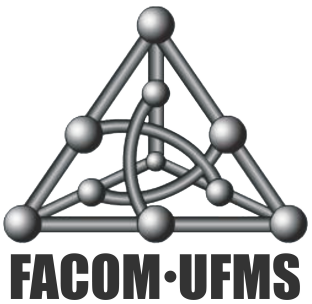
Dissertação de Mestrado

Infraestrutura para Codificação de Instruções Baseada em Fatoração de Padrões

Renan Albuquerque Marks

Orientação:Prof. Dr. Ricardo Ribeiro dos Santos

Área de Concentração: Sistemas de Computação



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
20 de novembro de 2012.

Pesquisa desenvolvida com auxílio financeiro da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) através de bolsa de mestrado.

Infraestrutura para Codificação de Instruções Baseada em Fatoração de Padrões

Campo Grande, 20 de novembro de 2012.

Banca Examinadora:

- Prof. Dr. Ricardo Ribeiro dos Santos (FACOM/UFMS) - orientador
- Prof. Dr. Rodolfo Jardim de Azevedo (IC/UNICAMP)
- Prof. Dr. Fábio Iaione (FACOM/UFMS)

Agradecimentos

À minha família, pelo apoio e participação em toda a minha formação. Agradeço por todas as condições oferecidas e por me mostrar desde cedo a importância, valor e gosto pelos estudos.

Ao professor Ricardo Ribeiro do Santos pelo profissionalismo e dedicação, com sábios e pertinentes conselhos que me guiaram durante todo este trabalho de mestrado. Agradeço por estar sempre disposto à ajudar quando requisitado e por ter ensinado a importância de se realizar um trabalho com qualidade.

Aos meus colegas e amigos de mestrado e pesquisa, pelo companheirismo, das horas de trabalho e estudo, dos desafios e conquistas alcançadas ao longo deste período.

Aos professores da Faculdade de Computação que colaboraram direta e indiretamente com a minha formação ao longo destes anos e que me transmitiram conhecimento suficiente para que eu tivesse a capacidade em realizar este trabalho.

Aos meus chefes superiores, amigos e colegas da Sanesul, que me apoiaram e incentivaram durante a conclusão deste trabalho de mestrado.

À CAPES pelo apoio financeiro.

A todos os outros amigos e pessoas que, por ser impossível enumerar e agradecer diretamente aqui, de forma direta ou indireta, colaboraram para a construção desse trabalho.

Resumo

Este trabalho apresenta o projeto e implementação da técnica de codificação de instruções PBIW (*Pattern Based Instruction Word*), baseada em fatoração de padrões, sobre o processador *softcore* embarcado denominado ρ-VEX. A técnica PBIW é implementada sobre uma infraestrutura de codificação de instruções que mapeia o código de saída de um compilador para o esquema de codificação PBIW em um processador alvo. A infraestrutura é construída com uma arquitetura modular e extensível que possibilita acoplar novos algoritmos codificadores e otimizadores, além de suporte a novos conjuntos de instruções e processadores. Neste trabalho foram projetadas e implementadas duas versões da técnica PBIW para o processador ρ-VEX e foram realizados experimentos estáticos e dinâmicos para validação e caracterização. Esses experimentos visam caracterizar detalhadamente a técnica PBIW sobre os efeitos gerados sobre o código do programa e sobre o processador alvo. Os resultados demonstram que a técnica PBIW oferece ganhos tanto na compressão do tamanho programa (resultados estáticos), quanto em área ocupada e dissipação de potência dinâmica. Os resultados também permitem notar que a utilização da técnica PBIW oferece oportunidades interessantes para exploração do espaço de projeto de decodificadores de código junto à via de dados e controle do processador alvo. A taxa de compressão média para as versões 1.0 e 2.0 foram de 85,82% e 76,87% respectivamente. As taxas de compressões mínimas variaram de 94,35% e 105,00% e as taxas de compressão máximas variaram de 56,51% e 59,70%, respectivamente. **Palavras-chave:** PBIW, ρ-VEX e codificador.

Abstract

This work presents the design and implementation of the PBIW (Pattern Based Instruction Word) instruction encoding technique, based on pattern factorization, on the ρ-VEX embedded softcore processor. The PBIW encoding technique is implemented on the top of an instruction encoding software infrastructure that maps the output code generated by a compiler into the PBIW encoding scheme designed for a target processor. The infrastructure is built based on an extensible and modular design that make it possible to couple to new encoding and optimization algorithms, new ISAs, and target processors. In this work we have designed and implemented two PBIW encoding schemes on the ρ-VEX processor. Static and dynamic experiments have been performed in order to characterize all effects of the PBIW technique on the generated code and on the target processor. The results show that PBIW provides gains of compression ratio, area, and dynamic power. The results also show that PBIW offers opportunities to explore the design space for instruction decoders on a target processor. The average compression ratio obtained for both the versions 1.0 and 2.0 developed for the ρ-VEX processor is 85.82% and 76.87%, respectively. The minimum and maximum compression ratios ranged from 94.35% and 105.00% to 56.51% e 59.70%.
Key-words: PBIW, ρ-VEX e codificador.

Sumário

Glossário

bss Segmento/Seção de dados contendo variáveis estaticamente alocadas que possuem valores inicialmente definidos como zero.

parsing Ato de processar um arquivo de entrada recolhendo informações para uso interno da aplicação.

profiling Análise dinâmica de um software medindo informações existentes somente em tempo de execução como, por exemplo, memória usada, tempo de execução, estatísticas de instruções e funções específicas ou a frequência e duração de chamadas de funções; o uso mais comum é para auxiliar na otimização de programas.

Siglas

ARM Advanced RISC Machine.

ASIC Application-Specific Integrated Circuit.

CPU Central Processing Unit.

ELF Executable and Linkable Format.

EPIC Explicitly Parallel Instruction Computing.

FPGA Field Gate Programable Array.

I-Cache Instruction Cache.

ISA Instruction Set Architecture.

LAT Local Address Table.

MIF Memory Initialization File.

MIPS Microprocessor without Interlocked Pipeline Stages.

PBIW Pattern Based Instruction Word.

P-Cache Pattern Cache.

RISC Reduced Instruction Set Computer.

SPARC Scalable Processor Architecture.

UF Unidade Funcional.

ULA Unidade de Lógica e Aritmética.

VEX VLIW Example.

VHDL VHSIC hardware description language.

VLIW Very Long Instruction Word.

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos últimos anos um grande número de técnicas têm sido desenvolvidas na tentativa de minimizar o problema de *Memory Wall* [5–7] que, de forma geral, consiste na diferença crescente entre o desempenho do processador e o tempo de acesso à memória. Muitas dessas técnicas focaram em novos esquemas de codificação de instruções enquanto outras utilizavam técnicas para compressão de instruções. Enquanto que as técnicas de codificação mudam a forma como as instruções são codificadas mas ainda mantendo a semântica dessas instruções, as técnicas de compressão focam diretamente na diminuição do tamanho das instruções para reduzir o tamanho final do código. Todas essas técnicas possuem um único objetivo: reduzir a quantidade de dados armazenados na memória principal e, consequentemente, diminuir os *cache misses* e o volume de dados transferidos entre memória e processador. Especificamente, essas técnicas têm sido aplicadas em arquiteturas genéricas e em arquiteturas voltadas para domínios específicos de aplicações, como os sistemas embarcados [8–11].

Os principais benefícios de um código menor são o menor consumo de energia, maior velocidade de execução e menor uso de memória. Tais características são especialmente desejáveis em sistemas embarcados, devido ao consumo de energia restrito, limitada capacidade de processamento (em comparação com computadores *desktop* e servidores) e baixa capacidade de memória.

A técnica de codificação de instruções PBIW [12, 13] foi projetada, inicialmente, para arquiteturas que buscam instruções longas na memória como processadores baseados em arquiteturas Explicitly Parallel Instruction Computing (EPIC) [14,15], Very Long Instruction Word (VLIW) [16], arquiteturas reconfiguráveis com várias unidades funcionais e arquiteturas de alto desempenho baseadas em matrizes de unidades funcionais. A técnica é composta por um algoritmo que realiza a fatoração de operandos [17–19] e por uma memória cache chamada de Pattern Cache (P-Cache).

Dada a limitação existente em comparar técnicas de codificação e compressão de código, este trabalho de mestrado volta-se para o projeto e implementação de uma infraestrutura em software para adição de algoritmos de codificação código e, consequentemente, para a investigação e exploração do espaço de projeto de decodificadores de código. Especificamente, o algoritmo PBIW foi adotado para codificar programas gerados com o conjunto de instruções VEX [20], a partir da infraestrutura implementada. Como estudo de caso, o processador *softcore* embarcado ρ-VEX [21] foi utilizado para execução das instruções codificadas por

PBIW. Dessa forma, o presente trabalho atua tanto na geração de instruções codificadas a partir do algoritmo PBIW, como também, explora o espaço de projeto de decodificadores de instruções junto à via de dados do processador ρ-VEX. Essa exploração objetivou minimizar o impacto desse circuito decodificador no fluxo de execução de instruções no processador alvo.

A justificativa para o presente trabalho se dá pela necessidade atual de reduzir o impacto da latência no acesso à memória sobre o desempenho final de programas. Também deve-se atentar para o impacto que um circuito de decodificação de instruções poderá acarretar no fluxo de execução das instruções. Assim, busca-se um compromisso entre a taxa de compressão de programas e a sobrecarga na utilização de um circuito decodificador junto ao hardware do processador.

O texto dessa desta dissertação está organizado da seguinte forma:

- no Capítulo 2 será realizada uma revisão da literatura de compressão e codificação de instruções. Devido à existência de uma vasta gama de esquemas de compressão e compactação de código disponíveis, a abordagem aqui utilizada apresenta os principais conceitos e discute os resultados das propostas mais próximas (quanto aos objetivos e metodologia) deste trabalho. No entanto, referências bibliográficas são fornecidas para o leitor interessado em conhecer, com mais detalhes, outras técnicas de compressão e codificação.

- no Capítulo 3 será apresentada a técnica de codificação de instruções PBIW e a infraestrutura de codificação projetada e implementada neste trabalho. Neste capítulo, tem-se como objetivo principal a compreensão dos conceitos e princípios de projeto para adoção da técnica PBIW para uma ISA específica. A apresentação da infraestrutura de codificação será de forma a permitir ao leitor o entendimento do fluxo de execução e das decisões a serem consideradas para adotar tal infraestrutura para um novo algoritmo de codificação ou um novo conjunto de instruções usando os algoritmos de codificação já implementados.

- O Capítulo 4 descreve as decisões de projeto para adoção da técnica PBIW sobre o processador *softcore* embarcado ρ-VEX. Além de apresentar formatos de instruções e padrões codificados, este capítulo também apresenta e discute o projeto e implementação de duas versões do decodificador de instruções PBIW junto à microarquitetura do processador ρ-VEX.

- O Capítulo 5 apresenta e discute todos os experimentos realizados e resultados obtidos visando à validação e avaliação da técnica de codificação PBIW sobre o processador ρ-VEX usando a infraestrutura de codificação descrita no Capítulo 3. Este capítulo abrange desde experimentos estáticos voltados para avaliar a capacidade de codificação da técnica PBIW sobre ρ-VEX até experimentos envolvendo área ocupada, potência dissipada (dinâmica e estática) e frequência máxima suportada pelo processador com a adição do hardware de decodificação PBIW.

- As conclusões deste trabalho assim como proposições para desenvolvimentos de trabalhos futuros são apresentadas no Capítulo 6.

Capítulo 2

Fundamentação Teórica

Na computação, a área de compressão de dados começou a ser desenvolvida no final da década de 1940. Claude Shannon e Robert Fano, em 1949, projetaram um algoritmo que construía um código de prefixos baseado em probabilidades que ficou conhecido como código de Shannon-Fano [22, 23]. Porém, tal algoritmo não construía códigos com tamanho ótimo. Posteriormente em 1952, Huffman propôs um algoritmo diferente que também gerava códigos de prefixos, mas de tamanho ótimo. Nas décadas posteriores, outros algoritmos foram criados, por exemplo LZW [24] e compressão aritmética [25]. A principal motivação para estudar e aplicar tais técnicas foi econômica: equipamentos de armazenamento de dados, independente da tecnologia, possuíam custo muito elevado, sendo então necessário minimizar espaço ocupado pelos dados removendo informação redundante.

Essa mesma motivação foi levada para o contexto de compressão de programas. Especificamente para o mercado de sistemas embarcados, as técnicas de compressão de programas/código mostraram-se muito atrativas e foram alvo de extensa pesquisa na década de 90.

Neste capítulo serão apresentadas técnicas para a redução do tamanho de programas na memória. Algumas dessas técnicas possuem características compressoras — retirando redundâncias do código original e gerando uma saída sem nenhuma estrutura semântica — e outras possuem características codificadoras — reduzindo o código original mas mantendo uma estrutura semântica. As técnicas de compressão serão abordadas na Seção 2.1 e as técnicas de codificação serão abordadas na Seção 2.2. Na Seção 2.3 apresenta-se as considerações finais do capítulo.

2.1 Técnicas de Compressão de Código

De acordo com a classificação sugerida em [26], existem quatro tipos de algoritmos principais usados em diversas técnicas de compressão de código: compressão de Huffman, compressão aritmética, compressão baseada em dicionários e via técnicas de modelagem (por exemplo, o modelo de Markov).

Diversos autores utilizam diferentes definições para a taxa de compressão de código. Neste trabalho, utiliza-se a Equação 2.1 como a taxa de compressão de um programa.

$$\text{Taxa de Compressão} = \frac{\text{Tamanho Comprimido} + \text{Overhead}}{\text{Tamanho Original}} \tag{2.1}$$

O *Tamanho Comprimido* corresponde ao tamanho (em bytes) após a aplicação da técnica de compressão de código. O termo *Overhead* corresponde ao tamanho (em bytes) de alguma informação adicional presente na compressão e o *Tamanho Original* corresponde ao tamanho (em bytes) do código original.

2.1.1 Compressão de Huffman

O sistema de compressão proposto por Huffman em 1952 [27] propõe utilizar as probabilidades de ocorrência dos símbolos de um conjunto de dados a ser comprimido para determinar códigos, de tamanho variável, para cada símbolo. O foco está em encontrar cadeias de bits mais curtas para representar os símbolos mais frequentes e cadeias mais longas para os símbolos menos frequentes.

Para a compressão deve-se, primeiramente, descobrir as frequências dos símbolos a serem codificados. Em seguida, deve-se atribuir os códigos aos respectivos símbolos. Por último, utiliza-se uma estrutura de dados para sua representação, de forma que seja pequena no processo de armazenamento e rápida ao ser empregada na descompressão. Para a realização das últimas duas etapas, Huffman encontrou uma solução baseada em árvore binária. Nela, cada nó folha contém um símbolo e sua respectiva frequência, cada nó intermediário contém somente a soma das frequências de suas sub-árvores e a raíz contém a soma das frequências de todos os símbolos do conjunto. As arestas esquerdas recebem peso zero e as direitas recebem um. O caminho da raíz até as folhas gera a compressão de Huffman.

Tenha como exemplo a cadeia AAAAAAAABBBBCCDE na qual cada letra pode ser representada por 3 bits(A = 000, B = 001 e assim por diante). Dessa forma, a cadeia inteira será representada por 48 bits: 000000000000000000000000001001001001010010011100. Depois da compressão de Huffman, como visto na Figura 2.1, a mesma cadeia pode ser representada como 000000001010101011011011101111 totalizando apenas 30 bits. A compressão foi de 18 bits, ou seja, a taxa de compressão foi de 62,5%, sem considerar o *overhead* da tabela de símbolos.

As vantagens do uso dessa técnica são a alta velocidade de compressão e descompressão. As desvantagens são perda de eficiência na compressão quando as frequências dos símbolos diferem de potências negativas de 2. A utilização desse esquema de compressão diretamente em instruções de máquina também possui algumas dificuldades. Devido a cada símbolo possuir um código com tamanho variável também é necessária uma análise prévia para solucionar o problema de endereçamento quando há saltos no fluxo de execução.

2.1.2 Compressão Aritmética

A compressão aritmética é, em determinadas situações, mais eficiente que a compressão de Huffman [25]. Ela utiliza probabilidades — representadas com números reais — dos símbolos ao invés de frequências. Dessa forma, as probabilidades dividem um intervalo —

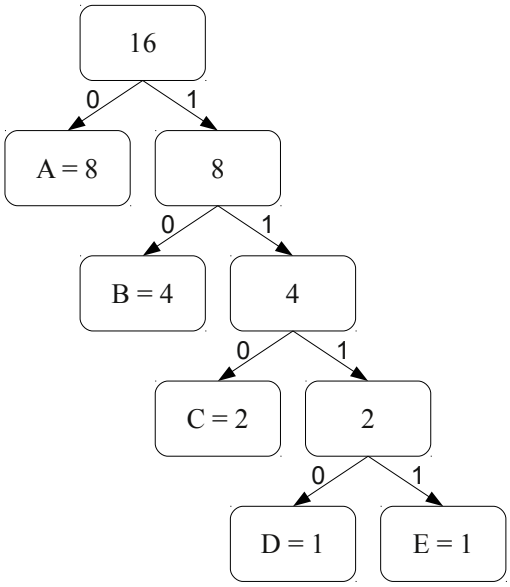


Figura 2.1: Exemplo de árvore de Huffman para a cadeia AAAAAAAABBBBCCDE.

normalmente o intervalo $0 \leq x < 1$ (representado por $[0, 1)$) onde $x \in \mathbb{R}$ — em sub-intervalos de tamanho proporcional a cada probabilidade, a qual cada símbolo é unicamente associado. A compressão se preocupa em sempre manter registro destes sub-intervalos.

A compressão aritmética consiste em representar uma mensagem em intevalos $[0, 1)$ de números reais. Quanto maior for a mensagem, menor será o intervalo necessário para representá-la e maior será o número de bits necessários para representar este intervalo. Símbolos sucessivos na mensagem reduzem o tamanho do intervalo de acordo com as probabilidades geradas pelo modelo em uso. Os símbolos mais comuns reduzem o intervalo menos que os símbolos menos comuns. Essa menor redução no intervalo pode ser representada com menos bits e então adicionam poucos bits à mensagem comprimida.

Como exemplo, suponha que deseja-se comprimir uma mensagem *eaii!* composta pelo alfabeto $\Sigma = \{a, e, i, o, u, !\}$. Usando um modelo fixo — que conta a frequência de cada item do alfabeto e calcula suas probabilidades — tem-se as probabilidades presentes na Tabela 2.1. Esta tabela contém valores aleatórios com o objetivo de auxiliar a explicação do exemplo.

Símbolo	**Probabilidade**	**Intervalo**
a	0.2	$[0, 0.2)$
e	0.3	$[0.2, 0.5)$
i	0.1	$[0.5, 0.6)$
o	0.2	$[0.6, 0.8)$
u	0.1	$[0.8, 0.9)$
$!$	0.1	$[0.9, 1.0)$

Tabela 2.1: Exemplo de modelo fixo para o alfabeto $\{a, e, i, o, u, !\}$.

Inicialmente sabe-se que o intervalo padrão é $[0, 1)$. Depois de ler o primeiro símbolo e, o compressor restringe o intervalo de compressão para $[0.2, 0.5)$. O segundo símbolo a

Inicialmente:		[0,	1)
Depois de ver:	e	[0.2,	0.5)
	a	[0.2,	0.26)
	i	[0.23,	0.236)
	i	[0.233,	0.2336)
	$!$	[0.23354,	0.2336)

Figura 2.2: Compressão sendo executada passo a passo.

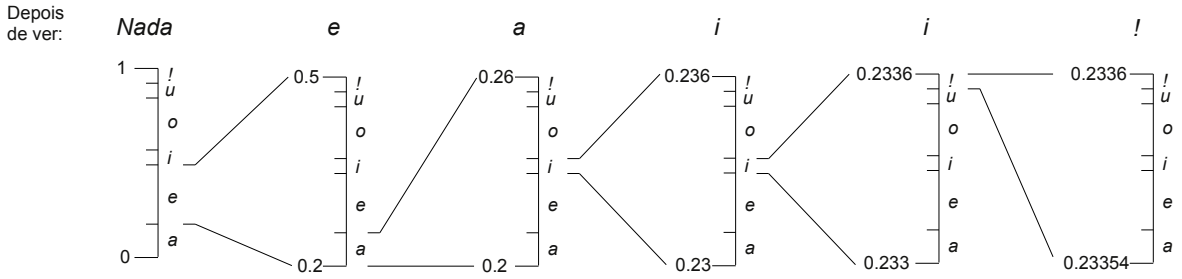


Figura 2.3: Processo da compressão passo a passo com cada estágio de restrição do intevalo sendo ampliado.

restringirá esse novo intervalo — $[0.2, 0.5)$, inicialmente alocado para comprimir o símbolo e — para o primeiro $1/5$ dele próprio, o intervalo $[0.2, 0.26)$.

O próximo símbolo, i, é alocado no intervalo $[0.5, 0.6)$ que aplicado proporcionalmente ao intervalo $[0.2, 0.26)$ gera o intervalo mais estreito $[0.23, 0.236)$. Procedendo desta maneira até o final, a mensagem comprimida pode ser vista nas Figuras 2.2 e 2.3, nas quais é possível notar que a compressão se dá principalmente por sub-divisões de cada intervalo ao longo do processamento da mensagem.

A descompressão é realizada através dos mesmos passos realizados na compressão. Para a descompressão não há necessidade de ser dado um intervalo, basta que seja um número contido nele. Por exemplo, no intervalo $[0.23354, 0.2336)$ os números 0.23355, 0.23354, 0.23357 ou até mesmo 0.23354321 representam a mesma informação comprimida.

Considera-se o número real 0.23355 para exemplificar a descompressão. Inicialmente, o descompressor busca em qual intervalo o número 0.23355 está contido. Começando com o maior intervalo disponível, que no caso é $[0, 1)$, ele restringe o intervalo que contém o número real dado, que no caso é o intervalo $[0.2, 0.5)$. Neste ponto, o descompressor já possui a informação que o primeiro símbolo comprimido é o símbolo e. A partir desse novo intervalo, o descompressor escolhe novamente o sub-intervalo que contenha o número real dado, que neste caso é o sub-intervalo $[0.2, 0.26)$. Ao fazer esta escolha, o descompressor tem uma nova informação a respeito do segundo símbolo comprimido, que é o símbolo a. A descompressão se dá realizando esses passos sucessivamente até encontrar o símbolo especial de finalização de mensagem, que neste caso é o símbolo !.

2.1.3 Compressão Baseada em Dicionários

A compressão baseada em dicionários é uma das mais utilizadas nos projetos de compressores e descompressores de códigos. A popularidade deve-se a facilidade de implementação em hardware, a velocidade e o nível de compressão alcançados.

O funcionamento dessa técnica é simples: um trecho de código que é repetido em diversas outras regiões é substituído por um índice em uma tabela que armazena esse código. Desta forma, ao invés de ser repetido diversas vezes pelo código, cada repetição é substituída por uma referência onde este código pode ser encontrado no dicionário, como pode ser visto na Figura 2.4.

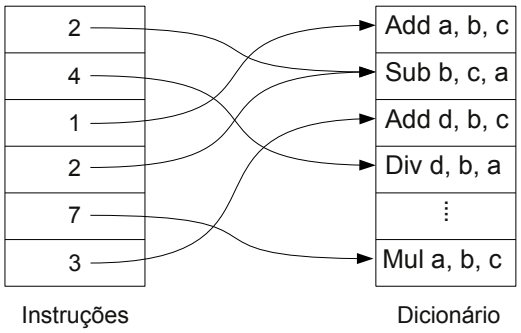


Figura 2.4: Exemplo de um dicionário simples de instruções.

A partir do conceito fundamental de guardar repetições em uma tabela (dicionário) foram derivadas diversas técnicas de compressão de código baseadas em dicionário conforme descrito nas subseções a seguir.

Equilíbrio de Tempo e Espaço

Os autores Sreejith K. Menon e Priti Shankar propõem em [8] uma compressão baseada em dicionários na qual conseguem eliminar a dificuldade de armazenamento de instruções de tamanho variável. Há a criação de diversos dicionários para cada classe de instruções, explorando a similaridade entre elas. A taxa média de compressão obtida é a partir de 73,5%.

Cada classe de instruções é um grupo e cada grupo contém instruções de uma determinada classe. Todas as instruções de cada grupo são separadas em dois segmentos. A posição do ponto de divisão dos segmentos, no qual as instruções são separadas em duas partes, varia de grupo para grupo. O primeiro segmento sempre contém um opcode — de tamanho fixo para todas as classes de instruções — que identifica a qual classe cada instrução pertence. No esquema utilizado pelos autores, eles asseguram que cada segmento esteja contido em dois bytes. O limite de tamanho simplifica a implementação das tabelas de índice e do hardware de descompressão.

Os autores utilizaram um processador VLIW da Texas Instruments® para teste de seu esquema de compressão. Como este processador despacha pacotes contendo oito instruções, o esquema de compressão variável proposto não garante que cada busca de pacote comece em um novo limite de byte. Por isso, é necessário um preenchimento com zeros no final

para que os pacotes sejam explicitamente endereçáveis. Isso reduz ligeiramente a taxa de compressão, em torno de 1,5%, em comparação com a taxa de compressão média.

Quanto à descompressão das instruções, foram descritas duas abordagens: a descompressão serial e a descompressão paralela.

Na descompressão serial, a instrução original é recuperada através de um *pipeline* de três estágios. No primeiro estágio há a identificação dos dicionários e classe da instrução. Desta forma, sabe-se a quantidade de bits ocupados pela instrução comprimida atual e assim o seu limite, colaborando na identificação do início da próxima instrução e sua descompressão no próximo ciclo. O segundo estágio é usado para enviar o deslocamento para o dicionário correspondente recuperando assim, no terceiro estágio, a instrução descomprimida.

Para a descompressão paralela, parte-se do princípio que todas as instruções comprimidas possuem dois segmentos distintos — lembrando que o primeiro segmento contém um opcode para identificação de sua classe — assim, um pacote de instruções pode ser rearranjado em duas regiões: uma região contendo somente os opcodes em sequência e outra região contendo, também em sequência, os restantes dos bits de cada instrução. A descompressão é feita depois de se identificar todas as classes de instruções presentes no pacote e relacionar com seus respectivos segmentos. Depois de descomprimidas, elas podem ser executadas em paralelo pelas unidades funcionais do processador.

A taxa de compressão média foi de 73,5%, incluindo o dicionário e a Local Address Table (LAT), para programas do MediaBench. No esquema com o dicionário de tamanho fixo os autores obtiveram uma compressão de 75,5% em média (MediaBench).

A vantagem observada nos dicionários de tamanho fixo, são o baixo *overhead* de descompressão. O descompressor foi testado em posições entre memória e cache. Assim a cache do processador poderia ser utilizada para armazenar as instruções já descomprimidas. Para os testes foi usada a versão de descompressão serial, devido ao seu projeto de hardware simples, obtendo-se uma velocidade média de descompressão de 22 bits/ciclo para o dicionário fixo e de 13 bits/ciclo para o dicionário variável.

Finalmente, conclui-se que a degradação da desempenho é de, em média, 6% com uma compressão de 78% utilizando uma cache de 64KBytes.

Isomorfismo de Instruções

Em [28] os autores propõem uma compressão baseada em dicionários mas estendem o conceito aplicando-o no isomorfismo de instruções. Instruções isomorfas são aquelas que possuem o mesmo opcode mas têm operandos ligeiramente diferentes — como instruções iguais que usam alguns registradores em comum — ou possuem opcodes diferentes com operandos iguais. Além disto, este método é aplicado em instruções que são usadas intensamente ao longo do programa.

Desta forma, as instruções são divididas em duas: o grupo de opcodes e o grupo de operandos. Cada grupo é mapeado para um dicionário específico. Esse esquema possibilita que o reúso de dados presentes em ambos os dicionários seja maior, como ilustrado na Figura 2.5. Assim, gera-se uma melhor compressão e menor uso de memória do código em execução.

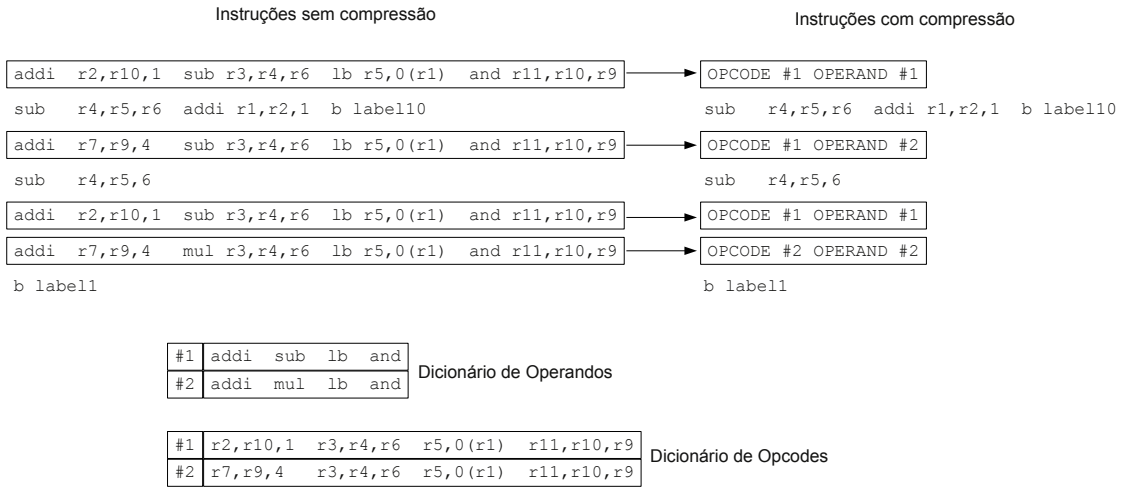


Figura 2.5: Compressão usando isomorfismo de instruções.

A compressão foi testada e explorada em instruções VLIW. Cada instrução VLIW contém uma quantidade fixa de operações que podem ser entendidas como "sub-instruções" que realizam cálculos simples. Devido às operações de cada instrução VLIW serem executadas concorrentemente, não é possível desviar o fluxo de execução para uma operação específica de uma instrução VLIW. O desvio só é possível se executado de e para instruções VLIW.

Como uma instrução compactada contém uma palavra que aponta para um item do dicionário de opcodes e outra palavra que aponta para um item do dicionário de operandos, o tamanho total de uma instrução compactada foi definido como o mesmo tamanho de uma operação não compactada. O motivo da adoção de tal medida foi para alinhamento das palavras com a cache. Isso pode gerar uma compressão pior do que palavras de tamanho variável, mas simplifica o projeto do hardware e acelera o processo de busca e descompressão das instruções.

Nos testes, os autores consideraram que operações NOP — que são utilizadas em instruções VLIW com o objetivo de manter o tamanho da instrução fixo — foram removidas dos programas originais. Também supõe-se que não há nenhuma limitação de tamanho dos dicionários. A média da taxa de compressão do código é de 63%, 69% e 71% em PI4, PI8 e PI12, respectivamente(os PI's são configurações de unidades funcionais simuladas que a arquitetura possui).

Notou-se também que a taxa de compressão é afetada pelo número de entradas disponíveis no dicionário e as frequências de aparecimento das instruções.

2.2 Técnicas de Codificação

As técnicas de codificação reduzem o espaço ocupado por um código em memória através da mudança da forma e/ou tamanho das representações binárias das instruções pertencentes a um processador. Enquanto as técnicas de compressão reduzem o espaço ocupado pelo código retirando, essencialmente, informações redundantes, as técnicas de codificação diminuem o espaço de representação das instruções, mantendo uma estrutura semântica para o código.

Muitas vezes as técnicas de codificação lançam mão da redução do tamanho dos campos presentes nas instruções originais, por exemplo, a redução do número de registradores endereçáveis (de trinta e dois para oito registradores, no caso do MIPS16) e a redução do tamanho dos campos de números imediatos. Também há uso da reorganização dos campos para otimizar o espaço reduzido de codificação. Desta forma, as instruções reduzidas mantêm uma semântica, facilitando o trabalho na etapa de decodificação das mesmas.

Nas seções 2.2.1, 2.2.2 e 2.2.3 serão explicadas as técnicas de codificação MIPS16, Thumb e SPARC16, respectivamente.

2.2.1 MIPS16

A extensão MIPS16 [1] é uma forma de codificação para instruções do processador Microprocessor without Interlocked Pipeline Stages (MIPS) criada com o objetivo de aumentar a densidade de código, visto que cada instrução presente no Instruction Set Architecture (ISA) do MIPS possui o tamanho fixo de 32 bits. Arquiteturas Reduced Instruction Set Computer (RISC) necessitam de uma grande largura de banda para alimentar o processador, visto que as instruções são longas (4 bytes no caso do MIPS) e de rápida execução. Para aplicações embarcadas que possuem pouca memória e necessitam de um baixo consumo de energia, esta não é uma solução atrativa [1]. Para minimizar estes problemas a extensão MIPS16 (para processadores MIPS) foi então desenvolvida.

Esta extensão é 100% compatível com as variantes de 32 bits(MIPS-I/II) e 64 bits(MIPS-III), tanto arquiteturalmente quanto no modelo de programação. Cada instrução MIPS16 corresponde a uma instrução MIPS, podendo ser traduzidas *on-the-fly* utilizando um hardware simples. O esquema de decodificação pode ser visto na Figura 2.6. Há possibilidade da tradução ser feita serialmente entre a cache e o sistema de decodificação das instruções, ou em paralelo em um decodificador de 16/32 bits.

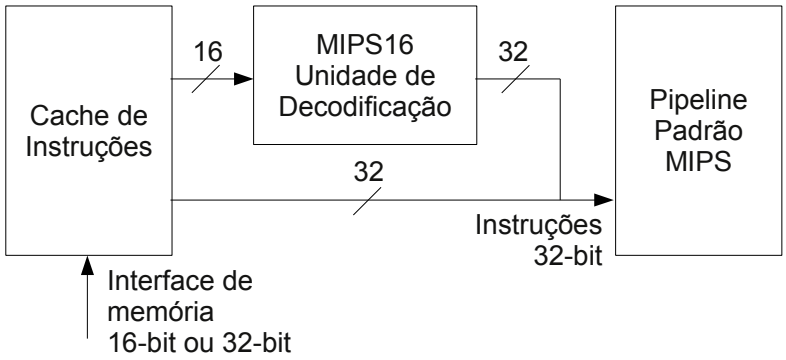


Figura 2.6: Exemplo de decodificação de uma instrução MIPS16 [1].

As instruções podem ser definidas como um conjunto reduzido das instruções originais. Assim, três componentes das instruções originais MIPS foram tratados: opcodes, números de registradores e imediatos. Os tamanhos dos campos foram reduzidos depois de um estudo estatístico [1] de quais instruções eram mais utilizadas em uma grande variedade de softwares. Os campos de opcode e *function* foram reduzidos de 6 para 5 bits; o número de operandos registradores especificados na instrução passaram de 3 para 2 também havendo uma redução

no tamanho do campo de 5 para 3 bits; e os imediatos foram os que mais obtiveram redução de tamanho: de 16 para 5 bits.

Um exemplo simples utilizando uma rotina em C demonstra o efeito da codificação. Esta rotina retorna o valor absoluto de um inteiro passado como parâmetro e pode ser vista na Figura 2.7. A versão equivalente em *assembly* MIPS e MIPS16 podem ser vistas respectivamente nas Figuras 2.8 e 2.9. O código MIPS requer vinte e quatro bytes (seis instruções, quatro bytes cada) enquanto que o código MIPS16 requer apenas doze bytes (seis instruções, dois bytes cada), um taxa de compressão de 50%.

```
if (x >= 0)
  return x;
else
  return -x;
```

Figura 2.7: Código C de exemplo para as rotinas em MIPS e MIPS16.

```
    ADD     $v0, $a0, $zero     # v0 <- a0
    SLT     $t0, $v0, $zero     # Compare v0 com zero;
    BNE     $t0, $zero, L1
    JR      $ra                 # Se v0>=0 Retorne v0;
L1: SUB     $v0, $zero, $v0     # Se v0<0 então v0=0-v0;
    JR      $ra
```

Figura 2.8: Código MIPS equivalente ao código C da Figura 2.7.

```
    ADDIU   $v0, $a0, 0     # v0 <- a0
    SLTI    $v0, 0          # Compare v0 com zero;
    BTNEZ   L1
    JR      $ra             # Se v0>=0 Retorne v0;
L1: NEG     $v0, $v0        # Se v0<0 então v0=0-v0;
    JR      $ra
```

Figura 2.9: Código MIPS16 equivalente ao código C da Figura 2.7.

Também foram adicionadas instruções auxiliares e funcionalidades extras, como instruções estendidas e *offsets load/store* deslocados (*shifted*). A funcionalidade de endereçamento relativo ao PC e SP do MIPS32 também foi mantida no MIPS16. Algumas instruções se referem a registradores implicitamente, para conservar o reduzido espaço de nomes dos registradores nas instruções.

