# Três Novos Métodos de Compressão Código usando Padrões de Blocos e Dicionário Multi-Nível

Wanderson Roger Azevedo Dias
Instituto de Computação – IComp
Universidade Federal do Amazonas – UFAM
Manaus, Amazonas, Brasil
wradias@gmail.com

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

***Resumo*—*Este trabalho propõe e implementa três novos métodos de compressão de código que são simples e eficientes. No artigo também se discute aspectos da implementação do respectivo hardware descompressor. Além disso, um novo tipo de dicionário, que é dividido em níveis, é introduzido neste artigo. O paradigma aplicado para este novo dicionário consiste em armazenar, ao mesmo tempo, instruções unitárias e padrões de blocos encontrados no código do programa, por esta razão, é chamado de Dicionário Multi-Nível. Nas simulações dos algoritmos desenvolvidos, usou-se alguns programas do benchmark MiBench, além de dois processadores embarcados de 32 bits (ARM e MIPS). A taxa média de compressão obtida em nossos métodos chegou a 34,5%. Assim, os algoritmos propostos oferecem uma melhor exploração da tríade compressão-desempenho-consumo.*

***Palavras-Chave*—*sistema embarcado; compressão de código; instrução unitária; padrão de bloco, dicionário multi-nível.***

## I. INTRODUÇÃO

Os sistemas de processamento de informação que estão incorporados em um produto maior e que normalmente não estão diretamente visíveis aos usuários são chamados de sistemas embarcados [2] e colaboram com esta nova tendência da computação, chamada de *disappearing computer* [11], que tem o princípio de que a computação acontece em todo lugar (computação ubíqua), porém de forma invisível, sendo realizados em dispositivos com aparências não tradicionais e cuja presença muitas vezes não se consegue identificar.

Então, define-se que os sistemas embarcados são quaisquer sistemas digitais que estejam incorporados a outros sistemas com a finalidade de acrescentar ou otimizar funcionalidades [2, 8]. Assim, os sistemas embarcados têm por função colaborar com tarefas de monitoramento e/ou controle do ambiente no qual esteja inserido.

Os sistemas embarcados apresentam inúmeras limitações físicas e de recursos computacionais, sendo a memória um dos recursos mais críticos [16]. Nos projetos de sistemas embarcados este recurso usualmente representa um dos componentes mais caros. Então, tudo isto justifica o esforço para otimizar o seu uso. Uma das técnicas desenvolvidas para isso é a compressão do código.

À medida que os sistemas embarcados tornam-se mais heterogêneos os mesmos admitem maior complexidade em seu desenvolvimento. Por outro lado, os softwares que neles operam também ampliam o seu grau de complexidade, causando assim o aumento em seu código e o respectivo uso de memória [16]. Desta forma, técnicas de compressão de código têm sido desenvolvidas com o intuito de reduzir o tamanho do código das instruções de uma aplicação. A taxa de compressão (TC) é amplamente aceita como a métrica para medir a eficiência de um método de compressão e é definida conforme a Equação 1.

$$TC = 100 - \left(\frac{Taman\ ho\ do\ código\ \ comprimido\ \ + Dicion\ ário}{Taman\ ho\ do\ código\ \ original}\ x\ 100\right)\% \quad (1)$$

Na busca pelo aperfeiçoamento das técnicas existentes, surge a necessidade de criar novas estratégias capazes de aumentar a capacidade de armazenamento de códigos na hierarquia de memória, visando melhorar o desempenho computacional em termos de tempo de execução assim também como reduzir o consumo de energia nos sistemas embarcados.

A principal contribuição deste artigo é propor e implementar três novos métodos de compressão de código para sistemas embarcados que sejam eficientes computacionalmente na tríade compressão-desempenho-consumo, nomeados de HDPB, CCHPB e CC-MLD. Para isso, temos pensado em métodos simples, porém eficientes, gerando menos *overhead* computacional no processo de descompressão assim como facilidade de implementação do respectivo hardware descompressor, trazendo menos *overhead* na fase de decodificação. Para isso, criou-se um novo tipo de dicionário, dividido em níveis, que permite armazenar instruções unitárias e padrões de blocos encontrados em instruções adjacentes. Os três métodos usam o algoritmo de *Huffman* na descoberta das instruções redundantes. O artigo apresenta detalhes do descompressor e estatísticas do projeto em FPGA.

O restante do artigo está organizado da seguinte forma: a Seção II apresenta os trabalhos correlatos; a Seção III apresenta a redundância das instruções; a Seção IV detalha os métodos HDPB, CCHPB e CC-MLD bem como suas simulações e análises; a Seção V apresenta o hardware descompressor do método CC-MLD e a Seção VI finaliza com as conclusões e ideias para trabalhos futuros.

## II. TRABALHOS CORRELATOS

Apesar de existirem diversos métodos para compressão de código, neste artigo apresentamos apenas alguns métodos de compressão que usam o algoritmo de *Huffman* e são baseados nas arquiteturas dos processadores embarcados ARM e MIPS.

Wolfe & Chanin [3] desenvolveram o CCRP (*Compressed Code RISC Processor*), que foi o primeiro hardware descompressor implementado em um processador RISC, MIPS

R2000, que usava as falhas de acesso à *cache* para acionar o mecanismo de descompressão. No CCRP os modelos dos programas são inalterados uma vez que tem uma arquitetura idêntica ao padrão do processador RISC. 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. A técnica CCRP utilizou o método de *Huffman* e mostrou uma taxa de compressão de 27%, em média, para seis programas do SPEC'95.

Haider & Nazhandali [12] desenvolveram um novo método de codificação híbrido, combinando a tradicional codificação baseada em *bitmask* e a codificação baseada em prefixo de *Huffman*, gerando assim um novo dicionário e uma nova técnica de seleção de instruções (algoritmo *non-greedy)*. A compressão baseada em *bitmask* gera a compressão baseada nas frequências e aproveita a curta distância de *hamming* entre algumas instruções. No método desenvolvido existem dois tipos distintos de instruções comprimidas, sendo: (i) as instruções de entrada no dicionário (DE) e (ii) as instruções de *children* no dicionário (DC). A *bitmask* contém informações mais detalhadas sobre as máscaras incluindo a sua localização e a alternância necessária entre os *bits* reais, conforme se explica em [12]. Os autores conseguiram uma taxa de compressão de 9% para os dicionários com 4k e 8k entradas (instruções) e 20% para os dicionários com 512k e 1k.

