UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GUSTAVO GIRÃO BARRETO DA SILVA

Estudo sobre o Impacto da Hierarquia de Memória em MPSoCs baseados em NoC

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, Fevereiro de 2009.

DADOS INTERNACIONAIS DE CATALOGRAÇÃO NA PUBLICAÇÃO (CIP)

S586e Silva, Gustavo Girão Barreto da.

Estudo sobre o Impacto da Hierarquia de Memória em MPSoCs baseados em NoC / Gustavo Girão Barreto da Silva. – Porto Alegre: PPGC da UFRGS, 2009.

96 p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2009. Orientador: Prof. Dr. Flávio Rech Wagner.

1. Sistemas embarcados. 2. Sistemas-em-chip multiprocessados. 3. Redes-em-chip. 4. Coerência em cache. 5. Migração de tarefas. I. Wagner, Flávio Rech. II. Título.

Ficha Catalográfica elaborada por: Alexsander Borges Ribeiro – CRB 10/1932

AGRADECIMENTOS

Agradeço, primeiramente, à minha mãe que sempre me apoiou em tudo que eu fiz e principalmente nos meus estudos. Agradeço a meu irmão que me influenciou em muitas coisas da minha vida inclusive meus estudos.

Agradeço ao meu orientador Flávio Rech Wagner que cumpriu com perfeição seu papel de orientador e me ajudou muito.

Sou grato também a todos os meus amigos, sem exceção, da turma de 2002 do curso de Ciências da Computação da UFRN. Vocês continuam sendo importantes para mim. Igualmente, agradeço aos Exilados de Natal em Porto Alegre: Monica, André, Renata, Marcela, Paulo, Alexsandra e principalmente Jonatas pelo apoio, camaradagem e companhia em momentos em que a saudade de casa é grande.

Não poderia ter feito este trabalho sem o apoio (científico ou não) de colegas do LSE como Daniel, Mateus, Vítor, Tomás, Márcio, Leonardo, Marco entre outros.

Obrigado.

"You can fool some of the people all of the time, and all of the people some of the time, but you can not fool all of the people all of the time."

Abraham Lincoln

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ATA	*Address Table*
BE	*Best Effort*
CA	*Communication Assist*
CaCoMa	*Cache Communication Manager*
CHESS	*Cache Hierarchy Estimator using Scalabe Simulator*
DCOS	*Directory Cache On Switch*
DCT	*Discrete cosine transform*
DMA	*Direct Memory Access*
DPM	*Dynamic Power Management*
DSP	*Digital Signal Processor*
DVS	*Dynamic Voltage Scaling*
FFT	*Fast Fourier Transform*
FIFO	*First In First Out*
GT	*Guaranteed Troughput*
HwMMU	*Hardware Memory Management Unit*
JVM	*Java Virtual Machine*
JPEG	*Joint Pictures Expert Group*
LRU	*Least Recently Used*
LSE	Laboratório de Sistemas Embarcados
LFU	*Least Frequently Used*
MCDSE	*Memory Configuration Design Space Exploration*
MHLA	*Memory hierarchy layer assignment*
MPEG	*Moving Picture Experts Group*
MPSoC	*Multi-processor System on Chip*
nDMA	NoC DMA
NoC	*Network on Chip*
NUMA	*Non Uniform Memory Access*

PE	*Processor Element*
PTA	*Processor Table*
SD-TLB	*Shared Translation Look-Aside Buffer*
STA	*Status Table*
QCIF	*Quarter Common Intermediate Format*
QoS	*Quality of Service*
QSDPCM	*Quad-Tree Structured Differential Pulse Code Modulation*
SIMPLE	*SIMPLE Multiprocessor Platform Environment*
SoC	*System on Chip*
TLB	*Translation Look-aside Buffer*
TLM	*Transaction Level Model*
UFRGS	Universidade Federal do Rio Grande do Sul
VLIW	*Very Large Instruction Word*
VTC	*Virtual Tree Coherence*
WTI	*Write-Through Invalidate*
WTU	*Write-Through Update*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Ao longo dos últimos anos, os sistemas embarcados vêm se tornando cada vez mais complexos tanto em termos de hardware quanto de software. Ultimamente têm-se adotado como solução o uso de MPSoCs (sistemas multiprocessados integrados em chip) para uma maior eficiência energética e computacional nestes sistemas. Com o uso de diversos elementos de processamento, redes-em-chip (NoC - networks-on-chip) aparecem como soluções de melhor desempenho do que barramentos. Nestes ambientes cujo desempenho depende da eficiência do modelo de comunicação, a hierarquia de memória se torna um elemento chave.

Baseando-se neste cenário, este trabalho realiza uma investigação sobre o impacto da hierarquia de memória em MPSoCs baseados em NoC. Dentro deste escopo foi desenvolvida uma nova organização de memória fisicamente centralizada com diferentes espaços de endereçamentos denominada nDMA. Este trabalho também apresenta uma comparação entre a nova organização e outras três organizações bastante difundidas tais como memória distribuída, memória compartilhada e memória compartilhada distribuída. Estas duas ultimas adotam um modelo de coerência de cache baseado em diretório completamente desenvolvido em hardware. Os modelos de memória foram implementados na plataforma virtual SIMPLE (*SIMPLE Multiprocessor Platform Environment*).

Resultados experimentais mostram uma forte dependência com relação à carga de comunicação gerada pelas aplicações. O modelo de memória distribuída apresenta melhores resultados conforme a carga de comunicação das aplicações é baixa. Por outro lado, o novo modelo de memória fisicamente compartilhado com diferentes espaços de endereçamento apresenta melhores resultados conforme a carga de comunicação das aplicações é alta.

Também foram realizados experimentos objetivando analisar o desempenho dos modelos de memória em situações de alta latência de comunicação na rede. Resultados mostram melhores resultados do modelo de memória distribuída quando a carga de comunicação das aplicações é alta e, caso contrário, o modelo nDMA apresenta melhores resultados.

Por fim, foram analisados os desempenhos dos modelos de memória durante o processo de migração de tarefas. Neste caso, os modelos de memória compartilhada e compartilhada distribuída apresentaram melhores resultados devido ao fato de que não se faz necessária o envio dos dados da aplicação nestes modelos e também devido ao menor tamanho de código se comparado com os outros modelos.

Palavras-Chave: sistemas embarcados, sistemas-em-chip multiprocessados, redes-em-chip, coerência de cache, migração de tarefas.

Study on the Impact of Memory Hierarchy in NoC-based MPSoCs

ABSTRACT

In the past few the years, embedded systems have become even more complex both on terms of hardware and software. Lately, the use of MPSoCs (Multi-Processor Systems-on-Chip) has been adopted on these systems for a better energetic and computational efficiency. Due to the use of several processing elements, Networks-on-Chip arise as better performance solutions than buses.

Considering this scenario, this work performs an investigation on the impact of memory hierarchy in NoC-based MPSoCs. In this context, a new physically centralized and shared memory organization with different address spaces named nDMA was developed. This work also presents a comparison between the new memory organization and three different well-known memory hierarchy models such as distributed memory and shared and distributed shared memories that make use of a fully hardware cache coherence solution. The memory models were implemented in the SIMPLE (SIMPLE Multiprocessor Platform Environment) virtual platform.

Experimental results shows a strong dependency on the application communication workload. The distributed memory model presents better results as the application communication workload is low. On the other hand, the new memory model (physically shared with different address spaces) presents better results as the application communication workload is high.

There were also experiments aiming at observing the performance of the memory models in situations where the communication latency on the network is high. Results show better results of the distributed memory model when the application communication workload is high, and the nDMA model presents better results otherwise.

Finally, the performance of the memory models during a task migration process were evaluated. In this case, the shared memory and distributed shared memory models presented better results due to the fact that in this case the data memory does not need to be transferred from one point to another and also due to the low size of the memory code in these cases if compared to other memory models.

Keywords: embedded systems, multiprocessor system-on-chip, network-on-chip, cache coherence, task migration.

1 INTRODUÇÃO

Ao longo dos últimos anos, os sistemas embarcados vêm se tornando cada vez mais complexos. Entre outros fatores, esta complexidade se dá devido ao uso destes sistemas como de propósito geral levando ao uso de softwares mais variados e mais complexos. Para dar suporte a esta situação, o hardware destes sistemas também se torna mais complexo. Em sistemas computacionais convencionais, o uso de elementos de processamento com frequências de operação cada vez maiores apresenta pouco aumento de performance computacional e um consumo de energia muito elevado. Levando-se em conta que sistemas embarcados apresentam severas restrições energéticas (MARWEDEL, 2003), este tipo de solução torna-se inviável.

Devido a este problema e graças ao alto grau de integração em um chip que se alcança nos dias atuais, tem-se adotado como solução o uso de sistemas multiprocessados em chip único (MPSoC, do inglês *Multiprocessor System-on-Chip*). Em MPSoCs utiliza-se múltiplos processadores que operam em uma frequência mais baixa do que a de processadores de propósito geral convencionais para que se obtenha eficiência energética mantendo a performance computacional necessária. O paralelismo deste sistema é a chave para obter o ganho de performance desejado.

A grande maioria dos sistemas multiprocessados apresentam barramentos como mecanismo de comunicação. Entretanto, estas estruturas sofrem de um baixo grau de escalabilidade no sentido de que a inclusão de poucos elementos no sistema causa uma degradação da comunicação como um todo. Com o objetivo de ter um mecanismo de comunicação mais eficiente foi proposta uma solução inspirada nas redes de computadores conhecida como rede-em-chip (NoC, do inglês *Network-on-Chip*) (BENINI, 2002). NoC é uma estrutura de comunicação composta por diversos roteadores, conectados entre si, seguindo uma determinada topologia (ex.: *mesh*, *torus*, cubo). Cada roteador está associado a um recurso da rede (processadores, memórias ou módulos de E/S, por exemplo) que envia mensagens que são roteadas pela rede até seu destino. Além do alto grau de escalabilidade (ZEFERINO, 2002), NoCs têm por principal característica a capacidade de comunicação paralela entre os elementos do sistema.

1.1 Objetivos

Dado o grande conjunto de aspectos a serem explorados em uma hierarquia de memória, diversos trabalhos têm abordado aspectos de implementação e desempenho. Nos últimos anos, devido à vasta proliferação dos sistemas embarcados, a eficiência energética vem sendo considerada um outro aspecto crucial. Entretanto poucos

trabalhos realizam um estudo de hierarquia de memória em MPSoCs que utilizam NoCs como mecanismo de comunicação. É importante perceber que o uso de NoCs leva a uma comunicação paralela no sistema, o que pode afetar diretamente o desempenho da hierarquia de memória. Além disso, o uso de NoCs acarreta em uma preocupação não existente quando se utiliza barramentos: o mapeamento físico dos recursos no sistema.

Considerando este contexto de MPSoCs baseados em NoCs, esta dissertação tem por objetivo principal analisar o impacto da hierarquia de memória neste ambiente. Pretende-se avaliar características importantes de sistemas embarcados tais como performance e eficiência energética de diferentes modelos de memória variando aspectos de organização física e lógica das mesmas. Além disso pretende-se analisar a tolerância de tais modelos de memória a diferentes níveis de latência de comunicação. Outro objetivo é o de avaliar o custo de migração de tarefas nestes modelos de memória a fim de avaliá-los quanto aos aspectos de performance e eficiência energética já mencionados.

1.2 Contribuições

A principal contribuição desta dissertação é a avaliação de diferentes organizações de memória em um cenário com múltiplos processadores que utilizam uma NoC como meio físico de comunicação. Esta avaliação leva em conta principalmente o desempenho de aplicações utilizando estas organizações bem como a eficiência energética de cada componente do sistema. No que diz respeito ao ambiente que faz uso da NoC, é realizada também uma análise quanto à tolerância em situações com diferentes latências de comunicação. Uma outra contribuição é a avaliação de um modelo de migração de tarefas previamente proposto em cada um dos ambientes com diferentes modelos de memória.

Outras contribuições de menor porte podem ser citadas, tais como:

- Desenvolvimento de um modelo de memória compartilhada com espaço de endereçamento distribuído bem como um mecanismo próprio de comunicação fazendo uso das vantagens do modelo.

- Implementação de diferentes modelos de memória em uma plataforma virtual previamente desenvolvida, provendo, assim, a possibilidade de avaliações futuras de outros aspectos considerando a hierarquia de memória.

- Desenvolvimento de diferentes aplicações utilizando diferentes mecanismos de comunicação.

