Design and Implementation of the PBIW Instruction Decoder in a Softcore Embedded Processor

Renan Marks, Felipe Araújo, Renato Santos, Felipe Yonehara, Ricardo Santos

LSCAD - High Performance Computing Systems Laboratory
School of Computing
Federal University of Mato Grosso do Sul
Campo Grande - MS - Brasil
Email: renanmarks@gmail.com, oli.engenharia@gmail.com, renatocox@gmail.com, felipeyonehara@gmail.com, ricardo@facom.ufms.br

Abstract

This paper presents the PBIW (Pattern Based Instruction Word) instruction encoding technique on the ρ-VEX embedded softcore processor. The PBIW encoding technique maps the assembly generated by a compiler into an encoding scheme of a target processor. The results obtained shows that the PBIW encoding has a compression ratio ranging from 60.97% to 115.91% among the evaluated programs. The impact of PBIW encoding in the memory access shows significant performance gains since there are improvements in hit ratio up to 58.93%. The PBIW decoder experiments show that the adoption of PBIW decoder circuit shrinks the processor total area in 15% (on average) and dynamic power reduction in 40%. In addition, the PBIW decoder reduces 56% and 52% the dynamic power consumption and the amount of data stored (memory bits of M4K memory blocks) in the instruction memory.

***Keywords*-PBIW; hardware decoder; FPGA; ρ-VEX ;**

I. Introdução

A evolução da tecnologia em torno de novos modelos arquiteturais tem chegado até o projeto de processadores embarcados. Atualmente, processadores embarcados utilizam *cores* de processamento e *caches on-chip* para aumentar o desempenho de forma que a redução da área ocupada, seja pelos programas ou mesmo pelo projeto do sistema embarcado, não é o principal motivo para otimizações no projeto.

Do objetivo inicial de reduzir o tamanho do código do programa para questões envolvendo o impacto do descompressor/decodificador no sistema computacional, as técnicas de compressão/codificação de código focaram no compromisso entre área do programa versus desempenho do código, projeto de baixo consumo de potência e aumento do tempo de ciclo do processador. Assim, embora a preocupação inicial tenha sido a eficiência na compressão/codificação do código, os impactos gerados pela inclusão de um descompressor/decodificador no processador não podem ser desconsiderados.

Este artigo apresenta a técnica de codificação PBIW (*Pattern Based Instruction Word*) [1] que objetiva codificar instruções de um processador alvo para instruções codificadas e padrões de instruções. A técnica é baseada na premissa que as instruções de um programa possuem informações redundantes internamente (entre campos da instrução) e externamente (entre instruções). Assim, a técnica extrai as redundâncias e gera instruções codificadas e padrões. Esses elementos complementam-se para recompor a instrução codificada em instrução original antes do estágio de execução. Este trabalho apresenta o projeto e implementação da codificação PBIW sobre o processador embarcado ρ-VEX [2]. Além disso, apresenta também os impactos da decodificação com relação à área, consumo de potência dissipada (estática e dinâmica) e frequência máxima do processador com e sem a codificação PBIW. Os experimentos que investigam os impactos do decodificador consideram a implementação sobre uma plataforma baseada em FPGAs.

O artigo é organizado conforme segue: A Seção II apresenta os trabalhos da literatura da área relacionados com

PBIW; A Seção III descreve as principais características do processador ρ-VEX ; A Seção IV apresenta a técnica PBIW; A Seção V apresenta e discute a implementação da técnica PBIW sobre o processador ρ-VEX ; A Seção VI descreve os experimentos realizados e discute os resultados obtidos com a codificação e decodificação PBIW; A Seção VII apresenta as principais conclusões e os resultados do trabalho.

II. Trabalhos Relacionados

Apresenta-se a seguir algumas propostas de codificação de instruções dependentes de conjuntos de instruções. Nos próximos parágrafos, considera-se que taxa de compressão/codificação (TC) é:

$$TC = \frac{\text{Tamanho Programa Codificado+Overhead}}{\text{Tamanho Programa Não-Codificado}} \quad (1)$$

O Overhead é devido à criação de palavras de dicionário ou padrões de código.

A técnica de codificação de instruções Thumb [3], [4], aumenta a densidade de código ARM com impacto mínimo no desempenho do programa. O conjunto de instruções Thumb é um subconjunto do conjunto de instruções ARM, codificado em 16 bits. O resultado da codificação de código é um aumento em 30% na densidade de código ARM nativo [3]. No entanto, os autores informam uma redução de desempenho de 15%-20% [5].

A técnica MIPS16 [6] é um mecanismo de codificação para instruções MIPS. Cada instrução MIPS16 corresponde a exatamente uma instrução MIPS. Da mesma forma que Thumb, instruções MIPS16 necessitam de um decodificador junto à via de dados do processador. Para reduzir o número de bits da instrução pela metade são utilizados os três componentes da instrução: opcodes, número de registradores e valores imediatos. Segundo os autores, a taxa de compressão alcançada foi de 60%.

