UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Gustavo Henrique Nihei

GERENCIAMENTO ENERGETICAMENTE EFICIENTE DE MEMÓRIA PARA MULTIPROCESSAMENTO EM CHIP EXPLORANDO MÚLTIPLAS SCRATCHPADS

Florianópolis

2012

Gustavo Henrique Nihei

GERENCIAMENTO ENERGETICAMENTE EFICIENTE DE MEMÓRIA PARA MULTIPROCESSAMENTO EM CHIP EXPLORANDO MÚLTIPLAS SCRATCHPADS

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

Florianópolis

2012

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

N691g Nihei, Gustavo Henrique
Gerenciamento energeticamente eficiente de memória para multiprocessamento em chip explorando múltiplas scratchpads [dissertação] / Gustavo Henrique Nihei ; orientador, José Luís Almada Güntzel. - Florianópolis, SC, 2012.
97 p.: il., grafs., tabs.

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

Inclui referências

1. Ciência da computação. 2. Gerenciamento de memória (Computação). 3. Processamento eletrônico de dados. 4. Multiprocessadores. I. Güntzel, José Luís Almada. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 681

Gustavo Henrique Nihei

GERENCIAMENTO ENERGETICAMENTE EFICIENTE DE MEMÓRIA PARA MULTIPROCESSAMENTO EM CHIP EXPLORANDO MÚLTIPLAS SCRATCHPADS

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

Florianópolis, 16 de fevereiro de 2012.

Ronaldo dos Santos Mello, Dr.
Coordenador

Banca Examinadora:

José Luís Almada Güntzel, Dr.
Orientador

Luiz Cláudio Villar dos Santos, Dr.
Coorientador

Marcelo de Oliveira Johann, Dr.

Mário Antônio Ribeiro Dantas, Dr.

À minha família

AGRADECIMENTOS

Aos meus pais, Celso e Marly, pelo amor e carinho incondicionais. Pela educação que recebi, de suma importância para minha formação pessoal e profissional. Também aos meus irmãos, Morgana e Leonardo, pela motivação e confiança demonstradas diariamente.

À minha namorada, Elisa, pelo amor e atenção a mim dedicados. Também pela paciência e compreensão durante os primeiros meses de nosso relacionamento, este que teve início em plena reta final de mestrado.

Ao Professor Dr. José Luís Almada Güntzel, pela orientação e incentivo durante todo este mestrado, além da amizade que data desde os tempos de minha graduação. A atenção e o esforço por ele dedicados tornaram a realização este trabalho possível.

Ao Professor Dr. Luiz Cláudio Villar dos Santos, pela coorientação deste trabalho. Suas sugestões e críticas foram fundamentais para a convergência deste trabalho.

Aos membros de banca, Professor Dr. Marcelo de Oliveira Johann e Professor Dr. Mário Antônio Ribeiro Dantas, por aceitarem o convite para avaliar este trabalho e pelas contribuições para sua melhoria.

Aos amigos e companheiros do grupo de pesquisa LAPS e NIME, pela convivência e auxílio para o desenvolvimento deste trabalho. Em particular, ao Renan Oliveira Netto, pelo grande apoio prestado, fundamental para a conclusão deste mestrado.

Aos amigos da Turma do Nheco-Nheco, pela amizade e companheirismo de longa data, com os quais pude compartilhar sucessos e frustrações que obtive com a realização deste trabalho.

Ao CNPq, no âmbito do Programa Nacional de Microeletrônica (PNM), pelo custeio parcial da execução deste trabalho (processos nº 131747/2009-6 e 130207/2010-1).

Ah, memória, inimiga mortal do meu repouso!

Miguel de Cervantes

RESUMO

A fim de proporcionar a alta capacidade de processamento requerida pelos dispositivos eletrônicos pessoais, sem ultrapassar os limites aceitáveis de potência e de consumo de energia, os sistemas em chip (SoCs) adotam o multiprocessamento. Para tanto, os SoCs possuem 2, 4 ou mais processadores, cada um com caches L1 privativas, conectados por meio de um barramento. Como o espaço de endereçamento visto pelos processadores é único, a programação do sistema pode assumir o modelo de memória compartilhada. A coerência entre as caches geralmente é assegurada pelo protocolo *snooping*.

Para tirar proveito do paralelismo dos SoCs multiprocessados (MPSoCs), aplicações são desenvolvidas com uso de múltiplas *threads* executando concorrentemente. Neste contexto, observa-se que os dados de pilha de uma dada *thread* são acessados somente pelo processador no qual a *thread* está executando. Desta forma, a relocação da pilha para memória *scratchpad* (SPM) pode ser explorada para reduzir a energia do subsistema de memória. Esta redução advém não apenas da menor energia gasta em cada acesso à pilha, mas também da redução das faltas nas caches L1 de dados e da penalidade imposta pelo protocolo *snooping*.

No presente trabalho propõe-se uma técnica para o gerenciamento dinâmico de dados de pilha em múltiplas SPMs, visando redução de energia no subsistema de memória em MPSoCs. A técnica utiliza um gerenciador totalmente em software, o qual é responsável por alocar e desalocar os dados de pilha de *thread* em SPM. A utilização da técnica dispensa intervenção do programador, pois as alterações necessárias no código da aplicação são realizadas por um compilador adaptado.

Foram obtidos resultados experimentais através da simulação de 400 aplicações geradas aleatoriamente, assumindo-se 20 plataformas multiprocessadas, totalizando 8000 casos de uso. Os resultados mostram que, variando-se o perfil das aplicações quanto à proporção de acessos a dados de pilha, a técnica proporciona reduções de energia no subsistema de memória entre 11% e 20%, em média, para plataformas com caches L1 de 32KB, e reduções entre 14,7% e 25,9%, em média, para plataformas com caches L1 de 64KB. Para plataformas com caches L1 de menor capacidade, a redução de energia é menor pois a penalidade de faltas nas caches L1 de instruções imposta pelo gerenciador torna-se relevante.

Palavras-chave: MPSoCs, *scratchpad*, gerenciamento *overlay*

ABSTRACT

In order to provide the high performance required by personal electronic devices, without exceeding the acceptable limits of power and energy consumption, systems-on-chip (SoC) adopt multiprocessing. Therefore, SoCs may have 2, 4 or more processors, each with its private L1 caches, connected through a bus. Since the address space seen by the processors is unique, the shared memory model is assumed in application development. Cache coherence is usually ensured by the snooping protocol.

To take advantage of parallelism in multiprocessor SoCs (MPSoCs), applications are developed using multiple concurrently-running threads. In this context, one can observe that the stack data of a given thread is accessed solely by the processor in which the thread is executed. Thus, the relocation of the stack to a scratchpad memory (SPM) can be exploited to reduce the memory subsystem energy. This reduction comes not only from lower energy spent in each stack access, but also from the reduction in both L1 data cache faults and snoops.

This work presents a technique for the dynamic management of stack data in multiple SPMs targeting the energy reduction in MPSoC memory subsystem. The technique uses a manager fully implemented in software which is responsible for allocating and deallocating thread stack data in SPMs. The technique does not require any programmer intervention, because the necessary changes to the application code are performed by a tailored compiler.

Experimental results were obtained through the simulation of 400 randomly generated applications, assuming 20 multiprocessor platforms, resulting in a total of 8000 use cases. The results show that by varying the applications' profile according to the proportion of accesses to data stack, the technique provides energy reductions in the memory subsystem ranging from 11% to 20%, on average, for platforms with 32KB L1 caches, and reductions ranging from 14.7% to 25.9%, on average, for platforms with 64KB L1 caches. For platforms with smaller L1 cache capacity, the energy reduction is smaller because the penalty imposed by the manager, in terms of L1 instruction cache faults, becomes relevant.

Keywords: MPSoCs, scratchpad, overlay management

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

CMP	*Chip Multiprocessing*
CRH	*Cashed-Region Hash*
DRAM	*Dynamic Random Access Memory*
DSP	*Digital Signal Processor*
ESP	*Essential Snoop Probes*
GPU	*Graphics Processing Unit*
ISA	*Instruction Set Architecture*
KB	*Kilobytes*
LLC	*Last-Level Cache*
LLM	*Limited Local Memory*
MMU	*Memory Management Unit*
MPSoC	*Multiprocessor System-on-Chip*
NSRT	*Non-Shared-Region Table*
NOB	*Non-overlay-based*
NoC	*Network-on-Chip*
OVB	*Overlay-based*
PGAS	*Partitioned Global Address Space*
RCA	*Region Coherence Array*
RISC	*Reduced Instruction Set Computer*
RS	*RegionScout*
RT	*RegionTracker*
SDRAM	*Synchronous Dynamic Random Access Memory*
SFT	*Snoop Filter Table*

SMP *Symmetric Multiprocessing*

SPM *Scratchpad Memory*

SoC *Systems-on-Chip*

SPMM *Scratchpad Memory Manager*

SRAM *Static Random Access Memory*

SSP *Selective Snoop Probes*

SUMÁRIO

1 INTRODUÇÃO

A extraordinária evolução da tecnologia de fabricação de circuitos integrados viabilizou a popularização de uma enorme variedade de produtos eletrônicos de consumo, tais como microcomputadores, televisores de LCD e de plasma, impressoras, *scanners* e telefones celulares, dentre outros. Em função de seus requisitos específicos em termos de desempenho ou tipo de processamento realizado, esses dispositivos fazem uso de sistemas em chip (*System on Chip* - SoC). Tipicamente, os SoCs são compostos por ao menos um processador de propósito geral e diversos blocos aceleradores, os quais podem ser processadores para uma única funcionalidade (por exemplo, codificador/decodificador JPEG) ou processadores programáveis para um domínio específico de aplicação (por exemplo, *Digital Signal Processor* - DSP).

Lançados mais recentemente no mercado, os dispositivos pessoais portáteis, tais como *smartphones*, *tablets* e consoles de jogos, passaram a ocupar uma fatia significativa do mercado de eletrônica de consumo, sendo inclusive responsáveis pela manutenção das altas taxas de expansão deste mercado nos últimos anos. Estes dispositivos requerem um poder de processamento bem superior aos produtos eletrônicos anteriores, uma vez que permitem a execução de diversas aplicações, estando as aplicações de multimídia sempre presentes. Estas últimas são intensivas tanto em termos de processamento quanto em termos de armazenamento e suprimento de dados. Não bastassem os requisitos de capacidade de processamento, os dispositivos pessoais portáteis ainda apresentam requisitos de baixo consumo de energia, uma vez que operam com baterias, cuja vida útil deve ser prolongada não apenas pela questão econômica, mas também para reduzir o impacto no ambiente natural causado pelo descarte das baterias usadas.

A fim de satisfazer os requisitos de desempenho e de baixo consumo de energia e, ao mesmo tempo, manter a competitividade econômica dos produtos, os dispositivos pessoais portáteis (e diversos outros produtos recentes) são construídos com SoCs multiprocessados (*Multiprocessor System on Chip* - MPSoC). Embora alguns autores, como Jerraya e Wolf (2005), sustentem que todos os SoCs são, na verdade, MPSoCs, este último termo passou a ser usado de maneira mais intensa a partir do uso de SoCs que possuem dois ou até quatro processadores simétricos em aplicações embarcadas. A figura 1 mostra o diagrama de blocos simplificado do OMAP5430, MPSoC da Texas Instruments (INSTRUMENTS, 2011), o qual é composto por dois processadores de

propósito geral para aplicações que exigem grande desempenho (ARM Cortex-A15), dois processadores para aplicações simples (ARM Cortex-M), um processador gráfico (*Graphics Processing Unit* - GPU), um DSP e diversos outros blocos aceleradores. Este MPSoC é voltado para aplicações multimídia e deverá equipar *smartphones* e *tablets* que serão lançados no mercado ao longo do ano de 2012.

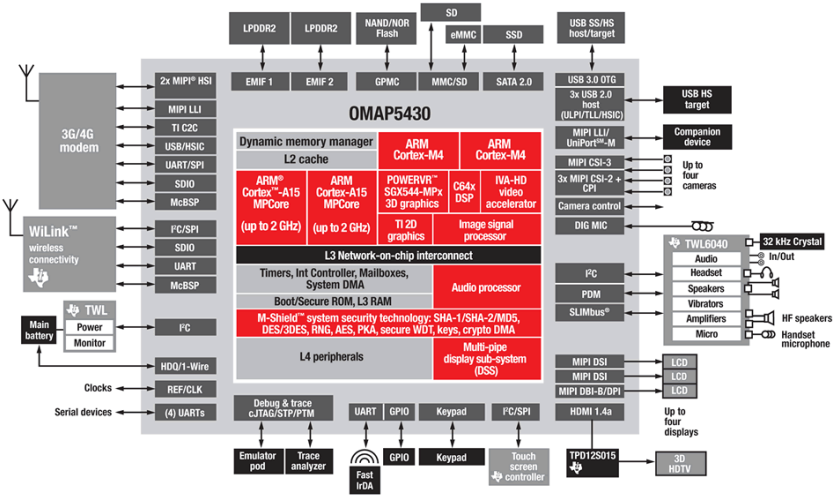

Figura 1: Exemplo de MPSoC heterogêneo, TI OMAP5430. FONTE: (INSTRUMENTS, 2011)

O uso de blocos aceleradores e de processadores específicos (GPUs, DSPs etc) provê aos MPSoCs a característica de **eficiência energética** requerida pelos dispositivos pessoais portáteis, a qual reside em atingir o desempenho exigido pelas aplicações, consumindo o mínimo de energia possível. Por outro lado, a demanda por capacidade de processamento de propósito geral nos dispositivos pessoais está aumentando constantemente, como decorrência do aumento da variedade de aplicações disponíveis aos usuários. Em função disso, o número de processadores de propósito geral nos MPSoCs tende a crescer nos próximos anos. Atualmente, já existem MPSoCs com dois e até quatro processadores de propósito geral simétricos, cada um destes com uma cache L1 de dados e uma cache L1 de instruções. A conexão entre os processadores é feita por meio de um barramento, ao qual também pode estar conectada uma memória cache L2 para dados e instruções, ou portas de saída para a

conexão com uma memória externa ao chip. Geralmente, o modelo de memória compartilhada é adotado, o que implica que todos os processadores compartilham um espaço de endereçamento único. Este modelo é particularmente interessante porque permite o desenvolvimento de aplicações paralelas com um esforço reduzido mediante o uso de técnicas de programação consagradas, como por exemplo *threads*. Por outro lado, a existência de memórias cache L1 privativas aos processadores e a adoção do modelo de memória compartilhada resulta no surgimento de um problema conhecido como **coerência de caches**. O problema da coerência de caches e o protocolo mais utilizado para sua solução são detalhados na seção 1.2.

Durante muito tempo o foco das otimizações de sistemas computacionais esteve centrado no processador. Contudo, trabalhos recentes têm colocado em evidência o fato de que o subsistema de memória é responsável por uma grande parcela do consumo energético de um sistema. A figura 2 dá uma ideia aproximada da parcela de energia consumida para suprir dados e instruções em sistemas monoprocessados. Em sistemas multiprocessados é de se esperar que o consumo energético do subsistema de memória seja ainda mais importante. Em função disto, a seção 1.1 descreve as principais características dos componentes do subsistema de memória.

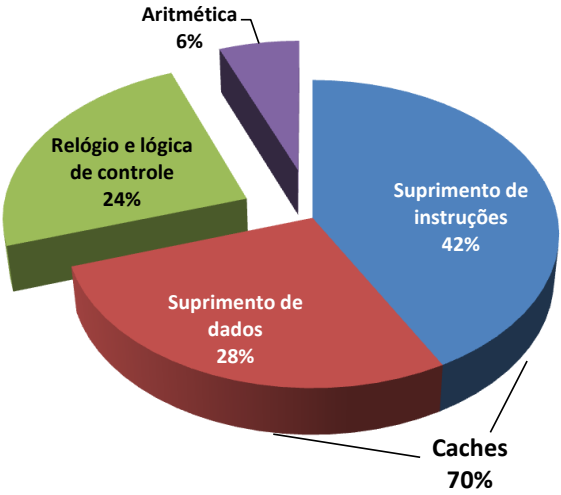

Figura 2: Distribuição de energia em um processador embarcado (DALLY et al., 2008)

1.1 COMPONENTES DO SUBSISTEMA DE MEMÓRIA

1.1.1 Memória principal

A memória principal (daqui em diante tratada por MP) é o maior dispositivo de armazenamento temporário de um sistema computacional. Predominantemente construída com a tecnologia DRAM, a MP é uma memória de acesso aleatório que pode residir tanto internamente quanto externamente ao SoC.

1.1.2 Memória cache

Uma memória cache é um bloco de memória utilizado pelo processador como um armazenamento temporário para cópias de dados e/ou instruções que estão presentes na MP. O gerenciamento de seu contéudo dispensa interferência do usuário, sendo realizado implicitamente por recursos adicionais de hardware, como conjunto de *tags*, comparadores e multiplexadores, conforme mostrado na figura 3.

Geralmente, as células das memórias cache são construídas com a tecnologia *Static Random Access Memory* (SRAM). Essa tecnologia possui um custo por KB mais elevado que o de uma *Dynamic Random Access Memory* (DRAM), fator esse que limita a capacidade de armazenamento. Em compensação, isso garante um acesso de alta velocidade às suas células (geralmente, sua latência pode ser acomodada dentro de 1 ciclo de processador para o caso de uma cache L1) e a torna mais eficiente energeticamente que a memória principal.

