UNIVERSIDADE FEDERAL DE SANTA CATARINA
INFORMÁTICA E ESTATÍSTICA

Daniel Pereira Volpato

GERENCIAMENTO EXPLÍCITO DE MEMÓRIA AUXILIAR A PARTIR DE ARQUIVOS-OBJETO PARA MELHORIA DA EFICIÊNCIA ENERGÉTICA DE SISTEMAS EMBARCADOS

Florianópolis

2010

Daniel Pereira Volpato

GERENCIAMENTO EXPLÍCITO DE MEMÓRIA AUXILIAR A PARTIR DE ARQUIVOS-OBJETO PARA MELHORIA DA EFICIÊNCIA ENERGÉTICA DE SISTEMAS EMBARCADOS

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: José Luís Almada Güntzel, Dr.

Florianópolis

2010

Catalogação na fonte pela Biblioteca Universitária
da
Universidade Federal de Santa Catarina

V931g Volpato, Daniel Pereira
Gerenciamento explícito de memória auxiliar a partir de arquivos-objeto para melhoria da eficiência energética de sistemas embarcados [dissertação] / Daniel Pereira Volpato ; orientador, Luís Almada Güntzel. - Florianópolis, SC, 2010.
142 p.: il., grafs., tabs.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da computação. 2. Arquitetura de computador. 3. Sistemas de memória de computadores. 4. Gerenciamento de memória (Computação). I. Santos, Luiz Claudio Villar dos. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 681

Daniel Pereira Volpato

GERENCIAMENTO EXPLÍCITO DE MEMÓRIA AUXILIAR A PARTIR DE ARQUIVOS-OBJETO PARA MELHORIA DA EFICIÊNCIA ENERGÉTICA DE SISTEMAS EMBARCADOS

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre em Ciência da Computação", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 20 de dezembro 2010.

Mario Antonio Ribeiro Dantas, Dr.
Coordenador

Banca Examinadora:

José Luís Almada Güntzel, Dr.
Orientador

Fernando Gehm Moraes, Dr.

Cesar Albenes Zeferino, Dr.

Luiz Cláudio Villar dos Santos, Dr.

À minha família, pelo que sou.

AGRADECIMENTOS

A Deus, pela vida e por amar-me de modo incondicional. Por toda a beleza, harmonia e ordem de Sua criação que nos rodeia, na qual podemos enxergar toques de Sua mão e a certeza de sua presença.

Aos meus pais, Volnei e Maria José, pela educação, formação humana, moral e religiosa. Juntamente com eles, também aos meus irmãos (Rafael, Mateus e Roberta) e a Iara pelo apoio e compreensão, principalmente quando não lhes dediquei o tempo devido.

A minha noiva, Hérica, pelo amor que me dedica e dedicou. Também pela paciência e compreensão nas diversas etapas deste mestrado, especialmente em sua reta final.

Ao Professor Dr. José Luís Almada Güntzel, pela orientação e amizade ao longo deste mestrado, pelas importantes contribuições, pela atenção e esforço empregados na revisão deste texto, e pela valiosa colaboração em minha formação profissional bem como pessoal.

Ao Professor Dr. Luiz Cláudio Villar dos Santos, pela coorientação deste trabalho, pelas sugestões, críticas e reflexões que tanto contribuíram para a melhoria da qualidade deste trabalho, e pela amizade já desde os tempos em que cursava a graduação.

Aos membros da banca, Professor Dr. Fernando Gehm Moraes e Professor Dr. Cesar Albenes Zeferino, por aceitarem o convite para avaliar este trabalho e pelas contribuições para sua melhoria.

Aos parceiros de grupo de pesquisa do LAPS e NIME, pela convivência e auxílio para o desenvolvimento deste trabalho. Em particular, aos colegas e amigos Alexandre K. I. de Mendonça e Rafael Westphal, com os quais colaborei mais diretamente.

Aos amigos do Grupo de Oração Universitário (GOU), pela amizade, torcida, oração, e por tornarem a vivência na universidade muito mais agradável. Aos amigos Luiz, Sayonara, Daiane, Letícia, e a todos aqueles que acompanharam o desenrolar deste trabalho.

Aos Professores, Técnicos e Funcionários do Departamento de Informática e Estatística (INE) da UFSC, pelo auxílio e colaboração. Ao INE pela infraestrutura concedida.

À CAPES, no âmbito do Programa de Fomento à Pós-Graduação (PROF), por bolsa de quota social, e ao CNPq, no âmbito do Programa Nacional de Microeletrônica (PNM), processo nº 136630/2008-1, pelo custeio parcial da execução deste trabalho. À CAPES, no âmbito do Programa Nacional de Cooperação Acadêmica (PROCAD), nº 0326054, pelo auxílio-moradia para missão de estudo na UNICAMP.

É uma doença natural no homem acreditar que possui a verdade.

Blaise Pascal

RESUMO

Memórias de rascunho (*Scratchpad Memories* — SPM) tornaram-se populares em sistemas embarcados por conta de sua eficiência energética. A literatura sobre SPMs parece indicar que a alteração dinâmica de seu conteúdo suplanta a alocação estática. Embora técnicas *overlay-based* (OVB) operando em nível de código-fonte possam beneficiar-se de múltiplos *hot spots* para uma maior economia de energia, elas não conseguem explorar elementos de programa oriundos de bibliotecas. Entretanto, quando operam diretamente em binários, as abordagens OVB conduzem a uma menor economia, frequentemente exigem hardware dedicado e às vezes impossibilitam a alocação de dados.

Por outro lado, a economia de energia reportada por todas as técnicas, até o momento, ignora o fato de que, em sistemas que possuem caches, estas deverão ser otimizadas antes da alocação para SPM. Este trabalho mostra evidência experimental de que, quando métodos *non-overlay-based* (NOB) são utilizados para manipulação de arquivos binários, a economia de energia em memória, por conta da alocação em SPM, varia entre 15% a 33%, e média, e é tão boa ou melhor do que a economia reportada para abordagens OVB que operam sobre binários.

Como esta economia (ao contrário dos trabalhos correlatos) foi medida após o ajuste-fino das caches — quando existe menos espaço para otimização —, estes resultados estimulam o uso de métodos NOB, mais simples, para a construção de alocadores capazes de considerar elementos de bibliotecas e que não dependam de hardware especializado.

Este trabalho também mostra que, dada uma capacidade C_T de uma cache pré-ajustada equivalente, o tamanho ótimo de SPM reside em $[C_T/2, C_T]$ para 85% dos programas avaliados.

Finalmente, mostram-se evidências contra-intuitivas de que, mesmo para arquiteturas baseadas em cache contendo SPMs pequenas, é preferível utilizar-se a granularidade de procedimentos à de blocos básicos, exceto em algumas poucas aplicações que combinam elementos frequentemente acessados e taxas de faltas relativamente altas.

Palavras-chave: Sistemas embarcados. Subsistema de memória. Scratchpad memory. Memória de rascunho. Gerenciamento overlay. Gerenciamento non-overlay.

ABSTRACT

Scratchpad memories (SPMs) became popular in embedded systems as energy efficiency boosters. The literature on SPMs seems to indicate that the use of dynamic overlaying supersedes static allocation. Although overlay-based (OVB) techniques operating at source-level code might benefit from multiple hot spots for higher energy savings, they cannot exploit libraries. When directly operating on binaries, OVB approaches lead to smaller savings, often require dedicated hardware, and sometimes prevent data allocation.

Besides, all saving reports published so far ignore the fact that, in cache-based systems, caches are likely to be optimized prior to SPM allocation. This work shows experimental evidence that, when non-overlay based (NOB) methods are used to directly handle binaries, the memory energy savings due to SPM allocation (from 15% to 33% on average) are as good as or better than the ones reported for OVB approaches that are also able to operate on binaries.

Since the savings obtained in the present work (as opposed to related works) were measured after cache tuning — when there is less room for optimization, they encourage the use of simpler NOB methods to build library-aware SPM allocators that cannot depend on dedicated hardware.

This work also shows that, given the capacity C_T of the equivalent pretuned cache, the optimal SPM size lies in $[C_T/2, C_T]$ for 85% of the programs under evaluation.

Finally, it shows counter-intuitive evidence that, even for cache-based architectures containing small SPMs, procedures should be preferred for allocation instead of basic blocks, except for a few applications combining frequently accessed elements and relatively high miss rates.

Keywords: Embedded systems. Memory subsystem. Scratchpad memory. Overlay management. Non-overlay management.

LISTA DE FIGURAS

Figura 1 Distribuição de energia em um processador embarcado (DALLY et al., 2008) ... 35
Figura 2 Exemplo de memória cache com mapeamento direto ... 37
Figura 3 Exemplo de memória de rascunho (SPM) ... 39
Figura 4 Arquiteturas-alvo possíveis para o subsistema de memória 41
Figura 5 Fluxo de trabalho genérico para técnicas de alocação em SPM ... 48
Figura 6 Mapa de memória da arquitetura MIPS (PATTERSON; HENNESSY, 2008) ... 59
Figura 7 Fluxo de trabalho da técnica estendida de alocação em SPM ... 77
Figura 8 Casos que podem ocorrer no mapeamento de blocos básicos para SPM ... 78
Figura 9 Impacto do ajuste-fino na economia de energia das caches de instruções e dados ... 92
Figura 10 Economia média de energia e taxa média de ocupação por capacidade de SPM ... 106
Figura 11 Sensibilidade da economia de energia ao dimensionamento da SPM (usando abordagem PRA) ... 108
Figura 12 Maior economia de energia, utilizando BBA e PRA, para cada programa ... 110
Figura 13 Capacidades de SPM que propiciam maior economia de energia ... 111
Figura 14 Correlação entre economia de energia e taxa de faltas global dos elementos candidatos, para SPMs grandes ($C_{SPM} \sim C_T$) ... 113

LISTA DE TABELAS

Tabela 1 Técnicas de alocação em SPM quanto à abordagem, fase, arquivo de entrada e arquitetura-alvo 62
Tabela 2 Técnicas de alocação em SPM quanto aos elementos de programa considerados 63
Tabela 3 Espaço de projeto considerado para ajuste-fino das memórias cache 91
Tabela 4 Resultado do ajuste-fino das memórias cache 91
Tabela 5 Descrição dos programas de *benchmark* utilizados 97
Tabela 6 Percentual de acessos acomodáveis em diferentes capacidades de uma memória qualquer 99
Tabela 7 Capacidade da SPM utilizada para cada configuração e programa 100
Tabela 8 Propriedades extraídas para caracterização dos programas-alvo 102
Tabela 9 Energia normalizada para a configuração de cache pré-ajustada 104
Tabela 10 Ocupação da SPM 105
Tabela 11 Correlação entre economia de energia total e de sistema 141

LISTA DE ALGORITMOS

1 SPCE 135
2 CONTA_CONFLITOS 135

LISTA DE ABREVIATURAS E SIGLAS

BB Bloco básico

BBA *Basic-block allocation* (alocação de blocos básicos)

CAM *Content-addressable memory* (memória endereçada por conteúdo)

CBA *Cache-based architectures* (arquiteturas baseadas em caches)

CT *Compilation time* (Tempo de compilação)

D-cache Cache de dados

DRAM *Dynamic Random Access Memory*

EDA *Eletronic Design Automation* (automação de projeto eletrônico)

EVA *Memory architecture under evaluation* (arquitetura de memória sob avaliação)

FCA *Fully-cached architectures* (arquiteturas somente com caches)

I-cache Cache de instruções

ILP *Integer Linear Programming* (programação linear inteira)

IP Intellectual Property

KB Kilo-bytes

MB Mega-bytes

MMU *Memory Management Unit* (unidade de gerenciamento de memória)

MP Memória principal

NOB *non-overlay-based*

OVB *overlay-based*

PC *Pos-compilation time* (Tempo de pós-compilação)

PR Procedimento

PRA *Procedure Allocation* (alocação de procedimentos)

RAM *Random Access Memory*

REF *Reference memory architecture* (arquitetura de memória de referência)

SoC *System-on-Chip* (sistema integrado)

SPM *Scratchpad Memory* (memória de rascunho)

SRAM *Static Random Access Memory*

T-cache Cache unificada equivalente

TCM *Tightly Coupled Memory* (memória fortemente acoplada)

UNA *Uncached architectures* (arquiteturas sem cache)

WCET *Worst-Case Execution Time* (tempo de execução do pior caso)

LISTA DE SÍMBOLOS

$\mathbf{M}$ Uma memória genérica.

$\mathbf{E_M}$ Energia consumida em um único acesso à memória M.

$\lambda_\mathbf{M}$ Latência da memória M (expressa em ciclos de relógio).

$\mathbf{C_M}$ Capacidade de memória M (expressa em bytes).

$\mathbf{T}$ Padrão de acessos à memória (*trace*).

$\alpha_\mathbf{i}$ i-ésimo endereço de acesso à memória.

$\mathbf{D_i}$ Elemento de programa candidato.

$\sigma_\mathbf{i}$ Tamanho (bytes) do elemento candidato D_i.

$\mathbf{a_i}$ Número de acessos a um elemento candidato D_i.

$\mathbf{m_i}$ Taxa de faltas (*miss rate*) do elemento candidato D_i.

$\mathbf{E_i}$ Energia consumida por um acesso ao elemento candidato D_i.

$\mathbf{p_i}$ Lucro de energia quando aloca-se o candidato D_i em SPM.

$\varepsilon_\mathbf{i}$ *Overhead* de energia quando aloca-se o candidato D_i em SPM.

$\mathbf{w_i}$ Espaço necessário quando aloca-se o candidato D_i em SPM.

$\sigma_\mathbf{extra}$ Tamanho total (bytes) das instruções extras necessárias quando aloca-se o candidato D_i em SPM.

$\mathbf{W}$ Matriz de caracterização de espaço dos elementos candidatos.

$\mathbf{P}$ Matriz de caracterização de lucro dos elementos candidatos.

$\mathbf{X}$ Matriz de mapeamento de elementos em SPM.

$\mathbf{x_i}$ Denota se um candidato D_i está ou não mapeado para alocação em SPM.

$\tau(\mathbf{D_i})$ Função que mapeia um elemento candidato D_i para seu tipo $(BB, proc$ ou $data)$.

$\varepsilon_\mathbf{i}^\mathbf{MP}$ *Overhead* de energia no espaço de endereçamento da MP quando aloca-se o candidato D_i em SPM.

$\varepsilon_\mathbf{i}^\mathbf{SPM}$ *Overhead* de energia no espaço de endereçamento da SPM quando aloca-se o candidato D_i em SPM.

$\mathbf{r_i}$ Taxa de invocação do bloco básico D_i.

$\mathbf{N_i}$ Número de invocações devidas às iterações do laço do bloco básico D_i.

$\mathbf{S_i}$ Número de invocações do bloco básico D_i a partir de outro bloco básico.

$(\imath/\theta)$ Taxa de amostragem do método de ajuste-fino: processa-se $\imath$ endereços de memória, ignoram-se os próximos θ.

$\mathbf{m_I}$ Taxa de faltas locais da I-cache.

$\mathbf{m_D}$ Taxa de faltas locais da D-cache.

LS Percentagem do número de instruções de carga (*load*) e escrita (*store*).

$\mathbf{m_T}$ Taxa de faltas combinada da I-cache e D-cache.

$\mathbf{V_{DD}}$ Tensão de alimentação (volts).

$\mathbf{\bar{m}}$ Taxa de faltas global dos candidatos.

$\mathbf{\bar{a}}$ Média do número de acessos dos candidatos.

σ Desvio-padrão do número de acessos dos candidatos.

H Conjunto dos elementos candidatos classificados como *hot spots*.

|H| Cardinalidade de H, i.e. número de elementos candidatos classificados como *hot spots*.

h Frequência de ocorrência dos elementos candidatos classificados como *hot spots*.

$\mathbf{E_N}$ Energia consumida pelo subsistema de memória, normalizada para a arquitetura de referência (REF).

$\mathbf{E_{EVA}}$ Energia consumida pelo subsistema de memória da arquitetura sob avaliação (EVA).

$\mathbf{E_{REF}}$ Energia consumida pelo subsistema de memória da arquitetura de referência (REF).

$\mathbf{E_{Mem}}$ Energia consumida pelo subsistema de memória.

$\mathbf{E_{Total}}$ Energia consumida por todo o sistema.

k Fator de proporcionalidade entre E_{Mem} e E_{Total}.

SUMÁRIO

Lista de Abreviaturas e Siglas . . . .
Lista de Símbolos . . . .
1 INTRODUÇÃO 31
1.1 SISTEMAS EMBARCADOS 31
1.2 O SUBSISTEMA DE MEMÓRIA 33
1.2.1 Principais componentes do subsistema de memória . 35
Memória Principal (MP) 36
Memória cache 36
Memória de rascunho (SPM) 38
1.2.2 Arquiteturas para o subsistema de memória 40
Arquiteturas somente com caches (FCAs) 40
Arquiteturas sem cache (UNAs) 40
Arquiteturas baseadas em cache (CBAs) 40
. . . . 41
1.3 ESCOPO DESTE TRABALHO 41
1.4 PRINCIPAIS CONTRIBUIÇÕES 44
1.5 ORGANIZAÇÃO DESTA DISSERTAÇÃO 45
2 ALOCAÇÃO EM MEMÓRIAS DE RASCUNHO 47
2.1 VISÃO GERAL DO PROCESSO DE ALOCAÇÃO 47
2.2 CARACTERÍSTICAS DAS TÉCNICAS DE ALOCAÇÃO . . 50
2.2.1 Elementos de programa 51
2.2.1.1 Tipo dos elementos 51
2.2.1.2 Origem dos elementos 52
2.2.2 Granularidade dos elementos 53
2.2.2.1 Granularidade de código 53
2.2.2.2 Granularidade de dados 56
2.2.3 Fase de alocação 57
2.2.4 Abordagem de alocação 58
. . . . 61
2.3 O ESTADO-DA-ARTE EM ALOCAÇÃO A PARTIR DE ARQUIVOS BINÁRIOS 61
2.4 CONSIDERAÇÕES SOBRE AS ABORDAGENS DE ALOCAÇÃO EM SPM 67
3 O PROBLEMA-ALVO 71
4 EXTENSÃO DE UMA TÉCNICA NOB PARA INCLUSÃO DE BLOCOS BÁSICOS NO ESPAÇO DE OTIMIZAÇÃO 75

4.1 FLUXO DE TRABALHO 76
4.2 CARACTERIZAÇÃO DOS ELEMENTOS 76
4.3 *PROFILING* DO PROGRAMA 80
4.4 CARACTERIZAÇÃO DE LUCRO E ESPAÇO 80
4.4.1 Lucro de energia de um bloco básico 81
4.4.2 Espaço necessário para alocar um bloco básico 82
4.5 MAPEAMENTO EM SPM 82
4.6 *PATCHING* DE BINÁRIOS 82
4.7 GERAÇÃO DE SAÍDA 84
5 AJUSTE-FINO DE CACHES PARA AVALIAÇÃO DA ALOCAÇÃO EM SPMS 85
5.1 AS TÉCNICAS DE AJUSTE-FINO DE CACHES 86
5.2 O MÉTODO SPCE 88
5.3 IMPLEMENTAÇÃO DO MÉTODO SPCE 88
5.4 DETERMINAÇÃO DAS CACHES PRÉ-AJUSTADAS 90
5.5 IMPACTO DO AJUSTE-FINO NA ECONOMIA DE ENERGIA 90
5.6 CÁLCULO DA CACHE UNIFICADA EQUIVALENTE 93
6 VALIDAÇÃO EXPERIMENTAL E RESULTADOS .. 95
6.1 CONFIGURAÇÃO EXPERIMENTAL 95
6.2 GERAÇÃO DOS EXPERIMENTOS 96
6.3 CARACTERIZAÇÃO DOS PROGRAMAS-ALVO 98
6.4 ANÁLISE DOS RESULTADOS 103
6.4.1 Sensibilidade da economia ao dimensionamento da SPM 103
6.4.2 Política de alocação de maior eficiência energética para uma determinada capacidade de SPM 107
6.4.3 Política de alocação de maior eficiência energética para um determinado programa 109
6.4.4 Capacidade ótima da SPM 109
6.4.5 Correlação entre economia de energia e taxa de faltas para SPMs grandes 112
6.4.6 Ocupação das SPMs ótimas 112
6.4.7 Determinação de um escopo para utilização de BBA 112
6.4.8 Comparação com trabalhos correlatos 114
7 CONCLUSÕES E PERSPECTIVAS 117
7.1 EVIDÊNCIA EXPERIMENTAL SÓLIDA 117
7.2 IMPORTÂNCIA DO AJUSTE-FINO 117
7.3 IMPORTÂNCIA DA CORRELAÇÃO ENTRE TAMANHO DA CACHE PRÉ-AJUSTADA EQUIVALENTE E TAMANHO DA SPM 118

7.4 DIMENSIONAMENTO DA SPM 118
7.4.1 Impacto do dimensionamento 118
7.4.2 Diretrizes para dimensionamento 119
7.4.3 SPMs grandes e as taxas de faltas 120
7.5 POLÍTICA DE ALOCAÇÃO (GRANULARIDADE DE CÓDIGO) 120
7.5.1 Alocação de procedimentos (PRA) 120
7.5.2 Alocação de blocos básicos (BBA) 120
7.6 REAVALIAÇÃO EXPERIMENTAL DAS TÉCNICAS NOB A PARTIR DE ARQUIVOS BINÁRIOS 121
7.7 PERSPECTIVAS 122
Referências Bibliográficas 123
APÊNDICE A – O método SPCE 133
APÊNDICE B – Correlação entre economia de energia total e de sistema 141

1 INTRODUÇÃO

1.1 SISTEMAS EMBARCADOS

Em pouco mais de duas décadas, os computadores, que causaram uma verdadeira revolução no modo de vida da sociedade contemporânea, passaram por sua própria revolução.

Inicialmente na forma de enormes *mainframes*, transformaram-se em compactos computadores pessoais, cada vez com maior capacidade de processamento, por conta do processo de miniaturização de seus circuitos integrados.

Paralelamente, a eletrônica de consumo evoluiu para muito além das calculadoras programáveis, incorporando "computadores portáteis" com um poder de processamento muito superior aos *mainframes* de outrora, embora mantendo sua característica de corresponderem a um domínio específico de aplicação — e.g. telecomunicações, entretenimento, etc. Estes dispositivos seguem evoluindo rapidamente. De um lado, cresce a demanda por uma maior capacidade de processamento e de armazenamento. De outro, aumentam os requisitos de portabilidade: redução de tamanho e peso, e baterias com duração satisfatória, pois tratam-se de sua principal fonte de alimentação.

Estes "sistemas de processamento de informação que estão incorporados em um produto maior e que normalmente não estão diretamente visíveis ao usuário" são chamados de **sistemas embarcados** e colaboram com esta nova tendência, chamada de "*disappearing computer*" (MARWEDEL, 2006): computação acontecendo em todo o lugar (computação ubíqua), porém invisível, sendo realizada em dispositivos com aparência que foge do tradicional "gabinete e monitor" e cuja presença não se consegue identificar.

A miniaturização e a queda no preço dos sistemas embarcados permitiu que eles se disseminassem por todas as áreas da vida humana. Como exemplos de produtos contendo sistemas embarcados, pode-se citar: telefones celulares, câmeras digitais, tocadores de MP3, fornos de microondas, conversores de TV digital, carros (no sistema de controle de freio ABS, do motor, de ajuste ao combustível nos carros bicombustíveis, do ar-condicionado, GPS, etc.), aeronaves (no computador de bordo, sistema anti-colisão, no sistema para pousos e decolagens guiadas, etc.), equipamentos médicos, casas inteligentes, e também em produtos militares, de robótica e de muitas outras áreas.

De acordo com Marwedel (2006) e Verma e Marwedel (2007),

pode-se classificar um sistema como embarcado quando este for dotado da maioria das características que seguem:

- **Dedicados a uma certa aplicação**. A maioria dos sistemas embarcados realiza um conjunto pré-determinado e dedicado de funções. Por exemplo, o controlador de ABS de um carro nunca tocará um CD de música. Esta característica também está relacionada com confiabilidade e com eficiência, que diminuem quando se permite a execução de outros softwares.

- **Confiabilidade**. Alguns sistemas são de alto-risco e necessitam ser confiáveis (*dependables*) quanto a falhas. A confiabilidade abrange aspectos como: baixíssima taxa de falhas (*reliability*); fácil e rápida manutenção no caso de eventuais falhas; disponibilidade (sistema sempre operando); segurança no funcionamento (*safety*), não causando nenhum mal caso falhe; e segurança dos dados (*security*), caso sejam confidenciais.

- **Eficiência**. A eficiência de um sistema embarcado deve acontecer em vários níveis: energética, pois muitos são alimentados por baterias; de tamanho de código, pois possuem memória limitada; de execução, realizando suas tarefas sempre com o mínimo de recursos e utilizando frequências de relógio e tensões de alimentação baixas, contribuindo também para redução do consumo de energia; de peso, por se tratar de um quesito muito desejado pelos consumidores de dispositivos portáteis; e, finalmente, de custo, para terem competitividade no mercado.

- **Sistemas reativos**. Frequentemente, sistemas embarcados são reativos, conectados ao mundo físico por meio de sensores que coletam informações às quais o sistema reage, inclusive por meio de atuadores, que permitem controlar o ambiente.

- **Interface de usuário dedicada**. Sistemas embarcados não utilizam a interface convencional dos computadores pessoais, como teclados e mouses, mas interfaces diferenciadas como telas sensíveis ao toque, botões de pressão, etc.

- **Restrições de tempo real**. Sistemas de tempo-real são aqueles em que a não-realização de uma computação em tempo hábil causa ou perda de qualidade (e.g. em transmissões de áudio ou vídeo) ou danos ao usuário (e.g. no controle de *airbag* de um automóvel ou mesmo em usinas nucleares).

- **Sistemas híbridos**. Muitos sistemas embarcados misturam componentes analógicos e digitais.

Mais especificamente no domínio de sistemas embarcados voltados à eletrônica de consumo, que corresponde a uma significativa parcela destes sistemas e abrange produtos como celulares, tocadores de vídeo e áudio e consoles de videogame, algumas das características comentadas anteriormente destacam-se das demais, sendo consideradas mais importantes por impactarem diretamente na experiência do usuário (VERMA; MARWEDEL, 2007). São elas: desempenho, eficiência energética e previsibilidade (responsividade em tempo real).

Em muitos projetos de sistemas embarcados, essas características não são meros objetivos, que podem ser otimizados, mas são elevadas a restrições de projeto (e.g. máxima potência ou máximo tempo de execução), que neste caso devem ser satisfeitas. Por conseguinte, os projetistas de sistemas embarcados (em particular de sistemas voltados ao consumidor) necessitam otimizar os componentes de hardware e software para garantir que estes objetivos ou restrições sejam satisfeitos.

Durante muito tempo, tais otimizações concentraram-se no processador destes sistemas, por se tratar do elemento de maior influência sobre estas características. No entanto, hoje em dia é amplamente reconhecido que a parte mais importante no projeto de um sistema embarcado é a organização, arquitetura e projeto do subsistema de memórias (WEHMEYER; MARWEDEL, 2006) (JACOB; NG; WANG, 2007) (VERMA; MARWEDEL, 2007), conforme será demonstrado na próxima seção.

1.2 O SUBSISTEMA DE MEMÓRIA

Por subsistema de memória entende-se o conjunto dos diversos componentes de memória de um sistema embarcado. Em sistemas embarcados podem coexistir espaços de endereçamento disjuntos (determinada faixa de endereços corresponde a uma memória X e outra faixa a uma memória Y), onde cada um destes espaços está associado ou a uma memória ou a uma hierarquia de memórias, com dois ou três níveis.

Desde o surgimento dos circuitos integrados, a velocidade dos processadores tem aumentado mais rapidamente do que a velocidade de acesso às memórias, resultando em uma diferença significativa de desempenho. Nos sistemas de propósito geral, o fator de crescimento da velocidade dos processadores já atingiu, nos últimos 30 anos, picos

de 2 a 2,5 vezes superior ao das memórias (MACHANIK, 2002).

Embora em sistemas de propósito geral a diferença de velocidade entre processador e memória tenha sido mais acentuada do que em sistemas embarcados, devido ao distinto funcionamento e projeto das memórias para cada sistema, sabe-se que este comportamento pode ser estendido também para sistemas embarcados: o desempenho de um sistema é dominado e limitado pelas lentas memórias (JACOB; NG; WANG, 2007). Isso acontece porque, apesar da evolução tecnológica ter levado a transistores cada vez menores — o que deveria proporcionar acessos mais rápidos —, há uma necessidade constante por mais capacidade de memória nos sistemas embarcados. Ou seja, enquanto de um lado há a diminuição do tempo de acesso por MB, de outro, o aumento da capacidade das memória eleva o tempo de acesso global. Na prática, as memórias, que deveriam ficar menores e mais rápidas, estão se tornando mais densas e com mais capacidade, do que resulta uma melhoria do tempo de acesso das memórias insuficiente para acompanhar a evolução dos processadores.

Desse hiato entre a velocidade dos processadores e das memórias decorre que o processador necessita esperar por muitos ciclos — possivelmente até centenas deles — para que a memória lhe forneça os dados desejados. Assim, o gargalo do desempenho de um sistema não se encontra no processador, mas no subsistema de memórias.

Como uma memória principal única, rápida e grande o suficiente é inviável na atual tecnologia de memórias, devido ao preço extremamente alto, a solução encontrada pelos projetistas de sistemas embarcados tem sido compor hierarquias de memórias de modo a atenuar a diferença de velocidade entre o processador e a memória principal. Estas hierarquias são elaboradas utilizando memórias menores e mais rápidas que a memória principal e, por isso, de maior eficiência energética do que estas. Tais hierarquias são posicionadas entre o processador e a memória principal, onde cada nível armazena temporariamente elementos previamente acessados no nível superior, evitando assim, acessos mais custosos em termos de tempo de acesso e energia aos níveis superiores.

