Uma Arquitetura para Compressão de Código em Processadores Embarcados

André B. R. Silva
CEFET Campos – UNED Macaé
Macaé, RJ, Brasil
abellieny@cefetcampos.br

Gabriel P. Silva
DCC/IM – Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brasil
gabriel@dcc.ufrj.br

Resumo

O uso eficiente de sistemas móveis e embarcados depende fortemente de estratégias adequadas para redução do consumo de energia. Esses sistemas são caracterizados também por uma grande restrição de recursos, entre eles a quantidade de memória disponível para as aplicações. Este trabalho apresenta um esquema de compressão de código para processadores embarcados compatíveis com o processador ARM, que visa apresentar soluções para essas duas demandas específicas dos sistemas móveis e embarcados. A compressão é feita diretamente no código objeto, após a compilação do código-fonte pelas ferramentas tradicionais, e usa um algoritmo de compressão por freqüência de opcodes*: o código de Huffman. Para a execução do código comprimido é necessário que haja um* hardware *de expansão das instruções associado ao núcleo do processador. O* hardware *de expansão é composto por estruturas de armazenamento e controle projetados para a realização eficiente das operações de expansão. As medidas de desempenho de compressão e os efeitos da expansão foram feitos a partir da simulação em uma versão modificada do SimpleScalar e com uso da suíte de avaliação MiBench. As simulações realizadas mostram que a taxa média de compressão foi de 76% para o conjunto de* benchmarks *estudados e com uma perda relativamente pequena no desempenho, quando comparado com a execução das aplicações na sua versão sem compressão.* [1]

1 Introdução

Os sistemas móveis e embarcados têm se popularizado nos dias atuais. Com a possibilidade de se fabricar elementos processadores, memórias e baterias recarregáveis a custos mais baixos e com tamanhos menores, várias aplicações antes baseadas em sistemas mecânicos, elétricos e eletrônicos (dedicados) passaram a utilizar microprocessadores. Incluem-se nestes casos os *handhelds*, telefones celulares, *notebooks* e *MP3 players*, entre outros.

Sistemas móveis devem ser projetados em função de um equilíbrio entre desempenho e consumo. Nos sistemas fixos como os microcomputadores de mesa (*desktops*) o controle do consumo muitas vezes não é fator mais importante no projeto, mas nos *notebooks* e sistemas embarcados o cuidado com o consumo pode ser o grande diferencial.

Uma forma de aumentar a autonomia dos sistemas é desenvolver baterias com maior capacidade, o que nem sempre é de custo viável. Outra forma é prover o *hardware* de meios de restrição dos recursos. 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 e compressão de código, entre outros.

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

A expansão do código deve ser feita por partes sob a demanda do fluxo de execução que não é seqüencial pois existem instruções de desvio. Um programa comprimido, por ocupar um tamanho menor de memória do que o seu equivalente sem compressão, necessita uma alteração dos endereços físicos para os desvios. Uma solução é modificar o compilador e substituir os endereços de desvio por endereços reais. Uma outra solução é criar uma tabela que indique a tradução dos endereços físicos pelos comprimidos. Esta última abordagem não necessita da alteração do compilador e é usada neste trabalho.

Na próxima seção, serão vistos os trabalhos correlatos, na Seção 3 o algoritmo de compressão é analisado, a Seção 4 trata da arquitetura proposta, a Seção 5 mostra as ferramentas e o método usado no levantamento dos resultados, a Seção 6 analisa os resultados obtidos e finalmente na Seção 7 são apresentadas as conclusões e futuros trabalhos.

[1] Este trabalho contou com o apoio financeiro da CAPES

2 Trabalhos Correlatos

Nesta seção são analisadas algumas propostas que têm correlação com o trabalho atual.

2.1 Wolfe e Chanin

Wolfe e Chanin [14] propuseram a arquitetura *Code Compression RISC Processor* (CCRP) que consiste de um núcleo de um processador RISC com uma memória *cache* adicional que recebe o código expandido além 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.