Bonny & Henkel [15] desenvolveram um método de compressão de código onde as instruções originais são divididas em padrões de comprimento variável e depois é aplicada a codificação de *Huffman* para os padrões obtidos. As novas instruções obtidas podem ser classificadas em dois tipos: (i) instruções que não se repetem dentro do aplicativo (frequência=1), chamadas de "instruções únicas"; (ii) instruções que se repetem dentro do aplicativo (frequência>1), também chamadas de "instruções repetidas". O algoritmo de divisão encontra a maioria dos padrões repetitivos de instruções entre todas as outras instruções. Os padrões podem ter comprimentos diferentes, sendo entre 2 até 32 *bits*. Depois disso, os novos códigos são comprimidos usando a codificação de *Huffman* [4].

Nesse método de compressão [15], as entradas da tabela de decodificação dos códigos de comprimento são variáveis e os índices (ou seja, as instruções codificadas) para essas entradas também possuem comprimento variável. Isto tornou difícil a decodificação quando foi implementada em hardware. Além disso, o tamanho necessário para armazenar a tabela de decodificação da memória é grande. Para resolver estes problemas, os autores usaram a *Codificação Canônica de Huffman* [13], como feito em [15]. Usando esta nova estratégia, os autores constataram que a decodificação tornou-se mais eficiente em espaço e tempo, porque requer que menos informações sejam armazenadas para a respectiva decodificação. O método desenvolvido é independente de qualquer arquitetura e de seu conjunto de instruções. As taxas de compressão alcançadas para os processadores ARM e MIPS foram de 47% e 49% respectivamente, usando 7 programas do *MiBench*.

Collin & Brorsson [10] desenvolveram um novo método de compressão de código que usa dois dicionários, um para lidar com a compressão de instruções individuais e o outro com a compressão de sequências de instruções no código. Os dois dicionários estão em estágio diferente do *pipeline* da arquitetura e trabalham juntos para descomprimir as instruções individuais e as sequências de instruções. Os autores afirmaram que o impacto dos dois dicionários sobre o tamanho total do armazenamento é pequeno, porém, as sequências no dicionário são armazenadas como instruções individuais em vez de instruções normais. O método possui dois tipos de *codewords* sendo: (i) *codeword* de instrução (ICW) e (ii) *codeword* de sequência (SCW). De acordo com Collin & Brorsson [10], sempre que a *cache* de instrução é acessado, assume-se que uma busca de palavra de 32 *bits* é obtida. Dependendo do código comprimido, um dos três casos acontece: (i) caso I, uma palavra buscada pode conter até quatro palavras de sequência; (ii) caso II, até três palavras de instrução de código ou (iii) caso III, uma instrução descomprimida.

É importante que as instruções descomprimidas possam coexistir com as instruções comprimidas com apenas uma pequena fração de todas as instruções que constituem o programa inteiro. Os autores fizeram alterações na ferramenta *SimpleScalar* e usou-se o conjunto de instruções ISA da plataforma MIPS IV juntamente com 15 programas do *benchmark MediaBench* para avaliar o desempenho desse novo método de compressão. Os resultados obtidos nos experimentos mostraram uma taxa de compressão média de 23% na compressão dinâmica e uma redução de 2% a 21% no consumo de energia da arquitetura.

## III. Redundância das Instruções e Dicionário Multi-Nível

O pequeno conjunto de instruções das arquiteturas RISC aliado à regularidade do código gerado pelos compiladores faz com que os programas possuam certa quantidade de instruções e/ou padrões de instruções que se repetem com alguma frequência nos códigos. Seguindo essa premissa, esta seção apresenta resultados prévios das análises realizadas que exemplificam e justificam o uso dos padrões de blocos pelos métodos de compressão de código propostos e desenvolvidos neste artigo.

A escolha dos tipos de instruções (unitária ou padrão de bloco) que compõem o dicionário usado por cada um dos métodos foi obtida após análises realizadas baseadas em informações estáticas dos códigos dos programas (*profile* estático). Por outro lado, fez-se necessário realizarmos alterações no código fonte do simulador arquitetural *SimpleScalar* (mais específico o simulador orientado à execução *Sim-outorder*) para assim obtermos tais informações. Usamos os programas do *benchmark MiBench* como entrada para as simulações e geramos relatórios com dados que justificam o uso dos padrões de blocos no processo de compressão dos códigos. A Tabela I mostra a contagem total das instruções, das instruções únicas e as repetições encontradas para cada programa do *MiBench*.

TABELA I. QUANTIDADE DE INSTRUÇÕES NO CÓDIGO DOS PROGRAMAS

| | Total de Instruções | | Únicas (%) | | Repetidas (%) | |
|---|---|---|---|---|---|---|
| **MiBench** | **ARM** | **MIPS** | **ARM** | **MIPS** | **ARM** | **MIPS** |
| bitcount | 10.132 | 11.013 | 3.060<br>30,2% | 3.072<br>27,9% | 7.072<br>69,8% | 7.941<br>72,1% |
| lame | 52.274 | 61.149 | 13.736<br>26,3% | 15.503<br>25,4% | 38.538<br>73,7% | 45.646<br>74,6% |
| patricia | 13.754 | 14.108 | 4.081<br>29,7% | 3.581<br>25,4% | 9.673<br>70,3% | 10.527<br>74,6% |
| stringsearch: small | 10.251 | 10.191 | 3.220<br>31,4% | 2.986<br>29,3% | 7.031<br>68,6% | 7.205<br>70,7% |
| stringsearch: large | 10.245 | 10.187 | 3.214<br>31,4% | 2.982<br>29,3% | 7.031<br>68,6% | 7.205<br>70,7% |
| sha | 9.822 | 10.454 | 3.063<br>31,2% | 2.949<br>28,2% | 6.760<br>68,8% | 7.505<br>71,8% |
| fft | 24.406 | 13.174 | 0<br>0% | 3.499<br>26,6% | 24.406<br>100% | 9.675<br>73,4% |
| **Média** | **18.697** | **18.610** | **4.339**<br>**23,2%** | **4.938**<br>**26,5%** | **14.358**<br>**76,8%** | **13.672**<br>**73,5%** |

Analisando os valores da Tabela I destaca-se que em média, apenas 24,8% das instruções dos códigos são únicas, o que indica a possibilidade de existir padrões de blocos no código dos programas, levando ao desenvolvimento de novos métodos que venham a explorar tal característica dos programas. Assim, o impacto na taxa de compressão com a substituição de um padrão de bloco por uma instrução de tamanho menor (*codeword*) provavelmente torna-se maior do que a substituição de uma única instrução por uma *codeword*.