No entanto, o uso de hierarquias não se mostrou suficientemente eficiente para atacar um outro problema: o consumo de energia, quesito de suma importância em sistemas embarcados alimentados por bateria. De acordo com Verma e Marwedel (2007), o subsistema de memória é responsável por 50 a 70% do orçamento total de potência do sistema. E dentro do subsistema, observa-se impacto significativo das caches neste consumo. Analisando o gasto energético relacionado apenas com um processador embarcado, Segars (2001) relata que o ARM920T dissipa

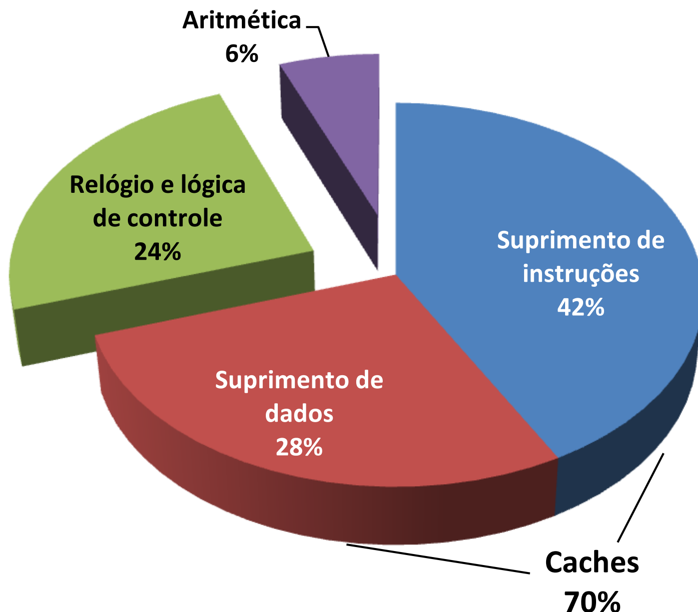


Figura 1: Distribuição de energia em um processador embarcado (DALLY et al., 2008)

43% da sua potência em caches. Dally et al. (2008) reportam, conforme a Figura 1, que 70% do total de energia gasto pelo processador é oriundo do suprimento de instruções (42%) e de dados (28%), com contribuição majoritária dos arranjos de dados e de *tag*, e dos controladores das caches. Já no caso de um sistema embarcado com processador RISC voltado para aplicações de processamento de imagem, ou seja, *data-intensive*, Ming, Yu e Lin (2008) reportam que somente a memória externa ao circuito integrado (*off-chip*) consome algo entre 50 a 80% do total de energia.

1.2.1 Principais componentes do subsistema de memória

Uma vez esclarecida a importância do projeto energeticamente consciente de um subsistema de memória, é conveniente que se revisem as características mais relevantes dos principais componentes de memória: memória principal, memória cache e memória de rascunho.

MEMÓRIA PRINCIPAL (MP)

É a maior, e por isso considerada a mais importante, memória do sistema. É ela que armazena todo, ou pelo menos a maior parte, do código e dos dados de um programa a ser executado no sistema. É uma memória de acesso aleatório, encontrada tanto externa (*off-chip*) quanto interna (*on-chip*) ao circuito integrado do processador e cujas células são normalmente fabricadas com tecnologia *Dynamic Random Access Memory* (DRAM), capaz de oferecer uma grande capacidade de armazenamento a baixo custo, pois demandam um único transistor e uma capacitância de transistor por bit. Como as cargas armazenadas neste capacitor se perdem via corrente de fuga, esta tecnologia exige uma lógica de controle mais complexa para restaurar periodicamente (*refreshment*) estas cargas, mantendo seu conteúdo armazenado.

MEMÓRIA CACHE

Uma *memória cache* (Figura 2) armazena cópias de instruções e/ou dos dados mais recentemente acessados. Seu conteúdo é gerenciado implicitamente (sem interferência do usuário) por um hardware dedicado: controlador de cache para permitir a transferência de informação entre níveis hierárquicos distintos e para o gerenciamento do conteúdo. Está localizada dentro do circuito integrado do processador (*on-chip*). Sua célula é construída com tecnologia *Static Random Access Memory* (SRAM), necessitando de 6 transistores por bit. Por esta tecnologia apresentar um maior custo por KB se comparada a uma DRAM, uma cache geralmente não possui grande capacidade de armazenamento. Isso, no entanto, torna-a extremamente rápida (geralmente, sua latência pode ser acomodada dentro de um ciclo de processador para o caso de uma cache primária) e de maior eficiência energética do que as MPs, motivo pelo qual passou a ser utilizada em sistemas embarcados.

A unidade mínima de informação em uma cache é chamada de bloco, sendo constituído de uma ou mais palavras de memória.

As caches foram construídas com o propósito de serem transparentes ao usuário e ao compilador, não necessitando de nenhum suporte adicional da parte destes para que se tire proveito de seus benefícios. Para tanto, contam com recursos extras de hardware (o arranjo de *tags*, comparadores e multiplexadores) que gerenciam implicitamente seu conteúdo: verificam se a cache possui ou não uma cópia válida do bloco que contém a palavra de memória requisitada, utilizando um conjunto

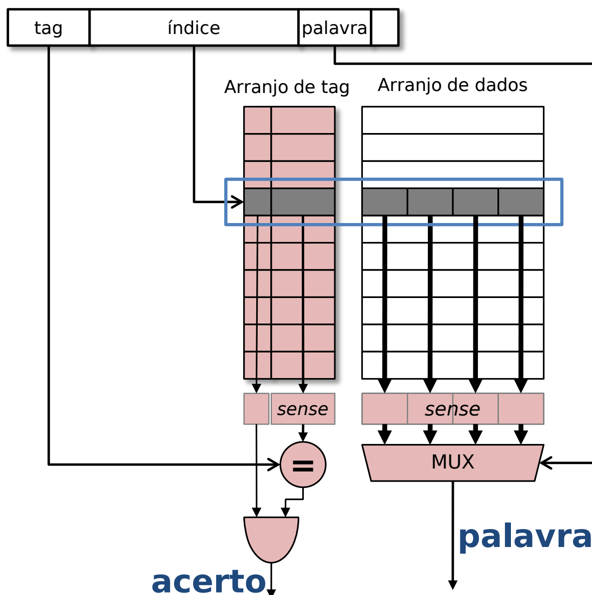


Figura 2: Exemplo de memória cache com mapeamento direto

de bits do endereço da palavra, denominado *tag*. Quando não contém, o controlador da cache recupera o bloco ao qual esse endereço pertence e armazena-o juntamente com sua *tag*.

Para ilustrar o funcionamento de uma cache, considere-se uma cache com mapeamento direito. Neste caso, ela é organizada utilizando um mapeamento $n : 1$ entre endereços de memória e entradas da cache: um certo endereço de memória estará sempre associado à uma mesma entrada da cache, embora uma mesma entrada seja compartilhada com n outros endereços. Sempre que o processador solicita a leitura de uma palavra, a *tag* de seu endereço é comparada com a *tag* armazenada na entrada da cache para a qual este endereço é mapeado. Caso sejam iguais, a palavra requisitada está presente na cache e é devolvida para o processador, o que se chama de acerto na cache (*cache hit*). Caso sejam diferentes, significa que a palavra não se encontra na cache e é buscada do próximo nível da hierarquia de memória, o que caracteriza uma falta na cache (*cache miss*).

Por conta desse comportamento de difícil previsibilidade (pois depende do padrão de acesso), as memórias cache são evitadas em sistemas de tempo real sob restrições rígidas (*hard real-time constraints*)

ou requerem análise cuidadosa com ferramentas sofisticadas para não se superestimar o pior caso de tempo de execução (*worst case execution time — WCET*) mas obter limites superiores seguros.

Além disso, o hardware extra necessário ao gerenciamento implícito de conteúdo da cache (*tag*, comparadores e multiplexadores) consome uma certa quantia de energia, o que pode influenciar negativamente a eficiência energética do sistema embarcado. Esta energia adicional pode ser evitada com outros tipos de memória de gerenciamento explícito de seu conteúdo, como as memórias de rascunho, mostradas abaixo.

MEMÓRIA DE RASCUNHO (SPM)

Uma *memória de rascunho* (*Scratchpad Memory* — SPM) ou memória fortemente acoplada (*Tightly Coupled Memory* — TCM) (Figura 3) é pequena, interna ao circuito integrado (*on-chip*), e construída com tecnologia SRAM, semelhantemente às caches. Contudo, ao contrário destas, possui um espaço de endereçamento próprio, disjunto da MP, e seu conteúdo é gerenciado explicitamente (pelo usuário), pois não há recursos adicionais de hardware.

Devido à ausência do arranjo de *tags*, dos comparadores e dos multiplexadores presentes nas caches, a SPM é mais eficiente em termos de área e energia. Todavia, o uso de SPM requer o gerenciamento explícito de seu espaço de endereçamento através de instrumentação do código. O projetista do sistema embarcado deve instrumentar o código através do uso de um *framework* de compilação com suporte à alocação em SPM, que pode ou não ser combinado com algum suporte para seu gerenciamento e/ou cópia do conteúdo — e.g. um controlador de SPM ou uma *Memory Management Unit* (unidade de gerenciamento de memória) (MMU).

A motivação para seu uso advém do fato que os sistemas embarcados executam aplicativos específicos (ou classes específicas de aplicações). Devido a especificidade dos aplicativos, é vantajoso gerenciar explicitamente seu conteúdo para obter uma redução significativa no consumo de energia e no tempo de execução, ainda que isto implique em perda de generalidade do software, que deve ser personalizado para cada sistema embarcado.

Além disso, como sua latência é fixa (geralmente um ciclo do processador), tem emprego certo em sistemas de tempo real, pois permitem o cálculo exato do WCET (ao contrário das caches).

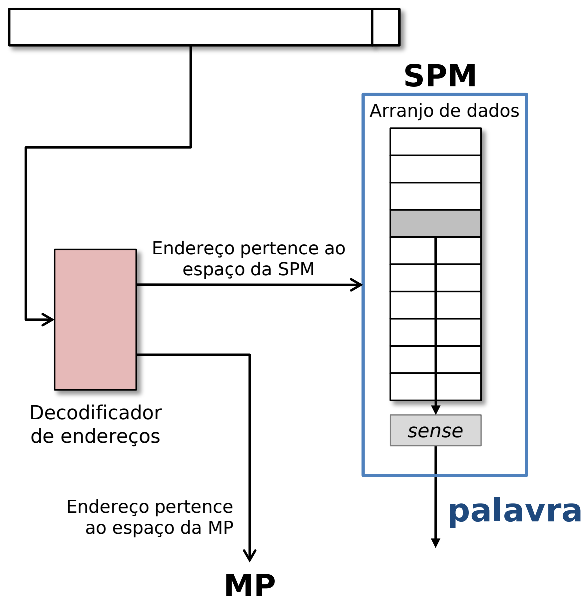


Figura 3: Exemplo de memória de rascunho (SPM)

A alocação de trechos de programa em SPM é conduzida de modo a otimizar um determinado objetivo — geralmente a redução do consumo de energia, de tempo de execução ou de ambos. Ao contrário das caches, a exploração de SPMs é uma área de pesquisa mais recente. Não se pode afirmar que existe uma técnica de alocação dominante, pois depende da aplicação-alvo e da arquitetura do subsistema de memória.

As técnicas de alocação em SPM são divididas costumeiramente em duas classes, de acordo com a abordagem de alocação utilizada: *non-overlay-based* (NOB) e *overlay-based* (OVB). Na primeira, os trechos de programa são alocados em SPM no início da execução da aplicação e são mantidos lá até o término da execução. Na segunda, os trechos presentes em SPM são alterados em tempo de execução, i.e. durante a execução do programa, a alocação de um trecho pode ocupar o mesmo espaço previamente alocado para outro trecho. A alocação de código e dados em SPM, as abordagens de alocação (estas duas classes) e as diversas técnicas propostas na literatura serão abordadas com mais profundidade no Capítulo 2.

1.2.2 Arquiteturas para o subsistema de memória

Dados os componentes de memória descritos acima e tendo sempre como base a presença de SPMs, centro deste trabalho, pode-se imaginar três principais arquiteturas para um subsistema de memória, como retratadas pelas Figuras 4(a), 4(b) e 4(c).

ARQUITETURAS SOMENTE COM CACHES (FCAS)

Arquiteturas somente com caches (*fully-cached architectures* — FCAs), ilustradas pela Figura 4(a), não possuem SPMs, sendo compostas apenas por um ou mais níveis de caches entre o processador e a MP. As caches podem ser unificadas (numa mesma cache são colocados instruções e dados) ou separadas (uma cache para instruções e outra para dados). Eventualmente, uma delas pode estar ausente — e.g. um subsistema apenas com cache de instruções e MP, sem cache de dados.

ARQUITETURAS SEM CACHE (UNAS)

Arquiteturas sem cache (*uncached architectures* — UNAs), ilustradas pela Figura 4(b), possuem uma ou mais SPMs, localizadas em espaço de endereçamento disjunto da MP. Pode-se pensar nas UNAs como arquiteturas onde a SPM substitui a cache, seja por conta de sua melhor previsibilidade em sistemas de tempo-real, seja pelo menor consumo de área ou pela eficiência energética.

ARQUITETURAS BASEADAS EM CACHE (CBAS)

Arquiteturas baseadas em cache (*cache-based architectures* — CBAs), ilustradas pela Figura 4(c), possuem cache(s) e SPM trabalhando conjuntamente. Como nas UNAs, a SPM é localizada num espaço de endereçamento próprio, porém a cache continua amenizando os acessos à MP. Por conta disso, mesmo quando parte do código ou dos dados não é alocado em SPM, o sistema se beneficia da presença de cache para a redução do consumo de energia e tempo de execução, diferentemente das UNAs.

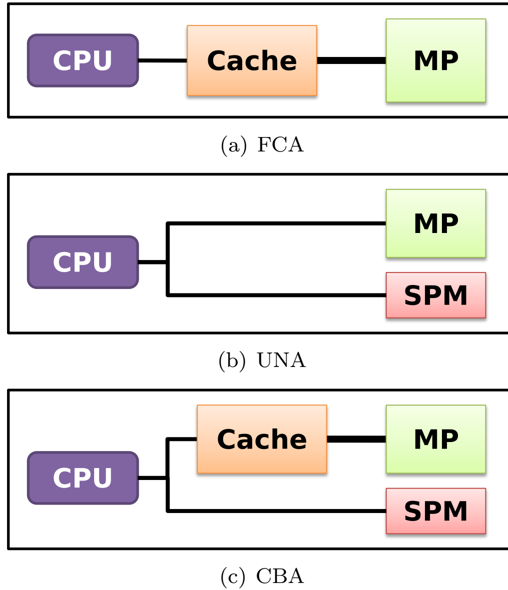


Figura 4: Arquiteturas-alvo possíveis para o subsistema de memória

1.3 ESCOPO DESTE TRABALHO

Nos últimos 6 anos, a abordagem *overlay-based* (OVB) tem dominado o cenário das técnicas de alocação em SPM. Aparentemente, a aptidão destas técnicas em explorar dinamicamente as propriedades de programa viabiliza maiores economias, pois permite que um determinado trecho de programa seja acessado em SPM somente enquanto promove economia, e, quando não mais, ceda seu lugar para outros trechos que permitam maiores lucros.

Contudo, uma análise mais atenta das técnicas OVB propostas na literatura revela fraquezas que podem ser eventualmente contornadas por técnicas *non-overlay-based* (NOB). Quando operam em tempo de compilação, as técnicas OVB mostram-se superiores às NOBs equivalentes. Porém, a manipulação de arquivos-fonte, mandatória em técnicas de tempo de compilação, inviabiliza a alocação de trechos contidos em bibliotecas, cujo código-fonte dificilmente é disponibilizado.

Para ser capaz de incluir bibliotecas no espaço de otimização, uma técnica qualquer — independente de OVB ou NOB — deve operar em tempo de pós-compilação, manipulando arquivos binários. Neste caso,

as técnicas OVB apresentam complicadores: necessitam de hardware dedicado, o que não é factível em certos sistemas, ou se limitam a alocar apenas código, desprezando dados. As técnicas NOB, contudo, mostram-se mais naturais para arquivos binários, alocando facilmente código e dados de bibliotecas sem a necessidade de recursos adicionais de hardware.

Assim, o presente trabalho apresenta uma reavaliação experimental da abordagem NOB, de modo a verificar qual a eficácia destas técnicas mais simples na redução do consumo de energia do subsistema de memórias, quando comparadas com suas correspondentes OVBs, aparentemente superiores.

Para uma comparação mais justa da economia obtida pelas técnicas, identificaram-se dois fatores com considerável influência: a arquitetura-alvo e a configuração das caches.

Observou-se que as diversas técnicas de alocação em SPM propostas na literatura consideram arquiteturas-alvo distintas, principalmente quanto à presença (CBAs) ou não (UNAs) de caches, e tratam este como um quesito secundário, desprezando o fato de que o subsistema de memória é extremamente sensível à configuração das memórias cache. Ignorando isto, tais técnicas às vezes estabelecem uma comparação direta com uma técnica proposta para uma arquitetura diferente da sua, superestimando seus resultados — por exemplo, Egger et al. (2006) (UNA) compara com Angiolini et al. (2004) (CBA) e relata ganhos de 24% em redução de energia, o que não é justo pois a presença da cache em uma CBA diminui o potencial de otimização da SPM, favorecendo os ganhos da UNA.

A análise das técnicas de alocação em SPM mais recentes também evidencia uma preferência por CBAs. Esta escolha é totalmente plausível, dado que a combinação de caches com SPMs está se tornando cada vez mais comum em sistemas embarcados (MALIK; MOYER; CERMAK, 2000; TALLA; GOLSTON, 2007). Logo, para fins de um subsistema de memória mais realista e de uma comparação mais justa com as técnicas mais recentes, este trabalho adota uma CBA como sua arquitetura-alvo[1].

Assim, o presente trabalho argumenta que a configuração arbitrária ou *ad hoc* das caches de referência[2] conduz a resultados superes-

[1]Tal escolha trata-se somente de limitação de escopo, uma vez que a técnica utilizada neste trabalho pode ser facilmente aplicada sobre UNAs.

[2]Uma cache de referência pertence à chamada *arquitetura de referência*. Trata-se de uma arquitetura cujo consumo de energia é comumente utilizado como referência para a comparação dos resultados obtidos pela arquitetura-alvo. No caso das técnicas de alocação em SPM, geralmente as arquiteturas de referência não possuem SPMs e a arquitetura-alvo é gerada adicionando-se uma SPM à arquitetura de referência.

timados de economia. Para superar tal inexatidão na literatura, este trabalho reporta resultados para caches pré-ajustadas (i.e., ajustada previamente à alocação em SPM) para cada programa de *benchmark* da seguinte maneira:

1. Por meio da técnica de ajuste-fino (*cache-tuning*) proposta por Viana (2006), são identificadas as caches de instruções e de dados de maior eficiência energética, para comporem a arquitetura de referência;

2. Estima-se o tamanho de uma "cache pré-ajustada equivalente unificada", o qual servirá como base para o dimensionamento da SPM. Com isso, pretende-se evitar que uma SPM de tamanho fixo e arbitrário influencie os resultados, da mesma forma como as caches não-ajustadas relatadas na literatura influenciam;

3. Utilizando uma técnica NOB, realiza-se a alocação de código e dados para tamanhos distintos de SPM, escolhidos como submúltiplos do tamanho da cache pré-ajustada equivalente;

4. Finalmente, os valores de energia obtidos são normalizados com relação à arquitetura de referência, o que permite avaliar a economia com respeito às caches pré-ajustadas.

Os resultados apresentados mostram que a economia obtida em CBAs pelas técnicas NOB que manipulam binários para a inclusão de bibliotecas no espaço de otimização pode ser tão boa quanto a obtida pela abordagem OVB (a qual, inclusive, é mais complexa). Trata-se de evidência experimental sólida, inferida sobre um conjunto significativo de 20 programas de *benchmark* que totalizam 240 casos avaliados, gerados por meio de variação do tamanho da SPM e da granularidade para divisão do código do programa. Tanto o número de programas como de casos avaliados é bem superior à maioria dos relatados pelos demais trabalhos publicados em alocação de SPM (de todos os trabalhos citados na bibliografia somente o de Falk e Kleinsorge (2009) apresenta resultados para um maior número de programas).

Estes resultados também traçam diretrizes para a identificação do tamanho ótimo de SPM (em termos de economia de energia), por meio da correlação dos resultados obtidos com SPMs cujo tamanho é parametrizado com relação à cache pré-ajustada equivalente.

Para a alocação de trechos dos programas em SPM, utilizou-se a técnica NOB de Mendonça (2009, 2010), que, dentre as técnicas até então propostas, é a que considera o maior espaço de otimização:

código e dados globais, incluindo aqueles oriundos de bibliotecas. Esta técnica opera em tempo de pós-compilação manipulando arquivos-objeto relocáveis, e considera trechos de código somente na granularidade de procedimentos. Sobre esta técnica realizou-se uma extensão que permite o suporte à alocação de código na granularidade de blocos básicos, como alternativa à granularidade de procedimentos. Entretanto, não é permitida uma granularidade mista (procedimentos e blocos básicos simultaneamente).

De maneira análoga à variação do tamanho da SPM, a variação da granularidade de código permite que sejam traçadas diretrizes para a identificação da melhor granularidade para um certo tamanho de SPM.

Os resultados aqui descritos, para uma CBA considerando caches pré-ajustadas, foram resumidos em artigo submetido ao *Symposium on Integrated Circuits and Systems Design* — SBCCI 2011 (VOLPATO et al., 2011). Os resultados tratando somente da extensão da técnica de Mendonça et al. foram publicados nos anais do *IEEE Computer Society Annual Symposium on VLSI* — ISVLSI 2010 (VOLPATO et al., 2010).

1.4 PRINCIPAIS CONTRIBUIÇÕES

Como **contribuições técnicas** deste trabalho, pode-se elencar:

- Implementação da técnica de ajuste-fino de memórias cache de Viana (2006), adaptada à infraestrutura experimental.
- Extensão da técnica de Mendonça (2009, 2010) para prover suporte à granularidade de blocos básicos (alternativamente à granularidade de procedimentos).
- A noção de cache equivalente unificada para correlacionar as capacidades da SPM e das caches determinadas via ajuste-fino.

O presente trabalho também apresenta as seguintes **contribuições científicas**:

- Reavaliação experimental, sob uma perspectiva de pré-ajuste da cache, da abordagem *non-overlay-based* (NOB), que aparentemente estaria suplantada pela abordagem *overlay-based* (OVB), a julgar por uma análise superficial da literatura.
- Evidência experimental sólida (baseada em um número de programas e de casos bem superior à maioria dos trabalhos correlatos)

de que a abordagem NOB deveria ser adotada na construção de alocadores para SPM capazes de considerar elementos de biblioteca, no contexto de sistemas que não podem dispor de hardware de suporte especializado.

1.5 ORGANIZAÇÃO DESTA DISSERTAÇÃO

O restante desta dissertação está organizado como segue.

O Capítulo 2 apresenta em mais detalhes o processo de alocação de trechos de programa em SPM, seu propósito e a imensa variedade de características relacionadas com as técnicas de alocação. Em seguida, descreve-se o estado-da-arte em alocação para SPM, e fazem-se considerações sobre os principais trabalhos correlatos e suas características.

No Capítulo 3 são revistos conceitos fundamentais para a formulação do problema-alvo.

O Capítulo 4 descreve a extensão proposta por este trabalho à técnica NOB de Mendonça (2009, 2010), com o intuito de permitir a escolha da granularidade de blocos básicos como alternativa à de procedimentos.

O Capítulo 5 aborda a necessidade do ajuste-fino de caches para cada programa, antes da alocação em SPM. Também descreve o método de ajuste-fino utilizado como infraestrutura para este trabalho e detalhes de sua implementação. Por fim, apresenta os resultados do ajuste-fino para um conjunto de programas de *benchmark* e, a partir destes, o cálculo de uma cache equivalente (unificada), para servir de referência ao dimensionamento da SPM.

No Capítulo 6 são abordados a configuração experimental utilizada, o procedimento utilizado na geração dos experimentos, e a validação experimental da técnica.

O Capítulo 7 mostra as conclusões deste trabalho, bem como algumas perspectivas para trabalhos futuros.

2 ALOCAÇÃO EM MEMÓRIAS DE RASCUNHO

Este capítulo trata em detalhes da alocação em memórias de rascunho (SPMs). Primeiramente, é oferecida uma visão geral do processo de alocação, seu propósito e a descrição das etapas que usualmente compõem as técnicas de alocação em SPM. Em seguida são definidas as diversas características que dizem respeito à alocação: os elementos de programa, sua origem, tipo e granularidade; a fase e a abordagem de alocação. Diversas técnicas propostas na literatura são classificadas quanto a estas características, o que é sumarizado na forma de tabelas. Dentre estas técnicas, enfatizam-se aquelas que são o foco desta dissertação: técnicas que manipulam arquivos binários (em tempo de pós-compilação), e, portanto, são capazes de considerar código encapsulado em bibliotecas. O estado-da-arte destas técnicas é exposto em maiores detalhes. Finalmente, são feitas considerações sobre a alocação em SPM, principalmente do ponto de vista da característica denominada abordagem de alocação.

2.1 VISÃO GERAL DO PROCESSO DE ALOCAÇÃO

A alocação de trechos de programa em SPM tem o objetivo de otimizar o sistema embarcado no seu consumo de energia, tempo de acesso, ou ambos. Estes são os objetivos mais usuais, pois, como discutido na Seção 1.2.1, SPMs possuem melhor eficiência energética e de desempenho do que outros componentes do subsistema de memória, como as caches.

Atingir estes objetivos exige que o software seja modificado de modo que parte dos endereços a serem acessados recaiam na SPM, a qual apresenta menor tempo de acesso e menor consumo de energia por acesso que os demais componentes do subsistema de memória. Para tanto, uma imensa gama de técnicas para alocação em SPM foram propostas. Apesar de bem variadas em suas características (conforme será explicado na Seção 2.2), é possível identificar etapas comuns à grande maioria destas técnicas.

Um fluxo de trabalho genérico para técnicas de alocação em SPM é ilustrado na Figura 2.1. Os pontos sólidos indicam o início e o fim do fluxo, suas entradas e saídas estão representadas por elipses e os retângulos simbolizam as etapas do processo.

As entradas da técnica são o conjunto de arquivos do programa

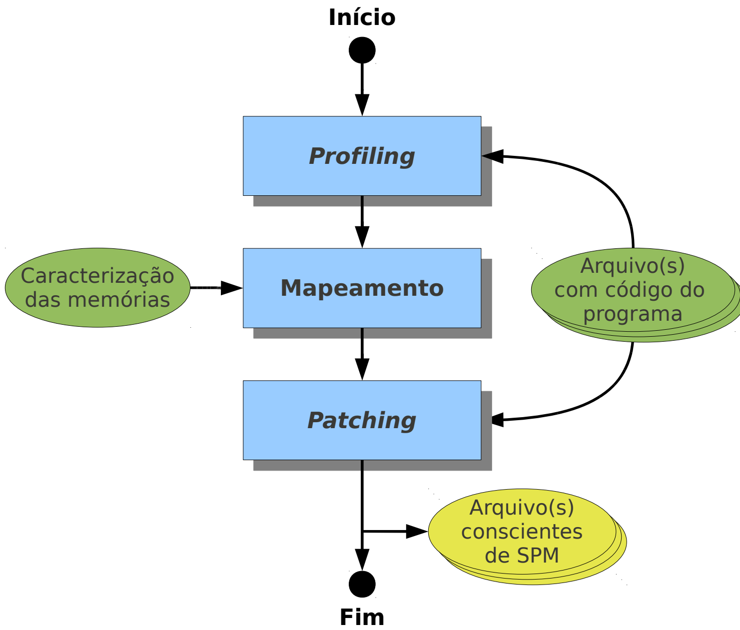


Figura 5: Fluxo de trabalho genérico para técnicas de alocação em SPM

(em formato de código-fonte ou binário) e as características das memórias que compõem o subsistema de memória do sistema embarcado alvo. Este conjunto de características geralmente é composto pela capacidade, pelo tempos de acesso e pela energia por acesso das memórias. Para tornar este exemplo mais realista, considere-se que o objetivo da otimização desejada consiste na redução do consumo de energia, ou seja, fazem-se também necessários os valores de energia por acesso para cada componente de memória.

Para otimizar o software de modo a reduzir o consumo de energia induzido por sua execução, deve ser traçado um perfil (*profile*), identificando a contribuição energética de cada trecho do programa. Isto ocorre na primeira etapa da técnica, denominada de ***profiling***.

O meio mais frequente para realização do *profiling* é via simulação: compilam-se os arquivos de entrada, caso estejam em formato de código-fonte, e submete-se o binário executável a um simulador do processador-alvo, obtendo-se a lista de todos os endereços de instruções e dados acessados. Por meio desta lista, denominada *trace*, calcula-se — analiticamente ou por meio de um simulador do subsistema de memória — o gasto energético de cada trecho do programa.

Nestas técnicas, conhecidas como *profile-driven*, o resultado final deriva muito da representatividade das entradas utilizadas para *profiling*. O uso de entradas com um padrão de acesso muito diferente das entradas reais mais frequentes da aplicação a ser otimizada tende a não trazer ganhos elevados para a maioria dos casos.

A segunda etapa é o **mapeamento**, o qual consiste em determinar quais trechos do programa serão alocados para SPM. Utilizando as características das memórias do sistema-alvo — em particular da SPM — e as informações obtidas pelo *profiling*, calcula-se o lucro (*profit*) obtido da alocação de cada trecho de programa para a SPM (a redução do consumo de energia), bem como o custo desta alocação (neste exemplo, a quantidade de espaço de armazenamento da SPM a ser gasto).

A etapa de mapeamento procura sempre identificar o conjunto dos trechos de programa que, quando alocado em SPM, melhor satisfaça o objetivo da otimização. De modo simplificado, pode-se dizer que os maiores lucros serão obtidos quando alocados para SPM os trechos de programa com maior gasto energético e menor tamanho. Também é preciso salientar que esta seleção ocorre sempre dentro do espaço de otimização da técnica, e que não foi proposta até o presente momento uma técnica que considere todo o código (instruções e dados) da aplicação como candidato à otimização. Por exemplo, acessos a trechos de pilha podem estar fora do alcance de uma técnica (JANAPSATYA;

IGNJATOVIĆ; PARAMESWARAN, 2006b) e não serão alocados, independentemente de seu consumo energético. Já uma outra técnica (AVISSAR; BARUA; STEWART, 2002) pode conseguir alocar estes trechos, mas é incapaz de alocar trechos de código, como a primeira.