A compressão do código foi feita por blocos de instruções de 32 *bytes* usando o código de Huffman [6]. Para esse sistema, duas modificações foram propostas. Na primeira, um *byte* não pode, em nenhum caso, ser codificado com mais de 16 *bits*. A segunda se refere à escolha de um código único para todos os programas. A escolha foi feita baseada na freqüência de *bytes* de dez programas diferentes. Os autores chamaram esta modificação de *Preselected Bounded Huffman*. Blocos de instruções que não podem ser eficientemente comprimidos foram deixados intactos.

A estrutura *Line Address Table* (LAT), incorporada ao circuito de reposição de *cache*, consiste de uma tabela que mapeia endereços de blocos de instruções em endereços de blocos de instruções comprimidas.

A expansão se faz por meio do decodificador implementado em *hardware*.

Verificou-se que para memórias (de programa) lentas o modelo de código comprimido teve melhor desempenho que o modelo tradicional, sem compressão, quando as taxas de falta na *cache* são altas. Entretanto, quando as memórias são mais rápidas, ocorreu a situação inversa.

2.2 CodePack da IBM

O CodePack da IBM [7], [12] 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 foi mudado nem foi preciso implementar extensões arquiteturais. A compressão nesta arquitetura é feita por meio de ferramentas em *software* em que o ponto inicial é o código executável ELF original do PowerPC, não sendo necessário o código fonte.

A compressão é baseada na observação de que as instruções usadas nas aplicações não são uniformemente distribuídas ao longo dos 32 *bits* usados para representar uma simples instrução. Algumas instruções aparecem no código com maior freqüência que outras, e se forem codificadas com uma seqüência menor de *bits*, o código poderá ser armazenado em uma memória menor. A decodificação é 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% no espaço de memória para armazenar o código.

2.3 Xu, Clarke e Jones

Xu, Clarke e Jones [15] descreveram um sistema de compressão de instruções do processador ARM/THUMB [8] 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 teve por objetivo mostrar que usando um processador ARM no modo THUMB, que tem uma densidade maior de código, pode-se ainda comprimir o código e obter-se economia de memória.

A arquitetura é uma versão modificada do CCRP [14] e se chama *Code Compression THUMB Processor* (CCTP). 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.

Uma tabela, a *Compressed Block Address Table* (CBAT) com a mesma função da LAT [14] foi usada para traduzir o espaço de endereçamento comprimido para o espaço sem compressão.

Definiu-se o desempenho relativo (DR) como sendo o quociente entre o número de ciclos de execução da aplicação comprimida e o número de ciclos da aplicação original. Esse valor variou entre 1,0 e 7,0 para as aplicações simuladas, aumentando conforme aumenta o tamanho do bloco de compressão. Algumas aplicações com baixa taxa de faltas na *cache* (como *susan-s* e *dijkstra*) mostraram-se com desempenho relativo próximo de 1,0 mesmo com blocos de compressão maiores.

Para efeito de equilíbrio entre compressão e desempenho, verificou-se que blocos comprimidos 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%.

3 Compressão

Dentre os vários métodos de se comprimir código [10], [13], o algoritmo de compressão escolhido para este trabalho é o *código de Huffman*, um algoritmo sem perdas e baseado na freqüência de ocorrência dos caracteres. Este algoritmo foi escolhido por ter boa taxa de compressão (quociente entre o tamanho do código comprimido e o tamanho do código sem compressão) e permitir que a decodificação possa ser feita por *hardware*.

Para adequar o código de Huffman ao trabalho, a parte executável do arquivo objeto é dividida em blocos de $2^n \times 32$ *bytes*, onde *n* é um inteiro, que serão comprimidos separadamente. Os 32 *bytes* correspondem à largura de uma linha de *cache* L1 do ARM SA-1110 [8]. Esses agrupamentos de *bytes* são denominados *blocos de compressão*. O parâmetro *n* é alterado nas simulações e assume valores iguais a 1, 2, 4, 8, 16 ou 32, o que corresponde a blocos de compressão de 32, 64, 128, 256, 512 ou 1024 *bytes*, respectivamente. Um *bloco comprimido* por sua vez se refere 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*. Isto é importante pois o *hardware* de expansão [14], [2] não é eficaz com grandes cadeias de *bits*.

