UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Alexandre Keunecke Ignácio de Mendonça

Alocação de Dados e de Código em Memórias Embarcadas: Uma Abordagem Pós-Compilação

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Luiz Cláudio Villar dos Santos, Dr. (Orientador)

Prof. José Luís Almada Güntzel, Dr. (Co-Orientador)

Florianópolis
2010

Alocação de Dados e de Código em Memórias Embarcadas: Uma Abordagem Pós-Compilação

Alexandre Keunecke Ignácio de Mendonça

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Florianópolis, Fevereiro de 2010

Prof. Mário Antônio Ribeiro Dantas, Dr. (Coordenador)

Banca Examinadora

Prof. Luiz Cláudio Villar dos Santos, Dr. (Orientador)

Prof. Luigi Carro, Dr.

Prof. Rodolfo Jardim de Azevedo, Dr.

Prof. José Luís Almada Güntzel, Dr.

O degrau de uma escada não serve simplesmente para que alguém permaneça em cima dele, destina-se a sustentar o pé de um homem pelo tempo suficiente para que ele coloque o outro um pouco mais alto.
-Thomas Huxley

À minha família pelo incondicional apoio.

Agradecimentos

Ao Professor Dr. Luiz Cláudio Vilar dos Santos pela excelente orientação, incentivo e oportunidades sem as quais este trabalho não seria possível.

Ao Professor Dr. José Luís Güntzel pela co-orientação e incentivo, fundamentais a esse trabalho.

Ao CNPq, no âmbito do Programa Nacional de Microeletrônica, pelo custeio parcial da execução deste trabalho (Processo número: 130071/2008-0).

Aos colegas do LAPS e NIME por todo o auxílio prestado. Em particular, aos colegas Daniel Pereira Volpato, Rafael Westphal e Leandro Freitas pelo suporte técnico.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de algoritmos

Lista de Acrônimos

ADL	*Architecture Description Language* (linguagem de descrição de arquiteturas)
ASIP	*Application-Specific Instruction Set Processor* (processador dedicado)
CI	Circuito Integrado
CPU	*Central Processing Unit* (processador)
DRAM	*Dynamic Random Access Memory* (memória dinâmica de acesso aleatório)
DSP	*Digital Signal Processor* (processador de sinais digitais)
DTSE	*Data Transfer and Storage Exploration*
DPRG	*Data-Program Relationship Graph* (grafo de relação dados-programa)
ECU	*Eletronic Control Unit* (unidade eletrônica de controle)
ESL	*Eletronic System Level*
ELF	*Executable and Linkable Format*
IP	*Intellectual Property*
ITRS	*International Technology Roadmap for Semiconductors*
MMU	*Memory Management Unit* (unidade de gerenciamento de memória)
MP	Memória Principal
MPEG	*Moving Picture Experts Group*
MPSoC	*Multiprocessor System-on-Chip*
RAM	*Random Access Memory* (Memória de acesso aleatório)
RTOS	*Real-Time Operating System*
RTL	*Register-Transfer Level*
SoC	*System-on-Chip* (sistema integrado)
SRAM	*Static Random Access Memory*
SPM	*Scratchpad Memory* (memória de rascunho)
KB	Kilo bytes

Lista de Símbolos

M: Uma memória genérica.
λ_M: Latência da memória M (expressa em ciclos de relógio).
C_M: Capacidade de memória M (expressa em bytes).
E_M: Energia consumida em um único acesso à memória M.
D_i: Elemento de programa
(variável, estrutura de dados ou segmento código).
m_i: Taxa de faltas ou *miss rate* do elemento de programa D_i.
a_i: Número de acessos a um elemento D_i.
w_i: Tamanho do elemento de programa D_i.
x_i: Denota se um elemento de programa D_i está ou não selecionado para alocação na SPM.
p_i: Ganho da alocação do elemento D_i na SPM.
e_i: Eficiência da alocação do elemento D_i na SPM (p_i/w_i).
X: Matriz de alocação de dados.
P: Matriz de lucro dos elementos.
W: Matriz de tamanho dos elementos.
R_n: Número de referências a um elemento D_n.
α_i: i-ésimo endereço de acesso à memória.
T: Padrão de acessos à memória (*trace*).
y_b: Item de corte.
s_i: Estado representando uma solução candidata.
π_i: Soma dos lucros dos itens alocados em uma solução associada ao estado s_i.
μ_i: Soma dos pesos dos itens alocados em uma solução associada ao estado s_i.
S^n: Espaço de soluções candidatas para o Problema da Mochila.
$[s, t]$: Posições-limite que definem o núcleo de pesquisa.
$S^{[s,t]}$: Subespaço de soluções candidatas induzido pelo núcleo de pesquisa.
$\overline{X}^{[s,t]}$: Vetor parcial de alocação associado ao núcleo de pesquisa.
S: Espaço de pesquisa explorado.
$z(S)$: Limite inferior de lucro do espaço de pesquisa S.

Resumo

Memórias do tipo *scratchpad* (SPMs) são alternativas promissoras para sistemas embarcados energeticamente eficientes. Muitas das técnicas de otimização para o mapeamento de dados e código para SPMs assumem a disponibilidade do código-fonte da aplicação. Porém, o desenvolvimento de software embarcado deve lidar com código legado, bibliotecas de terceiros e blocos de propriedade intelectual (IPs) para os quais podem estar disponíveis somente os arquivos-binários. As poucas técnicas que realizam otimizações diretamente em arquivos binários operam em arquivos executáveis e limitam-se a tratar somente código ou somente dados.

Este trabalho propõe uma nova técnica que pode alocar tanto código quanto dados para a SPM. Operando diretamente em binários, a técnica permite que elementos encapsulados em bibliotecas façam parte do mapeamento para SPMs. A técnica consiste de três principais mecanismos: o *profiler*, o mapeador e o *patcher*. O *patcher* foi projetado para operar utilizando arquivos-objeto relocáveis, contornando assim a limitação de se gerenciar relocações para SPM em arquivos-objeto executáveis. A maior eficiência ao se tratar arquivos binários relocáveis resultou em tempos de execução inferiores em pelo menos uma ordem de magnitude se comparados às técnicas relacionadas. O tempo médio de *patching* foi de 0,7s em uma estação de trabalho com quatro núcleos de processamento.

Comparando-se com o mapeamento de somente código, a proposta de também alocar dados em SPM resultou em economia extra de energia de 21%, em média, para um variado conjunto de programas do *benchmark* MiBench e diversas configurações de memória. Apesar de uma média relativamente baixa, instâncias de caso de uso reais com maior conteúdo de dados estáticos puderam ser encontradas, para as quais economias extra de energia de 67% e 91% foram observadas, quando dados também são passíveis de serem mapeados para SPM. Foi também observado que, para todos os casos de uso reais usados nos experimentos, os parâmetros de caracterização para mapeamento em SPM são bastante descorrelatados, o que significa - à luz de trabalhos anteriores - que o mapeamento exato obtido pelo assim-chamado algoritmo MINKNAP provavelmente manterá sua alta eficiência, mesmo diante do crescimento do número de elementos de programas resultante da crescente complexidade do software embarcado.

Ao se combinar a inclusão de dados e bibliotecas na alocação em SPM com a eficiência do mapeamento e a eficiência do *patching*, a técnica proposta torna-se um enfoque promissor para a otimização energeticamente consciente de código embarcado pré-compilado.

Abstract

Scratchpad memories (SPMs) are promising for energy-efficient embedded systems. Most optimizing techniques for mapping data and code elements to SPMs assume the availability of source code. However, embedded software development has to cope with legacy code, third-party software, and IP-protected applications for which only the binaries are available. The few techniques that directly handle binaries operate on executable files and are limited to either code or data.

This work proposes a new technique that addresses both data and code allocation into SPMs. Since it operates directly on binaries, the technique allows library elements to be eligible for SPM mapping. It consists of three main engines: a profiler, a mapper and a patcher. The patcher was designed to operate upon relocatable object binaries so as to overcome the inefficiency of bookkeeping SPM relocations on executable binaries. The higher efficiency of handling relocatable binaries resulted in runtimes one order of magnitude lower than those observed in related approaches. The average patching time was 0.7s on a quad-core workstation.

As compared to code-only SPM mapping, the proposed data-inclusive SPM allocation resulted in extra average energy savings of 21% for a varied set of programs from the benchmark MiBench and several memory configurations. Despite the relatively low overall average, instances of real-life use cases with higher static data content were found for which extra savings of 67% and 91% were observed, when SPM allocation becomes data-inclusive. It was also observed that, for all the real-life use cases employed in the experiments, the characterization parameters for SPM mapping are largely uncorrelated, which means - at the light of previous work - that the exact mapping obtained with the so-called MINKNAP algorithm is likely to preserve its high efficiency, even when the number of program elements scales up as a result of the growing complexity of embedded code.

The combination of data-inclusive and library-inclusive SPM allocation with efficient mapping and efficient patching, makes the proposed technique a promising approach for the energy-aware optimization of precompiled embedded code.

Capítulo 1

Introdução

Este capítulo apresenta, inicialmente, os limites tecnológicos impostos aos sistemas computacionais de propósitos gerais, os quais têm impacto tanto no desempenho quanto no consumo de energia. A seguir, é apresentado um panorama dos sistemas embarcados, dando ênfase às características que são peculiares às principais áreas de aplicação. A noção de sistema integrado (*System-on-a-chip* - SoC) é então apresentada, justificando-se seu uso no projeto de sistemas embarcados. Após, os componentes de um subsistema de memória são apresentados e suas características são ressaltadas. Então, é apresentado um exemplo ilustrativo para motivar o trabalho de pesquisa descrito nesta dissertação. Finalmente, define-se o escopo desta dissertação, resumem-se as principais contribuições e descreve-se a organização do texto.

1.1 Limites tecnológicos impostos aos sistemas computacionais

Segundo Hennessy e Patterson [HEN 06], o desempenho dos processadores aumentou a uma taxa de 25% ao ano até 1986, tendo passado para 52% ao ano entre 1986 e 2002. A partir de 2002 o desempenho dos processadores vem aumentando a uma taxa de 20% ao ano. Por outro lado, também segundo Hennessy e Patterson, o desempenho das memórias têm aumentado 10% ao ano. A Figura 1.1 (extraída de [HEN 06]) ilustra a diferença entre o desempenho dos processadores e o desempenho das memórias. Esta diferença de desempenho tem como consequência o fato de que a taxa na qual os dados podem ser supridos a um sistema computacional tende a ser inferior à taxa na qual o sistema pode processá-los. Tal limitação tecnológica tem sido referenciada por *The Memory Wall Problem* [WUL 95]. Assim, para maximizar o desempenho dos sistemas computacionais é necessário projetar cuidadosamente o subsistema de memória, de modo a atenuar, na medida do possível, o impacto do *Memory Wall Problem*.

A Figura 1.2 ilustra o incremento na frequência de operação e na potência dissipada para oito gerações de processadores da família x86 da Intel. Tanto a frequência de operação dos processadores quanto a potência dissipada aumentaram rapidamente até o ano de 2004, quando então pas-

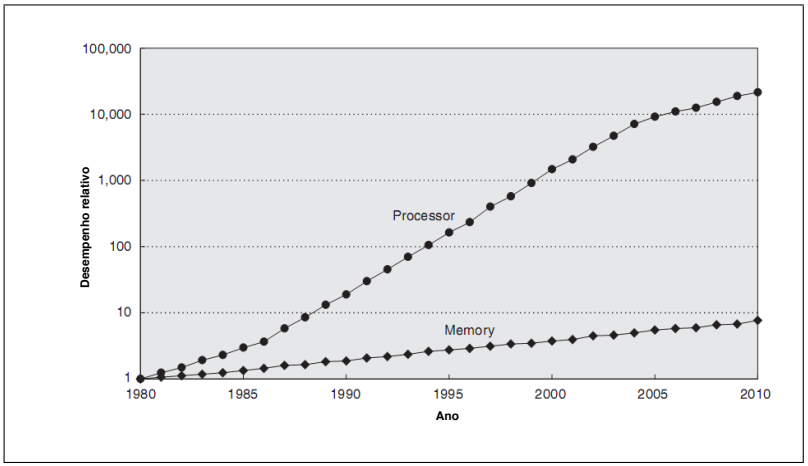

Figura 1.1: Diferença de desempenho entre processador e memória (latência) ao longo do tempo (Fonte:[HEN 06])

saram a cair. O comportamento semelhante destas duas medidas decorre do fato de que a frequência de operação e a potência dissipada estão relacionadas. Em 2004 ambas pararam de crescer porque o limite prático na refrigeração da potência dissipada foi atingido. Esse limite é denominado *The Power Wall* [PAT 08].

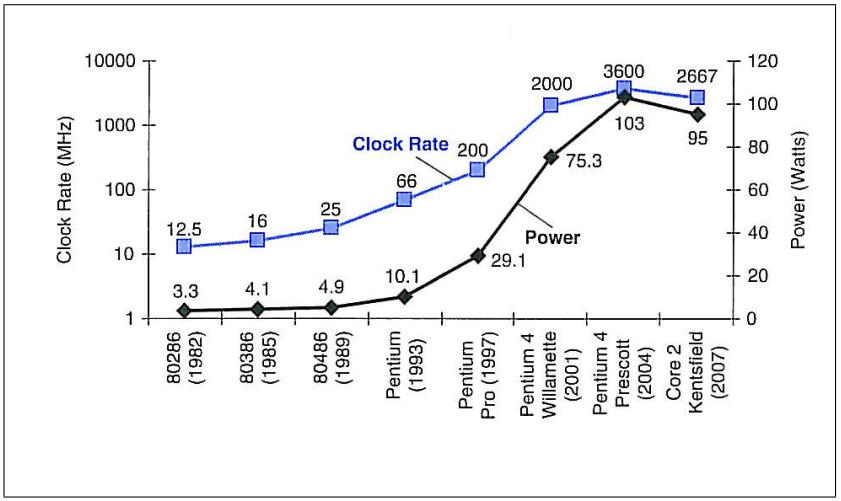

Figura 1.2: Frequência e potência para oito gerações dos processadores da família x86 (Fonte:[PAT 08])

1.2 Um panorama de sistemas embarcados

Um sistema embarcado é um conjunto de componentes de hardware e de software que cooperam para executar uma função específica [BER 01]. Às vezes, sistemas embarcados realizam operações muito simples, necessitando apenas de um microcontrolador. Porém, muitos sistemas embarcados demandam uma grande capacidade de processamento. Por exemplo, estima-se que os telefones celulares da próxima geração necessitem de um processador que seja capaz de executar 10 bilhões de instruções por segundo [WOL 07]. Além disto, muitos sistemas embarcados têm outras restrições como baixo consumo e/ou garantias de tempo real. Por exemplo, um telefone celular deve operar por horas em baterias de carga limitada e, ao mesmo tempo, cumprir com as restrições de tempo real inerentes à sua funcionalidade.

Os sistemas embarcados atuais podem ser classificados conforme sua área de aplicação [WOL 07]. Cada classe apresenta um conjunto de características que lhe são peculiares:

- Controle Automotivo: um automóvel contemporâneo possui entre uma dezena e uma centena de unidades eletrônicas de controle ou *Electronic Control Units* (ECUs) [COO 07]. Segundo Sangiovanni-Vincentelli [SV 07], sistemas eletrônicos automotivos são subdivididos em duas categorias: controle de partes mecânicas com restrição de tempo real severa (controle de chassis, controle de segurança, controle de freios, etc.) e informação/entretenimento (sistemas de informação, navegação, comunicação externa, etc.).

- Vídeo digital: os requisitos de desempenho para equipamentos de vídeo digital são comparáveis a um sistema de propósito geral, porém, com requisitos de consumo de energia de duas ordens de magnitude inferior e de custo inferior em mais de uma ordem de magnitude. Devido a isto, seu projeto é um grande desafio.

- Telefonia móvel: as primeiras gerações de telefones móveis utilizavam transmissões analógicas com microcontroladores simples para realizar tarefas básicas. Porém, telefones móveis lançados no mercado recentemente baseiam-se em microprocessadores para tarefas fundamentais, como compressão de voz e processamento de sinal de banda base (*baseband*), sem mencionar os recursos adicionais para realizar funções relacionadas à agenda telefônica e multimídia (áudio e vídeo), que fazem parte da interface com o usuário [RAM 07] [WOL 07]. É esperado que as novas soluções de processadores utilizados em telefonia móvel tenham um desempenho similar a de um processador atual, porém, com um baixo consumo de energia e baixa potência (menos de 0,5 W de potência dissipada) [RAM 07].

1.3 Sistemas integrados (SoCs)

Para atender aos requisitos de alto desempenho, de baixo consumo de energia e de tempo para mercado (*time-to-market*) e ainda propiciar um produto competitivo em termos de custo, os projetistas de sistemas embarcados precisam fazer uso de sistemas integrados em uma única pastilha de silício, os chamados SoCs (*Systems-on-chip*). Um SoC é composto por um ou mais processadores, subsistema de memória, rede de interconexão, um conjunto adequado (que pode ser extenso) de interfaces e eventualmente, um ou mais blocos aceleradores para executar partes críticas do processamento. O SoC pode ser projetado especificamente para o produto-alvo ou pode ter sido projetado para uma classe de aplicações, sendo fornecido por terceiros. No primeiro caso, os blocos aceleradores a serem utilizados, quando for o caso, podem pertencer a uma biblioteca própria, projetada pela mesma empresa que desenvolve o SoC, ou podem ser blocos de propriedade intelectual adquiridos de terceiros. Blocos aceleradores adquiridos de terceiros são comumente referenciados por núcleos de propriedade intelectual (*Intellectual Property Cores*), ou simplesmente "IPs".

A Figura 1.3 mostra o diagrama de blocos do *chip* DM6441, da Texas Instruments [TAL 07]. Trata-se de um SoC para aplicações multimídia, o qual contém um subsistema completo para aplicações de vídeo digital com dois processadores (um DSP e outro de uso geral), controladores de memória, memórias internas, interfaces de rede, interfaces para áudio e vídeo, portas de comunicação, etc. No caso de um SoC voltado para uma classe de aplicação, como o DM6441, a configuração da aplicação se dá por meio do software a ser embarcado.

Devido ao crescimento da complexidade do software embarcado, a quantidade de memória embarcada em um SoC vem aumentando significativamente, a ponto da área do *chip* ser dominada pela memória. Atualmente, o subsistema de memória já é o maior gargalo em termos de desempenho e de consumo de energia em um SoC [VER 07].

Segundo Dally [DAL 08], os blocos de memória respondem por até 70% do consumo de energia de um processador embarcado contemporâneo (ano 2008), sendo 42% gasto no suprimento das instruções e 28% no suprimento dos dados. Por outro lado, o consumo da unidade aritmética corresponde a 7%, conforme ilustra a Figura 1.4. Desta forma, para atingir-se o desempenho necessário com baixo consumo de energia é mandatório explorar-se o projeto do subsistema de memória.

Portanto, o desenvolvimento de técnicas para otimizar o uso de memória embarcada (presente no mesmo *chip* do processador) é um tópico relevante de investigação científica.

1.4 Componentes de um subsistema de memória

O subsistema de memória associado a um SoC é composto por um ou mais dos seguintes componentes:

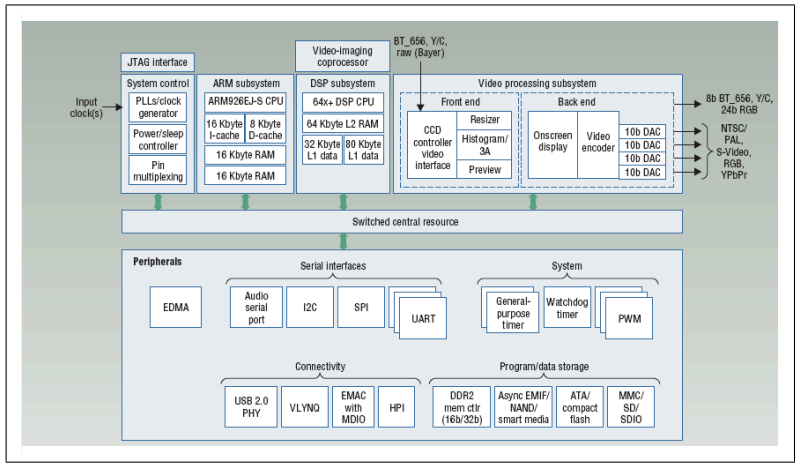

Figura 1.3: Diagrama de blocos de um SoC (Fonte:[TAL 07])

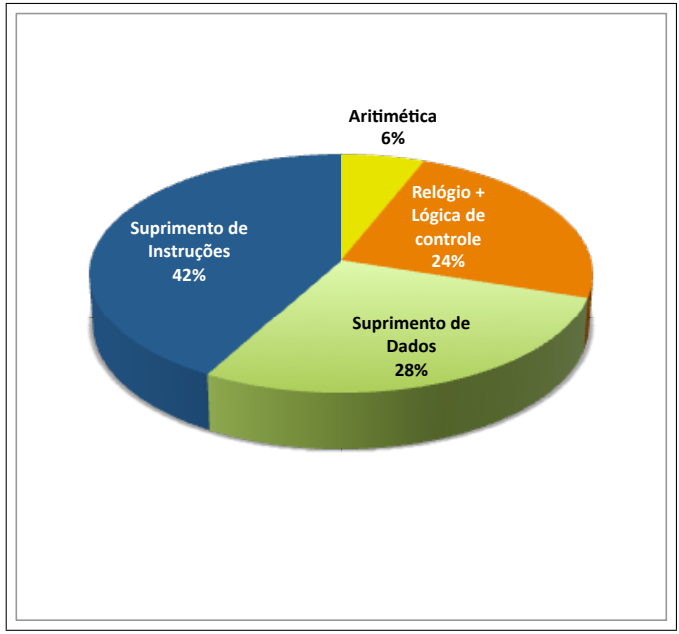

Figura 1.4: Distribuição do consumo de energia entre os componentes de um processador embarcado (Fonte:[DAL 08])

- **Memória principal (MP):** memória de acesso aleatório externa ao SoC (*off-chip*), predominantemente implementada com bancos de memória dinâmica (DRAM) de acesso aleatório.

- **Memória auxiliar (cache):** memória associativa interna ao SoC (*on-chip*), implementada com tecnologia SRAM, que contém cópias dos dados ou instruções residentes na memória principal. Seu espaço de endereçamento coincide, portanto, com parte do espaço de endereçamento da memória principal.

- **Memória de rascunho (SPM):** memória SRAM interna ao SoC (*on-chip*), conhecida como *scratchpad memory*, cujo espaço de endereçamento é disjunto do espaço de endereçamento em memória principal. Diferentemente das memórias cache, onde a alocação dos elementos de programa é controlada pelo próprio hardware, a alocação de elementos de programa à SPM (que neste caso podem ser dados e/ou instruções) deve ser realizada pelo projetista do software embarcado. Este mapeamento pode permanecer inalterado (alocação estática, também chamada de *non-overlaying*) ou pode ser alterado em tempo de execução da aplicação (alocação dinâmica, também chamada de *overlaying*) [VER 07].

A hierarquia de um subsistema de memória pode conter múltiplos níveis de memória cache, além de SPM e MP, conforme ilustra a Figura 1.5.

A estrutura e o comportamento de memórias cache são abordados em textos clássicos [HEN 06] [PAT 08]. SPMs são normalmente tratadas em textos especializados em sistemas embarcados [BAN 02] e [MAR 06].

A grande diferença entre uma SPM e uma cache é que a primeira garante tempo constante no acesso, enquanto o tempo de acesso à cache depende de ocorrer ou não as seguintes faltas:

- **Compulsória:** o primeiro acesso a um bloco sempre resultará em falta, sendo obrigatória a sua transferência a partir da memória principal.

- **Capacidade:** se a cache não puder acomodar todos os blocos necessários durante a execução do programa, a capacidade limitada da cache induz faltas.

- **Conflito:** se a estratégia de posicionamento de blocos não é totalmente associativa, faltas são induzidas quando blocos distintos da memória principal competem pela mesma posição da cache.

Para uma mesma capacidade, uma SPM ocupa uma menor área em silício, pois não requer células de memória para armazenamento de *tags* nem lógica de controle de busca associativa (comparadores e multiplexadores). Por exemplo, considerando um bloco de memória cache de 1024 bytes, mapeada diretamente e com tamanho de bloco de oito bytes, e um bloco de SPM de 1024 bytes, a SPM utiliza cerca de 68% menos de área, conforme dados obtidos com o uso do modelo físico de memórias CACTI [THO 08].

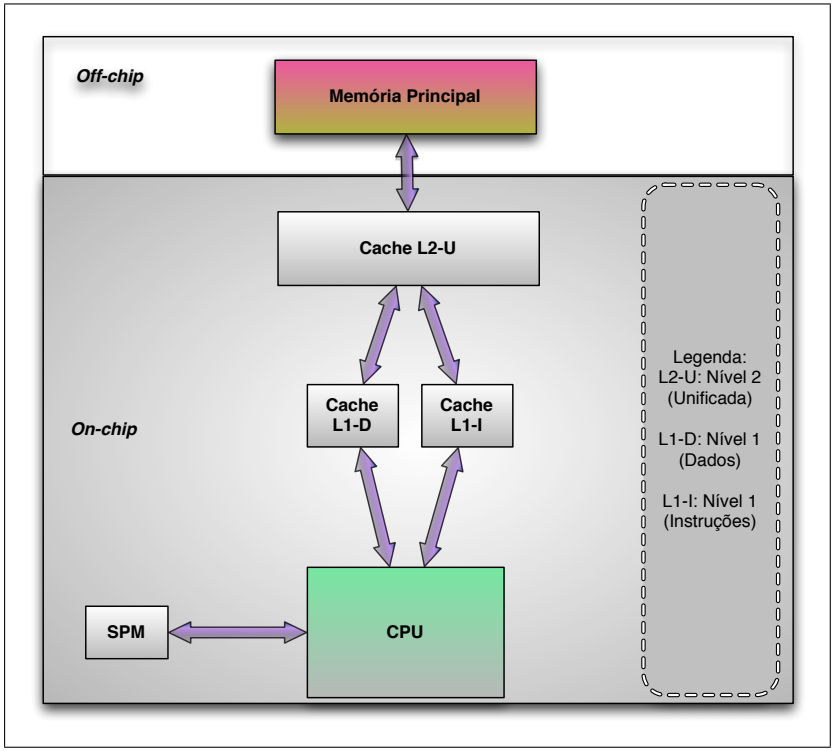

Figura 1.5: Representação esquemática de uma hierarquia de memória com dois níveis de cache

Para uma mesma capacidade, o consumo de energia de uma SPM é inferior ao de uma cache, pois esta consome mais energia devido à verificação de *tags* e ao roteamento do bloco selecionado. A Figura 1.6, extraída de [VER 07], ilustra, para cada tipo de memória, a energia consumida por acesso à cache com mapeamento direto, cache associativa de duas vias, cache associativa de quatro vias e SPM.

Para que se possa aproveitar o potencial de otimização oriundo da adoção de SPM faz-se necessária uma alocação eficiente de código e de dados.

Sem prejuízo para a generalidade da técnica proposta nesta dissertação, a partir de agora vamos assumir, por simplicidade, que a SPM e todas as memórias cache são internas ao SoC. Esta hipótese correlata com a maioria dos casos de interesse prático para SoCs.

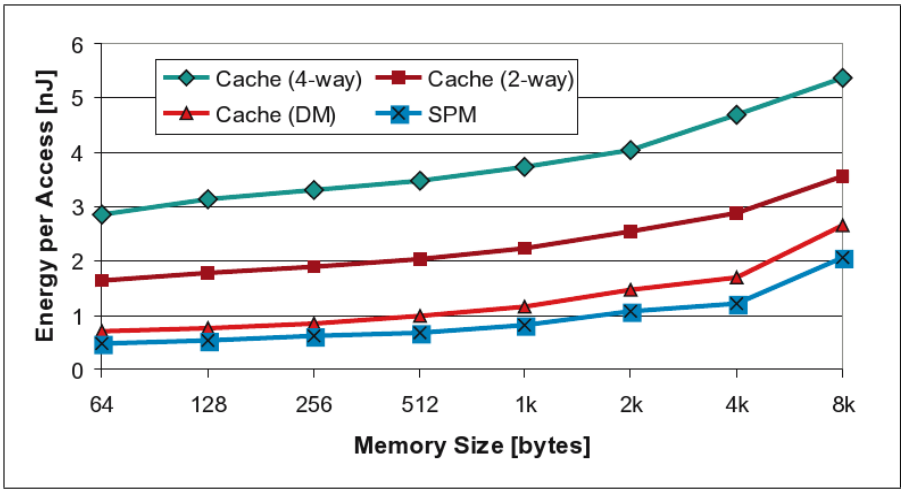

Figura 1.6: Consumo de energia por tipo de memória (Fonte:[VER 07])

Tabela 1.1: Caracterização dos elementos de programa cujo código-fonte é mostrado na Figura 1.7

Elemento de programa	Tamanho (bytes)
X	400
Y	400
Z	400
main	84

1.5 Exemplo motivacional

Suponha que o software cujo código-fonte é apresentado na Figura 1.7 deva ser executado em um processador MIPS R3000 embarcado em um SoC. Trata-se de um programa para soma de matrizes. Os tamanhos das matrizes e do código do procedimento `main` são mostrados na Tabela 1.1.