Na última etapa tem lugar o ***patching*** dos arquivos de entrada necessários, de modo a refletir o resultado da etapa de mapeamento. Os arquivos de entrada, independentemente de seu formato, são modificados para que os elementos mapeados para SPM sejam de fato copiados para o espaço de endereçamento desta memória quando da execução do programa.

Normalmente, o *patching* consiste no acréscimo de instruções no código, quando se trabalha com código-fonte, ou na cópia ou relocação de trechos de código, quando as entradas são arquivos binários. Por fim, pode ainda ser necessário executar novamente o compilador ou o linkeditor, para gerar o arquivo binário otimizado para SPM.

Para verificar os resultados obtidos pelo processo de alocação, costuma-se submeter este binário consciente de SPM a um simulador do processador e do subsistema de memória-alvo, utilizando os mesmos arquivos de entrada usados no *profiling* ou variando-os.

O fluxo de trabalho proposto na presente dissertação é uma especialização do fluxo apresentado na Figura 2.1 e será detalhado no Capítulo 4.

2.2 CARACTERÍSTICAS DAS TÉCNICAS DE ALOCAÇÃO

As técnicas de alocação para SPMs são influenciadas por uma série de características relacionadas à aplicação, aos elementos de programa, aos arquivos de entrada considerados e, finalmente, ao próprio processo de alocação. Estas características têm impacto direto nos ganhos obtidos pela alocação, motivo pelo qual se faz necessário entendê-las para bem explorar suas potencialidades, descobrir suas limitações e maneiras de sobrepô-las.

As principais características que influenciam as técnicas de alocação para SPM são:

- que elementos de programa são considerados candidatos à alocação;
- a origem, tipo e granularidade destes elementos;
- a fase em que ocorre a alocação dos elementos para SPM, se em tempo de compilação ou de pós-compilação;

- o tipo de arquivo de entrada utilizado (fonte, objeto ou executável);
- finalmente, a abordagem de alocação — se existe relocação de elementos durante a execução, ou se os elementos são copiados para a SPM no começo da execução e ali permanecem até seu término.

Estas características são descritas e exemplificadas nas seções posteriores.

2.2.1 Elementos de programa

Uma das principais características das técnicas de alocação para SPMs diz respeito aos elementos de programa considerados como candidatos para alocação. Por elementos de programa consideram-se trechos de um programa — sejam de código ou de dados — que possuem um significado lógico. Assim, por exemplo, tratando-se do código da aplicação, pode-se considerar como elementos de programa trechos que correspondam a procedimentos. Já tratando-se dos dados, pode-se considerar como elementos de programa trechos que correspondam a variáveis e estruturas de dados.

Elementos de programa podem ser caracterizados de acordo com seu tipo e sua origem. Neste sentido, quanto maior a abrangência dos elementos considerados no que diz respeito ao seu tipo e origem, maior o percentual de código da aplicação candidato à alocação, e portanto, maior a possibilidade de maximizar os ganhos decorrentes da alocação.

Por simplicidade, a partir deste ponto elementos de programa serão referenciados simplesmente como **elementos**.

2.2.1.1 Tipo dos elementos

Quanto ao tipo, os elementos podem ser divididos em elementos de código — grupos de instruções a serem executadas pelo processador — e elementos de dados.

Elementos de código são suportados pela grande maioria das técnicas. De fato, de todas as técnicas apresentadas aqui, apenas as de Kandemir et al. (2001), Avissar, Barua e Stewart (2002), Cho et al. (2007) e Deng et al. (2009) não suportam este tipo de elemento.

Por outro lado, os **elementos de dados** costumam ser categorizados, para fins de alocação em SPM, em globais escalares, globais

não-escalares, elementos de pilha e elementos de *heap*. São poucas as técnicas que não apresentam nenhum suporte para este tipo de elemento (STEINKE et al., 2002a) (ANGIOLINI et al., 2004) (JANAPSATYA; PARAMESWARAN; IGNJATOVIC, 2004) (JANAPSATYA; IGNJATOVIĆ; PARAMESWARAN, 2006b) (EGGER; LEE; SHIN, 2006, 2008) (EGGER et al., 2006, 2010).

Elementos de dados globais escalares e/ou não-escalares são considerados pelas técnicas de Kandemir et al. (2001), Steinke et al. (2002b), Avissar, Barua e Stewart (2002), Verma, Wehmeyer e Marwedel (2004b), Udayakumaran, Dominguez e Barua (2006), Cho et al. (2007), Mendonça (2009, 2010) e Deng et al. (2009).

Elementos de pilha são manuseados pelas técnicas propostas por Avissar, Barua e Stewart (2002), Verma, Wehmeyer e Marwedel (2004b), Udayakumaran, Dominguez e Barua (2006), Cho et al. (2007) e Deng et al. (2009).

Finalmente, elementos de *heap* são tratados pelas técnicas propostas por McIlroy, Dickman e Sventek (2008) e Deng et al. (2009).

2.2.1.2 Origem dos elementos

Quanto à sua origem, os elementos são classificados em elementos de aplicação e elementos de biblioteca.

Elementos de aplicação são de responsabilidade do desenvolvedor do software sistema embarcado. Todas as técnicas consideram elementos desta origem, pois o mínimo que os desenvolvedores de software dispõem é do código-fonte e/ou arquivos-objeto do programa.

Elementos de biblioteca geralmente são de responsabilidade de terceiros, que normalmente não disponibilizam seu código-fonte, mas somente uma biblioteca contendo arquivos-objeto pré-compilados. As únicas técnicas capazes de tratar elementos de dados provenientes de bibliotecas são as de Cho et al. (2007), Mendonça (2009, 2010) e Deng et al. (2009). Quando aos elementos de código, conseguem tratá-los as técnicas de Angiolini et al. (2004) Mendonça et al. (2009) Janapsatya, Parameswaran e Ignjatovic (2004) Janapsatya, Ignjatović e Parameswaran (2006b) Egger, Lee e Shin (2008) Egger et al. (2010).

Tipo e origem são características independentes, i.e. uma técnica que considera elementos de bibliotecas pode considerar somente elementos de código ou de dados, ou mesmo ambos.

2.2.2 Granularidade dos elementos

Os elementos, sejam eles de código ou de dados, podem ser divididos de acordo com diferentes fronteiras. É mais usual que estas sejam fronteiras naturais do próprio tipo do elemento, como procedimentos ou blocos básicos para códigos, e, vetores, no caso de dados. No entanto, nada impede que sejam utilizadas fronteiras diferentes, como um conjunto de instruções menor do que um bloco básico ou pedaços de um vetor. Definida a fronteira, a decomposição do programa em elementos dará origem a elementos de granularidade mais fina ou mais grossa. Estas duas granularidades podem ser exemplificadas pelas fronteiras de blocos básicos e procedimentos, respectivamente, para elementos de código.

2.2.2.1 Granularidade de código

A granularidade dos elementos de código pode ser classificada em: granularidade de procedimentos, de blocos básicos, de blocos de instruções (ou blocos lógicos) — um sub-bloco dentro de um bloco básico — e de *traces* (um conjunto de blocos básicos subsequentes).

A **granularidade de procedimentos** é possivelmente a mais natural para os elementos de códigos. A divisão dos elementos é simples e direta, resultando num mapeamento de um-para-um entre procedimentos e elementos. O *patching* de um elemento sob esta granularidade é simples, bastando o ajuste do desvio das instruções que chamam o procedimento, para que este aponte para o novo endereço no qual o procedimento residirá no espaço da SPM. Não se fazendo necessário o acréscimo de instruções extras, a granularidade não induz nenhum *overhead* de energia ou de espaço.

Exemplos de técnicas que adotam esta granularidade são Steinke et al. (2002a), Steinke et al. (2002b), Udayakumaran, Dominguez e Barua (2006), Mendonça (2009, 2010) e Egger et al. (2010).

A motivação para alocação de granularidades de código mais finas advém de duas razões. Primeiro, podem existir procedimentos com alta taxa de invocação, os quais são maiores em tamanho do que a capacidade da SPM. Em segundo lugar, o Princípio de Pareto (também conhecido como Princípio 80-20) indica, de forma geral, que 80% dos efeitos provém de 20% das causas (JURAN, 1975). Aplicado à otimização de software, este princípio indica que o foco das otimizações deve ser os *hot spots*, pequenos trechos de código responsáveis pela maioria do

tempo de execução gasto para rodar o programa (JENSEN, 2008). Ou seja, mesmo que um procedimento altamente invocado caiba em SPM, uma parcela muito pequena do seu código (frequentemente um laço) pode ser a responsável por grande parte dos acessos. A alocação desse procedimento como um todo em SPM pode diluir o ganho que a alocação desse elemento traz. Nestas duas circunstâncias, uma granularidade de código mais fina pode aumentar os ganhos decorrentes da alocação em SPM.

A **granularidade de blocos básicos** consiste na divisão dos procedimentos em unidades menores, denominadas blocos básicos. Um bloco básico (BB) é definido por Muchnick (1997) como “a máxima sequência de instruções da qual se pode entrar apenas pela primeira delas e sair apenas pela última delas”. A determinação dos BBs é feita por meio da identificação das primeiras instruções que os compõem, denominadas líderes. Um líder pode ser:

1. O ponto de entrada de uma rotina;
2. O alvo de um desvio (condicional ou incondicional);
3. A instrução que segue um desvio ou retorno de função.

O terceiro ponto citado anteriormente indica que um BB é determinado sempre no escopo de procedimentos, de modo que um BB nunca pode ultrapassar a fronteira do procedimento ao qual pertence.

A alocação de um BB consiste na cópia de seu conteúdo da memória principal (MP) para SPM e na modificação do seu ponto de entrada na MP (sua primeira instrução) por uma instrução de desvio incondicional, que transfira o fluxo de execução para o endereço do seu conteúdo no espaço de endereçamento da SPM. Dependendo do tipo de bloco básico, pode ser necessário inserir no seu fim uma instrução de desvio incondicional, para retornar o fluxo para a MP. Como o presente trabalho também investiga o impacto da granularidade dos elementos na redução de energia do subsistema de memória, a alocação de BBs em SPM será tratada em mais detalhes no Capítulo 4.

A vantagem da granularidade de BBs consiste na possibilidade de isolar os pontos mais acessados de um procedimento e alocá-los separadamente. Desta forma, pode-se evitar desperdício do limitado espaço em SPM com código pouco acessado. Sua desvantagem, porém, é que as instruções adicionadas (a instrução de desvio incondicional que invoca seu código na SPM e, quando necessário, a instrução de retorno para MP) causam *overhead* de tempo de execução e de consumo de

energia sempre, mesmo que o BB já esteja alocado na SPM desde o início da execução do programa, o que não acontece com procedimentos.

As técnicas de Banakar et al. (2002), Steinke et al. (2002b), Steinke et al. (2002a), Janapsatya, Parameswaran e Ignjatovic (2004), Janapsatya, Ignjatović e Parameswaran (2006b), Udayakumaran, Dominguez e Barua (2006) e Egger, Lee e Shin (2008) utilizam esta granularidade.

A **granularidade de blocos de instruções (ou blocos lógicos)** permite dividir os elementos em porções ainda menores do que um bloco básico: um bloco composto por, no mínimo, uma instrução. A alocação de blocos de instruções ocorre de maneira análoga à de BBs. Essa granularidade é motivada pelo seguinte cenário. Imagine que vários elementos já foram mapeados para SPM, restando um espaço muito pequeno. É possível que exista um BB muito acessado, porém maior do que o espaço disponível em SPM. Neste caso, a alocação de uma parcela desse BB tenderia a ser melhor do que a alocação de outro BB pouco acessado, mas com tamanho pequeno o bastante para ser alocado inteiramente na SPM. Porém, dado que blocos básicos já são geralmente muito pequenos (em média, 4 a 6 instruções em códigos de matemática inteira, sem instruções de ponto-flutuante (ROTENBERG; BENNETT; SMITH, 1999)), considerar blocos menores pode tornar o *overhead* de alocação proibitivo a ponto de predominar em relação ao ganho.

Como técnicas que usam esta granularidade, pode-se citar as técnicas de Angiolini et al. (2004) e Janapsatya, Parameswaran e Ignjatovic (2004).

A **granularidade de *traces*** é uma granularidade intermediária entre procedimentos e blocos básicos. Um *trace* é uma sequência linear de blocos básicos situados numa região contígua de memória. *Traces* melhoram o desempenho do processador aumentando a localidade espacial presente no programa. Devido à sua característica de sempre terminar com uma instrução de desvio incondicional (TOMIYAMA; YASUURA, 1996), *traces* compõem um bloco de instruções atômico e que pode ser alocado em outras regiões da memória sem a necessidade de alteração de outros *traces*. Dependendo das características da aplicação, a granularidade de *traces* pode conseguir uma otimização mais ou menos eficiente do que a granularidade de BBs. No cenário de um procedimento que concentra um enorme número de acessos em alguns BBs consecutivos ou muito próximos e que fazem parte de um mesmo *trace*, a alocação com granularidade de *trace* gerará um menor *overhead* de espaço do que a alocação de cada um destes blocos básicos isoladamente.

Exemplos de técnicas que utilizam esta granularidade são Verma, Wehmeyer e Marwedel (2004a), Verma, Wehmeyer e Marwedel (2004b)

e Ravindran et al. (2005).

Além das granularidades apresentadas aqui, algumas técnicas trabalham com **granularidade mista**, permitindo a alocação de elementos com mais de uma granularidade. Nestes casos, a técnica deve certificar-se de que não ocorram casos de alocação múltipla de um mesmo trecho de código sob granularidades diferentes (e.g. se um procedimento é alocado, nenhum dos BBs que o compõem deve ser alocado). As técnicas de Steinke et al. (2002b), Steinke et al. (2002a) e Udayakumaran, Dominguez e Barua (2006) lidam com elementos de granularidade de procedimentos e de blocos básicos ao mesmo tempo.

2.2.2.2 Granularidade de dados

Para dados escalares, não faz sentido falar em granularidade de dados. No entanto, quando se tratam de dados não-escalares (e.g. arranjos uni ou mesmo multidimensionais) é possível uma tentativa de classificação em granularidade plena e granularidade de blocos.

A **granularidade plena** considera os dados não-escalares sempre por completo, não ocorrendo alocação parcial dos dados, i.e., ou um dado está inteiramente na SPM ou está inteiramente em MP. Sua vantagem é a facilidade de implementação, pois é a fronteira mais natural para dados. No entanto, implicará na impossibilidade de alocação de dados maiores do que o tamanho da SPM. Esta é a abordagem utilizada pela maioria das técnicas, como, por exemplo, Steinke et al. (2002a), Avissar, Barua e Stewart (2002), Verma, Wehmeyer e Marwedel (2004b), Udayakumaran, Dominguez e Barua (2006), Cho et al. (2007), Deng et al. (2009) e Mendonça (2009, 2010).

Todavia, principalmente em sistemas embarcados no domínio de processamento de áudio e vídeo, com forte apelo comercial, aplicações baseadas no acesso a arranjos multi-dimensionais presentes no interior de laços aninhados são dominantes. Estes arranjos são frequentemente muito grandes e muito acessados, limitando consideravelmente os ganhos obtidos pela granularidade plena.

Para estes tipos de acessos, foi proposta a **granularidade de blocos (*tiles*) de dados**, que considera a divisão de um dado não-escalar em blocos menores e de igual tamanho (exceto eventualmente nas fronteiras do dado). Para permitir que os dados sejam transferidos em blocos, os laços aninhados são transformados utilizando uma técnica denominada de *loop tiling*, exemplificada da seguinte maneira. Considere-se uma matriz bidimensional. Ao invés do convencional acesso iterando

primeiramente sobre as linhas e depois sobre as colunas (ou vice-versa), a matriz é dividida em blocos e os laços são transformados de modo que se itere primeiro sobre os blocos, para depois se iterar sobre linha e coluna.

Esta granularidade melhora a localidade de dados e permite que dados extremamente grandes sejam alocados mesmo em SPMs muito pequenas. No entanto, por sua própria natureza, exige esta granularidade que os dados sejam copiados para a SPM em tempo de execução, o que acarreta um certo *overhead* de energia e tempo de acesso.

Exemplos destas técnicas são Kandemir et al. (2001), que trabalha com referências regulares a arranjos (cujo índice é computado diretamente), e Chen et al. (2006), que lida com referências irregulares a arranjos (cujo índice é calculado indiretamente, por exemplo, utilizando o valor de outro arranjo).

2.2.3 Fase de alocação

As técnicas também diferem quanto à fase na qual o processo de alocação para a SPM é realizado: em tempo de compilação ou em tempo de pós-compilação. A escolha da fase de alocação está diretamente associada com os arquivos de programa utilizados como entrada para a técnica.

As técnicas que operam **em tempo de compilação** necessitam obrigatoriamente do código-fonte da aplicação, operando ou no próprio código-fonte, ou no *frontend* do compilador. Por outro lado, as técnicas **em tempo de pós-compilação** operam justamente sobre os arquivos binários resultantes do processo de compilação. Não há dependência da linguagem de programação ou de compilador, mas sim da linguagem de máquina para a qual o binário foi gerado. Estas técnicas podem trabalhar sobre arquivos binários(executáveis ou arquivos-objeto relocáveis). A vantagem deste formato de arquivo sobre aquele é permitir que elementos de dados sejam mais facilmente considerados no espaço de candidatos à otimização devido à simplicidade de identificação e *patching* dos mesmos (MENDONÇA, 2010).

A relação entre o arquivo de entrada e a fase de alocação da técnica também influencia diretamente os ganhos obtidos com a otimização. A não-disponibilidade do código-fonte das bibliotecas de terceiros nas técnicas de tempo de compilação reduz consideravelmente o espaço de elementos candidatos à otimização (MENDONÇA, 2010), tornando

impossível a alocação de elementos de bibliotecas por estas técnicas. Isto não ocorre nas técnicas de tempo de pós-compilação, uma vez que o arquivo binário das bibliotecas está disponível.

2.2.4 Abordagem de alocação

Existem essencialmente duas abordagens para efetuar o gerenciamento do conteúdo da SPM: *non-overlay-based* (NOB) e *overlay-based* (OVB). A abordagem NOB mantém os mesmos elementos na SPM ao longo de toda a execução da aplicação, enquanto a abordagem OVB modifica o conteúdo da SPM ao longo da execução, por meio da cópia de elementos de código e/ou dados para a SPM. Apesar de uma nomenclatura bastante difundida denominar estas duas abordagens de "alocação estática" e "alocação dinâmica", respectivamente, ao longo desta dissertação não será utilizada tal nomenclatura para evitar ambiguidade com a alocação de dados estáticos e de dados dinâmicos na memória.

Na abordagem ***non-overlay-based*** **(NOB)** os elementos mapeados para a SPM são copiados para a mesma em tempo de carga da aplicação para a memória — processo este que, terminado, dará sequência ao início da execução —, e permanecem na SPM durante toda a execução do programa.

Para melhor entender o conceito de alocação NOB, faz-se necessário compreender como acontece a carga de um programa para memória, a fim de que possa ser executado. Antes do início da execução de uma aplicação, seus diversos elementos devem ser carregados para os segmentos de memória apropriados. Tome-se como exemplo a arquitetura MIPS (PATTERSON; HENNESSY, 2008), para a qual um possível mapa de memória é descrito na Figura 6. Os elementos de código (geralmente contidos na seção `.text` do arquivo binário) são carregados para o segmento de texto, que começa em `0x10`, e os elementos de dados estáticos (geralmente presentes nas seções `.data` e `.bss`) são carregados para o segmento de dados estáticos, que começa em `0x10000`. Sucede imediatamente a este segmento o de dados dinâmicos (*heap*) — a serem alocados durante a execução da aplicação. Por último, o segmento de pilha, que inicia no final do espaço de endereçamento do programa (`0x3ffffc`) e cresce em sentido oposto ao anterior. Estes dois últimos segmentos não possuem tamanho fixo, pois é impossível prever a quantidade de memória necessária para estes dados em tempo de execução.

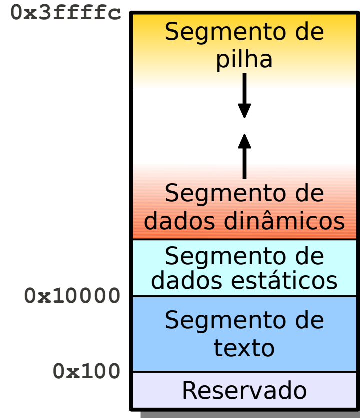


Figura 6: Mapa de memória da arquitetura MIPS (PATTERSON; HENNESSY, 2008)

No caso de uma aplicação consciente de SPM com alocação NOB, os elementos de código e/ou dados mapeados para SPM estarão contidos em uma seção própria (e.g. `.spm`), a ser carregada para o segmento de SPM. Este segmento possui tamanho fixo (no máximo igual à capacidade da SPM, mas eventualmente menor caso a SPM não seja inteiramente ocupada na alocação) e deve estar mapeado para um espaço de endereçamento disjunto (exclusivo) da MP. Neste exemplo do MIPS, pode-se imaginar que este segmento encontrar-se-ia após o segmento de pilha, começando em `0x400000`.

A alocação NOB não gera nenhum *overhead* em termos de hardware dedicado ou de espaço de memória e consumo de energia em decorrência da adição de instruções para cópia de elementos para SPM em tempo de execução. No entanto, no caso de programas grandes e bem modulares (cujo código possui padrões de acesso que se concentram em determinadas regiões de memória, caracterizando etapas distintas e com pouco código compartilhado entre elas), a eficácia da técnica tende a diminuir, pois seriam alocados elementos que seriam necessários apenas para uma determinada etapa da execução e depois poderiam ser removidos da SPM, liberando espaço para elementos das próximas etapas.

Como exemplos de trabalhos que utilizam a abordagem de alocação NOB, pode-se citar: Panda, Dutt e Nicolau (2000), Banakar et al. (2002), Steinke et al. (2002b), Verma, Wehmeyer e Marwedel (2004a),

Angiolini et al. (2004) e Mendonça (2009, 2010).

A abordagem ***overlay-based* (OVB)**, por sua vez, aloca os elementos mapeados para SPM em tempo de execução do programa. A motivação para tal é proporcionar uma melhor captura dos padrões de acesso da aplicação, alocando um determinado elemento na SPM apenas enquanto ele é extremamente requisitado, e removendo-o quando não mais, dando espaço para outros elementos requisitados. Neste sentido, pode-se pensar na SPM como uma "cache controlada por software" (CHERITON et al., 1988).

Nestas técnicas, a alocação de um determinado elemento consiste na cópia de seu conteúdo da MP para a SPM. Já a desalocação consiste na simples sobreposição do seu conteúdo por outros elementos, quando se tratam de elementos de código ou de dados constantes (somente para leitura). Quando se tratam de dados modificáveis, a sobreposição deve ser precedida pela cópia do conteúdo do elemento da SPM de volta para a MP, para que o conteúdo atualizado do elemento não se perca.

Esse processo de alocação e desalocação de elementos para a SPM repete-se diversas vezes durante a execução do programa. Vale a pena frisar que, apesar de a alocação acontecer durante a execução do programa, a escolha (mapeamento) dos elementos a serem alocados é feita na grande maioria das vezes em tempo de compilação ou pós-compilação, exceto por Deng et al. (2009), que a realiza em tempo de execução.

A cópia dos elementos para SPM pode acontecer por meio:

- da inserção de instruções convencionais de `load/store` no código (KANDEMIR et al., 2001) (STEINKE et al., 2002a) (VERMA; WEHMEYER; MARWEDEL, 2004b) (RAVINDRAN et al., 2005) (UDAYAKUMARAN; DOMINGUEZ; BARUA, 2006) (EGGER et al., 2006, 2010);

- de hardware extra, frequentemente controlado por instruções personalizadas, que precisam ser inseridas no código (CHO et al., 2007) (JANAPSATYA; PARAMESWARAN; IGNJATOVIC, 2004) (JANAPSATYA; IGNJATOVIĆ; PARAMESWARAN, 2006b);

- de funções de gerenciamento de memória dinâmica conscientes de SPM, como `malloc` e `free` da biblioteca `libc`, padrão da Linguagem C (DOMINGUEZ; UDAYAKUMARAN; BARUA, 2005) (MCILROY; DICKMAN; SVENTEK, 2008).

As técnicas de alocação em SPM e suas características apresentadas nesta seção encontram-se sumarizadas nas Tabelas 1 e 2. A Tabela 1 classifica as técnicas quanto à abordagem, fase (em tempo de compilação — TC — e de pós-compilação — PC), arquivo de entrada e arquitetura-alvo. As siglas I, D e U ao lado das CBAs indicam as caches presentes na configuração: cache de instruções, dados e unificada, respectivamente. Já a Tabela 2 classifica estas técnicas quanto aos elementos de programa considerados como candidatos, com relação aos tipos e origens ("Apl." para elementos de aplicação, e "Bib." para de bibliotecas). Para os elementos de código também é reportada a granularidade suportada ("PR" para procedimentos e "BB" para blocos básicos). A granularidade para elementos de biblioteca é sempre a mesma dos elementos de aplicação.

2.3 O ESTADO-DA-ARTE EM ALOCAÇÃO A PARTIR DE ARQUIVOS BINÁRIOS

Esta seção apresenta o estado-da-arte em alocação para SPM, tendo como foco técnicas OVB e NOB que operam a partir de arquivos binários (em tempo de pós-compilação) — o que permite considerar elementos de bibliotecas como candidatos — e que consideram arquiteturas baseadas em cache (*cache-based architectures* — CBAs) como alvo. As demais técnicas não são discutidas aqui em detalhes, embora suas características já tenham sido comentadas na Seção 2.2.

Egger, Lee e Shin (2008) propuseram uma técnica OVB de tempo de pós-compilação que divide o código (sob granularidade de BBs) da aplicação em duas regiões: *cacheable* e *pageable*. Na primeira região é colocado o código a residir no espaço de endereçamento da MP, acessado por meio de uma memória cache. Na segunda região, a técnica coloca páginas de código que serão copiadas para a SPM sob demanda, em tempo de execução. Para tanto, faz-se necessário hardware extra: um gerenciador de SPM e uma MMU. Acessos à região *cacheable* acontecem naturalmente: se o acesso a um endereço na cache resulta em falta, o bloco equivalente é buscado da MP. Já acessos à região *pageable* devem acontecer sempre por meio da SPM e são orientados por páginas de código. Uma página de SPM possui tamanho fixo, determinado pelo tamanho da página da MMU. Se a página contendo o endereço requisitado não se encontra ali, a MMU sinaliza uma exceção de falta de página. O gerenciador captura a exceção, copia a página requisitada para a SPM e a execução recomeça no endereço em que havia sido interrompida, desta vez acessado na SPM.

Tabela 1: Técnicas de alocação em SPM quanto à abordagem, fase, arquivo de entrada e arquitetura-alvo

Técnica	**Abordagem**	**Fase**	**Arquivo de entrada**	**Arquitetura de memória alvo**
Steinke et al. (2002b)	NOB	CT	fonte	UNA
Avissar, Barua e Stewart (2002)	NOB	CT	fonte	UNA
Angiolini et al. (2004)	NOB	PC	executável	CBA (I+D ou U)[a]
Mendonça (2009, 2010)	NOB	PC	obj. reloc.	CBA (I+D)
Este trabalho	NOB	PC	obj. reloc.	CBA (I+D)
Kandemir et al. (2001)	OVB	CT	fonte	CBA (D)
Steinke et al. (2002a)	OVB	CT	fonte	UNA
Verma, Wehmeyer e Marwedel (2004b)	OVB	CT	fonte	UNA
Udayakumaran, Dominguez e Barua (2006)	OVB	CT	fonte	UNA
Chen et al. (2006)	OVB	CT	fonte	UNA
Janapsatya, Parameswaran e Ignjatovic (2004)	OVB	PC	executável	UNA
Janapsatya, Ignjatović e Parameswaran (2006b)	OVB	PC	executável	UNA[b]
Cho et al. (2007)	OVB	PC	objeto	CBA (I+D)[c]
Egger, Lee e Shin (2008)	OVB	PC	executável	CBA (I+D)[d]
Deng et al. (2009)	OVB	PC	executável	CBA (I+D)
Egger et al. (2010)	OVB	PC	objeto	UNA e CBA (I)

[a] A cache é opcional, embora a arquitetura assuma apenas cache de instruções
[b] Apesar de possuir uma cache de dados, para fins de comparação deve ser considerada UNA, pois a técnica não otimiza dados
[c] Memória particionada horizontalmente: uma cache e uma SPM na partição de dados (foco do trabalho), e outra na de instruções
[d] Mesmo caso de *c*, porém com foco na partição de instruções

Tabela 2: Técnicas de alocação em SPM quanto aos elementos de programa considerados

Técnica	Dados								Código	
	Global escalar		Global não-escalar		Pilha		*Heap*			
	Apl.	Bib.	Apl.	Bib.	Apl.	Bib.	Apl.	Bib.	Apl.	Bib.
Steinke et al. (2002b)	✓	-	✓	-	-	-	-	-	PR e BB	-
Avissar, Barua e Stewart (2002)	✓	-	✓	-	✓	-	-	-	-	-
Angiolini et al. (2004)	-	-	-	-	-	-	-	-	BB (e menor)	✓
Mendonça (2009, 2010)	✓	✓	✓	✓	-	-	-	-	PR	✓
Este trabalho	✓	✓	✓	✓	-	-	-	-	*ou* PR *ou* BB	✓
Kandemir et al. (2001)	-	-	✓[a]	-	-	-	-	-	-	-
Steinke et al. (2002a)	-	-	-	-	-	-	-	-	PR e BB	-
Verma, Wehmeyer e Marwedel (2004b)	✓	-	✓	-	✓[b]	-	-	-	*Trace*	-
Udayakumaran, Dominguez e Barua (2006)	✓	-	✓	-	✓	-	-	-	PR e BB	-
Chen et al. (2006)	-	-	✓	-	✓	-	✓	-	-	-
Janapsatya, Parameswaran e Ignjatovic (2004)	-	-	-	-	-	-	-	-	BB (e menor)	✓
Janapsatya, Ignjatović e Parameswaran (2006b)	-	-	-	-	-	-	-	-	BB	✓
Cho et al. (2007)	✓	✓	✓	✓	✓	✓	-	-	-	-
Egger, Lee e Shin (2008)	-	-	-	-	-	-	-	-	BB	✓
Deng et al. (2009)	✓	✓	✓	✓	✓	✓	✓	✓	-	-
Egger et al. (2010)	-	-	-	-	-	-	-	-	PR[c]	✓

[a] Limitado para acessos a aplicações baseadas em arranjos, por meio de funções lineares
[b] Apenas dados não-escalares
[c] Realiza *function outlining* em laços muito acessados, transformando-os em procedimentos

O hardware especial garante que não se faça necessário conhecer o tamanho da SPM no momento da geração do binário otimizado pela técnica. No entanto, não é todo sistema embarcado que possui uma MMU e, mais do que isso, uma personalização deve ser feita, introduzindo-se um gerenciador em hardware. Além disso, como a unidade mínima de transferência de código entre MP e SPM pela MMU é de uma página, é necessário agrupar mais do que um BB sob uma mesma página, para evitar fragmentação e o desperdício de espaço na SPM, o que gera um novo problema a ser solucionado pelo método: a divisão do código na região *pageable* na menor quantidade de páginas possível e com as páginas contendo código com boa localidade espacial, para diminuir o número de transferências. Isso claramente afeta a localidade espacial do código. Além disso, como a técnica somente agrupa sob uma mesma página BBs oriundos de um mesmo procedimento, ocorre desperdício de espaço.

