PBIW: Uma Codificação de Instruções Alternativa para Arquiteturas de Alto Desempenho

Rafael Fernandes Batistella
Universidade Estadual de Campinas
Instituto de Computação
Avenida Albert Einstein, 1251 - Campinas, SP
rafael.batistella@ic.unicamp.br

Ricardo Ribeiro dos Santos
Universidade Católica Dom Bosco
Grupo de Pesquisa em Engenharia e Computação
Av. Tamandaré, 6000 - Campo Grande, MS
ricr.santos@ucdb.br

Rodolfo Jardim de Azevedo
Universidade Estadual de Campinas
Instituto de Computação
Avenida Albert Einstein, 1251 - Campinas, SP
rodolfo@ic.unicamp.br

Resumo

Este artigo apresenta a técnica de codificação PBIW. Essa técnica é baseada na fatoração de grupos de operações escalonadas em instruções codificadas e padrões. Uma instrução codificada não contém dados redundantes e é armazenada em uma cache de instruções. Os padrões são armazenados em uma cache de padrões. Foi realizado um estudo de caso desta técnica sobre esquemas de codificação de instruções denominados 2D-VLIW e EPIC em uma arquitetura de alto desempenho chamada 2D-VLIW. A técnica PBIW foi avaliada com os benchmarks MediaBench, SPECint e SPECfp. Os resultados revelam que a técnica PBIW produz programas até 81% menores que 2D-VLIW e até 46% menores que EPIC, além de programas até 96% mais rápidos que 2D-VLIW e até 69% mais rápidos que EPIC.

1 Introdução

Trabalhos não muito recentes comprovaram que o aumento na velocidade das memórias não acompanha o aumento na velocidade dos processadores. Como resultado, a diferença entre a velocidade do processador e da memória cresce exponencialmente levando a um fenômeno conhecido como *Memory Wall* [8].

Várias técnicas foram desenvolvidas para minimizar esse problema e boa parte delas foram aplicadas em arquiteturas que buscam instruções longas na memória ou em arquiteturas com limitação de memória, como os sistemas embarcados [9, 11]. Este artigo propõe uma nova abordagem para lidar com o *overhead* na busca de instruções em memória e seu impacto no desempenho dos programas em arquiteturas com instruções longas. A abordagem proposta é denominada PBIW (*Pattern-Based Instruction Word*) [2] e é composta por um algoritmo de codificação baseado na técnica de fatoração de operandos [1] e uma memória cache, chamada de *Pattern Cache* (P-cache). O algoritmo é responsável pela fatoração de instruções originais do programa em instruções codificadas PBIW e padrões. As instruções codificadas PBIW são armazenadas em uma I-cache e os padrões obtidos são armazenados na P-cache. A técnica PBIW foi avaliada através de experimentos com os *benchmarks* MediaBench, SPECint e SPECfp, implementando a codificação PBIW sobre uma arquitetura de alto desempenho conhecida como 2D-VLIW [12]. Os resultados revelam que a técnica PBIW produz programas até 81% menores e até 96% mais rápidos que 2D-VLIW além de até 46% menores e até 69% mais rápidos que EPIC [14].

A Seção 2 apresenta os trabalhos relacionados. Na Seção 3 são apresentados todos os conceitos relacionados à técnica PBIW. Um estudo de caso é apresentado na Seção 4. A Seção 5 mostra os resultados da técnica PBIW. Finalmente, a Seção 6 apresenta as conclusões.

2 Trabalhos Relacionados

Trabalhos anteriores sobre redução de código utilizaram conceitos e técnicas da área de compressão de código. Existem várias propostas, como em [10], descrevendo técnicas de compressão com foco em arquiteturas VLIW.

Trabalhos como em [3] mostraram que é possível obter redução no tamanho do código e no consumo de energia através de uma compressão de código eficiente e um descompressor bastante simples.

Em [10] é apresentada uma técnica de compressão baseada em dicionário que utiliza isomorfismo de instrução. Sua abordagem consiste em selecionar as instruções mais freqüentes e quebrá-las em *opcodes* e operandos, armazenando-os separadamente em dois dicionários. A lógica de decodificação acrescenta um novo estágio no *datapath* do processador, ao contrário da técnica PBIW que permite que a decodificação ocorra em paralelo a outras atividades do *datapath*.

A abordagem utilizada em PBIW difere de estratégias que reduzem o número de instruções como a técnica *Instruction Collapsing* [13], onde as instruções dependentes são agrupadas em uma única instrução e cujos experimentos mostram aumentos de até 20% no IPC. A técnica PBIW explora a sobrejeção entre instruções e seus padrões (reuso de padrões) com o objetivo de reduzir o tamanho da instrução em memória e o tamanho da cache de padrões.

