação dos blocos básicos consiste em identificar as instruções de desvios e as instruções que estão localizadas em endereços alvos deste tipo de instruções. As árvores de expressão são construídas, dentro de cada bloco básico, levando-se em conta se a instrução armazena na memória ou se o operando destino da instrução é fonte de uma ou mais instruções dentro do bloco básico. Os padrões de árvore e de operandos são derivados do processo de fatoração destas árvores de expressão. As figuras 5.8, 5.9 e 5.10 mostram este processo utilizando o código para o processador *TMS320C25*.

```
.....

LT    *+,AR6
MPY   *+,AR5
LDPK  STATIC_3
LAC   STATIC_3
APAC
SACL  STATIC_3
BANZ  L4,*-,AR7
ADDK  1
SACL  STATIC_3

.....
```

(a)

```
BLOCO BÁSICO #n-1

LT    *+,AR6
MPY   *+,AR5
LDPK  STATIC_3
LAC   STATIC_3
APAC
SACL  STATIC_3
BANZ  L4,*-,AR7

BLOCO BÁSICO #n

ADDK  1
SACL  STATIC_3
```

(b)

Figura 5.8 (a) Trecho do código original em linguagem de montagem; (b) Divisão em Blocos Básicos.

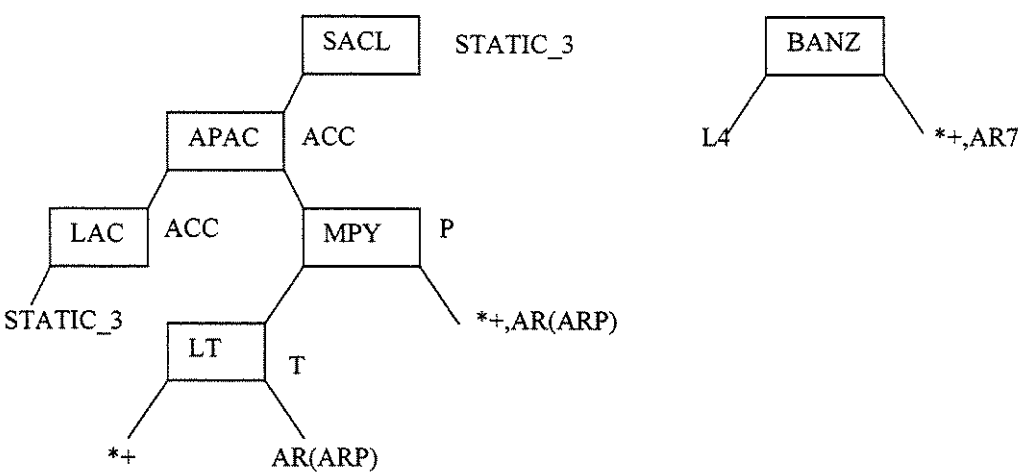


Figura 5.9 Árvores de expressão correspondentes ao bloco básico #n-1 da figura 5.8

PADRÃO DE ÁRVORE #1	PADRÃO DE OPERANDOS #1
SACL APAC LAC MPY LT	STATIC_3; *+; AR(ARP); *+,AR(ARP); STATIC_3
PADRÃO DE ÁRVORE #2	PADRÃO DE OPERNDOS #2
BANZ	L4; *+,AR7

Figura 5.10 Padrões de árvores e de operandos referentes às árvores da figura 5.9

5.2.2 Identificação de Padrões Distintos

Extraímos do conjunto de programas de testes os blocos básicos, as árvores de expressão e então fizemos a fatoração de operandos, gerando os padrões de árvores e de operandos. A tabela 5.2 mostra os números resultantes deste processo.

PROGRAMA	ÁRVORES DE EXPRESSÃO	PADRÕES DE ÁRVORES (%)	PADRÕES DE OPERANDOS (%)
aipint2	1043	90 (8.6)	285 (27.3)
bench	9483	572 (6.0)	2263 (23.9)
gnucrypt	3682	263 (7.1)	778 (21.1)
gzip	10835	582 (5.4)	2354 (21.7)
hill	920	121 (13.2)	279 (30.3)
jpeg	2305	190 (8.2)	563 (24.4)
rx	563	61 (10.8)	114 (20.3)
set	4565	319 (7.0)	1084 (23.7)

Tabela 5.2 Número de padrões de árvores e operandos nos programas

Podemos notar que o número de padrões de árvores e de operandos distintos é pequeno com relação ao número total de árvores de expressão. Por exemplo, para o programa *gzip*, os padrões de árvores distintos correspondem a apenas 5,4% do total de árvores do programa, enquanto que para os padrões de operandos distintos, correspondem a 21,7%. Os resultados encontrados reforçam ainda mais os argumentos, discutidos no item 5.1.2, de que um compilador tende a gerar padrões similares para estruturas tipo *if-then-else* e *for* e que quanto maior (menor) o programa mais (menos) padrões comuns ele gera. A partir da tabela 5.2 podemos calcular a média de freqüência das árvores distintas, que corresponde a apenas 8,3% das seqüências de *opcodes* do programa, enquanto que para os padrões de operandos, a média calculada foi de 24,1%.

5.2.3 Freqüência e Tamanho de Padrões nos Programas

Os experimentos feitos para o processador *MIPS R2000* foram repetidos aqui com o objetivo de avaliar se as conclusões tiradas no item 5.1.3, também seriam observadas para *TMS320C25*.

- **Freqüência de Padrões**

Neste experimento, calculamos e ordenamos decrescentemente as freqüências individuais de cada padrão, com relação à sua ocorrência no programa. As figuras 5.11 e 5.12 mostram, respectivamente, a cobertura dos padrões de árvores e de operandos, com relação ao total das árvores de expressão do programa. O que podemos concluir com os gráficos é que a freqüência dos padrões decresce exponencialmente à medida que os padrões se tornam menos freqüentes. Os resultados foram semelhantes aos encontrados para o processador *MIPS R2000* vistos no item 5.1.3.

- **Freqüência relacionada com o tamanho do padrão**

Os padrões, tanto de árvores como de operandos, foram ordenados de acordo com sua contribuição no programa, em número de *bits*. A contribuição cumulativa dos padrões ao tamanho do programa foram calculadas e podem ser vistas nas figuras 5.13 e 5.14.

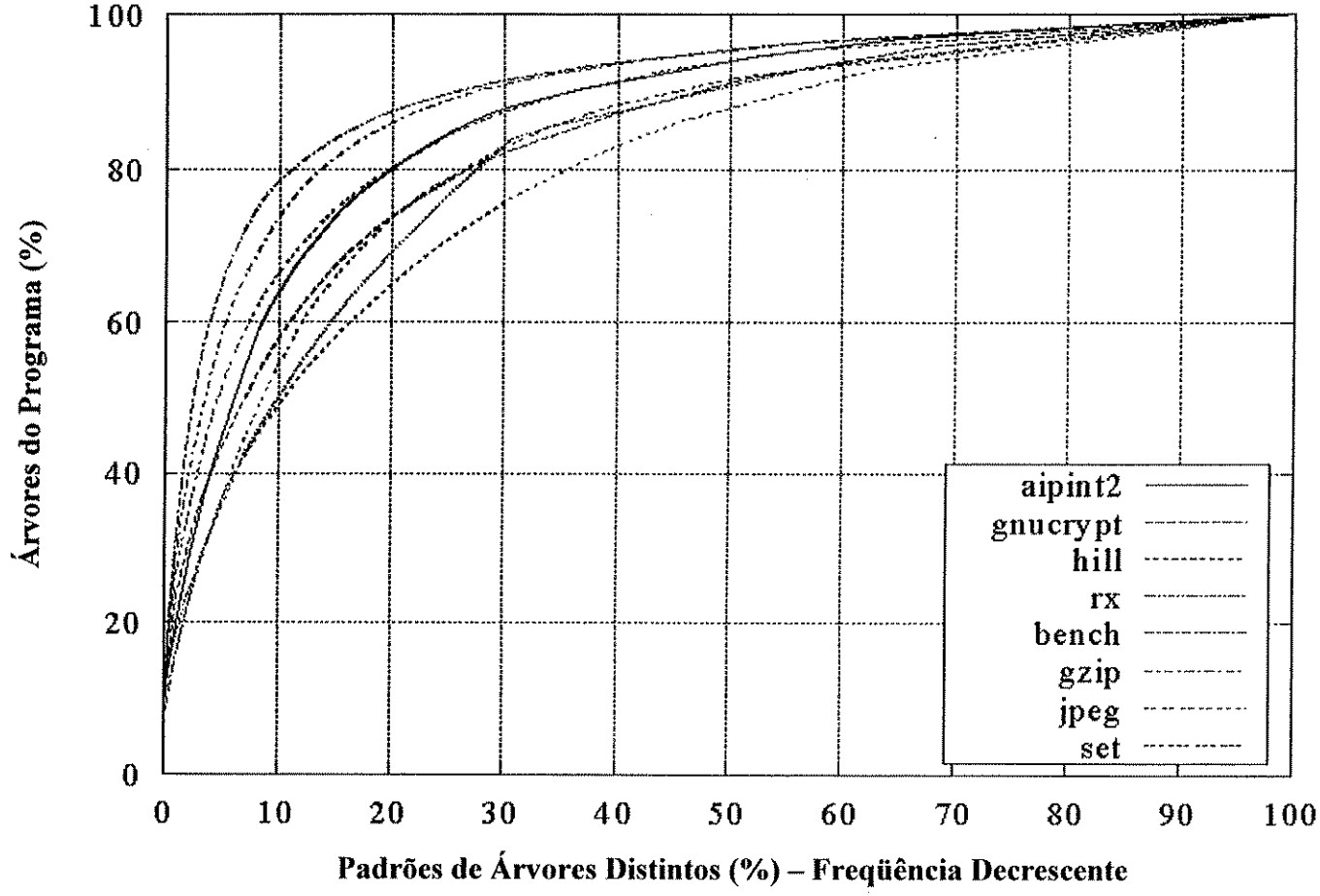


Figura 5.11 Porcentagem de cobertura de árvores de expressão pelos padrões de árvores distintos

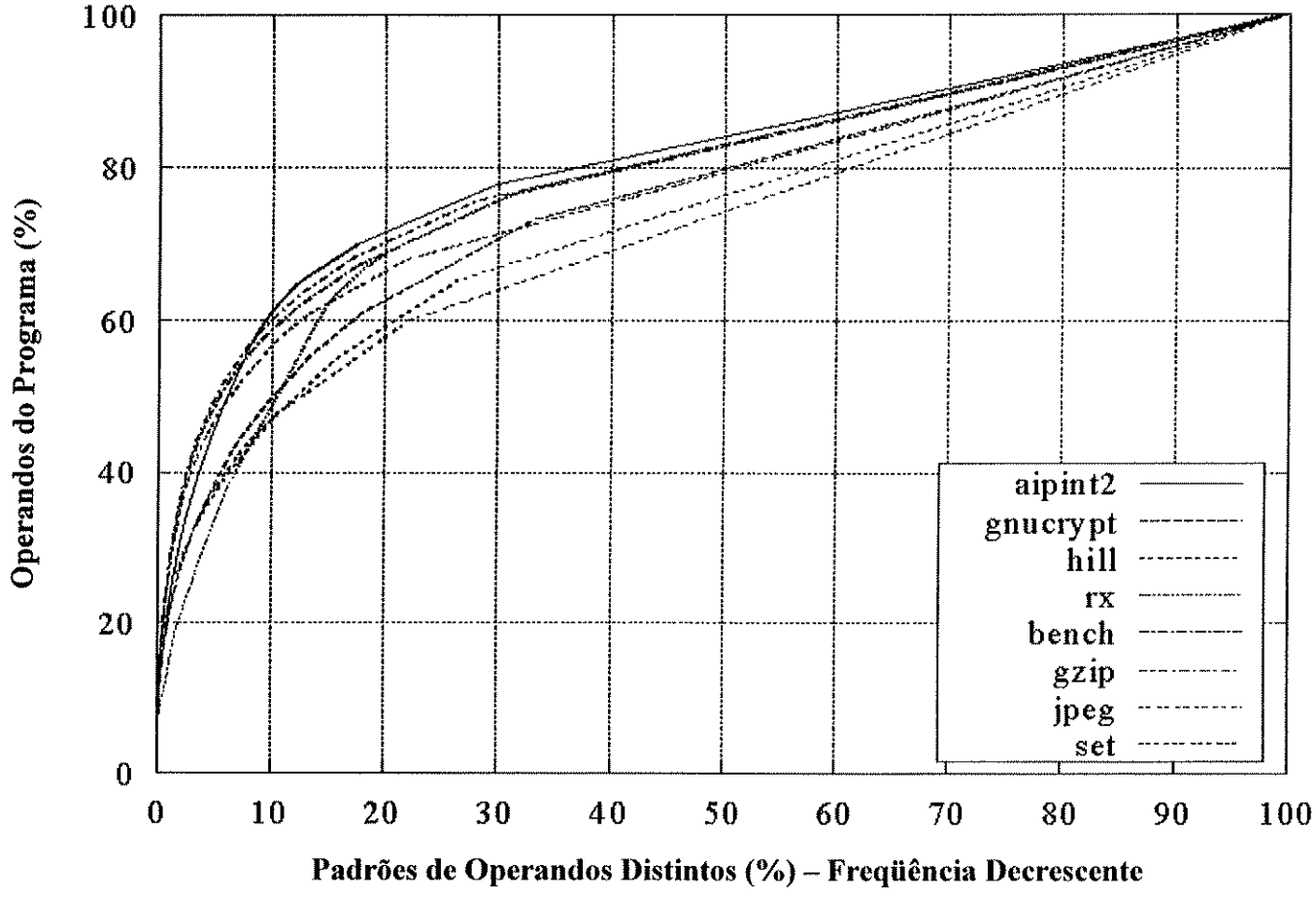


Figura 5.12 Porcentagem de cobertura das seqüências de operandos pelos padrões de operandos distintos