Os autores reportam um ganho médio de 14% em redução de consumo de energia e 16% de ganho em desempenho quando aplicam a técnica em uma CBA, tendo como referência uma FCA.

Algumas limitações da técnica de Egger, Lee e Shin (2008) foram sobrepujadas em **Egger et al. (2010)**, que desta vez trabalha com granularidade de procedimentos. O requisito de hardware extra (MMU e gerenciador de SPM) foi substituído por um gerenciador implementado via software. Informações sobre as páginas de código são armazenadas em estruturas de dados em formato de tabela: uma delas contém o endereço de cada procedimento residente na região *pageable* e o número de páginas que ocupa; a outra é similar a uma tabela de desvios (*jump table*). A cada procedimento corresponde uma entrada nessa tabela. Se o procedimento já está carregado na SPM, sua entrada contém uma instrução de desvio incondicional para lá. Se não está, contém uma instrução de interrupção de software que quando executada será capturada pelo gerenciador que, por sua vez, iniciará a transferência do procedimento da MP para a SPM, atualizará a entrada na tabela e desviará para a SPM.

A tabela de desvios elimina a necessidade de hardware extra, porém, limita o tipo dos elementos considerados para apenas código. Também decorre dela um certo *overhead* para lidar com chamadas de procedimentos via ponteiros, porque deve-se primeiro realizar uma busca binária sobre a tabela com os endereços de procedimentos paginados a fim de descobrir em qual região de memória o procedimento se encontra, para só então tratá-lo de acordo.

A mudança de granularidade para procedimentos também agravou

o problema de fragmentação e desperdício de espaço na SPM, porque a técnica não realiza agrupamento de procedimentos sobre uma mesma página. Para amenizar este problema, a técnica realiza a extração de procedimentos a partir de laços (denominado *function abstracting* ou *function outlining*). Além disso, a técnica apresenta resultados sub-ótimos porque assume conservadoramente que os procedimentos paginados sempre necessitam ser alocados na SPM, enquanto que podem ter sido carregados previamente.

Um grande diferencial desta técnica com relação às demais é que, pela primeira vez, tem-se uma técnica híbrida, simultaneamente OVB e NOB. O código é dividido em uma nova região (*pinned*), que contém código alocado sob NOB, carregado para a SPM quando a aplicação é carregada e permanecendo lá até o final da execução. Elementos são alocados desta forma quando existe uma certa vantagem em executá-los da SPM mas o *overhead* que advém da constante reposição dos elementos para a SPM é considerável, evitando estas transferências extras.

Os autores validaram a técnica sobre dois sistemas embarcados. Para o primeiro, simulado, obtiveram uma economia média de 19% em redução de energia e 12% em desempenho para uma UNA (comparado a uma FCA). Para o segundo, hardware real, economia média de 24% em desempenho e redução de energia para uma CBA, comparada a uma FCA.

Outra técnica OVB que inclui elementos de código oriundos de bibliotecas no espaço de otimização é a de **Janapsatya, Parameswaran e Ignjatovic (2004)**, de tempo de pós-compilação e que trabalha com granularidade de blocos lógicos.

Foi a primeira técnica a propor modificação de hardware de modo a tornar a cópia de elementos para SPM em tempo de execução mais eficiente. Uma instrução especialmente criada permite ativar o controlador da SPM e copiar um elemento inteiro de uma só vez, ao contrário de outras técnicas (e.g. Steinke et al. (2002a) necessita de duas instruções para a cópia de uma única palavra). Além disso, pela primeira vez o problema de particionamento de código entre SPM e MP foi modelado como um grafo. Por meio de análise de laço no grafo, os melhores blocos para SPM são selecionados e os pontos de inserção da instrução especial identificados. No entanto, seu algoritmo é ineficiente, pois usa uma heurística global que não trata de maneira adequada laços com BBs que estão muito distantes.

Redução de consumo de energia de 51% e melhoria de desempenho de 53% são relatados, em média, tendo como alvo uma UNA e como

referência uma FCA.

Janapsatya, Ignjatović e Parameswaran (2006b) relaxaram a necessidade de hardware especial de sua técnica anterior, de modo que a cópia de elementos poderia ser feita via software — muito embora continuaram a utilizar a cópia via hardware por motivo de eficiência. Eles também propuseram uma nova métrica, visando uma seleção mais eficiente (mapeamento) dos BBs para SPM. Esta métrica, denominada concomitância, mede a correlação temporal entre dois BBs. No entanto, esta técnica não produz um arquivo binário executável, que pode ser validado em um simulador. A economia esperada é computada analiticamente e, portanto, sujeita a erros. A redução média de consumo de energia relatada é de 41% para uma arquitetura-alvo que deve ser vista como uma UNA — pois muito embora possua cache de dados, a técnica aloca somente elementos de código — quando comparada a uma FCA.

A técnica proposta por **Mendonça (2009, 2010)** é uma técnica de tempo de pós-compilação que trabalha com arquivos-objeto (binários) relocáveis. A técnica modifica estes arquivos, realocando os elementos de código (sob granularidade de procedimentos) e dados selecionados para uma seção própria (`.spm`).

O *patching* desta técnica, particularmente de dados, é muito eficiente graças à presença da tabela de relocações nos arquivos-objeto. Nestes, o alvo de instruções de desvio, de chamada de procedimentos e de acesso a dados não é um endereço absoluto — isso será feito pelo linkeditor, no momento da geração do binário executável —, mas simbólico, marcado por uma relocação. As relocações são compostas pelo endereço da instrução a ser ajustada, pela ação de relocação (que dita o modo pelo qual o símbolo é convertido para codificação binária) e pela dependência simbólica (o nome do procedimento ou dado), e são armazenadas em tabela própria. O *patching* consiste unicamente em ajustar as entradas desta tabela que dizem respeito aos elementos mapeados para SPM, modificando a dependência simbólica para que aponte para o elemento em SPM.

Esta técnica, estendida pelo presente trabalho, ainda que puramente NOB, abrange um espaço de otimização maior do que as descritas anteriormente, quanto ao tipo e origem dos elementos. Embora todas as técnicas discutidas na presente seção sejam capazes de alocar elementos de biblioteca, aquelas consideram somente elementos de código, ao passo que esta considera também dados globais escalares e não-escalares, que não podem ser desprezados (MENDONÇA, 2010). De fato, trata-se da técnica NOB com maior abrangência de elementos até o presente momento. Além disso, é a única, dentre NOBs e OVBs, a alocar dados

de bibliotecas sem recursos extras de hardware. Vale lembrar que as técnicas OVB de Cho et al. (2007) e de Deng et al. (2009) conseguem isto utilizando hardware extra, enquanto que, dentre as NOBs, nenhuma possui esta aptidão.

Todas estas características de Mendonça (2009, 2010) continuam valendo para a extensão proposta neste trabalho. Além disso, a adição de suporte à granularidade de blocos básicos (detalhado no Capítulo 4) permite buscar maiores economias de energia em sistemas com SPM pequena, quando as aplicações exibem alta localidade.

2.4 CONSIDERAÇÕES SOBRE AS ABORDAGENS DE ALOCAÇÃO EM SPM

Muitas aplicações foram relatadas para as quais a maioria dos acessos à memória se concentram em um espaço de endereçamento suficientemente pequeno para caber na SPM (MENICHELLI; OLIVIERI, 2009), tornando a complexidade extra da alocação OVB desnecessária. A alocação NOB é especialmente adequada para tais aplicações com *hot spots*. Contudo, a abordagem NOB é sub-explorada quando aplicada em código-fonte: técnicas de tempo de compilação (AVISSAR; BARUA; STEWART, 2002) (STEINKE et al., 2002b) são incapazes de mapear elementos de bibliotecas ou de software protegido de IPs de terceiros para a SPM.

A alocação para SPM em tempo de compilação funciona mais naturalmente quando utilizada em conjunto com a abordagem OVB (KANDEMIR et al., 2001) (STEINKE et al., 2002a) (VERMA; WEHMEYER; MARWEDEL, 2004b) (UDAYAKUMARAN; DOMINGUEZ; BARUA, 2006). Quando múltiplos *hot spots* não cabem ao mesmo tempo na SPM, uma abordagem OVB de tempo de compilação conduzirá a uma maior economia por meio da exploração dinâmica de propriedades de programa observáveis no nível de código-fonte (as quais foram capturadas estaticamente em tempo de compilação). A potencial economia extra em se alocar elementos encapsulados em bibliotecas, que não seriam de qualquer forma exploráveis no nível de código-fonte, acaba sendo encoberta pela economia extra proveniente da alocação OVB.

Não obstante sua reconhecida superioridade quando aplicadas em tempo de compilação, as técnicas OVB não se mostram tão adequadas para manusear binários, salvo se hardware dedicado é utilizado para suportar o gerenciamento dinâmico da SPM (CHO et al., 2007) (DENG

et al., 2009) (EGGER; LEE; SHIN, 2006) (EGGER; LEE; SHIN, 2008) ou se a alocação em SPM é limitada a código apenas (EGGER et al., 2010) (JANAPSATYA; PARAMESWARAN; IGNJATOVIC, 2004).

Por consequência, para viabilizar sua maior aplicação e uma política de alocação mais inclusiva, um método consciente de bibliotecas deve preferencialmente adotar uma abordagem NOB, que não depende de hardware dedicado, desde que não negligencie a alocação de dados na SPM.

Para UNAs, a abordagem OVB leva a uma redução significativa do total de energia quando aplicada em tempo de compilação: em média, 34%, em Verma, Wehmeyer e Marwedel (2004b), e 31%, em Udayakumaran, Dominguez e Barua (2006), quando comparada com abordagens NOB; 30%, em Steinke et al. (2002a), quando comparada com FCAs. Esta redução é menor quando trabalha no nível de arquivos binários (em média, 21% em Egger et al. (2010) quando comparado com FCAs), visto que arquivos-objeto limitam a manipulação das propriedades de programa necessárias para o gerenciamento da sobreposição de elementos. Além disso, tais alocadores de SPM que não consideram a memória cache superestimam o lucro, o que é agravado em sistemas onde a cache é reconfigurável ou pré-ajustada para uma determinada aplicação (VIANA et al., 2006) (ZHANG; VAHID, 2003). Por exemplo, Zhang e Vahid (2003) reportam ganhos de 40% por meio do simples ajuste de parâmetros da cache para a aplicação (sem uma SPM). Por este motivo, em sistemas com caches pré-otimizadas, deve-se esperar ganhos muito menores que aqueles reportados em Steinke et al. (2002a), Verma, Wehmeyer e Marwedel (2004b) e Udayakumaran, Dominguez e Barua (2006).

Efetivamente, quando métodos OVB têm como alvo arquiteturas com caches, a economia de energia total reportada é marginal (em média, 8% para dados em Cho et al. (2007) e 14% e 24% para código em Egger, Lee e Shin (2008) e Egger et al. (2010), respectivamente, quando comparado com FCAs). Isto parece ser uma evidência que, na presença de caches, o impacto de métodos OVB é realmente menor do que o reportado para UNAs, uma vez que a própria cache atua como alocador dinâmico, evitando muitos acessos para a memória externa. Por esta razão, quando voltados para CBAs e lidando com binários, os métodos NOB podem ser quase tão eficazes quanto os métodos OVB. De fato, quando compara diretamente um método NOB com um método OVB sob exatamente o mesmo ambiente experimental, Egger et al. (2010) reporta essencialmente os mesmos ganhos para mais de metade dos programas avaliados.

Infelizmente, até onde se tem notícia, todos os trabalhos em alocação de SPM negligenciam o pré-ajuste de cache: eles reportam resultados normalizados para uma cache de referência cujo tamanho ou é fixo para todos os programas do *benchmark* ou é variável, porém definido por alguma outra métrica qualquer. Embora tal configuração experimental com um tamanho fixo de cache emule um caso prático (a otimização de código para memórias embarcadas preconcebidas), isto não provê uma base para o dimensionamento de memórias customizadas ou configuráveis (TALLA; GOLSTON, 2007) (MALIK; MOYER; CERMAK, 2000).

Como o objetivo deste trabalho não é propor uma nova técnica mas reavaliar quantitativamente a bem-conhecida abordagem NOB utilizando a perspectiva da pesquisa recente no ajuste-fino das caches (*cache tuning)* (ZHANG; VAHID, 2003) (VIANA et al., 2008), serão emprestadas noções de Angiolini et al. (2004), Mendonça (2010) e Volpato et al. (2010) para a formulação do problema-alvo, descrita no Capítulo 3.

3 O PROBLEMA-ALVO

Considere-se uma arquitetura baseada em cache (*cache-based architecture* — CBA) contendo uma memória principal, uma cache de instruções, uma cache de dados, e uma SPM.

A partir deste ponto, usaremos M para denominar um componente genérico de memória, o qual pode ser a memória principal (MP), uma memória cache de instruções ou de dados (I-cache e D-cache, respectivamente) ou uma SPM.

A **energia** consumida para acessar uma posição de M, E_M, é definida como a energia média gasta na leitura ou na escrita daquela posição.

A **latência** de uma memória, λ_M, é definida como o tempo gasto para acessar uma de suas posições, expressado em ciclos de relógio.

A **capacidade** de uma memória, C_M, é seu tamanho, especificado em bytes.

Definição 3.1. *Trace* de memória. Um *trace* T é uma tupla (α_1, α_2, ..., α_i, ... α_n) que representa uma sequência de endereços sucessivos, acessados no subsistema de memória (UHLIG; MUDGE, 1997), onde α_i denota o i-ésimo endereço.

Definição 3.2. Elemento candidato à alocação. Um elemento de programa candidato, denominado por D_i, é um contêiner de dados (uma constante, uma variável, uma estrutura de dados, etc.) ou um contêiner de instruções (um procedimento, um bloco básico, uma única instrução, etc.).

Dado o tamanho σ_i de um elemento D_i, sua *faixa de posições* varia entre um endereço-limite inferior α_i até um endereço-limite superior $\alpha_i + \sigma_i - 1$.

Definição 3.3. Número de acessos de um elemento candidato. Dado um *trace* T e um elemento candidato D_i, o número de acessos à memória em T que recaem na faixa de posições de D_i é:

$$a_i = \sum_{\forall \alpha | \alpha \in T} \delta_\alpha, \text{ onde: } \delta_\alpha = \begin{cases} 1 & \text{se } \alpha_i \leq \alpha < \alpha_i + \sigma_i \\ 0 & \text{caso contrário} \end{cases}$$

Definição 3.4. Taxa de faltas de um elemento candidato. Dado um trace T e um elemento candidato D_i, sua taxa de faltas é o número de vezes em que ele não é encontrado na cache para todas as referências

a ele em T, i.e.:

$$m_i = \frac{\sum\limits_{\forall \alpha | \alpha \in T} \phi_\alpha}{a_i}, \text{ onde: } \phi_\alpha = \begin{cases} 1 & \text{se } D_i \text{ não se encontra na cache} \\ 0 & \text{caso contrário} \end{cases}$$

Definição 3.5. Energia consumida em um acesso a um elemento candidato. A energia consumida por um acesso a um candidato D_i no espaço de endereçamento da MP é:

$$E_i = E_{cache} + m_i \times (E_{MP} + E_{cache}),$$

onde $E_{cache} = E_{I\text{-}cache}$ quando D_i é um contêiner de código e $E_{cache} = E_{D\text{-}cache}$, caso contrário.

Definição 3.6. Lucro (*profit*) de energia de um candidato. O lucro de energia de um elemento candidato D_i quando mapeado para a SPM é calculado por:

$$p_i = a_i \times (E_i - E_{SPM}) - \varepsilon_i,$$

onde ε_i é o *overhead* de energia associado com as instruções extras que desviam incondicionalmente o fluxo de execução da MP para a SPM e, depois, de volta para a primeira, para que o conteúdo do elemento seja acessado no espaço de endereçamento da SPM.

Quando D_i é um procedimento ou um dado, uma vez que nenhuma instrução extra se faz necessária (ANGIOLINI et al., 2004; MENDONÇA, 2010), tem-se $\varepsilon_i = 0$. Quando D_i é um bloco básico, decorre $\varepsilon_i > 0$, a ser detalhado na Seção 4.4.1.

Definição 3.7. Espaço necessário para um elemento candidato. O espaço w_i, em bytes, necessário para alocar um elemento D_i em SPM é

$$w_i = \sigma_i + \sigma_{extra},$$

onde σ_{extra} é o tamanho total das instruções extras necessárias para direcionar o fluxo de controle para aquele elemento quando D_i é um bloco básico alocado em SPM. O cálculo de σ_{extra} será detalhado na Seção 4.4.2. Quando D_i é um procedimento ou um dado, tem-se $w_i = \sigma_i$, pois não se fazem necessárias instruções extras (ANGIOLINI et al., 2004; MENDONÇA, 2010).

Definição 3.8. Caracterização de espaço dos elementos candidatos. Sejam $D_1, ..., D_i, ..., D_n$ os elementos candidatos acessados por

um código embarcado. Sua caracterização de espaço é dada por uma matriz-linha $W = [w_1, ..., w_i, ..., w_n]$, onde w_i significa o espaço necessário para alocar o elemento D_i em SPM, expresso em bytes.

Definição 3.9. Caracterização de lucro dos elementos candidatos. Sejam $D_1, ..., D_i, ..., D_n$ os elementos candidatos acessados por um código embarcado. Sua caracterização de lucro é denotada por uma matriz-linha $P = [p_1, ..., p_i, ..., p_n]$, onde p_i é o lucro pela alocação do elemento D_i em SPM.

Definição 3.10. Mapeamento de elementos para a SPM. Um mapeamento para a uma SPM é representado por uma matriz-coluna $X = [x_1, ..., x_i, ..., x_n]^{-1}$, onde $x_i = 1$ significa que um elemento D_i é mapeado para o espaço de endereçamento da SPM e $x_i = 0$ significa que D_i é mapeado para o espaço de endereçamento (disjunto) da MP.

O problema geral de otimização consiste em determinar quais elementos de programa serão mapeados para a SPM, de modo a minimizar uma função custo, ou equivalentemente, maximizar uma função lucro. O objetivo pode ser minimizar o tempo de acesso total à memória ou o consumo de energia. Essa possibilidade levará a diferentes instâncias do problema de otimização.

Como o número de acessos aos elementos depende da computação dos algoritmos utilizados no software embarcado, a otimização será feita capturando um padrão típico de acessos, caracterizado por um *trace* de memória T. Além disso, os elementos de programa candidatos devem ser caracterizados em termos de seus tamanhos.

O problema-alvo pode ser reformulado como um problema clássico de otimização combinatória chamado Problema Binário da Mochila (*0–1 Knapsack Problem*) (KARP, 1972), conforme formalizado abaixo.

Problema-alvo de Otimização: Dados um conjunto de elementos candidatos D_i, caracterizados por W e P, e um padrão de acesso capturado por um trace T, encontre a alocação X que maximize a função lucro $p(X) = P \times X$ tal que $W \times X \leq C_{SPM}$

A solução do Problema-alvo de Otimização por meio de técnicas exatas pode resultar em tempos de execução proibitivos para casos de uso reais, por tratar-se de um problema NP-completo. Embora muitos métodos utilizem Programação Linear Inteira (ILP), o problema também pode ser resolvido idealmente por meio de Programação Dinâmica, em tempo pseudo-polinomial (PAPADIMITRIOU, 1981).

O Problema-alvo é solucionado utilizando o algoritmo `MINKNAP`

(PISINGER, 1997), que garante solução ótima e eficiente para o Problema Binário da Mochila, da mesma forma que a técnica de Mendonça (2009, 2010), que foi estendida por este trabalho.

4 EXTENSÃO DE UMA TÉCNICA NOB PARA INCLUSÃO DE BLOCOS BÁSICOS NO ESPAÇO DE OTIMIZAÇÃO

A técnica proposta por Mendonça (2009, 2010) considera como candidatos elementos de código e dados globais estáticos (escalares e não-escalares), particularmente aqueles originados de bibliotecas. Para tornar isto possível, manipula arquivos-objeto relocáveis em tempo de pós-compilação. Entretanto, a técnica de Mendonça somente realiza *alocação de procedimentos* (*procedure allocation* — PRA), por utilizar a granularidade de procedimentos para definir as fronteiras entre os elementos de código.

O ajuste-fino das caches — fundamento da reavaliação das técnicas NOBs proposta por este trabalho —, quando precede a alocação em SPM, permite uma análise mais realista da economia obtida, impedindo que fatores secundários influenciem nesta economia. Aproveitando desta facilidade, optou-se por avaliar também o impacto da variação da granularidade dos elementos de código na abordagem NOB.

Para permitir a exploração da granularidade de elementos de código, foi necessário realizar uma extensão na técnica de Mendonça (2009, 2010), de modo a habilitar a *alocação de blocos básicos* (*basic-block allocation* — BBA), como alternativa à PRA.

A extensão implementada para que o código possa ser dividido na granularidade de blocos básicos (BBs) contemplou os seguintes aspectos:

- identificação das instruções de desvio condicional no código binário, para a demarcação dos blocos;
- identificação do caso de cada bloco básico. O caso é determinado verificando algumas características: se possui ou não instrução de desvio; caso sim, se o desvio é condicional ou incondicional; e se o bloco constitui um laço;
- adaptação do *profiler* para que este compute o número de acessos, taxa de faltas e o custo energético para o acesso aos candidatos BBs;
- adaptação da caracterização de espaço e de lucro dos candidatos para levar em conta os candidatos BBs;
- modificação do mecanismo de *patching* para que os BBs mapeados em SPM sejam alocados nesta quando do início da execução do programa.

É importante esclarecer que na versão estendida da técnica, apenas uma granularidade, BBs ou procedimentos, é considerada por vez como fronteira para os elementos de código.

4.1 FLUXO DE TRABALHO

Dados uma aplicação, um nodo tecnológico e um subsistema de memória pré-especificado, o objetivo último da extensão realizada permanece o mesmo da técnica original: produzir um arquivo executável relocado, de maneira que o melhor conjunto de elementos de programa seja mapeado para o espaço de endereçamento da SPM. Alguns passos são necessários, tanto para habilitar este mapeamento como para refleti-lo no código executável consciente de SPM. A Figura 7 mostra o fluxo de trabalho da técnica estendida. As etapas deste fluxo são detalhadas nas seções subsequentes. O processo é uma especialização da técnica ilustrativa explicada na Seção 2.1.

4.2 CARACTERIZAÇÃO DOS ELEMENTOS

O ponto de entrada do fluxo de trabalho é um conjunto de arquivos-objeto relocáveis, os quais podem compreender bibliotecas estáticas. Neste passo, algumas poucas propriedades de programa são extraídas destes arquivos para permitir a otimização pretendida, a saber, o conjunto de elementos de programa candidatos (D_i) e seus tamanhos (σ_i). A tarefa mais elaborada desse passo consiste na identificação dos elementos de programa, como descrito a seguir.

A técnica estendida permite manipular dados estáticos (escalares e vetores) e código (ou procedimentos ou BBs). A identificação de elementos de dados estáticos e procedimentos é direta. Entretanto, quando BBs são escolhidos como elementos de código, quatro diferentes casos podem ocorrer, como retratado pelas Figuras 8(a), 8(b), 8(c) e 8(d). O lado esquerdo de cada figura (à esquerda da seta) mostra a fronteira de um BB (em borda destacada) e as instruções que o sucedem no espaço de endereçamento da memória principal (MP) antes do mapeamento. O lado direito de cada figura (à direita da seta) ilustra os espaços de endereçamento disjuntos, após o mapeamento daquele BB para SPM. As instruções de desvio condicional e incondicional estão negritadas.

O lado esquerdo das Figuras 8(a), 8(b), 8(c) e 8(d) ilustram as características que permitem distinguir os BBs segundo quatro casos:

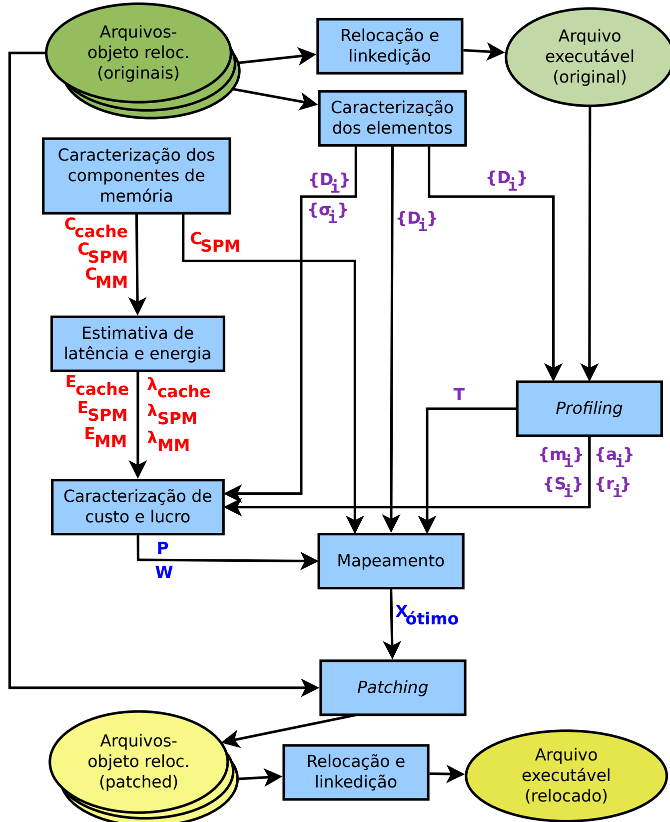
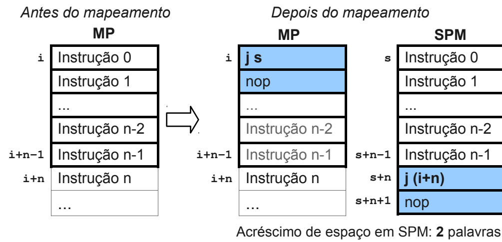


Figura 7: Fluxo de trabalho da técnica estendida de alocação em SPM

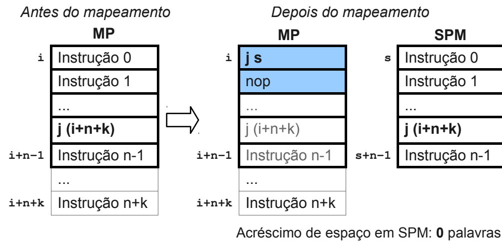
- **Caso I** (Figura 8(a)). O BB não contém nenhuma instrução de desvio. Este caso ocorre geralmente com código que inicializa variáveis ou armazena/recupera contextos.

- **Caso II** (Figura 8(b)). O BB termina com um desvio incondicional. Isto é tipicamente induzido por estruturas de seleção de múltipla escolha (`case/switch`) e estruturas de seleção `se-senão` (`if-else`) com múltiplas condições.

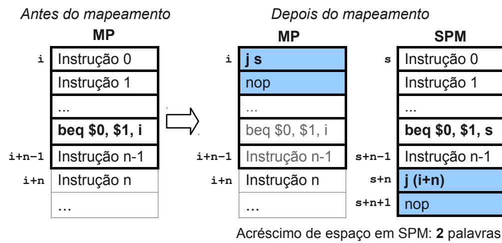
- **Caso III** (Figura 8(c)). O BB termina termina com um desvio condicional cujo alvo é o próprio BB, i.e. ele forma uma estrutura de laço. Isto é frequentemente induzido por estruturas de repetição pós-testada (`do-while`) sem estruturas condicionais aninhadas.



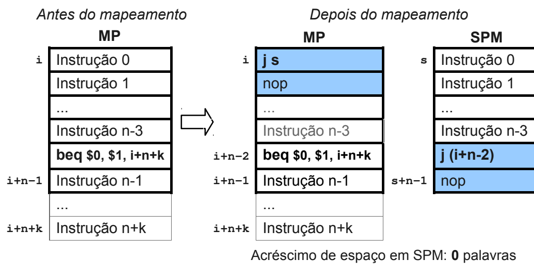
(a) Caso I



(b) Caso II



(c) Caso III



(d) Caso IV

Figura 8: Casos que podem ocorrer no mapeamento de blocos básicos para SPM

- **Caso IV** (Figura 8(d)). O BB termina com um desvio condicional cujo alvo é outro BB. Isso ocorre comumente quando os laços se estendem por mais de um BB (por exemplo, em estruturas de repetição `para` com estruturas condicionais `se-senão` aninhadas).

Como laços tendem a ser executados diversas vezes, os BBs que recaem nos Casos III e IV aparentam ser candidatos mais promissores para mapeamento em SPM quando comparados àqueles que recaem nos Casos I e II. Evidentemente, isso dependerá da taxa de invocação.