1.1.3 Memória de rascunho (SPM)

Memórias de rascunho (*Scratchpad Memory* - SPM), também conhecida por memória fortemente acoplada (*Tightly Coupled Memory* - TCM) são, assim como as caches, memórias pequenas, internas ao SoC, e construídas na tecnologia SRAM. Seu conteúdo, porém, deve ser gerenciado explicitamente, pois uma SPM não apresenta recursos adicionais de hardware além das estruturas de armazenamento de dados, como mostra a figura 4. Como não dispõe de arranjo de etiquetas (*tags*) nem de estruturas lógicas de comparação, as SPMs são energeticamente mais eficientes que as caches. A grande diferença entre uma SPM e uma cache é que, enquanto o tempo de acesso a uma cache depende

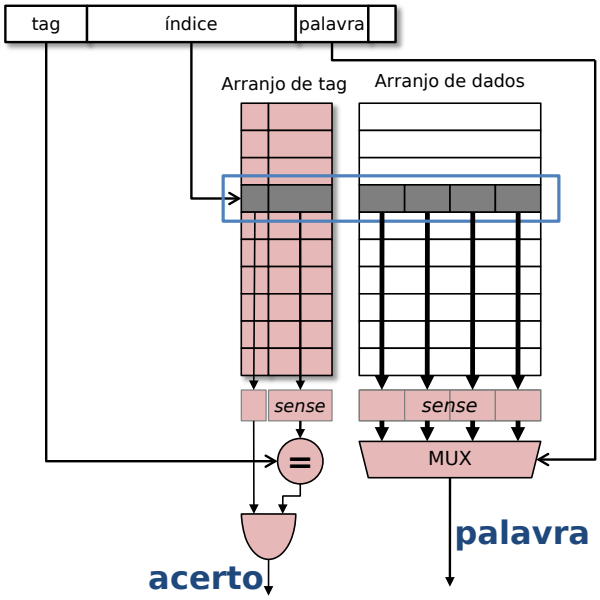

Figura 3: Exemplo de memória cache com mapeamento direto. FONTE: (VOLPATO, 2010)

da ocorrência ou não de uma falta de bloco, o acesso a uma SPM tem latência constante, geralmente um ciclo de processamento. Um exemplo recente de plataforma (KEUTZER et al., 2000) que emprega SPMs (ou TCMs) é o ARM Cortex-R7 (ARM Corporation, 2011c), cujo diagrama de blocos está mostrado na figura 5.

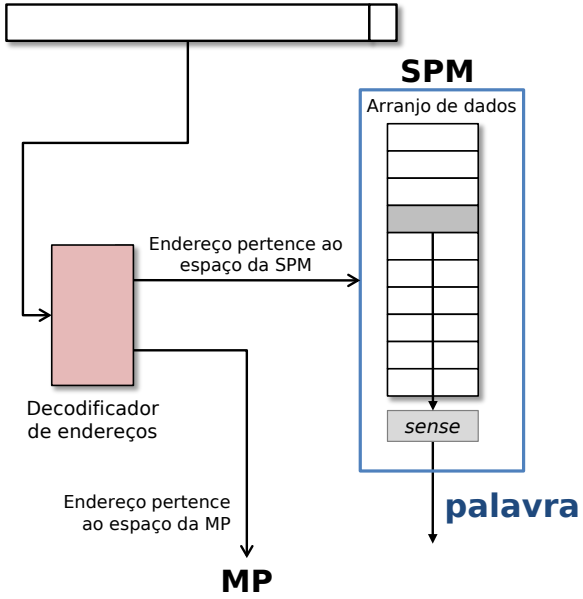

Figura 4: Exemplo de memória de rascunho (SPM). FONTE: (VOLPATO, 2010)

Existem duas possibilidades para se construir uma arquitetura de SPM em multiprocessadores. A arquitetura *Limited Local Memory* (LLM), por exemplo, é a empregada no IBM CELL BE (CHEN et al., 2007). O multiprocessador IBM CELL BE é composto por uma unidade de processamento de propósito geral, denominada PPE (*Power Processor Element*) e diversas unidades mais simples de processamento denominadas SPEs (*Synergistic Processing Elements*). Cada uma das SPEs possui uma SPM privativa. A SPE pode acessar apenas sua memória local, e um acesso remoto à memória local de outra SPE é dado como inválido. Qualquer transferência de dados entre SPEs (ou de/para a DRAM *off-chip*) deve ser explicitamente gerenciada pelo programador através de DMA.

Em outra arquitetura de SPM em multiprocessadores, conforme

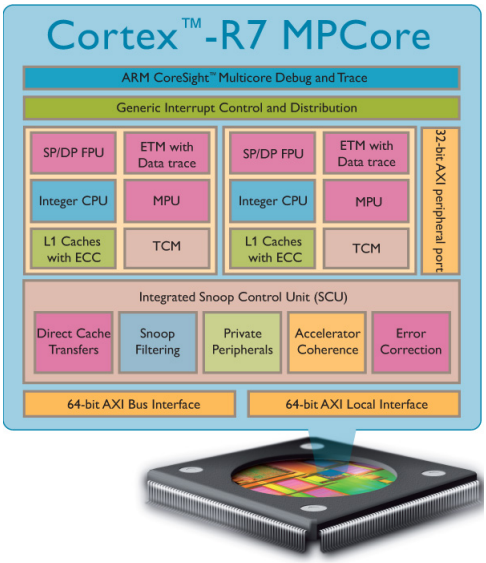

Figura 5: Diagrama de blocos da plataforma ARM Cortex-R7 MPCore. FONTE: (ARM Corporation, 2011c)

exemplificado pelo diagrama de blocos da figura 6, mesmo distribuídas fisicamente pelos processadores, as SPMs compartilham o mesmo espaço de endereçamento lógico, organizadas como um sistema *Partitioned Global Address Space* (PGAS) (COARFA et al., 2005). Nesse modelo, o espaço de endereçamento é logicamente compartilhado entre todos os processadores, porém uma parte dele é local para cada processador, sendo esta reservada para a respectiva SPM local.

1.2 COERÊNCIA DE CACHE

Existem dois aspectos importantes relacionados ao problema da coerência de cache (LILJA, 1993). O primeiro é o modelo do sistema de memória apresentado ao programador. O segundo aspecto é o protocolo utilizado pelo sistema para manter a coerência entre as caches e a memória principal.

Um sistema com caches consistentes é aquele que garante que o valor retornado por uma instrução *load* é sempre o valor dado pela instrução *store* mais recente para o mesmo endereço (CENSIER; FE-

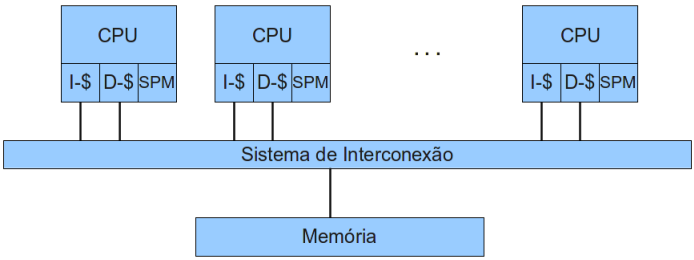

Figura 6: Exemplo de um sistema com arquitetura de SPM PGAS

AUTRIER, 1978). Essa definição causa confusão, pois o sentido de "instrução mais recente" não é preciso quando *loads* e *stores* ocorrem em processadores diferentes funcionando assincronamente entre si. Devido a atrasos em diferentes partes do sistema de interconexão entre processadores e memórias, e inclusive internamente aos mesmos, cada processador e cada módulo de memória pode enxergar uma ordem diferente dos eventos. O *modelo de consistência* de um processador define a visão do programador em relação à ordenação por tempo dos eventos que ocorrem em diferentes processadores. Constituem tais eventos operações de escrita e leitura à memória e de sincronização. Conforme o sistema faz menos restrições ao programador em relação à ordem dos eventos, há um maior potencial para sobrepor as operações de diferentes processadores entre si, e entre operações do mesmo processador, aumentando assim o desempenho do sistema. Contudo, o custo desse maior desempenho é o peso colocado sobre o programador (ou compilador) para garantir que as dependências entre as operações não sejam violadas.

Um aspecto relacionado ao modelo de consistência de memória é o protocolo usado pelo sistema para garantir que os processadores não acessem dados defasados. Em um multiprocessador com memória compartilhada, cada um dos processadores pode acessar qualquer local do espaço comum de endereçamento por meio de uma única instrução de leitura (*load*) ou escrita (*store*). Considerando que cada processador possua uma cache de dados privativa, uma cópia de um mesmo bloco compartilhado de memória pode estar presente em uma ou mais caches ao mesmo tempo. Quando um bloco compartilhado de memória é escrito por algum processador, essa alteração deve ser propagada para todos os outros processadores que possuem uma cópia do mesmo bloco em cache

para garantir que ninguém utilize uma versão antiga.

Por exemplo, suponha um sistema com três processadores, cada um com sua cache de dados privativa, no qual ocorre a sequência de eventos ilustrada na tabela 1.

Tabela 1: Sequência de leituras e escritas

Tempo	P_0	P_1	P_2
t_1	lê X		
t_2		lê X	
t_3	escreve 16 em X		
t_4	lê X	lê X	lê X

Assuma que inicialmente todas as caches estejam vazias e que o local de memória X contenha o valor 12. Depois que as duas leituras terminarem em t_2, as caches dos processadores P_0 e P_1 terão o valor 12 para a variável armazenada em X, conforme a figura 7.

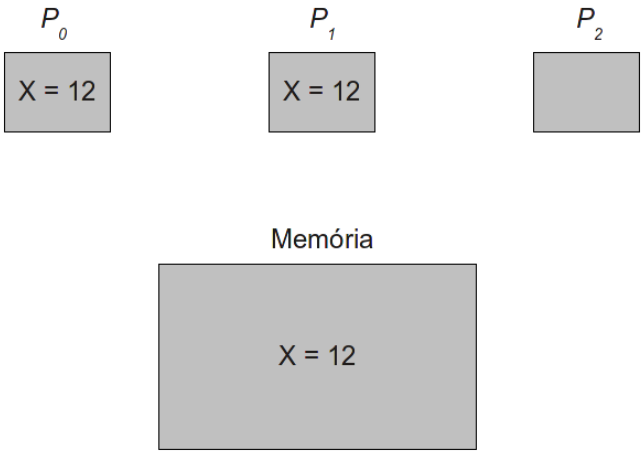

Figura 7: Conteúdo das caches após t_2

Em t_3, o processador P_0 armazena o valor 16 em X. Em um sistema sem um protocolo de coerência de caches, esse valor estará atualizado somente na cache de P_0. Quando P_1 reler X em t_4, será lido o

antigo valor 12 de sua cache, como está mostrado na figura 8. Do mesmo modo, o processador P_2 também lerá o valor antigo, pois a memória principal também não se encontra atualizada com o valor mais recente escrito por P_0.

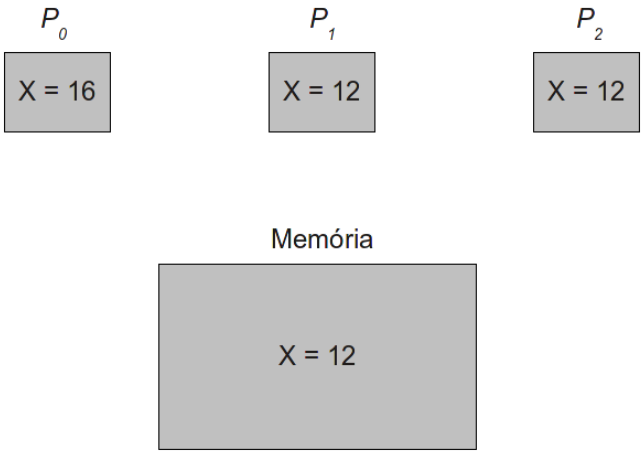

Figura 8: Conteúdo das caches após t_4

O protocolo de coerência de caches é necessário para garantir que os valores antigos de X nas caches de outros processadores e na memória principal não sejam propagados em futuras operações de leitura.

Segundo Lilja (1993), uma maneira útil de visualizar a relação entre o modelo de consistência de memória e o protocolo de coerência de caches é que o protocolo de coerência garante que todas as caches vejam todas as escritas a um bloco *específico* na mesma ordem lógica. O modelo de consistência, por outro lado, define ao programador a ordem de escritas a blocos *diferentes* que são percebidas por cada um dos processadores. Isto é, se o programador seguir as regras do modelo de consistência para o sistema em uso, o protocolo de coerência faz com que o valor retornado por qualquer *load* seja o valor garantido pelo modelo de consistência.

Existem diversos protocolos para tratar o problema de coerência de caches. A solução recomendada para um determinado sistema multiprocessado depende de diversos fatores, como número de processadores, propósito de utilização e o custo esperado do sistema. Os dois métodos mais utilizados para garantir coerência são implementações em *hardware*

e protocolos em *software* (LAWRENCE, 1998). Em uma implementação em *hardware*, dispositivos especiais são adicionados ao sistema para detectar acessos à cache e implementar um protocolo de consistência adequado. Já a abordagem em *software* depende do compilador para gerar código consistente.

Protocolos de coerência baseada em *snooping* (bisbilhotagem, em inglês) tornaram-se populares em multiprocessadores onde processadores e caches estão acoplados por meio de uma única memória compartilhada. Isso porque dependem de um sistema de interconexão compartilhado de baixa latência entre os processadores e os módulos de memória. Um processador, ao "bisbilhotar" as referências à memória de outros processadores, pode detectar quando um bloco que ele possui em cache foi modificado por outro processador.

Há dois modos de se manter a coerência através de *snooping*. Um método é assegurar que um processador tenha acesso exclusivo a um bloco antes que ele o sobrescreva. Esse protocolo é conhecido como *write invalidate*, pois ele invalida todas as outras cópias em uma escrita. O acesso exclusivo garante que não exista nenhuma cópia legível ou modificável de um bloco enquanto a escrita ocorre, ou seja, todas as outras cópias do bloco nas demais caches estão inválidas. Uma alternativa ao protocolo por invalidação baseia-se na atualização de todas as cópias de um bloco nas demais caches quando tal bloco for escrito. Esse protocolo é chamado de *write update* ou *write broadcast*. Como esse protocolo deve transmitir todas as escritas para as linhas compartilhadas de cache, a banda consumida é consideravelmente maior. Segundo Patterson e Hennessy (2008), devido a essa razão, todos os multiprocessadores recentes optaram por implementar o protocolo por invalidação. No caso de multiprocessadores embarcados, a utilização do protocolo *write update* torna-se inviável devido ao alto consumo de energia associado a cada *broadcast* de dados.

1.3 ESCOPO DESTE TRABALHO

A importância que a eficiência energética assumiu nos últimos anos, a crescente demanda por processamento requerida pelos dispositivos pessoais e a contribuição relevante do subsistema de memória no consumo energético justificam a investigação de métodos de otimização de energia no subsistema de memória de MPSoCs.

Conforme mencionado anteriormente, o modelo de memória compartilhada, geralmente adotado nos MPSoCs, viabiliza o uso de técnicas

de paralelização de aplicações consagradas, tais como programação com *threads*. Partindo-se da observação de que os dados da pilha de execução de uma *thread* são essencialmente privados, especula-se que a alocação destes dados em SPM privativa ao processador no qual a *thread* executa deve resultar em redução de energia, uma vez que uma SPM requer menos energia por acesso do que uma cache de mesma capacidade.

Este trabalho apresenta uma técnica para o gerenciamento dinâmico de dados da pilha de execução de *thread* em SPM, com o intuito de reduzir a energia consumida no subsistema de memória de MPSoCs. Embora tal estratégia pudesse ser implementada com o suporte de um bloco de hardware dedicado, escolheu-se implementar um gerenciador de SPM totalmente em software, como forma de minimizar a intervenção do programador da aplicação, tornando o método o mais transparente possível. A aplicação da técnica requer apenas a existência de uma SPM privativa a cada processador. Além do desenvolvimento do gerenciador, foi necessário adaptar um compilador, tornando-o ciente da técnica para o conjunto de instruções do processador escolhido para prova de conceito. A validação da técnica exigiu um substancial esforço para a criação da infraestrutura experimental, a qual envolveu a modelagem de 20 plataformas, cada uma com quatro processadores e hierarquia de memória, e o desenvolvimento de um gerador configurável de aplicações sintéticas.

1.4 PRINCIPAIS CONTRIBUIÇÕES

Este trabalho apresenta as seguintes **contribuições técnicas**:

- Criação de infraestrutura para a inclusão de SPMs privativas no *framework* de simulação arquitetural GEM5 (BINKERT et al., 2011);

- Criação de uma técnica de alocação dinâmica de dados de pilha de *threads* para sistemas multiprocessados com arquitetura PGAS, baseada no trabalho de Kannan et al. (2009), que não requer modificação manual do código-fonte da aplicação;

- Implementação de um gerador aleatório de aplicações, baseado no trabalho de Rambo, Henschel e Santos (2011), permitindo a criação automática de aplicações cujos perfis de acessos à memória são determinados pelo usuário.

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

- Identificação da correlação entre redução média de energia proporcionada pela técnica e proporção de acessos a dados de pilha de *thread*;
- Identificação da máxima redução de energia que a técnica proporciona para cada configuração de subsistema de memória avaliada;
- Identificação da correlação entre redução média de energia proporcionada pela técnica e características das memórias cache L1;
- Avaliação das penalidades impostas pela técnica;
- Avaliação da redução do número de requisições de coerência de cache (*snoops*) para as configurações de subsistema de memória avaliadas.

1.5 ORGANIZAÇÃO DESTA DISSERTAÇÃO

O restante desta dissertação está organizado da seguinte forma.

O Capítulo 2 aborda os trabalhos correlatos de redução de energia no subsistema de memória de MPSoCs, os quais utilizam uma dentre as duas principais estratégias, quais sejam: alocação em memória de rascunho (*Scratchpad Memory* - SPM) de elementos de programa frequentemente referenciados e otimização do protocolo de coerência de caches (*snooping*).

O Capítulo 3 descreve a técnica de otimização proposta, a qual se baseia na alocação *overlay* de dados de pilha de *threads* para reduzir a energia despendida no subsistema de memória de MPSoCs. Além da motivação para se propor tal técnica, são descritas as premissas para a sua aplicação e as alterações necessárias ao compilador para viabilizar o seu uso.

O Capítulo 4 detalha a implementação da técnica proposta. Além de descrever as modificações no compilador utilizado, também são mostradas as características da implementação de um protótipo de gerenciador *overlay* de pilha de *threads* em SPM. A seguir, ele descreve a infraestrutura desenvolvida para a validação experimental da técnica, a qual é composta por 20 plataformas multiprocessadas modeladas no *framework* de simulação denominado de GEM5 (BINKERT et al., 2006, 2011). Após, é apresentado o método que foi utilizado para gerar

as 400 aplicações sintéticas utilizadas na validação do método. Por fim, este Capítulo mostra as equações utilizadas para calcular o consumo de energia no subsistema de memória, bem como os valores de energia por acesso para cada componente de memória, conforme estimados pela ferramenta CACTI 5.3 (WILTON; JOUPPI, 1996).