A fim de realizar o estudo destes aspectos em sistemas multiprocessados baseados em NoC mencionados anteriormente, foram implementados três diferentes modelos de memória em um ambiente que utiliza NoC:

- **Memória compartilhada**: uso de um único módulo de memória com espaço de endereçamento compartilhado em um nó exclusivo da rede (não há processadores neste nó). Cada processador possui caches privadas com suporte a solução de coerência de cache baseada em diretório.

- **Memória compartilhada distribuída**: ambiente similar ao de memória compartilhada. A diferença está no fato de que existe mais de um módulo de memória em nós distintos da rede e todos os módulos, em conjunto, formam um único espaço de endereçamento. Neste ambiente também se utiliza caches privadas com solução de coerência de cache baseada em diretório implementada em trabalhos anteriores fora do escopo desta dissertação. É necessário saber em que módulo de memória se encontra o dado para realizar uma requisição de acesso (leitura ou escrita).
- **nDMA:** Neste modelo, existe somente um módulo de memória no sistema. Entretanto este módulo é formado por bancos de memória onde cada banco contém os dados de um processador. Desta maneira quando uma requisição chega no nó da memória, o controlador de memória verifica qual o processador requisitante e baseado nesta informação acessa o banco de memória correto. Nesta solução também existe o uso de caches locais e privadas, entretanto como o espaço de endereçamento é distribuído, não existe a necessidade de uma solução de coerência de cache. Este modelo foi desenvolvido no escopo desta dissertação e é considerado uma contribuição da mesma.

Como dito anteriormente, a hierarquia de memória impacta diretamente no modelo de programação. No caso da memória distribuída a comunicação se dá por troca de mensagens enquanto que nas memórias compartilhada e compartilhada distribuída os processadores se comunicam através de variáveis compartilhadas. No modelo nDMA, por se tratar de memória compartilhada com espaço de endereçamento distribuído, a comunicação se dá de maneira diferente dos outros casos. Aproveitando-se do fato de que as memórias de todos os processadores presentes no sistema estão presentes em um único nó da rede como uma memória compartilhada, um processador que queria enviar uma certa quantidade de dados (mensagem) só precisa indicar ao controlador da memória os dados que ele deseja transferir, a quantidade de bytes a serem transferidos e o processador destino. Desta maneira, os dados que trafegam pela rede no processo de comunicação são minimizados. Ao receber a mensagem o controlador de memória realiza a transferência dos dados de um banco de memória (do processador de origem) para outro banco (do processador de destino). Este mecanismo é similar a uma programação de um DMA fornecendo o endereço de origem, a quantidade de dados a serem copiados e o endereço de destino. Devido a este fato, este modelo de memória foi denominado de nDMA (NoC DMA).

Estes modelos foram implementados na plataforma virtual SIMPLE (*SIMPLE Multiprocessor Platform Environment*) (BARCELOS, 2008) que simula um ambiente com múltiplos processadores FemtoJava que implementam a máquina virtual Java em hardware. É importante salientar que todos os modelos de memória implementados dizem respeito à memória de dados. Esta decisão foi tomada tendo em vista que o acesso aos dados possui uma natureza não-linear, diferentemente do acesso a instruções. Esta característica torna o estudo dos modelos de memória com relação aos dados mais relevante do que no caso de memória de instruções. Todo o ambiente é apresentado com maiores detalhes no Capítulo 3.

Foram realizados experimentos com a plataforma SIMPLE e estas organizações de memória com aplicações representativas objetivando aspectos de performance, consumo de energia e tráfego na rede. Os resultados mostram que a memória distribuída

apresenta melhores resultados em situações em que a carga de dados comunicados no sistema é baixa. Os experimentos sugerem que conforme a carga de trabalho aumenta, este modelo de memória apresenta pior performance e maior consumo de energia além de gerar um maior tráfego. Este tipo de dependência não apresenta tanto impacto nos experimentos com outros modelos de memória.

As memórias compartilhada e compartilhada distribuída apresentaram resultados similares com alguma vantagem para a memória compartilhada distribuída no que diz respeito à performance e energia. Entretanto para menores cargas de trabalho, a memória nDMA apresenta melhores resultados do que as outras memórias compartilhadas. No que diz respeito ao tráfego na rede, a memória nDMA apresentou redução considerável. Este resultado era esperado devido ao modelo de comunicação que minimiza o envio de dados pela rede.

Também foram realizados experimentos avaliando a tolerância destes modelos de memória diante de situações de alta latência da rede. De modo a emular esta alta latência, foi incorporado à plataforma SIMPLE um gerador de tráfego. Este gerador tem o intuito de gerar um tráfego sintético para emular uma situação de alta latência na comunicação.

Resultados experimentais mostraram que o modelo de memória distribuída, apesar de apresentar resultados melhores para cara aplicação, aparenta perda de performance em situações de alta latência conforme a carga de comunicação de uma aplicação é baixa. Em uma situação oposta a esta, onde a carga de comunicação de uma aplicação é alta, o modelo nDMA apresentou melhores resultados.

Por fim foram realizados experimentos com o objetivo de mensurar o desempenho da migração de tarefas em cada um dos modelo de memória.

Devido à natureza dos modelos de memória, cada um deles tem sua própria maneira de realizar a migração de uma tarefa. Dadas estas diferenças, o modelo de memória distribuída apresentou piores resultados devido ao fato de que, neste modelo, existe a necessidade do envio dos dados da tarefa o que não se faz necessário nos outros modelos. Outro fator decisivo para este resultado diz respeito ao tamanho do código em um modelo de memória distribuída que, devido ao fato de realizar a comunicação de maneira explicita, resulta em um código maior do que nos modelos de memória compartilhada. Isto ocorre devido ao fato de que o modelo de memória distribuída exige o uso de funções de comunicação explicitas, as quais fazem parte do código a ser migrado. Esta característica também está presente no modelo nDMA, uma vez que também é preciso uma comunicação explicita, apesar de mais simples já que não há tráfego de dados e sim de controle. Por outro lado, modelos de memória compartilhada utilizam primitivas simples que têm por objetivo garantir a exclusão mútua.

1.3 Organização do texto

Uma vez estabelecidos os conceitos primários envolvidos neste trabalho, serão apresentados, no Capítulo 2, trabalhos relacionados a este nos vários níveis de estudos e exploração de espaço de projeto de hierarquia de memória em sistemas embarcados multiprocessados.

O Capítulo 3 apresenta de maneira mais abrangente a plataforma virtual SIMPLE. Neste mesmo capítulo são detalhadas as implementações dos modelos de memória mencionados anteriormente e que são alvo deste estudo.

No Capítulo 4 encontra-se o estudo que é objetivo deste trabalho. São analisados resultados de experimentos com organizações de memória largamente difundidas (memória distribuída, memória compartilhada distribuída) bem como a organização de memória nDMA proposta nesse trabalho.

Finalmente, são apresentadas, no último capítulo, as conclusões e algumas motivações para trabalhos futuros.

2 ESTADO DA ARTE

Esta seção apresenta trabalhos relacionados a esta dissertação. Inicialmente são apresentados conceitos sobre os quais são fundamentados esta dissertação e os trabalhos apresentados nesta seção. Em seguida, são detalhados e discutidos trabalhos que exploram o espaço de projeto de hierarquias de memórias em sistemas embarcados e também, de maneira mais específica, ambientes com MPSoCs baseados em NoC.

2.1 Hierarquia de memória

Em um sistema computacional, a hierarquia de memória tem um papel importante em termos de desempenho. Considerando sistemas multiprocessados ela se torna ainda mais importante devido à grande quantidade de alternativas possíveis para seu projeto.

A hierarquia de memória surgiu como uma tentativa de diminuir o tempo médio de acesso a dados ou instruções residentes na memória do sistema. A necessidade de diminuir este tempo de acesso vem da grande diferença de velocidade existente entre processadores e dispositivos de memória (PATERSON, 2003). A Figura 2.1 mostra um gráfico que representa esta diferença de velocidade com o passar dos anos.

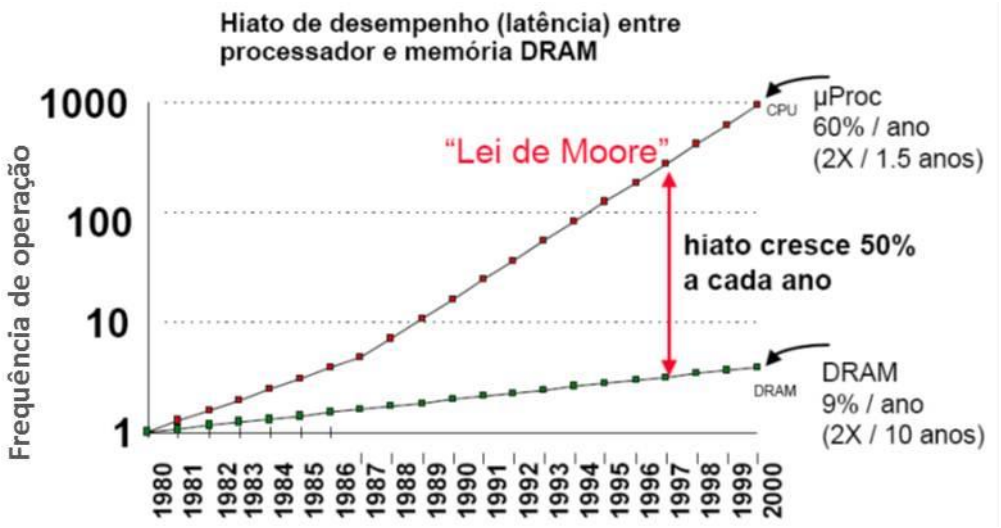


Figura 2.1 – Diferença de desempenho entre processadores e memória com o tempo.

Uma hierarquia de memória é composta por um conjunto de dispositivos de armazenamento com características de armazenamento e tempo de acesso distintos.

Estes dispositivos vão desde uma memória cache, por exemplo, com uma pequena capacidade de armazenamento e baixo tempo de acesso até uma memória principal de grande capacidade de armazenamento, mas com um alto custo temporal ao acessá-la. Os componentes de memória talvez sejam aqueles que apresentem a maior variedade de tipos, tecnologias, organizações, desempenhos e custos em um sistema computacional. Esta variedade abre espaço para uma grande exploração de espaço de projeto que é objeto de alguns estudos que serão apresentados no Capítulo 2.

No que diz respeito à organização física da memória as soluções basicamente dividem-se em memória compartilhada e memória distribuída. Uma grande consequência desta divisão é o modelo de programação adotado. Quando se utiliza memória compartilhada, em geral, utiliza-se um modelo de programação no qual a comunicação ocorre por meio de variáveis compartilhadas. Por outro lado, quando se utiliza memória distribuída é comum se utilizar comunicação por troca de mensagens.

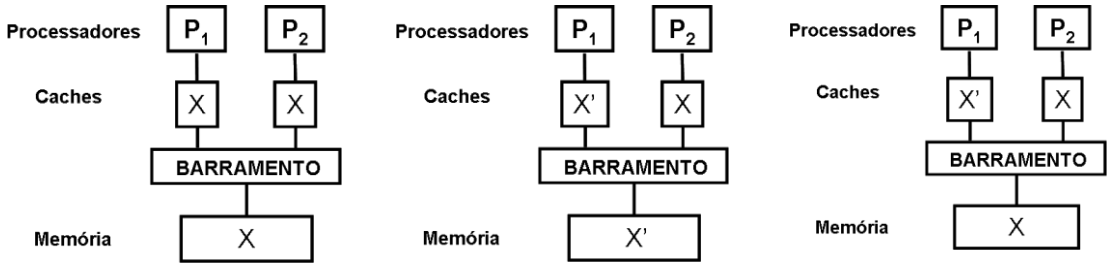
Outro aspecto ortogonal à organização física da memória é o espaço de endereçamento que pode vir a ser único (de tal forma que todos os processadores podem acessar a mesma faixa de endereços) ou distribuído (onde cada processador acessa um conjunto de endereços exclusivos). O uso de uma organização de memória fisicamente distribuída pode ser utilizado com um único espaço de endereçamento. Desta forma, diferentes processadores acessam diferentes módulos de memória dependendo do endereço requisitado. Da mesma maneira, uma única memória física no sistema pode ter diferentes espaços de endereçamento. Logo, diferentes processadores acessam o mesmo módulo de memória apesar de não poderem acessar o mesmo espaço de endereçamento. Esta última solução acarreta em uma operação a mais a ser realizada pelo controlador da memória que é identificar o processador que requisita o acesso à memória. Dependo do processador, o acesso será feito a um endereço diferente dentro do módulo.

