Instruction Compression in Runtime for Embedded Systems

Wanderson Roger Azevedo Dias, Raimundo da Silva Barreto
Universidade Federal do Amazonas – Departamento de Ciência da Computação
Manaus, Amazonas, Brasil
{roger, rbarreto}@dcc.ufam.edu.br

Edward David Moreno
Universidade Federal de Sergipe – Departamento de Ciência da Computação
Aracaju, Sergipe, Brasil
edwdavid@gmail.com

Abstract

The efficient use of embedded systems relies heavily on appropriate strategies to optimize the execution time and power consumption. These systems are characterized by resource restrictions, including the amount of memory available for applications. However, there are several techniques that make the embedded systems more efficient. One of those techniques is the code compression; the proposals found in the analyzed literature assume that the code is compressed at compilation time and decompressed at runtime. This article proposes the development of a new method of compression and decompression (on-the-fly) called of MIC (Middle Instruction Compression). The MIC was compared with the Huffman method and both were implemented in hardware using VHDL and FPGA. The results of our experiments showed that the MIC achieved better performance when compared to Huffman for some programs from MiBench. We have reduced 17% the number of logical elements of FPGA and 6% clock frequency (in MHz) and 42% rate of compression.

1. Introdução

Sistemas embarcados são quaisquer sistemas digitais que estejam inseridos a outros sistemas com a finalidade de acrescentar ou otimizar funcionalidades [14]. Os sistemas embarcados têm por função monitorar e/ou controlar o ambiente no qual esteja inserido. Esses ambientes podem estar presentes em dispositivos eletrônicos, eletrodomésticos, veículos, máquinas, motores e muitas outras aplicações.

A crescente demanda pelo uso de sistemas embarcados tem se tornado cada vez mais comum, motivando a implementação de complexos sistemas em um único chip, os chamados *System-on-Chip* (SoC). Neste caso, o processador embarcado é um dos principais componentes dos sistemas computacionais embarcados [3]. Atualmente, muitos processadores embarcados encontrados no mercado são baseados em arquiteturas de alto-desempenho (por exemplo, as arquiteturas RISC de 32 *bits*) que garantem um melhor desempenho computacional para as tarefas a serem executadas. Consequentemente, o projeto de sistemas embarcados para processadores de alto desempenho não é uma tarefa simples.

Sabe-se que muitos dos sistemas embarcados são alimentados por baterias. Por essa razão, é de suma importância que estes sistemas sejam capazes de controlar e gerenciar sua potência, possibilitando assim a redução no consumo de energia e no controle do aquecimento. Portanto, projetistas e pesquisadores concentraram-se no desenvolvimento de técnicas que diminuam o consumo de energia mantendo os requisitos de desempenho. Uma dessas técnicas é a compressão do código das instruções em memória.

A grande maioria das técnicas, metodologias e padrões de desenvolvimento de software, para o controle e o gerenciamento do consumo de energia, não se mostra viável para o desenvolvimento de sistemas embarcados, devido aos mesmos possuírem inúmeras limitações de recursos computacionais e físicos [10]. As atuais estratégias concebidas para o controle e o gerenciamento no consumo de energia, foram desenvolvidas para sistemas de propósitos gerais, em que os custos adicionais de processadores ou memórias são geralmente insignificantes [14].

O tamanho de código aumenta significativamente na medida em que os sistemas tornam-se mais heterogêneos e complexos. Neste sentido, surgiu uma técnica de alto nível que procura comprimir o código em tempo de compilação e a respectiva descompressão, por sua vez, é feita em tempo de execução [10, 11, 12].

A técnica de compressão foi desenvolvida com o intuito de reduzir o tamanho de um código [13]. Mas no decorrer do tempo, grupos de pesquisadores verificaram que essa técnica poderia trazer grandes benefícios para o desempenho e o consumo de energia nos sistemas de propósitos gerais e nos sistemas embarcados. A partir do momento que o código em memória está comprimido é possível em cada requisição do processador buscar uma quantidade bem maior de instruções contidas na memória. Assim, haverá uma diminuição nas atividades de transição nos pinos de acesso à memória, levando a um possível aumento no desempenho do sistema e a uma possível redução no consumo de energia do circuito [13].

Da mesma forma, quando se armazenam instruções comprimidas na memória *cache*, aumenta-se o número de instruções armazenadas na *cache* e aumenta-se sua taxa de acertos (*hit rate*), diminuindo a busca na memória principal, aumentando o desempenho do sistema e, por conseguinte, diminuindo o consumo de energia.

Este artigo apresenta o desenvolvimento de um novo método de compressão/descompressão de instruções (em tempo de execução), que foi implementado em VHDL (*Very High Speed Integrated Circuit Hardware Description Languages*) e prototipado em uma FPGA (*Field Programmable Gate Array*), denominado de *MIC* (*Middle Instruction Compression*), o qual foi comparado com o tradicional método de *Huffman* também implementado em hardware, e mostrou ter mais eficiência computacional que o método de *Huffman* a partir de uma comparação usando o *benchmark MiBench* [5].

