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

Alexandre Keunecke Ignácio de Mendonça[1]
José Luís Almada Güntzel [1], **Luiz Cláudio Villar dos Santos** [1]

[1]Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Santa Catarina – Florianópolis – SC – Brasil
{mendonca,guntzel,santos}@inf.ufsc.br

Resumo. *Este trabalho reporta as principais contribuições e resultados de uma dissertação de mestrado que aborda um problema contemporâneo de Computação Embarcada: o gerenciamento explícito de memória. Ao ser modelado como um problema clássico de Computação (Problema Binário da Mochila), ele pode ser eficientemente resolvido de forma exata (com um algoritmo de programação dinâmica) para minimizar o consumo energético do subsistema de memória, sem a necessidade de recompilação da aplicação, através do uso eficiente de uma combinação de memória cache com memória não-associativa do tipo scractchpad.*

Palavras-Chave: Computação Embarcada, Eficiência Energética, Problema da Mochila, Ligadores.

1. Introdução

Devido à sua maior latência e significativo consumo de energia, memórias *off-chip* limitam a eficiência energética em sistemas embarcados. Por este motivo, memórias cache embarcadas tem sido utilizadas. Porém, para algumas aplicações o mero uso de memórias cache não garante suficiente eficiência energética, pois os processadores embarcados contemporâneos podem gastar até 70% de seu consumo de energia suprindo dados e instruções via caches [Dally et al. 2008] (enquanto o consumo das operações aritméticas corresponde a 7%). Parte da energia gasta em caches é consumida na lógica que implementa sua estrutura associativa (e.g.: armazenamento e comparação de *tags*). Tendo essencialmente a mesma latência, porém consumindo menos energia que uma cache de mesma capacidade, uma memória do tipo *scratchpad* (SPM) pode substituir ou auxiliar a(s) cache(s). Ao contrário das caches (onde a alocação é implicitamente escolhida pelo hardware), elementos alocados em SPM (dados e instruções) são gerenciados pelo software embarcado: deve-se mapear explicitamente elementos de programa para o espaço de endereçamento da SPM, que é disjunto do da memória principal. Este mapeamento pode permanecer inalterado durante a execução da aplicação (alocação *non-overlaying*) ou pode ser alterado dinamicamente para atender às mudanças nos padrões de acesso da aplicação (alocação *overlaying*).

2. Principais contribuições deste trabalho

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. 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 do *benchmark* Mibench [Guthaus et al. 2001].

As principais contribuições dessa dissertação foram publicadas nos anais do *IEEE Computer Society Annual Symposium on VLSI* (ISVLSI 2009) [Mendonca et al. 2009]. As pesquisas nela iniciadas foram continuadas pelo autor em trabalho colaborativo posterior, dando origem a outro trabalho já publicado no ISVLSI 2010 [Volpato et al. 2010] e outro aceito para publicação no SBCCI 2011 [Volpato et al. 2011]. Além disso, um artigo foi submetido a um periódico da IEEE [Santos et al.]

3. Trabalhos Relacionados

a) Persistência do conteúdo em SPM: Algumas técnicas adotam a abordagem *non-overlaying* [Panda et al. 2000] e [Angiolini et al. 2004]. Com esta abordagem não há *overhead*

relacionado ao gerenciamento do conteúdo da SPM. 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. Outras técnicas adotam a abordagem *overlaying* [Kandemir et al. 2001], [Steinke et al. 2002b], [Steinke et al. 2002a], [Dominguez et al. 2005], [Udayakumaran et al. 2006], [Cho et al. 2007] e [McIlroy et al. 2008], que tenta manter em SPM os dados e instruções que são mais frequentemente acessados durante a execução, efetuando cópias da memória principal para a SPM e vice-versa. A decisão sobre quais dados serão copiados de ou para a SPM ocorre em tempo de compilação através de *profiling*.