Suponha também que se queira determinar uma hierarquia para o subsistema de memória cuja configuração minimize a energia consumida. Para isso vamos analisar os quatro cenários de subsistema de memória mostrados na Figura 1.8, com o objetivo de identificar um particionamento de dados e código entre os componentes da hierarquia que leve ao menor consumo de energia.

As memórias cache usadas nos cenários C e D mostrados na Figura 1.8 usam mapeamento direto com tamanho de bloco de duas palavras (8 bytes).

Seja C_M a capacidade de uma memória M e E_M a energia consumida para acessar uma posição de M. A Tabela 1.2 mostra as capacidades de cache (C_{CACHE}) e SPM (C_{SPM}), expressas em bytes, assumidas para cada um dos quatro cenários da Figura 1.8. A Tabela 1.2 mostra tam-

```
1  #define N 10
2  int X[N][N];
3  int Y[N][N];
4  int Z[N][N];
5
6  int main()
7  {
8    register int i = 0;
9    register int j = 0;
10   for (i=0;i<N;i++)
11     for (j=0;j<N;j++)
12       X[i][j] = Y[i][j] + Z[i][j];
13   return 0;
14 }
```

Figura 1.7: Código-fonte de um exemplo de software a ser embarcado em um SoC

Tabela 1.2: Parâmetros considerados para os quatro cenários de subsistema de memória mostrados na Figura 1.8

Cenário	Parâmetros			
	E_{CACHE}(nJ)	E_{SPM}(nJ)	C_{CACHE}(bytes)	C_{SPM}(bytes)
A	-	-	0	0
B	-	0,0060	0	1024
C	0,0066	-	1024	0
D	0,0040	0,0033	512	512

bém a energia por acesso, expressa em nJ, para cache (E_{CACHE}) e para SPM (E_{SPM}) para cada um dos quatro cenários, configurados com as capacidades citadas anteriormente[1]. Estes valores de energia foram obtidos simulando-se cada configuração (i.e., cenário com as capacidades de cache e SPM especificadas na Tabela 1.2) utilizando-se a infraestrutura descrita no Capítulo 5. Em todos os cenários supõe-se que a energia por acessos para a MP (E_{MP}) é $2,30nJ$.

O código da Figura 1.7, ao executar no processador MIPS R3000, produz um total de 2074 acessos à memória, dos quais 300 são acessos a dados e 1774 são acessos referentes ao procedimento `main`. A análise comparativa de custo entre as quatro configurações considerou a energia total gasta no acesso a código e a dados. A Figura 1.9 ilustra graficamente o custo de cada configuração, expresso em energia total consumida apenas no subsistema de memória.

[1]Note que os valores da Figura 1.6 diferem dos da Tabela 1.2 devido a três fatores principais: a tecnologia utilizada para obter os valores da Figura 1.6 foi de 130nm, enquanto a da Tabela 1.2 foi 90nm; o tamanho de bloco de caches da Figura 1.6 não é fornecido, podendo ser diferente do tamanho de bloco das caches da Tabela 1.2; embora a estimativa de energia das memórias da Figura 1.6 e da Tabela 1.2 tenham sido obtidas com o modelo físico CACTI [THO 08], diferentes versões e provavelmente, diferentes parâmetros de otimização, foram utilizados.

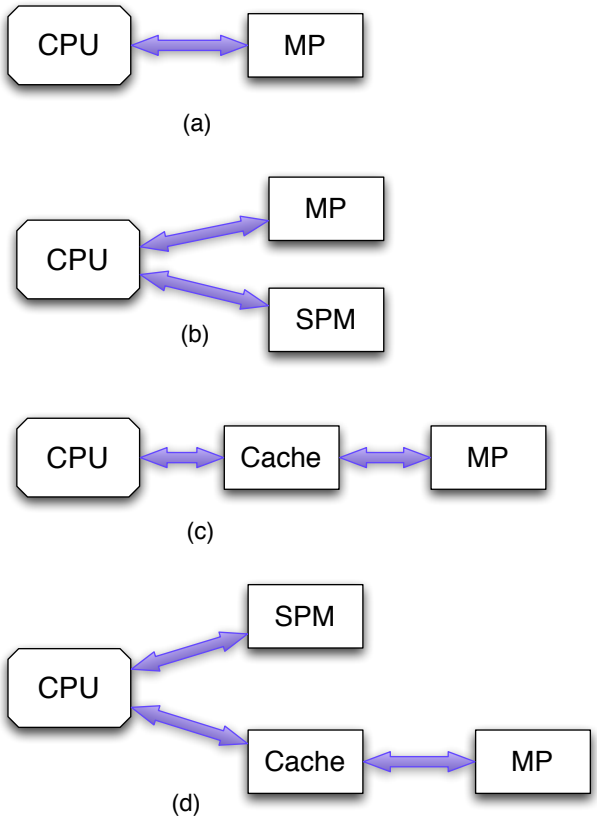

Figura 1.8: Cenários alternativos para o subsistema de memória

O cenário A (Figura 1.8a) resulta no maior consumo de energia por acesso, uma vez que a única memória existente é uma memória DRAM externa, que é utilizada como MP. Neste caso, a energia gasta por acesso é $E_{MP} = 2,30nJ$.

No cenário B (Figura 1.8b) assumiu-se SPM com 1024 bytes (conforme Tabela 1.2). Como o tamanho total dos elementos de programa é de 1284 bytes, um dos quatro elementos não pode ser alocado em SPM. Dada a ausência de cache, cada acesso ao elemento não alocado em SPM gasta $2,30nJ$ (pois $E_{MP} = 2,30nJ$).

No cenário C (Figura 1.8c) com a configuração especificada na Tabela 1.2 os acessos à memória principal ocorrem em menor proporção, mas o fato de todos os elementos de programa competirem por blocos na cache gera muitas faltas de conflito. Além disso, esses quatro elementos de programa sofrem o efeito de faltas compulsórias.

Finalmente, o cenário D (Figura 1.8d) com a configuração especificada na Tabela 1.2 resulta no menor custo porque `main` e dois dentre os três elementos restantes (`X`, `Y` e `Z`) podem ser alocados em SPM, ao passo que a cache minimiza a energia consumida nos acessos ao elemento de programa restante. Note que, devido ao fato de `main` e dois entre os três elementos de programa X, Y e Z terem sido alocados em SPM, o acesso a estes elementos não mais produz faltas compulsórias. Tampouco ocorrem faltas de conflito, pois um dentre os elementos X, Y e Z estará sozinho na cache.

Claramente, o cenário A não é interessante para sistemas embarcados, pois gasta muita energia e deixa o processador ocioso por muito tempo. O cenário B com 1024 bytes de SPM seria ideal se o tamanho total dos elementos de programa fosse menor ou igual à capacidade da SPM. (Neste caso, a memória principal nunca seria acessada.) Como para o exemplo da Figura 1.7 um dos elementos de programa não pode ser alocado em SPM, este cenário não é o mais adequado.

O cenário C com 1024 bytes de cache é interessante, mas devido a faltas por conflito e a faltas por capacidade na cache (entre `X`, `Y` ,`Z` e `main`), alguns blocos da cache podem ser descartados mesmo que sejam utilizados posteriormente. Além disso, em todos os primeiros acessos aos elementos de programa ocorrem faltas compulsórias.

O menor consumo de energia é obtido utilizando o cenário D com 512 bytes de cache e 512 bytes de SPM. Isso deve-se ao fato de não ocorrerem faltas compulsórias nem nos acessos a `main` nem nos acessos a dois dos três elementos de programa restantes. Tampouco ocorrem faltas de conflito na cache.

Este exemplo ilustra o fato de que, dada uma capacidade fixa de memória interna, o uso de SPMs pode ter impacto significativo na energia consumida devido à redução de faltas compulsórias e de conflito.

1.6 Escopo desta dissertação

Devido à sua grande latência e seu maior consumo de energia, memórias externas (*off-chip*) limitam a eficiência energética dos sistemas embarcados. Porém, para aplicações de alto desempenho, como de multimídia e telecomunicações, até mesmo o uso de memórias cache pode não ser suficiente para atingir o consumo de energia requerido, pois processadores embarcados contemporâneos podem consumir até 70% de sua energia no suprimento de dados e instruções via caches para o processador, conforme ilustra a Figura 1.4. Portanto, para atender aos requisitos de alto desempenho e baixo consumo de energia, fica evidente a necessidade de se utilizar memórias que tenham um desempenho similar às memórias cache, porém com um consumo de energia menor.

Tendo essencialmente a mesma latência, mas consumindo menos energia que uma memória cache de mesma capacidade, SPMs tem sido usadas como substitutas para caches ou como memórias complementares

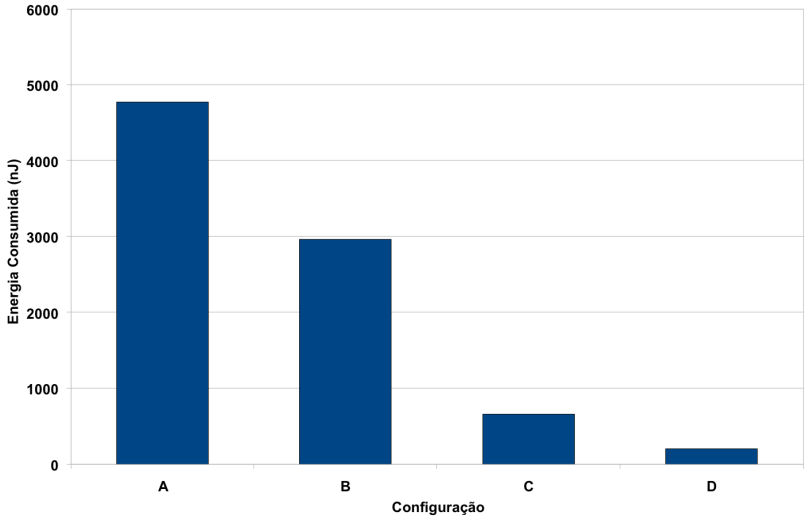

Figura 1.9: Energia consumida em diferentes configurações de memória para o exemplo ilustrativo

à cache (possivelmente através de reconfiguração [TAL 07]).

Esta dissertação aborda a alocação de código e de dados em SPM de forma a otimizar a energia consumida pelo subsistema de memória e/ou o seu desempenho.

1.7 Principais contribuições

A maioria dos trabalhos reportados na literatura faz a alocação em SPM a partir do código-fonte ou a partir de arquivo executável. Esta dissertação propõe uma nova técnica que viabiliza a alocação de código e dados em SPM a partir de arquivos-objeto relocáveis. Esta **abordagem pós-compilação** permite que, além de elementos de programas aplicativos, itens encapsulados em bibliotecas sejam passíveis de alocação em SPM, ao contrário das técnicas que fazem a alocação em tempo de compilação. Mostra-se que a adoção de arquivos-objeto relocáveis como ponto de partida melhora a eficiência da relocação em SPM, quando comparada com técnicas que manipulam diretamente arquivos executáveis. Embora a técnica proposta tenha sido concebida para não alocar dados dinâmicos (pilha e *heap*), os resultados experimentais mostram que essa limitação corresponde a se restringir a margem disponível para otimização em cerca de 61% da margem total (em média) para o conjunto de casos de uso reais do bem-conhecido *benchmark* Mibench [GUT 01]. Mostra-se também que a mera inclusão de dados estáticos como candidatos para a alocação em SPM resulta em economia extra de 21%, em média, para os programas

daquele *benchmark*.

As principais contribuições e trabalhos preliminares foram publicados em 2009, nos anais do IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2009) [MEN 09].

1.8 Organização desta dissertação

Esta dissertação está organizada da seguinte forma:

O Capítulo 2 apresenta os trabalhos correlatos em projeto de sistema de memória para SoCs, com especial atenção ao problema da alocação de código e/ou dados para as memórias embarcadas.

No Capítulo 3 são revisados alguns conceitos fundamentais necessários à formulação do problema-alvo e à descrição da técnica proposta para sua solução.

O Capítulo 4 descreve os algoritmos utilizados para resolver o problema-alvo.

O Capítulo 5 aborda a validação experimental da técnica proposta e a comparação com resultados de trabalhos correlatos.

As conclusões são mostradas no Capítulo 6, onde também são apresentadas alternativas para relaxar algumas das limitações do presente trabalho, com vistas a trabalhos futuros.

Capítulo 2

Trabalhos correlatos em hierarquia de memória para SoCs

Nos SoCs, o subsistema de memória representa o maior gargalo em termos de potência, desempenho e custo, conforme constatado em inúmeros trabalhos [KAN 02] [WUY 96] [VER 07].

Em um dos principais trabalhos em otimização de memória para sistemas de aplicação específica [CAT 98] [CAT 02], Catthor e colaboradores propuseram uma metodologia de projeto de sistema para a transferência de dados e exploração dos dispositivos de armazenamento. A metodologia denominada *Data Transfer and Storage Exploration* (DTSE) tem por objetivo reduzir o consumo de energia relacionado com os dados no subsistema de memória para uma dada aplicação sob restrições de tempo real. Ela consiste de duas principais etapas: otimização independente de plataforma e otimização dependente de plataforma. Na primeira etapa, o código-fonte da aplicação é otimizado, tornando os acessos à memória mais locais e regulares através de transformações de código, independentemente da plataforma-alvo. Na segunda etapa, o ordenamento dos acessos à memória é decidido com a alocação dos dados na memória da arquitetura-alvo. Entretanto, essa técnica não considera o uso de SPM em seu espaço de otimização. Como o foco desta dissertação é a alocação em SPM, técnicas como a de Catthor e colaboradores não serão analisadas em detalhe.

Trabalhos correlatos em otimização de hierarquia de memórias podem ser classificados em duas categorias. Dada uma aplicação-alvo, a primeira categoria aborda a otimização do hardware para uma alocação de memória pré-definida, enquanto a segunda categoria trata a otimização da alocação de memória para um hardware pré-definido.

2.1 Otimização do hardware para alocação pré-definida

Nesta estratégia, a hierarquia de memória é sintetizada ou reconfigurada levando em conta uma alocação de memória pré-definida da aplicação-alvo.

Panda, Dutt e Nicolau [PAN 97a] propuseram uma técnica analítica para a exploração de memórias internas para uma aplicação específica, a qual se baseia em um método de estimativa de desempenho. Para uma

dada capacidade total de memória interna, é feito um particionamento entre SPM (caracterizada pela sua capacidade) e cache (caracterizada por sua capacidade e tamanho de bloco). Para cada tamanho de memória interna, os autores consideram diferentes particionamentos entre cache e SPM, com a capacidade da cache variando sempre em múltiplos de dois. Para estimar o tamanho de bloco, o autor leva em conta os padrões de acesso à memória. Se esses são regulares e consecutivos (apresentam localidade espacial), então tamanhos de bloco maiores para as caches são desejáveis; por outro lado, se os acessos à memória são irregulares, um tamanho de bloco menor é desejável, pois isto reduz o tráfego de/para as memórias externas, evitando copiar dados desnecessários de/para a cache.

O método descrito no parágrafo anterior utiliza um modelo simplista para caches, que desconsidera outros parâmetros de configuração, tais como associatividade, política de escrita, etc. Esses parâmetros devem ser considerados pois podem ter uma influência significativa na energia consumida por esses tipos de memória (conforme ilustrado na Figura 1.6).

Já a abordagem proposta por Grunn, Dutt e Nicolau [GRU 01] analisa os padrões de acesso mais comuns de uma aplicação escrita na linguagem C de acordo com suas similaridades e conflitos. Como resultado dessa análise, essa abordagem personaliza a arquitetura, alocando um conjunto de módulos de memória a partir de uma biblioteca de módulos personalizados. Basicamente, os autores utilizam uma técnica de simulação por amostragem para retirar do espaço de pesquisa aquelas configurações que seriam inferiores. Em seguida, é executada uma simulação, que determina o consumo de energia e o desempenho somente para as configurações selecionadas. Para melhorar os acessos dos padrões mais frequentes da aplicação, módulos de memória personalizados são utilizados (normalmente simples e pequenos, o que acarreta em um melhor desempenho), enquanto os menos frequentes continuam sendo servidos pela memória cache interna.

Devido ao aumento significativo na complexidade dos sistemas, é comum existirem bibliotecas associadas à aplicação, as quais muitas vezes têm código-fonte proprietário. Como o método proposto por Grun, Dutt e Nicolau analisa os padrões a partir de um código-fonte na linguagem C, ele não é capaz de aplicar as otimizações levando em conta o código encapsulado na forma de bibliotecas pré-compiladas.

A partir de uma aplicação embarcada direcionada para uma família específica de processadores, o método proposto por Benini e colaboradores [BEN 02] tem como saída um leiaute completo de uma SPM particionada em múltiplos bancos, personalizada para a aplicação escolhida. A técnica consiste em dois passos: inicialmente, um algoritmo calcula a partição que minimiza o consumo de energia nos múltiplos bancos de SPM que podem ser acessadas simultaneamente (este particionamento é dependente da aplicação); em seguida, é gerada automaticamente a implementação física da arquitetura de memória particionada. A abordagem permite o acoplamento da fase de particionamento de memória com o projeto físico,

permitindo assim o uso de informações da implementação física na fase de particionamento. Infelizmente, esta técnica não é capaz de considerar o sistema de memória completo, uma vez que não considera caches.

As técnicas relatadas até aqui conduzem à economia de energia e melhora no desempenho, através da personalização do hardware. Entretanto, para aplicações que não admitem a personalização do subsistema de memória deve-se recorrer a técnicas de otimização de código, as quais são o objeto da próxima seção.

2.2 Otimização da alocação para hardware pré-definido

O problema de alocar dados e instruções frequentemente utilizados em memórias menores e mais rápidas vem sendo investigado desde o aparecimento das memórias cache, conforme relatado por Patterson e Hennessy [HEN 06]. Muitas técnicas baseadas em compilação para otimizar a alocação de cache foram desenvolvidas desde então. Uma revisão bastante ampla de tais técnicas pode ser encontrada em [KEN 01]. Mais recentemente, com o advento das SPMs, algumas dessas técnicas foram estendidas para hierarquias contendo este tipo de memória interna, técnicas estas que abordam tanto a alocação de código quanto a alocação de dados.

As próximas subseções sumarizam os trabalhos mais recentes em mapeamento de SPMs, levando em conta a persistência do conteúdo, a natureza das estruturas de dados mapeáveis, o método de otimização utilizado e a dependência do código-fonte.

2.2.1 Persistência do conteúdo em SPM

Algumas técnicas adotam alocação estática (*non-overlaying*). Nestas técnicas o conteúdo da SPM é carregado uma única vez, no início da execução da aplicação, sendo mantido até o final da execução. Esta técnica tem a vantagem de não introduzir *overhead*. Porém, quando a razão entre o tamanho do código e dados e o tamanho da SPM aumenta, este tipo de técnica pode não ser muito eficaz. Dentre as técnicas que adotam alocação estática pode-se citar [PAN 00] e [ANG 04].

Outras técnicas adotam alocação dinâmica (*overlaying*), as quais tentam manter em SPM os elementos de programa (dados e instruções) que são mais frequentemente acessados durante a execução do programa através de cópias da memória principal para a SPM e vice-versa (utilizando funções pré-definidas, como `memcpy` da biblioteca associada à linguagem C). A decisão sobre quais dados serão copiados de ou para a SPM ocorre em tempo de compilação através de *profiling* (com exceção do método apresentedo em [CHO 07] que utiliza uma abordagem pós-compilação). Dentre as técnicas que adotam alocação dinâmica pode-se citar [KAN 01], [STE 02b], [STE 02a], [DOM 05], [UDA 06], [CHO 07] e [MCI 08].

Os reposicionamentos dos elementos de programa de/para a SPM

incorrem no custo de instruções extras para copiar novos elementos para a SPM (e.g.: [KAN 01], [STE 02b], [STE 02a] e [UDA 06]) hardware extra para fazer tradução dinâmica de endereços ([CHO 07]) ou em alteração das funções de alocação de memória (`malloc`) e desalocação (`free`) (e.g.: [DOM 05], [MCI 08]).

2.2.2 Tratamento de estruturas de dados

Algumas técnicas mapeiam apenas código para a SPM (e.g.: [ANG 04] e [STE 02b]), enquanto outras mapeiam somente dados (e.g.: [PAN 00], [KAN 01], [CHO 07] e [UDA 06]). Dentre estas últimas uma aponta extensões para considerar também código ([UDA 06]). Finalmente, algumas mapeiam ambos, código e dados (e.g.: [STE 02a] e [VER 04]).

Panda, Dutt e Nicolau propuseram uma técnica para resolver o problema de mapeamento de variáveis escalares e vetores entre SPM e memória principal minimizando o número de ciclos gastos com acessos à memória [PAN 00]. A partir do código-fonte da aplicação, a técnica por eles proposta considera quatro critérios de seleção:

1. Alocação compulsória: para prevenir interferência com os vetores na cache de dados, todas as variáveis escalares e as constantes são alocadas em SPM.

2. Alocação baseada no tamanho dos vetores: todos os vetores com tamanho superior à capacidade da SPM são alocados em memória principal e são mapeadas via cache.

3. Alocação baseada no tempo de vida das variáveis: o tempo de vida é o período entre o primeiro acesso à variável e seu último acesso de leitura ou escrita. Variáveis com tempos de vida disjuntos podem compartilhar o mesmo espaço de endereçamento de memória cache sem gerar conflitos.

4. Alocação baseada na frequência de acesso e potencialidade de conflito em cache: para cada variável u, assume-se que a frequência de acesso a u e o potencial de conflito de seu acesso na cache determinam um fator de interferência $IF(u)$. Um alto fator de interferência indica que a variável tem uma grande probabilidade de estar envolvida em um grande número de conflitos de cache. Para o cálculo do fator de interferência, usa-se a equação $IF(u) = VAC(u) + IAC(u)$, onde $VAC(u)$ é o contador de acessos à variável u e $IAC(u)$ é o contador de interferências de acesso a u.

O problema é então mapeado através do Problema da Mochila, sendo então resolvido através de um algoritmo heurístico denominado de *value density aproximation* [GAR 79]. Os valores de lucro são os fatores de interferência e o peso dos itens, seu tamanho em bytes.

A técnica proposta em [KAN 01] escalona explicitamente as transferências de dados entre a memória externa e a SPM. As variáveis escalares da aplicação podem ser subdivididas em várias partes denominadas de *tiles*. Através de duas operações `read_tile` e `write_tile` os dados são copiados entre as memórias: a primeira indica uma cópia explícita da memória principal para a SPM; a segunda, uma cópia no sentido oposto. Note que essa é uma técnica de alocação dinâmica (*overlaying*).

Os *tiles* trazidos para a SPM devem ter um grande nível de reuso e, obviamente, não devem exceder a capacidade da SPM. Durante a execução, um número pré-determinado de *tiles* são copiados da memória principal para a SPM, sendo então ali acessados pela computação corrente. Se necessário, os *tiles* são re-alocados (e atualizados) em seus locais de origem em memória principal externa.

Cho e colaboradores propuseram um método dinâmico (*overlaying*) para um subsistema de memória particionado horizontalmente [EGG 06] (composto por uma SPM e uma mini-cache), utilizando uma unidade de gerenciamento de memória (MMU). A técnica proposta carrega dinamicamente os dados globais e de pilha por demanda quando a função que irá utilizar os dados é invocada. Um gerenciador de memória SPM, proposto pelos autores, carrega e descarrega as páginas de dados e mantém uma tabela de páginas para a MMU [CHO 07].

Udayakumaran e colaboradores propuseram uma técnica em que o compilador analisa o programa-fonte para identificar pontos onde possa ser benéfico inserir código para copiar um elemento de programa (variáveis globais escalares e não-escalares) da memória principal externa para a SPM. Tal inserção é benéfica se o ganho em latência ou energia ao alocar a variável em SPM for maior que o custo de sua transferência a partir da memória principal. Apesar de tal técnica somente alocar dados para a SPM, os autores apontam como alocar código e variáveis locais (localizadas na pilha) [UDA 06].

Angiolini e colaboradores propuseram uma técnica baseada na análise e em ajustes de endereços (*patching*) do código executável de uma aplicação [ANG 04]. Blocos de código de tamanho arbitrário são movidos para a SPM e desvios são inseridos antes e após o fim do bloco. Esta técnica gera expansão de código devido à presença de novas instruções nas fronteiras dos blocos (instrução de desvio para o bloco básico na SPM e instrução de retorno para a memória principal).

Steinke e colaboradores propuseram uma técnica que copia dinamicamente seções de código para a SPM [STE 02b]. Em pontos de programa pré-selecionados são inseridas chamadas para copiar uma parte de programa correspondente (por exemplo: um laço ou o corpo de uma função) para a SPM.

Steinke e colaboradores propuseram um algoritmo que analisa o código-fonte da aplicação e seleciona trechos de código e dados cuja cópia para a SPM contribui para minimizar o consumo de energia [STE 02a].

A técnica proposta por Verma, Wehmeyer e Marwedel tem como

finalidade compartilhar a SPM entre vários elementos de programa (variáveis globais, variáveis locais não-escalares e segmentos de código) utilizando uma técnica dinâmica (*overlaying*) [VER 04]. Esta técnica é capaz de tratar código e dados. Os autores fazem uma análise de tempo de vida dos elementos de programa através de seu padrão de acessos à memória.

Como uma extensão para métodos que tratam de vetores (por exemplo [STE 02a]) poderia se utilizar um analisador de *heap* que a convertesse em um grande vetor antes do mapeamento para SPM. Apesar de fragmentos da *heap* poderem ser mapeados para SPM, esta técnica seria fortemente limitada pelos tamanhos desconhecidos dos elementos em tempo de compilação, levando assim a um gerenciamento ineficiente de memória. Para superar o problema de se mapear elementos de *heap* de tamanhos desconhecidos, o método proposto em [DOM 05] aloca fragmentos de *heap* de tamanho fixo em porções de SPM, que são chamados *bins*. Os dados são movidos de/para a SPM em pontos de cópia (antes e depois de laços e em entradas e saídas de procedimentos). Procedimentos `malloc` e `free` modificados implementam a alocação e desalocação de *bins* (se existir um *bin* disponível, a função `malloc` aloca o fragmento de *heap* para a SPM; senão, para a memória principal). A desvantagem deste método é que variáveis sempre terão o mesmo endereço dentro de um *bin*, limitando assim o mapeamento para SPM.

Uma extensão desta idéia [MCI 08] divide o último fragmento utilizado para mapear um grande vetor (quando não completamento ocupado) em sub-fragmentos, os quais podem ser utilizados para mapear vetores menores. Infelizmente, nem sempre todos os fragmentos livres restantes podem ser efetivamente alocados.

2.2.3 Método de otimização utilizado no mapeamento

Usualmente as abordagens estáticas (*non-overlaying*) recaem na formulação do Problema da Mochila (*Knapsack Problem*) [KAR 72]. A solução adotada para o problema pode ser obtida através de programação linear inteira (ILP) (como em [STE 02a]) ou de programação dinâmica (como em [ANG 04] e [PAN 00]).

Por outro lado, as abordagens dinâmicas utilizam variadas formulações. Verma e colaboradores [VER 04], mostraram que a alocação dinâmica de SPM pode ser vista como um problema de alocação de registradores, o qual é então resolvido através de ILP.

