Um Esquema de Compressão de Código para Processadores Embutidos

André Bellieny Roberto da Silva
Orientador: Gabriel Pereira da Silva

Rio de Janeiro

2006

Um Esquema de Compressão de Código para Processadores Embutidos

por

André Bellieny Roberto da Silva

Dissertação submetida ao Corpo Docente do Instituto de Matemática e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.).

Aprovada por:

____

Gabriel Pereira da Silva, D.Sc.
(Presidente)

____

Adriano Joaquim de Oliveira Cruz, Ph.D

____

Edil Severiano Tavares Fernandes, Ph.D

____

Eugene Francis Vinod Rebello, Ph.D

À memória de Elzi, minha mãe, que plantou as sementes do amor e do respeito, sempre incentivando minha busca pelo crescimento.

Agradecimentos

A Gabriel Pereira da Silva pela valiosa orientação desta pesquisa e pelo apoio e confiança nos momentos decisivos.

Aos professores e profissionais do NCE que fizeram parte desta trajetória de pesquisa.

À Biblioteca do NCE em especial a Selma Mendes.

À família pelo incentivo.

A todos que, de uma forma ou outra, colaboraram para a realização deste trabalho.

Resumo

Esta dissertação propõe um esquema de compressão de código para processadores embutidos RISC. Utilizando o núcleo do processador ARM SA11xx da *Intel Corporation* adicionado de um *hardware* capaz de expandir instruções, cria-se uma arquitetura capaz de trabalhar com aplicações comprimidas. Como conseqüência, pode-se implementar uma memória de sistema de menor capacidade, trazendo benefícios como a redução do consumo de energia de todo o sistema.

O *hardware* de expansão consiste de várias estruturas de armazenamento e controle projetados para um melhor desempenho das operações de expansão.

Uma aplicação escrita para a plataforma ARM é compilada e ligada pelas ferramentas tradicionais gerando o código executável. Este código deve ser processado por uma ferramenta em *software* que divide o código executável em pequenos trechos chamados de *blocos de compressão*. Esses trechos são comprimidos usando o código de Huffman, um algoritmo de compressão sem perda, que codifica caracteres mais freqüentes em seqüências menores de bits. Todas as vezes que uma instrução for buscada e houver uma falta na cache de primeiro nível, o *hardware* de expansão deverá iniciar o gerenciamento capaz de trazer e expandir a linha de cache requerida pelo processador.

Vale a pena registrar que a arquitetura proposta não está limitada ao processador e algoritmo de compressão escolhidos. Com algumas modificações pode-se implementá-la com outros processadores e outros algoritmos de compressão que se enquadrem na filosofia da arquitetura.

Para medir a eficiência da compressão e os efeitos da expansão de código no desempenho do sistema, são feitas simulações na plataforma modificada do SimpleScalar, um conjunto bem conhecido de ferramentas de simulação, que em sua mais recente versão inclui o conjunto de instruções ARM. O código do SimpleScalar foi modificado para simular as características da arquitetura proposta e o *benchmark* MiBench foi utilizado para avaliar a eficiência desta arquitetura. O MiBench é um pacote gratuito com aplicações típicas para sistemas embutidos. Vários parâmetros da arquitetura são variados durante as simulações, incluindo o tamanho do *bloco de compressão*.

As simulações do esquema proposto mostram que o código de Huffman consegue uma taxa de compressão de 76% em média para o conjunto de *benchmarks* estudados e que o desempenho relativo, quando comparado com uma arquitetura sem compressão, se mantém abaixo de 3,29 para blocos de compressão com 256 bytes.

Abstract

This dissertation proposes a code compression scheme for embedded RISC processors. Using the *Intel Corporation* SA11xx ARM processor core added by a hardware able to expand instructions, an architecture is created to work with compressed code. Therefore, a small system memory can be achieved, as a means of reducing overall power consumption. The expansion hardware consists of many storage and control structures designed for a better performance on expansion operations.

An application written for ARM platform is compiled and linked by traditional tools to obtain an executable code. This code may be processed by a software tool that splits it in chuncks named *compressed blocks* that will be compressed by Huffman code, a lossless compression algorithm that codes frequently used characters in smaller bit sequences. Every time a instruction is fetched and there is a instruction L1 cache miss, the expansion hardware has to manage the block fetch and expansion operations required to restore the complete line cache to the processor.

We would like to note that the proposed architecture is not limited on the selected processor and compression code. With some modifications the architecture can be used with other processors and compression algorithms that fit in the architecture philosophy.

To measure the compression rate and the effects of code expansion in the architecture performance, the Simplescalar is used for simulation. It consists of a well-known set of simulation tools used by researches and the last version includes the ARM instruction set. To include the proposed architecture features, the Simplescalar code was changed and the MiBench, a free set of typical embedded benchmarks were used. Many parameters in the experimental architecture are varied including the compressed block size.

The simulations show that Huffman code can achieve an average compression rate of 76% for the benchmarks used and the relative performance is limited to 3,29 for compressed block size of 256 bytes.

Sumário

1 Introdução **12**

2 Trabalhos Correlatos **18**
2.1 Wolfe e Chanin, 1992 18
2.1.1 Compressão 19
2.1.2 Arquitetura 19
2.1.3 Decodificador 21
2.1.4 Método Experimental 22
2.1.5 Desempenho 23
2.2 CodePack da IBM 23
2.2.1 Compressão 23
2.2.2 Expansão 27
2.3 Xu, Clarke e Jones, 2004 29
2.3.1 Arquitetura 29
2.3.2 *Hardware* de Expansão 30
2.3.3 Método Experimental 32
2.3.4 Conclusões 32
2.4 Outros 33

3 Proposta **35**
3.1 Criação e Execução de Uma Aplicação 36
3.2 Processador StrongARM SA-1110 37
3.3 Compressão de Código 39
3.3.1 Algoritmo de Compressão 42
3.3.2 Tabela de Endereçamento de Bloco Comprimido (TEBC) 44
3.4 Arquitetura RISC com Código Comprimido (ARCC) 47

4 Método Experimental **53**
4.1 Ferramentas 54
4.1.1 SimpleScalar 54
4.1.2 MiBench 58
4.2 Simulação da ARCC 64
4.2.1 Compressão 64
4.2.2 Expansão 67

5 Análise de Resultados **72**
5.1 Compressão 73
5.2 Expansão 76
5.2.1 Perfil 1: Arquitetura Base 76
5.2.2 Perfil 2: Variações no Tamanho do BBE 81
5.2.3 Perfil 3: Variações no Tamanho do BEE 83
5.2.4 Perfil 4: Alteração do Decodificador Huffman 84

6 Conclusões **87**
6.1 Futuros Trabalhos 88

A Código-Fonte dos Programas **90**
A.1 compressao.h 90
A.2 compressao.c 91

Lista de Figuras

2.1 Line Address Table - LAT 20
2.2 Entrada da LAT 20
2.3 Componentes de endereço 21
2.4 Organização da CLB 22
2.5 Formato das instruções comprimidas 25
2.6 Entrada da Tabela de Indexação de Compressão 26
2.7 Fluxo de Expansão [1] 27
2.8 Arquitetura CCTP 30
2.9 Decompression Engine — DE 31
2.10 Quociente de Compressão — CR_O 33
2.11 Desempenho Relativo — RP 34

3.1 Processo de Criação e Execução da Aplicação 36
3.2 Código Antes e Após a Compressão 41
3.3 Alinhamento de Byte: Área Entre os Blocos Comprimidos n e n+1 41
3.4 Codificação no Algoritmo de Huffman 43
3.5 Componentes do endereço original 44
3.6 Translação de um endereço original em endereço alvo usando a TEBC . . . 45
3.7 Indexação da TEBC 46
3.8 ARCC - Arquitetura principal 47
3.9 Gerenciador de Expansão 48
3.10 Registrador de Bloco Expandido — RBE 49
3.11 Organização do Buffer de Entradas de Endereços — BEE 50
3.12 Organização do Buffer de Endereços do BBE — BEBBE 50

4.1 Arquitetura do SimpleScalar 56
4.2 Formato de um arquivo executável ELF 65
4.3 Inserção de código de na função sim_main no arquivo sim-outorder.c 67
4.4 Função il1_access_fn modificada no arquivo sim-outorder.c 71

5.1 Comparação entre o código Huffman tradicional e o modificado com código médio para um bloco de compressão de 32 *bytes*. 74
5.2 Influência da TEBC na taxa de compressão 75
5.3 Taxa de faltas na *cache* de instruções L1 para as aplicações escolhidas . . . 78
5.4 Desempenho relativo das aplicações separadas por grupos. 78

5.5 Desempenho Relativo x Taxa de Compressão 81
5.6 Taxa de acertos para tamanhos de BBE variáveis 83
5.7 Influência do aumento do BEE nas aplicações 84
5.8 Influência do decodificador Huffman para as aplicações *lame*, *patricia* e *crc* 86

Lista de Tabelas

2.1 Esquema de codificação para os 16 *bits* da parte alta das instruções 24
2.2 Esquema de codificação para os 16 *bits* da parte baixa das instruções . . . 25

4.1 Características dos simuladores do SimpleScalar 55

5.1 Aplicações do pacote MiBench usadas nas simulações 72
5.2 Taxa de compressão para os diversos *benchmarks* em função do tamanho do bloco de compressão 75
5.3 Execução dos *benchmarks* sem a utilização da ARCC 76
5.4 Perfil Base da ARCC com variação do tamanho do bloco de compressão. . 77
5.5 Desempenho relativo das aplicações para parâmetros da arquitetura base. . 77
5.6 Taxas de acertos para o RBE 79
5.7 Taxas de acertos para o BBE 80
5.8 Taxas de acertos para o BEE 80
5.9 Perfil Base da ARCC com variação do tamanho do bloco de compressão. . 82
5.10 Desempenho relativo das aplicações com a variação do tamanho do BBE. . 82
5.11 Perfil Base da ARCC com variação do tamanho do bloco de compressão. . 83
5.12 Perfil Base da ARCC com o decodificador Huffman assíncrono. 84
5.13 Desempenho relativo com decodificador Huffman assíncrono. 85

Capítulo 1

Introdução

Os sistemas móveis baseados em processadores e alimentados por baterias ganham cada vez mais evidência nos dias atuais. Com a possibilidade de serem fabricados elementos processadores, memórias e baterias recarregáveis a custos mais baixos e com tamanhos menores, várias áreas antes baseadas em sistemas mecânicos, elétricos e eletrônicos dedicados, ganharam a possibilidade de serem baseados em processadores e com isso houve um aumento e sofisticação das possibilidades desses aparelhos. Incluem-se nestes casos os computadores de mão (*handhelds*), telefones celulares, computadores portáteis (*notebooks*) e reprodutores móveis de música e vídeo (a maioria com formatos comprimidos de arquivos de som e de vídeo, exigindo grande poder de processamento), entre outros.

O elemento processador passou a figurar nos sistemas eletrônicos digitais a partir dos avanços na tecnologia de integração de circuitos. A integração em pequena escala ou *Small Scale Integration* (SSI) permitiu que até 12 transistores pudessem ser integrados em um *chip* ou Circuito Integrado (CI). Isto era suficiente para prover as funções lógicas básicas E, OU, NÃO (inversor) e os elementos básicos de armazenamento como os *flip-flops* tipo D e J–K. A integração em média escala ou *Medium Scale Integration* (MSI) permitiu que de 13 a 99 transistores ou equivalentes fossem integrados em um CI. Uma vez que o número de pinos associados a um CI é limitado, era necessário interconectar

essas portas e implementar funções simples como contadores, decodificadores, registradores, comparadores e somadores. A decisão dos fabricantes de CI sobre que funções MSI implementar era baseada nos tipos de função que freqüentemente apareciam nos sistemas digitais projetados com SSI.

Quando a capacidade de encapsular 100 ou mais transistores num único CI — a integração em larga escala ou *Large Scale Integration* (LSI) — foi alcançada, a escolha por funções a serem implementadas se tornou difícil. Pela despesa significativa de desenvolver um circuito LSI, a função que o circuito deveria implementar teria que alcançar larga aplicabilidade, e portanto um mercado maior. O número de funções fixas complexas que satisfaz a esses requerimentos são limitadas. A solução para este problema é integrar uma função que não é fixa mas definida pelo usuário do dispositivo LSI. A função mais geral é o computador programável, um dispositivo lógico universal. Neste caso, o projetista do sistema programa o computador de tal forma que ele implemente a função requerida. A primeira empresa a fabricar um dispositivo deste tipo foi a *Intel Corporation* em 1971 com o MCS-4, um grupo de 4 CIs LSI que formavam um computador completo [2] e o *chip* que continha a unidade central de processamento era o 4004 [3]. O termo *microprocessador* se refere à um CI que contém as funções lógicas que vão formar a unidade central de processamento de um sistema.

O poder dos microprocessadores cresceu com a capacidade de encapsular mais transistores ainda num único *chip*. A integração em muito grande escala ou *Very Large Scale Integration* (VLSI), permitiu que mais de um milhão de transistores pudessem ser encapsuladas em um CI, fazendo com que processadores cada vez mais complexos fossem criados. Atualmente os processadores da *Intel Corporation* chegam a ter centenas de milhões de transistores.

Nesta época inicial de desenvolvimento, outro avanço na tecnologia dos semicondutores crucial para a popularização dos microprocessadores foi a possibilidade de se fabricar memórias não-voláteis regraváveis. Com as memórias só para leitura, *Read Only Memory*

(ROM), caso o projetista errasse o programa, todas as ROMs fabricadas para aquela aplicação ficariam inutilizadas. Com o advento da memória não-volátil apagável e regravável, *Erasable Programmable Read Only Memory* (EPROM) um sistema poderia ser reprogramado de forma prática (basta apagar a memória numa câmara de raios ultravioleta) até a conclusão dos testes se mostrar sem erros e aí sim, encomendar as ROMs com o programa correto armazenado nelas.

Com os microprocessadores podendo realizar uma vasta gama de aplicações, tanto é possível usá-los nos computadores de propósitos gerais como em sistemas menores em que executam uma função digital bem delimitada. Nesta última utilização, diz-se que são *processadores embutidos* e que realizam uma computação especializada. Eles fazem parte de um *sistema embutido* que é parte de um sistema maior ou de uma máquina.

Os processadores embutidos podem ter sua complexidade variando de simples microcontroladores de baixo custo para aplicações mais simples (como controlador de teclado de um microcomputador, controle digital de uma TV, circuito de programação de fornos microondas, etc), até processadores complexos e com recursos de alto desempenho (usados em computadores de mão, telefones celulares, etc). Neste último caso, fica difícil delimitar as fronteiras entre sistemas embutidos e de computação de propósitos gerais.

Outro ponto importante na classificação dos sistemas microprocessados é a possibilidade de torná-los móveis (sem fio para alimentação) dotando-os de outras fontes de energia como baterias recarregáveis. A evolução da tecnologia da fabricação destas baterias tornou não só possível que a autonomia dos aparelhos móveis aumentasse mas também que sistemas mais robustos em termos de desempenho (e conseqüentemente em consumo de energia) pudessem evoluir nessa área.

Um aspecto importante a se notar na evolução dos sistemas móveis é a *portabilidade*. O aumento da complexidade das aplicações executadas e o grande número de processadores diferentes oferecidos pelos fabricantes faz com que o código seja preferencialmente escrito em linguagens de mais alto nível como C, C++ e Java pela portabilidade, reapro-

veitamento de código e menor complexidade de codificação, comparada ao uso de uma linguagem de baixo nível. O impacto disso nos sistemas se reflete através de uma exigência maior de elementos de *hardware* como memória, que ocupa física e economicamente papel de destaque no sistema.

Sistemas móveis são sempre projetados em função de um equilíbrio de dois fatores: desempenho e consumo. Nos sistemas fixos como os microcomputadores de mesa (*desktop*) o consumo muitas vezes não é fator mais importante no projeto, apesar de que a maior potência consumida leva ao maior o aquecimento e gera outros problemas como a necessidade de uma refrigeração mais efetiva. Para um *desktop* sendo alimentado pela rede concessionária de eletricidade, julga-se que a interrupção de energia seja uma falha e não uma certeza, como é nos *notebooks* alimentados exclusivamente por bateria.

Nos sistemas móveis deseja-se que a bateria dure o maior tempo possível. Uma forma de se fazer isso é desenvolver baterias capazes de armazenar mais energia, o que nem sempre é viável em termos de custo. A outra é desenvolver técnicas de economia de energia em *hardware* que são comumente restritivas em termos de utilização plena deste último. Muitas pesquisas feitas nesta área incluem o desligamento de partes do hardware, otimização de barramentos em termos de consumo de energia, reorganização da hierarquia de memória, reorganização da *cache*, otimização na busca de instruções para redução de consumo, mudança de estratégia no escalonamento das instruções, compressão de código, entre outros.

Esta dissertação investiga um esquema de compressão de código, os desdobramentos no *hardware* do processador em estudo e no desempenho global do sistema.

A compressão é feita no código executável de um determinado processador. Este código é gerado a partir das ferramentas convencionais de compilação, ligação e montagem que podem ser oriundas de linguagens de alto ou mais baixo nível como as linguagens de montagem (*assembly*). Usando um algoritmo de compressão sem perda — já que na expansão queremos o programa exatamente como ele foi concebido — permite-se que o

código seja reduzido em espaço e possa ser armazenado em uma memória de menor capacidade (comparando-se com o esquema sem compressão). Uma memória menor significa um sistema mais barato e que consome menos potência devido à redução da área do circuito integrado e conseqüentemente do sistema total.

Para reduzir o tamanho do código existem algumas dificuldades: primeiramente não faz sentido expandir a aplicação por completo de uma vez, pois neste caso o tamanho da memória para armazenar o código teria que ser do tamanho do código sem compressão. Este esquema de compressão é perfeitamente viável para dados, e, como exemplo, temos o *gzip* do UNIX que comprime sem perda. A expansão do código deve ser feita somente na parte do código que está sendo solicitado pelo processador. Em segundo lugar, o código de uma aplicação não é executado de forma linear e seqüencial dos endereços de memória, pois existem instruções de desvio. Um programa comprimido ocupa um tamanho menor de memória do que o seu equivalente sem compressão, o que leva à alteração dos endereços físicos para os desvios. Uma forma de resolver isso é mexer no compilador e substituir os endereços de desvio por endereços reais. Uma outra é criar uma tabela que indique a tradução dos endereços físicos pelos comprimidos. Esta última abordagem não necessita da alteração do compilador.

A compressão de código vai influenciar diretamente no desempenho do sistema, haja vista que para a execução das instruções é necessário que haja uma expansão. A expansão por *software* é sempre preterida pela degradação do desempenho do sistema e por isso sempre é necessário um *hardware* a mais no sistema, responsável pela tarefa da expansão das instruções.

As idéias apresentadas nesta dissertação têm como alvo alguns modelos da arquitetura *Advanced RISC Machine* (ARM) e com algumas modificações podem ser estendidas para outras arquiteturas, também não se restringindo apenas aos sistemas móveis com processadores de baixa potência.

Este trabalho está organizado da seguinte forma:

O Capítulo 2 analisa os trabalhos correlatos da compressão de código e contém pesquisas iniciais até as mais recentes na área além de uma implementação comercial de compressão de código. O Capítulo 3 explica a arquitetura proposta, o processador e o algoritmo de compressão usados. No Capítulo 4 são investigadas as ferramentas para a simulação da arquitetura, as medidas importantes relacionadas à compressão e expansão e as configurações referentes aos retardos de *hardware*. O Capítulo 5 exibe a análise dos resultados das simulações para vários perfis de *hardware* da arquitetura. O Capítulo 6 estabelece as conclusões do trabalho, faz comparações com trabalhos correlatos e indica os trabalhos futuros. O Apêndice A contém os códigos-fonte das modificações feitas no SimpleScalar para incluir as características da arquitetura proposta.

Capítulo 2

Trabalhos Correlatos

Várias arquiteturas e outros tantos algoritmos de compressão foram propostos na literatura ao longo do tempo. Neste capítulo serão analisados os trabalhos que têm correlação com as propostas apresentadas nesta dissertação.

2.1 Wolfe e Chanin, 1992

Em 1992, Wolfe e Chanin [4] propuseram a arquitetura *Code Compression RISC Processor* (CCRP) — processador RISC de código comprimido — que consiste de um núcleo de processador RISC com uma memória *cache* especial adicional que recebe o código expandido e de um dispositivo de *hardware* que gerencia a expansão das instruções. Foi usado nesta arquitetura o MIPS R2000, um processador RISC de 32 *bits* de dados e 24 *bits* de espaço de endereçamento físico e com a linha de *cache* de 32 *bytes*. Com esta construção, não há mudança no núcleo do processador nem no seu conjunto de instruções.

O código executável é gerado de forma tradicional, usando linguagens de programação tradicionais e ferramentas de compilação e ligação. Uma vez gerado o código executável, ele será processado por um aplicativo (em *software*) que o comprimirá, e o código comprimido poderá ser executado na arquitetura CCRP.

2.1.1 Compressão

A compressão é feita por blocos de instruções de 32 *bytes* através do código de Huffman [5], e para que a implementação em *hardware* do decodificador seja viável, duas modificações são propostas. Na primeira, um *byte* não deve ser codificado com mais de 16 *bits*, pois no pior caso é possível que um *byte* seja codificado em 255 *bits*. A segunda modificação se refere à escolha de um código único para todos os programas. Uma vez que o código de Huffman é gerado a partir da freqüência dos *bytes*, e esta freqüência muda a cada programa, a eficiência da compressão não é a mesma que o algoritmo sem modificação. A escolha foi feita baseada na freqüência de *bytes* de dez programas diferentes. O autor chamou esta modificação de *Preselected Bounded Huffman*, que é o código de Huffman pré-selecionado dos dez *benchmarks* e limitado em 16 *bits* de largura de código para um *byte*.

Blocos de instruções que não podem ser eficientemente comprimidos (blocos cuja compressão aumenta seu tamanho) são deixados intactos.

2.1.2 Arquitetura

A estrutura *Line Address Table* (LAT) — Tabela de endereçamento de bloco comprimido — é incorporada ao circuito de reposição de *cache* e consiste em uma tabela que mapeia endereços de blocos de instruções em endereços de blocos de instruções comprimidas. Os dados na LAT são gerados na ferramenta de compressão e armazenados com o programa comprimido na própria memória de programa. A Figura 2.1 mostra a LAT para uma linha de *cache* de 32 *bytes*. Um endereço dentro da linha de *cache* é deslocado de 5 *bits* ($\log_2 32$ *bytes* da linha de *cache*) à direita, sendo então o endereço da linha de *cache* que vai ser traduzida pela LAT para encontrar o endereço da mesma linha no código comprimido. Se um bloco de instruções é deixado sem compressão, este é sinalizado na LAT.

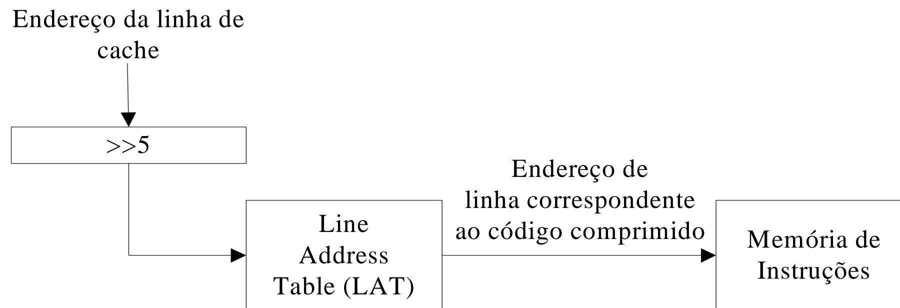

Figura 2.1: Line Address Table - LAT

A Figura 2.2 mostra que uma entrada da LAT tem largura de 8 *bytes* e contém informações sobre 8 blocos contíguos de instruções. Os primeiros 3 *bytes* contêm um ponteiro para o endereço físico do primeiro bloco de instruções comprimidas dentro do conjunto de 8 blocos. Em seguida há 8 grupos de 5 *bits* que indicam a largura em *bytes* do próximo bloco de instruções comprimidas. O tamanho do bloco em *bytes* varia de 1 a 31 sendo 0 reservado para blocos de 32 *bytes* (sem compressão). Desta forma para encontrar um endereço de um bloco de instruções comprimidas dentro dos oito blocos, basta somar o endereço base (3 primeiros *bytes*) com as larguras dos outros blocos.

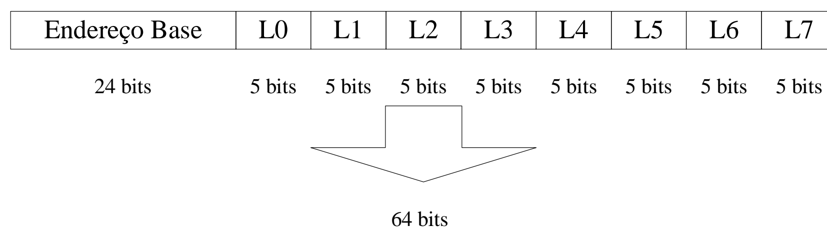

Figura 2.2: Entrada da LAT

O acesso à LAT aumenta o tempo da reposição da *cache* em pelo menos um ciclo de acesso à memória. Apesar do desempenho não cair tanto por isto só acontecer quando há reposição de *cache*, pode-se diminuir este efeito inserindo uma outra pequena *cache* para armazenar as entradas mais freqüentes da LAT. Esta *cache* chama-se Cache *Line Address Lookaside Buffer* (CLB). O mecanismo CLB/LAT é idêntico ao mecanismo TLB/Tabela de Página usado nos sistemas de memória virtual. A CLB é uma *cache* totalmente

associativa que pode conter de 4 a 16 entradas de LAT e com algoritmo LRU de reposição.