O Capítulo 5 descreve a configuração experimental utilizada, apresenta e justifica os resultados mais relevantes obtidos a partir das 8000 simulações realizadas para validar a técnica proposta.

O Capítulo 6 apresenta as conclusões deste trabalho e elenca trabalhos futuros.

2 OTIMIZAÇÃO DE ENERGIA NO SUBSISTEMA DE MEMÓRIA DE MPSOCS

Este capítulo aborda duas estratégias para a otimização de energia no subsistema de memória de MPSoCs. A primeira estratégia consiste na alocação em memória de rascunho (*Scratchpad Memory* - SPM) de elementos de programa frequentemente referenciados. A segunda estratégia consiste na otimização do protocolo *snooping*. Para cada uma destas duas estratégias são brevemente descritas as técnicas mais relevantes encontradas na literatura. No caso das técnicas de alocação em SPM, são consideradas apenas as técnicas com potencial de aplicação em multiprocessadores simétricos em chip, por serem estes o foco do presente trabalho.

2.1 TÉCNICAS DE ALOCAÇÃO EM SPM

As técnicas para alocação em SPM são influenciadas por uma série de características relacionadas à aplicação, aos elementos de programa, aos arquivos de entrada considerados e também ao próprio processo de alocação (VOLPATO, 2010). Essas características tem impacto direto na alocação e no eventual ganho de desempenho e melhora energética decorrentes.

As principais características que influenciam as técnicas para alocação em SPM são (VOLPATO, 2010):

- quais **elementos de programa** são considerados para alocação;
- a origem, tipo e granularidade destes elementos;
- a fase em que ocorre a alocação dos elementos para a SPM, se em tempo de compilação ou de pós-compilação;
- o tipo de arquivo de entrada utilizado (fonte, objeto ou executável);
- a abordagem de alocação.

Designam-se por **elementos de programa** trechos de código ou de dados que possuam significado lógico. Desta forma, procedimentos correspondem a **elementos de código**, ao passo que variáveis e estruturas de dados são **elementos de dados**. Os elementos de dados são ainda classificados em globais escalares, globais não-escalares, elementos de pilha e elementos de *heap*.

Segundo Volpato (VOLPATO, 2010) a maioria das técnicas consideram elementos de código e de dados como candidatos à alocação em SPM. Entretanto, dentre as técnicas que consideram elementos de dados, a maioria é incapaz de tratar os quatro tipos de elementos de dados citados anteriormente. Por uma questão de simplicidade, no restante deste texto elementos de programa serão referenciados simplesmente por **elementos**.

Com relação à origem, os elementos de programa (código e dados) podem pertencer à própria **aplicação**, ou podem ter sido desenvolvidos por terceiros e fornecidos como **biblioteca** de arquivos-objeto pré-compilados. Poucas técnicas são capazes de considerar elementos de biblioteca para alocação em SPM (VOLPATO, 2010). As possíveis granularidades dependem do tipo de elementos considerados. No caso de elementos de código, as possíveis granularidades são **procedimentos**, ***traces*** e **blocos básicos**. No caso de elementos de dados, o conceito de granularidade somente se aplica aos dados não-escalares e consiste em considerar o conjunto completo dos dados (**granularidade plena**) ou **blocos de dados**.

As técnicas de alocação em SPM são concebidas segundo uma das seguintes abordagens: *Non-overlay-based* (NOB) e *Overlay-based* (OVB).

Na abordagem NOB os elementos são alocados em SPM em tempo de carga da aplicação para a memória e lá permanecem durante toda a sua execução. A escolha dos elementos a serem alocados é realizada através de um mapeamento realizado em tempo de compilação ou pós-compilação. Devido a essa característica, as técnicas NOB são capazes de considerar apenas código e dados estáticos (globais escalares e não-escalares) como elementos candidatos à alocação em SPM, uma vez que os endereços de tais elementos podem ser determinados antes da execução da aplicação.

A abordagem OVB, por sua vez, aloca os elementos em SPM em tempo de execução da aplicação, escolhendo somente os elementos muito referenciados. Quando algum elemento deixa de ser muito referenciado, ele é removido da SPM, sendo substituído por outro elemento muito referenciado. Com tal funcionamento, a abordagem OVB tende a capturar os padrões de acesso da aplicação de maneira mais completa do que a abordagem NOB. É importante observar que, devido ao fato de realizar a alocação em tempo de execução, as técnicas OVB são capazes de considerar para alocação em SPM não somente código e dados estáticos, mas também dados dinamicamente alocados, como dados de pilha e *heap*.

A alocação de um determinado elemento por meio de uma técnica OVB ocorre através da cópia de seu conteúdo da memória principal para a SPM. Já a desalocação se dá apenas pela sobreposição desse elemento por um outro elemento a ser referenciado. Esse processo é válido para elementos de código e dados constantes (somente leitura). Para o caso de dados modificáveis, é necessário que o elemento da SPM a ser sobreposto seja previamente copiado de volta para a memória principal, para que o conteúdo atualizado do elemento não se perca. Esse processo de alocação e desalocação repete-se diversas vezes durante a execução da aplicação. A escolha (mapeamento) dos elementos a serem alocados pode ser realizada em tempo de compilação, pós-compilação, ou até mesmo em tempo de execução. Um fator que pode influenciar na decisão da etapa do mapeamento é o tipo dos dados candidatos à alocação. Para dados dinâmicos, por exemplo, geralmente é adotado um mapeamento em tempo de execução. A cópia dos elementos para SPMs pode ser feita através:

- da inserção de instruções convencionais de *load/store* no código da aplicação;
- de hardware adicional, controlado por instruções personalizadas inseridas no código da aplicação;
- de funções de gerenciamento de memória dinâmica, conscientes de SPM, como **malloc** e **free** da biblioteca **libc**;
- do redirecionamento dos ponteiros de memória de uma aplicação, apontando para a faixa de endereços abrangida pela SPM.

Considerando que o escopo do presente trabalho se restringe à alocação de dados de pilha em SPM, a seguir são descritos os trabalhos correlatos mais relevantes, os quais utilizam a abordagem OVB. Para detalhes sobre outros trabalhos relevantes em alocação para SPM, inclusos os trabalhos que utilizam a abordagem NOB, o leitor deve consultar (VOLPATO, 2010).

2.1.1 Exemplos de técnicas OVB correlatas ao presente trabalho

Diversos trabalhos implementam a abordagem OVB em sistemas que contam com auxílio de hardware através de uma *Memory Management Unit* (MMU). Park, Park e Ha (2007) propuseram uma técnica

para alocação de dados de pilha em SPM através de uma modificação da MMU. Partindo da premissa de que a pilha está localizada sempre próxima da região de memória indicada pelo *stack pointer*, o método dos autores faz com que a SPM acompanhe o movimento do *stack pointer* alterando a tabela de páginas da MMU em tempo de execução. Os autores definem dois casos para explicar a política de gerenciamento da técnica. O primeiro caso ocorre quando a pilha cresce além da área de memória da SPM e o segundo caso ocorre quando um acesso a uma área inferior à da SPM é realizado de dentro da pilha. Para realizar esse gerenciamento, tal método utiliza os bits de páginas referentes à permissão de acesso. Isso levanta preocupações quanto à segurança do sistema, pois qualquer aplicação maliciosa pode tirar vantagem da técnica para ter acesso a outros processos em execução.

Cho et al. (2007) e Egger, Lee e Shin (2006, 2008) propuseram uma arquitetura de memória com particionamento horizontal composta por uma SPM e uma mini-cache. Em Cho et al. (2007) e Egger, Lee e Shin (2006), através de conhecimento oriundo de *profiling* das aplicações, é feito o mapeamento de dados globais escalares, não-escalares e de pilha e o redirecionamento dos endereços de memória é realizado com o auxílio de uma MMU. Já Egger, Lee e Shin (2008) propõem uma técnica OVB de pós-compilação para a alocação de código. O código da aplicação é dividido em duas regiões: *cacheable*, que reside no espaço de endereçamento da memória principal, e *pageable*, a ser copiado para a SPM sob demanda, também sendo auxiliado pela MMU do sistema. Apesar de essas técnicas serem bastante promissoras, elas requerem modificações arquiteturais, reduzindo a aplicabilidade das mesmas.

Kannan et al. (2009) apresentam uma técnica de gerenciamento circular dinâmico de dados de pilha em SPMs, completamente em software e que dispensa a realização de *profiling*. O gerenciamento da pilha é realizado pelo *Scratchpad Memory Manager* (SPMM), cuja função principal é monitorar possíveis *overflows* e acomodar novos quadros (*frames*) em uma dada SPM. O SPMM é implementado como uma biblioteca ligada estaticamente à aplicação. As chamadas aos procedimentos do gerenciador são inseridas na aplicação em pares, antes e depois de cada chamada de função, conforme ilustrado na figura 9. O gerenciador utiliza uma tabela que contém os endereços de procedimentos e os tamanhos dos respectivos quadros de pilha, gerados em tempo de compilação. Os procedimentos do SPMM precisam de espaço em pilha para sua própria execução, o qual é alocado em uma área reservada da SPM.

As técnicas relatadas nos três parágrafos anteriores foram pro-

```
<código ASM altera $sp para pilha do SPMM>
spmm_check_in(F2);
<código ASM altera $sp para pilha de programa>
F2();
<código ASM altera $sp para pilha do SPMM>
spmm_check_out(F2);
<código ASM altera $sp para pilha de programa>
```

Figura 9: Chamadas de biblioteca inseridas na aplicação (KANNAN et al., 2009)

postas para sistemas monoprocessados, sendo que sua aplicabilidade em sistemas multiprocessados não é abordada pelos respectivos artigos. Por outro lado, nos parágrafos seguintes são abordadas técnicas desenvolvidas para sistemas multiprocessados.

Bai e Shrivastava (2010) e Bai, Shrivastava e Kudchadker (2011) estenderam a aplicação da técnica proposta por Kannan et al. (2009) ao multiprocessador IBM CELL BE (CHEN et al., 2007) para a alocação de dados de *heap* e pilha, respectivamente. A plataforma multiprocessada utilizada é um modelo típico de arquitetura LLM, onde a comunicação entre as tarefas é tratada explicitamente. Além disso, a utilização de memória local em uma arquitetura LLM é, ao contrário de MPSoCs com SPM, uma necessidade, e não uma otimização. Segundo os autores, o objetivo de ambos os trabalhos é apenas a compilação automática de aplicações para processadores que apresentem tal arquitetura de memória. Assim sendo, os autores não apresentaram quaisquer resultados experimentais que comprovassem a eficiência energética da técnica.

Seguindo a mesma linha dos dois trabalhos citados no parágrafo anterior, Deng et al. (2011) propõem um *framework* similar para gerenciamento de SPMs para multiprocessadores que suportam o modelo de memória em chip PGAS. Os dados de *heap* mais frequentemente referenciados são gerenciados de modo semi-automático, podendo ser distribuídos por todas as SPMs do sistema de modo a otimizar a latência média de acesso pelos processadores. Apesar da técnica não depender diretamente da utilização de técnicas de *profiling* nem de modificações no compilador, a escolha dos conjuntos de dados a serem alocados não é realizada automaticamente. Além disso, também é necessário fazer a modificação manual do código-fonte da aplicação, o que dificulta sua utilização.

As técnicas de alocação OVB baseadas em um gerenciador em

software e suas características estão sumarizadas na tabela 2 quanto à arquitetura-alvo, elementos de programa alocados e nível de interferência do programador necessário para seu melhor aproveitamento.

2.2 OTIMIZAÇÃO ENERGÉTICA DO PROTOCOLO *SNOOPING*

Protocolos de coerência de cache baseados em hardware podem ser classificados de acordo com duas dimensões ortogonais (STENSTRöM, 1990): quanto ao sistema de interconexão empregado no multiprocessador, ou quanto à política de coerência de cache.

Em sistemas multiprocessados com poucas dezenas de núcleos é comum a utilização de barramentos como sistema de comunicação entre os diversos núcleos e periféricos do sistema. Para garantir a coerência dos dados entre suas caches, um processador transmite mensagens de faltas através de *broadcasting* pelo barramento, e os processadores remotos verificam se podem ou não atender à requisição. Esse processo, descrito em maiores detalhes no capítulo 1, é conhecido como *snooping*. No caso de o sistema utilizar um meio de interconexão não compartilhado (uma NoC, por exemplo), protocolos de *snooping* são substituídos por protocolos baseados em diretório.

Quanto às políticas de coerência de cache, existem basicamente duas opções: uma baseada em invalidações (*write-invalidate*) e outra baseada em atualizações (*write-update*). Na primeira opção, sempre que uma determinada linha estiver para ser escrita, o protocolo invalida todas as cópias da mesma nas outras caches. Já na segunda opção, o protocolo atualiza tais linhas com o valor novo, o qual está sendo escrito. Loghi, Poncino e Benini (2006) realizam uma análise comparativa, em termos de energia e desempenho, de diferentes esquemas para suportar coerência de cache em MPSoCs. Além de tratar de políticas de coerência, também são consideradas diferentes políticas de escrita das caches. Para a análise, os autores consideraram diferentes esquemas de coerência baseados em *snooping*, um baseado em *software* que previne o armazenamento em cache de dados compartilhados, e um outro baseado em sistema operacional. Este último demonstrou ter um custo muito alto para aplicação em sistemas embarcados. Já os esquemas baseados em *snooping*, por serem baseados em hardware, oferecem um desempenho maior que as outras alternativas, com um acréscimo no consumo de energia proporcional ao tráfego de mensagens de coerência pelo sistema.

Diversos trabalhos encontrados na literatura demonstram que a grande maioria das requisições de coerência de memória não retornam

Tabela 2: Técnicas de alocação em SPM baseadas em um gerenciador em software

Técnica	Elementos de programa	Arquitetura-alvo	Interferência do programador
Kannan et al. (2009)	Pilha	Monoprocessada	Não
Bai e Shrivastava (2010)	Heap	Multiprocessada LLM	Não
Bai, Shrivastava e Kudchadker (2011)	Pilha	Multiprocessada LLM	Não
Deng et al. (2011)	Heap	Multiprocessada PGAS	Sim
Este trabalho	Pilha	Multiprocessada PGAS	Não

o bloco desejado, o que faz com que tanto o *broadcast* das requisições quanto as pesquisas às caches do sistema sejam realizados desnecessariamente. Observando a figura 10, retirada do trabalho de Ballapuram, Sharif e Lee (2008), podemos ver que, para a aplicação *cholesky* (*benchmark* SPLASH-2 (WOO et al., 1995)), a chance de uma requisição de coerência ser atendida por algum processador remoto é inferior a 10%.

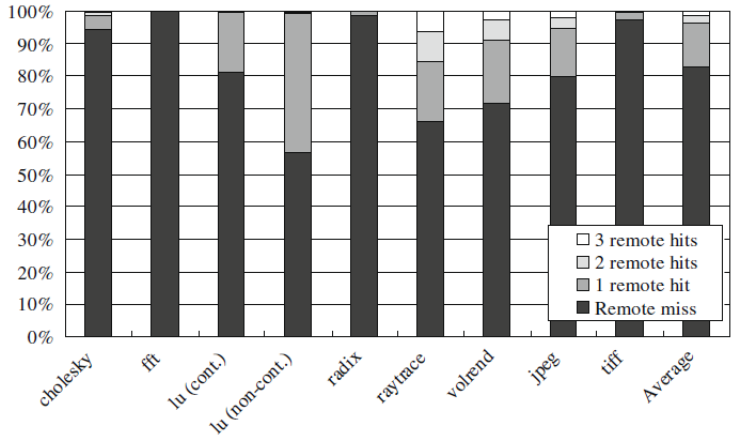

Figura 10: Distribuição de acertos a caches remotas em um MPSoC composto por 4 processadores (BALLAPURAM; SHARIF; LEE, 2008)

Para detectar e eliminar esses *snoops* desnecessários, várias técnicas foram propostas. Nesta seção, as técnicas mais relevantes são agrupadas e classificadas de acordo com o método adotado para reduzir o tráfego de mensagens de coerência:

- Filtragem de *snoops*
- Abordagens híbridas cientes da aplicação

2.2.1 Filtragem de *snoops* desnecessários

Para a implementação de um mecanismo de *snooping*, a abordagem paralela é a mais comumente adotada. *Snooping* em paralelo é trivial, uma vez que considera que os pacotes são enviados para todos os nodos do sistema ao mesmo tempo. Porém, ao realizar *broadcasts* de pacotes surge a preocupação com o consumo desnecessário de energia.

Várias requisições de coerência não encontram blocos correspondentes nas caches remotas e as verificações de *tags* falham, contribuindo para um desperdício de energia.

Uma maneira de otimizar o consumo energético do protocolo de coerência de memória é a eliminação de *snoops* inúteis. Diversas técnicas tem sido propostas com a intenção de detectar e eliminar esses *snoops* desnecessários.

Nos esquemas JETTY (MOSHOVOS et al., 2001), os autores introduziram estruturas semelhantes a caches entre as caches L2 locais e o barramento de memória, com o objetivo de identificar e, em seguida, filtrar todas as referências remotas a dados não presentes na cache local, conforme ilustra pictoriamente a figura 11. Quando a técnica não for capaz de afirmar que o dado não é compartilhado, ela permite a execução da operação de coerência.

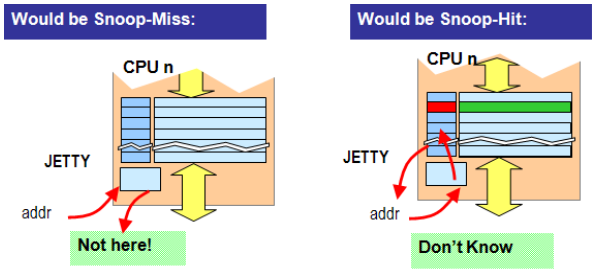

Figura 11: Sistema de filtragem JETTY, retirada de Moshovos et al. (2001)

Por ser uma técnica apenas em *hardware*, a cache responsável pela filtragem de *snoops* é atualizada sempre que o estado da cache local é alterado. Conforme demonstrado por Ekman, Dahlgren e Stenström (2002), ambas as variantes de JETTY, por terem sido originalmente projetadas para tratar *snoops* externos em servidores com multiprocessamento simétrico (SMP), não tem a mesma eficiência quando aplicadas em multiprocessadores simétricos em chip (CMP). Por um lado, *Include JETTY* não previne a maioria dos *snoops* desnecessários. Por outro lado, *Exclude JETTY* requer um acréscimo proibitivo de *hardware*.