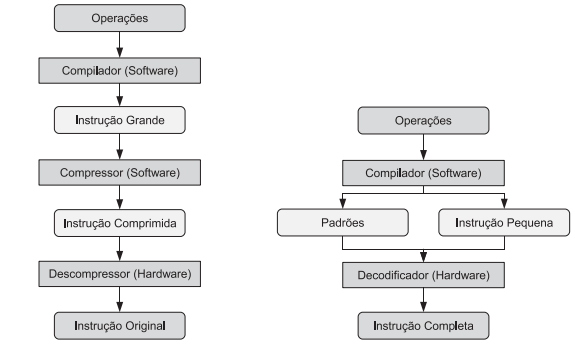
A técnica PBIW fatora as operações de um programa em instruções (conjunto de operações) codificadas e padrões. Diferente da abordagem em [1], a técnica PBIW armazena padrões fatorados em uma cache que pode ser acessada em paralelo à execução de outras atividades no *datapath*. A preparação da instrução utilizada nos estágios de execução é mais simples do que alguns mecanismos tradicionais de descompressão pois a instrução na memória possui uma referência para seu respectivo padrão na P-cache.

3 Esquema de Codificação PBIW

Nas técnicas convencionais de compressão de código, uma instrução grande é comprimida e passa a ser representada em memória por uma instrução pequena. Antes do estágio de execução, a instrução pequena volta a ser exatamente a mesma instrução grande inicial (Figura 1(a)). Na codificação PBIW a instrução inicial é pequena e seus dados são utilizados para complementar os dados armazenados em uma cache de padrões e assim formarem a instrução que será utilizada nos estágios de execução (Figura 1(b)).

Como em técnicas de compressão baseadas em fatoração de operandos [1], o algoritmo da técnica PBIW percorre as operações do programa extraindo padrões a partir de operandos redundantes. O fundamento da técnica PBIW é a sobrejeção intrínseca entre instruções e padrões. Pesquisas anteriores [5] descobriram que a função que mapeia o conjunto de instruções de um programa *CI* para o conjunto de padrões dessas instruções *CP* obedece ao comportamento de uma função sobrejetora.

Ao aplicar restrições arquiteturais durante a codificação de uma instrução PBIW, o número de instruções codificadas



(a) Compressão de Código (b) Codificação PBIW

Figura 1. Compressão vs Codificação

pode se tornar maior do que o número de instruções que seriam criadas por outra técnica de codificação. Desta forma, a existência da sobrejeção entre o conjunto de instruções e o conjunto de padrões é uma condição necessária para que essa estratégia possa reduzir a quantidade de memória utilizada por um programa.

Uma instrução PBIW representa, de forma compacta, um conjunto de operações de um programa. Os *opcodes* dessas operações são armazenados no padrão e os operandos (registradores e imediatos) na instrução codificada. Essas instruções codificadas são armazenadas na I-cache. Um padrão é uma estrutura de dados que contém os *opcodes* das operações e ponteiros para as posições da instrução codificada. Os padrões são armazenados na P-cache.

Uma instrução PBIW é composta por quatro conjuntos de informações: registradores cujos valores serão lidos, registradores de destino (escrita) ou valores imediatos, índice para a tabela de padrões em memória e bits que indicam quais unidades funcionais (UFs) de execução devem executar ou anular suas operações.

Um padrão é composto por C operações, onde C representa o número total de operações que estão agrupadas em uma instrução na memória. Cada operação é formada por um *opcode* e um número fixo de operandos. Com exceção dos *opcodes*, todos os outros campos (operandos) do padrão são ponteiros para campos da instrução codificada, ou seja, eles armazenam o índice referente ao respectivo campo da instrução.

A *Pattern Cache* é uma memória cache que armazena os padrões detectados durante a codificação das instruções PBIW. Durante o estágio de decodificação, os dados da P-cache são complementados com os dados da instrução codificada em memória para compor a instrução utilizada no estágio de execução.

A codificação de uma instrução para o esquema PBIW ocorre durante a fase final de compilação do programa. O compilador percorre todas as operações da instrução e, ba-

seado no conjunto de operandos utilizados, cria a instrução codificada e o respectivo padrão. O algoritmo não verifica dependências de dados entre as instruções pois elas estão diretamente associadas à arquitetura e, por isso, devem ser resolvidas estaticamente (pelo compilador) ou dinamicamente (pelo hardware).

O Algoritmo 1 apresenta todos os passos do processo de codificação PBIW. A entrada para o algoritmo é o conjunto de operações (CO) obtido após as fases de escalonamento e alocação de registradores. Observe que para compiladores que geram código para processadores que utilizam instruções longas, CO pode ser composto por instruções que agrupam, por sua vez, operações simples. Dois conjuntos estão disponíveis na saída: o conjunto de instruções codificadas (CI) e o conjunto de padrões (CP).