Durante a busca de cada instrução a CLB é varrida, e no caso de uma falta na *cache* ocorrer, a entrada na LAT estará pronta. O endereço físico é decomposto em 3 partes como mostra a Figura 2.3. Os cinco *bits* menos significativos indicam o deslocamento de um *byte* dentro da linha de *cache*. Os 16 *bits* mais significativos formam o índice da LAT e são comparados com os rótulos de cada local da CLB. Se um índice coincide com um rótulo da CLB e há uma falta na *cache*, então a correspondente entrada da LAT é usada para o cálculo do endereço para a reposição da *cache*. Se a entrada da LAT não está na CLB, então é necessária uma consulta na LAT para a computação do endereço correto. A organização da CLB é mostrada na Figura 2.4.

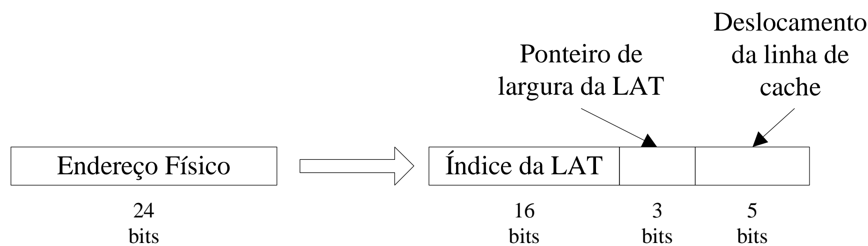

Figura 2.3: Componentes de endereço

A Unidade de Geração de Endereço (*Address Computation Unit*) usa os três *bits* restantes para determinar que elementos na entrada da LAT devem ser somados para calcular o começo do endereço do bloco comprimido. Esta unidade é implementada como uma árvore de circuitos somadores e o acesso à CLB mais a computação do endereço leva um ciclo de máquina.

2.1.3 Decodificador

Implementado em *hardware* o decodificador funciona com o carregamento de 2 *bytes* do programa comprimido para o registrador do decodificador, extrai a primeira seqüência válida de *bits* e traduz como o primeiro *byte* da linha de *cache*. Os *bits* remanescen-

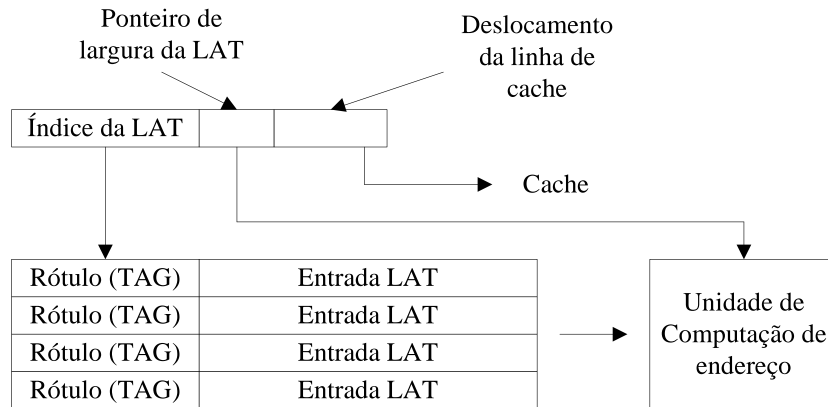

Figura 2.4: Organização da CLB

tes são deslocados e o registrador é preenchido com a próxima informação que vem do programa comprimido. Isto acontece duas vezes por ciclo de máquina, totalizando 16 ciclos para uma expansão de código de 32 *bytes* (linha de *cache*). Como é usado um código pré-selecionado, é possível consultar uma tabela gravada em ROM, PLA ou outra implementação de *hardware*.

2.1.4 Método Experimental

Foi implementado um simulador baseado em *trace* do processador MIPS R2000. Este simulador compara o desempenho de um R2000 com o R2000 adicionado da arquitetura de expansão de código. Os programas são todos escritos em C e FORTRAN e executados em estações de trabalho DECstations 3100.

O simulador produz uma variedade de estatísticas que incluem a taxa de falta na *cache*, tempo de substituição de linha de *cache* e penalidade da substituição na CLB. Os experimentos utilizaram modelos com *caches* variando de 256 *bytes* a 4096 *bytes*, CLB com 16, 8 e 4 entradas e memórias para programa como EPROM, *Burst* EPROM e DRAM.

2.1.5 Desempenho

O tamanho da CLB determina a freqüência com que a LAT vai ser acessada. Uma CLB grande é necessária quando a aplicação tem baixa localidade na execução de instruções.

Verificou-se que para memórias (de programa) lentas o modelo de código comprimido tem melhor desempenho que o modelo tradicional, sem compressão, para taxas altas de falta na *cache*, enquanto que o oposto é verdadeiro pra memórias mais rápidas.

2.2 CodePack da IBM

O CodePack da IBM [1] é uma solução comercial para a compressão de código. O processador RISC usado é o PowerPC da IBM, que tem 32 *bits* de dados e 32 *bits* de endereço. O núcleo do processador não é mudado nem é preciso implementar extensões arquiteturais. A compressão é feita através de ferramentas em *software* em que o ponto inicial é o código executável ELF original do PowerPC, não sendo necessário o código fonte.

2.2.1 Compressão

A compressão é baseada na observação de que as instruções usadas em aplicações práticas não são uniformemente distribuídas ao longo dos 32 *bits* usados para representarem uma simples instrução. Algumas instruções aparecem no código mais que outras e se forem codificadas com uma seqüência menor de *bits*, o código poderá ser armazenado em uma memória menor. A decodificação será feita substituindo a seqüência codificada pela instrução original que poderá ser executada no núcleo do processador PowerPC. Um refinamento que é efetivo nas instruções de 32 *bits* do PowerPC é comprimir separadamente a parte alta e baixa das instruções, ambas com 16 *bits*, já que várias instruções do processador PowerPC têm um deslocamento de 16 *bits* nos 16 *bits* de parte baixa da

instrução. Com esse esquema de codificação, há a redução de 20 a 30% de espaço na memória para armazenar o código.

As Tabelas 2.1 e 2.2 mostram o esquema de codificação para a parte alta e baixa da seqüência de 32 *bits* de uma instrução. O rótulo é o código inicial da instrução comprimida (em números) e o índice é representado pelos literais. O *software* de compressão processa o código original, uma instrução por vez, examinando a tabela de compressão e substituindo pela seqüência correspondente de *bits*.

Rótulo e Índice	**Largura (*bits*)**	**Descrição**
00 *aaa*	5	As 8 seqüências de 16 *bits* mais freqüentes com índice *aaa*
01 *bbbbb*	7	As próximas 32 seqüências de 16 *bits* mais freqüentes com índice *bbbbb*
100 *cccccc*	9	As próximas 64 seqüências de 16 *bits* mais freqüentes com índice *cccccc*
101 *ddddddd*	10	As próximas 128 seqüências de 16 *bits* mais freqüentes com índice *ddddddd*
110 *eeeeeeee*	11	As próximas 256 seqüências de 16 *bits* mais freqüentes com índice *eeeeeeee*
111 *yyyyyyyyyyyyyyyy*	19	*yyyyyyyyyyyyyyyy* não é um valor literal de 16 *bits* que ocorra freqüentemente para ter um código específico

Tabela 2.1: Esquema de codificação para os 16 *bits* da parte alta das instruções

Deve-se notar que quando uma instrução não pode ser comprimida, ou seja, quando as duas partes de 16 *bits* não podem ser representadas por um código comprimido, ela terá 38 *bits* de largura (19 *bits* representando a parte alta e 19 *bits* a parte baixa), tendo 6 *bits* a mais que a instrução original. Já a instrução comprimida com menor largura terá 7 *bits* (5 *bits* da parte alta e 2 *bits* da parte baixa).

A seqüência de *bits* armazenada na memória de programa será como indicado na

Rótulo e Índice	**Largura (*bits*)**	**Descrição**
00 (Sem índice)	2	Valor 0 (zero)
01 *aaaa*	6	As 16 seqüências de 16 *bits* mais freqüentes com índice *aaaa*
100 *bbbbb*	8	As próximas 32 seqüências de 16 *bits* mais freqüentes com índice *bbbbb*
101 *ccccccc*	10	As próximas 128 seqüências de 16 *bits* mais freqüentes com índice *ccccccc*
110 *dddddddd*	11	As próximas 256 seqüências de 16 *bits* mais freqüentes com índice *dddddddd*
111 *yyyyyyyyyyyyyyyy*	19	*yyyyyyyyyyyyyyyy* não é um valor literal de 16 *bits* que ocorra freqüentemente para ter um código específico

Tabela 2.2: Esquema de codificação para os 16 *bits* da parte baixa das instruções

Figura 2.5

Rótulo Alto	Rótulo Baixo	Índice Alto	Índice Baixo

Figura 2.5: Formato das instruções comprimidas

Para que uma instrução seja encontrada no código comprimido, já que saltos são possíveis na execução de um programa, a implementação de um mecanismo de tradução de endereços se faz necessário. A IBM resolveu isto dividindo o espaço de endereçamento lógico de 4GB em 64 regiões de 64MB de instruções comprimidas, cada um com sua própria tabela de indexação de compressão que servirá para traduzir o endereço alvo de programa (sem compressão) no endereço da instrução comprimida, armazenada na memória. As instruções comprimidas contidas dentro de uma região são divididas em grupos de 128 *bytes* (32 instruções) cada, e são conhecidas como grupos de compressão. Cada grupo de compressão é então subdividido em 2 blocos de 64 *bytes*, o de mais baixo endereço sendo o primeiro bloco e o de mais alto endereço sendo o segundo.

Uma entrada da tabela de indexação de compressão tem a largura de 4 *bytes* e aponta

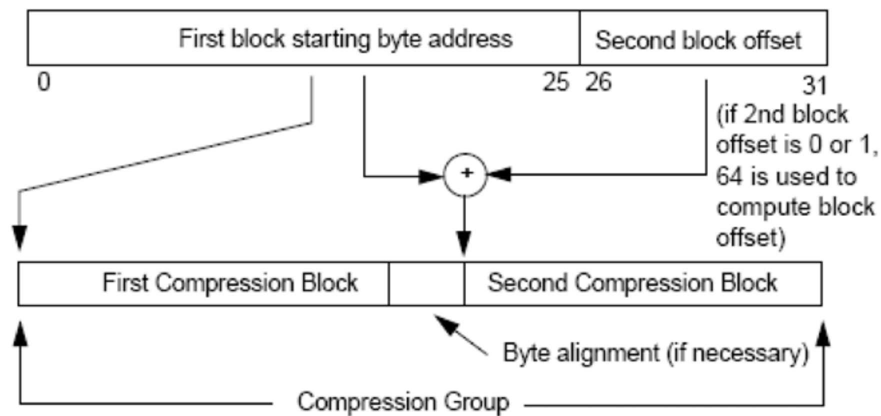

Figura 2.6: Entrada da Tabela de Indexação de Compressão

para um grupo de compressão. Esses 32 *bits* são divididos em 2 campos. Os primeiros 26 *bits* apontam para o deslocamento em *bytes* do início do primeiro bloco comprimido dentro da região de compressão de 64MB. Os próximos 6 *bits* indicam o deslocamento do segundo bloco comprimido após o início do primeiro bloco como indicado na Figura 2.6. Um valor de 1 nestes 6 *bits* indicam que o primeiro bloco contém instruções no formato original (não comprimido) e um valor 0 indica que ambos os blocos estão sem compressão.

As entradas da tabela de indexação de compressão são arranjadas por ordem de endereços. Dado que uma entrada desta tabela tem 4 *bytes* de largura e aponta para um grupo de compressão que representa 128 *bytes* de instruções sem compressão, segue que seu tamanho será de 2MB para uma região de compressão de 64MB.

As tabelas de indexação de compressão podem ficar em qualquer lugar do espaço de endereçamento de 4GB, começando em endereços que sejam múltiplos de 2MB. Os *bits* 4 e 5 de um endereço de programa alvo (sem compressão) selecionam um de 4 registradores do processador PowerPC (ITOR0-ITOR3) que apontam para a origem de uma tabela.

2.2.2 Expansão

A expansão das instruções consiste em usar as tabelas de índice para localizar o endereço inicial do bloco comprimido contendo o endereço real (sem compressão) da instrução. O bloco comprimido é trazido para o controlador de expansão, expandido para o seu formato original, e a instrução requerida é enviada para o núcleo do processador. É possível o PowerPC trabalhar com código comprimido tanto no modo real como no modo virtual.

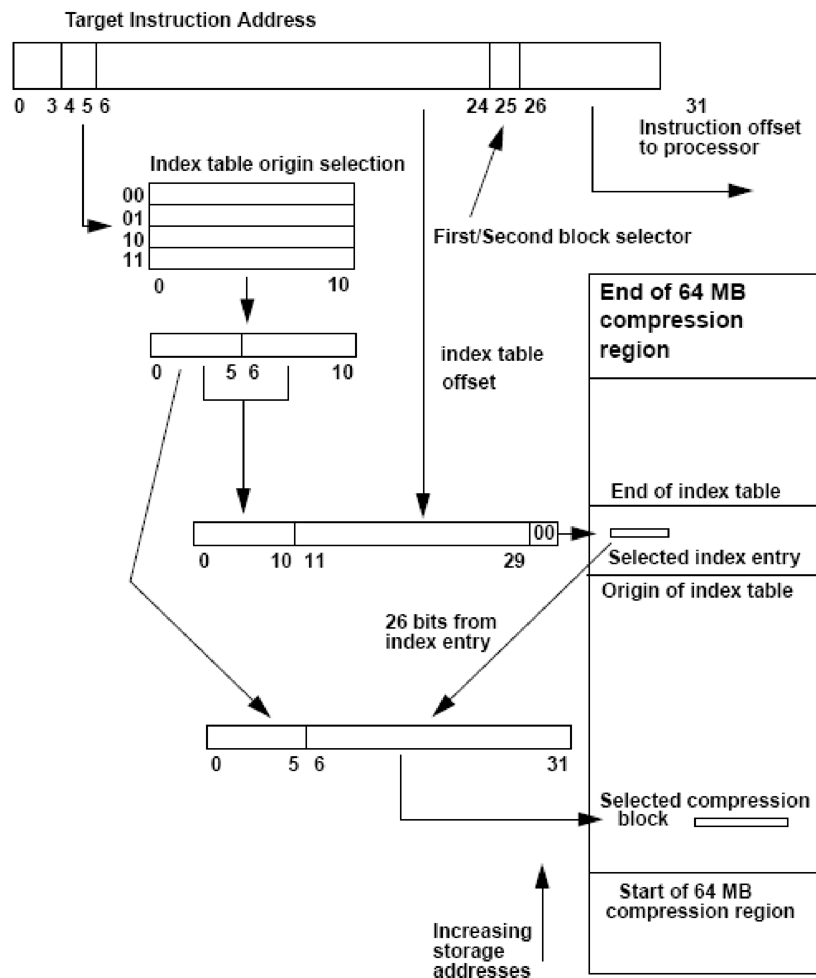

Figura 2.7: Fluxo de Expansão [1]

O hardware de expansão é composto de registradores auxiliares e unidades que calculam os endereços a partir dos dados contidos no registrador de instruções e na tabela de

índices. A Figura 2.7 mostra como um endereço alvo de uma instrução localizada na área de compressão é traduzida para o endereço do *byte* inicial de um bloco de compressão. A seguir uma explicação detalhada é apresentada.

- O núcleo do processador requisita uma instrução com endereço TIA (Target Instruction Address — endereço de instrução alvo).

- *Bits* 4 e 5 de TIA são usados para selecionar um de 4 ITOR (Index Table Origin — origem da tabela de índices) que são registradores do processador PowerPC que armazenam endereços iniciais de tabelas de índice. Os próximos 11 *bits* são responsáveis pelo endereço de uma entrada da tabela de índices.

- Os 11 *bits* do registrador ITOR selecionado são combinados com os 19 *bits* de deslocamento da tabela de índices dos *bits* 6:24 do endereço TIA ($TIA_{6:24}$) para formar o endereço da palavra da entrada selecionada da tabela de índices.

- O controlador de expansão lê os 4 *bytes* da entrada da tabela de índices localizado na memória externa. A informação nesta entrada da tabela aponta para um grupo de compressão na memória externa que contém a instrução desejada.

- TIA_{25} seleciona o primeiro ou segundo bloco de compressão na qual está a instrução requerida. Se o *bit* 25 é 0, selecionando o primeiro bloco, os 26 *bits* mais significativos da entrada da tabela de índices são combinados com os *bits* 0:5 do registrador ITOR para formar o endereço do *byte* inicial do primeiro bloco de compressão. Um deslocamento de 0 ou 1 no segundo bloco inicia um procedimento especial. Em qualquer caso o primeiro bloco não é comprimido. Então se o segundo bloco é endereçado, é adicionado o valor de 64 (que é a largura de um bloco não comprimido) ao endereço do primeiro bloco para formar o endereço do *byte* inicial do segundo bloco. Além disso, o valor especial de 0 indica que o segundo bloco não tem compressão também.

- O controlador de expansão lê e expande todo o bloco. No caso em que a entrada da tabela de índices indica que não há compressão no bloco, este é lido e o processo de expansão é inibido.

- As instruções sem compressão são fornecidas ao núcleo do processador. O controlador de expansão armazena todo o bloco sem compressão (64 *bytes* ou 16 instruções) até que o núcleo do processador requeira uma instrução localizada em um bloco diferente.

2.3 Xu, Clarke e Jones, 2004

Em 2004, Xu, Clarke e Jones [6] descreveram um sistema de compressão de instruções do processador ARM/THUMB com a expansão em *hardware* utilizando o algoritmo de compressão *X-Match PRO*.

Como nos outros exemplos, não foram feitas mudanças no núcleo do processador estudado nem no seu conjunto de instruções. Este trabalho se propõe a mostrar que mesmo com o processador ARM no modo THUMB, que tem uma densidade maior de código, pode ainda assim ter o código comprimido com vantagens na economia de energia.

2.3.1 Arquitetura

É uma versão modificada do CCRP [4] e se chama *Code Compression THUMB Processor* (CCTP) — processador THUMB com compressão de código. Enquanto o CCRP requer que o tamanho do bloco a ser comprimido seja o mesmo da linha de *cache* (32 *bytes* na maioria dos núcleos ARM), o CCTP trabalha com blocos de compressão que sejam alguma potência inteira de 2 vezes o tamanho da linha de *cache* L1. A Figura 2.8 mostra a arquitetura do CCTP.

Na impossibilidade de preencher uma linha de *cache* com um bloco de compressão

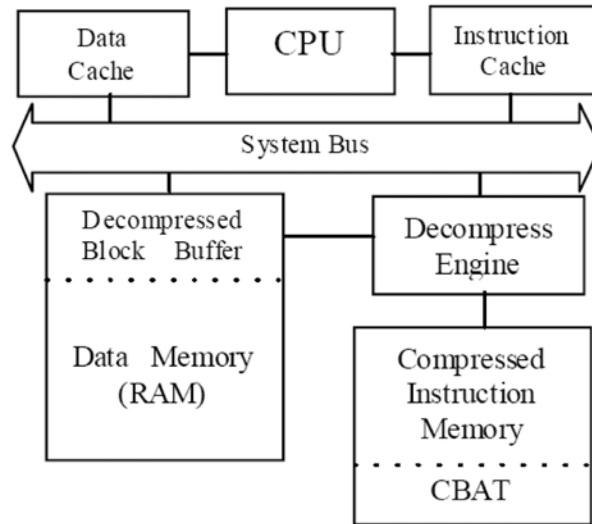

Figura 2.8: Arquitetura CCTP

expandido, é implementado um *buffer* na memória, chamado *Decompressed Block Buffer* (DBB) — *buffer* de bloco expandido, que armazena temporariamente os blocos de instruções expandidos e é tratado como uma *cache* L2. O instrumento de expansão (*Decompression Engine* ou DE) executa 4 tarefas:

- Controla todas as faltas (*misses*) na *cache*.
- Gerencia DBB como uma *cache* L2.
- Provê a linha de *cache* requerida.
- Providencia a execução da expansão.

Uma tabela, a *Compressed Block Address Table* — CBAT (tabela de endereçamento de bloco comprimido) com a mesma função da LAT [4] é usada para traduzir o espaço de endereçamento comprimido para o espaço sem compressão.

2.3.2 *Hardware* de Expansão

O DE faz fronteira com o DBB (dentro da memória), a memória de instruções (programa comprimido e CBAT) e barramentos do sistema. Ele é o centro de toda arquitetura.

A Figura 2.9 mostra o diagrama de blocos da DE.

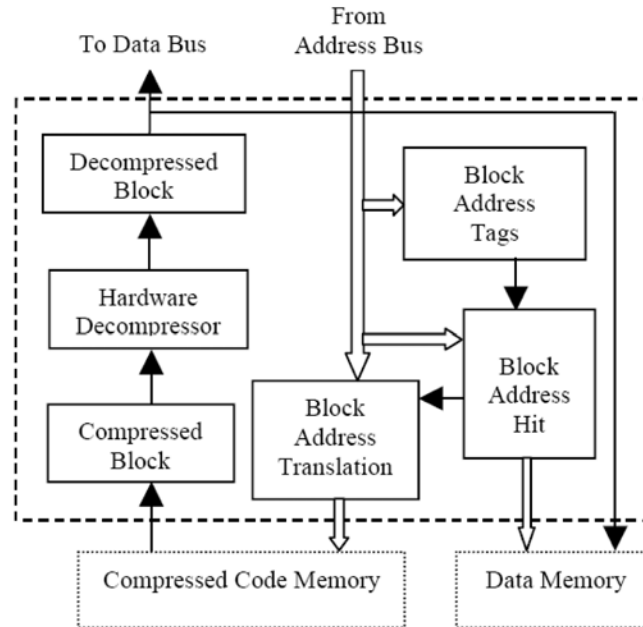

Figura 2.9: Decompression Engine — DE

Uma das mais importantes tarefas do DE é gerenciar o acerto do DBB. No DE uma memória RAM é incluída para armazenar os rótulos dos endereços dos blocos expandidos. O módulo *Block Address Translation* (tradução de endereço de bloco) converte o espaço de memória sem compressão para o espaço comprimido para a correta aquisição dos blocos comprimidos na memória de instruções. Dois *buffers* são integrados para conter o bloco expandido e o bloco comprimido corrente.

A arquitetura CCTP não se restringe a um só algoritmo. Isso propicia incorporar algoritmos eficazes na arquitetura. O *hardware* de expansão é escolhido segundo critérios de espaço e consumo de energia. Vale lembrar que nem todos os algoritmos podem ser implementados eficientemente em *hardware*. Um bom *hardware* de expansão deve ter execução em alta velocidade, compressão efetiva e uma baixa complexidade em *hardware*.

O Algoritmo de compressão/expansão escolhido foi o X-MatchPRO, baseado em tecnologias FPGA. X-MatchPRO é um compressor geral para qualquer tipo de dados que se

mostrou eficiente para código ARM/THUMB.

2.3.3 Método Experimental

Foram usadas as ferramentas do Simplescalar [7] como plataforma de simulação modificando a *cache* L2 para funcionar como DBB.

Como *benchmarks* foram usados programas do pacote MiBench [8]. Eles foram compilados pelo *tcc* da ferramenta *ARM Developer Suite* (ADS) versão 1.2 com o flag O2 de otimização ligado.

A DDB usada tem 4K *bytes*, reposição FIFO e associatividade total. Os blocos de compressão variaram entre 64, 128, 256, 512 e 1024 *bytes*.

Como medidas para quantificar a compressão e o desempenho, foram usadas: *Overall Compression Ratio* ($\mathrm{CR_O}$) que é o quociente de compressão e o *Relative Performance* RP que é o desempenho relativo da execução com compressão à execução sem compressão.

$$CR_O = \frac{\text{Tamanho da Aplicação Comprimida + Tamanho da CBAT}}{\text{Tamanho da Aplicação Original}}$$

$$RP = \frac{\text{Ciclos de Execução da Aplicação Comprimida}}{\text{Ciclos de Execução da Aplicação Original}}$$

2.3.4 Conclusões

Quanto à compressão, as aplicações se mostraram semelhantes com compressão alta para blocos de compressão maiores. Nas simulações, o $\mathrm{CR_O}$ ficou por volta de 0,95 para blocos de 64 *bytes* e 0,75 para blocos de 1024 *bytes* para todas as aplicações simuladas, o que é mostrado na Figura 2.10

A Figura 2.11 mostra que o desempenho relativo das aplicações variou muito. O RP das aplicações variou de valores próximos de 1,0 até 7,0, aumentando com o aumento do bloco de compressão. Algumas aplicações com baixa taxa de faltas na *cache* (como

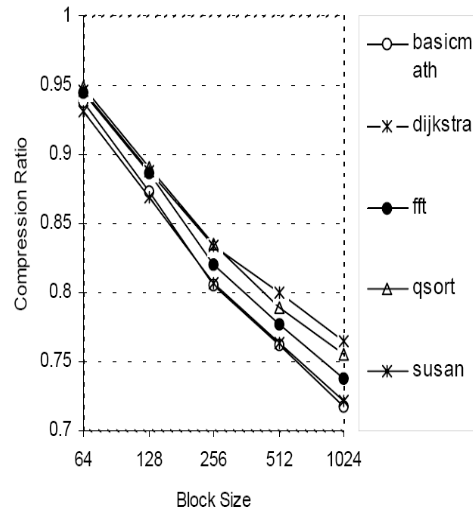

Figura 2.10: Quociente de Compressão — $\mathrm{CR_O}$

susan-s e *dijkstra*) se mostraram com desempenho relativo próximo de 1,0 mesmo com blocos de compressão de maior tamanho.