O restante do artigo está organizado da seguinte forma: a Seção 2 apresenta os trabalhos correlatos; a Seção 3 explana a arquitetura PDCCM desenvolvida para o método *MIC*; a Seção 4 detalha a compressão em tempo de execução; descreve o método *MIC* e a implementação realizada em FPGA e por fim, a Seção 6 apresenta as conclusões e idéias para trabalhos futuros.

2. Trabalhos Correlatos

Nesta seção são mostradas algumas arquiteturas para a execução de códigos de instruções comprimidas, encontradas na literatura.

WOLFE & CHANIN [15] desenvolveram o CCRP (*Compressed Code RISC Processor*), que foi o primeiro hardware descompressor implementado em um processador RISC (MIPS R2000) e também foi a primeira técnica a usar as falhas de acesso à *cache* para acionar o mecanismo de descompressão.

O CCRP tem uma arquitetura idêntica ao padrão do processador RISC e assim os modelos dos programas são inalterados. Isso implica em que todas as ferramentas de desenvolvimento existentes para a arquitetura RISC, incluindo compiladores otimizados, simuladores funcionais, bibliotecas gráficas e outras, também servem para a arquitetura CCRP.

A unidade de compressão usada é a linha da *cache* de instruções. A cada falha de acesso à *cache*, as instruções são buscadas na memória principal, descomprimidas e alimentam a linha da *cache* onde houve a falha [15]. O fato de o CCRP já fazer a descompressão das instruções antes de armazená-las na *cache* é vantajoso, no sentido que os endereços de saltos contidos na *cache* são os mesmos do código original. Isto resolve a maioria dos problemas de endereçamento, não havendo necessidade de recorrer a artifícios como: (I) Colocar hardware extra no processador para tratamento diferenciado dos saltos; e (II) Fazer *patches* de endereços de salto.

A técnica CCRP utilizou o algoritmo de *Huffman* [6] gerado através de um histograma de ocorrências de *bytes* de programa e mostrou uma razão de compressão de 73%, em média, para o pacote testado (composto pelos programas: *nasa1*, *nasa7*, *tomcatv*, *matrix25A*, *espresso*, *fpppp* e outros). Para modelos de memórias mais lentos DRAM (*Dynamic Random Access Memory*), o desempenho do processador foi na maioria das vezes suavemente melhorado. Para modelos mais rápidos de memória EPROM (*Erasable Programmable Read Only Memory*), o desempenho sofreu uma leve degradação.

AZEVEDO [2] propôs um método chamado de IBC (*Instruction Based Compression*), que tem por função realizar a divisão do conjunto de instruções do processador em classes, levando em consideração a quantidade de ocorrências juntamente com número de elementos de cada classe. Pesquisas realizadas por AZEVEDO [2] mostraram melhores resultados na compressão de 4 classes de instruções. O método de compressão desenvolvido consiste em agrupar pares no formato [prefixo, *codeword*] que substituem o código original. Nos pares formados, o prefixo indica a classe da instrução e o *codeword* serve como um índice para a tabela de instruções.

O processo de descompressão é realizado em 4 estágios de *pipeline*. O primeiro estágio é chamado de *INPUT* onde é convertido o endereço do processador (código não comprimido) em endereço da memória principal. O segundo estágio é chamado de *FETCH*, que é responsável pela busca da palavra comprimida na memória principal. O terceiro estágio é conhecido como *DECODE* onde verdadeiramente é realizada a decodificação dos *codewords*. E finalmente no quarto estágio, chamado de *OUTPUT*, é realizada a consulta no dicionário de instrução para ser fornecida a instrução ao processador. Nos testes realizados, AZEVEDO [2] obteve uma taxa de compressão de 53,6% para o processador MIPS (*Microprocessor without Interlocked Pipeline Stages*) e 61,4% para o processador SPARC (*Scalable Processor Architecture*). Quanto ao

desempenho, constatou-se uma perda de 5,89% utilizando o método IBC.

BENINI *et al* [3] desenvolveram um algoritmo de compressão que é adaptado para execução eficiente do hardware (descompressor). As instruções são compactadas em grupos que têm o tamanho de uma linha da *cache* e a sua descompactação ocorre no instante que são extraídas da *cache*. Experimentos foram realizados com o processador DLX, devido o mesmo ter uma arquitetura simples de 32 *bits* e também ser uma arquitetura RISC. Além disso, o processador DLX é semelhante a vários processadores comerciais da família ARM (*Advanced RISC Machine*) [1] e MIPS. Uma tabela de 256 posições foi utilizada para guardar as instruções mais executadas. Cada linha da *cache* é formada por 4 instruções originais ou um conjunto de instruções comprimidas e possivelmente intercaladas com outras não comprimidas, prefixado por uma palavra de 32 *bits*. A palavra não comprimida tem um posicionamento fixo na linha da *cache* e serve para diferenciar uma linha de *cache* com instruções comprimidas das outras linhas com as instruções originais. De fato, uma linha de *cache* comprimida não contém necessariamente todas as instruções comprimidas, mas sempre deve ter um número entre 5 e 12 instruções comprimidas na linha da *cache* para ser vantajoso o uso da compressão [3].