Algoritmo 1 Algoritmo de codificação

ENTRADA: Conjunto de operações CO.
SAÍDA: Conjunto de instruções codificadas CI e padrões CP.
Codifica(CONJUNTO_OPERACOES: CO**)**
1. **Para** $O \in CO$
2. cria novo I, cria novo P;
3. **Para** $op \in O$
4. $P.add(op.opcode)$;
5. **Para** $opnd \in op$
6. **Se** $op.opnd \notin I$
7. **Se** $espaco\_livre(I) < 1$
8. **Se** $P \notin CP$
9. $CP = CP \cup P$;
10. **Fim Se**
11. $I.add$(índice de P em CP);
12. $CI = CI \cup I$;
13. cria novo I, cria novo P;
14. **Fim Se**
15. $I.add(op.opnd)$;
16. **Fim Se**
17. $P.add$(índice de $op.opnd$ em I);
18. **Fim Para**
19. **Fim Para**
20. **Se** $P \notin CP$
21. $CP = CP \cup P$;
22. **Fim Se**
23. $I.add$(índice de P em CP);
24. $CI = CI \cup I$;
25. **Fim Para**

Para cada elemento O de CO (Linha 1), o algoritmo cria uma instrução PBIW vazia I e um padrão vazio P (2). Para cada operação *op* inserida em O é adicionado o respectivo *opcode* no padrão (4). Para cada operando $opnd$ de uma operação *op* é verificado se o operando já existe na instrução codificada I (6). Caso o operando não exista em I é verificado se ainda há espaço livre na instrução codificada (7).

Caso não exista mais espaço na palavra, é verificado se o padrão P já existe no conjunto de padrões CP (8). Se o padrão ainda não existir, então ele é adicionado ao conjunto de padrões (9). Ainda no caso onde não existe mais espaço livre na instrução I, o índice que aponta para o conjunto de padrões é atualizado na instrução com a posição do padrão P nesse conjunto (11). A instrução I passa a fazer parte do conjunto de instruções CI (12) e são criados uma nova instrução I e um novo padrão P (13).

Se existir espaço na palavra, o operando $op.opnd$ é adicionado à instrução (15) e a posição do operando na instrução é inserida no padrão (17). O último padrão criado é adicionado ao conjunto de padrões (21) caso ainda não exista e o índice da instrução que aponta para o padrão no conjunto de padrões é atualizado (23). Finalmente, a instrução codificada I é adicionada ao conjunto de instruções CI (24).

Como o número de operações de cada elemento O, o número de operandos em *op* e o tamanho de I são constantes, a complexidade do algoritmo é limitada pelo tamanho do conjunto CO, $|CO|$, e pela complexidade da consulta ao conjunto de padrões CP. Em uma situação de pior caso $|CP| = |CO|$. Considerando que a complexidade de uma busca no conjunto de padrões possui a mesma complexidade $\mathcal{O}(1)$ de uma busca de chave em uma tabela de dispersão, a complexidade do Algoritmo 1 é $\mathcal{O}(|CO|)$.

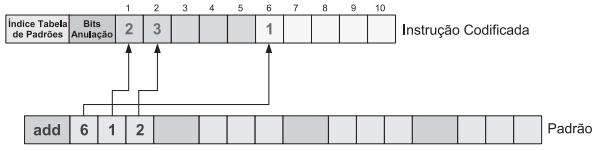
O fragmento de código apresentado na Figura 2 corresponde a quatro operações simples que formam uma instrução de um programa. Este conjunto de operações é usado como base para o exemplo de codificação de instruções PBIW apresentado na Figura 3. As Figuras 3(a) à 3(d) descrevem os passos para criar uma instrução codificada e seu padrão com apenas quatro operações de 32 bits. As setas indicam os operandos utilizados por cada operação inserida no padrão. Neste exemplo, a instrução codificada possui 10 campos. Os campos 1 até 5 são reservados para registradores de leitura e os campos de 6 a 10 para registradores de escrita e imediatos. Foi considerado que um imediato é representado por 5 bits.

```
add r1, r2, r3
addu r4, r2, r6
addi r7, r6, 9
subu r9, r10, r6
```

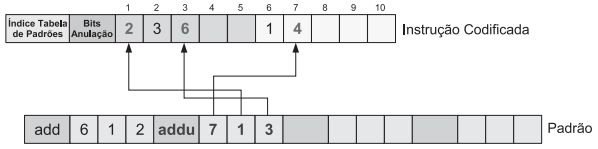
Figura 2. Fragmento de código

A codificação é iniciada pela operação `add r1, r2, r3` (Figura 3(a)). O *opcode* `add` é inserido no padrão e, em seguida, o registrador `r1` (escrita), é armazenado no campo "6" da instrução codificada. Um ponteiro para o campo "6" é armazenado em uma linha da cache de padrões.

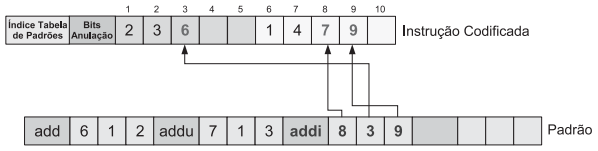
A codificação dos registradores de leitura (`r2` e `r3`) segue o mesmo procedimento, mas utiliza os campos reservados para registradores de leitura. Os mesmos procedimentos são utilizados para a segunda (Figura 3(b)), terceira (Figura 3(c)) e quarta (Figura 3(d)) operações. Depois da codificação da última operação, a informação sobre o endereço do padrão é adicionada à instrução codificada no campo "Índice Tabela de Padrões".