A técnica SPARC16 [7] representa uma escolha entre oportunidade de codificação e impacto do resultado. Os campos mais importantes, da instrução SPARC original, para redução são os de valores imediatos que podem ocupar mais da metade da instrução. A tradução é realizada por meio de um decodificador localizado entre a *cache* de instruções e o pipeline do SPARCv8. Conforme experimentos realizados pelos autores, SPARC16 obteve uma taxa de codificação média de 60% em relação ao SPARCv8 nativo. Também houve redução significativa no tamanho do código e no número de *cache misses* para *caches* de vários tamanhos.

A técnica Codepack^{TM} [8] foi desenvolvida com objetivo de codificar instruções para processadores IBM PowerPC. Os autores informam um ganho de desempenho da ordem de $\pm 10\%$ utilizando Codepack. O decodificador Codepack foi projetado e implementado entre o processador PowerPC e o controlador de memória.

As diferenças principais entre as técnicas de codificação apresentadas e PBIW reside na dependência quanto a um conjunto de instruções específicos para utilização. PBIW propõe que, independente do conjunto de instruções/processador alvo sob estudo, a codificação seja baseada na extração da redundância do código por meio da construção de padrões. Esses padrões, de forma parecida ao que acontece com Codepack, são armazenados em uma memória ou tabela de padrões e são acessados durante a fase de decodificação de instruções. Os padrões funcionam como ponteiros que indicam para o decodificador em *hardware* como a instrução original deve ser obtida. Outra característica que diferencia PBIW das demais técnicas é a existência de uma função sobrejetora entre o conjunto de instruções codificadas e o conjunto de padrões. O mapeamento de diferentes instruções para um mesmo padrão possibilita projetar tabelas específicas de padrões junto à via de dados sem gerar impactos significativos na área do processador.

III. O Processador ρ-VEX

O ρ-VEX [2] é um processador VLIW *soft-core* configurável descrito em VHDL baseado no ISA VEX [9]. O ISA VEX é inspirado no ISA da família de processadores HP/ST Lx [9] (o mesmo usado para os processadores ST200). O ρ-VEX possui uma micro-arquitetura VLIW minimalista: não possui unidade de gerenciamento de memória nem unidade para cálculo de números de ponto-flutuante. Apesar dessas limitações, o processador ρ-VEX possui diversas características, como instruções longas em memória e projeto *soft-core* embarcado, que o torna adequado a adotar uma técnica de codificação de instruções.

A Figura 1 apresenta a via de dados do processador ρ-VEX . Depois dos estágios de busca e decodificação, as sílabas de uma instrução são despachadas para unidades de execução individuais. Cada instrução contém 4 sílabas. Todas as sílabas podem ser operações ALU, porém operações de controle só estão presentes na primeira sílaba. Operações de multiplicação e divisão podem estar presentes somente nas segunda e terceiras sílabas e operações de acesso à memória só estão presentes na quarta sílaba.

O conjunto de ferramentas de geração de código e simulação para ρ-VEX contém um compilador C, bibliotecas ANSI C, um simulador em *software* (disponibilizado pela HP [10]) e o montador ρ-ASM. Apesar da existência de ferramentas para geração e simulação de código, tais ferramentas não suportam a compilação e simulação de qualquer tipo de programa. Especificamente, não há ferramentas de link-edição, forçando, assim, a remoção de

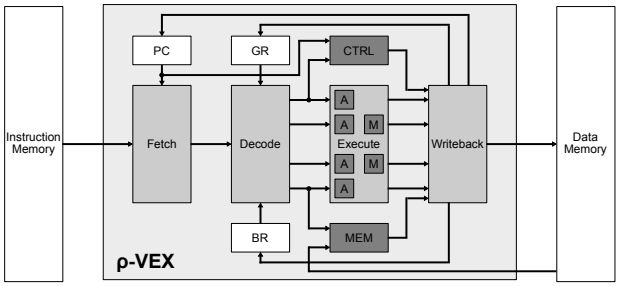

Figura 1. Via de dados arquitetural do processador ρ-VEX .

todas as chamadas a funções presentes em bibliotecas externas. Todas as funções em um programa devem ser expandidas inline na função *main* de forma a preservar o endereçamento global de variáveis.

IV. A Técnica de Codificação PBIW

A técnica de codificação *Pattern Based Instruction Word* (PBIW) [1] percorre o programa realizando a fatoração de operandos redundantes e opcodes em dois elementos: instruções codificadas e padrões. PBIW explora a sobrejeção entre o conjunto de instruções e o conjunto de padrões. A técnica é embasada na técnica de fatoração de operandos [11], [12].