Foi criada uma instrução JALX para a troca dos modos de operação entre 16 e 32 bits. Assim, sub-rotinas podem alterar entre código MIPS16 e MIPS (e vice-versa). Como somente oito registradores estão disponíveis diretamente para as instruções, foram criadas duas novas instruções para o MIPS16: MOV32R e MOVR32. O objetivo de ambas é copiar dados entre registradores gerais do MIPS e específicos[1] do MIPS16. Há também as instruções de comparação — utilizadas para desvio de fluxo — que utilizam um registrador implícito como destino.

[1]Registradores acessíveis através do novo espaço de representação de três bits das instruções reduzidas.

A taxa de compressão obtida no código MIPS16 por [1] foi de 60%, mas os efeitos no desempenho foram complexos e variáveis: mesmo reduzindo o tamanho das instruções para aumentar a taxa de acerto no cache, foram necessárias mais instruções para executar os mesmos processamentos que as instruções originais não reduzidas. Para MicroMIPS, a taxa de compressão foi de aproximadamente 65%.

2.2.2 Thumb

A codificação Thumb [2] e sua extensão, Thumb 2, partem do mesmo princípio utilizado pelas instruções MIPS16 e MicroMIPS: o aumento da densidade de instruções na memória. Este objetivo é alcançado na redução do tamanho das instruções originais pela metade: as instruções Thumb têm tamanho de 16 bits contra os 32 bits das instruções Advanced RISC Machine (ARM) originais. Um exemplo de decodificação de uma instrução Thumb pode ser visto na Figura 2.10. Cada instrução Thumb possui uma instrução ARM equivalente, logo há uma relação um-a-um entre os conjuntos de instruções.

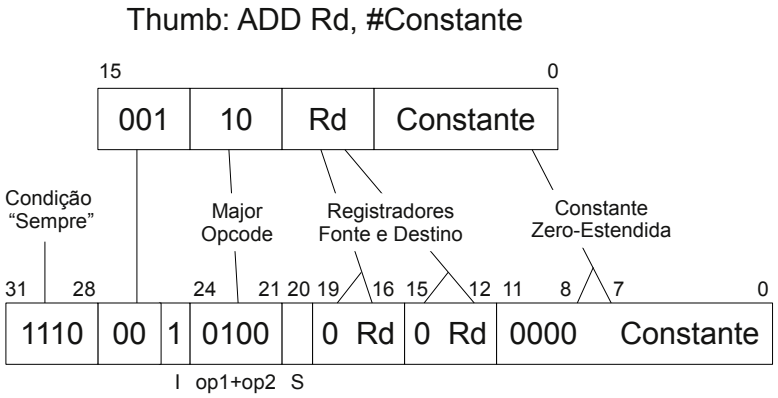


Figura 2.10: Exemplo de mapeamento de uma instrução [2].

Devido a natureza embarcada dos processadores ARM — como restrições no consumo de energia e tamanho dos circuitos — três categorias de instruções ARM foram consideradas para redução de tamanho: as usadas mais frequentemente em softwares de clientes, as necessárias ao compilador para emitir código compactado e aquelas com alguma redundância nos opcodes fixos.

Assim, as instruções Thumb contêm um subconjunto de trinta e seis classes constituídas a partir das instruções de 32 bits. O resultado é um aumento de 30% na densidade do código sobre o código ARM nativo [2].

Como os processadores "Thumb-aware" podem executar tanto códigos Thumb como ARM nativamente, o projetista pode escolher quais instruções utilizar, usando em cada sub-rotina o tipo de código que tiver melhor apelo. Um exemplo é a utilização de instruções Thumb em trechos de código onde alto desempenho não é necessário.

O decodificador Thumb foi desenvolvido para uma implementação em um ciclo de *clock* não utilizado no pipeline. Desta forma, as instruções codificadas podem ser executadas

utilizando o mesmo número de ciclos das instruções comuns. O esquema do processador ARM7TDMI contendo o decodificador Thumb pode ser visto na Figura 2.11.

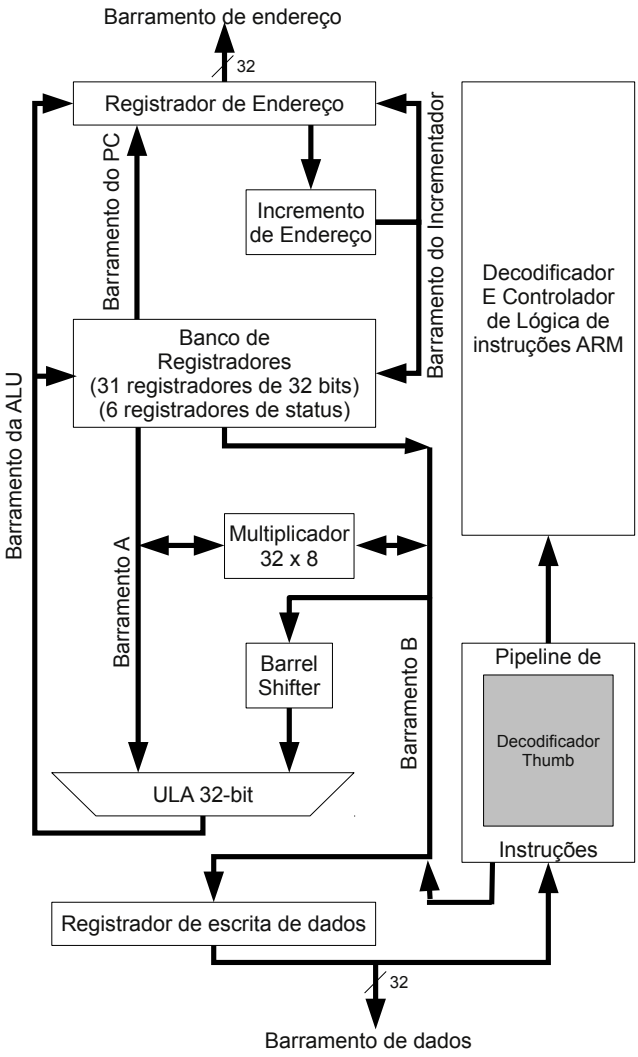


Figura 2.11: Núcleo do processador ARM7TDMI contendo decodificador Thumb [3].

Instruções Thumb e ARM também usam o mesmo espaço de memória, isto é, rotinas contendo código Thumb e ARM não necessitam estar em memórias distintas. Rotinas que necessitem de menos espaço podem ser compiladas em Thumb enquanto rotinas de desempenho crítico podem ser compiladas em ARM. Esta solução foi obtida utilizando um bit no registrador de status do processador que representa qual conjunto de instruções a ser processado (Thumb ou ARM). Este bit de estado pode ser alterado ao se alternar entre cada sub-rotina (para processadores com suporte a extensão Thumb 2, não existe mais a necessidade de alteração do estado do processador através deste bit).

Em testes com memórias acessadas através de barramentos largos (32 bits), o código Thumb executa 15% mais lento que o código ARM devido a instruções extras para completar uma tarefa. Entretanto, em memórias acessadas via barramento estreito (8 a 16 bits), o código Thumb consegue ser 30% mais rápido que o código ARM devido ao reduzido número de acessos à memória para receber as instruções. Mesmo sendo necessárias mais instruções

Thumb de 16 bits para realizar uma tarefa, elas também reduzem em 30% o consumo médio de energia [2].

Um exemplo simples utilizando uma rotina em C demonstra o efeito da codificação. Esta rotina retorna o valor absoluto de um inteiro passado como parâmetro e pode ser vista na Figura 2.7. A versão equivalente em *assembly* ARM e Thumb podem ser vistas respectivamente nas Figuras 2.12 e 2.13. O código ARM requer doze bytes (três instruções, quatro bytes cada) enquanto que o código Thumb requer apenas oito bytes (dois bytes cada), ou seja, um taxa de compressão de 66%.

```
CMP     r0, #0;         %Compare r0 com zero;
RSBLT   r0, r0, #0;     %Se r0<0 então r0=0-r0;
MOV     pc.lr;          %Retorne;
```

Figura 2.12: Código ARM equivalente ao código C da Figura 2.7.

```
CMP   r0, #0;     %Compare r0 com zero;
BGE   return;     %Retorne se é maior ou igual;
NEG   r0, r0;     %Se não, negue r0;
MOV   pc.lr;      %Retorne;
```

Figura 2.13: Código Thumb equivalente ao código C da Figura 2.7.

As instruções Thumb não têm como objetivo a substituição completa das instruções ARM. O objetivo principal é permitir ao projetista de software otimizar o código em nível de sub-rotinas com duas opções: ou reduzir o tamanho em memória do código original em 30% com instruções Thumb/Thumb2 ou ganhar 15% de desempenho com instruções ARM.

2.2.3 SPARC16

A arquitetura Scalable Processor Architecture (SPARC) foi escolhida pelos autores [4] para a criação de conjunto de mini-instruções de 16 bits depois de uma análise de 15 variações de sete ISA's — Alpha, MIPS, PowerPC, SPARC, ARM, i686/x86_64 e m68k, em ordem crescente de densidade de código — e representa um bom balanço entre oportunidade de compressão (tamanho grande de código) e resultado de impacto (considerável gama de usuários utilizam processadores SPARC como plataforma de hardware para suas aplicações).

Na revisão da literatura, os autores notaram que o hardware de decodificação, se bem desenhado, pode esconder a hierarquia de latências da memória e aumentar o desempenho do sistema, enquanto diminui o consumo de energia. Também durante a análise da literatura, notaram que em alguns resultados apresentados pela ARM, o conjunto de instruções Thumb obteve uma taxa de compressão de 55% a 70%, com o ganho de performance médio de 30% para barramentos 16 bits e 10% de perda para os de 32 bits. Quanto ao MIPS16, o nível de compressão foi de 60%.

Ao analisarem os softwares da arquitetura SPARC, notaram que 80% das instruções aritméticas precisam de seis ou menos bits para codificar imediatos. Também perceberam que os imediatos de 80% das instruções load/store podem ser representadas com nove ou menos bits. Baseados em toda a análise do conjunto de instruções do SPARC, eles desenvolveram

um modelo de programação linear para representar o problema e resolvê-lo de modo a obter os melhores tamanhos de campos para cada instrução, definindo um novo conjunto de formatos.

As instruções SPARC16 são simples o suficiente para serem traduzidas para suas instruções correspondentes de 32 bits em tempo de execução. O esquema de decodificação pode ser visto na Figura 2.14.

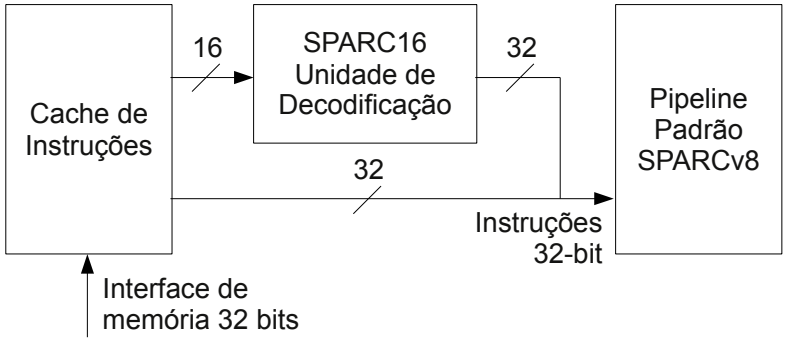


Figura 2.14: Exemplo de decodificação de uma instrução SPARC16 [4].

SPARC16 usa um subconjunto dos registradores presentes no SPARCv8: somente oito estão disponíveis para referência direta nas instruções SPARC16. Os autores adotaram uma solução semelhante à adotada no MIPS16: duas instruções (MOVT8to32 e MOV32to8) podem ser utilizadas para transferir dados entre os registradores "escondidos" e os "visíveis" pelas instruções. Estas instruções podem ser usadas para transferir dados (*spill data*) para os registradores escondidos, evitando transferir os mesmos dados para a memória, durante a etapa de alocação de registradores do compilador.

Um exemplo simples utilizando uma rotina em C demonstra o efeito da codificação. Esta rotina retorna o valor absoluto de um inteiro passado como parâmetro e pode ser vista na Figura 2.7. As versões equivalentes em *assembly* SPARCv8 e SPARC16 podem ser vistas respectivamente nas Figuras 2.15 e 2.16. O código SPARC requer 20 bytes (cinco instruções, quatro bytes cada) enquanto que o código SPARC16 requer apenas 12 bytes (seis instruções, dois bytes cada), ou seja, um taxa de compressão de 60%.

```
        SUBcc   %i0, %g0, %g0     # Compare i0 com zero;
        BL      L1
        RETL                      # Se i0>=0 Retorne i0;
L1:     SUB     %g0, %i0, %i0     # Se i0<0 então i0=0-i0;
        RETL
```

Figura 2.15: Código SPARCv8 equivalente ao código C da Figura 2.7.

Existem também instruções auxiliares que acessam registradores implicitamente, para resolver os mesmos problemas encontrados durante o desenvolvimento do MIPS16. Apesar da alocação de formatos de instruções ser fornecido por um método de programação linear, o SPARC16 foi desenvolvido para deixar um espaço de codificação reservado para futuras extensões (como foi o caso das instruções MOV8to32, MOV32to8 e EXTEND).

Os resultados de taxa de compressão dos programas do MediaBench foi de 58,1%, o MiBench foi de 57,6% e a média dos binários do kernel Linux de 64,3%. Também foi

```
    MOV     0, %i1
    CMP     %i0, %i1          # Compare i0 com zero;
    BL      L1
    RETL                      # Se i0>=0 Retorne i0;
L1: SUB     %g0, %i0, %i0     # Se i0<0 então i0=0-i0;
    RETL
```

Figura 2.16: Código SPARC16 equivalente ao código C da Figura 2.7.

comprovado que com um tamanho menor das instruções, o SPARC16 reduz substancialmente o número de cache *misses* para cada tamanho testado de cache.

2.3 Considerações Finais

Este capítulo apresentou uma revisão bibliográfica contendo pesquisas relacionadas a este trabalho. O capítulo foi organizado em duas seções principais: técnicas de compressão e técnicas de codificação de instruções. Percebe-se uma diferença fundamental entre as técnicas das duas seções. As técnicas baseadas em compressão têm como foco primordial a redução do tamanho dos programas e a minimização da latência na busca de instruções. As técnicas baseadas em codificação mantém semântica nas instruções codificadas possibilitando assim que tais instruções possam ser buscadas na memória e decodificadas já no *pipeline* do processador. Apesar das taxas de compressão provenientes das técnicas de compressão serem atrativas, o emprego dessas técnicas tem sido mais efetivo com a adoção de dicionários de código. Nas técnicas de codificação de instruções, apesar de efetivas na adoção, tais técnicas são específicas para determinados conjuntos de instruções. Nesse sentido, nota-se a necessidade de proposição de técnicas de mais alto nível, que possam ser facilmente estendidas e aplicadas em diferentes conjuntos de instruções.

A técnica de codificação PBIW [12], objeto de estudo deste trabalho, oferece tal possibilidade e será apresentada e discutida com mais detalhes no Capítulo 3. Em adição, apresenta-se também uma infraestrutura de codificação de código, inicialmente projetada para a técnica PBIW, que pode ser estendida para diferentes algoritmos de codificação e, consequentemente, utilizada para gerar codificação para programas de diferentes conjuntos de instruções.

Capítulo 3

Infraestrutura para Codificação Baseada na Técnica PBIW

Neste capítulo apresenta-se o projeto, funcionamento e detalhes de implementação de uma infraestrutura em software para codificar instruções com suporte para a técnica de codificação de instruções baseadas em padrões (Pattern Based Instruction Word). Na Seção 3.1 apresenta-se os principais conceitos da técnica PBIW. A Seção 3.2 descreve um algoritmo de otimização para a técnica de codificação PBIW cujo objetivo é minimizar o número de padrões codificados. Na Seção 3.3 são apresentados os detalhes de funcionamento e extensibilidade da infraestrutura de codificação. A Seção 3.4 detalha o projeto e fluxo de execução dessa infraestrutura. Os contextos de montagem e codificação PBIW são apresentados nas Seções 3.5 e 3.6, respectivamente. As considerações finais do capítulo são apresentadas na Seção 3.7.

3.1 Técnica de Codificação PBIW

A técnica de codificação de instruções PBIW [12] é focada na exploração da sobrejeção entre os conjuntos de instruções codificadas e seus respectivos padrões buscando reduzir o tamanho da instrução na memória. Esta técnica foi inicialmente projetada para arquiteturas que buscam instruções longas na memória. A técnica é composta por um algoritmo baseado em fatoração de operandos [17–19] e por uma tabela de padrões (P-Cache).

Depois das atividades de escalonamento das instruções e alocação dos registradores desenvolvidas pelo *back-end* do compilador, o algoritmo de codificação PBIW extrai operandos redundantes da instrução original, criando uma instrução codificada e um padrão.

Considerando que PBIW não é uma técnica de codificação com enfoque em uma ISA ou processador específico, o projeto e formato do padrão e instrução codificada devem levar em consideração algumas características fundamentais:

- Número de registradores de leitura;
- Número de registradores de escrita;

- Número de imediatos;
- Tamanho do índice para a tabela de padrões em memória (P-Cache);
- Quantidade de bits que indicam quais Unidade Funcionals (UFs)s devem executar ou cancelar suas operações.

Além dessas características, deve-se levar em conta, no projeto de instruções codificadas PBIW, que os operandos de uma instrução devem ser mantidos na instrução codificada, enquanto que sinais de controle devem ser armazenados no padrão obtido. A razão para isso é que tais operandos podem ser lidos no banco de registradores, paralelamente ao processo de decodificação de instruções.

A instrução codificada não possui redundância de operandos. Essa instrução é armazenada em uma Instruction Cache (I-Cache). O algoritmo de codificação mantém o registro da posição original dos operandos no padrão criado que é armazenado na P-Cache.

Cada instrução codificada PBIW tem $\mathcal{R}$ $(0 < \mathcal{R} \leq \mathcal{P})$ campos para representar registradores de leitura, onde $\mathcal{P}$ é o número máximo de portas de leitura disponível do banco de registradores. A instrução precisa ter $\lceil \log_2 \mathit{Regs} \rceil$ 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 de índice para a tabela de padrões, cujo tamanho é $\lceil \log_2 |P - Cache| \rceil$, em que $|P - Cache|$ indica a quantidade de entradas da tabela de padrões. Esse padrão é uma estrutura de dados que contém informações necessárias para compor a instrução codificada utilizada nos estágios de execução. O Algoritmo 1 mostra a versão original (não dedicada para uma ISA específica) do algoritmo de codificação PBIW.

A complexidade do algoritmo de codificação PBIW é diretamente relacionada ao algoritmo de busca utilizado para verificar se existe um padrão similar ao recém codificado (linhas 11 e 24). Caso seja usada uma busca linear sobre o conjunto de padrões, a complexidade assintótica do algoritmo de codificação será $O(n^2)$ onde n é o número de padrões criados. Dependendo da estrutura de dados utilizada, pode-se reduzir a complexidade assintótica do algoritmo de codificação. Como exemplo, se os padrões estiverem organizados numa estrutura do tipo árvore, a busca por um padrão usa $O(\log n)$ passos, tornando o algoritmo de codificação com complexidade $O(n \log n)$.

Um exemplo da técnica de codificação de instruções PBIW sobre uma ISA VLIW genérica é apresentada na Figura 3.1. Nessa figura são apresentadas duas instruções originais, duas instruções codificadas e um padrão após a codificação PBIW. O padrão é compartilhado entre ambas instruções codificadas, demonstrando a possibilidade de reúso.

O esquema de codificação PBIW também prevê o uso de bits de cancelamento na instrução codificada. Os bits de cancelamento informam quais operações presentes no padrão serão efetivamente executadas, possibilitando assim que somente alguns dos resultados sejam escritos. Essa funcionalidade pode ser explorada com uma otimização denominada Junção de Padrões (Seção 3.2). A quantidade N de bits de cancelamento corresponde ao número de operações existentes no padrão PBIW. Cada operação contém um opcode e um número fixo de operandos. Cada operando do padrão tem $\lceil \log_2 X \rceil$ bits, onde X é o número de campos dedicados para operandos (leitura e escrita da instrução PBIW).

Algoritmo 1: Algoritmo PBIW Genérico

Entrada: Conjunto de operações CO
Saída: Conjunto de instruções codificadas CI e padrões CP

```
1  início
2      Seja O um grupo de N (total de UFs da arquitetura) operações escalonadas;
3      para cada O ∈ CO faça
4          cria novo I;
5          cria novo P;
6          para cada op ∈ O faça
7              P.add(op.opcode);
8              para cada opnd ∈ op faça
9                  se op.opnd ∉ I então
10                     se espaço_livre(I) < 1 então
11                         se P ∉ CP então
12                             CP = CP ∪ P;
13                         fim
14                         I.add(índice de P em CP);
15                         CI = CI ∪ I;
16                         cria novo I;
17                         cria novo P;
18                     fim
19                     I.add(op.opnd);
20                 fim
21                 P.add(índice de op.opnd em I);
22             fim
23         fim
24         se P ∉ CP então
25             CP = CP ∪ P;
26         fim
27         I.add(índice de P em CP);
28         CI = CI ∪ I;
29     fim
30 fim
```

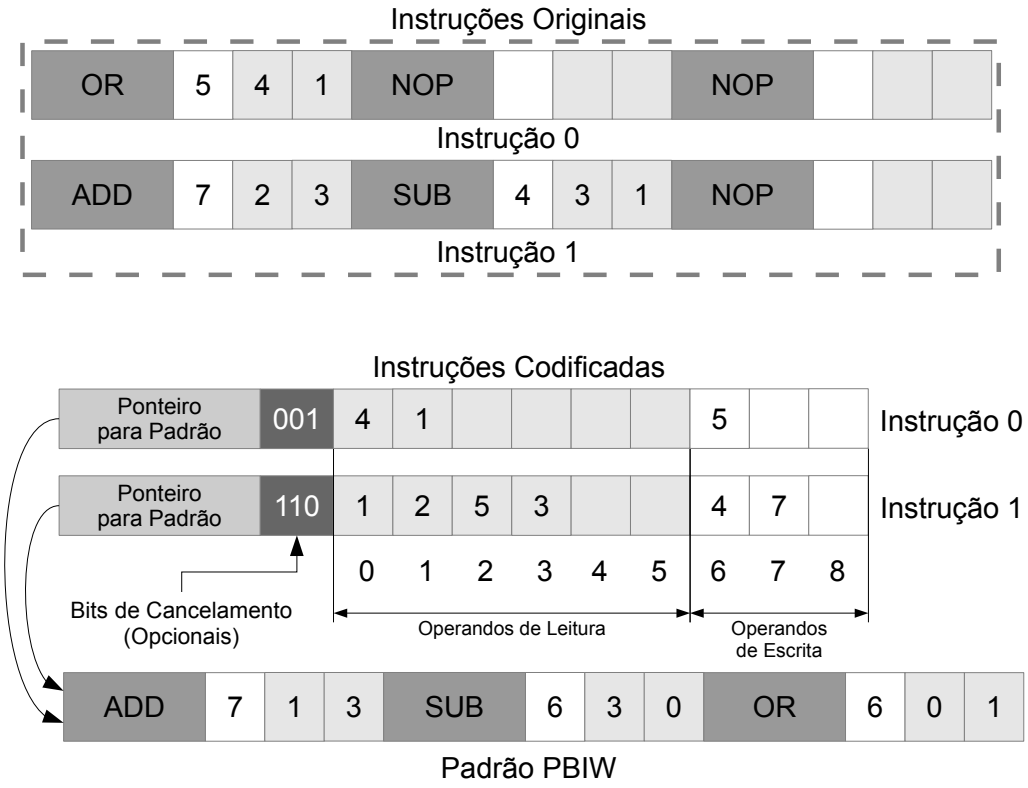


Figura 3.1: Instruções originais, codificadas e padrão PBIW.

Na Figura 3.1, pode-se observar que as operações ADD e SUB são executadas apenas pela instrução 1 e a operação OR é executada pela instrução 0. Os bits de cancelamento para ambas instruções são 110 e 001, respectivamente. Nos campos de cancelamento da instrução 1, a sequência 110 indica que a primeira e segunda operações (ADD e SUB) do padrão devem ser escritas em um registrador (ou memória se fosse uma operação STORE) e a terceira operação (OR) não deve ser executada: o resultado não será escrito em registrador ou memória (efeito NOP).

3.2 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. A otimização de junção de padrões é uma aplicação do problema de subconjuntos de soma máxima. Se a junção de padrões gera um padrão C de dois padrões A e B:

1. Todas as instruções que apontam para os padrões 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 cancelamento, para cancelar operações do padrão B e vice-versa;

3. Os padrões A e B são apagados do conjunto de padrões do programa.

Apresenta-se, a seguir, um conjunto de passos de uma heurística desenvolvida para construção do algoritmo de junção de padrões. Apesar de exemplificado para o conjunto de instruções VEX, esse algoritmo pode ser estendido para outras codificações PBIW. Informações mais detalhadas sobre a otimização de junção de padrões PBIW podem ser encontradas em [29].

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 compõem o padrão, considerando as restrições aquiteturais.

   Por exemplo, considerando a ISA do processador ρ-VEX (Figura 3.2), 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.

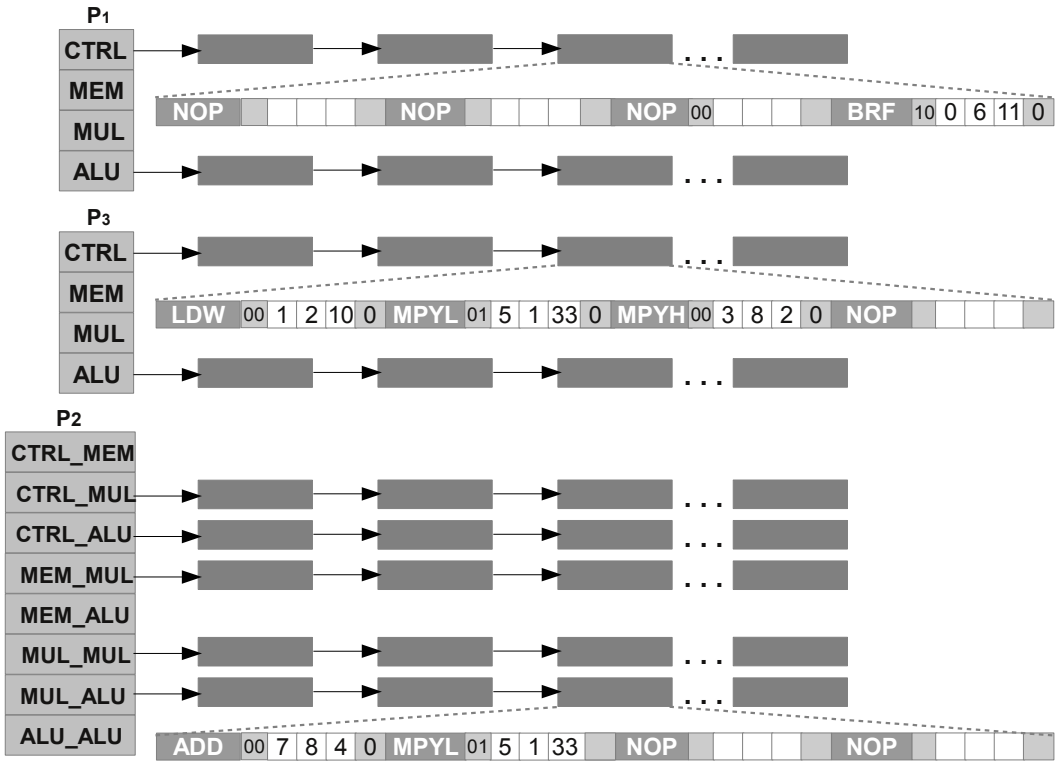


Figura 3.2: Categorias de padrões para o conjunto de instruções VEX.

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. Um exemplo de junção de padrões da categoria P2 é mostrado na Figura 3.3.

3. **Pós-processamento**: Remove padrões redundantes do conjunto de padrões e atribui endereços contíguos aos padrões do conjunto final.

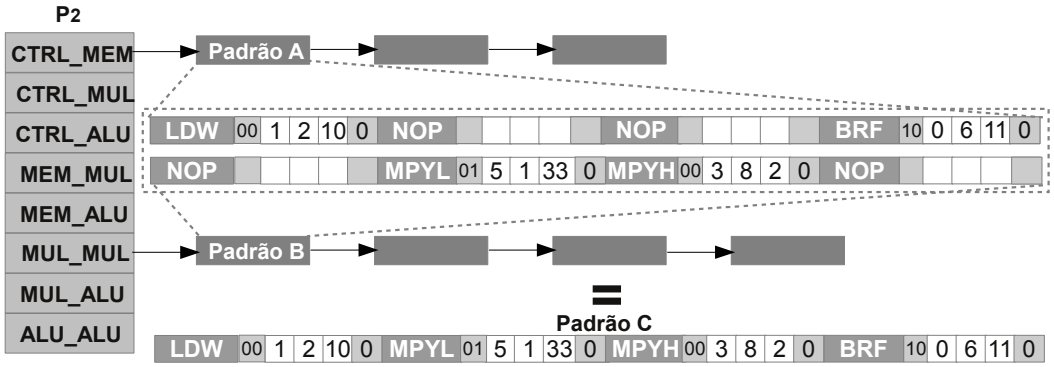


Figura 3.3: Junção entre padrões da categoria P2.

3.3 Infraestrutura de Codificação PBIW

A codificação PBIW possui como principal característica a aplicabilidade para diferentes tipos de ISAs. Assim, existe a necessidade de desenvolvimento de uma infraestrutura de software que seja modular e extensível, facilitando seu uso em diversos tipos de ISAs e representações específicas de entrada, saída e a possibilidade de troca do algoritmo genérico por uma versão especificamente otimizada para uma determinada ISA. A infraestrutura de codificação desenvolvida neste projeto procura atender esses objetivos. A infraestrutura de codificação permite ser estendida para suportar alguns tipos de formatos de programas (entradas para a infraestrutura).

- Texto em linguagem de montagem: arquivos texto de linguagem de montagem (`.s`, `.asm`, etc.) gerados por um compilador;
- Estruturas de dados finais providas pelo *back-end* de um compilador: possibilidade de integração direta entre o codificador PBIW e o *back-end* de um compilador;
- Binários Executable and Linkable Format (ELF): arquivos executáveis no formato ELF gerados por um compilador.

A saída da infraestrutura pode ser estendida para emitir, por exemplo, os seguintes formatos:

- Informações de depuração: informações técnicas acerca do processo de codificação PBIW;

- Arquivos fonte de descrição de hardware: arquivos fonte VHDL, Verilog ou outra linguagem de descrição de hardware, contendo as instruções codificadas de forma a serem sintetizadas e prototipadas em hardware (Field Gate Programable Array (FPGA) ou Application-Specific Integrated Circuit (ASIC)) para execução;
- Arquivos binários ELF: para execução no hardware de destino;

Exemplos de algoritmos PBIW genéricos e especificamente otimizados utilizados neste trabalho incluem:

- Codificação PBIW genérica: algoritmo parametrizável capaz de se adequar para diversas ISAs de entrada;
- Codificação VEX PBIW:
  - Versão 1.0: utilizado neste trabalho para gerar instruções codificadas VEX PBIW **com** restrições;
  - Versão 2.0: utilizado neste trabalho para gerar instruções codificadas VEX PBIW **sem** restrições;

Algumas funcionalidades também foram incluídas no projeto da infraestrutura de codificação, como:

- Possibilidade de incluir e encadear algoritmos de otimização no fluxo de execução da codificação PBIW;
- Independência dos conjuntos de instruções e padrões gerados no processo de codificação: cada contexto de codificação e otimizador PBIW possui seu próprio conjunto de instruções e padrões sem interferir nos dados dos demais.

Para o desenvolvimento dessa infraestrutura de software foi escolhida a linguagem de programação C++. As razões da escolha desta linguagem foram a portabilidade entre diferentes sistemas e arquiteturas e também por ser baseada no paradigma de orientação a objetos, que provê modularidade e extensibilidade.

Na aplicação da técnica PBIW sobre o conjunto de instruções VEX, realizada neste trabalho, usou-se como entrada um arquivo contendo linguagem de montagem VEX. Esta escolha foi motivada pela facilidade de *parsing* provida por uma gramática Flex-Bison presente no pacote do compilador C VEX. Esta gramática reconhece de forma completa o arquivo de montagem gerado pelo compilador C VEX. A escolha pelo *parsing* também resolve a limitação do compilador de não gerar um arquivo binário nativo para o processador ρ-VEX. Esta limitação existe porque o compilador C VEX é construído sobre a ISA VEX que é totalmente configurável (quantidade de unidades funcionais, tamanho do *pipeline*, entre outros). Assim, como a implementação específica da arquitetura (e codificação das instruções) do processador não é conhecida, ele não pode gerar código binário nativo para execução. Mais detalhes sobre o funcionamento do compilador C VEX podem ser encontrados em [20].

As saídas do codificador PBIW utilizadas neste trabalho são a de depuração e a geração de arquivos de descrição de hardware. As saídas de depuração foram usadas para checagem e teste durante o processo de construção do codificador. As saídas de descrição de hardware são arquivos fonte contendo código VHDL que implementam as memórias de instruções e de padrões (P-Cache), que contém as instruções ρ-VEX codificadas e padrões PBIW. Essas memórias são usadas para a síntese e prototipação para execução e validação em FPGA utilizando o processador ρ-VEX.

Os algoritmos de codificação PBIW otimizados usados neste trabalho codificam as instruções ρ-VEX utilizando duas organizações: a organização com restrições e sem restrições, versões 1.0 e 2.0, respectivamente. Na versão com restrições, os campos de leitura e escrita na instrução ρ-VEX PBIW estão fisicamente separados na instrução. Assim, operandos de leitura não podem ocupar campos de operandos de escrita e vice-versa. Na codificação sem restrições essa separação entre operandos de leitura e escrita não existe: ambos os operandos podem ocupar qualquer campo livre presente na instrução PBIW ρ-VEX.

3.4 Arquitetura e Fluxo de Dados

A arquitetura da infraestrutura para codificação é composta por três conjuntos principais: o contexto de montagem, contexto de codificação e o contexto de otimização.

No contexto de montagem encontra-se o *front-end* do codificador e as estruturas de dados necessárias para a representação em memória das instruções do(s) programa(s) a serem codificadas. Esse contexto possui rotinas de processamento dos dados de entrada, pré-processando e deixando-os prontos para a codificação.

No contexto de codificação encontra-se o algoritmo de codificação assim como as rotinas responsáveis pela manipulação e armazenamento das estruturas de dados necessárias para a representação em memória das instruções e padrões do(s) programa(s) codificados. Considerando a técnica PBIW, o contexto de codificação PBIW possui todas as informações necessárias para a codificação do programa original. No contexto de codificação PBIW estão os algoritmos de codificação (e decodificação, caso seja necessário) e todas as instruções e padrões PBIW do programa codificado.

A possibilidade da inclusão de um algoritmo de decodificação nesse contexto tem como objetivo testar e validar o algoritmo de codificação PBIW durante o desenvolvimento do mesmo. Dessa forma, é possível escrever testes de unidade cujo objetivo é validar tanto a corretude do algoritmo de codificação via código PBIW gerado como na sua decodificação para a representação original.

Por fim, no contexto de otimização são implementados os algoritmos de otimização para a técnica de codificação incluída no contexto de codificação. No caso da codificação PBIW, o contexto de otimização possui rotinas responsáveis pela manipulação e armazenamento das estruturas de dados necessárias para a representação em memória das instruções e padrões PBIW otimizados do(s) programa(s) codificados.

O fluxo de dados e controle do processo de codificação PBIW simplificado pode ser visto na Figura 3.4.

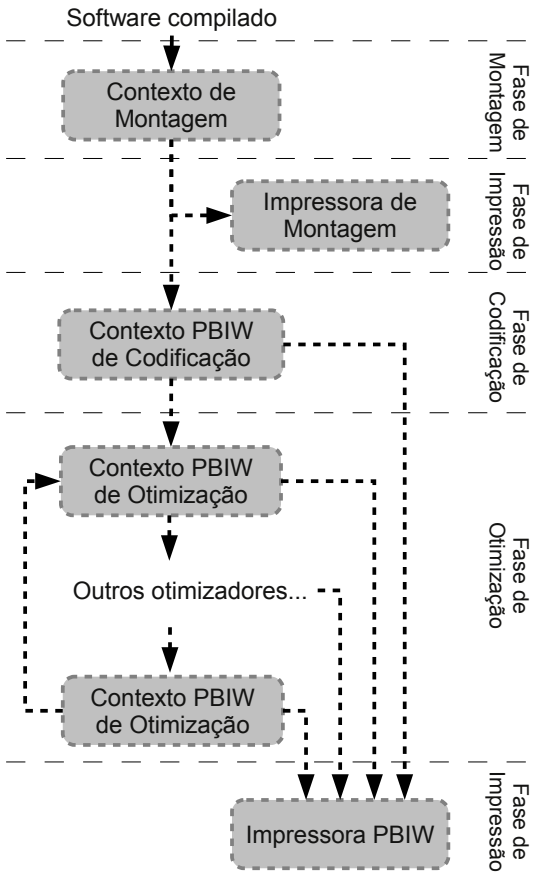


Figura 3.4: Fluxo de dados da infraestrutura de codificação utilizando a técnica PBIW.