O método em [CHO 07] também utiliza ILP. Porém, sua formulação baseia-se em uma estrutura de dados específica (o grafo de chamadas de sistema). Os métodos em [UDA 06] e [DOM 05], por sua vez, utilizam uma extensão do grafo de chamadas de sistema denominado Grafo de Relação Dados-Programa (DPRG). Nesse grafo, os vértices representam os elementos de programa e as arestas representam as chamadas entre os elementos.

A técnica proposta por McIlroy e colaboradores também trata a alo-

cação de memória dinâmica (*heap*) [MCI 08]. Para cada requisição de alocação de memória dinâmica tal técnica tenta alocar o dado requisitado na *heap* presente em SPM. Se a alocação falhar por falta de espaço em SPM, o dado é então alocado em memória principal. A representação da *heap* em SPM baseia-se em blocos de tamanho fixo com uma lista de bits para codificar a disponibilidade de cada bloco. O autor escolheu uma técnica de *first fit* para decidir em qual endereço da SPM o dado irá residir. Infelizmente, esta técnica pode resultar em um mapeamento não-ótimo em termos de energia. Para ilustrar tal deficiência, consideremos o exemplo a seguir. Suponha uma SPM de 1KB e duas estruturas de dados de 1KB cada: a primeira (denominada de estrutura `A`) corresponde a 5% da energia global gasta no programa; a segunda (denominada de estrutura `B`) corresponde a 50% da energia global gasta no programa. Agora suponha que seja feita uma requisição de alocação para a estrutura `A`. Portanto, esta estrutura será alocada em SPM esgotando assim o seu espaço livre. Agora imagine que seja feita uma requisição para a alocação da estrutura `B` (a estrutura `A` ainda reside em SPM). Claramente, esta estrutura será alocada em memória principal. Obtem-se assim uma economia de energia de cerca de 5%. Por outro lado, se a alocação levasse em conta o padrão de acessos da aplicação, poder-se-ia ter obtido cerca de 50% de economia de energia para o mesmo programa.

2.2.4 Dependência do código-fonte e tratamento de bibliotecas

Grande parte dos trabalhos em mapeamento de SPMs consiste de técnicas que operam em tempo de compilação (por exemplo: [KAN 01], [STE 02b], [STE 02a], [DOM 05], [UDA 06], [MCI 08], [PAN 00]). Nessas técnicas, o impacto reportado assume implicitamente o acesso a todo o código-fonte da aplicação. Porém, nos casos mais realistas, onde bibliotecas de terceiros, código legado ou código protegido por propriedade intelectual devem ser utilizados no projeto de sistemas embarcados, a melhoria esperada tende a ser menor do que a reportada, pois elementos de programa relevantes (possivelmente frequentemente invocados) podem ser excluídos do espaço de otimização se estiverem encapsulados em arquivos-binários (cujo código-fonte não está disponível).

Por outro lado, poucas técnicas otimizam a partir de arquivos binários (e.g.: [ANG 04] e [CHO 07]). Uma alternativa é identificar elementos de programa canditados à otimização diretamente do arquivo executável, como feito em [ANG 04]. Porém, o gerenciamento das relocações de SPM em arquivos executáveis é ineficiente, pois uma dada relocação para a SPM tende a gerar muitos ajustes no código e nos elementos de dados (isto será melhor explicado no Capítulo 4).

Outra abordagem é mapear fragmentos de memória para a SPM em tempo de execução utilizando um mecanismo de paginação por demanda, similar aos utilizados em sistemas com memória virtual [CHO 07]. Este método carrega páginas de dados globais e de pilha para a SPM quando

um procedimento é chamado. Como o mapeamento da SPM é feito em tempo de execução, esta abordagem trata de elementos de dados, independentemente de virem de uma biblioteca ou não. Porém, esta técnica requer a inserção de rotinas de gerenciamento de SPM no arquivo binário executável da aplicação e necessita de hardware dedicado.

2.3 A proposta deste trabalho frente aos trabalhos correlatos

É difícil (se não for impossível) conceber um método unificado que possa harmonizar todos os aspectos citados anteriormente. Uma divisão de tarefas entre abortagens complementares é provalvelmente mais pragmática. Por um lado, a maioria das técnicas dinâmicas apresentam um tratamento mais natural de dados dinâmicos (pilha e/ou *heap*). Porém, o tratamento de bibliotecas é geralmente limitado. Por outro lado, o tratamento de bibliotecas requer o tratamento de arquivos binários, os quais internamente particionam o código em elementos de dados estáticos e procedimentos.

Como a falta de suporte a bibliotecas e a código protegido por propriedade intelectual impede maiores economias de energia, como a alocação para SPMs de dados estáticos e procedimentos não induz expansão de código e como arquivos-objeto relocáveis conduzem a um *patching* mais eficiente, é provável que uma técnica estática (*non-overlaying*) que mapeie elementos de dados estáticos e procedimentos a partir de arquivos-objeto possa ser utilizada em conjunto com uma técnica dinâmica (*overlaying*) que mapeie dados dinâmicos[1].

Esta dissertação propõe um método de alocação estática de código e dados em SPM para uma dada aplicação-alvo. O método proposto não requer a disponibilidade do código-fonte nem de um hardware especial, o que amplia sua aplicação a bibliotecas pré-compiladas.

Dado um padrão de acessos à memória e os arquivos-objeto relocáveis dos módulos da aplicação e de eventuais bibliotecas externas, o método identifica os dados e código a serem alocados em SPM que minimizam o consumo de energia e/ou tempo de execução e, através de *patching*, implementa a alocação obtida, fazendo os ajustes necessários nas seções, na tabela de símbolos e nas tabelas de relocação dos arquivos-objeto da aplicação e nas bibliotecas (se necessário).

O problema-alvo abordado nesta dissertação e o método proposto para sua resolução são descritos nos dois próximos capítulos.

[1] Obviamente, cada técnica deveria operar sobre partições distintas de uma mesma SPM ou sobre distintos SPMs.

Capítulo 3

O problema-alvo e a solução proposta

Este capítulo inicialmente formaliza o problema-alvo abordado nesta dissertação, com o uso de nomenclatura apropriada. A seguir, a técnica proposta é apresentada. Tal técnica está baseada na decomposição do problema-alvo em três etapas distintas. A infraestrutura de suporte a sua implementação é então discutida. Os algoritmos utilizados para implementar a técnica proposta são objeto do Capítulo 4, ao passo que os resultados experimentais são apresentados e discutidos no Capítulo 5.

3.1 Formalização do problema-alvo

Considerando-se um sistema embarcado que possui SPM e caches, quando uma aplicação embarcada é compilada, os dados da aplicação podem ser armazenados tanto na SPM quanto em memória principal externa. Neste último caso, o acesso aos dados e instruções é feito via cache (exceto para faixas do espaço de endereçamento deliberadamente não mapeáveis para caches, para prover a inicialização das próprias caches ou para não incorrer na imprevisibilidade das caches).

A **latência** de acesso a uma memória M, denotada por λ_M, é definida como o tempo gasto, em ciclos de processador, para acessar uma posição na memória M.

A **energia consumida** para acessar uma posição na memória M, denotada por E_M, é definida como a energia média consumida ao ler ou escrever naquela posição. Uma memória M pode ser a memória principal (MP), uma memória de rascunho (SPM), uma cache de instruções ou de dados (I-cache ou D-cache), ou uma cache unificada (U-cache).

A **capacidade** de uma memória M, denotada por C_M, é seu tamanho expresso em bytes.

O problema geral de otimização consiste em determinar que dados ou segmentos de código serão alocados para uma SPM de forma a minimizar uma função custo (ou maximizar uma função lucro). A função custo pode ser mono-objetivo ou multi-objetivo, podendo capturar, por exemplo, apenas o tempo total gasto no acesso à memória, apenas o consumo de energia ou ambos (eficiência energética). Isso dará origem a diferentes instâncias do problema de otimização.

Como o acesso aos elementos depende do cômputo dos algoritmos usados no software embarcado, a otimização será feita capturando-se um padrão típico de acesso, caracterizado por um *trace* de memória T. Além disso, a aplicação precisa ser caracterizada em termos dos tamanhos de seus elementos de programa. A seguir, essas noções são formalizadas antes de se formular o problema-alvo.

Definição 3.1. (*trace* de memória): Um *trace* de memória `T` é uma tupla (α_1, α_2, ..., α_i, ... α_n) que representa a sequência de sucessivos endereços a serem acessados no subsistema de memória [UHL 97], onde α_i denota o i-ésimo endereço.

Definição 3.2. (Caracterização de custo dos elementos de programa): Sejam $D_1, ..., D_i, ..., D_n$ os elementos de programa (variáveis, instruções ou estruturas) acessados por um código embarcado. Sua caracterização é denotada por uma matriz-linha $W = [w_1, ..., w_i, ..., w_n]$, onde w_i é o espaço ocupado em SPM, expresso em bytes, para alocar o elemento D_i.

Definição 3.3. (Alocação de dados e instruções em SPM): Uma alocação para uma SPM é representada por uma matriz-coluna $X = [x_1, ..., x_i, ..., x_n]^{-1}$, onde $x_i = 1$ denota a alocação do elemento D_i na SPM e $x_i = 0$ denota sua não-alocação.

Problema 1. (Problema-alvo): Dados um conjunto de elementos D_i caracterizados pelo vetor W e um padrão de acesso caracterizado por um trace T, encontrar a alocação X que minimize a função custo $c(X)$ e seja tal que $W * X \leq C_{SPM}$.

O Problema 1 pode ser reformulado para recair num problema clássico de otimização combinatória denominado de Problema Binário da Mochila (*Binary Knapsack Problem*) [KAR 72], para o qual existem algoritmos propostos na literatura. Como em sua versão convencional o Problema Binário da Mochila aborda a maximização de uma função lucro (ao invés da minimização de uma função custo), o Problema 1 é reformulado para guardar máxima semelhança com o problema clássico, como mostrado a seguir.

Problema 2. (Reformulação baseada no Problema da Mochila): Dados um conjunto de elementos D_i caracterizados pelo vetor W e um padrão de acesso caracterizado por um trace T, encontrar a alocação X que maximize a função lucro $p(X) = P * X$ e seja tal que $W * X \leq C_{SPM}$, onde $P = [p_1, ..., p_i, ..., p_n]$, é uma matriz-linha onde p_i representa o lucro da alocação x_i.

Como o Problema 2 é NP-Completo [GAR 79], sua resolução usando técnicas exatas pode resultar em tempos de execução proibitivos para casos de uso reais. Embora muitos métodos utilizem Programação Linear Inteira (ILP: *integer linear programming*), ele pode também ser resolvido

de forma exata via Programação Dinâmica em tempo pseudo-polinomial[1], como foi provado em [PAP 81].

Note que, no Problema 2, a função lucro é determinada pela escolha de como avaliar o lucro de cada alocação. Para computar os valores de lucro, esta dissertação utiliza heurísticas que capturam o número de ciclos e o consumo de energia gastos no acesso a cada elemento, a partir do *trace* que caracteriza o padrão de acesso aos dados.

Deve-se ressaltar que, para uma dada função lucro, esta dissertação resolve o Problema 2 de forma exata através de Programação Dinâmica. Entretanto, a construção da função lucro baseia-se em uma aproximação do lucro real.

3.2 Fluxo de resolução do problema-alvo

Esta seção descreve a instrumentação necessária para obter e implementar a alocação ótima de código e dados em SPM a partir de propriedades do programa (extraídas de código executável) e de propriedades dos componentes do subsistema de memória (extraídas de uma descrição de suas características temporais e energéticas).

O fluxo de resolução do problema é ilustrado na Figura 3.1, o qual assume que, como resultado de pré-compilação, o código-objeto relocável para o software embarcado esteja disponível.

A partir do código-objeto, são extraídos os tamanhos dos elementos de programa D_i, os quais são anotados na matriz-linha W. Esta caracterização será abordada na Seção 4.3. Também a partir do código-objeto, é gerado um arquivo executável através do processo de relocação e linkedição. A partir do código executável gerado, obtém-se o trace T correspondente ao padrão de acessos induzidos pela execução do programa (aplicação), a taxa de faltas (m_i) induzida na memória cache para cada elemento de programa D_i e o número de acessos a_i de cada elemento de programa. Este processo será discutido na Seção 4.4.

A partir da descrição do subsistema de memória, extraem-se as latências de cada componente de memória (λ_{MP}, λ_{CACHE}, λ_{SPM}), bem como a energia média gasta no acesso a cada um deles (E_{MP}, E_{CACHE}, E_{SPM}) e a capacidade da SPM (C_{SPM}). Os parâmetros dependentes de tecnologia foram obtidos através do modelo físico de memórias CACTI [THO 08]. As configurações de memória utilizadas para instrumentação do CACTI serão discutidas no Capítulo 5.

As propriedades extraídas permitem a criação da matriz-linha P associada à função lucro, a partir da escolha de critérios heurísticos que

[1] Trata-se de um algoritmo cujo tempo de execução é, no pior caso, assintoticamente limitado por um polinômio em duas variáveis, digamos a e n. Por exemplo, para um problema de programação linear inteira com dimensões $n \times m$, o limite assintótico tem a forma $O(na^{p(m)})$ [PAP 81], onde $a = \max_{i,j}\{|a_{ij}|, |b_j|\}$ e $p(m)$ é um polinômio para um dado valor de m fixo. Uma discussão mais detalhada desse limite para a resolução do Problema 2 será feita na Seção 4.1

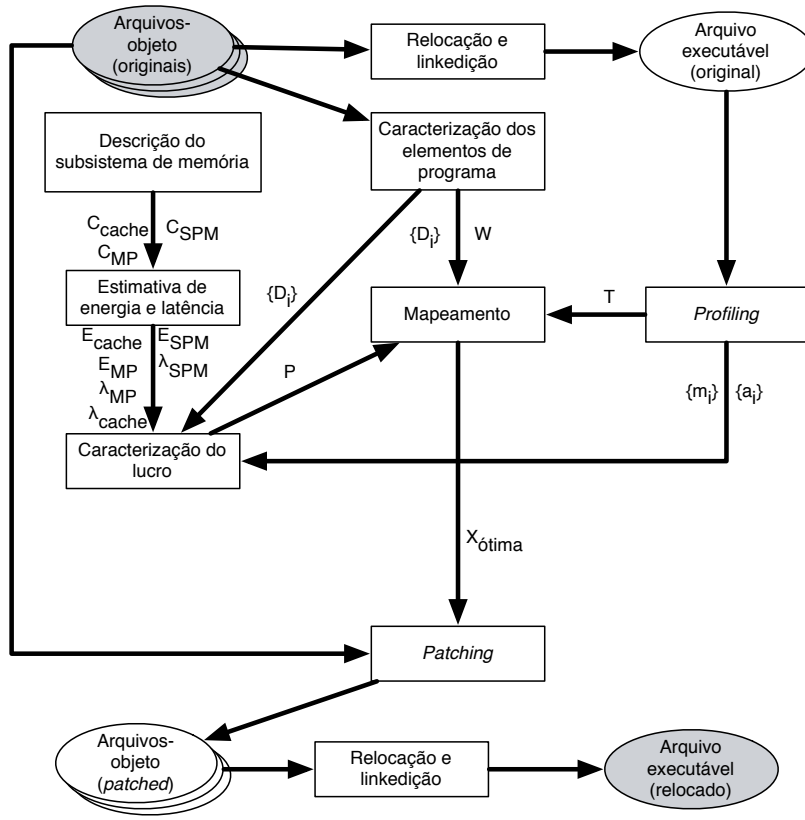

Figura 3.1: Fluxo proposto de pós-compilação para alocação de dados em SPM.

serão discutidos na Seção 4.5.

Com base nas propriedades do programa (aplicação) e do subsistema de memória onde será executado, a etapa de mapeamento consiste na resolução do Problema 2, instrumentado com a função lucro pré-caracterizada. O algoritmo para resolução do Problema 2 é discutido na Seção 4.2.

Uma vez obtida a alocação ótima ($X_{ótima}$), os elementos de programas dos arquivos-objeto são relocados para obedecê-la. Finalmente, as referências aos elementos alocados em SPM sofrem linkedição, dando origem ao código executável relocado para a SPM. Este processo será descrito na Seção 4.6.

Capítulo 4

A implementação da abordagem proposta

Este capítulo inicia descrevendo a resolução do problema-alvo: a Seção 4.1 discute a complexidade do Problema 2, a Seção 4.2 aborda a resolução do Problema da Mochila para uma função lucro arbitrária. As seções 4.3 e 4.4 abordam a caracterização dos elementos de programa e seus padrões de acesso. Em seguida, a Seção 4.5 mostra a caracterização de funções lucro, e a Seção 4.6 descreve a instrumentação e as ações necessárias para obter-se o arquivo executável otimizado.

4.1 A complexidade do Problema 2

Papadimitriou provou que a versão de decidibilidade de um programa linear inteiro do tipo $Ax = b$ ($x \geq 0$, inteiro), onde A é uma matriz $m \times n$, pode ser resolvida por um algoritmo cujo tempo de execução é limitado assintoticamente por $O(n^{2m+2}(ma)^{(m+1)(2m+1)})$, onde $a = \max_{i,j}\{|a_{ij}|, |b_j|\}$ [PAP 81]. Ele também provou que, para resolver a versão de otimização daquele problema, é preciso resolver o respectivo problema de decibilidade para todo inteiro z no intervalo $-(\sum_{j=1}^{n} |c_j| * M) - 1 \leq z \leq \sum_{j=1}^{n} |c_j| * M$, onde c_j é um dos coeficientes da função lucro e $M = n^2(ma^2)^{2m+3}$. Portanto, o problema de decidibilidade precisa ser resolvido no máximo $2 + 2M \sum_{j=1}^{n} |c_j|$ vezes.

Como o Problema da Mochila pode ser visto como um programa linear inteiro com uma única linha ($m = 1$), particularizando aquele limite assintótico para o Problema 2, temos que o problema de decidibilidade pode ser resolvido em tempo $O(n^4a^5)$ e é necessário resolvê-lo $2 + 2M \sum_{j=1}^{n} |p_j|$ vezes, onde $M = (n^2a^{10})$ e $a = \max_{i=1,n}\{w_1, w_2, ..., w_n, C\}$.

Como p_j não cresce[1] com a ou n para um dado *trace* (conforme Equações 4.3 e 4.4, a serem discutidas na Seção 4.5), o Problema 2 pode ser resolvido em tempo $O(n^7 a^{15})$ no pior caso, ou seja, em tempo pseudo-polinomial.

Note que o número de vezes em que o problema de decidibilidade precisa ser resolvido pode ser reduzido computando-se limites iterativamente e utilizando pesquisa binária. Por isso, a complexidade do caso médio pode ser bem menor, utilizando-se uma estratégia *Branch and Bound*. A eficiência do caso médio pode ser ainda melhor se o algoritmo escolhido utilizar Programação Dinâmica, onde a decomposição em subproblemas menores permite reduzir o grau dos polinômios envolvidos. Estas foram as diretrizes utilizadas na escolha de um algoritmo para resolver o Problema 2, o qual é descrito na próxima seção.

4.2 Escolha de um algoritmo para resolver o Problema 2

A evolução dos algoritmos para resolver eficientemente o Problema da Mochila resultou em uma série de noções importantes [HOR 72] [MAR 77] [NAU 76] [BAL 80] [MAR 88] [PIS 97] que são exploradas pelo algoritmo escolhido nesta dissertação. Assim, para facilitar a compreensão daquele algoritmo, as noções mais relevantes são primeiramente definidas na Seção 4.2.1, antes de serem utilizadas no algoritmo descrito na Seção 4.2.2. A Seção 4.2.3 discute aspectos de implementação relevantes para a eficiência daquele algoritmo.

4.2.1 Noções importantes

Embora algoritmos eficientes não devam resultar em pesquisa exaustiva, é conveniente enumerar o espaço de soluções do problema, ou seja, todas as alternativas de alocação das variáveis de decisão x_i.

Definição 4.1. (Espaço de soluções candidatas): Dada uma tupla de itens a alocar $(D_1, D_2, ..., D_n)$, seu espaço de soluções candidatas é $S^n = \{X = [x_1, x_2, ..., x_n]^{-1} \mid x_i \in \{0, 1\}, i = 1, 2, ..., n\}$.

A construção de uma solução para o Problema da Mochila é um processo de refinamento que parte de uma solução inicial (não necessariamente ótima) e a refina iterativamente até que se tenha garantia de que a solução refinada é a ótima.

Para garantir a exatidão sem incorrer na enumeração das $|S^n| = 2^n$ possibilidades, é preciso:

a) identificar os itens mais promissores,

[1] Ou seja, $O(\sum_{j=1}^{n} |p_j|) = \sum_{j=1}^{n} O(|p_j|) = \sum_{j=1}^{n} O(1) = O(n)$

b) enumerar apenas as alternativas de alocação dos itens mais promissores e

c) remover itens que, garantidamente, não levariam a soluções melhores.

Para fins de priorização, os itens candidados são mantidos em uma lista. Para evitar ambuiguidade ao se designar um item (propriamente dito), seu identificador e sua posição na lista ordenada, vamos fazer y denotar um item a alocar e y_j denotar o fato de que y ocupa a j-ésima posição na lista. Para nos referirmos aos atributos de um item que ocupa a j-ésima posição na lista, vamos adotar a seguinte notação: se y_j denota o fato de que um item, digamos D_i, reside na j-ésima posição da lista e a é um atributo do item D_i, então $\overline{a}$ denota o atributo a_i. Por exemplo, $\overline{p}_j = p_i$, $\overline{w}_j = w_i$, $\overline{e}_j = e_i$ e $\overline{x}_j = x_i$.

Para utilizar a eficiência como heurística de priorização, os itens candidatos são mantidos na lista em ordem decrescente de eficiências e são alocados à mochila sucessivamente até que seja atingido um item que, se incluído, excederia sua capacidade C. O último item alocado na mochila, antes de se exceder sua capacidade, é denominado **item de corte**, conforme formalizado abaixo.

Definição 4.2. (Item de corte): Dadas a capacidade C da mochila e a lista de itens a alocar $(y_1, y_2, ..., y_b, ..., y_n)$ tal que, para todo $p, q \in [1, n], p < q \Rightarrow \overline{e}_p \geq \overline{e}_q$, o item de corte é o item y_b tal que $\sum_{p=1}^{b-1} \overline{w}_p \leq C$ e $\sum_{p=1}^{b} \overline{w}_p > C$.

A solução tentativa que aloca na mochila todos os itens da lista ordenada anteriores ao item de corte, é denominada de **solução de corte**. Esta solução tentativa é o ponto de partida para o refinamento em busca da solução ótima. Esta noção é formalizada a seguir.

Definição 4.3. (Solução de corte): Denomina-se de solução de corte, a solução tal que: $\overline{x}_p = 1$ para $p = 1, ..., b-1$ e $\overline{x}_p = 0$, para $p = b, ..., n$, onde y_b é o item de corte.

Embora a solução de corte não seja necessariamente a solução ótima para o Problema Binário da Mochila, provou-se [BAL 80] que ela é a solução ótima do Problema Linear da Mochila[2]. Assim, é provável que seja necessário permutar somente alguns elementos com eficiência próxima à do item de corte y_b para obter a solução ótima da instância binária daquele problema.

[2]É o problema onde a restrição binária $x_j \in \{0, 1\}$ é relaxada para a restrição linear $0 \leq x_j \leq 1, x_j \in \Re$ para um determinado item j.

Por este motivo deve-se verificar se candidatos promissores (com eficiência próxima à do item de corte), mas não alocados na mochila, levariam a uma solução melhor se substituíssem algum (alguns) item (itens) nela alocado(s). Para sistematizar a pesquisa, o conjunto de itens a alocar é particionado de forma que:

a) uma primeira partição contenha os itens já alocados na mochila e cuja alocação não se queira no momento questionar,

b) uma segunda partição contenha os itens cuja alocação se queira avaliar e

c) uma terceira partição contenha os itens cuja não-alocação não se queira no momento questionar.

Como resultado desse particionamento, o espaço de soluções candidatas a ser pesquisado restringe-se ao espaço de soluções induzidas pela segunda partição, que será denominada de **núcleo de pesquisa**.

Para manter as partições de forma eficiente, o conjunto de itens a alocar é representado como uma lista e a posição de um item nessa lista indica a qual partição ele pertence. Assim, as partições são representadas como sub-listas. É conveniente designar uma sub-lista (e, consequentemente, uma partição) através dos intervalos que delimitam suas posições-limite.

Diferentes particionamentos podem ser explorados pelos algoritmos dependendo do número de elementos no núcleo de pesquisa, ou seja, do número de itens cuja alocação se queira pesquisar. Como os elementos de mais alta eficiência têm menor probabilidade de serem desalocados da mochila e os elementos de mais baixa eficiência têm menor probabilidade de serem alocados, o núcleo de pesquisa conterá itens de eficiência intermediária, próxima à eficiência do item de corte. Ora, para usar a solução de corte como ponto de partida, o particionamento deve garantir que o item de corte pertença ao núcleo de pesquisa. Essas noções são formalizadas a seguir.

Definição 4.4. (Particionamento de corte): Dados a lista de itens a alocar $(y_1, y_2, ..., y_b, ..., y_n)$ e o item de corte y_b, um particionamento é dito de corte se, e somente se, todas as condições abaixo forem verdadeiras:

- $\forall p \in [1, s-1] : \overline{e}_p \geq \overline{e}_s$,
- $\forall p \in [t+1, n] : \overline{e}_p \leq \overline{e}_t$,
- $b \in [s, t]$.

Note que a propriedade fundamental do núcleo de pesquisa, designado pelo intervalo $[s, t]$, é a de que seus elementos tem eficiência menor do que os da primeira partição (induzida pelo intervalo $[1, s-1]$) e eficiência maior do que os da terceira partição (induzida pelo intervalo $[t+1, n]$).

Para garantir a exatidão sem pesquisa exaustiva, enumerando primeiramente as alternativas de alocação do itens mais promissores, é eficiente restringir-se a enumeração aos elementos do núcleo de pesquisa, conforme sugerido por Balas e Zemel [BAL 80] e expandir o núcleo de pesquisa à medida que soluções comprovadamente inferiores não precisem ser pesquisadas.

Quando o núcleo contém apenas o item de corte, a pesquisa seria restringida a avaliar se sua alocação ($\overline{x}_b = 1$) ou não-alocação ($\overline{x}_b = 0$) levaria a soluções melhores do que a solução de corte. Embora a alocação isolada do item de corte não resulte (por definição) em solução factível, ela poderia ser considerada se um item i que já tenha sido alocado devido à sua maior eficiência ($i \in [1, s-1]$), fosse desalocado para dar-lhe lugar e que a não-alocação de um item j devido à sua menor eficiência ($j \in [t+1, n]$) possa ser reconsiderada.

Como o núcleo supõe que os itens à sua esquerda estejam alocados e os itens à sua direita não alocados, mas os itens do núcleo não tem alocação definida, isso dá origem à noção de subespaço de soluções por ele induzido, conforme formalizado a seguir.

Definição 4.5. (Subespaço de soluções candidatas induzido pelo núcleo): Dado um núcleo com k itens, delimitado pelo intervalo $[s, t]$, o subespaço de soluções candidatas por ele induzido, denotado por $S^{[s,t]}$, é o conjunto $\{\overline{X} = [\overline{x}_1, \overline{x}_2, ..., \overline{x}_{s-1}, \overline{x}_s, ..., \overline{x}_t, \overline{x}_{t+1}, ..., \overline{x}_n]^{-1} \mid \overline{x}_i = 1$ para $i \in [1, s), \overline{x}_i = 0$ para $i \in (t, n]$ e $\overline{x}_i \in \{0, 1\}$ para $i \in [s, t]\}$, onde $k = t - s + 1$.

Como a enumeração de soluções vai se restringir ao núcleo, pode-se representar implicitamente a alocação dos itens à esquerda e a não-alocação dos itens à sua direita. Como consequência, cada solução candidata nesse subespaço pode ser representada por um **vetor parcial de alocação** $\overline{X}^{[s,t]} = [\overline{x}_s, \overline{x}_{s+1}, ..., \overline{x}_t]$.

Além disso, convém caracterizar cada solução candidata através de dois de seus atributos, as somas dos lucros e pesos dos itens alocados na mochila ou seja, o lucro e o peso acumulados, dando origem à noção de **estado de uma solução candidata**, como formalizado abaixo.