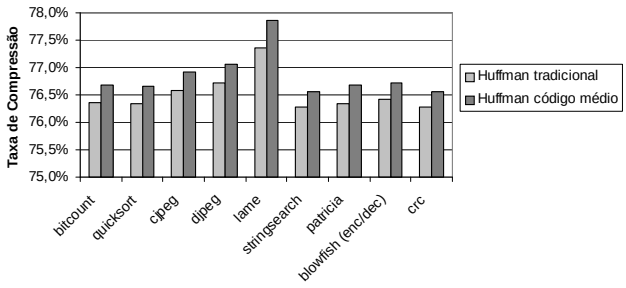

Figura 1. Comparação entre o código Huffman tradicional e o modificado com código médio para um bloco de compressão de 32 bytes

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 deve ser carregada no *hardware* de expansão. Uma 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. Uma forma simples de encontrar um código "médio" é tirar a média ponderada das freqüências de vários códigos de programas com relação ao tamanho do seu código. Para determinar o código de Huffman médio, foram usadas as aplicações *bitcount*, *quicksort*, *cjpeg*, *djpeg*, *lame*, *stringsearch*, *patricia*, *blowfish*, *crc*, *ghostscript*, *dijkstra*, *rijndael* e *fft*. Com base na Figura 1 vemos que o código de Huffman médio é bastante satisfatório para esta arquitetura.

4 Arquitetura

A arquitetura proposta chama-se *Arquitetura RISC com Código Comprimido* (ARCC) e está representada na Figura 2. Nesta arquitetura, a UCP é o núcleo do processador ARM.

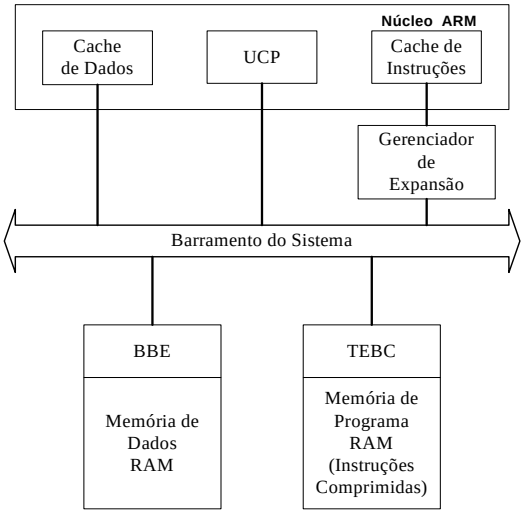

Figura 2. ARCC - Arquitetura principal

A *Tabela de Endereçamento de Bloco Comprimido* (TEBC) tem a mesma função da LAT em [14] e da CBAT em [15] e contém as informações necessárias à tradução de um endereço do código sem compressão (endereço original) em um endereço na memória de programa comprimida (endereço alvo) independente do sistema de memória ser real ou virtual. Esta tabela é carregada na memória de programa junto com o código comprimido. Suas informações são estabelecidas pela ferramenta que comprime o código executável. Uma entrada da tabela de endereçamento contém um endereço alvo de 32 *bits*. A Figura 3 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 em

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 tabela e obter uma entrada com o endereço para o bloco comprimido na memória de programa.

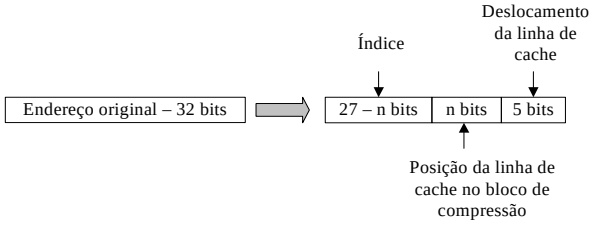

Figura 3. Componentes do endereço original

Quando os blocos de compressão são grandes, o tamanho da tabela de endereçamento pode ser um fator importante de custo extra da taxa de compressão, uma vez que a ela deve ser carregada junto com o código comprimido. A tabela tem um custo de 12,5% do tamanho do código sem compressão para blocos de compressão de 32 *bytes* e de 0,4% com blocos de 1024 *bytes*.