Considerando organizações de memória compartilhada tem-se ainda o uso de soluções para diminuir o tempo médio de acesso à memória. A solução mais adotada é o uso de caches. Caches apresentam vários outros aspectos relevantes para o desempenho do sistema, tais como tamanho, associatividade, política de substituição e política de escrita. Outra solução para diminuir o tempo médio de acesso à memória é o uso de *scratchpads*. Esta alternativa consiste no uso de uma pequena memória próxima ao processador, preenchida (em tempo de projeto) com dados que serão frequentemente utilizados pelo processador. *Scratchpads* se diferenciam de caches no que diz respeito ao fato de que, na primeira, os dados são previamente alocados e também por não apresentar uma lógica de controle tão complexa.

2.2 Coerência de cache

A incoerência da cache pode ser causada por vários fatores. Os principais são o uso de dados compartilhados e a migração de processos (HWANG, 1993). É importante esclarecer que a política de escrita da cache não melhora ou piora o problema da coerência de cache. Se a política de escrita for *write-through* (significando que ao realizar uma escrita o dado alterado será imediatamente mandado para a memória para mantê-la atualizada) ou se for *write-back* (que é o caso do bloco alterado somente ser atualizado na memória quando este precisar ser retirado da cache), em ambos os casos, poderá ocorrer incoerência de cache uma vez que ela pode ocorrer não só entre a cache e a memória como também entre as caches dos diferentes processadores.

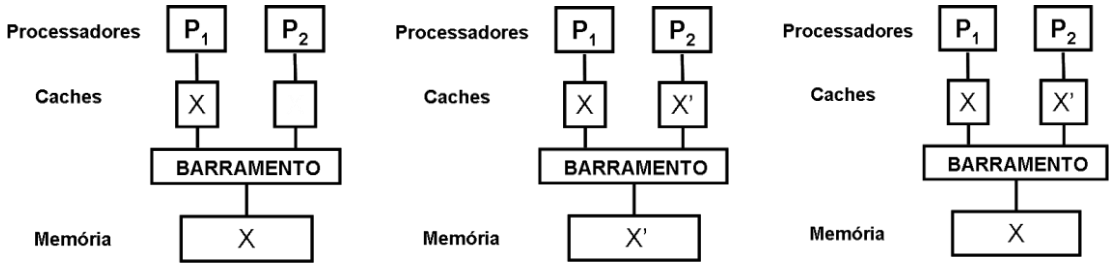
No caso do uso de dados compartilhados, o problema da incoerência de dados pode ocorrer da seguinte maneira: considerando dois processadores cada um com sua própria cache e ambos compartilhando a memória principal, suponha-se que existe uma estrutura de dados X qualquer que é referenciada por ambos os processadores e ambas as cópias estão consistentes (Figura 2.2a). Em caso de uma escrita por um dos processadores, se a cache utilizar a política de *write-through*, a memória terá a atualização do bloco (modificando a estrutura para X', por exemplo), entretanto a outra cache que contém este dado não terá sido atualizada (Figura 2.2b) Caso a cache utilize uma política de *write-back* tanto a cache do outro processador quanto a própria memória ficarão desatualizadas (apesar de que esta última será atualizada, ocasionalmente, quando a cache substituir este bloco) (Figura 2.2c).



a) b) c)

Figura 2.2 – Incoerência devido ao uso de dados compartilhados.

Quando lidando com migração de processos, a coerência de cache também pode ser prejudicada. Utilizando um exemplo similar ao dado acima, suponham-se dois processadores e suas caches privadas e uma estrutura de dados compartilhada X (Figura 2.3a). Caso um processo alocado ao processador P_1 faça acesso a X e este processo faça uma migração para um outro processador e ocorra uma nova atualização do dado, o bloco anteriormente modificado permanecerá na cache do processador P_1. Caso a política seja de *write-through*, na primeira situação (antes da migração) o dado será copiado da memória para a cache. Na segunda situação (após a migração) o dado será modificado na memória principal, entretanto o dado X continuará na cache do processador P_1. Assim caso o processador P_1 execute um processo que faça acesso ao dado X, ocorrerá a inconsistência (Figura 2.3b). Caso a política de escrita seja *write-back* o dado será ocasionalmente modificado na memória principal assim que uma das caches substituir o bloco. Contudo, enquanto isto não acontecer, ambas as caches estarão com o mesmo dado de maneira inconsistente (Figura 2.3c).



a) b) c)

Figura 2.3 - Incoerência devido à migração de tarefas.

De maneira geral pode-se dizer que existem duas categorias de protocolos para manter a coerência de cache em sistemas multiprocessados: *snoop* e diretório. Existem outros protocolos, entretanto praticamente todos são baseados nestas duas categorias (HWANG, 1993).

Uma das grandes diferenças entre o protocolo *snoop* e o protocolo de diretório é como a informação sobre os estados dos blocos está difundida no sistema. Na solução *snoop*, cada cache é responsável por manter o estado do bloco que possui. Desta forma é acrescentada à cache a complexidade de um controlador *snoop* para monitorar o tráfego. Esta necessidade de monitorar o tráfego faz com que esta solução seja impossível quando se utiliza uma rede-em-chip como mecanismo de interconexão. Esta impossibilidade vem do fato de que a comunicação entre dois nós na rede, um processador e a memória, por exemplo, pode nunca passar pelo roteador no qual se encontra um outro processador. Na solução de diretório os estados dos blocos são mantidos em um módulo centralizador (geralmente localizado na memória). Nesta segunda classe de soluções o diretório representa um controlador de coerência centralizado. Cada solicitação de acesso à memória passa por este módulo e com isso mantém uma tabela que relaciona que blocos da memória estão presentes em que caches e qual o estado deles: limpo, indicando que não houve escritas neste bloco, ou sujo, sinalizando que o bloco foi modificado.

2.3 Networks-on-Chip

No contexto de sistemas multiprocessados, a comunicação é um componente crítico devido ao fato de que o grau de paralelismo das aplicações em execução é, geralmente, dependente da capacidade de escalabilidade e baixa latência do mecanismo físico de comunicação. No que diz respeito a MPSoCs uma das primeiras soluções apresentadas é a mais simples no aspecto relacionado à implementação e também de mais baixo custo: o uso de um barramento para efetuar a comunicação dos dispositivos. O problema do barramento se encontra na baixa escalabilidade disponível, ou seja, na crescente degradação na comunicação do sistema à medida que o número de dispositivos aumenta.

Uma solução mais atual e que é objeto de pesquisas mais recentes é o uso de uma NoC (BENINI, 2002) como elemento de interconexão. Esta solução apresenta maior flexibilidade devido ao fato de poder ser configurável no que diz respeito às topologias de conexão entre os componentes e também por apresentar uma maior escalabilidade (LEE, 2007). Uma NoC utiliza características existentes nas redes de computadores para atender aos propósitos dos sistemas paralelos. Por esta razão, o principal componente de uma NoC é o roteador que contém características de topologia, arbitragem, roteamento, chaveamento, controle de fluxo e memorização. Muitos estudos foram feitos para comparar topologias de NoCs (BARTIC, 2005; BRIÈRE, 2005; CHING, 2004), com o objetivo de caracterizar suas vantagens e desvantagens. Algumas topologias mais conhecidas são Grelha 2D (também conhecida como mesh), Torus 2D, Cubo 3D, Cubo 4D e Hipercubo.

2.4 Modelos de memória

Um dos principais aspectos no que diz respeito à memória em sistemas multiprocessados é a escolha do modelo a ser utilizado. Por modelo, se entende como a

localização física da memória com relação aos elementos de processamento do sistema bem como a disponibilidade de acesso em relação a eles. Tipicamente, esta escolha se resume a duas categorias: memória compartilhada e memória distribuída.

No modelo de memória compartilhada, os processadores disputam o acesso a um dispositivo de memória único no sistema. Neste modelo fica claro que a disputa pelo acesso à memória causa um grande gargalo no sistema. Este problema pode ser amenizado utilizando bancos de memória entrelaçados, ou seja, diferentes conjuntos de endereços da memória que podem ser acessados independentemente. Esta solução se baseia na probabilidade de que mais de um processador não deseje acessar o mesmo endereço em um mesmo momento (HWANG, 1993).

Alternativamente, com o modelo de memória compartilhada, cada elemento de processamento tem uma memória privada. Desta maneira, deixa de existir a disputa pelo acesso a memória por mais de um processador.

É importante perceber que o modelo de memória adotado influencia diretamente no modelo de programação do sistema. Intuitivamente, em um modelo de memória distribuída utiliza-se um mecanismo de troca de mensagens (através de primitivas de envio e recebimento) para realizar a comunicação entre os componentes do sistema. Por outro lado, no modelo de memória compartilhada toma-se proveito do acesso à mesma memória por parte dos elementos de processamento para realizar a comunicação através de variáveis compartilhadas (FORSELL, 2002).

Percebe-se que cada modelo tem suas vantagens e desvantagens. O modelo de memória distribuída tem a vantagem da escalabilidade do sistema enquanto que o modelo de memória compartilhada é de fácil programação já que não envolve o uso de primitivas de envio e recebimento de mensagens. Ao invés disso, neste modelo, em geral, são utilizadas simples marcações (*mutexes*) para garantir exclusão mútua. A Figura 2.4 mostra a relação de escalabilidade e programabilidade dos dois modelos.

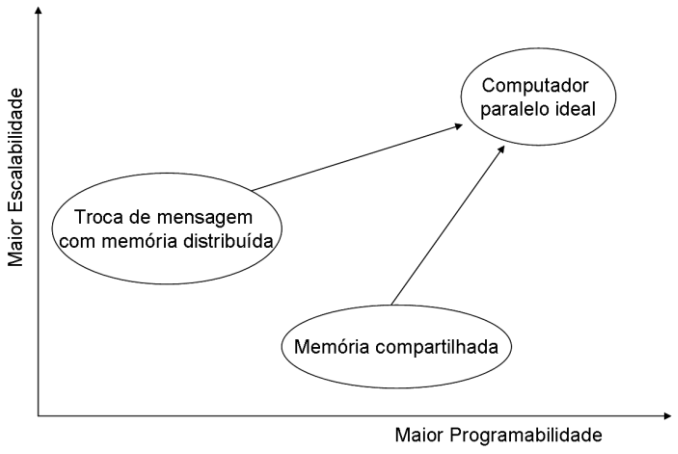


Figura 2.4 – Programabilidade versus escalabilidade.

No que diz respeito ao espaço de endereçamento, (ou seja, a faixa de endereços que cada elemento de processamento é capaz de acessar) fica implícito que no modelo de memória compartilhada o espaço de endereçamento seja único e no modelo de memória distribuída o espaço de endereçamento seja também distribuído. Entretanto, o uso de um

espaço de endereçamento único em um modelo de memória distribuída dá origem a um novo modelo de memória híbrido conhecido como memória compartilhada distribuída. Neste modelo, memórias fisicamente distribuídas no sistema formam um mesmo espaço de endereçamento.

O desempenho do modelo de memória compartilhada distribuída é influenciado diretamente pelo mecanismo de comunicação do sistema. Dependendo da solução adotada, o acesso a diferentes módulos de memória do sistema pode ser realizado em paralelo (no caso do uso de NoCs), o que traz grandes benefícios em termos de desempenho diminuindo o tempo gasto com acesso à memória do sistema como um todo.

2.5 Performance de hierarquia de memória em sistemas embarcados

Devido à grande variedade de componentes de uma hierarquia de memória bem como a grande variedade de características possíveis de cada um destes componentes faz-se necessário uma exploração do espaço de projeto a fim de obter uma solução que atenda os requisitos de projeto tais como desempenho, área e consumo de energia.

Alguns artigos abordam a exploração automática deste espaço de projeto através de ferramentas que se baseiam no *profiling* da aplicação, ou seja, a partir do código da aplicação conclui-se características comportamentais da mesma. Neste caso específico, este comportamento diz respeito ao acesso à memória.

Hiser (2007) apresenta um processo para exploração de espaço de projeto de hierarquia de memória de maneira rápida e precisa chamado MCDSE (*Memory Configuration Design Space Exploration*). Este processo inclui um passo de exploração de hierarquia de memória durante o projeto do hardware como está destacado na Figura 2.5. Assume-se que neste projeto do hardware são conhecidos os códigos da aplicação alvo, as restrições do hardware bem como os objetivos de performance desejados.

O processo consiste nos seguintes passos. Primeiro, o MCDSE coleta o comportamento de acesso a uma variável no programa (a partir do *profile*). A partir daí, é mantida uma informação sobre o uso da variável durante a execução, armazenando informações sobre a quantidade de acessos à variável entre um tempo e outro (ciclos). Esta informação é chamada de *dynamic live range*. Com esta informação e o resultado do *profiling* é possível realizar a associação das variáveis a diferentes partições da memória. A partir deste particionamento é possível estimar padrões de acesso à cache de dados. Com estas estimações e os objetivos de performance desejados pode-se obter um conjunto das melhores soluções de hierarquias de memória. Os passos deste processo podem ser vistos na Figura 2.6.