Uma instrução codificada PBIW é uma estrutura de dados que não contém dados redundantes (registradores ou imediatos) e é armazenada em uma *Cache* de Instruções. Um padrão PBIW é uma estrutura de dados que contém as informações necessárias para reconstituir, junto a um decodificador de instruções, a instrução original para o estágio de execução. O padrão contém ponteiros para as posições da instrução codificada e é armazenado em uma *cache* ou tabela chamada de Tabela de Padrões (P-Tabela ou P-*Cache*). O custo de um novo decodificador na via de dados do processador é reduzido, tendo em vista que o projeto do decodificador PBIW explora o paralelismo entre as atividades de *hardware*, como decodificação de instruções na unidade de controle e leitura de operandos no banco de registradores.

Considerando que PBIW não é uma técnica de codificação com enfoque em uma ISA ou processador específico, o projeto do padrão e instrução codificada deve levar em consideração algumas informações: número de registradores de leitura e escrita, número e tamanho de imediatos na instrução codificada e tamanaho da tabela de padrões.

Cada instrução codificada PBIW tem $\mathcal{R} > 0$ campos para representar registradores de leitura, onde $\mathcal{R}$ é o número máximo de portas de leitura do banco de registradores. A instrução precisa ter $\lceil \log_2 Regs \rceil$ bits para endereçar cada registrador, onde *Regs* é o número total de registradores da arquitetura. A instrução codificada aponta para seu respectivo padrão através do campo ponteiro do padrão, cujo tamanho é $\lceil \log_2 |\text{P-Tabela}| \rceil$, ou seja, a quantidade de bits para endereçar as entradas da tabela de padrões. A instrução codificada PBIW pode oferecer campos de cancelamento que são utilizados para anular operações/sílabas específicas da instrução. A quantidade de bits de cancelamento é N bits, onde N é o número de operações que constituem o padrão PBIW. Cada operação contém um opcode e um número fixo de operandos. Cada operando tem $\lceil \log_2 X \rceil$ bits, onde X é o número de campos dedicados para operandos (escrita e da instrução PBIW).

Os campos de cancelamento na instrução codificada PBIW, indicam que operações do padrão não serão executadas em tempo de execução. A utilização de campos de cancelamento permite que o padrão seja utilizado por mais que uma instrução codificada (reúso de padrões). Ao fazer isso, uma etapa especial no codificador PBIW, otimiza os padrões juntado-os.

O algoritmo de codificação PBIW acrescenta um novo passo na infraestrutura de complicação ρ-VEX . O algoritmo de codificação e a otimização de junção de padrões têm complexidade de tempo $(O(N^2)$, onde $N =$ número de instruções não codificadas$)$.

A. Exemplo de Codificação PBIW

Um exemplo da técnica de codificação de instruções PBIW sobre o processador ρ-VEX é apresentada na Figura 2. Nessa figura são mostradas duas instruções originais (128 bits), duas instruções codificadas e um padrão após a codificação PBIW. O padrão é compartilhado entre ambas instruções codificadas, demonstrando a possibilidade de reúso.

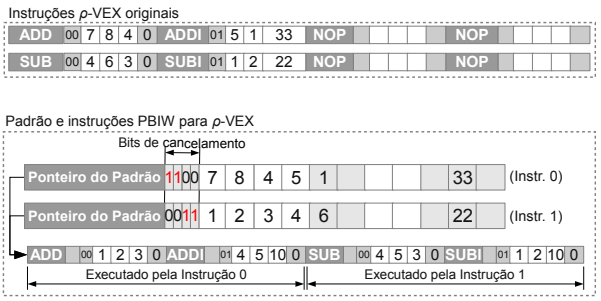

Figura 2. Instruções originais, codificadas e padrão PBIW ρ-VEX .

Pode-se observar que as operações ADD e ADDI são executadas apenas pela instrução 0 e as operações SUB e

SUBI são executadas pela instrução 1. Os bits de cancelamento para ambas instruções são 1100 e 0011, respectivamente. Nos campos de cancelamento da instrução 0, a sequência 1100 indica que a primeira e segunda operações (ADD e ADDI) do padrão, devem ser escritas em um registrador ou memória e, a terceira e quarta operações (SUB e SUBI) não devem tornar o efeito de seus resultados em registradores ou memória (efeito NOP) .

B. Otimização de Junção de Padrões

A junção de padrões é uma otimização com objetivo de reduzir a quantidade total de padrões na tabela de padrões. Essa otimização é aplicada sobre padrões cuja soma da quantidade de operações é menor ou igual à capacidade de um padrão. Se a junção de padrões gera um padrão C de dois padrões A e B:

1) Todas as instruções que apontam para o padrão A e B têm o campo de ponteiro para o padrão PBIW atualizado com o endereço do novo padrão (C);
2) Todas as instruções que apontam para o padrão A devem atualizar os bits do campo de anulação, para anular operações do padrão B e vice-versa;
3) Os padrões A e B são apagados do conjunto de padrões do programa.