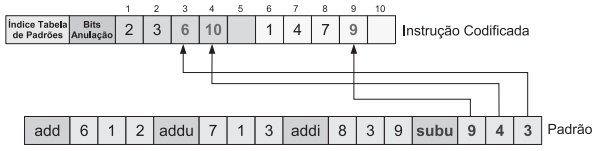
(a) Operação `add r1, r2, r3`



(b) Operação `addu r4, r2, r6`



(c) Operação `addi r7, r6, 9`



(d) Operação `subu r9, r10, r6`

Figura 3. Exemplo de Codificação

Após encerrar o algoritmo de codificação, obtém-se um conjunto de instruções codificadas e um conjunto de padrões, sendo que cada um desses padrões pode ser compartilhado por mais de uma instrução. Porém, alguns destes padrões podem possuir muitos espaços vazios dependendo das restrições arquiteturais aplicadas durante o escalonamento das operações. Com o objetivo de obter um conjunto de padrões menor e mais denso e, conseqüentemente, reduzir o tamanho do programa, utiliza-se uma técnica para unir padrões [2]. A junção de padrões é uma otimização que une dois padrões quando a soma das operações de ambos é menor ou igual a N (número de UFs da arquitetura). Após a junção de dois padrões, as instruções que apontavam para os padrões antigos devem apontar para o novo padrão originado a partir da junção. Mais importante, cada instrução deve anular operações no padrão que não serão utilizadas durante a execução. Assim, deve-se indicar em cada instrução (campo "Bits Anulação"), quais operações serão executadas e quais não serão executadas (anuladas).

No estágio de decodificação, enquanto os registradores globais são lidos, busca-se um padrão de acordo com o campo de endereço presente na instrução codificada. Este campo indica a posição do padrão dentro da cache de padrões. O padrão é buscado na memória caso não se encontre na P-cache.

Depois que o padrão é recuperado da cache de padrões, o hardware de decodificação utiliza os ponteiros que estão armazenados no padrão e monta a instrução utilizada nos estágios de execução. Durante a decodificação, a instrução codificada funciona como um dicionário de operandos para os ponteiros do padrão. Deve-se observar que a leitura dos registradores globais é realizada concomitante com a decodificação da instrução PBIW.

4 Estudo de Caso: Arquitetura 2D-VLIW

A arquitetura 2D-VLIW [12] é uma arquitetura de alto desempenho que utiliza uma codificação semelhante à VLIW composta por operações RISC e possui os estágios de: Busca, Decodificação e L estágios de Execução (L é o número de linhas de uma matriz de execução).

A Figura 4 apresenta uma visão geral da arquitetura 2D-VLIW onde é possível observar o *datapath* e a organização das UFs através de uma matriz de L linhas $\times$ C colunas, onde, neste caso, $L = C = 4$.

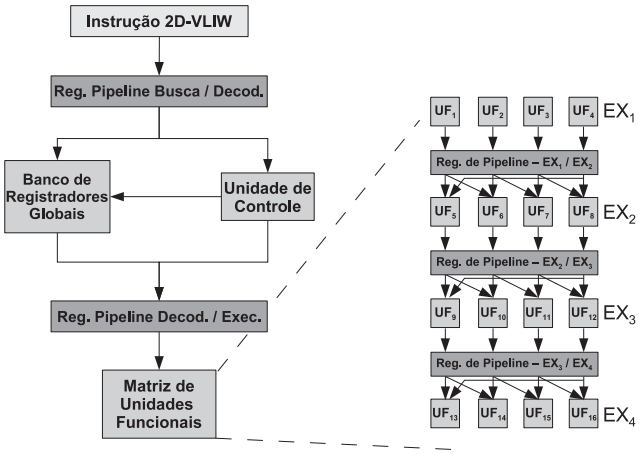


Figura 4. *Datapath* **da arquitetura 2D-VLIW**

As instruções 2D-VLIW são de tamanho fixo e formadas por operações simples (`add, sub, ld, st`, entre

outras). Todas as operações possuem 32 bits e são executadas no interior da matriz de unidades funcionais. O número de operações de uma instrução é equivalente à quantidade de unidades funcionais na matriz. No exemplo da Figura 4, uma instrução 2D-VLIW possui 16 operações e 512 bits (16×32) de tamanho. As instruções 2D-VLIW são compostas por operações dependentes e independentes, sendo que todo o controle para evitar conflitos de dados é gerenciado pelo compilador. A estrutura arquitetural 2D-VLIW permite apenas a leitura de $(2 \times C)$ registradores globais por instrução.

A primeira proposta considerada para a instrução PBIW nesta arquitetura possui 128 bits (Figura 5). Esta instrução é composta por oito campos de cinco bits para os registradores de leitura (40 bits), 13 campos de cinco bits para registradores de escrita e imediatos (65 bits), 8 bits para anulação e 15 bits para índice da tabela de padrões em memória.