Ainda vale ressaltar que os métodos de compressão que utilizam a abordagem de *profile* estático na criação do dicionário, em geral, obtêm maiores taxas de compressão, em relação aos métodos que utilizam a abordagem de *profile* dinâmico, os quais apresentam maior ganho de desempenho e/ou menor consumo de energia, conforme [7]. Portanto, o que se espera de um algoritmo de compressão de código é que ele possa fazer uso destas redundâncias das instruções nos programas para diminuir o tamanho do código e, por conseguinte, aumentar o desempenho e reduzir o consumo de energia nos sistemas embarcados.

Os resultados da Tabela I induzem a pensar na possibilidade de ter múltiplos dicionários na compressão de código. Assim, nesse trabalho desenvolvemos um novo tipo de dicionário (chamado de *Dicionário Multi-Nível*) composto por vários níveis, onde em cada nível do dicionário é possível armazenar tipos diferentes de instruções comprimidas no código dos programas. A Figura 1 exemplifica um dicionário multi-nível (composto por dois níveis), sendo que o nível 1 armazena padrões de blocos formados por duas instruções adjacentes (devido às instruções repetidas) e o nível 2 apenas instruções unitárias.

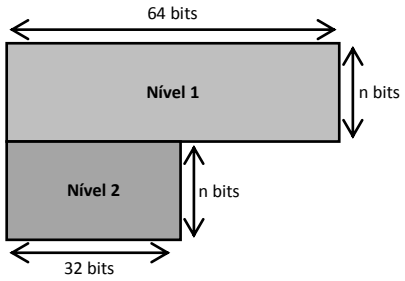


Fig. 1. Exemplo do dicionário multi-nível.

## IV. MÉTODOS BASEADOS EM PADRÕES DE BLOCOS

Os três métodos de compressão de código propostos são: **HDPB** (*Huffman and Dictionary-Based Pattern Blocks*), **CCHPB** (*Huffman + Pattern Blocks and Multi-Level Dictionary*) e **CC-MLD** (CC *Huffman-Based Multi-Level Dictionary*). Eles foram implementados em software (Linguagem C) e tentam localizar as instruções que se repetem com maior frequência no código dos programas (tanto unitárias quanto padrões de blocos) e atribuem uma *codeword* de menor tamanho a essas instruções, utilizando um dicionário multi-nível para armazenar as instruções que são comprimidas. De forma similar ao trabalho de Lekatsas *et at* [9], as *codewords* geradas após usar o algoritmo de *Huffman*, possuem comprimento fixo. Isso simplifica a lógica da descompressão em acessar o dicionário multi-nível e reduz a latência do processo da descompressão, e facilita a implementação em hardware, do respectivo descompressor, nos três métodos HDPB, CCHPB e CC-MLD.

Uma das características destes métodos é alterar o mínimo possível o projeto do processador de um sistema embarcado. Por esse motivo, se escolheu a arquitetura CDM (*Cache Decompressor Memory*), uma vez que é possível obter melhores resultados na taxa de compressão sem afetar o projeto do processador e do sistema embarcado. Neste artigo usamos os processadores embarcados: ARM (modelo ARMv5, versão SA-1110) [5] e MIPS (modelo MIPS-I, versão R3000) [6]. Estes processadores já se encontram modelados na *SimpleScalar*. E para análise e validação desses métodos, foram realizadas simulações com programas das seis categorias do *MiBench* (ver Tabela I), específicos para sistemas embarcados.

A Figura 2 mostra detalhadamente os passos executados pelo compressor de código desses métodos.

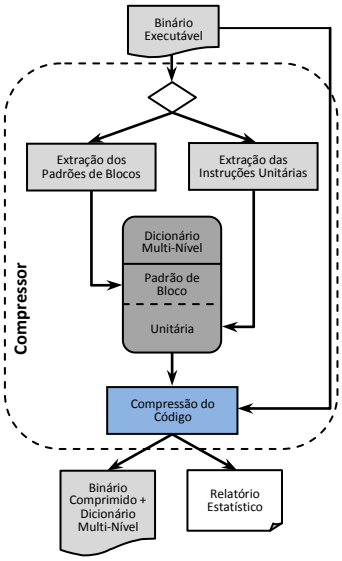


Fig. 2. Passos executados pelo compressor de código.

### A. Método HDPB

Com o objetivo de alcançar uma maior taxa na compressão do código dos programas, o método HDPB implementa três técnicas semelhantes que são usadas sequencialmente, a saber:

(i) compressão usando *Huffman* em instruções unitárias, que usa o nível 1 do dicionário; (ii) compressão usando *Huffman* em padrões de blocos nas *codewords* geradas pela técnica-1, que usa o segundo nível do dicionário e por fim, (iii) compressão usando *Huffman* em padrões de blocos (formados por duas instruções) em instruções que não foram comprimidas pelas técnicas anteriores, usando o terceiro nível do dicionário.

*1) Técnica-1 – Compressão de Huffman em Instruções Unitárias*: A ideia básica é atribuir códigos menores de *bits* para Instruções Unitárias (INU) que se repetem com maior frequência no código dos programas. O processo de compressão primeiramente gera uma tabela com as INUs e suas respectivas frequências de repetições. Esta tabela é ordenada de forma decrescente pelo campo frequência. Em seguida, cria-se o primeiro nível do dicionário. O código original do programa é lido linha a linha, e a cada INU encontrada neste código e que também esteja inserida no dicionário nível 1 é realizada a troca pela *codeword* correspondente. Este processo se repete para todas as instruções do programa original. Assim, ao final do processo um novo código parcialmente comprimido é obtido juntamente com o primeiro nível do dicionário.

*2) Técnica-2 – Compressão de Huffman em Padrões de Blocos nas Codewords*: O processo de compressão nesta técnica é semelhante ao processo realizado antes mencionado, usa-se como entrada para análise da compressão o arquivo pré-comprimido que foi gerado pela técnica-1. Nesta técnica, o algoritmo procura apenas padrões de blocos nas *codewords* originadas pela técnica-1. Novamente uma tabela ordenada de forma decrescente é gerada para guardar todos os padrões de blocos das *codewords* e suas respectivas frequências de repetições no código. Assim, o dicionário nível 2 é formado e o processo de compressão inicia executando novamente a análise linha a linha. Ao término do processo de compressão um novo código pré-comprimido é obtido juntamente com o segundo nível do dicionário multi-nível.