Para efeito de equilíbrio entre compressão e desempenho, verifica-se que blocos de 128 *bytes* têm um melhor compromisso entre esses parâmetros, uma vez que o tempo de execução fica em torno de 50% mais lento com relação à aplicação original, mas o ganho em espaço de memória é de 15 a 20% como mostra a Figura [**?**].

2.4 Outros

A partir da arquitetura proposta por Wolfe e Chanin [4] , vários algoritmos foram propostos, uma vez que as particularidades da compressão de código requerem certos cuidados. Lekatsas e Wolf [9] e [10] desenvolveram alguns algoritmos. Em [9] propõem dois algoritmos, um independente do conjunto de instruções: *Semiadaptative Markov Compression* (SAMC) — compressão semi-adaptativa Markov — que usa um método de codificação estatístico e outro dependente do conjunto de instruções: Semiadaptative Dictionary Compression (SADC) — compressão semi-adaptativa baseada em dicionário — em que um dicionário é construído com seqüências comuns de instruções. Em [10] surge

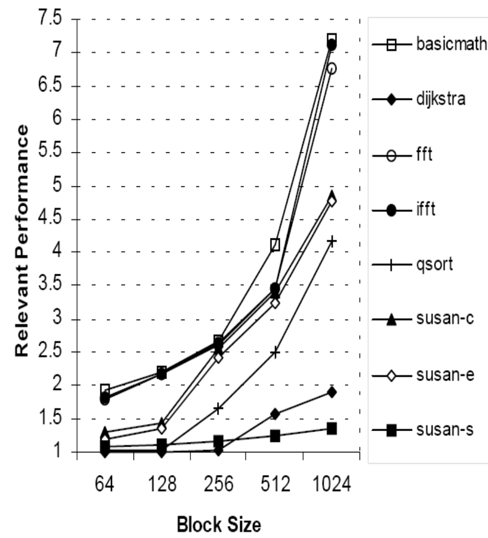

Figura 2.11: Desempenho Relativo — RP

a proposta de um algoritmo baseado em codificação aritmética que permite a expansão das instruções ser iniciada em qualquer parte do arquivo comprimido.

O CodePack da IBM foi também objeto de estudo de Lefurgy, Piccininni e Mudge em [11] que analisa o comportamento de outras arquiteturas (usando a plataforma de simulação do Simplescalar [7]) com o esquema de compressão proposto pela IBM e descrito na Seção 2.2.

Capítulo 3

Proposta

Este trabalho de pesquisa tem por objetivo investigar os efeitos da compressão de código em processadores embutidos, em especial os processadores *StrongARM* da *Intel Corporation* [12] que são representativos dentro do conjunto de processadores embutidos RISC (*Reduced Instruction Set Computer*, ou computadores com conjunto de instruções reduzido). A arquitetura proposta nesta dissertação denomina-se *Arquitetura RISC com Compressão de Código* (ARCC) e é composta de um processador StrongARM SA-1110 adicionado de um *hardware* auxiliar responsável pelo gerenciamento das funções decorrentes da compressão de código.

A compressão do código executável permite a diminuição da memória física que armazena este código. Com a redução do número de acessos à memória e esta necessitando de uma área menor a ser alimentada, baixa-se também o consumo de potência como mostra [13]. Isto leva à diminuição do preço da unidade e do uso de energia, este último fator importante para sistemas alimentados por bateria.

As próximas seções trarão detalhes sobre os elementos envolvidos no esquema de compressão de código proposto nesta dissertação.

3.1 Criação e Execução de Uma Aplicação

A Figura 3.1 mostra o processo total de criação e execução de uma aplicação com o esquema de compressão de código proposto. Vale aqui salientar que o núcleo do processador em estudo não precisa de alterações, mas é preciso aumentar o *hardware* ao redor dele segundo as necessidades do esquema proposto. A expansão (processo oposto à compressão) das instruções comprimidas será feita por *hardware* e existirá um circuito que gerencia esta transformação. A opção por uma expansão em *hardware* e não por *software* é feita para que o desempenho do sistema não tenha degradação significativa. Em alguns trabalhos mais recentes, [14], [15] e [16], a expansão é feita por *software*, mas existe uma diferença na compressão que é feita apenas em partes do código pouco usadas escolhidas pela observação de perfis de execução (*profiles*).

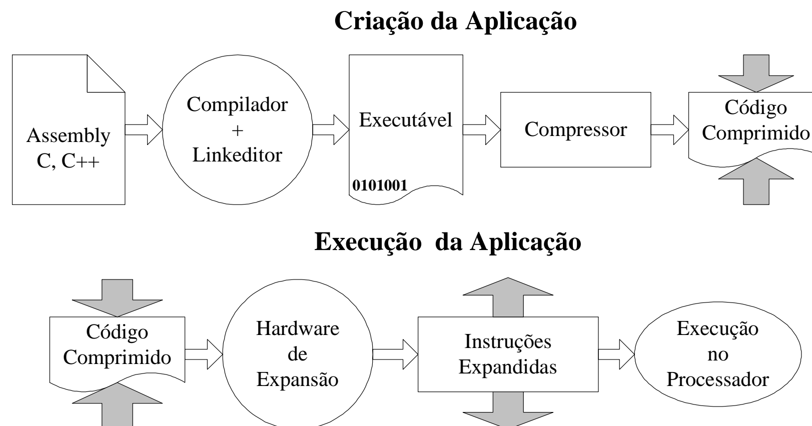

Figura 3.1: Processo de Criação e Execução da Aplicação

De uma forma geral, o processo total de criação e execução de uma aplicação para a arquitetura proposta consta dos seguintes passos:

Criação da aplicação: O código fonte é compilado e ligado pelas ferramentas tradicionais relativas ao processador. O código executável é então comprimido através de uma ferramenta de *software* que pode ser executada em um outro sistema. O código comprimido pode ser transferido para a memória da máquina alvo através

das ferramentas disponibilizadas para esta finalidade pelos projetistas do sistema alvo.

Execução da aplicação na máquina alvo: O *hardware* de expansão, responsável pela transformação da informação comprimida em instruções capazes de serem executadas pelo núcleo do processador, entra em ação todas as vezes que uma instrução requerida pelo processador não se encontra na sua *cache* de nível 1 (L1).

Pela descrição anterior pode-se concluir que a unidade mínima de expansão deve ser o número de *bytes* de uma linha de *cache*. Como a proposta é não mudar o núcleo do processador, sua *cache* L1 deve ser provida de informação não comprimida. Em [17] há uma outra abordagem em que a *cache* L1 recebe o código comprimido mas deve-se salientar que o núcleo do processador alvo tem que ser alterado para que ele realize a expansão das instruções em algum momento.

3.2 Processador StrongARM SA-1110

O processador escolhido para integrar a arquitetura com compressão de código é o *StrongARM SA-1110* fabricado pela *Intel Corporation.* É um processador RISC de propósitos gerais de 32 *bits* usado em computadores de mão, comutação em redes de alta interconexão, sistemas de armazenamento e dispositivos de acesso remoto. Este processador apresenta as seguintes características:

- Alta performance:
  - 150 Dhrystone 2.1 MIPS @ 133 MHz
  - 235 Dhrystone 2.1 MIPS @ 206 MHz
- Baixa potência:
  - <240 mW @1.55 V/133 MHz

  - <400 mW @ 1.75 V/206 MHz

- Gerador de pulsos de relógio integrado;
- Características de gerenciamento de energia;
- Modos de operação *big endian* e *little endian*;
- *Caches* associativas por conjunto *32-way*
  - 16 *Kbytes* para *cache* de instruções
  - 8 *Kbytes write-back* para *cache* de dados
- Unidade de gerenciamento de memória (MMU) com 32 entradas;
- *Buffers* de entrada e saída;
- Barramento de memória contendo interfaces para vários tipos de memória.

Além de todas as características positivas de seu *hardware*, o StrongARM é suportado pela plataforma de simuladores Simplescalar [7] bem conhecido no meio acadêmico e que é explicado com detalhes na Seção 4.1.1. Com isto é possível levantar medidas importantes que quantifiquem os efeitos da compressão de código para este processador.

O núcleo deste processador é derivado do StrongARM SA-110. Pelas características semelhantes destes processadores, é possível utilizar qualquer processador da família StrongARM SA-11xx com a arquitetura proposta sem alterações. As idéias gerais também se encaixam a processadores RISC embutidos em geral, com poucas modificações.

Processadores RISC têm um conjunto de instruções reduzido, sendo instruções de largura fixa e capazes de serem executadas em 1 ciclo de máquina. Com estas características o paralelismo por *pipeline* torna-se natural nestes processadores.

A motivação para o projeto dos processadores RISC se deveu a mudanças dos parâmetros nas arquiteturas dos computadores como resultado da evolução tecnológica. No meio dos

anos 70 a memória era cara e desejava-se menor complexidade dos compiladores, daí arquiteturas com conjuntos de instruções complexas — *Complex Instruction Set Computer* (CISC) — realizando muitas tarefas para uma mesma instrução, o que levou à popularidade da microprogramação [18]. Deve-se notar que os processadores CISC têm a vantagem das aplicações terem uma densidade de código maior que nas aplicações para RISC.

Com o surgimento das memórias dinâmicas mais rápidas e mais baratas fora do núcleo da CPU, e mais tarde com as *caches*, a microprogramação acaba sendo preterida pela programação externa já que o tempo de acesso a microprogramas se equipara ao tempo de acesso dos programas na memória. Além disso com a freqüência de relógio aumentando cada vez mais, a filosofia dos processadores RISC se estabeleceu e estes se tornaram populares.

Pela natureza pouco densa do código RISC, a compressão de código é bastante efetiva para as aplicações destes processadores.

3.3 Compressão de Código

A compressão de código é um caso especial da compressão de dados e esta distinção deve ser feita por algumas razões importantes. Diferentemente de algumas aplicações na área da compressão de dados (compressão de imagens, sons, etc), a compressão de código não pode ser feita com perda, pois na reconstituição da informação o código tem que ser igual ao original antes da compressão. Além disso, a forma como se comprime o código tem que prever uma possível expansão em pontos aleatórios da parte comprimida pois não é possível uma expansão completa dos dados (perdendo o efeito da compressão) e a expansão das instruções não é seqüencial já que o código contém instruções de salto. Vale ressaltar a diferença entre os termos comprimir e compactar. A compactação de código não requer uma descompactação pois é um processo irreversível e não há diferença semântica entre o código original e o compactado. Como exemplo de compactação, temos

as otimizações feitas por métodos tradicionais de um compilador como a eliminação de código não utilizado.

A unidade de compressão é tradicionalmente o caractere ou grupos de 8 *bits*. Alguns tipos de compressão utilizam unidades com maior largura em *bits*, dependendo das características envolvidas e como exemplo temos o CodePack da IBM, explicado na Seção 2.2.

Existem diferentes métodos de compressão de código como mostrado em [19]. Dentre as diversas classificações e características podemos citar 3 grupos de métodos que se destacam na compressão de código executável.

Codificação por freqüência: Os símbolos com maior freqüência são codificados com seqüências menores de *bits* e, dependendo de como se faz a codificação pode-se classificá-la como estática ou dinâmica. Na forma estática os códigos dos símbolos são fixos para toda a informação a ser comprimida, já na dinâmica os códigos podem variar de acordo com as freqüências em um determinado instante da codificação. Dentro destes métodos destaca-se o Código de *Huffman* [5] e suas variações.

Codificação aritmética: Esta codificação usa probabilidades diretamente no lugar das freqüências. A idéia é ter uma linha de probabilidades [0–1) e associar a cada símbolo uma faixa desta linha onde uma maior probabilidade do símbolo corresponde a uma faixa maior. A codificação é feita por transformações lineares e colocação dos símbolos em suas faixas de freqüência. A saída é um número real mas com algumas modificações é possível transformá-los em números inteiros para o processamento digital [20].

Métodos baseados em dicionário: Nestes métodos, seqüências de símbolos comuns são codificados e substituídos pelos seus códigos menores. Há muitas variações deste código mas se destacam alguns como o *Move-to-Front* (MTF) *coding* e a família de compressores Lempel-Ziv [21].

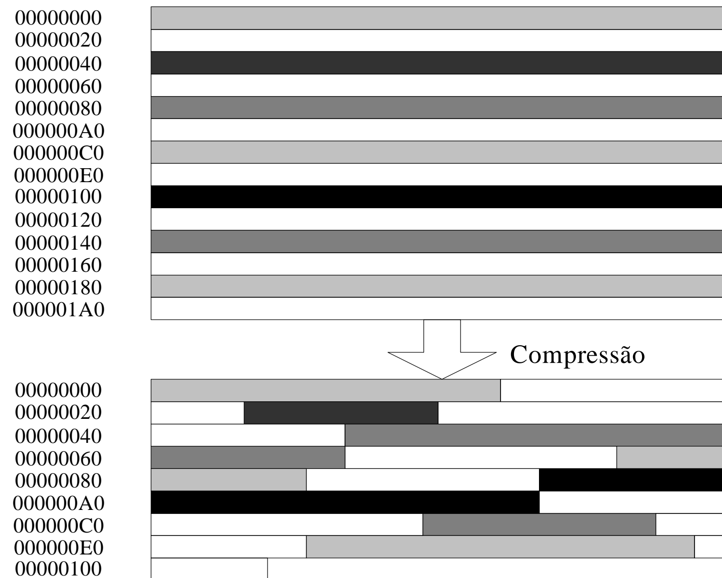

Figura 3.2: Código Antes e Após a Compressão

A Figura 3.2 mostra como fica a compressão em 32 *bytes* (tamanho de uma linha de *cache* L1 do SA-1110). Esta abordagem tem a pequena desvantagem de diminuir a compressão do código executável, pois tomando como exemplo uma codificação variável com uma largura menor para os *bytes* mais freqüentes, é possível que a largura de um bloco comprimido não seja alinhada em *bytes* como mostra a Figura 3.3, perdendo vários *bits* por bloco, o que não chega a comprometer o sistema de compressão. A grande vantagem é a simplicidade de se reconhecer o endereço do início de um bloco comprimido que começa em um *byte* e não em um *bit* qualquer no meio do byte.

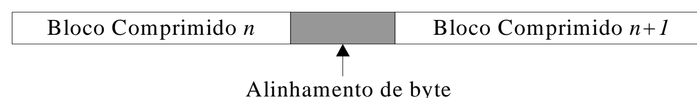

Figura 3.3: Alinhamento de Byte: Área Entre os Blocos Comprimidos n e n+1

Uma vez com o código comprimido, o sistema pode carregá-lo na memória para ser executado. Como o núcleo ARM não foi alterado, é preciso que no caso de uma falha no acesso à *cache*, o gerenciador de expansão entre em ação disparando o *hardware* de expansão do bloco correto e fornecendo à *cache* a linha cujo endereço foi buscado.

3.3.1 Algoritmo de Compressão

O algoritmo de compressão escolhido para este trabalho foi o código de *Huffman* com algumas alterações. Este algoritmo se baseia na codificação de símbolos mais freqüentes por cadeias de *bits* menores [5] [22]. A escolha foi feita levando em conta algumas características importantes. Este método tem boa taxa de compressão, é facilmente modificado para poder corresponder às características da compressão de código, e pode ter seu decodificador implementado em *hardware* com desempenho bem satisfatório como citado em [4].

A freqüência dos caracteres é determinada em uma leitura dos dados a serem comprimidos. A partir daí as duas menores freqüências são somadas gerando uma nova freqüência que substitui as outras duas durante o algoritmo de criação da árvore. Uma das pontas agrupadas recebe 0 e outra 1. O algoritmo segue buscando novamente as próximas duas menores freqüências e procedendo como anteriormente. Quando o último agrupamento soma 1, é o fim da geração da árvore e o código para um determinado símbolo está estabelecido pela seqüência de *bits* que vai da raiz da árvore até o símbolo. Estes passos estão detalhados na Figura 3.4.

Os símbolos são substituídos pelo seu código no arquivo comprimido. Para expandir o código comprimido, basta seguir a árvore binária através dos *bits* do código e quando encontrar uma folha, o código do caractere está terminado e basta substituir pelo seu byte original consultando-se a tabela de codificação dos símbolos.

Para adequar o código de Huffman ao trabalho, a parte executável do arquivo objeto será codificado em blocos de $2^n \times 32$ *bytes* por vez, onde n é um inteiro e 32 *bytes* correspondem à largura de uma linha de *cache* L1 do ARM SA-1110. Esses agrupamentos de *bytes* serão denominados *blocos de compressão*. O parâmetro n será alterado nas simulações para avaliações e poderá valer 1, 2, 4, 8, 16 ou 32 correspondendo a blocos de compressão de 32, 64, 128, 256, 512 ou 1024 *bytes*, respectivamente. Um *bloco comprimido*

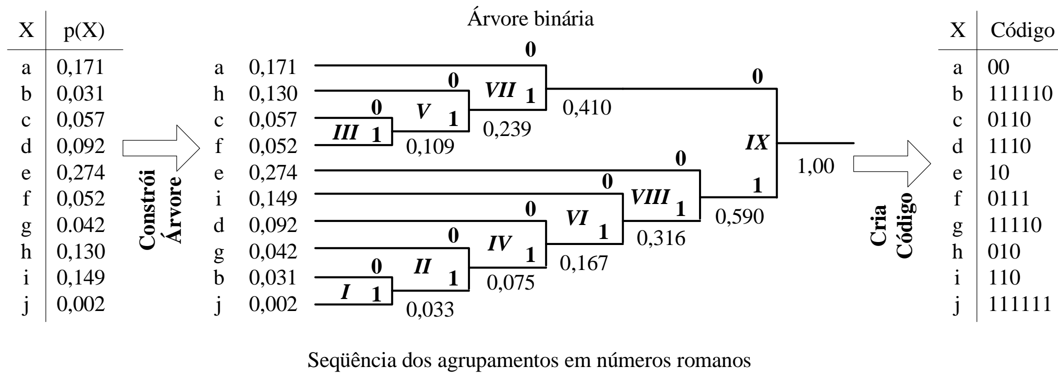

Figura 3.4: Codificação no Algoritmo de Huffman

por sua vez se referirá ao bloco de compressão codificado pelo algoritmo de Huffman.

O código de Huffman permite que um *byte* seja codificado com até 255 *bits*. O algoritmo utilizado na compressão para este trabalho, é modificado e não permite que um símbolo seja codificado com mais de 16 *bits* como em [4]. Isto é importante pois o *hardware* de expansão [4], [23] não é eficaz com grandes cadeias de *bits*.

Outra modificação feita no algoritmo diz respeito às freqüências dos símbolos. Para que a codificação seja a melhor possível para o algoritmo dado, os símbolos devem ser codificados de acordo com as freqüências respectivas do código, o que faz com que cada programa tenha sua própria tabela de códigos. Com isto a cada execução de um programa diferente, uma tabela de códigos e informações complementares devem ser carregadas no *hardware* de expansão para que o decodificador Huffman possa expandir as instruções convenientemente. Uma solução a esta alternativa é ter um código fixo para todos os programas e então o código e as informações complementares podem ser implementados no *hardware* de expansão como em [4]. Uma forma simples de encontrar um código "médio" é tirar a média ponderada das freqüências de vários códigos de programas com relação ao seu tamanho. No Capítulo 5 é mostrado o resultado desta simplificação que traz resultados bastante satisfatórios.

3.3.2 Tabela de Endereçamento de Bloco Comprimido (TEBC)

Ao mesmo tempo que os blocos de compressão estão sendo comprimidos pelo *software* responsável, é construída uma tabela, a *Tabela de Endereçamento de Bloco Comprimido* (TEBC), com a mesma função da LAT em [4] e da CBAT em [6], que deverá ser carregada para a memória na área de programa junto com o código comprimido quando o programa for carregado para execução. A TEBC deverá conter informações para que o gerenciador de reposição de *cache* possa localizar qualquer bloco de compressão na área de memória de programa comprimido (*endereço alvo*) a partir de seu endereço correspondente ao endereço do programa sem compressão (*endereço original*), independente do sistema de memória ser real ou virtual. Uma entrada TEBC contém um endereço alvo de 32 *bits*. A Figura 3.5 mostra os componentes de um endereço original, onde os 5 *bits* menos significativos correspondem ao deslocamento na linha de *cache*, os próximos *n bits* correspondem à posição que a linha de *cache* se localiza dentro dos 2^n locais permitidos dentro de um bloco de compressão, e os restantes *27 - n bits* armazenam a informação para indexar a TEBC e obter uma entrada com o endereço para o bloco comprimido na memória de programa.

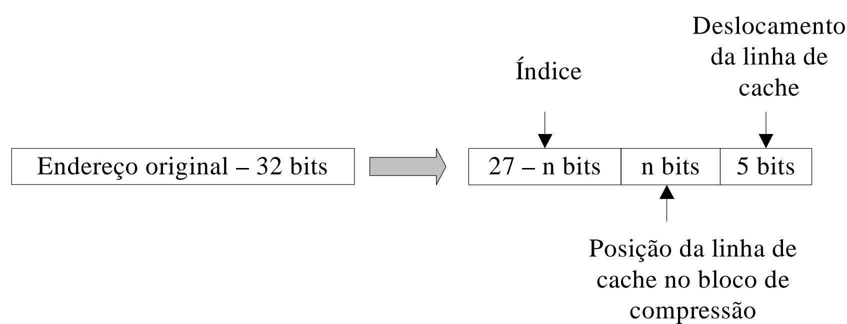

Figura 3.5: Componentes do endereço original

Quando os blocos de compressão são pequenos, o tamanho da TEBC pode ser fator importante de custo extra da taxa de compressão, uma vez que a TEBC deve ser carregada junto com o código comprimido. Com o esquema da TEBC proposto acima, tendo cada

entrada apontando para um bloco comprimido na memória de programa, a TEBC será 12,5% do tamanho do código sem compressão. Como na seqüência dos tamanhos dos blocos de compressão o próximo é o dobro do anterior, a TEBC reduz seu tamanho pela metade neste sentido até chegar ao bloco de 1024 com 0,39%. A alternativa de organização da TEBC com ponteiros para mais de um bloco de compressão como em [4] esbarra na concepção dos blocos de compressão de tamanhos variáveis e no barramento de endereços de 32 *bits* do processador em questão. Como exemplo, usando um bloco de compressão de 1024 *bytes* são precisos 10 *bits* para codificá-lo, e usando uma entrada de TEBC de 64 *bits*, 32 *bits* são para o endereço alvo do primeiro bloco e mais 3 conjuntos de 10 *bits* para indexar os 3 blocos apontados por esta entrada, com uma sobra de 2 *bits*. Neste caso, 3 não é potência de dois o que complica a codificação de uma entrada de TEBC a partir do endereço original se quisermos usar todos os 3 ponteiros. Da mesma forma um bloco de compressão de 32 *bytes* pode apontar para 6 blocos de compressão com sobra de 2 *bits*, o que é também pouco prático.

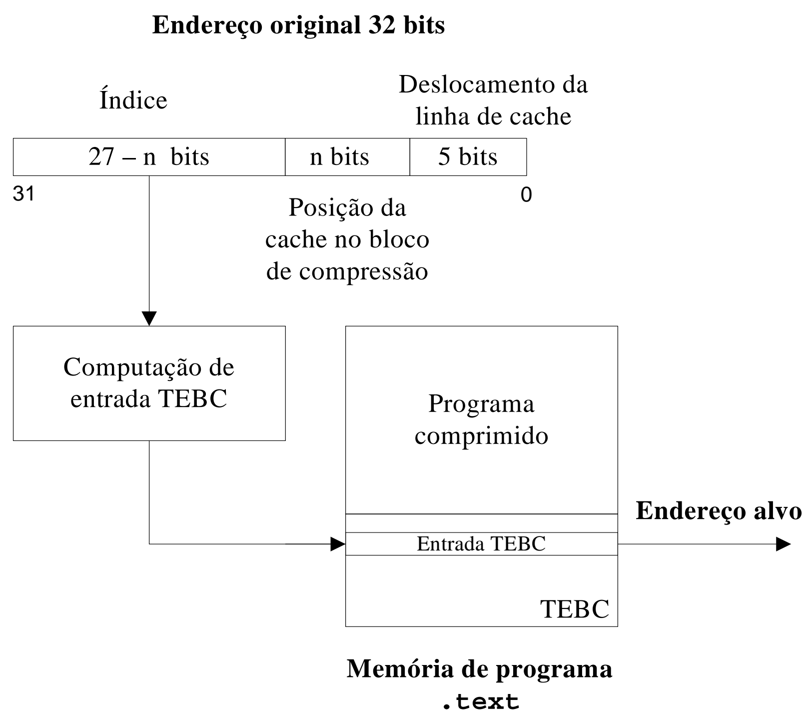

Figura 3.6: Translação de um endereço original em endereço alvo usando a TEBC

A Figura 3.6 mostra como encontrar o endereço alvo do bloco requerido usando a TEBC. O *hardware* de expansão deve ser capaz de identificar o início da TEBC na memória de programa assim como fazer a correspondência entre um endereço original e o endereço correspondente na TEBC, que é o que faz o bloco *Computação de entrada TEBC*. Se o endereçamento for contíguo para a área de memória de programa, esta tarefa é bem simples bastando uma transformação de endereços simples e sem custo a mais em tempo de processamento (ciclos extras). A Figura 3.7 mostra como os endereços originais dentro de um bloco de compressão de 2^{n+5} *bytes* indexam uma entrada TEBC. Vale notar que a TEBC deve ser carregada na memória de programa não importando sua localização, ficando a cargo do projetista de *hardware* da arquitetura escolher o esquema com menor custo de translação do endereço.