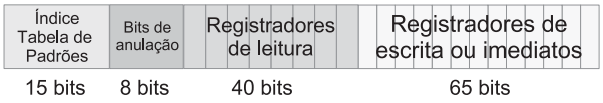


Figura 5. PBIW com 128 bits para 2D-VLIW

O padrão para uma instrução PBIW de 128 bits possui 528 bits divididos em 16 campos de 33 bits. Cada um é dividido em um campo de 8 bits para *opcode* e 5 campos de 5 bits para os operandos: três campos representam os operandos de escrita (1 campo) e de leitura (2 campos) e dois campos são utilizados para representar os imediatos junto com um dos campos de leitura (um imediato 2D-VLIW utiliza 15 bits, então estes dois campos são utilizados juntos à um outro campo de operando). Cada campo de operando aponta para um dos 21 campos existentes na instrução codificada (Figura 6).

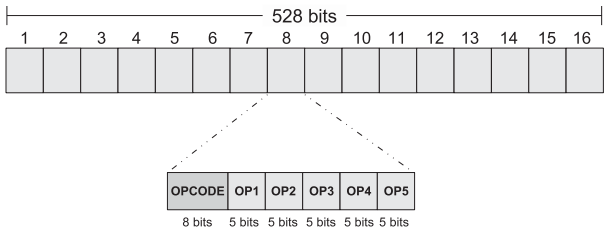


Figura 6. Padrão com 528 bits para 2D-VLIW

O gráfico apresentado na Figura 7 mostra o percentual de instruções 2D-VLIW seria possível representar como instruções codificadas PBIW sem que houvesse necessidade de dividir uma instrução 2D-VLIW em mais de uma instrução PBIW. O eixo x representa a quantidade de campos de possíveis configurações de instruções codificadas PBIW. O eixo y indica a porcentagem de instruções 2D-VLIW de um programa que é representada pelo mesmo número de instruções codificadas PBIW. Após a realização dos primeiros experimentos com a instrução codificada de 128 bits, constatou-se que o número máximo de padrões obtidos é aproximadamente 6.000 (programa 255vortex na Tabela 1).

Da Figura 7 nota-se que com 8 campos, o que equivale a uma instrução PBIW de 64 bits, pode-se representar 88% das instruções 2D-VLIW e com 21 campos (excluindo os campos "Índice Tabela de Padrões" e "Bits de Anulação") (PBIW de 128 bits), em média, poderiam ser representadas todas as instruções 2D-VLIW sem necessidade de qualquer divisão em duas ou mais instruções PBIW.

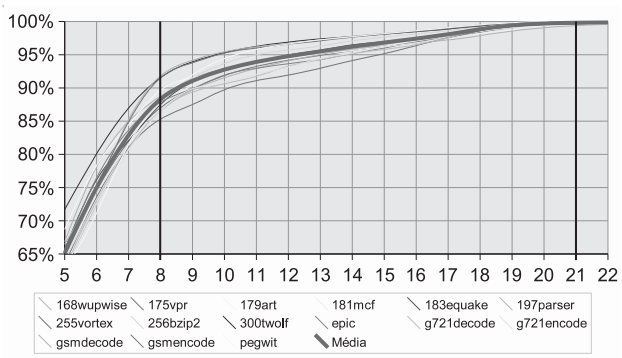


Figura 7. Utilização: campos PBIW 128 bits

A nova instrução PBIW com 64 bits é dividida em 8 campos de 5 bits que armazenam registradores de leitura, escrita ou imediatos, 1 campo de 16 bits para o endereço do padrão (índice para a tabela de padrões), 4 bits para anulação de linha e 4 bits para anulação de colunas. Seu respectivo padrão possui 368 bits divididos em 16 campos de 23 bits. As informações a respeito de uma operação são armazenados no padrão por meio de 1 campo de 8 bits para *opcode* e 5 campos de 3 bits para os operandos: três campos representam os operandos de escrita (1 campo) e de leitura (2 campos) e dois campos são utilizados para representar os imediatos junto com um dos campos de leitura.

Cada campo de operando do padrão aponta para um dos 8 campos existentes na instrução codificada. O total de bits da instrução completa (64 + 368 = 432) é menor do que uma instrução 2D-VLIW (512 bits).

5 Experimentos e Resultados

Os experimentos foram realizados através da geração de código com instruções PBIW para a arquitetura 2D-VLIW conforme Figura 4. O resultado foi comparado ao

código gerado de acordo com os esquemas de codificação 2D-VLIW original e EPIC derivado de 2D-VLIW. Foram utilizados 15 programas provenientes dos $benchmarks$ MediaBench (epic, g721decode, g721encode, gsmdecode, gsmencode e pegwit), SPECfp (168wupwise, 179art e 183equake) e SPECint (175vpr, 181mcf, 197parser, 255vortex, 256bzip2 e 300twolf).

Todos os programas foram compilados com o compilador Trimaran [4] (versão 3.7) e as opções de formação de hiperblocos e desenrolamento de laços (em até 32 vezes) habilitadas. Foram realizados experimentos estáticos para avaliação do reuso de padrões e do fator de redução que PBIW consegue atingir e também experimentos dinâmicos com o objetivo de avaliar o desempenho dos programas. Os experimentos dinâmicos foram realizados utilizando ferramentas próprias para geração de *traces* de execução de programas e também a ferramenta Dinero [6] para simulação de hierarquia de memória. Em ambos os experimentos, o modelo de execução 2D-VLIW foi especificado através da linguagem de descrição de arquiteturas HMDES [7].