Quando um BB é mapeado para SPM, seu código é replicado no espaço de endereçamento da SPM. Contudo, somente a réplica de seu conteúdo não é suficiente para que o BB possa ser acessado a partir da SPM. Ajustes fazem-se necessários em suas posições no espaço de endereçamento da MP e, eventualmente, no espaço da SPM. As posições que necessitam de ajuste estão hachurados, à direita da seta nas Figuras 8(a), 8(b), 8(c) e 8(d). Estes ajustes estão explicados em maiores detalhes a seguir.

No espaço de endereçamento da MP, o ponto de entrada de um BB (i.e. sua primeira instrução) é substituído por um desvio incondicional (`jump`) para a posição (digamos, `s`) do código do BB no espaço de endereçamento da SPM. É importante perceber que, como muitas arquiteturas possuem desvios adiados (*delayed branches*), os *delay slots* devem ser considerados sempre que um desvio incondicional é adicionado. Por simplicidade, *delay slots* são preenchidos com instruções de não-operação (`nop`), embora pudessem ser preenchidos com instruções úteis (e.g. instruções de um BB precedente).

No espaço de endereçamento da SPM, embora as mudanças necessárias dependam do caso do BB, estas consistem basicamente em possuir um desvio incondicional que salta de volta para a posição adequada no espaço de endereçamento da MP. Estes ajustes tem os seguintes impactos:

- **Caso II:** não necessita de mudanças pois o código original já termina com um desvio incondicional (e seus alvos não necessitam de ajuste nos deslocamentos porque são codificados como endereços absolutos).

- **Casos I e III:** requerem a inserção de desvio incondicional (*jump*) no final do BB, o que aumenta levemente seu tamanho e, por conseguinte, o espaço necessário.

- **Caso IV:** o desvio condicional final é transformado em um desvio incondicional cujo alvo é a posição daquele desvio no espaço de

endereçamento da memória principal.

A solução adotada no Caso IV tem como intuito reduzir o *overhead* de espaço na SPM, permitindo assim que mais elementos sejam mapeados, embora ocasione aumento na energia gasta no espaço de endereçamento da MP pelo acesso à instrução de desvio condicional. Se decidíssemos pela execução do desvio condicional no espaço de endereçamento da SPM, seria necessário não apenas uma mudança no deslocamento do desvio, como também a adição de dois desvios incondicionais (um para o alvo do desvio condicional e outro para a primeira instrução após a saída do BB).

4.3 *PROFILING* DO PROGRAMA

Submetem-se os arquivos-objeto relocáveis ao ligador para realizar a relocação e edição de referências, do que resulta um arquivo executável utilizado para fins de *profiling*. Por meio da execução deste executável, determina-se o *trace* T.

Dado o *trace* T, para cada elemento de programa candidato D_i, o *profiler* calcula seu número de acessos a_i e sua taxa de faltas m_i, conforme as Definições 3.3 e 3.4, respectivamente.

4.4 CARACTERIZAÇÃO DE LUCRO E ESPAÇO

Para um dado nodo tecnológico, a caracterização de lucro requer a estimativa de energia consumida (ou do atraso) por acesso para cada componente do subsistema de memória. Além disso, já que a energia média depende dos padrões de acesso, a caracterização do lucro igualmente leva em conta a configuração do subsistema de memória, tal como as capacidades dos componentes (C_{MP}, C_{cache}, C_{SPM}) e outros parâmetros de configuração (número de portas, associatividade da cache, tamanho de bloco, etc.).

Dada a energia média e a latência média de cada componente (E_{MP}, E_{cache}, E_{SPM}, λ_{MP}, λ_{cache}, λ_{SPM}), a caracterização de lucro e a de espaço consistem em encontrar um lucro (p_i) e um espaço (w_i) para cada elemento candidato (D_i) mapeado para SPM, conforme as Definições 3.9 e 3.8.

Seja τ uma função que mapeia cada elemento de programa D_i para um tipo de elemento, tal que:

$$\tau : D \rightarrow \{BB, proc, data\}$$

onde *BB*, *proc* e *data* significam um bloco básico, um procedimento ou um dado estático, respectivamente.

4.4.1 Lucro de energia de um bloco básico

Da Definição 3.6, tem-se que o lucro de energia obtido em mapear um elemento para a SPM é dado por $p_i = a_i \times (E_i - E_{SPM}) - \varepsilon_i$.

Para facilitar a visualização do quanto cada espaço de endereçamento (MP e SPM) contribui para o *overhead* ε_i, este pode ser subdividido nas seguintes componentes:

$$\varepsilon_i = \varepsilon_i^{MP} + \varepsilon_i^{SPM},$$

onde ε_i^{MP} e ε_i^{SPM} representam os *overheads* de energia nos espaços de endereçamento da MP e da SPM, respectivamente.

Seja s o número de *delay slots* na arquitetura alvo e r_i a taxa de invocação do BB D_i (i.e. o número de vezes que seu ponto de entrada é acessado). Quando $\tau(D_i) = BB$ sob o Caso III (laço), sua taxa de invocação pode ser escrita como $r_i = N_i + S_i$, onde N_i denota o número de invocações devido às iterações do laço e S_i representa o número de vezes que o BB é alcançado partindo de um outro BB, digamos D_k, tal que $k \neq i$.

Considerando os casos de BBs apresentados na Seção 4.2, os *overheads* podem ser escritos como:

$$\varepsilon_i^{MP} = \begin{cases} (1+s) * r_i * E_i & \text{se } \tau(D_i) = BB \text{ sob os Casos I e II} \\ (1+s) * S_i * E_i & \text{se } \tau(D_i) = BB \text{ sob o Caso III} \\ 2 * (1+s) * r_i * E_i & \text{se } \tau(D_i) = BB \text{ sob o Caso IV} \end{cases} \tag{4.1}$$

$$\varepsilon_i^{SPM} = \begin{cases} (1+s) * r_i * E_{SPM} & \text{se } \tau(D_i) = BB \text{ sob os Casos I e III} \\ 0 & \text{caso contrário} \end{cases} \tag{4.2}$$

Quando $\tau(D_i) = BB$, seu cálculo leva em consideração a energia gasta acessando as instruções de desvio incondicional (e seus respectivos *delay slots*) adicionadas: um desvio da MP para a SPM e outro no caminho oposto.

Quando $\tau(D_i) = \{proc, data\}$, o *overhead* é sempre nulo.

4.4.2 Espaço necessário para alocar um bloco básico

Da Definição 3.7, temos que o espaço ocupado em SPM quando da alocação de D_i é: $w_i = \sigma_i + \sigma_{extra}$.

Da mesma forma que para o lucro, pode-se compreender σ_{extra} como um *overhead* de espaço, em bytes, devido às instruções extras que direcionam o fluxo de controle do espaço de endereçamento da SPM para o da MP. Para o direcionamento do espaço da MP para o da SPM não se tem acréscimo de instruções, pois as próprias posições do BB na MP são utilizadas.

Para um candidato D_i, o espaço extra necessário para sua alocação em SPM é dado por:

$$\sigma_{extra} = \begin{cases} \sigma_{jmp} + s \times \sigma_{nop} & \text{se } \tau(D_i) = BB \text{ sob os Casos I e III} \\ 0 & \text{caso contrário} \end{cases} \quad (4.3)$$

onde σ_{jmp} e σ_{nop} são, respectivamente, os tamanhos de uma instrução de desvio incondicional (*jump*) e de uma instrução de não-operação (*nop*), expressos em bytes.

Comparando esta equação com a Figura 8, fica claro que apenas dois casos de BBs necessitam de espaço extra na SPM. Para os outros dois casos, bem como para procedimentos e dados, nenhuma instrução adicional é necessária.

4.5 MAPEAMENTO EM SPM

Este passo consiste em resolver o Problema de Otimização Alvo, para o qual existem muitos algoritmos propostos na literatura. Neste trabalho, adota-se um eficiente algoritmo de programação dinâmica conhecido como MINKNAP (PISINGER, 1997). Como resultado, um mapeamento ótimo de elementos para SPM ($X_{ótimo}$) é encontrado.

A etapa de mapeamento procede da mesma forma que na técnica de Mendonça (2009, 2010).

4.6 *PATCHING* DE BINÁRIOS

Dado o mapeamento ótimo, os arquivos relocáveis originais são alterados (*patching*) de modo a refletir o mapeamento consciente de SPM.

O esforço para realizar o *patching* aumenta quando elementos de código são divididos de procedimentos em BBs. No entanto, o *patching* em arquivos-objeto relocáveis tende a ser mais eficiente do que em arquivos executáveis, devido às facilidades providas pelas relocações (MENDONÇA, 2010).

O patching de um procedimento consiste basicamente em mover seu código para uma seção de SPM, que posteriormente será realocado para o espaço de endereçamento da SPM, em tempo de ligação, e em editar a tabela de símbolos de cada arquivo-objeto.

Por outro lado, o *patching* de um BB não é simples. Embora um BB possua um único ponto de entrada, uma vez que ele poderia ser atingido por BBs distintos e não necessariamente pelo situado em uma faixa de posições vizinhas, necessitaria potencialmente do *patching* de muitos deslocamentos de desvios condicionais no código.

Por esta razão, decidiu-se por copiar o BB para a SPM, mantendo seu código na MP (apenas sobreescrevendo a posição inicial por uma instrução de desvio incondicional e as posições seguintes que correspondem aos *delay slots* por não-operações, conforme ilustrado na Figura 8). Ao fazer isso, evita-se visitar os desvios condicionais que requerem ajuste de deslocamento. Note que o desperdício de espaço de memória total ocupado pelos BBs originais é marginal, pois está atrelado ao tamanho da SPM (tipicamente, alguns KB), que é muito menor do que o da MP (tipicamente, na ordem de MBs).

O *patching* é realizado no nível de arquivos-objeto relocáveis e cada arquivo contendo um elemento de programa mapeado para SPM necessita de ajustes. Cada BB mapeado para SPM tem seu código copiado para uma seção de SPM exclusiva (e.g. `.spm.bb0`) criada no arquivo. Após a cópia, o código do BB é ajustado tanto no espaço de endereçamento da MP como no da SPM, de acordo com os casos ilustrados na Figura 8. Uma vez que o endereço final de cada seção (no arquivo executável) é desconhecido antes do tempo de ligação, o alvo dos desvios incondicionais que redirecionam para e do espaço de endereçamento da SPM não podem ser codificados como endereços absolutos. Ao invés disso, atribui-se uma referência simbólica para a posição desejada. Cada referência simbólica resultante corresponde a uma entrada na *tabela de relocações* do arquivo. Durante relocação e *linkedição*, todas as referências nesta tabela são substituídas por endereços efetivos. Sempre que um BB é copiado, entradas de relocações que originalmente apontavam para ele são redirecionadas para apontarem para sua nova seção de SPM.

O *patching* de elementos de dados e de procedimentos é realizado

conforme descrito por Mendonça (2010).

4.7 GERAÇÃO DE SAÍDA

Depois que todos os arquivos-objeto são submetidos ao *patching*, o ligador (instrumentado com um *script* modificado, ciente de SPM) corrige cada referência simbólica, mescla todas as seções de SPM (`.spm.*`) e as realoca para o espaço de endereçamento de SPM, dando origem ao arquivo executável otimizado.

5 AJUSTE-FINO DE CACHES PARA AVALIAÇÃO DA ALOCAÇÃO EM SPMS

O subsistema de memórias de uma arquitetura somente com caches (FCA) ou de uma arquitetura baseada em cache (CBA) é muito sensível à configuração das caches que o compõem. Tal sensibilidade acarreta grande impacto no total de energia consumida pelo subsistema. Zhang e Vahid (2003) reportam uma economia de 40% pelo simples ajuste de parâmetros em uma cache configurável para a aplicação-alvo. Para certos programas, Zhang, Vahid e Najjar (2005) obtiveram diferenças de energia de quase 60% pela simples variação do tamanho de bloco de uma cache de dados.

Essa variação no consumo de energia pode ser explicada pelo seguinte exemplo. Imagine um subsistema de memória composto por I-cache e MP, o qual está sendo ajustado para uma dada aplicação. Um aumento na capacidade da cache pode, por exemplo, aumentar a energia consumida por acesso, mas ao mesmo tempo, diminuir os acessos à MP, resultando em redução da energia total consumida no subsistema de memória. No entanto, supondo que a taxa de faltas na cache antes do aumento de capacidade já era muito próxima de zero, i.e. o número de acessos à MP já era pequeno, é de se esperar que este ajuste acabe aumentando o total de energia consumida no subsistema de memória.

Quando a alocação em SPM despreza o impacto da(s) cache(s) no subsistema de memória, a economia obtida é superestimada. Se as caches da arquitetura de referência não estão ajustadas à aplicação, elas consomem mais energia, elevando assim o consumo de energia da arquitetura de referência utilizada para normalização dos resultados. Quando a técnica de alocação em SPM é aplicada sobre a arquitetura-alvo, a SPM acaba por absorver boa parte da energia consumida pelas caches, pois espera-se que sejam alocados em SPM justamente os elementos de programa com maior taxa de faltas na cache (que, por isso, induzem maior consumo de energia na cache e na MP). Disto, pode-se deduzir que a economia resultante da alocação em SPM é superestimada, pois absorveu parte do consumo de energia de uma cache não-ajustada, o que não aconteceria se a cache tivesse sido ajustada à aplicação.

Considerando primeiramente técnicas OVB tendo arquiteturas sem cache (UNAs) como arquitetura-alvo, e FCAs como arquitetura de referência, Steinke et al. (2002a) reportam uma economia média de energia de 30%, Verma e Marwedel (2007) de 20% e Egger et al. (2010) de 21%. Como as caches de referência não foram pré-ajustadas,

o ganho da alocação em SPM está superestimado. Em sistemas com caches pré-otimizadas, deve-se esperar ganhos muito menores que aqueles reportados.

Quando as arquiteturas-alvo são CBAs, Cho et al. (2007) relatam economia média de energia de 8% para dados, Egger, Lee e Shin (2008) e Egger et al. (2010) relatam, 14% e 24% para código, respectivamente. Os ganhos são menores mas — pode-se supor — superestimados, pois as caches também não foram otimizadas.

Além disso, algumas técnicas ignoram as diferenças entre as arquiteturas-alvo e estabelecem uma comparação direta com uma técnica proposta para uma arquitetura diferente da sua, superestimando os resultados. Por exemplo, Egger et al. (2006) (UNA) relatam uma economia de energia de 24% sobre Angiolini et al. (2004) (CBA), o que não é justo, pois a presença de cache em uma CBA diminui o potencial de otimização da SPM, favorecendo os ganhos da UNA.

Por conta disso, faz-se necessária a escolha apropriada da configuração das caches da arquitetura-alvo, de modo que os resultados obtidos não sejam mascarados por conta de uma determinada configuração, mas, sim, verdadeiramente significativos, evidenciando o real impacto da técnica sobre o programa otimizado.

Para evitar que a escolha arbitrária dos parâmetros das caches de uma CBA favoreça os ganhos obtidos através da alocação em SPM, optou-se por estimar, para cada programa, as caches de instruções e de dados que resultam no menor consumo de energia. Desta forma é possível avaliar adequadamente o impacto da granularidade de código e do tamanho da SPM no consumo de energia. Tal otimização dos parâmetros da cache pode ser feita por meio de técnicas de ajuste-fino de caches (*cache-tuning*).

5.1 AS TÉCNICAS DE AJUSTE-FINO DE CACHES

As técnicas de ajuste-fino das memórias cache (*cache-tuning*) procuram estimar o comportamento de um determinado espaço de projeto de caches para uma aplicação. Dependendo da técnica, este espaço de projeto pode corresponder a uma ou a diversas configurações, obtidas pela variação de um ou mais parâmetros da cache. A captura do padrão de acesso da aplicação é feita pelo processamento do seu *trace* de execução.

As principais técnicas de ajuste-fino de memórias cache são enumeradas em Viana (2006). A seguir, as principais técnicas de ajuste-fino

de caches são descritas muito brevemente, uma vez que neste trabalho não se pretende propor nova técnica, mas apenas adaptar uma das existentes, ao contexto da infraestrutura experimental.

As técnicas mais simples são aquelas dirigidas pelo *trace* de endereços, que simulam, a partir de uma sequência de acessos, o comportamento de acertos e faltas para uma dada configuração.

Outro grupo de técnicas simula não apenas o comportamento da cache mas sua interação com o processador, por meio da simulação da cache simultaneamente à execução do programa, realizada por um simulador de conjunto de instruções. Estes dois grupos de técnicas, porém, permitem a avaliação de uma única configuração de cache por vez e são demasiadamente lentas, o que torna seu uso proibitivo para a exploração de um espaço de projeto contendo centenas ou até mesmo milhares de configurações de caches diferentes.

Na tentativa de solucionar este problema, diversos autores propuseram um terceiro grupo de técnicas, denominadas de **técnicas de passagem única** (MATTSON; GECSEI; TRAIGER, 1970) (HILL; SMITH, 1989) (SUGUMAR; ABRAHAM, 1995) (CONTE; HIRSCH; HWU, 1998) (CASCAVAL; PADUA, 2003) (JANAPSATYA; IGNJATOVIĆ; PARAMESWARAN, 2006a) (VIANA, 2006; GORDON-ROSS et al., 2007; VIANA et al., 2008).

Estas técnicas são mais interessantes, pois permitem a simulação de um conjunto de caches de uma única vez, utilizando também o *trace* de endereços. O conjunto de configurações de caches a ser simulado é determinado pela restrição quanto aos valores máximos e mínimos de alguns parâmetros da cache, sendo os principais: capacidade da cache, tamanho de bloco e associatividade, nesta ordem (ZHANG; VAHID, 2003).

Porém, a grande maioria das técnicas não permite a variação simultânea destes três parâmetros, sendo um ou mais fixos, caso em que possivelmente se fará necessária mais de uma simulação para a cobertura de todo o espaço de projeto considerado.

Para o ajuste-fino das memórias cache a serem utilizadas na configuração experimental, optou-se por implementar e utilizar o método SPCE (VIANA, 2006; GORDON-ROSS et al., 2007; VIANA et al., 2008), por este levar em consideração os três parâmetros enumerados anteriormente em uma passagem única.

5.2 O MÉTODO SPCE

Dados um *trace* de memória e restrições de espaço de projeto (tamanho mínimo e máximo de cache, tamanho mínimo e máximo de bloco e maior grau de associatividade permitido), o método SPCE identifica, para cada endereço α_i do trace, todas as configurações possíveis dentro do espaço de projeto para as quais o acesso ao endereço resulta em acerto. Uma estrutura de tabela em múltiplas camadas armazena o número de acertos computados ao longo da execução da técnica, e uma pilha armazenando os endereços já processados auxilia no cálculo do número de conflitos, o que é usado para a determinação de acerto ou falta para o acesso. Uma vez que o trace todo foi analisado, valores de energia são computadas para cada configuração de cache, usando estimativas de energia obtidas do pacote CACTI 5.3[1] (THOZIYOOR et al., 2008) combinado com o número de acertos e faltas derivados da tabela. Finalmente, a configuração de melhor eficiência energética é selecionada.

Mais detalhes sobre o método SPCE encontram-se no Apêndice A.

5.3 IMPLEMENTAÇÃO DO MÉTODO SPCE

Com base na literatura disponível (VIANA, 2006) (GORDON-ROSS et al., 2007) (VIANA et al., 2008), foi feita uma implementação do método SPCE como um programa *standalone* C++. As restrições de espaço de projeto são passadas como argumentos via linha de comando. O *trace* de endereços é obtido via simulador (modelo executável) para o processador-alvo MIPS (descrito na Seção 6.1), e fornecido à ferramenta por comunicação entre processos.

Para esta implementação do método de Viana, diversas dificuldades tiveram que ser vencidas. Por exemplo, o algoritmo mostrado em Gordon-Ross et al. (2007) e Viana et al. (2008) não apresenta insumos suficientes para viabilizar sua implementação. Em sua tese de doutorado (VIANA, 2006), Viana não apresenta nenhum algoritmo e os poucos exemplos demonstram-se insuficientes para preencher as lacunas deixadas pelo algoritmo descrito em seus outros trabalhos, enumerados anteriormente. Estas lacunas somente foram preenchidas por meio de

[1]CACTI é um modelo físico de memórias cache, SPM e DRAM. Utilizando um conjunto de mais de 30 parâmetros de entrada para a caracterização da memória desejada, o modelo do CACTI permite a estimativa de uma série de atributos temporais, energéticos e de área desta memória.

dedução e comparação com os exemplos apresentados.

Um problema que atinge não só o método SPCE, mas todos os métodos de ajuste-fino de caches, é a velocidade de execução. Os autores relatam em Viana et al. (2008) uma aceleração de quase 15 vezes (VIANA et al., 2008) em relação à ferramenta Dinero[2] (HILL et al., 1993) e uma aceleração de quase 2 vezes com relação à uma heurística de ajuste-fino (ZHANG; VAHID; NAJJAR, 2005). Isso tudo, aliado ao baixo tempo de execução médio (120s), passa a falsa sensação de que a técnica é extremamente rápida. Uma análise mais detalhada, porém, revela que a explicação para tal rapidez encontra-se nos programas-alvo escolhidos. O método é experimentado sobre um conjunto de 9 programas do *benchmark* Powerstone (SCOTT et al., 1998), que são programas extremamente pequenos e rápidos. A aplicação sobre programas mais realistas (como os relatados na Seção 6.1 do presente trabalho) demanda um tempo de execução muito maior. Exemplos destes tempos são apresentados no final desta seção.

A explicação para tal aumento no tempo de execução é porque, para programas muito grandes, uma enorme gama de endereços distintos são processados e necessitam ser armazenados em uma pilha. Uma operação de busca é realizada para cada endereço acessado, o que pode tomar um tempo considerável, dependendo do tamanho da pilha.

Para contornar estas limitações, Viana (2006) propõe duas alternativas: uma amostragem (ι/θ) do *trace* de endereços, i.e. são processados ι endereços sequenciais, os θ seguintes são ignorados, e assim sucessivamente; ou o limite do tamanho da pilha.

Para a primeira sugestão, Gordon-Ross et al. (2007) relatam que a amostragem de apenas 3% do total de endereços (usando as taxas $\iota = 4/\theta = 128$, $\iota = 16/\theta = 512$ e $\iota = 64/\theta = 2048$) conduz a resultados médios que diferem por apenas 4% do valor ótimo. Dentre estas taxas, $\iota = 64/\theta = 2048$ é a que apresenta o melhor resultado. Para alguns programas de *benchmark* relatados, contudo, o uso de amostragem levou a erros consideráveis, independentemente da taxa considerada. A causa não foi identificada pelos autores até o presente momento.

Para a segunda sugestão, Viana (2006) não apresenta valores factíveis nem o erro causado por esta restrição.

A opção adotada pelo presente trabalho não foi nem uma nem outra. Para não comprometer de nenhuma maneira a estimativa do mé-

[2]Dinero é uma conhecida ferramenta dirigida por *trace* para o cálculo das taxas de acertos e faltas de uma determinada configuração de cache. Para a exploração de espaço de projeto contendo múltiplas caches, pode ser enquadrada como técnica de ajuste-fino de passagens múltiplas (uma para cada cache a ser avaliada).

todo, optou-se por paralelizar o algoritmo utilizando diretivas OpenMP. Mesmo assim, o ajuste-fino de certos programas levou muito tempo. Por exemplo, para o programa *basicmath*, o ajuste-fino da I-cache e da D-cache demoraram, respectivamente, 3 dias e 15 horas, e quase 19 horas. Para o programa *fft*, o ajuste da I-cache levou 2 dias e 15 horas, e o da D-cache levou 12 horas. Estes tempos foram medidos em uma máquina com quatro-núcleos, descrita em maiores detalhes na Seção 6.1.

5.4 DETERMINAÇÃO DAS CACHES PRÉ-AJUSTADAS

O espaço de projeto considerado para o pré-ajuste das memórias cache é ilustrado na Tabela 3, com todos os valores permitidos para cada um dos três parâmetros considerados pela técnica SPCE. A capacidade da SPM, expressa em bytes (B), variou entre 1K e 8K. O tamanho de bloco variou entre 8B e 32B. E, para a associatividade, considerou-se desde o mapeamento direto (1) até 8-vias.

A Tabela 4 apresenta os parâmetros de configuração obtidos do ajuste-fino das caches de instruções (I-cache) e dados (D-cache) para cada programa-alvo do *benchmark* MiBench (GUTHAUS et al., 2001) para o nodo tecnológico (*technology node*) CMOS de 90nm. As configurações são representadas por uma tupla (capacidade da cache, associatividade, tamanho de bloco), onde os tamanhos de cache e de bloco são expressados em bytes. Mais sobre os programas-alvo e a configuração experimental utilizada encontra-se na Seção 6.1.

5.5 IMPACTO DO AJUSTE-FINO NA ECONOMIA DE ENERGIA

O algoritmo de ajuste-fino permite a estimativa do consumo de energia de cada uma das caches do espaço de projeto considerado. Determinando-se as caches de melhor e pior eficiência energética — i.e. menor e maior consumo de energia — para cada programa, é possível normalizar o consumo das primeiras sobre o das últimas e estimar o impacto do ajuste-fino de caches na economia de energia para o espaço de projeto considerado, apresentado na Figura 9.

Em média, a economia de energia obtida pelo ajuste-fino foi de 88% para a I-cache e de 80% para a D-cache. A maior redução foi obtida para a I-cache do programa *basicmath* (próximo de 100%). As menores reduções foram obtidas para a D-cache de FFT e $IFFT$: 38% e 37%, respectivamente. Estes resultados corroboram a necessidade do ajuste-

Tabela 3: Espaço de projeto considerado para ajuste-fino das memórias cache

	Min.			Máx.
Capacidade (B)	1K	2K	4K	8K
Tamanho de bloco (B)	8	16		32
Associatividade	1	2	4	8

Tabela 4: Resultado do ajuste-fino das memórias cache

	Configuração		
Programa	**I-cache**	**D-cache**	$\mathbf{C_T}$
basicmath	(4K,16,1w)	(1K, 8 ,1w)	32K
bitcount	(1K, 8 ,1w)	(1K, 8 ,1w)	2K
qsort	(8K,16,2w)	(1K, 8 ,2w)	16K
susan_edges	(8K,16,2w)	(2K, 8 ,1w)	16K
susan_smoothing	(1K, 8 ,1w)	(2K, 8 ,2w)	4K
cjpeg	(4K, 8 ,2w)	(8K,16,2w)	16K
stringsearch	(4K, 8 ,2w)	(4K, 8 ,2w)	32K
dijkstra	(2K, 8 ,1w)	(8K,16,2w)	16K
blowfish_enc	(4K, 8 ,2w)	(8K, 8 ,1w)	16K
blowfish_dec	(4K, 8 ,1w)	(8K, 8 ,1w)	16K
rijndael_enc	(8K,16,2w)	(8K,16,2w)	16K
rijndael_dec	(8K, 8 ,4w)	(8K, 8 ,2w)	32K
sha	(1K, 8 ,1w)	(1K,16,1w)	2K
crc32	(1K, 8 ,1w)	(4K, 8 ,2w)	8K
fft	(4K, 8 ,2w)	(1K, 8 ,1w)	32K
ifft	(4K, 8 ,2w)	(1K, 8 ,1w)	32K
adpcm_code	(1K, 8 ,1w)	(4K, 8 ,1w)	8K
adpcm_decode	(1K, 8 ,1w)	(4K, 8 ,1w)	8K
gsm_toast	(2K, 8 ,1w)	(2K, 8 ,1w)	8K
gsm_untoast	(2K, 8 ,1w)	(4K, 8 ,1w)	16K

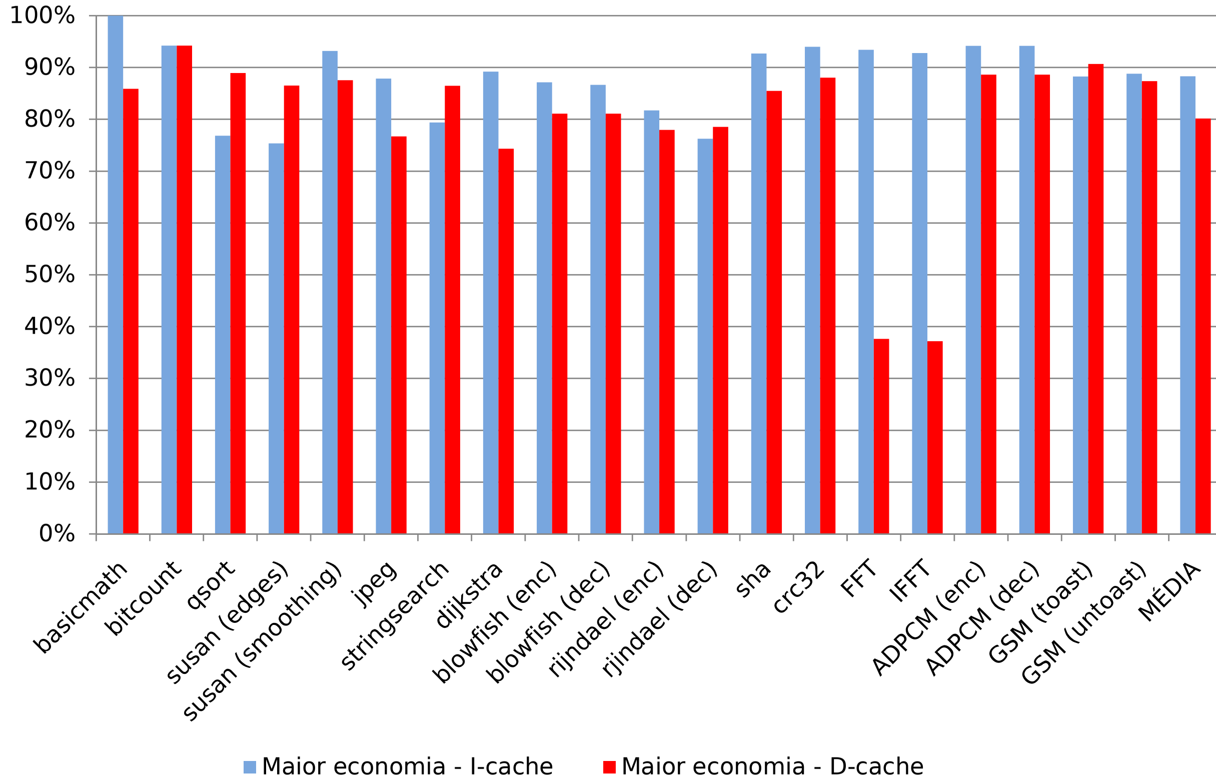


Figura 9: Impacto do ajuste-fino na economia de energia das caches de instruções e dados

fino previamente à alocação em SPM. Em sistemas sem o pré-ajuste das caches, uma quantia considerável dessa economia — senão toda ela — teria sido atribuída intuitiva e indevidamente à alocação em SPM.