O algoritmo de junção de padrões é constituído de três fases:

1) **Pré-processamento**: organiza os padrões em categorias e subcategorias, de acordo com a quantidade de operações nos padrões e o tipo de cada operação. Cada subcategoria é a combinação simples dos tipos de operações que compõe o padrão, considerando as restrições aquiteturais.
   Por exemplo, para o processador ρ-VEX (Figura 3), as categorias P1, P2, e P3 correspondem a padrões com uma operação, duas operações e três operações, respectivamente. Padrões completos (com o total de operações suportadas) não são considerados. Os padrões P1 e P3 são organizados em categorias de acordo com o tipo da operação (por exemplo: operações de *branch*, operações de memória, operações lógicas e aritméticas, etc.). Padrões P2 seguem uma organização diferente de P1 e P3, de acordo com a combinação das operações suportadas pela arquitetura.
2) **Processamento**: executa o algoritmo de junção de padrões tendo como entrada os padrões organizados. A junção é realizada em pares de padrões. O algoritmo cria os pares com base nas categorias, criadas na fase anterior. Se houver junção entre um padrão com uma operação e um padrão com duas operações, resultará em um padrão com três operações, que poderá ser unido a outro padrão com uma operação. O mesmo acontece na junção entre padrões com uma operação.
3) **Pós-processamento**: Remove padrões redundantes do conjunto de padrões e atribui endereços contíguos aos padrões do conjunto final.

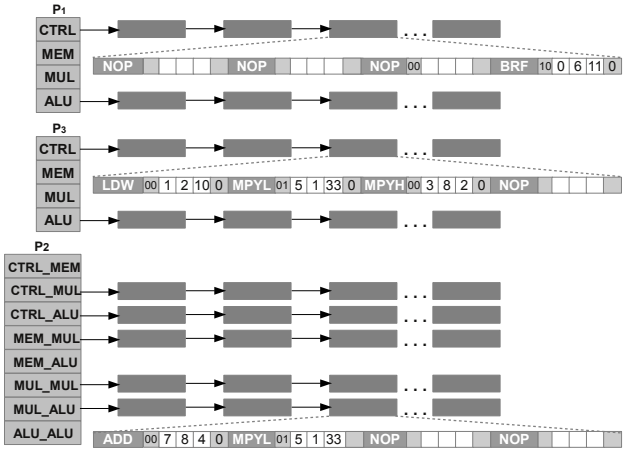

Figura 3. Categorias de padrões para o conjunto de instruções ρ-VEX .

V. Projeto da Codificação e Decodificação PBIW sobre o Processador ρ-VEX

A instrução PBIW codificada pode ser vista na Figura 4. A instrução ρ-VEX original possui 128 bits de comprimento enquanto que a nova instrução codificada 64 bits. Cada instrução codificada possui um campo de ponteiro para padrão PBIW (7 bits), o campo de anulação (4 bits), 7 campos indexados de 1 a 7 para os operandos de leitura, 4 campos para operandos de escrita. Os campos de operandos possuem 5 bits cada, exceto o último campo, que possui 3 bits. O operando no campo zero é aterrado e não existe fisicamente na instrução, mas é útil e usado na codificação para prover o valor zero. Os últimos 3 campos são usados para armazenar valores imediatos.

O padrão PBIW (Figura 5) possui 96 bits divididos em 4 sílabas de 24 bits cada. Uma sílaba contém um campo de opcode de 7 bits, um campo de 2 bits para seletor de imediato, três campos de 4 bits e um campo de 3 bits usados como índices para os operandos na instrução PBIW.

Durante a decodificação de uma instrução codificada, os endereços dos registradores que devem ser lidos são enviados para as portas de leitura do banco de registradores BR e GR. Concomitantemente com a leitura dos registradores, o padrão é recuperado da cache de padrões utilizando o índice armazenado na instrução PBIW. Sendo o tamanho do campo do Índice da Tabela de Padrões de 7 bits, a implementação da cache de padrões (P-*cache*) tem

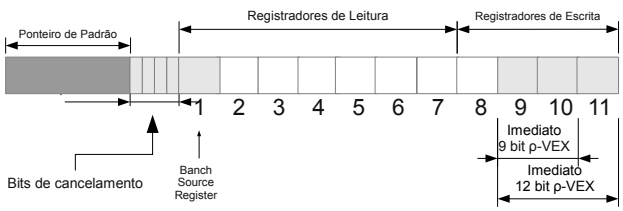

Figura 4. Instrução codificada PBIW ρ-VEX .

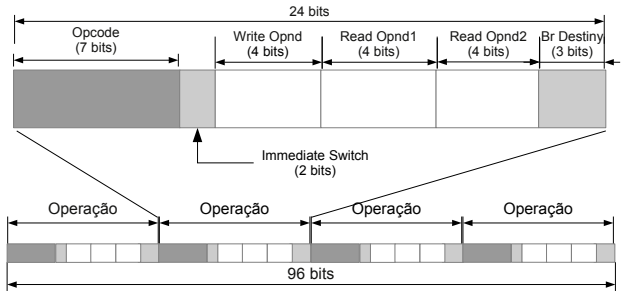

Figura 5. Padrão PBIW ρ-VEX .

um tamanho de 1.5 Kbytes (128 linhas x 96 bits = 12.288 kbits= 1.5 Kbytes) e está implementada como uma tabela de acesso direto junto a unidade *Decode*.