Para evitar o uso das tabelas de tradução de endereços, BENINI *et al*, exigem que os endereços de destino estejam sempre alinhados a 32 *bits* (palavra). A primeira palavra (32 *bits*) da linha de *cache* contém uma marca L e um conjunto de *bits* de *flags*. A marca é um *opcode* de instrução não utilizada, ou seja, um *opcode* inválido que sinaliza uma linha comprimida (no processador DLX os *opcodes* são de 6 *bits*).

O algoritmo de compressão desenvolvido por BENINI *et al* [3], analisa o código sequencialmente, a partir da primeira instrução (supondo que cada linha da *cache* já esteja alinhada) e tenta acondicionar instruções adjacentes em linhas comprimidas. Os experimentos realizados em vários pacotes de código C do *benchmark* fornecido pelo projeto *Ptolemy* [4], comprovaram que houve uma redução média de 28% no tamanho do código e de 30% no consumo de energia.

LEKATSAS *et al* [8, 9], desenvolveram uma unidade de descompressão com um único ciclo. A descompressão pode ser aplicada para instruções de qualquer tamanho de um processador RISC (16, 24 ou 32 *bits*). A única aplicação específica é a parte do interfaceamento entre o processador e a memória (principal ou *cache*). O mecanismo de descompressão é capaz de descomprimir uma ou duas instruções por ciclo para atender a demanda da CPU sem aumentar o tempo de execução. Foi desenvolvida uma técnica para criar um dicionário que contém as instruções que aparecem com mais frequência. O dicionário de código refere-se a uma classe de métodos de compressão que substitui sequencias de símbolos com os índices de uma tabela. Essa tabela é chamada de "dicionário" e os índices são os "*codewords*" no programa compactado [9]. A principal vantagem dessa técnica é que os índices geralmente são de comprimento fixo, e assim, simplifica a lógica da descompressão em acessar o dicionário e também reduz a latência da descompressão.

Os resultados obtidos nos testes realizados demonstraram que houve um ganho médio de 25% de desempenho no tempo de execução dos aplicativos usando a compressão de código e uma média de 35% na redução do tamanho do código. Essa tecnologia desenvolvida não está limitada em apenas um processador, mas sim pode ser aplicada e obter resultados similares em outros processadores.

LEFURGY *et al* [7], propuseram uma técnica de compressão de código baseado na codificação do programa usando um dicionário de códigos. Assim, a compressão é realizada após a compilação do código fonte, porém, o código objeto é analisado e as sequencias comuns de instruções são substituídas por uma palavra codificada (*codeword*), como na compressão de texto. Apenas as instruções mais frequentes são comprimidas. Um *bit* (*escape bit*) é utilizado para distinguir uma palavra comprimida (codificada) de uma instrução não comprimida. As instruções correspondentes às instruções comprimidas são armazenadas em um dicionário no hardware de descompressão. As instruções comprimidas são usadas para indexar as entradas do dicionário. O código final consiste de *codewords* misturadas com instruções não comprimidas.

Observa-se que um dos problemas mais comuns encontrados na compressão de código se refere à determinação dos endereços alvo das instruções de salto. Normalmente este tipo de instrução (desvio direto) não é codificado para evitar a necessidade de reescrever as palavras de códigos que representam estas instruções [6]. Já os desvios indiretos podem ser codificados normalmente, pois, como seus endereços alvos estão armazenados em registradores, apenas as palavras de códigos necessitam ser rescritas. Neste caso, é necessária apenas uma tabela para mapear os endereços originais armazenados no registrador para os novos endereços comprimidos.

Este método diverge dos demais métodos vistos na literatura no sentido que, os endereços alvos estejam sempre alinhados a 4 *bits* (tamanho de um *codeword*) e não ao tamanho da palavra do processador (32 *bits*). Como vantagem destaca-se uma melhor compressão; mas como desvantagem verifica-se a necessidade de alterações no *core* do processador (um hardware extra) para tratar desvios para endereços alinhados a 4 *bits*. Entretanto, não fica claro os detalhes sobre a interação do hardware descompressor com os processadores

experimentados (PowerPC, ARM e i386). O funcionamento do hardware descompressor é realizado basicamente da seguinte maneira: A instrução é buscada da memória, caso seja um *codeword*, a lógica de decodificação dos *codewords* obtém o deslocamento e o tamanho do *codeword* que servirá como índice para acessar a instrução não comprimida no dicionário e repassar ao processador. No caso de instruções não comprimidas, elas são repassadas diretamente ao processador. Com o método proposto em [7], foram obtidas taxas de compressões de 61% para o processador PowerPC, 66% para o processador ARM e 75% para o processador i386. As métricas de desempenho e consumo de energia não foram expressas.