A entrada para a infraestrutura de codificação, neste exemplo, é a saída do compilador em linguagem *assembly*. Cada unidade de compilação gerada pelo compilador será processada pelo contexto de montagem e armazenado em uma representação interna presente no mesmo. Esta representação interna é totalmente dependente do montador, tipo de arquitetura e estrutura do arquivo de montagem. Para tornar essa representação interna acessível ao contexto de codificação PBIW, a infraestrutura provê interfaces que devem ser implementadas por essas estruturas de dados. Essas interfaces provêem métodos de acesso aos dados internos sem expor sua implementação, sendo uma camada de abstração entre as diferentes implementações possíveis dos contextos de montagem e os algoritmos de codificação PBIW.

Logo após o processo de codificação, há a possibilidade de se executar otimizações sobre o conjunto de instruções e padrões presentes neste contexto. Cada otimização contém um conjunto independente de instruções e padrões: uma cópia — das instruções, padrões e outros dados — do contexto original ou do contexto de outra otimização. As otimizações que utilizam a saída de um contexto de codificação são aplicadas sobre uma cópia dos conjuntos de instruções e padrões do contexto de codificação, não alterando os conjuntos de instruções e padrões dele. Também é possível executar otimizações em sequência quantas vezes seja necessário.

Depois das etapas de montagem, codificação e/ou otimização, pode-se realizar a impressão dos dados gerados utilizando classes chamadas de impressoras. Cada impressora tem a responsabilidade de retornar dados de saída da aplicação. Por terem acesso às instruções,

padrões e outros dados disponíveis das classes de contexto, as impressoras podem imprimir desde informações de depuração, passando por arquivos de descrição de hardware e até gerar arquivos binários ELF completos.

Um exemplo de uso da infraestrutura de codificação PBIW pode ser visto na Figura 3.5 com as cinco fases do processo: montagem, impressão da montagem (opcional), codificação PBIW, otimização PBIW (opcional) e impressão da codificação ou otimização. Cada uma dessas fases é descrita a seguir:

Fase de montagem: há o processamento do programa que será codificado, convertendo-o para uma representação intermediária de seções, funções, instruções, operações, rótulos e outros elementos necessários para representar o programa de forma estruturada em memória. Caso o programa a ser codificado seja representado em "alto nível" — como um código *assembly* gerado por um compilador — também é necessário, nesta etapa, realizar o cálculo de endereços de desvios e chamadas de funções. Caso o programa a ser codificado seja representado em um nível já montado e link-editado — como uma representação ELF — o cálculo de endereços não é necessário. Entretanto, a representação intermediária do contexto de montagem deve saber quais instruções ou funções são os destinos dos desvios já calculados. Esta fase de montagem pode ser adaptada de um montador ou do *back-end* de um compilador;

Fase de impressão da montagem: na (opcional) fase de impressão da montagem, os dados coletados e armazenados na representação intermediária do contexto de montagem podem ser impressos de diversas maneiras. A saída mais comumente desejada é uma saída de depuração cujos dados impressos podem ser usados para detectar inconsistências no processamento do código de entrada. Outra saída, também comum, é uma representação binária do código de entrada, que pode variar desde uma *string* de bits contendo somente instruções até arquivos ELF;

Fase de codificação PBIW: o programa de entrada presente no contexto de montagem (como uma representação intermediária) será passado ao contexto de codificação PBIW. Isso gerará uma versão codificada do programa internamente ao contexto de codificação PBIW. Essa versão codificada pode seguir um de dois caminhos: ser impressa ou ser otimizada. Ao ser impressa, um processo análogo à impressão de montagem será realizado. Ao ser otimizada, essa versão codificada será repassada a um ou mais otimizadores PBIW que realizarão otimizações no código gerado;

Fase de otimização PBIW: nesta fase, a representação intermediária PBIW gerada no contexto de codificação é copiada para dentro do contexto de otimização. Isso significa que o otimizador trabalhará em um conjunto de dados/objetos completamente independente dos dados/objetos presentes no contexto de codificação. Dessa forma, é possível comparar, caso seja necessário, os resultados da(s) otimização(ões) com o resultado inicial da codificação PBIW;

Fase de impressão PBIW: nesta fase as instruções (e padrões) PBIW gerados nos contextos de codificação e/ou otimização são impressos em um processo análogo à impressão de montagem. A saída comumente desejada é uma saída de depuração, mostrando instruções com seus operandos e os padrões apontados pelas instruções, cada um com suas operações. Assim como na impressão dos dados do contexto de montagem, esta

impressão pode ser usada para detectar inconsistências no processo de codificação e otimização PBIW. Saídas formatadas para execução do programa codificado também podem ser criadas. No Capítulo 4 serão exibidos exemplos de saídas para execução implementadas para o processador ρ-VEX.

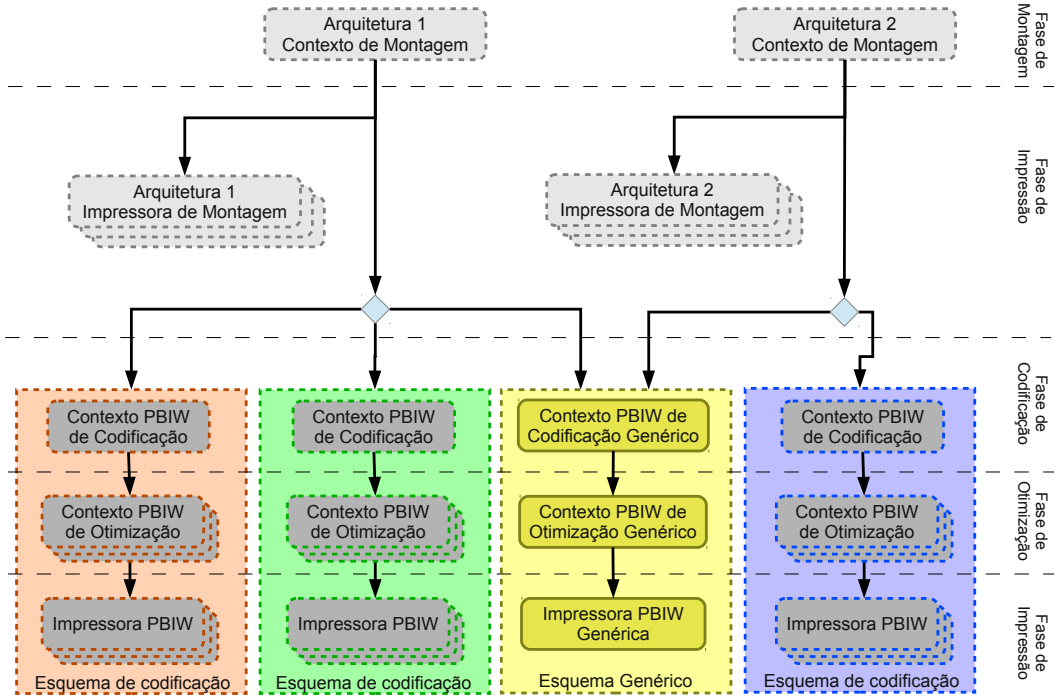


Figura 3.5: Exemplos de fluxos de dados para diversas arquiteturas e codificadores PBIW.

3.5 Contexto de Montagem

O contexto de montagem é definido por um conjunto relativamente pequeno de interfaces. Cada interface define um conjunto de métodos que devem ser implementados para que haja interoperabilidade de dados entre ele e o contexto de codificação. O conjunto de interfaces disponíveis pode ser visto na Figura 3.6.

O contexto de montagem agrega seções que podem agregar funções e/ou rótulos (*labels*). As funções agregam rótulos e instruções, que por sua vez agregam operações, que por sua vez agregam operandos. As impressoras, por sua vez, conseguem acessar toda a hierarquia descrita para impressão de alguma informação de saída, seja ela de depuração ou algum arquivo para execução.

3.5.1 Interfaces

O contexto de montagem possui um conjunto de interfaces que exportam funcionalidades necessárias ao fluxo de codificação. Informações detalhadas sobre cada interface (implementadas considerando a codificação PBIW) podem ser encontradas no Apêndice A.

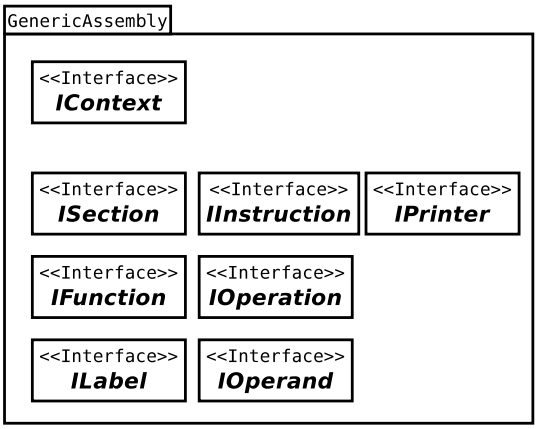


Figura 3.6: Interfaces de montagem da infraestrutura de codificação.

A interface `IContext` define métodos para acessar os dados contidos dentro dos contextos de montagem. Um contexto de montagem é o *front-end* do algoritmo de codificação. Contextos de montagem podem ser implementados como montadores completos ou parciais, podem ser uma representação em memória da saída do back-end de um compilador, podem ser extratores de informações de arquivos ELF, entre outros.

A interface `ISection` foi projetada para definir uma estrutura para seções (texto, dados e bss) de arquivos de montagem ou binários. Pode conter funções, *labels* e alocação de memória usada ao longo do programa.

A interface `IFunction` define uma estrutura para funções presentes em arquivos de montagem ou binários. Funções são conjuntos de instruções de uma ISA que processam dados de entrada, podendo retornar outros dados como saída.

A interface `ILabel` define uma estrutura para rótulos presentes em arquivos de montagem ou binários. Um rótulo é uma cadeia de caracteres usada como auxílio na marcação de um endereço de memória interno em um programa. Rótulos podem ser usados para endereçar funções e variáveis em seções de arquivos de montagem ou binários.

A interface `IInstruction` define uma estrutura para trabalhar com instruções de uma ISA em arquivos de montagem ou binários. Uma instrução pode ser formada por uma ou mais operações. O endereço de uma instrução também pode ser marcado por um rótulo, como no caso de início de blocos básicos e funções.

A interface `IOperation` define uma estrutura para trabalhar com operações pertencentes a instruções de uma ISA em arquivos de montagem ou binários. Uma operação é composta por um opcode e operandos.

A interface `IOperand` define uma estrutura para trabalhar com operandos pertencentes a operações de uma instrução. Um operando é um valor passado ou recebido de uma operação ao ser executada. Comumente é definido como um número e pode representar semanticamente um registrador (leitura ou escrita), um valor imediato ou um rótulo.

A interface `IPrinter` define métodos para imprimir as informações de montagem de um código objeto processado pelo contexto de montagem. Esta interface modela o padrão de projeto *Visitor* [30]. Este padrão de projeto permite separar um algoritmo que será

executado sobre uma estrutura de objetos sem precisar alterar os objetos em si. Assim, pode-se definir diversos tipos de impressoras (depuração, arquivos executáveis, entidades VHSIC hardware description language (VHDL), etc.) que gerarão uma saída para os dados de montagem.

3.5.2 Classes Utilitárias

Algumas classes utilitárias foram criadas para auxiliar no desenvolvimento dos contextos de montagem em duas vertentes: a vertente de funcionalidade de montagem e de checagem de tipos por parte do compilador. A vertente de funcionalidade de montagem contém as classes auxiliares de verificação de conflitos de dados entre instruções e de escopos de rótulos. A vertente de checagem de tipos por parte do compilador possui a classe auxiliar que verifica a herança entre classes. Essas classes utilitárias auxiliares são descritas em detalhes a seguir.

DependencyChains

A classe `DependencyChains` possui a responsabilidade de verificação de conflitos de dados entre instruções e operações. A necessidade de tal verificação ocorre em situações em que o algoritmo de codificação "divide" a instrução original em mais de uma instrução codificada. Esse é o caso da utilização da codificação PBIW sobre conjuntos de instruções de arquiteturas VLIW. Essa divisão, porém, pode acabar expondo conflitos de dependência verdadeira ou anti-dependência de dados, também conhecidos como conflitos *read after write* (RAW) e *write after read* (WAR), respectivamente. Dessa forma, há a necessidade de identificar tais conflitos para conseguir decidir qual é a melhor posição de divisão, caso seja possível, na instrução VLIW original.

Caso não seja possível a divisão da instrução — devido a um conflito de dados entre a primeira e a última operação de uma instrução VLIW, por exemplo — a classe `DependencyChains` possui um algoritmo de ordenação topológica que reordena as operações de modo a eliminar o conflito de dados ou, caso não seja possível sua eliminação, permitir a divisão da instrução reduzindo a distância entre as operações conflituosas.

LabelScope

Classe não instanciável que contém uma enumeração com definições de dois tipos de rótulos (*labels*): global e local. Rótulos globais são rótulos que definem funções, endereços de memória específicos de variáveis e outros locais que podem ser globalmente acessados. Rótulos locais são rótulos que servem para delimitar blocos básicos internos a funções.

DerivedFrom

A classe `DerivedFrom` verifica, em tempo de compilação, se uma classe filha herda (ou é derivada) de uma classe pai específica. Esta checagem é feita para prover ao usuário da infraestrutura de codificação uma validação do uso correto das interfaces presentes.

Essa checagem é necessária pois a linguagem C++ não disponibiliza funcionalidades-padrão presentes na sua sintaxe e/ou bibliotecas para cobrir este caso de uso. Um exemplo de uso pode ser visto no construtor da classe paramétrica DependencyChains, presente na Figura 3.7. Se alguma das classes declaradas nos parâmetros do template não for um tipo derivado dos tipos informados, compiladores exibirão uma mensagem de erro amigável identificando exatamente em que trecho de código a verificação foi violada.

```
template <class TInstruction, class TOperation, class TPrinter>
class DependencyChains
{
public:
  DependencyChains()
  {
    DerivedFrom<TInstruction,GenericAssembly::Interfaces::IInstruction>();
    DerivedFrom<TOperation,GenericAssembly::Interfaces::IOperation>();
    DerivedFrom<TPrinter,GenericAssembly::Interfaces::IPrinter<TInstruction, TOperation> >();
  }
(...)
```

Figura 3.7: Exemplo de uso da classe `DerivedFrom` .

Um exemplo de mensagem de erro gerada pelo compilador GNU G++ pode ser visto na Figura 3.8.

```
src/rVex/PBIWPartial/PartialPBIW.cpp:135:70: error: invalid static_cast from type
'GenericAssembly::Interfaces::IInstruction* const' to type 'rVex::Instruction*'
```

Figura 3.8: Exemplo de mensagem de erro emitida em tempo de compilação pelo compilador GNU C++ ao usar a classe `DerivedFrom` .

3.6 Contexto de Codificação

O contexto de codificação é definido por um conjunto relativamente pequeno de interfaces. Cada interface define um conjunto de métodos que devem ser implementados para que haja abstração entre os tipos de dados providos pelo contexto de montagem e as instruções codificadas.

No caso da utilização da técnica de codificação PBIW, as interfaces desse contexto devem ser implementadas de forma a prover a conexão necessária entre os dados providos pelo contexto de montagem e as instruções codificadas e padrões PBIW. Utilizando princípios de abstração de tipos concretos, é possível manter um algoritmo genérico de codificação PBIW compatível com diversos tipos de contexto de montagem e configurações PBIW de entrada que vierem a ser desenvolvidos no futuro. O conjunto de interfaces para o contexto de codificação PBIW pode ser visto na Figura 3.9.

3.6.1 Interfaces

Nesta seção será realizada uma descrição de todas as interfaces do contexto de codificação PBIW. Informações detalhadas sobre cada interface podem ser encontradas no Apêndice B.

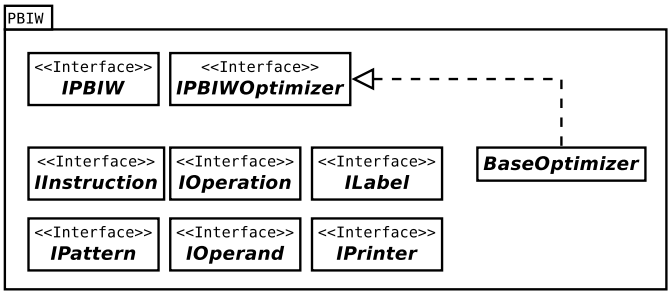


Figura 3.9: Interfaces de codificação PBIW da infraestrutura de codificação.

- `IPBIW` define métodos comuns para todos os contextos de codificação PBIW. Um contexto de codificação PBIW contém o algoritmo de codificação e estruturas de dados e componentes PBIW responsáveis para representar em memória o programa codificado.
- `IPBIWOptimizer` define métodos comuns a todos os contextos de otimização PBIW. Um contexto de otimização PBIW recebe conjuntos de instruções e padrões e realiza otimizações em uma cópia desses conjuntos armazenada internamente.
- `IPBIWFactory` implementa o padrão de projeto *Factory*. Este padrão de projeto define uma interface comum para criar objetos sem expor a lógica de instanciação ao usuário da *Factory* e referencia os objetos recentemente criados através de uma interface comum.
- `ILabel` define métodos para acessar os dados de um rótulo da infraestrutura de codificação PBIW. Um rótulo PBIW contém informações sobre o endereço absoluto e a instrução destino que ele referencia, o escopo (global ou local) e a *string* que o define. Um rótulo com escopo global serve para referenciar nomes de funções e outros endereços globais ao arquivo de entrada e um rótulo com escopo local referencia blocos básicos internos a uma função.
- `IPBIWInstruction` define métodos para acessar os dados da instrução PBIW. Uma instrução PBIW contém informações sobre os operandos que possui, um ponteiro para o padrão que utiliza e os bits de cancelamento (caso a codificação adote bits de cancelamento).
- `IPBIWPattern` define métodos para atuar sobre operações e índices que apontam aos campos de operandos das instruções PBIW.
- `IOperand` define métodos para atuar sobre dados de operandos presentes na instrução PBIW. Um operando contém um valor, um tipo (registrador de leitura ou escrita, valor imediato, etc.) e um índice (sua posição (posição > 0) indexada no vetor de operandos da instrução PBIW).
- `IOperation` define métodos para manipular cada operação presente no padrão PBIW. Uma operação contém um opcode e um vetor de índices para cada operando presente na instrução PBIW que usa esse padrão.
- `IPBIWPrinter` (assim como a `IPrinter`) define métodos comuns para imprimir as informações de um código presente em um contexto de codificação ou otimização PBIW.

BaseOptimizer

A classe abstrata `BaseOptimizer` foi desenvolvida para auxiliar no desenvolvimento de otimizadores para a codificação PBIW. Ela implementa alguns recursos comuns a todos os contextos de otimização. Um desses recursos é a clonagem dos conjuntos de instruções e padrões PBIW juntamente com a correção dos ponteiros que definem desvios presentes no grafo de fluxo.

3.7 Considerações Finais

Este capítulo apresentou os principais conceitos da técnica de codificação PBIW assim como apresentou a infraestrutura de software para codificação de instruções. No projeto da infraestrutura, considerou-se parâmetros como extensibilidade e portabilidade da técnica assim que o núcleo do algoritmo de codificação pode ser utilizado para atuar com diferentes otimizações e contextos de entrada e saída. Esta infraestrutura de software já está implementada e utilizada na instanciação do algoritmo PBIW em duas ISAs distintas: VEX e SPARC. A utilização da infraestrutura de codificação PBIW sobre o processador ρ-VEX e detalhes sobre o projeto das instruções codificadas e padrões serão apresentados no Capítulo 4.

Capítulo 4

PBIW Aplicada ao Processador ρ-VEX

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 seja 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 energia 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.

Neste capítulo apresenta-se a implementação da codificação PBIW, através da infraestrutura de codificação apresentada no Capítulo 3, na ISA VEX do processador embarcado ρ-VEX. Este capítulo aborda desde especificidades da arquitetura do processador embarcado ρ-VEX, passando pelas decisões do projeto de codificação indo até a implementação de duas abordagens de codificação (e decodificação) PBIW para o processador ρ-VEX. O capítulo está organizado da seguinte forma: nas Seções 4.1 e 4.2 são detalhadas a microarquitetura e implementação da ISA VEX no processador ρ-VEX; na Seção 4.3 detalha-se o projeto das codificações PBIW desenvolvidas para o processador ρ-VEX; as considerações finais do capítulo são apresentadas na Seção 4.4.

4.1 Processador *Softcore* ρ-VEX

O ρ-VEX [21] é um processador VLIW *softcore* multiciclo configurável descrito em VHDL baseado na ISA VEX [20]. A ISA VEX é inspirada na ISA da família de processadores HP/ST Lx [20] (o mesmo usado para os processadores ST200 [20]). O processador ρ-VEX implementa uma arquitetura Harvard e possui uma micro-arquitetura VLIW minimalista: não

possui unidade de gerenciamento de memória nem unidade para cálculo de números de ponto-flutuante, o endereçamento na memória é realizado com referências absolutas, não há implementação de caches, mecanismos de predição de desvios e não há bancos de registradores distribuídos. Apesar dessas limitações, o processador ρ-VEX possui diversas características, como instruções longas em memória e projeto *softcore* embarcado, que o torna adequado para adotar uma técnica de codificação de instruções. A Figura 4.1 apresenta a via de dados do processador ρ-VEX.

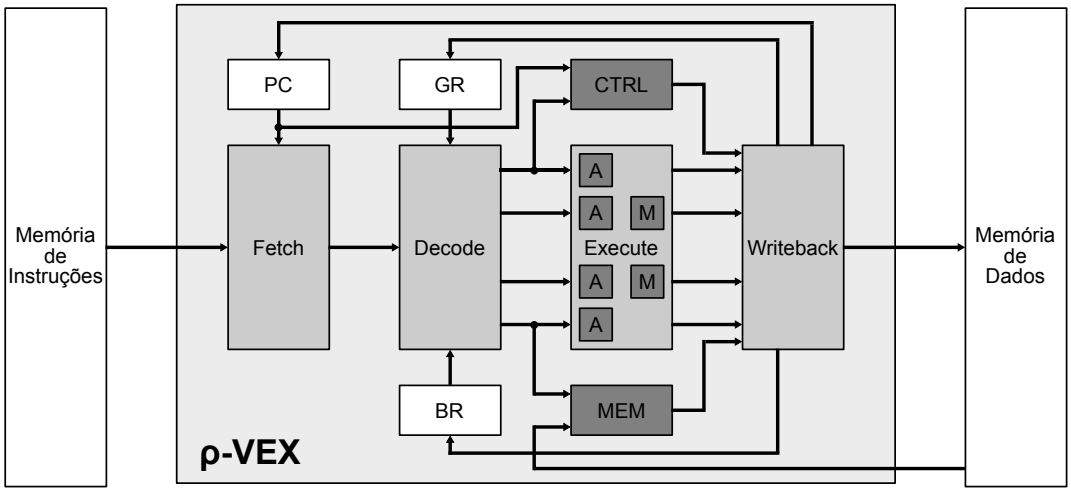


Figura 4.1: Via de dados do processador ρ-VEX.

Depois dos estágios de busca e decodificação, as sílabas de uma instrução são despachadas para unidades de execução individuais. Cada instrução contém quatro 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.

A arquitetura possui dois bancos de registradores: os registradores gerais (*General Registers*, ou GR) e registradores de *branch* (*Branch Registers*, ou BR). Os registradores gerais são um conjunto de 32 registradores de 32 bits, indexados de 0 a 31, usados para computação geral dentro da via de dados. Os registradores de *branch* são um conjunto de 8 registradores de 1 bit, indexados de 0 a 7, usados para armazenar resultados booleanos de operações de comparação e lidos em operações de desvio condicional. Ao longo deste capítulo, as siglas GR e BR serão usadas para se referir a um registrador geral ou registrador de *branch*, respectivamente.

O conjunto de ferramentas de geração de código (Figura 4.2) e simulação para ρ-VEX contém um compilador C, bibliotecas ANSI C, ferramentas para facilitar o processo de *profiling* de código, um simulador em *software* (disponibilizado pela HP [31]) e o montador ρ-ASM. Apesar da existência de ferramentas para geração e simulação de código VEX, tais ferramentas não suportam a compilação e simulação de uma variedade de programas para o processador ρ-VEX.

Especificamente, não há ferramentas de link-edição, forçando, assim, a remoção de 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.

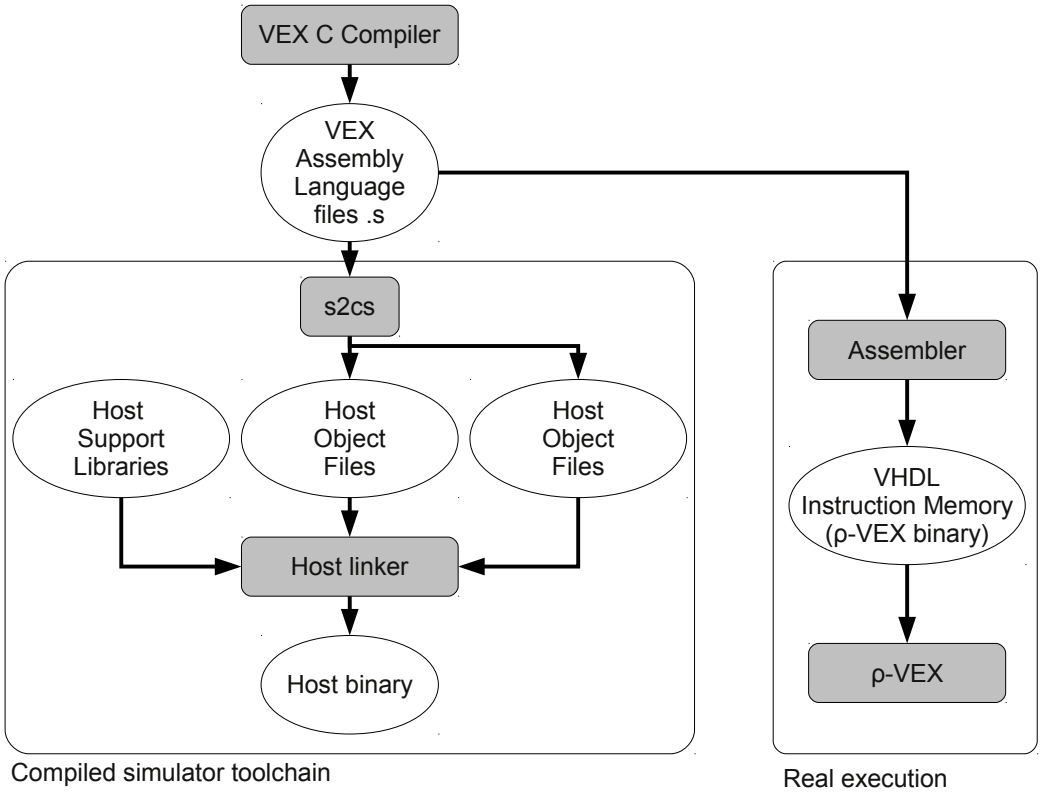


Figura 4.2: Fluxo de ferramentas para geração de código VEX e ρ-VEX.

4.2 Conjunto de Instruções VEX

Como VEX é uma arquitetura flexível, pode-se distinguir dois tipos de restrições: o conjunto de regras que todas as implementações devem obedecer (como a ISA base, conectividade de registradores, coerência de memória, estado da arquitetura) e o conjunto de regras de uma instância VEX específica (como a largura do despacho de instruções, número de *clusters*, conjunto de unidades funcionais, latências e instruções customizadas).

Um *cluster* é uma instância de um processador que implementa a ISA VEX. Dentro de uma Central Processing Unit (CPU) podem existir um ou mais *clusters*, cada um com regras de instância específicas.

Para compreender a notação da linguagem *assembly* de uma operação VEX, considere o exemplo presente na Figura 4.3.

O primeiro símbolo, "c0", denota o *cluster* no qual a operação será executada (neste caso, *cluster* zero). O(s) operando(s) de destino é(são) dado(s) por uma lista de símbolos à esquerda do símbolo "=", enquanto que o(s) operando(s) de origem é(são) listado(s) à direita do símbolo "=". Neste caso, um único operando de destino é o registrador BR de índice 3 do *cluster* zero. O identificador do *cluster* é opcional nos símbolos dos operandos quando representam registradores. Entretanto, o identificador do *cluster* sempre está presente para reforçar a identificação do *cluster* cujo registrador operando será usado na execução. Os símbolos ";;" indicam o fim de um grupo de operações a serem emitidas em um mesmo ciclo.

Instruções que possuem várias operações simplesmente listam cada operação em linhas separadas, com um duplo sinal de ponto-e-vírgula como separador de instruções. Na Figura 4.3 observa-se um trecho de código em linguagem de montagem VEX no qual há duas instruções: a primeira com duas operações paralelas e a segunda com três.

```
c0 cmpne $b0.3 = $r0.4, $r0.5 # instr 0, op 0
c0 sub $r0.16 = $r0.6, 3      # instr 0, op 1
;; ## fim da primeira instrução

c0 shl $r0.13 = $r0.13, 3  # instr 1, op 0
c0 shr $r0.15 = $r0.15, 9  # instr 1, op 1
c0 ldw.d $r0.14 = 0[$r0.4] # instr 1, op 2
;; ## fim da segunda instrução
```

Figura 4.3: Exemplo de instruções VEX.

A arquitetura VEX possui um modelo de execução para as instruções e operações:

- Deve-se emitir múltiplas operações por instrução em um único ciclo de *clock*;
- Operações são executadas como uma única unidade atômica;
- Instruções são executadas estritamente na ordem apresentada do programa mas, internamente à execução de uma instrução, todos os operandos são lidos antes de qualquer resultado ser escrito (isso permite, por exemplo, trocar valores de um par de registradores em uma única instrução);
- Instruções não podem conter restrições de execução sequecial entre suas operações;

O comportamento da execução é o mesmo de uma máquina em-ordem: cada instrução executa até completar antes de iniciar a execução da próxima. Assim, todas as sílabas de uma instrução começam a executar juntas e realizam o *commit* de seus resultados juntas. O *commit* de resultados inclui desde modificar o estado dos registradores, atualizar memória até gerar exceções.

O conjunto de instruções VEX padrão não possui suporte para trabalhar com números em ponto-flutuante. Assim, todas suas 73 operações (excluindo `NOP` - *no operation*) trabalham com números inteiros. Dentre o conjunto de 73 operações originais, existem 42 operações lógico-aritméticas, 11 operações de multiplicação e divisão de inteiros, 11 operações de controle e nove operações de memória.

Para a implementação da ISA VEX no processador ρ-VEX o conjunto original foi estendido com mais duas operações adicionais: `STOP` e `LONG_IMM`. A primeira informa para o processador ρ-VEX parar de buscar instruções da memória de instruções. A segunda é usada quando operandos *long immediate* são usados. Para definir pelo menos um *opcode* único a cada operação, foram necessários sete bits para codificá-los.

Somente as operações de transferência de dados inter-*clusters*(`SEND` e `RECV`) não foram implementadas no processador ρ-VEX, apesar de seus *opcodes* permanecerem reservados.

Tipo de imediato	Tamanho	*Immediate switch*
Sem imediato	N/A	00
Short immediate	9 bit	01
Branch offset immediate	12 bit	10
Long immediate	32 bit	11

Tabela 4.1: Tipos de imediatos.

Isso deve-se ao processador ρ-VEX não ser capaz de trocar dados entre *clusters* por suportar, no estágio de desenvolvimento atual, somente um *cluster*.

Uma instrução ρ-VEX possui 16 bytes (128 bits) de comprimento e seu formato é apresentado na Figura 4.4. Ela é formada por quatro sílabas (operações), cada uma com quatro bytes (32 bits) que podem ser organizadas da seguinte forma:

- Sílaba zero com operação lógico-aritmética ou de controle;
- Sílabas um e dois com operação lógico-aritmética ou de multiplicação/divisão;
- Sílaba três com operação lógico-aritmética ou de memória;

As posições das sílabas na instrução ρ-VEX são equivalentes a quantidade de unidades funcionais disponíveis para uso e suas respectivas posições no estágio de execução.

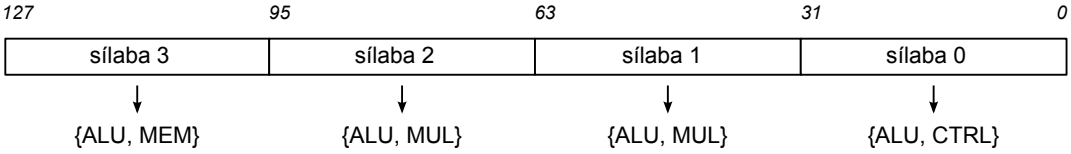


Figura 4.4: Formato de instrução do processador ρ-VEX.

A Figura 4.5 mostra os possíveis formatos das operações. Nos sete bits mais significativos encontra-se o campo dos opcodes. Os próximos dois bits definem qual o tipo de imediato usado na operação; sua codificação pode ser vista na Tabela 4.1. Nos bits 22 a 2 estão presentes os operandos usados pela operação que podem ter organizações específicas conforme o valor presente no campo *immediate switch*.

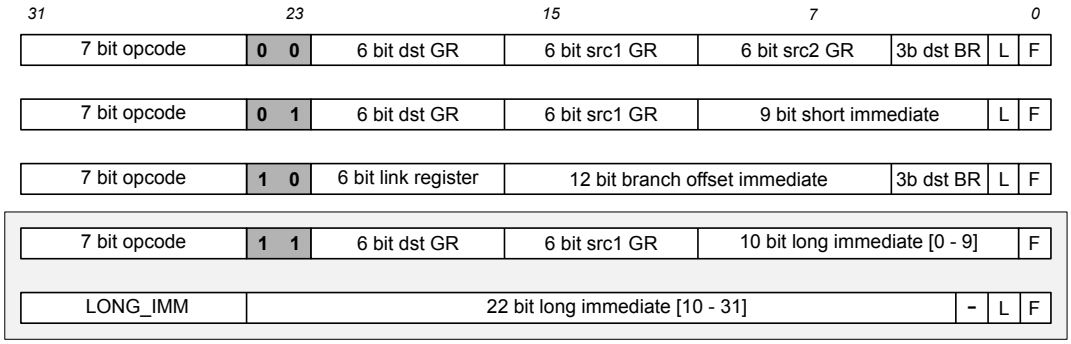


Figura 4.5: Formatos de operações implementadas no processador ρ-VEX.

Os últimos dois bits de cada operação estão definidos como bits L e F, respectivamente. O bit L (*last*) informa se a operação é a última sílaba da instrução. O bit F (*first*) informa se a operação é a primeira sílaba da instrução. Sua existência é justificada como informação necessária para implementação de possíveis compressores/codificadores de instruções.

Os últimos dois formatos destacados dos demais pertencem ao formato de *long immediate*: um operando de quatro bytes (32 bits) utilizado por uma operação. Este formato utiliza duas sílabas para conseguir representar uma instrução com um operando *long immediate*. Este formato não estava disponível na versão ρ-VEX utilizada neste trabalho, apenas planejado como futura melhoria.

4.3 Projeto da Técnica PBIW Sobre o Processador ρ-VEX

A codificação PBIW foi aplicada sobre o processador ρ-VEX em duas versões (ou esquemas) de codificação: uma versão denominada "Codificação com restrições" e outra denominada "Codificação sem restrições". O primeiro esquema foi definido como a versão 1.0 e o segundo como versão 2.0. O motivo da escolha e definição de versões deve-se ao período em que foram projetados e implementados. A versão 1.0 (com restrições) foi a primeira a ser projetada e implementada utilizando decisões de projeto. A versão 2.0 (sem restrições) foi uma evolução natural da versão 1.0 otimizada para tratar os piores casos da versão 1.0. O fluxo completo de geração de código (com e sem codificação PBIW) para o processador ρ-VEX pode ser visto na Figura 4.6.

Ambas as versões dos algoritmos de codificação possuem complexidade $O(n^2)$ por realizarem uma busca linear no conjunto de padrões PBIW a cada novo padrão gerado. Essa decisão de projeto teve como objetivo simplificar a implementação dos algoritmos de codificação. A troca por uma busca utilizando estruturas de dados mais elaboradas (estruturas em árvores ou tabelas *hash*) pode ser realizada em um trabalho futuro para diminuir a complexidade dos algoritmos e acelerar o processo de codificação.

Na Figura 4.7 é possível observar o fluxo de dados e controle do codificador PBIW aplicado ao processador ρ-VEX. Este fluxo de dados usa a infra-estrutura de codificação introduzida no Capítulo 3. O fluxo de dados da codificação PBIW para o processador ρ-VEX segue o padrão discutido nas Figuras 3.4 e 3.5. Nota-se, também, que o contexto de montagem ρ-VEX é compartilhado entre os dois algoritmos de codificação PBIW específicos para o processador ρ-VEX juntamente com o algoritmo PBIW genérico.

Internamente aos algoritmos PBIW específicos é usada uma otimização em comum: a otimização de junção de padrões. Cada rotina de codificação (versões 1.0 e 2.0) utiliza um contexto de codificação e impressoras específicos para cada versão. Isso é devido à estrutura interna das instruções e padrões serem diferentes entre os esquemas de codificação 1.0 e 2.0. Essa diferença exige uma adequação entre as impressoras e algoritmos de codificação para manter sua corretude na geração de código. Também é usada uma impressora de dados estatísticos comum entre as versões 1.0 e 2.0. Essa impressora imprime informações referentes ao nível de compressão estático, número de reúso médio de cada padrão PBIW gerado e comparações de tamanho com o código ρ-VEX sem codificação.