Assim sendo, é preciso ter sempre em mente que as economias apresentadas na Seção 6.4, medidas após o ajuste-fino, são certamente menores do que aquelas avaliadas em sistemas com caches não-ajustadas, que acabam por consumir mais energia por acesso e sofrem mais faltas, ocasionado mais acessos aos níveis superiores da hierarquia de memória. Para alguns casos, como será mostrado na Seção 6.4.2, o pré-ajuste diminui o espaço de otimização a ser explorado pela alocação em SPM.

5.6 CÁLCULO DA CACHE UNIFICADA EQUIVALENTE

Para correlacionar as capacidades da SPM e das caches pré-ajustadas, é conveniente definir a capacidade de uma cache unificada equivalente (T-cache), equivalente em termos de taxa de faltas àquelas previamente ajustadas. Tal capacidade, denotada por C_T, é calculada como segue.

Sejam m_I and m_D as taxas de faltas locais da I-cache e D-cache, respectivamente. Seja LS a fração do número de instruções executadas que representam instruções de carga (*load*) e escrita (*store*). Então, a taxa de faltas combinada é o número total de faltas, dividido pelo número total de acessos para ler instruções ou ler/escrever dados:

$$m_T = \frac{m_I + LS \times m_D}{1 + LS} \quad (5.1)$$

Por simplicidade, considera-se que a taxa de faltas e a capacidade de uma cache são inversamente proporcionais, i.e.

$$\begin{aligned} k_I &= m_I \times C_I, \\ k_D &= m_D \times C_D, \\ k_T &= m_T \times C_T \end{aligned}$$

As constantes k_I, k_D e k_T podem ser interpretadas como a média do número de bytes transferidos de ou para a MP. Como a T-cache deve transferir a mesma quantia de bytes que I-cache e D-cache juntas, tem-se $k_T = k_I + k_D$, o que leva a:

$$C_T = \frac{C_I \times m_I + C_D \times m_D}{m_T} \quad (5.2)$$

A última coluna da Tabela 4 descreve as capacidades das caches equivalentes unificadas, obtidas a partir das Equações 5.1 e 5.2 para cada programa.

6 VALIDAÇÃO EXPERIMENTAL E RESULTADOS

6.1 CONFIGURAÇÃO EXPERIMENTAL

Para a avaliação do impacto do pré-ajuste das memórias cache, bem como da granularidade de código, utilizou-se da seguinte infraestrutura.

Os arquivos-objeto relocáveis foram produzidos utilizando o compilador cruzado (*cross-compiler*) `gcc` (versão 4.4.1) combinado com a biblioteca *newlib* (RedHat Inc, 2010), uma implementação reduzida da biblioteca padrão da linguagem C, própria para sistemas embarcados. Como o modelo executável do processador (descrito abaixo) não suporta instruções de ponto-flutuante, utilizou-se a biblioteca `soft-float` para emular estas instruções por outras de matemática inteira. A compilação utilizou o parâmetro de otimização `-Os`, que previne otimizações que resultem em aumento do tamanho de código, uma escolha apropriada para sistema embarcados com capacidade limitada de memória. Para gerar os arquivos executáveis a partir dos relocáveis, utilizou-se o linkeditor `ld` disponível no pacote GNU Binutils (BINUTILS, 2007).

O ambiente de simulação utilizado consistiu de um modelo executável do processador e um modelo do subsistema de memória. O modelo executável do processador MIPS (um simulador do conjunto de instruções gerado pela ADL ArchC (RIGO et al., 2004)) gera o *trace* de endereços acessados, o qual serviu de entrada para os seguintes artefatos computacionais: o algoritmo de ajuste-fino, o *profiler* e o modelo parametrizável do subsistema de memória[1]. Este modelo (pré-validado) permitiu a composição de distintas arquiteturas (do subsistema de memória), utilizando os seguintes componentes: uma memória principal (MP) externa (*off-chip*), caches de instrução e de dados (nível 1) e, opcionalmente, uma SPM. Também permitiu a estimativa do consumo de energia dinâmica de cada componente — por conseguinte, do subsistema de memória como um todo, com exceção de barramentos de interconexão.

Como MP, assumiu-se o uso de uma memória *off-chip Micron MT48H8M16LF low-power SDRAM*, a qual é a mesma de outros trabalhos, como Deng et al. (2009), Egger, Lee e Shin (2006) e Egger et al. (2006). Os principais parâmetros desta memória são: C_{MM} = 128MB,

[1]Ferramenta implementada por Rafael Westphal, Alexandre Keunecke I. de Mendonça e Daniel Pereira Volpato.

V_{DD}=1,8V, e barramento da memória operando a $100MHz$. Os valores de energia e latência modelados foram os mesmos relatados naqueles trabalhos — os quais, coincidentemente, são também os mesmos adotados por Cho et al. (2007), Kannan et al. (2009), Egger, Lee e Shin (2008) e Egger et al. (2010) para uma memória diferente.

Considera-se que a MP é organizada como uma memória larga — i.e., a largura de banda da MP e a largura do barramento de interconexão da MP com o processador são do mesmo tamanho do bloco das caches. No caso de a I-cache e a D-cache possuírem tamanhos de bloco diferentes, considera-se o maior deles. Esta organização permite que um bloco seja transferido das caches sempre de uma única vez, sem a necessidade de acessos extras à MP.

Para as memórias cache e SPM, os parâmetros dependentes de tecnologia (consumo de energia por acesso e tempo de acesso) foram obtidos através do modelo físico de memórias CACTI 5.3 (THOZIYOOR et al., 2008), para um nodo tecnológico de 90nm.

Os experimentos foram realizados em uma máquina com processador Intel Xeon E5430 (*quad-core*) 2,66GHz com 4GB de memória principal, rodando o sistema operacional Ubuntu GNU/Linux (*kernel* 2.6.31, 32-bit).

6.2 GERAÇÃO DOS EXPERIMENTOS

O conjunto de programas-alvo consistiu-se de 20 programas pertences a todas as seis classes de aplicações do *benchmark* Mibench (GUTHAUS et al., 2001)[2]. A relação dos programas bem como uma breve descrição de sua função são apresentados na Tabela 5. O tamanho do arquivo executável de cada programa (incluídas as bibliotecas estáticas), quando compilado na infraestrutura mencionada anteriormente, é apresentado na segunda coluna da Tabela 6. Os parâmetros de entrada aplicados em cada programa foram retirados dos casos de teste correspondentes, denominados `large`, do mesmo *benchmark*.

Para cada programa, avaliou-se a energia consumida por duas arquiteturas de memória distintas. Para fins de normalização, empregou-se uma FCA (com I-cache, D-cache e MP externa) como *arquitetura de memória de referência* (*reference memory architecture* - REF). Ao contrário dos trabalhos anteriores, adotou-se uma referência particular para

---

[2] Apesar de a técnica utilizada ser capaz de realizar a alocação em SPM para todos os benchmarks propostos, a infraestrutura de compilador cruzado (*cross-compiler*) e simulador de conjunto de instruções utilizada impediu o *profiling* de alguns dos programas do *benchmark* Mibench (GUTHAUS et al., 2001).

Tabela 5: Descrição dos programas de *benchmark* utilizados

Programa	**Descrição**
basicmath	Série de cálculos matemáticos simples, sem suporte usual de hardware em sistemas embarcados.
bitcount	Testa as operações de manipulação de bits do processador.
qsort	Usa o algoritmo *quick sort* para ordenar um grande arranjo de *strings*.
susan (edges)	Pacote para reconhecimento de bordas e cantos em imagens de ressonância magnética do cérebro. Usa o modo para bordas.
susan (smoothing)	O mesmo que o anterior, exceto que utiliza o modo de suavização.
jpeg	Compactação de imagens usando o algoritmo JPEG de compressão com perda de dados.
stringsearch	Procura por certas palavras em frases, sem sensibilidade à caixa (*case insensitive*).
dijkstra	Aplica o algoritmo de Dijkstra para caminho de custo mínimo sobre um grafo grande.
blowfish (enc)	Usa o cifrador blowfish para criptografar blocos.
blowfish (dec)	Usa o cifrador blowfish para descriptografar blocos.
rijndael (enc)	Usa o cifrador rijndael para criptografar blocos.
rjindael (dec)	Usa o cifrador rijndael para descriptografar blocos.
sha	Aplica o algoritmo de dispersão (*hashing*) SHA (*Secure Hash Algorithm*).
crc32	Realiza uma Verificação de Redundância Cíclica (*Cyclic Redundancy Check*) sobre um arquivo.
FFT	Aplica a Transformada Rápida de Fourier sobre um arranjo de dados.
IFFT	Aplica a Inversa da Transformada Rápida de Fourier sobre um arranjo de dados.
ADPCM (enc)	Codifica uma amostra de voz usando a Modulação Diferencial Adaptativa por Código de Pulsos (ADPCM).
ADPCM (dec)	O mesmo que o anterior, exceto que trata-se da descodificação.
GSM (toast)	Codifica uma amostra de voz usando o padrão GSM (*Global Standard for Mobile*).
GSM (untoast)	O mesmo que o anterior, exceto que trata-se da descodificação.

cada programa de *benchmark*, com parâmetros de cache pré-ajustados conforme a Tabela 4. Como *arquitetura de memória sob avaliação* (*memory architecture under evaluation* - EVA), utilizou-se uma CBA que consiste na adição de uma SPM à REF, i.e. a SPM coexiste com as caches pré-ajustadas e a MP externa. Para cada programa, foram avaliadas 6 variações da CBA, determinadas pelo dimensionamento da capacidade da SPM (C_{SPM}) como múltiplo da capacidade (C_T) da cache equivalente unificada (T-cache): $\frac{1}{16}C_T$, $\frac{1}{8}C_T$, $\frac{1}{4}C_T$, $\frac{1}{2}C_T$, C_T e $2C_T$. O valor de C_{SPM} para cada programa é apresentado na Tabela 7.

Para cada programa, os mapeamentos ótimos foram procurados sob dois cenários distintos — *alocação de procedimentos* (*procedure allocation* - PRA) ou *alocação de blocos básicos* (*basic blocks allocation - BBA*) — e para as 6 diferentes capacidades de SPM. Para encontrar tais mapeamentos, empregou-se a abordagem NOB descrita nos Capítulos 3 e 4, a qual considera como candidatos tanto elementos de código como dados estáticos, independentemente de sua origem (da aplicação ou de bibliotecas), e que não necessita de hardware dedicado.

A solução ótima do Problema Alvo de Otimização (formalizado no Capítulo 3) foi encontrada pelo algoritmo MinKnap (PISINGER, 1997).

6.3 CARACTERIZAÇÃO DOS PROGRAMAS-ALVO

Como o impacto da alocação em SPM na redução de energia é fortemente dependente das propriedades dos programas-alvo, a simples avaliação da economia média para um conjunto de programas qualquer seria muito limitada e questionável, uma vez que os resultados poderiam ser influenciados pela escolha desse conjunto.

Para uma avaliação apropriada, deve-se correlatar a economia obtida da alocação em SPM com propriedades do programa, de modo a fornecer informação útil aos arquitetos de projeto e desenvolvedores de ferramentas de automação de projeto eletrônico (*Eletronic design automation* — EDA).

Para extrair estas propriedades de programa, submeteram-se todos os programas selecionados ao *profiler* para os dados de entrada mencionados na Seção 6.2.

A primeira propriedade extraída a partir de *profiling* foi o percentual de acessos que são acomodáveis em uma determinada capacidade, como mostram as últimas cinco colunas da Tabela 6.

Os valores claramente indicam que os programas apresentam *hot*

Tabela 6: Percentual de acessos acomodáveis em diferentes capacidades de uma memória qualquer

		Capacidade da memória (KB)				
Programa	**Tamanho**	**0,5**	**1**	**2**	**4**	**8**
basicmath	138KB	58,2%	72,5%	85,0%	93,8%	97,7%
bitcount	114KB	88,9%	100,0%	100,0%	100,0%	100,0%
qsort	170KB	31,1%	46,7%	70,2%	84,9%	96,9%
susan (edges)	193KB	40,6%	52,2%	67,0%	82,2%	94,5%
susan (smoothing)	193KB	91,7%	93,5%	94,4%	94,8%	95,0%
cjpeg	241KB	53,5%	66,4%	76,7%	90,5%	95,8%
stringsearch	128KB	64,0%	75,0%	86,4%	94,9%	99,5%
dijkstra	176KB	88,0%	90,2%	94,4%	97,3%	98,0%
blowfish (enc)	38KB	57,9%	68,1%	88,5%	94,8%	99,5%
blowfish (dec)	38KB	57,7%	68,0%	88,6%	95,3%	99,5%
rijndael (enc)	143KB	29,5%	34,6%	44,9%	65,4%	93,0%
rijndael (dec)	143KB	29,4%	34,6%	44,8%	65,4%	93,5%
sha	116KB	76,0%	93,4%	97,3%	98,4%	99,3%
crc32	112KB	94,8%	96,3%	99,2%	100,0%	100,0%
fft	147KB	54,5%	72,5%	88,8%	96,1%	98,5%
ifft	147KB	54,2%	72,4%	88,9%	96,3%	98,7%
adpcm (enc)	112KB	97,5%	98,3%	99,2%	100,0%	100,0%
adpcm (dec)	112KB	96,9%	97,9%	98,9%	100,0%	100,0%
gsm (toast)	177KB	78,9%	91,6%	95,4%	98,0%	99,3%
gsm (untoast)	177KB	78,9%	91,6%	95,4%	98,0%	99,3%
Média	172KB	66,1%	75,8%	85,2%	92,3%	97,9%

Tabela 7: Capacidade da SPM utilizada para cada configuração e programa

Programa	C_{SPM} (B)					
	$\frac{C_T}{16}$	$\frac{C_T}{8}$	$\frac{C_T}{4}$	$\frac{C_T}{2}$	C_T	$2C_T$
basicmath	2K	4K	8K	16K	32K	64K
bitcount	128	256	512	1K	2K	4K
qsort	1K	2K	4K	8K	16K	32K
susan (edges)	1K	2K	4K	8K	16K	32K
susan (smoothing)	256	512	1K	2K	4K	8K
jpeg	1K	2K	4K	8K	16K	32K
stringsearch	2K	4K	8K	16K	32K	64K
dijkstra	1K	2K	4K	8K	16K	32K
blowfish (enc)	1K	2K	4K	8K	16K	32K
blowfish (dec)	1K	2K	4K	8K	16K	32K
rijndael (enc)	1K	2K	4K	8K	16K	32K
rjindael (dec)	2K	4K	8K	16K	32K	64K
sha	128	256	512	1K	2K	4K
crc32	512	1K	2K	4K	8K	16K
FFT	2K	4K	8K	16K	32K	64K
IFFT	2K	4K	8K	16K	32K	64K
ADPCM (enc)	512	1K	2K	4K	8K	16K
ADPCM (dec)	512	1K	2K	4K	8K	16K
GSM (toast)	512	1K	2K	4K	8K	16K
GSM (untoast)	1K	2K	4K	8K	16K	32K

spots, pois concentram a maior parte dos acessos em uma capacidade muito pequena de memória. Para 16 dos 20 programas analisados, pelo menos 90% dos acessos poderiam acontecer no espaço de endereçamento de uma SPM de 4KB, o que corresponde a 2,3%, em média, do tamanho do programa.

Diante da variedade de programas e domínios de aplicação, estes resultados indicam que programas contendo *hot spots* não apenas são bastante comuns, como estes trechos frequentemente acessados podem ser alocados em SPMs de capacidade relativamente pequena, sem a necessidade de alterar dinamicamente sua alocação. Em outras palavras, a Tabela 6 apresenta evidências de que a alocação NOB é provavelmente uma abordagem pragmática para muitos programas. Estas evidências são confirmados por tabela similar exibida em Menichelli e Olivieri (2009), para outra implementação da biblioteca *libc* (que não a *newlib*) e para um menor conjunto de programas.

Outra propriedade extraída foi a *taxa de faltas global dos candidatos* ($\bar{m}$) para cada programa, dada pela Equação 6.1 e reportada na segunda coluna da Tabela 8. Como a taxa média de faltas dos BBs que formam um procedimento deve ser equivalente à taxa de faltas do próprio procedimento, $\bar{m}$ é independente da granularidade de código selecionada.

$$\bar{m} = \frac{(\sum_{i=1}^{n} a_i \times m_i)}{(\sum_{i=1}^{n} a_i)} \tag{6.1}$$

Para capturar as propriedades dependentes das granularidades de código, realizou-se *profiling* adicional para dois cenários distintos: um assumindo procedimentos (PRA) como elementos candidatos para alocação em SPM, outro assumindo blocos básicos (BBA). Para identificar os elementos considerados *hot spots* de um programa, avaliaram-se os elementos cujo número de acessos é muito superior ao número médio de acessos, conforme a Equação 6.2, onde $\bar{a}$ e σ são, respectivamente, a média e o desvio padrão do número de acessos.

$$H = \{D_i \mid a_i > \bar{a} + 3\sigma\} \tag{6.2}$$

O número absoluto de *hot spots* ($|H|$) e sua frequência de ocorrência, i.e. $h = (\sum_{i \in H} a_i)/(\sum_{i=1}^{n} a_i)$, para cada um dos cenários, são relatados nas demais colunas da Tabela 8.

Pode ser observado, por exemplo, que o programa crc32 exibe a maior frequência de *hot spots* combinado com uma das menores taxas de

Tabela 8: Propriedades extraídas para caracterização dos programas-alvo

		BBA		PRA	
Programa	$\bar{m}$	$\|H\|$	h	$\|H\|$	h
basicmath	5,32%	11	47,7%	4	62,3%
bitcount	0,00%	8	85,3%	2	57,0%
qsort	1,98%	8	24,1%	4	42,5%
susan (edges)	2,10%	12	45,8%	2	37,0%
susan (smoothing)	0,45%	1	93,7%	1	98,0%
cjpeg	0,75%	14	65,5%	4	66,5%
stringsearch	8,28%	2	38,6%	1	31,5%
dijkstra	5,42%	5	79,7%	1	67,3%
blowfish (enc)	0,40%	2	46,6%	2	63,8%
blowfish (dec)	0,84%	2	46,2%	2	64,4%
rijndael (enc)	1,50%	4	69,1%	1	61,6%
rijndael (dec)	2,10%	3	60,7%	1	61,7%
sha	5,22%	7	89,2%	1	76,1%
crc32	0,09%	4	95,9%	1	95,9%
fft	2,59%	11	57,0%	4	84,7%
ifft	2,54%	11	56,8%	4	84,8%
adpcm (enc)	0,88%	7	62,6%	1	93,9%
adpcm (dec)	2,09%	6	62,5%	1	92,7%
gsm (toast)	2,03%	3	65,5%	2	72,5%
gsm (untoast)	2,00%	11	71,6%	1	65,6%

onde:

$\bar{m}$: Taxa de faltas global dos candidatos;

$|H|$: Número de elementos candidatos classificados como *hot spots*;

h: Frequência de ocorrência dos elementos candidatos classificados como *hot spots*.

faltas. Apesar de mostrar uma frequência igualmente alta de *hot spots*, o programa sha apresenta, contudo, uma das maiores taxas de faltas. Como será mostrado na Seção 6.4, seus distintos níveis de localidade conduzem a padrões de economia muito diferentes, conforme varia-se o tamanho da SPM.

6.4 ANÁLISE DOS RESULTADOS

A energia consumida pelo subsistema de memória em cada caso, normalizada para a arquitetura de memória de referência, é dada por $E_N = E_{EVA}/E_{REF}$, e apresentada na Tabela 9. Ao todo, foram avaliados 240 casos (20 programas $\times$ 6 capacidades de SPM $\times$ 2 cenários de alocação. A economia corresponde a uma redução de energia, a qual pode ser determinada a partir dos valores normalizados (*economia* $= (1 - E_N) \times 100$). Todos os valores de energia reportados referem-se apenas ao subsistema de memória (e não a valores totais do sistema).

Também o espaço ocupado em SPM, normalizado para sua capacidade ($w_N = w_{EVA}/C_{SPM}$), é apresentado na Tabela 10 para estes mesmos casos. A (taxa de) ocupação da SPM pode ser determinada a partir dos valores normalizados, i.e. *ocupação* $= w_N \times 100$.

Os valores médios de economia de energia e taxa de ocupação para cada capacidade de SPM são apresentados graficamente na Figura 10.

Diversos aspectos destes resultados — relacionados ao dimensionamento da SPM, política de alocação, taxa de faltas, taxa de ocupação da SPM, etc. — são analisados nas seções subsequentes.

6.4.1 Sensibilidade da economia ao dimensionamento da SPM

Olhando-se simplesmente para a média calculada para o conjunto de programas, a economia não varia tanto de acordo com o tamanho da SPM. Os valores mínimo e máximo de economia média observada foram de 15% a 33% para PRA, e de 17% a 30% para BBA, conforme pode ser observado na Figura 10.

Entretanto, para os programas *stringsearch*, *sha*, *susan (smoothing)* e *adpcm (dec)*, a economia variou bastante com o tamanho da SPM. O aumento de economia proporcionado pelo dimensionamento da SPM para estes programas, usando abordagem PRA, é, respectivamente, de 25, 39, 39 e 94 pontos percentuais. Esses e os demais valores são ilustrados na Figura 11, que apresenta a menor e a maior economia,

Tabela 9: Energia normalizada para a configuração de cache pré-ajustada

Programa	E_N (BBA)						E_N (PRA)					
	$\frac{C_T}{16}$	$\frac{C_T}{8}$	$\frac{C_T}{4}$	$\frac{C_T}{2}$	C_T	$2C_T$	$\frac{C_T}{16}$	$\frac{C_T}{8}$	$\frac{C_T}{4}$	$\frac{C_T}{2}$	C_T	$2C_T$
basicmath	0,92	0,90	0,87	0,86	0,87	0,87	0,86	0,79	0,75	0,70	0,68	0,69
bitcount	0,99	0,99	0,99	0,99	1,00	1,00	1,00	0,99	1,23	0,99	0,99	0,99
qsort	0,93	0,91	0,88	0,88	0,89	0,86	0,89	0,91	0,90	0,84	0,83	0,83
susan (edges)	0,87	0,79	0,72	0,72	0,72	0,73	0,93	0,73	0,70	0,70	0,71	0,71
susan (smoothing)	0,86	0,80	0,73	0,69	0,66	0,66	0,95	0,91	0,80	0,63	0,56	0,47
cjpeg	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,91	0,91	0,91	0,92	0,95
stringsearch	0,52	0,48	0,49	0,49	0,49	0,49	0,64	0,58	0,44	0,39	0,40	0,40
dijkstra	0,33	0,30	0,28	0,28	0,29	0,30	0,27	0,28	0,23	0,22	0,21	0,22
blowfish (enc)	0,97	0,95	0,95	0,96	0,96	0,96	0,96	0,95	0,95	0,95	0,96	0,97
blowfish (dec)	0,95	0,92	0,92	0,93	0,93	0,94	0,92	0,91	0,91	0,91	0,92	0,93
rijndael (enc)	0,92	0,89	0,85	0,81	0,78	0,79	1,20	0,85	0,84	0,83	0,78	0,79
rijndael (dec)	0,79	0,78	0,78	0,78	0,73	0,74	0,79	0,79	0,78	0,72	0,73	0,73
sha	0,75	0,72	0,44	0,40	0,40	0,40	0,84	0,84	0,55	0,45	0,45	0,45
crc32	0,99	0,99	0,99	0,99	0,99	0,99	0,99	0,99	0,99	0,99	0,99	1,00
fft	0,91	0,87	0,85	0,83	0,83	0,84	0,90	0,86	0,81	0,79	0,78	0,78
ifft	0,91	0,87	0,84	0,83	0,83	0,84	0,90	0,86	0,81	0,79	0,78	0,78
adpcm (enc)	0,11	0,10	0,10	0,10	0,11	0,11	0,10	0,06	0,06	0,08	0,09	0,10
adpcm (dec)	0,99	1,00	0,05	0,04	0,04	0,05	0,97	0,98	0,05	0,04	0,04	0,04
gsm (toast)	0,97	0,97	0,94	0,91	0,88	0,87	0,98	1,01	0,95	0,91	0,88	0,84
gsm (untoast)	0,93	0,87	0,75	0,75	0,76	0,76	0,92	0,78	0,69	0,69	0,69	0,69
Média	0,83	0,80	0,72	0,71	0,70	0,71	0,85	0,80	0,72	0,68	0,67	0,67

Tabela 10: Ocupação da SPM

Programa	**w_N (PRA)**											
	$\frac{C_T}{16}$	$\frac{C_T}{8}$	$\frac{C_T}{4}$	$\frac{C_T}{2}$	C_T	$2C_T$	$\frac{C_T}{16}$	$\frac{C_T}{8}$	$\frac{C_T}{4}$	$\frac{C_T}{2}$	C_T	$2C_T$
basicmath	1,00	1,00	1,00	1,00	0,81	0,40	1,00	1,00	1,00	1,00	1,00	0,75
bitcount	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
qsort	1,00	1,00	1,00	1,00	0,45	0,91	1,00	1,00	1,00	1,00	1,00	0,99
susan (edges)	1,00	1,00	1,00	1,00	0,80	0,32	1,00	1,00	1,00	1,00	1,00	0,69
susan (smoothing)	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
jpeg	1,00	1,00	1,00	1,00	1,00	0,68	1,00	1,00	1,00	1,00	1,00	1,00
stringsearch	1,00	1,00	0,61	0,27	0,13	0,07	1,00	1,00	0,79	0,79	0,39	0,19
dijkstra	1,00	1,00	1,00	1,00	0,63	0,31	1,00	1,00	1,00	1,00	1,00	0,84
blowfish (enc)	1,00	1,00	1,00	1,00	0,57	0,28	1,00	1,00	1,00	1,00	0,90	0,43
blowfish (dec)	1,00	1,00	1,00	0,78	0,64	0,31	1,00	1,00	1,00	1,00	0,90	0,45
rijndael (enc)	1,00	1,00	1,00	1,00	0,97	0,47	1,00	1,00	0,88	1,00	1,00	0,55
rjindael (dec)	1,00	1,00	1,00	1,00	0,56	0,27	1,00	1,00	1,00	1,00	0,70	0,35
sha	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
crc32	1,00	1,00	1,00	1,00	0,72	0,30	1,00	1,00	1,00	1,00	1,00	0,49
FFT	1,00	1,00	1,00	1,00	0,71	0,36	1,00	1,00	1,00	1,00	1,00	0,65
IFFT	1,00	1,00	1,00	1,00	0,68	0,34	1,00	1,00	1,00	1,00	1,00	0,64
ADPCM (enc)	1,00	1,00	1,00	1,00	0,69	0,34	1,00	1,00	1,00	1,00	0,77	0,39
ADPCM (dec)	1,00	1,00	1,00	1,00	0,70	0,35	1,00	1,00	1,00	1,00	0,77	0,39
GSM (toast)	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00
GSM (untoast)	1,00	1,00	1,00	1,00	1,00	0,62	1,00	1,00	1,00	1,00	1,00	0,83
Média	1,00	1,00	0,98	0,95	0,75	0,52	1,00	1,00	0,98	0,99	0,92	0,68

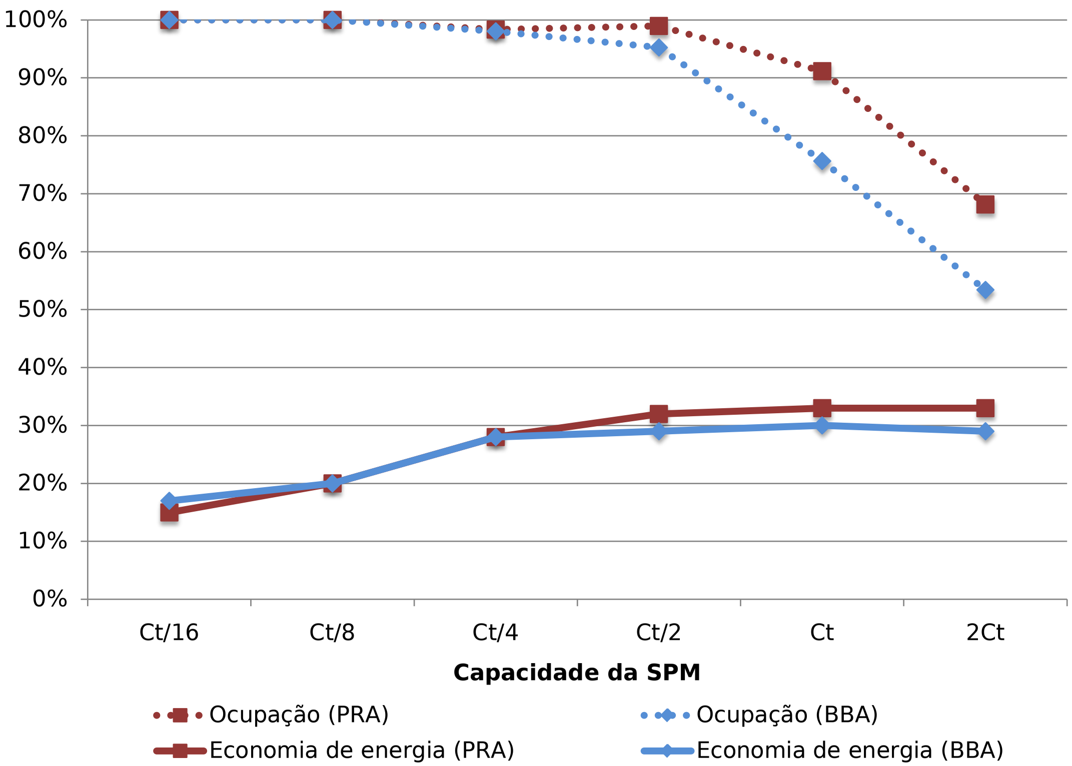


Figura 10: Economia média de energia e taxa média de ocupação por capacidade de SPM

usando abordagem PRA, obtidas pelo dimensionamento da SPM, bem como a diferença média, cujo valor é de aproximadamente 18%.