3. Arquiteturas para Compressão

Na literatura são encontrados dois tipos básicos de arquiteturas de compressão de código, CDM e PDC, que indicam o posicionamento do descompressor em relação ao processador e subsistema de memória, como mostra a Figura 1. Segundo NETTO [10], a arquitetura CDM (*Cache Decompressor Memory*) indica que o descompressor está posicionado entre a *cache* e a memória principal, enquanto que a arquitetura PDC (*Processor Decompressor Cache*) posiciona o descompressor entre o processador e a *cache*.

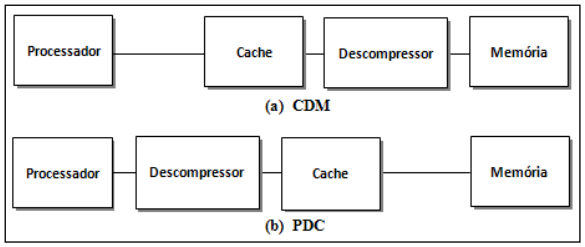

Figura 1. Arquiteturas de descompressão de código: (a) CDM e (b) PDC [10]

Como foi visto anteriormente (Seção 2), o desenvolvimento de arquiteturas para compressão ou descompressão de código de instrução é feito de forma separada, ou seja, na grande maioria dos trabalhos desenvolvidos só é tratado o hardware descompressor porque a compressão das instruções geralmente é feita por meio de modificações no compilador. Assim, a compressão é realizada em tempo de compilação e a descompressão é feita em tempo de execução usando um hardware específico para descompressão.

Para o funcionamento do método *MIC*, proposto neste trabalho, foi necessário o desenvolvimento de uma nova arquitetura, em hardware, que realize tanto a compressão quanto a descompressão dos códigos de instruções em tempo de execução. A arquitetura criada foi intitulada de PDCCM (*Processor Decompressor Cache Compressor Memory*) no qual é mostrado que o hardware de compressão foi inserido entre as memórias *cache* e principal e o hardware de descompressão foi inserido entre o processador e a memória *cache*. A arquitetura PDCCM foi implementada em VHDL e prototipada em uma FPGA *Cyclone-II* modelo EP2C20F484C7 do fabricante ALTERA®.

A arquitetura PDCCM trabalha com instruções do tamanho de 32 *bits*, ou seja, cada linha da *cache* de instrução é composta por 4 *bytes*. Assim, a arquitetura desenvolvida é compatível com sistemas que usam o processador ARM como núcleo do seu sistema embarcado, pois este processador tem como característica um conjunto de instruções de 32 *bits*. Na arquitetura PDCCM, usando o método de compressão/descompressão *MIC* todas as instruções que serão salvas na *cache* de instrução sofrerão uma compressão de 50% no seu tamanho original.

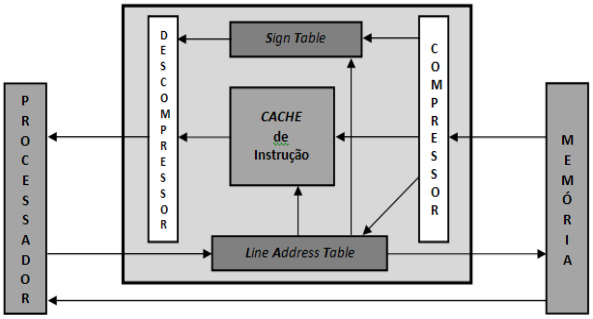

Figura 2. Arquitetura PDCCM

A Figura 2 mostra a arquitetura PDCCM desenvolvida para implantar o novo método de compressão/descompressão de instruções em hardware (*MIC*), que é composta por quatro componentes básicos, sendo eles:

- **LAT** (Tabela da Linha de Endereços - *Line Address Table*): é uma tabela que tem por função fazer o mapeamento dos endereços das instruções com o seu novo endereço na *cache* de instrução;
- **ST** (Tabela de Sinais - *Sign Table*): é uma tabela que contém *bits* que servem como *flags* para indicar ao descompressor qual dupla de *bits* deverá ser reconstituída, ou seja, descomprimida;
- **Compressor**: tem por função fazer a compressão de todos os códigos das instruções que serão salvas na *cache* de instrução. O compressor é acionado toda vez que houver um *miss* na *cache* de instrução;
- **Descompressor**: tem por função fazer a descompressão de todas as instruções que estão armazenadas na *cache* de instrução e serão repassadas ao processador. O descompressor é

acionado toda vez que houver um *hit* na *cache* de instrução.

4. Compressão em Tempo de Execução

Assim como a **tabela de *Huffman*** usada por WOLFE e CHANIN [15] na arquitetura do CCPR; a **tabela de instrução** e **tabela de tradução de endereço** usada por AZEVEDO [2] na arquitetura do IBC; o **dicionário de instruções** usada por BENINI *et al* [3] e o **dicionário de códigos** usado por LEKATSAS *et al* [8, 9] e LEFURGY *et al* [7], o método *MIC* também requer tabelas adicionais que serão usadas pelos componentes ST e LAT, para armazenarem o conjunto de *flags* da instrução comprimida e o mapeamento dos novos endereços das instruções comprimidas na *cache*, respectivamente.