5.1 Avaliação Estática

Estes experimentos foram realizados para comprovar o reuso de padrões pelas instruções e também o fator de redução atingido pela técnica PBIW quando comparada a outras técnicas de codificação. A Tabela 1 mostra os resultados da estratégia de codificação PBIW, utilizando instruções de 64 e de 128 bits. As primeiras colunas, I_{2d}, I_{e2d} mostram o número de instruções codificadas no formato 2D-VLIW e EPIC baseado em 2D-VLIW. As colunas I_{p64} e I_{p128} mostram a quantidade de instruções PBIW de 64 e 128 bits respectivamente. As colunas P_{64} e P_{64} apresentam o número de padrões obtidos para PBIW de 64 e 128 bits. As colunas R_{64} e R_{128} indicam a taxa de reuso dos padrões (número de instruções dividido pelo número de padrões) para as codificações com 64 e 128 bits. As colunas F_{x64} e F_{x128} indicam o fator de redução (Equação 1) de um programa que utiliza instruções PBIW de 64 e 128 bits quando comparado ao mesmo programa codificado em outras técnicas. Nos experimentos apresentados nesta Seção, x é utilizado para representar 2D-VLIW ou EPIC.

$$\textbf{Fator de Redução} = 1 - \frac{((I_p \times \alpha) + (P \times \beta))}{(I_x \times \gamma)} \quad (1)$$

onde: α é o tamanho da instrução PBIW. No exemplo considerado neste experimento, cada instrução possui 64 ou 128 bits. β é o tamanho de cada padrão de instrução. Cada padrão possui 368 bits (instrução de 64 bits) ou 528 bits (instrução de 128 bits). I_x e γ são a quantidade de instruções codificadas e o tamanho de cada instrução, respectivamente, para alguma das outras duas técnicas. As instruções 2D-VLIW utilizam 512 bits. Uma instrução EPIC baseada em 2D-VLIW utiliza 576 bits (512 bits para armazenar 16 operações de 32 bits e os outros 64 bits para indicar dependências entre as operações).

Os resultados da Tabela 1 comprovam a existência da sobrejeção entre padrões e instruções. Uma sobrejeção de $14,05$ (coluna R_{64} do programa pegwit) indica que, em média, mais de 14 instruções PBIW utilizam o mesmo padrão.

O número de instruções PBIW de 64 bits aumentou entre 19% e 39%. Contudo, como pode ser observado pelas colunas F_{2d64} e F_{e2d64}, esse aumento não foi significativo a ponto de impedir a redução no tamanho do programa. A codificação PBIW de 64 bits produz programas entre 67% e 81% menores do que a codificação 2D-VLIW(coluna F_{2d64}).

A codificação PBIW de 64 bits (coluna I_{p64}) apresentou um grande aumento no número de instruções quando comparada à EPIC baseado em 2D-VLIW (coluna I_{e2d}), de 234% até 471%, com uma média de 387%. Isto era esperado pois não existem *NOPs* em instruções EPIC. Essa diferença entre o número de instruções não foi suficiente para a codificação EPIC conseguir programas menores que PBIW. A codificação PBIW de 64 bits produz programas até 46% menores do que a codificação EPIC (coluna F_{e2d64}).

5.2 Avaliação Dinâmica

A avaliação dinâmica visa mensurar o impacto da utilização de duas caches (I-cache + P-cache) durante a busca e execução das instruções, comparando com a utilização apenas da I-cache nas outras técnicas de codificação. Devido às restrições no simulador da infraestrutura de compilação Trimaran, foram considerados apenas 10 programas nos experimentos envolvendo a avaliação dinâmica. Além disso, o foco da avaliação dinâmica foi dedicado à codificação PBIW com 64 bits (Tabela 3) já que essa apresenta melhores resultados que a configuração com 128 bits.

Os experimentos adotam 1 ciclo de relógio para executar cada instrução por causa do *pipeline* e 5 ciclos de *Custo de Miss* para cada palavra lida da memória pela cache. O *Overhead por Miss* foi calculado como o *Custo de Miss* menos o tempo de execução (1 ciclo).

O *Overhead Total* foi calculado como $Misses \times Overhead\ por\ Miss \times Palavras\ por\ Bloco$ e o *Tempo de Execução* é o *Número de Instruções* executadas somado ao *Overhead Total*. O ganho de desempenho foi calculado como $S = 1 - (T_p/T_x)$ onde T_p corresponde ao tempo de execução do programa com instruções PBIW de 64 bits e T_x pode ser um dos seguintes valores: $2d$ para 2D-VLIW e $e2d$ para EPIC baseado em 2D-VLIW.