*3) Técnica-3 – Compressão de Huffman em Padrões de Blocos*: Com base nas evidências expostas na Tabela I, o objetivo desta técnica é localizar os padrões de blocos de instruções que mais se repetem no código do programa e que ainda não foram comprimidas pela técnica-1 e assim gerar uma nova instrução de tamanho reduzido (*codeword*). Igualmente nas técnicas 1 e 2, é criado um novo nível do dicionário (nível 3) contendo as instruções dos padrões de blocos encontradas no código pré-comprimido. A cada duas ou mais instruções sequenciais ainda não comprimidas no código pré-comprimido é formado um novo padrão de bloco que é armazenado em um *buffer* e em seguida é consultado se o mesmo está ou não contido no terceiro nível do dicionário; se estiver as instruções que formaram o padrão de bloco são substituídas pela *codeword* correspondente, senão permanecem como estão, ou seja, descomprimidas. Este processo se repete para todas as instruções do programa pré-comprimido. Ao término desse processo, o código final comprimido é obtido juntamente com o dicionário multi-nível.

Para todas as instruções do código é preciso identificar se houve ou não a compressão da mesma, assim, usa-se um identificador (Id) de tamanho fixo para esta identificação:

- $00_2$: instruções que não foram comprimidas no código;
- $01_2$: instruções comprimidas pela técnica-1;
- $10_2$: instruções comprimidas pela técnica-2;
- $11_2$: instruções comprimidas pela técnica-3.

Para a escolha do tamanho do dicionário multi-nível que obtém a maior taxa de compressão, foram realizadas simulações (variando os tamanhos para cada nível entre 128 até 2048 entradas) para encontrar qual o melhor tamanho para cada um dos níveis do dicionário. A Tabela II e a Figura 3 apresentam resultados das simulações da compressão de código usando *Huffman* em instruções unitárias (U) e padrões de blocos formados com duas (PB-2), três (PB-3) e quatro (PB-4) instruções, e a Tabela III mostra um resumo das médias nas taxas de compressão apresentadas na Tabela II.

TABELA II. TAXA DE COMPRESSÃO USANDO HUFFMAN

| | | Tamanho do primeiro nível do dicionário (entradas) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **Processador** | **128** | **256** | **512** | **768** | **1024** | **1280** | **1536** | **2048** |
| **U** | **ARM** | 16,2% | 21,1% | 25,5% | 26,7% | **28,1%** | 27,1% | 27,2% | 26,4% |
| | **MIPS** | 17,2% | 21,7% | 25,9% | 27% | **28,3%** | 27,1% | 27,5% | 26,7% |
| **PB-2** | **ARM** | 9,6% | 12,4% | **14,4%** | 13,7% | 11% | 7,2% | 3,9% | -2,7% |
| | **MIPS** | 7,8% | 10,6% | 12,6% | **12,8%** | 10,3% | 6,1% | 2,3% | -5,3% |
| **PB-3** | **ARM** | 5,4% | **6,6%** | 4,8% | -0,2% | -4,9% | -10% | -14,7% | -24,8% |
| | **MIPS** | 4,4% | **6,1%** | 5,4% | -0,2% | -5,9% | -11,9% | -17,7% | -29,3% |
| **PB-4** | **ARM** | 3,8% | **4,8%** | -1,2% | -7,7% | -14,3% | -21,1% | -27,8% | -41,2% |
| | **MIPS** | 3,2% | **4,3%** | -2,4% | -10,2% | -17,9% | -25,7% | -33,5% | -49% |

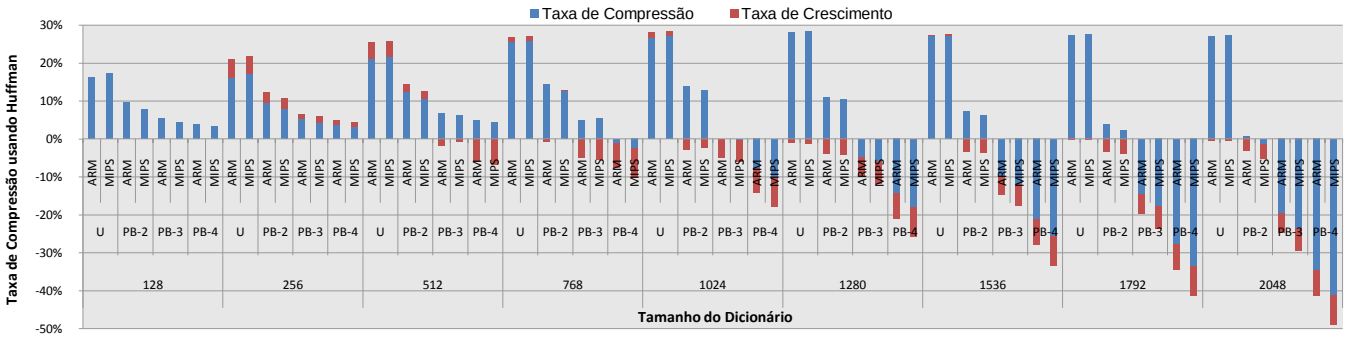


Fig. 3. Taxas de compressão e crescimento usando Huffman.

TABELA III. MÉDIA DAS TAXAS DE COMPRESSÃO USANDO HUFFMAN

| | Tamanho do primeiro nível do dicionário (entradas) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **128** | **256** | **512** | **768** | **1024** | **1280** | **1536** | **2048** |
| **U** | 16,7% | 21,4% | 25,7% | 26,8% | **28,2%** | 27,1% | 27,3% | 26,5% |
| **PB-2** | 8,7% | 11,5% | **13,5%** | 13,2% | 10,6% | 6,6% | 3,1% | -4% |
| **PB-3** | 4,9% | **6,3%** | 5,1% | -0,2% | -5,4% | -10,9% | -16,2% | -27% |
| **PB-4** | 3,5% | **4,5%** | -1,8% | -8,9% | -16,1% | -23,4% | -30,6% | -45,1% |

Observando a Figura 3 conclui-se que os padrões de blocos formados com três e quatro instruções não apresentaram resultados satisfatórios, sendo que na maioria das vezes as taxas de compressão e crescimento foram até negativas. Portanto, concluímos que o uso de padrões de blocos formados com três ou quatro instruções não são adequados para serem comprimidos e armazenados em dicionário multi-nível.