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*. Neste trabalho a expansão é feita entre a memória principal e a cache, diferentemente de outras arquiteturas em que a expansão é feita entre a cache e o processador [9].

Como o bloco de compressão pode ser maior que a linha de *cache*, existe nesta arquitetura o *Buffer de Blocos Expandidos* (BBE) que tem a mesma função do DBB proposto em [15]. Ele recebe blocos expandidos e pode armazenar um número de blocos que é uma potência de 2. Como forma de investigar seus efeitos na arquitetura, seu tamanho será variado.

A Figura 4 mostra os componentes internos do gerenciador de expansão. Dentro dele temos o *Hardware* de Expansão (HE) que equivale a um decodificador Huffman para o algoritmo de compressão 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 [15] 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.

Os decodificadores Huffman em *hardware* podem ser

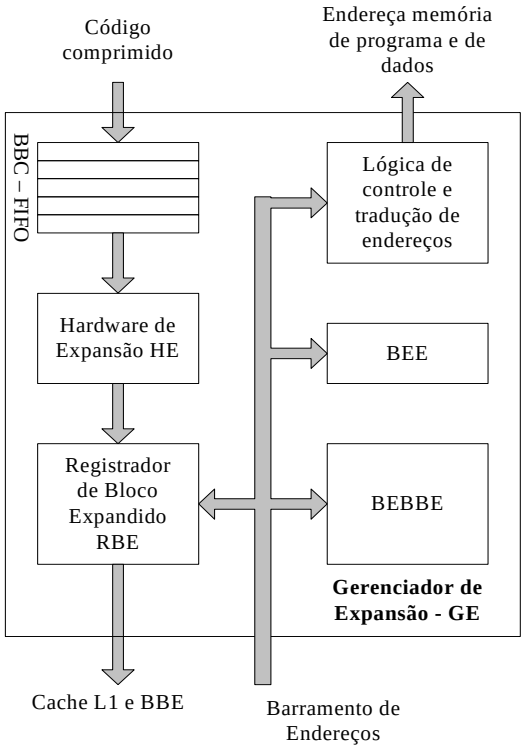

Figura 4. Gerenciador de Expansão

síncronos ou assíncronos. São analisados os dois tipos para verificar o impacto na arquitetura estudada. O decodificador síncrono usado tem taxa da saída constante e independente da compressão [14] [4] com uma taxa de entrada constante de 16 *bits* e saída de um caractere (*byte*) a cada ciclo. Se as operações forem feitas na subida e descida do ciclo de relógio, então o decodificador leva $m \div 2$ ciclos onde m é o número de *bytes* do bloco de compressão (no estado original). O decodificador assíncrono [2] [1] é 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 que o primeiro grupo leva para chegar a memória.

O Registrador de Bloco Expandido (RBE) tem a função de armazenar o bloco expandido que é saída do decodificador Huffman. Este registrador é usado como acesso rápido para faltas na *cache* cujos dados requisitados podem ser encontrados lá. Este tipo de acesso é mais rápido que na RAM pois o hardware de expansão deve ser construído com memórias estáticas. O registrador deve ter os campos *Dados*, *Rótulo* e *Validade*. O campo *Dados* é o local de arma-

zenamento do bloco de compressão na sua forma expandida ou original. O campo *Rótulo* armazena o índice do endereço que desencadeou a expansão. Para saber se um endereço se encontra neste registrador, 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 registrador à *cache*, basta selecionar um dos *n* blocos do campo *Bloco de compressão* através do campo de *n bits* do endereço (ver Figura 3).

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. Ele funciona como uma TLB em memória virtual e o número de entradas é um dos parâmetros variáveis da pesquisa. Com um acerto neste *buffer* a tabela de endereçamento (TEBC) não necessita ser lida para encontrar um endereço alvo, diminuindo em pelo menos um acesso de memória na busca pelo bloco comprimido correspondente. Se o este buffer 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 termos de tempo de execução.