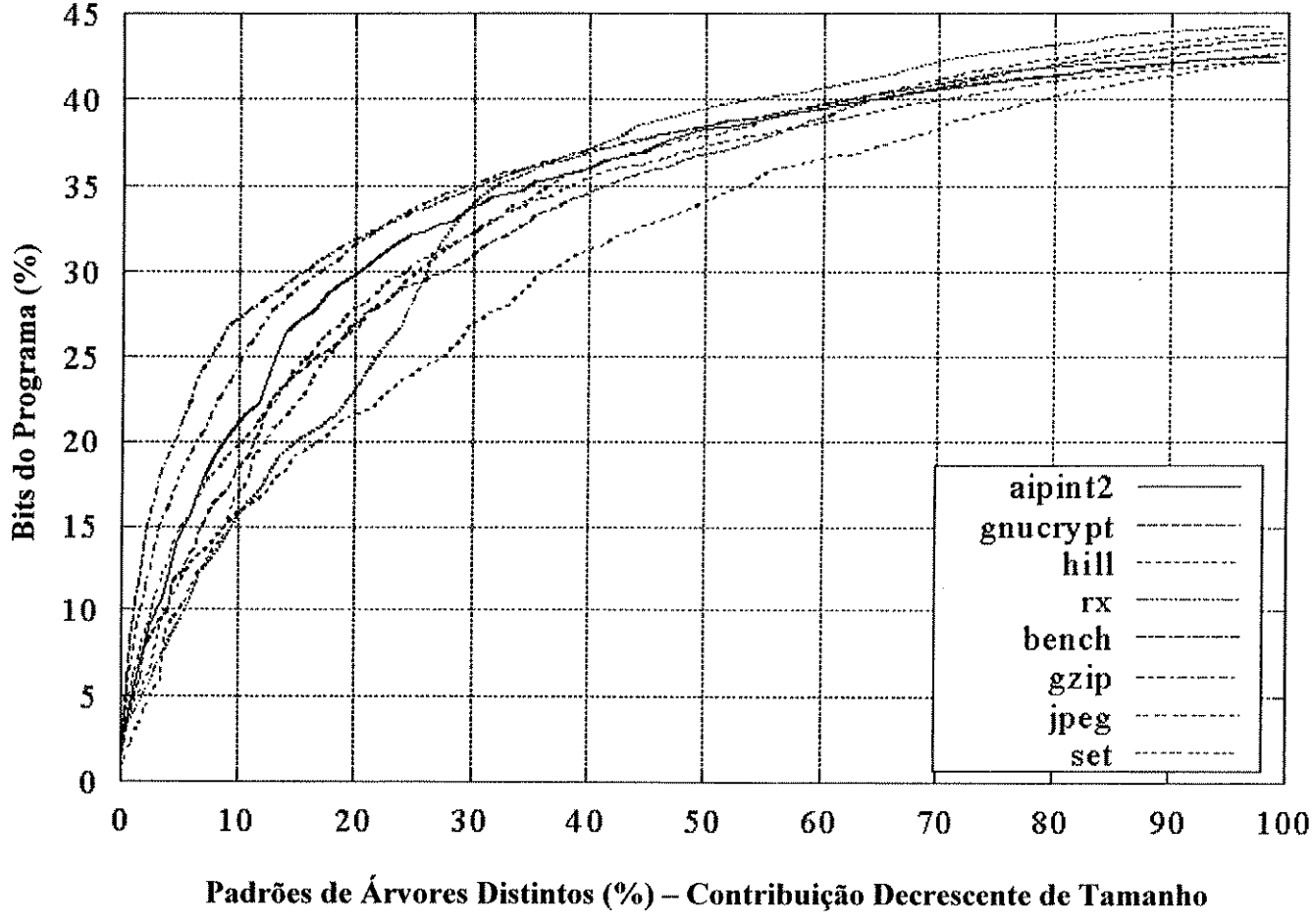


Figura 5.13 Porcentagem do tamanho do programa em *bits* devido aos padrões de árvores distintos

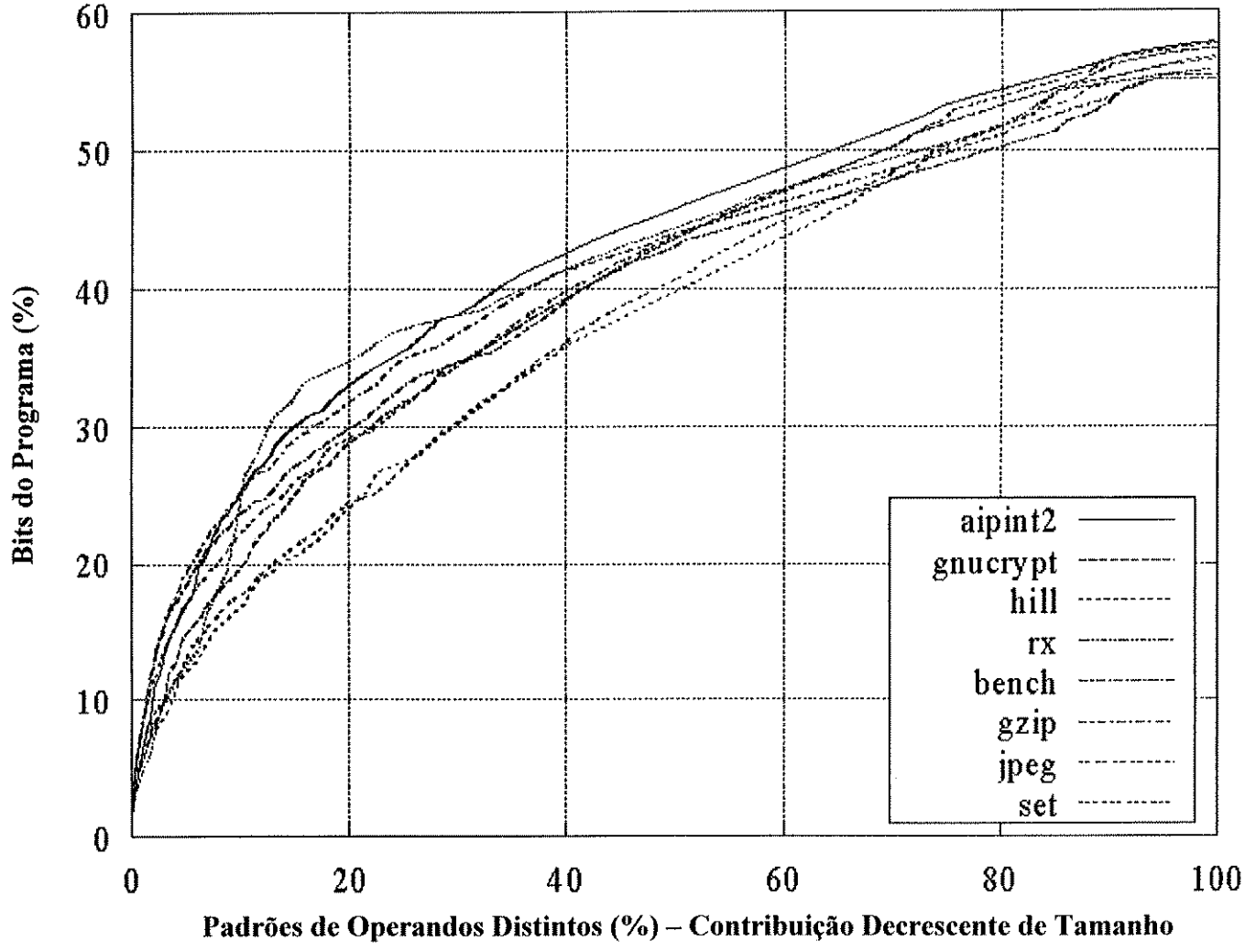


Figura 5.14 Porcentagem do tamanho do programa em *bits*, devido aos padrões de operandos distintos

A figura 5.13 mostra a porcentagem dos *bits* dos programas coberta pelos padrões de árvores, enquanto que a figura 5.14 mostra a porcentagem coberta pelos padrões de operandos. Podemos ver que a contribuição de padrões de árvores e de operandos em programas para *DSP* não é uniforme, eles têm uma distribuição exponencial, isto é, um pequeno conjunto de padrões cobre um número grande de árvores do programa. Nas figuras, podemos ver que quase 20% dos *bits* dos programas são cobertos por apenas 10% dos padrões de árvores, enquanto que os 10% mais freqüentes dos padrões de operandos cobrem 19% dos *bits* dos programas.

5.3 Algoritmo de Compressão

Os experimentos vistos nos itens 5.1 e 5.2 nos mostraram que uma pequena porcentagem de padrões pequenos de árvores e operandos cobrem a maioria dos *bits* do programa. Isto confirma, para o caso de instruções, a observação feita por Lefurgy [25] sobre o papel desempenhado por pequenas cadeias de *bits* no código do programa. A fatoração de operandos possibilita a utilização da correlação existente entre os padrões de árvores e de operandos, características não levadas em consideração pelas outras técnicas de compressão de programas. Por exemplo, um algoritmo que faz compressão seqüencial, como o LZ [19], não é capaz de detectar um padrão simples formado por *lw* *, *, * seguido por *add* *, *, *. Qualquer algoritmo não seqüencial que considera um programa como um conjunto de cadeias de *bits* também não o detectará. Considere, por exemplo, o padrão de árvores *lw* *, *, * e um processador que codifica o *opcode* e os registradores de destino, nesta ordem, usando 6 *bits* cada. Se um *byte* é escolhido como tamanho do símbolo codificado, então o primeiro *byte* de cada uma das instruções *lw r2*, *, * e *lw r15*, *, * serão codificadas como duas palavras de código diferentes, mesmo se o padrão *lw* *, *, * cobrir uma grande parte do programa. A fatoração de operandos identifica os padrões de operandos que são utilizadas por dois ou mais padrões de árvores. Por exemplo, no programa *gcc* , o padrão de operandos *[r2, r0, r4]* é usado nas instruções *xor r2, r0, 4* e *sub r2, r0, r4*. Esta redundância é usada para gerar códigos comprimidos menores. O

aproveitamento destas características é a razão pela qual codificamos padrões de árvores e operandos separadamente.

No processo de compressão, codificamos as árvores de expressões como um par de palavras código [*Tp,Op*], em que *Tp* é uma palavra código para um padrão de árvore e *Op* para um padrão de operandos. Dividimos o algoritmo de codificação em duas fases: na primeira, os padrões de árvores e de operandos são codificados para na segunda ser feita a codificação das palavras de código em um arquivo, gerando assim o programa comprimido.

5.3.1 Codificação dos Padrões

As análises feitas nos itens 5.2 e 5.3 mostram-nos que os padrões de árvores e de operandos têm distribuições não uniformes. Isto sugere que algoritmos de codificação, com comprimento variável, possam resultar em razões de compressão melhores. Por outro lado, a codificação com comprimento variável implica em uma decodificação com baixa eficiência. As principais características envolvidas na codificação são: detecção do comprimento da palavra de código, extração da palavra código e alinhamento dos *bits* desta palavra. Embora estes problemas possam ser tratados pelo projeto de uma máquina eficiente de decodificação, como mostrado por Bènes [24], eles ainda representam um trabalho adicional para descompressão.

Estudamos quatro métodos para codificar padrões de árvores e de operandos. Os dois primeiros métodos envolvem a codificação com comprimento fixo e a codificação utilizando o método de *Huffman*. As figuras 5.15 e 5.16 ilustram estes métodos. Nestas figuras, *CODEWORD* indica a coluna onde estão as palavra de código geradas para os padrões de árvore distintos. Cada padrão de árvore tem um conjunto de instruções que está descrito em *NAME*. Cada instrução do padrão de árvores tem o tipo da instrução (que nos dá o formato da instrução) descrito em *#TYPE* e o número de operandos que ela usa descrito em *#NRANDS*. Por exemplo, seja a árvore de expressão:

addu r8,r0,r3
addu r2,r2,r4

Em *CODEWORD*, estaria a palavra código *100*, correspondente ao padrão de árvore descrito em *NAME* (*addu, addu*). Em *OPCODE*, estariam os *opcodes* correspondentes a estas duas instruções (*100001, 100001*). Em *#TYPE*, o tipo delas, no caso ambas do tipo 3, ou seja, *R-TYPE* [9] e em *#RANDS* o número de operandos de cada instrução, aqui ambas com 3 operandos.

Na figura 5.15, podemos ver que as palavras de código têm o mesmo tamanho, pois utilizamos o método de codificação fixa. O tamanho de cada palavra é igual a 2^n, onde n é o número de padrões de árvores distintos. A figura 5.16 nos mostra a codificação usando *Huffman*. Podemos ver que as palavras de códigos têm tamanho variável e que os padrões que mais influenciam no tamanho do programa têm os menores códigos.

CODEWORD

000

# OPCODE	# TYPE	# RANDS	NAME
100101	11	11	or

001

# OPCODE	# TYPE	# RANDS	NAME
100001	11	11	addu

010

# OPCODE	# TYPE	# RANDS	NAME
100011	01	10	lw

011

# OPCODE	# TYPE	# RANDS	NAME
001001	10	10	jalr

100

# OPCODE	# TYPE	# RANDS	NAME
100001	11	11	addu
100001	11	11	addu

Figura 5.15 Exemplo de codificação usando comprimento fixo

CODEWORD

01

# OPCODE	# TYPE	# RANDS	NAME
100101	11	11	or

111

# OPCODE	# TYPE	# RANDS	NAME
100001	11	11	addu

110

# OPCODE	# TYPE	# RANDS	NAME
100011	01	10	lw

0010

# OPCODE	# TYPE	# RANDS	NAME
100001	11	11	addu
100001	11	11	addu

0001

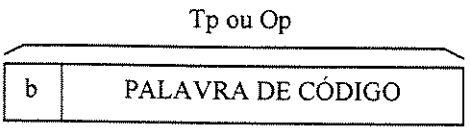
# OPCODE	# TYPE	# RANDS	NAME
101011	01	10	sw

00001

# OPCODE	# TYPE	# RANDS	NAME
000100	10	10	beq

Figura 5.16 Exemplo de codificação usando o método de *Huffman*

Os outros dois métodos levam em consideração o impacto que os métodos de codificação devem ter no desempenho da máquina de descompressão. Eles são chamados *Bounded Huffman* (BH) [18] e *Variable Length Coding* (*VLC*) [50]. Em ambos os métodos adicionamos um *bit* diferenciador no início de cada palavra código para diferenciar quando estamos usando a codificação pelo método *Huffman* ou *VLC* de quando estamos utilizando a codificação com comprimento fixo. A figura 5.17 ilustra a utilização do bit diferenciador. Nesta figura podemos ver que a palavra de código é formada pelo bit diferenciador (*escape bit*) e pela palavra de código do padrão de árvore (*Tp*) ou de operandos (*Op*) distinto. Quando a palavra é codificada em Huffman ou VLC, o valor do escape bit será 0 e, quando ela estiver codificada em comprimento fixo, assumirá valor 1.