Após a leitura dos registradores, os valores são disponibilizados na entrada dos multiplexadores referentes aos operandos de origem (leitura), e os números dos registradores/imediatos que não são lidos do banco de registradores são disponibilizados na entrada dos multiplexadores referentes aos operandos de destino (escrita). Os sinais que selecionam os valores disponibilizados nas entradas dos multiplexadores são os ponteiros existentes no padrão.

A visão geral do circuito decodificador PBIW é apresentada na Figura 6. Em virtude da complexidade e da limitação de espaço, apresenta-se aqui o resultado da decodificação de uma operação apenas.

VI. Experimentos e Resultados

Esta seção apresenta os resultados obtidos com a utilização da técnica PBIW sobre o processador ρ-VEX . Inicialmente, apresentam-se os resultados da codificação PBIW em um conjunto de 13 programas de propósito geral. Em seguida discute-se o efeito do circuito decodificador PBIW junto à via de dados do processador com relação à área, consumo de potência dinâmica e frequência máxima de operação. A Tabela I enumera os programas e as instâncias de entrada adotadas nos experimentos. Todos os 13 programas foram escritos em linguagem C. Alguns (5 deles) fazem parte do Trimaran Compiler Suite (*Simple*

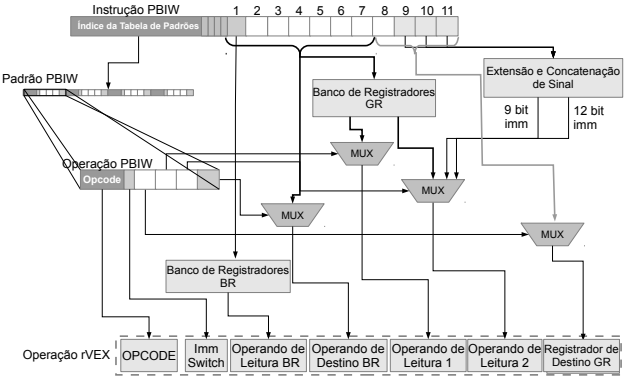

Figura 6. Visão geral do decodificador PBIW ρ-VEX .

benchmarks) [13]. Os programas *mergesort*, *counting_sort*, *linked_list*, *doubled_linked_list*, *avl_tree*, *kruskal*, *prim* e *floyd_warshal* são implementações de algoritmos clássicos de ordenação e manipulação de estruturas de dados. Todos os programas foram compilados e simulados usando o toolchain VEX [9].

Nos experimentos com o decodificador PBIW, a tecnologia alvo utilizada foi a FPGA da família Cyclone II (EP2C35F672C6). Como ρ-VEX não possui hierarquia de memória, todos os programas foram inseridos na memória de instruções do *soft-core* do ρ-VEX antes da síntese do projeto. As memórias de instrução e dados foram prototipadas utilizando os blocos de memória embarcadas denominados M4K [14].

Programas	Descrição	Entrada
avl_tree	Balan. de Árvores	Inserção: 1 a 12 inteiros
nested_complex	Encadeamento de loops	Default
mergesort	Ordenação	Vetor: 15 inteiros
counting_sort	Ordenação	Vetor: 20 inteiros
linked_list	Estrutura de dados	Inserção: 28 inteiros
double_linked_list	Estrutura de dados	Inserção: 17 inteiros
kruskal	Árvore geradora mín.	5x5 matriz de adj.
strcpy	Cópia de strings	strings de 55 bytes
prim	Árvore geradora mín.	3x3 matriz de adj.
bmm	Mult. de Matrizes	vars: NUM=5, BLOCK=1
floyd_warshall	Algo. Caminho mínimo	5x5 matriz de adj.
sqrt	Método Newton Raphson	var: NUM=40
hyper	Desvios internos à loop	200 iterações

Tabela I. Programas usados nos experimentos.

A. Experimentos com a Codificação PBIW

A Figura 7 mostra o tamanho do código estático para programas não-codificados e codificados com PBIW (eixo Y). As barras dos programas codificados (segunda barra de cada programa) também mostra a porcentagem do tamanho das instruções e padrões em relação ao tamanho total do

programa.

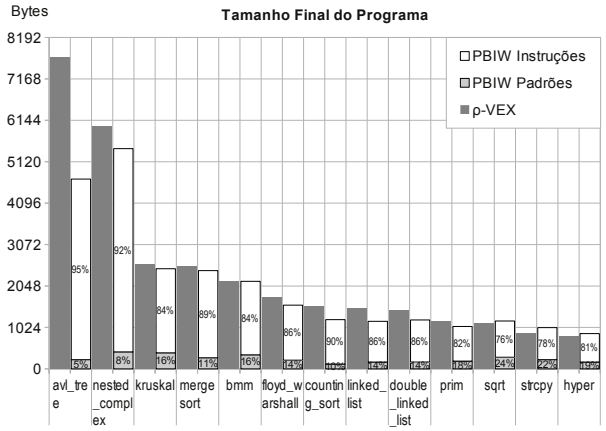

Figura 7. Resultados estáticos dos programas ρ-VEX PBIW.