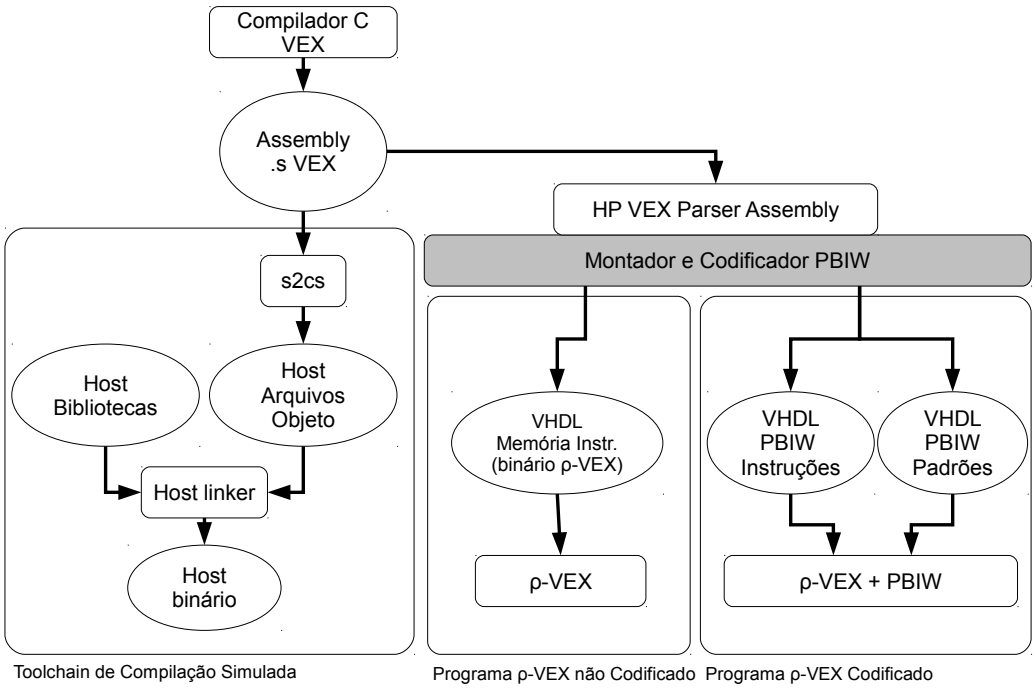


Figura 4.6: Fluxo de geração de código com e sem a codificação PBIW para ρ-VEX.

4.3.1 Codificação com Restrições (Versão 1.0 da Codificação PBIW)

Conforme apresentado no Capítulo 3, o projeto de instruções codificadas baseadas na técnica PBIW deve considerar alguns parâmetros para decidir sobre o formato e tamanho das instruções codificadas e padrões gerados. A partir de experimentos e da análise de códigos gerados sobre programas que implementavam algoritmos clássicos de ordenação, busca em árvores, as seguintes considerações foram tomadas para o formato da instrução codificada com restrições para o processador ρ-VEX:

- sete campos dedicados aos operandos de leitura;
- quatro campos dedicados aos valores imediatos e operandos de escrita;
- quatro campos de um bit para cancelamento de operações (implica na utilização da otimização de junção de padrões);
- um campo de sete bits que serve como ponteiro para o padrão corresponde na tabela de padrões.

A instrução PBIW codificada pode ser vista na Figura 4.8. A instrução ρ-VEX original possui 128 bits de comprimento enquanto que a nova instrução codificada 64 bits.

A instrução codificada possui o campo de anulação com quatro bits (um bit para cada operação presente no padrão), sete campos indexados (1-7) para os operandos de leitura, quatro campos para operandos de escrita. Os campos de operandos possuem cinco bits cada, exceto o último campo, que possui três bits. A decisão de dimensionar a instrução

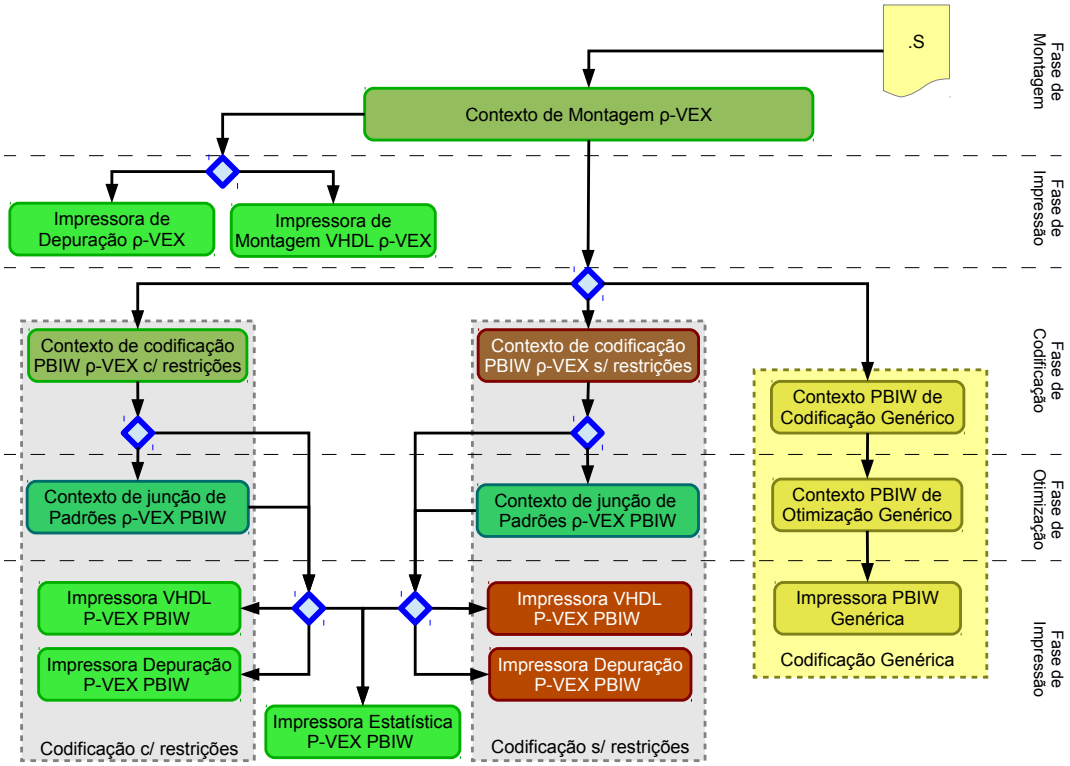


Figura 4.7: Exemplos de fluxo de dados para codificação PBIW na arquitetura ρ-VEX. Losângulos representam pontos de decisão de fluxo.

codificada com 64 bits foi motivada para manter o alinhamento da instrução em bytes. Todos os campos de cinco bits armazenam, individualmente, valores de 0 a 31. O último campo, com três bits, armazena valores de 0 a 7.

O operando no campo de índice zero é oculto não existe fisicamente na instrução: sua utilidade é ser usado na codificação para prover a constante de valor zero quando necessário. Os últimos três campos, índices 9-11, são usados para armazenar valores imediatos. O imediato de nove bits é armazenado nos operandos de índice 9-10, tendo seu último bit extendido para ocupar os 10 bits de ambos os operandos. O imediato de 12 bits é armazenado nos operandos de índice 9-11, também tendo seu último bit extendido para ocupar os 13 bits dos três campos de operandos.

Nessa versão, os registradores gerais de leitura das operações ρ-VEX serão armazenados nos campos 1-7, enquanto que os registradores gerais de escrita nos campos de 8-11. Esta divisão nos locais de armazenamento levando em consideração operandos de leitura e escrita deu o nome a esta versão de "Codificação com restrições".

Caso a instrução possua alguma instrução com registrador BR de leitura, o mesmo deverá ser armazenado no campo de índice 1. Registradores BR de escrita são armazenados nos mesmos campos de leitura, isto é, com índices 1-7 devido à limitação do tamanho do ponteiro existente na operação do padrão: somente três bits de endereçamento.

O padrão PBIW (Figura 4.9) possui 96 bits divididos em quatro sílabas de 24 bits cada. Uma sílaba contém um campo de opcode de sete bits, um campo de dois bits para seletor

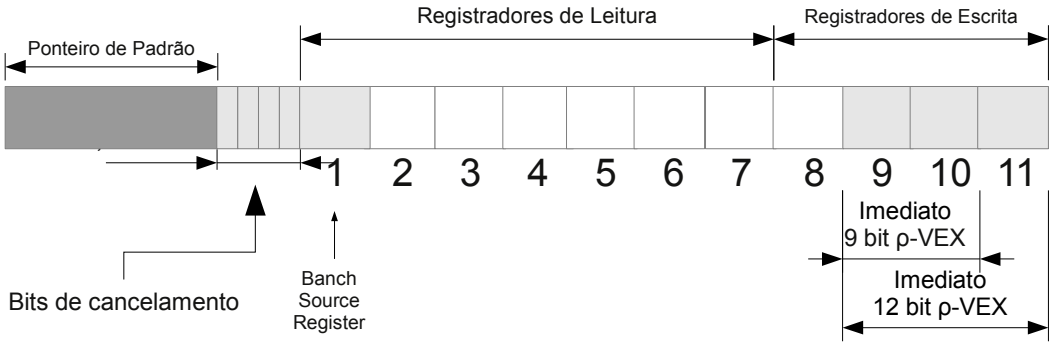


Figura 4.8: Instrução codificada PBIW ρ-VEX versão 1.0

de imediato, três campos de quatro bits e um campo de três bits usados como índices para os operandos na instrução PBIW.

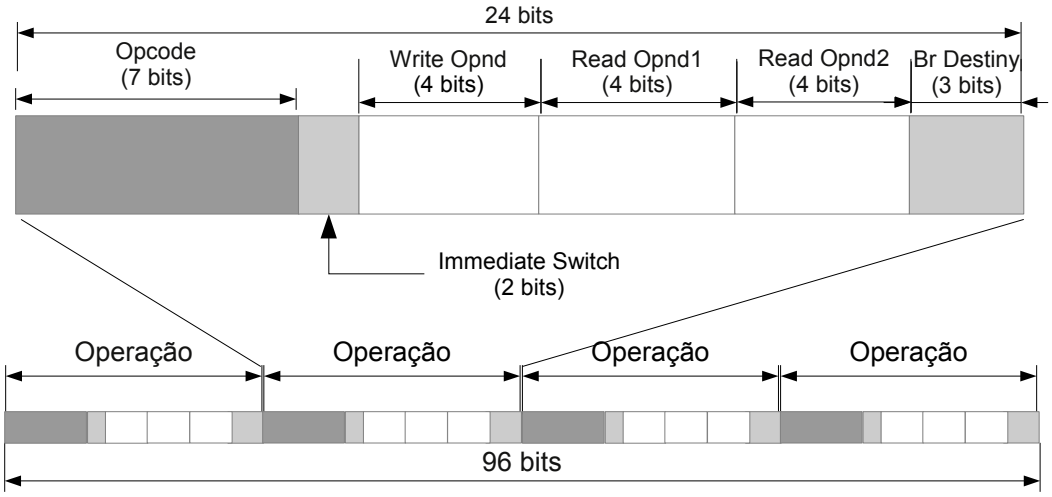


Figura 4.9: Padrão PBIW ρ-VEX versão 1.0.

Para as operações ADDCG, DIVS, SLCT e SLCTF, dos sete bits presentes para o opcode, são usados somente os quatro bits mais significativos. Os três bits menos significativos são usados para armazenar o número do registrador BR de leitura. Esses três bits são ignorados no processo de decodificação para simplificar o projeto do decodificador em hardware visto que o registrador BR de leitura é armazenado, por definição, no 1^o campo (índice 1) da instrução PBIW.

Se uma instrução ρ-VEX possui quatro operações que usem um registrador BR de leitura e a instrução PBIW só possui uma posição para este operando, a instrução ρ-VEX original será dividida em quatro instruções PBIW no processo de codificação. Esta limitação foi corrigida na versão 2.0 da codificação, como será visto na Seção 4.3.2.

O campo de seletor de imediato do operando é usado pelo processador ρ-VEX para decidir quais os tipos do operandos presentes na operação depois da decodificação PBIW. Dos três campos de quatro bits da operação, o primeiro aponta para o operando de escrita e os restantes para operandos de leitura presentes na instrução PBIW. O último campo, de três bits, aponta para o registrador BR de destino.

A visão geral do circuito decodificador PBIW é apresentada na Figura 4.10. Em virtude da complexidade e da limitação de espaço, apresenta-se aqui o esquema de decodificação de uma operação.

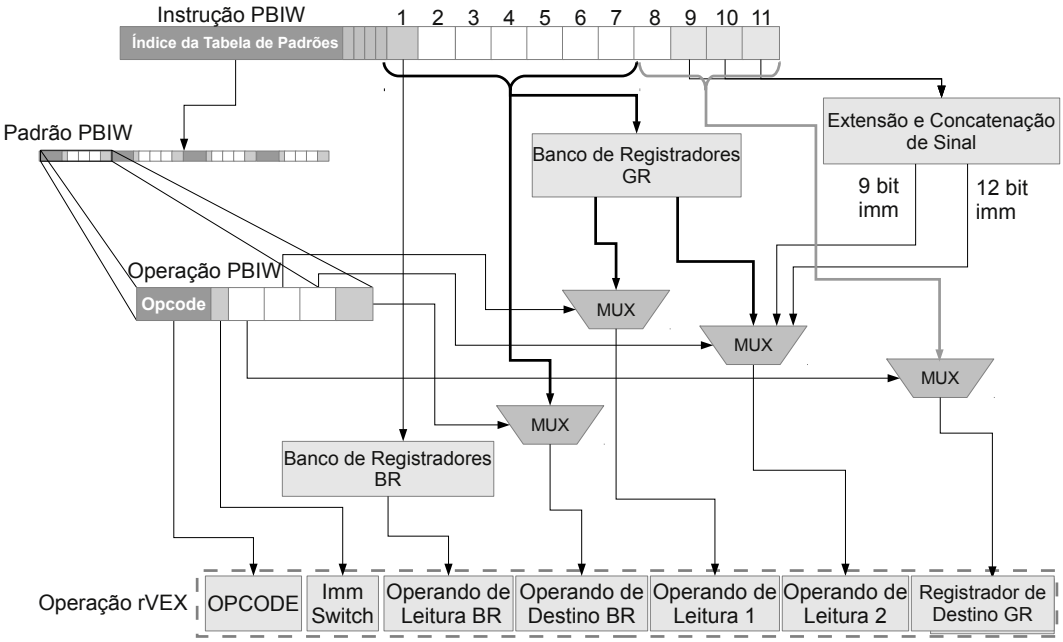


Figura 4.10: Visão geral do decodificador PBIW ρ-VEX versão 1.0.

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. Concomitante com a leitura dos registradores, o padrão é recuperado da tabela de padrões utilizando o índice armazenado na instrução PBIW. Sendo o tamanho do campo do Índice da Tabela de Padrões de sete bits, a implementação da tabela de padrões (P-*cache*) tem um tamanho de 1,5 Kbytes (128 linhas × 96 bits = 12.288 kbits = 1,5 Kbytes) e está implementada como uma tabela de acesso direto junto à 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 Figura 4.11 exemplifica a codificação PBIW de versão 1.0. Inicialmente tem-se duas instruções ρ-VEX, a primeira contém as operações ADD e ADDI e a segunda SUB e SUBI. Ao final do processo de codificação PBIW e junção de padrões são obtidas duas instruções PBIW e um padrão compartilhado entre as instruções. É importante notar, também, os bits de cancelamento paras as instruções codificadas: os bits definidos com o valor "1" simbolizam que as operaçãos presentes no padrão nas respectivas posições serão executadas, enquanto que bits em "0" simbolizam que a operação não será executada.

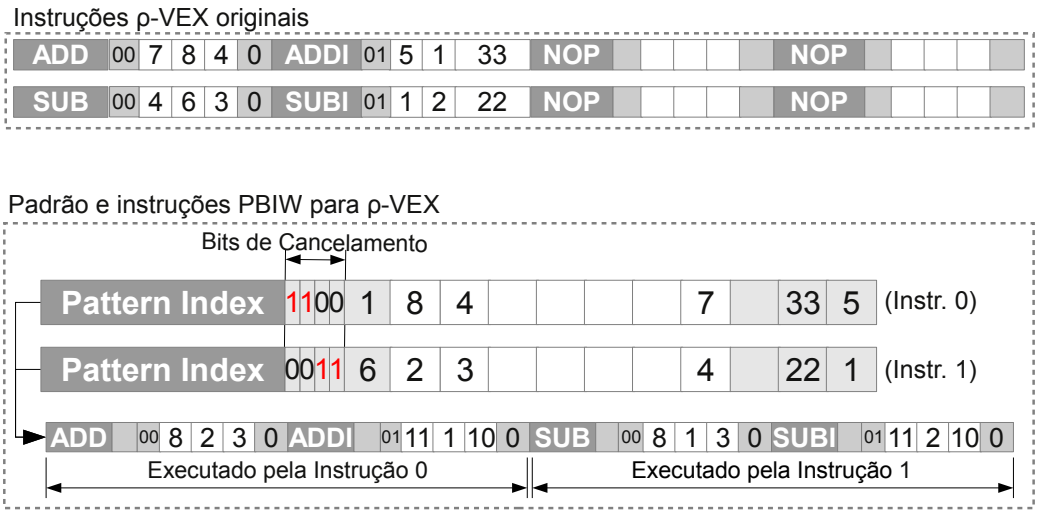


Figura 4.11: Instruções originais, codificadas e padrão para PBIW ρ-VEX versão 1.0.

4.3.2 Codificação sem Restrições (Versão 2.0 da Codificação PBIW)

A versão 2.0 da codificação PBIW para o processador ρ-VEX possui muitas semelhanças a versão 1.0 e seguem as decisões de projeto conforme apresentadas no Capítulo 3. A principal mudança, no entanto, refere-se à eliminação de campos específicos para leitura e escrita de operandos. Após análise dos resultados da codificação na versao 1.0, notou-se a ocorrência de muitas situações em que a instrução PBIW ρ-VEX codificada não suportava todos os operandos da instrução ρ-VEX original e, com isso, mais de uma instrução codificada e um novo padrão associado eram gerados. A instrução ρ-VEX codificada, Figura 4.12, continua tendo 64 bits dos quais sete bits são para ponteiro de padrão, quatro bits para cancelamento e onze campos de operandos, dos quais os dez iniciais possuem cinco bits e o último possui três bits.

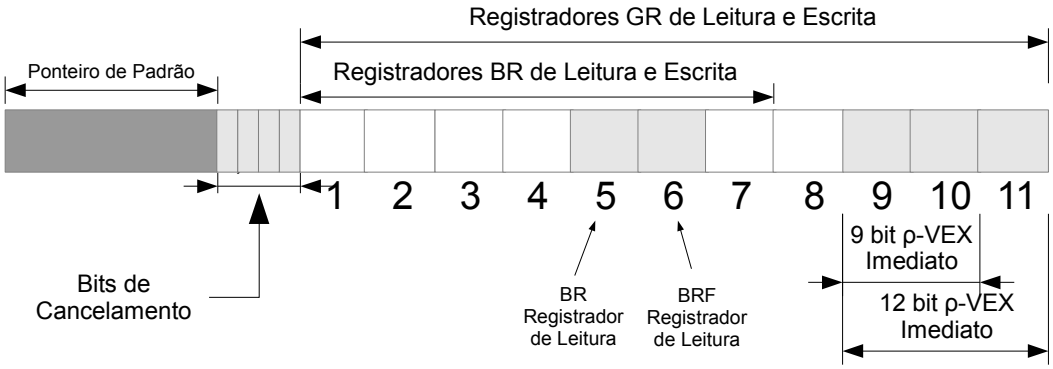


Figura 4.12: Instrução codificada PBIW ρ-VEX versão 2.0.

O operando no campo de índice zero continua existindo: ele é oculto e portanto não existe fisicamente na instrução, fornecendo sempre o valor zero. Os últimos três campos, de índices de 9-11, também continuam sendo usados para armazenar valores imediatos. O imediato de nove bits é armazenado nos operandos de índice 9-10, assim como o imediato

de 12 bits é armazenado nos operandos de índice 9-11. Ambos têm seu último bit estendido para ocupar os dois ou três campos de operandos, respectivamente.

Na versão 2.0, os registradores GR de leitura e escrita das operações ρ-VEX serão armazenados nos campos de 1 a 11, sem separação. A possibilidade de armazenamento de qualquer tipo de operando em qualquer campo da instrução deu o nome a esta versão de "Codificação sem restrições". Outra alteração foi a possibilidade de armazenamento dos registradores BR de leitura e escrita nos campos de índice 1-7 da instrução.

O padrão PBIW versão 2.0 (Figura 4.13) também não foi alterado: possui 96 bits divididos em quatro sílabas de 24 bits cada. Uma sílaba contém um campo de opcode de sete bits, um campo de dois bits para seletor de imediato, três campos de quatro bits e um campo de três bits usados como índices para os operandos na instrução PBIW.

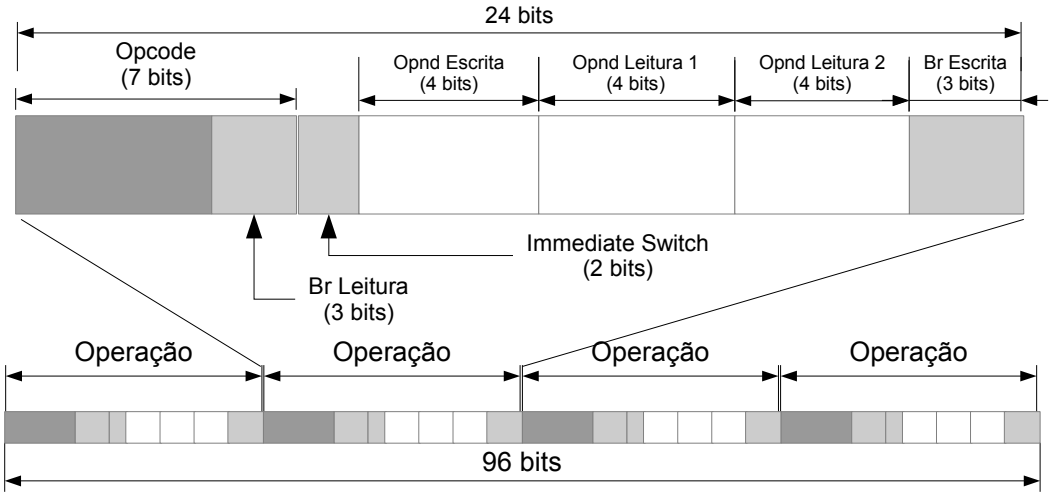


Figura 4.13: Padrão PBIW ρ-VEX versão 2.0.

Para as operações ADDCG, DIVS, SLCT, SLCTF, BR e BRF também houveram mudanças: os sete bits presentes para o opcode são divididos em dois campos. O primeiro campo, com os quatro bits mais significativos, contém o opcode. O segundo campo, com os três bits menos significativos, é usado para armazenar o ponteiro para o registrador BR de leitura armazenado na instrução PBIW nos campos de índices 0-7. Aqui é necessário ressaltar: as operações ADDCG, DIVS, SLCT e SLCTF possuem opcodes de quatro bits que se encaixam perfeitamente na divisão anterior, mas as operações BR e BRF não. Os últimos três bits menos significativos de seus opcodes (valores cinco e seis, respectivamente) fazem parte do mesmo, como pode ser visto na Tabela 4.2. Neste caso, os campos com índices 5-6 da instrução PBIW são definidos para armazenar os registradores BR lidos pelas operações BR e BRF.

Dessa forma, elimina-se a limitação de uma operação que usa registrador BR de leitura por instrução. Caso haja uma instrução ρ-VEX com quatro operações que usem um registrador BR de leitura ela não será mais dividida, no processo de codificação, em quatro instruções PBIW por falta de espaço.

A visão geral do circuito decodificador PBIW é apresentada na Figura 4.14. Em virtude da complexidade e da limitação de espaço, apresenta-se aqui o esquema de decodificação para

ADDCG	1111---
DIVS	1110---
SLCT	0111---
SLCTF	0111---
BR	0100101
BRF	0100110

Tabela 4.2: Operações que utilizam registradores BR de leitura e respectivos opcodes.

uma operação apenas. Observa-se, assim, que o fluxo de decodificação seguem os seguintes passos:

1. O padrão é recuperado da tabela de padrões utilizando o índice armazenado na instrução PBIW;
2. Cada operação do padrão seleciona os seus respectivos operandos na instrução PBIW;
3. Cada operando (caso sejam registradores) recuperam seus valores nos bancos de registradores;
4. Ao final, *opcodes*, seletores de imediato, operandos e imediatos foram decodificados para formar cada operação original, prontas para serem executadas.

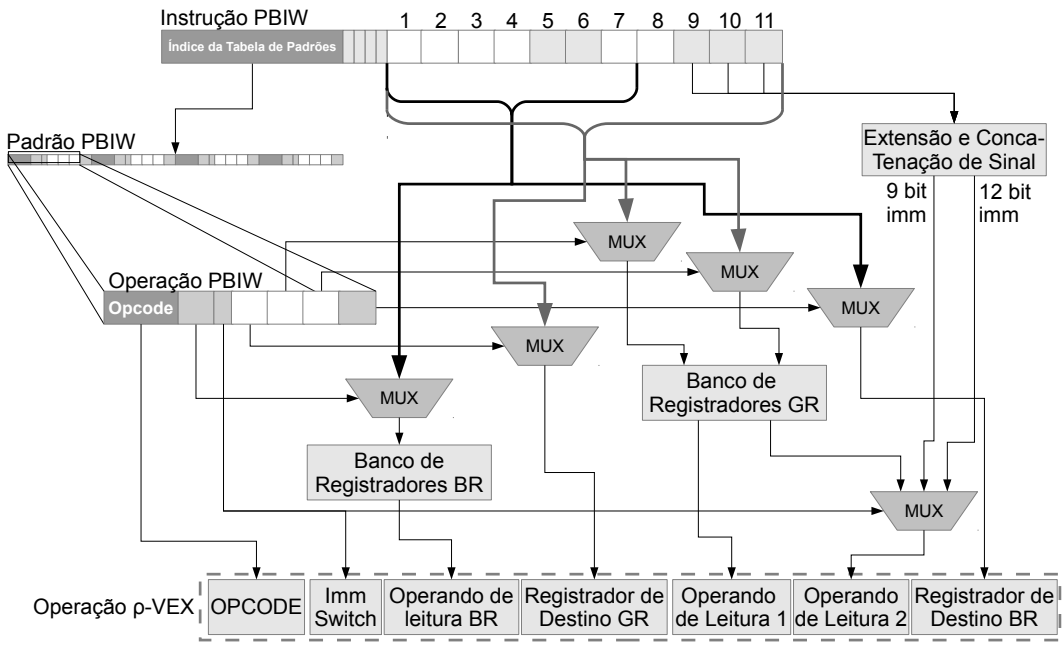


Figura 4.14: Visão geral do decodificador PBIW ρ-VEX versão 2.0.

Sendo o tamanho do campo do Índice da Tabela de Padrões de sete bits, a implementação da tabela de padrões (P-*cache*) também tem o mesmo tamanho da cache da codificação 1.0: de 1,5 KBytes (128 linhas × 96 bits = 12.288 kbits= 1,5 KBytes). A tabela de padrões está implementada como uma tabela de acesso direto junto à unidade *Decode*.

Importante observar que a codificação 2.0 é menos restritiva que a 1.0, tornando seu decodificador compatível com programas codificados na versão 1.0, que tem codificação mais restritiva. Assim, é possível executar tanto programas codificados na versão 1.0 quanto na versão 2.0 no decodificador 2.0, mantendo uma compatibilidade binária retroativa.

4.4 Considerações Finais

Este capítulo apresentou a aplicação da técnica de codificação PBIW sobre o processador ρ-VEX. No projeto da codificação PBIW ρ-VEX, duas versões de codificação PBIW foram desenvolvidas. Além disso, decodificadores de instruções, para cada uma dessas versões, foram projetados, implementados e prototipados junto ao projeto existente do processador ρ-VEX. Pode-se observar também que o projeto de codificações PBIW é extensível e flexível para possibilitar a otimização dos programas codificados enquanto mantém-se uma compatibilidade binária retroativa. Experimentações, resultados e análises de ambas as técnicas de codificação PBIW serão apresentadas no Capítulo 5.

Capítulo 5

Experimentos e Resultados

Neste capítulo serão apresentados experimentos e resultados relativos à implementação da codificação PBIW no processador embarcado ρ-VEX. Discutir-se-á implicações referentes à escolha das versões de codificação PBIW de acordo com os resultados dos experimentos. Os resultados obtidos consideram métricas como taxa de compressão, potência consumida, frequência de operação, área ocupada, simulação da taxa de *miss* na cache de padrões e de instruções via Dinero e tamanho do código gerado. Este capítulo está organizado da seguinte forma: na Seção 5.1 será introduzido o método de coleta dos dados; nas Seções 5.2, 5.3 e 5.3.3 serão apresentados os resultados estáticos e dinâmicos para ambas as versões de codificação e suas implicações; na Seção 5.4 serão apresentadas considerações finais do capítulo.

5.1 Introdução

Para avaliar o impacto de um novo esquema de codificação de instruções deve-se levar em consideração diferentes indicadores de avaliação. Alguns desses se referem à codificação do software em si e o impacto de sua execução no sistema. Destacam-se os seguintes indicadores referentes à codificação: indicadores estáticos (tamanho original versus codificado dos programas escolhidos para avaliação) e dinâmicos (taxa de *misses* na memória cache e quantidade de bytes transferidos da memória principal) de cada programa. Também existem indicadores relativos ao processo de decodificação no hardware que devem ser considerados, destacando-se: a área ocupada pelo novo decodificador no hardware, o número de elementos lógicos usados, o consumo de energia e a frequência de operação.

A Tabela 5.1 enumera os programas e as instâncias de entrada adotadas nos experimentos deste capítulo. Todos os treze programas avaliados foram escritos em linguagem C. Cinco programas fazem parte do Trimaran Compiler Suite (*Simple benchmarks*) [32]. 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 [20].

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 5.1: Programas usados nos experimentos.

Não foram usados programas maiores de benchmarks conhecidos (como SPEC, MediaBench e Mibench, por exemplo) pois não há ferramentas de link-edição disponíveis para o processador ρ-VEX. Isso obriga a remoção de todas as chamadas a funções presentes em bibliotecas externas, como manipulação dinâmica de memória, impressão e leitura de textos em tela e arquivos, entre outras.

Há, também, outro fator limitante: devido ao ρ-VEX suportar somente imediatos de nove bits em operações da Unidade de Lógica e Aritmética (ULA) e memória, não é possível realizar o carregamento e armazenamento de dados em endereços de memória muito distantes entre si. Isso se torna um problema quando um programa possui muitas funções e um trabalho intenso de pilha com chamadas de funções com muitas variáveis locais, gerando um *stack frame* muito extenso. Também é impossível a essas funções endereçarem e acessarem variáveis globais do programa, que se encontram em uma posição muito distante na pilha com relação ao *stack frame* da função em execução. Dessa forma, todas as funções em um programa devem ser expandidas *inline* internamente na função *main* de forma a preservar o endereçamento global de variáveis, cuidando-se para que o tamanho da pilha de variáveis locais não ultrapasse a capacidade de deslocamento do imediato de nove bits. Devido ao compilador C VEX possuir código fechado, as limitações descritas não podem ser contornadas devido a impossibilidade de configuração ou adaptação (via código fonte) do compilador. Isso impede que ele emita código compatível com as limitações de endereçamento presentes no processador ρ-VEX .

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 *softcore* ρ-VEX antes da síntese do projeto. As memórias de instrução e dados foram prototipadas sobre os blocos de memória embarcadas denominados M4K [33]. Os programas (instruções e padrões, no caso dos programas codificados) foram inseridos nas memórias embarcadas utilizando o arquivo "Memory Initialization File (MIF)" que contém definições do tamanho da memória a ser prototipada e a lista de endereços com a instrução correspondente.

5.2 Resultados Estáticos

Para experimentação e obtenção de resultados estáticos os programas foram codificados utilizando a versão com restrição (1.0) e a versão sem restrição (2.0) da codificação PBIW desenvolvida para ρ-VEX com e sem otimização de junção de padrões.

5.2.1 Versão 1.0

As Figuras 5.1 e 5.2 mostram o tamanho do código estático sem e com a otimização de junções de padrões, respectivamente. No eixo Y encontra-se o tamanho dos programas não-codificados (originais) e codificados com PBIW versão 1.0. As barras dos programas codificados (segunda barra de cada programa) também mostram a porcentagem do tamanho das instruções e padrões em relação ao tamanho total do programa.

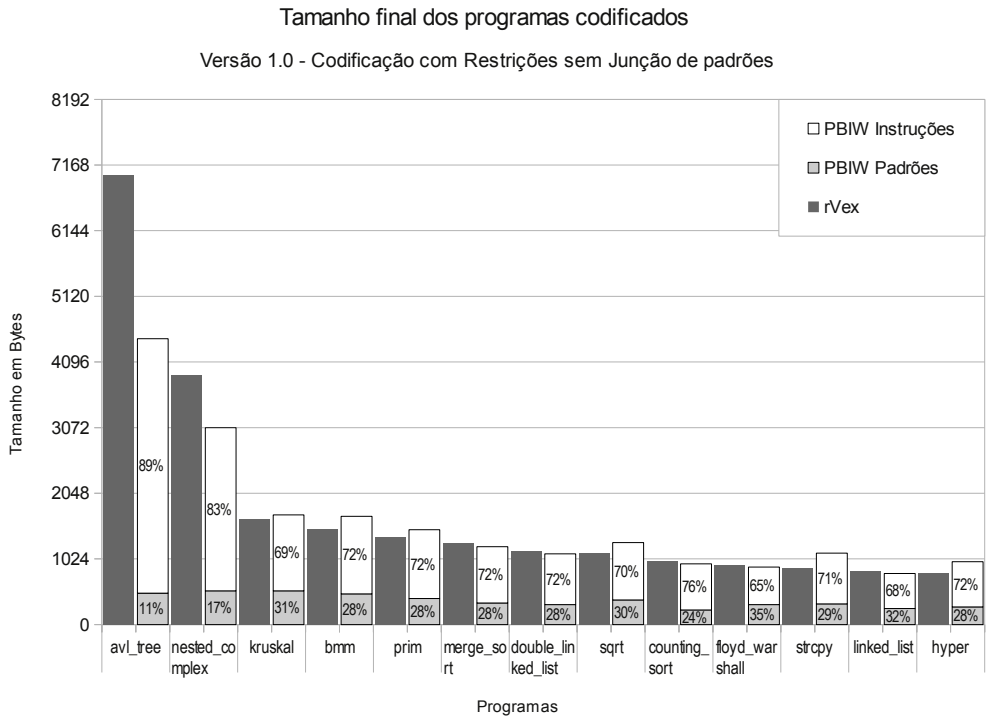


Figura 5.1: Resultados estáticos dos programas PBIW ρ-VEX versão 1.0 sem otimização de junção de padrões.

Os resultados mostram que a adoção do PBIW provê a redução do tamanho de código para sete dos treze (53,8%) programas analisados, quando a otimização de junção de padrões não é aplicada. Quando a otimização de junção de padrões é aplicada, a redução do tamanho de código ocorreu em dez dos treze (76,9%) programas analizados.

Ao adotar a técnica PBIW, a taxa de compressão segue a definição da Equação 2.1, apresentada no Capítulo 2, mas considera que o tamanho do código comprimido é dado por:

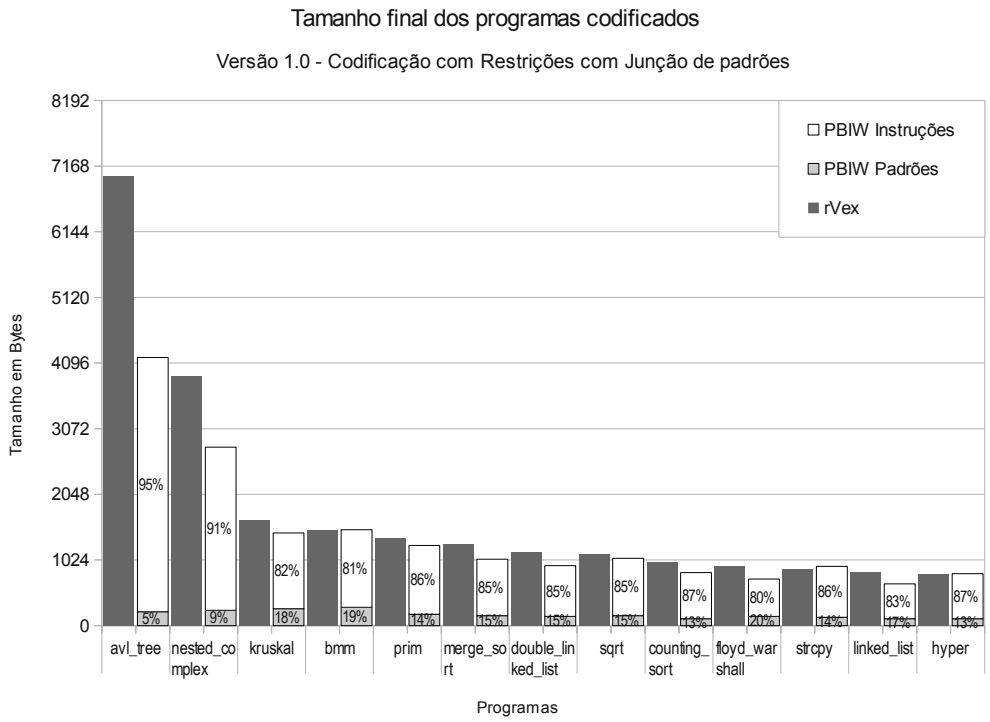


Figura 5.2: Resultados estáticos dos programas PBIW ρ-VEX versão 1.0 com otimização de junção de padrões.

$$\begin{aligned} \text{Tamanho Comprimido} &= (\text{Número instruções codificadas} \times \text{Tamanho instrução codificada}) \\ &+ (\text{Número padrões codificados} \times \text{Tamanho padrão codificado}) \end{aligned}$$

A taxa de compressão variou entre 63,64% e 126,82% para os programas sem a otimização de junção de padrões, e a taxa de compressão média foi de 100,94%. A taxa de compressão variou entre 59,70% e 105,00% para os programas com a otimização de junção de padrões, e a taxa de compressão média foi de 85,82%.