ESC BIT(b)	CODEWORD HUFF.	CODEWORD VLC
0	10	10
	010	11
	0110	010
	0111	011
	1101	0010
	0000	0011
	0001	00010
	1111	00011
	.....	.....

ESC BIT(b)	CODEWORD FIXO (5bits)
1	00000
	00001
	00010
	00011
	00100
	00101
	00110
	00111
	.....

Figura 5.17 Codificação de padrões de árvores ou de operandos utilizando o *escape bit*

O método *Bounded Huffman* também é usado por Haskell [50] e por Wolfe [20]. É um método que possibilita restringir o tamanho das palavras código codificadas em *Huffman*, reduzindo assim a complexidade da máquina de descompressão, pois não precisa ir na memória mais de uma vez para pegar a palavra de código e trabalha com poucos tamanhos diferentes destas palavras. Apesar deste método utilizar a codificação baseada em *Huffman*, que é uma técnica otimizada, como mostrada por Bell [17], este tipo de codificação apresentará taxas de compressão maiores, devido à introdução de códigos de comprimento fixo. A figura 5.18 mostra este método.

O método *VLC* é uma variação da codificação *MPEG-VLC* proposta por Haskell [50]. Neste método, formamos as palavras de código selecionando alguns símbolos pré-determinados e introduzimos um certo número de zeros no início de cada uma, a fim de codificar o tamanho da palavra código. O principal objetivo deste método é simplificar a lógica associada à máquina de decodificação responsável pela extração das palavras código. A figura 5.19 ilustra este método.

CODEWORD

	# OPCODE	# TYPE	# RANDS	NAME
01	100101	11	11	or
111	100001	11	11	addu
110	100011	01	10	lw
0000	001001	10	10	jalr
0001	100001	11	11	addu
	100001	11	11	addu
0010	101011	01	10	sw

	#OPCODE	#TYPE	#RANDS	NAME
0011	000100	10	10	beq
0100	100011	01	10	lw
0101	001001	10	10	jalr
0110	100001	11	11	addu
	100001	11	11	addu
0111	101011	01	10	sw

Figura 5.18 – Exemplo de codificação usando o método de *Bounded-Huffman*.

CODEWORD

10

# OPCODE	# TYPE	# RANDS	NAME
100101	11	11	or

11

# OPCODE	# TYPE	# RANDS	NAME
100001	11	11	addu

010

# OPCODE	# TYPE	# RANDS	NAME
001000	11	01	jr

011

# OPCODE	# TYPE	# RANDS	NAME
001001	10	10	jalr

0010

# OPCODE	# TYPE	# RANDS	NAME
000001	00	10	bgez

0011

# OPCODE	# TYPE	# RANDS	NAME
000100	10	11	beq

00010

#OPCODE	#TYPE	#RANDS	NAME
100011	01	10	lw

0000

# OPCODE	# TYPE	# RANDS	NAME
001001	10	10	jalr

0001

# OPCODE	# TYPE	# RANDS	NAME
100001	11	11	addu
100001	11	11	addu

0010

# OPCODE	# TYPE	# RANDS	NAME
101011	01	10	sw

0011

# OPCODE	# TYPE	# RANDS	NAME
100001	11	11	addu
100001	11	11	addu

Figura 5.19 – Exemplo de codificação usando o método de *VLC*.

No capítulo seguinte mostraremos os dois conjuntos de experimentos que foram feitos para os processadores *MIPS R2000* e *TMS320C25*, para determinar a melhor técnica de codificação para padrões de programas.

5.4 Compressão de Código para o Processador *MIPS R2000*

Para a geração do código comprimido tendo como processador alvo o *MIPS R2000*, procedemos da seguinte maneira. Após as codificações dos padrões de árvores e de operandos, eles são colocados juntos, seqüencialmente para formar uma lista de pares de palavras de código: $Tp_1,Op_1|\ Tp_2,Op_2|...\ Tp_n,Op_n$. As barras são usadas para marcar o fim de um par de palavras de código e o início de outro. A figura 5.17 mostra a formação de uma palavra de código que, além do par (Tp_i,Op_i), tem ainda o *bit* diferenciador *b* para mostrar qual método foi utilizado para codificar Tp_i ou Op_i. As palavras de códigos podem ser quebradas a fim de formar palavras de 32 *bits*. Os *bits* restantes da palavra de código quebrada são colocados na próxima palavra de 32 *bits*. O tamanho de uma palavra código é limitado em 16 *bits*. Portanto, no pior dos casos, uma palavra conterá, pelo menos, um padrão. A possibilidade de dividir palavras de código e usar códigos de tamanho variável torna o projeto da máquina de descompressão um pouco crítico. Por outro lado, os trabalhos que apresentam razões de compressão baixas (< 50%) somente alcançam estas taxas porque utilizam esta abordagem. O motivo disto, também discutido por Lefurgy et al [25], está relacionado com o fato de que muitos padrões comuns são originados de uma simples palavra de instrução. Entretanto, quando restringimos as palavras de código em uma simples palavra de memória, limitamos consideravelmente a razão de compressão.

5.5 Compressão de Código para o Processador *TMS320C25*

Para este processador, de forma semelhante ao item anterior, o programa compactado é formado por pares (Tp_i,Op_i), em que Tp_i é uma palavra de código para padrões de árvores e Op_i para padrões de operandos. As palavras de código podem ser quebradas com

o objetivo de formar palavras de 16 *bits*, sendo o tamanho de uma palavra de código limitada a 8 *bits*. A codificação de um padrão de árvores ou de operandos é semelhante ao do processador *MIPS*, e pode também ser vista na figura 5.17

CAPÍTULO 6

Arquitetura do Descompressor

Neste capítulo, propomos a máquina de descompressão tanto para o processador MIPS R2000 como para o TMS320C25 com o objetivo de avaliar o impacto destas máquinas na razão final de compressão dos programas.

6.1 A Máquina de Descompressão para Código *MIPS R2000*

A despeito de sua alta taxa de compressão, a codificação utilizando comprimento variável implica em baixa eficiência de decodificação. Entretanto, acreditamos que, para conseguirmos altas taxas de compressão, temos que explorar a natureza não balanceada dos padrões dos programas. Propomos aqui uma máquina de decodificação para nosso método de compressão que pode ser vista na figura 6.1. O objetivo é trocar parte da área de silício ganha pela compressão por um projeto melhorado de uma máquina de descompressão. A seguir, discutiremos o funcionamento da máquina proposta.

6.1.1 Geração dos Padrões de Árvores

O *Dicionário de Padrões de Árvores* (*Tree Patterns Dictionary - TPD*) armazena, para cada palavra de código, os *opcodes* dos padrões de árvores. A palavra *Tp* é então decodificada pelo *Gerador de Padrão de Árvore* (*TGEN*) em um endereço *tpaddr* correspondente a uma entrada do dicionário *TPD*. Cada entrada do *TDP* é composta de três campos : *OPCODE*, *ITYPE* e *END*. O campo *OPCODE* tem o código de operação da instrução no padrão de árvore. O campo *ITYPE* codifica o tipo, ou seja, o formato da instrução.

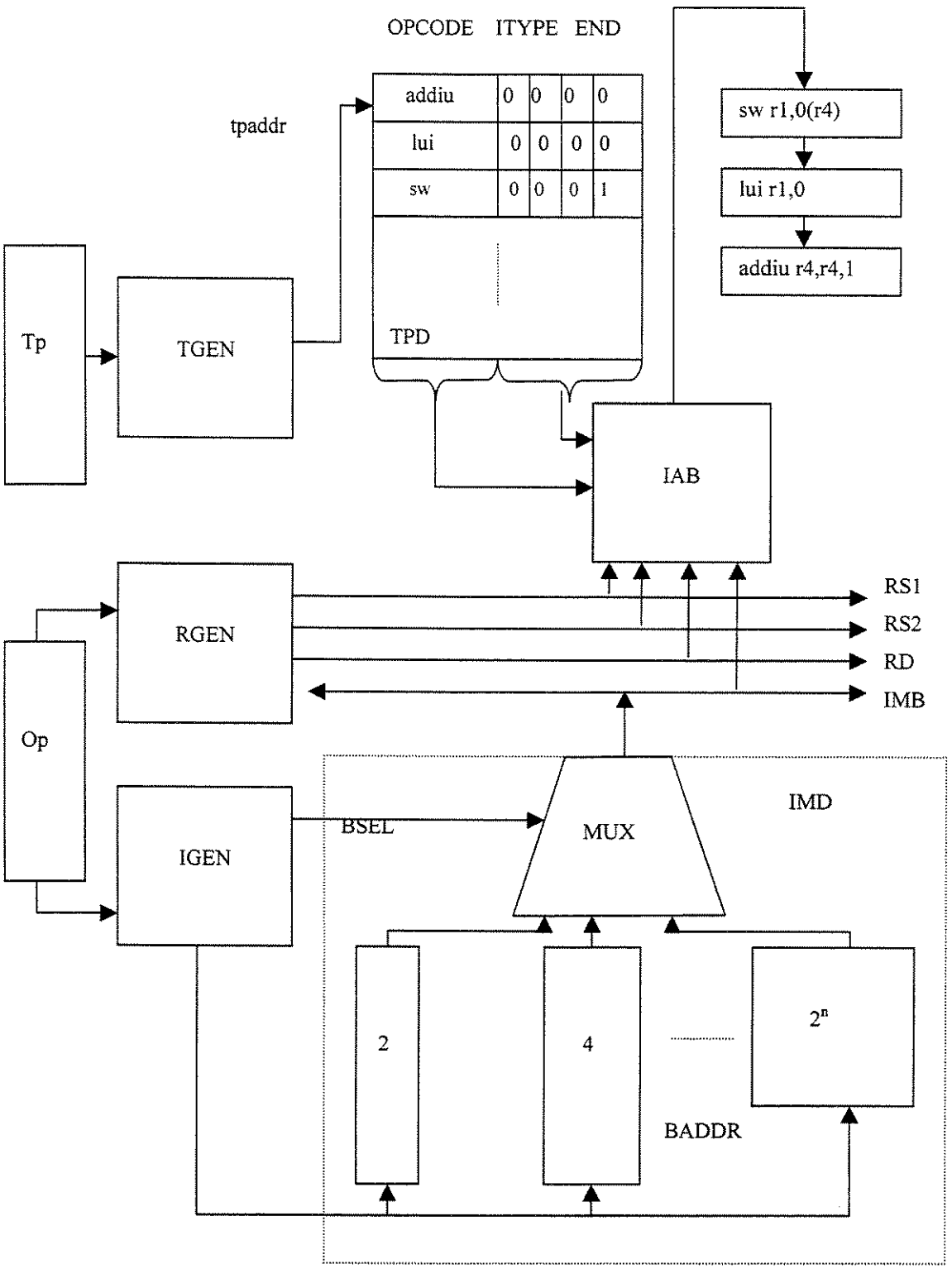


Figura 6.1 Máquina de descompressão para códigos *MIPS*

A informação armazenada em *ITYPE* é usada pelo módulo *IAB (Instruction Assembler Buffer)*, para saber como montar uma instrução descomprimida. O *IAB* põe o conjunto de *bits* do *OPCODE* junto com os *bits* dos registradores (*RS1, RS2, RD*) e dos imediatos (*IMB*) para formar uma determinada instrução. As entradas do *TPD* são feitas seqüencialmente a partir do *tpaddr*. O campo *END* é usado para marcar a última instrução na árvore padrão. O tamanho adicional, calculado em número de bits, devido ao *TPD*, com relação ao programa não comprimido, é mostrado na tabela 6.1. Em média, o *TPD* representa apenas 1,7% do tamanho do programa original.

PROGRAMA	TPD % (a)	IMD % (b)	RGEN % (c)	Compressão dos Imediatos % (d)
go	1,2	1,1	5,5	17,5
li	0,7	2,7	2,4	32,9
compress	7,3	5,1	12,6	32,9
perl	1,0	2,1	3,0	28,4
gcc	0,7	1,2	2,4	21,6
vortex	0,5	1,6	1,7	30,8
ijpeg	2,7	1,8	6,1	22,5

Tabela 6.1: Porcentagem com relação ao programa não comprimido (a) do tamanho, em número, de *bits* do *TPD*; (b) do tamanho em número de *bits* do *IMD*; (c) do tamanho, em número de *bits*, do *RGEN*; (d) Razão de compressão dos imediatos.

6.1.2 Geração de Registradores

O *Gerador de Registradores* (*RGEN*) é uma máquina de estados finitos que decodifica o campo *OP* da palavra comprimida numa seqüência de registradores necessários às instruções do padrão de árvores. A saída de *RGEN* é formada por três barramentos, um para cada registrador : *RS1, RS2* e *RD*. Os dois primeiros barramentos (*RS1* e *RS2*)

correspondem aos campos associados aos registradores fonte da instrução, enquanto que o barramento *RD* corresponde ao registrador destino. O número de estados de *RGEN* é limitado pelo número de instruções do maior padrão de árvore do programa. Como a maioria das árvores de expressões são pequenas, avaliamos que *RGEN* não terá muitos estados. Para os casos em que o tamanho do padrão de árvore seja menor que o maior padrão de árvore do programa, os estados das variáveis não utilizadas ficam em *don't care*. Os padrões de operandos que têm imediatos são usados para simplificar a lógica do *RGEN*.