Atrai muita atenção o resultado da alocação para o programa *adpcm (dec)*, cujas economias mínima e máxima observadas sob PRA são de 2% e 96%. Isto pode ser explicado pelo seguinte: *adpcm (dec)* contém uma estrutura de dados (`sbuf`) frequentemente acessada na MP por conta de sua alta taxa de faltas ($m_{sbuf} = 0.999$) e seu grande número de acessos ($a_{sbuf} = 13305600$), sendo responsável por 95% da energia consumida pelo sistema de memória da REF. Uma vez que o tamanho de `sbuf` é maior do que a capacidade das menores SPMs ($C_T/16 < C_T/8 < \sigma_{sbuf} = 2000B$), a economia resultante é marginal. Tão logo essa estrutura possa ser alocada nas SPMs maiores, a economia torna-se extremamente alta.

Estes resultados mostram o **quão sensível ao dimensionamento da SPM alguns programas podem ser**.

6.4.2 Política de alocação de maior eficiência energética para uma determinada capacidade de SPM

Para um determinado tamanho de SPM, PRA e BBA conduzem essencialmente à mesma economia média. Todavia, dentre todos os casos avaliados, **PRA conduz à maior economia na maioria (61%) deles**, BBA em 20%, e ambas empatam nos 19% restantes.

Alguns casos particulares merecem comentários.

De um lado, para o programa *sha*, as economias de BBA prevalecem para todos os tamanhos de SPM. Isto pode ser explicado da seguinte maneira. Sob PRA, é impossível alocar os dois procedimentos mais acessados (sha_update e sha_transform) nas SPMs menores ($C_T/16 < C_T/8 < \sigma_{sha_update} < \sigma_{sha_transform}$). Entretanto, sob BBA, os BBs mais acessados destes procedimentos cabem nestas SPMs pequenas, permitindo maiores economias. Para as SPMs maiores, uma estrutura de dados muito acessada (`W`) participa da alocação: como ambas as políticas podem alocá-la, ambas conseguem economias maiores. Embora PRA possa agora mapear sha_update e sha_transform para as SPMs maiores, BBA ainda consegue melhores resultados porque uma pequena fração do código destes procedimentos reside em *hot spots*: sob BBA os BBs que correspondem ao código pouco acessado são substituídos por outros BBs com maior lucro.

Por outro lado, para o programa *crc32* praticamente não ocorre economia, independentemente de capacidade da SPM e de política de

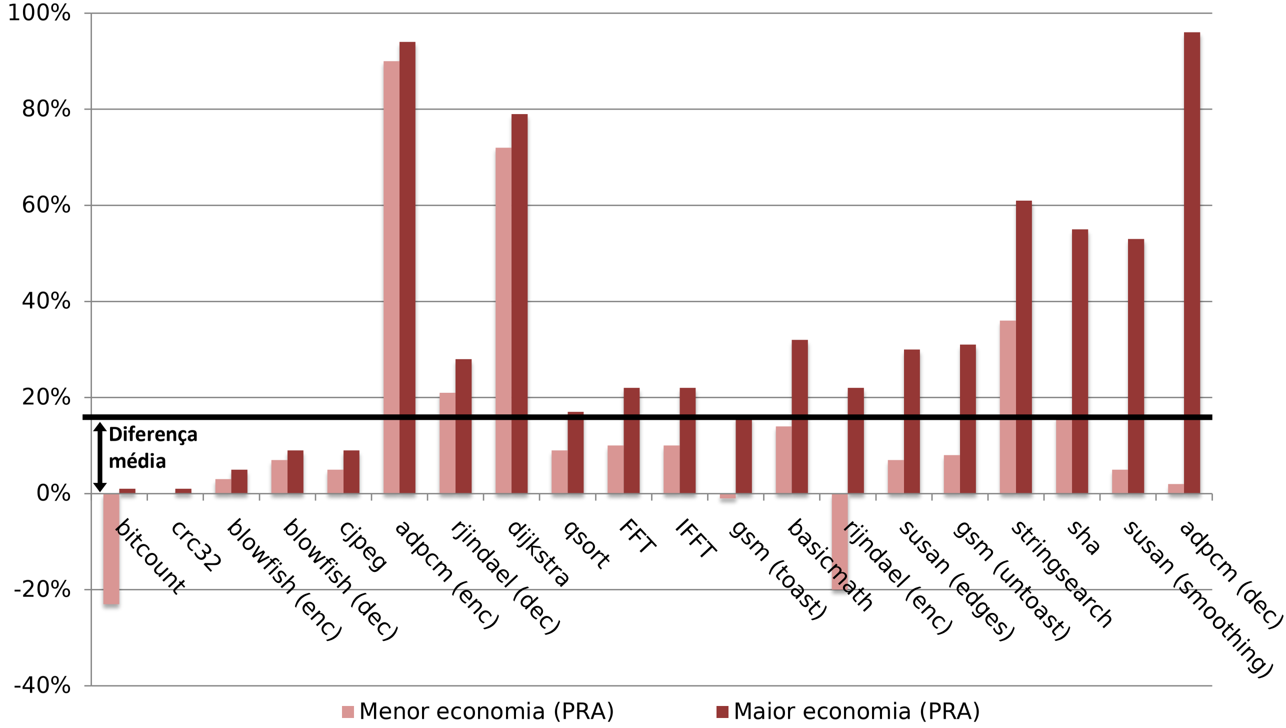


Figura 11: Sensibilidade da economia de energia ao dimensionamento da SPM (usando abordagem PRA)

alocação (BBA ou PRA). A razão é que, devido ao ajuste-fino das caches, a configuração REF exibe uma taxa de faltas extremamente baixa ($m_D = 0.05\%$, $m_I = 0.09\%$). Mesmo descartando-se o efeito devido aos elementos não-alocáveis (de pilha e de *heap*), a taxa de faltas dos candidatos continua pequena ($\bar{m} = 0.09\%$).

O efeito do pré-ajuste das caches aqui é interessantíssimo: **a otimização induzida pelo ajuste-fino deixa pouquíssimo espaço para uma economia adicional via alocação em SPM**. Conforme apresentado na Seção 5.5, o programa *crc32* experimentou uma economia de energia de 94% para a I-cache e de 88% para a D-cache pelo ajute-fino das caches. O ajuste-fino diminuiu a taxa de faltas das caches, o que impossibilita maiores ganhos via alocação em SPM. Se as caches não tivessem sido pré-ajustadas, uma parcela dessa economia teria sido atribuída indevidamente à alocação em SPM.

6.4.3 Política de alocação de maior eficiência energética para um determinado programa

A Figura 12 mostra a maior economia de energia obtida para cada um dos programas de *benchmark* considerados, utilizando as duas políticas de alocação suportadas, e dentro de todo o intervalo considerado ($[C_T/16, 2C_T]$). O primeiro programa apresentado, *sha*, é o único programa para o qual BBA resultou em maior economia do que PRA (cujo comportamento foi explicado anteriormente). Para os 5 programas seguintes, as duas políticas resultaram no mesmo valor de economia de energia. **PRA resultou em menor consumo de energia para 70% de todos os programas**, ou seja, os 14 programas apresentados a seguir.

6.4.4 Capacidade ótima da SPM

Em 85% dos programas (17/20), a capacidade de SPM que levou ao menor consumo de energia reside no **intervalo $[\mathbf{C_T/2, C_T}]$, o qual pode ser usado como "regra de ouro" para o dimensionamento da SPM em CBAs**, conforme mostrado pela Figura 13. Dos 3 programas restantes, em 2 destes (*susan (smoothing)* e *gsm (toast)*) a capacidade que permitiu uma maior redução do consumo de energia foi de $2C_T$, e no outro programa (*adpcm (enc)*) foi de $C_T/4$.

Alguns poucos programas obtiveram maior redução de consumo

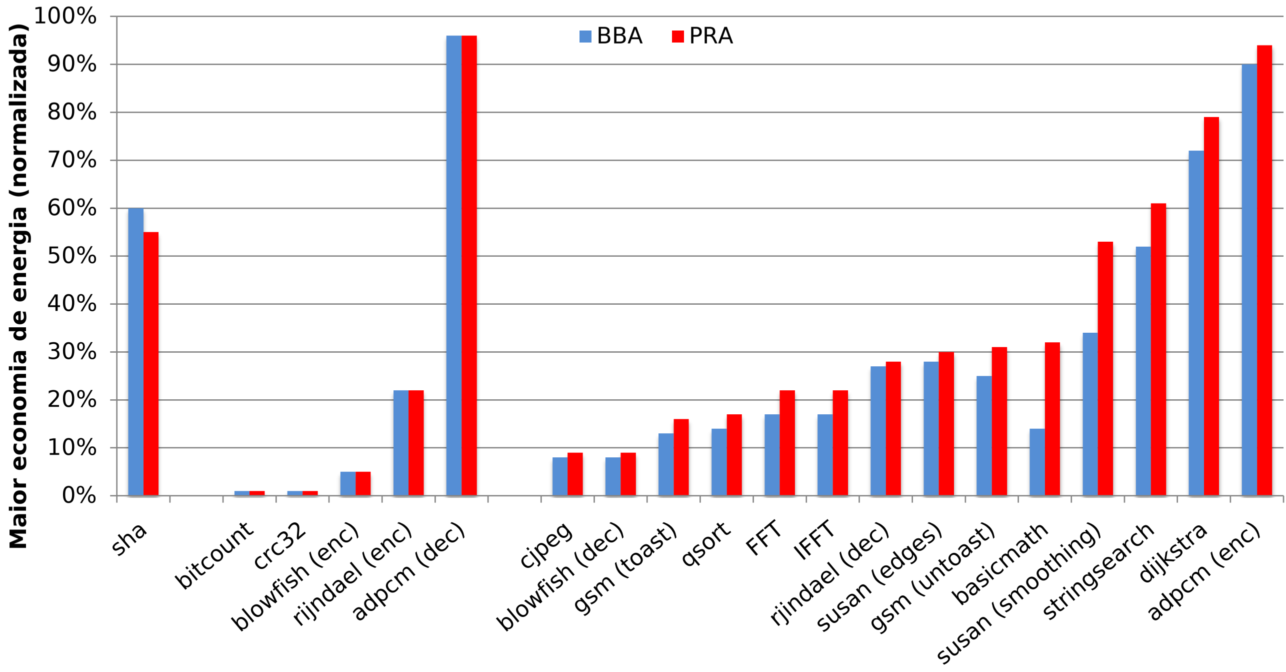


Figura 12: Maior economia de energia, utilizando BBA e PRA, para cada programa

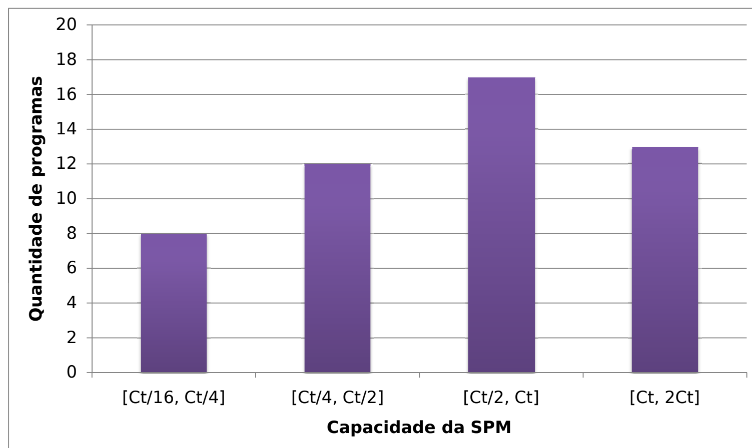


Figura 13: Capacidades de SPM que propiciam maior economia de energia

de energia também para SPMs de capacidade pequena ($[C_T/16, C_T/8]$). Entretanto, esta maior redução de energia não foi exclusividade destas capacidades, pois todos estes programas apresentaram consumo equivalente para SPMs de capacidades maiores. Como, além disso, a economia de energia destes programas não varia muito com o dimensionamento da SPM, estas capacidades pequenas podem ser descartadas do espaço de projeto da SPM. Isto permite que maiores economias de energia sejam obtidas, em média, para sistemas que executem mais de um programa.

Além disso, percebe-se que quando a capacidade da SPM passa de $C_T/8$ para $C_T/4$ ocorre o maior aumento médio de economia de energia, em torno de 8%, para ambas as políticas de alocação. Entre estas capacidades ocorre aumento significativo de economia de energia para os programas *sha* e *adpcm (decode)*. Da capacidade de SPM de $C_T/4$ para o intervalo da capacidade ótima, a economia de energia varia pouco (2% para BBA e 5% para PRA). Esta capacidade pode ser usada como regra para sistemas embarcados com restrição de área, levando ainda assim a uma economia satisfatória.

6.4.5 Correlação entre economia de energia e taxa de faltas para SPMs grandes

Considere-se o comportamento de programas para SPMs grandes, digamos $C_{SPM} \sim C_T$. Para esta capacidade de SPM, é possível observar uma correlação entre a economia de energia e a taxa de faltas global dos candidatos, apresentada na Figura 14. Novamente, a economia de energia foi medida usando a política de alocação PRA.

Para os 10 programas com menor $\bar{m}$, a economia média é de 23%, enquanto nos 10 programas com maior $\bar{m}$, a média é de 43%. A explicação para este comportamento é que, quando a SPM e a cache equivalente tem tamanhos comparáveis, os lucros tornam-se marginais ($C_{SPM} \sim C_T \Rightarrow E_{cache} - E_{SPM} \sim 0$), exceto para elementos faltando frequentemente nas caches (para os quais $m_i \times E_{MP}$ é maior, conforme indicado pelas Definições 3.5 e 3.6).

Em suma, **para SPMs grandes**, como $E_{SPM} \sim E_{cache}$, a alocação em SPM é dominada por elementos frequentemente acessados na MP, de modo que uma **maior taxa de faltas permite maior economia**.

6.4.6 Ocupação das SPMs ótimas

Dentro do intervalo contendo a maioria das capacidades ótimas de SPM para cada programa ($[C_T/2, C_T]$), PRA utiliza as capacidades quase ao máximo, enquanto **BBA resulta em SPMs menos povoadas**: em média, 96% da SPM é ocupada sob PRA e 85% sob BBA.

A chave para este fenômeno é a localidade. Quanto menor a taxa de faltas, menor a utilização da SPM sob BBA, porque o *overhead* de um candidato (ε_i) pode ser visto como um **limiar de lucro** ($p_i > 0 \Rightarrow a_i \times (E_i - E_{SPM}) > \varepsilon_i$ na Definição 3.6. Sob BBA e SPMs grandes, este limiar só pode ser atingido por alguns BBs com alta taxa de faltas, ao contrário de PRA, cujo limiar nulo ($\varepsilon_i = 0$) frequentemente permite alocação completa (apesar do lucro marginal de muitos dos procedimentos alocados).

6.4.7 Determinação de um escopo para utilização de BBA

Tal subpovoamento em SPMs grandes indica que BBA pode valer a pena em arquiteturas com SPMs pequenas, **especialmente para programas com taxa de faltas relativamente alta**. Esta

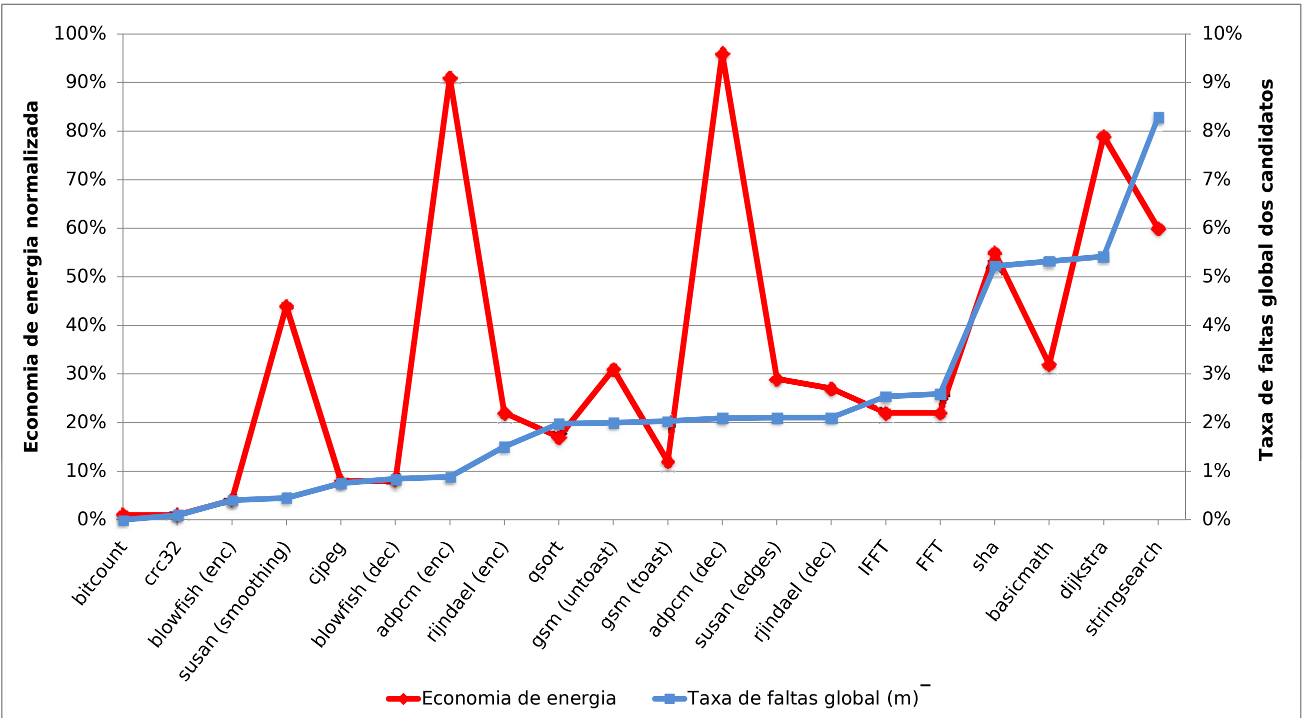


Figura 14: Correlação entre economia de energia e taxa de faltas global dos elementos candidatos, para SPMs grandes ($C_{SPM} \sim C_T$)

evidência pode ser reforçada a partir de outra perspectiva, como segue.

Dentro do intervalo $[C_T/16, 2C_T]$, BBA resulta na melhor economia de energia somente para um programa. Contudo, quando C_{SPM} é limitada para $[C_T/16, C_T/4]$, BBA consegue melhor economia para 3 programas (*sha*, *susan (smoothing)*, *gsm (toast)*). Além disso, do primeiro para o segundo intervalo, a percentagem de casos com energia mínima por configuração cresce de 20% para 28%.

6.4.8 Comparação com trabalhos correlatos

Estabelecer uma comparação direta com outros trabalhos exige uma configuração experimental equivalente, o que raramente é viável em trabalhos que envolvem muitas variáveis como: conjunto de instruções (ISA) do processador, tipo de MP (*on-chip* ou *off-chip*), implementação para sistemas embarcados da biblioteca *libc* utilizada, uso ou não de emulação de ponto-flutuante, etc.

Apesar destas diferenças, foram tomadas algumas medidas para amenizar o problema de uma comparação direta, como a utilização de uma MP também utilizada nos trabalhos correlatos (como descrito na Seção 6.1). Outras destas variáveis não puderam ser equiparadas devido ao pouco detalhamento na descrição da configuração experimental da grande maioria dos trabalhos correlatos. Para tais variáveis, buscou-se uma configuração usual e realista, sempre que possível. Por exemplo, como implementação da *libc* para sistemas embarcados, escolheu-se o pacote *newlib* (conforme descrito na Seção 6.1), por este ser amplamente difundido.

Contudo, algumas destas diferenças não puderam ser amenizadas pela configuração experimental. A infraestrutura experimental utilizada não permite que se estime a energia total consumida pelo sistema, pois ela não disponibiliza um modelo com precisão de ciclos (*cycle-accurate*) do processador. Logo, a infraestrutura experimental permite que seja mensurado apenas o consumo energético do subsistema de memória.

Para contornar esta limitação e permitir que os resultados de energia do subsistema de memória (E_{Mem}) deste trabalho sejam comparados aos resultados de energia total do sistema (E_{Total}) apresentados na literatura, considerou-se um **fator de proporcionalidade** k dado por:

$$k = \frac{E_{Mem}}{E_{Total}} \tag{6.3}$$

A análise de diversos trabalhos correlatos (ANGIOLINI et al., 2004; CHO et al., 2007; EGGER; LEE; SHIN, 2008; EGGER et al., 2010) permitiu estimar $0,98 \leq k \leq 1,01$, conforme apresentado em mais detalhes no Apêndice B. Ou seja, a alocação em SPM induz economia semelhante no consumo de energia do subsistema de memória e do processador, o que permite a comparação direta entre E_{Mem} e E_{Total} sem prejuízos quanto às conclusões daí derivadas.

Assim sendo, estabeleceu-se uma comparação com técnicas também propostas para CBAs e que também operam em arquivos binários. A economia média de memória obtida por este trabalho varia entre 15% a 33% para 6 capacidades distintas de SPM. Estes resultados são melhores do que a economia total relatada pelas técnicas OVB de Cho et al. (2007), Egger, Lee e Shin (2008) e Egger et al. (2010), que são de 8%, 14% e 24%, respectivamente.

Deve-se ressaltar que, ao contrário dos trabalhos correlatos, os resultados de economia de memória relatados por este trabalho são medidos em um subsistema de memória cujas caches foram ajustadas previamente à alocação em SPM. E, conforme já foi mostrado, o pré-ajuste das caches diminui o impacto da alocação em SPM.

Uma comparação com técnicas OVB para CBAs, mas que operam em código-fonte, revela que a economia de memória média deste trabalho é tão boa quanto a economia destas técnicas. Steinke et al. (2002a), por exemplo, relatam uma economia total de 30%.

Finalmente, os resultados deste trabalho, quando comparados com as técnicas OVB propostas para arquiteturas sem cache (UNAs) de Verma, Wehmeyer e Marwedel (2004b) e Udayakumaran, Dominguez e Barua (2006) mostram-se inferiores. Isto é explicado porque economias estimadas para UNAs tornam-se superestimativas para CBAs devido à interferência das caches.

7 CONCLUSÕES E PERSPECTIVAS

Este capítulo apresenta conclusões globais sobre a reavaliação experimental das técnicas NOB, realizada por esta dissertação. Tais conclusões são válidas apenas para arquiteturas com SPM baseadas em cache (CBAs), não podendo ser aplicadas para arquiteturas sem caches (UNAs) antes de maiores estudos. Também faz considerações sobre o dimensionamento da SPM (para as 6 capacidades de SPM consideradas) e a política de alocação (procedimentos ou blocos básicos), de acordo com os resultados obtidos. São, ainda, apresentadas conclusões sobre o ajuste-fino das memórias cache e o impacto de sua utilização como etapa anterior à alocação em SPM. Todas estas conclusões são detalhadas nas Seções 7.1 a 7.6. O capítulo encerra-se com perspectivas para trabalhos futuros, apresentadas na Seção 7.7.

7.1 EVIDÊNCIA EXPERIMENTAL SÓLIDA

O grande número de casos avaliados permite derivar conclusões a partir de evidências experimentais sólidas. O conjunto de 20 programas de *benchmark* utilizados para experimentação, bem como dos 240 casos avaliados, constituem número bem superior à maioria dos relatados pelos demais trabalhos de alocação em SPM encontrados na literatura. Dentre todos os trabalhos reportados na literatura até o momento, apenas o trabalho de Falk e Kleinsorge (2009) apresenta resultados para um maior número de programas.

7.2 IMPORTÂNCIA DO AJUSTE-FINO

A economia marginal obtida por alguns programas mostra que a SPM é inócua em alguns casos específicos, e que tão somente uma cache bem ajustada seria suficiente para prover uma economia de energia bastante satisfatória. Tal fato, por si, comprova a necessidade do ajuste-fino das caches no processo de melhoria da eficiência energética de um sistema embarcado.

Outro indicativo desta necessidade é a considerável variação nos parâmetros das configurações de caches pré-ajustadas pelo algoritmo de ajuste-fino. O resultado das configurações pré-ajustadas atesta a afirmação feita por Zhang e Vahid (2003) de que os parâmetros mais

importantes são o tamanho da cache, seguido pelo tamanho de bloco e, finalmente, pela associatividade.

A capacidade de ambas as caches (I-cache e D-cache) variou bastante, englobando todos os valores de tamanho possíveis no espaço de projeto, sem o predomínio de nenhum valor.

O tamanho de bloco variou pouco: 16 dos 20 programas (tanto para instruções como para dados) tiveram suas caches ajustadas para um tamanho de bloco de 8 bytes. Os programas restantes foram ajustados para 16 bytes e nenhum dos programas para 32 bytes.

A associatividade das caches teve variação um pouco maior do que o tamanho de bloco. Houve predomínio das configurações com mapeamento direto, embora algumas caches tenham sido ajustadas como associativas de 2 duas e, algumas poucas, de 4 vias.

7.3 IMPORTÂNCIA DA CORRELAÇÃO ENTRE TAMANHO DA CACHE PRÉ-AJUSTADA EQUIVALENTE E TAMANHO DA SPM

Teve fundamental importância a fixação das capacidades de SPM como múltiplos da capacidade da cache equivalente unificada (sobre as caches pré-ajustadas de dados e instruções). Isto permitiu que as conclusões pudessem ser feitas sobre capacidades da SPM diretamente relacionadas com uma propriedade dos programas-alvo, a taxa de faltas. Isto evita que uma seleção particular de programas ou de tamanhos de SPM possa influenciar a generalidade da análise dos resultados experimentais.

7.4 DIMENSIONAMENTO DA SPM

7.4.1 Impacto do dimensionamento

Neste trabalho, a capacidade da SPM (C_{SPM}) foi dimensionada como um múltiplo da capacidade da cache pré-ajustada equivalente (C_T). Ao todo, seis tamanhos distintos de SPM foram considerados: $\frac{1}{16}C_T$, $\frac{1}{8}C_T$, $\frac{1}{4}C_T$, $\frac{1}{2}C_T$, C_T e $2C_T$

O cálculo da economia média de energia para cada um destes 6 tamanhos de SPM (considerando os programas do *benchmark* MiBench) mostrou que a diferença entre a maior e a menor economia média de energia foi considerável. No caso de BBA, a maior economia média foi

de 30% ($C_{SPM} = C_T$), ao passo que a menor foi de 17% ($C_{SPM} = C_T/16$). Para PRA, a maior economia foi de 33% ($C_{SPM} = C_T$), ao passo que a menor foi de 15% ($C_{SPM} = C_T/16$). Ou seja, a economia praticamente dobrou com o aumento da capacidade da SPM[1].

Para alguns programas, o dimensionamento proporcionou uma melhora ainda mais significativa na redução de consumo de energia. Sob política PRA, destacam-se os programas *stringsearch*, *sha*, *susan (smoothing)* e, notadamente, o programa *adpcm (dec)*, para o qual houve redução de energia de apenas 2% no pior caso, enquanto no melhor caso a redução foi de 96%.

7.4.2 Diretrizes para dimensionamento

O pré-ajuste das memórias cache permitiu a identificação do intervalo de capacidades de SPM que levam às maiores reduções de energia: $[C_T/2, C_T]$. Neste intervalo obteve-se a maior redução de consumo de energia para 17 dos 20 programas-alvo. Assim, ele pode ser utilizado como diretriz para o dimensionamento de SPMs visando a maior redução de energia possível. Conforme observado nos resultados, nos casos em que a capacidade ótima de SPM não se encontra neste intervalo, ela certamente encontrar-se-á próxima dele, e portanto, ele pode ser utilizado como ponto de partida para exploração de redução de consumo de energia pelo dimensionamento da SPM.

Quando o sistema embarcado tiver restrição severa de área, uma SPM com capacidade de $C_T/4$ pode ser adotada como diretriz, para permitir um compromisso satisfatório entre área no circuito integrado e economia de energia.

Além disso, os resultados observados sustentam que, para uma determinação mais rápida da capacidade ótima da SPM, o intervalo $[C_T/16, C_T/8]$ pode ser descartado do espaço de projeto de CBAs, sem prejuízo à eficiência energética do sistema.

---

[1] *A priori* esta conclusão pode parecer paradoxal quando confrontada com a literatura. Contudo, o paradoxo não se configura, visto que a maioria dos trabalhos correlatos tratam de arquiteturas-alvo somente com SPM (UNAs), onde SPMs maiores levam a um maior consumo de energia. Para CBAs, contudo, as SPMs conseguem acomodar os elementos de baixa localidade que a cache não conseguiria acomodar.

7.4.3 SPMs grandes e as taxas de faltas

A correlação entre economia de energia e taxa de faltas foi investigada para SPMs grandes (i.e., $C_{SPM} \sim C_T$). Averiguou-se que a economia de energia é proporcional à taxa de faltas, i.e. os programas com maior taxa de faltas apresentaram maior economia de energia. Este comportamento é esperado pois, desta forma, a alocação de candidatos com grande taxa de faltas evitará vários acessos à MP, que consomem grande quantidade de energia.

7.5 POLÍTICA DE ALOCAÇÃO (GRANULARIDADE DE CÓDIGO)

7.5.1 Alocação de procedimentos (PRA)

Os resultados comprovam que a política de alocação de maior eficiência energética é PRA, embora, em média, os resultados de economia média sejam apenas levemente superiores aos de BBA. De modo geral, o limiar de lucro (*overhead*) nulo de PRA, ao mesmo tempo que leva a uma ocupação muito maior do espaço em SPM, permite a alocação de vários elementos com lucros muitíssimo pequenos. Estes lucros, quando somados, conduzem a uma maior economia de energia, ainda que marginal.