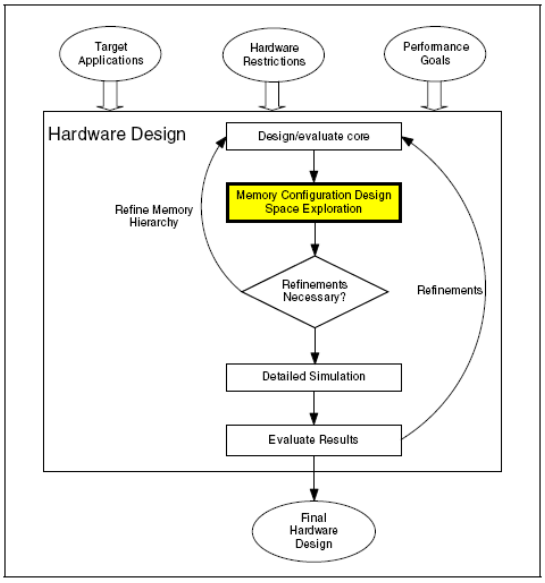


Figura 2.5: Inclusão do MCDSE no projeto de hardware. Retirado de (HISER, 2007).

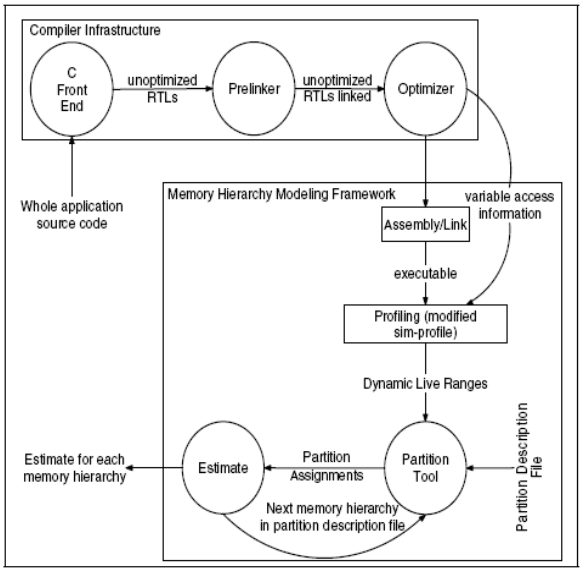


Figura 2.6: Processo MCDSE. Retirado de (HISER, 2007).

Os resultados apresentados mostram que o processo de estimação é rápido (duas ordens de grandeza mais rápido do que uma simulação completa) bem como preciso, uma vez que 70% de todas as estimativas tiveram um erro de até 1% enquanto que 99% de todas as estimativas obtiveram um erro máximo de 10%.

De maneira similar, Abraham (1999) propõe uma abordagem hierárquica para avaliação dos componentes de um sistema formado por um processador VLIW (*Very Large Instruction Word*), caches de dados e instruções e uma cache unificada de segundo nível. Nesta abordagem, o uso de diferentes tipos de processadores VLIW (com diferentes unidades de execução, número de registradores, etc) é avaliado de maneira separada da hierarquia de memória. Dessa maneira, a partir da aplicação alvo são feitas estimativas de custo-performance sobre as várias opções de organização de processadores bem como da hierarquia de memória a partir dos traces da aplicação gerados a partir de um processador referência. Os resultados destas estimativas são combinados para formar a estimativa de performance do sistema como um todo.

Entretanto, os resultados mostram que as estimativas com relação ao número de *misses* da cache unificada do sistema são otimistas.

Shiue (2001) apresenta uma metodologia para exploração no projeto de memória para sistemas embarcados com baixo consumo de energia. O primeiro passo desta metodologia é realizar transformações de *loop* no código. A partir dos resultados destas transformações são utilizados algoritmos para busca do melhor conjunto de soluções de memória. Estas soluções levam em consideração três aspectos: tamanho da cache, tamanho da linha da cache e consumo de energia. Sabidamente para um tamanho específico de linha, conforme o tamanho da cache aumenta, a taxa de *misses* diminui. Também é conhecido que para um tamanho específico de linha, conforme o tamanho da cache de dados aumenta, o número de ciclos de execução do sistema diminui. Entretanto, os experimentos relatados neste trabalho mostram que para um tamanho específico de linha, conforme o tamanho da cache aumenta, o consumo de energia é reduzido em um primeiro momento e em seguida aumenta. Isto significa que existe um tamanho de cache para o qual o consumo de energia é mínimo.

Em Milenkovic (2003) não é feita uma exploração automática do espaço de projeto de caches, entretanto demonstra-se que, para caches pequenas de mapeamento direto, o uso de caches separadas de instruções e dados apresenta melhores resultados do que o uso de uma cache unificada. Por outro lado, ao utilizar caches de mapeamento conjunto-associativo o uso de uma cache unificada obtém melhores resultados do que o uso de caches separadas. Todos estes resultados são baseados exclusivamente no número de *misses*.

Apesar de serem trabalhos interessantes já que visam a obtenção de uma hierarquia de memória eficiente no contexto de sistemas embarcados, nenhum destes trabalhos prevê um ambiente multiprocessado. Este tipo de ambiente acarretaria em um espaço de projeto ainda maior a ser explorado.

2.6 Organizações de memória em MPSoCs baseados em NoC

Em Marescaux (2007) é apresentada uma comparação entre o uso de caches e *scratchpads* em um MPSoC baseado em NoC que utiliza um modelo de memória compartilhada distribuída.

Neste trabalho é utilizada uma plataforma composta por 6 nós DSP e uma memória L2 compartilhada. São apresentados dois tipos de NoC com suporte a tipos de QoS diferentes: *Best Effort* (BE) e *Guaranteed Troughput* (GT). Além disso, também é configurável o uso de caches para a comunicação entre os processos ou o uso de *scratchpads* gerenciadas por dispositivo similar a um DMA controlado por software chamado CA (*Communication Assist*). A Figura 2.7 mostra a plataforma utilizada neste trabalho. Cada nó da rede tem um elemento responsável por encapsular (e desencapsular) transações de memória em pacotes que irão trafegar pela rede (*Bridge*).

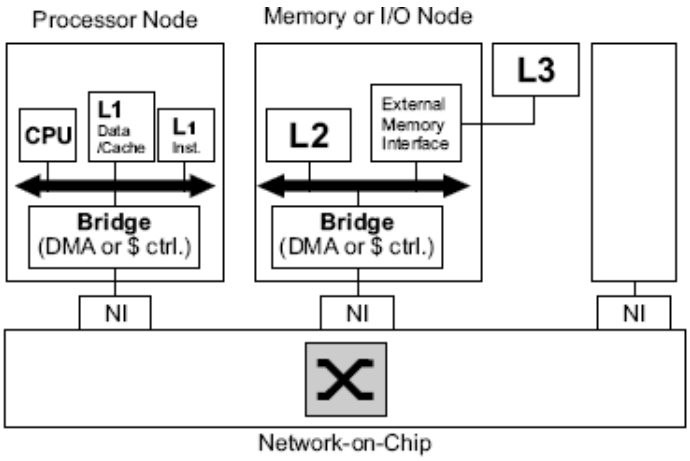


Figura 2.7: Plataforma MPSoC composta por nós de processamento e nós de memória. Retirado de (MARESCAUX, 2007).

Na versão da plataforma com *scratchpads*, o gerenciamento utilizando CAs funciona de modo que as transferências de blocos de dados entre as diferentes memórias no sistema são controladas pelos CAs independentemente do processador. Os CAs são controlados por um código incluído na aplicação rodando no processador local.

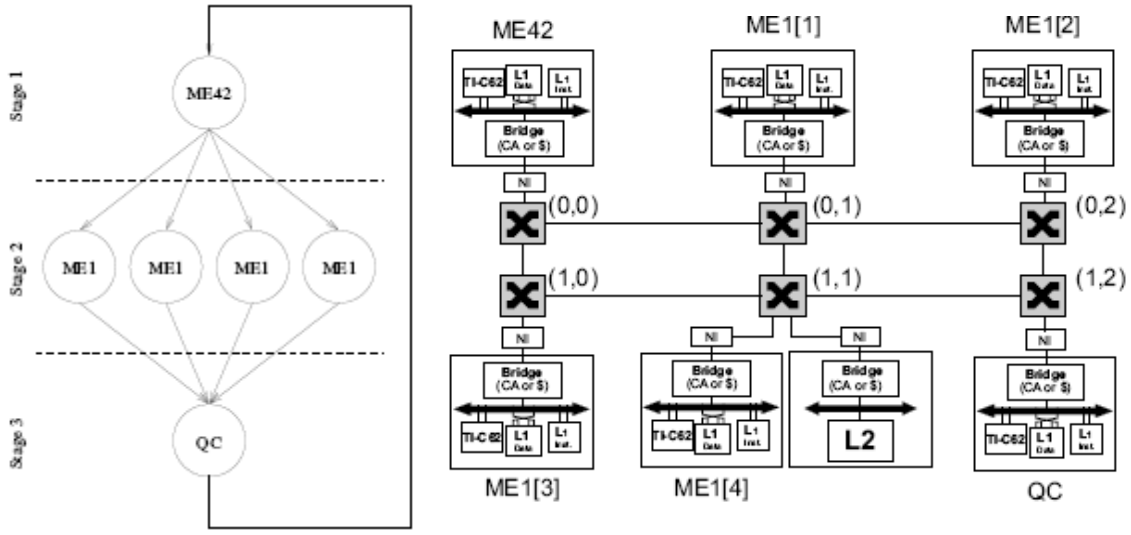
Na versão com o uso de caches, os *Bridges* se tornam controladores de cache. Nesta plataforma, as caches utilizam política de escrita *write-back* e política de substituição LRU (*Least Recently Used*). O tamanho da linha da cache é configurado para 16 bytes e a associatividade bem como o tamanho do conjunto são configuráveis. Não há o fornecimento de nenhum mecanismo de consistência de dados a não ser o determinismo de entrega de pacotes em ordem da própria rede. Também não há um mecanismo de coerência de cache em hardware. Os *Bridges* dão suporte a soluções de coerência de cache controladas por software uma vez que mantêm controle de estado dos blocos e permitem operações explícitas de invalidação e *flushes* de endereços da cache requisitados por um processador.

Neste artigo é apresentado também o mapeamento de uma aplicação de compressão inter-frame de vídeo conhecida como QSDPCM (*Quad-Tree Structured Differential Pulse Code Modulation*). A aplicação é dividida em 3 tipos de tarefas: estimação de movimento *full resolution* (ME1), *half resolution* e *quarter resolution* (ME42) e quantização *Quad-Tree* (QC). A tarefa ME1 por ser a mais complexa foi paralelizada entre 4 DSPs (dividindo o processamento de cada quadro em 4 áreas sem intersecção) enquanto que as tarefas ME42 e QC foram mapeadas para um DSP cada uma. A aplicação funciona como um pipeline como mostra a Figura 2.8a. Um exemplo de mapeamento na plataforma é mostrado na Figura 2.8b.

Para mapear os dados na versão com *scratchpads* foi utilizada uma versão para multiprocessadores da ferramenta MHLA (*Memory hierarchy layer assignment*). O resultado da análise dos acessos à memória previstos é entrada da ferramenta MHLA que decide a associação dos dados nas diferentes camadas de memória bem como o escalonamento de transferência de blocos (*prefetching*). Este escalonamento é gerado na forma de um código C.

A versão mapeada para caches da aplicação QSDPCM tem 3 principais diferenças com relação à versão para *scratchpads*. Primeiramente, não há explícito *prefetching* dos dados. Por conseqüência, a versão com cache ocasionalmente acarretará na necessidade

de parar o processador para efetuar a busca dos dados (*stall*). A segunda diferença é que apesar dos processadores serem mapeados para manipularem dados distintos dos quadros (sem sobreposição) ainda é possível, que quando uma cache leia sua parte do quadro, uma parte do quadro que diz respeito a outro processador compartilhe o mesmo bloco na memória. Isto ocorre porque cada processador lê um *array* de 300 bytes que não é múltiplo de 16 (tamanho da linha da cache). Para evitar este tipo de problema, os dados foram armazenados de modo que a cada parte do quadro destinada a um processador seja adicionado de mais 4 bytes (já que 304 é múltiplo de 16). A terceira diferença é que devido ao fato de não se adotar uma solução de coerência de cache em hardware, o código necessita conter operações explícitas de flush da cache ou invalidação de linhas. Dessa maneira, todo dado compartilhado que é lido precisa de uma operação de invalidação. Esta é uma conseqüência que prejudica sensivelmente o desempenho da cache.



a) b)

Figura 2.8: Paralelização do algoritmo QSDPCM. Retirado de (MARESCAUX, 2007).