Definição 4.6. (Estado de uma solução candidata): Dados o i-ésimo vetor parcial de alocação $\overline{X}_i^{[s,t]}$ explorado, o estado s_i a ele associado é $s_i = (\pi_i, \mu_i, \overline{X}_i^{[s,t]})$, onde:

- $\pi_i = \sum_{j=1}^{s-1} \overline{p}_j + \sum_{j=s}^{t} \overline{p}_j * \overline{x}_j$
- $\mu_i = \sum_{j=1}^{s-1} \overline{w}_j + \sum_{j=s}^{t} \overline{w}_j * \overline{x}_j$

Pode-se agora enumerar todos os estados associados ao espaço de soluções candidatas induzidos pelo núcleo, como formalizado abaixo.

Definição 4.7. (Espaço de pesquisa): Dado o subespaço $S^{[s,t]}$ de soluções candidatas induzidos pelo núcleo, seu espaço de pesquisa é o conjunto $S = \{(\pi_i, \mu_i, \overline{X}_i^{[s,t]})\}$.

Como a intenção é a de diminuir o número de estados a serem pesquisados pelo algoritmo, estados que garantidamente levariam a soluções inferiores devem ser removidos do espaço de pesquisa. Dados dois estados, um estado é dito inferior se, e somente se, seu lucro acumulado for menor que o lucro acumulado do outro e seu peso acumulado for maior que o peso acumulado do outro. A noção de dominância, formalizada abaixo, é a chave para descartar soluções inferiores do espaço de pesquisa.

Definição 4.8. (Dominância): Dados dois estados $s_i, s_j \in S$, s_i domina s_j se $\pi_i \geq \pi_j$ e $\mu_i \leq \mu_j$.

Foi provado em [TOT 80] que, dados dois estados s_i e s_j tais que s_i domina s_j, então o estado s_j pode ser descartado, pois a solução ótima é preservada no espaço de pesquisa remanescente.

Para verificar se a inclusão de um novo item no núcleo é promissora, é necessário guardar o lucro acumulado do estado associado à melhor solução factível encontrada até então. Isto dá origem à noção de **limite inferior de lucro do espaço de pesquisa**, formalizada abaixo.

Definição 4.9. (Limite inferior de lucro do espaço pesquisa): Dado o espaço de pesquisa S, seu limite inferior de lucro, denotado por $z(S)$, pode ser definido como $z(S) = \max\{\pi_i\}, \forall\, s_i \in S \wedge \mu_i \leq C$.

4.2.2 O algoritmo MINKNAP

Ao invés de simplesmente reproduzir o algoritmo original [PIS 97], o algoritmo MINKNAP será reformulado para explicitar conceitualmente as noções que garantem a exatidão e a eficiência, sem entrar em detalhes de implementação. Esta reformulação permitirá uma análise crítica da aplicabilidade desse algoritmo a casos de uso reais (Seção 4.2.3).

Antes de explicar o algoritmo MINKNAP propriamente dito (Algoritmo 6), serão explicados os principais procedimentos que aquele algoritmo invoca: ADD (Algoritmo 1), ENUMERATE (Algoritmo 2), REDUCESET (Algoritmo 3), PROMISING (Algoritmo 4) e DEFINESOLUTION (Algoritmo 5).

Assuma que, para o atual núcleo de pesquisa, foram explorados todos os estados representando soluções do problema. Seja s_i um desses estados já explorados. Quando o núcleo for estendido com um novo item promissor que ocupa a posição ℓ na lista de itens a alocar, digamos y_ℓ, um novo estado s_j será obtido, a partir de s_i, para representar uma nova solução a ser explorada (induzida por aquela extensão).

Dados um estado s_i e um item y_ℓ, o Algoritmo 1 considera a inclusão de y_ℓ no núcleo de pesquisa e retorna um novo estado s_j a ser explorado. Inicialmente, o algoritmo verifica se y_ℓ é proveniente da partição à esquerda do núcleo (linha 3). Em caso positivo, y_ℓ estava previamente alocado à mochila e, portanto, deve-se revogar essa alocação implícita. Com a revogação, sua contribuição indevida ao peso e ao lucro acumulados precisa ser compensada (veja Definição 4.6). A compensação é obtida invertendo-se os sinais do lucro e do peso daquele item (linhas 4 e 5). Caso y_ℓ provenha da partição à direita do núcleo, sua não-alocação implícita precisa ser revogada de forma similar. Neste caso, entretanto, os sinais do lucro e do peso do item são preservados, pois precisam ser acumulados (linhas 9 e 10). Para refletir a inclusão de y_ℓ no núcleo, o algoritmo estende (à esquerda ou à direita) o vetor parcial de alocação para acomodá-lo (linhas 6 e 11). No caso de o item estar à esquerda do núcleo, sua alocação já foi explorada no estado s_i; por isso, sua variável de decisão não será enumerada ($\overline{x}_\ell = 0$ à linha 6). Algo complementar ocorre quando o item situar-se à direita do núcleo ($\overline{x}_\ell = 1$ à linha 11), pois sua não-alocação já foi explorada no estado s_i.

Algoritmo 1 ADD

Entradas: $\ell, s_i = (\pi_i, \mu_i, \overline{X}_i^{[s,t]})$
Saídas: s_j
1: Seja $Y = (y_1, y_2, ..., y_b, ..., y_n)$ a lista de itens a alocar;
2: Seja y_b o item de corte conforme a Definição 4.2;
3: **se** $(\ell < b)$ **então**
4: $\quad \pi_j \leftarrow \pi_i - \overline{p}_\ell$;
5: $\quad \mu_j \leftarrow \mu_i - \overline{w}_\ell$;
6: $\quad \overline{X}_j^{[s-1,t]} \leftarrow [0, \overline{x}_s, \overline{x}_{s+1}, ..., \overline{x}_t]^{-1}$, onde $\overline{x}_k \in \overline{X}_i^{[s,t]}$ e $k = s, s+1, ..., t$;
7: $\quad$ **retorne** $s_j = (\pi_j, \mu_j, \overline{X}_j^{[s-1,t]})$;
8: **senão**
9: $\quad \pi_j \leftarrow \pi_i + \overline{p}_\ell$;
10: $\quad \mu_j \leftarrow \mu_i + \overline{w}_\ell$;
11: $\quad \overline{X}_j^{[s,t+1]} \leftarrow [\overline{x}_s, \overline{x}_{s+1}, ..., \overline{x}_t, 1]^{-1}$, onde $\overline{x}_k \in \overline{X}_i^{[s,t]}$ e $k = s, s+1, ..., t$;
12: $\quad$ **retorne** $s_j = (\pi_j, \mu_j, \overline{X}_j^{[s,t+1]})$;

Considere agora o conjunto de todos os estados já explorados para o atual núcleo de pesquisa, digamos S. Quando o núcleo for estendido com um novo item promissor y_ℓ, será induzido um novo conjunto de estados a explorar, digamos S'.

Dados o conjunto S de estados já explorados e o item promissor y_ℓ (a ser incluído no núcleo), o Algoritmo 2 cria um novo estado a explorar

para cada estado já explorado e retorna o conjunto S' de estados a explorar, dele previamente removendo estados que resultariam em soluções inferiores.

Inicialmente, o novo conjunto de estados S' preserva todos os estados explorados até então (linha 1). Para cada estado $s_i \in S$, um novo estado s'_i é criado em S', refletindo a inclusão de y_ℓ no núcleo (linhas 2 a 4). Se algum estado em S' domina o novo estado s'_i, sua inclusão em S' é revogada (linhas 6 e 7). Caso o novo estado não seja dominado, o algoritmo remove de S' todo estado explorado s'_j dominado por cada s'_i a explorar (linhas 9 e 10).

Como o objetivo é o de mostrar conceitualmente o cômputo do novo conjunto de estados, o Algoritmo 2 foi escrito (deliberadamente) de forma não otimizada, induzindo uma complexidade $O(|S|^2)$, no pior caso. Para fins de implementação, Pisinger [PIS 97] propôs uma versão em que, ao se manter o conjunto de estados em ordem crescente do atributo μ_i, o algoritmo pode ser reescrito de forma que sua complexidade reduz-se a $O(|S|)$.

Algoritmo 2 ENUMERATE

Entradas: S, ℓ
Saídas: S'
1: $S' \leftarrow S$;
2: **para** todo $s_i \in S$ **faça**
3: $\quad s'_i \leftarrow \text{ADD}(s_i, \ell)$;
4: $\quad S' \leftarrow S' \cup \{s'_i\}$;
5: $\quad$ **para** todo $s'_j \in S'$ **faça**
6: $\quad\quad$ **se** (s'_j domina s'_i) **então**
7: $\quad\quad\quad S' \leftarrow S' - \{s'_i\}$
8: $\quad\quad$ **senão**
9: $\quad\quad\quad$ **se** (s'_i domina s'_j) **então**
10: $\quad\quad\quad\quad S' \leftarrow S' - \{s'_j\}$;
11: **retorne** S';

Além de se explorar a relação de dominância entre estados já visitados (como no Algoritmo 2), pode-se remover estados que não levariam a estados promissores. Essa noção é usada pelo Algoritmo 3.

Dados o conjunto de estados já explorados (S), o Algoritmo 3 remove os estados cuja exploração não é promissora, com exceção de todo estado cujo lucro acumulado é igual ao limite inferior de lucro (o lucro da melhor solução encontrada até então). Ao final, o algoritmo retorna o conjunto reduzido de estados. O critério para remoção é o seguinte: um estado s_i é removido se os estados imediatamente alcançáveis a partir de s_i induzem soluções comprovadamente inferiores e se o lucro acumulado de s_i (π_i) não corresponder ao limite inferior de lucro do espaço de pesquisa ($z(S)$); ou seja, preserva-se no espaço de pesquisa todos os estados

com lucro igual ao da melhor solução até então encontrada. Dado um estado s_i, os estados alcançáveis a partir de s_i são aqueles que resultariam da expansão do núcleo à esquerda ou à direita. Para se decidir o sentido de expansão do núcleo, o Algoritmo 3 utiliza o critério de factibilidade de uma solução: um estado s_i induz uma solução factível se seu peso acumulado (μ_i) excede a capacidade da mochila (linha 3).

Caso a solução induzida por s_i seja infactível, o próximo estado a ser explorado corresponderia à desalocação do item na posição $s-1$ da lista (expansão do núcleo à esquerda). No caso de s_i induzir uma solução factível, o próximo estado corresponderia à alocação do item na posição $t+1$ da lista (expansão do núcleo à direita). Por isso, o Algoritmo 3 calcula uma estimativa de lucro acumulado (u) considerando a desalocação do item imediatamente à esquerda de s (linha 4) ou a alocação do item imediatamente à direta de t (linha 6).

Se a estimativa de lucro acumulado for inferior ao atual limite inferior de lucro do espaço de pesquisa (linha 7), o estado s_i comprovadamente não induziria solução superior. Assim, caso o lucro acumulado do estado s_i (π_i) não corresponda ao limite inferior de lucro ($z(S)$), s_i pode ser removido do espaço de pesquisa (linha 8), com a garantia de que soluções ótimas são preservadas no espaço de pesquisa remanescente. A complexidade do Algoritmo 3 é claramente $O(|S|)$, no pior caso.

Algoritmo 3 REDUCESET

Entradas: S
Saídas: S
1: Sejam s e t as posições-limite do núcleo;
2: **para** todo $s_i \in S$ **faça**
3: **se** $(\mu_i > C)$ **então**
4: $u \leftarrow \pi_i + (C - \mu_i) * \overline{e}_{s-1}$;
5: **senão**
6: $u \leftarrow \pi_i + (C - \mu_i) * \overline{e}_{t+1}$;
7: **se** $(u < z(S) \wedge \pi_i \neq z(S))$ **então**
8: $S \leftarrow S - \{s_i\}$;
9: **retorne** S;

Antes de enumerar uma nova solução é preciso investigar se ela é promissora. Dados o conjunto S de estados já explorados e um item candidato y_ℓ, o Algoritmo 4 verifica se a inclusão de y_ℓ no núcleo é ou não promissora. Para isso, o algoritmo estima o lucro máximo (max) obtido ao se expandir o núcleo e o compara com o atual limite inferior de lucro do espaço de pesquisa ($z(S)$). Se a inclusão de y_ℓ elevar o limite inferior de lucro, ela é considerada promissora.

Para estimar o lucro máximo induzido pela inclusão de y_ℓ no núcleo, o Algoritmo 4 estima preliminarmente o impacto individual de sua alocação (linha 4) ou não-alocação (linha 6). Lembre que, no caso de re-

vogação de alocação, as contribuições de y_ℓ previamente acumuladas ao lucro e ao peso globais precisam ser compensadas; no caso de revogação de não-alocação, as contribuições de y_ℓ precisam ser acumuladas ao lucro e ao peso globais (veja Definição 4.6).

Em seguida, o Algoritmo 4 computa a estimativa de lucro máximo (max) ao se expandir o espaço de pesquisa considerando a inclusão de y_ℓ ao núcleo (linhas 7 a 13). Essa estimativa deve considerar o fato de que, se a inclusão do item resultar em transbordo da mochila (linha 8), para ocorrer a inclusão de y_ℓ, o primeiro item à esquerda do núcleo (posição $s-1$ na lista) necessita ser desalocado para dar-lhe espaço (linha 9) ou, em caso contrário, ao ocorrer a inclusão de y_ℓ no núcleo, o primeiro item à direita do núcleo (posição $t+1$ na lista) pode ser alocado (linha 11), pois na inclusão de y_ℓ no núcleo não acontece um transbordo da mochila. Se a estimativa de lucro máximo (max) elevar o atual limite inferior de lucro (linha 14), o Algoritmo 4 retorna predicado "Verdadeiro" e, em caso contrário, predicado "Falso". A complexidade do Algoritmo 4 é $O(|S|)$, no pior caso.

Algoritmo 4 PROMISING

Entradas: S, ℓ
Saídas: Verdadeiro se a inclusão de ℓ é promissora, Falso em caso contrário

```
1:  Sejam s e t as posições-limite do núcleo;
2:  max ← 0;
3:  se (ℓ < s) então
4:      p' ← −p̄_ℓ;  w' ← −w̄_ℓ;
5:  senão
6:      p' ← p̄_ℓ;  w' ← w̄_ℓ;
7:  para todo s_i ∈ S faça
8:      se (μ_i + w' > C) então
9:          u ← (π_i + p') + (C − (μ_i + w')) ∗ ē_{s−1}
10:     senão
11:         u ← (π_i + p') + (C − (μ_i + w')) ∗ ē_{t+1}
12:     se (u ≥ max) então
13:         max ← u;
14: se (max > z(S)) então
15:     retorne Verdadeiro;
16: senão
17:     retorne Falso;
```

Uma vez encontrada uma solução ótima, é preciso construir sua matriz de alocação. Dado o estado com máximo lucro encontrado durante o processo de exploração, digamos s_k, o Algoritmo 5 retorna a respectiva matriz de alocação X. Para cada uma das posições da lista Y de itens a alocar (linhas 2 e 3), o algoritmo identifica o item, digamos D_i, que ocupa

Algoritmo 5 DEFINESOLUTION

Entradas: $s_k = (\pi_k, \mu_k, \overline{X}_k^{[s,t]})$
Saídas: $X = [x_1, x_2, ..., x_i, ..., x_n]^{-1}$
1: Sejam s e t as posições-limite do núcleo;
2: Seja $Y = (y_1, y_2, ..., y_j, ..., y_n)$ a lista de itens a alocar;
3: **para** todo $j \in [1, n]$ **faça**
4: Seja D_i o item na j-ésima posição de Y;
5: **se** $(j < s)$ **então**
6: $x_i \leftarrow 1$;
7: **se** $(s \leq j \leq t)$ **então**
8: $x_i \leftarrow \overline{x}_j$;
9: **se** $(j > t)$ **então**
10: $x_i \leftarrow 0$;
11: **retorne** X;

uma determinada posição j naquela lista e atribui um valor à sua variável de decisão x_i na matriz X, dependendo de sua posição em relação ao núcleo de pesquisa, de acordo com a Definição 4.5: se o item estiver à esquerda (linha 5) ou à direita (linha 9) do núcleo, isto significa, respectivamente, que naquele estado o item D_i foi implicitamente alocado (linha 6) ou implicitamente não-alocado (linha 10); se o item D_i pertencer ao núcleo, sua alocação ou não-alocação foi explicitamente representada naquele estado pela variável de decisão $\overline{x}_j$ do vetor $\overline{X}_k^{[s,t]}$.

Agora que todos os procedimentos a serem invocados foram explicados, pode-se descrever com clareza o algoritmo principal. Dadas a matriz de lucros P, a matriz de pesos W e a capacidade da mochila C, o Algoritmo 6 retorna a matriz de alocação X correspondente a uma solução ótima para o Problema 2, ao se atribuir a C o valor C_{SPM}.

O primeiro passo do Algoritmo 6 é encontrar o item de corte (linha 2), que, inicialmente, torna-se o primeiro elemento do núcleo (linha 3). Em seguida, o espaço de pesquisa é inicializado com os dois estados resultantes da inclusão do item de corte no núcleo (linha 4), um deles representando sua não-alocação e outro representado sua alocação à mochila. Como o único estado factível é o estado que corresponde à não-alocação do item de corte (Definição 4.2), a variável $best$, que armazena o estado factível com maior lucro acumulado, é inicializada com esse estado (linha 5).

Em seguida, verifica-se se a inclusão dos itens vizinhos ao atual núcleo são promissoras (linhas 6 e 7); isto é, se sua inclusão no núcleo eleva o limite inferior de lucro do espaço de pesquisa. Os predicados resultantes dessa verificação para os itens nas posições $s - 1$ e $t + 1$ são denotados por Ps e Pt, respectivamente. Enquanto houverem vizinhos promissores, o Algoritmo 6 explora o espaço de pesquisa S, buscando enumerar estados

associados à exploração de itens promissores e buscando remover todos os estados que induziriam soluções comprovadamente inferiores (linhas 8 a 19). Como resultado da remoção de estados, serão preservados no espaço de pesquisa S somente estados cujos lucros acumulados (π_i) são iguais ao da melhor solução encontrada ($best$), que corresponde à solução com maior lucro acumulado que não excede a capacidade da mochila (veja Definição 4.9). Ao final de cada iteração, com a expansão do núcleo, Ps e Pt são atualizados para os novos vizinhos (linhas 18 e 19).

Algoritmo 6 MINKNAP

Entradas: P, W, C
Saídas: X

1: Seja $Y = (y_1, y_2, ..., y_j, ..., y_n)$ a lista de itens a alocar;
2: encontre o item de corte y_b conforme a Definição 4.2;
3: $[s, t] \leftarrow [b, b]$;
4: $S \leftarrow \{(\pi_b, \mu_b, \{0\}), (\pi_b, \mu_b, \{1\})\}$;
5: $best \leftarrow (\pi_b, \mu_b, \{0\})$;
6: $Ps \leftarrow$ PROMISING$(s - 1, S)$;
7: $Pt \leftarrow$ PROMISING$(t + 1, S)$;
8: **enquanto** $(Ps \vee Pt)$ **faça**
9: **se** (Ps) **então**
10: $S \leftarrow$ ENUMERATE$(S, s - 1)$;
11: $s \leftarrow s - 1$;
12: REDUCESET(S);
13: **se** (Pt) **então**
14: $S \leftarrow$ ENUMERATE$(S, t + 1)$;
15: $t \leftarrow t + 1$;
16: REDUCESET(S);
17: $best \leftarrow s_k \in S \mid \pi_k = z(S) \wedge \mu_i \leq C$
18: $Ps \leftarrow$ PROMISING$(s - 1, S)$;
19: $Pt \leftarrow$ PROMISING$(t + 1, S)$;
20: $X \leftarrow$ DEFINESOLUTION$(best)$;
21: **retorne** X;

Vamos agora detalhar a exploração de itens promissores (linhas 9 a 19). O Algoritmo 6 primeiramente investiga se o item imediatamente à esquerda do núcleo é promissor (linha 9) e, em caso afirmativo, enumera novos estados a serem pesquisados sob essa hipótese (linha 10). Após registrar a expansão do núcleo à esquerda (linha 11), o algoritmo reduz o espaço de pesquisa, eliminado estados que induziriam soluções inferiores (linha 12). De forma similar, o Algoritmo 6 trata o cenário complementar em que se investiga um item promissor imediatamente à direita do núcleo (linhas 13 a 16). Assim, a cada iteração do laço delimitado pelas linhas 8 e 19, o Algoritmo 6 expande o núcleo até que se atinjam os limites da lista de elementos a alocar ou se, com a inclusão de um novo item, o limite

inferior de lucro do espaço de pesquisa não possa ser aumentado.

Ao final, o Algoritmo 6 invoca o Algoritmo 5 (linha 20) para construir a matriz de alocação correspondente à melhor solução. Como nenhuma solução comprovadamente superior deixou de ser explorada, a solução retornada é garantidamente ótima.

4.2.3 Discussão da eficiência do algoritmo MINKNAP

Esta seção inicialmente discute dois dos principais mecanismos utilizados para reduzir a complexidade do caso médio e, consequentemente, contribuir para a redução do tempo médio de execução do Algoritmo 6. O primeiro mecanismo refere-se à ordenação dos itens a alocar e o segundo refere-se à priorização de itens do núcleo ao se explorar o espaço de pesquisa. Ao final, discute-se a correlação entre o tempo médio de execução do Algoritmo 6 e a natureza da instância específica do Problema 2 abordada. Estas discussões fornecerão insumos para a expectativa de eficiência do algoritmo em casos de uso reais distintos (em tamanho e natureza) daqueles utilizados nos experimentos a serem reportados no Capítulo 5.

No intuito de reduzir a complexidade do caso médio, Pisinger optou por encontrar o item de corte por meio de um ordenamento parcial, através de uma modificação no algoritmo Quicksort que foi originalmente proposta por Balas e Zemel [BAL 80].

Como se sabe, quando o algoritmo Quicksort é instrumentado para ordenação decrescente de um arranjo delimitado pelo intervalo $[f, l]$, ele o particiona recursivamente em dois sub-arranjos delimitados pelos intervalos $[f, i-1]$ e $[i, l]$ de forma que todos os elementos correspondentes ao sub-intervalo $[f, i-1]$ sejam maiores ou iguais a um elemento pivô escolhido, digamos m, e que os elementos correspondentes ao sub-intervalo $[i, l]$ sejam menores do que m, embora os elementos correspondentes a cada sub-intervalo não resultem ordenados entre si.

Quando a eficiência é utilizada como chave de ordenação, Balas e Zemel [BAL 80] notaram que itens delimitados em um intervalo $[f, i-1]$ satisfazendo a condição $\sum_{j=1}^{i-1} w_j \leq C$ podem ser descartados, pois o item de corte certamente não estará entre eles (veja Definição 4.2). Pelo mesmo motivo, os itens delimitados em um intervalo $[i, l]$ satisfazendo a condição $\sum_{j=1}^{i-1} w_j > C$ podem também ser descartados. Essa modificação no algoritmo Quicksort reduz a complexidade da linha 2 do Algoritmo 6 de $O(n^2)$ para $O(n)$, no pior caso.

É então efetuado um processo de redução em duas listas (a primeira contendo os itens com eficiência maior que a do item de corte e a segunda contendo os itens com eficiência menor), utilizando-se o limite superior proposto por Dembo e Hammer [DEM 80]. Esse processo consiste em

testar, através daquele limite superior, se a não-alocação dos itens implicitamente alocados (ver Definição 4.3) é promissora; em caso contrário, os itens podem ser descartados do espaço de soluções, diminuindo-se assim o número de elementos a serem enumerados. Ao final do processo, os itens que não foram descartados resultam completamente ordenados. De maneira similar, testa-se a potencial alocação dos elementos (previamente não-alocados) pertencentes à lista de elementos com eficiência menor que a do item de corte. Segundo Pisinger [PIS 97], este processo pode diminuir em até 80% o tempo de execução médio pois reduz-se o número médio de iterações do laço entre as linhas 8 a 18 do Algoritmo 6, embora a complexidade de pior caso permaneça inalterada.

O mecanismo de priorização de itens do núcleo advém da observação de Balas e Zemel [BAL 80] de que a solução ótima para o Problema 2 pode ser obtida tomando-se a solução de corte como solução inicial e trocando-se um ou mais itens já alocados por um ou mais itens não alocados. Ao se analisar todas as trocas factíveis (respeitando-se a restrição da capacidade da mochila), Balas e Zemel observaram que os itens em torno do item de corte, são os que levam ao maior número de trocas. Pode-se denominar de **frequência de trocas** de um item a relação entre o número de vezes em que ele está presente em uma troca e o número total de trocas efetuadas.

A Figura 4.1 ilustra a frequência média de trocas como função da distância entre a posição do j-ésimo elemento da lista de itens a alocar e a posição b do item de corte. Essa função captura quão frequentemente um item y_j, com eficiência maior do que a do item de corte y_b, deixa de ser alocado na mochila, apesar de sua maior eficiência.

A Figura 4.1 resume o resultado de um experimento reportado por Pisinger [PIS 97], no qual foram resolvidas 1000 instâncias do Problema 2 com $n = 1000$ itens cada, onde os valores de lucros e pesos foram escolhidos aleatoriamente, mas tais que as eficiências resultantes determinassem a posição do item de corte exatamente no meio do intervalo $[1, n]$.

A Figura 4.1 indica que a frequencia de trocas é inversamente proporcional à distância entre um item arbitrário e o item de corte. Em média, 0,34% dos itens tem sua alocação para a mochila modificada [PIS 97].

Esse é o motivo que leva a se priorizar a enumeração a partir de itens pertencentes ao núcleo.

Vamos agora discutir o comportamento do Algoritmo 6 para diferentes instâncias do Problema 2. Dado um limite superior U para os valores das propriedades lucro e peso, Martello e Toth [MAR 90] classificaram as instâncias do Problema Binário da Mochila, de acordo com as distribuição de valores de lucro e peso no intervalo de valores $[1, U]$:

- Classe 1 - Instâncias com propriedades não correlacionadas: p_j e w_j são aleatoriamente distribuídos em $[1, U]$.
- Classe 2 - Instâncias com propriedades fracamente correlacionadas: w_j aleatoriamente distribuído em $[1, U]$ e p_j uniformemente distri-

buído em $[w_j - U/10, w_j + U/10]$ com a restrição de que $p_j > 0$.

- Classe 3 - Instâncias com propriedades fortemente correlacionadas: w_j aleatoriamente distribuído em $[1, U]$ e $p_j = w_j + 10$.

Pisinger [PIS 97] realizou um experimento com o Algoritmo 6 para diferentes tamanhos de problema (n) e diferentes limites superiores para as propriedades lucro e peso (U), analisando os resultados de acordo com a classificação acima. No experimento, esses parâmetros foram assim escolhidos: $U \in \{100, 1000, 10000\}$ e $n \in \{100, 300, 1000, 3000, 10000, 30000, 100000\}$. A capacidade da mochila foi escolhida de maneira que a soma dos pesos fosse igual ao dobro da capacidade da mochila, ou seja, $C = \frac{1}{2}\sum_{j=1}^{n} w_j$, para assegurar um problema não trivial[3].

Os resultados desse experimento revelaram que, em média, as instâncias da Classe 1 foram resolvidas em 0,03s, as instâncias da Classe 2 em 0,04s e as instâncias da Classe 3 em 151s (em uma estação de trabalho modelo HP 9000/740, que utilizava um processador PA-RISC operando a 66MHz).

Assim, a escolha do Algoritmo 6 para o mapeador, baseou-se não somente na existência de mecanismos para acelerar o caso médio, mas também na expectativa de que a instância do Problema 2 aqui abordada (alocação em SPMs) pertença às Classes 1 ou 2. Segundo Pisinger [PIS 97] a eficiência da resolução de instâncias da Classe 1 tem pouca relação com o tamanho do problema. Na Seção 5.4, os casos de uso reais adotados nos experimentos serão observados com respeito à correlação de pesos e lucros.

4.3 Caracterização de elementos de programa

Embora a técnica aqui proposta opere sobre arquivos-objeto relocáveis para fins de re-alocação na SPM, o fluxo de projeto requer um arquivo executável (que pode ser obtido através do processo de linkedição nos arquivos-obeto relocáveis) para fins de caracterização do programa (*profiling*). O arquivo-objeto executável difere do arquivo-objeto relocável em alguns pontos, mas o principal é que no arquivo-objeto executável todos os endereços de variáveis, vetores, métodos, desvios, etc. são endereços finais, ou seja, não são modificados e correspondem aos endereços efetivos de acesso à memória.