Os resultados mostram que a adoção do PBIW provê a redução do tamanho de código para 9 dos 13 programas analisados. A taxa de compressão (baseado na Seção II) variou entre 60,97% e 115,91%. A taxa de compressão média foi de 91,78%. Como a técnica PBIW explora o reúso (número de instruções que acessam o mesmo padrão) entre instruções e padrões, programas maiores criam melhores oportunidades de codificação. Essa premissa também é confirmada na Figura 7 pois a diferença entre os tamanhos não-codificado e codificado decresce da esquerda para a direita. Alguns programas codificados obtiveram um tamanho maior devido a geração desproporcional de instruções PBIW. Este comportamento é devido à limitação de espaço na instrução PBIW que força a separação das sílabas presentes na instrução ρ-VEX original em até outras 4 instruções.

Os experimentos dinâmicos mostram os efeitos dos *misses* em *cache* devido à redução do tamanho da instrução em troca da adoção de uma *cache* de padrões. O objetivo é mostrar que a adoção de uma nova *cache* (padrões) não implica em perda de desempenho pelos programas codificados. Os resultados apresentados na Figura 8 foram obtidos através do simulador de caches Dinero [15].

As barras na Figura 8 mostram o melhor (menor) número de bytes buscados da memória (*cache miss*) para cada programa codificado e não-codificado. As c*caches* possuem tamanhos diferentes entre os programas, mas o mesmo tamanho para as versões codificada e não-codificada do mesmo programa. Os tamanhos de cada *cache* para cada programa são exibidas acima dos nomes dos programas. O programa que mais se beneficiou da adoção de PBIW para minimizar a taxa de *miss* na *cache* foi double_linked_list que buscou 58,93% menos bytes

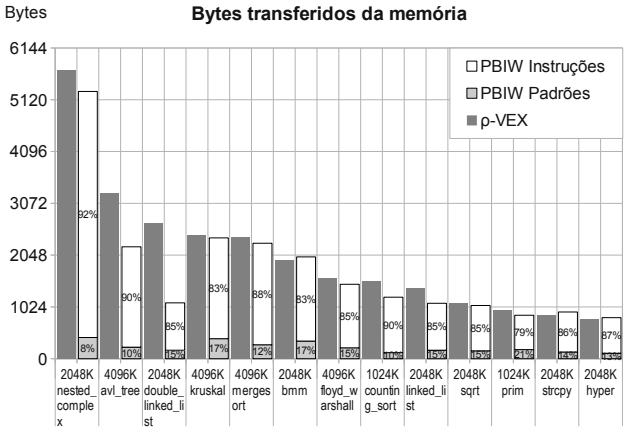

Figura 8. Bytes totais buscados na memória para cada programa.

na memória, enquanto que o programa que buscou 6,94% bytes a mais na memória foi o programa strcpy. A média de melhoria ficou em 12,10%.

B. Experimentos com o Decodificador PBIW

Nesta seção discute-se o efeito do circuito decodificador PBIW junto à via de dados do processador ρ-VEX sobre uma plataforma FPGA. Os experimentos de caracterização de dissipação de potência, e frequência máxima foram feitos utilizando as ferramentas PowerPlay Power Analyzer e o TimeQuest Timing Analyzer integradas ao ambiente do Altera Quartus® [16], [17]. A taxa de comutação, ou taxa de atividade, para o experimento de dissipação de potência foi fornecida para a ferramenta através de um ficheiro de simulação (*.vcd) [16] construído a partir do *testbench* da execução de cada um dos 13 programas.

1) Área e Dissipação de Potência: A potência total dissipada por um dispositivo FPGA é constituída por três componentes: Potência Dinâmica, Potência Estática e Potência de I/O (Entrada e Saída). A potência de I/O (Entrada e Saída) é a energia consumida devida à carga e descarga dos condensadores externos ligados aos pinos dos dispositivos externos. A potência dinâmica é a energia consumida para a operação do dispositivo. A potência dinâmica é proporcional ao número de transições de sinal e as cargas capacitivas em um circuito. Já a potência estática é procional ao número de transistores em *off-stage*.