Uma análise dos resultados de simulações com várias características de caches (graus de associatividade diferentes e tamanhos de cache diferentes) apontaram o uso da cache de associatividade 2 e tamanho 4K como a de melhor desempenho. A Figura 2.9 apresenta os resultados de número de ciclos de execução comparando as versões de *scratchpad* em uma NoC com BE e versões de caches em NoCs GT e BE. O tempo de execução está dividido em 5 categorias:

- **Preâmbulo (*Preamble*)**: corresponde ao tempo gasto para a primeira sincronização de *threads* esperando que o pipeline da aplicação encha.
- **Sincronização de threads (*Thread Sync*):** é o numero de ciclos que um processador espera para que o outro estágio do pipeline da aplicação complete seu processamento.
- **Processamento (*Processing*)**: tempo gasto efetivamente com a execução do código.

- **Conflitos de acesso à L1 (*L1 bank conflicts*):** ocorre quando ambas as unidades de *load* e *store* do DSP tentam acessar a L1.
- **Acesso à L2 (*L2 stalls*):** corresponde ao tempo gasto pelos *misses* na versão com caches e pelo tempo de transferência de blocos no caso da versão com *scratchpads*.

Os resultados mostram que o número de *stalls* por parte das caches foi o diferencial nos experimentos. Não há como saber quantos destes *misses* nas caches foram causados por invalidações desnecessárias devido à adoção de uma solução de coerência de cache em software que realiza invalidação dos dados a cada leitura de dados compartilhados. A diferença do número de ciclos para a sincronização das *threads* também é resultado do número elevado de *stalls*.

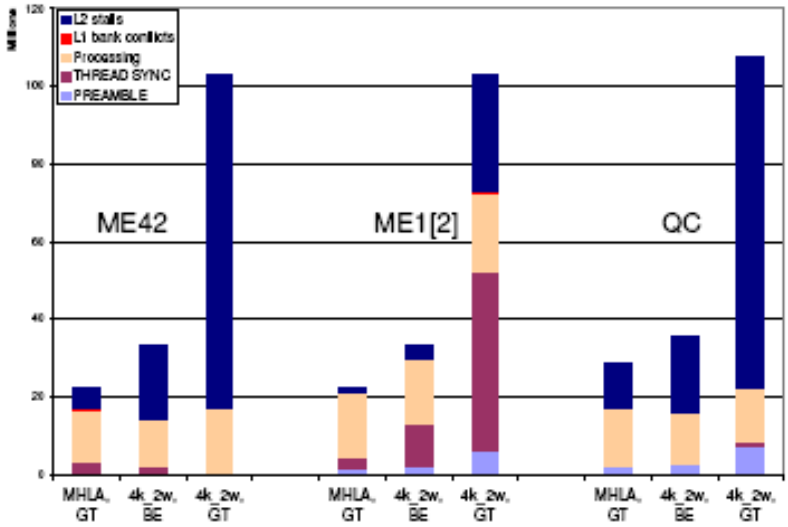


Figura 2.9: Resultado comparativo entre Scratchpad e Caches. Retirado de (MARESCAUX, 2007).

Monchiero (2006) apresenta a exploração do uso de memória compartilhada distribuída em uma plataforma MPSoC com uso de NoC. Neste artigo é feita uma análise em termos de consumo de energia e desempenho de memórias fisicamente distribuídas com um mesmo espaço de endereçamento.

O trabalho apresenta uma plataforma que utiliza a NoC PIRATE cujo roteamento é baseado em *crossbar*, chaveamento *wormhole*, arbitragem distribuída. O sistema de memória é baseado no paradigma NUMA (*Non Uniform Memory Access*) devido ao uso de memórias fisicamente distribuídas e, por consequência, o tempo de acesso a cada uma delas por parte de um processador pode levar tempos (*hops*) diferentes. A alocação e desalocação de dados na memória são gerenciadas de maneira dinâmica por uma unidade de gerenciamento de memória em hardware (HwMMU).

A Figura 2.10 apresenta a plataforma utilizada neste artigo, basicamente composta por diversos elementos de processamento, módulos de memória e a HwMMU. Cada elemento de processamento é composto por um *core*, uma cache de dados e uma cache de instruções L1 privadas, uma TLB para dados compartilhados (*Shared Data* TLB), uma TLB para dados privados e instruções e um co-processador para realizar a comunicação com a interface de rede (*Network Interface*). Uma cache L2 unificada para dados privados de cada processador e instruções é utilizada neste ambiente.

No que diz respeito ao subsistema de memória, esta plataforma apresenta espaço de memória privada composto por caches L1 privadas, cache L2 e a memória principal que se encontra fora do chip (note-se que o controlador de acesso à memória é um componente interno ligado à NoC). Cada processador tem o seu espaço de endereçamento dentro destes elementos de memória. Por outro lado, também existe um conjunto de memórias fisicamente distribuídas na rede que formam um único espaço de endereçamento e que pode ser acessado por qualquer elemento de processamento. Este espaço de memória compartilhado é utilizado para sincronização e troca de dados entre os processadores. É importante esclarecer que neste ambiente não há a necessidade de manutenção da coerência, pois no que diz respeito ao espaço de endereçamento privado (L1 privadas, L2 unificada e memória principal) não há o problema, enquanto que os dados do espaço de endereçamento compartilhado não são copiados para as caches privadas.

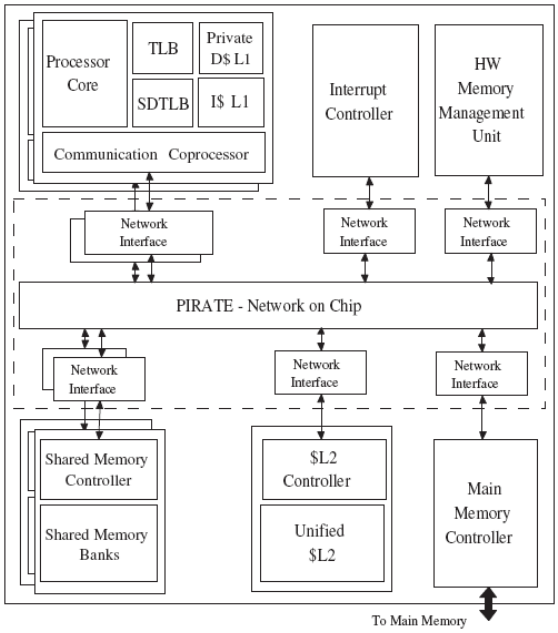


Figura 2.10: Plataforma MPSoC com a NoC PIRATE. Retirado de (MONCHIERO, 2006).

As funções exercidas por uma unidade de gerenciamento de memória foram implementadas em hardware para diminuir o *overhead* decorrente da adoção de um sistema operacional para realizar esta tarefa. Esta solução acarretaria em um maior gasto de energia e uma pior performance. A solução em hardware torna o gerenciamento de memória rápido e determinístico. Nesta implementação, a HwMMU mantém uma tabela de páginas compartilhadas alocadas (*Shared Page Table*) bem como a relação entre o endereço virtual e o endereço físico de cada página compartilhada. A SD-TLB (*Shared Translation Look-Aside Buffer)* presente em cada elemento de processamento funciona como uma cache desta tabela, de tal forma que, quando um *miss* ocorre, é feita uma leitura dos dados da tabela de páginas compartilhadas da HwMMU.

Como dito, esta unidade de gerenciamento de memória dá suporte a funções de alocação e desalocação de memória utilizando duas estruturas básicas:

- *MMU_MALLOC(MEM, size)*: realiza a alocação de "*size*" bytes e "*MEM*" é o numero do módulo de memória onde o dado precisa ser alocado;

- *MMU_FREE(address)*: libera o espaço antes alocado no endereço indicado por "*address*".

Além destas funções, a HwMMU também dá suporte a outras duas operações de manipulação de dados:

- *MMU_COPY(MEM, address, number)*: inicialmente aloca um espaço de "*number*" páginas a partir do endereço "*address*" na memória de destino apontada por "*MEM*", então a cópia é realizada e ao fim da operação o endereço da nova página é enviado de volta ao elemento de processamento que originou a cópia;
- *MMU_MOVE(MEM, address, number)*: inicialmente aloca um espaço de "*number*" páginas a partir do endereço "*address*" na memória de destino apontada por "*MEM*". Uma vez realizada a alocação, a HwMMU envia uma solicitação de invalidação para todas as SD-TLBs que tenham referência ao endereço cujo conteúdo será movido. A partir de então a movimentação dos dados pode ser realizada.

Para realizar os experimentos foram utilizadas aplicações dos benchmarks SPLASH-2 e PARMACS. Estas aplicações têm características diferentes com relação ao tempo gasto em computação, acesso à memória e sincronização, como é mostrado na Figura 2.11.

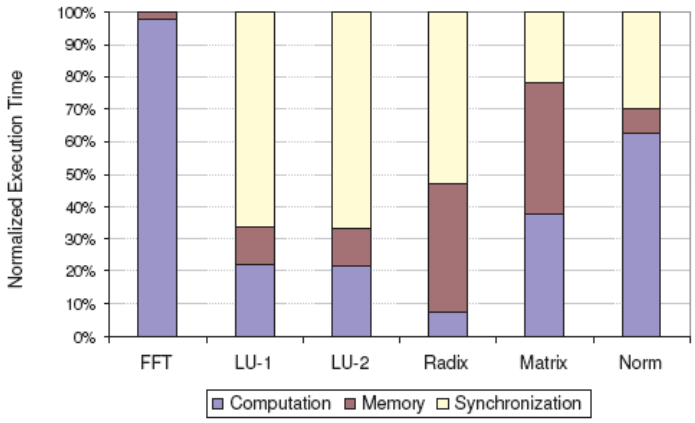
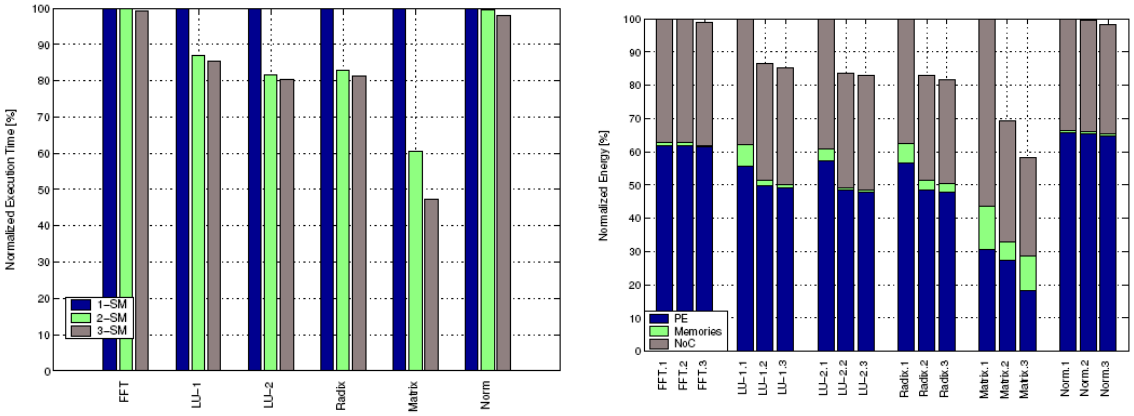


Figura 2.11: Benchmark de aplicações utilizado. Retirado de (MONCHIERO, 2006).

Primeiramente são apresentados dados de latência de acesso à memória compartilhada distribuída. Os resultados mostram que para um único módulo de memória a latência tem os valores mais altos. Este valor decresce conforme o número de módulos de memória no sistema aumenta. Este crescimento ocorre até a inclusão de 5 módulos de memória onde a latência aumenta com relação ao uso de 4 módulos de memória. Os autores afirmam que isto se deve ao fato de que o aumento do número de módulos de memória aumenta o tamanho da rede causando uma maior latência na comunicação.

A Figura 2.12a apresenta os resultados de tempo de execução normalizados (em relação ao uso de um único módulo de memória) para as 6 aplicações utilizando de 1 a 3 módulos de memória. Conforme o número de módulos de memória aumenta, melhor é o desempenho. Segundo o autor, as aplicações *LU-1*, *LU-2*, *Radix* e *Matrix* obtiveram um melhor desempenho devido a sua maior característica de acesso à memória ou sincronização (que é feita por meio da memória compartilhada).