Nesta arquitetura, existe também uma estrutura de armazenamento que é o *Buffer* de Endereços do BBE (BEBBE) responsável por armazenar os endereços dos inícios dos blocos (expandidos) no *buffer* de blocos expandidos (BBE). Ao ser buscada uma instrução, o BEE e o BEBBE são varridos simultaneamente na busca de um acerto de um índice de endereço com o campo *Rótulo*.

Finalmente a *Lógica de controle e tradução de endereços* controla toda a movimentação de dados, as comparações de endereços, o disparo do decodificador Huffman 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* que foram determinadas em função de parâmetros arquiteturais típicos [4].

5 Método Experimental

A simulação da arquitetura foi feita na plataforma SimpleScalar [3] tendo como máquina-alvo o processador ARM. Alterando-se o código fonte do simulador foram incluídas as características particulares da arquitetura ARCC. Um *software* foi desenvolvido para gerar o código comprimido segundo as especificações da Seção 3.

O conjunto de aplicativos MiBench [5] foi escolhido para compor o conjunto de programas de teste (*benchmarks*). O MiBench é um conjunto de códigos-fonte gratuitos e propõe-se a representar as aplicações mais relevantes no universo dos sistemas embarcados. As aplicações do pacote de *benchmarks* usadas nas simulações foram escolhidas de forma a representar todas as categorias e estão descritas na Tabela 1.

Tabela 1. Aplicações MiBench Utilizadas

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

Para verificar o impacto dos diferentes componentes no desempenho do sistema, usaremos uma arquitetura base que é descrita na Tabela 2. Seus parâmetros foram escolhidos de acordo com os melhores resultados dos trabalhos correlatos.

Tabela 2. Arquietura Base da ARCC

Tamanho do bloco de compressão (*bytes*)	256 (*bytes*)
Tamanho da BBE (*bytes*)	4096
Número de entradas da BEE	32
Tipo de Dec. Huffman	Tx. de Saída Constante

6 Análise de Resultados

6.1 Compressão

A Figura 5 mostra a média das taxas de compressão das aplicações para cada tamanho de bloco de compressão simulado, destacando a influência do tamanho da tabela de endereçamento (TEBC). Deve-se notar que quanto maior o tamanho dos blocos de compressão, menor vai ser a tabela e menor vai ser a perda por alinhamento dos blocos comprimidos.

A taxa de compressão obtida sem a tabela de endereçamento é em média de 76,9% para blocos de compressão de 32 *bytes* e de 75,5% para blocos de 1024 *bytes*.

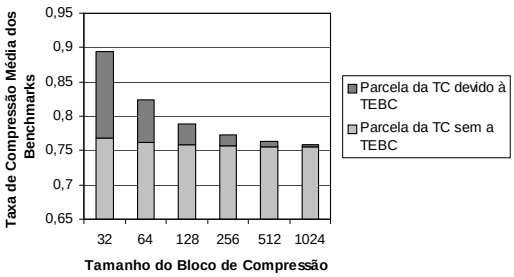

Figura 5. Influência da TEBC na taxa de compressão

6.2 Expansão

Na simulação, várias medidas de desempenho foram feitas com a variação de alguns parâmetros em torno de uma arquitetura base (ver Tabela 2).

6.2.1 Variações no Tamanho do Bloco de Compressão

A Figura 6 mostra o desempenho relativo (DR) em função das variações do tamanho do bloco de compressão. Note que as aplicações estão separadas por grupos. A diferença nestas medidas é explicada pelo número de faltas na *cache* de instruções de primeiro nível (IL1). As aplicações que não têm esta taxa desprezível são *patricia* com 4,9%, *crc* com 2,4% e *lame* com 0,2%. Todas as outras aplicações têm taxas de faltas na *cache* IL1 próximas de zero. O aumento na taxa de faltas em IL1 causa a degradação do desempenho.

6.2.2 Variações no Tamanho do BBE

Nesta seção, é avaliado o impacto da diminuição do tamanho do BBE na arquitetura proposta. A tentativa é de não degradar o desempenho e ao mesmo tempo diminuir a memória do *hardware* de expansão. Os tamanhos testados são de 2048 e 1024 bytes, uma vez que o tamanho original do BBE é de 4094 bytes.