Surpreendeu a eficiência de PRA sobre SPMs pequenas (digamos, $[C_T/16, C_T/4]$). Para estes casos, a intuição diria que BBA deveria suplantar PRA. Entretanto, encontraram-se evidências de que isto geralmente não ocorre: em média, as duas políticas mostraram-se equivalentes.

Em termos de eficiência energética, PRA mostrou-se superior a BBA para uma determinada capacidade de SPM, permitindo maior economia em 61% dos casos e empatando em 19% deles. Adicionalmente, considerando-se cada programa com SPM no intervalo $[C_T/16, 2C_T]$, PRA obteve maior redução de energia que BBA para 70% dos programas.

7.5.2 Alocação de blocos básicos (BBA)

A política BBA apresentou, em média, resultados equivalentes a PRA. Analisando-se os resultados de energia para o intervalo de SPM $[C_T/16, 2C_T]$, percebe-se que BBA apresentou maior economia de energia somente para o programa *sha*. Por outro lado, restringindo o

intervalo para $[C_T/16, C_T/4]$ (o que equivale a SPMs pequenas), observa-se que BBA teve uma eficiência energética levemente superior a PRA, apresentando maior economia média para $C_{SPM} = C_T/16$. Em especial, neste intervalo BBA apresenta maior economia para 3 programas — *sha*, *susan (smoothing)* e *gsm (toast)*.

Os resultados permitiram identificar o escopo de maior eficiência energética para BBA como sendo a união de SPMs pequenas com programas-alvo que apresentam o seguinte comportamento: elementos candidatos frequentemente acessados que exibem taxas de faltas relativamente altas. Neste caso, o lucro da alocação destes elementos ultrapassa o limiar de lucro da política BBA (apresentado na Seção 6.4.6), resultando em maior economia.

Além disso, BBA possui uma vantagem sobre PRA, mesmo para SPMs grandes. O limiar de lucro de BBA impede que candidatos de lucro muito pequeno sejam alocados, resultando em uma menor ocupação da SPM.

7.6 REAVALIAÇÃO EXPERIMENTAL DAS TÉCNICAS NOB A PARTIR DE ARQUIVOS BINÁRIOS

A economia obtida sob uma abordagem NOB que considera bibliotecas foi, em média, de 15% a 33% para SPMs com capacidade entre $[C_T/16, 2C_T]$. Esta economia é melhor ou tão boa quanto aquelas reportadas por abordagens OVB que manipulam binários. Sua simplicidade, combinada com sua independência de hardware dedicado, fazem da abordagem NOB uma escolha pragmática para a alocação de SPMs a partir de binários, para CBAs.

Como as caches foram dimensionadas previamente à alocação em SPM, pode-se afirmar que as economias obtidas resultam única e exclusivamente da alocação em SPM.

Diante de tudo isso, pode-se verificar que a abordagem NOB não está ultrapassada. Os resultados obtidos após o ajuste-fino das caches reabilitam a abordagem NOB diante das OVBs, mostrando que possuem uma aplicação efetiva para viabilizar um maior espaço de otimização (incluindo elementos de bibliotecas) para a redução de energia em sistemas que não fazem uso de hardware dedicado para gerenciamento de SPM.

7.7 PERSPECTIVAS

Na análise da literatura sobre alocação em SPMs, foram identificadas duas técnicas que fazem extração de procedimentos (*function outlining*) a partir de laços (UDAYAKUMARAN; DOMINGUEZ; BARUA, 2006) (EGGER et al., 2010). Isto permite que BBs frequentemente acessados (cujo limiar de lucro é maior do que zero) sejam transformados em procedimentos frequentemente acessados (com limiar de lucro igual a zero), assim concedendo a PRA as vantagens de BBA sobre SPMs pequenas. Como um dos trabalhos futuros, vislumbra-se a avaliação do uso de extração de procedimentos a partir de laços no contexto da técnica NOB desta dissertação.

Para aumentar a eficácia da técnica, outra possibilidade seria incluir o suporte a dados dinâmicos, conforme proposto por Mendonça (2009, 2010) (ainda não implementado). Para que possa ser mantida a característica da técnica de não-uso de hardware dedicado, estes dados deverão ser tratados em nível de código-fonte. Deste modo, teria-se uma técnica de tempo misto: em tempo de compilação (arquivos-fonte) seriam manipulados os dados dinâmicos, e no tempo de pós-compilação (arquivos-objeto relocáveis), continuariam sendo manipulados código e dados estáticos.

Esta técnica mista parece ainda mais promissora quando combinada com a alocação de código sob BBA. Para a alocação de código, BBA geralmente parece não valer a pena frente a PRA. Entretanto, quando dados dinâmicos são incluídos no espaço de otimização, o menor povoamento da SPM (resultante do limiar de alocação de BBA) é muito conveniente, permitindo que mais dados dinâmicos sejam alocados em SPM, o que deve proporcionar uma maior economia de energia.

REFERÊNCIAS BIBLIOGRÁFICAS

ANGIOLINI, F. et al. A Post-Compiler Approach to Scratchpad Mapping of Code. In: *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES).* [S.l.: s.n.], 2004. p. 259–267.

AVISSAR, O.; BARUA, R.; STEWART, D. An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, New York, NY, USA, v. 1, n. 1, p. 6–26, 2002.

BANAKAR, R. et al. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign.* New York, NY, USA: ACM, 2002. p. 73–78.

BINUTILS, G. *The GNU Binutils Website.* 2007. <http://www.gnu.org/software/binutils>.

CASCAVAL, C.; PADUA, D. A. Estimating cache misses and locality using stack distances. In: *Proceedings of the 17th annual international conference on Supercomputing.* New York, NY, USA: ACM, 2003. p. 150–159.

CHEN, G. et al. Dynamic scratch-pad memory management for irregular array access patterns. In: *Proc. of the Conference on Design, Automation and Test in Europe.* [S.l.: s.n.], 2006. p. 931–936.

CHERITON, D. et al. The vmp multiprocessor: initial experience, refinements and performance evaluation. In: *Proceedings of the 15th Annual International Symposium on Computer Architecture.* [S.l.: s.n.], 1988. p. 410–421.

CHO, H. et al. Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU. In: *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES).* [S.l.: s.n.], 2007. p. 195–206.

CONTE, T. M.; HIRSCH, M. A.; HWU, W.-M. W. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, v. 47, n. 6, p. 714–720, 1998. ISSN 0018-9340.

DALLY, W. et al. Efficient Embedded Computing. *IEEE Computer*, v. 41, n. 7, p. 27–32, July 2008. ISSN 0018-9162.

DENG, N. et al. A Novel Adaptive Scratchpad Memory Management Strategy. In: *Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. [S.l.: s.n.], 2009. p. 236–241.

DOMINGUEZ, A.; UDAYAKUMARAN, S.; BARUA, R. Heap Data Allocation for Scratch-Pad Memory in Embedded Memories. *Journal of Embedded Computing*, IOS Press, Amsterdam, The Netherlands, v. 1, n. 4, p. 521–540, 2005.

EGGER, B. *Dynamic Scratchpad Memory Management based on Post-Pass Optimization*. Tese (Doutorado) — Seoul National University, Feb 2008.

EGGER, B. et al. A dynamic code placement technique for scratchpad memory using postpass optimization. In: *Proc. of the International conference on Compilers, architecture and synthesis for embedded systems*. [S.l.: s.n.], 2006. p. 223–233.

EGGER, B.; LEE, J.; SHIN, H. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In: *Proc. of the ACM & IEEE International Conference on Embedded Software*. [S.l.: s.n.], 2006. p. 321–330.

EGGER, B.; LEE, J.; SHIN, H. Dynamic Scratchpad Memory Management for Code in Portable Systems with an MMU. *ACM Transactions Embedded Computer Systems*, ACM, New York, NY, USA, v. 7, n. 2, p. 1–38, 2008.

EGGER, B. et al. Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU. *IEEE Transactions on Computers*, v. 59, n. 8, p. 1047–1062, 2010.

FALK, H.; KLEINSORGE, J. C. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In: *Proc. of the Design Automation Conference*. [S.l.: s.n.], 2009. p. 732–737.

GORDON-ROSS, A. et al. A one-shot configurable-cache tuner for improved energy and performance. In: *Proceedings of the conference on Design, Automation and Test in Europe*. San Jose, CA, USA: EDA Consortium, 2007. p. 755–760.

GUTHAUS, M. et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: *Proc. of the IEEE International Workshop on Workload Characterization*. Washington, DC, USA: IEEE Computer Society, 2001. p. 3–14.

HILL, M. D. et al. Wisconsin architectural research tool set. *SIGARCH Comput. Archit. News*, ACM, New York, NY, USA, v. 21, p. 8–10, September 1993.

HILL, M. D.; SMITH, A. J. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, IEEE Computer Society, Washington, DC, USA, v. 38, n. 12, p. 1612–1630, 1989.

JACOB, B.; NG, S.; WANG, D. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2007.

JANAPSATYA, A.; IGNJATOVIĆ, A.; PARAMESWARAN, S. Finding optimal l1 cache configuration for embedded systems. In: *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 2006. p. 796–801.

JANAPSATYA, A.; IGNJATOVIĆ, A.; PARAMESWARAN, S. A novel instruction scratchpad memory optimization method based on concomitance metric. In: *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*. Piscataway, NJ, USA: IEEE Press, 2006. p. 612–617.

JANAPSATYA, A.; PARAMESWARAN, S.; IGNJATOVIC, A. Hardware/software Managed Scratchpad Memory for Embedded System. In: *Proc. of the IEEE/ACM International conference on Computer-aided design*. [S.l.: s.n.], 2004. p. 370–377.

JENSEN, D. Developing System-on-Chips with Moore, Amdahl, Pareto and Ohm. In: *IEEE International Conference on Electro/Information Technology*. [S.l.: s.n.], 2008. p. 13–18.

JURAN, J. The non-pareto principle; mea culpa. *Quality Progress*, v. 8, n. 5, p. 8–9, 1975.

KANDEMIR, M. et al. Dynamic management of scratch-pad memory space. *Design Automation Conference, 2001. Proceedings*, p. 690–695, 2001. ISSN 0738-100X.

KANNAN, A. et al. A Software Solution for Dynamic Stack Management on Scratch pad Memory. In: *Proc. of the Asia and South Pacific Design Automation Conference*. [S.l.: s.n.], 2009. p. 612–617.

KARP, K. M. Reducibility among Combinatorial Problems. In: *Complexity of Computer Computations*. [S.l.]: Plenum Press, 1972.

MACHANIK, P. *Approaches to addressing the memory wall*. Brisbane, Nov. 2002.

MALIK, A.; MOYER, B.; CERMAK, D. The M-CORE(TM) M340 Unified Cache Architecture. In: *Proc. of the IEEE International Conference on Computer Design*. [S.l.: s.n.], 2000. p. 577.

MARWEDEL, P. *Embedded System Design*. [S.l.]: Springer Verlag, 2006.

MATTSON, R. L.; GECSEI, D. R. S. J.; TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, v. 9, n. 2, p. 78–117, 1970.

MCILROY, R.; DICKMAN, P.; SVENTEK, J. Efficient Dynamic Heap Allocation of Scratch-Pad Memory. In: *Proc. of the 7th ACM International Symposium on Memory Management*. [S.l.: s.n.], 2008. p. 31–40.

MENDONÇA, A. K. I. *Alocação de dados e de código em memórias embarcadas*: uma abordagem pós-compilação. Dissertação (Mestrado em Ciência da Computação) — Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Santa Catarina, Florianópolis, 2010.

MENDONÇA, A. K. I. et al. Mapping data and code into scratchpads from relocatable binaries. In: *Proceedings of the 2009 IEEE Computer Society Annual Symposium on VLSI*. Washington, DC, USA: IEEE Computer Society, 2009. p. 157–162.

MENICHELLI, F.; OLIVIERI, M. Static Minimization of Total Energy Consumption in Memory Subsystem for Scratchpad-Based Systems-on-Chips. *IEEE Transactions on VLSI*, v. 17, n. 2, p. 161–171, February 2009. ISSN 1063-8210.

MING, L.; YU, Z.; LIN, S. An alternative choice of scratch-pad memory for energy optimization in embedded system. In: *IEEE International Conference on Networking, Sensing and Control*. [S.l.: s.n.], 2008. p. 1641–1647.

MUCHNICK, S. S. *Advanced compiler design and implementation.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

PANDA, P. R.; DUTT, N. D.; NICOLAU, A. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 5, n. 3, p. 682–704, 2000.

PAPADIMITRIOU, C. H. On the complexity of integer programming. *J. ACM*, ACM, New York, NY, USA, v. 28, n. 4, p. 765–768, 1981.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface.* 4. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

PISINGER, D. A Minimal Algorithm for the 0-1 Knapsack Problem. *Operations Research*, v. 45, p. 758–767, 1997.

RAVINDRAN, R. A. et al. Compiler managed dynamic instruction placement in a low-power code cache. In: *CGO '05: Proceedings of the international symposium on Code generation and optimization.* Washington, DC, USA: IEEE Computer Society, 2005. p. 179–190.

RedHat Inc. *Newlib.* 2010. `http://sources.redhat.com/newlib/`.

RIGO, S. et al. ArchC: A SystemC-Based Architecture Description Language. In: *Proc. of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* [S.l.: s.n.], 2004. p. 66–73.

ROTENBERG, E.; BENNETT, S.; SMITH, J. E. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, IEEE Computer Society, Washington, DC, USA, v. 48, n. 2, p. 111–120, Feb 1999.

SCOTT, J. et al. Designing the low-power m*core architecture. In: *Proc. IEEE Power Driven Microarchitecture Workshop.* [S.l.: s.n.], 1998. p. 145–150.

SEGARS, S. *Tutorial 4: Low-Power Design Techniques for Microprocessors.* Feb. 2001. International Solid-State Circuits Conference.

STEINKE, S. et al. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In: *Proc. of the International*

Symposium on System Synthesis. New York, NY, USA: ACM, 2002. p. 213–218.

STEINKE, S. et al. Assigning program and data objects to scratchpad for energy reduction. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2002. p. 409.

SUGUMAR, R. A.; ABRAHAM, S. G. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, ACM, New York, NY, USA, v. 13, n. 1, p. 32–56, 1995.

TALLA, D.; GOLSTON, J. Using DaVinci Technology for Digital Video Devices. *Computer*, v. 40, p. 53–61, 2007.

THOZIYOOR, S. et al. *CACTI 5.1*. [S.l.], abr. 2008.

TOMIYAMA, H.; YASUURA, H. Optimal code placement of embedded software for instruction caches. *European Design and Test Conference*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 96, 1996.

UDAYAKUMARAN, S.; DOMINGUEZ, A.; BARUA, R. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, New York, NY, USA, v. 5, n. 2, p. 472–511, 2006.

UHLIG, R. A.; MUDGE, T. N. Trace-Driven Memory Simulation: a Survey. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 29, n. 2, p. 128–170, 1997.

VERMA, M.; MARWEDEL, P. *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2007.

VERMA, M.; WEHMEYER, L.; MARWEDEL, P. Cache-aware scratchpad allocation algorithm. In: *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004. p. 21264.

VERMA, M.; WEHMEYER, L.; MARWEDEL, P. Dynamic overlay of scratchpad memory for energy minimization. In: *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2004. p. 104–109.

VIANA, P. *A Methodology to Explore Memory Hierarchy Architectures for Embedded Systems*. Tese (Doutorado) — Universidade Federal de Pernambuco, September 2006.

VIANA, P. et al. A table-based method for single-pass cache optimization. In: *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. [S.l.]: ACM, 2008. p. 71–76.

VIANA, P. et al. Configurable cache subsetting for fast cache tuning. In: *Proceedings of the Design Automation Conference*. [S.l.: s.n.], 2006. p. 695–700.

VOLPATO, D. P. et al. Cache-tuning-aware allocation from binaries: a fresh perspective on scratchpad usage. 24th Symposium on Integrated Circuits and Systems Design (SBCCI'11), submetido. 2011.

VOLPATO, D. P. et al. A post-compiling approach that exploits code granularity in scratchpads to improve energy efficiency. In: *Proceedings of the 2010 IEEE Computer Society Annual Symposium on VLSI*. Washington, DC, USA: IEEE Computer Society, 2010. p. 127–132.

WEHMEYER, L.; MARWEDEL, P. *Fast, efficient and predictable memory accesses: optimization algorithms for memory architecture aware compilation*. [S.l.]: Springer, 2006.

ZHANG, C.; VAHID, F. Cache Configuration Exploration on Prototyping Platforms. In: *Proc. of the IEEE International Workshop on Rapid Systems Prototyping*. [S.l.: s.n.], 2003. p. 164–170. ISSN 1074-6005.

ZHANG, C.; VAHID, F.; LYSECKY, R. A self-tuning cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems*, ACM, New York, NY, USA, v. 3, p. 407–425, May 2004. <http://doi.acm.org/10.1145/993396.993405>.

ZHANG, C.; VAHID, F.; NAJJAR, W. A highly configurable cache for low energy embedded systems. *ACM Transactions on Embedded Computer Systems*, ACM, New York, NY, USA, v. 4, p. 363–387, May 2005.

APÊNDICE A – O método SPCE

O método SPCE realiza o ajuste-fino, a partir dos endereços do programa, para um conjunto de caches e em uma única passada. As entradas do método são um *trace* T, um conjunto de parâmetros que delimitam o espaço de projeto de caches, o deslocamento (*offset*) de palavra w da arquitetura do processador, e algumas estruturas de dados.

O **trace** T contém a sequência de endereços acessados no subsistema de memória para um programa qualquer, conforme a Definição 3.1. O tipo de endereço (instruções, dados ou ambos) contido no trace determinará qual a cache sendo ajustada (cache de instruções, de dados ou unificada).

O **espaço de projeto (*design space*)** de caches é delimitado pelos parâmetros $s_{min}, s_{max}, b_{min}, b_{max}, \bar{a}_{max}$, que representam, respectivamente, o número mínimo e máximo de conjuntos que uma cache pode possuir, o tamanho mínimo e máximo de um bloco de cache (em bytes) e o maior grau de associatividade permitido. O menor grau de associatividade considerado pelo método é sempre $\bar{a}_{min} = 1$, o que configura uma cache com mapeamento direto.

Além destas entradas, o método utiliza duas estruturas de dados: uma estrutura de matriz tridimensional, denominada Tabela de Conflitos, e uma pilha de endereços, apresentados pelas definições que seguem.

Definição A.1. Pilha de endereços. Uma pilha de endereços P é uma tupla (p_1, p_2, ..., p_i, ... p_n) que armazena uma sequência de endereços de bloco processados (derivados dos endereços de T) durante a execução do método SPCE, onde p_i denota o i-ésimo endereço de bloco armazenado num dado momento. Seu topo é indicado por p_n, e sua base por p_1. As características do método SPCE são tais que cada endereço armazenado é único.

Uma pilha P é uma extensão da pilha LIFO (*last in, first out*) convencional. A operação de inserção é realizada da maneira tradicional, i.e. um elemento novo é empilhado (no topo da pilha). Todavia, a operação de remoção permite que um elemento seja retirado de qualquer posição da pilha, ao invés de somente do topo.

Definição A.2. Configuração de cache. Uma configuração de cache, denotada por (a_i, s_i, b_i), representa uma cache com grau de associatividade a_i, s_i conjuntos e tamanho de bloco b_i (em bytes), tal que sua capacidade é dada por $C = s_i \times b_i \times a_i$, expressa em bytes.

Definição A.3. Tabela de Conflitos. Uma Tabela de Conflitos K é uma matriz tridimensional $K_{\bar{a}_{max} \times s_{max} \times b_{max}}$, onde $\bar{a}_{max}$ matrizes bidimensionais são formadas de s_{max} linhas e b_{max} colunas. Cada célula da tabela, denotada por $K_{\bar{a}_i, s_i, b_i}$, está relacionada a uma configuração

(a_i, s_i, b_i), de modo a proporcionar o cômputo do número de acertos desta configuração.

O funcionamento do método SPCE consiste em descobrir, para cada configuração de cache (a_i, b_i, s_i) do espaço de projeto, quantos acertos ocorreram para a sequência de endereços acessados informada por T. Isto consiste, em última instância, em determinar se cada acesso a um dado endereço α_i induz um acerto ou uma falta na cache.

Para tanto, quando processa α_i, o método procura calcular o número de conflitos (κ) no conjunto da cache para o qual α_i está mapeado, ocorridos desde o último acesso a este mesmo endereço, digamos α_h, onde $\alpha_h = \alpha_i \mid h \in \mathbb{N}, 1 < h < i$.

Obtido κ, calcula-se o menor grau de associatividade da cache necessário para que o acesso ao endereço α_i resulte em acerto, denotado por $\bar{a}'$. Se, desde o último acesso à α_i, não houve nenhum conflito em sua entrada na cache ($\kappa = 0$), então um acerto ocorrerá para uma cache com mapeamento direto ou qualquer grau de associatividade ($\bar{a}' = 1 \therefore \bar{a}_i \geq 1$). Se, no entanto, houve um conflito ($\kappa = 1$), isto significa que α_i não estará mais presente se a cache em questão operar sob mapeamento direto. Contudo, caso a cache seja associativa de pelo menos duas vias ($\bar{a}' = 2 \therefore \bar{a}_i \geq 2$), o endereço que conflitaria com α_i pode ser acomodado juntamente com ele no mesmo conjunto da cache, de modo que o acesso resultaria em um acerto. De forma análoga, para dois conflitos ($\kappa = 1$), uma cache associativa de quatro ou mais vias ($\bar{a}' = 4 \therefore \bar{a}_i \geq 4$) seria necessária para garantir um acerto. Em outras palavras, uma cache de grau de associatividade $\bar{a}_i$ consegue suportar até $\kappa - 1$ conflitos por conjunto sem que haja uma falta.

O cálculo de κ e $\bar{a}'$ é feito para todas as configurações de caches formadas a partir de variações no tamanho de bloco b e no número de conjuntos s. Após a determinação da associatividade $\bar{a}'$ para uma configuração com parâmetros b_i e s_i, sabe-se que toda cache $(a_i, b_i, s_i) \mid \bar{a}_i \geq \bar{a}'$ resultará em acerto.

Finalmente, calcula-se o número de faltas como sendo o complemento do número de acertos com relação ao total de endereços acessados, e, a partir do número de faltas, pode ser estimado o consumo de energia de cada configuração (a_i, b_i, s_i).

A.1 PROCESSAMENTO DOS ENDEREÇOS DO TRACE T

O Algoritmo 1 apresenta o procedimento principal do método SPCE. O método funciona processando cada endereço α_i de T (linha 1)

Algoritmo 1 SPCE

Entrada(s): T, s_{min}, s_{max}, b_{min}, b_{max}, $\bar{a}_{max}$, w, P, K

1: **para todo** $\alpha_i \in T$ **faça**
2:    $end \Leftarrow$ shift_right(α_i, w)
3:    **para** $b_i = b_{max}$ até b_{min} **faça**
4:       $end_{bloco} \Leftarrow$ shift_right(end, $\log_2(b_i)$)
5:       **se** $end_{bloco} \in P$ **então**
6:          **para** $s_i = s_{min}$ até s_{max}, onde $s_i \in \{n^2 \mid n \in \mathbb{N}, s_{min} \leq n \leq s_{max}\}$ **faça**
7:             $\kappa \Leftarrow$ CONTA_CONFLITOS(P, s_i, end_{bloco})
8:             **se** $\kappa \leq \bar{a}_{max}$ **então**
9:                $\bar{a}' \Leftarrow$ múltiplo de 2 que sucede κ
10:                $K_{\bar{a}',s_i,b_i} \Leftarrow K_{\bar{a}',s_i,b_i} + 1$
11:             **fim se**
12:          **fim para**
13:          Mova end_{bloco} para o topo de P
14:       **senão** { $end_{bloco} \notin P$ }
15:          Empilhe end_{bloco} em P
16:       **fim se**
17:    **fim para**
18: **fim para**

Algoritmo 2 CONTA_CONFLITOS

Entrada(s): P, s_i, end_{bloco}

Saída(s): κ

1: $\kappa \Leftarrow 0$
2: $c \Leftarrow end_{bloco} \bmod s_i$
3: **para** $p_i = p_n$ até p_1 **faça**
4:    **se** $p_i = end_{bloco}$ **então**
5:       **retorne** κ
6:    **fim se**
7:    $c' \Leftarrow p_i \bmod s_i$
8:    **se** $c = c'$ **então**
9:       $\kappa \Leftarrow \kappa + 1$
10:    **fim se**
11: **fim para**
12: **retorne** κ

da seguinte maneira. Primeiramente, elimina-se o deslocamento (*offset*) de palavra w do endereço α_i: α_i deslocado bit-a-bit w vezes para a direita é armazenado em *end* (linha 2). Então, para cada tamanho de bloco b_i (linha 3), é realizado o o seguinte processamento.

Elimina-se o deslocamento de bloco de cache do endereço (linha 4), dando origem ao endereço de bloco (end_{bloco}).

Se o endereço de bloco não está na pilha P de endereços já processados, ele é simplesmente empilhado em P (linhas 14 e 15) e parte-se para o próximo endereço (α_{i+1}).

Entretanto, se o endereço de bloco encontra-se na pilha P (linha 5), então, para cada tamanho de conjunto s_i, é calculada a quantidade de conflitos (κ) ocorridos desde o último acesso ao endereço α_i, na respectiva entrada de α_i na cache (linhas 6 e 7). O Algoritmo 2 apresenta o cálculo do número de conflitos de maneira detalhada.

Para obter o menor grau de associatividade que garante um acerto na cache ($\bar{a}'$), κ é arredondado para a próxima potência de dois, e a célula correspondente na Tabela de Conflitos é incrementada de um (linhas 9 e 10).

Contudo, pode ser que o maior grau de associatividade ($\bar{a}_{max}$) considerado no espaço de projeto não seja grande o suficiente para acomodar os κ conflitos e garantir que o acesso à α_i resulte em acerto (o que é verificado na linha 8). Neste caso, nenhuma célula da Tabela de Conflitos é incrementada.

Finalmente, α_i é movido de sua atual posição em P para o topo (linha 13), e parte-se para o próximo endereço, α_{i+1}.

A.2 CONTABILIZAÇÃO DO NÚMERO DE CONFLITOS EM UM CONJUNTO

O Algoritmo 2 detalha o procedimento de contagem do número de conflitos em um conjunto, para um endereço de bloco end_{bloco} (derivado de α_i), ocorridos deste o último acesso a end_{bloco}. Além do endereço de bloco, este procedimento recebe como entradas a pilha P e o número de conjuntos da cache (s_i). A saída do algoritmo é o número de conflitos ocorridos, denotado por κ.

Inicialmente, o valor de retorno é inicializado. Determina-se o conjunto da cache para o qual α_i está mapeado, denotado por c, onde $c \in \mathbb{N} \mid 1 \leq c \leq s_i$ (linha 2), e onde a mod b representa o resto da divisão inteira de a por b.

Então, cada endereço p_i contido na pilha P é processado (linha 3),

partindo do topo (p_n) para a base (p_1), como segue:

1. Caso p_i e end_{bloco} (e, portanto, α_i) correspondam a um mesmo bloco de cache, este algoritmo chegou ao fim, e o número de conflitos é retornado (linhas 4 e 5).

2. Caso contrário, determina-se o conjunto c' da cache para o qual o endereço p_i está mapeado (linha 7).

3. Então, caso este conjunto seja o mesmo para p_i e α_i, um conflito é contabilizado (linhas 8 e 9).

Finalmente, o número de conflitos é retornado pelo algoritmo (linha 12).

A.3 CÁLCULO DO NÚMERO DE ACERTOS E ESTIMATIVA DE ENERGIA

Após a execução do Algoritmo 1 para cada endereço α_i de T, a quantidade de acertos e de faltas de cada configuração pode ser calculada a partir da Tabela de Conflitos K e do número de endereços processados (dado pela cardinalidade de T), conforme mostram as Equações A.1 e A.2:

$$acertos_{\bar{a}_i,s_i,b_i} = \sum_{j=\bar{a}_{min}=1}^{\bar{a}_i} K_{j,s_i,b_i} \tag{A.1}$$

$$faltas_{\bar{a}_i,s_i,b_i} = |T| - acertos_{\bar{a}_i,s_i,b_i} \tag{A.2}$$

A estimativa de energia é feita a partir do número de acertos e faltas, fazendo-se uso de um modelo físico de memórias para estimar o consumo de energia por acesso para os diversos componentes do subsistema de memória. Neste trabalho, utilizamos uma adaptação do esquema apresentado por Zhang, Vahid e Lysecky (2004), que consistiu na totalização do consumo de energia decorrente dos acertos ($acertos_{\bar{a}_i,s_i,b_i} \times E_{cache}$) com o consumo decorrente das faltas ($faltas_{\bar{a}_i,s_i,b_i} \times (E_{cache} + E_{MP})$). Como modelo físico de memórias, foi utilizado o CACTI (THOZIYOOR et al., 2008).

APÊNDICE B – Correlação entre economia de energia total e de sistema

A correlação entre a economia de energia total (E_{Total}) e a energia do subsistema de memória (E_{Mem}) pode ser capturada por um **fator de proporcionalidade** k que, conforme a Equação 6.3, é dado por:

$$k = \frac{E_{Mem}}{E_{Total}}$$

A Tabela 11 apresenta os valores de k calculados, neste trabalho, a partir dos resultados apresentados por trabalhos anteriores da literatura de SPM. Para cada trabalho correlato (primeira coluna), são apresentados o processador (segunda coluna), a localização da memória principal quanto ao circuito integrado do processador (terceira coluna), e, finalmente, o valor de k (última coluna). Frente aos valores apresentados, cabe relembrar que este trabalho utiliza um processador da arquitetura MIPS e uma memória principal *off-chip*.

Tabela 11: Correlação entre economia de energia total e de sistema

Referência	**Processador**	**Memória Principal**	**k**
Angiolini et al. (2004)	ARMv7	*on-chip*	0,980
Cho et al. (2007)	ARM9E-S	*off-chip*	0,997
Egger (2008)	ARM926EJ-S	*off-chip*	0,966
Egger et al. (2010)	ARM1136JF-S	*off-chip*	1,010
Média			**0,988**