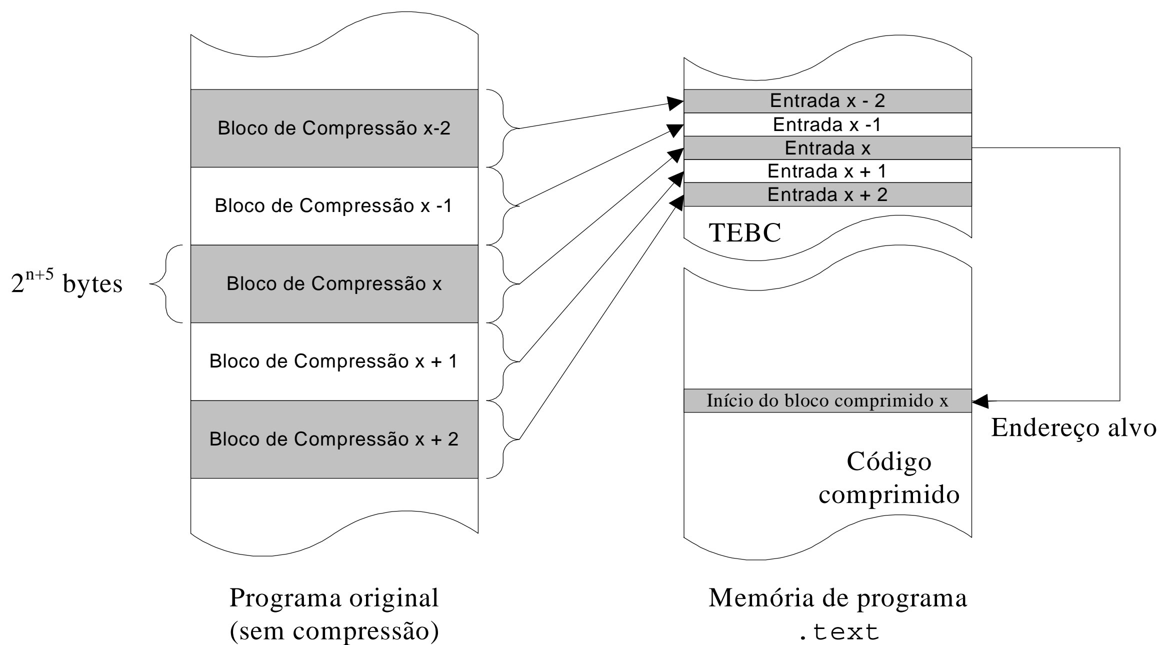

Figura 3.7: Indexação da TEBC

3.4 Arquitetura RISC com Código Comprimido (ARCC)

Na arquitetura do sistema visto na Figura 3.8, a UCP é o núcleo RISC, no nosso caso, o ARM. Para que o núcleo ARM não tenha que ser mudado, a compressão/expansão de código deve ser totalmente transparente para o núcleo. Toda vez que uma requisição de um endereço de uma instrução causar uma falta na *cache*, o Gerenciador de Expansão (GE) deve interceptar a requisição e prover a linha de *cache*.

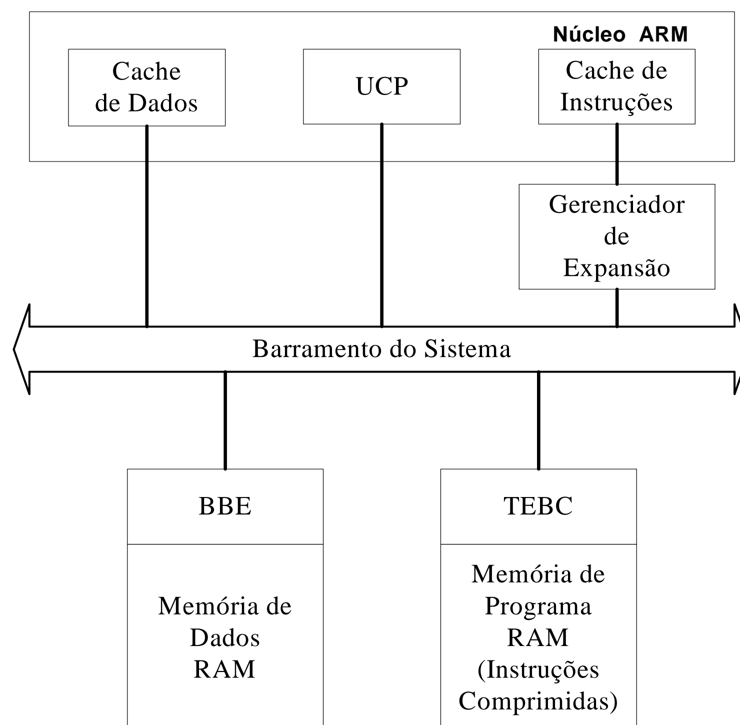

Figura 3.8: ARCC - Arquitetura principal

Como o bloco de compressão pode ser maior que a linha de *cache*, existe um buffer que receberá os blocos expandidos. Este buffer se chama Buffer de Blocos Expandidos (BBE) e pode conter uma quantidade de blocos expandidos equivalente a uma potência de dois. O BBE é equivalente ao DBB proposto em [6] e seu tamanho será variado a fim de investigar os efeitos na arquitetura.

A Figura 3.9 mostra os componentes internos do GE. Dentro dele temos o *Hardware* de Expansão (HE) que equivale a um decodificador Huffman para o algoritmo de compressão

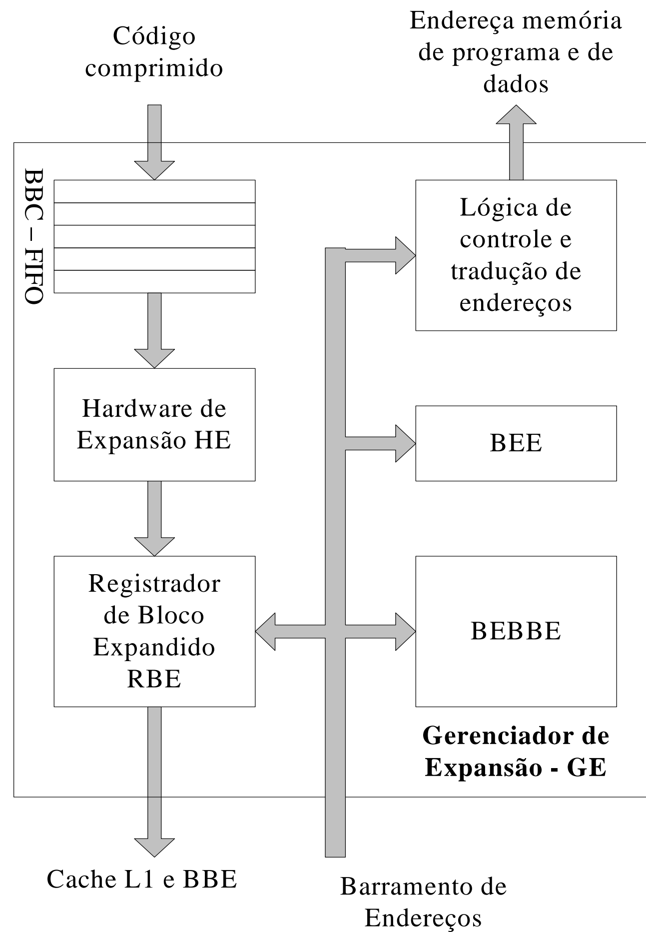

Figura 3.9: Gerenciador de Expansão

escolhido e está entre dois *buffers*. O *Buffer* de Bloco Comprimido (BBC) é uma estrutura FIFO com largura da memória RAM (no nosso caso 32 *bits*) que recebe os *bytes* do bloco comprimido que serão lidos na memória de programa. Sua estrutura lógica é FIFO para que o decodificador Huffman possa consumir os dados vindos do bloco comprimido de acordo com o seu ritmo (*throughput*). Diferentemente de [6] não se sabe *a priori* quantos *bytes* o bloco comprimido tem, sendo o decodificador Huffman responsável pela sinalização do final da leitura na memória quando forem decodificados os *bytes* correspondentes ao tamanho do bloco de compressão. O outro buffer, o Registrador de Bloco Expandido (RBE) tem a função de armazenar o bloco expandido que é saída do HE. O RBE é usado como acesso rápido para faltas na *cache* cujos dados requeridos podem ser encontrados

lá. Este tipo de acesso é mais rápido que na RAM pois o hardware de expansão deve ser construído com unidades estáticas de armazenamento.

A Figura 3.10 mostra os campos do RBE. O campo *Bloco expandido* de 2^{n+5} *bytes* é o local de armazenamento do bloco de compressão na sua forma expandida ou original. O campo *Rótulo* armazena o índice do endereço que desencadeou a expansão (ver Figura 3.5). Para saber se um endereço se encontra no RBE basta verificar se o rótulo é válido (no campo *Válido*) e se coincide com o índice do endereço procurado. No momento de prover uma linha que está no RBE à *cache*, basta selecionar um dos n blocos do campo *Bloco de compressão* através do campo de n *bits* do endereço (ver Figura 3.5).

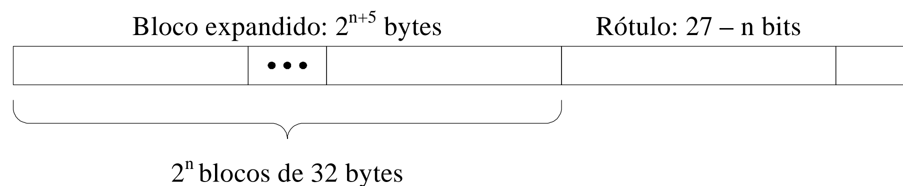

Figura 3.10: Registrador de Bloco Expandido — RBE

O *Buffer* de Entradas de Endereços (BEE), armazena os rótulos dos endereços originais dos blocos que foram expandidos recentemente e junto com estes, o endereço alvo correspondente e é equivalente à CLB proposta em [4]. Ele funciona como uma TLB em memória virtual e o número de entradas será um dos parâmetros da pesquisa. Com um acerto no BEE a TEBC não necessita ser lida para encontrar um endereço alvo, diminuindo de pelo menos um acesso de memória a busca pelo bloco comprimido correspondente. Se o BEE tiver poucas entradas, a busca pelo acerto de um índice de endereço com o campo *Rótulo* na sua entrada pode ser feita em paralelo para todas as entradas, com custo baixo em tempo de execução. A Figura 3.11 ilustra a organização do BEE. Notar que o campo *Válido* indica se a entrada está armazenando dados válidos.

Além do BEE, existe uma estrutura semelhante que é o *Buffer* de Endereços do BBE (BEBBE) que armazena os endereços dos inícios dos blocos (expandidos) no BBE. Ao

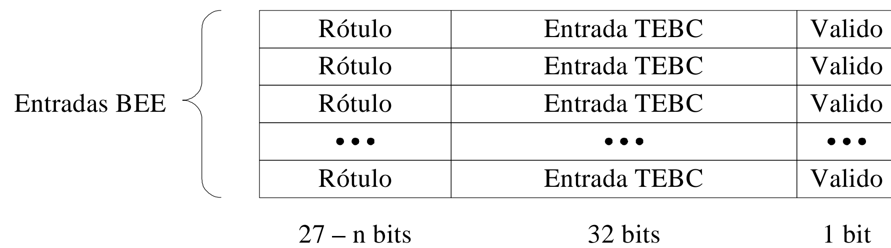

Figura 3.11: Organização do Buffer de Entradas de Endereços — BEE

ser buscada uma instrução, o BEE e o BEBBE são varridos simultaneamente em busca de um acerto de um índice de endereço com o campo *Rótulo*. A Figura 3.12 ilustra a organização do BEBBE. Vale notar que se os blocos de compressão expandidos residentes no BBE estão armazenados com alinhamento de 2^{n+5} *bytes*, ao invés de armazenar um endereço completo de 32 *bits* para cada entrada BEBBE no campo *Endereço de bloco expandido*.

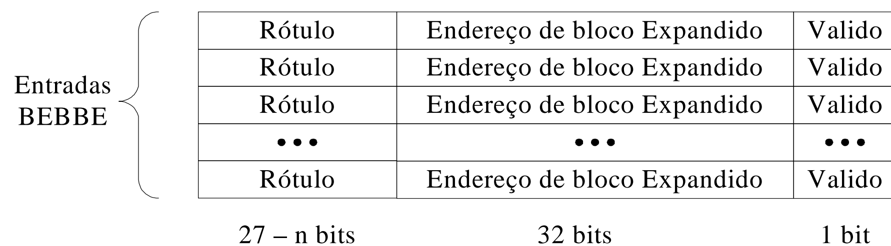

Figura 3.12: Organização do Buffer de Endereços do BBE — BEBBE

Finalmente a *Lógica de controle e tradução de endereços* controla toda a movimentação de dados, as comparações de endereços, o disparo do HE e os acessos à memória segundo o algoritmo abaixo.

1. O endereço da instrução é passado ao GE.

2. O GE verifica se o índice do endereço coincide com o rótulo em RBE. Caso isto ocorra, significa que a linha de *cache* está no RBE e vai para o passo 7. Senão vai para o próximo passo.

3. O GE compara o índice do endereço requerido com os rótulos armazenados em BEE e BEBBE paralelamente. Se houver coincidência com o BEBBE, é porque a linha requerida se encontra no BBE, e basta uma leitura na memória de dados para obter a linha de *cache* requerida e vai para o passo 8. Caso haja coincidência em um dos rótulos do BEE é porque ele já foi utilizado antes e está na memória de programa comprimida. Neste caso o endereço alvo está na entrada do BEE e não há necessidade de ir até a TEBC. Vai para o passo 5. Se não houver acerto do índice com os rótulos de nenhuma das estruturas anteriores vai para o próximo passo.

4. O GE calcula o endereço da entrada da TEBC correspondente ao endereço original e dispara um acesso de leitura à memória de programa recebendo a entrada da TEBC correspondente que contém o endereço alvo.

5. Se forem válidas as informações em RBE, o campo *Bloco expandido* deve ser salvo em algum lugar do BBE que pode ser um *slot* ainda não preenchido (inválido) ou um *slot* substituído por LRU. O GE deve então disparar um acesso de escrita na memória de dados com o tamanho do bloco de compressão. O campo *Rótulo* da entrada do BEBBE que corresponde ao bloco de compressão no RBE tem que ser atualizado com o valor do campo *Rótulo* do RBE e validado, mas antes seu conteúdo deve ser salvo em alguma entrada do BEE (seguindo os mesmos procedimentos de substituição de uma entrada do BEBBE, com a diferença que se houve um acerto no BEE, a entrada substituída deverá ser a do acerto pois o RBE vai ter ao final da execução do algoritmo, seu campo *Rótulo* preenchido com este valor) que deverá também ser validada. Desta forma ficam preservados os registros aos últimos acessos a endereços originais de instruções.

6. O GE dispara um acesso à memória de programa para ler as palavras do bloco comprimido correspondente e vai enfileirando-os no BBC, ao mesmo tempo que dispara o HE que consumirá as palavras segundo seu ritmo de execução e preencherá

o RBE. Ao final da expansão completa de um bloco comprimido, o GE encerra o acesso à memória, limpa o BBC e atualiza o campo *Rótulo* do RBE com o índice do endereço expandido

7. O GE seleciona a linha de *cache* dentro do campo *Bloco expandido* do RBE através do campo de n *bits* do endereço original.

8. O GE retorna a linha de *cache* requerida.

Em cada operação do GE, estão associadas latências ao uso de *hardware*. No Capítulo 4 os custos de cada operação são calculados com base no *hardware* da arquitetura.

Capítulo 4

Método Experimental

Como forma de validar e medir os parâmetros da arquitetura (ARCC) proposta neste trabalho de pesquisa, a simulação por *software* foi escolhida. Apesar de existirem linguagens de descrição de *hardware* bastante acessíveis (como o VHDL), a escolha da plataforma de simulação SimpleScalar [7] [24], é a mais apropriada por uma série de motivos: ter a simulação orientada à execução (executa cada instrução da máquina alvo); conter o conjunto de instruções do processador ARM; ser bastante detalhada a nível arquitetural a ponto de emular o *pipeline* de alguns modelos do ARM; ser modular e bem comentado além de ser um sistema estável e gozar de bastante reconhecimento no meio acadêmico.

A modelagem e simulação por *software* é indicada para que seja feita a validação do desempenho e correção do projeto. No caso da produção industrial, o projetista pode testar sua arquitetura através da construção ou modificação de um programa em horas ou dias ao invés de meses, no caso de se construir o *hardware* real da arquitetura. Isto agiliza o processo de projetar, testar e produzir as peças de silício.

Algumas modificações foram implementadas no SimpleScalar para que a modelagem da ARCC fosse feita. Os *benchmarks* usados para o levantamento das medidas pertinentes pertencem ao pacote Mibench [8], uma coleção de programas representativos para sistemas embutidos. Nas próximas seções as ferramentas de *software* usadas neste trabalho de

pesquisa serão descritas com detalhes assim como as as modificações feitas nas mesmas para dar suporte à arquitetura proposta.

4.1 Ferramentas

Os relatórios detalhados gerados pelo SimpleScalar modificado para abrigar o modelo da ARCC ao rodar os *benchmarks* do pacote MiBench, mostram com detalhes como a arquitetura se comporta quantitativa e qualitativamente. Nas próximas seções serão mostrados detalhes do SimpleScalar e do MiBench, ferramentas de software usadas neste trabalho de pesquisa.

4.1.1 SimpleScalar

O SimpleScalar foi escrito em 1992 como parte do projeto Multiscalar na Universidade de Wisconsin, sob direção de Gurindar Sohi. Em 1995, com a assistência de Doug Burger as ferramentas foram liberadas como distribuição em código aberto e gratuita para aplicações acadêmicas não-comerciais. SimpleScalar LCC mantém agora as ferramentas e disponibiliza-as no sítio de internet `http://www.simplescalar.com`. A versão atual disponível é a 3.0d.

Em 2000 mais de um terço de todos os artigos publicados em conferências de arquitetura de computadores usou as ferramentas do SimpleScalar para avaliar seus projetos [24]. Isto mostra a popularidade que desfruta no meio acadêmico entre pesquisadores e instrutores.

Modelagem

O conjunto de ferramentas do SimpleScalar apresenta uma infra-estrutura para simulação e modelagem arquitetural. As ferramentas podem modelar uma grande variedade de problemas que vão desde um processador simples sem *pipeline* até micro-arquiteturas

detalhadas com escalonamento dinâmico de instruções e hierarquia de memória com vários níveis. Durante a simulação as ferramentas medem as características dinâmicas do modelo de *hardware* e o desempenho do *software* sendo executado nele. Para usuários com necessidades mais específicas, SimpleScalar oferece uma codificação bem documentada que permite extender as ferramentas para ampliar a modelagem das arquiteturas.

Os simuladores do SimpleScalar reproduzem a execução de um programa através de um interpretador que executa todas as instruções do programa na máquina alvo. Os interpretadores aceitam conjuntos de instruções de várias arquiteturas populares como Alpha, PowerPC, x86 e ARM.

Simulador	**Descrição**	**Linhas de Código**	**Velocidade de Simulação**
sim-safe	Simulador simples funcional	320	6 MIPS
sim-fast	Simulador funcional otimizado	780	7 MIPS
sim-profile	Analisador dinâmico de programa	1300	4 MIPS
sim-bpred	Simulador de predição de desvio	1200	5 MIPS
sim-cache	Simulador de *cache* multinível	1400	4 MIPS
sim-fuzz	Gerador de instruções aleatório	2300	2 MIPS
sim-outorder	Modelo microarquitetural detalhado	3900	2 MIPS

Tabela 4.1: Características dos simuladores do SimpleScalar

A Tabela 4.1 lista as características dos simuladores incluídos na versão 3.0 do SimpleScalar. Os simuladores vão desde o *sim-safe*, um simulador que emula somente o conjunto de instruções, até o *sim-outorder*, um modelo microarquitetural detalhado com escalonamento dinâmico, execução especulativa agressiva e um sistema hierárquico de memória com vários níveis. Além disso o sim-outorder inclui arquivos de configuração que torna possível a simulação do pipeline do processador StrongARM SA-1110, o que é bastante conveniente para este trabalho de pesquisa.

Todos os simuladores têm o código pequeno e uma coleção de rotinas que serve para uma ampla gama de tarefas de modelagem pode ser usada nos vários simuladores. Estas tarefas incluem a simulação do conjunto de instruções, emulação de E/S, gerenciamento

de eventos, carregamento da aplicação a ser interpretada e modelagem de componentes micro-arquiteturais básicos como preditores de desvio, filas de instruções e *caches*. Em geral, quanto mais detalhado é o simulador, maior o código e mais lenta a execução da simulação graças ao aumento do processamento por cada instrução simulada. A Figura 4.1 mostra a arquitetura do software do SimpleScalar.

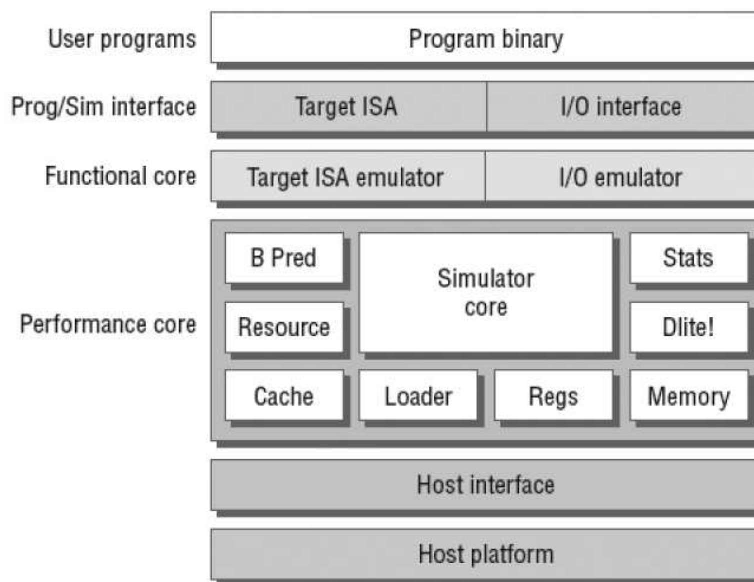

Figura 4.1: Arquitetura do SimpleScalar

Simulação Orientada à Execução

As aplicações são executadas no SimpleScalar usando uma técnica chamada simulação orientada à execução, que reproduz a execução de instruções da máquina simulada, e requer a inclusão de um emulador do conjunto de instruções e um módulo de emulação de E/S.

Uma outra forma popular alternativa é a simulação orientada à *trace* (*trace-driven*). Neste modo, é empregado um fluxo de instruções pré-gravadas para o desenvolvimento do modelo temporal de hardware.

Simulação orientada à execução tem vantagens interessantes comparadas às técnicas

baseadas em rastro. A primeira tem acesso à todo o dado produzido e consumido durante a execução do programa. Estes valores são indispensáveis para o estudo de otimizações em predição de valores, sistemas de memória comprimida e análise dinâmica de energia.

O SimpleScalar inclui interpretadores para o conjunto de instruções do ARM, x86, PPC, e Alpha. No caso do processador ARM, o SimpleScalar emula o conjunto de instruções inteiras da família de processadores ARM7 e extensões FPA de ponto-flutuante. Os interpretadores são escritos numa linguagem de definição que descreve convenientemente como as instruções alteram os registradores e memória. Um pré-processador usa essas definições de máquina para sintetizar os interpretadores, análise de dependência e os geradores de microcódigo que os modelos do SimpleScalar exigem.

O módulo de E/S supre os programas simulados com facilidades de E/S. O SimpleScalar suporta vários módulos de emulação E/S desde chamadas do sistema até a simulação do sistema total. Para uma emulação de chamada de sistema, o simulador executa o módulo de E/S sempre que o programa simulado tenta executar uma chamada de sistema. Neste caso, o sistema emula a chamada traduzindo-a para uma chamada equivalente do sistema operacional hospedeiro e direcionando o simulador para executar a chamada no programa simulado. Isso acontece quando o a aplicação requer uma abertura de arquivo que será traduzida para a chamada equivalente do sistema hospedeiro.

A versão SimpleScalar/ARM inclui um emulador de E/S para o *IPaq*, um computador de mão popular fabricado pela Compaq. Este emulador é tão detalhado a ponto de ser possível executar o *boot* do sistema operacional Linux/ARM.

Módulos

O núcleo do simulador (*simulator core* na Figura 4.1) define o laço principal, que executa uma iteração para cada instrução do programa até o fim das instruções. Para um modelo temporal o laço principal deve cuidar da progressão do tempo de execução medidos em ciclos de relógio para este modelo. Para determinar o desempenho relativo

dos programas, o modelo compara o número total de ciclos para completar a execução. O valor definido de tempo para uma instrução é de um ciclo de relógio, para *loads* e *stores* (operações na memória) é de 2 ciclos de relógio e uma falta na *cache* corresponde a 10 ciclos de relógio.