Não foram percebidas grandes variações no DR das aplicações [4], porém a taxa de acertos caiu. Isso pode ser problemático se a memória RAM se tornar mais rápida, levando a uma degradação do desempenho geral do sistema. A Figura 7 mostra estas variações.

6.2.3 Variações no Tamanho do BEE

São investigados os efeitos da variação no tamanho do BEE para as 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.

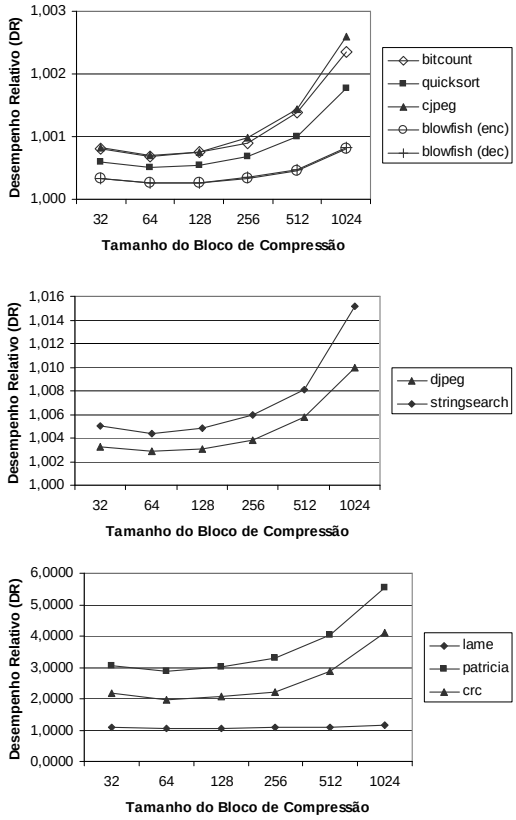

Figura 6. Desempenho relativo das aplicações por grupo

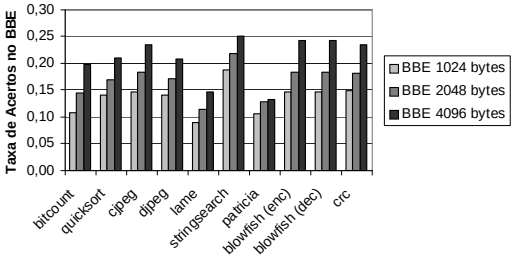

Figura 7. Taxa de acertos por tamanho do BBE

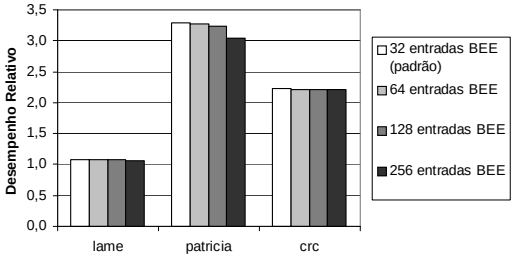

Figura 8. Influência do aumento do BEE

A Figura 8 confirma a baixa influência do número de entradas BEE no desempenho do sistema. Para conseguir uma redução no DR de 7,6% 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.

6.2.4 Alteração do Decodificador Huffman

Um decodificador mais rápido aumenta o desempenho do sistema, e sabendo-se que a maioria das aplicações já tem o DR perto do valor unitário, são analisadas somente as três aplicações do MiBench com DR piores (maiores), que são: *lame*, *patricia* e *crc*. Para comparar com os resultados anteriores, o tamanho do bloco de compressão assume os valores de 128, 256, 512 e 1024 *bytes*.

A Figura 9 compara os valores do DR 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 37,7% do valor do DR 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 uma melhora no desempenho do *hardware* de expansão.

7 Conclusões e Futuros Trabalhos

As simulações mostram que é possível conseguir boas taxas de compressão com o código de Huffman médio, sendo de 76% para o conjunto de aplicações estudado, e, lembrando que quanto maior o tamanho do bloco de compressão, menor o *overhead* da TEBC o que aumenta a eficiência da compressão.