b) Tratamento de estruturas de dados: Algumas técnicas mapeiam apenas código para a SPM ([Angiolini et al. 2004] e [Steinke et al. 2002b]), enquanto outras mapeiam somente dados ([Panda et al. 2000], [Kandemir et al. 2001], [Cho et al. 2007] e [Udayakumaran et al. 2006]). Finalmente, algumas mapeiam ambos, código e dados ([Steinke et al. 2002a] e [Verma et al. 2004]).

c) Dependência do código-fonte e tratamento de bibliotecas: Grande parte dos trabalhos correlatos consiste de técnicas que operam em tempo de compilação ([Kandemir et al. 2001], [Steinke et al. 2002b], [Steinke et al. 2002a], [Dominguez et al. 2005], [Udayakumaran et al. 2006], [McIlroy et al. 2008], [Panda et al. 2000]). Nessas técnicas, o impacto reportado assume implicitamente o acesso a todo o código-fonte da aplicação. Porém, nos casos mais realistas em que 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 relevantes são 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 ([Angiolini et al. 2004] e [Cho et al. 2007]). Uma alternativa é identificar elementos de programa canditados à otimização diretamente do arquivo executável, como feito em [Angiolini et al. 2004]. Porém, o gerenciamento das relocações em SPM a partir de arquivos executáveis é ineficiente, pois uma dada relocação para a SPM tende a gerar muitos ajustes no código e nos elementos de dados.

d) 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 do tipo *overlaying* 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 a manipulação de arquivos binários, os quais internamente particionam o código em elementos de dados estáticos e procedimentos. Esta dissertação propõe um método de alocação do tipo *non-overlaying* de código e dados em SPM para uma dada aplicação-alvo. O método proposto não requer a disponibilidade do código-fonte nem de hardware dedicado, o que amplia sua aplicação a bibliotecas pré-compiladas.

4. O problema-alvo e a solução proposta

Considerando-se um sistema embarcado que possua SPM e caches, quando uma aplicação embarcada é compilada, seus dados podem ser armazenados tanto na SPM quanto em memória principal externa. Neste último caso, o acesso aos dados e instruções é feito via cache. O objetivo é encontrar quais elementos de programa devem ser alocados para a SPM de maneira a maximizar a economia de energia. Antes de formular o problema-alvo, é preciso formalizar algumas noções importantes.

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 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. 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, onde α_i denota o i-ésimo endereço.

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. 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.

O problema-alvo pode ser formulado para recair no clássico Problema Binário da Mochila:

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 maximize $P * X$ 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 será definido na Seção 5).

Como o problema-alvo é NP-Completo [Garey and Johnson 1979], 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.

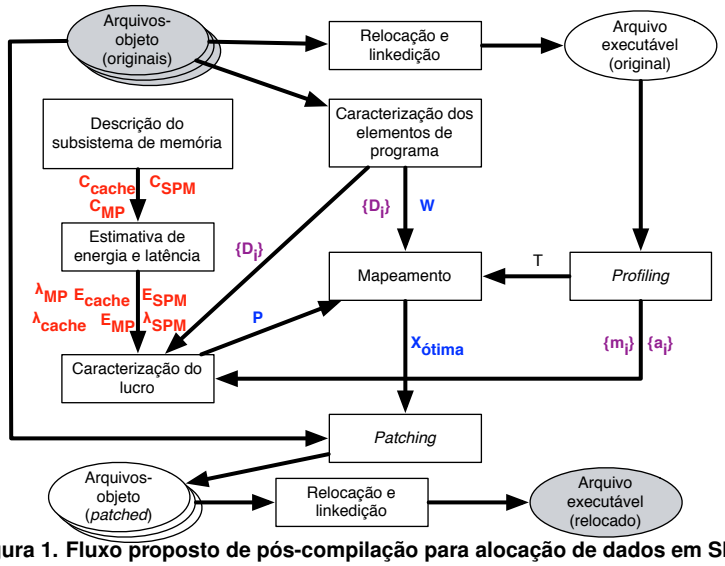

Figura 1. Fluxo proposto de pós-compilação para alocação de dados em SPM.