Em Young et al. (2008) é feita uma avaliação de algumas técnicas de filtragem de *snoop*, dentre as quais estão a *RegionScout* (RS) (MOSHOVOS, 2005) e a técnica proposta por Cantin, Lipasti e Smith (2005). Ambas aproveitam as taxas de faltas de região e buscam informações de baixa granularidade. Uma região é definida como um bloco

de endereços de memória contíguos e possui tamanho pré-definido.

RS consiste de duas estruturas de dados, *Cashed-Region Hash* (CRH) e *Non-Shared-Region Table* (NSRT), em cada último nível de cache (do inglês *Last-Level Cache*), conforme ilustrado na figura 12. Essencialmente, a CRH indica quais regiões estão atualmente no LLC local e a NSRT indica quais regiões estão nos LLCs vizinhos e determina se uma falta no LLC acarreta em uma requisição de coerência. Quando um processador requisita uma operação de coerência, as CRHs dos outros processadores são verificadas para saber se eles possuem linhas da mesma região em cache. Se a resposta indicar que nenhuma linha da mesma região está nas caches de outros processadores, então é criada uma entrada para tal região na NSRT do processador solicitante. Desse modo, quaisquer outras linhas daquela região podem ser acessadas diretamente, dispensando o *broadcast*.

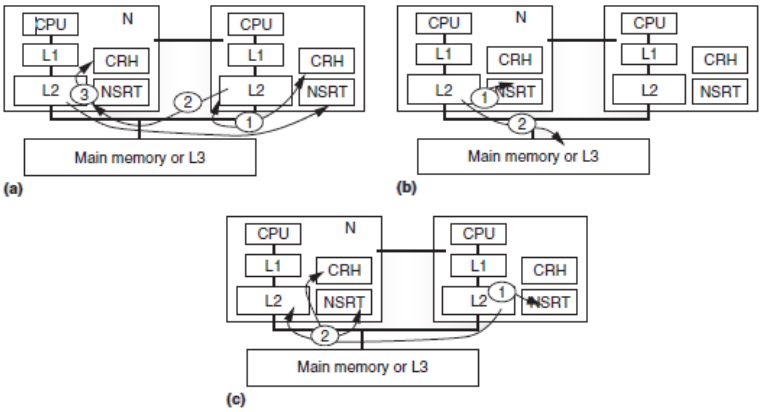

Figura 12: Arquitetura do RegionScout, retirada de Moshovos (2005)

A ocorrência de colisões de mapeamento na CRH e o consequente aumento da taxa de falsos positivos são os fatores limitantes do desempenho da RS. Isso porque regiões de tamanhos menores ou CRHs menores podem fazer com que mais regiões sejam mapeadas para uma mesma entrada da CRH e, consequentemente, sejam gerados falsos positivos. Além disso, a necessidade de um *broadcast* inicial para adquirir as informações das regiões impede que a RS atinja uma maior economia energética.

A técnica proposta por Cantin, Lipasti e Smith (2005) é similar a RegionScout e requer uma estrutura em *hardware* denominada *Region*

Coherence Array (RCA), bem como bits adicionais no sistema de interconexão. A RCA é uma pequena estrutura que mantém registros de regiões de memória compartilhada e não compartilhada, com associatividade por conjuntos composta por *tags* em cada LLC, conforme ilustrado na figura 13. Cada entrada da RCA contém os seguintes atributos:

- um endereço de *tag* para uma região
- um contador para indicar o número de linhas de cache em uma região atualmente no LLC local
- um bit de estado para indicar se a região está compartilhada ou não
- um bit de validade

Uma entrada RCA é alocada para uma região quando qualquer linha dessa região é trazida para a cache pela primeira vez. Em seguida, qualquer linha dessa região trazida para a cache irá incrementar o contador para aquela região. No caso de exclusão da linha, o contador é decrementado para aquela região. A RCA é acessada a cada falta no LLC para determinar se o *broadcast* de uma operação *snoop* é necessário e acessado quando requisições *snoop* são recebidas para atualizar o bit de estado de compartilhamento da RCA local. Se a entrada na RCA indicar que a região solicitada não está compartilhada, então nenhum *snoop* é enviado para aquela região.

Para garantir precisão, quando uma entrada da RCA é excluída devido tanto a um conflito de associatividade quanto a estouro da capacidade da RCA, todas as linhas válidas naquela região necessitam ser invalidadas, o que ocasiona novos acessos à cache e novas faltas no LLC.

Uma técnica de redução de energia de acessos à TLB e *snooping* através da utilização de caches virtuais é proposta por Ekman, Stenström e Dahlgren (2002). Para determinar se uma página está compartilhada por outros processadores, junto ao *snoop broadcast* é enviada uma informação de compartilhamento de página através de um barramento separado, denominado *shared vector bus*. Lendo tal informação, os processadores remotos podem eliminar pesquisas desnecessárias nas caches. Apesar de os autores afirmarem que o sistema proposto elimina o acréscimo de *snoops* relacionados a buscas de instruções, caches de instruções não são consideradas.

Patel e Ghose (2008) propõem um esquema de filtragem de *snoops* em processadores CMP com caches coerentes utilizando o protocolo

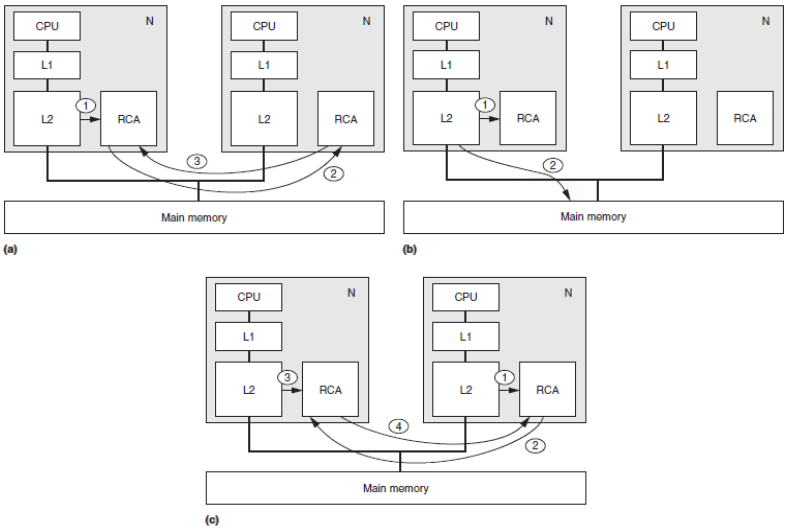

Figura 13: Arquitetura do RCA, retirada de Cantin, Lipasti e Smith (2005)

MESI, através da coleta pró-ativa de informações de compartilhamento de regiões. Isso é realizado por meio de acréscimo de vias no sistema de interconexão, pelas quais são coletados os estados de compartilhamento de todas as regiões. As informações são mantidas em estruturas associativas anexadas ao controlador de coerência de cada núcleo, conhecidas como *Snoop Filter Table* (SFT), conforme ilustrado na figura 14. As decisões de filtragem ocorrem em paralelo aos acessos a cache L2, não ocasionando *overhead* nos acessos à memória.

2.2.2 Abordagens híbridas cientes da aplicação

A grande parte dos trabalhos concentram-se em arquiteturas de processadores CMP de propósito geral e, na maioria dos casos, a abordagem introduzida considera a identificação e utilização de características de programas em tempo de execução. Para suportar tal abordagem, faz-se necessária a inclusão de diversas estruturas em hardware como caches e tabelas, o que acarreta em um acréscimo de área e consumo energético, tornando frequentemente a solução inviável para a utilização em sistemas em chip.

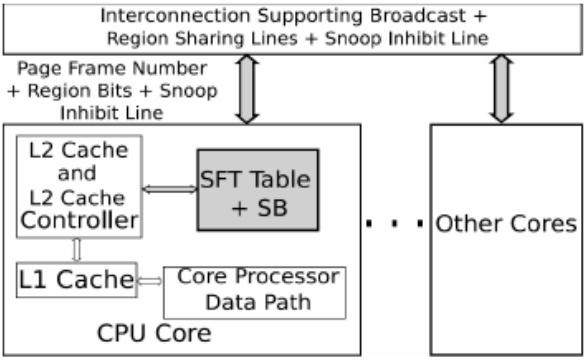

Figura 14: Arquitetura da técnica de Patel e Ghose (2008)

Em Zhou et al. (2008) é proposta uma técnica de filtragem de *snoops* com base em informações de aplicações coletadas previamente. Isso é feito através da identificação das regiões de memória compartilhadas por cada tarefa e, então, fornecendo essas informações ao sistema operacional e ao controlador *snoop* da cache para a utilização em tempo de execução. A identificação estática de dados compartilhados entre *threads* é realizada por um algoritmo heurístico de análise de ponteiros, cujo objetivo é determinar todos os valores possíveis de um ponteiro para obter informações pertencentes aos objetos de memória que serão potencialmente acessados por aquele ponteiro. Para a máxima eficiência da técnica proposta, é necessário que o desenvolvedor do *software* forneça ao compilador as informações dos objetos compartilhados entre as *threads*. Além disso, a técnica não considera quaisquer mudanças dinâmicas no padrão de acessos aos dados compartilhados, ou seja, eventuais acessos a dados não-compartilhados podem gerar *snoops* desnecessários.

Uma técnica de particionamento da cache de dados é apresentada em Yu, Zhou e Petrov (2009). Através da separação dos dados privados dos dados compartilhados, é possível reduzir o consumo energético devido ao fato de que apenas uma partição da cache é acionada a cada acesso, utilizando menos vias de associatividade. É necessária a inclusão de um dispositivo denominado *Way Allocation Bitmap Set* que determina quais vias de associatividade serão alocadas para um determinado acesso à memória.

Partindo da premissa de que os dados contidos na pilha de uma *thread* sejam privados a essa *thread*, os *snoops* ocasionados por consequência de acessos a tais dados nunca resultarão em um *hit*, pois

os mesmos certamente não estarão presentes nas caches privativas de processadores remotos. Segundo Ballapuram, Sharif e Lee (2008), cerca de 30% dos acessos à memória que atingem o último nível de cache são referentes à pilha, conforme ilustra a figura 15.

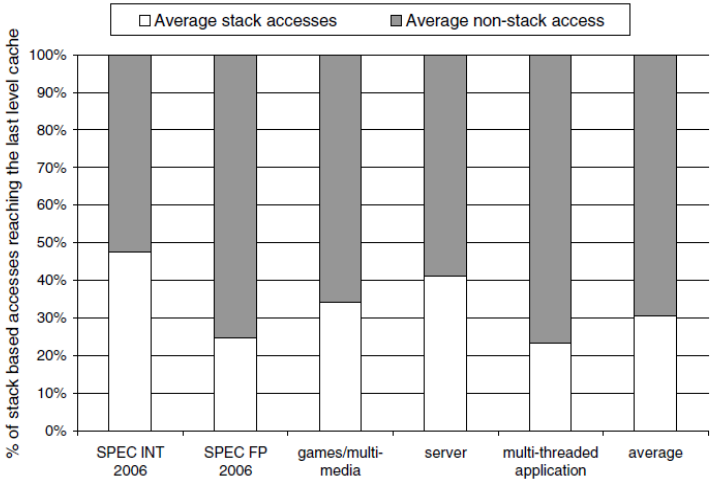

Figura 15: Acessos de pilha ao último nível de cache (BALLAPURAM; SHARIF; LEE, 2008)

Seguindo essa linha, Ballapuram, Sharif e Lee (2008) propuseram dois mecanismos de filtragem de *snoops*. Diferentemente da técnica proposta por Zhou et al. (2008), *Selective Snoop Probes* (SSP) e *Essential Snoop Probes* (ESP) identificam as regiões de memória que certamente não possuem dados compartilhados entre processadores. Para tanto, os acessos à memória que atingem o LLC são divididos em dois grupos, quanto ao destino: os que são referentes à pilha e os que não são. Em relação aos acessos que não vão para a pilha, esses podem ser subdivididos em dois grupos: os que recebem uma resposta de acerto ou modificado (dado encontrado em algum processador remoto), e os que recebem uma resposta limpa (dado inexistente nos processadores remotos). As técnicas são complementares, sendo a SSP aplicada em tempo de execução e a ESP com auxílio de compiladores. A necessidade de modificação de alguns dispositivos-padrão e a inclusão de estruturas de dados para realizar a filtragem dinâmica de *snoops* podem comprometer a eficiência de área e energia do sistema, inviabilizando a implementação

das técnicas em ambientes computacionais embarcados.

Também considerando dados privados de *threads* como foco para otimização, Meng e Skadron (2009) propõem uma organização de cache não-inclusiva e semi-coerente que remove o requisito de inclusão para dados privados, esses compostos tanto de pilha e de *heap*. Isso permite que dados privados existam apenas nas caches L1 e dados privados despejados da LLC não precisam invalidar suas cópias armazenadas nas caches L1, reduzindo o tráfego de requisições de coerência pelo sistema. A técnica requer a adição de um bit de controle ao registrador de estado de cada linha de cache, responsável por indicar se a linha de cache armazena dados compartilhados ou privados, de modo a habilitar ou desabilitar a emissão de requisições de coerência, respectivamente. Tal modificação do *hardware* pode inviabilizar sua aplicação em MPSoCs. Além disso, os autores não realizaram uma análise de consumo de energia em seus resultados experimentais, tendo considerado apenas o impacto da técnica no desempenho do sistema.

3 UMA TÉCNICA PARA O GERENCIAMENTO *OVERLAY* DE SPMS PRIVATIVAS EM MPSOCS

Conforme mencionado anteriormente, o alto desempenho requerido pelos dispositivos eletrônicos pessoais contemporâneos, tais como *smartphones* e *tablets*, tem sido atingido graças à adoção de multiprocessamento em *chip*. Se um sistema embarcado exige a adoção de uma plataforma com multiprocessamento simétrico, então as aplicações para este sistema deverão ser implementadas usando alguma técnica de paralelização, a fim de tirar o devido proveito da capacidade de processamento disponível. Neste contexto, o uso de *threads* no desenvolvimento de aplicações paralelas surge como uma solução pragmática e eficiente, basicamente porque a maioria das plataformas para sistemas embarcados disponíveis no mercado (e.g., famílias Cortex-R (ARM Corporation, 2011b) e Cortex-A (ARM Corporation, 2011a), da ARM) são desenvolvidas visando o modelo de memória compartilhada.

Com o intuito de maximizar a banda passante e ao mesmo tempo, minimizar a latência no acesso ao espaço de endereçamento único, os MPSoCs usados nas plataformas voltadas a aplicações de multimídia e de comunicação possuem um subsistema de memória hierárquico, com duas caches L1 (uma para dados e uma para instruções) privativas a cada processador e, no caso das plataformas mais recentes (e.g., ARM Cortex-A), memória cache L2 unificada em *chip*. Como resultado, baixa latência nos acessos a dados e instruções é alcançada, mas mediante penalidade em termos de consumo de energia no subsistema de memória. Além disso, a existência de múltiplas caches L1 exige a implementação de algum protocolo de coerência, conforme já mencionado no capítulo 1. De fato, o protocolo *snooping* é comumente encontrado nas plataformas comerciais, como ARM Cortex-R e ARM Cortex-A. Ainda que a implementação de protocolo de coerência de caches utilize técnicas de filtragem e blocos de hardware dedicados, as penalidades em termos de área, de tempo de acesso e de consumo de energia não são desprezíveis, conforme relatado no capítulo 2.

O fato de os dados da pilha de execução de uma *thread* serem essencialmente privados e apresentarem boa localidade permite especular que a alocação da pilha em uma SPM de uso privativo ao processador na qual a *thread* executa resultará em economia de energia no subsistema de memória, uma vez que a energia por acesso à SPM é inferior à energia por acesso a uma cache com mesma capacidade de armazenamento. Naturalmente, a economia a ser alcançada com a relocação da pilha de

cada *thread* em um MPSoC dependerá da organização de seu subsistema de memória (número de níveis de caches, capacidades da SPM e das caches, protocolo de coerência) e de características das aplicações (basicamente, da frequência na qual os dados de pilha são acessados). É interessante perceber que a relocação da pilha de *thread* para SPM traz como benefício extra a redução do número de requisições de coerência nas caches L1, desde que a área de endereçamento da SPM e o espaço de endereços acessáveis via caches sejam disjuntos. Desta forma, ocorre uma filtragem de *snoops* por construção, o que também colabora para a redução de energia no subsistema de memória. Por outro lado, em função do comportamento dinâmico da pilha, seu gerenciamento precisa ser realizado durante a execução da aplicação por um gerenciador em software, ou totalmente em hardware ou parte em software e parte em hardware (i.e., híbrido). Assim, parte da penalidade imposta pelo gerenciamento da pilha em SPM dependerá de tal implementação.

Neste trabalho propõe-se uma técnica para o gerenciamento *overlay* de SPMs privativas em MPSoCs, visando a redução da energia consumida no subsistema de memória. O objetivo da técnica é alocar a pilha de cada *thread* na SPM privativa ao processador onde a *thread* executa. Para tanto, a técnica pressupõe que:

1. Os dados de pilha de uma dada *thread* sejam essencialmente privados[1].

2. A plataforma a ser utilizada proporcione multiprocessamento simétrico, com modelo de memória compartilhada.

3. Cada processador possua duas caches L1 privativas (uma para dados e uma para instruções), através das quais o espaço de endereçamento comum é acessado.

4. Cada processador possua uma SPM de uso privativo, cujo espaço de endereçamento é disjunto do espaço de endereçamento compartilhado, seguindo o modelo PGAS.