Quando o padrão de operandos codificado por *OP* contiver um imediato, o barramento do registrador associado ao imediato não será usado, ficando no estado *don't care*. Por exemplo, o padrão operando *[r2,r5,4]* resultaria em *RD = 00010, RS1 = 00101* e *RS2 = XXXXX* (*don't care*). Podemos observar que os padrões que compartilham registradores podem também ser usados para minimizar a lógica do *RGEN*. Por exemplo, o padrão *[r2,r0,r5]* resulta em um valor similar para os barramentos *RD* e *RS2*, se comparamos com o padrão *[r2,r3,r5]*. Esta maneira de codificação de padrões de operandos naturalmente se traduz em um problema de minimização para a lógica do *RGEN*, podendo o tamanho final do circuito se beneficiar das ferramentas de minimização disponíveis atualmente [35].

Uma outra abordagem é implementar um dicionário para as seqüências de registradores, tal como feito no item anterior. Com isto, a média de contribuição do módulo *RGEN* será de 4,8%, como pode ser visto na tabela 6.1.

6.1.3 Geração de Imediatos

O *Dicionário de Imediatos* (*Immediate Dictionary - IMD*), armazena os imediatos usados pelo programa. Para cada imediato do programa, é necessário apenas um acesso simples ao dicionário, não importando a maneira que a instrução o usa ou quantas vezes ele é usado. Por exemplo, a constante 4 tem diferentes significados para as instruções *bgez r5,4*; *lw r6, 4(r29)* e *srl r5,r3,4*. O significado do imediato 4, como *offset* de endereço, *offset* de pilha ou quantidade de deslocamento não é relevante para a maneira como ele será armazenado em *IMD*. Nós usamos a variação dos tamanhos dos imediatos para minimizar o número de *bits* necessários para armazená-los no *IMD*. A avaliação do número de *bits* necessários para codificar os imediatos revela que, em média, mais de 70% dos imediatos

de um programa podem ser codificados com menos que 16 *bits*, como apresentado também em Haskell et al [50]. Os imediatos são agrupados em bancos de memória de acordo com o número de *bits* que eles necessitam para ser representados. O endereço do banco de memória *BADDR* e o sinal de seleção do banco *BSEL* são gerados pelo módulo *IGEN* através da palavra código *OP*. O *IGEN* é uma máquina de estados finitos que trabalha paralelamente com *RGEN* e *TGEN*. O número ótimo de bancos de memória depende de como os imediatos são distribuídos no programa e do aumento de área, devido à introdução de novos bancos. Este enfoque melhora sensivelmente a razão de compressão para imediatos. Esta razão é calculada dividindo o número de *bits* armazenados na *IMD* pelo número total de *bits* dos campos dos imediatos das instruções do programa não comprimido. Para os programas analisados, o resultado foi, em média, 26.7%. O tamanho adicional do *IMD* com relação ao programa não comprimido corresponde, em média, a apenas 2,2% do código original, como mostrado na tabela 6.1.

6.1.4 Endereço de Desvios

Aproveitamos aqui as idéias desenvolvidas por Lefurgy et al [25]. O endereço alvo de desvio é dividido no endereço *addr* com 21 *bits* e um deslocamento *offset* com 5 *bits*. Diferente da abordagem feita por Lefurgy [25], aqui as instruções de desvios são comprimidas. Durante a descompressão, os valores *addr* e *offset* são lidos do dicionário *IMD* para montar a instrução. Assumimos que a unidade de controle do processador trata deslocamentos com alinhamento por *bit* dentro da palavra de código. Durante a operação de busca, a palavra armazenada no endereço *addr* é lida da memória e a instrução comprimida é extraída da palavra codificada, a partir da posição *offset*. Para os programas analisados, apenas a um pequena porcentagem de endereços alvo necessita mais que 21 *bits*. Para estes casos, usamos uma tabela de saltos que armazena os novos endereços alvos, a mesma abordagem utilizada por Lefurgy et al [25]. Uma outra alternativa é utilizar a técnica de mapeamento de endereços *LAT/CLB* para poder calcular os endereços de desvios, como proposto por Wolfe [20].

6.2 A Máquina de Descompressão para Código *TMS320C25*

A figura 6.2 mostra a máquina de descompressão proposta para códigos compactados, tendo como alvo o processador *TMS320C25*. Este sistema consiste num conjunto de máquinas de estados e de dicionários que geram os vários campos da instrução descomprimida. As máquinas de estado *OPGEN, ARGEN, INGEN* e *IMGEN* trabalham em paralelo para gerar os *opcodes* e operandos das instruções. Os vários campos gerados vão para o módulo *IAB* que monta as instruções e as envia para o processador.

6.2.1 Geração dos Padrões de Árvores

A responsabilidade da extração dos padrões de árvores é da máquina de estado *OPGEN* e de um dicionário de padrões de árvores *OPD* (*Opcode Dictionary*), como mostra a figura 5.19. Em *OPD* estão armazenados os padrões de árvores e os respectivos *opcodes* de suas instruções. A máquina de estado *OPGEN* extrai o código *Tp* da memória e decodifica-o, gerando os sinais *OPSEL* e *OPADR*. Estes sinais são responsáveis pela seleção dos bancos e posterior geração dos *opcodes* referentes às instruções do padrão de árvore. Os *opcodes* são agrupados de acordo com seu tamanho e armazenados nos respectivos bancos de memória.

6.2.2 Geração dos Operandos

Para a geração dos operandos, o sistema de descompressão conta com 3 máquinas de estados *ARGEN, INGEN* e *IMGEN* e os respectivos dicionários *ARGEND, INGEND* e *IMD*. Depois da extração do padrão de operando *Op*, as máquinas de estado *ARGEN, INGEN* e *IMGEN* decodificam os operandos das instruções.

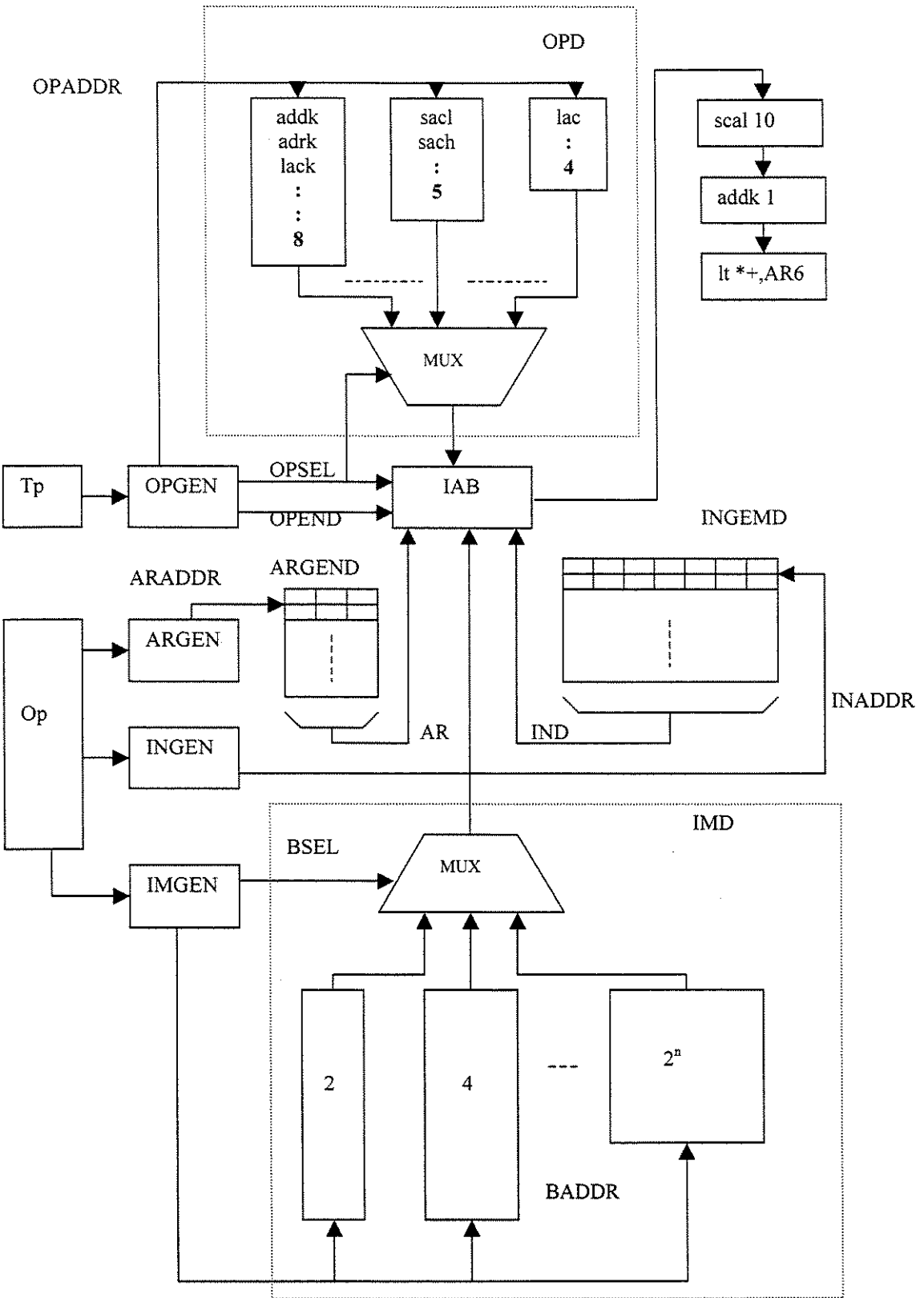


Figura 6.2 Máquina de descompressão para código TMS320C25

A máquina *ARGEN* gera o sinal *ARADDR* que seleciona o campo das instruções relativo ao registrador *AR*, e *INGEN* gera o sinal *INADDR* que é responsável por extrair de *INGEND* os 7 *bits* que codificam os campos <*ind,next*> das instruções. Os imediatos são extraídos do dicionário *IMD* através dos sinais *BSEL* e *BADDR*. Estes sinais são gerados pela máquina de estado *IMGEN*, a fim de selecionar quais dos bancos de imediatos serão acessados. Um *bit* extra é utilizado para habilitar automaticamente a extensão do sinal, quando os imediatos são recuperados do *IMD*.

CAPÍTULO 7

Resultados Experimentais

O objetivo dos experimentos, tanto para o processador *MIPS R2000* como para o *TMS320C25*, foi achar o método de compressão que resultasse a melhor razão de compressão, levando-se em conta a complexidade da descompressão. A seguir descreveremos os resultados que encontramos para ambos os processadores.

7.1 Experimentos Utilizando o Processador *MIPS R2000*

Para a análise dos resultados experimentais, fizemos dois conjuntos de experimentos com o objetivo de determinar a melhor maneira de codificar os padrões de árvores e de operandos. O primeiro teve o objetivo de determinar a razão de compressão usando codificação com os métodos de comprimento fixo e de *Huffman*. Este experimento serviu também para determinar o ponto de separação entre estes dois métodos quando utilizamos a abordagem *Bounded Huffman*. O segundo experimento teve os mesmos objetivos do primeiro só que utilizando os métodos de comprimento fixo e o *VLC*. Nos itens que se seguem, mostraremos os resultados obtidos.

7.1.1 Experimentos Envolvendo Comprimento Fixo e *Huffman*

Neste experimento, ordenamos os padrões de árvores e de operandos separadamente em duas listas, de acordo com suas contribuições para o programa, em números de *bits*. Para o primeiro conjunto de experimentos, os padrões foram codificados utilizando *Huffman* e comprimento fixo. As figura 7.1 e 7.2 mostram os resultados destes experimentos.

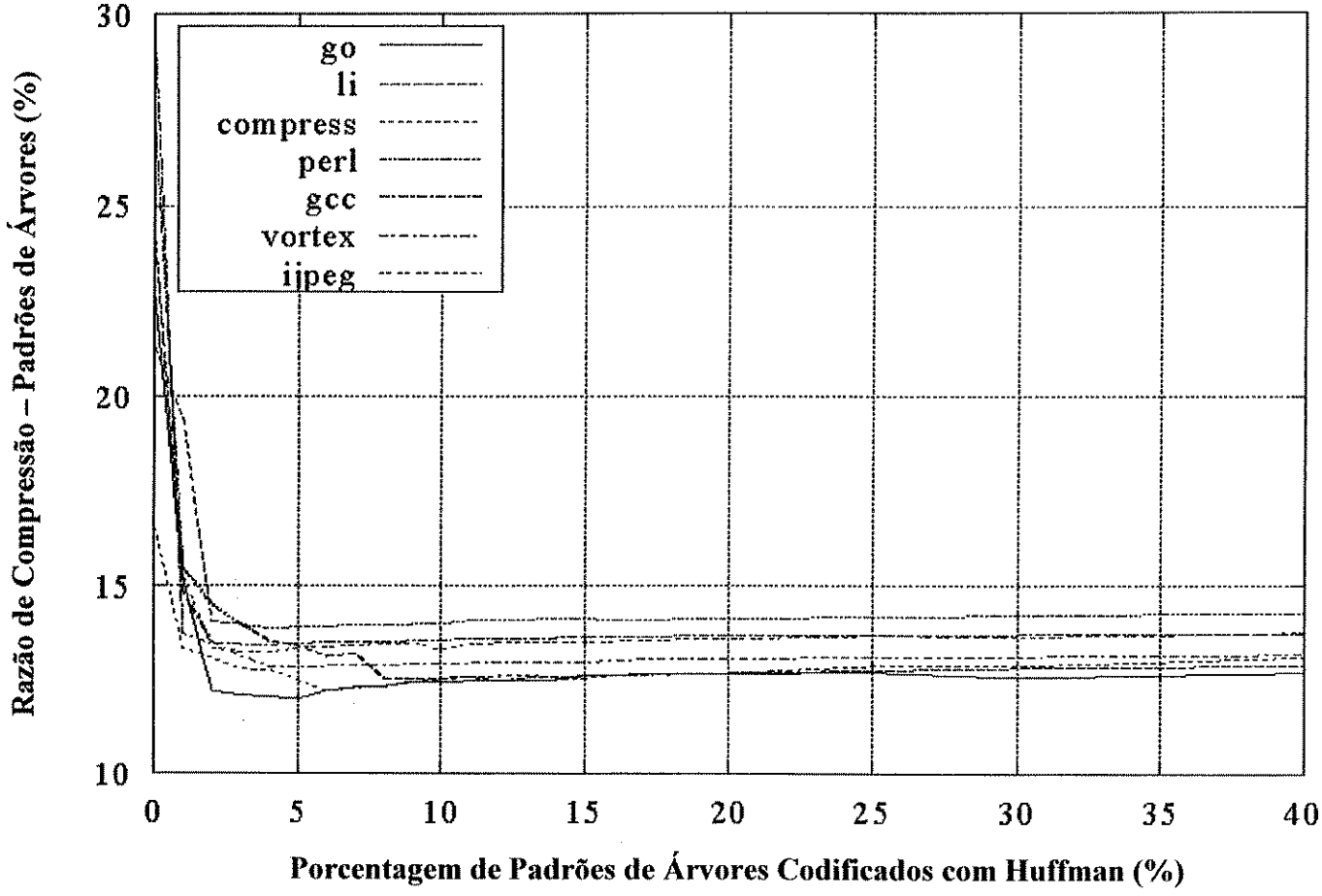


Figura 7.1 Razão de compressão devido à codificação de padrões de árvores usando Bounded-Huffman