A técnica de codificação PBIW, apesar de aumentar significativamente o número de elementos lógicos utilizados na unidade *Decode* (Figura 10), trouxe uma redução média de área de 15% no sistema como um todo. Interessante, no entanto, é observar que o decodificador PBIW não alterou o consumo de potência estática do projeto (considerando

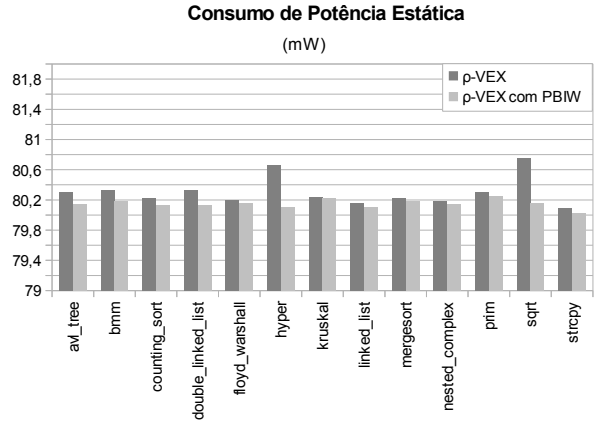

Figura 9. Potência estática consumida.

os 13 programas) como mostra a Figura 9. Dessa forma, nesse artigo investiga-se apenas os impactos que a técnica de codificação PBIW traz sobre a potência dinâmica.

A Figura 10 mostra o número médio de elementos lógicos das unidades existentes na via de dados do processador. A inserção do circuito decodificador trouxe um aumento na unidade de decodificação de 87% (média entre os 13 programas utilizados) quando comparado com a versão original do ρ-VEX . Esse aumento pode ser explicado pelo fato da tabela de padrões e a lógica do circuito decodificador (multiplexadores) estarem instanciadas na unidade *Decode*. Apesar do número de operações que lêem dados do Banco de Registradores GR terem aumentado, não houve um aumento significativo no número de elementos lógicos nesta unidade.

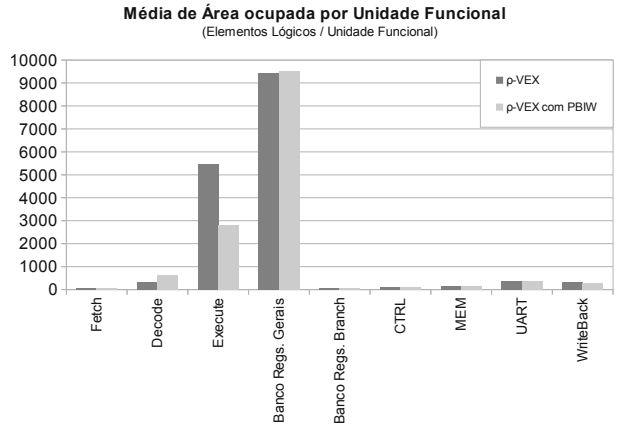

Figura 10. Média de área ocupada por unidade funcional do processador ρ-VEX .

Embora a unidade *Decode* tenha tido um aumento em sua área, o consumo da potência dinâmica nesta unidade foi reduzida cerca de 10% (em média). Pode-se observar que a dissipação de potência na unidade *Decode* aumenta à medida que há um número maior de campos da sílaba com a mesma funcionalidade. A sobreposição de funcionalidade entre os campos de uma sílaba aumenta a quantidade de blocos lógicos de diferentes topologias (comparadores, multiplexadores e portas lógicas) que são necessários para identificar o tipo de layout da sílaba a ser executada. A adoção da codificação PBIW reduz a sobreposição pois a instrução codificada PBIW possui campos com finalidades fixas.

O reúso de padrões reduziu a taxa de comutação (*toggle*) de sinal na memória de instrução em 27% em média. Outro efeito que a codificação PBIW trouxe sobre a memória de instrução foi a redução no número de bits de memória em 52% em média. A consequência dessas reduções tanto na taxa de comutação como no total de número de bits na memória de instrução pode ser vista na Figura 11 que mostra uma redução de cerca 56% no consumo de potência dinâmica na memória de instrução (*imen*). Em média, o circuito decodificador trouxe um redução de 40% no consumo de potência dinâmica.

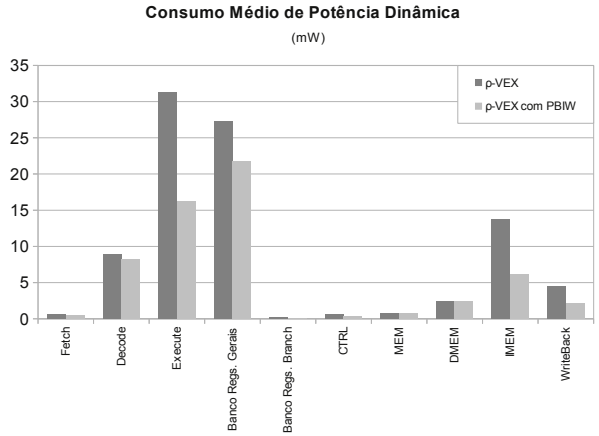

Figura 11. Média de potência dinâmica consumida por unidade funcional.

2) Frequência Máxima: A frequência máxima obtida para cada programa para o processador ρ-VEX com e sem circuito decodificador PBIW é apresentada na Figura 12.