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 pois a sobrejeção entre o conjunto de intruções e o conjunto de padrões aumenta. Essa premissa também é confirmada nas Figuras 5.1 e 5.2 pois a diferença entre os tamanhos dos programas originais e codificados decresce da esquerda para a direita. Alguns programas codificados obtiveram um tamanho maior devido a geração desproporcional de instruções PBIW. Este comportamento é devido ao formato da instrução codificada PBIW que força a separação das sílabas presentes na instrução ρ-VEX original em até outras quatro instruções (por exemplo, no caso de ocorrência de operações `DIVS` na instrução original). Esse comportamento foi melhorado significativamente na versão 2.0 da codificação e poderá ser conferido na seção 5.2.2.

A taxa de reúso TR dos padrões mostrada na Equação (5.1) indica quantas instruções, em média, de um programa utilizam um mesmo padrão. Dentre os programas avaliados, a taxa de reúso nos programas codificados sem a otimização de junção de padrões oscila entre

2,8 (programa floyd_warshall) e 12,1 (programa avl_tree), com uma taxa média de reúso de padrões de 12,1. Já a taxa de reúso nos programas codificados com a otimização de junção de padrões oscila entre 6,1 (programa floyd_warshall) e 27,6 (programa avl_tree), com uma taxa média de reúso de padrões de 10,2.

$$TR = \frac{\text{número de instruções PBIW}}{\text{número de padrões PBIW}} \tag{5.1}$$

A média de padrões gerados sem e com a otimização de junção de padrões foi de respectivamente 31,15 e 14,38. A junção de padrões reduziu em aproximadamente 42% o número de padrões nas aplicações codificadas. Como cada padrão possui 12 bytes, é possível notar uma redução significativa no tamanho final de cada programa, como visto na Figura 5.3.

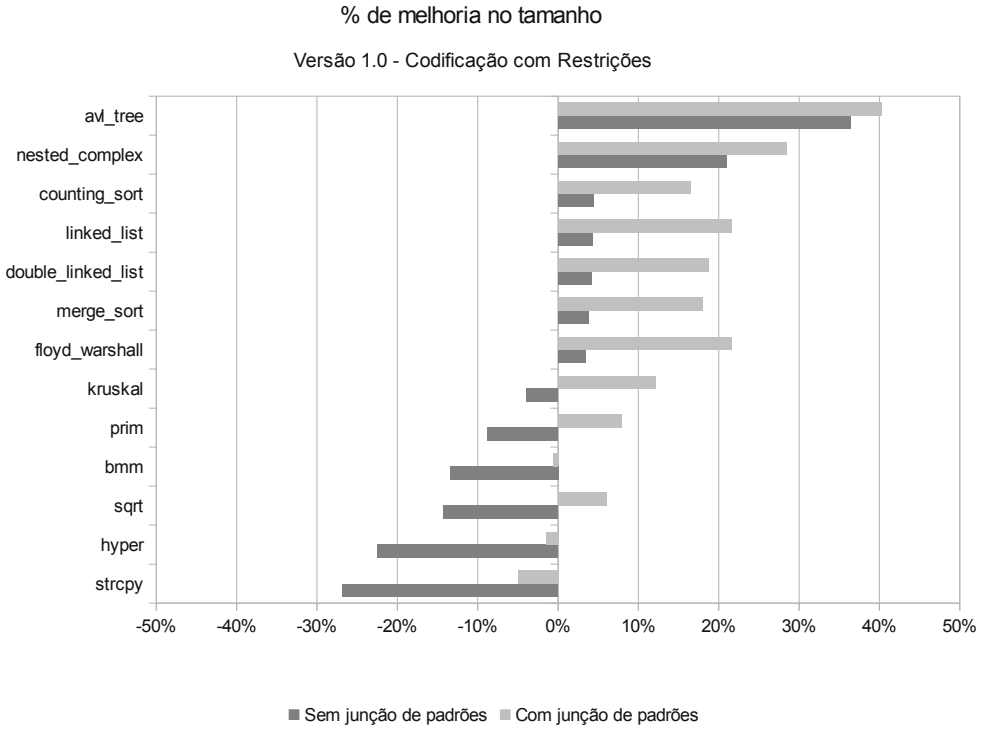


Figura 5.3: Porcentagem de melhoria no tamanho final dos programas PBIW ρ-VEX versão 1.0.

Na versão 1.0 da codificação PBIW (sem a otimização de junção de padrões), tem-se um total de 405 padrões somando-se a quantidade de padrões gerados para cada um dos treze programas codificados. Com a otimização de junção de padrões, os mesmos treze programas geram 187 padrões. Uma redução de 218 padrões ou 46%.

Entretanto, é possível reduzir ainda mais esse número: aplicando-se novamente a otimização de junção de padrões nestes 187 padrões, reduz-se a quantidade para 161 padrões únicos para todos os treze programas codificados. Ao final desse processo, é possível armazenar — caso fosse necessário — todos os treze programas utilizando 17820 bytes em memória, contra 23360 bytes dos programas não codificados. Uma redução de 23,72% no tamanho final.

Na Figura 5.4 é possível notar o efeito da otimização de junção de padrões: há uma redução na quantidade média de padrões que podem ser unidos — isto é, padrões que contém

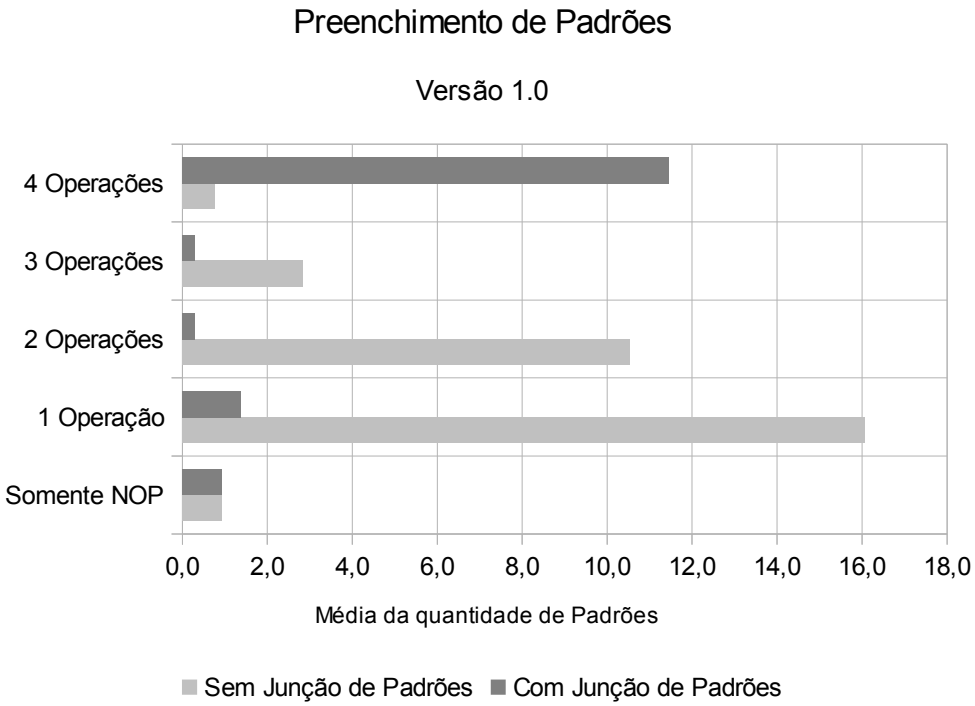


Figura 5.4: Nível de preenchimento dos padrões dos programas PBIW ρ-VEX versão 1.0.

pelo menos 1 operação `NOP` — e um aumento de padrões completos — que possuem 4 operações diferentes de `NOP` — que não podem ser mais unidos a nenhum outro.

5.2.2 Versão 2.0

As Figuras 5.5 e 5.6 mostram o tamanho do código estático sem e com a otimização de junções de padrões, respectivamente. No eixo Y encontra-se o tamanho dos programas não-codificados (originais) e codificados com PBIW versão 2.0. As barras dos programas codificados (segunda barra de cada programa) também mostram a porcentagem do tamanho das instruções e padrões em relação ao tamanho total do programa.

Os resultados mostram que a adoção de PBIW provê a redução do tamanho de código para onze dos treze programas analisados, quando a otimização de junção de padrões não é aplicada. Quando a otimização de junção de padrões é aplicada, a redução do tamanho de código ocorreu em todos os treze programas analizados.

A taxa de compressão variou entre 60,62% e 106,47% para os programas sem a otimização de junção de padrões, e a taxa de compressão média foi de 92,05%. A taxa de compressão variou entre 56,51% e 94,35% para os programas com a otimização de junção de padrões, e a taxa de compressão média foi de 76,87%.

Dentre os programas avaliados, a taxa de reúso (Equação 5.1) nos programas codificados sem a otimização de junção de padrões oscila entre 2,1 (programa floyd_warshall) e 9,6 (programa avl_tree), com uma taxa média de reúso de padrões de 3,3. Já a taxa de reúso

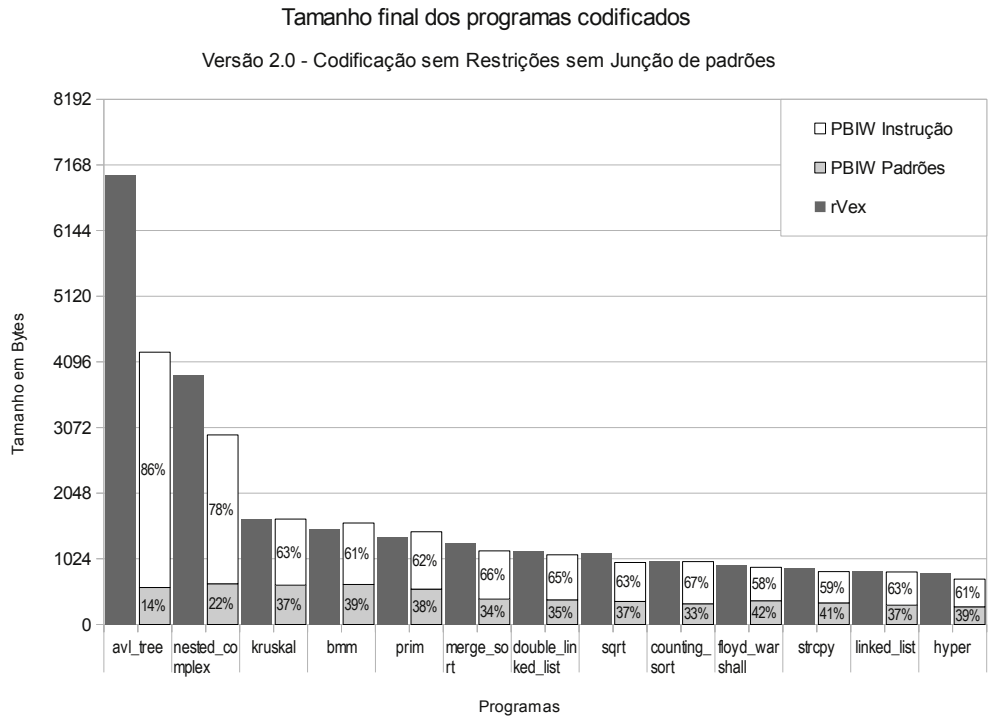


Figura 5.5: Resultados estáticos dos programas PBIW ρ-VEX versão 2.0 sem otimização de junção de padrões.

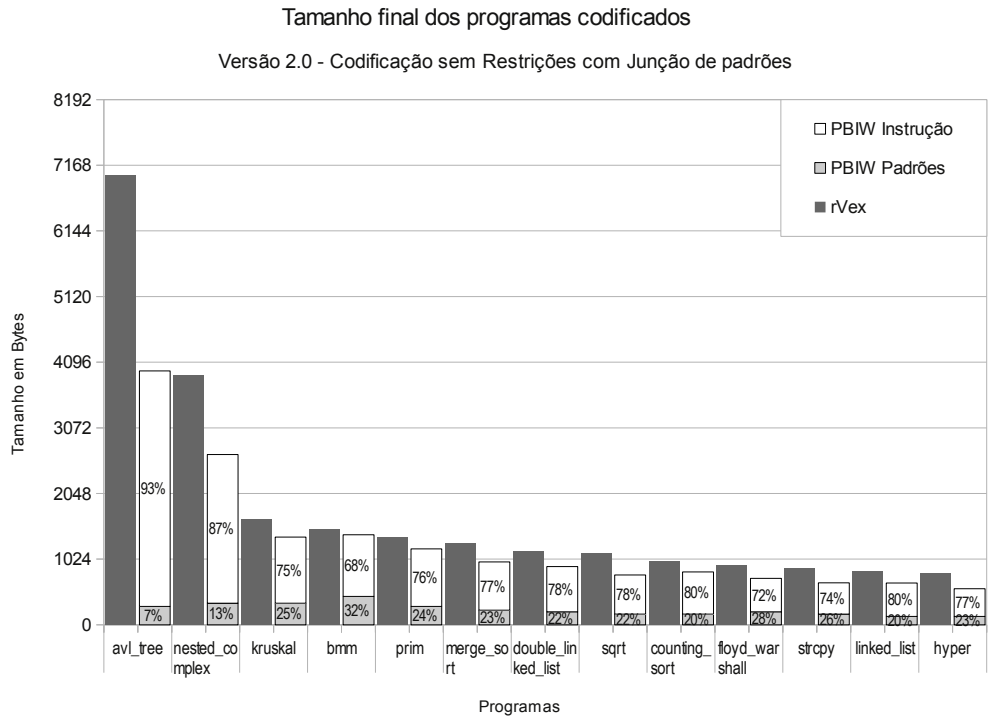


Figura 5.6: Resultados estáticos dos programas PBIW ρ-VEX versão 2.0 com otimização de junção de padrões.

nos programas codificados com a otimização de junção de padrões oscila entre 3,2 (programa bmm) e 19,1 (programa avl_tree), com uma taxa média de reúso de padrões de 6,3.

Nota-se uma melhoria de aproximadamente 9% na taxa de compressão entre as versões 1.0 e 2.0. Essa melhoria é devido a maior disponibilidade de espaços nas instruções PBIW, evitando a separação de instruções originais em diversas instruções PBIW. A menor quantidade de instruções PBIW geradas pode ser vista na Figura 5.7.

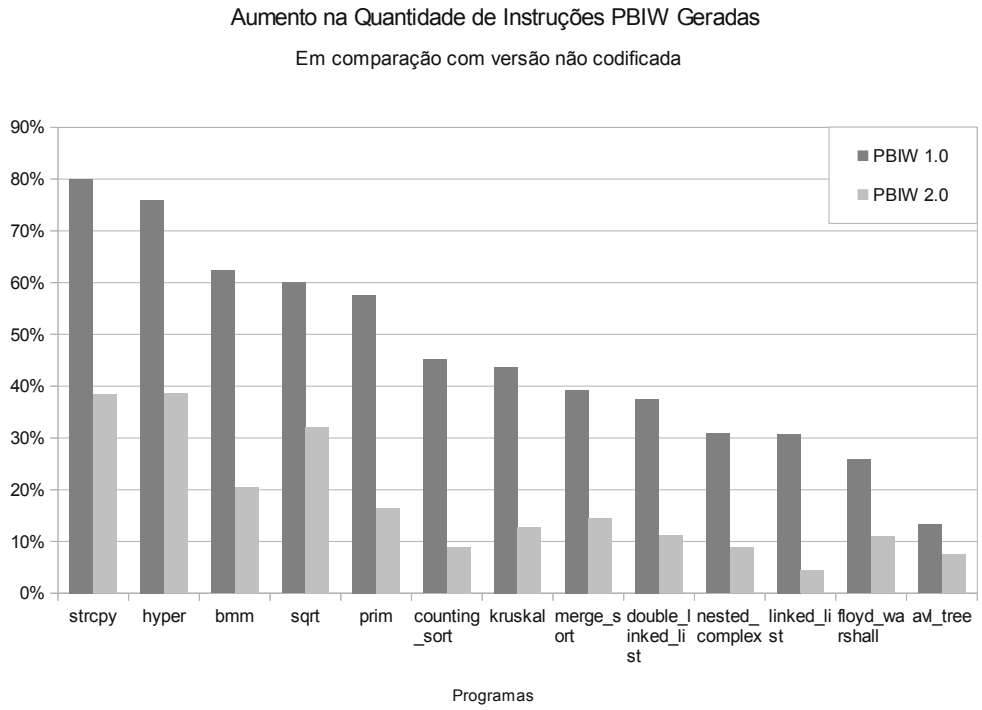


Figura 5.7: Quantidade de instruções PBIW geradas em comparação com quantidade de instruções não codificadas.

A média de padrões gerados sem e com a otimização de junção de padrões foi de, respectivamente, 36,85 e 19,85. Assim como na versão 1.0, a junção de padrões reduziu a quantidade de padrões praticamente pela metade: em média, aproximadamente 53% o número de padrões nas aplicações codificadas.

Na versão 2.0 da codificação PBIW (sem a otimização de junção de padrões), existem 479 padrões somando-se a quantidade de padrões gerados para cada um dos treze programas codificados. Somando-se a quantidade os padrões, agora com a otimização de junção de padrões, tem-se 258 padrões. Entretanto, é possível reduzir ainda mais esse número: aplicando-se novamente a otimização de junção de padrões nestes 258 padrões, reduz-se a quantidade para 224 padrões únicos para todos os treze programas codificados. Ao final desse processo, é possível armazenar — caso fosse necessário — todos os treze programas utilizando 16248 bytes em memória, contra 17820 bytes da versão 1.0 e 23360 bytes dos programas não codificados. Uma redução de 8,8% em relação à versão 1.0 e 30,45% nos programas não codificados.

Apesar da melhoria na taxa de compressão, houve uma redução de 1,3 e 3,9 (sem e com junção de padrões) na taxa de reúso de padrões comparada com a versão 1.0. Essa redução na

taxa de reúso deve-se à existência de instruções mais preenchidas o que, como consequencia, minimiza o reúso de padrões. A Figura 5.8 confirma este fato, mostrando uma quantidade média menor de padrões gerados na versão 1.0 quando comparada com a versão 2.0.

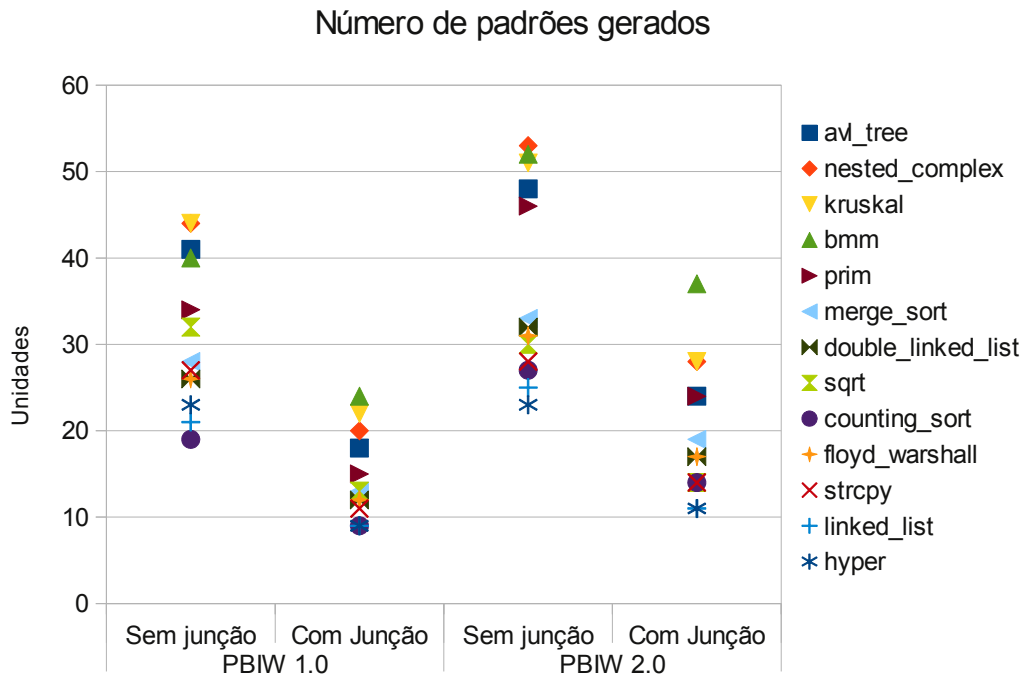


Figura 5.8: Quantidade de padrões gerados.

A Figura 5.9 mostra a porcentagem de melhoria no tamanho final dos programas. Barras apontando para o lado direito indicam uma redução no tamanho final do programa codificado, enquanto que barras apontando para a esquerda indicam um aumento no tamanho final.

Observa-se que a versão 2.0 (sem restrições) da codificação obteve uma redução no tamanho de todos os programas muito maior quando a otimização de junção de padrões é aplicada. Quando comparada com a versão 1.0 (com restrições), a versão sem restrições trouxe uma melhor compressão do programa original às custas da redução da taxa de reúso e quantidade de padrões gerados.

Na Figura 5.10 é possível notar o mesmo efeito da otimização de junção de padrões presente na versão 1.0 (Figura5.4): há uma redução na quantidade média de padrões que podem ser unidos e um aumento de padrões completos que não podem ser mais unidos a nenhum outro. Isso demonstra que a heurística utilizada no algoritmo de junção de padrões consegue ser eficiente ao maximizar a quantidade de padrões completos que não podem mais ser unidos a outros padrões.

5.3 Resultados Dinâmicos

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

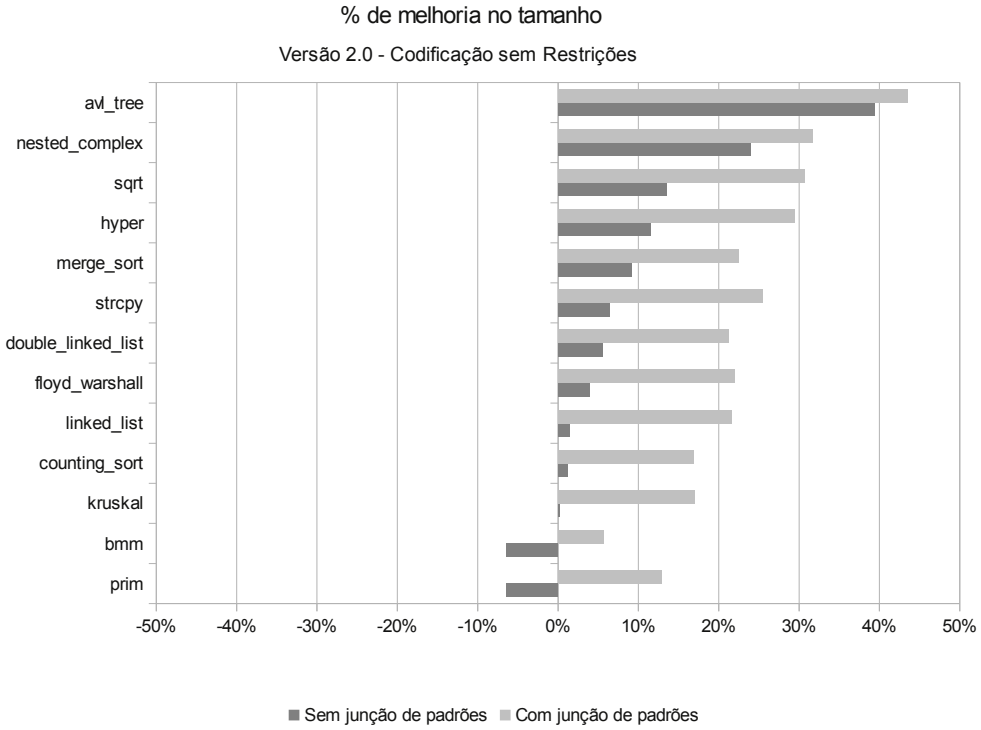


Figura 5.9: Porcentagem de melhoria no tamanho final dos programas PBIW ρ-VEX versão 2.0.

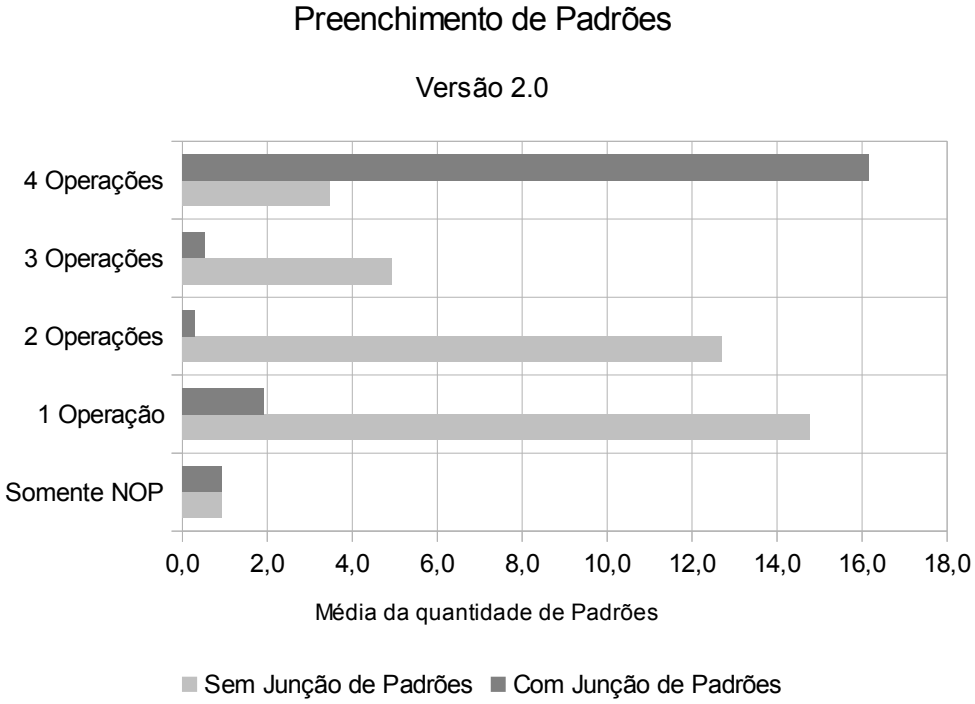


Figura 5.10: Nível de preenchimento dos padrões dos programas PBIW ρ-VEX versão 2.0.

que a adoção de uma nova *cache* (de padrões) não implica em perda de desempenho pelos programas codificados. Os resultados apresentados foram obtidos utilizando o simulador de caches Dinero [34].

Depois da compilação e simulação de cada programa no VEX *toolchain*, converte-se os traços de execução em dois traços válidos do programa para o processador ρ-VEX: um que representa o programa original e outro que representa o programa codificado. Em seguida, converte-se ambos os traços (original e codificado) de cada programa em um traço Dinero que é simulado considerando o impacto de *misses* das caches. Nos experimentos, todas as caches foram organizadas com mapeamento direto e tamanhos variando entre 128 bytes a 4 KBytes. Como a codificação PBIW necessita de duas caches (instruções e padrões), o tamanho de uma cache inteira foi dividida utilizando as seguintes proporções (Equações (5.2) e (5.3)):

$$\text{Tamanho Cache de Instruções PBIW} = \frac{\text{Tamanho total da cache}}{2} \tag{5.2}$$

$$\text{Tamanho Cache de padrões PBIW} = \frac{\text{Tamanho total da cache}}{2} \times 0,75 \tag{5.3}$$

Por exemplo, uma cache de 128 bytes para PBIW foi dividida em uma cache de 64 bytes para instruções e 48 bytes para padrões. A redução de 25% no tamanho da cache de padrões (proporcionada pela multiplicação do coeficiente 0,75) é necessária para o alinhamento dos padrões internamente.

Devido à ferramenta Dinero simular caches em tamanhos de potências de 2, os experimentos foram executados com caches de 16 bytes por linha. Como os padrões possuem 12 bytes, há um excedente de quatro bytes por linha. Nas estatísticas geradas pelo Dinero, o número de bytes transferidos leva em consideração os 16 bytes em vez de 12 bytes que cada padrão possui. Assim, esse excesso de 25% na quantidade de bytes transferidos é reduzido na quantidade de bytes transferidos da memória e no tamanho da cache original de forma a manter a real quantidade de bytes transferidos entre a cache de padrões e a memória principal.

Os experimentos dinâmicos foram executados com as versões 1.0 e 2.0 aplicando a otimização de junção de padrões devido a possuírem menor quantidade de padrões, além de padrões mais densos, melhorando o reúso e a probabilidade de *hit* na cache.

5.3.1 Versão 1.0

Os experimentos dinâmicos aqui apresentados mostram os efeitos dos *misses* na cache ao reduzir o tamanho da instrução em troca de adotar uma cache de padrões. O objetivo é que a adoção de uma nova cache (de padrões) não implique na perda de desempenho dos programas codificados com a versão 1.0 da codificação PBIW.

A Figura 5.11 mostra o melhor (menor) número de bytes buscados da memória (*cache miss*) para cada programa original e codificado considerando a codificação PBIW ρ-VEX versão 1.0 com junção de padrões. As caches possuem tamanhos diferentes entre os programas, mas o mesmo tamanho para as versões original e 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 avl_tree que buscou 30,32% menos bytes na memória, enquanto que o programa que buscou 12,04% bytes a mais na memória foi o programa strcpy. A média de melhoria ficou em 8,2%.

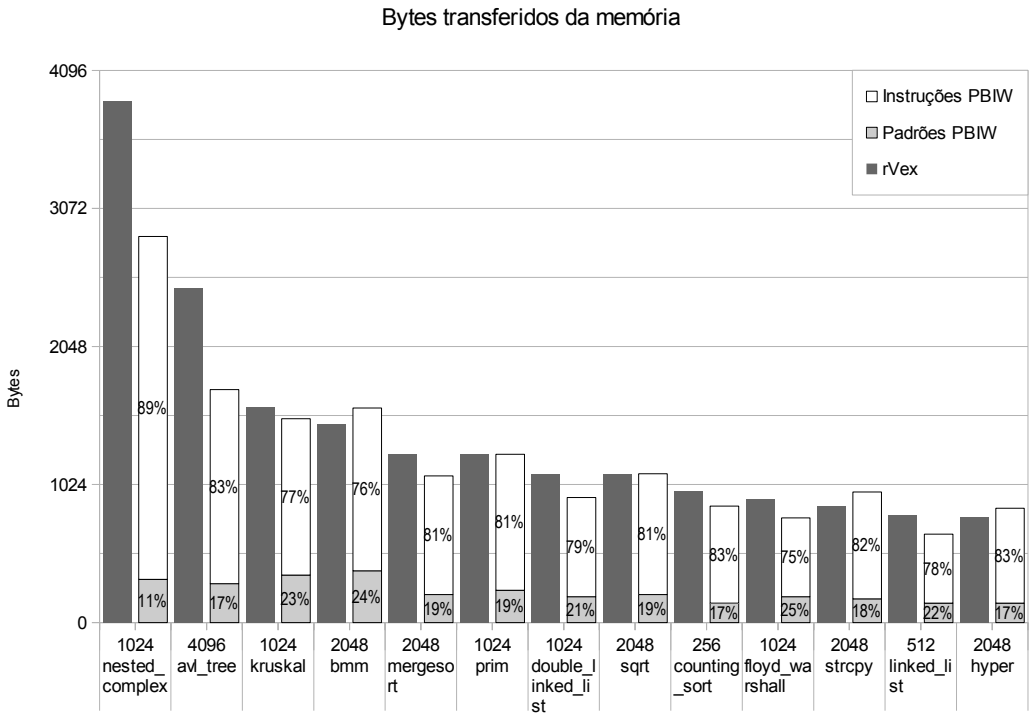


Figura 5.11: Total de Bytes buscados na memória para cada programa.

A Figura 5.12 mostra a porcentagem de melhoria $(1 - \frac{\text{tamanho codificado}}{\text{tamanho original}})$ na quantidade de bytes buscados na memória principal com a técnica PBIW. Barras à direita mostram programas beneficiados que buscaram menos bytes na memória (das caches de instruções e padrões juntas) em comparação com a cache de instruções originais. Já barras à esquerda mostram programas prejudicados (que buscaram mais bytes na memória).

Decodificador PBIW versão 1.0

Nesta seção discute-se o efeito do circuito decodificador PBIW versão 1.0 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® [35, 36].

A taxa de comutação, ou taxa de atividade, para o experimento de dissipação de potência foi fornecida para a ferramenta através de um arquivo de simulação (*.vcd) [35] construído a partir do *testbench* da execução de cada um dos treze programas.

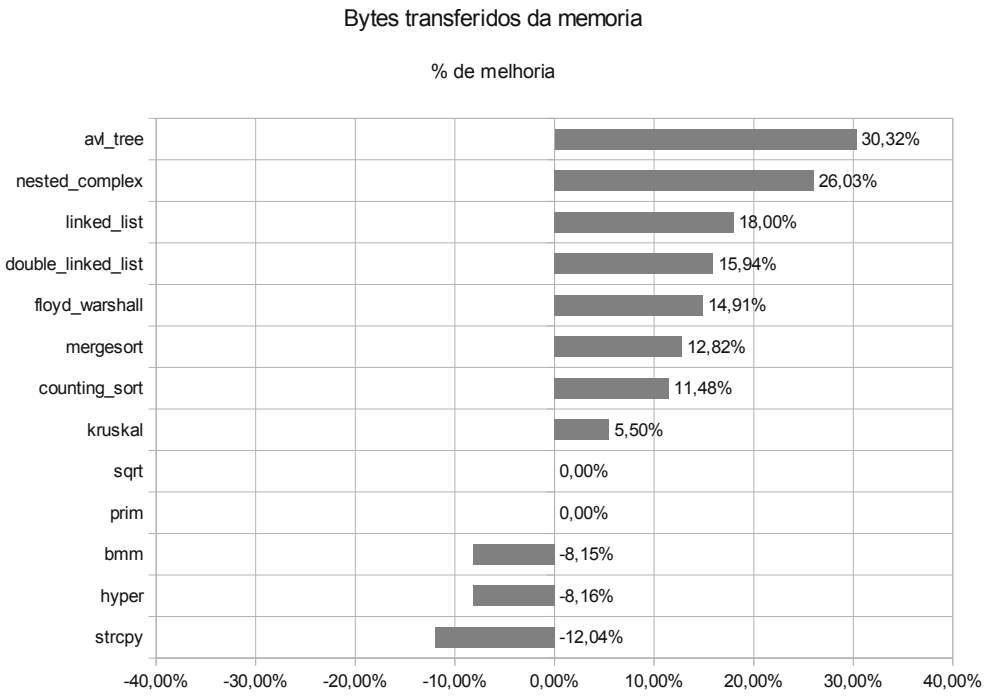


Figura 5.12: Porcentagem de melhoria na quantidade de bytes buscados da memória.

A potência total dissipada por um dispositivo FPGA é constituída por três componentes: Potência Dinâmica, Potência Estática e Potência de E/S. A potência de E/S é a energia consumida devido à carga e descarga das capacitâncias parasitas apresentadas pelos dispositivos conectados à FPGA. 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 capacitâncias parasitas em um circuito. Já a potência estática é proporcional 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 5.14), 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 os treze programas) como mostra a Figura 5.13. Dessa forma, será investigado apenas os impactos que a técnica de codificação PBIW traz sobre a potência dinâmica.

A Figura 5.14 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 treze programas utilizados) quando comparado com a versão original do ρ-VEX. Esse aumento é justificado pela inserção da tabela de padrões e a lógica do circuito decodificador (multiplexadores) estarem localizadas 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. Esse aumento no número de operações deve-se ao fato que a otimização da junção de padrões aumenta o número de operações presentes em um padrão.