Portanto, mesmo com o acréscimo adicional das tabelas (ST e LAT) no método *MIC*, que inicialmente são grandes ao ponto de parecer não ter tido ganho na compressão pois se requer uma tabela adicional equivalente à metade do tamanho da *cache* existente, e sendo a compressão de 50%, não há ganho no uso de memória, considerado como recurso físico. Apesar disso, essas tabelas foram inseridas para avaliarmos a estrutura da nova arquitetura e do método proposto, que levanta a opção de ter compressão em tempo de execução, sendo diferente das outras propostas que baseiam-se na construção de dicionários em tempo de compilação. Neste momento o grupo está trabalhando com o *MIC* no intuito de retirar (ou diminuir) essas tabelas adicionais e assim obter resultados mais expressivos num futuro próximo.

Para compressão, a cada instrução que for lida na memória RAM e salva na *cache* de instrução será dividida em duplas de *bits*, sendo cada dupla formada por: 00, 01, 10 e 11. O **compressor** *MIC* realiza a seguinte lógica: duplas com *bits* iguais (00 ou 11) são substituídas pelo *bit* 0 (zero) e duplas com *bits* diferentes (10 ou 01) são substituídas pelo *bit* 1 (um). Assim, uma dupla de *bits* é reduzida a um *bit* único.

Uma tabela auxiliar (ST) é usado para guardar o conjunto de *flags* da dupla de *bits* comprimidos. Em duplas de *bits* que iniciam com o valor 0 (zero), como no caso 00 ou 10, é salvo na ST o *bit* 0 e em duplas de *bits* que iniciam com o valor 1 (um), como no caso 11 ou 01, é salvo na ST o *bit* 1. Vale ressaltar que o modo de endereçamento das linhas de instrução para essa arquitetura é *Big-Endian*.

Portanto, o método *MIC* (*Middle Instruction Compression*) é um método de compressão que tem por função reduzir em 50% o tamanho dos códigos de instruções que são salvas na *cache* de instrução, passando então o tamanho dessas instruções de 32 *bits* (tamanho original) para 16 *bits* (tamanho comprimido).

Para um melhor esclarecimento e sempre que possível, os nomes dos componentes, variáveis e pinos de entrada e saída (*input* e *output*) são semelhantes aos usados no código implementado em VHDL.

4.1. Descrição do Método *MIC*

O processador requisita uma instrução à *cache* através de um pino, que para essa implementação foi chamado de `end_inst_proc` (PC atual). Será pesquisada na LAT a existência ou não do endereço fornecido pelo processador. Se a instrução for encontrada na *cache* a LAT sinalizará com um ACERTO (*hits*). Então, a LAT fornecerá o novo endereço da instrução na *cache* de instrução, o endereço do conjunto de *flags* da instrução na ST e o posicionamento da dupla de *bytes* (primeira ou segunda), ou seja, onde se encontra a instrução e os *flags* na *cache* de instrução e na ST, respectivamente. Todas essas informações são repassadas ao descompressor que descomprimirá a instrução e a retornará de forma descomprimida ao processador através da variável `returnD_inst_proc`.

A **descompressão dos códigos de instrução** é realizada da seguinte forma:

- O novo endereço da instrução que foi repassado pela LAT, foi localizado na *cache* de instrução e na ST;
- A *cache* de instrução e a ST retornam para o descompressor os 16 *bits* da instrução comprimida e os 16 *bits* do conjunto de *flags*;
- Se o *bit* lido da instrução comprimida na *cache* de instrução for 0 (zero), a dupla de *bits* a ser reconstituída será 00 ou 11. O que definirá qual será a dupla de *bits* é o *bit flag*, ou seja, se o *bit flag* for 0 a dupla de *bits* a ser reconstituída será 00 e se o *bit flag* for 1 a dupla de *bits* a ser reconstituída será 11;
- Mas se o *bit* lido da instrução comprimida na *cache* de instrução for 1 (um), a dupla de *bits* a ser reconstituída será 10 ou 01. Então, novamente o *bit flag* que definirá qual será a dupla de *bits*, ou seja, se o *bit flag* for 0 a dupla de *bits* a ser reconstituída será 10 e se o *bit flag* for 1 a dupla de *bits* a ser reconstituída será 01;
- Para cada instrução a ser descomprimida, são analisados os 16 *bits* da instrução salva na *cache* de instrução, transformando assim as instruções comprimidas de 16 *bits* em instruções descomprimidas de 32 *bits*.

Agora, se o endereço fornecido pelo processador não se encontrar na LAT, significa que não existe essa instrução na *cache* de instrução. A LAT sinalizará uma FALHA (*miss*) na linha da *cache* de instrução. O endereço fornecido pelo processador será repassado

para a memória RAM (*Random Access Memory*), onde a mesma será consultada e verificada se existe ou não essa instrução. Caso a pesquisa na RAM indique uma FALHA, a instrução será buscada em uma memória auxiliar *flash*. Agora se a pesquisa indicar um ACERTO significa que a instrução encontra-se na RAM. Em seguida, a RAM retornará uma cópia da instrução no formato original (não comprimida) para o processador através da variável `returnC_inst_proc` e outra cópia para o compressor, que realizará todo o processo de compressão.