O módulo `cache.c` que vem no SimpleScalar implementa as *caches*. O módulo de *cache* usa uma tabela *hash* para armazenar os blocos que ela contém. Se o acesso a um endereço de memória combina com uma entrada da tabela, o acesso retorna a latência de acerto. Se o acesso a um endereço não combina com uma entrada da tabela, então o sistema chama o manipulador de faltas na *cache* que retorna o número de ciclos de relógio requerido para suprir uma falta de *cache*. O construtor do modelo especifica o manipulador de faltas que pode ser outro módulo de *cache* ou memória DRAM. O módulo de *cache* não retorna o valor acessado na *cache* porque este valor não tem efeito na latência de acesso da *cache*.

Além dos componentes padrão, o SimpleScalar tem ainda uma variedade de módulos de ajuda que implementam ferramentas úteis que alguns módulos requerem. Estes módulos incluem um depurador, carregador de programa, leitor de tabela de símbolos, processador de linha de comando e pacote estatístico.

4.1.2 MiBench

O desempenho de aplicações executadas na arquitetura projetada é uma parte crítica do processo de pesquisa e desenvolvimento [25]. Aplicações representativas (*benchmarks*) na grande variedade de aplicações disponíveis são necessárias para validar e testar a arquitetura proposta pelo projetista. Uma grande variedade de *benchmarks* tem sido proposta incluindo Dhystone, Whetstone, LINPACK, CPU2, MediaBench, SSBA, TPC e muitos outros. Muitos destes *benchmarks* são usados em áreas específicas da computação. O Dhrystone serve para verificar o desempenho com inteiros; Whetstone e CPU2 são ambos aplicativos com intensiva operação de ponto flutuante; o LINPACK se refere a

computação vetorial; o MediaBench para aplicações multimídia; o SSBA uma coleção de rotinas de processamento de dados e o TPC verifica o desempenho de sistemas de transações multi-usuário, como por exemplo as transações no sistema de um grande banco.

O *benchmark* mais amplamente usado é o *Standard Performance Evaluation Corporation* (SPEC). A versão atual é a SPEC2000 e as aplicações se caracterizam por serem feitas para computadores de propósitos gerais e com categorias para dados em formato inteiro e ponto-flutuante.

Apesar dessa classe de máquinas ser o foco da comunidade de arquitetura de computadores, menos de 1% dos processadores são usados neste segmento de mercado. A grande maioria (mais de 99%) é empregada em sistemas embutidos [26]. Embora muitos deles sejam simples microcontroladores de baixo custo, a soma das vendas de todos eles é quase a metade do valor total de todos os processadores. Além disso o mercado de processadores para sistemas embutidos tem crescido rapidamente.

É difícil caracterizar o domínio dos sistemas embutidos quanto às aplicações. Um controlador de uma máquina de lavar pode ter sua programação e dados bem simples mas um aparelho de telefonia celular por exemplo, pode conter aplicações a nível de desktop combinada com comunicação *bluetooth*. Um conjunto de *benchmarks* para sistemas embutidos deve portanto enfatizar a diversidade.

O exemplo de um esforço no sentido de delimitar os espaços para os *benchmarks* nos sistemas embutidos é o *Embedded Microprocessor Benchmark Consortium* (EEMBC). O pacote de aplicações produzido por eles traz subconjuntos para cada tipo diverso de aplicação, mostrando que é realmente difícil caracterizar com uma só aplicação uma determinada área de atuação. Uma desvantagem desse pacote é o alto custo para o meio acadêmico, assim como o é para o SPEC.

Aplicações

O MiBench é um pacote de benchmarks cujos códigos fonte são gratuitos. Com algumas pequenas modificações é possível torná-lo portável para diversas plataformas, o que alguns mantenedores o fazem, deixando disponível o código compilado em sítios da Internet. Desta forma pode-se encontrar o código pronto para ser executado no SimpleScalar/ARM. São disponibilizados também conjuntos de dados de entrada grandes e pequenos. Um conjunto pequeno de dados de entradas representa uma carga leve para as aplicações do *benchmark* enquanto o conjunto de dados grande provê uma carga maior e mais parecida com a aplicações reais.

O MiBench é dividido em seis conjuntos (assim como o EEMBC) em que cada conjunto é responsável por caracterizar uma área das aplicações representativas para os sistemas embutidos. As seis categorias são: Automotiva e controle industrial; dispositivos de consumo; automação de escritório; rede; segurança e telecomunicações. A seguir com detalhes as seis categorias.

Automotiva e Controle Industrial: Nesta categoria a intenção é demonstrar o uso do processadores embutidos em sistemas de controle embutidos. Esses processadores requerem desempenho em matemática básica, manipulação de *bits*, entrada e saída de dados e organização simples de dados. Aplicações típicas são controladores de *air bag*, monitores de desempenho de mecanismos e sistemas de sensores. Abaixo são explicados os aplicativos para esta categoria.

basicmath: Testa cálculos matemáticos simples, que freqüentemente não encontram suporte em *hardware* para tal. Por exemplo, resolução de função cúbica, raiz quadrada de inteiro e conversão de ângulos de grau para radianos, todos são cálculos necessários para determinar velocidade na estrada ou outros valores vetoriais.

bitcount: O algoritmo testa as habilidades de manipulação que o processador tem de contar o número de *bits* em um vetor de inteiros. Ele o faz usando cinco métodos incluindo

um contador de 1 *bit* por laço otimizado, contador recursivo de *bits* em *nibbles*, contador de *bit* não-recursivo de *bits* em *nibbles* usando uma tabela e deslocamento e contagem de *bits*. Como dados de entrada temos um vetor de inteiros com número igual de 1's e 0's.

qsort: Ordena um longo vetor de strings em ordem ascendente usando o algoritmo *quick sort*. Ordenação de informação é importante em eventos em que há prioridades, saídas podem ser melhor interpretadas, dados pode ser organizados e o tempo total de execução dos programas reduzidos.

susan: É um pacote de reconhecimento de imagem. Ele foi desenvolvido para reconhecer extremidades e fronteiras em imagens de Ressonância Magnética do cérebro. É típico de um programa real que pode ser empregado em uma aplicação baseada em compromisso com qualidade de visão.

Dispositivos de Consumo: Estes *benchmarks* têm a função de representar os vários dispositivos que cresceram em popularidade nos últimos anos como *scanners*, câmeras digitais e *Personal Digital Assistants* (PDAs). Esta categoria trata principalmente de aplicações multimídia com algoritmos de codificação/decodificação de jpg, conversão de formato de cores de imagens, difusão de imagem, redução da paleta de cores, codificação/decodificação MP3 e composição de HTML. Todos os benchmarks de imagem usam imagens grandes e pequenas como entrada de dados.

codificação/decodificação jpg: JPG é um formato de imagem padrão com perda de qualidade. É incluída no MiBench porque é um algoritmo representativo de compressão e expansão de imagem e é usado freqüentemente para exibir imagens embutidas em documentos.

tiff2bw: converte uma imagem colorida no formato TIFF em preto e branco.

tiff2rgba: Converte uma imagem colorida no formato TIFF em TIFF RGB.

tiffdither: faz a difusão de um bitmap em preto e branco no formato TIFF para diminuir a resolução e tamanho da imagem em prejuízo da clareza.

tiffmedian: Diminui a paleta de cores de uma imagem usando médias das cores na paleta corrente.

lame: É o codificador GPL de MP3 que suporta codificação constante, média e variável da taxa de *bits*.

mad: É um codificador de MPEG de alta qualidade.

typeset: É uma ferramenta de composição geral que tem como base um compositor de HTML.

Automação de Escritório: Esta categoria compõe-se basicamente de algoritmos de manipulação de textos para representar as máquinas comuns de escritório como impressoras, fax e processadores de texto. Um PDA que se inclui na categoria anterior pode também ser incluído aqui pois existem aplicações de processamento de texto.

ghostscript: É um interpretador de linguagem *postscript* sem interface gráfica. É incluído para representar a crescente importância da capacidade de trabalhar com *postscript* dos dispositivos como impressoras.

stringsearch: Procura por palavras em frases com algoritmo sem sensibilidade à caixa.

ispell: É um corretor ortográfico rápido similar ao *spell* do Unix porém mais rápido. Suporta sugestões de correção e outras línguas que não o Inglês.

rsynth: É um sintetizador de texto para voz que inclui várias partes de código de domínio público em um só programa.

sphinx: Decodificador de reconhecimento de fala que opera em segmentos de largura finita.

Rede: A categoria de rede representa processadores embutidos em dispositivos de rede como *switches* e roteadores. O trabalho feito por estes processadores inclui cálculo de menor caminho, busca em árvore e tabelas.

dijkstra: Constrói um grafo grande em representação matricial e então calcula o menor

caminho entre dois nós aplicando repetidamente o algoritmo de Dijkstra. Este é um algoritmo bastante conhecido e completa em $O(n^2)$.

patricia: Uma árvore Patricia é uma estrutura usada no lugar de árvores completas esparsas. Ramos com uma só folha são concatenados para cima na árvore. Freqüentemente as árvores Patricia são usadas para representar tabelas de roteamento em aplicações de rede.

Segurança: Segurança está crescendo em importância na medida em que o comércio virtual pela internet aumenta em popularidade. As aplicações para esta categoria incluem algoritmos de criptografia, decriptografia e *hashing* de dados.

criptografia/decriptografia blowfish: Blowfish é um algoritmo criptográfico de chave simétrica desenvolvido por Bruce Schneider. Consiste de um cifrador em blocos de 64 *bits* com chaves de tamanho variável (de 32 até 448 *bits*) e é ideal para criptografia doméstica.

sha: É o algoritmo seguro de *hash* que produz uma mensagem de 160 *bits* agrupados para uma determinada entrada. É sempre usada em troca segura ou chaves criptográficas e para gerar assinatura digital.

criptografia/decriptografia rijndael: O Algoritmo de Rijndael foi selecionado pelo *National Institute of Standards and Technologies* como *Advanced Encryption Standard* (AES). É um algoritmo criptográfico que pode operar com chaves e blocos de 128, 192 e 256 *bits*.

assinatura/verificação pgp: Prety Good Privacy (PGP) é um algoritmo de criptografia com uma chave pública desenvolvido por Phil Zimmerman. Permite a comunicação segura de pessoas que nunca interagiram usando assinatura digital e sistema criptográfico RAS de chave pública.

Telecomunicações: Com o crescimento da internet, vários dispositivos portáteis estão incluindo comunicação sem-fio. Esta categoria gera carga para estas características e os algoritmos consistem de codificação e decodificação de voz, análise de freqüências e

checksum.

FFT/IFT: Executa a transformada rápida de Fourier (Fast Fourier Transform) e a transformação inversa em um vetor de dados. A transformada de Fourier é usada em processamento de sinal digital para encontrar as freqüências contidas em um sinal de entrada.

codificação/decodificação GSM: *Global Standard for Mobile* (GSM) é um padrão para codificação e decodificação na Europa e em vários países. Usa uma combinação de acesso múltiplo por divisão de tempo e freqüência (TDMA/FDMA) para codificar e decodificar *streams* de dados.

codificação/decodificação ADPCM: *Adaptive Differential Pulse Code Modulation* (ADPCM) é uma variação do PCM. Uma implementação freqüente usa uma amostragem PCM de 16 *bits* e a transforma em amostragem de 4 *bits* com uma compressão de 4:1.

CRC32: Executa uma CRC (*Cyclic Redundancy Check*) de 32 *bits* em um arquivo e é usado para detectar erros de transmissão de dados.

4.2 Simulação da ARCC

Como forma de validar, testar e investigar o desempenho da ARCC, modificações foram feitas no código fonte do SimpleScalar para incluir os detalhes arquiteturais a serem simulados. A seguir as descrições da compressão e expansão de código no SimpleScalar.

4.2.1 Compressão

A compressão da aplicação é feita no código executável. Para um sistema em *hardware* da ARCC operando com todas as funcionalidades previstas, deve-se incluir uma ferramenta que gere o código comprimido tendo como entrada o código executável para o processador ARM como na Figura 3.1. O código comprimido será consumido pelo hardware de expansão da ARCC. Na simulação é mais interessante incluir o código de

compressão diretamente nos arquivos fonte do SimpleScalar pois o acesso à memória da máquina-alvo é imediato e evita-se o trabalho duplicado de criação e identificação de um novo formato de arquivo.

O arquivo `loader.c` pertencente ao pacote de arquivos fonte do SimpleScalar, contém a função `ld_load_prog` que carrega as informações contidas no código executável da aplicação nas estruturas alocadas em memória.

O formato do arquivo executável é o *Executable and Linking Format* (ELF) [27], um formato bastante conhecido e documentado. A Figura 4.2 mostra a organização de um arquivo ELF executável, que inclui um cabeçalho ELF, contendo a identificação e as informações de como o arquivo está organizado; uma tabela de cabeçalhos de programa, que se estiver presente indica ao sistema operacional como criar uma imagem do processo; seções, onde estão o *.text*, *.data*, *.bss*, etc; e a tabela de cabeçalhos das seções onde ficam as informações, por exemplo, de tamanho e localização de cada seção.

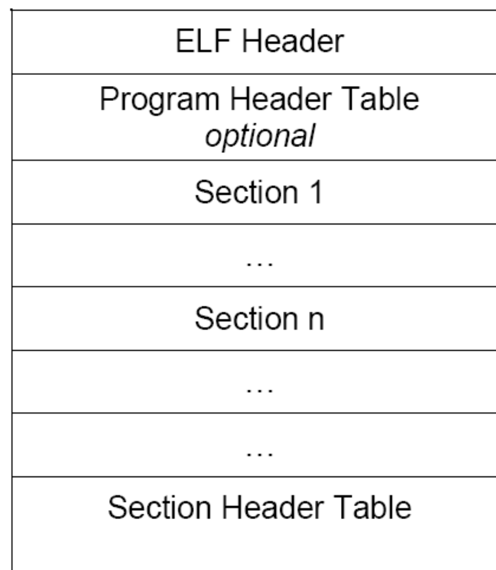

Figura 4.2: Formato de um arquivo executável ELF

Inicialmente, o carregador verifica se o arquivo preenche os requisitos de um arquivo ARM executável, procurando no cabeçalho ELF as indicações. Em caso afirmativo, entra

em uma rotina em que todas as seções são percorridas e uma estrutura de memória mem é preenchida com todas as informações do arquivo executável. É neste momento que os trechos de arquivo contendo código são levados à esta estrutura e seu início e tamanho são indicados pelos endereços relativos contidos respectivamente nas variáveis globais ld_text_base e ld_text_size que são inteiros de 32 *bits* — unsigned int renomeado para o tipo md_addr — que é da mesma largura dos endereços do processador ARM.

O arquivo Makefile foi alterado para incluir os arquivos compressao.c e compressao.h. O primeiro contém as rotinas de compressão e expansão das instruções. O SimpleScalar utiliza macros para a leitura/escrita de posições de memória da máquina-alvo, definidas no arquivo memory.h. A macro MEM_READ_BYTE(MEM, ADDR), que retorna um *byte* da memória, foi usada nas rotinas de compressão, já que os endereços de memória contendo instruções da máquina-alvo são conhecidos. A função de compressão segundo o algoritmo de Huffman, cria o código comprimido e a TEBC, armazenando-os em estruturas acessíveis. O cálculo da *Taxa de Compressão* (TC) é:

$$TC = \frac{\text{Tam. da Parte Executável Comprimida + Tam. da TEBC}}{\text{Tam. da Parte Executável Original}}$$

A Figura 4.3 mostra inserções de código referentes à compressão/expansão no arquivo sim-outorder.c. sim_main é a função chamada pelo módulo main.c indicando o início da simulação. A função main é a mesma para todos os módulos simuladores do Simplescalar, mas chama o simulador de um módulo específico a partir da função sim_main. A função comprime_código chama as rotinas que lêem a configuração de *hardware* da ARCC e comprimem a parte que contém instruções no arquivo. Outras inserções de código C foram necessárias em mais locais dentro e fora de sim-outorder.c para a criação de um código "médio" de Huffman como explicado na Seção 3.3.1. Um programa utilitário foi criado para combinar as freqüências de cada programa individual. A função inicializa_expansao inicializa as variáveis globais referentes à expansão e cria

espaço na memória (`malloc`) para as estruturas em *software* correspondentes ao *hardware* da ARCC.

```
void sim_main(void) {
  int setPC;

  comprime_codigo (mem);
  inicializa_expansao ();

  ...
```

Figura 4.3: Inserção de código de na função sim_main no arquivo sim-outorder.c

4.2.2 Expansão

O processo e gerência da expansão das instruções é feita quando um endereço requerido na *cache* de instruções L1 não é encontrado. Isto faz com que o algoritmo apresentado na Seção 3.4 seja executado. A simulação deste algoritmo é feita através das funções localizadas em `compressão.c`. Em `sim-outorder.c` existe a função `il1_access_fn` que é chamada quando há uma falta na *cache* de instruções L1 e retorna a latência da operação de preenchimento da linha de *cache*. Esta função é alterada e passa a chamar a rotina de cálculo da latência da operação. A Figura 4.4 mostra a função com a alteração chamando a função return `calcula_latencia_expansao` ao invés de `mem_access_latency`.

O *Desempenho Relativo* (DR) é a razão entre o tempo de execução da aplicação comprimida e o tempo de execução da aplicação original sem compressão:

$$DR = \frac{\text{Num. Ciclos de Execução da Aplicação Comprimida}}{\text{Num. Ciclos de Execução da Aplicação Original}}$$

Esta medida dá a exata noção de quanto a execução da arquitetura proposta variou em relação à execução sem o esquema de compressão.

Configuração de Parâmetros

Na simulação da ARCC, vários parâmetros podem ser mudados através de um arquivo de configuração implementado para a simulação da ARCC denominado `config_compressao.txt`. Ele deve ficar no diretório dos programas executáveis do Simplescalar. Dentre os parâmetros que podem ser alterados incluem-se: O tamanho do bloco de compressão, os retardos associadas às operações executadas pelo *hardware* de expansão, o número de entradas do BEE e o tamanho da BBE.

Nas simulações executadas, os blocos de compressão poderão assumir os valores de 32, 64, 128, 256, 512 e 1024 *bytes* como já explicado anteriormente. O BBE assumirá o valor base de 4096 *bytes* e o número de entradas do BEE será inicialmente de 32. Estes parâmetros serão variados em algumas simulações e os efeitos investigados no Capítulo 5.

Retardos do Hardware de Expansão (*Delay*)

Sabendo-se que nenhum *hardware* é instantâneo, podemos estimar os retardos de cada parte da ARCC para incluí-los nas simulações. Vamos supor que o *hardware* do GE tem a mesma tecnologia de integração do núcleo ARM.

Um acerto no RBE (Figura 3.10) significa uma comparação positiva do índice do endereço com o campo *Rótulo* do RBE, isto pode ser feito sem custo adicional pois a lógica é simples e rápida. A escolha da posição da linha de *cache* no bloco expandido em RBE, é feita através de um multiplexador (lógica combinacional rápida e sem custo extra) e o custo para se fornecer uma linha de *cache* através do RBE será de 1 ciclo, já que as estruturas de armazenamento são todas estáticas e a transferência se dá dentro do mesmo CI.

Um acerto no BEE ou BEBBE (ver Figuras 3.12 e 3.11) já não é tão rápido pois o índice do endereço será comparado com os campos *Rótulo* destas estruturas. Um acerto na TLB do SA-1110 no Simplescalar corresponde a um ciclo de máquina. Para um número

(total) de entradas de até 64, será assumido um valor de 2 ciclos de retardo e para valores maiores, 3 ciclos. Como os rótulos do RBE, BBEE e BEE são buscados em paralelo, não há custo adicional numa falta no RBE.

Numa falta no BEE e BEBBE ao mesmo tempo, computa-se o retardo de um acerto mais o retardo de uma translação de endereço original em endereço da TEBC. Dependendo das características do endereçamento da área de programa, isto pode ser uma operação simples se o endereçamento for contíguo (ver Seção 3.4) ou mais complicada, tendo o GE que calcular a translação para espaços não contíguos de endereçamento. O Simplescalar cria o espaço de endereçamento virtual contíguo, o que faz com que 1 ciclo de processador seja suficiente para caracterizar o retardo de uma translação.

Foi escolhida uma memória SDRAM com largura de dados de 4 *bytes* capaz de prover o primeiros grupos de 4 *bytes* em 6 ciclos e os grupos restantes de 4 *bytes* em 1 ciclo, portanto para ler uma entrada da TEBC o retardo é de 6 ciclos (desde que a memória esteja pronta), e não há retardo para a computação do endereço alvo pois a TEBC não precisa ser decodificada.

Os decodificadores Huffman podem ser classificados em dois grupos quanto a sincronização: Síncronos — que necessitam de um *clock* geral — e assíncronos. A grande maioria das implementações serve à aplicações de vídeo digital como a decodificação de MPEG-2. Os síncronos podem ser ainda classificados de acordo com a taxa constante de entrada ou saída. Os decodificadores com taxa de entrada constante têm a limitação de só poderem processar 1 *bit* por ciclo. Isto é problemático pois para alcançar uma boa taxa de decodificação, o *clock* tem que ser alto. Os decodificadores com taxa de saída constante processam um grupo de *bits* que corresponde ao máximo que um código pode ter em um ciclo de máquina. No nosso caso um código não pode ter mais que 16 *bits*, isto significa que a decodificação de um *byte* se faz carregando os primeiros 16 *bits* de entrada (codificados) no decodificador que reconhece a seqüência, traduz o primeiro byte, elimina o código de *bits*, e os *bits* restantes são deslocados (*shift*). Na próxima iteração

ele recebe os próximos 16 *bits* aloja-os convenientemente no seu registrador, e procede como no primeiro passo. Ele termina seu trabalho quando o número requerido de *bytes* decodificados termina. Se cada operação é realizada na subida e descida de clock (como propõe [4] em seu trabalho) leva-se $m \div 2$ ciclos para completar a decodificação onde m é o tamanho em *bytes* do bloco de compressão. Desta forma o tempo de decodificação independe do tamanho do bloco no estado comprimido. Benes *et alii* em [23] propõe um decodificador Huffman assíncrono com alto *throughput*, trabalhando a taxas de entrada e saída variáveis e o aperfeiçoa em [28].

Nas simulações deste trabalho, há duas opções para o funcionamento do decodificador Huffman que serão investigadas: A primeira é o decodificador síncrono com taxa de saída constante e com latência de $m \div 2$ ciclos para a decodificação de qualquer bloco comprimido, sendo m o número de *bytes* do bloco de compressão. A segunda é o decodificador assíncrono capaz de processar 4 *bytes* de entrada (largura da memória) por ciclo. Neste caso ele deve ser rápido no processamento de códigos com poucos *bits*. Este é o caso ideal para esta arquitetura e a latência da operação de decodificação será de $p \div 4$ ciclos (arredondando para o próximo inteiro), onde p é o tamanho do bloco de compressão no seu estado comprimido. Em ambos os casos deve-se contabilizar o tempo para o primeiro grupo chegar da memória.

A organização FIFO do BBC justifica-se pelo funcionamento dos decodificadores Huffman. Para as simulações, supõe-se que seu tamanho seja grande suficiente para que nunca fique cheio totalmente e tenha que parar a leitura da memória para recomeçar mais tarde.

```
il1_access_fn(enum mem_cmd cmd,     /* access cmd, Read or Write */
              md_addr_t baddr,      /* block address to access */
              int bsize,        /* size of block to access */
              struct cache_blk_t *blk,  /* ptr to block in upper level */
              tick_t now)       /* time of access */
{
  unsigned int lat;

  if (cache_il2)
    {
      /* access next level of inst cache hierarchy */
      lat = cache_access(cache_il2, cmd, baddr, NULL, bsize,
             /* now */now, /* pudata */NULL, /* repl addr */NULL);
      if (cmd == Read)
        return lat;
      else
        panic("writes to instruction memory not supported");
    }
  else
    {
      /* access main memory */
      if (cmd == Read)
        /*return mem_access_latency(bsize);*/

       return calcula_latencia_expansao (baddr, sim_cycle,
       &mem_ready, mem_lat[0], mem_lat[1], mem_pipelined);

      else
        panic("writes to instruction memory not supported");
    }
}
```

Figura 4.4: Função il1_access_fn modificada no arquivo sim-outorder.c

Capítulo 5

Análise de Resultados

As aplicações do pacote de *benchmarks* MiBench, usadas nas simulações foram escolhidas de forma a representar todas as categorias (ver Seção 4.1.2) e estão descritas na Tabela 5.1. Um outro critério de seleção foi com relação às taxas de faltas na *cache* de instruções L1. Em geral os *benchmarks* exibem uma taxa bem baixa de faltas em IL1, sendo próximo de 0% na grande maioria das vezes. Com a execução prévia de todos os programas a partir do Simplescalar, pode-se escolher as que apresentam variações em taxas de falta em IL1 que sejam representativas do conjunto de aplicações. Mais tarde quando se falar em expansão, estes dados serão apresentados.

Benchmark	**Tamanho do Código (*bytes*)**
bitcount	260124
quicksort	239244
cjpeg	298916
djpeg	311108
lame	1659868
stringsearch	188484
patricia	243756
blowfish (enc)	190900
blowfish (dec)	190900
crc	186884

Tabela 5.1: Aplicações do pacote MiBench usadas nas simulações

Deve-se notar que *cjpeg* e *djpeg* referem-se ao mesmo tipo de aplicativo: *jpeg*, porém são códigos diferentes. Já o *blowfish (enc)* e *blowfish (dec)* se referem ao mesmo programa sendo a codificação e decodificação caracterizadas a partir de parâmetros de entrada, fazendo com que a análise de compressão dos aplicativos *blowfish* seja a mesma para os dois.