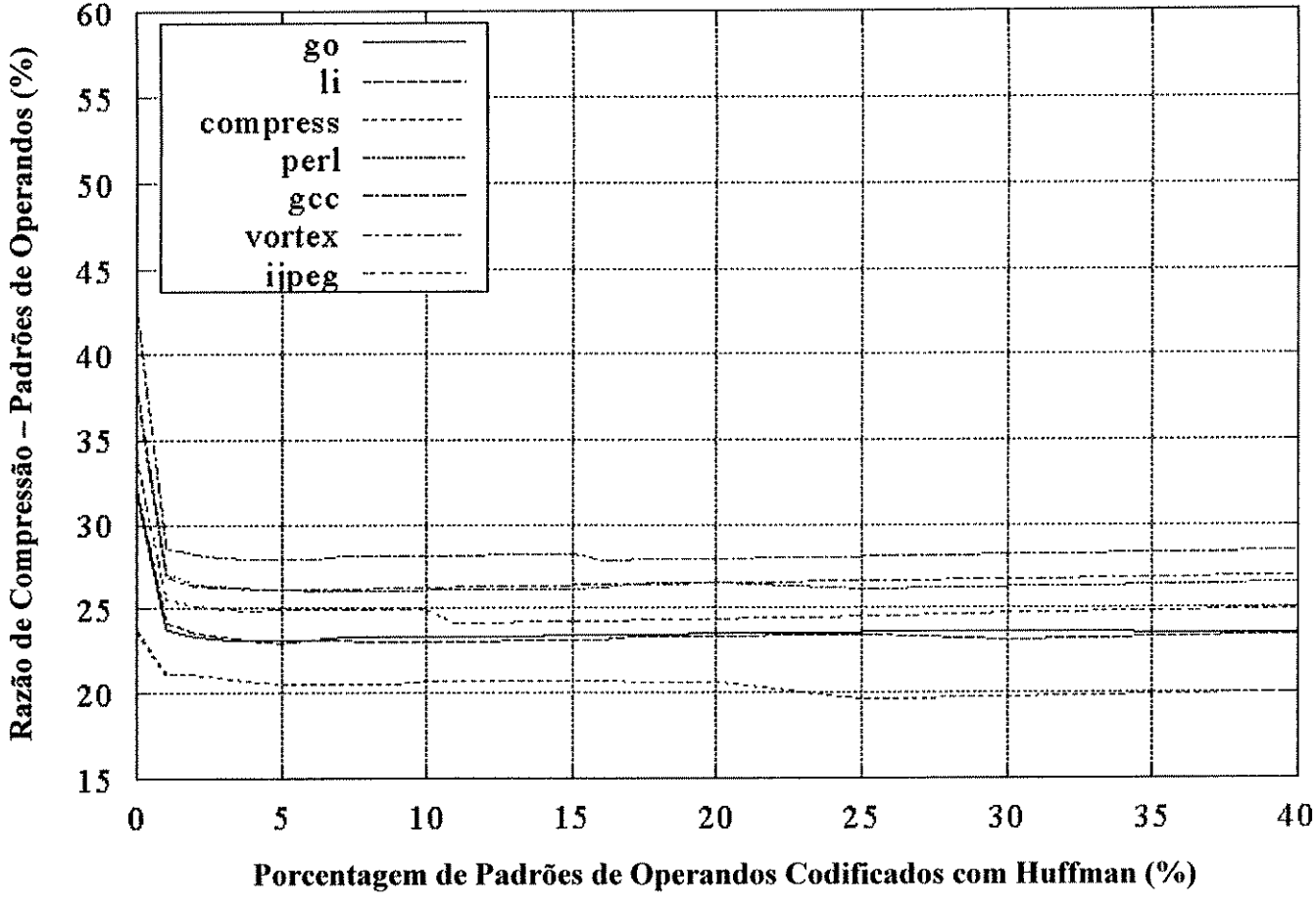


Figura 7.2 Razão de compressão devido à codificação de padrões de operandos, usando Bounded- Huffman

Nas figuras, os eixos horizontais representam a porcentagem de padrões codificados pelo método *Huffman*. Para o ponto 0% temos todos os padrões codificados com o método de comprimento fixo. Os pontos entre 0% e 40% utilizam os dois métodos combinados, mais o *bit* diferenciador. As figuras nos mostram também que a razão de compressão tende a saturar, pois, quanto maior o número de padrões codificados com *Huffman*, menor será sua contribuição para a razão de compressão. Isto é reflexo da distribuição exponencial dos padrões dos programas discutidos no capítulo 5. Os padrões que têm pequena contribuição não mudam a entropia do programa. Uma pequena entropia resulta em pequenas redundâncias e faz com que a codificação utilizando *Huffman* se aproxime da codificação com comprimento fixo, fato também observado por Franz [48]. A mudança da codificação *Huffman* para a de comprimento fixo é feita a partir do ponto onde a distribuição de padrões torna-se uniforme; isto minimiza o impacto da não utilização do método de *Huffman*. Considere, por exemplo, o programa *go* na figura 6.1. A contribuição para a razão de compressão, quando 5% dos padrões de árvores são codificados usando *Bounded Huffman*, é 12%, ou seja, apenas 1,4% maior do que se todos os padrões fossem totalmente codificados utilizando *Huffman*. Um comportamento parecido pode também ser encontrado nos padrões de operandos, como podemos ver na figura 6.2. A contribuição para a razão de compressão do programa *go*, devido à codificação de 4% dos padrões de operandos, usando *Bounded Huffman* é de 23,1%, muito perto da razão de compressão achada usando apenas *Huffman*, que é de 22,6%.

7.1.2 Experimentos Envolvendo Comprimento Fixo e *VLC*

Aqui também ordenamos, de forma separada, os padrões de árvores e de operandos em duas listas, de acordo com suas contribuições em números de bits, só que utilizando a codificação *VLC* e de comprimento fixo. As figuras 6.3 e 6.4 mostram a razão de compressão usando codificação *VLC*. O ponto de mudança de *VLC* para comprimento fixo foi em torno de 2,5% para padrões de árvores e de 0,2% para de operandos. Os valores acima são menores do que os valores encontrados quando utilizamos *Huffman* porque as palavras de código *VLC* são maiores que as de *Huffman*.

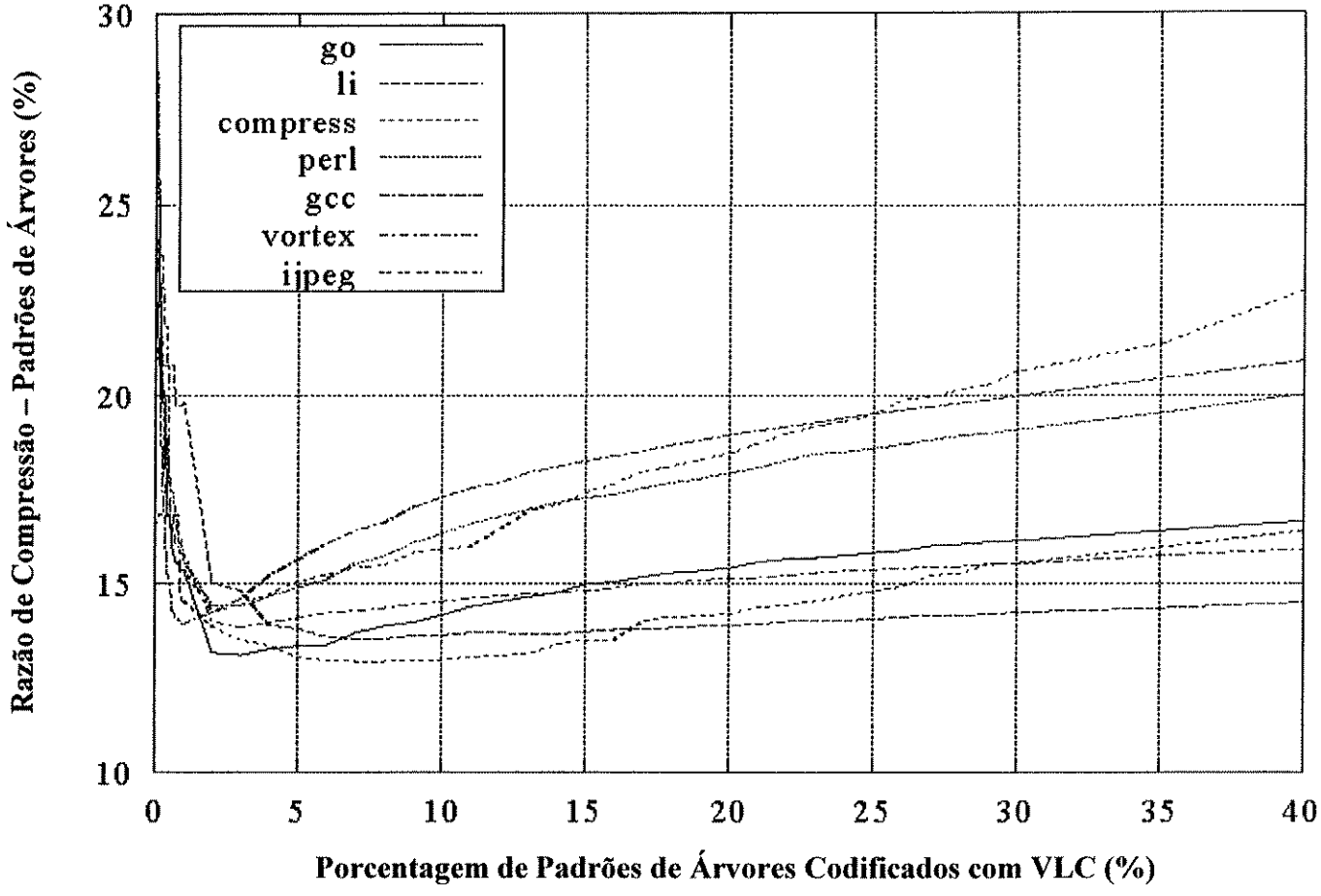


Figura 7.3 Razão de compressão devido à codificação de padrões de árvores usando VLC

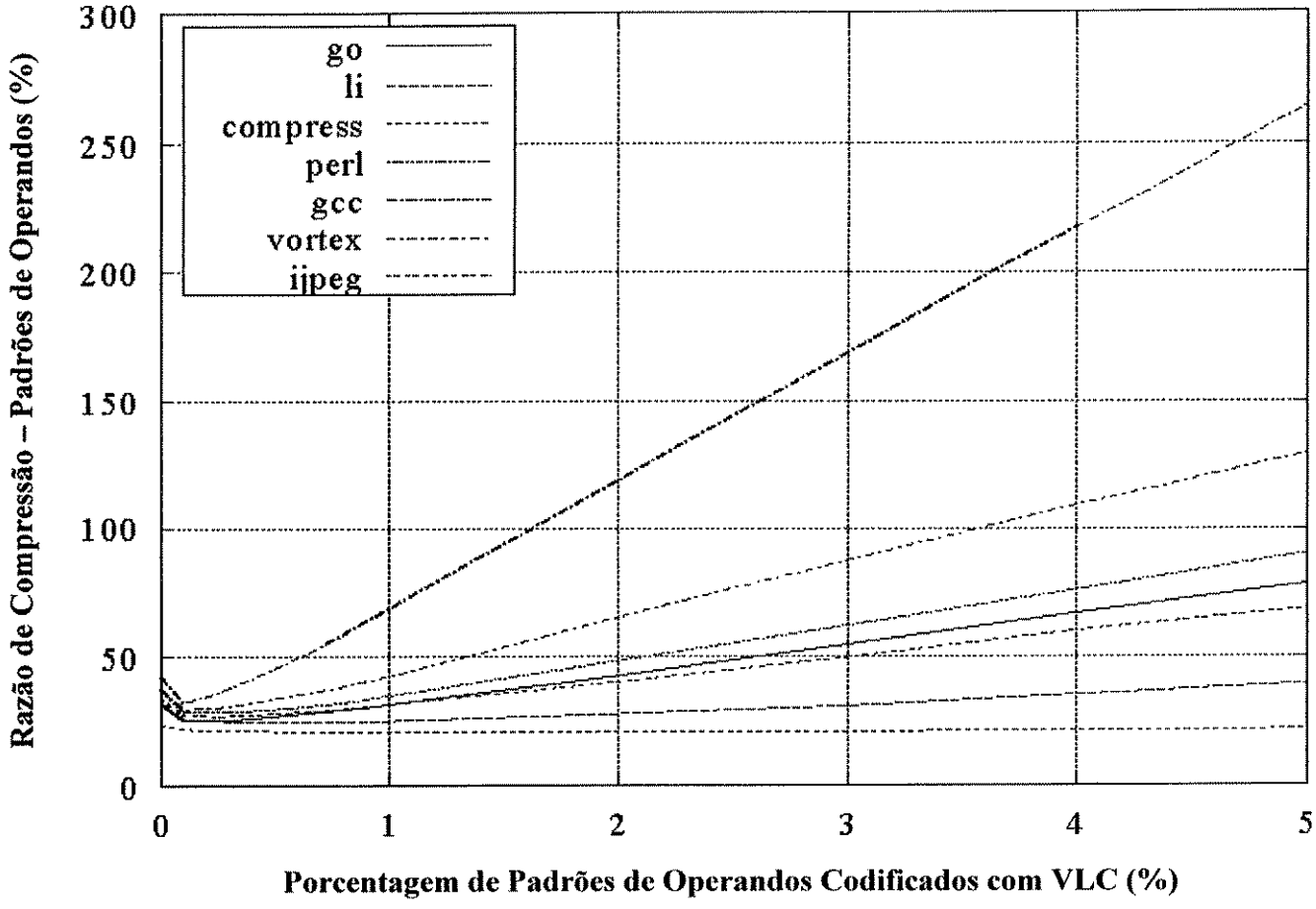


Figura 7.4 Razão de compressão devido à codificação de padrões de operandos usando VLC

É por isto que a razão de compressão ultrapassa 100%, à medida que mais padrões são codificados com VLC, como podemos ver nas figuras 6.3 e 6.4. A média da contribuição para a razão de compressão dos padrões de árvores foi de 13,7%, enquanto que para os padrões de operandos foi de 26,9%.