Como a técnica proposta opera sobre arquivos-objeto utilizou-se como infra-estrutura a biblioteca BFD, que é parte do conjunto de utilitários binários *GNU Binutils* [BIN 07], para a caracterização dos elementos de programa.

[3]Um problema cuja soma dos pesos w_i fosse menor que a capacidade da mochila seria um problema trivial, pois pouco exigiria do algoritmo.

A biblioteca BFD tem como objetivo prover um conjunto padronizado de métodos para operar em diferentes formatos e arquiteturas de arquivos-objeto. Vários tipos de arquivos-objeto são suportados pela biblioteca que é basicamente dividida em duas partes, como ilustra a Figura 4.2, extraída de [BAL 05]:

- *Frontend*: A interface com o usuário, que permite administrar a memória e várias estruturas de dados canônicas (independentes de arquitetura e formato de arquivo binário). Como a tabela de símbolos, tabela de relocações, seções presentes no arquivo, entre outros. O *frontend* também decide qual *backend* utilizar e quando chamar as rotinas do *backend*.

- *Backend*: Cada *backend* provê um conjunto de chamadas as quais o *frontend* do BFD pode usar para manter sua forma canônica. Um novo formato de arquivo objeto (como, por exemplo: `a.out`, `coff`, `ecoff` e `elf`) pode ser suportado simplesmente criando um novo BFD *backend* e adicionando-o à biblioteca.

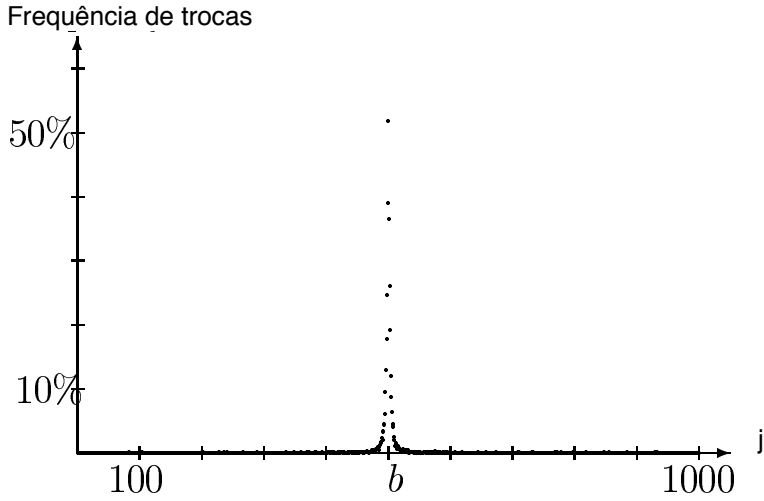

Figura 4.1: Frequência média de trocas como função da distância do item de corte [PIS 97].

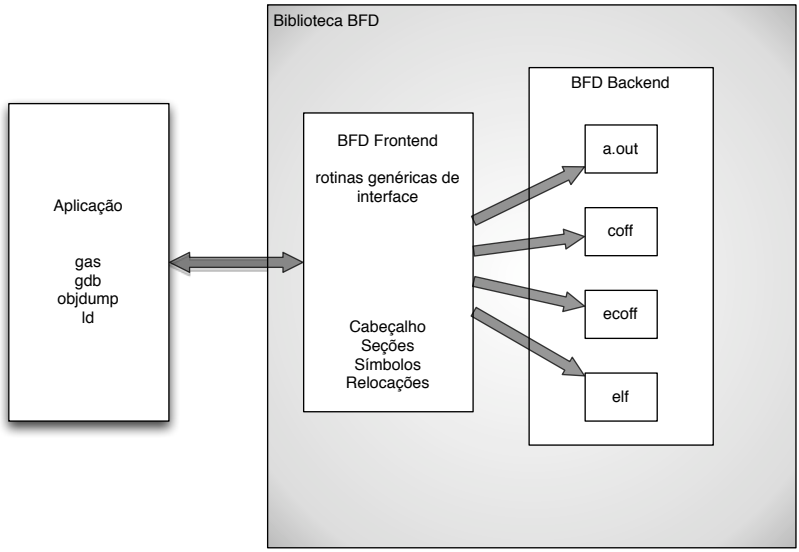

Figura 4.2: Estrutura da biblioteca GNU Binutils (Fonte:[BAL 08])

As funções do *frontend* da biblioteca BFD são utilizadas, por exemplo, para obter os elementos da matriz W (definida na Seção 3.1). Os tamanhos w_i de cada elemento D_i são extraídos através da tabela de símbolos do arquivo-objeto executável, que por sua vez é obtida através da invocação de funções do *frontend* da biblioteca BFD. Como pode ser constatado em [BAL 08], essa biblioteca pode ser redirecionada automaticamente, garantindo assim em princípio a redirecionabilidade do método proposto.

4.4 Caracterização do padrão de acessos

Como infra-estrutura para a caracterização do padrão de acesso são utilizadas três ferramentas:

1) Gerador automático de simuladores de processadores: utilizou-se o gerador disponível no pacote ArchC [RIG 04]. Os simuladores gerados por ArchC foram modificados para interceptar todos os acessos à memória gerados por uma aplicação específica (trace de memória), redirecionando os respectivos endereços para um simulador de hierarquia de memória.

2) Gerador automático de simuladores de hierarquia de memórias: utilizou-se uma ferramenta desenvolvida no Laboratório de Automação e Projeto de Sistemas (LAPS) da UFSC[4], a qual gera automaticamente simuladores de hierarquias de memória que permitem obter o número de ciclos gastos e a energia consumida no subsistema de memória.

3) *Profiler*: utilizou-se um programa para coletar a taxa de faltas associada a cada item. Dado um trace T e um elemento de programa D_i, o *profiler* calcula o número de acessos à memória em T que recaem no intervalo de endereços denotado por a_i, conforme a Equação 4.1.

$$a_i = \sum_{\forall \alpha | \alpha \in T} \delta_\alpha, \quad onde: \delta_\alpha = \begin{cases} 1 & se \quad \alpha_i \leq \alpha < \alpha_i + w_i \\ 0 & senão \end{cases} \tag{4.1}$$

A taxa de faltas (*miss rate*) de cada elemento é calculada utilizando-se a Equação 4.2.

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

$$m_i = \sum_{\forall \alpha | \alpha \in T} (\varphi_\alpha)/a_i, \quad onde: \varphi_\alpha = \begin{cases} 1 & se\ item\ não \\ & encontrado\ na\ cache \\ 0 & se\ item\ encontrado \\ & na\ cache \end{cases} \tag{4.2}$$

4.5 Critérios heurísticos para a função lucro

Ao apresentar a fórmula proposta, será apresentada também a formulação do trabalho correlato mais próximo, para fins de comparação.

O espaço requerido para alocar um elemento de programa D_i para a SPM é o próprio tamanho deste elemento, pois a técnica proposta nesta dissertação não necessita da adição de instruções para alocar código ou dados, ao contrário do método proposto em [ANG 04].

Nesta dissertação são propostas duas fórmulas para o cálculo do lucro (*profit*), as quais serão comparadas com a fórmula proposta em [ANG 04]. A primeira fórmula diz respeito à otimização voltada ao menor consumo de energia. Como o método aqui proposto não gera expansão de código, o cômputo do lucro em se alocar o elemento na SPM é a energia economizada por acesso vezes o número de acessos àquele elemento.

$$p_i = a_i \times (E_i - E_{SPM}), \quad onde: E_i = E_{cache} + m_i \times E_{MM} \tag{4.3}$$

A segunda fórmula é relativa à otimização voltada ao desempenho do subsistema de memória (mínimo número de ciclos). O cômputo do lucro em se alocar o elemento D_i em SPM é igual ao número de ciclos economizados por acesso vezes o número total de acessos àquele elemento:

$$p_i = a_i \times (\lambda_i - \lambda_{SPM}), \quad onde: \lambda_i = \lambda_{cache} + m_i \times \lambda_{MM} \tag{4.4}$$

A fórmula proposta em [ANG 04], traduzida para a notação desta dissertação e eliminando-se os termos relativos à expansão de código, pode ser assim escrita:

$$p_i = a_i \times (E_i - E_{SPM}), onde: E_i = E_{cache} + m_{cache} \times E_{MM} \tag{4.5}$$

O lucro em se alocar o elemento D_i é a energia economizada por acesso vezes o número de acessos àquele elemento. Para todos os elementos D_i o valor de E_i é constante, pois baseia-se na taxa de faltas global da memória cache.

Repare que na fórmula de [ANG 04] a taxa de faltas (*miss rate*) utilizada pelos autores é a taxa global da aplicação e, na equação proposta nesta dissertação, a taxa de faltas é local para cada elemento de programa D_i.

Para ilustrar uma falha na equação de lucro proposta por Angio-

lini, suponha dois procedimentos pertencentes ao mesmo programa com o mesmo número de acessos (a_i) e de mesmo tamanho: o primeiro baseado em desvios condicionais (com alta taxa de faltas na cache e baixa localidade espacial); o segundo baseado em um laço (com baixa taxa de faltas na cache e com alta localidade espacial). O primeiro procedimento terá um desempenho fraco na cache, enquanto o segundo terá um melhor desempenho. Pela fórmula proposta em [ANG 04] será atribuído o mesmo lucro (p_i) para os dois procedimentos. Isto pode ocasionar em uma alocação sub-ótima para a SPM (devido a taxa de faltas utilizada naquele método ser global do programa).

4.6 Relocação e linkedição

Vamos ilustrar através de um exemplo as idéias básicas para suportar a otimização de bibliotecas mantendo seus elementos no espaço de otimização, sem se deixar limitar por um *patching* computacionalmente ineficiente. Isto é obtido ao se utilizar arquivos-objeto relocáveis, ao invés de se utilizar somente o arquivo executável.

Sejam as estruturas de dois arquivos relocáveis ilustradas esquematicamente na Figura 4.3. Os arquivos correspondem aos procedimentos `main` (parte superior) e `getK` (parte inferior). A primeira coluna ilustra as seções do arquivo. Por simplicidade, o código é representado em linguagem *assembly* do processador *MIPS*. Repare que os segmentos de dados contêm referências simbólicas para variáveis estáticas (N e K). A tabela de símbolos contém a relação entre os elementos de programa e seus respectivos rótulos. Note que uma única referência externa continua indefinida (`getK`). Esta referência será resolvida mais tarde pelo linkeditor (todas as referências definidas foram omitidas por simplicidade).

O que diferencia um arquivo-objeto relocável é basicamente a seção que contém as informações de relocação. Nessa seção, cada instrução que necessita de relocação é identificada pelo seu endereço no segmento de código e sua depêndencia a uma referência simbólica (um elemento de programa). Como os ajustes necessários a cada relocação dependem do tipo de instrução e de seu formato, uma ação de relocação é associada a cada instrução que necessita de *patching* (uma ação pode ser vista como uma regra para converter uma referência simbólica em uma codificação binária).

Repare que a instrução no endereço 0x8 no arquivo superior (na qual o endereço-alvo é inicializado com zero) necessita de *patching*. A partir da respectiva relocação, podemos dizer que este endereço deve ser substituído pelo endereço de `getK` (quando este for resolvido pelo linkeditor). Note também que as instruções nos endereços 0xC e 0x10 dependem da variável `N` (porém, diferentes ações de relocação são requeridas). Em resumo, a relação entre endereços-alvo e elementos de programa são explicitamente mantidas no arquivo-objeto através de referências simbólicas.

Na Figura 4.3 (parte superior), observe que as instruções nos ende-

Cabeçalho	Nome	main	
	...	...	...
Segmento de texto	Endereço	Instrução	
	0	addiu sp,sp, -24	
	4	sw ra, 16(sp)	
	8	jal 0	
	C	lui at, 0x0	
	10	sw v0,0(at)	
	14	lw ra, 16(sp)	
	18	jr ra	
	1C	addiu sp, sp, 24	
Segmento de dados	Endereço	Valor	Referência
	0	0x0	N
Relocações	Endereço	Ação	Dependência
	8	R_MIPS_26	getK
	C	R_MIPS_HI16	N
	10	R_MIPS_LO16	N
Tabela de símbolos	Rótulo	Endereço	
	getK	–	

Cabeçalho	Nome	getK	
	...	...	...
Segmento de texto	Endereço	Instrução	
	0	lui v0, 0x0	
	4	lw v0, 0(v0)	
	8	jr ra	
	C	nop	
Segmento de dados	Endereço	Valor	Referência
	0	0x5	K
Relocações	Endereço	Ação	Dependência
	0	R_MIPS_HI16	K
	4	R_MIPS_LO16	K
Tabela de símbolos	Rótulo	Endereço	
	–	–	

Figura 4.3: Representação esquemática de arquivos-objeto relocáveis

reços 0x4 e 0x14 não necessitam de relocação apesar de serem instruções de acesso à memória, isto se deve ao fato de que as variáveis por elas referenciadas estão alocadas em pilha. A técnica aqui proposta não foi projetada para tratar tais elementos que residem em áreas dinâmicas de memória (pilha e *heap*), pois abordagens dinâmicas [STE 02b] [STE 02a] [DOM 05] [UDA 06] [CHO 07] [MCI 08] são mais adequadas para esses tipos de dados. Desta maneira, podemos focar no tratamento de procedimentos, variáveis estáticas (escalares ou não) e constantes definidas em arquivos binários. Assim, elementos de programa encapsulados em bibliotecas são preservados no espaço de endereçamento alocável em SPM (sem a necessidade de hardware dedicado como em [CHO 07]).

Arquivos executáveis estáticos exibem uma estrutura similar à ilustrada na Figura 4.3, não possuindo referências simbólicas nem as informações de relocação que caracterizam os arquivos-objeto relocáveis. Por simplicidade, a partir de agora vamos nos referir a arquivos-objeto relocáveis meramente como arquivos-objeto.

Quando um elemento de programa é mapeado para a SPM, o arquivo-objeto no qual este foi declarado deve sofrer a ação de *patching*. A idéia-chave é criar em cada arquivo-objeto uma seção dedicada (`.spm`) e editar as tabelas de símbolos de cada arquivo-objeto de forma que a referência à antiga seção (`.data`, `.text` ou `.bss`) seja substituída por uma referência à nova seção (`.spm`). Obviamente, existe a necessidade de se ajustar as referências a cada elemento mapeado em SPM para sua nova seção. Como resultado, novos arquivos-objeto são gerados nos quais elementos mapeados para SPM são anotados com ações de relocação. Quando os arquivos-objeto modificados são passados para o linkeditor, todas as seções `.spm` são concatenadas e são fisicamente alocadas para o espaço de endereçamento da SPM (definido previamente por um *script* do linkeditor). As ações de *patching* são diferentes dependendo do tipo do elemento de programa (variável global ou procedimento) alocado para a SPM.

Vamos começar com a alocação de procedimentos. Quando um procedimento, digamos D_i, é mapeado para a SPM, os alvos de seus desvios devem ser ajustados para fazer referência à nova seção de D_i, neste caso `.spm`. Em arquivos-objeto, os alvos de desvios são anotados como referências simbólicas que são relativas ao início da seção de código à qual esse desvio pertence (por exemplo, `.text`). Em arquivos executáveis, ocorre o oposto, os alvos de desvios são codificados como endereços absolutos. Digamos que $|J_i|$ represente o número de desvios em um procedimento D_i e $|J|$ represente o número total de desvios mapeados para SPM. O número de operações de *patching* é $\sum_i |J_i|$ para arquivos-objetos e $|J|$ para arquivos executáveis. Assim podemos afirmar que a eficiência de *patching* é a mesma para arquivos-objeto e arquivos executáveis ($\sum_i |J_i| = |J|$), desprezando-se um pequeno acréscimo de complexidade de se efetuar *patching* em endereços absolutos. É por este motivo que o tempo de *patching* reportado em [ANG 04] para arquivos executáveis é relativamente pequeno, pois aquele método somente aloca código em SPM

(mas não dados).

Por outro lado, o suporte de alocação de dados para SPM a partir de arquivos executáveis levaria a um *patching* computacionalmente menos eficiente. Ao se escolher arquivos-objeto como entrada, podemos habilitar o mapeamento de dados para a SPM com um custo computacional marginal. A seguir serão resumidos os principais motivos que levam o *patching* de arquivos-objeto a ser mais eficiente quando a alocação de dados faz parte da otimização.

Quando uma variável global (escalar ou não), digamos D_i, é mapeada para a SPM, não somente suas referências devem sofrer *patching*, mas o conteúdo da sua seção de origem deve também ser deslocado para evitar um desperdício de espaço de armazenamento.

Vamos analisar primeiramente o *patching* das referências. Apenas a tabela de símbolos onde D_i foi declarado necessita de *patching* (como as referências a D_i em outros arquivos são externas, estas são tratadas mais tarde pelo linkeditor). Se fossem utilizados arquivos-executáveis como em [ANG 04], todas as referências a D_i necessitariam de *patching*.

Agora consideremos o deslocamento de conteúdo. Cada elemento de dados mapeado para a SPM, digamos D_i, sofre *patching* por vez. Seja $succ(D_i) = D_j | \alpha_j > \alpha_i$. No pior caso, o número de elementos que necessitam deslocamento é $|succ(D_i)|$ e todas as referências para cada D_j necessitam de *patching*, bem como todas as referências para todos os elementos D_i. Suponha que M é o número total de elementos de dados mapeados, e que N seja o número total de elementos de dados, e que R_n é o número de referências a um elemento D_n no código da aplicação. Desta maneira, o número de referências que necessitam de *patching* no arquivo executável é $M \times (\sum_{i=1}^{N} R_n)$ no pior caso. No arquivo-objeto, por sua vez, este número é reduzido para $M \times N$, pois somente as referências na tabela de símbolos necessitam de *patching* ($R_n = 1$).

Em resumo, a operação de *patching* em arquivos-objeto é muito mais fácil e mais eficiente que em arquivos executáveis, pois em arquivos-objeto as referências a serem relocadas (sendo simbólicas) não estão ainda codificadas como endereços.

Capítulo 5

Validação experimental e resultados

Este capítulo inicia descrevendo a infra-estrutura experimental utilizada para validar o protótipo sob os diferentes cenários de configuração de memória e para diferentes aplicações reais. Em seguida, analisam-se o impacto potencial de se alocar bibliotecas e o impacto relativo entre alocação de código e dados. À luz dessa análise preliminar do impacto potencial, são reportados e interpretados os resultados experimentais observados para avaliar a eficácia e a eficiência da técnica proposta. Ao final, comparam-se os resultados obtidos nesta dissertação com outros propostos na literatura.

5.1 Configuração experimental

Para se fazer a experimentação do protótipo foram utilizados treze programas extraídos do conjunto de *benchmarks* MiBench [GUT 01]. Infelizmente, não foi possível utilizar todos os programas do referido conjunto de *benchmarks* devido a não se conseguir compilar alguns dos programas com a infra-estrutura utilizada nos experimentos. Para exercitar os programas, foram escolhidos os estímulos chamados de *large inputs*, que acompanham aquele conjunto de *benchmarks*.

Para produzir arquivos-objeto relocáveis, foi utilizado o compilador `gcc` (versão 3.3.1), sob a opção de otimização `-Os` (esta opção previne otimizações que aumentam o tamanho de código, uma escolha típica para sistemas embarcados). Para gerar arquivos executáveis, foi utilizado o linkeditor `ld` obtido do pacote GNU Binutils [BIN 07]. O ambiente de simulação utilizado na execução do *profiling* consiste de um modelo executável do processador MIPS (um simulador do conjunto de instruções gerado pela ADL ArchC [RIG 04]) e um modelo de memória parametrizável prévalidado, que implementa a representação de memória principal proposta em [PAN 97b]. Os parâmetros dependentes de tecnologia foram obtidos através do modelo físico de memórias CACTI [THO 08]. Os experimentos foram executados em um processador Intel Xeon E5430 (*quad-core*), rodando à frequência de 2,66GHz, com 4GB de memória principal, sob o sistema operacional Debian GNU/Linux (kernel 2.6.26).

A Tabela 5.1 enumera as diferentes configurações para o subsistema de memória utilizando cache unificada (U-cache), caches separadas

Tabela 5.1: Configuração do subsistema de memória utilizado nos experimentos

Configuração	Capacidade (kB)			
	$C_{U-cache}$	$C_{I-cache}$	$C_{D-cache}$	C_{SPM}
A	0	1	1	0
B	0	4	1	0
C	0	1	1	1
D	1	0	0	1
E	0	1	1	4
F	0	4	1	4
G	1	0	0	4
H	4	0	0	4

de instruções (I-cache) e dados (D-cache), bem como a SPM. Essas configurações são as mesmas utilizadas em [ANG 04] e foram escolhidas para fins de comparação com aquele trabalho.

As caches de instruções foram configuradas com mapeamento direto e tamanho de bloco de quatro palavras (16 bytes), enquanto para as caches de dados e para as caches unificadas adotou-se um mapeamento associativo de quatro vias (*4-way*) com um tamanho de bloco de quatro palavras (16 bytes). Esses parâmetros são os mesmos utilizados em [ANG 04]. No trabalho aqui proposto, utiliza-se um nodo tecnológico de 90nm; infelizmente, em [ANG 04] esta informação não é fornecida, o que limita a comparação de valores absolutos.

Todos os valores de energia, ciclos e área reportados referem-se apenas ao subsistema de memória (e não a valores totais no sistema).

5.2 Impacto da inclusão de dados e de código no espaço de otimização

5.2.1 Caracterização do impacto potencial

Para as configurações de memória descritas na Tabela 5.1, a Tabela 5.2 mostra a contribuição relativa, em termos de consumo de energia e de número de ciclos, para cada segmento de memória, para a configuração-base (A). Esses valores são expressos em porcentagem para código (CO), dados estáticos (SD) e dados dinâmicos (DD). Como estes valores foram obtidos antes da introdução de SPMs, eles nos permitem quantificar o impacto potencial em se alocar código e dados estáticos para SPMs. Os valores foram medidos para todos elementos de programa acessados, independentemente de estarem encapsulados em bibliotecas ou não. Para facilitar a vizualização dos dados da Tabela 5.2, seus dados estão plotados nas Figuras 5.1 e 5.2.

Em média, 61% da energia e 77% do número de ciclos são consumidos quando se acessam dados estáticos ou código e 39% da energia e 23% do número de ciclos quando se acessam dados dinâmicos (pilha e *heap*). Essas porcentagens dão clara evidência de que, mesmo que uma técnica não aborde a alocação em SPM de dados dinâmicos, seu uso prático se justifica em face do significativo potencial de otimização.

Tabela 5.2: Caracterização da contribuição ao consumo de energia e número de ciclos

Programa	CO		SD		DD	
	Energia	Ciclos	Energia	Ciclos	Energia	Ciclos
qsort	**59,50**	**74,29**	**1,75**	**1,53**	38,75	24,18
susan	**52,97**	**85,46**	**0,03**	**0,00**	47,00	14,54
cjpeg	**31,73**	**66,27**	**4,34**	**2,48**	63,93	31,25
lame	**59,27**	**64,60**	**0,39**	**0,20**	40,34	35,20
dijkstra	**22,54**	**67,33**	**46,4**	**19,88**	31,06	12,79
blowfish	**38,52**	**60,23**	**0,02**	**0,01**	61,45	39,76
rijndael	**51,44**	**62,87**	**14,69**	**9,50**	33,87	27,63
sha	**8,27**	**71,18**	**67,79**	**21,22**	23,94	7,60
adpcm_code	**18,95**	**91,34**	**80,52**	**7,54**	0,54	1,12
adpcm_decode	**10,07**	**86,92**	**89,54**	**11,64**	0,38	1,44
crc32	**10,21**	**59,45**	**29,58**	**3,50**	60,21	37,05
FFT	**56,15**	**58,88**	**0,09**	**0,05**	43,76	41,07
GSM	**27,39**	**78,37**	**6,67**	**2,23**	65,94	19,40
Média	**34,39**	**71,32**	**26,29**	**6,14**	39,32	22,54

Podemos agora definir a **margem de otimização para energia** do método proposto nesta dissertação como a contribuição relativa ao consumo de energia quando acessamos código e dados estáticos (a soma das contribuições relativas SD e CO, destacadas em negrito na Tabela 5.2). Espera-se que a técnica aqui proposta tenha maior impacto para programas com maior margem de otimização, tais como: `adpcm_code` (99%), `adpcm_decode` (99%) e `sha` (76%). Para programas com menor margem de otimização, tais como `blowfish` (38%) e `GSM` (34%) espera-se que a técnica aqui proposta tenha um desempenho inferior. Em média, para o conjunto de programas escolhido, o método proposto tem uma margem de otimização de 61% ao se minimizar o consumo de energia.

A **margem de otimização para número de ciclos** gastos é definida de forma similar. Em média, para o conjunto de programas escolhido, o método proposto tem uma margem de otimização de 77% ao se minimizar o número de ciclos.

Embora o potencial de impacto da alocação de elementos de programa em SPM varie entre programas distintos (devido aos diferentes padrões de acesso), a eficácia da técnica proposta deve ser avaliada como função do aproveitamento da margem de otimização *disponível*. Essa avaliação é objeto das próximas subseções.