Nas próximas seções serão mostrados os resultados das simulações para a compressão dos dados, funcionamento da ARCC na expansão de dados e o relacionamento entre compressão e expansão.

5.1 Compressão

Para ilustrar a eficiência em utilizar o algoritmo de Huffman modificado, explicado na Seção 3.3.1, foram incluídas as aplicações *ghostscript*, *dijkstra*, *rijndael* e *fft*, às da Tabela 5.1 para gerar o *código médio* de Huffman. A inclusão de mais aplicações não se presta a melhorar a taxa de compressão individual, pelo contrário, a possibilidade de piorar é maior. A inclusão serve para mostrar que mesmo para uma gama maior de amostras, que é o que se faria para uma implementação em *hardware* da ARCC, o *código médio* traz excelentes resultados como mostra a Figura 5.1 que compara a taxa de compressão do código Huffman tradicional com o código médio proposto incluindo ou não o tamanho da TEBC. O tamanho de bloco de compressão utilizado nesta comparação é de 32 *bytes*, sendo o que traz a pior taxa de compressão devido ao maior número de *bits* perdidos por alinhamento de *bytes*, já que são necessários mais blocos de compressão para determinar a aplicação completamente, comparando-se com blocos de compressão maiores. A TC média para os programas foi de 76,52% e 89,02% para o código de Huffman tradicional sem e com TEBC respectivamente e 76,85% e 89,35% para o *código médio* sem e com TEBC respectivamente e sem grandes variações individuais, o que mostra um resultado excelente para esta proposta. Notar que a taxa de compressão sem a TEBC é

simplesmente o quociente entre o tamanho do código comprimido pelo tamanho do código sem compressão.

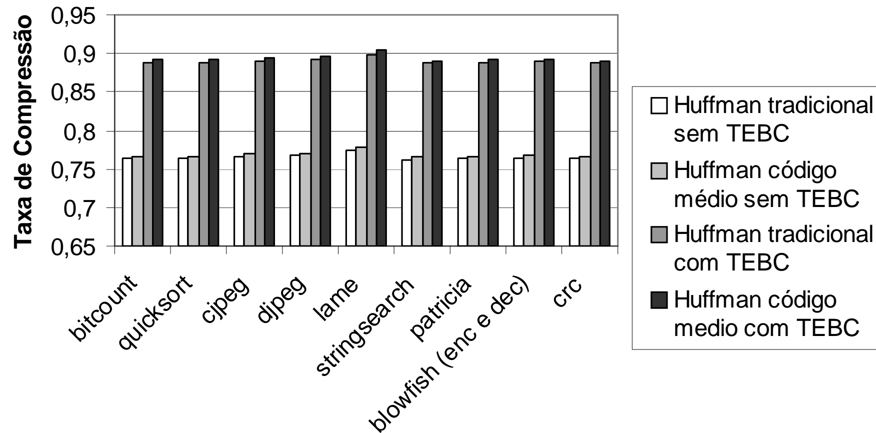

Figura 5.1: Comparação entre o código Huffman tradicional e o modificado com código médio para um bloco de compressão de 32 *bytes*.

Usando a codificação de Huffman proposta com o código médio, a Tabela 5.2 mostra como fica a taxa de compressão para os vários *benchmarks* em função da variação do tamanho do bloco de compressão. Se nada for dito, a taxa de compressão inclui o tamanho da TEBC.

A taxa de compressão sem a TEBC deste trabalho que é em média de 76,85% para blocos de compressão de 32 *bytes* e de 75,53% para blocos de 1024 *bytes* é compatível com taxas de outros trabalhos, em especial o citado em [28] que diz que o processador MIPS pode atingir uma taxa de compressão de 76% para aplicações típicas.

O tamanho do bloco de compressão atua nas duas causas que influenciam diretamente a taxa de compressão: O tamanho da TEBC e o número de *bits* perdidos por alinhamento de *bytes* (em menor escala). Ao aumentar o tamanho do bloco de compressão, diminuem-se as entradas da TEBC e conseqüentemente seu tamanho e também diminui o número de blocos e a possibilidade de *bits* perdidos em alinhamentos de *bytes*. O algoritmo de Huffman não sofre influência do tamanho do bloco de compressão. Com o código

	Taxa de Compressão					
bitcount	0,8918	0,8223	0,7878	0,7704	0,7617	0,7574
quicksort	0,8916	0,8223	0,7877	0,7703	0,7617	0,7573
cjpeg	0,8941	0,8248	0,7901	0,7727	0,7641	0,7598
djpeg	0,8955	0,8263	0,7914	0,7742	0,7655	0,7611
lame	0,9037	0,8343	0,7997	0,7824	0,7737	0,7694
stringsearch	0,8906	0,8215	0,7866	0,7693	0,7607	0,7563
patricia	0,8917	0,8224	0,7877	0,7704	0,7617	0,7574
blowfish (enc e dec)	0,8922	0,8229	0,7882	0,7709	0,7622	0,7579
crc	0,8906	0,8213	0,7865	0,7693	0,7607	0,7563
	32	**64**	**128**	**256**	**512**	**1024**
	Tamanho do bloco de compressão em *bytes*					

Tabela 5.2: Taxa de compressão para os diversos *benchmarks* em função do tamanho do bloco de compressão

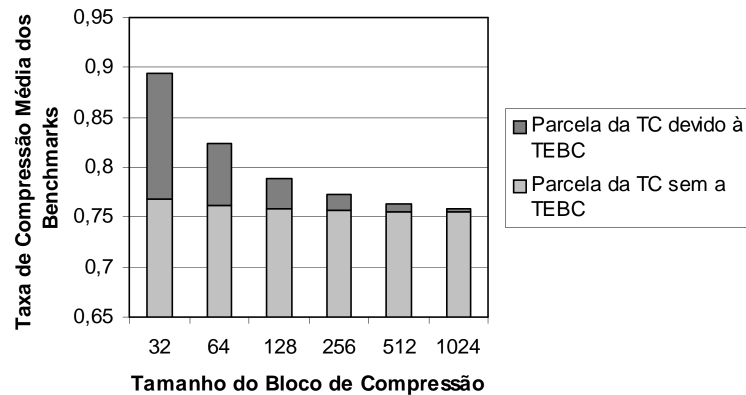

Figura 5.2: Influência da TEBC na taxa de compressão

para cada *byte* estabelecido *a priori*, a compressão de uma seqüência de *bytes* vai ser a mesma se estiver em um mesmo bloco ou em blocos diferentes (salvo o número de *bits* perdidos em alinhamento que foi tratado acima). A Figura 5.2 ilustra a média das taxas de compressão das aplicações para cada tamanho de bloco de compressão simulado, destacando a influência do tamanho da TEBC.

5.2 Expansão

A Tabela 5.3 mostra o número de ciclos de máquina e o número de instruções executadas pelas aplicações escolhidas do MiBench. Notar que agora o aplicativo *blowfish* terá valores diferentes para a codificação e decodificação.

Benchmark	**Número de Ciclos de máquina**	**Número de Instruções executadas**
bitcount	64604895	76229145
quicksort	65042444	85245509
cjpeg	138893468	157928215
djpeg	30746065	157928215
lame	164039305	168015602
stringsearch	5297382	6495757
patricia	231118842	177862982
blowfish (enc)	91658817	125803039
blowfish (dec)	91339474	125795579
crc	31153	26412

Tabela 5.3: Execução dos *benchmarks* sem a utilização da ARCC

As simulações foram feitas a partir de uma arquitetura base descrita na Seção 4.2.2 incluindo o decodificador Huffman com taxa de saída constante. A partir disso, podemos variar os parâmetros para avaliar o comportamento da expansão, medido pelo desempenho relativo das aplicações com relação à execução sem o esquema de compressão da ARCC. As próximas seções mostram cada perfil de *hardware* da ARCC e as variações de parâmetros.

5.2.1 Perfil 1: Arquitetura Base

A Tabela 5.4 mostra a configuração base da ARCC, e o parâmetro alterado será o tamanho do bloco de compressão.

Para este perfil, foram feitas as simulações com os benckmarks escolhidos e a Tabela 5.5 mostra o desempenho relativo para cada caso.

Observando os números, percebe-se que a maioria das aplicações tem o desempenho relativo bem próximo de 1, para todos os valores de bloco de compressão, enquanto

Tamanho do bloco de compressão (*bytes*)	32, 64, 128, 256, 512, 1024 (variável)
Tamanho da BBE (*bytes*)	4096
Número de entradas da BEE	32
Tipo de Decodificador Huffman	Taxa de Saída Constante

Tabela 5.4: Perfil Base da ARCC com variação do tamanho do bloco de compressão.

	Desempenho Relativo					
bitcount	1,0008	1,0007	1,0007	1,0009	1,0014	1,0024
quicksort	1,0006	1,0005	1,0005	1,0007	1,0010	1,0018
cjpeg	1,0008	1,0007	1,0008	1,0010	1,0014	1,0026
djpeg	1,0033	1,0029	1,0031	1,0038	1,0057	1,0099
lame	1,0836	1,0663	1,0626	1,0719	1,0939	1,1463
stringsearch	1,0051	1,0043	1,0049	1,0060	1,0081	1,0152
patricia	3,0577	2,8841	3,0224	3,2928	4,0247	5,5274
blowfish (enc)	1,0003	1,0003	1,0003	1,0003	1,0005	1,0008
blowfish (dec)	1,0003	1,0003	1,0003	1,0003	1,0005	1,0008
crc	2,1724	1,9658	2,0533	2,2225	2,8897	4,1000
	32	**64**	**128**	**256**	**512**	**1024**
	Tamanho do bloco de compressão em *bytes*					

Tabela 5.5: Desempenho relativo das aplicações para parâmetros da arquitetura base.

algumas têm valores mais altos, como *patricia* que chega a ter um valor de mais de 5 em um bloco de compressão de 1024 *bytes*. Isto pode ser explicado pela taxa de faltas (miss rate) na *cache* de instruções L1. A Figura 5.3 mostra como é a distribuição desta taxa para as aplicações escolhidas.

As aplicações que não têm esta taxa desprezível são *patricia* com 4,86%, *crc* com 2,43% e *lame* com 0,19%. Todas as outras aplicações têm taxas de faltas na *cache* IL1 próximas de zero.

A Tabela 5.5 é transformada em três gráficos para melhor visualização dos resultados e são mostrados na Figura 5.4. As aplicações são separadas por grupos próximos de valores de desempenho relativo.

Os gráficos além de mostrarem as diferenças de DR entre os três grupos, mostram também que o DR aumenta (com poucas exceções) com o aumento do tamanho do bloco de compressão a partir de 64 *bytes*. Os blocos de compressão de tamanho 32 *bytes* se mostram

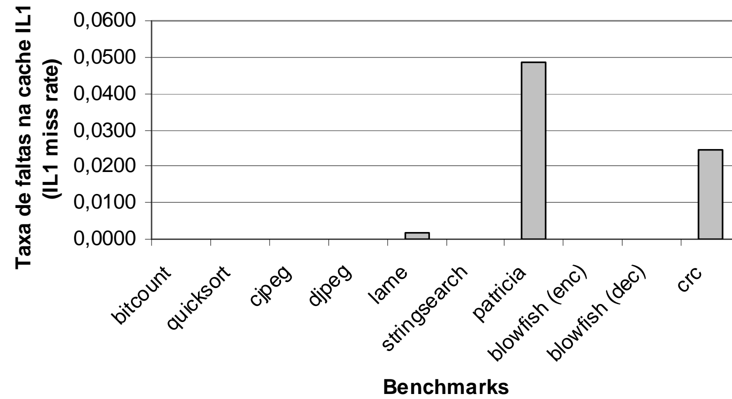

Figura 5.3: Taxa de faltas na *cache* de instruções L1 para as aplicações escolhidas

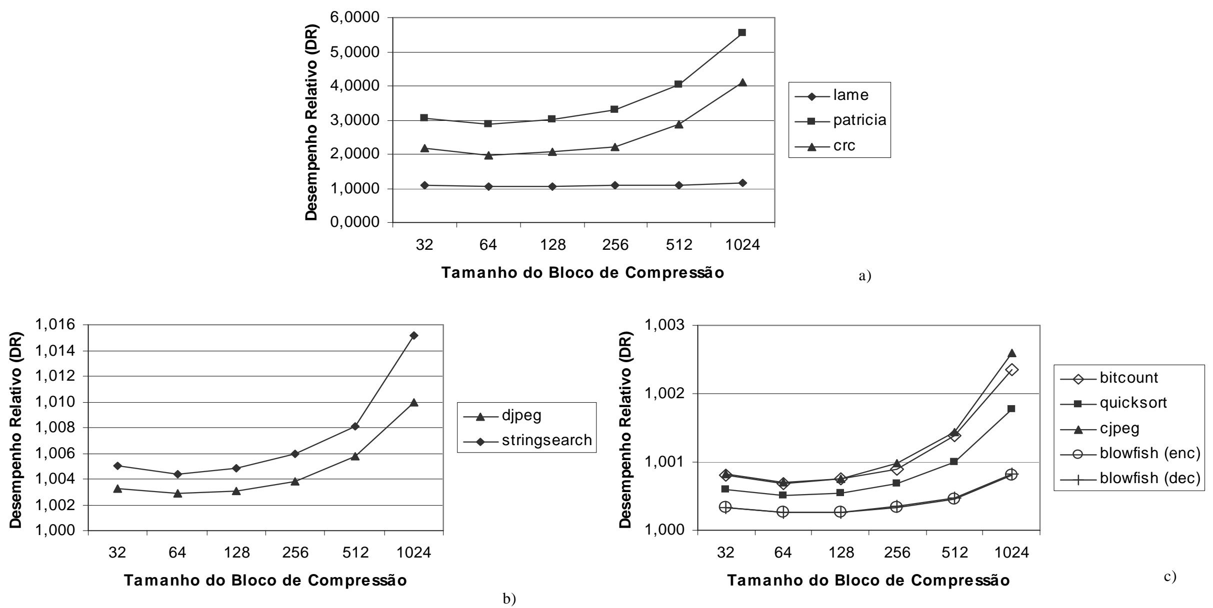

Figura 5.4: Desempenho relativo das aplicações separadas por grupos.

Taxa de Acertos no RBE

bitcount	0,0000	0,3209	0,5048	0,5457	0,5685	0,5409
quicksort	0,0000	0,3102	0,4691	0,5293	0,5201	0,5694
cjpeg	0,0000	0,3119	0,4422	0,4982	0,5288	0,5472
djpeg	0,0000	0,3194	0,4791	0,5341	0,5221	0,5634
lame	0,0000	0,3925	0,5972	0,6598	0,6725	0,7009
stringsearch	0,0000	0,3356	0,4362	0,4922	0,5011	0,5101
patricia	0,0000	0,3302	0,5117	0,6204	0,6605	0,6530
blowfish (enc)	0,0000	0,3550	0,4888	0,5193	0,5112	0,4929
blowfish (dec)	0,0000	0,3529	0,4868	0,5172	0,5112	0,4929
crc	0,0000	0,3196	0,4519	0,5258	0,5619	0,5653
	32	**64**	**128**	**256**	**512**	**1024**

Tamanho do bloco de compressão em *bytes*

Tabela 5.6: Taxas de acertos para o RBE

com DR equivalente ou até mesmo um pouco superiores aos blocos de 128 *bytes*. Isto pode ser explicado pela impossibilidade de uma linha de *cache* requerida se encontrar no RBE. Como o tamanho do bloco de compressão é igual à largura da linha de *cache*(32 *bytes*), todas as vezes que uma nova linha de *cache* é requerida ao Gerenciador de Expansão, ela não vai ser encontrada no RBE pois os dados que ele contém já foram para a *cache* antes e será disparada uma escrita na memória destes 32 *bytes* sem ter sido aproveitado mais de uma vez. A Tabela 5.6 mostra a taxa de acertos na RBE (quociente entre o número de acertos e o número total de consultas à GE pela *cache* IL1). Como podemos ver, a taxa de acertos não cresce indefinidamente, o que faz com que o desempenho caia a medida que os blocos de compressão aumentam e uma expansão de um bloco maior fica mais custoso para o sistema. As Tabelas 5.7 e 5.8 mostram respectivamente as taxas de acerto para o BBE (quociente entre o número de vezes que o bloco requerido está no BBE pelo número de consultas ao GE pela *cache* IL1) e para o BEE (quociente entre o número de vezes que uma entrada TEBC do bloco requerido está no BEE pelo número de consultas ao GE pela *cache* IL1). Lembrando que a ordem crescente de latência da operação do GE é de: Acerto em RBE, acerto no BBE e acerto no BEE (a última opção é falta em todos eles).

Pelos resultados, podemos concluir que os blocos de compressão de tamanho inter-

Taxa de Acertos no BBE						
bitcount	0,0000	0,0781	0,1214	0,1983	0,2296	0,2873
quicksort	0,0000	0,0957	0,1466	0,2099	0,2855	0,2562
cjpeg	0,0000	0,1029	0,1831	0,2348	0,2697	0,2718
djpeg	0,0000	0,0825	0,1465	0,2081	0,2763	0,2602
lame	0,0000	0,0501	0,0842	0,1463	0,1979	0,2028
stringsearch	0,0000	0,0805	0,1834	0,2506	0,3110	0,3087
patricia	0,0000	0,0524	0,0964	0,1329	0,1735	0,2270
blowfish (enc)	0,0000	0,0751	0,1724	0,2434	0,3185	0,3550
blowfish (dec)	0,0000	0,0751	0,1724	0,2434	0,3185	0,3550
crc	0,0000	0,0962	0,1787	0,2354	0,2474	0,2835
	32	**64**	**128**	**256**	**512**	**1024**
	Tamanho do bloco de compressão em *bytes*					

Tabela 5.7: Taxas de acertos para o BBE

Taxa de Acertos no BEE						
bitcount	0,0000	0,0084	0,0144	0,0264	0,0541	0,0781
quicksort	0,0000	0,0046	0,0170	0,0432	0,0556	0,0880
cjpeg	0,0000	0,0111	0,0306	0,0533	0,0734	0,1124
djpeg	0,0000	0,0090	0,0203	0,0377	0,0622	0,1005
lame	0,0000	0,0043	0,0071	0,0128	0,0239	0,0334
stringsearch	0,0000	0,0157	0,0228	0,0470	0,0604	0,0895
patricia	0,0004	0,0034	0,0051	0,0088	0,0193	0,0338
blowfish (enc)	0,0000	0,0081	0,0101	0,0385	0,0548	0,0811
blowfish (dec)	0,0000	0,0081	0,0101	0,0385	0,0548	0,0811
crc	0,0000	0,0120	0,0206	0,0344	0,0756	0,0928
	32	**64**	**128**	**256**	**512**	**1024**
	Tamanho do bloco de compressão em *bytes*					

Tabela 5.8: Taxas de acertos para o BEE

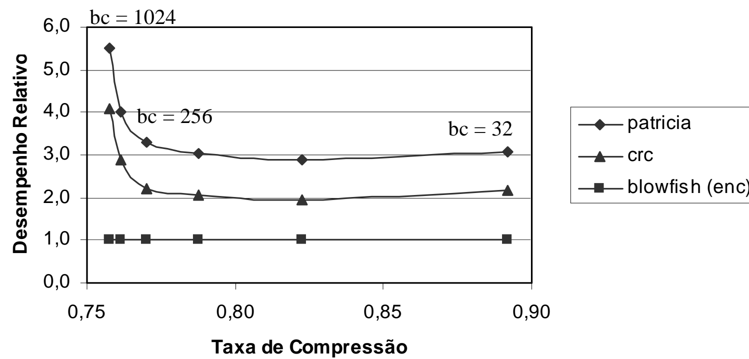

Figura 5.5: Desempenho Relativo x Taxa de Compressão

mediário podem balancear melhor o desempenho da compressão e da expansão em *hardware*. O bloco de compressão de tamanho 32 *bytes* se mostrou ruim na compressão e equivalente ao de 128 *bytes* na expansão, que por sua vez tem a taxa de compressão melhor. A Figura 5.5 mostra o gráfico do DR $\times$ TC das aplicações *patricia*, *crc* e *blowfish (enc)*. Pelas informações obtidas podemos concluir que para os blocos de compressão de 128 e 256 *bytes*, podemos ter um DR limitado a 3,29 (pela aplicação *patricia*) e uma TC nunca maior que 0,80 (para o caso da aplicação lame com bloco de compressão de 128 *bytes* — ver Tabela 5.5), sendo os blocos com maior equilíbrio entre as funções de compressão e expansão.

As próximas seções investigam os efeitos de outros parâmetros levando em conta a arquitetura base com blocos de compressão de 256 *bytes*.

5.2.2 Perfil 2: Variações no Tamanho do BBE

Nesta Seção serão investigados os efeitos da diminuição do tamanho do BBE na expansão. Isto é importante pois diminuir o tamanho da memória usada é sempre desejável. A Tabela 5.9 mostra os parâmetros da arquitetura.

A Tabela 5.10 mostra que a tentativa para economizar memória na diminuição do BBE para blocos de compressão de 256 *bytes* pode ser uma opção interessante se os programas

Tamanho do bloco de compressão (*bytes*)	256
Tamanho da BBE (*bytes*)	1024 e 2048 (variável)
Número de entradas da BEE	32
Tipo de Decodificador Huffman	Taxa de Saída Constante

Tabela 5.9: Perfil Base da ARCC com variação do tamanho do bloco de compressão.

não tiverem altas taxas de falta na *cache* IL1 como é o caso dos executáveis ARM pré-compilados dos benchmarks fornecidos pelo MiBench. Para o pior caso, em que a BBE assume 1024 *bytes*, o maior acréscimo de DR é observado no *crc* e vale 13%, seguido do *patricia* com 4,48%. Os outros programas exibiram uma variação desprezível mais uma vez creditada à uma taxa de faltas baixa na *cache* IL1.

	Desempenho Relativo		
bitcount	1,0010	1,0010	1,0009
quicksort	1,0008	1,0008	1,0007
cjpeg	1,0012	1,0011	1,0010
djpeg	1,0044	1,0041	1,0038
lame	1,0830	1,0763	1,0719
stringsearch	1,0068	1,0064	1,0060
patricia	3,4402	3,3072	3,2928
blowfish (enc)	0,0014	1,0004	1,0003
blowfish (dec)	1,0004	1,0004	1,0003
crc	2,5105	2,3980	2,2225
	1024	**2048**	**4096**
	Tamanho do BBE		

Tabela 5.10: Desempenho relativo das aplicações com a variação do tamanho do BBE.

Deve-se ter cuidado ao proceder com a diminuição do tamanho do BBE pois pela Figura 5.6 a taxa de acertos na BBE cai bastante, chegando a ter aproximadamente a metade para a aplicação *bitcount* no pior caso de uma BBE de 1024 *bytes*. Isto significa que se o acesso à memória RAM se tornar mais rápido seja por diminuição da latência ou aumento de largura de barramento, o desempenho pode ser prejudicado pois a memória poderá suprir de forma mais rápida as linhas de *cache* se não houver uma queda do número de acertos. Esta observação é importante pois larguras de barramento maiores e

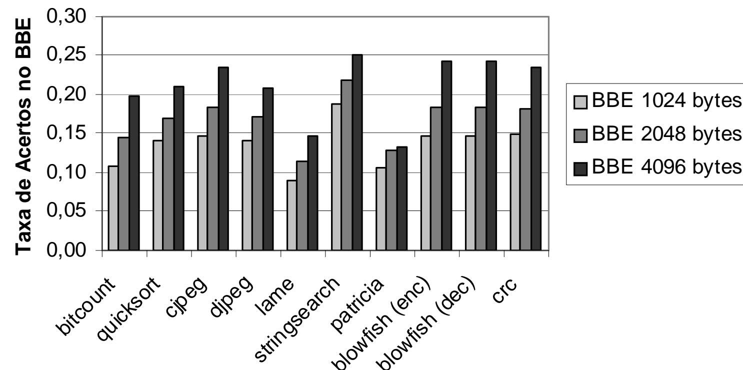

Figura 5.6: Taxa de acertos para tamanhos de BBE variáveis

tecnologia de fabricação de memórias mais rápidas podem se tornar populares em sistemas embutidos.

5.2.3 Perfil 3: Variações no Tamanho do BEE

Nesta seção é avaliada a relação entre o aumento de entradas BEE e o desempenho relativo das aplicações (ver a Tabela 5.11). Só serão vistos os resultados das aplicações *lame*, *patricia* e *crc* pois são as aplicações com o maior valor de DR e pela arquitetura estudada, sabe-se que um acerto no BEE significa apenas que uma entrada da TEBC não vai ser lida, ou seja, um acesso à memória economizado. Além disso pela Tabela 5.8 vemos que as taxas são baixas, chegando no máximo a 9,28% para o *crc* com um bloco de compressão de 1024 *bytes*. Para um bloco de compressão de 256 *bytes* chega a ser no máximo de 5,33% para a aplicação *cjpeg*.

Tamanho do bloco de compressão (*bytes*)	256
Tamanho da BBE (*bytes*)	4096 *bytes*
Número de entradas da BEE	64, 128, 256 (variável)
Tipo de Decodificador Huffman	Taxa de Saída Constante

Tabela 5.11: Perfil Base da ARCC com variação do tamanho do bloco de compressão.