7.1.3 Razão de Compressão

Para achar a razão de compressão ideal para os programas, construímos uma tabela com o espaço de soluções quando os diversos métodos são combinados. Isto auxilia na escolha do método de codificação ideal. Esta tabela foi montada a partir da média das menores razões de compressão obtidas das figuras 7.1, 7.2, 7.3 e 7.4. A tabela 7.1 é o resultado deste processo e nos mostra a razão de compressão média para todos os programas, combinando os vários métodos para padrões de árvores e operandos.

		PADRÕES DE OPERANDOS			
PADRÕES DE ÁRVORES	METODO DE CODIFICAÇÃO	COMP. FIXO	HUFFMAN	BOUNDED-HUFFMAN	VLC
	COMPRIMENTO FIXO	57,7	46,2	48,1	50,5
	HUFFMAN	45,4	35,0	35,8	38,2
	BOUNDED-HUFFMAN	46,0	36,6	37,4	39,8
	VLC	47,9	37,5	38,3	40,7

Tabela 7.1 Razão de compressão média após a combinação de métodos de codificação de padrões de árvores e operandos.

A pior razão de compressão foi de 57%, obtida quando os padrões de árvores e de operandos foram codificados utilizando comprimento fixo. Como esperado, a melhor razão

foi resultado da codificação utilizando *Huffman* para ambos os padrões (35%). O problema da codificação *Huffman* é que as palavras código não têm a informação do seu tamanho. Por outro lado, usando *VLC* para codificar ambos padrões, resulta numa razão de compressão de 41%. Esta razão é apenas 6% maior que a codificação com *Huffman*. Este é o preço a ser pago para diminuir a latência da lógica de detecção do tamanho da palavra de código na máquina de descompressão. A razão de descompressão, mostrada na tabela 7.1, não deixa clara a eficiência de compressão usando fatoração de operandos, para isto, a área de silício da máquina de descompressão deve ser considerada. O tamanho dos dicionários de padrões de árvores e de imediatos pode ser estimado, baseado no número de *bits* por eles utilizados. A figura 7.5 mostra a razão de compressão final para os métodos de *Huffman*, com as estimativas dos tamanhos dos dicionários. Podemos ver que a razão de compressão foi em média 43% para o método *Huffman* e 48% para o *VLC*.

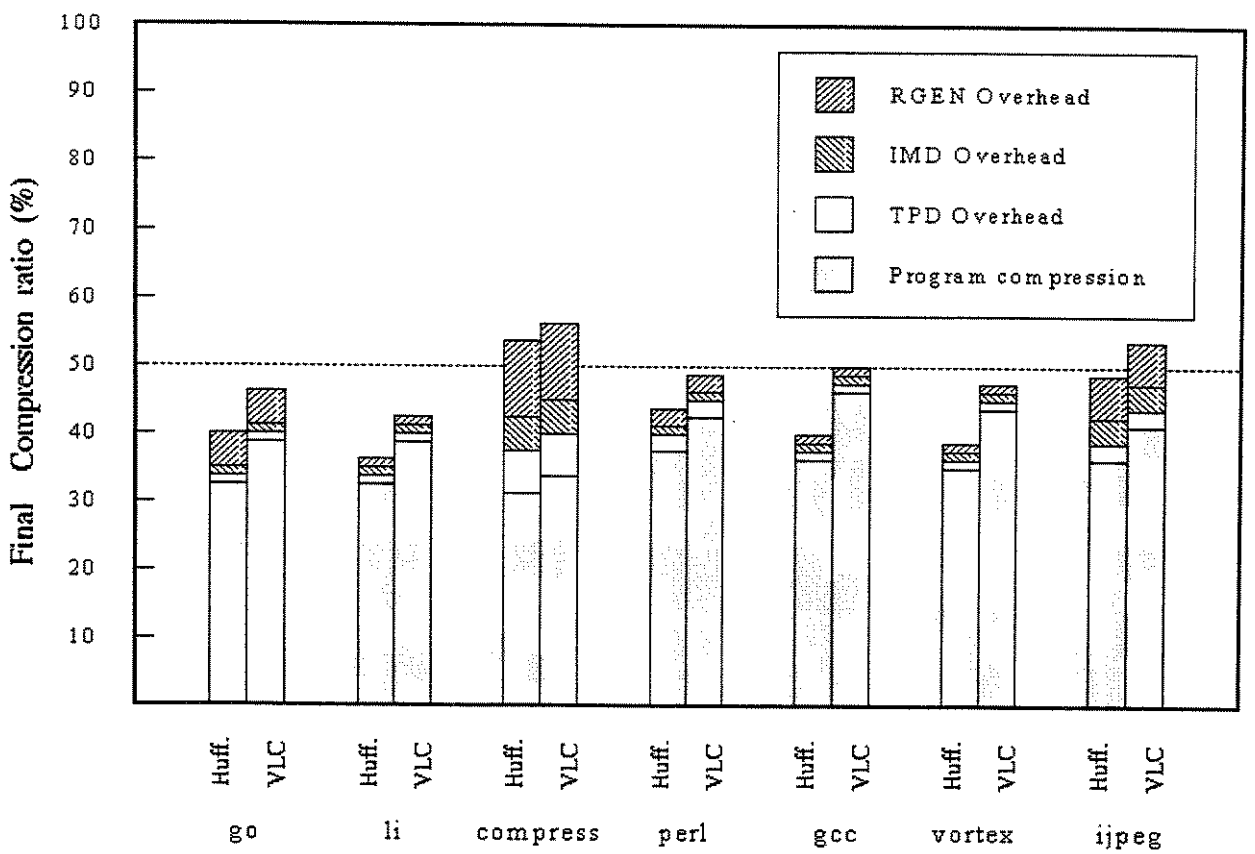


Figura 7.5 – Razão de compressão final usando *Huffman* e *VLC*

Como podemos ver nesta figura, a influência dos dicionários na razão de compressão final é pequena, em média 2%. Uma estimativa mais precisa do tamanho e do desempenho dos módulos *TGEN*, *RGEN* e *IGEN* pode ser feita apenas após sua implementação. O *TGEN* é o decodificador de um pequeno dicionário (*TPD*), portanto ocupa uma pequena área de silício. Por outro lado, *RGEN* e *IGEN* codificam um número grande de padrões de operandos e deve contribuir mais para o tamanho da máquina.

7.2 Experimentos Utilizando o Processador *TMS320C25*

Como no item 7.1, montamos também dois conjuntos de testes a fim de avaliar qual a melhor solução para a compressão de códigos de programas para o processador *TMS320C25*. O primeiro conjunto avaliou o método utilizando comprimento fixo e o método de *Huffman*, assim como o melhor ponto de separação entre eles, quando utilizamos estes dois métodos juntos. O segundo avaliou os métodos *VLC* e comprimento fixo e também a melhor separação entre eles quando ambos são usados.

7.2.1 Experimentos Envolvendo Comprimento Fixo e *Huffman*

Como nos experimentos anteriores, ordenamos separadamente os padrões de árvores em duas listas, de acordo com sua contribuição, em número de *bits*. Os padrões foram codificados utilizando os métodos de comprimento fixo e *Huffman*. As figuras 7.6 e 7.7 mostram a razão de compressão para este caso. Como visto no Capítulo 5, a distribuição de padrões é algo próximo de uma curva exponencial, portanto, quanto mais os padrões são codificados em *Huffman*, menores serão as suas contribuições para a razão de compressão. A razão disto é a saturação das curvas das figura 7.6 e 7.7. O menor ponto onde devemos mudar de comprimento fixo para *Huffman* corresponde ao mínimo na curva mostrada pelas figuras, seja para padrões de árvore, seja para padrões de operandos. Por exemplo, para o programa *gzip*, na figura 7.6, a contribuição para a razão de compressão, quando 50% dos padrões de árvores são codificados utilizando *Bounded-Huffman*, é de 30%. Para os padrões de operandos, na figura 7.7, temos um comportamento parecido, pois a contribuição para a razão de compressão do mesmo programa, devido à codificação de 45% dos padrões, é de 37%.

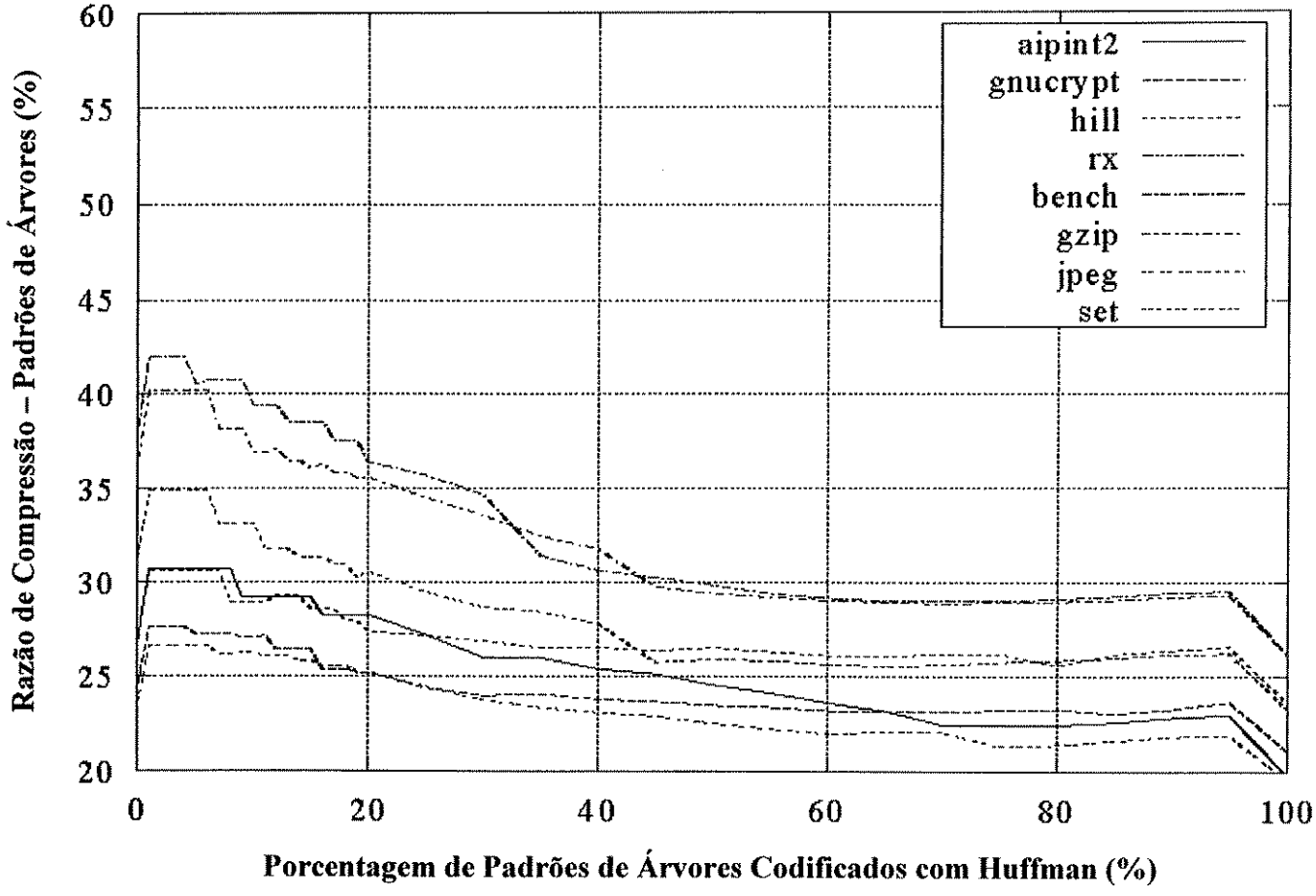


Figura 7.6 Razão de compressão devido à codificação de padrões de árvores usando Huffman