Todos os experimentos foram realizados com tamanhos

Tabela 1. Número de instruções e padrões ($\times$ mil) e Fator de Redução (%) PBIW sobre programas *SPEC* e $MediaBench$

Programa	I_{2d}	I_{e2d}	I_{p64}	I_{p128}	P_{64}	P_{128}	R_{64}	R_{128}	F_{2d64}	F_{2d128}	F_{e2d64}	F_{e2d128}
168wupwise	7,66	1,80	10,03	7,90	0,89	1,06	11,29	11,01	75	73	6	-3
175vpr	40,18	10,04	53,32	40,63	3,56	4,91	14,99	14,91	77	75	18	10
179art	9,07	2,05	11,71	9,32	1,00	1,22	11,67	11,47	76	73	5	-5
181mcf	5,85	1,53	7,69	5,90	0,80	0,98	9,57	9,46	74	70	11	-2
183equake	8,47	2,01	11,11	8,65	1,09	1,30	10,20	9,99	74	71	4	-7
197parser	41,69	11,39	55,55	41,95	3,93	5,36	4,15	14,08	77	74	24	16
255vortex	79,60	25,92	110,41	80,12	2,88	5,92	38,39	37,40	80	80	46	45
256bzip2	17,57	4,13	22,13	17,71	1,70	2,12	13,00	12,99	77	75	14	5
300twolf	66,20	15,34	78,47	66,35	4,14	5,52	18,95	18,92	81	79	26	19
epic	3,03	1,21	4,04	3,08	0,57	0,67	7,11	6,90	70	65	33	21
g721decode	1,64	0,53	2,19	1,64	0,37	0,39	5,91	5,94	67	63	9	-2
g721encode	1,65	0,53	2,20	1,65	0,37	0,39	6,02	6,02	67	63	10	-1
gsmdecode	9,91	2,65	11,80	9,91	0,75	0,93	15,75	15,32	80	78	33	26
gsmencode	13,01	3,20	15,56	13,02	1,07	1,30	14,48	14,21	79	77	25	18
pegwit	13,64	3,12	16,90	13,65	1,20	1,50	14,05	13,83	78	76	15	7
Média	21,28	5,70	27,54	21,43	1,62	2,24	13,70	13,50	75	73	19	10

de caches entre 4KB e 256KB, associatividade de mapeamento direto a 4-way e o número de palavras por bloco variando de 1 a 4. De forma equivalente à análise estática, foram realizadas simulações para instruções PBIW de 64 bits. A política de substituição foi LRU em todos os casos e a taxa de transferência foi de 8 bytes (PBIW de 64 bits) por acesso.

Para cada programa, foram escolhidos os tamanhos de P-cache e I-cache que apresentaram melhores valores de Tempo de Execução. Como os tamanhos de cache não são exatamente iguais entre as codificações devido à diferença no tamanho das palavras, cada configuração PBIW escolhida foi comparada a um tamanho de cache menor e também a um tamanho maior nas outras técnicas de codificação. A Tabela 2 mostra os tamanhos de cache comparados entre as codificações. A colunas C_I e C_P indicam respectivamente o tamanho da I-cache e da P-cache e a soma destes valores é apresentado na coluna C_{PB} que indica o tamanho total para PBIW. As colunas C_{2D} e C_{E2D} indicam, respectivamente, os tamanhos de I-caches para 2D-VLIW e EPIC baseado em 2D-VLIW.

Tabela 2. Tamanhos (KB) de caches considerados

Palavra	C_I	C_P	C_{PB}	C_{2D}	C_{E2D}
64 bits	8	5,8	13,8	8	9
				16	18
	8	2,9	10,9	8	9
				16	18
	32	2,9	34,9	32	18
				64	36

Na Tabela 3, as colunas T_{2d}, T_{e2d} e T_p representam o Tempo de Execução, em ciclos, respectivamente para as estratégias de codificação 2D-VLIW, EPIC baseado em 2D-VLIW e PBIW e as colunas S_{2d}, S_{e2d} representam o ganho de desempenho alcançado pela técnica PBIW sobre 2D-VLIW e EPIC baseado em 2D-VLIW respectivamente. (M) indica valores referentes a caches maiores que a cache PBIW enquanto que (m) indica valores referentes a caches menores que a cache PBIW.

A estratégia de codificação PBIW de 64 bits obteve o melhor desempenho em todos os casos quando comparada à estratégia de codificação 2D-VLIW. O valor de S_{2d} (m) variou de 81% à 96% com média de 91%. As principais razões para este ganho são o tamanho da instrução (64 bits vs 512 bits) e o $overhead$ por $miss$ (4 ou 28 ciclos vs 39 ciclos, ou no pior caso 4 + 28 vs 39 ciclos). O valor de S_{e2d} (m) variou de -27% à 69%, com média de 19%.

Apesar do número de instruções ser bem maior na técnica PBIW de 64 bits do que em EPIC baseado em 2D-VLIW, o tamanho de uma instrução PBIW é $9\times$ menor (64 vs 576 bits) e o reuso de padrões é bastante alto (16 instruções para o programa gsmdecode na Tabela 1). Além disso, o $overhead$ de $miss$ de PBIW (4 ou 28 ciclos e no pior caso 4 + 28 ciclos) é menor do que o $overhead$ de um $miss$ de EPIC (44 ciclos). Ao comparar com caches maiores EPIC, a codificação PBIW de 64 bits apresentou melhor desempenho em apenas alguns programas.