A Figura 5.7 confirma a baixa influência do número de entradas BEE no desempenho

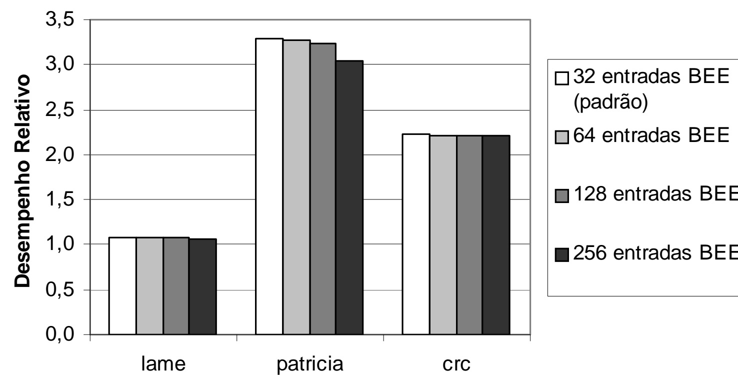

Figura 5.7: Influência do aumento do BEE nas aplicações

do sistema. Para conseguir uma redução no DR de 7,58% na aplicação mais sensível (*patricia*) foi necessário aumentar o número de entradas BEE de 32 para 256. Nas outras aplicações a melhora de desempenho é muito pequena. Como a busca nas entradas o BEE é em paralelo com as entradas do RBE e BEBBE, um número muito alto de entradas BEE passa a ser custoso para o sistema, não sendo necessário seu aumento.

5.2.4 Perfil 4: Alteração do Decodificador Huffman

Nesta Seção o decodificador Huffman é substituído pelo decodificador assíncrono explicado na Seção 4.2.2. Os parâmetros da simulação são exibidos na Tabela 5.12.

Tamanho do bloco de compressão (*bytes*)	128, 256, 512 e 1024 *bytes* (variável)
Tamanho da BBE (*bytes*)	4096
Número de entradas da BEE	32
Tipo de Decodificador Huffman	Assíncrono e consumindo 4 *bytes* por ciclo

Tabela 5.12: Perfil Base da ARCC com o decodificador Huffman assíncrono.

Uma vez que um decodificador mais rápido vai melhorar o desempenho do sistema, e sabendo-se que a maioria das aplicações já tem o DR perto de 1,00, serão analisadas somente as três aplicações do MiBench com o DR maior, que são: *lame*, *patricia* e *crc*. Serão analisados blocos de compressão de 128, 256, 512 e e 1024 *bytes* para que tenhamos

a noção da melhora de desempenho em valores de blocos intermediários e grandes.

A Tabela 5.13 mostra como fica o desempenho relativo das aplicações com o decodificador assíncrono proposto que consome 4 *bytes* por ciclo.

	Desempenho Relativo			
lame	1,0434	1,0490	1,0638	1,0988
patricia	2,3664	2,5214	3,0050	4,0129
crc	1,7190	1,8223	2,2619	3,0672
	128	**256**	**512**	**1024**
	Tam. do Bloco de Compressão			

Tabela 5.13: Desempenho relativo com decodificador Huffman assíncrono.

A Figura 5.8 compara os valores de desempenho relativo para os tamanhos de bloco de compressão acima citados para os dois tipos de decodificador de Huffman usados neste trabalho. Verifica-se que há uma melhora de desempenho significativo principalmente para os blocos de compressão maiores, chegando a uma redução de 37,74% do valor do desempenho relativo para um bloco de compressão de 1024 *bytes* da aplicação *patricia*. Estes dados comprovam mais uma vez que as aplicações com taxas maiores de faltas na *cache* IL1 são mais sensíveis a um aumento do desempenho do *hardware* de expansão. Outra informação interessante que os gráficos trazem é que para as três aplicações, o DR de um bloco de compressão de 256 *bytes* com decodificador síncrono tem o valor bem próximo (no caso do *lame* e *patricia*, tem um valor menor) do DR de um bloco de compressão de 512 com decodificador assíncrono. Se um decodificador assim for disponível, o tamanho de bloco de compressão de 512 passa a ser o mais equilibrado com relação a compressão/expansão.

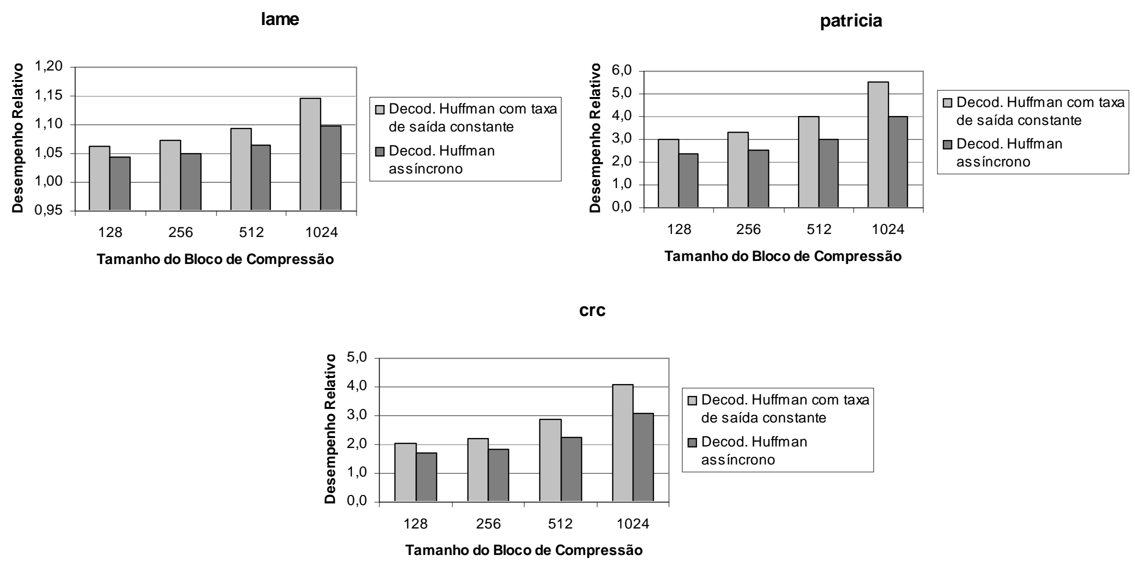

Figura 5.8: Influência do decodificador Huffman para as aplicações *lame*, *patricia* e *crc*

Capítulo 6

Conclusões

O esquema de compressão de código proposto nesta dissertação se mostrou viável e com resultados bastante positivos. As simulações mostraram que é possível conseguir taxas razoáveis de compressão sem tornar um fator degradante de desempenho. Esta arquitetura pode ser estendida a outros processadores RISC com conjunto de instruções diferentes e a outros algoritmos de compressão, desde que possam ser implementados decodificadores rápidos em *hardware*.

A implementação do algoritmo que simula a ARCC no conjunto de simuladores do Simplescalar se mostrou bem complexo de início, quando ainda não havia conhecimento suficiente do seu funcionamento. São muitos módulos, funções implementadas com macros, o que dificulta a depuração e verificação do funcionamento do programa através de um depurador (foi usado o *front-end* gráfico *ddd* 3.3 do gdb). Além disso os acessos à memória virtual simulada e aos dados do simplescalar muitas vezes têm que ser feitos com suas próprias funções, o que pode criar certos problemas se não houver cautela pois as variáveis globais são muitas.

O algoritmo de compressão escolhido, o código de Huffman, é um algoritmo simples de ser implementado em *hardware*. Para conseguir altas taxas de decodificação um circuito de decodificação tem que ser bem elaborado e existem diversas propostas e implementações

[28]. Em [6], Xu *et alii* citam o algoritmo *X-Match PRO* em seu artigo e dizem que o algoritmo tem implementação rápida em *hardware*. Pelos seus resultados a TC para blocos de compressão de 256 *bytes* fica em torno de 0,82 enquanto a média para a TC no presente trabalho é de 0,77. Para blocos maiores o *X-Match PRO* consegue seus melhores resultados e para blocos de 1024 *bytes* a TC é melhor que no algoritmo Huffman aqui apresentado.

A expansão se mostrou também bastante eficaz, com valores de DR maiores para aplicações com taxas de faltas na *cache* IL1 maiores. O Capítulo 5 mostra todas a variações de parâmetros implementadas. Para uma arquitetura base, estabelecida na Seção 4.2.2 vimos que os blocos de compressão de tamanho 128 e 256 *bytes* podem ser escolhidos como tendo o melhor compromisso entre TC e DR. Se houver disponível um decodificador Huffman que possa consumir imediatamente os dados que recebe, os blocos de compressão de 512 *bytes* passam a ter o melhor compromisso compressão/expansão.

A possibilidade de obter as taxas de acerto na RBE, BBE e BEE ajudaram a explicar alguns fatos nas simulações. A taxa de acerto 0 para a RBE nos blocos de compressão de 32 *bytes* mostra que a arquitetura só deve ser implementada com blocos maiores que a linha de *cache*. Estas taxas não são exibidas nos trabalhos correlatos de pesquisa e o conhecimento delas é importante no entendimento do funcionamento do programa simulado.

6.1 Futuros Trabalhos

Uma vez com o simulador da ARCC implementado, é possível testar novas configurações como algoritmos de compressão e expansão com algumas modificações. Além disso, quando o Simplescalar aceitar conjuntos de instruções de outros processadores embutidos, será possível investigar os efeitos da ARCC para eles.

Tendo em mãos os resultados de TC e DR, um próximo trabalho interessante seria a

pesquisa da modelagem de *hardware* para a ARCC e a simulação do consumo de energia para uma comparação com a arquitetura sem o esquema de compressão. Neste caso, pode-se ver o efeito da diminuição do tráfego de memória na economia de energia em uma arquitetura com compressão de código.

Apêndice A

Código-Fonte dos Programas

Neste apêndice são listados os códigos-fonte em linguagem C dos arquivos contendo as modificações feitas no SimpleScalar.

A.1 compressao.h

```
/***************************************************************************
*
*   Este arquivo é parte integrante da Tese de Mestrado em Arquitetura
*   e SO, NCE/UFRJ de André Bellieny Roberto da Silva, orientado por
*   Gabriel Pereira da Silva
*
***************************************************************************/

/*
*   Este arquivo contem as definicoes de funcoes
*   e estruturas usadas na compressao de dados.
*/


#ifndef COMPRESSAO_H
#define COMPRESSAO_H

#include "host.h" /* contem as definicoes de tam de dados */
#include "memory.h" /* contem as definicoes de memória */

#define TRUE 1
#define FALSE 0
```

```
#define TAM_LINHA_CACHE 32
#define LARG_BARR_MEM 4
/*em bytes */

typedef char boolean;

struct ent_TEBC_t {   /* entrada da tabela TEBC*/
  md_addr_t end_original; /* endereco alvo */
  md_addr_t end_alvo; /* endereço do codigo comprimido que sera o indice
                         do array de bytes juntos no struct mem_prog_comp*/
  half_t tam_bloco;
  half_t deslocamento;/* o deslocamento serve pro caso de uma entrada da TEBC
                         com dois blocos comprimidos */
  boolean comprimido; /* pro caso da TEBC antiga, indica se o bloco eh
                         comprimido ou nao */
  half_t tam_bloco_2; /* pra facilitar a conta da latencia */
};

struct ent_rotulo_t {
  md_addr_t   rotulo;          /* entrada da TEBC */
  boolean     valido;          /* flag de endereco valido */
  tick_t      ciclo_sim;       /* timestamp que eh o sim_cycle e vai servir para
reposicao rlu*/
};


void reporta_estat (void);
void reporta_estat_expansao (void);
void reporta_estat_2 (void);
void retorna_tam_executavel (char *nome_exec);
void captura_nome_executavel (char *nome_exec);
void comprime_codigo (struct mem_t *mem);
void computa_frequencias (struct mem_t *mem);
void le_arquivo_configuracao (void);
void constroi_heap_inicial (void);
void reheap (unsigned short  heap_entry);
void constroi_arvore (void);
unsigned short  gera_tabela_de_codigo (void);
void comprime (struct mem_t *mem);
void comprime_2 (struct mem_t *mem);


#endif
```