A **compressão dos códigos de instrução** é realizada da seguinte forma:

- A instrução é localizada na memória RAM, uma cópia da mesma é repassada para o processador e outra para o compressor;
- A instrução no compressor é dividida em 16 duplas de *bits*, sendo que cada dupla é formada no instante que é lida pela função de compressão. O início da leitura da instrução vinda da memória RAM é pelo modo MSB (*bit* mais significativo);
- O compressor sempre analisará em qual parte da dupla de *bytes* (primeira ou segunda) deverá ser salva a instrução comprimida na *cache* de instrução e na ST;
- Se a dupla de *bits* lida para a compressão for 00 ou 11, então essa dupla de *bits* será substituída pelo *bit* 0 e salva na *cache* de instrução. Agora se a dupla de *bits* lida for 10 ou 01 então essa dupla de *bits* será substituída pelo *bit* 1 e salva na *cache* de instrução;
- O conjunto de *flags* da ST será formado através da seguinte lógica: se o primeiro *bit* da dupla de *bits* que está sendo comprimido for 0, então o *bit flag* salvo será 0. Agora se o primeiro *bit* da dupla de *bits* que está sendo comprimido for 1, então o *bit flag* salvo será 1;
- Após o compressor fazer toda a compressão dos 32 *bits* da instrução original nos 16 *bits* comprimidos e seus respectivos *bits flags*, o compressor irá salvar na dupla de *bytes* (primeira ou segunda) a instrução comprimida na *cache* de instrução e o conjunto de *flags* na ST;
- A tabela LAT será atualizada com o novo endereço da instrução salva na *cache* de instrução;
- Para cada instrução que for buscada na memória RAM, repetirá esse processo de compressão.

Importante salientar que essa técnica de compressão/descompressão é realizada em tempo de execução, via hardware específico que foi prototipado em FPGA. Na implementação em hardware verificou-se que é similar aos trabalhos de LEKATSAS [8, 9], obtivemos um componente que precisa de apenas um único ciclo para o processo de compressão ou descompressão, além dos benefícios mostrados na próxima seção.

Para uma análise de desempenho do método desenvolvido neste projeto (*MIC*), foi implementado em hardware o tradicional método de compressão de *Huffman*, pois o mesmo foi utilizado por WOLFE & CHANIN [15] na arquitetura do CCRP e também por BENINI *et al* [3] e LEFURGY *et al* [7]. Portanto, a comparação do *MIC* com *Huffman* permite verificar pontos fortes e fracos desta nova abordagem em compressão de códigos em relação a um método já conceituado e altamente usado no meio cientifico.

4.2. Implementação em FPGAs

Os *benchmark* utilizados nas simulações de compressão e descompressão dos métodos *MIC* e *Huffman* são do pacote *MiBench* [5] específicos para sistemas embarcados e de categorias diferentes, os quais estão em código Assembly do processador ARM9, conforme obtido através da ferramenta IDA Pro [17]. A categoria e funcionalidade dos *benchmark MiBench* usados nas simulações são:

- **Dijsktra** (Rede de Computadores): é um algoritmo que calcula o menor caminho em um grafo;
- **FFT** (Telecomunicação): é um algoritmo que executa a *Transformada de Fourier* usado no tratamento de sinais digitais para encontrar as frequências em um sinal de entrada;
- **MAD** (Dispositivos de Consumo): é um decodificador de áudio MPEG de alta qualidade;
- **QuickSort** (Controle Automotivo e Industrial): é um algoritmo que faz ordenação de dados;
- **SHA** (Segurança): é um algoritmo que gera chaves de criptografia para troca segura de dados e assinaturas digitais;
- **Stringsearch** (Automação): é um algoritmo que faz a busca de uma string em um texto.

Foi utilizado o conjunto de instruções do processador embarcado ARM (família ARM9, versão ARM922T, ISA ARMv4T) para simular o funcionamento do compressor e descompressor dos métodos *MIC* e *Huffman* na arquitetura PDCCM. No entanto, esse processador escolhido (ARM) é do tipo RISC e possui um conjunto de instruções formado por 32 *bits* (instrução) que o habilitou como sendo uma boa plataforma para simular a arquitetura PDCCM.

Vale ressaltar que a única alteração necessária na arquitetura PDCCM para o uso do método de *Huffman* foi a troca do componente da ST pela a HT (Tabela de *Huffman*) que contem a árvore de *Huffman* dos códigos das instruções comprimidas.

Para as simulações de compressão/descompressão dos métodos *MIC* e *Huffman*, foram selecionadas as 4.096 primeiras instruções de cada *MiBench* (devido as

limitações físicas do FPGA usado para a prototipação), obtidas através do código compilado (*Assembly*) para o processador embarcado ARM, formando assim o conjunto de sequências de instruções que foram usadas para carregar um trecho das memórias RAM e a *cache* de instrução. Para mais detalhes, ver em [16].