Após simulações, constata-se que o melhor tamanho para o dicionário multi-nível usando o método HDPB é quando os níveis 1, 2 e 3 do dicionário tiveram 1024, 128 e 256 entradas, respectivamente, alcançando uma taxa de compressão de 27,9%. A Tabela IV apresenta uma sumarização das médias na taxa de compressão para cada nível do dicionário multi-nível e a Figura 4 apresenta detalhadamente a taxa de compressão obtida pelo método HDPB.

TABELA IV. MÉDIA GERAL NA TAXA DE COMPRESSÃO DO HDPB

| | | Nível 1 | |
|---|---|---|---|
| **Nível 2** | **Nível 3** | **512** | **1.024** |
| **128** | **256** | 27% | **27,9%** |
| | **512** | 23,5% | 24,3% |
| **256** | **256** | 24,7% | 25,5% |
| | **512** | 24,6% | 25,6% |

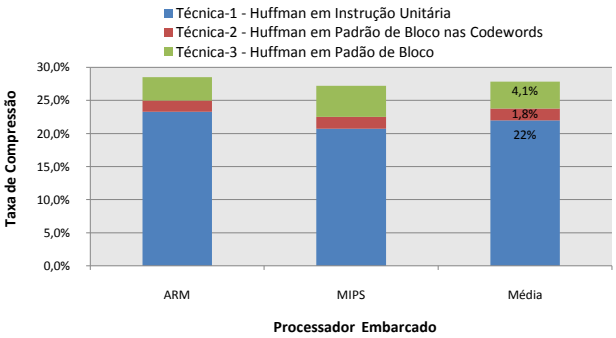


Fig. 4. Taxa de compressão obtida pelo método HDPB.

Analisando os valores das Tabelas IV e III (apenas as instruções unitárias, ou seja, U), constata-se que o método HDPB não apresentou nenhuma configuração que obtivesse uma taxa de compressão superior às taxas de compressão apresentadas na Tabela III (*Huffman*), dando origem ao método CCHPB.

## B. *Método CCHPB*

O método *Compressed Code using Huffman + Pattern Blocks and Multi-Level Dictionary* (CCHPB) é basicamente idêntico ao método HDPB. Após algumas análises verificou-se que alterando a ordem sequencial da execução das técnicas, o resultado alcançado na taxa de compressão pode ser melhorado. Então, o método CCHPB primeiramente executa a técnica de compressão usando *Huffman* em padrões de blocos (formados com duas instruções), que usa o nível 1 do dicionário, em seguida executa-se a técnica de compressão usando *Huffman* em instruções unitárias, que usa o segundo nível do dicionário e por último a técnica de compressão usando *Huffman* em padrões de blocos nas *codewords* geradas pela nova técnica-2, que usa o terceiro nível do dicionário.

As mesmas análises feitas no método HDPB são mantidas para este método, assim, constata-se que a melhor configuração para o dicionário multi-nível usando o método CCHPB é quando os níveis 1, 2 e 3 do dicionário possuem os tamanhos 512, 1024 e 128 entradas, respectivamente, obtendo uma taxa de compressão média de 34,5% para os programas do *MiBench*. A Tabela V apresenta as médias na taxa de compressão para os melhores tamanhos dos níveis do dicionário multi-nível e a Figura 5 mostra a taxa de compressão obtida por cada uma das técnicas do método CCHPB.

TABELA V. MÉDIA GERAL NA TAXA DE COMPRESSÃO DO CCHPB

| | | Nível 1 | | |
|---|---|---|---|---|
| **Nível 2** | **Nível 3** | **256** | **512** | **768** |
| **512** | **128** | 31,8% | 32,1% | 32,1% |
| | **256** | 31,6% | 31,9% | 31,9% |
| | **512** | 31,6% | 31,8% | 31,8% |
| **768** | **128** | 32,9% | 33,1% | 33,1% |
| | **256** | 32,7% | 32,9% | 32,9% |
| | **512** | 32,6% | 32,8% | 32,8% |
| **1024** | **128** | 31,8% | **34,5%** | 31,8% |
| | **256** | 31,6% | 31,8% | 31,6% |
| | **512** | 31,5% | 31,7% | 31,5% |

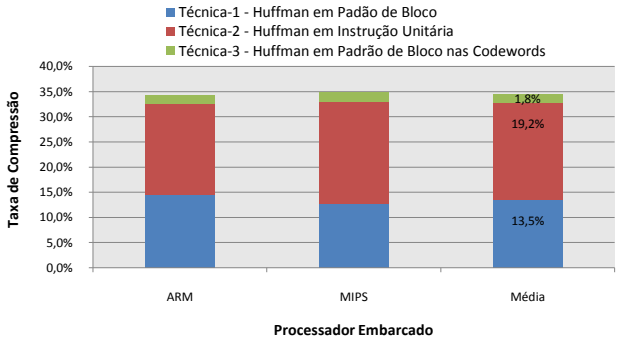


Fig. 5. Taxa de compressão obtida pelo método CCHPB.

Verificando os valores das Tabelas IV e V, destacamos que apenas fazendo uma reordenação na sequência de execução das técnicas desenvolvidas para o método HDPB foi possível obteruma melhora na taxa de compressão, e a melhor configuração (1024, 128 e 256 para os níveis 1, 2 e 3) do dicionário multi-nível para o método HDPB obteve uma média na taxa de compressão de 27,9%, comparando (com o mesmo tamanho de dicionário) ao método CCHPB que obteve uma média de 31,8% esta diferença foi de 3,9% a favor do método CCHPB. Este valor foi ainda mais significativo quando a comparação foi feita de forma inversa, sendo agora que a melhor configuração (512, 1024 e 128 para os níveis 1, 2 e 3) para o dicionário multi-nível do método CCHPB obteve uma taxa média de compressão de 34,5% e para o método HDPB esta taxa média foi de 24,3%, atingindo assim uma diferença de 10,2% a mais para o método CCHPB.

### C. *Método CC-MLD*

Na Figura 5 constata-se que a técnica-3 do método CCHPB contribuiu apenas com 1,8% na taxa final da compressão do código dos programas do *MiBench*, sendo este um valor quase insignificante e o que induza pensar que esta técnica-3 produz apenas mais complexidade ao método proposto, gerando assim uma possível perda de desempenho do sistema devido ao processo da descompressão das *codewords* comprimidas. Então, mediante essa observação, uma forma de otimizar o método, é excluir a técnica-3 do CCHPB e assim renomeá-lo para o método chamado de CC-MLD (*Compressed Code using Huffman-Based Multi-Level Dictionary*). O método CC-MLD executa apenas as técnicas de compressão usando *Huffman* em padrões de blocos e em instruções unitárias, nessa ordem.