Na técnica aqui proposta, os dados de pilha de uma *thread* são gerenciados dinamicamente por um gerenciador de pilha em software, o qual é também armazenado na SPM privativa do processador a fim de reduzir a penalidade de tempo e de energia nos acessos a suas instruções

[1] Pode ocorrer uma situação em que uma *thread* referencia um elemento de pilha de outra *thread* através de um ponteiro. Apesar de que a técnica proposta tratará esse cenário quando necessário, tal caso atípico raramente ocorre, uma vez que isso seria resultado de má prática de programação.

e dados. Tipicamente, as primeiras instruções de um procedimento (conhecidas como prólogo) preparam a pilha para acomodar variáveis locais, enquanto as últimas instruções (conhecidas como epílogo) restauram a pilha para o seu estado anterior. Na técnica proposta, a interação entre o código da aplicação e o gerenciador de pilha ocorre no prólogo e no epílogo de cada procedimento executado por uma *thread*. Desta forma, por estarem presentes no espaço de endereçamento compartilhado, as instruções adicionadas no prólogo e no epílogo dos procedimentos impõem alguma penalidade à técnica.

Apesar de depender da disponibilidade do código-fonte da aplicação, diferentemente de outras técnicas semelhantes (e.g.,(ZHOU et al., 2008)), a técnica aqui apresentada dispensa a intervenção do desenvolvedor, pois as alterações necessárias no código da aplicação são realizadas por um compilador devidamente adaptado, conforme será explicado na seção 3.2.

As seções seguintes abordam os aspectos teóricos mais relevantes da técnica de gerenciamento *overlay* de múltiplas SPMs proposta neste trabalho. Os detalhes de implementação são explicados no capítulo 4.

3.1 A PILHA DE EXECUÇÃO

A noção de pilha de execução, ou simplesmente pilha, teve origem com linguagens de programação como Lisp e Algol 60, as quais introduziram o uso de variáveis locais (FISCHER; CYTRON; LEBLANC, 2010). Uma variável é dita local quando é acessível apenas durante a execução de um procedimento. A execução de um procedimento requer que seja alocado espaço em memória para armazenar os parâmetros do procedimento e suas variáveis. Além disso, também é necessário espaço em memória para informações de controle, como o endereço de retorno de procedimento. Assim, os dados e informações de controle do procedimento são armazenados na pilha, a qual ocupa uma parte do espaço de endereçamento do sistema. A pilha, por sua vez, cresce em unidades denominadas quadros de pilha (*stack frames*), ou simplemente quadros, cujo tamanho depende da arquitetura-alvo e é calculado pelo compilador. Quando o procedimento retorna, seu quadro é retirado e o espaço utilizado por suas variáveis locais é liberado. Desta forma, somente os procedimentos em execução ocupam memória.

Para ilustrar a alocação de dados em pilha, considere o procedimento p mostrado na figura 16, o qual foi escrito em linguagem C++.

```
p(int a) {
     double b;
     double c[10];
     b = c[a] * 2.51;
}
```

Figura 16: Exemplo de procedimento em linguagem C++

Considere que a pilha seja implementada em uma memória organizada em palavras de 4 bytes, a qual é endereçada pelo processador usando alinhamento por palavras duplas (*doubleword alignment*). Desta forma, todo e qualquer quadro deverá ter tamanho múltiplo de 8 bytes. Suponha que as informações de controle de p, o parâmetro a, a variável local b e o arranjo local c requeiram, respectivamente, 8 bytes, 4 bytes, 8 bytes, e 80 bytes. Nestas condições, o quadro de pilha para o procedimento p possui tamanho de 104 bytes, sendo ilustrado pela figura 17. Nesta figura, o espaço livre rotulado por *padding* serve exclusivamente para garantir o alinhamento por palavras, fazendo com que o tamanho do quadro seja múltiplo de 8.

3.2 ADEQUAÇÃO DO COMPILADOR À TÉCNICA

As etapas do processo de compilação em um compilador convencional podem ser agrupadas em *front-end*, *middle-end* e *back-end* (FISCHER; CYTRON; LEBLANC, 2010).

O *front-end* corresponde às etapas da compilação que dependem basicamente da linguagem de entrada, sendo portanto independentes da arquitetura-alvo (ALFRED; SETHI; ULLMAN, 1986). Nesse conjunto de etapas normalmente estão incluídas as análises léxica e sintática, a criação da tabela de símbolos, a análise semântica, e a geração de código intermediário. Algumas otimizações de código podem ser realizadas no *front-end*. O *front-end* também inclui o tratamento de erros associado a cada etapa.

O *back-end* inclui as etapas do compilador que dependem da arquitetura-alvo. No *back-end* podem ser encontrados aspectos da etapa de otimização de código e também é onde se realiza a geração de código, juntamente às operações necessárias sobre a tabela de símbolos e de

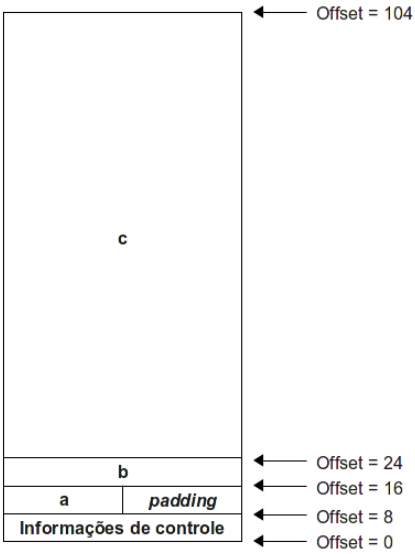

Figura 17: Quadro de pilha do procedimento *p*

tratamento de erros.

No *middle-end* são realizadas as transformações típicas que levam à otimização do código, tais como remoção de código inútil ou inalcançável, descoberta e propagação de valores constantes, relocação de uma computação para um local executado com menor frequência (para fora de um laço, por exemplo), ou especialização de computação baseada em um contexto. Como as etapas do *middle-end* são realizadas após o *front-end* e antes do *back-end*, elas são independentes tanto da linguagem de entrada quanto da arquiteturas-alvo, o que simplifica muito a construção de compiladores para várias linguagens de entrada e várias arquiteturas-alvo.

Para que o compilador gere código adequado para a utilização da técnica proposta neste trabalho, é necessária a adição de dois passos ao processo de compilação da aplicação, sendo um no *middle-end*, durante a análise do grafo de chamadas, e outro no *back-end*, precedendo a geração do código em linguagem de montagem.

3.2.1 Identificação dos procedimentos cuja pilha será alocada em SPM

Quando o compilador finaliza a análise de fluxo da aplicação, ele gera dois tipos de **grafos de fluxo**: o **grafo de chamadas de procedimentos** e um **grafo de fluxo de controle** para cada procedimento (FISCHER; CYTRON; LEBLANC, 2010).

Definição 1. *Um grafo de fluxo* $G = (N, E, raiz)$ *é um grafo direcionado:* N *é um conjunto (de vértices) e* E *é uma relação binária em* N*. O vértice raiz é o vértice de entrada distinto do grafo de fluxo:* $\forall X \in N, (X, raiz) \notin E$*.*

Em um grafo de fluxo, os vértices representam diversos pontos de um programa e as arestas indicam transições possíveis entre esses pontos. A técnica aqui apresentada utiliza um tipo de grafo de fluxo cujos caminhos representam uma sequência de chamadas de procedimento. Neste grafo de fluxo, denominado grafo de chamadas de procedimento, cada vértice em N corresponde a um procedimento de um programa e cada aresta em E representa uma possível chamada de procedimento.

O primeiro passo da técnica tem a função de identificar todos os procedimentos que são invocados em uma dada *thread*, o que é realizado com o auxílio do grafo de chamadas de procedimentos da aplicação. Uma vez que o grafo de chamadas de procedimento esteja disponível, o primeiro passo da técnica pode ser executado. Inicialmente, é necessário identificar o procedimento que representa o ponto de entrada (*entry-point*) de uma *thread*. O grafo de chamadas de procedimentos é então percorrido em busca do vértice referente a esse procedimento e, quando encontrado, o mesmo é anotado pelo compilador. Em seguida, são identificados todos os procedimentos que são invocados a partir do ponto de entrada da *thread*.

O tamanho do quadro de pilha de cada um dos procedimentos identificados é então computado, sendo mantido em uma estrutura de dados auxiliar. Tal estrutura será integrada aos binários da aplicação no processo de linkedição, de modo a ser utilizada pelo gerenciador de pilha em tempo de execução tanto para alocar os quadros de pilha em SPM quanto para desalocá-los.

3.2.2 Geração de código de suporte

O segundo passo fornece suporte à geração de código para invocar o gerenciador de pilha. Para cada procedimento identificado durante o primeiro passo, o compilador gerará um prólogo e um epílogo adequados para invocar apropriadamente as ações do gerenciador.

Além das instruções próprias para invocação do gerenciador de pilha, o código gerado tem o objetivo de garantir que seja mantido o fluxo de execução da aplicação, salvando o contexto de registradores que venham a ser utilizados no processo.

3.2.3 Geração de saída

Uma simples customização do *script* de linkedição permite que o ligador acomode os binários do gerenciador de pilha em um espaço de endereçamento reservado em cada SPM privativa, dando origem ao arquivo executável otimizado.

3.3 O GERENCIADOR DE PILHA

Conforme já mencionado no início deste capítulo, a técnica proposta assume que o gerenciador de pilha é totalmente implementado em *software* e fica armazenado na mesma SPM por ele gerenciada. Além disso, a técnica assume que os binários do gerenciador estarão replicados em cada SPM. Semelhantemente ao proposto no trabalho de Kannan et al. (2009), o gerenciador a ser usado na técnica aqui proposta deve ser capaz de:

1. Verificar se existe espaço suficiente na SPM para o quadro de pilha do procedimento antes que ele seja chamado;
2. Rastrear o quadro mais antigo armazenado na SPM;
3. Transferir quadros da SPM para a memória principal em caso de transbordo, e trazê-los de volta quando necessário.

Detalhes sobre a implementação do protótipo de gerenciador são fornecidos no próximo capítulo.

4 INFRAESTRUTURA E IMPLEMENTAÇÃO

Neste capítulo são apresentados detalhes da implementação da técnica de gerenciamento *overlay* de SPMs introduzida no capítulo 3, os quais abrangem a adaptação do compilador e o gerenciador de dados de pilha de *thread* em SPM. O capítulo também descreve a infraestrutura utilizada para modelar as arquiteturas de MPSoCs e para simular a execução de aplicações nas mesmas. A seguir, o método de geração aleatória de aplicações é apresentado e as características mais relevantes do conjunto de aplicações gerado são detalhadas. Por último, é mostrada a formulação utilizada para se obter os valores de energia a partir dos números de acessos a cada componente do subsistema de memória e dos respectivos valores de energia por acesso.

4.1 MODIFICAÇÃO DO COMPILADOR C DO GCC

Conforme descrito no capítulo 3, a técnica aqui proposta requer que um compilador seja adaptado para a geração de código ciente do gerenciador de dados de pilha em SPMs. Para tanto, decidiu-se utilizar o compilador C do GCC (GNU, 2012).

Um novo passo de compilação inserido no *middle-end* do compilador é responsável pela identificação dos procedimentos que serão otimizados. No *middle-end* do GCC a linguagem de entrada já se encontra traduzida para uma representação intermediária denominada GIMPLE, fazendo com que a implementação do passo seja independente da arquitetura do processador-alvo. O novo passo, a ser executado durante o processo de otimização interprocedural, faz uso do grafo de chamadas de procedimentos da aplicação, o qual já está disponível nessa etapa do processo de compilação. Primeiramente, o compilador faz uma análise do código para identificar o primeiro procedimento a ser executado em uma dada *thread*, também conhecido como ponto de entrada (*entry-point*) de uma *thread*. A implementação aqui descrita se baseia no padrão POSIX Threads para a manipulação de *threads*. Desta forma, o procedimento desejado será aquele cujo endereço é passado como parâmetro de uma chamada ao procedimento *pthread_create*. Assim que o ponto de entrada é identificado, o compilador busca pelo vértice a ele associado no grafo de chamada de procedimentos da aplicação. Para cada um dos seus vértices descendentes, o compilador anota todos os procedimentos a eles associados.

O segundo passo que compõe a adaptação do compilador à técnica é realizado no *back-end* do GCC, durante a geração de código de máquina. Para cada um dos procedimentos anotados anteriormente, o compilador cria um par de prólogo e epílogo com instruções especiais destinadas à comunicação com o gerenciador de dados de pilha. Tais instruções são representadas em RTL (*Register-Transfer Language*), uma linguagem dependente da arquitetura do processador-alvo. No presente trabalho, esse passo foi implementado para a arquitetura SPARC V9. Caso surja a necessidade de se portar a implementação da técnica para uma outra arquitetura, apenas esse passo precisará ser adaptado para o novo alvo.

A figura 18 mostra o código do início de um procedimento ciente de gerenciador de pilha em SPM, que no caso se trata de um ponto de entrada de *thread*. O prólogo do procedimento corresponde às linhas 1 a 17. As linhas de código 1 a 4 e 13 a 16 são responsáveis pelo salvamento e restauração, respectivamente, do conteúdo dos registradores aproveitados para realizar a comunicação com o gerenciador de pilha. As chamadas aos procedimentos de inicialização e *check-in* do gerenciador ocorrem nas linhas 8 e 11, respectivamente. Os parâmetros de cada uma das chamadas são setados nas linhas 5 a 7, e também na linha 10. A instrução *save* na linha 17 é responsável por realizar as operações referentes à troca de contexto do processador a cada novo procedimento chamado.

A figura 19 mostra as últimas instruções de um procedimento ciente de gerenciador de pilha em SPM. O epílogo do procedimento corresponde às linhas 3 a 7. O parâmetro do procedimento de *check-out* é setado na linha 3, e na linha 4 ocorre a chamada ao procedimento. Semelhante à instrução *save*, a instrução *rett* na linha 6 é responsável pelas operações referentes à restauração de contexto do processador ao final de cada procedimento.

No fluxograma mostrado na figura 20 é possível identificar em quais etapas do processo de compilação os novos passos foram inseridos.

A geração de código ciente do gerenciador de dados de pilha em SPMs é ativável através da inclusão da opção *-fspm* aos parâmetros de execução do compilador.

Conforme dito na seção 3.3, os binários do gerenciador de pilha são alocados nas SPMs de cada um dos processadores do sistema em tempo de carga da aplicação. O compilador gera os dados e procedimentos do gerenciador em seções de SPM exclusivas, *.spm.data* e *.spm.func*, respectivamente. O ligador, instrumentado com um *script* modificado, mescla ambas as seções de SPM, e as realoca para o espaço de endereçamento de SPM, originando o arquivo executável otimizado.

```
     <thread_0>:
1         sethi     %hi(0x80002000), %g1
2         stx       %o0, [ %g1 + -8 ]
3         stx       %o1, [ %g1 + -16 ]
4         stx       %o7, [ %g1 + -24 ]
5         sethi     %hi(0x80000400), %o0
6         sethi     %hi(0x80001c00), %g1
7         or        %g1, 0x33f, %o1
8         call      80000000 <spmm_init>
9         nop
10        mov       4, %o0
11        call      80000044 <spmm_check_in>
12        nop
13        sethi     %hi(0x80002000), %g1
14        ldx       [ %g1 + -8 ], %o0
15        ldx       [ %g1 + -16 ], %o1
16        ldx       [ %g1 + -24 ], %o7
17        save      %sp, -256, %sp
18        stx       %i0, [ %fp + 0x87f ]
19        stx       %i1, [ %fp + 0x887 ]
          ...
```

Figura 18: Prólogo de um procedimento ciente de SPM

```
          ...
1         cmp  %g1, 0
2         nop
3         mov  4, %o0
4         call  80000158 <spmm_check_out>
5         nop
6         rett  %i7 + 8
7         nop
```

Figura 19: Epílogo de um procedimento ciente de SPM

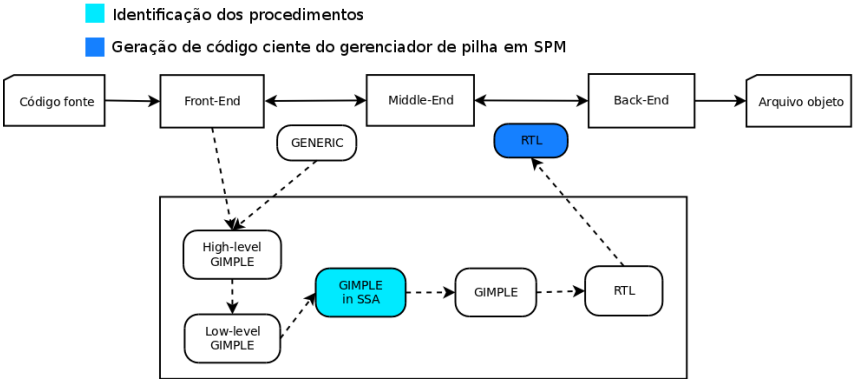

Figura 20: Fluxo de compilação do GCC

4.2 IMPLEMENTAÇÃO DE GERENCIADOR DE PILHA EM SPM

O gerenciador de pilha em SPM descrito na seção 3.3 é implementado como uma biblioteca ligada estaticamente à aplicação. Os binários (código e dados) do gerenciador são completamente armazenados em uma faixa de endereços específica, dentro da faixa reservada à SPM. A interface da biblioteca possui três procedimentos para se comunicar com a aplicação:

- **init**: Inicializa as estruturas de dados do gerenciador. É chamada no prólogo do primeiro procedimento a ser executado dentro de uma *thread* (*entry-point*).

- **check-in**: Notifica o gerenciador de pilha que um procedimento foi invocado. Após verificar se existe espaço disponível para a chamada de procedimento, o gerenciador pode enviar quadros antigos para a memória principal com o intuito de acomodar o quadro novo. O procedimento de *check-in* é chamado no prólogo de cada procedimento, antes de qualquer instrução relacionada à pilha. A figura 21 ilustra os passos que compõem a operação de *check-in*.

- **check-out**: É chamado no epílogo dos procedimentos, exatamente antes da instrução de retorno. O gerenciador verifica se ele teve de enviar algum quadro para a memória principal durante o *check-in* desse mesmo procedimento. Caso tenha sido necessário, o

gerenciador traz de volta para a SPM tais quadros, permitindo que a aplicação continue normalmente. A figura 22 ilustra os passos que compõem a operação de *check-out*.