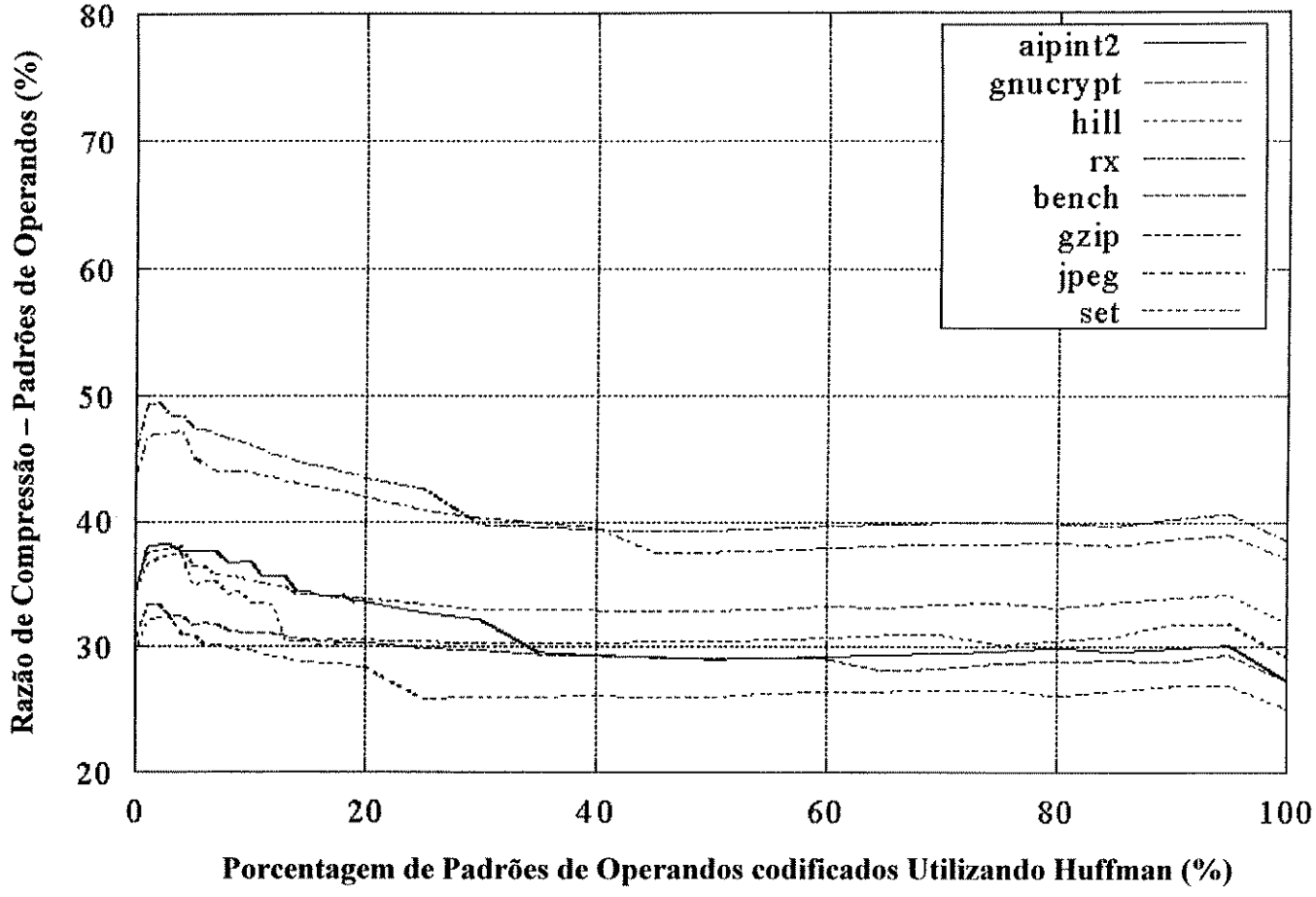


Figura 7.7 Razão de Compressão devido à codificação de padrões de operandos usando Huffman

7.2.2 Experimentos Envolvendo Comprimento Fixo e *VLC*

O processo aqui foi semelhante ao descrito no item 7.2.1, só que ao invés de utilizarmos *Huffman*, usamos *VLC*. As figuras 7.8 e 7.9 mostram a razão de compressão obtida. O ponto de mudança de *VLC* para comprimento fixo foi em torno de 27% para os padrões de árvore e 23% para os de operandos. A média da contribuição para a razão de compressão dos padrões de árvores foi de aproximadamente 23% e para os de operandos foi de aproximadamente 33%.

7.2.3 Razão de Compressão

Na figura 7.10, podemos ver a razão de compressão final para os métodos de *Huffman* e *VLC*, para o conjunto de programas utilizados no teste. Nesta figura já estão considerados os tamanhos dos dicionários da máquina de descompressão, em termos de números de *bits*. A razão de compressão média, utilizando *Huffman*, foi de 60% enquanto que com o *VLC* foi de 67%. A razão de compressão foi determinada pelos pontos mínimos encontrados nos gráficos das figuras 5.6, 5.7, 5.8, e 5.9. O aumento de área devido às máquinas de estados não deve alterar muito este resultado, pois muitos padrões distintos têm partes similares que podem ser traduzidas por lógicas compartilhadas nas máquinas de estados. Por exemplo, a máquina de estado *ARGEN* produzirá o mesmo endereço de entrada no dicionário para os padrões de operandos distintos *AR3, *+* e *AR3, *-, AR4*, pois ambos utilizam o mesmo registrador de endereços *AR3*.

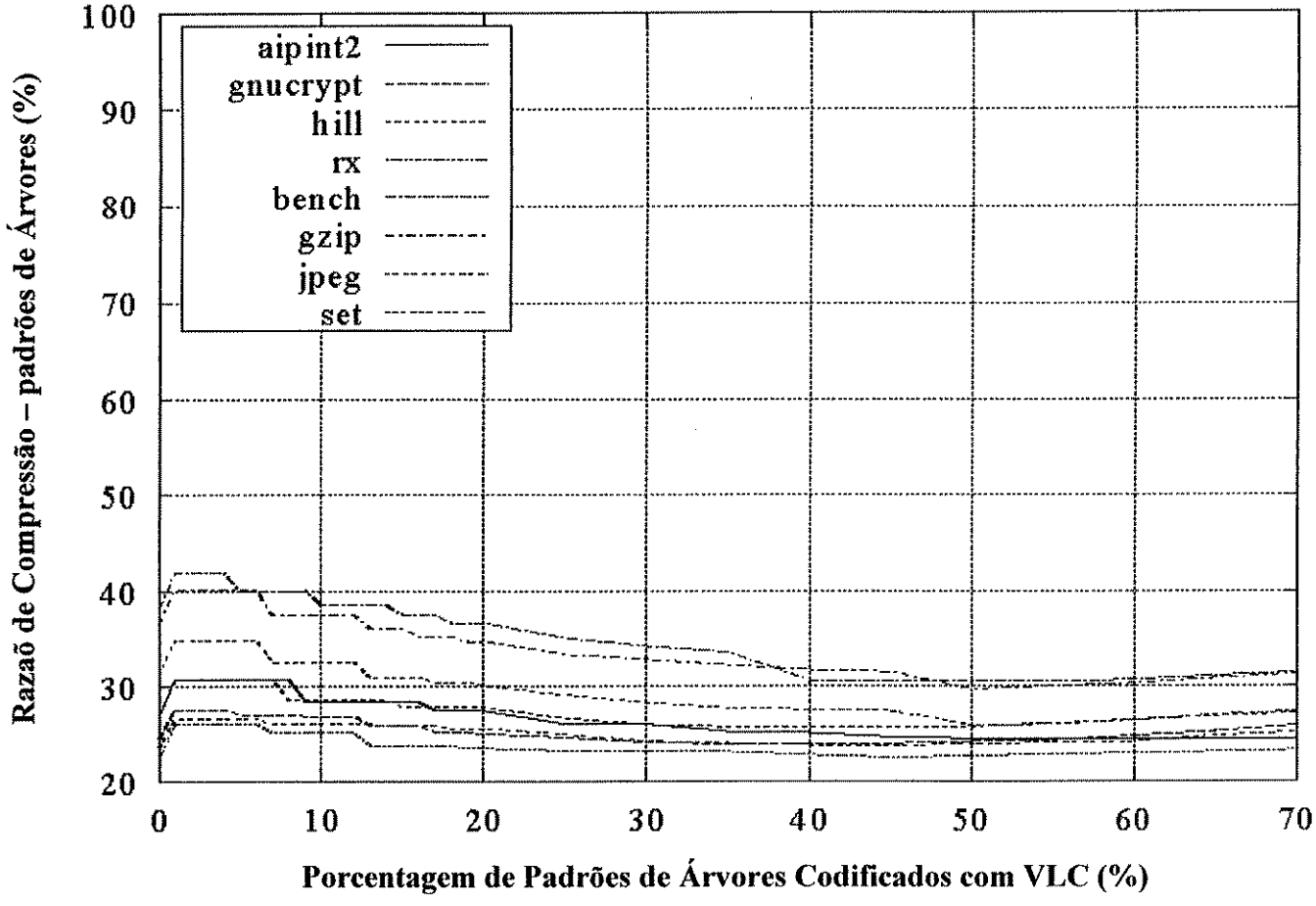


Figura 7.8 Razão de compressão devido à codificação de padrões de árvores usando VLC

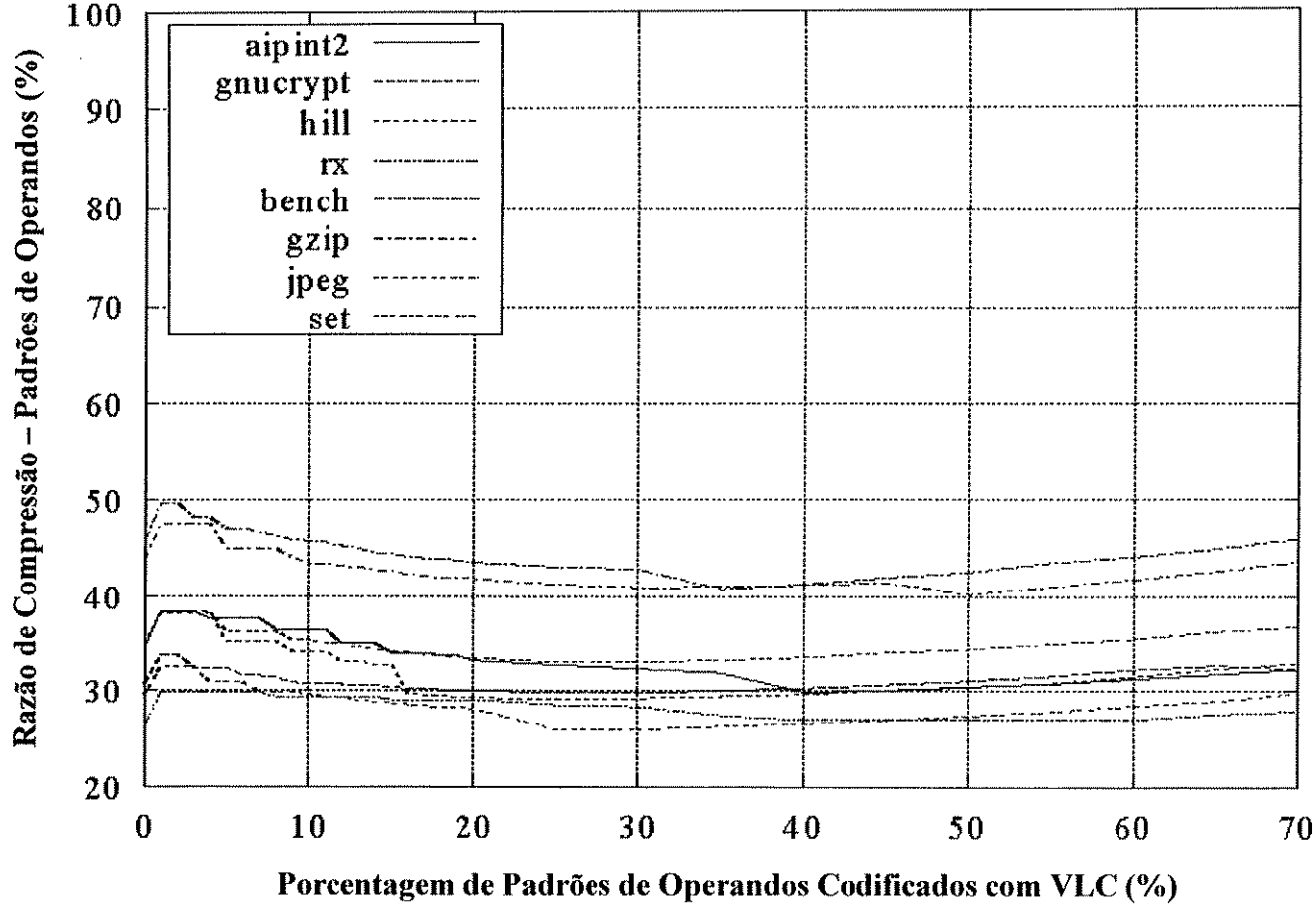


Figura 7.9 Razão de compressão devido à codificação de padrões de árvores de operandos usando VLC

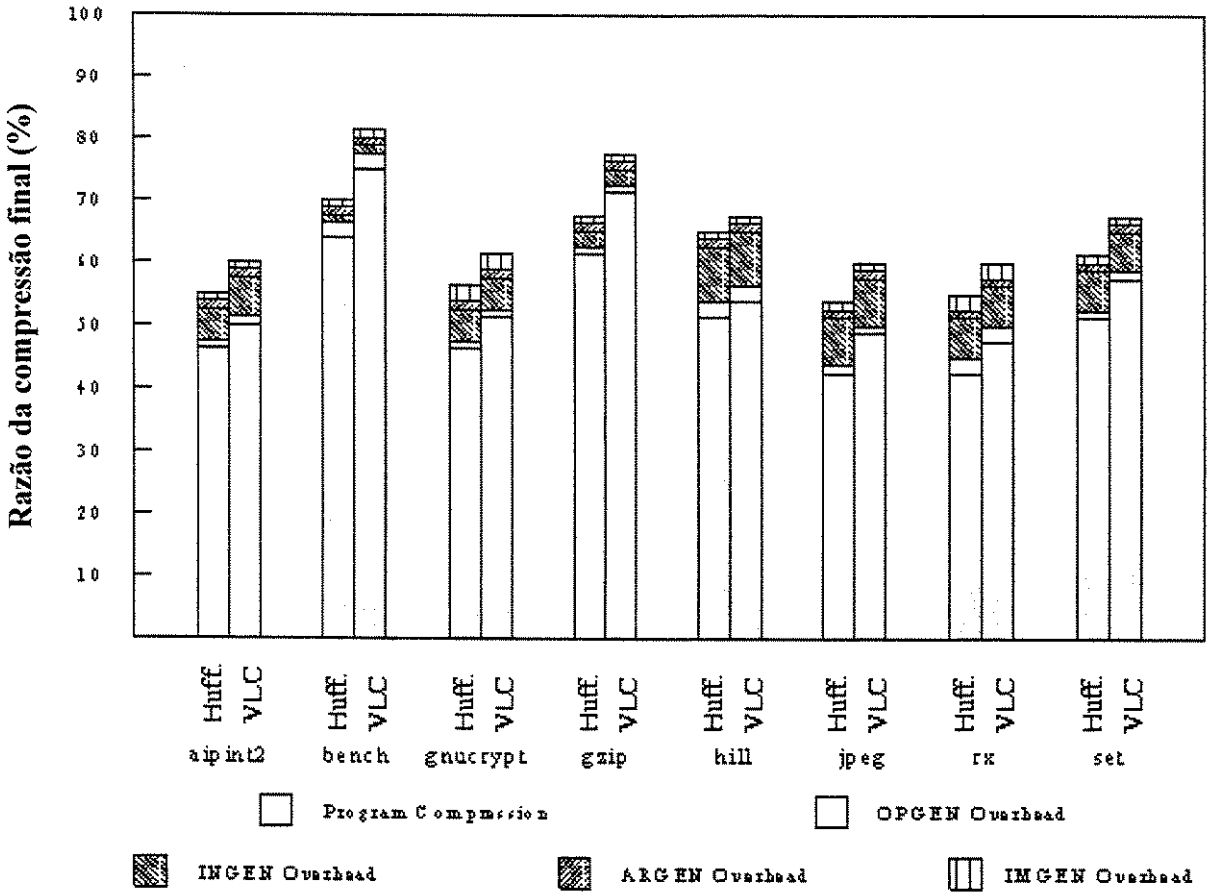


Figura 7.10 Razão da compressão final, considerando o tamanho dos dicionários

CAPÍTULO 8

Conclusões e Trabalhos Futuros

8.1 Conclusões

Nossa contribuição com este trabalho foi propor uma técnica eficiente de compressão para programas chamada fatoração de operandos capaz de ser utilizada em sistemas embutidos. Para os processadores utilizados nos experimentos, mostramos que os padrões de árvores e de operandos têm uma distribuição de freqüência exponencial, isto é, padrões de árvores ou de operandos mais freqüentes cobrem mais árvores de expressões que os não freqüentes. Os experimentos para o processador *MIPS R2000* mostraram uma razão de compressão média de 43% , utilizando a codificação com o método de *Huffman*. Com codificação a *VLC* o resultado foi de 48%. Este é o melhor resultado nesta área, publicado até o momento (circa junho 1999), como podemos verificar na tabela 8.1.