As simulações para o método CC-MLD são basicamente as mesmas simulações que foram realizadas para os métodos HDPB e CCHPB. Os melhores tamanhos encontrados para o primeiro nível do dicionário foram 256, 512 e 768 e para o segundo nível 512, 1024 e 1280 entradas. A Tabela VI mostra uma sumarização das médias nas taxas de compressão dos programas do *MiBench* com o método CC-MLD.

TABELA VI. MÉDIA GERAL NA TAXA DE COMPRESSÃO DO CC-MLD

| | Nível 1 | | |
|---|---|---|---|
| **Nível 2** | **256** | **512** | **768** |
| **512** | 29,3% | 28,3% | 26,3% |
| **1024** | **32%** | 29,7% | 27,2% |
| **1280** | 30,3% | 28,6% | 25,9% |

Observando os valores das Tabelas III e VI destaca-se que, se fixarmos os tamanhos para o primeiro e segundo nível do dicionário da Tabela VI e compararmos os valores das taxas de compressão com os valores dos mesmos tamanhos de dicionários apresentados na Tabela III, nota-se que na maioria das vezes o método CC-MLD conseguiu obter uma taxa de compressão maior quando comparado com a compressão de *Huffman*, seja comprimindo instruções unitárias ou padrões de blocos. Então, fixando o tamanho dos níveis 1 e 2 do dicionário multi-nível em 256 e 512 (768 entradas) foi alcançado um aumento de 2,5% na compressão do código comparada com a taxa das instruções unitárias; Já para os tamanhos 256 e 1024 esta diferença foi de 4,9%. O mesmo ocorreu para os tamanhos 256 e 1280 atingindo 3% de crescimento na taxa de compressão. No entanto, para o tamanho 768 e 1280 (2048 entradas) o método CC-MLD obteve uma taxa de compressão menor (-0,6%) do que o método tradicional de *Huffman*.

Destaca-se ainda ao observar a Tabela VI, que na medida em que o tamanho do nível 1 do dicionário começa a crescer, a taxa de compressão realiza um efeito inverso, ou seja, começa a diminuir. Então, conclui-se que não é vantajoso no método CC-MLD o uso de um dicionário multi-nível com o tamanho de níveis maiores, conforme provado pelos resultados expressos nas Tabelas III e VI para os tamanhos dos níveis 1 e 2 igual a 768 e 512; 768 e 1024; 768 e 1280 do dicionário multi-nível. Portanto, ressalta-se que para projetos de sistemas embarcados onde o espaço físico é um requisito de alta prioridade o uso de um dicionário multi-nível de tamanho menor provavelmente terá um efeito mais significativo do que um dicionário onde os tamanhos dos níveis são maiores.

Um dos principais focos dos métodos apresentados nesse artigo é mostrar que existem outras formas de aumentar a taxa de compressão utilizando ainda o algoritmo de *Huffman*. Sendo que, para os métodos HDPB, CCHPB e CC-MLD desenvolveu-se uma nova abordagem que é a compressão usando *Huffman* em padrões de blocos formados por duas ou mais instruções adjacentes. No entanto, ao analisar as Tabelas III, IV, V e VI conclui-se que mesmo sendo necessário espaço adicional para um novo dicionário multi-nível os métodos apresentados nesse artigo conseguiram ser tão eficiente na compressão dos códigos quanto os resultados apresentados por outros métodos estudados na Seção II desse artigo e que utilizaram como base o algoritmo de *Huffman* e dicionário, mas com complexidade computacional maior. Também ressalta-se que a melhor configuração do dicionário multi-nível para a compressão dos programas do *MiBench* usando o método CC-MLD é quando os níveis 1 e 2 do dicionário possuem 256 e 1024 entradas, respectivamente (ver Tabela VI). Na Figura 6 mostra-se uma média do tamanho dos programas (em *bytes*, originais e comprimidos).

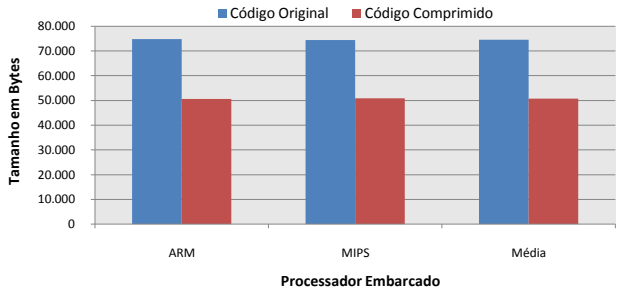


Fig. 6. Média do tamanho dos programas originais e comprimidos.

Observa-se na Figura 6 que em ambas as arquiteturas (ARM e MIPS) houve uma redução (em *bytes*) no tamanho dos programas do *MiBench* quando aplicados ao processo de compressão, em média 32%, ou seja, houve variação do tamanho médio original de 74.619 *bytes* para 50.771 *bytes* comprimido. Comprovando esta afirmação mostra-se detalhadamente na Figura 7 a taxa de compressão obtida por cada uma das técnicas do método CC-MLD.

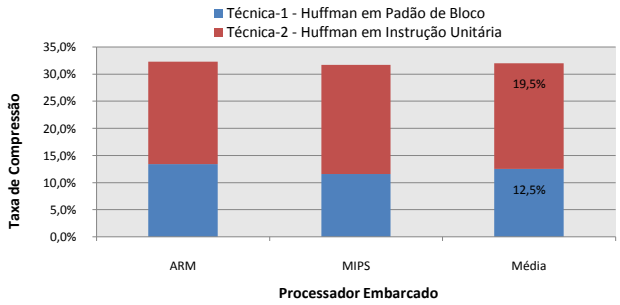


Fig. 7. Taxa de compressão obtida pelo método CCHPB.

E por fim, a Figura 8 apresenta um comparativo (para a média dos dois processadores embarcados) no processo de compressão mostrando o tamanho do código original e também do código comprimido para cada uma das técnicas do método CC-MLD, confirmando assim que a compressão nas duas técnicas obteve resultados satisfatórios.

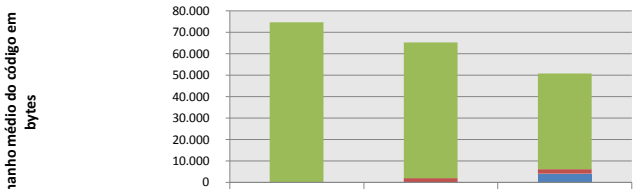


Fig. 8. Comparativo das técnicas de compressão do método CC-MLD.