5.2.2 Minimização do consumo de energia

As Tabelas 5.3 e 5.4 mostram os resultados obtidos usando a função-lucro da Equação 4.3, ou seja, minimizando o consumo de energia (o que, algumas vezes, pode levar a um maior número de ciclos). As tabelas reportam os valores relativos obtidos para cada configuração, normalizados em relação à configuração A, em termos de número de ciclos (AT), energia (EN) e área (AR). Os resultados mostrados em ambas as tabelas são referentes somente ao subsistema de memória e foram obtidos em dois diferentes cenários: no primeiro cenário (Tabela 5.3), somente elementos

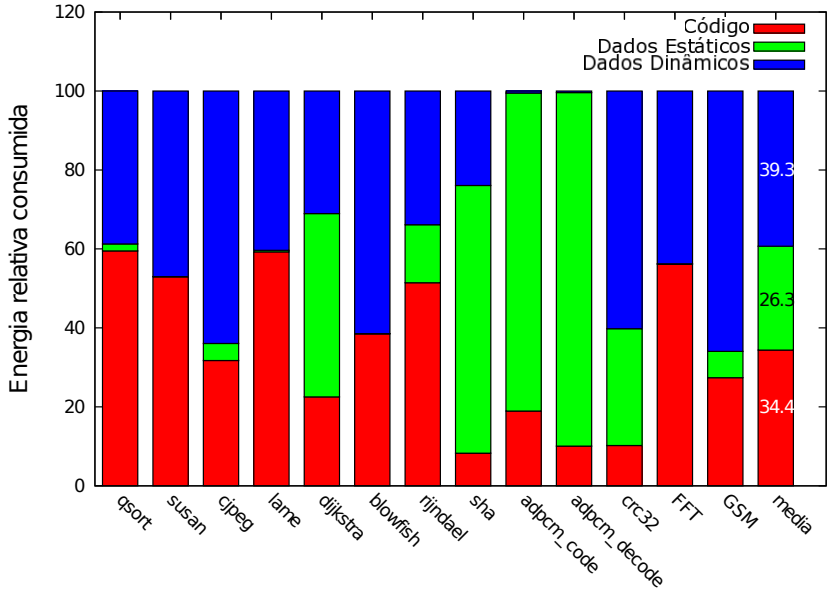

Figura 5.1: Caracterização da contribuição ao consumo de energia

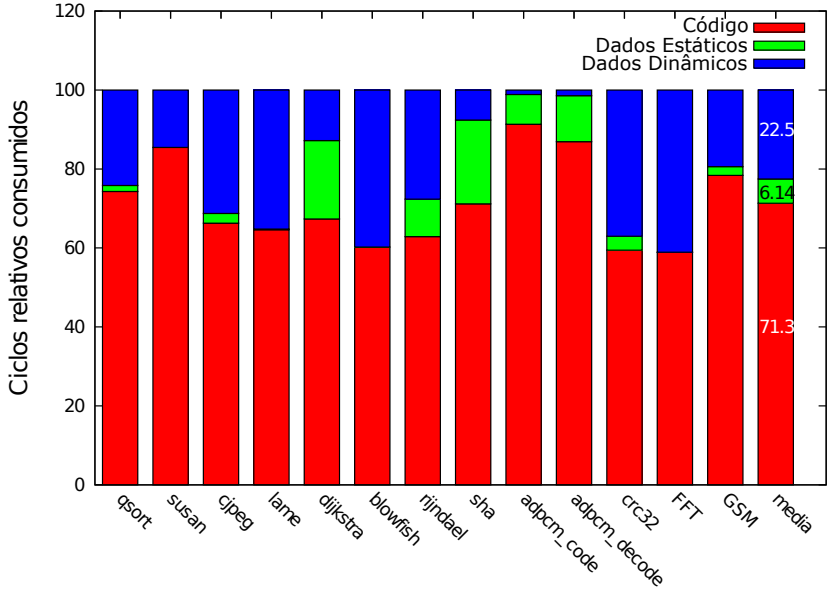

Figura 5.2: Caracterização da contribuição ao número de ciclos

Tabela 5.3: Resultados da minimização do consumo de energia: somente código com inclusão de bibliotecas (Cenário 1)

Programa	Métrica	B	C	D	E	F	G	H
qsort	AT	0,88	0,84	0,87	0,66	0,57	0,70	0,58
	EN	**0,66**	**0,85**	**0,98**	**0,71**	**0,48**	**0,85**	**0,51**
susan	AT	1,00	0,99	1,01	0,17	0,17	0,17	0,17
	EN	**1,14**	**0,96**	**2,01**	**0,60**	**0,57**	**0,62**	**0,41**
cjpeg	AT	0,94	0,81	0,86	0,44	0,43	0,45	0,42
	EN	**0,79**	**0,78**	**0,96**	**0,71**	**0,69**	**0,78**	**0,66**
lame	AT	0,87	0,82	0,85	0,62	0,52	0,65	0,52
	EN	**0,65**	**0,83**	**0,94**	**0,70**	**0,45**	**0,81**	**0,44**
dijkstra	AT	0,96	0,49	0,52	0,37	0,35	0,37	0,30
	EN	**0,85**	**0,88**	**0,66**	**0,84**	**0,78**	**0,87**	**0,54**
blowfish	AT	0,86	0,80	0,86	0,51	0,51	0,51	0,46
	EN	**0,65**	**0,91**	**1,00**	**0,62**	**0,62**	**0,63**	**0,51**
rijndael	AT	0,92	0,93	0,94	0,84	0,73	0,85	0,83
	EN	**0,85**	**0,96**	**0,98**	**0,91**	**0,68**	**0,93**	**0,90**
sha	AT	0,99	0,38	0,38	0,29	0,29	0,29	0,28
	EN	**0,99**	**0,92**	**0,95**	**0,90**	**0,90**	**0,91**	**0,88**
adpcm_code	AT	1,00	0,09	0,09	0,09	0,09	0,09	0,08
	EN	**1,19**	**0,81**	**0,82**	**0,81**	**0,81**	**0,82**	**0,62**
adpcm_decode	AT	1,00	0,14	0,14	0,14	0,14	0,14	0,13
	EN	**1,10**	**0,90**	**0,90**	**0,90**	**0,90**	**0,90**	**0,77**
crc32	AT	1,00	0,17	0,18	0,14	0,14	0,15	0,14
	EN	**1,02**	**0,37**	**0,39**	**0,32**	**0,32**	**0,34**	**0,30**
FFT	AT	0,88	0,75	0,78	0,44	0,43	0,44	0,43
	EN	**0,66**	**0,74**	**0,85**	**0,51**	**0,48**	**0,53**	**0,48**
GSM	AT	0,98	0,45	0,45	0,33	0,33	0,33	0,32
	EN	**0,92**	**0,84**	**0,90**	**0,43**	**0,43**	**0,47**	**0,43**
média	AT	0,94	0,59	0,61	0,39	0,36	0,40	0,36
	EN	**0,88**	**0,83**	**0,95**	**0,69**	**0,62**	**0,73**	**0,57**
–	AR	2,01	1,32	0,85	2,15	3,16	1,67	2,64

de código foram levados em consideração pelo algoritmo de otimização; no segundo cenário (Tabela 5.4), foram também levados em consideração elementos de dados estáticos.

Para facilitar a interpretação da grande quantidade de informações nas Tabelas 5.3 e 5.4, vamos comparar o impacto da técnica para configurações com a mesma quantidade de cache, mas com diferentes tamanhos de SPM. Isso permite avaliar o impacto da inclusão de SPM no consumo de energia da hierarquia de memória. Por isso, vamos nos concentrar nas configurações C e E, que possuem a mesma quantidade de cache da configuração-base, mas possuem também SPMs de 1KB e 4KB, respectivamente.

A Tabela 5.5 resume o melhor e o pior resultados obtidos, bem como a média obtida no conjunto de programas, para as configurações C e E, em ambos os cenários, comparando-os com as respectivas margens de otimização para energia.

Note que, para o programa `blowfish`, apenas 23% da margem de otimização disponível foi aproveitada na configuração C em ambos os cenários. O fato de se ter obtido a mesma economia em ambos os cenários deve-se ao fato de que a energia relativa consumida em dados estáticos é praticamente nula (ver Tabela 5.2).

Tabela 5.4: Resultados da minimização do consumo de energia: código e dados com inclusão de bibliotecas (Cenário 2)

Programa	Métrica	B	C	D	E	F	G	H
qsort	AT	0,88	0,84	0,88	0,65	0,58	0,69	0,58
	EN	**0,66**	**0,85**	**0,97**	**0,69**	**0,48**	**0,83**	**0,50**
susan	AT	1,00	0,99	1,01	0,17	0,17	0,17	0,17
	EN	**1,14**	**0,96**	**2,01**	**0,60**	**0,57**	**0,62**	**0,41**
cjpeg	AT	0,94	0,82	0,86	0,42	0,41	0,43	0,40
	EN	**0,79**	**0,74**	**0,90**	**0,67**	**0,65**	**0,72**	**0,62**
lame	AT	0,87	0,82	0,85	0,62	0,52	0,65	0,52
	EN	**0,65**	**0,83**	**0,94**	**0,70**	**0,45**	**0,81**	**0,44**
dijkstra	AT	0,96	0,80	0,83	0,20	0,18	0,21	0,15
	EN	**0,85**	**0,51**	**0,66**	**0,42**	**0,36**	**0,46**	**0,20**
blowfish	AT	0,86	0,80	0,86	0,51	0,51	0,51	0,46
	EN	**0,65**	**0,91**	**1,00**	**0,62**	**0,62**	**0,63**	**0,51**
rijndael	AT	0,92	0,93	0,94	0,91	0,83	0,91	0,91
	EN	**0,85**	**0,96**	**0,98**	**0,86**	**0,65**	**0,88**	**0,87**
sha	AT	0,99	0,74	0,78	0,13	0,13	0,13	0,13
	EN	**0,99**	**0,49**	**0,73**	**0,45**	**0,45**	**0,46**	**0,44**
adpcm_code	AT	1,00	0,06	0,06	0,02	0,02	0,02	0,02
	EN	**1,19**	**0,18**	**0,19**	**0,01**	**0,01**	**0,01**	**0,01**
adpcm_decode	AT	1,00	0,12	0,12	0,02	0,02	0,02	0,02
	EN	**1,10**	**0,87**	**0,87**	**0,01**	**0,01**	**0,01**	**0,01**
crc32	AT	1,00	0,17	0,18	0,12	0,12	0,12	0,12
	EN	**1,02**	**0,37**	**0,39**	**0,30**	**0,30**	**0,30**	**0,30**
FFT	AT	0,88	0,75	0,78	0,44	0,43	0,45	0,43
	EN	**0,66**	**0,74**	**0,85**	**0,51**	**0,48**	**0,54**	**0,48**
GSM	AT	0,98	0,45	0,45	0,33	0,33	0,33	0,32
	EN	**0,92**	**0,84**	**0,90**	**0,43**	**0,43**	**0,47**	**0,43**
média	AT	0,94	0,64	0,66	0,35	0,33	0,36	0,32
	EN	**0,88**	**0,71**	**0,88**	**0,48**	**0,48**	**0,52**	**0,40**
–	AR	2,01	1,32	0,85	2,15	3,16	1,67	2,64

Tabela 5.5: Impacto da minimização de energia para as configurações C e E

Caso	Programa	Cenário 1: somente código			Cenário 2: código+dados		
		Margem	Configurações		Margem	Configurações	
			C	E		C	E
Pior	blowfish	39%	9%	38%	39%	9%	38%
Melhor	adpcm_decode	10%	10%	10%	99%	13%	99%
Média	–	34%	17%	31%	61%	29%	52%

Por outro lado, observe que, para o programa `adpcm_decode`, toda a margem de otimização foi aproveitada no Cenário 1 em ambas as configurações. Entretanto, no Cenário 2, o aproveitamento completo da margem de otimização só ocorreu na configuração E. A diferença de impacto observada no Cenário 2 explica-se pelo fato de aquele programa utilizar uma estrutura de dados frequentemente acessada para a decodificação de áudio, a qual é maior que 1KB e menor que 4KB, ou seja, ela não pode ser alocada em SPM na configuração C (mas apenas na configuração E). Por isso, na configuração C, apenas 13% da margem de otimização foi aproveitada.

Em média, cerca de metade da margem de otimização disponível foi aproveitada na configuração C e cerca de 85% na configuração E, não havendo diferença de impacto sensível entre os cenários.

Agora vamos considerar todas as configurações que possuem SPM (C a H), mesmo sem se conservar o mesmo tamanho da cache da configuração-base. Nesse universo, as energias relativas médias dos Cenários 1 (Tabela 5.3) e 2 (Tabela 5.4) são 0,73 e 0,58, respectivamente. Portanto, do Cenário 1 para o Cenário 2, observou-se um fator de redução de energia de 0,79, em média, o que corresponde a uma economia extra de 21% (em relação ao Cenário 1).

Fatores de redução de energia de 0,33 (67% de economia extra) e 0,09 (91% de economia extra) foram obtidos para os programas `adpcm_decode` e `adpcm_code`, respectivamente. O maior impacto da otimização sobre esses dois programas deve-se à sua maior contribuição de energia no acesso a dados estáticos, quando comparada à dos demais programas.

Assim, pode-se concluir que, para o conjunto de programas utilizado, embora o impacto de se incluir dados no espaço de otimização seja pequeno em média, podem haver casos importantes em que esse impacto é substancial (como em `adpcm_decode` e `adpcm_code`), pois a margem de otimização pode aumentar consideravelmente diante de estruturas de dados estaticamente alocadas e frequentemente acessadas.

Apesar de seu relativamente baixo impacto potencial frente ao código (contribuições médias de 26% em energia e de 6% em número de ciclos, conforme Tabela 5.2) dados estáticos não devem ser descartados do espaço de otimização. O argumento de que a otimização de dados é melhor tratado por *dynamic prefetching* [1] [ANG 04] não é necessariamente verdadeiro para dados estáticos (seus endereços são definidos em tempo de compilação-ligação), embora sejam verdadeiros para métodos que alocam dados da pilha em SPM (e.g.: [AVI 02] e [VER 04]) e para métodos que alocam dados da *heap* (e.g.: [MCI 08] e [DOM 05]).

Embora nem sempre a minimização de energia correlate com a melhoria de desempenho, a aplicação da técnica proposta resultou, em média, numa redução do número de ciclos gastos. Por exemplo, para a configuração C, as economias resultantes no número de ciclos gastos foram de 41% (Tabela 5.3) e 36% (Tabela 5.4). Para a configuração E, essas economias foram 61% e 65%, respectivamente.

Vamos nos concentrar agora na última linha das Tabelas 5.3 e 5.4, que mostra a área relativa (AR) em relação à configuração A, para cada configuração (B a H). Note que, para implementar a configuração C, foi necessário aumentar a área de um fator de 1,32 em relação à configuração-base para se obter reduções médias de 0,83 (Cenário 1) e 0,71 (Cenário 2) no consumo de energia. Para a configuração E, a área precisou ser aumentada de um fator de 2,15 para se obter reduções de energia de 0,69 e 0,48, respectivamente. Observe que a configuração D é a única que reduz a área ocupada em relação à configuração-base. Na configuração D as caches independentes de instruções e dados (1KB cada) foram substituídas por uma

[1] A alocação em SPM é ajustada dinamicamente através da inserção de funções que copiam, em tempo de execução, os elementos de programa entre a SPM e a memória externa.

Tabela 5.6: Resultados da minimização do número de ciclos: somente código com inclusão de bibliotecas (Cenário 1)

Programa	Métrica	B	C	D	E	F	G	H
qsort	AT	**0,88**	**0,81**	**0,84**	**0,64**	**0,56**	**0,65**	**0,56**
	EN	0,66	0,86	0,96	0,74	0,52	0,78	0,53
susan	AT	**1,00**	**0,99**	**1,01**	**0,17**	**0,17**	**0,17**	**0,17**
	EN	1,14	0,96	2,01	0,61	0,57	0,63	0,41
cjpeg	AT	**0,94**	**0,74**	**0,79**	**0,44**	**0,43**	**0,46**	**0,42**
	EN	0,79	0,90	1,07	0,72	0,69	0,79	0,66
lame	AT	**0,87**	**0,92**	**0,97**	**0,65**	**0,58**	**0,67**	**0,58**
	EN	0,65	0,90	1,03	0,64	0,45	0,70	0,44
dijkstra	AT	**0,96**	**0,46**	**0,43**	**0,37**	**0,35**	**0,37**	**0,30**
	EN	0,85	1,07	0,99	0,84	0,78	0,87	0,54
blowfish	AT	**0,86**	**0,80**	**0,86**	**0,51**	**0,51**	**0,51**	**0,46**
	EN	0,65	0,91	1,06	0,62	0,62	0,63	0,51
rijndael	AT	**0,92**	**0,89**	**0,90**	**0,84**	**0,73**	**0,85**	**0,83**
	EN	0,85	0,98	0,98	0,91	0,68	0,93	0,90
sha	AT	**0,99**	**0,38**	**0,38**	**0,29**	**0,29**	**0,29**	**0,28**
	EN	0,99	0,92	0,95	0,92	0,92	0,92	0,88
adpcm_code	AT	**1,00**	**0,09**	**0,09**	**0,09**	**0,09**	**0,09**	**0,08**
	EN	1,19	0,81	0,82	0,81	0,81	0,82	0,62
adpcm_decode	AT	**1,00**	**0,14**	**0,14**	**0,14**	**0,14**	**0,14**	**0,13**
	EN	1,10	0,90	0,90	0,90	0,90	0,90	0,77
crc32	AT	**1,00**	**0,41**	**0,41**	**0,41**	**0,41**	**0,41**	**0,39**
	EN	1,02	0,97	0,97	0,97	0,97	0,97	0,90
FFT	AT	**0,88**	**0,94**	**0,99**	**0,93**	**0,79**	**0,98**	**0,78**
	EN	0,66	0,91	1,08	0,90	0,52	1,08	0,51
GSM	AT	**0,98**	**0,33**	**0,33**	**0,21**	**0,20**	**0,21**	**0,20**
	EN	0,92	0,84	0,89	0,43	0,41	0,47	0,41
média	AT	**0,94**	**0,63**	**0,65**	**0,45**	**0,41**	**0,46**	**0,41**
	EN	0,88	0,92	1,05	0,77	0,68	0,81	0,62
–	AR	2,01	1,32	0,85	2,15	3,16	1,67	2,64

cache unificada de 1KB, reduzindo-se assim a quantidade de cache pela metade. Em contrapartida, esta configuração foi dotada de um SPM de 1KB. Embora uma cache unificada no primeiro nível da hierarquia de memória possa causar uma deteriorização no desempenho do sistema devido aos possíveis conflitos estruturais no processador, ela foi utilizada como exemplo devido a sua simplicidade. Como resultado, a área do subsistema de memória foi diminuída de um fator 0,85 para se obter reduções médias de energia de 0,95 e 0,88 nos Cenários 1 (Tabela 5.3) e 2 (Tabela 5.4), respectivamente. Isso é uma forte evidência de que a técnica proposta pode obter economias de energia mesmo utilizando menos área do que um sistema composto somente por caches (como é o caso da configuração-base).

5.2.3 Minimização do número de ciclos gastos

As Tabelas 5.6 e 5.7 reportam os resultados obtidos usando a função-lucro da Equação 4.4, ou seja, minimizando o número de ciclos gastos na hierarquia de memória (o que algumas vezes pode levar a um maior consumo de energia). A estrutura dessa tabela é idêntica às das Tabelas 5.3 e 5.4, que já foram anteriormente descritas.

De forma similar à da seção anterior, vamos primeiramente compa-

Tabela 5.7: Resultados da minimização do número de ciclos: código e dados com inclusão de bibliotecas (Cenário 2)

Programa	Métrica	B	C	D	E	F	G	H
qsort	AT	**0,88**	**0,81**	**0,84**	**0,64**	**0,55**	**0,65**	**0,56**
	EN	0,66	0,86	0,94	0,74	0,52	0,79	0,54
susan	AT	**1,00**	**0,99**	**1,01**	**0,17**	**0,17**	**0,17**	**0,17**
	EN	1,14	0,96	2,01	0,61	0,57	0,63	0,41
cjpeg	AT	**0,94**	**0,74**	**0,77**	**0,42**	**0,41**	**0,43**	**0,40**
	EN	0,79	0,88	1,03	0,67	0,65	0,72	0,62
lame	AT	**0,87**	**1,20**	**1,25**	**0,65**	**0,58**	**0,67**	**0,58**
	EN	0,65	1,18	1,34	0,64	0,45	0,70	0,44
dijkstra	AT	**0,96**	**0,26**	**0,28**	**0,20**	**0,18**	**0,20**	**0,15**
	EN	0,85	0,54	0,65	0,41	0,36	0,45	0,20
blowfish	AT	**0,86**	**0,80**	**0,86**	**0,51**	**0,51**	**0,51**	**0,46**
	EN	0,65	0,91	1,06	0,62	0,62	0,63	0,51
rijndael	AT	**0,92**	**0,89**	**0,90**	**0,84**	**0,73**	**0,85**	**0,83**
	EN	0,85	0,98	0,98	0,91	0,68	0,93	0,90
sha	AT	**0,99**	**0,34**	**0,34**	**0,13**	**0,13**	**0,13**	**0,13**
	EN	0,99	0,75	0,78	0,46	0,46	0,46	0,44
adpcm_code	AT	**1,00**	**0,06**	**0,06**	**0,02**	**0,02**	**0,02**	**0,02**
	EN	1,19	0,18	0,02	0,01	0,01	0,01	0,01
adpcm_decode	AT	**1,00**	**0,12**	**0,12**	**0,02**	**0,02**	**0,02**	**0,02**
	EN	1,10	0,89	0,89	0,01	0,01	0,01	0,01
crc32	AT	**1,00**	**0,41**	**0,41**	**0,37**	**0,37**	**0,37**	**0,37**
	EN	1,02	0,97	0,98	0,91	0,91	0,91	0,89
FFT	AT	**0,88**	**0,94**	**0,99**	**0,92**	**0,79**	**0,98**	**0,78**
	EN	0,66	0,91	1,08	0,90	0,52	1,08	0,51
GSM	AT	**0,98**	**0,33**	**0,33**	**0,21**	**0,20**	**0,21**	**0,20**
	EN	0,92	0,84	0,89	0,43	0,41	0,47	0,41
média	AT	**0,94**	**0,63**	**0,65**	**0,41**	**0,38**	**0,42**	**0,38**
	EN	0,88	0,84	0,99	0,56	0,47	0,60	0,45
	AR	2,01	1,32	0,85	2,15	3,16	1,67	2,64

Tabela 5.8: Impacto da minimização de ciclos para as configurações C e E

Caso	Programa	Cenário 1: somente código			Cenário 2: código+dados		
		Margem	Configurações		Margem	Configurações	
			C	E		C	E
Pior	blowfish	60%	20%	49%	60%	20%	49%
Melhor	adpcm_code	91%	91%	91%	99%	94%	98%
Média	–	71%	37%	55%	77%	37%	59%

rar o impacto da técnica para configurações com a mesma quantidade de cache, mas com diferentes tamanhos de SPM (configurações C e E).

A Tabela 5.8 resume o melhor e pior resultados obtidos, bem como a média obtida no conjunto de programas, para as configurações C e E, em ambos os cenários, comparando-os com as respectivas margens de otimização para número de ciclos.

Para o programa `blowfish`, apenas 33% da margem de otimização disponível foi aproveitada na configuração C em ambos os cenários. Por outro lado, para a configuração E, pôde-se alcançar 82% da margem de otimização disponível, não havendo diferença de impacto entre os cenários. O fato de se ter obtido a mesma economia em ambos os cenários deve-se ao fato de que o número de ciclos relativos consumidos em dados estáticos é praticamente nulo (ver Tabela 5.2).

Por outro lado, observe que, para o programa `adpcm_code`, toda a margem de otimização foi aproveitada no Cenário 1 em ambas as configurações. A eficiência observada explica-se pelo fato de esse programa invocar frequentemente um procedimento de pequeno tamanho (340 bytes). Por isso toda a margem de otimização pôde ser aproveitada no Cenário 1. No Cenário 2, os 8% da margem de otimização adicionados pela inclusão dos dados estáticos são referentes em grande parte a duas estruturas de dados: a primeira com tamanho de 600 bytes e a segunda com tamanho de 3000 bytes. Portanto, utilizando-se 1KB de SPM, somente foi possível a alocação da estrutura menor (600 bytes) e do procedimento de 340 bytes citado anteriormente. Por outro lado, utilizando-se 4KB de SPM, foi possível a alocação de ambas as estruturas e do procedimento, sobrando assim pouco espaço em SPM (156 bytes) para os elementos de programa restantes (os quais são responsáveis por 1% da margem de otimização restante).

Em média, cerca de metade da margem de otimização disponível foi aproveitada na configuração C e 77% na configuração E, não havendo diferença de impacto sensível entre os cenários.

Agora vamos considerar todas as configurações que possuem SPM (C a H), mesmo sem se conservar o mesmo tamanho da cache da configuração-base. Nesse universo, os ciclos relativos médios dos Cenários 1 (Tabela 5.6) e 2 (Tabela 5.7) são 0,50 e 0,48, respectivamente. Portanto, do Cenário 1 para o Cenário 2, observou-se um fator de redução do número de ciclos de 0,96, em média, o que corresponde a uma economia extra de 4% (em relação ao Cenário 1).

Fatores de redução do número de ciclos de 0,34 (66% de economia extra) e 0,38 (62% de economia extra) foram obtidos para os programas `adpcm_code` e `adpcm_decode`, respectivamente. O maior impacto da otimização sobre esses dois programas deve-se à sua maior contribuição de número de ciclos no acesso a dados estáticos, quando comparada à dos demais programas.

Note que o impacto da otimização foi bastante inferior ao se minimizar o número de ciclos quando comparado com a minimização de energia. Isto pode ser explicado pelas diferentes configurações utilizadas nas caches: supondo um grau de localidade similar no acesso a código e no acesso a dados, o fato de a cache de instruções ser mapeada diretamente tende a induzir um maior número de ciclos do que a cache de dados, que é de conjunto associativo de quatro vias. Portanto, essa configuração assimétrica das caches de instruções e de dados torna mais impactante a alocação de *código* em SPM, pois a diferença no número de ciclos de acesso entre a SPM e a cache de instruções (mapeada diretamente) é maior que a diferença entre a SPM e a cache de dados (associativa de quatro vias). Assim as configurações adotadas beneficiam técnicas que só alocam código em SPM, como a própria técnica proposta em [ANG 04] (de onde as configurações foram extraídas para realizar comparações aqui apresentadas).

Embora nem sempre a minimização de ciclos correlate com a melhoria no consumo de energia, a aplicação da técnica proposta resultou, em

média, numa redução no consumo de energia. Por exemplo, para a configuração C, as economias resultantes no consumo de energia foram de 8% (Tabela 5.6) e 16% (Tabela 5.7). Para a configuração E, essas economias foram 23% e 44%, respectivamente.

Vamos nos concentrar agora nas últimas linhas das Tabelas 5.6 e 5.7, que mostram a área relativa (AR) em relação à configuração A, para cada configuração (B a H). Note que, para implementar a configuração C, foi necessário aumentar a área de um fator de 1,32 em relação à configuração-base para se obter reduções médias de 0,63 (para ambos os cenários) no número de ciclos. Para a configuração E, a área precisou ser aumentada de um fator de 2,15 para se obter reduções de 0,45 e 0,41, respectivamente. Conforme já mencionado na sub-seção anterior, a configuração D é a única que reduz a área ocupada em relação à configuração-base. Neste caso, a área do subsistema de memória foi diminuída de um fator 0,85 para se obter reduções de 0,65 no número de ciclos de memória para ambos os cenários. Isso é uma forte evidência de que a técnica proposta pode resultar em reduções do número de ciclos mesmo utilizando menos área do que um sistema composto somente por caches (como é o caso da configuração-base).

5.3 Impacto da inclusão de bibliotecas no espaço de otimização

5.3.1 Caracterização do impacto potencial de bibliotecas

A Tabela 5.9 ilustra, para cada programa, a quantidade de código e de dados em termos de aplicativo (app), bibliotecas de terceiros (lib) e bibliotecas da linguagem C (clib). Para facilitar a vizualização dos dados, os valores da Tabela 5.9 estão plotados nas Figuras 5.3 e 5.4. A Tabela 5.9 também mostra, para cada programa, o percentual do tamanho dos arquivos binários que correspondem a bibliotecas, ou seja, $((lib+clib)/(app+lib+clib))$, a assim-chamada **densidade de bibliotecas** (LD). Para os programas selecionados, em média 92% dos elementos de código (procedimentos) e 61% dos dados estáticos estão encapsulados em bibliotecas (em particular todos os elementos de dados provêm de bibliotecas para os programas `blowfish`, `lame`, `FFT` e `susan`). Isto é uma forte evidência da necessidade de se incluir as bibliotecas no espaço de otimização.

Esta seção reporta os resultados da aplicação da técnica proposta em quatro cenários distintos, enumerados na Tabela 5.10, para avaliar o impacto de se incluir ou não as bibliotecas no espaço de otimização. Os dois primeiros cenários permitem avaliar o impacto de bibliotecas quando somente código é alocado na SPM, enquanto os dois últimos permitem avaliar o impacto quando código e dados podem ser alocados na SPM.

O Cenário 1 corresponde à otimização de somente código sem (1a) e com (1b) a inclusão de bibliotecas. O Cenário 2 corresponde à otimização de código e dados sem (2a) e com (2b) a inclusão de bibliotecas no espaço de otimização.

Tabela 5.9: Caracterização dos benchmarks

Programa	Código tamanho (bytes) clib	lib	app	LD (%)	Dados tamanho (bytes) clib	lib	app	LD (%)
qsort	63888	0	712	99	3966	0	2400	62
susan	52632	0	24680	68	3788	0	0	100
cjpeg	56248	46520	18888	84	3666	2008	224	96
lame	75600	112756	492	100	5068	457423	0	100
dijkstra	54760	0	1180	98	3634	0	40832	8
blowfish	13628	3916	632	97	2908	4168	0	100
rijndael	40452	0	14064	74	3592	0	40832	8
sha	39700	0	1980	95	2820	0	8552	25
adpcm_code	39376	0	820	98	2820	0	2924	49
adpcm_decode	39376	0	800	98	2820	0	2924	49
crc32	41280	0	524	99	2820	0	1024	73
FFT	49500	0	3572	93	4344	0	0	100
GSM	49344	22128	8304	90	2844	622	13568	61
Média	–	–	–	92	–	–	–	61

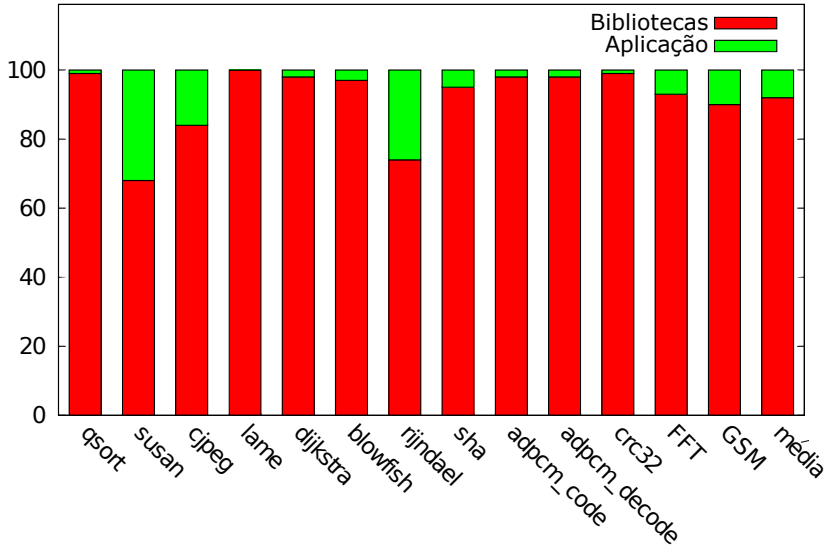

Figura 5.3: Caracterização dos benchmarks (somente código)

Também na comparação do impacto das bibliotecas optou-se por configurações com a mesma quantidade de cache da configuração-base (A), mas com diferentes tamanhos de SPM (C e E).