A.2 compressao.c

```
/***************************************************************************
```

```
*
*   Este arquivo e parte integrante da Tese de Mestrado em Arquitetura
*   e SO, NCE/UFRJ de Andre Bellieny Roberto da Silva, orientado por
*   Gabriel Pereira da Silva
*
*****************************************************************************/

/*
*   Este arquivo contem funcoes
*   usadas na compressao de dados.
*/

#include "host.h"
#include "compressao.h"
#include "memory.h"
#include "sim-outorder.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>


/* var globais */

extern md_addr_t ld_text_base;
extern md_addr_t ld_text_bound;
extern md_addr_t ld_text_size;
extern md_addr_t ld_data_size;
/*extern int mem_bus_width;*/
extern int mem_lat[];

/* variaveis globais da compressao */


static half_t          tam_bloco_compressao; /* tamanho do bloco comprimido a ser lido
no
                                                         arquivo de configuracao */
static half_t          pai[512];
static half_t          codigo[256], heap_length;
static qword_t         num_bytes_comprimidos, num_bytes_prog_comp,
num_bytes_prog_sem_comp, heap[257];
static byte_t          larg_codigo[256];
static qword_t         cont_frequencia[512];
static struct ent_TEBC_t      *TEBC;
static unsigned long   num_ent_TEBC= 0;

char                   nome_arquivo_freq[256]; /* nome do arquivo de frequencias */

/* variaveis globais da expansao */

static struct ent_rotulo_t *rotulos_BEE; /* buffer de entradas de enderecos */
static int                 num_ent_BEE;
```

```
static struct ent_rotulo_t *rotulos_BBE; /* buffer de bloco expandido
                                neste caso so colocar as entradas com os rotulos
                                e usa-se a estrutura do BEE. os rotulos ficam no
                                hardware de expansao */
static int                 tam_BBE;
static int                 num_ent_BBE;

static struct ent_rotulo_t rotulo_RBE;  /* rotulo de registrador de bloco expandido -
fica no hardware de expansao */

static md_addr_t           mascara;     /* mascara do endereco para rotulo*/

/*static int                 lat_mem_bbe_le_prim, lat_mem_bbe_le_outros,
                           lat_mem_bbe_es_prim, lat_mem_bbe_es_outros;*/
                           /* sao as latencias de leitura e escrita de memorias
pipelined do hardware de
                              expansao na memoria onde ficara o BBE */

/*static int                 lat_exp_chunk;  /* latencia da expansao de um chunck - 4
bytes */

static int                 lat_hd_prim, lat_hd_outros;
                           /* sao as latencias pipelined do hardware de expansao para
repor uma linha de cache */

static int                 lat_decod_TEBC; /* latencia de decodificacao da TEBC */
static int                 lat_compara_BEBBE_BEE;    /* latencia para comparar rotulos
com os que estao em BEBBE e BEE */

static int                 permite_bloco_maior=0; /* informa se permite que o bloco 2
seja maior que o bloco de compressao.
                                                    1 - permite, 0 - nao permite.*/

static int                 tipo_de_decod_huff;  /* 0 - sincrono com saida constante -
decodifica 2 bytes na saida por ciclo
                                                    1 - assincrono e melhor hipotese -
decodifica */


static qword_t             st_BEE_hits = 0, st_BBE_hits = 0, st_RBE_hits = 0;
static half_t              st_bl_maior_exp = 0, st_bl_maior_comp = 0;
static float               st_media_bl_maiores_comp = 0.0, st_media_bl_maiores_exp =
0.0;
static qword_t             st_soma_tam_bl_maiores_comp = 0, st_soma_tam_bl_maiores_exp =
0;

void reporta_estat (void)
{
  unsigned int i, larg_max_cod=larg_codigo[0], larg_min_cod=larg_codigo[0];
  for (i=1; i<256; i++)
    if (larg_max_cod < larg_codigo[i])
      larg_max_cod = larg_codigo[i];
  num_bytes_prog_comp = num_bytes_prog_sem_comp - ld_text_size + num_bytes_comprimidos +
```

```
num_ent_TEBC*4;
  st_media_bl_maiores_comp = (float) st_soma_tam_bl_maiores_comp/st_bl_maior_comp;
  fprintf (stderr, "Estatisticas de compressao:\n\n");
  fprintf (stderr, "Largura minima do codigo: %d\n", larg_min_cod);
  fprintf (stderr, "Largura maxima do codigo: %d\n", larg_max_cod);
  fprintf (stderr, "Tamanho da parte executavel SEM compressao: %d\n", ld_text_size);
  fprintf (stderr, "Tamanho da parte executavel COM compressao + bytes de apoio:
%d\n\n", num_bytes_comprimidos + num_ent_TEBC*4);
  fprintf (stderr, "Numero de blocos comprimidos: %d\n\n", num_ent_TEBC);

/*  fprintf (stderr, "Taxa de compressao: %f\n", (float)(num_bytes_comprimidos +
num_ent_TEBC*4 + 256*3)/ld_text_size);*/
  fprintf (stderr, "Tamanho da TEBC: %d\n", num_ent_TEBC*4);
  fprintf (stderr, "Tamanho da parte executavel COM compressao: (puro): %d\n\n",
num_bytes_comprimidos);
  fprintf (stderr, "Tamanho do arquivo arm SEM compressao: %d\n",
num_bytes_prog_sem_comp);
  fprintf (stderr, "Tamanho do arquivo arm COM compressao: %d\n\n",
num_bytes_prog_comp);
  fprintf (stderr, "Numero de blocos maiores ou iguais ao bl de compressao: %d\n",
st_bl_maior_comp);
  if (st_bl_maior_comp)
    fprintf (stderr, "Media dos blocos maiores ou iguais ao bl de compressao: %f\n\n",
st_media_bl_maiores_comp);
}

void reporta_estat_expansao (void)
{
  st_media_bl_maiores_exp = (float) st_soma_tam_bl_maiores_exp/st_bl_maior_exp;
  fprintf (stderr, "Estatisticas de expansao\n\n");
  fprintf (stderr, "RBE hits: %d\n", st_RBE_hits);
  fprintf (stderr, "BBE hits: %d\n", st_BBE_hits);
  fprintf (stderr, "BEE hits: %d\n\n", st_BEE_hits);
  fprintf (stderr, "Numero de blocos maiores ou iguais ao bl de compressao: %d\n",
st_bl_maior_exp);
  if (st_bl_maior_exp)
    fprintf (stderr, "Media dos blocos maiores ou iguais ao bl de compressao: %f\n\n",
(float)(st_media_bl_maiores_exp));
}

void reporta_estat_2 (void)
{
  unsigned int i, larg_max_cod=larg_codigo[0], larg_min_cod=larg_codigo[0];
  for (i=1; i<256; i++)
    if (larg_max_cod < larg_codigo[i])
      larg_max_cod = larg_codigo[i];
  /* 256 * 3 se refere ao codigo 2 bytes * 256 + 256 bytes que indica o tamanho do
codigo */
  fprintf (stderr, "Largura maxima do codigo: %d\n", larg_max_cod);
  fprintf (stderr, "Largura minima do codigo: %d\n", larg_min_cod);
  fprintf (stderr, "Tamanho do bloco comprimido: %d\n\n", tam_bloco_compressao);
  fprintf (stderr, "Tamanho da parte executavel SEM compressao: %d\n", ld_text_size);
  fprintf (stderr, "Tamanho da parte executavel COM compressao + bytes de apoio: %d\n",
```

```
num_bytes_comprimidos + num_ent_TEBC*4 + 256*3);
  fprintf (stderr, "Taxa de compressao: %f\n", (float)(num_bytes_comprimidos +
num_ent_TEBC*4 + 256*3)/ld_text_size);
  fprintf (stderr, "Tamanho da TEBC: %d\n\n", num_ent_TEBC*4);
  fprintf (stderr, "Tamanho da parte executavel COM compressao: (puro): %d\n\n",
num_bytes_comprimidos);
  fprintf (stderr, "Tamanho do arquivo arm SEM compressao: %d\n",
num_bytes_prog_sem_comp);
  fprintf (stderr, "Tamanho do arquivo arm COM compressao: %d\n\n",
num_bytes_prog_comp);
}


void retorna_tam_executavel (char *nome_exec)
{
  struct stat arq;

  if(stat(nome_exec,&arq))
    fatal ("Problemas na identificacao do tamanho do executavel arm");

  num_bytes_prog_sem_comp = arq.st_size;
}


void captura_nome_executavel (char *nome_exec)
{
   strcpy (nome_arquivo_freq, nome_exec);
   strcat (nome_arquivo_freq, ".freq");
}


void comprime_codigo_2 (struct mem_t *mem)
{
  permite_bloco_maior = 1; /* permite blocos maiores que o bloco de compressao */
  tam_bloco_compressao = 32;
  mascara = ~(0L  + tam_bloco_compressao - 1);

  computa_frequencias (mem);
  constroi_heap_inicial ();
  constroi_arvore ();
  if (!gera_tabela_de_codigo ())
    fatal ("Nao e possivel comprimir - codigo errado");
  comprime (mem);
  reporta_estat_2 ();
  exit (0);
}

void comprime_codigo (struct mem_t *mem)
{
  le_arquivo_configuracao ();
  constroi_heap_inicial ();
  constroi_arvore ();
  if (!gera_tabela_de_codigo ())
```

```
    fatal ("Nao e possivel comprimir - codigo errado");
  comprime (mem);
  reporta_estat ();
}


void computa_frequencias (struct mem_t *mem)
{
   register unsigned long   loop;
   FILE             *fp_freq;

   if (!(fp_freq = fopen (nome_arquivo_freq, "wt")))
     fatal ("Arquivo de frequencias nao pode ser aberto");

   for (loop = ld_text_base; loop < ld_text_bound; loop++)
      cont_frequencia[MEM_READ_BYTE(mem, loop)]++;

   for (loop = 0L; loop < 256; loop++)
      fprintf (fp_freq, "%d\n", cont_frequencia[loop]);

   fclose (fp_freq);
}

void le_arquivo_configuracao (void)
/* le os parametros que o programa precisa para simular:
   tam_bloco_compressao

   tam_BBE

   num_ent_BEE,

   lat_compara_BEBBE_BEE

   lat_decod_TEBC

   lat_hd_prim
   lat_hd_outras

   permite_bloco_maior

   tipo_de_decod_huff

   le as frequencias
*/
{
   register int  loop;
   FILE          *fp;

   if (!(fp = fopen ("/home/bellieny/sim/config_compressao.txt", "rt")))
     fatal ("Arquivo de configuracao da compressao nao pode ser aberto");

   fscanf (fp, "%d", &tam_bloco_compressao);
```

```
/*   fscanf (fp, "%d", &lat_exp_chunk);*/

   fscanf (fp, "%d", &tam_BBE);
/*   fscanf (fp, "%d", &lat_mem_bbe_le_prim);
   fscanf (fp, "%d", &lat_mem_bbe_le_outros);
   fscanf (fp, "%d", &lat_mem_bbe_es_prim);
   fscanf (fp, "%d", &lat_mem_bbe_es_outros);*/

   fscanf (fp, "%d", &num_ent_BEE);

   fscanf (fp, "%d", &lat_compara_BEBBE_BEE);

   fscanf (fp, "%d", &lat_decod_TEBC);

   fscanf (fp, "%d", &lat_hd_prim);
   fscanf (fp, "%d", &lat_hd_outros);

   fscanf (fp, "%d", &permite_bloco_maior);

   fscanf (fp, "%d", &tipo_de_decod_huff);

   for (loop = 0; loop < 256; loop++)
     fscanf (fp, "%d", (cont_frequencia+loop));

   fclose (fp);
   mascara = ~(0L  + tam_bloco_compressao - 1);

   fprintf (stderr, "Configuracao: \n\n");
   fprintf (stderr, "Tamanho do Bloco de Compressao: %d\n", tam_bloco_compressao);
   fprintf (stderr, "Tamanha da BBE em bytes: %d\n", tam_BBE);
   fprintf (stderr, "Numero de entradas da BEE: %d\n", num_ent_BEE);
   fprintf (stderr, "Latencia da comparacao dos rotulos em BEBBE e BEE: %d\n",
lat_compara_BEBBE_BEE);
   fprintf (stderr, "Latencia da decodificacao da TEBC: %d\n", lat_decod_TEBC);
   fprintf (stderr, "Latencia da reposicao da linha de cache pelo RDB: %d\n",
lat_hd_prim);
   fprintf (stderr, "Permite Bloco 2 maior que o bloco de compressao: %d\n",
permite_bloco_maior);
   fprintf (stderr, "Tipo de decodificacao Huffman: %d\n\n\n", tipo_de_decod_huff);
}

void constroi_heap_inicial (void)
{
   register unsigned short  loop;


   heap_length = 0;

   for (loop = 0; loop < 256; loop++)
      if (cont_frequencia[loop])
         heap[++heap_length] = (unsigned long) loop;

   for (loop = heap_length; loop > 0; loop--)
```

```
      reheap (loop);
}

void reheap (unsigned short  heap_entry)
{
   register unsigned short  index;
   register unsigned short  flag = 1;

   unsigned long   heap_value;


   heap_value = heap[heap_entry];

   while ((heap_entry <= (heap_length >> 1)) && (flag))
   {
      index = heap_entry << 1;

      if (index < heap_length)
         if (cont_frequencia[heap[index]] >= cont_frequencia[heap[index+1]])
            index++;

      if (cont_frequencia[heap_value] < cont_frequencia[heap[index]])
     flag--;
      else
      {
         heap[heap_entry] = heap[index];
         heap_entry       = index;
      }
   }

   heap[heap_entry] = heap_value;
}

void constroi_arvore (void)
{
   register unsigned short  findex;
   register unsigned long   heap_value;


   while (heap_length != 1)
   {
      heap_value = heap[1];
      heap[1]    = heap[heap_length--];

      reheap (1);
      findex = heap_length + 255;

      cont_frequencia[findex] = cont_frequencia[heap[1]] +
                                cont_frequencia[heap_value];
      pai[heap_value] =  findex;
      pai[heap[1]]    = -findex;
      heap[1]         =  findex;
```

```
      reheap (1);
   }

   pai[256] = 0;
}

unsigned short  gera_tabela_de_codigo (void)
{
   register unsigned short  loop;
   register unsigned short  current_length;
   register unsigned short  current_bit;

   unsigned short  bitcode;
   short           parent;


   for (loop = 0; loop < 256; loop++)
      if (cont_frequencia[loop])
      {
         current_length = bitcode = 0;
         current_bit = 1;
         parent = pai[loop];

         while (parent)
         {
            if (parent < 0)
            {
               bitcode += current_bit;
               parent = -parent;
            }
            parent = pai[parent];
            current_bit <<= 1;
            current_length++;
         }

         codigo[loop] = bitcode;

         if (current_length > 16)
            return (0);
         else
            larg_codigo[loop] = (unsigned char) current_length;
      }
      else
         codigo[loop] = larg_codigo[loop] = 0;

   return (1);
}

void comprime (struct mem_t *mem)
{
  register byte_t          thebyte = 0;
  register short           loop1;
  register half_t          current_code;
```

```
  register unsigned long   loop, loop2;

  byte_t                   *bloco_comp_aux;
  half_t                   cont_bloco_comp_aux=0, ent_TEBC=0;

  unsigned short  current_length, dvalue;
  unsigned long   curbyte = 0;
  short           curbit = 7;
  FILE            *fp_mem_comp;  /* contem o codigo comprimido */

  /* cria espaco na memoria para as estruturas */

  if (!(fp_mem_comp = fopen ("memcomp.mem", "w+b")))
    fatal ("Arquivo de apoio nao pode ser aberto");

  num_ent_TEBC = (unsigned long) ((ld_text_bound - (ld_text_base & mascara) +
(tam_bloco_compressao - 1)) / tam_bloco_compressao);

  if (!(TEBC = malloc (sizeof (struct ent_TEBC_t) * (num_ent_TEBC))))
    fatal ("Erro de alocacao de memoria para TEBC");

  if (!(bloco_comp_aux = malloc (sizeof(byte_t)*2*tam_bloco_compressao)))
    fatal ("Erro de alocacao de memoria para variavel auxiliar");

  /* O algoritmo fica complicado se for ficar verificando se o bloco de
    compressao eh maior que tam_bloco_compressao e tomar providencias ao
    mesmo tempo. Eh melhor entao preencher tudo
    e depois resolver. Por isso tem o tamanho de 16 bits * tam_bloco_compressao.
  */

  TEBC[ent_TEBC].end_original = ld_text_base; /* Inicializa a TEBC */
  TEBC[ent_TEBC].end_alvo = 0;
  ent_TEBC++;

  /*mem_prog_comp.inst_comp = NULL;*/

  for (loop = ld_text_base; loop < ld_text_bound; loop++)
  {
    dvalue         = MEM_READ_BYTE(mem, loop);
    current_code   = codigo[dvalue];
    current_length = (unsigned short) larg_codigo[dvalue];

    for (loop1 = current_length-1; loop1 >= 0; --loop1)
    {
        if ((current_code >> loop1) & 1)
          thebyte |= (byte_t) (1 << curbit);

        if (--curbit < 0)
        {
          /*putc (thebyte, fp_mem_comp);*/
          bloco_comp_aux[cont_bloco_comp_aux] = thebyte;
          cont_bloco_comp_aux++;
```

```
          thebyte = 0;
          /*curbyte++;*/
          curbit = 7;
        }
    }
    if (!((loop - (ld_text_base & mascara) + 1) % tam_bloco_compressao)) {
    /* ultimo byte do bloco comprimido */
      if (curbit < 7) {
          /*putc (thebyte, fp_mem_comp);*/
          bloco_comp_aux[cont_bloco_comp_aux] = thebyte;
          cont_bloco_comp_aux++;

          thebyte = 0;
          /*curbyte++;*/
          curbit = 7;
      }
      if ((cont_bloco_comp_aux >= tam_bloco_compressao) && (!permite_bloco_maior)) {
        for (loop2 = loop - tam_bloco_compressao + 1; loop2 <= loop; loop2++)
          putc (MEM_READ_BYTE(mem, loop2), fp_mem_comp);
        TEBC[ent_TEBC - 1].comprimido = FALSE;
        TEBC[ent_TEBC - 1].tam_bloco =  tam_bloco_compressao;
        curbyte += tam_bloco_compressao;
      }
      else {
      /* verifica se foi eficazmente comprimido se o tamanho do bloco aux for maior ou
igual
          a tam_bloco_compressao eh melhor entao deixar sem compressao */
        for (loop2 = 0; loop2 < cont_bloco_comp_aux; loop2++)
          putc (bloco_comp_aux[loop2], fp_mem_comp); /* escreve inst. comprimidas*/
        TEBC[ent_TEBC - 1].comprimido = TRUE;
        TEBC[ent_TEBC - 1].tam_bloco =  cont_bloco_comp_aux;
        curbyte += cont_bloco_comp_aux;
      }
      if (cont_bloco_comp_aux >= tam_bloco_compressao) {
        st_bl_maior_comp ++;
        st_soma_tam_bl_maiores_comp += cont_bloco_comp_aux;
      }

      cont_bloco_comp_aux = 0; /* ja escreveu o bloco de compressao entao pode zerar o
contador */

      if (loop != (ld_text_bound - 1)) { /*se nao eh o ultimo byte da ultima inst */
        /* atualiza a TEBC */
        TEBC[ent_TEBC].end_original = loop + 1;
        TEBC[ent_TEBC].end_alvo = curbyte;
        ent_TEBC++;
      }
    }
  }

  if (curbit < 7) { /* saiu do loop, o thebyte tem alguma coisa ainda pra ser escrita */
    /*putc (thebyte, fp_mem_comp);
    curbyte++; */
```

```
    bloco_comp_aux[cont_bloco_comp_aux] = thebyte;
    cont_bloco_comp_aux++;
  }

  if (cont_bloco_comp_aux) { /* se existe alguma coisa no bloco auxiliar, tem que ser
escrito */
    if ((cont_bloco_comp_aux >= tam_bloco_compressao) && (permite_bloco_maior)) {
      for (loop2 = TEBC[ent_TEBC - 1].end_original; loop2 < ld_text_bound; loop2++)
        putc (MEM_READ_BYTE(mem, loop2), fp_mem_comp);
      TEBC[ent_TEBC - 1].comprimido = FALSE;
      TEBC[ent_TEBC - 1].tam_bloco =  ld_text_bound - TEBC[ent_TEBC - 1].end_original;
      curbyte += ld_text_bound - TEBC[ent_TEBC - 1].end_original;
    }
    else {
    /* verifica se foi eficazmente comprimido se o tamanho do bloco aux for igual
        a tam_bloco_compressao eh melhor entao deixar sem compressao */
      for (loop2 = 0; loop2 < cont_bloco_comp_aux; loop2++)
        putc (bloco_comp_aux[loop2], fp_mem_comp); /* escreve inst. comprimidas*/
      TEBC[ent_TEBC - 1].comprimido = TRUE;
      TEBC[ent_TEBC - 1].tam_bloco = cont_bloco_comp_aux;
      curbyte += cont_bloco_comp_aux;
    }
    if (cont_bloco_comp_aux >= tam_bloco_compressao) {
      st_bl_maior_comp ++;
      st_soma_tam_bl_maiores_comp += cont_bloco_comp_aux;
    }
  }

  num_bytes_comprimidos = curbyte;
  free (bloco_comp_aux);
  fclose (fp_mem_comp);
  if (ent_TEBC != num_ent_TEBC)
    fatal ("Problemas com a TEBC. Cuidar disso!");
}



/*******************************************************************************

  Expansao

*******************************************************************************/

void inicializa_expansao (void)
{    register int i;

    num_ent_BBE = tam_BBE/tam_bloco_compressao;

    if (!(rotulos_BBE = malloc (sizeof (struct ent_rotulo_t) * (num_ent_BBE))))
      fatal ("Erro de alocacao de memoria para rolulos do BEE");

    if (!(rotulos_BEE = malloc (sizeof (struct ent_rotulo_t) * (num_ent_BEE))))
```

```
      fatal ("Erro de alocacao de memoria para rolulos do BEE");

    rotulo_RBE.valido = FALSE;

    for (i=0; i<num_ent_BEE; i++) {
      rotulos_BEE[i].valido = FALSE;
      rotulos_BEE[i].ciclo_sim = 0L;
    }

    for (i=0; i<num_ent_BBE; i++) {
      rotulos_BBE[i].valido = FALSE;
      rotulos_BBE[i].ciclo_sim = 0L;
    }
}

unsigned int            /* total latency of access */
latencia_acesso_memoria (int blk_sz, int mem_lat_0, int mem_lat_1, tick_t *mem_ready,
tick_t sim_cycle, int mem_pipelined)
/* block size accessed */
{
  int chunks = (blk_sz + (LARG_BARR_MEM - 1)) / LARG_BARR_MEM;

  if (mem_pipelined)
    return (/* first chunk latency */mem_lat_0 +
        (/* remainder chunk latency */mem_lat_1 * (chunks - 1)));
  else /* !mem_pipelined */
    {
      unsigned int total_lat, lat =
    (/* first chunk latency */mem_lat_0 +
     (/* remainder chunk latency */mem_lat_1 * (chunks - 1)));
      if (*mem_ready > sim_cycle)
    {
      total_lat = (lat + (*mem_ready - sim_cycle));
      *mem_ready += lat;
    }
      else /* mem_ready <= sim_cycle */
    {
      total_lat = lat;
      *mem_ready = sim_cycle + lat;
    }

      return total_lat;
    }
}


unsigned int calcula_latencia_expansao (md_addr_t endereco, tick_t ciclo_sim,
                                        tick_t *p_mem_ready, int lat_mem_0, int
lat_mem_1,
                                        int mem_pipelined)
/* se entrou aqui eh porque aconteceu um cache miss no endereco
   agora a expansao vai acontecer de acordo com o algoritmo de
   expansao descrito no texto da dissertacao */
```

```
{
  md_addr_t             rotulo = endereco & mascara;
  register int          i;
  int                   chunks_cache = (TAM_LINHA_CACHE + (LARG_BARR_MEM - 1)) /
LARG_BARR_MEM;
  int                   chunks_bloco_comp = (tam_bloco_compressao + (LARG_BARR_MEM - 1))
/ LARG_BARR_MEM;
  unsigned int          lat=0;
  half_t                ent_TEBC;
  half_t                tam_bl_comp_1, tam_bl_comp_2, tam_bl_comp;
  struct ent_rotulo_t   rotulo_aux;

  half_t                ent_BEE, ent_BBE;
  boolean               rotulo_na_BEE = FALSE, bl_1_eh_comp, bl_2_eh_comp, bl_eh_comp;

  if (rotulo_RBE.valido && (rotulo_RBE.rotulo == rotulo)) {
    st_RBE_hits++;
    return (lat_hd_prim + (chunks_cache - 1) * lat_hd_outros);
  }

  lat += lat_compara_BEBBE_BEE; /* ja que nao esta no RBE vai procurar no BEBBE e no BEE
*/

  for (i=0; i<num_ent_BBE; i++)
    if (rotulos_BBE[i].valido  && (rotulos_BBE[i].rotulo == rotulo)) {/* esta no BBE e
basta mandar ler */
      rotulos_BBE[i].ciclo_sim = ciclo_sim; /* atualiza o timestamp pra mostrar que foi
acessado por ultimo */
      lat += latencia_acesso_memoria (TAM_LINHA_CACHE, lat_mem_0, lat_mem_1,
p_mem_ready, ciclo_sim, mem_pipelined);
      st_BBE_hits ++;
      return lat;
      /*return (lat_mem_bbe_le_prim + (chunks_cache -1) * lat_mem_bbe_le_outros);*/
    }

  /* nao esta na BBE, vai ter que procurar na memoria, expandir e mandar pra cache L1 */

  for (i=0; i<num_ent_BEE; i++) /* procura no BEE primeiro. Se estiver ai nao vai ter
que ir na TEBC  mas a
                                   nivel de latencias ja foi computado em cima pois eh
feito em paralelo */
    if (rotulos_BEE[i].valido  && (rotulos_BEE[i].rotulo == rotulo)) {
      ent_BEE = i;
      rotulo_na_BEE = TRUE;
      st_BEE_hits ++;
      break;
    }

  if (!rotulo_na_BEE) {/* nao esta na BEE tem que ir ate a TEBC */
    lat += latencia_acesso_memoria (LARG_BARR_MEM, lat_mem_0, lat_mem_1, p_mem_ready,
ciclo_sim, mem_pipelined);
    /* Leitura de uma entrada da TEBC eh mais um acesso a memoria */
```

```
    lat += lat_decod_TEBC; /* ao receber a entrada da TEBC, decodifica */
  }

  /*descobrindo quantos bytes tem o bloco comprimido de rotulo "rotulo" */
  ent_TEBC = (rotulo - (ld_text_base & mascara)) / (tam_bloco_compressao);

  tam_bl_comp = TEBC[ent_TEBC].tam_bloco;
  bl_eh_comp = TEBC[ent_TEBC].comprimido;

  if (tam_bl_comp >= tam_bloco_compressao) {
    st_bl_maior_exp ++;
    st_soma_tam_bl_maiores_exp += tam_bl_comp;
  }

  if (rotulo_RBE.valido) {
    tick_t ciclo_aux = rotulos_BBE[0].ciclo_sim;
    boolean encontrou_ent_BBE_inv = FALSE;
    ent_BBE = 0;
    for (i = 0; i<num_ent_BBE; i++)
      if (!rotulos_BBE[i].valido){
        ent_BBE = i;
        encontrou_ent_BBE_inv = TRUE;
        break;
      }
    if (!encontrou_ent_BBE_inv) {
      for (i = 0; i<num_ent_BBE; i++)
        if (ciclo_aux > rotulos_BBE[i].ciclo_sim) {
          ciclo_aux = rotulos_BBE[i].ciclo_sim;
          ent_BBE = i;
        }
    }

    lat += latencia_acesso_memoria (tam_bloco_compressao, lat_mem_0, lat_mem_1,
p_mem_ready, ciclo_sim, mem_pipelined);

    if (!encontrou_ent_BBE_inv) /* se a substituicao nao eh por haver rotulos invalidos
*/
      memcpy (&rotulo_aux, (rotulos_BBE + ent_BBE), sizeof(struct ent_rotulo_t));
    /* salvando em rotulo_aux o rotulo do bloco que foi para o BBE */
    memcpy ((rotulos_BBE + ent_BBE) , &rotulo_RBE, sizeof(struct ent_rotulo_t));
    /* salva no BBE o rotulo correspondente ao transferido */


    if (!rotulo_na_BEE) {
    /* se nao esta na BEE, procura um rotulo invalido ou usado com o timestamp menor -
LRU */
      tick_t ciclo_aux_2 = rotulos_BEE[0].ciclo_sim;
      boolean encontrou_ent_BEE_inv = FALSE;
      ent_BEE = 0;
      for (i = 0; i<num_ent_BEE; i++)
        if (!rotulos_BEE[i].valido){
          ent_BEE = i; /* nao preciso mais do antigo ent_BEE, ja posso descarta-lo */
          encontrou_ent_BEE_inv = TRUE;
```

```
          break;
        }
      if (!encontrou_ent_BEE_inv) {
        for (i = 0; i<num_ent_BEE; i++)
          if (ciclo_aux > rotulos_BEE[i].ciclo_sim) {
            ciclo_aux = rotulos_BEE[i].ciclo_sim;
            ent_BEE = i;
          }
      }
    }
    if (!encontrou_ent_BBE_inv)
      memcpy ((rotulos_BEE + ent_BEE), &rotulo_aux, sizeof(struct ent_rotulo_t));
    /* passa para a BEE o rotulo que estava na BBE e ia ser descartado e la embaixo
passa
        o rotulo atual para o RBE o que completa o rodizio*/
  }

  if (bl_eh_comp) { /* so vai calcular as latencias de expansao se o bloco for
comprimido */
    if (tipo_de_decod_huff == 0) { /* taxa de saida fixa */
      int lat_aux = tam_bloco_compressao/2; /* fornece 2 bytes por ciclo */
      int tam_bl_comp_aux = LARG_BARR_MEM * (lat_aux + 1 + 1); /* 1 refere-se a esperar
lat_mem_0 para receber o bloco */
      if (tam_bl_comp_aux < tam_bl_comp)
        tam_bl_comp_aux = tam_bl_comp; /* isso nao deve ocorrer */
      lat += latencia_acesso_memoria (tam_bl_comp_aux, lat_mem_0, lat_mem_1,
p_mem_ready, ciclo_sim, mem_pipelined);
    }
    else /* gasta um ciclo pra 4 bytes */
      lat += latencia_acesso_memoria (tam_bl_comp, lat_mem_0, lat_mem_1, p_mem_ready,
ciclo_sim, mem_pipelined);
  } /* se nao for comprimido, a latencia eh nula */

  rotulo_RBE.rotulo = rotulo;
  rotulo_RBE.ciclo_sim = ciclo_sim; /* atualiza o rotulo */
  rotulo_RBE.valido = TRUE;

  lat += lat_hd_prim + (chunks_cache - 1)* lat_hd_outros; /* reposicao na cache */

  return lat;
}
```

Glossário

ARCC Arquitetura RISC com Código Comprimido — Arquitetura proposta nesta dissertação composta de um núcleo RISC adicionado de um hardware de expansão. Esta arquitetura é capaz de executar instruções comprimidas segundo o algoritmo de Huffman, 35

BBC *Buffer* de Bloco Comprimido — Estrutura FIFO na ARCC encarregada de receber e armazenar as palavras comprimidas para que o decodificador Huffman possa consumi-las de acordo com seu ritmo (throughput), 47

BBE *Buffer* de Blocos Expandidos — Região da memória RAM (área de dados) da ARCC que contém blocos de compressão mais recentemente expandidos, 47

BEBBE *Buffer* de Endereços do BBE — Estrutura na ARCC que armazena os endereços dos blocos contidos no BBE. Todas as entradas devem ser comparadas simultaneamente para que não haja degradação do desempenho, 49

BEE *Buffer* de Entradas de Endereços — Estrutura da ARCC que registra as traduções mais recentes da TEBC. Funciona como uma TLB em memória virtual. Um acerto na BEE equivale à economia de um acesso à TEBC e da latência da respectiva tradução, 49

Bloco de Compressão Na ARCC corresponde a uma ou mais linhas de cache do processador usado como unidade de compressão. Uma operação de compressão ou expansão é feita sempre em um bloco de compressão, 42

CCRP *Code Compression RISC Processor* — Arquitetura proposta por Wolfe e Chanin em [4] que consiste de um núcleo de processador RISC com uma memória *cache* especial adicional que recebe o código expandido e de um dispositivo de *hardware* que gerencia a expansão das instruções, 18

CLB *Cache Line Address Lookaside Buffer* — Uma pequena *cache* que armazena as entradas mais freqüentes da LAT da arquitetura CCRP [4], 20

CodePack É uma solução comercial da IBM para a compressão de código. O processador RISC usado é o PowerPC da IBM, 23

DR Desempenho Relativo — Relação entre o tempo de execução do programa com compressão e o tempo de execução do programa sem o esquema de compressão, 67

GE Gerenciador de Expansão — *Hardware* da ARCC composto de estruturas de armazenamento e controle capaz de gerenciar a busca e expansão de um bloco de compressão para fornecer uma linha de cache requerida pelo núcleo do processador, 46

HE *Hardware* de Expansão — Circuito responsável pela expansão de um bloco de compressão. No caso da ARCC trata-se do decodificador Huffman, 47

LAT *Line Address Table* — Consiste em uma tabela que mapeia endereços de blocos de instruções sem compressão em endereços de blocos de instruções comprimidas na arquitetura CCRP [4], 19

RBE Registrador de Bloco Expandido — *Buffer* responsável por armazenar o mais recente bloco de compressão expandido pelo HE na ARCC, 47

TC Taxa de Compressão — Medida principal da compressão que consiste na razão entre o tamanho em bytes do código executável com compressão e o tamanho do código executável sem compressão, 66

TEBC Tabela de Endereçamento de Código Comprimido — Tabela da ARCC que traduz um endereço do código sem compressão (endereço original) em um endereço na memória comprimida de programa (endereço alvo), 43

Referências Bibliográficas

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

[2] SHORT, K. L. *Microprocessors and programmed logic (2nd ed.).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987. ISBN 0-13-580606-2.

[3] FAGGIN, F. et al. The History of the 4004. *IEEE Micro*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 16, n. 6, p. 10–20, 1996. ISSN 0272-1732.

[4] WOLFE, A.; CHANIN, A. Executing compressed programs on an embedded RISC architecture. In: *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1992. p. 81–91. ISBN 0-8186-3175-9.

[5] HUFFMAN, D. A. A method for the construction of minimum redundancy codes. *Proceedings of the IERE*, v. 40, p. 1098–1101, 1952.

[6] XU, X. H.; CLARKE, C. T.; JONES, S. R. High performance code compression architecture for the embedded ARM/THUMB processor. In: *CF '04: Proceedings of the 1st conference on Computing frontiers.* New York, NY, USA: ACM Press, 2004. p. 451–456. ISBN 1-58113-741-9.

[7] BURGER, D.; AUSTIN, T. M. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, ACM Press, New York, NY, USA, v. 25, n. 3, p. 13–25, 1997. ISSN 0163-5964.

[8] GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, Dezembro 2001.

[9] LEKATSAS, H.; WOLF, W. Code compression for embedded systems. In: *DAC '98: Proceedings of the 35th annual conference on Design automation.* New York, NY, USA: ACM Press, 1998. p. 516–521. ISBN 0-89791-964-5.

[10] LEKATSAS, H.; WOLF, W. Random Access Decompression using Binary Arithmetic Coding. In: *DCC '99: Proceedings of the Conference on Data Compression.* Washington, DC, USA: IEEE Computer Society, 1999. p. 306. ISBN 0-7695-0096-X.

[11] LEFURGY, C.; PICCININNI, E.; MUDGE, T. Evaluation of a high performance code compression method. In: *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1999. p. 93–102. ISBN 0-7695-0437-X.

[12] INTEL. *Intel StrongARM SA-1100 Microprocessor Developer's Manual*. [S.l.], Outubro 2001. Disponível em: <http://developer.intel.com/design/strong/manuals/278240.htm>.

[13] LEKATSAS, H.; HENKEL, J.; WOLF, W. Code compression for low power embedded system design. In: *DAC '00: Proceedings of the 37th conference on Design automation*. New York, NY, USA: ACM Press, 2000. p. 294–299. ISBN 1-58113-187-9.

[14] DEBRAY, S.; EVANS, W. Profile-guided code compression. In: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2002. p. 95–105. ISBN 1-58113-463-0.

[15] DEBRAY, S.; EVANS, W. S. Cold code decompression at runtime. *Commun. ACM*, ACM Press, New York, NY, USA, v. 46, n. 8, p. 54–60, 2003. ISSN 0001-0782.

[16] XIE, Y.; WOLF, W.; LEKATSAS, H. Profile-Driven Selective Code Compression. In: *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003. p. 10462. ISBN 0-7695-1870-2.

[17] CORLISS, M. L.; LEWIS, E. C.; ROTH, A. The implementation and evaluation of dynamic code decompression using DISE. *Trans. on Embedded Computing Sys.*, ACM Press, New York, NY, USA, v. 4, n. 1, p. 38–72, 2005. ISSN 1539-9087.

[18] ESPONDA, M.; ROJAS, R. *The RISC Concept - A Survey of Implementations*. Berlim, Alemanha, Setembro 1991. Disponível em: <http://www.inf.fu-berlin.de/lehre/WS94/RA/RISC-9.html>.

[19] BESZÉDES Árpád et al. Survey of code-size reduction methods. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 35, n. 3, p. 223–267, 2003. ISSN 0360-0300.

[20] Eric Bodden and Malte Clasen and Joachim Kneis. Arithmetic Coding in revealed. In: RWTH AACHEN University. *Proseminar Datenkompression 2001*. 2002. German version available: Proseminar Datenkompression, Arithmetische Kodierung. Disponível em: <http://www.bodden.de/publications>.

[21] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, v. 23, n. 3, p. 337–343, 1977. Disponível em: <citeseer.ist.psu.edu/ziv77universal.html>.

[22] SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de Dados e Seus Algoritmos*. 2ª ed. Rio de Janeiro, RJ: LTC Editora Livros Técnicos e Científicos S.A., 1994.

[23] BENES, M.; WOLFE, A.; NOWICK, S. M. A high-speed asynchronous decompression circuit for embedded processors. In: *ARVLSI '97: Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*. Washington, DC, USA: IEEE Computer Society, 1997. p. 219. ISBN 0-8186-7913-1.

[24] AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 35, n. 2, p. 59–67, 2002. ISSN 0018-9162.

[25] PATTERSON, D. A.; HENNESSY, J. L. *Computer Architecture: A Quantitative Approach*. 3ª ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1-55860-596-7.

[26] TURLEY, J. Embedded Processors by the Numbers. *Embedded Systems Programming, http://www.embedded.com/1999/9905/9905turley.htm*, Maio 1999.

[27] ARM. *ARM ELF Specification*. [S.l.], Junho 2001.

[28] BENES, M.; NOWICK, S. M.; WOLFE, A. A fast asynchronous huffman decoder for compressed-code embedded processors. In: *ASYNC '98: Proceedings of the 4th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. Washington, DC, USA: IEEE Computer Society, 1998. p. 0043. ISBN 0-8186-8392-9.