Analisando a Figura 8 destaca-se que para o melhor tamanho do dicionário multi-nível usado pelo método CC-MLD, o espaço ocupado pelo dicionário não influencia muito no tamanho final do código comprimido, ou seja, apenas 13,7% no total, sendo que o nível 1 do dicionário corresponde por somente 4,6% e os outros 9,1% pelo segundo nível do dicionário. Já quando comparado com o tamanho do código original o dicionário multi-nível influencia em apenas 8,2%. Portanto, conclui-se que o uso do dicionário multi-nível pelo método CC-MLD trouxe ainda mais benefícios na compressão do código dos programas do *MiBench*.

A Tabela VII mostra um resumo dos resultados obtidos pelos autores dos trabalhos correlatos (apresentado na Seção II) através das simulações e análises realizadas. Para uma possível comparação, nesta tabela também aparece à taxa de compressão média obtida com os métodos propostos neste artigo.

TABELA VII. RESUMO DOS TRABALHOS CORRELATOS

| Autor | Plataforma | benchmark | Nº de Programas | Taxa de Compressão |
|---|---|---|---|---|
| Wolfe &Chanin [3] | MIPS | SPEC'95 | 6 | 27% |
| Haider &Nazhandali [12] | ARM | MiBench | 9 | 20% DicSmall |
| | | | | 9% DicLarge |
| Bonny& Henkel [15] | ARM | MiBench | 7 | 47% |
| | MIPS | | | 49% |
| Collin&Brorsson [10] | MIPS IV | Mediabench | 15 | 23% |
| **HDPB** | **ARM** | **MiBench** | **7** | **28,5%** |
| | **MIPS** | | | **27,2%** |
| **CCHPB** | **ARM** | **MiBench** | **7** | **34,8%** |
| | **MIPS** | | | **34,3%** |
| **CC-MLD** | **ARM** | **MiBench** | **7** | **32,3%** |
| | **MIPS** | | | **31.7%** |

## V. HARDWARE DESCOMPRESSOR DO MÉTODO CC-MLD

O tempo gasto na descompressão de uma instrução comprimida é considerado crítico, em virtude deste processo ocorrer em tempo de execução. Então, o hardware descompressor deve ser capaz de fornecer a respectiva descompressão evitando que ocorra alto *overhead* na execução de um código comprimido. Assim, o projeto do hardware descompressor deve ter sua implementação baseada na execução de apenas um único ciclo.

O hardware descompressor do método CC-MLD foi projetado para atuar como um módulo independente das memórias principal e *cache*. Uma vez que a arquitetura escolhida para ser utilizada pelo método foi a CDM. A Figura 9 mostra a arquitetura do hardware descompressor desenvolvida para o método CC-MLD e a Figura 10 apresenta a estrutura básica do componente "Lógica do Descompressor".

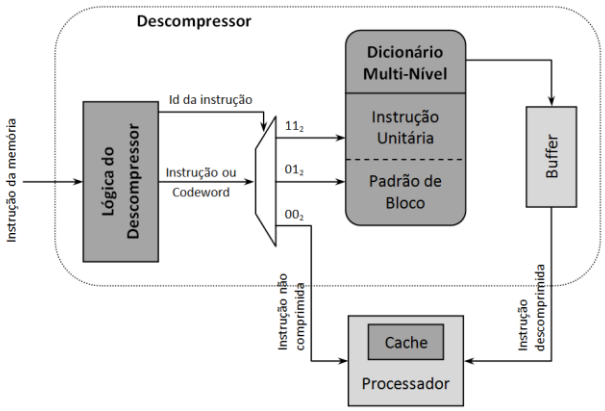


Fig. 9. Arquitetura do hardware descompressor do método CC-MLD.

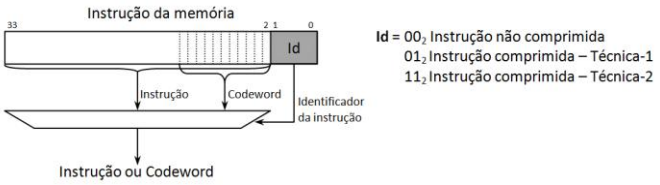


Fig. 10. Estrutura básica do componente Lógica do Descompressor.

O processo da descompressão de uma instrução unitária ou de um padrão de bloco é executado da seguinte forma: supondo que a instrução buscada na memória principal do sistema possua 34 *bits* (32 *bits* do tamanho da instrução do processador tipo RISC e 2 *bits* de controle para a compressão - Id), sendo que os dois primeiros *bits* são o identificador da instrução (Id). Então, primeiramente a instrução passa pelo componente "Lógica do Descompressor", pois este componente é responsável em identificar se a instrução está ou não comprimida. Caso a instrução não esteja comprimida então imediatamente ela é repassada para o processador e a memória *cache*. Agora se a instrução estiver comprimida então o nível correto do "Dicionário Multi-Nível" é acessado diretamente na posição indicada pela *codeword* e assim a instrução unitária ou o padrão de bloco contido nesta posição é repassado para o componente *Buffer* que se encarrega de entregar a instrução para o processador e a memória *cache*. Quando um padrão de bloco é carregado no *Buffer*, somente após a execução de todas as instruções pertencentes ao padrão de bloco é que um novo acesso à memória principal precisa ser feito (exceto no caso das instruções de desvio).

Foi implementada uma versão do hardware descompressor do método CC-MLD em software (usando a linguagem C) e inserida no código fonte do simulador *SimpleScalar*. Assim, foi possível obtermos dados da execução de códigos originais quanto comprimidos, o que permitiu fazer uma análise comparativa e apontar os benefícios alcançados com o uso da compressão de código. A Figura 11 mostra o desempenho final do sistema usando o método CC-MLD, onde pode-se constatar que a execução dos códigos comprimidos dos programas do *MiBench* tiveram um *overhead* médio de 3,7%.

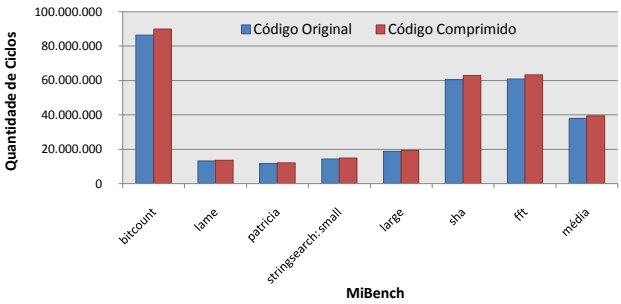


Fig. 11. Desempenho do sistema usando o método CC-MLD.