A Tabela 5.11 mostra os resultados de economia de energia para todos os cenários. A interpretação dos resultados será realizada nas Seções 5.3.2 e 5.3.3

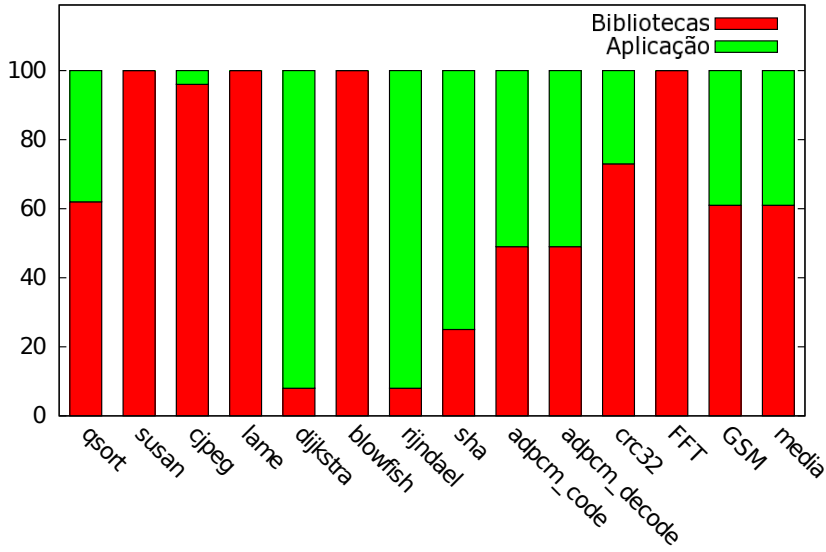

Figura 5.4: Caracterização dos benchmarks (código e dados)

Tabela 5.10: Cenários para avaliar o impacto de bibliotecas

Cenários	Código	Dados	Bibliotecas
1a	Sim	Não	Não
1b	Sim	Não	Sim
2a	Sim	Sim	Não
2b	Sim	Sim	Sim

Tabela 5.11: Impacto da alocação de bibliotecas na economia de energia

Programa	Cenário 1: somente código				Cenário 2: código + dados			
	1a		1b		2a		2b	
	C	E	C	E	C	E	C	E
qsort	0,95	0,94	0,85	0,71	0,95	0,94	0,85	0,69
susan	1,06	0,80	0,96	0,60	1,06	0,80	0,96	0,60
cjpeg	1,01	0,99	0,78	0,71	1,01	0,99	0,74	0,67
lame	1,01	1,01	0,83	0,70	1,01	1,01	0,83	0,70
dijkstra	0,80	0,86	0,88	0,84	0,70	0,48	0,51	0,42
blowfish	0,92	0,82	0,91	0,62	0,91	0,82	0,91	0,62
rijndael	0,98	0,96	0,96	0,91	0,98	0,86	0,96	0,86
sha	0,92	0,90	0,92	0,90	0,50	0,45	0,49	0,45
adpcm_code	0,81	0,81	0,81	0,81	0,18	0,01	0,18	0,01
adpcm_decode	0,90	0,90	0,90	0,90	0,87	0,01	0,87	0,01
crc32	0,37	0,32	0,37	0,32	0,37	0,30	0,37	0,30
FFT	1,03	0,99	0,74	0,51	1,03	0,99	0,74	0,51
GSM	0,84	0,43	0,84	0,43	0,84	0,43	0,84	0,43
Média	0,89	0,83	0,83	0,69	0,80	0,62	0,71	0,48

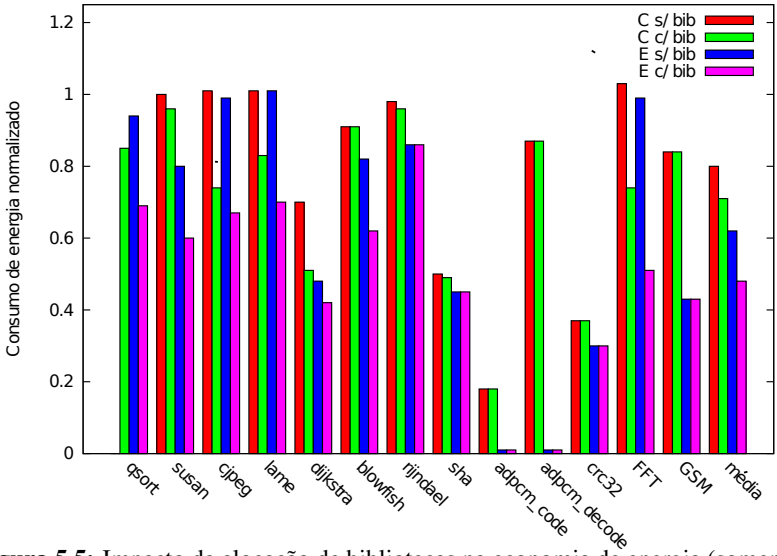

Figura 5.5: Impacto da alocação de bibliotecas na economia de energia (somente código)

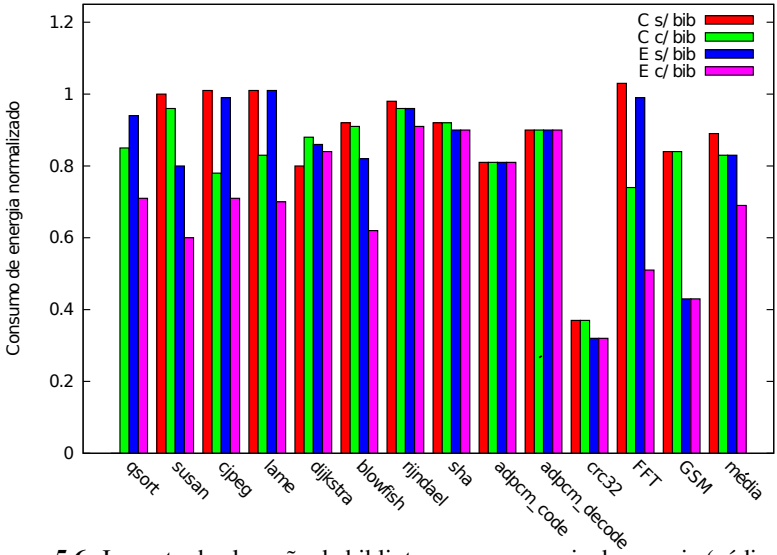

Figura 5.6: Impacto da alocação de bibliotecas na economia de energia (código e dados)

5.3.2 Impacto das bibliotecas quando só código é alocado em SPM

Esta seção compara os resultados da Tabela 5.11 para os Cenários 1a e 1b. Para facilitar a interpretação dos dados, os resultados obtidos nos Cenários 1a e 1b estão plotados na Figura 5.5. O menor impacto foi observado para o programa `adpcm_code`. A razão de o impacto ter sido nulo em ambas as configurações deve-se ao fato de que grande parte da energia consumida neste programa é proveniente de uma estrutura de dados responsável pela codificação do áudio, a qual não faz parte de quaisquer bibliotecas.

Os maiores impactos da inclusão de bibliotecas foram observados para os programas `cjpeg` e `FFT`. Para o programa `cjpeg`, economias de energia de 23% e 28% puderam ser obtidas para as configurações C e E, respectivamente, o que correlata com a maior contribuição de energia proveniente de bibliotecas (conforme verificado através de *profiling*). Para o programa `FFT`, economias de energia de 28% e 48% puderam ser obtidas para as configurações de memória C e E, respectivamente. Ora, a aplicação `FFT` utiliza cálculos matemáticos em ponto flutuante e o simulador do conjunto de instruções do processador utilizado nos experimentos não tem suporte a tal função. Por isto, teve-se que emular operações de ponto flutuante, através de uma biblioteca do compilador `gcc` chamada de *soft-float* (*clib*). Consequentemente, grande parte da energia consumida nessa aplicação provém da execução das rotinas dessa biblioteca.

Em média economias de energia de 7% (configuração C) e 17% (configuração E) puderam ser obtidas.

5.3.3 Impacto das bibliotecas quando código e dados são alocados em SPM

Esta seção compara os resultados da Tabela 5.11 para os Cenários 2a e 2b. Para facilitar a interpretação dos dados, os resultados obtidos nos Cenários 2a e 2b estão plotados na Figura 5.6. O menor impacto foi observado para o programa `GSM`. A razão de o impacto ter sido nulo em ambas as configurações C e E deve-se ao fato de que a energia relativa consumida por elementos de programa encapsulados em bibliotecas é muito baixa (conforme verificado através de *profiling*), embora o tamanho relativo de código encapsulado em bibliotecas seja de 90% e o tamanho relativo de dados encapsulados em bibliotecas seja de 61%.

Os maiores impactos foram observados para os programas `cjpeg` e `FFT`. Para o programa `cjpeg`, economias de energia de 27% e 32% puderam ser obtidos para as configurações de memória C e E, respectivamente. Isto correlata com a maior contribuição de energia proveniente de bibliotecas, especialmente de bibliotecas de terceiros (*lib*). Para o programa `FFT`, economias de energia de 28% e 48% puderam ser obtidos para as configurações C e E, respectivamente. Conforme já explicado anteriormente, isto se deve à biblioteca `soft-float` utilizada para emulação de ponto flutuante.

Em média economias de energia de 11% (configuração C) e 23% (configuração E) puderam ser obtidas.

5.4 Eficiência da ferramenta

Os tempos de execução da ferramenta para os diferentes programas e para as diferentes configurações de memória estão ilustrados na Tabela 5.12.

Dentre as etapas da técnica proposta, a que consome mais tempo de execução é a etapa de *profiling*. Isto se deve em parte à necessidade de se executar o programa para se extrair seu *trace* de acesso à memória (assim, há uma grande variação nos tempos de geração do *trace*). Essa etapa acomoda também a simulação do conjunto de instruções do processador-alvo e a hierarquia de memória da configuração-base. Além disso, existe um *overhead* devido à comunicação, através do uso de *sockets*, das três ferramentas necessárias para o *profiling* (simulador do conjunto de instruções do processador, simulador da hierarquia de memória e *profiler* proposto neste trabalho).

O método proposto nesta dissertação mostrou-se bastante eficiente computacionalmente, resultando em um tempo médio de *patching* de 0,7s. Note que, ao contrário do tempo de *profiling*, este tempo praticamente não varia entre os programas. Na Tabela 5.12 estão ilustrados os tempos de *patching* para os diferentes programas testados e para dois tamanhos de SPM (configurações C e E). Conforme visto no Capítulo 4, o número de operações de *patching* é, no pior caso, o número total de elementos mapeados (M) vezes o número total de elementos de dados (N). Porém, o caso médio da ferramenta mostrou-se muito mais eficiente. Na Figura 5.7, o número máximo de operações de *patching* (pior caso) é ilustrado pela curva em azul (linha sólida), enquanto o tempo de execução da ferramenta é ilustrado na curva em laranja (linha tracejada). Como a variação no número máximo de operações (105 vezes) é bem maior do que a variação no tempo de execução medido (1,37 vezes), há um índice de que o número médio de operações é em torno de 76,9 vezes inferior que o pior caso, para o conjunto de programas e configurações exploradas.

Todas as instâncias do Problema da Mochila induzidas pelos programas adotados nos experimentos realizados puderam ser classificadas na Classe 1 (veja Seção 4.2.3), ou seja, as propriedades peso e lucro não são correlacionadas. O tamanho médio das instâncias resolvidas foi de 223 itens, enquanto o maior tamanho observado foi de 463 itens. Na resolução, o número de soluções enumeradas pelo Algoritmo 2 ($|S|$) foi de 2^7 em média e 2^{11} no pior caso observado. Esses números justificam por que os tempos de execução da etapa de mapeamento são desprezíveis frente às demais etapas.

A julgar pelos experimentos aqui reportados, há evidência preliminar de que instâncias do Problema 2 que abordam alocação em SPMs recaiam na Classe 1, pois observou-se a não-correlação entre pesos e lucros

Tabela 5.12: Tempos de execução da ferramenta

		Tempo(s)				Tempo(s)	
Programa	**Etapa**	**C e D**	**E a H**	**Programa**	**Etapa**	**C e D**	**E a H**
qsort	Patching	0,64	0,66	susan	Patching	0,64	0,66
	Mapeamento	0	0		Mapeamento	0	0
	Profiling	2,4	2,4		Profiling	50,41	50,41
cjpeg	Patching	0,81	0,85	lame	Patching	0,76	0,83
	Mapeamento	0	0		Mapeamento	0	0
	Profiling	22,6	22,6		Profiling	9600	9600
dijkstra	Patching	0,67	0,71	blowfish	Patching	0,65	0,67
	Mapeamento	0	0		Mapeamento	0	0
	Profiling	10,86	10,86		Profiling	59,78	59,78
rijndael	Patching	0,62	0,62	sha	Patching	0,7	0,74
	Mapeamento	0	0		Mapeamento	0	0
	Profiling	62,54	62,54		Profiling	26,86	26,86
adpcm_code	Patching	0,66	0,69	adpcm_decode	Patching	0,65	0,67
	Mapeamento	0	0		Mapeamento	0	0
	Profiling	82,09	82,09		Profiling	76,49	76,49
crc32	Patching	0,6	0,72	FFT	Patching	0,63	0,7
	Mapeamento	0,01	0,01		Mapeamento	0	0
	Profiling	109	109		Profiling	1800	1800
GSM	Patching	0,67	0,75	**Média**	Patching	0,67	0,72
	Mapeamento	0	0		Mapeamento	0	0
	Profiling	78,23	78,23		Profiling	921,64	921,64

para todos os casos de uso utilizados. Assim, se essa observação se mantiver para outros casos de uso reais de complexidade similar mas natureza distinta, é de se esperar que casos de uso de maior tamanho também sejam resolvidos eficientemente, pois a eficiência de sua resolução teria pouca correlação com seu tamanho [PIS 97].

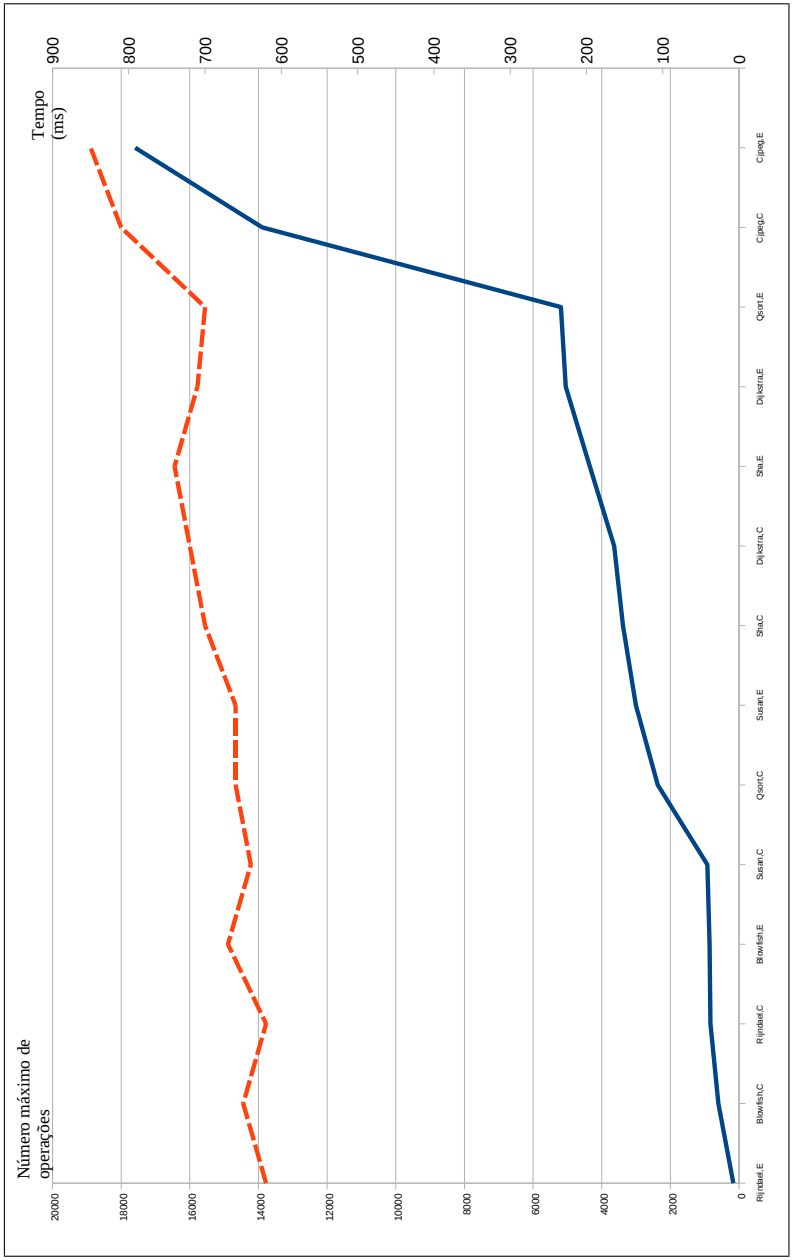

Figura 5.7: Número de operações e tempo de execução para diferentes programas e configurações

5.5 Comparação com outros métodos

Esta seção trata da comparação do método aqui proposto com outros reportados na literatura. A comparação restringe-se aos métodos que, como o aqui proposto, incluem bibliotecas no espaço de otimização.

5.5.1 Comparação com o método de Angiolini et al.

O primeiro método a ser comparado é a técnica proposta por Angiolini et al. [ANG 04]. Como fonte de informação para a otimização, a técnica de Angiolini e colaboradores utiliza arquivos binários executáveis, enquanto o método proposto nesta dissertação utiliza arquivos binários relocáveis.

O método proposto por Angiolini et al. limita-se a alocar código em SPM (mas não dados). Uma vantagem desse método é a sua flexibilidade na alocação de código com diferentes granularidades: por exemplo, funções, blocos básicos ou até mesmo uma única instrução. Isto é possível devido à inserção de instruções extras que desviam o fluxo de execução da memória principal para a SPM e vice-versa. Isto gera uma expansão de código de duas instruções por elemento mapeado para a SPM (desvio para/do SPM).

Infelizmente, uma comparação direta entre os dois métodos não é possível devido a diferentes condições experimentais que serão explicadas a seguir:

- **Processador-alvo:** o método proposto por Angiolini utiliza como processador-alvo um ARM v7, enquanto o método aqui proposto utiliza o MIPS R3000. As maiores discrepâncias podem resultar dos diferentes tamanhos de código e números distintos de instruções.

- **Nodo tecnológico:** No método de Angiolini esta informação não é fornecida, enquanto o método aqui proposto utiliza 90nm.

- **Emulação de ponto flutuante:** Como o modelo de processador utilizado (gerado pelo pacote ArchC [RIG 04]) não fornece suporte a ponto flutuante, utilizamos emulação por software através da biblioteca `soft-float`. Embora o processador ARM v7 utilizado nos experimentos de Angiolini et al. possua suporte para ponto flutuante através de um co-processador (opcional), aquele trabalho não informa se esse co-processador foi ou não usado nos experimentos.

O método proposto em [ANG 04] fornece resultados experimentais para **apenas** dois programas: `FFT` e `jpeg`. Porém, justamente para esses programas a energia relativa consumida quando acessando dados estáticos

é muito pequena: 0% para o `FFT` e 4% para o `jpeg` (ver Tabela 5.2). Portanto, ao restringir-se a programas cujos dados impactam marginalmente a otimização, os autores acabam negligenciando o impacto potencial da limitação de seu método à mera alocação de código.

Apesar das diferenças de condições experimentais, as Tabelas 5.13 e 5.14 mostram resultados da comparação entre os dois métodos. Para comparar as diferenças de configurações das máquinas hospedeiras, os tempos de execução reportados em [ANG 04] foram ajustados com base na razão SpecInt2000 [COU 00]. Para ambos os programas, o método proposto nesta dissertação obteve economia similar, utilizando-se 4KB de SPM (Configuração E). Por outro lado, o protótipo da técnica aqui proposta resultou em um tempo de execução muito inferior para ambos os programas. Isto é devido a dois fatores principais:

- **Fonte de informação para a otimização:** o método de Angiolini utiliza arquivos binários executáveis, o que gera mais operações devido ao *patching* de endereços absolutos e também ao deslocamento dos dados, como visto na Seção 4.6.
- **Granularidade de otimização:** o método de Angiolini utiliza uma granularidade menor para a alocação dos elementos na SPM, o que gera mais operações no Mapeador e no *Patcher* devido à inserção de instruções extra, que são responsáveis pela alteração do fluxo de execução da aplicação.

Tabela 5.13: Comparação com Angiolini et al. para o programa `FFT`

Método	Processador	Segmento de código	Tecnologia	Economia (conf. E)	Tempo
Angiolini et al. [ANG 04]	ARM v7	10.164 bytes	N/D	50%	14s
Proposto	MIPS R3000	54.912 bytes	90nm	49%	0,7s

Tabela 5.14: Comparação com Angiolini et al. para o programa `jpeg`

Método	Processador	Segmento de código	Tecnologia	Economia (conf. E)	Tempo
Angiolini et al. [ANG 04]	ARM v7	N/D	N/D	36%	28s
Proposto	MIPS R3000	123.496 bytes	90nm	37%	0,85s

Assim, supondo o mesmo nodo tecnológico e segmentos de código de mesmo tamanho, a técnica proposta atinge economias de energia bastante similares com tempos de execução inferiores em pelo menos uma ordem de grandeza.

5.5.2 Comparação com o método de Cho

O segundo método a ser comparado foi proposto por Cho et al. [CHO 07] e utiliza arquivos binários executáveis como fonte de informação para a otimização. É um método dinâmico (*overlaying*) que se limita a alocar em SPM dados estáticos e dinâmicos, não alocando código.

Há uma inadequação nos experimentos reportados em [CHO 07]. Como configuração-base, os autores utilizaram uma cache de instruções oito vezes maior que a cache de dados. Repare que, como aquele método não aloca código para a SPM, esse super-dimensionamento faz com que a energia **relativa** consumida quando acessando elementos de código seja bastante reduzida, pois irão ocorrer menos faltas na cache de instruções. Para ilustrar essa inadequação utilizamos um *profiler* com duas configurações de memória. Uma delas com a configuração escolhida no método de Cho e outra com uma configuração alternativa com 1KB de cache de instruções e 1KB de cache de dados.

Os resultados do *profiling* podem ser vistos na Tabela 5.15, onde: $SD = (E_{DadosEstáticos}/E_{total})$, $CO = (E_{Código}/E_{total})$ e $DD = (E_{DadosDinâmicos}/E_{total})$. A primeira linha da Tabela 5.15 refere-se ao *profiling* da aplicação utilizando à configuração da hierarquia de memória reportada em [CHO 07]; a segunda linha refere-se a configuração alternativa utilizada anteriormente nesta dissertação (configuração A). A configuração-base utilizada em [CHO 07] ameniza a limitação da não-alocação de código neste método. Utilizando-se a configuração-base de Cho et al., aquele método induz uma margem de otimização de 86% (devido à alocação de dados dinâmicos), enquanto a margem de otimização disponível para nosso método é 14% (aproximadamente seis vezes menos).

Por outro lado, se utilizarmos a configuração equitativa (1KB de cache de instruções e 1KB de cache de dados), as margens de otimização se alteram para 56% e 44%, respectivamente.

De forma similar ao que foi descrito na Seção 5.5.1, não foi possível uma comparação direta com o método proposto por Cho devido à não-reproductibilidade da infra-estrutura experimental. Por isso, os resultados apresentados na Tabela 5.16 devem ser interpretados à luz das diferentes configurações experimentais.

A grande diferença na economia de energia (sete vezes) pode ser explicada principalmente pela diferença na margem de otimização introduzida pela configuração experimental que o método de Cho et al. utilizou (cache de instruções super-dimensionada). Ademais, parte da discrepância pode ser explicada pelo diferente nodo tecnológico utilizado nos dois métodos.

Um ponto que chama a atenção é a diferença nos tempos de execução dos dois métodos (os tempos de execução reportados em [CHO 07] foram ajustados na razão do SpecInt2000 [COU 00]). O tempo de execução do método de Cho é três ordens de magnitude superior, embora capture

apenas o tempo associado à fase de mapeamento, enquanto que os tempos reportados para a técnica aqui proposta combinam os tempos de mapeamento (Algoritmo 6) e o tempo de *patching*. Provavelmente, o altíssimo tempo de mapeamento é resultado do uso de um resolvedor ILP no método de Cho et al., enquanto o método proposto nesta dissertação usa programação dinâmica. Além disso, é de se esperar que os tempos totais no método de Cho sejam ainda maiores devido à ineficiência do *patching* em arquivos executáveis.

Tabela 5.15: Resultado do *profiling* para as duas configurações-base

Configuração-base	I-Cache	D-Cache	SD	CO	DD
Configuração utilizada em Cho	8KB	1KB	0%	14%	86%
Configuração alternativa	1KB	1KB	0%	56%	44%

Tabela 5.16: Comparação com Cho para o programa FFT

Método	Processador	Segmento de código	Tecnologia	Economia	Tempo
Cho[CHO 07]	ARM926EJ-S	67 KB	130nm	35%	225s
Proposto	MIPS R3000	5 KB	90nm	5%	0,7s

Capítulo 6

Conclusões e perspectivas

Este capítulo inicia com conclusões globais a respeito da eficácia, eficiência e limitações atuais da técnica proposta (Seção 6.1). Em seguida, a Seção 6.2 mostra como contornar algumas das limitações atuais. Por fim, a Seção 6.3 propõe um método para combinar a técnica estática pós-compilação com técnicas dinâmicas incorporadas em um compilador.

6.1 Eficiência, eficácia e restrições

Impacto das limitações impostas à alocação:
Os resultados mostram que o impacto aparentemente marginal de se limitar o espaço de otimização somente a dados [CHO 07] ou somente a código [ANG 04] é resultado de experimentação com um conjunto muito limitado de casos de uso reais.

A mera inclusão de dados estáticos pode resultar em economia extra de 21% no consumo médio de energia. Foram encontradas instâncias de casos de uso reais com economias extra de 67% e 91% ao incluir dados estáticos.

Impacto da restrição adotada na técnica proposta:
A restrição da técnica proposta em não tratar dados dinâmicos resulta em se limitar a margem disponível para otimização em 61% da margem total (em média). Entretanto, a técnica mostrou-se eficaz em explorar a margem disponível, cuja metade foi aproveitada, em média, e cuja quarta parte foi aproveitada no pior caso.

Como para tratar dados dinâmicos, as alternativas seriam a alocação via compilador [VER 04] (inviabilizando a inclusão de bibliotecas em SPM), ou a adoção de hardware especializado [CHO 07], a restrição a dados estáticos parece representar uma alternativa simples e pragmática para explorar SPMs a partir de arquivos binários. Infelizmente, a limitada quantidade de casos de uso reportados para o método apresentado em [ANG 04] não permitiu quantificar quanto se deixa de economizar em média devido à restrição a dados estáticos.

Um trabalho futuro que pode viabilizar o tratamento dos três tipos de elementos de programa (código, dados estáticos e dados dinâmicos) em uma única técnica híbrida é discutido na Seção 6.3.

Eficiência da técnica proposta:

Ficou comprovada a maior eficiência da técnica devido ao uso de arquivos-objeto relocáveis, pois os tempos de execução são pelo menos uma ordem de magnitude inferiores aos observados em outras abordagens que operam sobre arquivos binários [CHO 07] [ANG 04].