O fluxo de alocação ótima em SPM é ilustrado na Figura 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. Também a partir do código-objeto, gera-se um arquivo executável através 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. A partir da descrição do subsistema de memória, extraem-se 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 [Thoziyoor et al. 2008]. As propriedades extraídas permitem a criação da matriz-linha P associada à função lucro (definida na Seção 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-alvo, instrumentado com a função lucro pré-caracterizada. 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.

5. Aspectos mais relevantes da implementação

a) Formulação da função-lucro: Como o método aqui proposto não gera expansão de código, o cômputo do lucro em se alocar um elemento D_i na SPM é a energia economizada por acesso vezes o número de acessos a D_i, ou seja:

$$p_i = a_i \times (E_i - E_{SPM}), \quad onde: E_i = E_{cache} + m_i \times E_{MM} \tag{1}$$

b) Relocação e linkedição: Para suportar a otimização de bibliotecas mantendo seus elementos no espaço de otimização, sem se deixar limitar por um *patching* computacionalmente ineficiente esta dissertação propõe o uso de arquivos-objeto relocáveis, ao invés de se utilizar somente o arquivo executável. 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). 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.

6. Validação experimental e resultados

Para se fazer a experimentação do protótipo foram utilizados treze programas extraídos do *benchmark* MiBench [Guthaus et al. 2001]. Neste resumo apenas duas configurações serão consideradas para o subsistema de memória, ambas com $C_{I-cache} = C_{D-cache} = 1KB$, mas com SPMs de diferentes capacidades: 1KB e 4KB.

6.1. Impacto da inclusão de dados e de código no espaço de otimização

A Figura 2 mostra a contribuição de cada tipo de elemento de programa (código, dados estáticos e dados dinâmicos) ao consumo de energia de uma configuração-base com caches de capacidade $C_{I-cache} = C_{D-cache} = 1KB$, mas sem SPM. Como estes valores foram obtidos antes da introdução de SPM, eles nos permitem quantificar o impacto potencial em se alocar código e dados estáticos para SPM. Os valores foram medidos para todos os elementos de programa acessados, independentemente de estarem encapsulados em bibliotecas ou não.

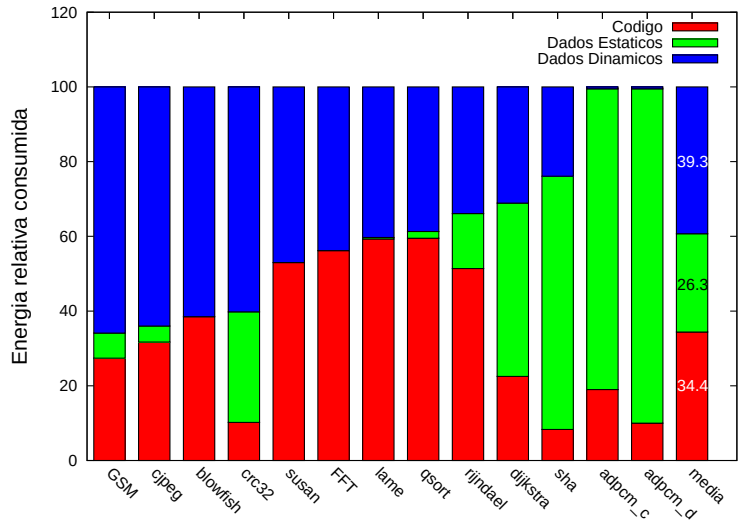

Figura 2. Contribuição ao consumo de energia

Em média, 61% da energia é consumida quando se acessam dados estáticos ou código. 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. Assim, podemos definir a **margem disponível para otimização** do método proposto como a soma das contribuições ao consumo de energia quando se acessam código e dados estáticos. 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.