MÉTODO	PROCESSADOR					
	R2000	TMS320C25	PowerPC	ARM	i386	PENTIUM
FATORAÇÃO OPERANDOS	43%	67%	---	---	---	---
LIAO ET AL.	---	82%	---	---	---	---
LEKATSAS & WOLFE	57%	---	---	---	---	75%
WOLF & CHANIN	73%	---	---	---	---	---
LEFURGY ET AL.	---	---	63%	66%	74%	---

Tabela 8.1 – Comparação entre os método de compressão de código existentes.

O motivo disto é que a nossa abordagem explora a correlação existente entre operandos e operadores nas instruções de um programa não tratando este apenas como uma seqüência de *bits*. É bom lembrar que nos dados acima já está computada uma estimativa das áreas relativas aos dicionários da máquina de descompressão, o que a maioria dos trabalhos propostos não fazem.

Com relação ao processador *TMS320C25*, apesar dos números serem aparentemente maiores se comparados com os do processador *MIPS R2000* (60% utilizando *Huffman* e 67% usando *VLC*), eles são ainda 15% menores que aqueles mostrados em outros trabalhos. Estes números também incorporam o aumento de área de silício devido aos dicionários da máquina de descompressão. Além dos algoritmos de compressão, propomos também as respectivas máquinas de descompressão, responsáveis pela montagem de padrões de árvores e de operandos em instruções não comprimidas. Nos trabalhos propostos até hoje, o impacto desta máquina na razão de compressão final não é considerado.

8.2 Trabalhos Futuros

Alguns trabalhos podem ser feitos para tentar melhorar e estender os resultados obtidos.

Como nos padrões de operandos codificamos os registradores temporários duas vezes, uma quando o registrador é usado como destino de uma instrução e outra quando é fonte de outra, poderemos eliminar a segunda referência ao registrador temporário, minimizando assim o tamanho do padrão de operandos, se tivermos um compilador que escalone árvores de expressões em uma ordem pré-definida, o que é muito comum em geração de código [41].

Outro trabalho decorrente desta tese, seria a implementação das máquinas descompressoras, a fim de avaliar a alteração de desempenho dos sistemas.

Outro trabalho que consideramos interessante é analisar a maneira de embutir a máquina de descompressão no próprio processador.

Além disso, podemos também avaliar este método de descompressão para o processador Pentium, um processador CISC de uso geral.

Summary

The increasing use of embedded systems is a clear trend in the telecommunication, multimedia and consumer electronics industry. Because these systems are designed for high-volume market, a cost reduction can have a large impact in the final price of the product. Driven by the need to reduce cost, these systems are implemented by putting together a core processor, an ASIC (*Application Specific Integrated Circuits*) and a program/data memory into a single chip. These systems are known as *SOC - System-On-a-Chip*.

As embedded systems are becoming more complex, the size of embedded programs is growing considerably large. The results are systems in which program memories account for the largest share of total die area, more than the area of the others modules. Thus, minimizing program size has become an important part of the design effort of these systems.

We propose a program code compression technique called operand factorization. The key idea of operand factorization is the separation of program expression trees into sequences of operators and operands (registers and immediates). Using operand factorization we show that tree and operand sequences have exponential frequency distributions. A set of experiments is performed to determine the best encoding technique that explores this feature. The experimental results, using the studied techniques, show an average compression ratio of 43% for SPEC CINT95 programs, running on a MIPS R2000 processor and 60% for a set embedded programs running on the TMS320C25.

A decompression engine is also proposed which assembles operators and operands sequence into uncompressed instructions, using a combination of dictionaries and state machines.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Guido Araújo, Ricardo Pannain, Paulo Centoducatte e Mário Cortes. Code Compression Based on Operand Factorization. *In Proceddings of Micro-31: The 31th Annual International Symposium on Microarchitecture*, pags 194-201, dezembro 1998.

[2] Editorial. The Future of Computing. *The Economist,* pags. 79-81, setembro de 1998.

[3] J.L. Hennessy e D.A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 1996.

[4] K.D. Kissell. MIPS16: High-Density MIPS for the Embedded Market. *Proceedings RTS*, 1997.

[5] Texas Instruments. TMS320C2x Users's Guide, 1993.

[6] C. W. Fraser, D.R. Hanson e T.A. Proebsting. Enginnering a Simple, Efficient Code Generator. *Journal of the ACM*, 22(12):248-262, março 1993.

[7] M . Jonhson. *Superscaler Microprocessor Design.* Prentice-Hall, 1981.

[8] P. M. Kogge. *The Archtiecture of Pipelined Computer*. McGraw-hill, 1981.

[9] Gerry Kane e Joe Heinrich. MIPS RISC Architecture. Prentice Hall, 1992.

[10] www.mips.com /products/Jade1030.pdf.

[11] Guido Araújo. Code Generation Algorithms for Digital Signal Processors. Tese de Doutorado, Universidade de Princeton – EUA, junho de 1997.

[12] Guido Araújo e S. Malik. Optimal Code Generation for Embedded Memory Non Homogeneous Register Architecture. *In Proceedings of 1995 International Symposium on System Synthesis*, setembro de 1995.

[13] Lee, E. A. Programmable DSP architectures: Part I. *IEEE ASSP* Magazine, pags 4-19, outubro de 1988.

[14] G. Goossens, P. G. Paulin, J. V. Praet, D. Lanneer, W. Geurts, A. Kiffi e C. Lierm. Embedded Software in Real-Time Signal Processing Systems : Design Technologies. *Proc. IEEE – Special Issue on Hardware/Software Co-Design*, setembro de 1997.

[15] P. G. Paulin, G. Goossens, C. Liem, M. Cornero e F. Naçabal. Embeddee Software in Real-Time Signal Processing Systems : Applications and Architecture Trends. *Proc. IEEE – Special Issue on Hardware/Software Co-Design*, setembro de 1997.

[16] Fabíola G. P. de Souza. Métodos Universais de Compressão de Dados. Tese de Mestrado, IC-UNICAMP, dezembro de 1991.

[17] Timothy C. Bell, John G. Cleary e Ian H. Witten. *Text Compression*. Advanced References Series. Prentice Hall, New Jersey, 1990.

[18] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098-1101, setembro 1952.

[19] J. Ziv e A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transaction on Information Theory*, 23(3), pags. 337 – 343, maio 1977.

[20] Andrew Wolfe e Alex Channim. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of Micro-25: The 25th Annual International Symposium on Microarchitecture*, pags 81-91, dezembro 1992.

[21] Michael Kozuc e Andrew Wolfe. Compression of Embedded Systems Programs. *In Proceedings of the IEEE International Conference on Computer Design*, pags 270-277, outubro, 1997.

[22] Michael Kozuch e Andrew Wolfe. Performance Analysis of the Compressed Code RISC Processor. *Tecnichal Report CE-A95-2*, Princeton University, 1995.

[23] Martin Benès, Andrew Wolfe e Steven M. Nowick. A High-Speed Assynchronous Decompression Circuit for Embedded Processors. *In Proceedings of 17th Conference on Advanced Research in VLSI*, Los Alamitos, CA, 1997. IEEE Society Press.

[24] Martin Benès, Steven M. Nowick e Andrew Wolfe. A Fast Asynchronous Huffman Decoder for Decompressed-Code Embedded Processors. In *Async98*. ACM, 1998.

[25] Charles Lefurgy, Peter Bird, I-Cheng Chen e Trevor Mudge. Improving Code Density Using Compression Techniques. *In Proceddings of Micro-30: The 30th Annual International Symposium on Microarchitecture*, pags 194-203, dezembro 1997.

[26] SPEC CPU'95, Technical Manual, agosto de 1995.

[27] Inc. Staff Apple Computer, Inc Staff International Bussiness Machine. PowerPc Microprocessor Common Hardware Reference Plataform - A System Architecture. Ed. Morgan Kauffman – 1997.

[28] www.gnu.ai.mit.edu/software

[29] Advanced RISC Machines. *ARM 7TDMI Data Sheet,* 1995.

[30] Intel Corporation. *Intel Architecture Developer's Manual*, 1997.

[31] Stan Y. Liao, Srinivas Devadas e Kurt Keutzer. Code Density Optimization for Embedded DSP Processors using Data Compression Techniques. *In Proceedings of 16th Conference on Advanced Research in VLSI*, pags 272-285, Los Alamitos, CA, outubro de 1995. IEEE Society Press.

[32] S. Liao, S. Devadas, e K. Keutzer. A Text-Compression-Based Method for Code Size Minimization in Embedded Systems, *ACM Transaction on Design Automation of Electronic Systems*, 4(1) ,1998.

[33] S. Liao, S. Devadas, K. Keutzer, S. Tjiang e A. Wang. Code Optimization Techiques in Embedded DSP Microprocessors. *In Proceedings of the 32nd Design Automation Conference*, pags 599-604, junho 1995.

[34] J. A. Storer e T. G. Szymanski. Data Compression via Textual Substitution . *Journal of the ACM*, 29(4), pags 928-951, outubro de 1982.

[35] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. Ed. McGraw-Hill, 1994.

[36] S. E. Goodman e S. T. Hedetniemi. *Introduction to the Design and Analysis of Algorithims.* Ed. McGraw-Hill, 1977.

[37] Grasselli, A. e Luccio, F. A Method for Minimization the Number of Internal States in Incompletely Specified Machines. *IEEE Transaction on Electronic Computers*, 14(1), pags 350-359, junho de 1965

[38] J. Gimpel. The Minimization of TANT Networks. *IEEE Transaction on Electronic Computers*, 16(1), pags 18-38, fevereiro de 1967.

[39] R. Rudell. Logic Synthesis for VLSI Design. *U. C. Berkley, ERL Memo* 89/49, 1989.

[40] O. Coudert. On Solving Binate Covering Problems. *Proceedings of the 33rd Design Automation Conference*, pags. 197-202, julho de 1995.

[41] A. Aho , R. Sethi J. Ullman. Compilers Principles, Techniques and Tools. Addison-Wesley, 1986.

[42] A. Aho e S. Johnson. Optimal Code Generation for Expression Tress. *Journal of the ACM*, 23, pags 488-501, julho 1976.

[43] A. Aho , S. Johnson e J. Ullman. Code Generation for Expressions with Commom Subexpression, *Journal of the ACM*, pags 146-160, janeiro 1977.

[44] Harris Lekatsas e Wayne Wolf. Code Compression for Embedded System. *In Proceedings of 35th Desing Automation Conference*, São Francisco, CA, julho de 1988.

[45] I. Witten, R. Neal e J. Cleary. Arithmetic Coding for Data Compression. *Communications of the Association for Computing Machinery*, 30(6), pags 520-540, junho 1987.

[46] G. Cormak e R. Horspool. Data Compression Using Dynamic Markov Modelling. *The Computer Journal*, 30(6), 1987.

[47] Jean Ernst, Willian Evans, Christopher W. Fraser, Steven Lucco e Todd A. Proebsting. Code Compression. In *SIGPLAN Programming Languages Design and Implementation*, 1997.

[48] Michael Franz e Kistler Thomas. Slim Binaries. *Comunication of the ACM*, 40(12):87-94, dezembro 1997.

[49] Todd A. Proebsting. Optimization an ANSI C Interpreter with Superoperators. *In ACM Conference on Principles of Programming Languages*, pags 322-332, janeiro 1995.

[50] Barry G. Haskell, Atul Puri e Arun N. Netraveli. *Digital Video: An Introduction to MPEG-2*. Chapman&Hill.

[51] C. E. Shannon A Mathematical Theory of Communication. *The Bell System Techinical Journal*, julho de 1948.

[52] R. M. Fano. *Transmission of Information*. M.I.T. Press, 1949.

[53] N. Abramson. *Information Theory and Coding*. McGraw-Hill, 1963.

[54] P. Elias. Universal Codeword Sets and Representation of Integers. *IEEE Transaction on Information Theory*, 21(2) , pags. 194-203, março de 1975.

[55] J. L. Bentley e C. McGeoch. Amortized Analyses of Self-Organizing Sequential Search Heuristcs. *Communications of ACM*, 28(4), pags. 404-411, abril de 1985.

[56] E. McCluskey. Minimizationof Booleans Functions. *The Bell System Techinical Journal*, pags. 1417-1444, novembro 1956.

[57] W. Quine. The Problem of Simplifying Truth Functions. *Americam Mathematical Monthly*, vol. 59, pags. 521-533, 1952.