Visando reduzir a penalidade imposta pelo gerenciador de pilha, cada um de seus procedimentos é implementado sem utilizar quaisquer chamadas a bibliotecas padrões.

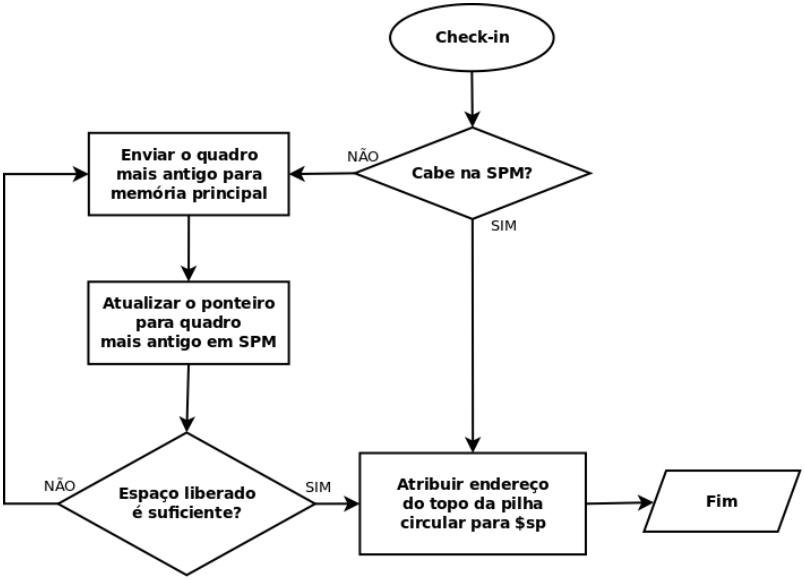

Figura 21: Fluxograma do procedimento de *check-in*

O gerenciamento de dados de pilha em SPM segue uma pilha circular. Na presente implementação, o gerenciador utiliza granularidade de quadro de procedimento tanto para alocar como para remover dados da SPM.

4.3 INFRAESTRUTURA DE SIMULAÇÃO

A técnica proposta foi validada e avaliada com o auxílio do *framework* de simulação dirigida a eventos denominado de GEM5 (anteriormente chamado de M5) (BINKERT et al., 2006, 2011), que serve de representação executável de uma arquitetura CMP. A disponibilidade de código-fonte tornou possível a adaptação do *framework* para a inclusão

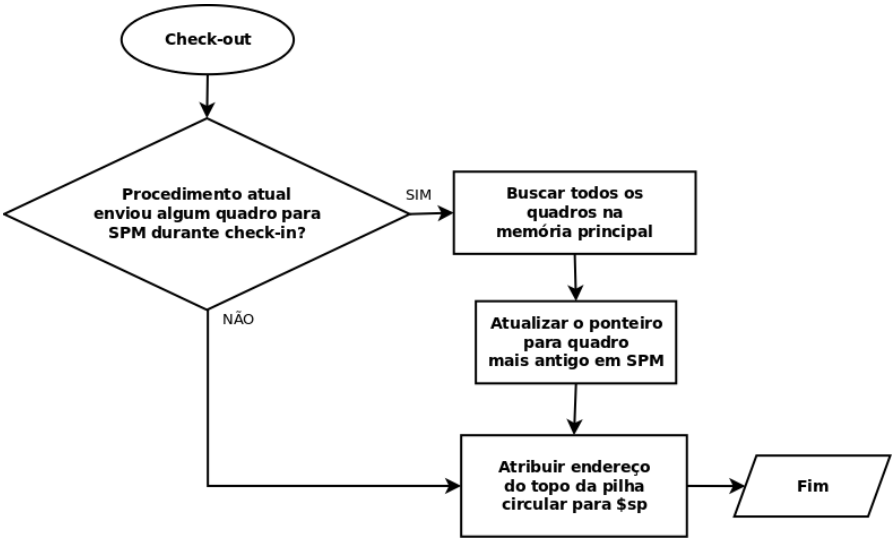

Figura 22: Fluxograma do procedimento de *check-out*

de SPMs, o que fez com que o GEM5 fosse considerada como uma opção bastante adequada para o propósito dessa dissertação. Além disso, a possibilidade de configurar diversos parâmetros da plataforma-alvo, como número de processadores e capacidades e associatividades das caches, permitiu que fossem modeladas diversas plataformas sem necessidade de recompilação. O *framework* GEM5 garante a coerência entre as caches privativas e a memória compartilhada através de uma implementação simples do protocolo *snooping*.

As simulações foram realizadas em um modelo funcional temporizado (*cycle approximate*) da plataforma, no modo de emulação de chamadas de sistema (*syscall emulation*), onde o binário da aplicação é executado diretamente sobre a plataforma, dispensando a simulação de um sistema operacional. Uma vez concluída uma simulação, o *framework* gera um arquivo de texto com uma estatística completa, composta por dados como tempo total de simulação, número de ciclos despendidos por cada processador, número de acessos a cada bloco de memória do sistema, quantidade de faltas das caches, entre outras.

Para a avaliação quantitativa da técnica proposta para alocação de dados de pilha em SPMs foram modeladas 2 arquiteturas de sistema multiprocessado que refletem os MPSoCs estado-da-arte, ambas compartilhando as características mostradas na tabela 3.

Tabela 3: Características comuns às arquiteturas modeladas com o GEM5

Nó tecnológico	32nm
Núcleos	4 SPARC V9, 64 bits, com execução em-ordem (o sistema pode executar 4 threads em paralelo)
Caches L1	1 I-Cache e 1 D-Cache por núcleo, ambas com as mesmas características: • Capacidade e grau de associatividade, conforme Tabela 4 • Capacidade de bloco: 32 bytes, *write-back* • Latência de acerto: 1 ciclo
Cache L2	Unificada, compartilhada entre todos os núcleos Capacidade: 256 KB Associatividade: 16 vias Capacidade de bloco: 32 bytes, *write-back* Latência de acerto: 10 ciclos
Memória Principal	Capacidade: 128 MB Latência de acesso: 30 ciclos

A uma das arquiteturas modeladas foi adicionada uma SPM de 8KB por processador (com latência de acesso de 1 ciclo), criando assim um modelo de plataforma multiprocessada PGAS com SPM, servindo para avaliar a técnica proposta neste trabalho. A outra arquitetura, que não inclui SPMs, é utilizada como um modelo de referência. A partir dessas 2 arquiteturas, foram criadas 10 diferentes plataformas através da variação da capacidade e da associatividade das caches L1. A tabela 4 resume as características que diferenciam as 20 plataformas utilizadas. Nesta tabela, "REF" identifica as plataformas baseadas na arquitetura de referência e "cSPM" identifica as que são baseadas na arquitetura com SPM.

Tabela 4: Configuração de sistema para as plataformas baseadas nas arquiteturas REF e cSPM

Arquitetura	I-Cache L1, D-Cache L1		SPM	Total
	Capacidades (KB)	Associatividades		
REF	4, 8, 16, 32 ou 64	2, 4	-	10
cSPM	4, 8, 16, 32 ou 64	2, 4	8 KB	10
Total				20

As capacidades e associatividades empregadas para as caches L1 são baseadas nas configurações utilizadas por implementações de MPSoCs comerciais, como o ARM Cortex-R7.

4.4 GERAÇÃO AUTOMÁTICA DE APLICAÇÕES

Com o objetivo de explorar o espaço de otimização disponível para a técnica proposta, surgiu a necessidade de se gerar uma quantidade de dados significativos que permitissem realizar uma análise estatística. A indisponibilidade de um conjunto de *benchmarks* de teste suficientemente grande e cujas características de proporção de acessos a dados de pilha sejam conhecidos *a priori* motivou a adoção de uma abordagem de simulação com aplicações sintéticas, estas criadas automaticamente por meio de um gerador dedicado.

A geração automática de aplicações permite produzir uma grande quantidade de dados com características conhecidas, o que facilita a

análise dos resultados para identificar tendências em um dado conjunto de simulações com diferentes parâmetros. O algoritmo 1 mostra o pseudocódigo do gerador de aplicações, este desenvolvido com base no trabalho de Rambo, Henschel e Santos (2011). O gerador requer 5 parâmetros de entrada para sua execução:

- Porcentagem de operações de chamadas de procedimentos (t_C);
- Porcentagem de operações de escrita (t_E);
- Porcentagem de operações de leitura (t_L);
- Porcentagem de acessos referentes a dados de pilha (t_{AL});
- Porcentagem de acessos referentes a dados globais (t_{AG}).

A soma dos parâmetros t_C, t_E e t_L deve totalizar exatamente 1 (100%). O mesmo requisito se aplica para os parâmetros t_{AL} e t_{AG}.

Em uma aplicação gerada pelo gerador automático, além do procedimento principal (*main*), são definidos 4 procedimentos. O código de cada um deles foi gerado aleatoriamente obedecendo a seguinte distribuição de tipos de operação ($tipo_{OP}$), de modo que a aplicação sintética gerada representasse um padrão de acessos arbitrário de uma aplicação real: 10% de chamadas de procedimentos, 60% de leituras e 30% de escritas. A quantidade de operações, variáveis locais e parâmetros de cada procedimento, representadas por n_{OP}, n_{VARS} e n_{PARAMS}, respectivamente, também são definidas aleatoriamente.

Foi imposta uma limitação na profundidade da árvore de chamada de procedimentos de cada *thread* para que a altura das mesmas não fosse superior a 7. A não imposição de um limite faria com que o tempo de simulação das aplicações fosse demasiadamente longo, considerando que o mesmo cresce exponencialmente à medida que a altura e o grau da árvore de chamadas da aplicação aumenta.

Os acessos de leitura e escrita podem ser divididos em dois grupos quanto ao tipo de dado acessado ($tipo_A$): os que acessam dados de pilha e os que acessam dados globais. A proporção entre esses dois tipos de acesso, tanto para leitura quanto para escrita, obedece a definição 2.

Definição 2. *Sejam G e S, respectivamente, o número de acessos a dados globais e o números de acessos a dados de pilha em uma aplicação. A proporção de acessos a dados de pilha em uma aplicação é definida por $P_S = S/(S+G)$ e, por conseguinte, a proporção de acessos a dados globais é $P_G = G/(S+G) = 1 - P_S$.*

Algoritmo 1 Gerador aleatório de aplicações

Entrada(s): t_C, t_E, t_L, t_{AL}, t_{AG}

```
 1: GERAR_VARS_GLOBAL()
 2: para todo procedimento faça
 3:    n_PARAMS ← random(1,8)
 4:    n_VARS ← random(1,15)
 5:    n_OP ← random(10,100)
 6:    GERAR_VARS_LOCAL(n_VARS)
 7:    para i = 0 até n_OP faça
 8:       tipo_OP ← DEFINE_TIPO_OPERACAO(t_C,t_E,t_L)
 9:       se tipo_OP = chamadaProcedimento então
10:          GERAR_CHAMADA_PROCEDIMENTO()
11:       senão, se tipo_OP = leitura então
12:          tipo_A ← DEFINE_TIPO_ACESSO(t_AL,t_AG)
13:          se tipo_A = local então
14:             GERAR_LEITURA_LOCAL()
15:          senão, se tipo_A = global então
16:             GERAR_LEITURA_GLOBAL()
17:          fim se
18:       senão, se tipo_OP = escrita então
19:          tipo_A ← DEFINE_TIPO_ACESSO(t_AL,t_AG)
20:          se tipo_A = local então
21:             GERAR_ESCRITA_LOCAL()
22:          senão, se tipo_A = global então
23:             GERAR_ESCRITA_GLOBAL()
24:          fim se
25:       fim se
26:    fim para
27: fim para
28: GERAR_PROCEDIMENTO_MAIN()
```

Para a obtenção de resultados experimentais, foram gerados 4 grupos de aplicações, cada um destes caracterizado por um P_S distinto, pertencente ao conjunto $0,4$; $0,6$; $0,8$; $1,0$. Para cada grupo foram geradas 100 aplicações, totalizando 400 aplicações, conforme resume a tabela 5.

Tabela 5: Proporção de acesso a dados para as aplicações geradas aleatoriamente

P_S	P_G	Número de aplicações geradas
$1,0$	$0,0$	100
$0,8$	$0,2$	100
$0,6$	$0,4$	100
$0,4$	$0,6$	100
Total		400

Para a compilação das aplicações foi utilizado o parâmetro *-O0* para desativar todas as otimizações de código do compilador C do GCC, pois isso acarretaria na descaracterização das aplicações, umas vez que o código gerado aleatoriamente não apresenta computação útil. Todas as 400 aplicações foram então simuladas em cada uma das 20 plataformas descritas na tabela 4, totalizando 8000 simulações, as quais tiveram suas estatísticas coletadas para análise.

4.5 ESTIMATIVA DE ENERGIA

Uma vez feita a simulação com o GEM5 e coletados os dados de número de acessos para cada tipo de memória, foram utilizadas as seguintes fórmulas para o cálculo da energia no subsistema de memória.

4.5.1 Especificação das componentes de energia

- ε_{L1}: Energia por acesso a uma cache L1 (de instruções ou de dados);
- ε_{SPM}: Energia por acesso a uma memória de rascunho;

- ε_{L2}: Energia por acesso à cache L2;
- ε_{MM}: Energia por acesso à memória principal.

Conjunto das variáveis cujos valores foram obtidos através de simulações:

- α_{IL1}: Quantidade de acessos a uma cache L1 de instruções;
- α_{DL1}: Quantidade de acessos a uma cache L1 de dados;
- α_{SPM}: Quantidade de acessos a uma memória de rascunho;
- α_{L2}: Quantidade de acessos à cache L2;
- α_{MM}: Quantidade de acessos à memória principal.

A energia consumida por cada componente do subsistema de memória pode ser representada através das seguintes equações:

$$E_{IL1} = \alpha_{IL1} \times \varepsilon_{L1} \tag{4.1}$$

$$E_{DL1} = \alpha_{DL1} \times \varepsilon_{L1} \tag{4.2}$$

$$E_{SPM} = \alpha_{SPM} \times \varepsilon_{SPM} \tag{4.3}$$

$$E_{L2} = \alpha_{L2} \times \varepsilon_{L2} \tag{4.4}$$

$$E_{MM} = \alpha_{MM} \times \varepsilon_{MM} \tag{4.5}$$

Seja P o número de processadores presentes no sistema. A seguinte equação representa o consumo energético total da plataforma REF:

$$E_{TotalREF} = \sum_{i=1}^{P} (E^i_{IL1} + E^i_{DL1}) + E_{L2} + E_{MM} \tag{4.6}$$

A seguinte equação representa o consumo energético total da plataforma cSPM:

$$E_{TotalSPM} = \sum_{i=1}^{P} (E^i_{IL1} + E^i_{DL1} + E^i_{SPM}) + E_{L2} + E_{MM} \tag{4.7}$$

Também é possível calcular energias parciais, de blocos de memória individuais, tal como energia somente da cache L1 de dados.

4.5.2 Estimativa de energia por acesso

Os valores de energia por acesso da SPM e das caches L1 (instruções e dados) e L2 foram obtidos através do modelo físico CACTI 5.3 (WILTON; JOUPPI, 1996), assumindo-se nó tecnológico de 32nm. Para cada dispositivo de memória foi considerado o respectivo valor de energia por acesso de leitura retornado pelo CACTI.

Para as energias por acesso de leitura e escrita da memória principal, assumiu-se os valores referentes a uma memória *off-chip Micron MT48H8M16LF low-power SDRAM*, a qual é a mesma de outros trabalhos, como Egger, Lee e Shin (2006), Egger, Lee e Shin (2008) e Volpato (2010).

Os valores de energia por acesso para cada dispositivo de memória (em pJ) se encontram sumarizados na tabela 6.

Tabela 6: Consumo de energia por acesso conforme reportado pelo CACTI 5.3 (32nm) em pJ

Caches L1	4KB	8KB	16KB	32KB	64KB
2 vias	8,3	13,8	15,8	20,5	31,6
4 vias	14,2	14,6	19,8	21,8	31,4

SPM	
8KB	6,8

Cache L2	
256KB	184

Memória Principal	Acesso de Leitura	Acesso de Escrita
128MB	3370	1470

5 RESULTADOS EXPERIMENTAIS

A fim de se avaliar a eficácia da técnica proposta, as 400 aplicações geradas automaticamente (conforme detalhado na seção 4.4) foram simuladas nas 20 plataformas (10 baseadas na arquitetura REF e 10 baseadas na arquitetura cSPM, detalhadas na seção 4.3), com o uso da GEM5, resultando em 8000 simulações. As simulações foram realizadas em uma máquina com processador Intel Xeon E5430 (quad-core) 2,66GHz com 4GB de RAM, executando o sistema operacional Gentoo GNU/Linux (*kernel* 2.6.39, 64-bit). Foram necessárias aproximadamente 185 horas para completar as 8000 simulações.

Ao final de cada simulação foram coletados o número de acessos a cada bloco de memória (SPM, caches L1 de dados e de instruções, cache L2 e memória principal) bem como o número de faltas. Também foi coletado o número de ciclos despendidos por cada processador para executar a aplicação na plataforma considerada. O número de acessos a cada bloco de memória foi utilizado, juntamente com os valores de energia por acesso fornecidos pelo CACTI (ver tabela 6), para calcular a energia gasta em cada memória, e a energia total do subsistema de memória. Tais cálculos fizeram uso das equações 4.1 a 4.7. A energia gasta nos barramentos do sistema não foi contabilizada.

5.1 CORRELAÇÃO ENTRE REDUÇÃO MÉDIA DE ENERGIA E PERFIL DAS APLICAÇÕES

Como na técnica proposta a otimização de energia é obtida por meio da alocação de dados de pilha em SPM, é de se esperar que quanto maior for a proporção de acessos a dados de pilha da aplicação (P_S), maior deva ser a redução de energia proporcionada pela técnica. Por outro lado, considerando-se uma dada configuração do subsistema de memória (i.e., capacidades de SPM, caches e memória principal), o limite teórico para a redução de energia que a técnica pode proporcionar corresponde ao caso em que a aplicação acessa somente dados em pilha, i.e., $P_S = 1,0$.