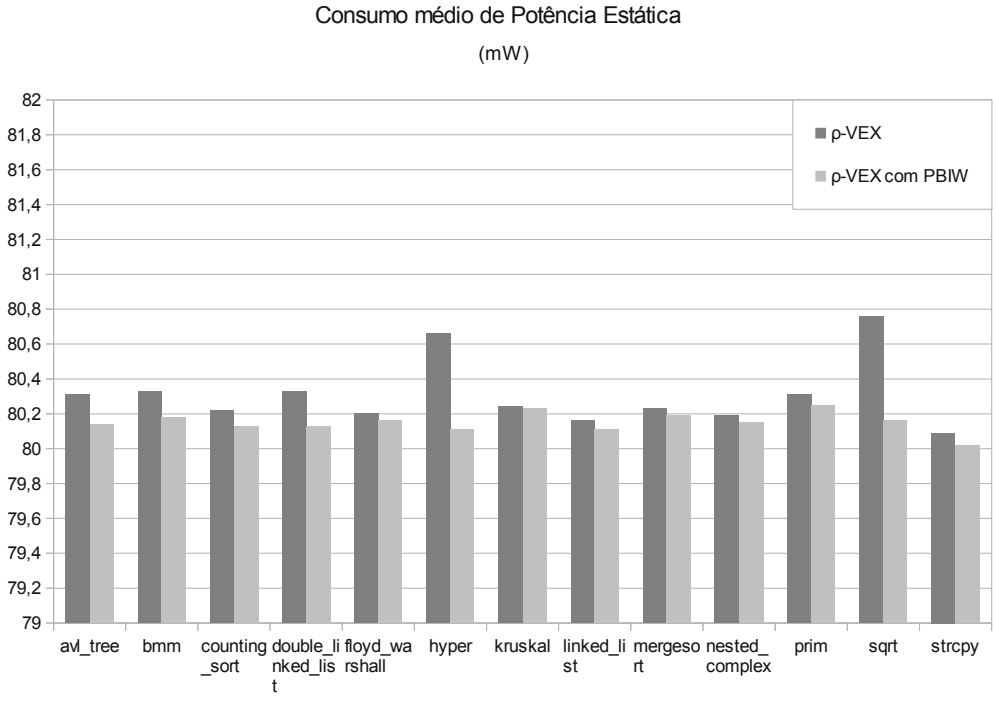


Figura 5.13: Média de potência estática consumida pela FPGA.

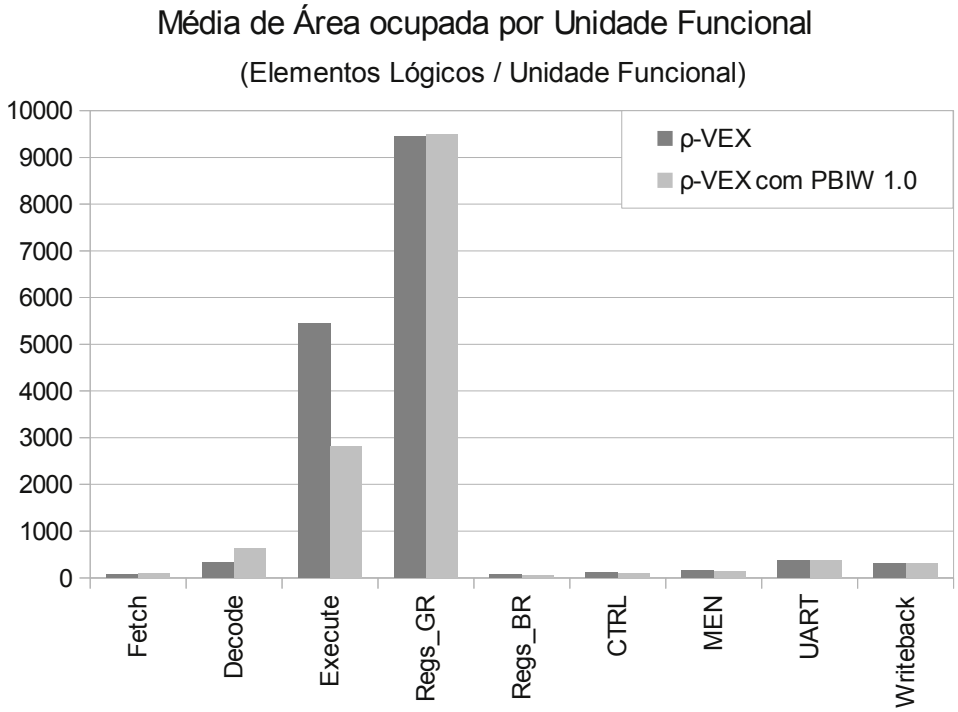


Figura 5.14: Média de área ocupada pelos principais componentes do processador ρ-VEX.

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). Na versão original do ρ-VEX, instruções distintas possuem campos com funcionalidades distintas na mesma posição da instrução. 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.

A técnica de codificação PBIW faz com que o processo de decodificação de instrução torne-se menos complexo, uma vez que o bloco lógico responsável pela decodificação é um simples multiplexador e o campo do padrão que indica o campo na instrução PBIW a ser utilizado. Esse novo projeto de decodificação na unidade Decode influencia o processo de roteamento da FPGA permitindo otimizações e com isso minimizando a lógica, em até mesmo outra unidade, da via de dados do processador.

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 5.15 que mostra uma redução de cerca 56% no consumo de potência dinâmica na memória de instrução (*imem*). Em média, o circuito decodificador trouxe redução de 40% no consumo de potência dinâmica de todo o processador e 15% na redução no número de elementos lógicos presentes no processador.

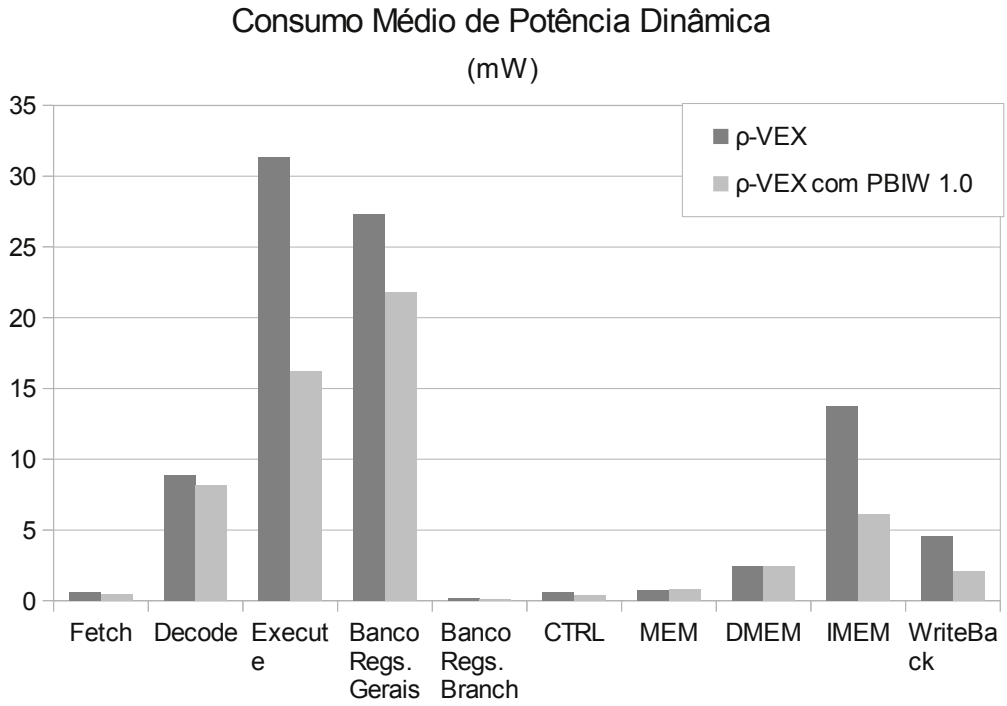


Figura 5.15: Média de potência dinâmica consumida por unidade funcional do processador ρ-VEX

A Figura 5.16 mostra o impacto sobre a freqüência de operação máxima do *clock* com a inserção do circuito decodificador. O circuito decodificador PBIW influencia o caminho crítico da via de dados do processador. O caminho crítico é definido como o caminho entre uma entrada e uma saída com atraso máximo. Especificamente, a redução na frequência máxima de *clock* é devido ao aumento do *slack time* no caminho crítico da via de dados.

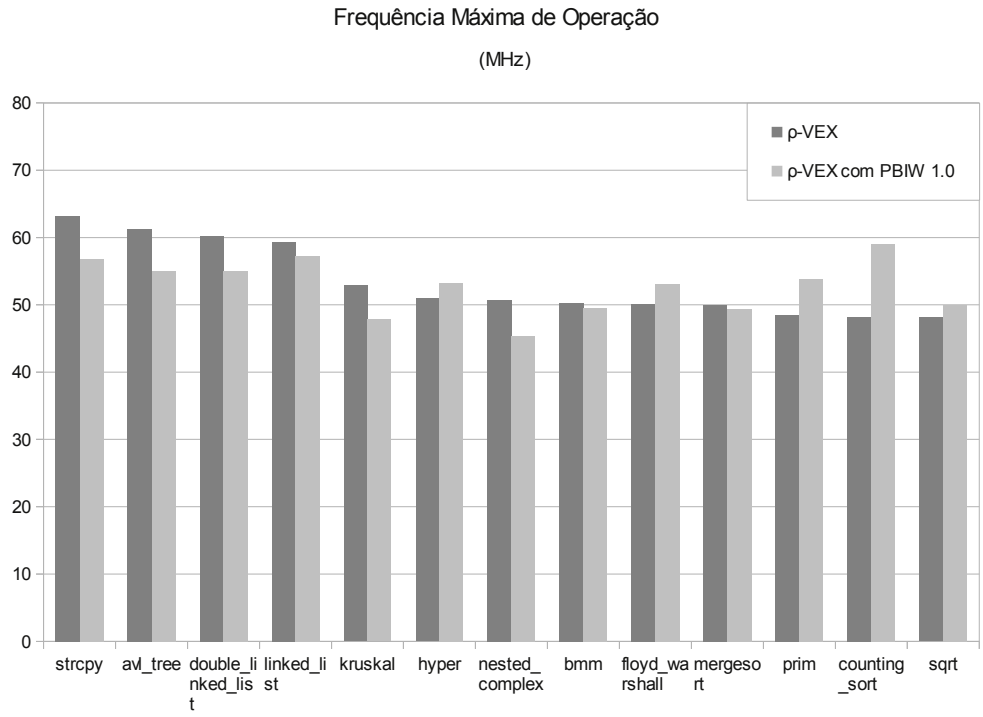


Figura 5.16: Frequência máxima de operação resultante da síntese de programas sobre o processador ρ-VEX com decodificador para a versão 1.0.

Conforme Equação 5.4, o *slack* (ST) consiste na diferença entre o tempo requerido (RT) para manter a frequência de referência e o tempo de chegada (AT). O tempo de chegada é o tempo necessário para que um sinal possa alcançar um ponto do circuito. O tempo requerido é o tempo máximo para que um sinal possa alcançar um ponto do circuito sem a necessidade de incrementar o tempo de ciclo do *clock*. Um *slack time* positivo de S em um nó implica que o tempo de chegada nesse nó pode ser aumentado por S sem afetar o atraso do circuito. Da mesma forma, um *slack time* negativo implica que o caminho é lento e que a frequência do sinal de *clock* deve ser reduzida para que o circuito respeite as restrições de tempo.

$$ST = RT - AT \tag{5.4}$$

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

Tabelas 5.2 e 5.3 mostram o caminho de dados para os programas avl_tree (pior caso de frequência máxima) e counting_sort (melhor caso de frequência máxima) respectivamente.

Caminho de dados por unidade	ρ-VEX	ρ-VEX com circuito decodificador
Clock	2,746	3,314
Memória de Instruções	3,202	3,202
Decode	4,889	1,794
Writeback	2,952	0
Banco de Regs GR	5,266	0
Execute	0	14,777
Tempo de chegada total (ns)	19,055	23,087

Tabela 5.2: Caminho de chegada de dados para o programa avl_tree.

Caminho de dados por unidade	ρ-VEX	ρ-VEX com circuito decodificador
Clock	2,723	3,118
Memória de Instruções	3,202	3,202
Decode	1,505	1,572
Execute	16,008	11,841
Tempo de chegada total (ns)	23,438	19,733

Tabela 5.3: Caminho de chegada de dados para o programa counting_sort.

Tabelas 5.2 e 5.3 mostram que o aumento dos elementos lógicos (circuitos multiplexadores) na Unidade de Decode impactam no tempo de chegada do sinal. De outra forma, o projeto de decodificador PBIW junto ao estágio de Decode tornou o processo de decodificação de instruções menos complexo do que na versão original devido à simplicidade da organização das instruções codificadas.

O efeito negativo que minimiza a frequência máxima de operação do *clock* é devido ao aumento do tempo de chegada do sinal na unidade Execute. Esse atraso deve-se à mudança na manipulação dos valores imediatos armazenados na instrução. Na codificação PBIW, um mesmo imediato pode ser reusado pelas quatro operações de uma instrução. Essa situação força a cópia dos bits do imediato para diferentes multiplexadores e, como consequência, sobrecarrega a lógica de decodificação mesmo em casos onde existem operações que são canceladas. No programa avl_tree, diversos padrões criados com o algoritmo de junção de padrões passaram a ter operações stw, ldw e addi. Em muitos casos, apenas uma dessas operações foi executada mas, conforme já apresentado, o imediato foi copiado para as demais operações.

5.3.2 Versão 2.0

A Figura 5.17 mostra o melhor (menor) número de bytes buscados da memória (*cache miss*) para cada programa original e codificado na versão 2.0 da codificação PBIW (com otimização de junção de padrões) para ρ-VEX. Da mesma forma como ocorreu nos experimentos

com a versão 1.0, as *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 apresentados acima dos nomes dos programas.

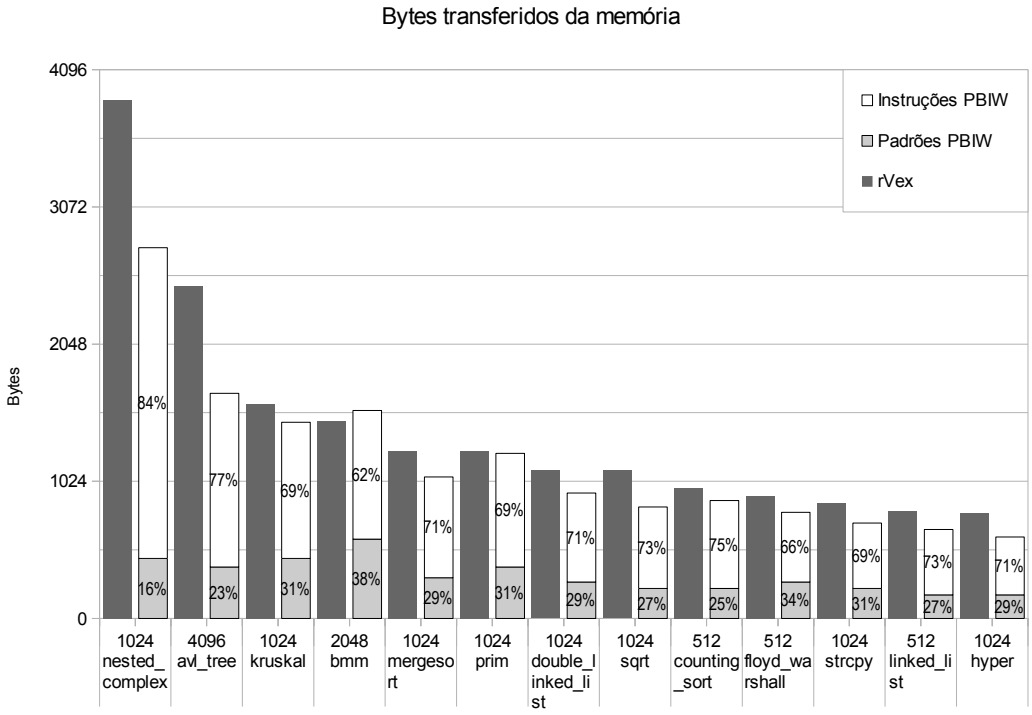


Figura 5.17: Total de Bytes buscados na memória para cada programa.

O programa que mais se beneficiou da adoção de PBIW para minimizar a taxa de *miss* na *cache* foi avl_tree que buscou 32,26% menos bytes na memória, enquanto que o programa que buscou 5,43% bytes a mais na memória foi o programa bmm. A média de melhoria ficou em 15,41%.

Nota-se um aumento significativo em comparação com a versão 1.0: a média de melhoria na quantidade de bytes trazidos da memória quase duplicou. A maioria dos programas trouxeram menos dados da memória em comparação com a versão 1.0 da codificação, sendo o programa bmm a única exceção. Os programas counting_sort, floyd_warshall, hyper, mergesort, sqrt e strcpy obtiveram melhor desempenho executando em uma cache de tamanho menor do que a usada quando codificados com a versão 1.0.

A Figura 5.18 mostra a porcentagem de melhoria na quantidade de bytes buscados na memória principal com a versão 2.0 da técnica PBIW. Barras à direita mostram programas beneficiados que buscaram menos bytes na memória (das caches de instruções e padrões juntas) em comparação com a cache de instruções não codificadas. As barras à esquerda mostram programas prejudicados (que buscaram mais bytes na memória).

Decodificador PBIW versão 2.0

A versão 2.0 do decodificador PBIW para o processador ρ-VEX consome em média 15% mais recursos do hardware da FPGA que a versão 1.0 previamente apresentada. Há um

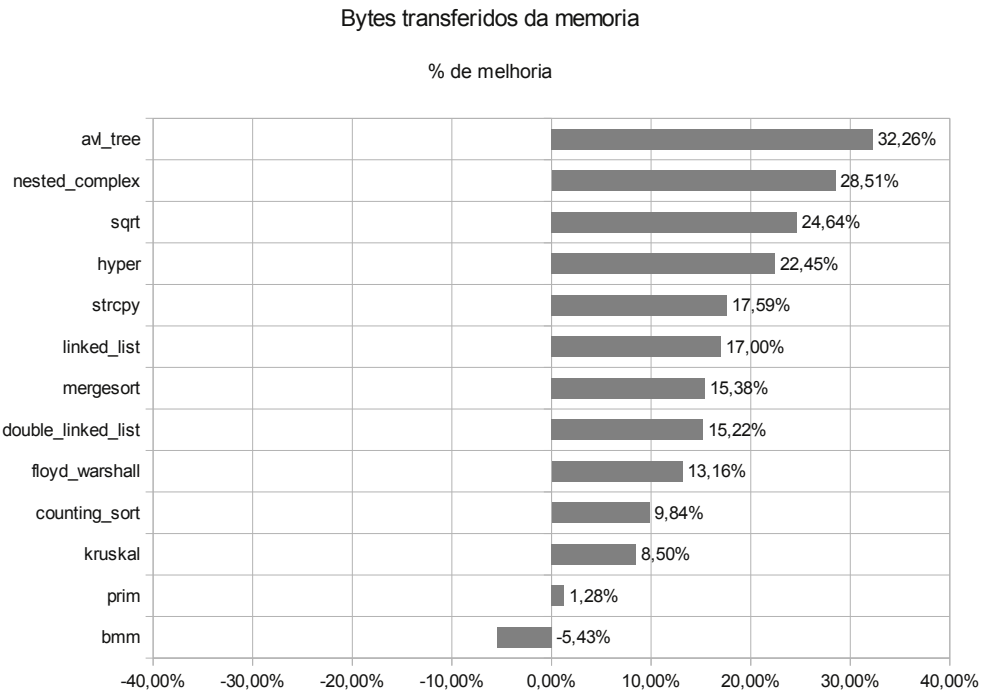


Figura 5.18: Porcentagem de melhoria na quantidade de bytes buscados da memória.

aumento na quantidade de elementos lógicos das unidades de *Decode*, *Execute* e do banco de registradores GR, em relação à versão 1.0, de respectivamente 75,1%, 7,4% e 9,5%, como pode ser visto nas comparações entre as Figuras 5.14 e 5.19. Isso é devido a maior quantidade de multiplexadores existentes no decodificador versão 2.0 (Figura 4.14).

Apesar do aumento da área, o consumo de potência dinâmica, Figura 5.20, teve um aumento de apenas 3% em comparação com a versão 1.0. Em comparação com o ρ-VEX sem o circuito decodificador PBIW, a versão 2.0 reduziu em 27,5% o consumo de potência dinâmica, enquanto que a versão 1.0 reduziu em 29,9%.

A Figura 5.21 mostra o impacto sobre a frequência de operação máxima do *clock* com a inserção do circuito decodificador para a codificação versão 2.0.

Tanto a redução da taxa de comutação quanto do número de bits da memória de instruções foram resultados esperados pelo emprego da técnica PBIW. Como o objetivo inicial da técnica é a codificação de instruções em instruções menores, inevitavelmente, a quantidade de bits da memória de instruções sofrerá reduções. Nesse sentido, a taxa de comutação também reduz, pois a quantidade de bits buscados da memória de instruções é menor.

Com o aumento do reúso dos padrões, as instruções codificadas adquirem características padronizadas e menos redundantes. Por exemplo, considere que primeiramente deve-se codificar a instrução 1 da Figura 5.22. A instrução codificada não precisa mais armazenar, por exemplo, o endereço dos operandos `r0.4` três vezes como na instrução original. Basta escolher um campo na instrução codificada para armazenar o endereço do operando `r0.4` e fazer todas operações que necessite desse endereço, como operando de leitura ou escrita, ter

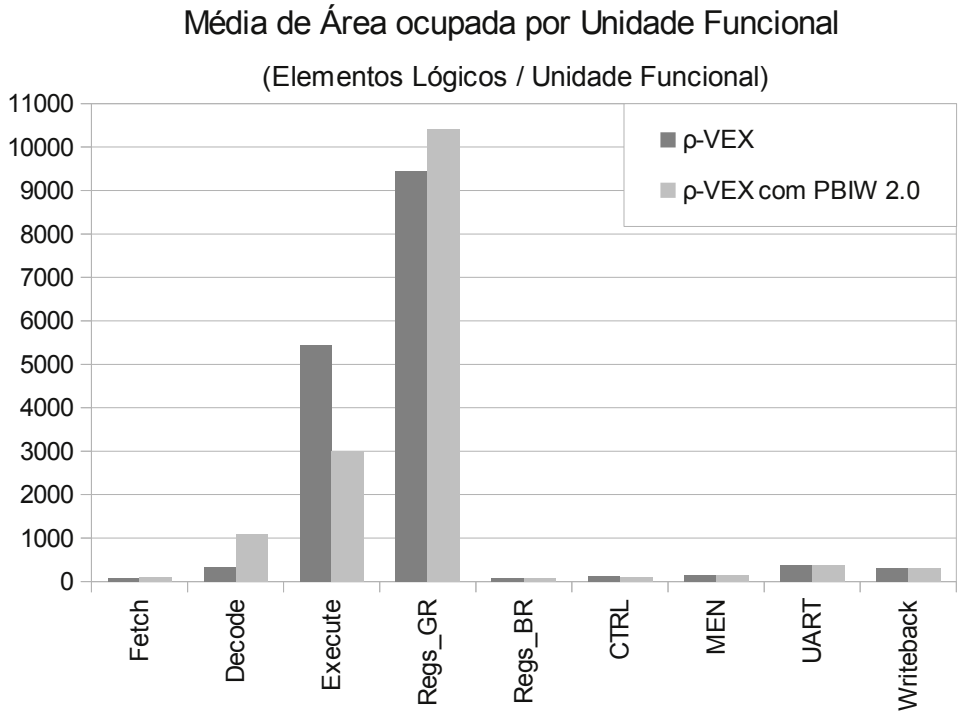


Figura 5.19: Média de área ocupada por componente do processador ρ-VEX.

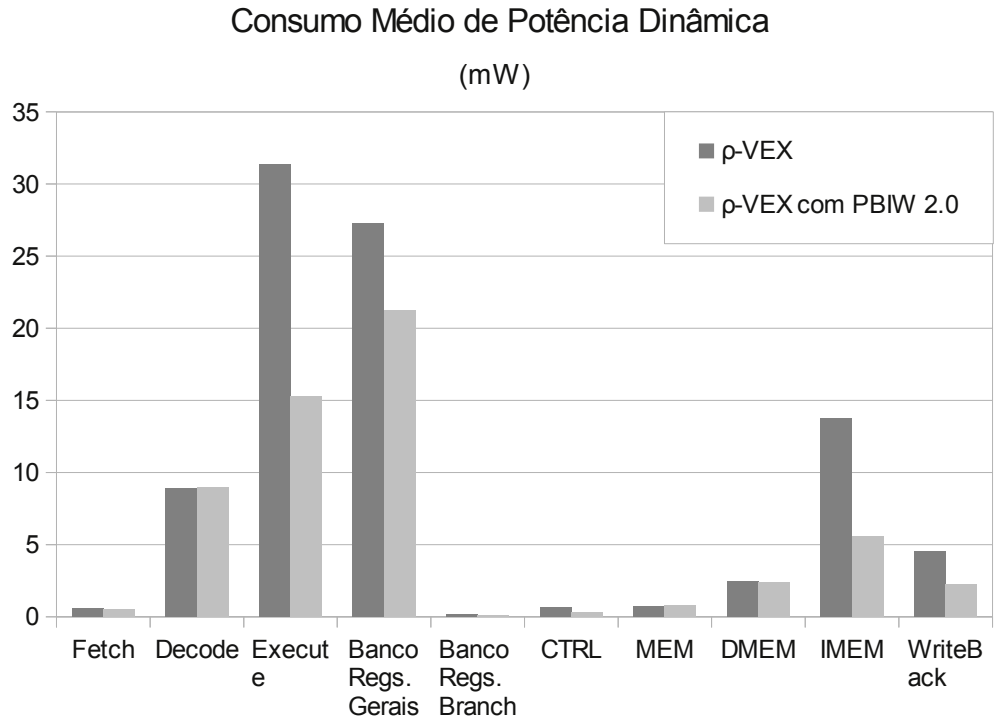


Figura 5.20: Média de potência dinâmica consumida por unidade funcional do processador ρ-VEX com decodificador da versão 2.0.

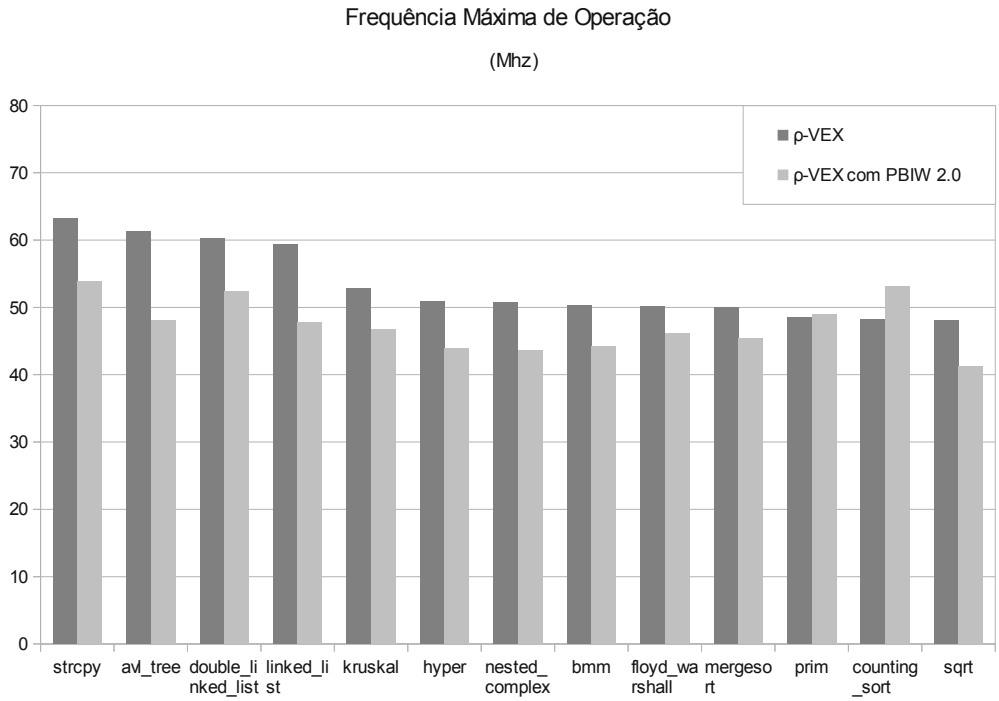


Figura 5.21: Frequência máxima de operação por programa codificado na versão 2.0.

seu índice no padrão apontando para esse campo na instrução codificada. Com isso, cria-se uma instrução codificada com uma característica de não ser redundante.

Ao codificar a instrução 2 da Figura 5.22, nota-se que possui as mesmas operações que a instrução 1, pode-se utilizar o mesmo padrão criado na codificação da instrução 1. Para reutilizar o mesmo padrão da instrução anterior, armazena-se os endereços dos novos operandos, nos mesmos campos da instrução codificada anterior, para reutilizar os mesmos ponteiros do padrão anterior. Dessa forma, torna-se a instrução codificada padronizada, pois apenas determinados campos não redundantes sofrerão comutação.

5.3.3 Taxa de *Miss*: Instruções e Padrões

Esta subseção apresenta um detalhamento sobre o comportamento de programas codificados com a versão 2.0 nas caches de instruções e padrões. A motivação para detalhar esse comportamento é desmistificar como acontecem os acessos às caches de instruções e padrões. Deve-se atentar para o fato que, ao organizar instruções e padrões em memórias caches, cada instrução buscada gerará um novo acesso para a cache de padrões. Embora os experimentos e resultados aqui apresentados são para traços gerados a partir do programa counting_sort com a versão 2.0, esse mesmo comportamento foi observado para os programas codificados na versão 1.0 (Apêndice C).

A Figura 5.23 apresenta todos os acessos na memória para o traço codificado do programa counting_sort. As linhas representam o comportamento durante a execução do programa para os acessos das caches de instruções e padrões PBIW e instruções originais (eixo X). O eixo Y mostra a taxa de *miss*. O programa counting_sort foi escolhido por ser representativo para

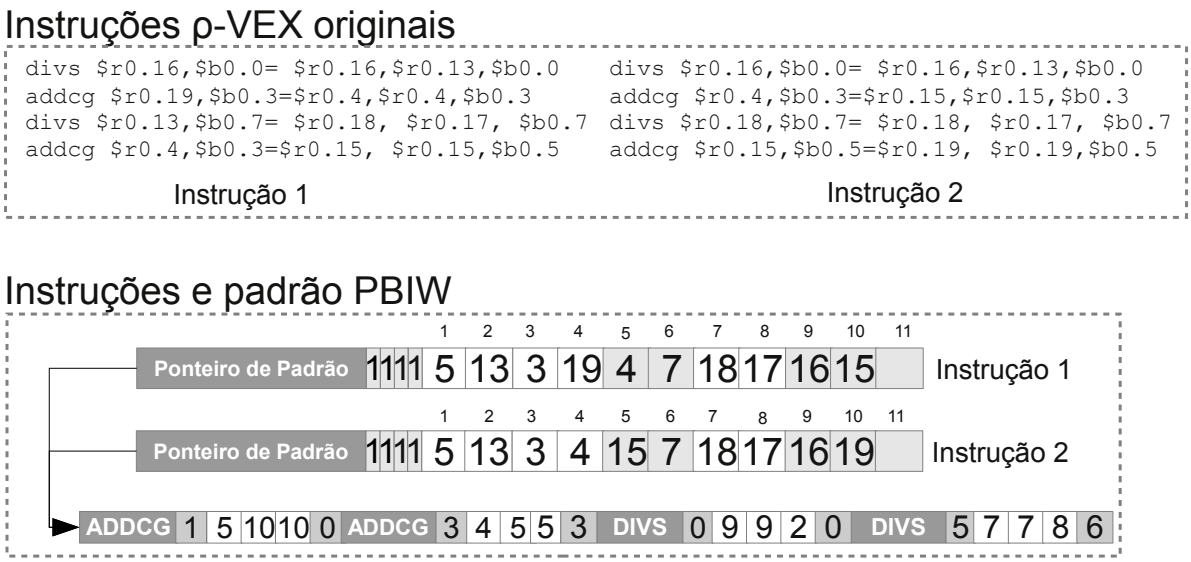


Figura 5.22: Trecho do código do programa Hyper para a codificação 2.0.

a maior parte dos programas. Os gráficos de miss de todos os programas são apresentados no Apêndice C.

A Figura 5.23 confirma o reúso dinâmico de padrões ao longo da execução do programa. É possível notar que os *misses* nos padrões são devidos à *misses* nas instruções codificadas. Entretanto, devido a existência do reúso de padrões pelas instruções, nem todos os *misses* das instruções codificadas implicam em *misses* nos padrões.

O comportamento da linha dos *misses* na cache de instruções (linha escura) é representada por um valor constante (coeficiente de angulação=0) no eixo Y. Nota-se que essa linha permanece constante por um período de execução maior que a linha representando os *misses* na cache de padrões. Os *hits* de instruções e padrões são notados quando as linhas caem de forma ininterrupta (coeficiente de angulação $\neq 0$). O reúso de padrões acelera o processo de preenchimento da cache diminuindo a quantidade de misses de conflito e compulsórios, tornando os *hits* mais frequentes.

Interessante observar o relacionamento entre os comportamentos dos *misses* nas instruções ρ-VEX originais e nas instruções PBIW ρ-VEX. Nota-se que a linha representando os acessos das instruções ρ-VEX originais finaliza antes das linhas de instruções e padrões PBIW ρ-VEX. Isso acontece porque, no processo de codificação, gerou-se mais instruções codificadas do que instruções originais. No caso específico do programa counting_sort, foram buscadas 547 instruções ρ-VEX originais e 694 instruções PBIW ρ-VEX. No entanto, a taxa de *miss* ao final dos acessos é igual (em torno de 11%) tanto para o traço das instruções originais quanto para o traço das instruções codificadas. Deve-se, entretanto, lembrar ainda que a memória cache das instruções codificadas é metade do tamanho da memória cache das instruções originais.

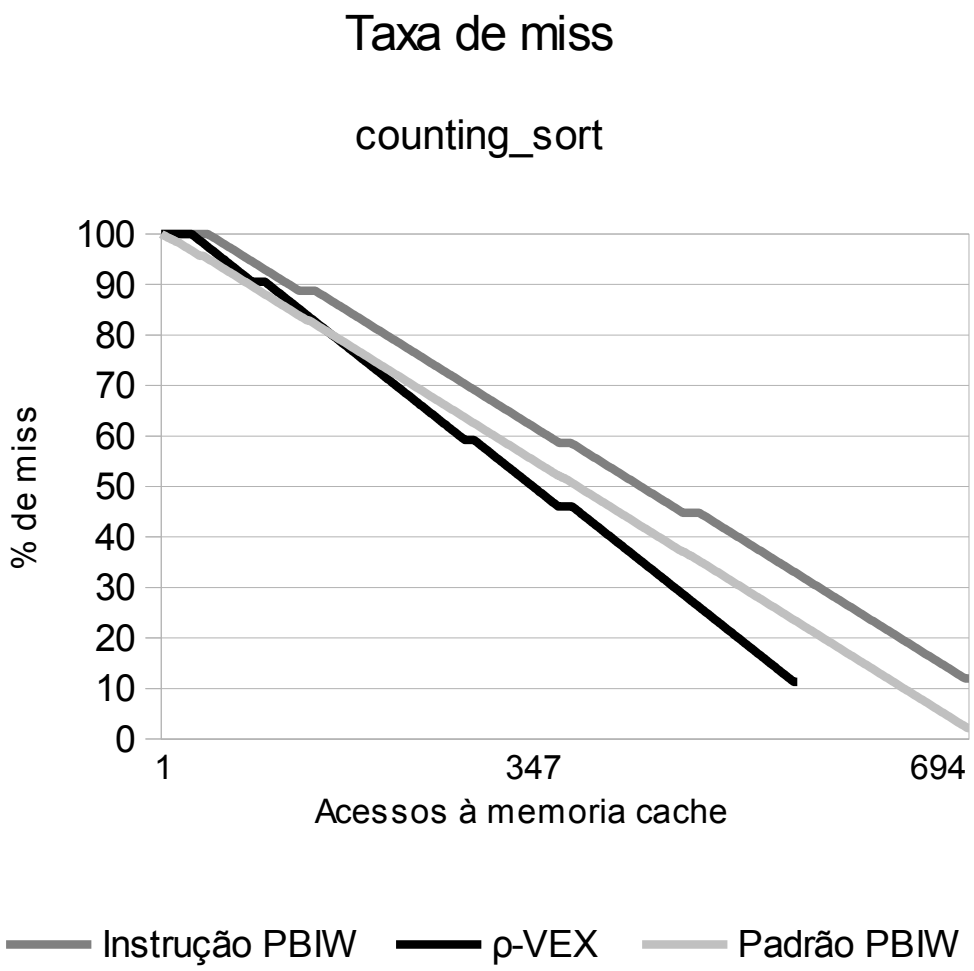


Figura 5.23: Taxa de *miss* PBIW ρ-VEX para o programa counting_sort.