Na Figura 2.12b são apresentados resultados de consumo de energia. A economia no consumo de energia é justificada principalmente por dois fatores: primeiro, devido à redução do tempo de execução e segundo devido ao custo energético por acesso à memória quando a mesma é dividida em mais de 1 módulo.



a) b)

Figura 2.12: Resultados de desempenho e energia. Retirado de (MONCHIERO, 2006).

Os trabalhos apresentados nesta seção mostram que o uso de memória compartilhada distribuída vem sendo largamente adotado em ambientes multiprocessados baseados em NoC. Apesar de apresentarem bons resultados estes trabalhos adotam soluções alternativas para evitar o uso de uma solução de coerência de cache em hardware. É possível que o não uso de uma solução em hardware possa ter levado a um *overhead* desnecessário.

A seção seguinte apresenta trabalhos sobre soluções de coerência de cache. A maioria adota soluções em ambientes multiprocessados baseados em NoC.

2.7 Soluções de coerência de cache em sistemas multiprocessados

O uso de caches em sistemas multiprocessados pode levar ao problema da coerência de cache como explicado na Seção 2.2. Esta seção apresenta alguns trabalhos que propõem soluções de coerência de cache ou comparações entre estas soluções.

O trabalho apresentado em Zahran (2003) tenta avaliar a melhor solução de hierarquia de caches para sistemas multiprocessados. Além disso, são apresentadas 5 soluções de protocolo *snoop* para o problema da coerência de caches. Destas 5 soluções 4 são do tipo *write-invalidate* (Berkeley, Goodman, Illinois e Synapse), ou seja, invalida blocos nas outras caches caso haja uma modificação por parte de uma cache, e uma solução do tipo *write-update* que consiste em enviar o bloco modificado por uma cache para todas as outras caches do sistema que detiverem este bloco.

Ainda neste trabalho são propostas quatro alternativas de projeto de hierarquia de caches. As quatro soluções são:

- Uso de caches L1 e L2 privadas;
- Uso de caches L1 privadas e L2 compartilhada;
- Caches L1 e L2 compartilhadas;

- Cache L1 dividida em bancos diferentes porém compartilhados e L2 compartilhada.

Nas duas primeiras soluções se faz necessário o uso de uma solução para o problema da coerência de cache.

São realizados experimentos com as soluções apresentadas utilizando um simulador de hierarquia de caches dirigido por trace chamado CHESS (*Cache Hierarchy Estimator using Scalabe Simulator*). Este simulador tem por características simular vários níveis de cache e ter associado informações de tempo de modo que seja possível obter informações de latência nas simulações.

Os resultados das simulações utilizando um conjunto de aplicações do *benchmark* SPLASH mostram que a solução de uso de caches L1 e L2 privadas com o protocolo Illinois apresentou os melhores resultados em termos de taxa de *miss* e o uso da política de *Write Update* nas soluções de coerência de cache apresentou melhores resultados no que diz respeito a latência. São mostrados outros resultados adotando o protocolo *snoop* para solucionar o problema da coerência de cache nos modelos listados anteriormente que necessitavam de coerência.

Em Loghi (2005) é apresentado um estudo comparativo entre diferentes soluções de coerência de cache. São analisadas soluções de coerência de cache em hardware (protocolo *snoop*) utilizando políticas de atualização *Write-Through Invalidate* (WTI) e *Write-Through Update* (WTU). Na primeira política, a cache que realizar uma modificação no bloco envia uma mensagem para todas as outras notificando e ocorre uma invalidação deste bloco nas caches que estiverem com ele. A segunda política adota postura de atualizar as outras caches quando uma delas faz uma modificação em um bloco. Esta segunda solução faz com que o controlador *snoop* seja um pouco mais complexo.

As outras soluções de coerência de cache apresentadas são baseadas em software: na primeira, é adotada a solução de não armazenar na cache dados compartilhados. Esta é uma solução que simplesmente evita o problema ao invés de solucioná-lo utilizando primitivas de compilação. A ultima solução é tornar o problema tratável pelo sistema operacional. Nesta solução o acesso a variáveis compartilhadas é encapsulado em operações que utilizam travas equivalentes a operações de *test-and-set*. Para o programador, a comunicação passa a ser por troca de mensagens através de primitivas de envio e recebimento.

As simulações são realizadas na plataforma MPARM que utiliza um número configurável de processadores ARM e utiliza como mecanismo de interconexão um barramento que pode ser AMBA ou ST-Bus. Estas simulações utilizam aplicações como produtor-consumidor, multiplicação de matrizes, FFT paralela e decomposição de matrizes.

A Figura 2.13 apresenta o resultado da simulação utilizando as soluções de coerência de cache para a aplicação produtor consumidor. Fica claro o grande custo em termos de tempo de execução e consequentemente em termos de energia gasta por parte da solução de coerência de cache por sistema operacional. Devido ao péssimo resultado, esta solução foi descartada dos experimentos seguintes.

A Figura 2.14 mostra a performance, potência e energia e produto *energy-delay* para a aplicação produtor consumidor variando o tamanho das caches. Nestes resultados pode-se observar que a quantidade de ciclos diminui conforme o tamanho da cache

aumenta, entretanto a queda é ainda maior na solução WTU. Também é possível observar que a solução WTU é a que consome mais potência devido ao fato de ser uma solução de hardware complexa. Entretanto, devido a um menor número de ciclos de execução conforme o tamanho de cache aumenta, a quantidade de energia gasta também diminui e por conseqüência o produto *energy-delay* decai de maneira mais drástica.

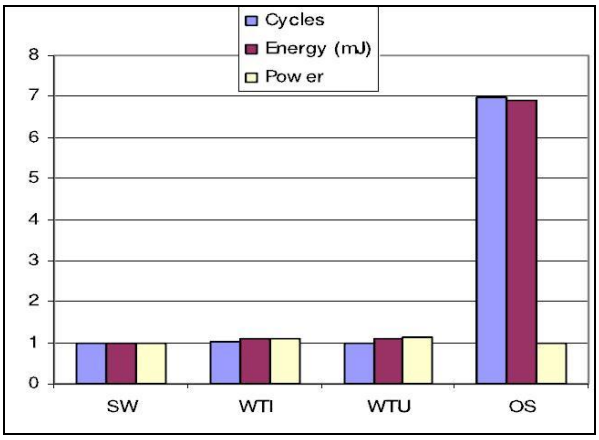


Figura 2.13: Energia e performance para a aplicação produtor consumidor. Retirado de (LOGHI, 2005).

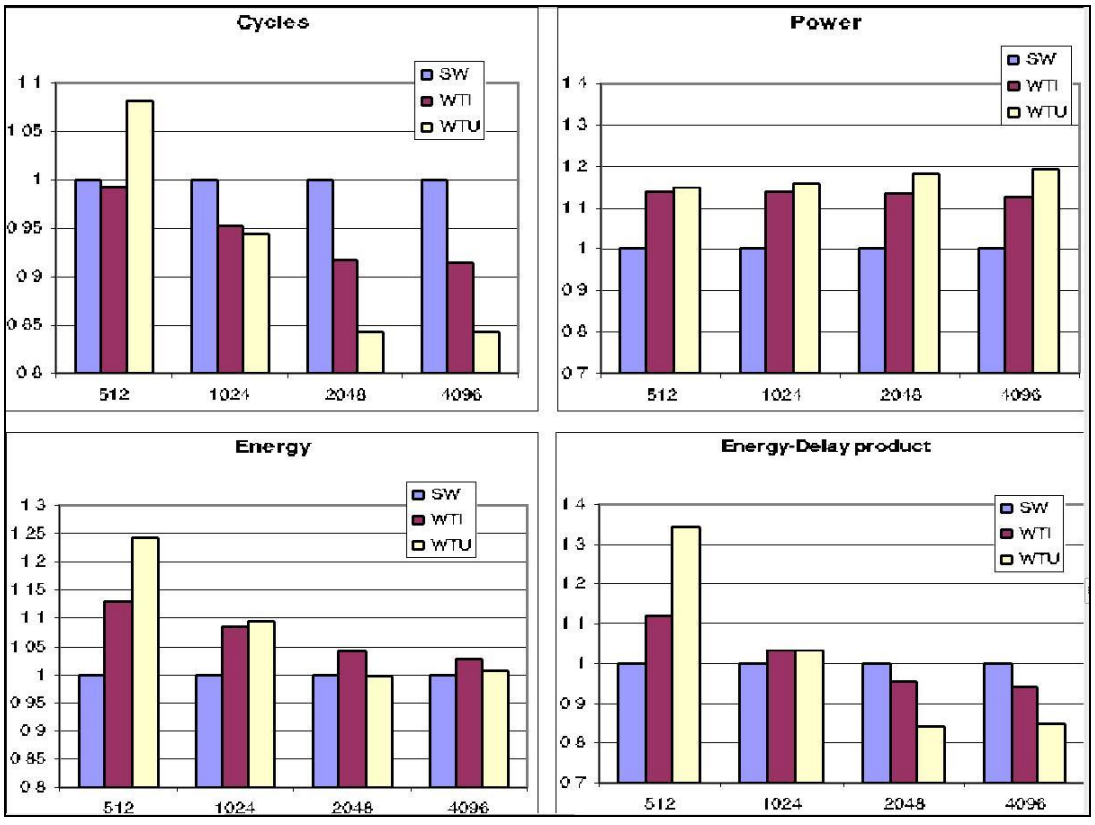


Figura 2.14: Performance, Potência, Energia e *Energy-delay* para a aplicação produtor consumidor variando o tamanho da cache. Retirado de (LOGHI, 2005).

Após a simulação de outras aplicações variando outras características do sistema, o autor conclui que a abordagem do uso do sistema operacional como solução para a coerência de cache apresentou um alto custo em termos de performance e energia. Por

outro lado as soluções de hardware apresentam resultados competitivos em termos de performance, porém com um significativo custo de potência. Por fim, os usos de técnicas para evitar o armazenamento em cache de variáveis compartilhadas apresentam eficiência energética, entretanto implicam em um grande esforço por parte do programador.

Os trabalhos apresentados até aqui nesta seção realizaram investigações de performance para soluções de coerência de cache. Entretanto em nenhum dos dois ambientes foi utilizada solução de NoC. A seguir são apresentados três artigos que apresentam soluções de coerência em MPSoCs baseados em NoCs.

Kim (2006) propõe em seu trabalho uma solução de diretório para a coerência de cache embarcada em *switch crossbar* da rede. Esta solução tem o nome de DCOS (*Directory Cache On Switch*). Nesta abordagem é apresentada uma plataforma multiprocessada que utiliza como mecanismo de interconexão uma NoC. Esta plataforma utiliza ainda o modelo de memória compartilhada distribuída que ainda conta com memórias compartilhadas L2 e memórias cache L1 de dados e de instruções privadas. Neste ambiente é adotada uma solução de diretório para a manutenção da coerência das caches tanto no nível da memória principal quanto no nível L2.

A grande diferença desta solução está no fato de que a tabela e o controlador do diretório estão embutidos nos *switches crossbar* da rede. Entradas nos *switches* da rede são acrescidas de forma a alertar o controle de diretório embutido de que ocorreram modificações nas memórias (Figura 2.15). A informação relativa à tomada de decisão por parte do diretório é enviada diretamente para o arbitro. Esta solução tem o intuito de diminuir o tempo de transferência entre as caches e por consequência o desempenho do sistema como um todo.