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.

A Tabela 1 resume o melhor e o pior resultados obtidos, bem como a média obtida no conjunto de programas para as duas configurações de SPM, em ambos os cenários, comparando-os com as respectivas margens de otimização. Note que, para `blowfish`, apenas 23% da margem de otimização disponível foi aproveitada para $C_{SPM} = 1KB$ e quase totalmente aproveitada para $C_{SPM} = 4KB$ 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 Figura 2). Por outro lado, observe que, para `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 para $C_{SPM} = 4KB$. A diferença de impacto observada no Cenário 2 explica-se pelo fato de aquele programa utilizar uma estrutura de dados frequentemente

Tabela 1. Impacto da minimização de energia

Caso	Programa	Cenário 1: somente código			Cenário 2: código+dados		
		Margem	C_{SPM}		Margem	C_{SPM}	
			1KB	4KB		1KB	4KB
Pior	blowfish	39%	9%	38%	39%	9%	38%
Melhor	adpcm_decode	10%	10%	10%	99%	13%	99%
Média	–	34%	17%	31%	61%	29%	52%

acessada para a decodificação de áudio, a qual é maior que 1KB e menor que 4KB. Por isso, para $C_{SPM} = 1KB$ apenas 13% da margem de otimização foi aproveitada.

Em média, cerca de metade da margem de otimização disponível foi aproveitada para $C_{SPM} = 1KB$ e quase totalmente aproveitada para $C_{SPM} = 4KB$ (85%), não havendo diferença de impacto sensível entre os cenários. 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. Assim, apesar de seu relativamente baixo impacto potencial frente ao código (contribuições médias de 26%), dados estáticos não devem ser descartados do espaço de otimização.

6.2. Impacto da inclusão de bibliotecas no espaço de otimização

Para avaliar o impacto de bibliotecas, dois cenários são considerados na Figura 3: exclusão (Cenário 1) e inclusão (Cenário 2) de elementos de bibliotecas no espaço de otimização.

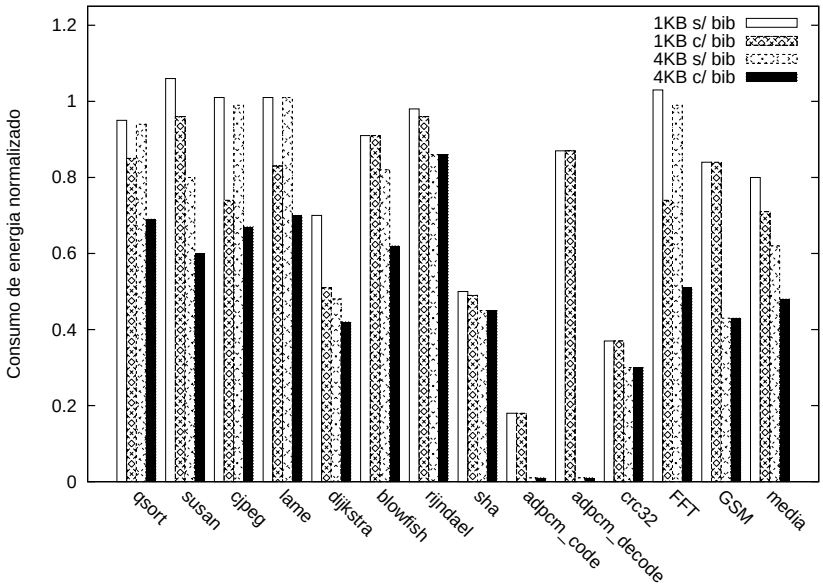

Figura 3. Impacto da alocação de bibliotecas na economia de energia

O menor impacto foi observado para `GSM`. A razão de o impacto ter sido nulo em ambas as configurações 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 `cjpeg` e `FFT`. Para `cjpeg`, economias de energia de 27% e 32% puderam ser obtidos para $C_{SPM} = 1KB$ e $C_{SPM} = 4KB$, respectivamente. Isto correlata com a maior contribuição de energia proveniente de bibliotecas, especialmente de bibliotecas de terceiros. Para `FFT`, economias de energia de 28% e 48% puderam ser obtidos, respectivamente.