Limitação do tipo de código binário tratável:

A desvantagem em se utilizar arquivos-objeto relocáveis ao invés de arquivos executáveis é a de que, embora se possa tratar bibliotecas-padrão e código de propriedade intelectual protegido na forma de bibliotecas, a técnica atualmente não trata diretamente código executável legado. Para viabilizar o tratamento de código legado, a técnica proposta requer um pré-processamento do arquivo-objeto executável com o objetivo de transformá-lo em um arquivo-objeto relocável. O pré-processamento consiste em uma análise do arquivo-objeto executável com a finalidade de criar relocações para as instruções de desvio incondicional. Infelizmente este pré-processamento não é pragmático quando aplicado a dados, pois o problema de determinar se dois acessos à memória apontam para o mesmo endereço (conhecido por *alias analysis*) é NP-Completo e os algoritmos aproximados que o resolvem não têm precisão suficiente [LAN 91] [CHA 02]. O algoritmo mais eficiente encontrado na literatura para resolver o problema de *alias analysis* em arquivos executáveis tem precisão entre 30% e 60% [DEB 98].

Devido a essa limitação intrínseca de utilizar-se arquivos-objeto executáveis, nenhum método proposto na literatura faz alocação de dados globais em SPM a partir de arquivos-objeto executáveis (sem suporte dedicado de hardware). Quando se utilizam arquivos-objeto relocáveis, essa dificuldade é contornada devido à presença de relocações que indicam a qual elemento se refere cada acesso à memória.

Métricas de qualidade para a alocação em SPM:

Nos experimentos foram utilizados a energia e o número de ciclos como métricas para se avaliar a qualidade da solução. Pretende-se repetir a análise para outras métricas importantes, tais como potência, produto energia-tempo (ET) [RAB 96], etc.

Limitação da plataforma-alvo:

Os experimentos reportados foram realizados em uma plataforma que contém um único processador. Ora, como grande parte dos SoCs contemporâneos são MPSoCs, deve-se esclarecer como a técnica é utilizada na alocação de SPMs numa plataforma *multicore*.

Como a técnica proposta assume implicitamente que o SPM é de uso exclusivo de um processador, quando SPMs são utilizadas como memórias privadas de cada *core*, a técnica pode ser aplicada diretamente, desde que os padrões de acesso à memória utilizados na caracterização refiram-se exclusivamente ao processo ou *thread* alocada para um dado *core*. Como a alocação de processos em *cores* é em si um outro problema de otimização [WOL 05] [BEN 05], supõe-se que uma solução para o particionamento de processos tenha sido obtida antes de a técnica aqui pro-

posta ser aplicada. Embora o uso de SPM como memória compartilhada seja menos comum, a técnica proposta poderia ser aplicada de maneira similar, pois SPMs não requerem protocolo de coerência. Um protocolo de coerência só se faz necessário se existir mais de uma cópia do mesmo elemento de programa em memórias de processadores diferentes. Ora, como o espaço de endereçamento da SPM é disjunto de outras memórias, inclusive de SPMs de processadores diferentes, nenhum mecanismo de coerência é necessário.

Duas linhas promissoras de trabalhos futuros podem ser vislumbradas, conforme discutido nas Seções 6.2 e 6.3.

6.2 Extensões à técnica proposta

Nesta seção, são apontadas três extensões para a técnica proposta. Estas extensões podem impactar o desempenho ou a generalidade. Primeiramente, será apresentada a idéia básica de cada extensão e, em seguida, será avaliado seu impacto potencial.

6.2.1 Diminuição da granularidade de dados

A diminuição da granularidade dos dados consiste na alocação de somente um pedaço de uma variável não-escalar (vetor) em SPM. Teoricamente, esta extensão teria muito impacto em aplicações que gastam muita energia em acesso a dados (principalmente pelo fato de que vetores maiores que o tamanho da SPM poderiam ser alocados parcialmente). Isto é extremamente difícil de se obter a partir de arquivos binários (tanto em executáveis quanto em relocáveis), pois normalmente os acessos a vetores se dão via a carga de endereço-base em registrador (no MIPS, por exemplo, uma alternativa é utilizar duas instruções `lui` e `addi` com informações de relocação para as duas instruções). Em seguida, os índices do vetor são somados a um deslocamento em uma instrução subsequente, permitindo assim acessar um endereço arbitrário dentro do vetor.

A Figura 6.1 apresenta um código simples que servirá de exemplo para ilustrar a dificuldade dessa extensão. A versão em linguagem de montagem do MIPS é mostrada na Figura 6.2.

Repare que na Figura 6.2 as duas únicas informações de relocação tem como dependência o elemento `a` (que é o elemento que deve ser particionado). As instruções dos endereços 0x4 e 0xc têm como finalidade carregar o endereço de `a` em um registrador e são as duas únicas instruções que possuem informação de relocação. Repare que, na instrução no endereço 0x8, o endereço do índice do vetor `a` é multiplicado por 4 e, em seguida, na instrução residente no endereço 0x10 o valor obtido por esta multiplicação é somado com o endereço-base de `a`. Isto permite o acesso a um elemento arbitrário dentro do vetor (pode-se acessar qualquer posição do vetor com somente uma instrução de carga de memória). A instrução no endereço 0x14 é a responsável pela carga de um elemento do vetor `a`

```
1 int a[5]={1,2,3,4,5};
2 int main()
3 {
4 int i=0;
5 for (i=0;i<5;i++)
6    a[i]=a[i]+10;
7 }
```

Figura 6.1: Exemplo ilustrativo para a diminuição da granularidade de dados

em registrador (neste caso `a[i]`).

Com base nestes fatos podemos argumentar sobre a possibilidade de se alocar um pedaço da estrutura `a` em SPM. Suponha que o elemento `a[2]` fosse escolhido para alocação em SPM. Teríamos as seguintes ordens de elementos na memória:

- Memória Principal: `a[0]`, `a[1]`, `a[3]`, `a[4]`
- SPM: `a[2]`

Conforme dito anteriormente, as únicas informações de relocação no código são relativas ao endereço-base de `a` em memória principal. Note também que na tabela de símbolos é gerada somente uma entrada para todos os elementos de `a` (corresponde ao endereço-base desse vetor). Ao alocar o elemento `a[2]` a ordem dos endereços internos de `a` foi alterada (não há mais contiguidade de todos os elementos de a). Assim, o código da Figura 6.2 não poderia ser utilizado (seria necessário reescrever quase que completamente o método `main` para efetuar os ajustes). Além disso, seria necessária a criação de um novo elemento na tabela de símbolos do arquivo para permitir ao linkeditor calcular o endereço de `a[2]`. Por esses motivos, a alocação em SPM de trechos de vetores, a partir de arquivos binários, não parece ser pragmática.

Por outro lado, a alocação em SPM de trechos de vetores pode ser feita em código-fonte, como reportado em [KAN 01]. Porém, quando acessos a um vetor são irregulares, o impacto desse tipo de alocação é pequeno (em muitos casos, o consumo de energia pode aumentar [CHE 06]).

6.2.2 Diminuição da granularidade de código

Na técnica implementada nesta dissertação, além de dados, foram alocadas funções. A técnica poderia ser estendida para acomodar a granularidade de blocos básicos[1]. É comum que a alocação ótima para a SPM leve em consideração apenas um laço predominante de uma função.

[1]Um bloco básico pode ser definido como uma sequência de instruções sem desvios, exceto talvez ao final, não contendo endereços-alvo de desvios, exceto possivelmente no início.

Cabeçalho	Nome	main	
	...	...	...
Segmento de texto	Endereço	Instrução	Comentário
	0	move a1,zero	i=0
	4	lui v0,0x0	
	8	sll v1,a1,0x2	v1 = &(i*4)
	C	addiu v0,v0,0	v0=&a
	10	addu v1,v1,v0	v1=&a+i
	14	lw v0,0(v1)	v0=a[i]
	18	addiu a1,a1,1	i=i+1
	1c	addiu v0,v0,10	a[i]=a[i]+10
	20	slti a0,a1,5	
	24	bnez a0,4	
	28	sw v0,0(v1)	&a[i]=a[i]
	2c	jr ra	
	30	nop	
Segmento de dados	Endereço	Valor	Referência
	0	0x0	a
Relocações	Endereço	Ação	Dependência
	4	R_MIPS_HI16	a
	C	R_MIPS_LO16	a
Tabela de símbolos	Rótulo	Endereço	Segmento
	a	0	Dados

Figura 6.2: Representação esquemática de arquivos-objeto relocáveis

Na Figura 6.2 três blocos básicos podem ser identificados, correspondentes ao prólogo, ao corpo do laço e ao seu epílogo (endereço 0x0, endereço 0x4 até 0x28 e endereço 0x2c até 0x30). Como o corpo do laço é invocado mais frequentemente, a granularidade de código poderia ser diminuida alocando-se somente o bloco básico correspondente ao corpo do laço, ao invés da função inteira, aumentando a eficiência (lucro similar e menor espaço ocupado em SPM). O espaço adicional poderia ser utilizado para acomodar outros blocos básicos ou elementos de dados.

Os seguintes passos descrevem o que deve ser modificado para a implementação da alocação de blocos básicos em SPMs sob a ótica das três etapas da técnica proposta: *Profiler*, Mapeamento e *Patching*.

Duas alterações são necessárias na etapa de *Profiling*:

1) Identificação de cada bloco básico.

2) Criação de um elemento (alocável em SPM) correspondente ao bloco básico identificado.

Na etapa de mapeamento, será necessária a alteração da equação de lucro para levar em consideração a expansão de código devida à necessidade de desviar o fluxo de execução (instruções de desvio de/para o bloco básico relocado em SPM). O novo lucro p'_i, calculado pela Equação 6.1, é baseado no lucro p_i, calculado na Equação 4.3, mas levando-se em consideração a energia E_i gasta com o desvio de entrada para SPM (residente em memória principal) e a energia E_{SPM} gasta com desvio de saída da SPM (residente na própria SPM).

$$p'_i = p_i - (E_i + E_{SPM}), \quad onde: E_i = E_{cache} + m_i \times E_{MM} \tag{6.1}$$

Como a alocação de blocos básicos também gera expansão de código (desvios de entrada e saída para a SPM), é necessária a alteração da equação que calcula o tamanho dos elementos de programa que são utilizados no algoritmo da mochila. No cálculo do tamanho w'_i do bloco básico é necessário levar em conta o tamanho, digamos Δ, de uma instrução de desvio incondicional (aquele que reside em SPM), conforme Equação 6.2.

$$w'_i = w_i + \Delta \tag{6.2}$$

Para a etapa de *Patching* são necessárias três ações para cada bloco básico selecionado para alocação em SPM:

1) Cópia do bloco básico: copiar as instruções referentes às faixas de endereços do bloco básico para a seção `.spm`.

2) Inserção de um desvio incondicional no início do bloco básico em MP: fazer com que o fluxo de execução do programa desvie para o endereço em SPM onde o bloco básico foi relocado (ou seja, o código do bloco básico sempre será executado em SPM).

3) Inserção de um desvio incondicional ao fim do bloco básico em SPM: devemos retornar o fluxo de execução para a memória principal (o endereço de retorno é o da instrução que sucede o bloco básico em memória principal no código original).

Repare que podemos utilizar um desvio incondicional (por exemplo, a instrução `j` no MIPS). A semântica de execução do programa não será alterada; sendo assim, ao transferirmos o fluxo de execução para a SPM não se faz necessário o salvamento de contexto em pilha e não é necessária a alteração de registradores de retorno (como o registrador `$ra` no MIPS).

Espera-se que esta alteração tenha um impacto significativo na economia de energia, pois isto evita o desperdício de espaço em SPM com as instruções de uma função que são pouco acessadas. Esta extensão é objeto de trabalho em andamento.

6.2.3 Redirecionamento automático

Uma das grandes vantagens em se utilizar arquivos-objeto relocáveis é que todos os elementos que necessitam de alteração ficam marcados com informações de relocação. Isto gera uma certa independência do conjunto de instruções da arquitetura conforme mostrado a seguir.

A abstração de relocação utilizada pela biblioteca GNU BFD compõe-se dos seguintes campos:

- **Ponteiro para o símbolo que necessita de relocação:** todas as relocações devem ser relativas a um símbolo que terá seu endereço definido pelo linkeditor.

- **Deslocamento da relocação na seção:** como as relocações são relativas à seção, este campo indica o endereço da instrução que necessita de relocação.

- **Adendo:** é um valor a ser somado ao deslocamento da relocação (utilizado em poucas arquiteturas).

- **Ponteiro para um tipo *howto*:** o campo *howto* pode ser imaginado como uma instrução de relocação, conforme descrição abaixo.

O campo *howto* é um ponteiro para uma estrutura que indica a ação requerida para aquele tipo de relocação. Os principais campos desta estrutura são:

- **rightshift:** Indica o número de bits de deslocamento para a direita que a relocação deve sofrer. Isto é necessário pois, em algumas arquiteturas, os valores de campos imediatos são deslocados para esquerda[2]. Isto permite que, ao codificar o endereço-alvo como um endereço de palavra, o alcance dos desvios seja ampliado.

- **name:** indica o nome textual da relocação utilizado para fins de depuração (por exemplo, R_MIPS_HI16 na Figura 6.2).

- **dst_mask:** este é o campo mais importante de uma relocação e indica, através de uma máscara, quais bits da instrução serão alteradas pela relocação (por exemplo, endereços-alvo de desvios incondicionais).

Pode-se perceber que a abstração de relocação da biblioteca BFD é bastante independente da arquitetura do conjunto de instruções do processador-alvo. Assim, ao se apoiar nessa abstração e ao utilizar, como ponto de partida, arquivos-objeto relocáveis, é possível construir com ela uma ferramenta automaticamente redirecionável para alocação de SPMs.

Essa extensão requer apenas pequenas alterações na implementação como, por exemplo, o suporte a diferentes *endianess*. A única restrição prática é a de que o código binário seja produzido no formato ELF [STA 95].

[2]Nas instruções `j` e `jal` do MIPS, por exemplo, o valor é codificado deslocado de dois bits para a esquerda.

6.3 Hibridização com técnicas complementares

Uma possível generalização consiste em viabilizar o uso de uma técnica que mapeia dados dinâmicos em tempo de compilação com uma técnica pós-compilação que mapeia código e dados, tal como a aqui proposta.

6.3.1 Motivação

Os programas utilizados nos experimentos desta dissertação revelam uma contribuição de energia balanceada entre os diferentes segmentos de memória. Em média, 34% da energia é consumida quando acessando código, 26% quando acessando dados estáticos e 40% quando acessando dados dinâmicos (pilha e *heap*). Um método que possa alocar em SPM os três segmentos principais de memória teria uma margem de otimização de 100%. Esses percentuais parecem indicar que a combinação de técnicas sugeridas poderia levar a reduções significativas de energia.

6.3.2 Particionamento do espaço de endereçamento

O particionamento do espaço de endereçamento da SPM entre as técnicas complementares pode ser obtido utilizando-se duas ferramentas principais empregados na técnica proposta: o *profiler* (responsável por identificar os segmentos de memória) e o linkeditor (responsável por implementar o particionamento).

A função do *profiler* será de coletar as informações do programa, tais como a energia relativa consumida em cada segmento de memória, o número de ciclos relativos gastos em cada segmento de memória, etc. A partir desta informação pode-se particionar o espaço de endereçamento da SPM entre os segmentos de memória, criando assim SPMs virtuais. Cada SPM virtual é dedicada a um segmento de memória. O tamanho de cada SPM virtual pode ser determinado a partir da contribuição relativa de energia ou de ciclos para seu segmento dedicado. A seguir, a idéia básica deste particionamento será ilustrada através de um exemplo.

Suponha que a energia relativa gasta pelo segmento de código e dados estáticos para um programa arbitrário seja de 70% e que a energia relativa consumida pelo segmento de dados dinâmicos seja de 30%. Suponha também, que o tamanho total da SPM seja de 4096 bytes. Para este exemplo seriam criados duas SPMs virtuais. A primeira SPM virtual teria uma capacidade de 2.268 bytes (70% do tamanho total do SPM físico). A segunda SPM virtual teria 1.228 bytes (30% do tamanho total do SPM físico). As faixas de endereços resultantes induzem o particionamento desejado.

Através desse pré-particionamento, a técnica em tempo de compilação seria restrita a alocações na faixa 0x0 a 0x8DB e a técnica pós-compilação à faixa 0x8DC a 0xFFF.

Como as diferentes técnicas operam sobre segmentos de memória distintos, descartam-se quaisquer conflitos potenciais. O resultado da aplicação de cada técnica é o mapeamento para SPMs virtuais distintas. Ora, basta elaborar um *script* de linkedição adequado para induzir a relocação dos endereços dos SPMs virtuais para as partições no SPM físico. A hibridização aqui proposta será objeto de trabalho futuro.

Referências bibliográficas

[ANG 04] ANGIOLINI, F. et al. A Post-compiler Approach to Scratchpad Mapping of Code. **Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)**, New York, NY, USA, p.259–267, 2004.

[AVI 02] AVISSAR, O.; BARUA, R.; STEWART, D. An Optimal Memory Allocation Scheme for Scratchpad-based Embedded Systems. **Transactions on Embedded Computing Systems**, New York, NY, USA, v.1, n.1, p.6–26, 2002.

[BAL 80] BALAS, E.; ZEMEL, E. An Algorithm for Large zero-one Knapsack Problems. **Operations Research**, [S.l.], v.28, n.5, p.1130–1154, 1980.

[BAL 05] BALDASSIN, A. **Geração automática de montadores em ArchC**. Dissertação de Mestrado: Instituto de Computação, UNICAMP, Campinas, 2005.

[BAL 08] BALDASSIN, A. et al. An Open-source Binary Utility Generator. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, New York, NY, USA, v.13, n.2, p.1–17, 2008.

[BAN 02] BANAKAR, R. et al. Scratchpad memory: Design Alternative for Cache On-chip Memory in Embedded Systems. **Proceedings of the International Symposium on Hardware/software Codesign (CODES)**, New York, NY, USA, p.73–78, 2002.

[BEN 02] BENINI, L. et al. Layout-driven Memory Synthesis for Embedded Systems-on-chip. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, [S.l.], v.10, n.2, p.96–105, Apr 2002.

[BEN 05] BENINI, L. et al. Allocation and Scheduling for MpSoCs via Decomposition and No-good Generation. **Lecture Notes in Computer Science**, [S.l.], 2005.

[BER 01] BERGER, A. S. **Embedded Systems Design: An Introduction to Processes, Tools and Techniques**. CMP Books, 2001.

[BIN 07] BINUTILS, G. **The GNU Binutils Website**.

[CAT 98] CATTHOOR, F.; DE GREEF, E.; SUYTACK, S. **Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design**. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[CAT 02] CATTHOOR, F. et al. **Data Access and Storage Management for Embedded Programable Processors**. Kluwer Academic Publisher, 2002.

[CHA 02] CHAKARAVARTHY, V.; HORWITZ, S. On the Non-approximability of Points-to Analysis. **Acta Informatica**, [S.l.], v.38, n.8, p.587–598, 2002.

[CHE 06] CHEN, G. et al. Dynamic Scratch-pad Memory Management for Irregular Array Access Patterns. **DATE '06: Proceedings of the conference on Design, automation and test in Europe**, 3001 Leuven, Belgium, Belgium, p.931–936, 2006.

[CHO 07] CHO, H. et al. Dynamic Data Scratchpad Memory Management for a Memory Subsystem With an MMU. **Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES)**, New York, NY, USA, p.195–206, 2007.

[COO 07] COOK, J. et al. Control, Computing and Communications: Technologies for the Twenty-First Century Model T. **Proceedings of the IEEE**, [S.l.], v.95, n.2, p.334–355, February, 2007.

[COU 00] COUNCIL, S. **SPECint2000 Benchmark**.

[DAL 08] DALLY, W. et al. Efficient Embedded Computing. **IEEE Computer**, [S.l.], v.41, n.7, p.27–32, July, 2008.

[DEB 98] DEBRAY, S.; MUTH, R.; WEIPPERT, M. Alias Analysis of Executable Code. **POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**, New York, NY, USA, p.12–24, 1998.

[DEM 80] DEMBO, R.; HAMMER, P. A Reduction Algorithm for Knapsack Problems. **Methods of Operations Research**, [S.l.], v.36, p.49–60, 1980.

[DOM 05] DOMINGUEZ, A.; UDAYAKUMARAN, S.; BARUA, R. Heap Data Allocation to Scratch-pad Memory in Embedded Systems. **Journal Embedded Computing**, Amsterdam, The Netherlands, v.1, n.4, p.521–540, 2005.

[EGG 06] EGGER, B. et al. A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization. **Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)**, New York, NY, USA, p.223–233, 2006.

[GAR 79] GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1979.

[GRU 01] GRUN, P.; DUTT, N.; NICOLAU, A. Apex: Access pattern based memory architecture exploration. **Proceedings of the 14th International Symposium on Systems Synthesis (ISSS)**, New York, NY, USA, p.25–32, 2001.

[GUT 01] GUTHAUS, M. R. et al. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. **Proceedings of the International Workshop on Workload**, [S.l.], p.3–14, 2001.

[HEN 06] HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 4. ed. Morgan Kaufmann, September, 2006.

[HOR 72] HOROWITZ, E.; SAHNI, S. Computing Partitions with Applications to the Knapsack Problem. Ithaca, NY, USA, 1972. Relatório técnico .

[KAN 01] KANDEMIR, M. et al. Dynamic Management of Scratch-pad Memory Space. **Proceedings of the**

38th Design Automation Conference, [S.l.], v., p.690–695, 2001.

[KAN 02] KANDEMIR, M.; CHOUDHARY, A. Compiler-directed Scratch Pad Memory Hierarchy Design and Management. **Proceedings of the 39th annual Design Automation Conference**, New York, NY, USA, p.628–633, 2002.

[KAR 72] KARP, K. M. Reducibility Among Combinatorial Problems. **In Complexity of Computer Computations**, [S.l.], 1972.

[KEN 01] KENNEDY, K.; ALLEN, J. R. **High-Performance Compilers**. Elsevier Science and Technology Books, 2001.

[LAN 91] LANDI, W.; RYDER, B. G. Pointer-induced Aliasing: A Problem Taxonomy. **POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**, New York, NY, USA, p.93–103, 1991.

[MAR 77] MARTELLO, S.; TOTH, P. An Upper Bound for the Zero-one Knapsack Problem and a Branch and Bound Algorithm. **European Journal of Operational Research**, [S.l.], v.1, n.169-175, p.849, 1977.

[MAR 88] MARTELLO, S.; TOTH, P. A New Algorithm for the 0-1 Knapsack Problem. **Management Science**, Institute for Operations Research and the Management Sciences (INFORMS), Linthicum, Maryland, USA, v.34, n.5, p.633–644, 1988.

[MAR 90] MARTELLO, S.; TOTH, P. **Knapsack problems: algorithms and computer implementations**. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[MAR 06] MARWEDEL, P. **Embedded System Design**. Springer Verlag, 2006.

[MCI 08] MCILROY, R.; DICKMAN, P.; SVENTEK, J. Efficient Dynamic Heap Allocation of Scratch-pad Memory. **Proceedings of the International Symposium on Memory Management (ISMM)**, New York, NY, USA, p.31–40, 2008.

[MEN 09] MENDONCA, A. K. I. et al. Mapping Data and Code into Scratchpads from Relocatable Binaries. **IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**, Washington, DC, USA, p.157–162, 2009.

[NAU 76] NAUSS, R. An Efficient Algorithm for the 0-1 Knapsack Problem. **Management Science**, [S.l.], v.23, n.1, p.27–31, 1976.

[PAN 97a] PANDA, P. R.; DUTT, N. D.; NICOLAU, A. Architectural Exploration and Optimization of Local Memory in Embedded Systems. **Proceedings of the 10th International Symposium on System Synthesis (ISSS)**, Washington, DC, USA, p.90–97, 1997.

[PAN 97b] PANDA, P. R.; DUTT, N. D.; NICOLAU, A. Exploiting Off-chip Memory Access Modes in High-level Synthesis. **Proceedings of the IEEE/ACM international conference on Computer-aided design (ICCAD)**, Washington, DC, USA, p.333–340, 1997.

[PAN 00] PANDA, P. R.; DUTT, N. D.; NICOLAU, A. On-chip vs. Off-chip Memory: The Data Partitioning Problem in Embedded Processor-based Systems. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, New York, NY, USA, v.5, n.3, p.682–704, 2000.

[PAP 81] PAPADIMITRIOU, C. H. On the Complexity of Integer Programming. **Journal of the ACM (JACM)**, New York, NY, USA, v.28, n.4, p.765–768, 1981.

[PAT 08] PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design: The Hardware/Software Interface**. 4. ed. Morgan Kaufmann, June, 2008.

[PIS 97] PISINGER, D. A Minimal Algorithm for the 0-1 Knapsack Problem. **Operations Research**, [S.l.], v.45, p.758–767, 1997.

[RAB 96] RABAEY, J.; CHANDRAKASAN, A.; NIKOLIĆ, B. **Digital Integrated Circuits: A Design Perspective**. Prentice Hall New Jersey, 1996.

[RAM 07] RAMACHER, U. Software-Defined Radio Prospects for Multistandard Mobile Phones. **IEEE Computer**, [S.l.], v.40, n.10, p.62–69, October, 2007.

[RIG 04] RIGO, S. et al. ArchC: A SystemC-Based Architecture Description Language. **Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**, Washington, DC, USA, p.66–73, 2004.

[STA 95] STANDARD, T. Executable and Linking Format (ELF) Specification Version 1.2. **TIS Committee, May**, [S.l.], 1995.

[STE 02a] STEINKE, S. et al. Assigning Program and Data Objects to Scratchpad for Energy Reduction. **Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)**, Los Alamitos, CA, USA, v.0, p.0409, 2002.

[STE 02b] STEINKE, S. et al. Reducing Energy Consumption by Dynamic Copying of Instructions Onto Onchip Memory. **Proceedings of the International Symposium on System Synthesis (ISSS)**, New York, NY, USA, p.213–218, 2002.

[SV 07] SANGIOVANNI-VINCENTELLI, A.; DI NATALE, M. Embedded System Design for Automotive Applications. **IEEE Computer**, [S.l.], v.40, n.10, p.42–51, October, 2007.

[TAL 07] TALLA, D.; GOLSTON, J. Using DaVinci Technology for Digital Video Devices. **IEEE Computer**, Los Alamitos, CA, USA, v.40, n.10, p.53–61, October, 2007.

[THO 08] THOZIYOOR, S. et al. CACTI 5.1. Hewlett-Packard Laboratories, Abril, 2008. Relatório técnico .

[TOT 80] TOTH, P. Dynamic programming algorithms for the zero-one knapsack problem. **Computing**, [S.l.], v.25, n.1, p.29–45, 1980.

[UDA 06] UDAYAKUMARAN, S.; DOMINGUEZ, A.; BARUA, R. Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. **Transactions on Embedded Computing Systems**, New York, NY, USA, v.5, n.2, p.472–511, 2006.

[UHL 97] UHLIG, R. A.; MUDGE, T. N. Trace-driven Memory Dimulation: A Survey. **ACM Computing Surveys (CSUR)**, New York, NY, USA, v.29, n.2, p.128–170, 1997.

[VER 04] VERMA, M.; WEHMEYER, L.; MARWEDEL, P. Dynamic Overlay of Scratchpad Memory for Energy Minimization. **Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/software Codesign and System Synthesis (CODES+ISS)**, New York, NY, USA, p.104–109, 2004.

[VER 07] VERMA, M.; MARWEDEL, P. **Advanced Memory Optimization Techniques for Low-Power Embedded Processors**. Springer Verlag, 2007.

[WOL 05] WOLF, W.; JERRAYA, A. **Multiprocessor Systems-on-chips**. Morgan Kaufmann, 2005.

[WOL 07] WOLF, W. Guest Editor's Introduction: The Embedded Systems Landscape. **IEEE Computer**, [S.l.], v.40, n.10, p.29–31, October, 2007.

[WUL 95] WULF, W. A.; MCKEE, S. A. Hitting the Memory Wall: Implications of the Obvious. **ACM SIGARCH Computer Architecture News**, New York, NY, USA, v.23, n.1, p.20–24, 1995.

[WUY 96] WUYTACK, S. et al. Power Exploration for Data Dominated Video Applications. **Proceedings of the International Symposium on Low power Electronics and Design (ISLPED)**, Piscataway, NJ, USA, p.359–364, 1996.