A tabela 7 mostra a redução média (percentual) de energia no subsistema de memória obtida pelo uso da técnica proposta nas 5 plataformas cSPM que possuem caches L1 de 2 vias. As aplicações utilizadas possuem 4 valores distintos de P_S, $0,4$, $0,6$, $0,8$, $1,0$, tendo sido utilizadas 100 aplicações distintas para cada valor de P_S. Considerando

as aplicações que possuem um dado valor de P_S, a redução média de energia proporcionada pela técnica em uma dada plataforma cSPM (ou seja, o valor que consta em uma célula qualquer da tabela 7) foi obtida da seguinte forma. Para uma dada aplicação *app*, calculou-se a redução de energia (representada por R) no subsistema de memória utilizando-se a equação 5.1:

$$R(app) = (1 - E_{TotalSPM}(app)/E_{TotalREF}(app)) \times 100\% \quad (5.1)$$

onde $E_{TotalSPM}(app)$ é a energia do subsistema de memória quando *app* executa em cSPM e $E_{TotalREF}(app)$ é a energia do subsistema de memória quando *app* executa em REF. Uma vez calculado o valor de $R(app)$ para cada uma das 100 aplicações, calculou-se a redução média de energia utilizando-se a equação 5.2:

$$R_{med} = \sum_{i=1}^{100} R(app_i)/100 \quad (5.2)$$

Este cálculo foi realizado para cada célula da tabela 7.

A fim de facilitar sua interpretação e viabilizar a identificação de possíveis tendências, os dados da tabela 7 também estão representados sob forma de gráfico na figura 23.

Tabela 7: Redução média (percentual) de energia no subsistema de memória proporcionada pela técnica executando nas plataformas cSPM com caches L1 de 2 vias

Capacidade	Proporção de acessos à pilha nas aplicações			
das caches L1	$P_S = 0,4$	$P_S = 0,6$	$P_S = 0,8$	$P_S = 1,0$
4KB	-5,0%	-5,6%	-6,4%	-7,0%
8KB	5,8%	8,0%	9,9%	11,9%
16KB	8,1%	10,4%	12,6%	15,1%
32KB	11,0%	14,0%	16,7%	20,0%
64KB	14,7%	18,3%	21,6%	25,9%

Analisando-se o gráfico da figura 23 e os dados da tabela 7 é possível perceber que para todas as plataformas cSPM, exceto aquela com caches L1 de 4KB (a ser justificado mais adiante), a técnica proposta

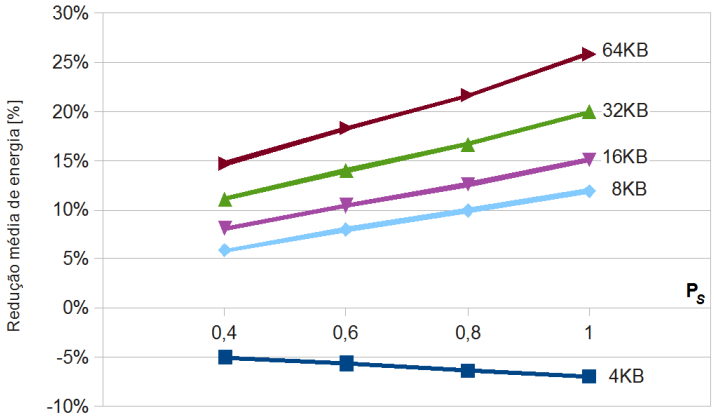

Figura 23: Redução média (percentual) de energia no subsistema de memória proporcionada pela técnica executando nas plataformas cSPM com caches L1 de 2 vias

resultou em redução de energia no subsistema de memória. Além disso, para essas plataformas, a redução de energia é maior para as aplicações que tem P_S maior, o que demonstra que a técnica explora, em certa proporção, o espaço de otimização disponível. Tal comportamento confirma parcialmente a expectativa inicial.

Considerando particularmente a plataforma com caches L1 de 8KB, a técnica proporcionou uma redução de energia de 5,8%, em média, para $P_S = 0{,}4$ e de 11,9%, em média, para $P_S = 1{,}0$. Assim, pode-se afirmar que 11,9% é a máxima redução de energia que a técnica pode proporcionar para esta plataforma. Conforme mencionado no início do capítulo 3, tal redução de energia advém do fato de que na arquitetura cSPM cada acesso a um dado de pilha gasta menos energia do que na arquitetura REF. Por exemplo, no caso das plataformas com caches L1 de 8KB em questão, a energia por acesso à cache L1 é de 13,8 pJ, ao passo que a energia por acesso à SPM de 8KB é de 6,8 pJ. Logo, para cada acesso há uma redução de energia de 50,7%. Além disso, o uso de SPM para alocar dados de pilha tem como consequência a redução no número de faltas na cache L1 de dados, como mostra o gráfico da figura 24. Com a redução do número de faltas, o acesso à cache L2 (cuja energia por acesso é uma ordem de magnitude maior do que a

das caches L1) é menos frequente, resultando em menor consumo de energia.

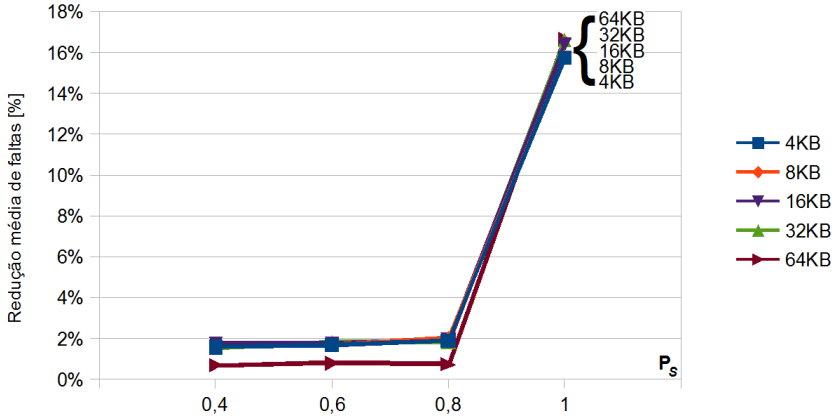

Figura 24: Redução média (percentual) de faltas na cache L1 de dados proporcionada pela técnica executando nas plataformas cSPM com caches L1 de 2 vias

Por outro lado, a redução de energia que a técnica proporciona está limitada pela penalidade imposta pela invocação do gerenciador de pilha. Basicamente, ao iniciar (ou terminar) a execução de um procedimento, as instruções referentes ao seu prólogo (ou ao seu epílogo) deverão ser executadas. Como tais instruções estão no espaço de endereçamento compartilhado, elas são acessadas através da cache L1 de instruções do processador no qual o procedimento executa, o que aumenta o número de acessos a esta cache. O gráfico da figura 25 mostra o aumento médio percentual no número de acessos às caches L1 de instruções para as plataformas cSPM com caches L1 de 2 vias, tomando-se como referência as plataformas REF também com caches L1 de 2 vias. Este gráfico mostra que, para um conjunto de 100 aplicações que possuem um dado valor de P_S (por exemplo, $P_S = 0{,}4$), o aumento médio no número de acessos às caches L1 de instruções é o mesmo para todas as plataformas cSPM (exceto para a plataforma cSPM com caches L1 de 4KB). Ou seja, o aumento médio no número de acessos às caches L1 de instruções depende principalmente do perfil das aplicações. Nota-se também que o aumento médio de acessos às caches L1 de instruções ficou entre 5% e

7%, aproximadamente.

O aumento no número de acessos às caches L1 de instruções também altera a localidade nestas caches, resultando em um maior número de faltas, conforme mostra o gráfico da figura 26. Neste gráfico, observa-se que o aumento no número de faltas é pequeno para as caches de maior capacidade (16KB, 32KB e 64KB), ficando entre 1,8% e 2,5%. Já para as caches menores (notadamente, a de 4KB), o aumento do número de faltas foi significativo. Particularmente, o aumento no número de faltas na cache L1 de instruções de 4KB (e o consequente aumento nos acessos à cache L2), aliado à pequena redução da energia resultante da relocação de dados desta cache para a SPM de 8KB (de 8,3 pJ para 6,8 pJ), justifica o aumento da energia no subsistema de memória da plataforma cSPM com caches L1 de 4KB e 2 vias, conforme mostrado no gráfico da figura 23.

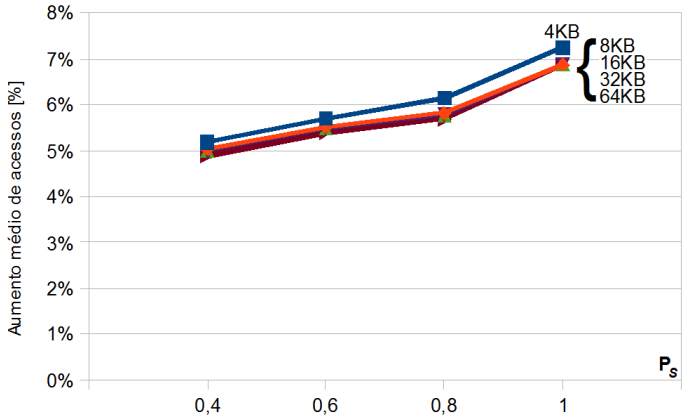

Figura 25: Aumento médio (percentual) de acessos à cache L1 de instruções imposto pela técnica executando nas plataformas cSPM com caches L1 de 2 vias

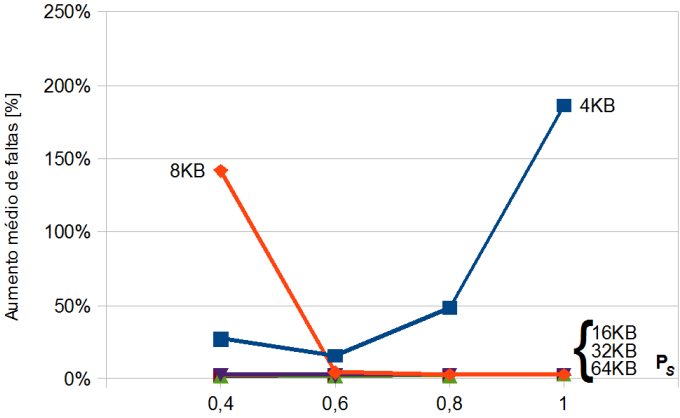

Figura 26: Aumento médio (percentual) de faltas na cache L1 de instruções imposto pela técnica executando nas plataformas cSPM com caches L1 de 2 vias

5.2 CORRELAÇÃO ENTRE REDUÇÃO MÉDIA DE ENERGIA E CARACTERÍSTICAS DAS CACHES L1

Considerando duas plataformas cSPM que se diferenciam somente pela capacidade das caches L1, a diferença da redução de energia no subsistema de memória proporcionada pela aplicação da técnica proposta é diretamente proporcional à diferença entre as energias gastas por acesso às respectivas caches L1. Por exemplo, no caso das plataformas com caches L1 de 2 vias, a energia por acesso às caches L1 de 8KB, 16KB e 32KB é 13,8 pJ, 15,8 pJ e 20,5 pJ, respectivamente. Assim, a diferença percentual entre as energias por acesso às caches L1 de 16KB e 8KB é 14,5%, ao passo que a diferença percentual entre as energias por acesso das caches L1 de 32KB e 16KB é 29,7%. Essas diferenças estão refletidas, a grosso modo, nas distâncias entre as respectivas curvas de redução média de energia mostradas no gráfico da figura 23. Repetindo tal análise para as plataformas com caches L1 de 4 vias, observa-se que a energia por acesso às caches L1 de 8KB, 16KB e 32KB é 14,6 pJ, 19,8 pJ e 21,8 pJ, respectivamente. Então, a diferença percentual entre as energias por acesso às caches L1 de 16KB e 8KB é 35,6%, ao passo que a diferença percentual entre as energias por acesso das caches L1

de 32KB e 16KB é 10,1%. Novamente, observa-se que estas diferenças correlatam com as distâncias entre as respectivas curvas de redução média de energia mostradas no gráfico da figura 27. Com base nesse comportamento, é possível afirmar que a redução de energia tende a ser diretamente proporcional ao aumento da capacidade das caches L1, já que a capacidade das SPMs nas plataformas cSPM foi fixada em 8KB.

As plataformas cSPM com caches de 4 vias associativas apresentam padrão de redução de energia do subsistema de memória semelhante às plataformas com caches de 2 vias, conforme mostram a tabela 8 e o gráfico da figura 27. A principal diferença reside no fato de que a técnica de gerenciamento de dados de pilha proposta resulta em uma redução de energia maior nas plataformas cSPM com caches L1 de 4 vias do que nas plataformas cSPM com caches L1 de 2 vias, exceto quando aplicações com $P_S = 1,0$ executam na plataforma cSPM com caches L1 de 64KB. Em particular, a técnica proposta resultou em redução de energia para a plataforma cSPM com caches L1 de 4KB e 4 vias, o que não havia ocorrido no caso da plataforma cSPM com caches L1 de 4KB e 2 vias. Tal comportamento justifica-se pela diferença entre a energia por acesso da cache L1 de 4 vias e 4KB e a energia por acesso da SPM de 8KB as quais, neste trabalho, correspondem a 14,2 pJ e 6,8 pJ, respectivamente, resultando em uma redução de energia de 52,1% a cada acesso a um dado de pilha. Outra diferença reside no fato da curva de redução de energia para a plataforma cSPM com caches de 16KB e 4 vias ter ficado bem próxima da curva de redução de energia para a plataforma cSPM com caches de 32KB e 4 vias. No caso das plataformas cSPM com caches de 2 vias esta distância é maior. Conforme exposto no parágrafo anterior, a distância entre as curvas de redução média de energia depende diretamente da relação entre as energias por acesso das respectivas caches L1.

Finalmente, observa-se que as reduções médias de energia para a plataforma cSPM com caches L1 de 64KB e 2 vias (tabela 7) são praticamente iguais às reduções médias de energia para a plataforma cSPM com caches L1 de 64KB e 4 vias (tabela 8). A mesma afirmação também é válida para as plataformas cSPM com caches L1 de 32KB. Assim, pode-se concluir que, para as plataformas cSPM com caches de maior capacidade, o grau de associatividade tem menor influência na redução de energia. Tal comportamento também é decorrente da diferença entre as energias por acesso das caches consideradas. Por exemplo, a energia por acesso para a cache L1 de 64KB de 2 vias é 31,6 pJ ao passo que a energia por acesso para a cache L1 de 64KB e 4 vias é 31,4 pJ.

Tabela 8: Redução média (percentual) de energia no subsistema de memória proporcionada pela técnica executando nas plataformas cSPM com caches L1 de 4 vias

Capacidade das caches L1	Proporção de acessos à pilha nas aplicações			
	$P_S = 0,4$	$P_S = 0,6$	$P_S = 0,8$	$P_S = 1,0$
4KB	4,2%	5,3%	6,5%	8,7%
8KB	6,6%	8,9%	10,9%	13,2%
16KB	10,7%	13,5%	16,1%	19,4%
32KB	11,6%	14,7%	17,5%	21,0%
64KB	14,7%	18,3%	21,6%	25,8%

5.3 REDUÇÃO MÉDIA DE *SNOOPS* NAS CACHES L1

A técnica proposta apresentou redução média de *snoops* nas caches L1 de até 3%, tanto para as plataformas cSPM com 2 vias quanto para as com 4 vias, conforme mostrado nas tabelas 9 e 10, respectivamente.

Entretanto, para as plataformas cSPM com caches de 4KB, a utilização da técnica proposta resultou em aumento no número de *snoops*. Para compreender o motivo deste aumento, faz-se necessário analisar o impacto da técnica sobre o número de faltas nas caches L1 de dados e nas caches L1 de instruções para as plataformas cSPM com caches de 4KB. A aplicação da técnica proposta proporcionou a redução da quantidade de faltas nas caches L1 de dados para todas as plataformas cSPM, inclusive para aquelas com caches L1 de 4KB. Porém, para as plataformas cSPM com caches de baixa capacidade (notadamente, 4KB), a aplicação da técnica resultou em aumento significativo das faltas nas caches L1 de instruções. No balanço final, o total de faltas em caches L1 aumentou (notadamente, nos casos com maior P_S), resultando assim em aumento na quantidade de *snoops*.

É importante observar que, em todas as plataformas utilizadas nos experimentos, os conjuntos de *tags* das caches L1 (dados e instruções) estão duplicados. Tal duplicação tem por objetivo permitir que, ao ocorrer uma operação de *snoop*, a busca por um dado ou instrução possa ser feita mediante o acesso ao conjunto extra de *tags* das caches L1, evitando a interrupção de um eventual acesso à cache efetuado

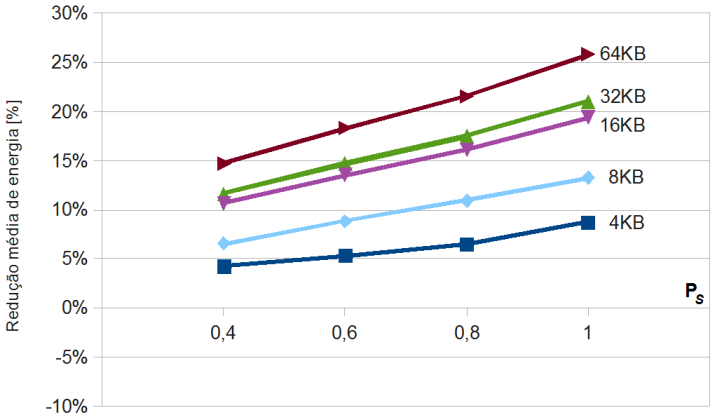

Figura 27: Redução média (percentual) de energia no subsistema de memória proporcionada pela técnica executando nas plataformas cSPM com caches L1 de 4 vias

pelo respectivo processador. Por outro lado, devido às limitações de tempo, não foi desenvolvido um modelo para o consumo de energia por transação de barramento. Desta forma, com relação à energia consumida por operações de *snoop*, os valores de redução média de energia no subsistema de memória reportados neste capítulo não levaram em conta a energia do barramento, mas somente a energia referente aos acessos aos conjuntos extras de *tags*. A inclusão do modelo de energia de barramento e a análise dos resultados de redução de energia no subsistema de memória decorrentes desta inclusão estão elencados nos trabalhos futuros.