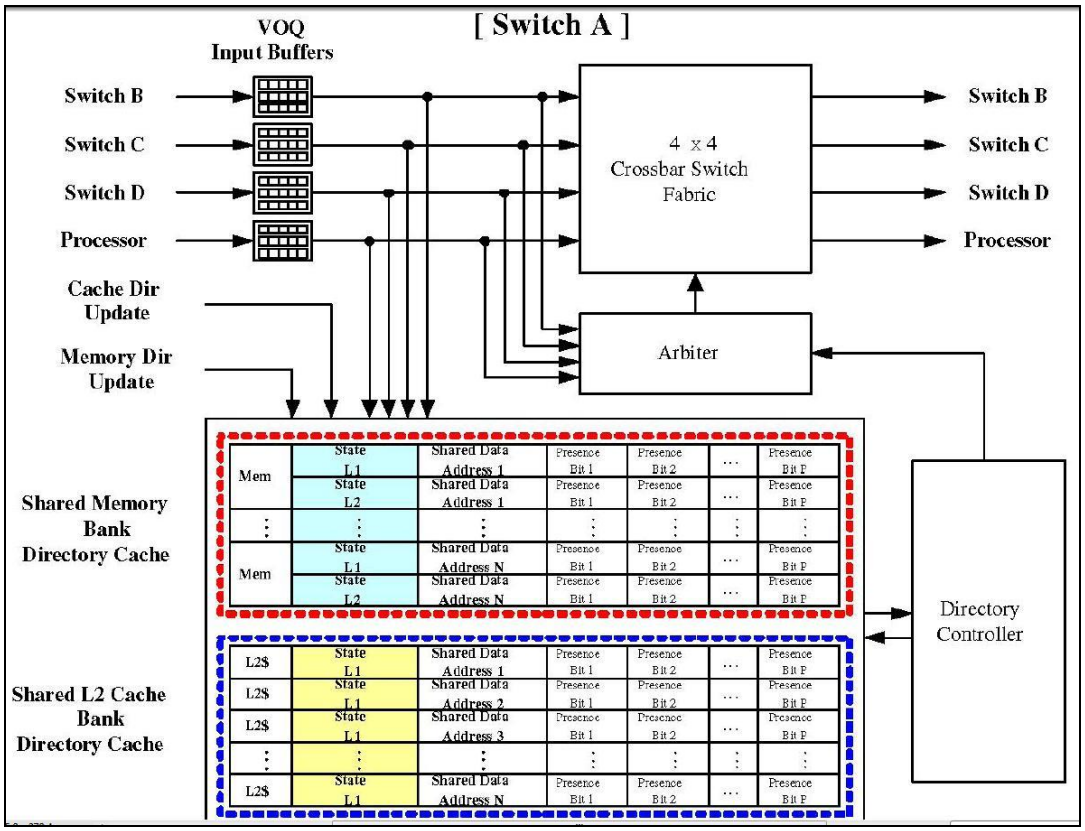
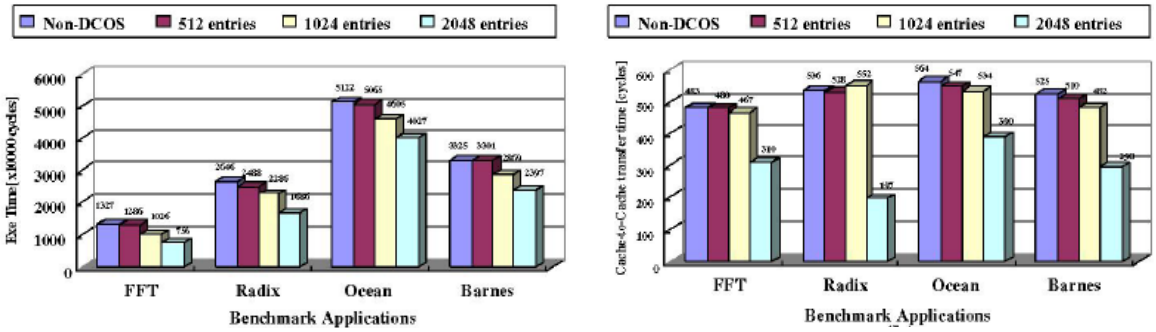


Figura 2.15: Arquitetura DCOS. Retirado de (KIM, 2006).

Utilizando aplicações do benchmark SPLASH-2, são apresentados resultados sobre o tempo de execução bem como o tempo de transferência entre caches. As Figuras 2.16a e 2.16b mostram estes resultados que confirmam o bom desempenho da solução DCOS no que diz respeito ao tempo de execução e tempo de transferência entre caches, respectivamente. Nestes resultados são feitas comparações entre a solução de diretório comum e a solução DCOS variando o número de entradas do diretório da memória compartilhada.



a) b)

Figura 2.16: Resultados da solução DCOS. Retirado de (KIM, 2006).

Petrot (2006) apresenta um solução completamente em software para o problema do compartilhamento de dados em um SoC multiprocessado executando aplicações multi-thread. A solução apresentada se baseia em um modelo de alocação de dados na memória que é estaticamente particionada em dois tipos de segmentos: segmentos locais e segmentos compartilhados. Um segmento é uma entidade lógica definida por um endereço base e um tamanho no espaço de endereçamento. O problema da coerência de cache é solucionado fazendo com que dados dos segmentos compartilhados não sejam armazenados em cache.

Para ganhar acesso a regiões de dados compartilhados seria necessário um mecanismo de *test-and-set* atômico. Para diminuir o tempo de espera para acesso a um determinado endereço, é sugerido o uso de um mecanismo de *lock*. Neste mecanismo, uma leitura a um determinado endereço deve ser interpretado pelo mecanismo como uma operação de *read-and-set to 1*. A *thread* que lê o valor 0 no endereço X detém de maneira exclusiva a trava X. Uma thread que lê o valor 1 deve requerer o acesso até ler o valor 0.

Na solução proposta, é utilizado o kernel MUTEK para que seja possível criar multi-threads na plataforma multiprocessada. Este kernel também permite associar tarefas a processadores no momento da sua criação bem como alocar memória estática ou dinamicamente tanto nos segmentos compartilhados quanto em segmentos locais.

Na Figura 2.17, os autores apresentam resultados de uma aplicação de decodificação JPEG comparando o número de ciclos por instrução para a solução proposta, uma versão da plataforma sem o uso de caches e uma versão onde somente as variáveis utilizadas como *buffers* de comunicação não ficam em cache. É demonstrado um aumento de performance de 50% a 100% da solução proposta se comparado com uma solução completamente sem caches.

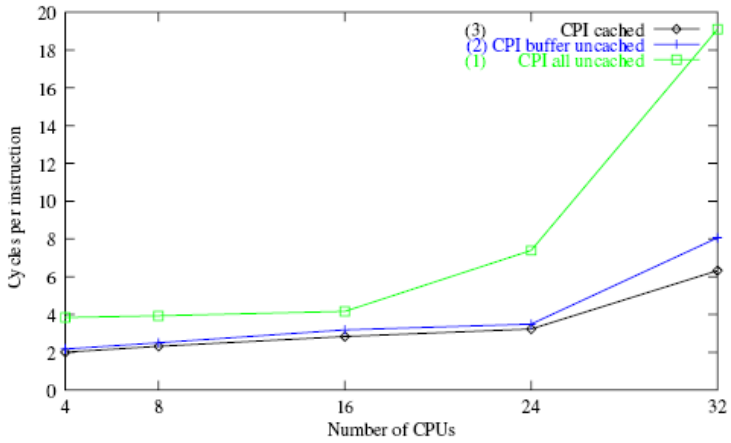
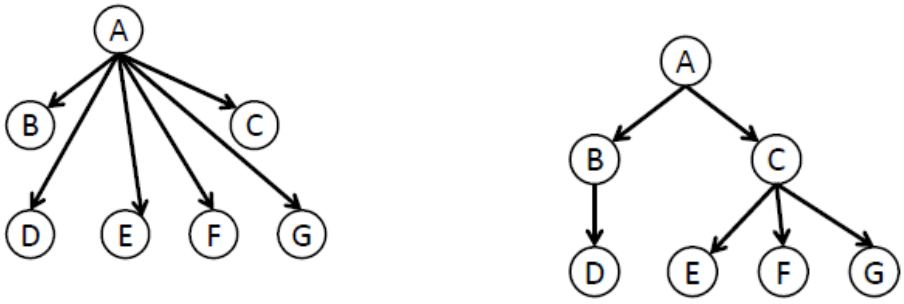


Figura 2.17: Ciclos por Instrução dos três experimentos. Retirado de (PETROT, 2006).

Jerger et al. (2008) apresentam a proposta de um novo protocolo de coerência de cache em ambientes com NoCs chamado *Virtual Tree Coherence* (VTC). Este protocolo se baseia no uso de árvores virtuais para criar uma hierarquia lógica entre nós que compartilham os mesmos dados. Com esta hierarquia, a solicitação de um dado não precisa ser enviada em *broadcast* para todos os nós da rede (para fins de manutenção da coerência) mas apenas para os nós que de fato compartilham este dado.

Tendo em vista a necessidade de operações de *multicast* entre membros de uma mesma árvore lógica e também devido ao fato de que redes-em-chip atuais realizam estas operações na forma de *unicasts* sequenciais, este trabalho também propõe um suporte a operações desta natureza. O *virtual circuit tree multicast* (VCTM) é um mecanismo que se utiliza da hierarquia lógica em árvores para realizar o envio de dados. A Figura 2.18a apresenta um exemplo de envio de mensagens em um sistema em uma NoC sem suporte a operações de *multicast*. Neste caso, um mesmo nó envia mensagens sequenciais para um único nó. A Figura 2.18b ilustra a mesma situação com o suporte da VCTM. Neste segundo caso, o envio da mensagem é enviado para nós em um mesmo nível que, em um segundo passo, enviam estas mensagens aos nós do próximo nível e assim por diante.



a) b)

Figura 2.18: *multicast* em redes-em-chip sem e com suporte da VCTM. Retirado de (JERGER, 2008).

Outra característica deste modelo de coerência de cache é que o mesmo utiliza uma granularidade maior do que o tamanho de um bloco. O mecanismo de *Coarse Grain Coherence Tracking* realiza o gerenciamento de regiões de dados compostas por mais

de um bloco. Esta solução visa utilizar melhor o princípio da localidade espacial das caches. Além disso, este mecanismo permite manter registro de que processadores mantêm em cache que regiões de dados. Isto possibilita que um processador que queira acessar um dado que nenhum outro tenha em cache possa enviar a solicitação de acesso ao dado diretamente à memória principal.

São apresentadas comparações entre a solução VTC com suporte a multicast (VTC-M), com suporte a broadcast (VTC-B), uma solução baseada em *snooping* (Greedy) e uma solução de coerência baseada em diretório. Todos os resultados foram normalizados com relação à solução de diretório. Em termos de performance (ilustrados na Figura 2.19) a solução VTC se mostra, em média, 19% melhor do a solução de diretório e 11% melhor do que a solução de *snooping*. Resultados de tráfego (mostrados na Figura 2.20), por outro lado, apresentam uma clara vantagem da solução de diretório. A solução baseada em *snooping* apresenta uma geração de trafego 3,8 vezes maior, em média, que a solução de diretório. Além disso, a solução VTC baseada em multicast apresenta uma redução de 35% de tráfego com relação à VTC com broadcast.

Neste artigo não são apresentados quaisquer resultados referentes ao consumo de energia destas soluções.

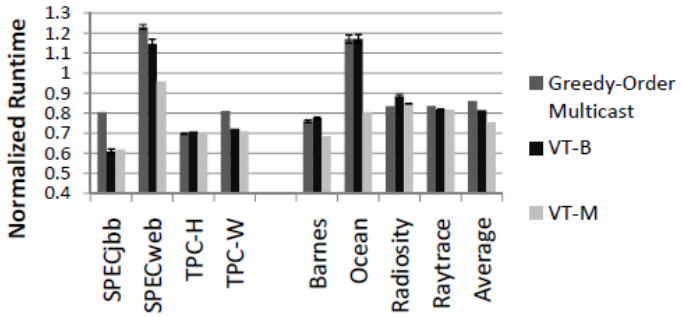


Figura 2.19: Resultados de performance da solução VTC. Retirado de (JERGER, 2008).

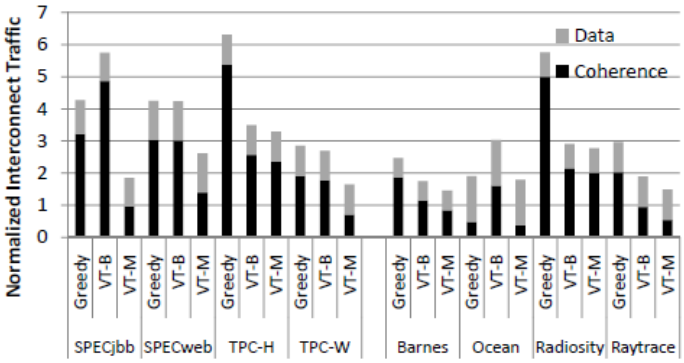


Figura 2.20: Resultados sobre o tráfego na rede da solução VTC. Retirado de (JERGER, 2008).

Como mostrado nos artigos desta seção, diversas soluções para o problema da coerência de cache podem ser adotadas. O custo destas soluções deve ser bem avaliado levando em consideração aspectos relevantes no âmbito dos sistemas embarcados como eficiência energética, desempenho e potência.

Soluções alternativas (mas não completamente excludentes) ao uso de caches em sistemas embarcados são as *scratchpads*. A seção seguinte apresenta dois artigos que falam sobre estes dispositivos no contexto de sistemas embarcados.