A Figura 12 mostra equilíbrio no impacto sobre a freqüência de operação máxima do *clock* com a inserção do PBIW circuito decodificador. O circuito decodificador PBIW influencia o caminho crítico da via de dados do processador. A origem na redução da frequência máxima com a inserção do circuito decodificador PBIW está no aumento do tempo de chegada de dados na unidade *Execute*. Este atraso acontece porque a codificação PBIW possibilita que um mesmo imediato seja usado por até quatro diferentes

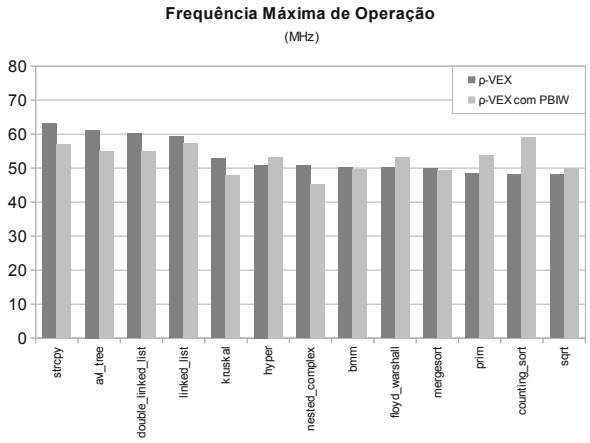

Figura 12. Frequência máxima de operação.

operações na instrução, ao contrário da instrução VEX que possui campos separados para armazenar imediato de uma determinada operação quando necessário.

VII. Considerações Finais

Este artigo apresentou a técnica de codificação de instruções PBIW aplicada sobre um processador soft-core embarcado denominado ρ-VEX . Embora não seja limitada a apenas um tipo de conjunto de instruções, a técnica PBIW mostra-se uma alternativa viável para reduzir a utilização de memória, a redução do tempo de busca e a dissipação de potência dinâmica em processadores como ρ-VEX .

Os resultados apresentados ratificam a possibilidade de adoção de PBIW visando atender os objetivos de projeto assim como alinhar com outras restrições. A taxa de compressão de código estático foi de 60,97%. Considerando o impacto da codificação na cache de instruções e padrões, houve aumento de taxa de acertos de até 58,93%. Os experimentos procuraram observar o comportamento do decodificador de instruções, junto à via de dados do processador, analisando o impacto na área total do projeto do processador, com redução média de 15% em elementos lógicos utilizados, e na dissipação de potência dinâmica com redução média de 40% na potência dinâmica dissipada pelo circuito.

VIII. Acknowledgments

The authors would like to thank Brazilian research funding agencies CNPq, CAPES, and FUNDECT-MS for their financial support to this work and research projects of LSCAD/FACOM/UFMS.

Referências

[1] R. Santos, R. Batistella, and R. Azevedo, "A pattern based instruction encoding technique for high performance architectures," *IJHPSA*, vol. 2, pp. 71–80, 2009.

[2] T. van As, S. Wong, and G. Brown, "ρ-VEX: A Reconfigurable and Extensible VLIW Processor," in *Proceedings of the FPT*. IEEE, 2008.

[3] L. Goundge and S. Segars, "Thumb: Reducing the cost of 32-bit risc performance in portable and consumer applications," *Proceedings of Computer Society Conference*, 1996.

[4] A. Holdings, "ARM1156T2-S Technical Reference Manual," Tech. Rep., 2005.

[5] A. R. M. Ltda, "An introduction to thumb," Tech. Rep., 1995.

[6] K. D. Kissell, "Mips16: High-density MIPS for the embedded market." Real Time Systems, 1997.

[7] L. L. Ecco, B. C. Lopes, E. C. Xavier, R. Pannain, P. Centoducatte, and R. J. de Azevedo, "Sparc16: A new compression approach for the sparc architecture," in *Proceedings of the SBAC-PAD*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 169–176.

[8] M. Game and A. Booker, "Codepack: Code compression for powerpc processors," *Micronews*, vol. 5, no. 1, 1999.

[9] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Elsevier, 2005.

[10] Hewlett-Packard Laboratories, "VEX Toolchain," http://www.hpl.hp.com/downloads/vex/, May 2012.

[11] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code Compression Based on Operand Factorization," in *Proceedings of the MICRO*. IEEE Computer Society, 1998, pp. 194–201.

[12] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code compression," in *Proceedings of the PLDI*. ACM, 1997, pp. 358–365.

[13] L. N. Chakrapani, J. Gyllenhaal, W. Mei, W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah, "Trimaran - An Infrastructure for Research in Instruction-Level Parallelism," *Lecture Notes in Computer Science*, vol. 3602, pp. 32–41, 2004.

[14] ALTERA, *Cyclone II Memory Blocks*, Altera Corporation, 2008.

[15] J. Edler and M. D. Hill, "Dinero iv cache simulator," [online] http://pages.cs.wisc.edu/ markhill/DineroIV/, 2003.

[16] ALTERA, *PowerPlay Power Analysis*, Altera Corporation, 2012.

[17] ——, *The Quartus II TimeQuest Timing Analyzer*, Altera Corporation, 2012.