5.4 Considerações Finais

Este capítulo apresentou os experimentos e resultados sobre o projeto da codificação PBIW (com e sem restrição) sobre o processador *softcore* ρ-VEX. Os resultados possibilitam concluir que a estratégia de codificação PBIW traz ganhos significativos no espaço ocupado pelo programa na memória de instruções. Dados preliminares, porém promissores, coletados da aplicação da codificação do benchmarks SPECINT2000 podem ser encontrados no Apêndice D. Além disso, pode-se também notar vantagens na adoção de decodificação PBIW sobre a potência dinâmica dissipada e a área total ocupada pelo projeto. O decodificador PBIW impactou no caminho crítico da via de dados do processador ρ-VEX e, como consequência, a frequência máxima alcançável pelo projeto reduziu em torno de 10%. No entanto, sabe-se que a utilização de técnicas de redução de potência como *clock gating* e/ou *guarded evaluation* podem minimizar os impactos do decodificador na frequência máxima de *clock* do projeto. Uma descrição detalhada desses experimentos considerando apenas o projeto e implementação da técnica de codificação PBIW com restrições, sobre o processador ρ-VEX, pode ser encontrada em [37, 38]. O próximo Capítulo apresenta as conclusões finais

sobre o trabalho, as contribuições geradas e possibilidades para desenvolvimento de trabalhos futuros.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho apresentou uma nova infraestrutura extensível e modular de codificação de software utilizando a técnica PBIW. Essa infraestrutura possui uma arquitetura que possibilita um fraco acoplamento entre suas diversas partes, desde estruturas de dados até algoritmos de codificação e (des)montagem de arquivos (de montagem ou executáveis).

Utilizando essa infraestrutura, foram implementadas duas versões de codificação da técnica PBIW para o processador *softcore* embarcado ρ-VEX : uma com restrições nas posições dos operandos (referenciada como versão 1.0) e uma sem restrição nas posições dos operandos (versão 2.0). Também foi implementada a otimização de junção de padrões no fluxo de codificação. Esta otimização trabalha sobre o conjunto de padrões gerados independentemente da versão utilizada (1.0 ou 2.0), gerando um novo conjunto de padrões reduzidos.

Com essa otimização aplicada, a versão 1.0 conseguiu uma taxa de compressão média de 85,82%, sendo a mínima de 105,00% e a máxima de 59,70%. Já a versão 2.0 conseguiu uma taxa de compressão média de 76,87%, sendo a mínima de 94,35% e a máxima de 56,51%. Ambas as versões de codificação PBIW para o processador ρ-VEX obtiveram resultados satisfatórios no quesito de redução de memória necessária para os programas experimentados.

Na versão 1.0 apenas três programas codificados tiveram seu tamanho final maior em relação a sua versão original. Já na versão 2.0 todos os programas obtiveram uma redução no tamanho final. O motivo é a limitação de espaço de operandos de leitura e escrita presente nas instruções da versão 1.0. Quando não há espaço restante para adicionar algum operando de escrita ou leitura, a instrução ρ-VEX original tem que ser dividida em duas, três ou até quatro instruções PBIW distintas. Esse efeito é drasticamente reduzido na codificação versão 2.0, onde os operandos de leitura e escrita podem ser armazenados em qualquer posição disponível na instrução.

Na simulação do comportamento dinâmico na memória, usando a ferramenta Dinero, a adição de mais uma memória cache (tabela de padrões) no circuito decodificador não causou impactos significativos de desempenho na execução dos programas avaliados. Em diversos casos, inclusive, foi possível reduzir o tamanho da memória cache total em comparação com o processador ρ-VEX original. Isso implicaria em uma menor área e consumo de energia do mesmo.

Quanto ao circuito decodificador, tanto pelas versões 1.0 e 2.0, houve uma redução na área ocupada e potência dinâmica consumida em relação ao processador ρ-VEX original. Essa redução é efeito da simplicidade do circuito decodificador PBIW, em comparação com o original empregado pelo processador ρ-VEX. Com menos sobreposição de campos e instruções menores, os softwares de síntese da FPGA conseguem otimizar o espaço ocupado pelo circuito impactaando no restante de consumo de potência dinâmica em outras unidades relacionadas.

Os resultados obtidos permitem concluir que a técnica PBIW mostra-se uma alternativa viável para reduzir a utilização de memória, a redução do tempo de busca de dados na memória e a dissipação de potência dinâmica em processadores embarcados, como ρ-VEX. Sua versatilidade é um ponto chave para conseguir atingir bons resultados em codificação de código e redução de potência dinâmica dissipada, que são características importantes em aplicações embarcadas restritas.

Além dos resultados demonstrando a viabilidade da técnica, percebe-se que a PBIW é um esquema de codificação extensível e mostra-se adequada para aplicação em diferentes ISAs voltadas tanto para processadores superscalares quanto ISAs de máquinas escalares. Este trabalho também ratifica essa premissa. O projeto e resultados obtidos com a codificação versão 1.0 foram aceitos para publicação nos anais do XIII Simpósio de Sistemas Computacionais (WSCAD-SCC) realizado em Petrópolis-RJ, e uma outra publicação derivada deste trabalho também foi realizada na forma de relatório técnico da FACOM, ambas em 2012.

Alguns trabalhos futuros podem ser enumerados para a melhoria da infraestrutura de codificação PBIW desenvolvida neste trabalho:

1. Desenvolvimento de um algoritmo paramétrico para codificação PBIW: o algoritmo codifica e gera instruções e padrões com base em um arquivo de configuração onde são descritos os parâmetros e formatos de instrução e padrão PBIW desejados;
2. Integração da infraestrutura de codificação como *back-end* de algum compilador de produção para emissão de código já codificado em PBIW ao final da compilação;
3. Desenvolvimento de novas técnicas de otimização para qualquer codificação PBIW;
4. *Front-end* e *back-end* para leitura e escrita, respectivamente, de arquivos binários ELF;
5. Suporte de outras ISAs na infraestrutura de codificação como exemplo: SPARCv8, MIPS, x86, etc;
6. Extensão do processador ρ-VEX para suportar faixas de endereços mais longos e, com isso, possibilitar a execução de programas maiores;
7. Implementação da tabela de padrões como memória cache em hardware a fim de avaliar o desempenho "real" dos programas codificados com a técnica PBIW;

Referências Bibliográficas

[1] K. D. Kissell, "Mips16: High-density MIPS for the Embedded Market," Real Time Systems, 1997.

[2] 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.

[3] A. Holdings, *ARM7TDMI Technical Reference Manual*, 2001. Página 1-7.

[4] 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 21st International Symposium on Computer Architecture and High Performance Computing*, (Washington, DC, USA), pp. 169–176, IEEE Computer Society, 2009.

[5] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SigArch*, vol. 23, no. 1, pp. 20–24, 1995.

[6] A. Saulsbury, F. Pong, and A. Nowatzyk, "Missing the Memory Wall: the Case for Processor/Memory Integration," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 90–101, October 1996.

[7] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the 1st Conference on Computing Frontiers*, pp. 162–167, April 2004.

[8] S. K. Menon and P. Shankar, "Space/Time Tradeoffs in Code Compression for the TMS320c62x Processor," Tech. Rep. IISc-CSA-TR-2004-4, Indian Institute of Science, 2004.

[9] R. Montserrat and P. Sutton, "Compiler Optimization and Ordering Effects on VLIW Code Compression," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 95–103, ACM, November 2003.

[10] J. Prakash, C. Sandeep, P. Shankar, and Y. N. Srikant, "Experiments with a New Dictionary Based Code-Compression Tool on a VLIW Processor," Tech. Rep. IISc-CSA-TR-2004-5, Indian Institute of Science, 2004.

[11] M. Ros and P. Sutton, "Code Compression Based on Operand-Factorization for VLIW Processors," in *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 559–569, ACM, September 2004.

[12] R. Batistella, "PBIW: Um Esquema de Codificação Baseado em Padrões de Instrução," Master's thesis, Instituto de Computação - Universidade Estadual de Campinas, Campinas-SP, Fevereiro 2008.

[13] R. Santos, R. Batistella, and R. Azevedo, "A pattern based instruction encoding technique for high performance architectures," *Int. J. High Perform. Syst. Archit.*, vol. 2, pp. 71–80, Mar. 2009.

[14] M. S. Schlansker and B. R. Rau, "EPIC: An Architecture for Instruction-Level Parallel Processors," Tech. Rep. 99-111, Hewlett Packard Laboratories Palo Alto, February 2000.

[15] M. S. Schlansker and B. R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *IEEE Computer*, vol. 33, pp. 37–45, February 2000.

[16] M. Len and I. Vaitsman, "VLIW: Old Architecture of the New Generation," Mar. 2011. http://ixbtlabs.com/articles2/vliw/.

[17] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code Compression Based on Operand Factorization," in *31st International Symposium on Microarchitecture*, pp. 194–201, ACM, 1998.

[18] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting, "Code Compression," in *Proceedings of Programming Language Design and Implementation*, pp. 358–365, ACM, 1997.

[19] M. Franz and K. Thomas, "Slim Binaries," *Communications of the ACM*, vol. 40, no. 2, pp. 87–94, 1997.

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

[21] T. van As, S. Wong, and G. Brown, "ρ-VEX: A Reconfigurable and Extensible VLIW Processor," in *IEEE International Conference on Field-Programmable Technology*, IEEE, 2008.

[22] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, pp. 379–423, July 1948.

[23] R. M. Fano, "The Transmission of Information," tech. rep., Cambridge (Mass.), USA: Research Laboratory of Electronics at MIT, 1949.

[24] T. Welch, "A Technique for High-Performance Data Compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[25] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[26] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of Code-Size Reduction Methods," *ACM Computing Survey*, vol. 35, no. 3, pp. 223–267, 2003.

[27] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the I.R.E.*, pp. 1095–1102, September 1952.

[28] S.-J. Nam, I.-C. Park, and C.-H. Kyung, "Improving Dictionary-Based Code Compression in VLIW Architectures," *IEICE Transactions on Fundamentals*, vol. E82-A, pp. 2318–2324, November 1999.

[29] R. F. dos Santos, "Otimizações para Codificadores de Instruções." Exame de Qualificação de Mestrado, 2012. Faculdade de Computação - Universidade Federal de Mato Grosso do Sul - Campo Grande-MS.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1 ed., 1994.

[31] Hewlett-Packard Laboratories, "VEX Toolchain." Disponível em: http://www.hpl.hp.com/downloads/vex/, Março 2011.

[32] 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.

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

[34] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator." [online], 1995. http://www.cs.wisc.edu/~markhill/DineroIV/.

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

[36] ALTERA, *The Quartus II TimeQuest Timing Analyzer*. Altera Corporation, 2012.

[37] R. Marks, F. Araújo, R. Santos, F. Yonehara, and R. Santos, "Design and Implementation of the PBIW Instruction Decoder in a Softcore Embedded Processor," in *13th Simpósio de Sistemas Computacionais (WSCAD-SSC)*, pp. 110–117, IEEE, 2012.

[38] R. Marks, F. Araujo, and R. Santos, "PBIW Instruction Encoding Technique on the ρ-Vex Processor: Design and First Results," tech. rep., UFMS, Abril 2012.

[39] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millenium," *IEEE Computer*, vol. 33, no. 7, pp. 28–35, 2000.

Apêndice A

Interfaces de Montagem

Neste apêndice serão descritos detalhes técnicos a respeito das interfaces de montagem. Essas interfaces provêem métodos de leitura para o codificador PBIW e métodos de escrita para preenchimento com os dados provenientes de um programa a ser codificado.

A.1 IContext

```
namespace GenericAssembly
{
 namespace Interfaces
 {
   class IContext
   {
   public:
     typedef std::deque<IOperation*> OperationDeque;
     typedef std::deque<IInstruction*> InstructionDeque;
     typedef std::deque<ILabel*> LabelDeque;

     virtual ~IContext() { }

     virtual void enableDebugging(bool) = 0;
     virtual bool isDebuggingEnabled() const = 0;

     virtual InstructionDeque getInstructions() const = 0;
     virtual OperationDeque getOperations() const = 0;
     virtual LabelDeque getLabels() const = 0;
   };
 }
}
```

Assinatura e finalidade dos principais métodos são descritas a seguir:

```
virtual void enableDebugging(bool) = 0;
```

Habilita ou desabilita o modo de depuração do objeto de contexto de montagem através do parâmetro informado. Usado para alterar o estado do contexto de montagem quando em execução normal ou em execução de depuração.

```
virtual bool isDebuggingEnabled() const = 0;
```

Informa se habilitado ou desabilitado o modo de depuração do objeto de contexto de montagem.

```
virtual InstructionDeque getInstructions() const = 0;
```

Retorna um deque de ponteiros para instruções presentes no contexto de montagem. Usado como entrada do contexto de codificação PBIW.

```
virtual OperationDeque getOperations() const = 0;
```

Retorna um deque de ponteiros para operações presentes no contexto de montagem. Também é usado como entrada do contexto de codificação PBIW.

```
virtual LabelDeque getLabels() const = 0;
```

Retorna um deque de ponteiros para rótulos presentes no contexto de montagem. Assim como os métodos anteriores, também é usado como entrada do contexto de codificação PBIW.

A.2 ILabel

```
namespace GenericAssembly
{
 namespace Interfaces
 {
   class ILabel
   {
   public:
     virtual ~ILabel() { }

     virtual void setAbsoluteAddress(unsigned int) = 0;
     virtual unsigned int getAbsoluteAddress() const = 0;

     virtual void setDestiny(IOperation* destiny) = 0;
     virtual IOperation* getDestiny() const = 0;

     virtual void setScope(Utils::LabelScope::Type scope) = 0;
     virtual Utils::LabelScope::Type getScope() const = 0;

     virtual void setName(const std::string& name) = 0;
     virtual std::string getName() const = 0;
   };
 }
}
```

Assinatura e finalidade dos principais métodos são descritas a seguir:

```
virtual void setAbsoluteAddress(unsigned int) = 0;
virtual unsigned int getAbsoluteAddress() const = 0;
```

Métodos *getter* e *setter* do endereço absoluto da instrução apontada.

```
virtual void setDestiny(IOperation* destiny) = 0;
virtual IOperation* getDestiny() const = 0;
```

Métodos *getter* e *setter* do ponteiro que aponta para a operação referenciada. Em arquiteturas VLIW a operação referenciada normalmente é a primeira operação presente na instrução referenciada. Esta convenção é usada para não haver erros de semantica ao realizar a codificação PBIW. Como a codificação PBIW pode dividir uma instrução VLIW original em duas ou mais instruções PBIW (devido a limitações de espaço livre destas) o rótulo deve sempre apontar para a primeira operação, de forma que, mesmo que haja a divisão, a corretude semântica do código seja mantida.

```
virtual void setScope(Utils::LabelScope::Type scope) = 0;
virtual Utils::LabelScope::Type getScope() const = 0;
```

Métodos *getter* e *setter* do escopo do rótulo. A classe que define os escopos existentes na infra-estrutura é detalhada na Seção 3.5.2.

```
virtual void setName(const std::string& name) = 0;
virtual std::string getName() const = 0;
```

Métodos *getter* e *setter* do nome do rótulo.

A.3 IInstruction

```
namespace GenericAssembly
{
 namespace Interfaces
 {
   class ILabel;
   class IOperation;

   class IInstruction
   {
   public:
     typedef std::deque<IOperation*> OperationDeque;

     virtual ~IInstruction() { }

     virtual void addReferencePBIWInstruction(const PBIW::Interfaces::IPBIWInstruction&) = 0;
     virtual std::set<const PBIW::Interfaces::IPBIWInstruction*> getPBIWInstructions() const = 0;

     virtual unsigned int getAddress() const = 0;
     virtual void setAddress(unsigned int) = 0;

     virtual bool haveLabel() const = 0;
     virtual ILabel* getLabel() const = 0;

     virtual OperationDeque getOperations() const = 0;
     virtual bool canSplitInstruction(const IOperation& operation) const = 0;
   };
 }
}
```

Assinatura e finalidade dos principais métodos são descritas a seguir:

```
virtual void addReferencePBIWInstruction(const PBIW::Interfaces::IPBIWInstruction&) = 0;
```

Adiciona a informação de quais instruções PBIW foram geradas a partir desta instrução não codificada.

```
virtual std::set<const PBIW::Interfaces::IPBIWInstruction*> getPBIWInstructions() const = 0;
```

Retorna a informação de quais instruções PBIW foram geradas a partir desta instrução não codificada.

```
virtual unsigned int getAddress() const = 0;
virtual void setAddress(unsigned int) = 0;
```

Métodos *getter* e *setter* do endereço global da instrução no programa.

```
virtual bool haveLabel() const = 0;
```

Informa se esta instrução possui um rótulo apontando para ela.

```
virtual ILabel* getLabel() const = 0;
```

Retorna um ponteiro para o rótulo que aponta para esta instrução.

```
virtual OperationDeque getOperations() const = 0;
```

Retorna um deque de ponteiros de operações que esta instrução contém. Por exemplo: caso seja uma instrução VLIW que contenha N operações, este método retornará um deque contendo N ponteiros para objetos `IOperation`.

```
virtual bool canSplitInstruction(const IOperation& operation) const = 0;
```

Informa se esta instrução pode ser dividida em duas a partir desta operação, isto é, operações anteriores a informada em uma instrução e o restante em outra. Método essencial em casos de instruções VLIW que não puderem ser codificadas em somente uma instrução PBIW.

A.4 IOperation

```
namespace GenericAssembly
{
 namespace Utils
 {
class OperandVector;
 }

 namespace Interfaces
 {
   class IOperation
   {
   public:
     virtual ~IOperation() { }

     virtual void setBelongedInstruction(IInstruction* instruction) = 0;
     virtual IInstruction* getBelongedInstruction() const = 0;

     virtual unsigned int getAddress() const = 0;
     virtual void setAddress(unsigned int address) = 0;

     virtual void setLayoutType(int layoutType) = 0;

     virtual unsigned int getOpcode() const = 0;
     virtual bool isOpcode(unsigned int opcode) const = 0;

     virtual void setTextRepresentation(std::string textRepresentation) = 0;
     virtual std::string getTextRepresentation() const = 0;

     virtual Utils::OperandVector exportOperandVector() const = 0;
   };
 }
}
```

Assinatura e finalidade dos principais métodos são descritas a seguir:

```
virtual void setBelongedInstruction(IInstruction* instruction) = 0;
virtual IInstruction* getBelongedInstruction() const = 0;
```

Métodos *getter* e *setter* de ponteiro para a instrução que contém esta operação.

```
virtual unsigned int getAddress() const = 0;
virtual void setAddress(unsigned int address) = 0;
```

Métodos *getter* e *setter* do endereço global referente a essa operação, independente do endereço da instrução que a contém. Por exemplo: caso existam 2 instruções com 4 operações cada, as instruções terão os endereços 0 e 1. Já as operações terão endereços de 0 a 7: a primeira instrução com as operações de endereço de 0 a 3 e a segunda instrução com operações de endereço de 4 a 7.

```
virtual unsigned int getOpcode() const = 0;
```

Recupera o valor do *opcode* desta operação.

```
virtual bool isOpcode(unsigned int opcode) const = 0;
```

Retorna verdadeiro ou falso caso o *opcode* desta operação seja igual ao apresentado no parâmetro.

A.5 IOperand

```
namespace GenericAssembly
{
 namespace Interfaces
 {
   class IOperand
   {
   public:
     virtual ~IOperand() { }

     virtual IOperand* clone() const = 0;

     virtual void setTypeCode(int) = 0;
     virtual int getTypeCode() const = 0;

     virtual bool isImmediate() const = 0;
     virtual bool isRegister() const = 0;

     virtual void setValue(int) = 0;
     virtual int getValue() const = 0;

     virtual void setOperationBelonged(IOperation*) = 0;
     virtual IOperation* getOperationBelonged() const = 0;

     virtual bool operator==(const IOperand&) const = 0;
     virtual bool operator!=(const IOperand&) const = 0;
   };
 }
}
```

```
virtual void setTypeCode(int) = 0;
virtual int getTypeCode() const = 0;
```

Métodos *getter* e *setter* do código de tipo deste operando. Um código de tipo é um valor inteiro dos elementos de uma enumeração. Tais elementos codificam tipos de operandos existem disponíveis na ISA usada como, por exemplo, registradores de leitura, escrita e imediatos.

```
virtual bool isImmediate() const = 0;
virtual bool isRegister() const = 0;
```

Métodos auxiliares que informam se este operando é um registrador ou um valor imediato.

```
virtual void setValue(int) = 0;
virtual int getValue() const = 0;
```

Métodos *getter* e *setter* do valor deste operando.

```
virtual void setOperationBelonged(IOperation*) = 0;
virtual IOperation* getOperationBelonged() const = 0;
```

Métodos *getter* e *setter* do ponteiro para a operação que este operando pertence.

A.6 IPrinter

```
namespace GenericAssembly
{
 namespace Interfaces
 {
   template <class TInstruction, class TOperation>
   class IPrinter
   {
   public:
     virtual ~IPrinter() { }

     virtual void printOperation(const TOperation&, const std::vector<unsigned int>&) = 0;
     virtual void printInstruction(const TInstruction&) = 0;
     virtual void printHeader() = 0;
     virtual void printFooter() = 0;
     virtual std::ostream& getOutputStream() = 0;
   };
 }
}
```

```
virtual void printOperation(const TOperation&, const std::vector<unsigned int>&) = 0;
```

Imprime a operação passada como parâmetro. Também recebe um vetor de inteiros contendo a representação em binário da operação a ser impressa, caso necessário.

```
virtual void printInstruction(const TInstruction&) = 0;
```

Imprime a instrução passada como parâmetro.

```
virtual void printHeader() = 0;
```

Imprime o cabeçalho de saída do arquivo.

```
virtual void printFooter() = 0;
```

Imprime o rodapé de saída do arquivo.

```
virtual std::ostream& getOutputStream() = 0;
```

Retorna o *stream* usado para escrever o arquivo de saída.

Apêndice B

Interfaces de Codificação

Neste apêndice serão descritos detalhes técnicos a respeito das interfaces de codificação PBIW. Essas interfaces provêem métodos para a comunicação de dados entre diversos componentes do fluxo de codificação PBIW. O objetivo é prover fraco acoplamento entre os contextos de codificação e otimização PBIW.

B.1 IPBIWSet

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIWSet
    {
    public:
      virtual ~IPBIWSet () { }

      virtual unsigned int getOriginalInstructionCount() const = 0;

      virtual std::deque<IPBIWPattern*> getPatterns() const = 0;
      virtual std::deque<IPBIWInstruction*> getInstructions() const = 0;
      virtual std::deque<ILabel*> getLabels() const = 0;

      virtual void printStatistics(IPBIWPrinter&) = 0;
      virtual void printInstructions(IPBIWPrinter&) = 0;
      virtual void printPatterns(IPBIWPrinter&) = 0;
    };
  }
}
```

A interface IPBIWSet define operações comuns de um container de estruturas PBIW, como contextos de codificação e otimização.

```
virtual void printStatistics(IPBIWPrinter&) = 0;
virtual void printInstructions(IPBIWPrinter&) = 0;
virtual void printPatterns(IPBIWPrinter&) = 0;
```

Imprime estatísticas, instruções e padrões deste conjunto de estruturas PBIW usando a impressora passada como referência.

```
virtual std::vector<IPBIWPattern*> getPatterns() const = 0;
virtual std::vector<IPBIWInstruction*> getInstructions() const = 0;
virtual std::vector<ILabel*> getLabels() const = 0;
```

Retorna rótulos, instruções e padrões deste conjunto de estruturas PBIW.

B.2 IPBIW

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIW : public IPBIWSet
    {
      public:
        virtual ~IPBIW() {}

        virtual void setDebug(bool) = 0;
        virtual bool isDebug() const = 0;

        virtual void
        encode(const std::deque<GenericAssembly::Interfaces::IInstruction*>&) = 0;

        virtual void
        decode(const std::deque<IPBIWInstruction*>&, const std::deque<IPBIWPattern*>&) = 0;

        virtual void registerOptimizer(IPBIWOptimizer&) = 0;
        virtual void runOptimizers() = 0;
    };
  }
}
```

```
virtual void setDebug(bool) = 0;
virtual bool isDebug() const = 0;
```

Métodos *getter* e *setter* para definir ou checar o estado de depuração do contexto de codificação PBIW.

```
virtual void
encode(const std::deque<GenericAssembly::Interfaces::IInstruction*>&) = 0;
```

Método principal de codificação. Recebe um conjunto de ponteiros para instruções vindas do contexto de montagem que serão codificadas.

Este parâmetro foi pensado de forma que seja possível codificar qualquer conjunto de instruções: desde instruções específicas, passando por blocos básicos, tracos específicos indo até todas as instruções presentes em um programa.

```
virtual void
decode(const std::deque<IPBIWInstruction*>&, const std::deque<IPBIWPattern*>&) = 0;
```

Método principal de decodificação PBIW. Recebe dois conjuntos: de instruções e de padrões PBIW. A partir destes dois conjuntos, consegue restaurar a codificação das instruções originais do programa codificado. Este método existe por motivo de conveniência caso seja necessária sua utilização.

```
virtual void registerOptimizer(IPBIWOptimizer&) = 0;
```

Método que registra um otimizador que poderá ser executado depois do processo de codificação PBIW. Os otimizadores registrados são colocados em uma fila, na qual a entrada do próximo otimizador registrado é a saída do anterior.

Frisa-se que cada contexto, tanto de codificação quanto de otimização PBIW, possui seu próprio conjunto de instruções e padrões criados a partir dos dados de entrada. Nenhum contexto de otimização deve alterar as instruções e padrões PBIW de outro contexto.

```
virtual void runOptimizers() = 0;
```

Método que dispara a execução dos otimizadores PBIW registrados neste contexto de codificação.

B.3 IPBIWFactory

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIWInstruction;
    class IPBIWPattern;
    class IOperation;
    class IOperand;
    class ILabel;

    class IPBIWFactory
    {
    public:
      virtual ~IPBIWFactory() { }

      virtual IPBIWInstruction* createInstruction() const = 0;
      virtual IPBIWPattern* createPattern() const = 0;
      virtual IOperation* createOperation() const = 0;
      virtual IOperand* createOperand() const = 0;
      virtual ILabel* createLabel() const = 0;

      virtual IOperation*
        createOperation(const GenericAssembly::Interfaces::IOperation&) const = 0;

      virtual IOperand*
        createOperand(const GenericAssembly::Interfaces::IOperand&) const = 0;

      virtual ILabel*
        createLabel(const GenericAssembly::Interfaces::ILabel&) const = 0;
    };
  }
}
```

```
virtual IPBIWInstruction* createInstruction() const = 0;
virtual IPBIWPattern* createPattern() const = 0;
virtual IOperation* createOperation() const = 0;
virtual IOperand* createOperand() const = 0;
virtual ILabel* createLabel() const = 0;
```

Criação de objetos da infra-estrutura de codificação PBIW. Os objetos criados por estes métodos não possuem qualquer informação, estão ”vazios”, esperando para serem preenchidos com os dados importantes a cada um.

```
virtual IOperation* createOperation(const GenericAssembly::Interfaces::IOperation&) const = 0;
```

Cria uma nova operação a partir dos dados de uma operação vinda do contexto de montagem.

```
virtual IOperand* createOperand(const GenericAssembly::Interfaces::IOperand&) const = 0;
```

Cria um novo operando a partir dos dados de um operando vindo do contexto de montagem.

```
virtual ILabel* createLabel(const GenericAssembly::Interfaces::ILabel&) const = 0;
```

Cria um novo rótulo a partir dos dados de um rótulo vinda do contexto de montagem.

B.4 ILabel

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIWInstruction;

    class ILabel
    {
    public:
      virtual ~ILabel() { }

      virtual ILabel* clone() const = 0;

      virtual void setAbsoluteAddress(unsigned int absoluteAddress) = 0;
      virtual unsigned int getAbsoluteAddress() const = 0;

      virtual void setDestiny(IPBIWInstruction* destiny) = 0;
      virtual IPBIWInstruction* getDestiny() const = 0;

      virtual void setScope(GenericAssembly::Utils::LabelScope::Type scope) = 0;
      virtual GenericAssembly::Utils::LabelScope::Type getScope() const = 0;

      virtual void setName(const std::string& name) = 0;
      virtual std::string getName() const = 0;
    };
  }
}
```

```
virtual void setAbsoluteAddress(unsigned int absoluteAddress) = 0;
virtual unsigned int getAbsoluteAddress() const = 0;
```

Métodos *getter* e *setter* para definir o endereço absoluto que este rótulo aponta.

```
virtual void setDestiny(IPBIWInstruction* destiny) = 0;
virtual IPBIWInstruction* getDestiny() const = 0;
```

Métodos *getter* e *setter* de um ponteiro para a instrução PBIW que ele aponta.

```
virtual void setScope(GenericAssembly::Utils::LabelScope::Type scope) = 0;
virtual GenericAssembly::Utils::LabelScope::Type getScope() const = 0;
```

Métodos *getter* e *setter* para definir o escopo do rótulo.

```
virtual void setName(const std::string& name) = 0;
virtual std::string getName() const = 0;
```

Métodos *getter* e *setter* para definir a string deste módulo.

B.5 IPBIWInstruction

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIWInstruction
    {
      public:
        virtual ~IPBIWInstruction() { }

        virtual IPBIWInstruction* clone() const = 0;
```

```
        virtual void setAddress(unsigned int) = 0;
        virtual unsigned int getAddress() const = 0;

        virtual void setLabel(const ILabel*) = 0;
        virtual ILabel* getLabel() const = 0;

        virtual void pointToPattern(IPBIWPattern&) = 0;
        virtual const IPBIWPattern* getPattern() const = 0;

        virtual void setCodingOperation(IOperation& operation) = 0;
        virtual IOperation* getCodingOperation() const = 0;

        virtual const IOperand& containsOperand(const IOperand&) const = 0;

        virtual void addOperand(IOperand&) = 0;

        virtual void addReadOperand(IOperand&) = 0;
        virtual void addWriteOperand(IOperand&) = 0;

        virtual bool hasOperandSlot(const PBIW::Utils::OperandItemDTO&) = 0;asdf

        virtual bool hasReadOperandSlot() const = 0;
        virtual bool hasWriteOperandSlot() const = 0;
        virtual bool hasControlOperationWithLabelDestiny() = 0;

        virtual std::string getLabelDestiny() const = 0;
        virtual void setLabelDestiny(std::string) = 0;

        virtual void setBranchDestiny(const IPBIWInstruction& branchDestiny) = 0;
        virtual IPBIWInstruction* getBranchDestiny() const = 0;

        virtual int readOperandQuantity() const = 0;
        virtual int writeOperandQuantity() const = 0;

        virtual void print(IPBIWPrinter&) const = 0;

        struct OperandComparer {
          bool operator() (const IOperand& lhs, const IOperand& rhs) const
          { return lhs < rhs; }
        };

        virtual PBIW::Utils::OperandVector getOperands() const = 0;

        virtual void
          setOperationReferences(const std::list<GenericAssembly::Interfaces::IOperation*>&) = 0;

        virtual std::list<GenericAssembly::Interfaces::IOperation*>
          getOperationReferences() const = 0;

        typedef std::vector<bool> AnnulationBits;
        virtual const AnnulationBits& getAnnulBits() const = 0;

        virtual void setAnnulBit(int, bool) = 0;
        virtual void setAnnulBits(const AnnulationBits& vectorBits) = 0;
        virtual void updateAnnulBits(int index1, int index2) = 0;
    };
  }
}
```

```
virtual void setAddress(unsigned int) = 0;
virtual unsigned int getAddress() const = 0;
```

Métodos *getter* e *setter* para o endereço absoluto da instrução PBIW.

```
virtual void setLabel(const ILabel*) = 0;
virtual ILabel* getLabel() const = 0;
```

Métodos *getter* e *setter* para o rótulo que aponta para esta instrução PBIW.

```
virtual void pointToPattern(IPBIWPattern&) = 0;
virtual const IPBIWPattern* getPattern() const = 0;
```

Métodos *getter* e *setter* para padrão apontado por esta instrução PBIW.

```
virtual void setCodingOperation(IOperation& operation) = 0;
virtual IOperation* getCodingOperation() const = 0;
```

Métodos *getter* e *setter* para definir qual operação está sendo codificada no momento para esta instrução PBIW. Útil para obter dados e tomar melhores decisões durante o processo de codificação da instrução atual pelo algoritmo de codificação PBIW.

```
virtual const IOperand& containsOperand(const IOperand&) const = 0;
```

Busca o operando passado como parâmetro na instrução PBIW atual. Por convenção, deve retornar o mesmo operando passado como parâmetro caso não encontre nenhum operando na instrução com o mesmo conteúdo. Caso encontre algum operando na instrução deve retorná-lo.

```
virtual void addReadOperand(IOperand&) = 0;
virtual void addWriteOperand(IOperand&) = 0;
virtual void addBranchOperand(IOperand&) = 0;
```

Métodos que adicionam um operando de leitura, escrita e de controle (desvios, ou *branch*) respectivamente na instrução PBIW.

```
virtual bool hasReadOperandSlot() const = 0;
virtual bool hasWriteOperandSlot() const = 0;
virtual bool hasControlOperationWithLabelDestiny() = 0;
```

Métodos que retornam se ainda há slots para operandos de leitura, escrita ou de controle, respectivamente, na instrução PBIW. São usados para checar se a instrução ainda possui espaço disponível para receber mais um operando do tipo especificado durante o processo de codificação. Devem ser usadas em conjunto com o método `hasOperandSlot()` para facilitar o desenvolvimento do codificador.

```
virtual bool hasOperandSlot(const PBIW::Utils::OperandItemDTO&) = 0;
```

Recebe um objeto de transferencia de dados (*Data Transfer Object*) contendo um ponteiro para o operando não codificado e codificado em PBIW. O operando não codificado é necessário para prover mais informações ao algoritmo de codificação PBIW. A versão codificada do mesmo é para ser adicionada internamente na instrução PBIW.

```
virtual std::string getLabelDestiny() const = 0;
virtual void setLabelDestiny(std::string) = 0;
```

Métodos *getter* e *setter* para retornar ou definir, respectivamente, o rótulo da instrução PBIW destino apontado por esta instrução PBIW.

```
virtual void setBranchDestiny(const IPBIWInstruction& branchDestiny) = 0;
virtual IPBIWInstruction* getBranchDestiny() const = 0;
```

Métodos *getter* e *setter* para retornar ou definir, respectivamente, a instrução PBIW destino apontado por esta instrução PBIW.

```
virtual int readOperandQuantity() const = 0;
virtual int writeOperandQuantity() const = 0;
```

Retornam a quantidade de operandos de leitura e escrita presentes nesta instrução PBIW, respectivamente.

```
virtual void print(IPBIWPrinter&) const = 0;
```

Imprime a instrução PBIW atual utilizando a impressora passada como parâmetro.

```
struct OperandComparer {
  bool operator() (const IOperand& lhs, const IOperand& rhs) const
  { return lhs < rhs; }
};
```

```
virtual PBIW::Utils::OperandVector getOperands() const = 0;
```

Retorna o conjunto de operandos presentes nesta instrução PBIW.

```
virtual void
  setOperationReferences(const std::list<GenericAssembly::Interfaces::IOperation*>&) = 0;

virtual std::list<GenericAssembly::Interfaces::IOperation*>
  getOperationReferences() const = 0;
```

Métodos *getter* e *setter* para refinir quais foram as operações codificadas por esta instrução PBIW.

```
virtual const AnnulationBits& getAnnulBits() const = 0;
virtual void setAnnulBit(int, bool) = 0;
virtual void setAnnulBits(const AnnulationBits& vectorBits) = 0;
```

Métodos *getter* e *setter* para realizarem o retorno do vetor de bits de cancelamento (ou anulação), a definição do valor de um bit de cancelamento específico ou a atribuição/cópia de todos bits de execução para a instrução PBIW.