Quanto à expansão, o *desempenho relativo* (DR) piora com o aumento do tamanho do bloco de compressão, o que faz com que este tamanho tenha de ser escolhido em função de um compromisso entre a eficiência da compressão e da expansão.

A arquitetura se mostrou pouco sensível às variações de tamanho da BBE, o que pode ser uma economia no custo do *hardware* comparado a [15] que utiliza 4096 *bytes* numa es-

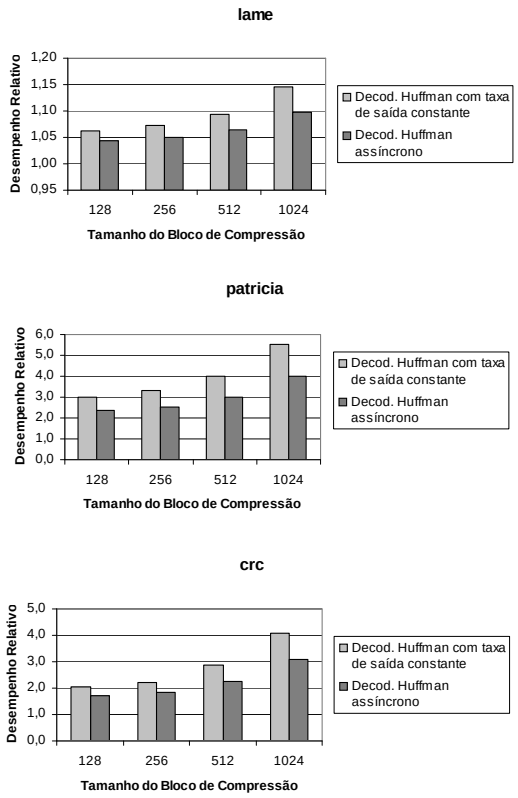

Figura 9. Influência do decodificador Huffman

trutura similar. Desta forma, pode-se usar um BBE de 1024 *bytes* sem maiores prejuízos no desempenho do sistema.

O *hardware* de expansão foi avaliado com dois tipos de decodificador, e esta parte da pesquisa se mostrou bastante promissora com relação à melhora do DR. Com um decodificador mais rápido (assíncrono) pode-se aumentar o tamanho do bloco de compressão e manter o DR dentro de níveis aceitáveis. Nesta arquitetura, para blocos de compressão de 256 bytes, o DR não é maior que 2,52 para o decodificador Huffman assíncrono.

Os próximos passos deste trabalho, incluem a avaliação do ganho de energia por meio de simuladores como o *eCACTI* [11] e o custo do *hardware* de expansão pela síntese a partir de uma descrição em *VHDL*.

Referências

[1] M. Benes, S. M. Nowick, and A. Wolfe. 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*, page 0043, Washington, DC, USA, 1998. IEEE Computer Society.

[2] M. Benes, A. Wolfe, and S. M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *ARVLSI '97: Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, page 219, Washington, DC, USA, 1997. IEEE Computer Society.

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[4] A. B. R. da Silva. Um Esquema de Compressão de Código para Processadores Embutidos. Master's thesis, IM-NCE-UFRJ, 08 2006.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, Dezembro 2001.

[6] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IERE*, 40:1098–1101, 1952.

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

[8] Intel. *Intel StrongARM SA-1100 Microprocessor Developer's Manual*, Outubro 2001.

[9] H. Lekatsas, J. Henkel, and W. Wolf. Code compression for low power embedded system design. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 294–299, New York, NY, USA, 2000. ACM Press.

[10] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 516–521, New York, NY, USA, 1998. ACM Press.

[11] M. Mamidipaka and N. Dutt. ecacti: An enhanced power estimation model for on-chip caches. Technical report, University of California, 2004.

[12] A. Orpaz and S. Weiss. A study of CodePack: optimizing embedded code space. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 103–108, New York, NY, USA, 2002. ACM Press.

[13] Árpád BeszÉdes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.

[14] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 81–91, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[15] X. H. Xu, C. T. Clarke, and S. R. Jones. High performance code compression architecture for the embedded ARM/THUMB processor. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 451–456, New York, NY, USA, 2004. ACM Press.