O trecho da memória RAM descrita em VHDL foi utilizada em todas as simulações com os *benchmark MiBench* e teve tamanho fixo de 4.096 linhas de 4 *bytes* cada, contabilizando assim 131.072 *bits* e a *cache* de instrução teve o tamanho de 512 linhas de 32 *bits* cada. Assim, se observa que há uma relação de 8:1 entre os tamanhos da memória RAM e da respectiva *cache* de instruções. A Tabela 1 mostra as métricas nas estatísticas de desempenho e na Tabela 2 é possível visualizar as métricas na temporização da arquitetura PDCCM no FPGA; ambas usando os métodos *MIC* e *Huffman* na compressão/descompressão das instruções de alguns programas do *MiBench*.

Tabela 1. Estatística de desempenho do FPGA

	MIC	***Huffman***
Compressão		
Elementos Lógicos	1.460 (32%)	1.317 (28%)
Registradores	825 (18%)	738 (16%)
Pinos	177 (56%)	170 (54%)
Descompressão		
Elementos Lógicos	3.464 (74%)	4.414 (94%)
Registradores	1.045 (23%)	1.482 (32%)
Pinos	177 (56%)	170 (54%)

Observando a Tabela 1, no processo de **compressão**, o método de *Huffman* mostrou ser um pouco mais eficiente no uso dos recursos do FPGA para a arquitetura PDCCM, sendo que a quantidade de elementos lógicos, registradores e pinos entre os dois métodos não passou de 4% de diferença a favor do método de *Huffman*. Já no processo de **descompressão**, o método *MIC* apresentou melhores resultados nos recursos do FPGA, sendo que a quantidade de elementos lógicos, registradores e pinos ficaram em média aproximadamente 9% a menos a favor do método *MIC*.

Tabela 2. Temporização do FPGA

	MIC	***Huffman***
Compressão		
Tempo no pior caso	9.228 ns	9.712 ns
Clock em MHz	75.98 MHz	69.88 MHz
Clock em tempo	13.257 ns	14.579 ns
Descompressão		
Tempo no pior caso	8.632 ns	10.708 ns
Clock em MHz	68.42 MHz	66.97 MHz
Clock em tempo	14.668 ns	14.900 ns

Nota-se na Tabela 2, que o método *MIC* apresentou melhor temporização no FPGA para todos os *benchmark MiBench* analisados. Na **compressão**, é observada uma diferença de 6.1 MHz (na frequência do *clock* em MHz) a mais para o método *MIC* sendo este um dos pontos que o torna mais eficiente que o método de *Huffman*. Já o tempo, no pior caso, para os dois métodos foram bem idênticos. Na **descompressão**, são observados que o *clock* em MHz e em tempo são idênticos para os dois métodos, sendo que o método *MIC* tem uma pequena vantagem comparada ao método de *Huffman*.

Tendo como base as primeiras 4K instruções dos programas, obtidas através do código *Assembly* compilado para a plataforma ARM, observamos na Tabela 3 que o método *MIC* comprimiu em 50% o tamanho das instruções, ou seja, após o processo de compressão o programa passou de 4K para apenas 2K linhas na *cache* de instrução. Enquanto isso o método de *Huffman* obteve uma média geral na compressão de 30% a menos em relação ao tamanho da memória RAM usada na simulação.

Tabela 3. Comparativo na taxa de compressão

MiBench	***MIC***	***Huffman***
Dijsktra	2.048 (50%)	2.622 (36%)
FFT	2.048 (50%)	2.803 (32%)
MAD	2.048 (50%)	3.002 (27%)
QuickSort	2.048 (50%)	3.245 (21%)
SHA	2.048 (50%)	2.785 (32%)
StringSearch	2.048 (50%)	2.913 (29%)
Médias	**2.048 (50%)**	**2.895 (30%)**

Com base nos resultados apresentados, constatamos que para a arquitetura PDCCM usando as 4.096 primeiras instruções dos *benchmarks MiBench* (Dijsktra, FFT, MAD, QuickSort, SHA e StringSearch), o método *MIC* mostrou-se mais eficiente na taxa de compressão, ou seja, um percentual de ≈42% menor (para o tamanho do código final) se comparado ao método de *Huffman* para todos os *MiBenchs* analisados.

5. Conclusões e Trabalhos Futuros

Este artigo descreveu um novo método de compressão, chamado de *MIC*, o qual foi prototipado em FPGAs, e mostrou-se viável para os sistemas embarcados que usam arquitetura RISC. Pois futuramente esta técnica pode tornar-se um componente necessário em projetos de sistemas embarcados. Com a utilização das técnicas de compressão de código, as arquiteturas RISC conseguem minimizar um de seus maiores problemas, que é a quantidade de memória para armazenar os programas.