6 Conclusões

Neste artigo foi apresentada a técnica PBIW para codificação de instruções que reduz o tamanho dos programas em memória e minimiza a latência na busca de

Tabela 3. Tempo de Execução ($\times 10^6$ ciclos) e Desempenho (%) para PBIW de 64 bits

Programa	$T_{2d}\ (m)$	$T_{2d}\ (M)$	$T_{e2d}\ (m)$	$T_{e2d}\ (M)$	T_p	$S_{2d}\ (m)$	$S_{2d}\ (M)$	$S_{e2d}\ (m)$	$S_{e2d}\ (M)$
168wupwise	18.124	18.124	5.122	3.443	3.484	81	81	32	-1
175vpr	97	50	9	6	9	90	81	-1	-46
179art	12.801	4.447	419	418	532	96	88	-27	-27
183equake	471	350	22	16	19	96	95	12	-22
197parser	1.546	861	261	205	280	82	68	-7	-36
256bzip2	56.735	45.523	2.180	1.878	2.253	96	95	-3	-20
g721decode	4.670	4.494	954	199	292	94	94	69	-47
g721encode	4.727	4.603	1.024	213	335	93	93	67	-58
gsmdecode	4.242	508	328	294	275	94	46	16	6
pegwit	1.641	1.116	188	133	129	92	88	31	3
Média	10.505	8.007	1.051	680	761	91	83	19	-25

instruções. A estratégia utilizada consiste em fatorar as operações de um programa em instruções codificadas (I-cache) e padrões (P-cache).

O esquema de codificação PBIW pode ser aplicado em arquiteturas que recuperam instruções grandes da memória como processadores EPIC, arquiteturas reconfiguráveis com várias unidades funcionais e arquiteturas de alto desempenho baseada em matriz de unidades funcionais.

Os resultados da Seção 5 mostraram que um programa PBIW pode ser até 69% (média de 19%) mais rápido do que o mesmo programa codificado como EPIC e até 96% (média de 91%) mais rápido do que 2D-VLIW (Tabela 3). Também é possível observar reduções de até 46% (média de 19%) no tamanho de código quando PBIW é comparada à EPIC e até 81% (média de 75%) quando comparada à 2D-VLIW. Além disso, o grande reuso de padrões contribui para que a técnica PBIW seja uma alternativa viável para minimizar o gargalo na busca de instruções grandes da memória.

7 Agradecimentos

Os autores agradecem às agências Capes e CNPq pelo apoio financeiro durante o projeto 2D-VLIW. R. F. Batistella agradece ao CPqD por permitir sua dedicação ao projeto PBIW.

Referências

[1] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code Compression Based on Operand Factorization. In *Procs. of the* 31^{st} *IEEE MICRO*, pages 194–201. IEEE Computer Press, 1998.

[2] R. Batistella. PBIW: Um Esquema de Codificação Baseado em Padrões de Instrução. Master's thesis, Unicamp, Brazil, Fevereiro 2008.

[3] E. Billo, R. Azevedo, G. Araujo, P. Centoducatte, and E. W. Netto. Design of a Decompressor Engine on a SPARC Processor. In *Procs. of the* 18^{th} *SBCCI*, pages 110–114, Florianópolis, SC, Brazil, 2005.

[4] 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. *LNCS*, 3602:32–41, 2004.

[5] D. Citron and D. G. Feitelson. Revisiting Instruction Level Reuse. In *Procs. of the WDDD*, pages 62–70, May 2002.

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

[7] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3, Center for Reliable and High-Performance Computing - University of Illinois at Urbana-Champaign, Urbana-Champaign-Illinois, 1996.

[8] S. A. McKee. Reflections on the Memory Wall. In *Procs. of the* 1^{st} *ACM Computing Frontiers*, pages 162–167. ACM, April 2004.

[9] S. K. Menon and P. Shankar. Space/Time Tradeoffs in Code Compression for the TMS320C62x Processor. Technical Report IISc-CSA-TR-2004-4, Indian Institute of Science, India, 2004.

[10] S.-J. Nam, I.-C. Park, and C.-H. Kyung. Improving Dictionary-Based Code Compression in VLIW Architectures. *IEICE Transactions on Fundamentals*, E82-A(11):2318–2324, November 1999.

[11] M. Ros and P. Sutton. Code Compression Based on Operand-Factorization for VLIW Processors. In *Procs. of the CASES*, pages 559–569. ACM Press, September 2004.

[12] R. Santos, R. Azevedo, and G. Araujo. 2D-VLIW: An Architecture Based on the Geometry of the Computation. In *IEEE ASAP*, Steamboat Springs - Colorado, 2006. IEEE Computer Society.

[13] P. G. Sassone and D. S. Wills. Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication. In *Procs. of the* 37^{th} *IEEE/ACM MICRO*, pages 7–17, Washington, DC, USA, 2004. IEEE Computer Society.

[14] M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, February 2000.