Em média, economias extra de 11% ($C_{SPM} = 1KB$) e 23% ($C_{SPM} = 4KB$) puderam ser obtidas ao se incluir bibliotecas.

7. Conclusões

a) 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 et al. 2007] ou

somente a código [Angiolini et al. 2004] 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.

b) 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.

c) Eficiência da técnica proposta: Ficou comprovada a maior eficiência da técnica devido ao uso de arquivos-objeto relocáveis, pois os tempos de execução medidos são pelo menos uma ordem de magnitude inferiores aos observados em outras abordagens que operam sobre arquivos binários [Cho et al. 2007] [Angiolini et al. 2004].

Referências

Angiolini, F. et al. (2004). A Post-compiler Approach to Scratchpad Mapping of Code. *CASES (2004)*, pages 259–267.

Cho, H. et al. (2007). Dynamic Data Scratchpad Memory Management for a Memory Subsystem With an MMU. *LCTES (2007)*, pages 195–206.

Dally, W. et al. (2008). Efficient Embedded Computing. *IEEE Computer*, 41(7):27–32.

Dominguez, A., Udayakumaran, S., and Barua, R. (2005). Heap Data Allocation to Scratch-pad Memory in Embedded Systems. *Journal Embedded Computing*, 1(4):521–540.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Guthaus, M. R. et al. (2001). MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *Proceedings of the International Workshop on Workload*, pages 3–14.

Kandemir, M. et al. (2001). Dynamic Management of Scratch-pad Memory Space. *DAC (2001)*, pages 690–695.

McIlroy, R., Dickman, P., and Sventek, J. (2008). Efficient Dynamic Heap Allocation of Scratch-pad Memory. *ISMM (2008)*, pages 31–40.

Mendonca, A. K. I. et al. (2009). Mapping Data and Code into Scratchpads from Relocatable Binaries. *ISVLSI (2009)*, pages 157–162.

Panda, P., Dutt, N., and Nicolau, A. (2000). On-chip vs. Off-chip Memory: The Data Partitioning Problem in Embedded Processor-based Systems. *TODAES (2000)*, 5(3):682–704.

Santos, L. C. V., Volpato, D. P., Mendonca, A. K. I., and Güntzel, J. L. A Fresh Perspective on Scratchpad Usage. *Submetido ao IEEE Transactions on VLSI Systems (2011)*.

Steinke, S. et al. (2002a). Assigning Program and Data Objects to Scratchpad for Energy Reduction. *DATE (2002)*, 0:0409.

Steinke, S. et al. (2002b). Reducing Energy Consumption by Dynamic Copying of Instructions Onto Onchip Memory. *ISSS (2002)*, pages 213–218.

Thoziyoor, S. et al. (2008). CACTI 5.1. Technical report, Hewlett-Packard Laboratories.

Udayakumaran, S., Dominguez, A., and Barua, R. (2006). Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. *TECS (2006)*, 5(2):472–511.

Verma, M., Wehmeyer, L., and Marwedel, P. (2004). Dynamic Overlay of Scratchpad Memory for Energy Minimization. *CODES+ISSS (2004)*, pages 104–109.

Volpato, D. P., Mendonca, A. K. I., Güntzel, J. L., and Santos, L. C. V. (2011). Cache-tuning-aware allocation from binaries: a fresh perspective on scratchpad usage. In *SBCCI*.

Volpato, D. P., Mendonca, A. K. I., Santos, L. C. V., and Güntzel, J. L. (2010). A Post-compiling Approach that Exploits Code Granularity in Scratchpads to Improve Energy Efficiency. In *ISVLSI (2010)*, pages 127–132.