Mediante as simulações realizadas com alguns programas do *benchmark MiBench* verificamos que o

método *MIC* apresentou as seguintes médias: 17% a menos na utilização dos elementos lógicos do FPGA, uma frequência (MHz) de operação em aproximadamente 6% maior para os processos de compressão/descompressão dos códigos de instruções e 42% mais eficiente na taxa de compressão dos *MiBench* analisados em relação ao método de *Huffman*, que também foi prototipado em hardware.

Portanto, analisando os dados obtidos por meio das simulações, apesar da memória adicional usada pelo dicionário (ST) do método *MIC*, conclui-se que o método desenvolvido e apresentado neste artigo mostrou ser mais eficiente computacionalmente em comparação com o método de *Huffman* também implementado em hardware. As simulações utilizaram aplicações de categorias diferentes, sendo eles: Dijsktra, FFT, MAD, QuickSort, SHA e StringSearch do *benchmark MiBench* para obter as medições de desempenho.

Como trabalhos futuros ficam: realizar medições reais do consumo de energia da arquitetura PDCCM usando os métodos *MIC* e *Huffman*; projetar e implementar um processador de arquitetura RISC que já tenha o hardware compressor e descompressor embutido em seu núcleo; realizar testes de compressão e descompressão usando os métodos *MIC* e *Huffman* com mais programas do *benchmark MiBench* e chegar a uma implementação em ASIC, de modo que este projeto transcenda o âmbito acadêmico, servindo como uma contribuição, também, para o meio industrial.

6. Referências

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

[2] R. J. de Azevedo. *Uma Arquitetura para Código Comprimido em Sistemas Dedicados*. Tese de Doutorado, Instituto de Computação da UNICAMP, Brasil, Junho de 2002. 136 p.

[3] L. Benini, A. Macii and A. Nannarelli. "Cached-Code Compression for Energy Minimization in Embedded Processor". *In Proceedings of the International Symposium on Low-Power, Electronics and Design (ISPLED'01)*. Huntington Beach, California, USA, August 2001, pp. 322-327.

[4] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong. *Overview of the Ptolemy Project*, ERL Technical Memorandum UCB/ERL, Technical Report Nº M-99/37, Department of Electrical Engineering and Computer Science, Univ. of California, Berkeley, California, USA, July 6, 1999.

[5] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite". *In Proceedings of the IEEE 4th Annual Workshop on Workload Characterization (WWC-4)*. Austin, Texas, USA, December 2001, pp. 3-14.

[6] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". *Proceedings of the Institute of Radio Engineers (IRE)*, 40(9):1098-1101, September 1952.

[7] C. Lefurgy, P. Bird, I-C. Chen and T. Mudge. "Improving Code Density Using Compression Techniques". *In Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO 30)*. Research Triangle Park, NC, USA, December 1997, pp. 194-203.

[8] H. Lekatsas, J. Henkel and V. Jakkula. "Design of One-Cycle Decompression Hardware for Performance Increase in Embedded Systems". *In Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. New Orleans, Louisiana, USA, June 2002, pp. 34-39.

[9] H. Lekatsas and W. Wolf. "Code Compression for Embedded Systems". *In Proceedings of the 35th Annual Design Automation Conference (DAC'98)*. San Francisco, California, USA, June 1998, pp. 516-521.

[10] E. B. W. Netto. *Compressão de Código Baseada em Multi-Profile*. Tese de Doutorado, Instituto de Computação, Universidade Estadual de Campinas, Maio de 2004. 137 p.

[11] E. B. W. Netto, R. Azevedo, P. Centoducatte and G. Araújo. "Mixed Static/Dynamic Profiling for Dictionary Based Code Compression". *In Proc. of the Intl. Symposium on System-on-Chip (SoC'03)*. Tampere, Finland, November 2003, pp. 159-163.

[12] E. B. W. Netto, R. Azevedo, P. Centoducatte and G. Araújo. "Multi-Profile Based Code Compression". *In Proceedings of the 41th Annual Design Automation Conference (DAC'04)*. San Diego, California, USA, June 2004, pp. 244-249.

[13] E. B. W. Netto, R. S. de Oliveira, R. Azevedo, P. Centoducatte. *Compressão de Código em Sistemas Embarcados*. HOLOS CEFET-RN. Ano 19, páginas 23-28, Dezembro, 2003. 94p.

[14] A. S. de Oliveira, F. S. de Andrade. *Sistemas Embarcados - Hardware e Firmware na Prática*. Editora Érica, 2006, 316p.

[15] A. Wolfe and A. Chanin. "Executing Compressed Programs on an Embedded RISC Architecture". *In Proceedings of 25th Annual Intl. Symposium on Microarchitecture (MICRO 25)*. Portland, Oregon, USA, December 1992, pp. 81-91.

[16] W. R. A. Dias. *Arquitetura PDCCM em Hardware para Compressão/Descompressão de Instruções em Sistemas Embarcados*. Dissertação de Mestrado, Departamento de Ciência da Computação da UFAM, Brasil Abril de 2009. 152 p.

[17] IDA - The Interactive Disassembler. Disponível em: http://www.hex-rays.com. Acessado em 03 de maio de 2010.