```
virtual void updateAnnulBits(int index1, int index2) = 0;
```

Troca os bits de anulação nos índices especificados: o índice 1 é trocado pelo índice 2. Ao realizar a troca, as respectivas operações no padrão apontado pela instrução PBIW tambem tem suas posições trocadas.

B.6 IPBIWPattern

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIWInstruction;
    class IOperation;
    class IPBIWPrinter;

    class IPBIWPattern
    {
      public:
        typedef std::vector<IOperation*> OperationVector;

        virtual ~IPBIWPattern() {}

        virtual IPBIWPattern* clone() const = 0;

        virtual void setAddress(unsigned int) = 0;
        virtual unsigned int getAddress() const = 0;

        virtual void addOperation(IOperation*) = 0;

        virtual unsigned int getOperationCount() const = 0;
        virtual OperationVector getOperations() const = 0;

        virtual IOperation* getOperation(unsigned int) const = 0;
        virtual void setOperation(IOperation&, int) = 0;

        virtual IOperation* operator[](const unsigned int) const = 0;
```

```
        virtual bool hasControlOperation() const = 0;

        virtual void updateIndexes(int oldIndex, int newIndex) = 0;

        virtual void referencedByInstruction(IPBIWInstruction*) = 0;

        virtual std::deque<IPBIWInstruction*> getInstructionsThatUseIt() = 0;

        virtual void reorganize() = 0;
        virtual void reorganize(bool) = 0;

        virtual int getUsageCounter() const = 0;

        virtual void resetUsageCounter() = 0;

        virtual void print(IPBIWPrinter&) const = 0;

        virtual bool operator<(const IPBIWPattern&) const = 0;
        virtual bool operator>(const IPBIWPattern&) const = 0;
        virtual bool operator<=(const IPBIWPattern&) const = 0;
        virtual bool operator>=(const IPBIWPattern&) const = 0;

        virtual bool operator==(const IPBIWPattern&) const = 0;
        virtual bool operator!=(const IPBIWPattern&) const = 0;
    };
  }
}
```

```
virtual void setAddress(unsigned int) = 0;
virtual unsigned int getAddress() const = 0;
```

Métodos getter e setter do endereço deste padrão PBIW.

```
virtual void addOperation(IOperation*) = 0;
```

Adiciona uma operação as presentes no padrão.

```
virtual unsigned int getOperationCount() const = 0;
```

Retorna o número de operações presentes neste padrão.

```
virtual OperationVector getOperations() const = 0;
```

Retorna um vetor contendo ponteiros para as operações presentes neste vetor.

```
virtual IOperation* getOperation(unsigned int) const = 0;
virtual void setOperation(IOperation&, int) = 0;
```

Métodos getter e setter de uma operação em uma posição específica dentro do padrão.

```
virtual IOperation* operator[](const unsigned int) const = 0;
```

Retorna um ponteiro para uma operação presente no índice passado como argumento.

```
virtual bool hasControlOperation() const = 0;
```

Informa se este padrão contém alguma operação de controle que desvia o fluxo de código, como uma instrução de branch (in)condicional.

```
virtual void updateIndexes(int oldIndex, int newIndex) = 0;
```

Troca, em todas as operações do padrão, o índice de valor informado por um novo índice. Por exemplo: ao atualizar o índice 5 para 12, se no padrão existem 2 operações que fazem referência ao índice 5 da instrução PBIW, essas duas operações terão esse índice 5 atualizado para o valor 12.

```
virtual void referencedByInstruction(IPBIWInstruction*) = 0;
```

Informa ao padrão que ele é referenciado por uma instrução. A semântica deste comportamento é tornar o padrão ciente de quais instruções o referenciam.

```
virtual std::deque<IPBIWInstruction*> getInstructionsThatUseIt() = 0;
```

Retorna uma coleção de ponteiros para as instruções PBIW que referenciam este padrão. É o método "getter" equivalente ao "setter" `referencedByInstruction(IPBIWInstruction*)`.

```
virtual void reorganize() = 0;
virtual void reorganize(bool) = 0;
```

A primeira versão reorganiza (ou reordena) as operações do padrão (de acordo com as limitações arquiteturais presentes na máquina alvo) e, como consequência, também atualiza os bits de cancelamento das instruções que apontam para este padrão. A segunda versão do método (com o argumento booleano) torna explícita a escolha de atualizar ou não os bits de cancelamento.

```
virtual int getUsageCounter() const = 0;
```

Retorna a quantidade de instruções que apontam para este padrão.

```
virtual void resetUsageCounter() = 0;
```

Limpa a lista de instruções que apontam para este padrão.

```
virtual void print(IPBIWPrinter&) const = 0;
```

Imprime este padrão usando a impressora especificada.

```
virtual bool operator<(const IPBIWPattern&) const = 0;
virtual bool operator>(const IPBIWPattern&) const = 0;
virtual bool operator<=(const IPBIWPattern&) const = 0;
virtual bool operator>=(const IPBIWPattern&) const = 0;

virtual bool operator==(const IPBIWPattern&) const = 0;
virtual bool operator!=(const IPBIWPattern&) const = 0;
```

Operadores de igualdade devem ser implementados para facilitar a legibilidade do código quando realizada a comparação do conteúdo de dois padrões.

Os operadores maior e menor ajudam a facilitar o trabalho de busca de padrões durante o processo de codificação PBIW por proverem uma ordenação aos padrões. O algoritmo de codificação terá sua complexidade reduzida de forma significativa se usada em uma coleção que realize busca binária.

B.7 IOperation

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIWInstruction;

    class IOperation
    {
    public:
      virtual ~IOperation() {}
```

```
      virtual IOperation* clone() const = 0;

      virtual void setOpcode(unsigned short opcode) = 0;
      virtual unsigned short getOpcode() const = 0;

      virtual void addOperand(const IOperand&) = 0;

      typedef std::vector<int> OperandIndexVector;
      virtual OperandIndexVector getOperandsIndexes() const = 0;

      virtual void updateIndexes(int oldIndex, int newIndex) = 0;

      virtual void setType(int) = 0;
      virtual int getType() const = 0;

      virtual bool operator<(const IOperation&) const = 0;
      virtual bool operator>(const IOperation&) const = 0;
      virtual bool operator<=(const IOperation&) const = 0;
      virtual bool operator>=(const IOperation&) const = 0;

      virtual bool operator==(const IOperation&) const = 0;
      virtual bool operator!=(const IOperation&) const = 0;

    };
  }
}
```

```
virtual void setOpcode(unsigned short opcode) = 0;
virtual unsigned short getOpcode() const = 0;
```

Métodos getter e setter para definição do opcode.

```
virtual void addOperand(const IOperand&) = 0;
```

Adiciona um operando na operação atual.

```
typedef std::vector<int> OperandIndexVector;
virtual OperandIndexVector getOperandsIndexes() const = 0;
```

Retorna um vetor de índices dos operandos da operação atual.

```
virtual void updateIndexes(int oldIndex, int newIndex) = 0;
```

Troca, em todos os operandos da operação, o índice de valor informado por um novo índice. Por exemplo: ao atualizar o índice 5 para 12, se na operação existem 2 operandos que fazem referência ao índice 5 da instrução PBIW, esses dois operandos terão esse índice 5 atualizado para o valor 12.

```
virtual void setType(int) = 0;
virtual int getType() const = 0;
```

Métodos getter e setter para definição do tipo da operação. No caso do processador ρ-VEX , por exemplo, as operações podem ser dos tipos ALU, MEM, MUL e CTRL.

```
virtual bool operator<(const IOperation&) const = 0;
virtual bool operator>(const IOperation&) const = 0;
virtual bool operator<=(const IOperation&) const = 0;
virtual bool operator>=(const IOperation&) const = 0;

virtual bool operator==(const IOperation&) const = 0;
virtual bool operator!=(const IOperation&) const = 0;
```

Operadores de igualdade devem ser implementados para facilitar a legibilidade do código quando realizada a comparação do conteúdo de duas operações. Os operadores maior e

menor ajudam a facilitar o trabalho de busca de operações durante o processo de codificação PBIW por proverem uma ordenação as operações internamente a um padrão.

B.8 IOperand

```
namespace PBIW
{
  namespace Interfaces
  {
    class IOperand
    {
    public:
      virtual ~IOperand() {}

      virtual IOperand* clone() const = 0;

      virtual void setIndex(unsigned int) = 0;
      virtual unsigned int getIndex() const = 0;

      virtual bool isImmediate() const = 0;

      virtual void setTypeCode(int) = 0;
      virtual int getTypeCode() const = 0;

      virtual void setValue(int) = 0;
      virtual int getValue() const = 0;

      virtual bool operator<(const IOperand&) const = 0;
      virtual bool operator>(const IOperand&) const = 0;
      virtual bool operator<=(const IOperand&) const = 0;
      virtual bool operator>=(const IOperand&) const = 0;

      virtual bool operator==(const IOperand&) const = 0;
      virtual bool operator!=(const IOperand&) const = 0;
    };
  }
}
```

```
virtual void setIndex(unsigned int) = 0;
virtual unsigned int getIndex() const = 0;
```

Métodos getter e setter para o índice do operando.

```
virtual bool isImmediate() const = 0;
```

Informa se o operando é um valor imediato.

```
virtual void setTypeCode(int) = 0;
virtual int getTypeCode() const = 0;
```

Métodos getter e setter para o código do tipo do operando.

```
virtual void setValue(int) = 0;
virtual int getValue() const = 0;
```

Métodos getter e setter para o valor do operando.

```
virtual bool operator<(const IOperand&) const = 0;
virtual bool operator>(const IOperand&) const = 0;
virtual bool operator<=(const IOperand&) const = 0;
virtual bool operator>=(const IOperand&) const = 0;

virtual bool operator==(const IOperand&) const = 0;
virtual bool operator!=(const IOperand&) const = 0;
```

Operadores de comparação.

B.9 IPBIWPrinter

```
namespace PBIW
{
  namespace Interfaces
  {
    class IPBIWPattern;
    class IPBIWInstruction;

    class IPBIWPrinter
    {
    public:
      virtual ~IPBIWPrinter() { }

      virtual void printInstructionsHeader() = 0;
      virtual void printInstruction(const IPBIWInstruction&, const std::vector<unsigned int>&) = 0;
      virtual void printInstructionsFooter(unsigned int) = 0;

      virtual void printPatternsHeader() = 0;
      virtual void printPattern(const IPBIWPattern&, const std::vector<unsigned int>&) = 0;
      virtual void printPatternsFooter(unsigned int) = 0;

      virtual std::ostream& getOutputStream() = 0;
    };
  }
}
```

```
virtual void printInstructionsHeader() = 0;
virtual void printInstruction(const IPBIWInstruction&, const std::vector<unsigned int>&) = 0;
virtual void printInstructionsFooter(unsigned int) = 0;
```

Imprime cabeçalho, conteúdo e rodapé do arquivo de instruções PBIW. No método de impressão da instrução, são recebidos como parametros uma referencia para a instrução a ser impressa e uma referencia ao vetor contendo sua representação binária.

```
virtual void printPatternsHeader() = 0;
virtual void printPattern(const IPBIWPattern&, const std::vector<unsigned int>&) = 0;
virtual void printPatternsFooter(unsigned int) = 0;
```

Imprime cabeçalho, conteúdo e rodapé do arquivo de padrões PBIW. No método de impressão do padrão, são recebidos como parametros uma referencia para o padrão a ser impresso e uma referencia ao vetor contendo sua representação binária.

```
virtual std::ostream& getOutputStream() = 0;
```

Retorna uma referencia ao stream de saída usado por esta impressora.

B.10 IPBIWOptimizer

```
namespace PBIW
{
 namespace Interfaces
 {
  class IPBIWOptimizer : public IPBIWSet
  {
    public:
      virtual ~IPBIWOptimizer() {}

      virtual void useContext(const IPBIW&) = 0;

      virtual void useLabels(const std::deque<ILabel*>&) = 0;
```

```
      virtual void useInstructions(const std::deque<IPBIWInstruction*>&) = 0;
      virtual void usePatterns(const std::deque<IPBIWPattern*>&) = 0;

      virtual void setupOptimizer() = 0;

      virtual void run(IPBIWFactory&) = 0;
  };
 }
}
```

```
virtual void useLabels(const std::vector<ILabel*>&) = 0;
virtual void useInstructions(const std::vector<IPBIWInstruction*>&) = 0;
virtual void usePatterns(const std::vector<IPBIWPattern*>&) = 0;
```

Realiza uma cópia profunda completa dos conjuntos de rótulos, instruções e padrões PBIW para uso interno do otimizador.

```
virtual void setupOptimizer() = 0;
```

Atualiza os ponteiros internos aos conjuntos de rótulos, instruções e padrões copiados para apontar às suas novas versões.

```
virtual void run(IPBIWFactory&) = 0;
```

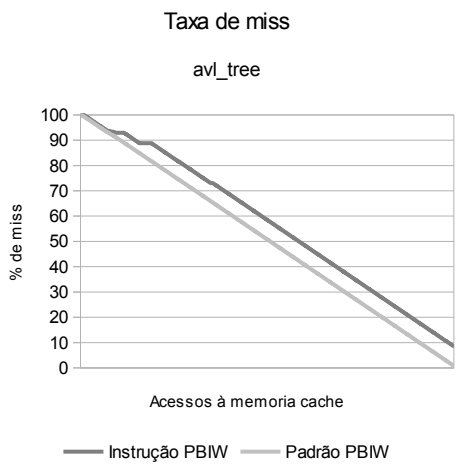
Executa a otimização utilizando uma fábrica PBIW passada como referência para a criação dos novos padrões, instruções ou rótulos necessários ao algorítmo de otimização utilizado.

Apêndice C

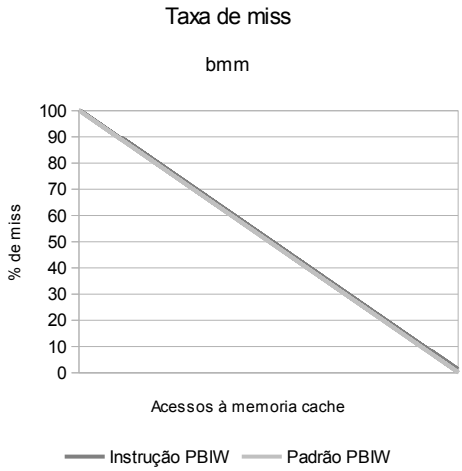
Gráficos para taxa de miss

Os gráficos contidos neste apêndice são referentes ao comportamento dinâmico da execução dos treze benchmarks utilizados neste trabalho.

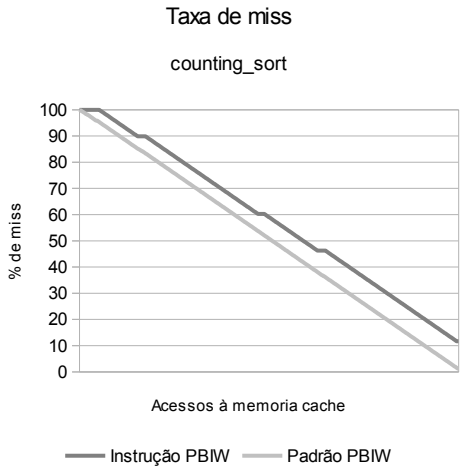
C.1 PBIW Versão 1.0



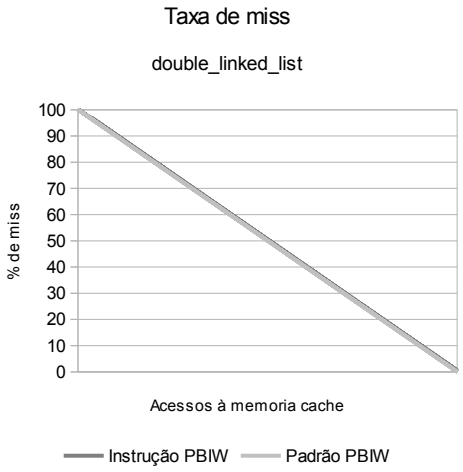
(a) avl_tree



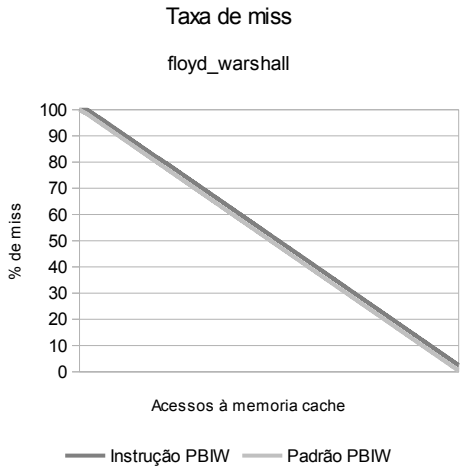
(b) bmm



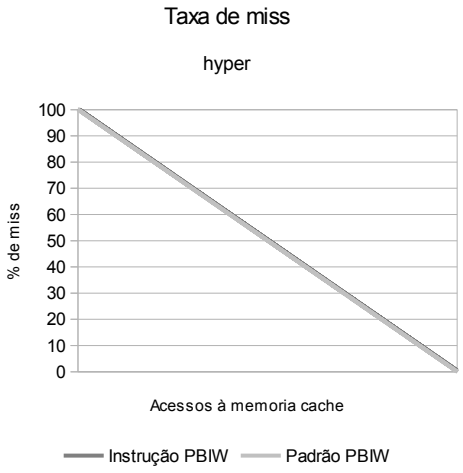
(c) counting_sort



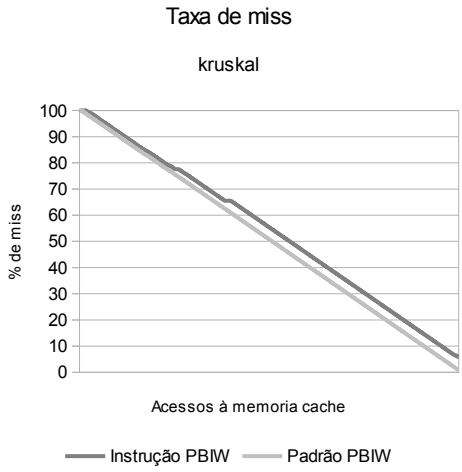
(d) double_linked_list



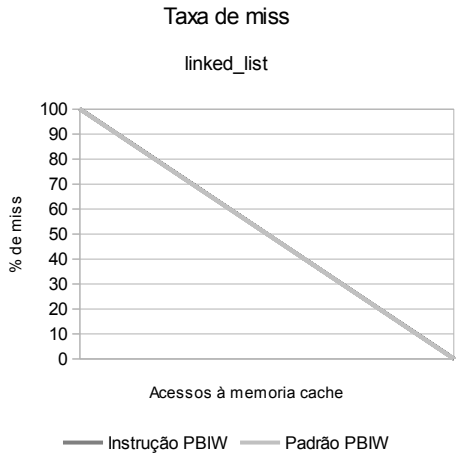
(e) floyd_warshall



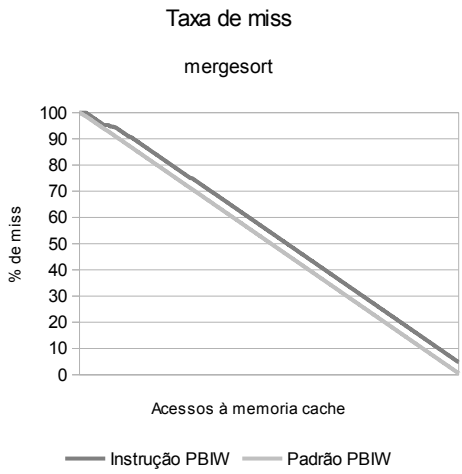
(f) hyper



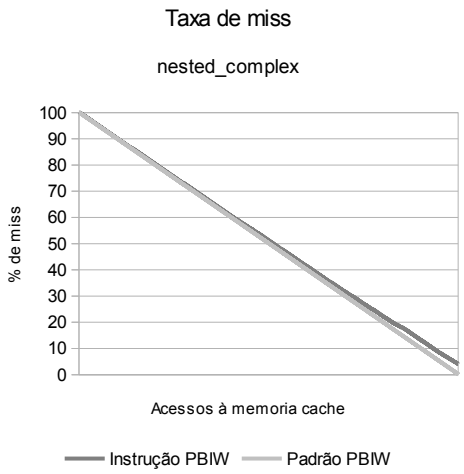
(g) kruskal



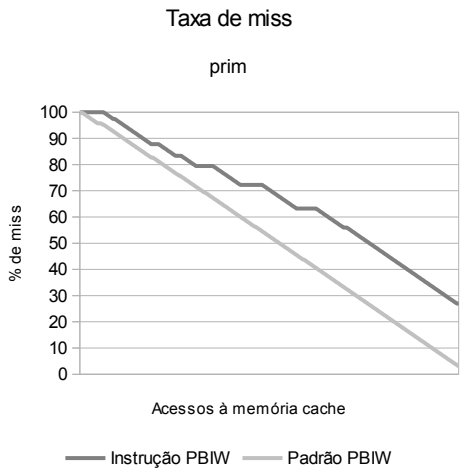
(h) linked_list



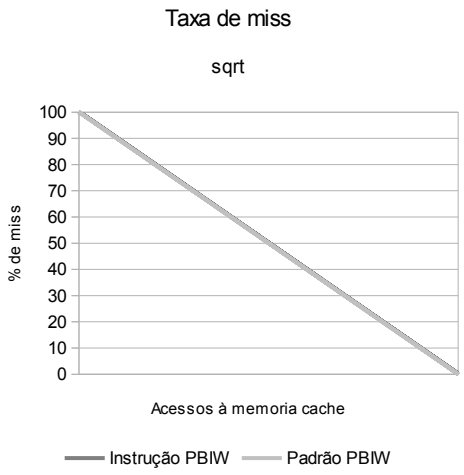
(i) mergesort



(j) nested_complex



(k) prim



(l) sqrt

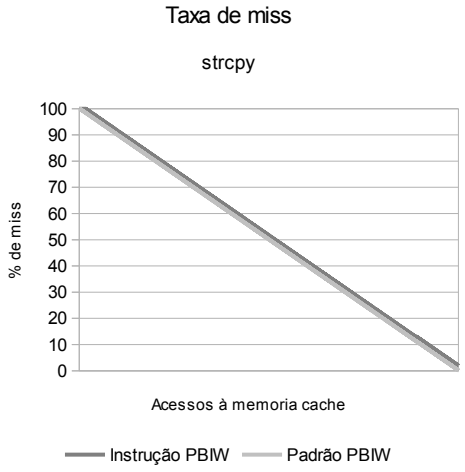
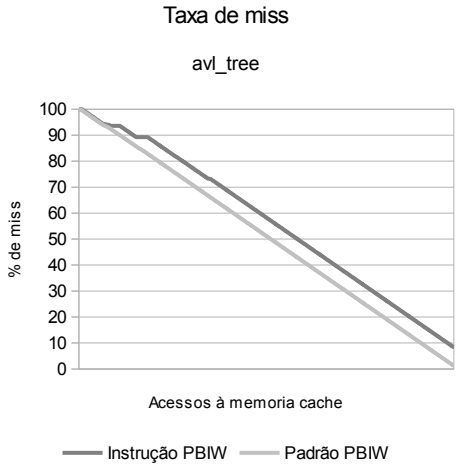
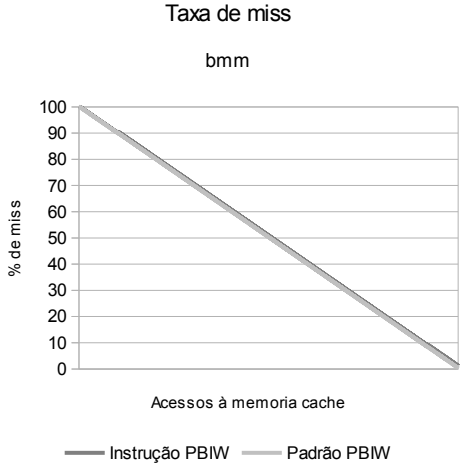


Figura C.1: Taxa de miss para Programas PBIW Versão 1.0

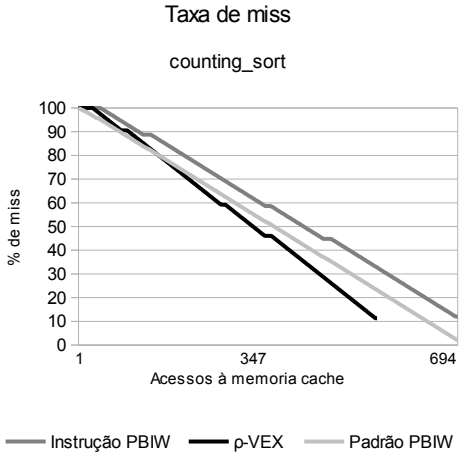
C.2 PBIW Versão 2.0



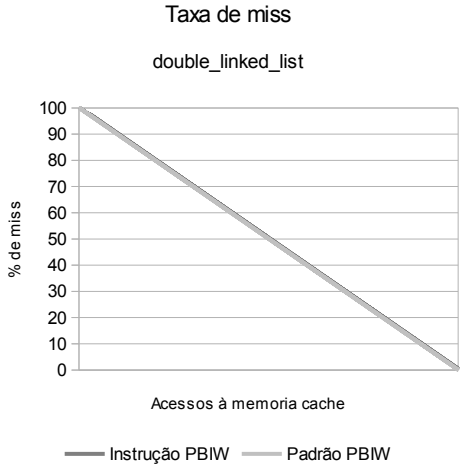
(a) avl_tree



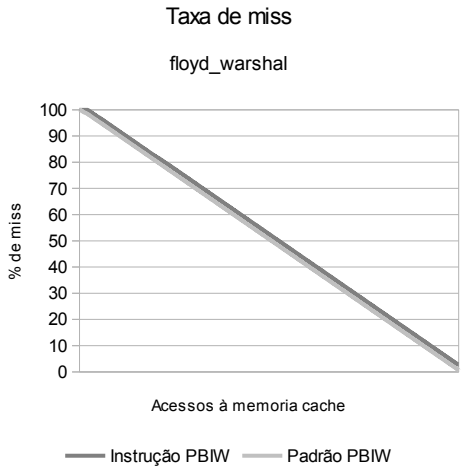
(b) bmm



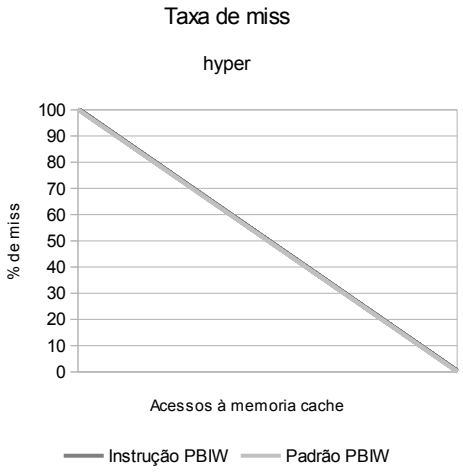
(c) counting_sort



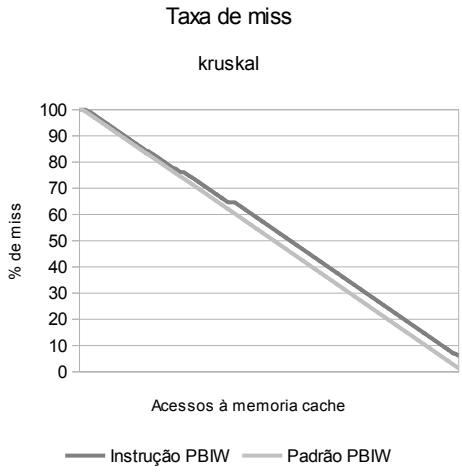
(d) double_linked_list



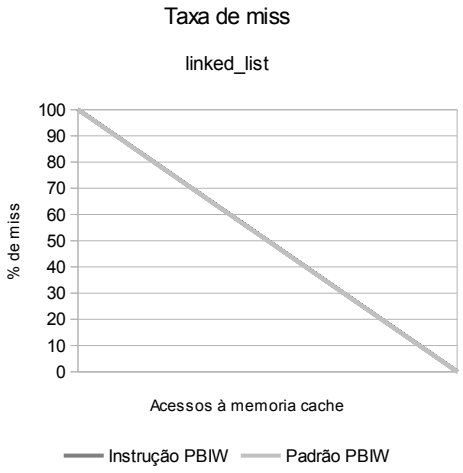
(e) floyd_warshall



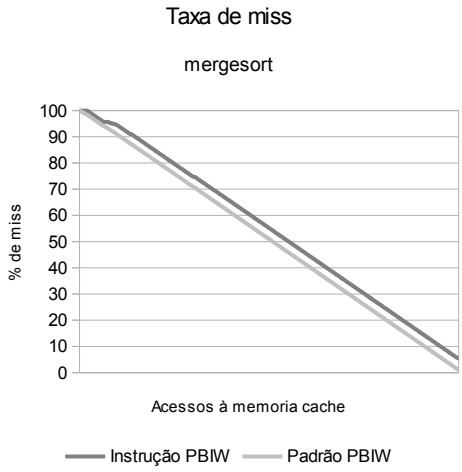
(f) hyper



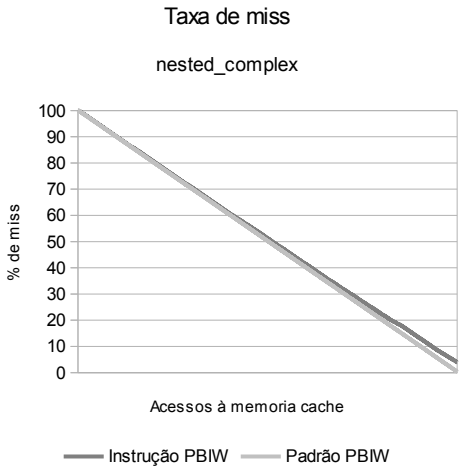
(g) kruskal



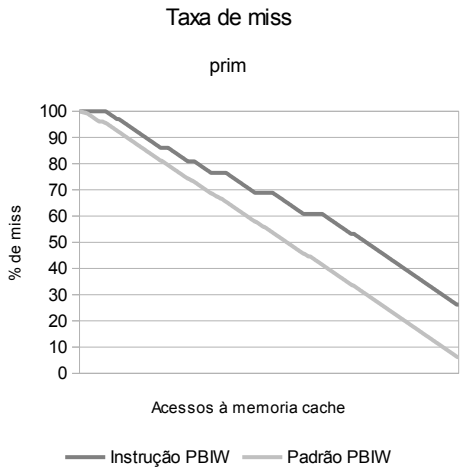
(h) linked_list



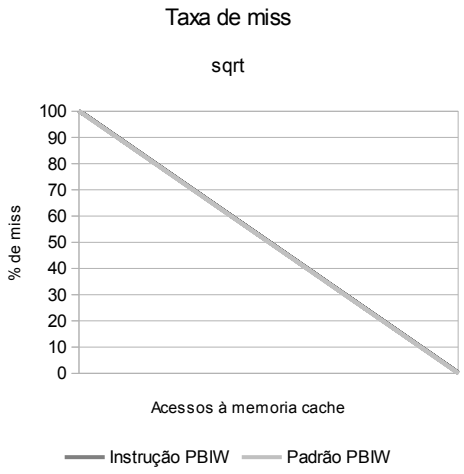
(i) mergesort



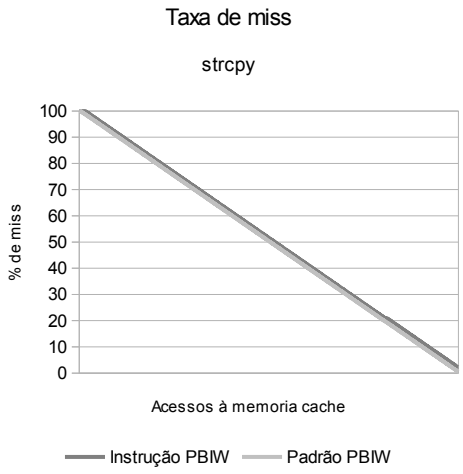
(j) nested_complex



(k) prim



(l) sqrt



(m) strcpy

Figura C.2: Taxa de miss para Programas PBIW Versão 2.0

Apêndice D

SPEC - Resultados Estáticos Preliminares

Neste apêndice serão mostrados os resultados estáticos da codificação PBIW para programas do benchmark SPECINT2000 [39] compilados para o processador ρ-VEX . Os dados estáticos coletados são preliminares pois não houve tempo hábil para a validação da inexistência de erros de codificação dos programas. Por esse motivo, dois programas — 164.gzip e 175.vpr — não conseguiram ser codificados e/ou otimizados em alguma das versões da codificação PBIW desenvolvidas para ρ-VEX. Entretanto, os valores apresentados podem ser levados em consideração como uma estimativa de valores reais visto que os programas codificados são grandes.

Ao codificar os programas do benchmark SPECINT2000 em PBIW na versão 1.0, nota-se a partir da Figura D.1 uma melhoria no tamanho final de aproximadamente 36% (ou uma taxa de compressão de 64%) em média para os programas otimizados com junção de padrões. Sem junção de padrões obteve-se uma melhoria no tamanho final de aproximadamente 31% (ou uma taxa de compressão de aproximadamente 69%). Para a versão 2.0 os valores são ainda mais expressivos: uma melhoria no tamanho final de aproximadamente 42% (ou uma taxa de compressão de 58%) em média para os programas otimizados com junção de padrões. Sem junção de padrões obteve-se uma melhoria no tamanho final de aproximadamente 40% (ou uma taxa de compressão de aproximadamente 60%). Assim nota-se que programas maiores oferecem oportunidades e taxas de compressão muito maiores que programas pequenos.

Na Figura D.2 é possível notar que a taxa de reúso de padrões aumenta de forma expressiva: no programa 255.vortex a taxa de reúso de padrões na codificação 1.0 com junção de padrões atingiu mais de 250 instruções por padrão. No caso do programa 176.gcc, as codificações 1.0 e 2.0 com junção de padrões obtiveram uma taxa de reúso próxima a 175 instruções por padrão. Valores muito significativos que poderão trazer resultados interessantes quando experimentos dinâmicos forem realizados.

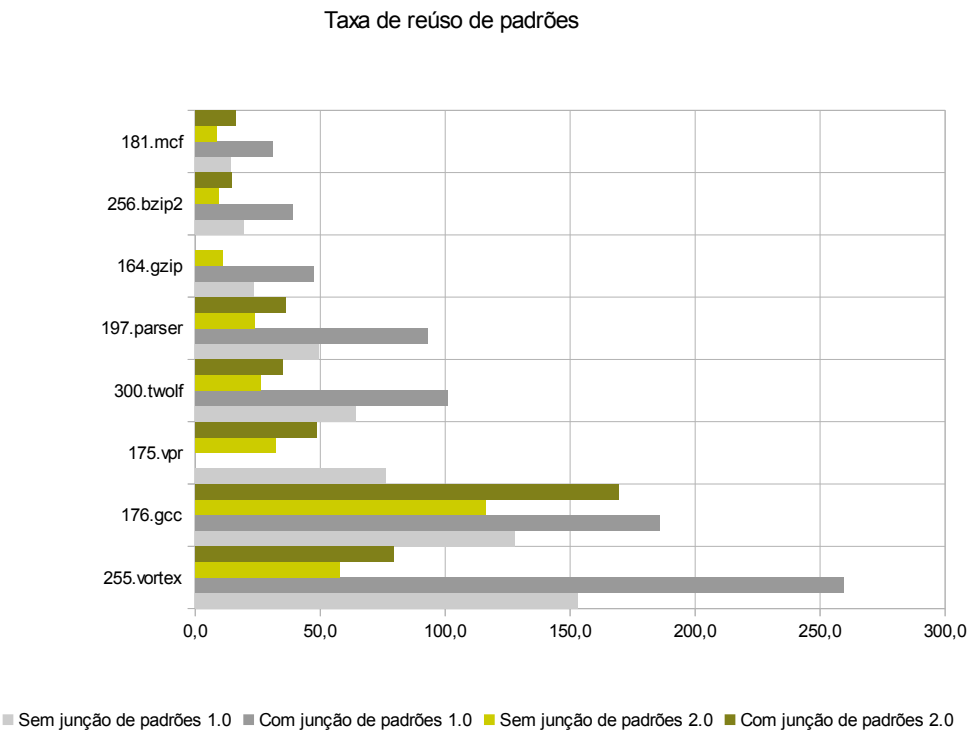


Figura D.1: Taxa melhoria do tamanho dos programas SPECINT2000 codificados.

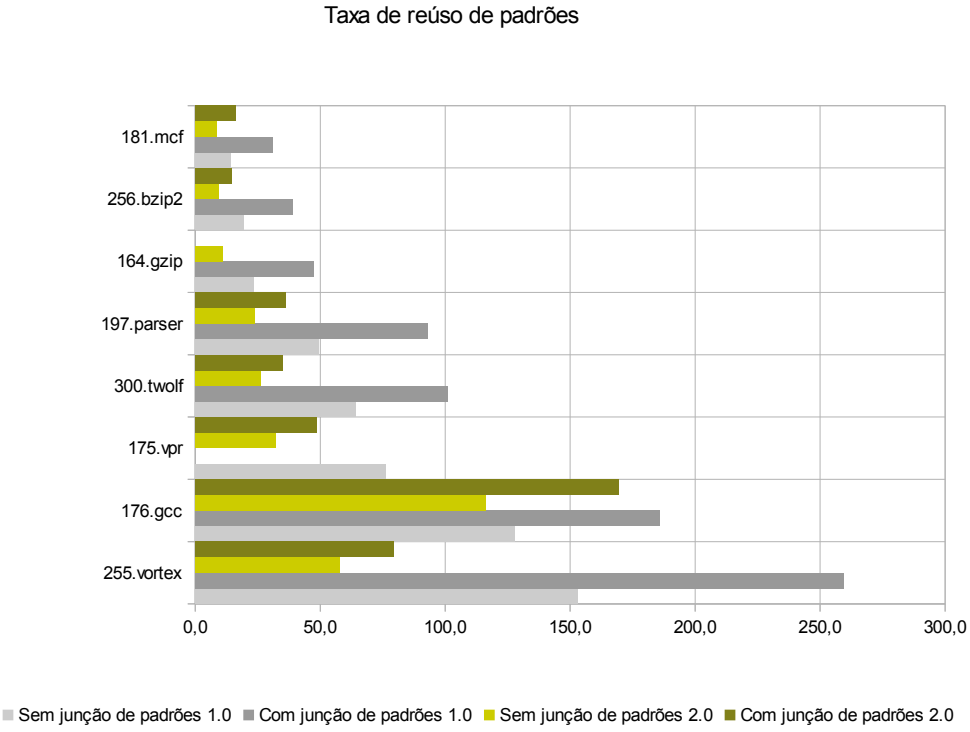


Figura D.2: Taxa de reúso de padrões PBIW para programas SPECINT2000 codificados.