5.4 PENALIDADE DE CICLOS DE CPU

A utilização de um gerenciador de dados de pilha de *threads* em SPMs impõe um aumento no número de ciclos de CPU necessários para concluir a execução de uma aplicação. Para o conjunto de aplicações geradas, a técnica resulta em um aumento médio de 16,7% no número de ciclos. Tal penalidade é diretamente influenciada pela quantidade de chamadas de procedimentos que são executados por *threads* em uma

Tabela 9: Redução média (percentual) de *snoops* no subsistema de memória proporcionada pela técnica executando nas plataformas cSPM com caches L1 de 2 vias

Capacidade das caches L1	Proporção de acessos à pilha nas aplicações			
	$P_S = 0,4$	$P_S = 0,6$	$P_S = 0,8$	$P_S = 1,0$
4KB	-3,6%	-5,5%	-10,7%	-150,9%
8KB	0,6%	1,2%	1,7%	2,0%
16KB	1,5%	1,6%	1,6%	2,3%
32KB	1,4%	1,7%	1,6%	2,9%
64KB	0,5%	0,7%	0,5%	3,0%

aplicação. Quanto maior o número de chamadas de procedimentos em uma aplicação, maior será também o número de chamadas ao gerenciador de pilha.

Outro fator determinante é o tamanho de código desses procedimentos. Notadamente, a técnica proposta tende a ser mais eficiente para aplicações cujos procedimentos são longos, uma vez que nestes casos o número de acessos à SPM é maior, resultando em economia de enegia que compensa o aumento nos acessos à cache L1 de instruções oriundo das instruções extras referentes ao prólogo e ao epílogo dos procedimentos.

5.5 COMPARAÇÃO COM TRABALHOS CORRELATOS

Para o presente trabalho, não foi possível realizar uma análise comparativa com as técnicas de trabalhos correlatos.

Os trabalhos de Bai e Shrivastava (2010) e Deng et al. (2011), apesar de apresentarem uma infraestrutura similar à utilizada neste trabalho, consideram apenas dados de *heap* para a alocação dinâmica em SPMs. Já Bai, Shrivastava e Kudchadker (2011) realizam a alocação de dados de pilha, porém não apresentam análises de consumo de energia, inviabilizando uma comparação entre as técnicas.

Tabela 10: Redução média (percentual) de *snoops* no subsistema de memória proporcionada pela técnica executando nas plataformas cSPM com caches L1 de 4 vias

Capacidade das caches L1	Proporção de acessos à pilha nas aplicações			
	$P_S = 0,4$	$P_S = 0,6$	$P_S = 0,8$	$P_S = 1,0$
4KB	-2,7%	-5,0%	-10,7%	-152,1%
8KB	0,5%	1,0%	1,1%	2,1%
16KB	1,3%	1,4%	1,1%	2,8%
32KB	1,3%	1,3%	1,1%	3,0%
64KB	1,2%	1,4%	1,2%	3,0%

6 CONCLUSÕES

A presente dissertação apresentou uma técnica para redução de energia no subsistema de memória de MPSoCs a qual baseia-se na alocação em memória de rascunho (*Scratchpad Memory* - SPM) dos dados de pilha de *threads*. A técnica pressupõe que a aplicação execute em um MPSoC cujo subsistema de memória possua memória cache L2 compartilhada por todos os processadores (ou seja, um espaço de endereçamento compartilhado) e que cada processador possua uma cache L1 de dados, uma cache L1 de instruções e uma SPM (cujos endereços são disjuntos em relação ao espaço de endereçamento compartilhado), todas privativas. A técnica é posta em prática por meio de um gerenciador totalmente em software, o qual é responsável por alocar e desalocar os dados de pilha de *thread* na SPM privativa ao processador no qual a *thread* é executada. A fim de reduzir a penalidade imposta pelo uso da técnica, há uma cópia dos binários do gerenciador em cada SPM.

Considerando-se os resultados obtidos nos experimentos realizados, foi possível observar que a redução média de energia proporcionada pela técnica proposta neste trabalho é dependente principalmente de dois fatores:

- da proporção de acessos a dados de pilha (representada por P_S ao longo deste texto);
- da relação entre capacidade da cache L1 de dados e capacidade da SPM.

O primeiro fator está relacionado ao perfil das aplicações, ao passo que o segundo diz respeito à organização do subsistema de memória.

Ao se propor a técnica, já se esperava que a redução de energia dependeria da proporção de acessos a dados de pilha. Entretanto, os resultados experimentais permitiram quantificar tal dependência. Em particular, utilizando-se aplicações que acessam predominantemente dados de pilha ($P_S = 1,0$), foi possível determinar a máxima redução de energia proporcionada pela técnica para cada organização de subsistema de memória (i.e., para cada uma das plataformas cSPM consideradas). Por exemplo, para as plataformas cSPM com caches L1 de 2 vias, as máximas reduções de energia observadas foram 11,9%, 15,1%, 20,0% e 25,9% para as plataformas com caches L1 de 8KB, 16KB, 32KB e 64KB, respectivamente. No caso das plataformas cSPM com caches de 4 vias, as máximas reduções de energia observadas foram 8,7%, 13,2%,

19,4%, 21,0% e 25,8% para as plataformas com caches L1 de 4KB, 8KB, 16KB, 32KB e 64KB, respectivamente.

A redução média de energia obtida através da aplicação da técnica proposta é consequência direta da relação entre a energia por acesso à cache L1 de dados e a energia por acesso à SPM. Considerando-se aplicações com uma dada característica de proporção de acessos à dados de pilha, quanto maior for a diferença entre tais energias, maior tende a ser a economia de energia obtida ao se relocar os dados de pilha para a SPM. Logo, tendo-se mantido o tamanho da SPM em 8KB, a maior economia de energia foi observada nas plataformas com caches L1 de dados de 64KB e de 32KB, sendo que nestes casos não houve diferença significativa entre as plataformas com caches L1 de 2 vias e aquelas com caches L1 de 4 vias. Já nos casos das plataformas com caches L1 de 8KB e 16KB, a maior economia de energia foi observada nos casos em que as caches L1 possuem grau de associatividade 4.

Para a plataforma cSPM com caches L1 de 4KB e grau de associatividade 2, a aplicação da técnica resultou em aumento de energia. Isso ocorreu porque o aumento do número de faltas na cache L1 de instruções resultou em um aumento de energia que não pode ser compensado pela redução de energia nos acessos à cache L1 de dados. Esse aumento pode ser justificado pelo fato de a energia por acesso à cache L1 de dados com 2 vias ser de apenas 1,32 vezes a energia por acesso à SPM (de 8KB). Já no caso da plataforma cSPM com caches L1 de 4KB e grau de associatividade 4, a aplicação da técnica resultou em redução de energia, pois o aumento de energia decorrente do aumento das faltas de cache L1 de instruções foi mais que compensado pela redução de energia nos acessos aos dados de pilha. Observa-se que a energia por acesso à cache L1 com associatividade 4 é 2,09 vezes a energia por acesso à SPM (de 8KB).

Considerando que a área de endereçamento das SPMs e o espaço de endereços acessáveis via caches são disjuntos, a relocação da pilha de *threads* para SPM também contribui para a redução do número de requisições de coerência (*snoops*) nas caches L1, o que resulta em redução de energia consumida no subsistema de memória. A técnica proposta apresentou redução média de *snoops* nas caches L1 de até 3%, ganho esse que foi limitado pelo alto número de faltas nas caches L1 de instruções.

A utilização de um gerenciador de dados de pilha de *threads* em SPMs impõe um aumento no número de ciclos de CPU necessários para concluir a execução de uma aplicação. Para o conjunto de aplicações geradas, a técnica resulta em um aumento médio de 16,7% no número de

ciclos. Pode-se concluir que tal penalidade é diretamente influenciada pela quantidade de chamadas de procedimentos que são executados por *threads* em uma aplicação. Quanto maior o número de chamadas de procedimentos em uma aplicação, maior será também o número de chamadas ao gerenciador de pilha. Por outro lado, dado um número de chamadas de procedimentos, tanto maior será a redução de energia obtida pela técnica proposta quanto maior for o número de instruções dos procedimentos.

6.1 TRABALHOS FUTUROS

A implementação da técnica proposta no presente trabalho possui algumas limitações que são oriundas da infraestrutura utilizada. Como exemplo, cita-se o uso de processador de 64 bits, quando o mais frequente em sistemas embarcados é o uso de processadores de 32 bits. Além disso, para sua validação foram utilizadas aplicações sintéticas geradas de forma a cobrir um conjunto de características. Esse conjunto de características teve que ser limitado em função do longo tempo de simulação (quase 48 horas para simular cada conjunto de 100 aplicações nas 20 plataformas). Por outro lado, a partir da análise dos resultados obtidos percebeu-se que seria conveniente gerar mais aplicações com outras características, como por exemplo, variando-se a proporção de chamadas de procedimentos, de leituras e de escritas em dados de pilha. Também percebeu-se que pode ser útil gerar resultados a partir de aplicações nas quais outros parâmetros, como por exemplo o número de operações em cada procedimento, sejam restringidos.

As características das plataformas utilizadas também se mostraram muito importantes para a redução de energia proporcionada pela técnica. Neste contexto, observa-se que a técnica poderia ser avaliada de maneira mais justa fixando-se a quantidade de memória disponível para dados. Assim, ao invés de derivar uma plataforma cSPM mediante a simples adição de uma SPM (de capacidade arbitrária) à plataforma REF, a plataforma cSPM teria uma cache L1 de dados e uma SPM, cada uma destas com capacidade igual à metade da capacidade da cache L1 de dados da plataforma REF. Fazendo isso, a capacidade total de armazenamento em cache L1 seria a mesma para ambas arquiteturas, cSPM e REF.

Conforme comentado na seção 5.3, a energia associada aos *snoops* foi considerada apenas parcialmente, somando-se a energia referente aos acessos aos conjuntos extras de *tags* das caches L1. Para que a

energia de *snoops* possa ser considerada com maior precisão é necessário utilizar um modelo de energia por transação no barramento, o que permitiria verificar o impacto energético da redução no número de *snoops* proporcionada pela técnica. Esta redução ficou entre 0,5% e 5,0%, em média.

Assim, vislumbram-se os seguintes trabalhos futuros:

- Criar infraestrutura que possibilite avaliar a aplicação da técnica em sistemas multiprocessados de 32 bits;
- Avaliar a influência da proporção de tipos de operações na redução de energia e na penalidade de número de ciclos decorrentes da aplicação da técnica;
- Modelar a energia por transação no barramento e avaliar a redução de energia oriunda da redução do número de *snoops*;
- Avaliar o impacto da técnica quando uma plataforma cSPM possui a mesma capacidade de memória que uma plataforma REF.

REFERÊNCIAS BIBLIOGRÁFICAS

ALFRED, V.; SETHI, R.; ULLMAN, J. *Compilers: Principles, Techniques and Tools*. [S.l.]: Addison-wesley, 1986.

ARM Corporation. *ARM Cortex-A Series*. 2011. <http://www.arm.com/products/processors/cortex-a>.

ARM Corporation. *ARM Cortex-R Series*. 2011. <http://www.arm.com/products/processors/cortex-r>.

ARM Corporation. *ARM Cortex-R7 Processor*. 2011. <http://www.arm.com/products/processors/cortex-r/cortex-r7.php>.

BAI, K.; SHRIVASTAVA, A. Heap data management for Limited Local Memory (LLM) multi-core processors. In: IEEE. *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*. [S.l.], 2010. p. 317–325.

BAI, K.; SHRIVASTAVA, A.; KUDCHADKER, S. Stack data management for Limited Local Memory (LLM) multi-core processors. In: IEEE. *2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. [S.l.], 2011. p. 231–234.

BALLAPURAM, C. S.; SHARIF, A.; LEE, H.-H. S. Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. In: *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2008. p. 60–69. ISBN 978-1-59593-958-6.

BINKERT, N. et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, ACM, v. 39, n. 2, p. 1–7, 2011.

BINKERT, N. et al. The m5 simulator: Modeling networked systems. *Micro, IEEE*, IEEE, v. 26, n. 4, p. 52–60, 2006.

CANTIN, J. F.; LIPASTI, M. H.; SMITH, J. E. Improving multiprocessor performance with coarse-grain coherence tracking. In: *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005. p. 246–257. ISBN 0-7695-2270-X.

CENSIER, L. M.; FEAUTRIER, P. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, v. 27, p. 1112–1118, 1978.

CHEN, T. et al. Cell Broadband Engine Architecture and Its First Implementation — A Performance View. *IBM Journal of Research and Development*, IBM, v. 51, n. 5, p. 559–572, 2007.

CHO, H. et al. Dynamic data scratchpad memory management for a memory subsystem with an MMU. *ACM SIGPLAN Notices*, ACM, v. 42, n. 7, p. 195–206, 2007.

COARFA, C. et al. An evaluation of global address space languages: co-array fortran and unified parallel c. In: ACM. *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. [S.l.], 2005. p. 36–47.

DALLY, W. et al. Efficient embedded computing. *Computer*, IEEE, v. 41, n. 7, p. 27–32, 2008.

DENG, N. et al. A semi-automatic scratchpad memory management framework for CMP. *Advanced Parallel Processing Technologies*, Springer, p. 73–87, 2011.

EGGER, B.; LEE, J.; SHIN, H. Scratchpad memory management for portable systems with a memory management unit. In: ACM. *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. [S.l.], 2006. p. 321–330.

EGGER, B.; LEE, J.; SHIN, H. Dynamic scratchpad memory management for code in portable systems with an MMU. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, v. 7, n. 2, p. 11, 2008.

EKMAN, M.; DAHLGREN, F.; STENSTRöM, P. Evaluation of snoop-energy reduction techniques for chip-multiprocessors. In: *Workshop on Duplicating, Deconstructing, and Debunking*. [S.l.: s.n.], 2002.

EKMAN, M.; STENSTRöM, P.; DAHLGREN, F. Tlb and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In: *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2002. p. 243–246. ISBN 1-58113-475-4.

FISCHER, C. N.; CYTRON, R.; LEBLANC, R. *Crafting a Compiler*. [S.l.]: Pearson Education, 2010.

GNU. *GNU Compiler Collection*. 2012. <http://gcc.gnu.org>.

INSTRUMENTS, T. *TI OMAP*. 2011.

JERRAYA, A.; WOLF, W. *Multiprocessor Systems-on-Chips*. [S.l.]: Morgan Kaufmann, 2005. (The Systems on Silicon Series). ISBN 9780123852519.

KANNAN, A. et al. A software solution for dynamic stack management on scratch pad memory. In: *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 2009. p. 612–617. ISBN 978-1-4244-2748-2.

KEUTZER, K. et al. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, v. 19, n. 12, p. 1523–1543, 2000.

LAWRENCE, R. A Survey of Cache Coherence Mechanisms in Shared Memory Multiprocessors. *Department of Computer Science, University of Manitoba, Manitoba, Canada*, 1998.

LILJA, D. J. Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 25, p. 303–338, September 1993. ISSN 0360-0300. <http://doi.acm.org/10.1145/158439.158907>.

LOGHI, M.; PONCINO, M.; BENINI, L. Cache coherence tradeoffs in shared-memory mpsocs. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 5, n. 2, p. 383–407, 2006. ISSN 1539-9087.

MENG, J.; SKADRON, K. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In: *Proceedings of the 2009 IEEE international conference on Computer design*. Piscataway, NJ, USA: IEEE Press, 2009. (ICCD'09), p. 282–288. ISBN 978-1-4244-5029-9. <http://portal.acm.org/citation.cfm?id=1792354.1792409>.

MOSHOVOS, A. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. *Computer Architecture, International Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 234–245, 2005. ISSN 1063-6897.

MOSHOVOS, A. et al. Jetty: Filtering snoops for reduced energy consumption in smp servers. In: *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2001. p. 85. ISBN 0-7695-1019-1.

PARK, S.; PARK, H.; HA, S. A novel technique to use scratch-pad memory for stack management. In: IEEE. *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*. [S.l.], 2007. p. 1–6.

PATEL, A.; GHOSE, K. Energy-efficient mesi cache coherence with pro-active snoop filtering for multicore microprocessors. In: *ISLPED '08: Proceeding of the 13th international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2008. p. 247–252. ISBN 978-1-60558-109-5.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. 4th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 0123744938, 9780123744937.

RAMBO, E. A.; HENSCHEL, O. P.; SANTOS, L. C. V. dos. Automatic Generation of Memory Consistency Tests for Chip Multiprocessing. In: *2011 18th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. [S.l.: s.n.], 2011. p. 542–545.

STENSTRöM, P. A survey of cache coherence schemes for multiprocessors. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 6, p. 12–24, 1990. ISSN 0018-9162.

VOLPATO, D. P. *Gerenciamento Explícito de Memória Auxiliar a partir de Arquivos-Objeto para Melhoria da Eficiência Energética de Sistemas Embarcados*. Dissertação (Mestrado em Ciência da Computação) — Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Santa Catarina, Florianópolis, 2010.

WILTON, S. J. E.; JOUPPI, N. P. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, v. 31, p. 677–688, 1996.

WOO, S. C. et al. The SPLASH-2 programs: characterization and methodological considerations. In: *ISCA '95: Proceedings of the 22nd*

annual International Symposium on Computer Architecture. New York, NY, USA: ACM, 1995. p. 24–36. ISBN 0-89791-698-0.

YOUNG, J. et al. To snoop or not to snoop: Evaluation of fine-grain and coarse-grain snoop filtering techniques. In: *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*. Berlin, Heidelberg: Springer-Verlag, 2008. p. 141–150. ISBN 978-3-540-85450-0.

YU, C.; ZHOU, X.; PETROV, P. Low-power inter-core communication through cache partitioning in embedded multiprocessors. In: *SBCCI '09: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design*. New York, NY, USA: ACM, 2009. p. 1–6. ISBN 978-1-60558-705-9.

ZHOU, X. et al. Application-aware snoop filtering for low-power cache coherence in embedded multiprocessors. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 13, n. 1, p. 1–25, 2008. ISSN 1084-4309.