Uma versão do hardware descompressor também foi implementada em VHDL e prototipada em uma FPGA da família *Cyclone-II*, modelo EP2C35F672C6, usou-se a ferramenta *Quartus-II Web Edition* da *Altera*® [1] para sintetizar o hardware descompressor do método CC-MLD.

Na Figura 12 é possível observar que o processo de descompressão é executado em apenas um *clock* levando em consideração o tempo de *clock* em 10 *ns*. Comparando este resultado com os resultados apresentados em [14], onde o autor desenvolveu uma ferramenta de comparação e avaliação para diferentes métodos de compressão de código, e usou vários parâmetros para analisar a compensação entre espaço-tempo-custo e mostrou resultados obtidos na execução do hardware descompressor para sete métodos baseados em dicionário (dicionário com tamanho fixo, dicionário usando LAT com tamanho fixo, distância de *Hamming*, múltiplos dicionários para os processadores TI, ARM e MIPS e o método de *Huffman*), quando implementados usando HDL e prototipados em uma FPGA os métodos utilizaram entre 10 a 119 *clocks*. Assim, o hardware descompressor do método CC-MLD mostrou-se mais eficiente que aqueles descritos na Seção II.

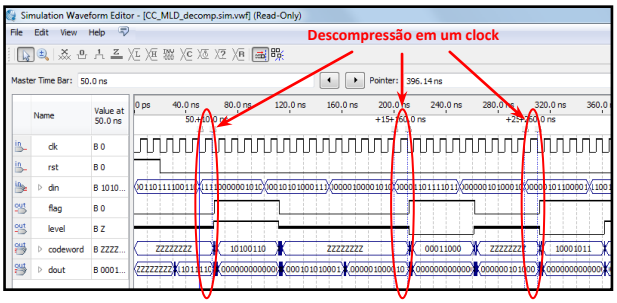


Fig. 12. Processo de execução do hardware descompressor.

Por outro lado, usando a ferramenta *PowerPlay Power Analyzer*, inserida na ferramenta *Quartus-II*, constata-se que o hardware descompressor do método CC-MLD consome 114.04 *mW* (*micro Watts*) de energia da FPGA e o hardware do método de *Huffman* consome 112.02 *mW*, ou seja, uma diferença de 2.02 *mW*, aumento de 1,8%. Assim, verifica-se que em termos de energia o consumo em utilizar um dicionário multi-nível é praticamente igual ao consumo de se usar um dicionário normal.

## VI. Conclusões e Trabalhos Futuros

Neste artigo apresentamos três novos métodos de compressão de código chamados de HDPB, CCHPB e CC-MLD, que usam *Huffman* para instruções unitárias e padrões de blocos baseando-se em dicionário multi-nível. Os métodos são independentes de arquitetura. Realizou-se simulações com o *SimpleScalar* e aplicações do *MiBench*, e para dois processadores embarcados de 32 *bits* (ARM e MIPS) foi possível alcançar uma taxa de compressão média de 34,5%, sendo que o processo de descompressão requer somente um *clock* quando implementado em FPGA simples do tipo *Cyclone-II* de uma placa DE-2. O descompressor com dois níveis no dicionário aumenta somente 2% o consumo de energia, quando comparado a métodos que usam um único dicionário *Huffman*. Como trabalhos futuros sugerem-se: (i) testar a compressão e descompressão de códigos usando outros pacotes de programas de *benchmark*; (ii) analisar o comportamento dos métodos usando a arquitetura PDC (*Processor Decompressor Cache*) e (iii) melhorar a forma de identificação das instruções normais ou comprimidas.

## Referências

[1] Altera® Corporation. Disponível em: http://www.altera.com. Acessado em 01 de março de 2013.

[2] A. S. Oliveira, and F. S. Andrade, "Embedded Systems - Hardware and Firmware in Practice". São Paulo: Publisher Érica, 2006, 316p.

[3] A. Wolfe, and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture". In Proc. of 25th Annual Intl. Symp. on Microarchitecture (MICRO 25), USA, p. 81-91, Dec. 1992.

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

[5] D. Seal, "ARM Architecture Reference Manual". Addison-Wesley Professional, 2nd edition, 2001, 816p.

[6] D. Sweetman, "See MIPS Run". – San Francisco, California, USA: Morgan Kaufmann Publishers, 2nd edition, 2006, 512p.

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

[8] G. de Micheli, "Computer-Aided Hardware-Software Codesign". IEEE Micro, 14(4):10-16, August 1994.

[9] H. Lekatsas, J. Henkel, and W. Wolf, "Code Compression for Low Power Embedded System Design". In Proc. of the 37th Annual Design Automation Conference (DAC'00), USA, pages 294-299, June 2000.

[10] M. Collin, and M. Brorsson, "Two-Level Dictionary Code Compression: a New Scheme to Improve Instruction Code Density of Embedded Applications". In Proc. of 17th Intl. Symposium on Code Generation and Optimization (CGO'09), USA, p. 231-242, March 2009.

[11] P. Marwedel, "Embedded System Design". New York, USA: Springer-Verlag, 2nd edition, 2006, 241p.

[12] S. I. Haider, and L. Nazhandali, "A Hybrid Code Compression Technique using Bitmask and Prefix Encoding with Enhanced Dictionary Selection". In Proc. of the Intl. Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'07), Salzburg, Austria, pages 58-62, September 2007.

[13] S. Klein, "Space and Time-Efficient Decoding with Canonical Huffman Trees". In Proc. of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997), Aarhus, Denmark, pages 65-75, June 1997.

[14] S. K. Menon, "An Efficient Tool-Chain for Analyzing Tradeoffs of Code Compression Schemes in Embedded Processors". In Proc. of 18th IEEE-Intl. Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2012), Korea, p. 192-201, Aug. 2012.

[15] T. Bonny, and J. Henkel, "Instruction Splitting for Efficient Code Compression". In Proc. of the 44th Annual Design Automation Conference (DAC'07), USA, pages 646-651, June 2007.

[16] W. R. A. Dias, and E. D. Moreno, "Code Compression in ARM Embedded Systems using Multiple Dictionaries". In Proc. of 15th IEEE Intl. Conference on Computational Science and Engineering (CSE 2012), Paphos, Cyprus, pages 209-214, December 2012.