2.8 Scratchpads em sistemas embarcados

Devido a sua já comprovada eficiência energética se comparado com o uso de caches, as *scratchpads* têm sido amplamente utilizadas no contexto dos sistemas embarcados. Por ser um dispositivo que requer um suporte explícito do compilador faz-se necessário uma tomada de decisão prévia sobre que conjunto de dados será alocado na *scratchpad* de maneira a obter um melhor desempenho. Grande parte dos artigos sobre scratchpads propõe o uso de um algoritmo de alocação mais eficiente. Dois destes artigos são apresentados a seguir.

Verma (2004), modela o comportamento de uma cache de instruções como um grafo de conflito e, a partir deste, propõe um algoritmo de alocação de dados em uma *scratchpad* que leva em consideração o uso de caches no sistema (CASA - *Cache Aware Scratchpad Allocation*). O artigo mostra que o problema de alocar o melhor conjunto de objetos em uma *scratchpad* pode ser formulado como uma variante do problema do conjunto independente máximo, um conhecido problema NP-completo.

Como dito, este algoritmo de alocação se baseia em um grafo de conflito. Este grafo representa o conflito de endereços dos *memory objects* na cache de instruções. Estes *memory objects* nada mais são do que um conjunto de blocos básicos que estão em um mesmo caminho no fluxo da aplicação. Estes *memory objects* foram utilizados como unidade de alocação de memória, pois são conjuntos de instruções mais frequentemente utilizados.

No grafo de conflito, os vértices são os *memory objects* e os pesos das arestas representam a penalidade em termos de *misses* quando se aloca um *memory object* em detrimento de outro *memory object* (já que, por vezes, dois *memory objects* são mapeados para um mesmo lugar da cache). A abordagem deste algoritmo se baseia em alocar a maior parte dos *memory objects* mais utilizados e que tenham uma maior penalidade de *misses* na scratchpad de modo que os outros *memory objects* que são conflitantes com estes sejam alocados na memória cache. Devido ao fato de que *scratchpads* são mais eficientes energeticamente do que as caches, se a maior parte dos *memory objects* estiver na *scratchpad*, haverá uma maior economia de energia. A Figura 2.21 apresenta um exemplo de grafo de conflito.

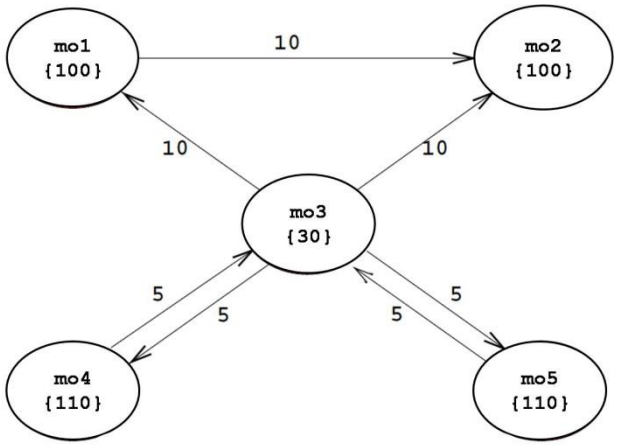


Figura 2.21: Grafo de conflito. Retirado de (VERMA, 2004).

O artigo apresenta resultados comparativos entre a abordagem proposta, outro algoritmo de alocação em *scratchpads* proposto em Steinke (2002) e uma abordagem utilizando uma *loop cache*.

A principal diferença entre a abordagem CASA e a proposta em Steinke (2002) é que na segunda abordagem foi considerada uma hierarquia de memória somente com uma *scratchpad* e uma memória principal enquanto que, em CASA, a hierarquia de memória conta também com uma memória cache. Outra diferença é que em CASA foi considerado apenas o código do programa enquanto que em Steinke (2002) os dados também foram alocados na *scratchpad*.

No que diz respeito à abordagem de *loop cache*, ela se diferencia da proposta no artigo devido ao fato de que na *loop cache*, são armazenados *loops* e funções mais utilizadas enquanto que, na abordagem CASA, os *memory objects* são alocados na memória cache e scratchpad de forma a minimizar os conflitos.

Os resultados mostram a vantagem da abordagem proposta com relação às outras duas. CASA consegue uma redução no consumo de energia, em relação ao outro algoritmo de alocação em scratchpad, diminuindo o número de *misses* na cache de instruções por alocar os *memory objects* também na scratchpad diminuindo o número de conflitos e consequentemente de acessos à memória principal. A vantagem de CASA sobre a solução com *loop cache* se dá devido à pequena quantidade de *memory objects* previamente carregados nesta. A Figura 2.22 mostra os resultados comparativos entre CASA e Steinke (2002) enquanto que na Figura 2.23 são apresentados os resultados de CASA em relação à solução com *loop cache*. Em ambas as figuras os resultados da abordagem CASA com relação à quantidade de acesso à cache de instruções (*I-Cache Access*), quantidade de acesso à scratchpad (*scratchpad access*), taxa de miss da cache de instruções (*I-Cache Miss*) e energia consumida pela memória de instruções (*I-Mem Energy*) foram normalizados de acordo com os resultados de Steinke (2002) que corresponde à marca de 100%.

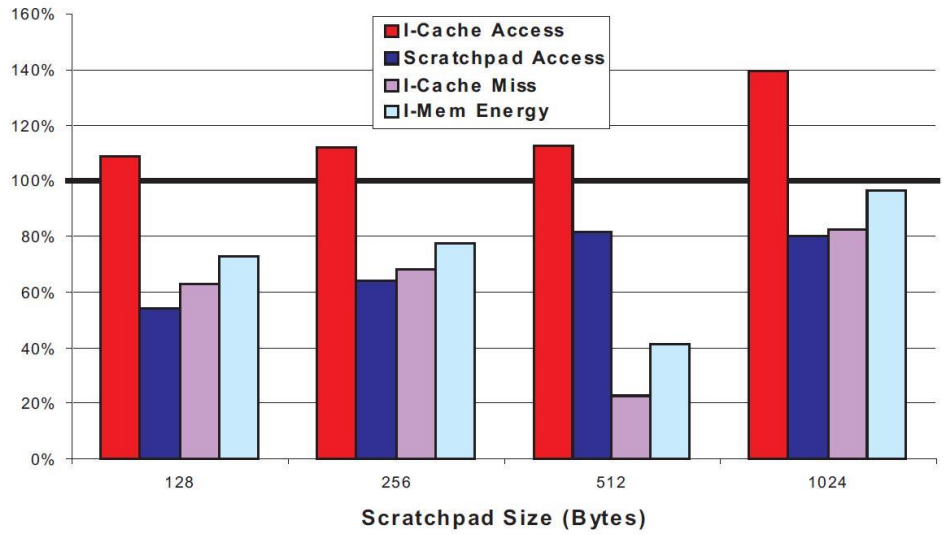


Figura 2.22: Comparação entre CASA e STEINKE. Retirado de (VERMA, 2004).

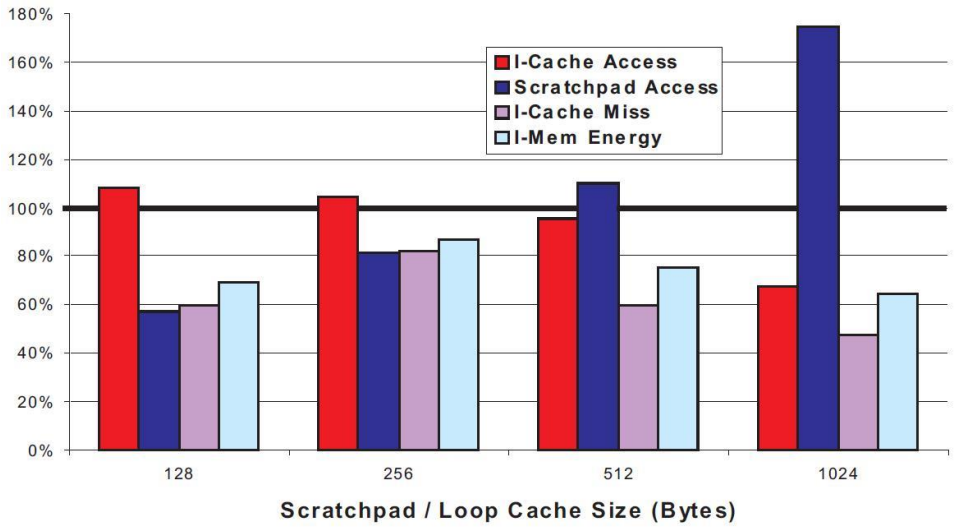


Figura 2.23: Comparação entre CASA e *loop* cache. Retirado de (VERMA, 2004).

Em Suhendra (2006) é apresentada uma técnica de programação linear inteira que leva em consideração o mapeamento de dados nas *scratchpads* de uma plataforma MPSoC para otimizar o mapeamento e escalonamento de tarefas no sistema. O autor defende que a eficiência do sistema depende diretamente da prévia alocação dos dados que uma tarefa utiliza em *scratchpads* mais próximas. A Figura 2.24 representa a plataforma MPSoC utilizada neste trabalho. Cada processador tem uma memória *scratchpad* localizada em seu próprio nó da rede, entretanto é possível acessar qualquer uma das *scratchpads* do sistema.

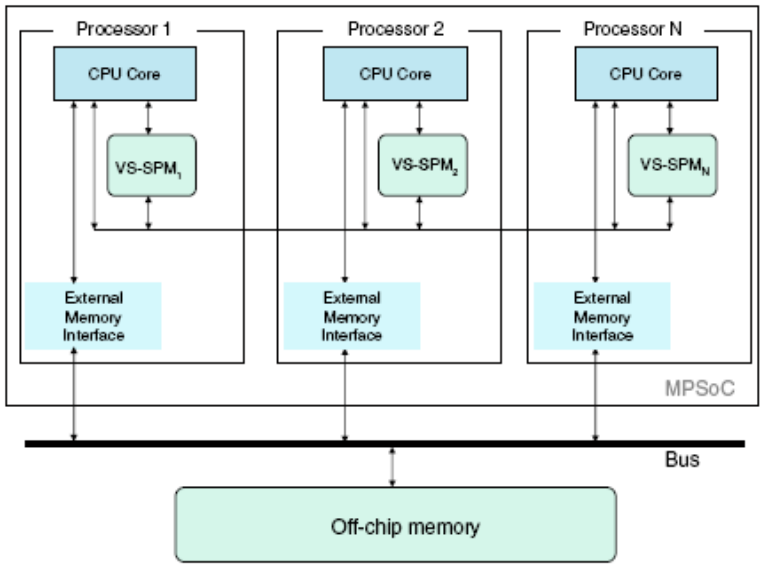


Figura 2.24: Plataforma MPSoC com *scratchpads* fisicamente distribuídas. Retirado de (SUHENDRA, 2006).

A formulação da técnica de programação linear inteira para escalonamento de tarefas, particionamento de *scratchpads* e alocação de dados gera uma solução ótima. Como dito, o objetivo da técnica é explorar a interação entre os diferentes estágios do processo de exploração de espaço de projeto. O autor afirma que isto ajuda a identificar os limites de arquiteturas MPSoC para aplicações embarcadas.

Para a avaliação da técnica são utilizadas três técnicas diferentes para alocação de dados nas *scratchpads* listadas a seguir:

- **EQ:** O escalonamento de tarefas ignora a alocação de dados nas *scratchpads* e as *scratchpads* do sistema mapeiam o mesmo número de endereços;
- **PF:** O escalonamento de tarefas ignora a alocação de dados nas *scratchpads* entretanto o particionamento das *scratchpads* bem como a alocação dos dados são feitos utilizando a técnica de programação linear inteira proposta;
- **CF:** O escalonamento de tarefas, o particionamento das *scratchpads* e a alocação dos dados são feitos utilizando a técnica proposta.

As simulações foram realizadas utilizando aplicações do benchmark MiBench. A Figura 2.25 mostra os resultados das simulações (para duas aplicações distintas) no que diz respeito ao *initiation interval* que representa a diferença de tempo entre o inicio da execução de duas iterações consecutivas do grafo de tarefas. São apresentados resultados para diferentes tamanhos de *scratchpads*. Estes tamanhos dizem respeito à soma das *scratchpads* do sistema. São assumidos valores de 100 ciclos de latência para acesso à memória externa e 4 ciclos para acesso a uma *scratchpad* remota (fora do nó onde se encontra o processador). De acordo com os resultados, ocorre uma melhora de até 60% de desempenho do sistema com o uso da técnica para particionamento das *scratchpads* e até 80% de desempenho se a técnica for utilizada também para o escalonamento de tarefas.

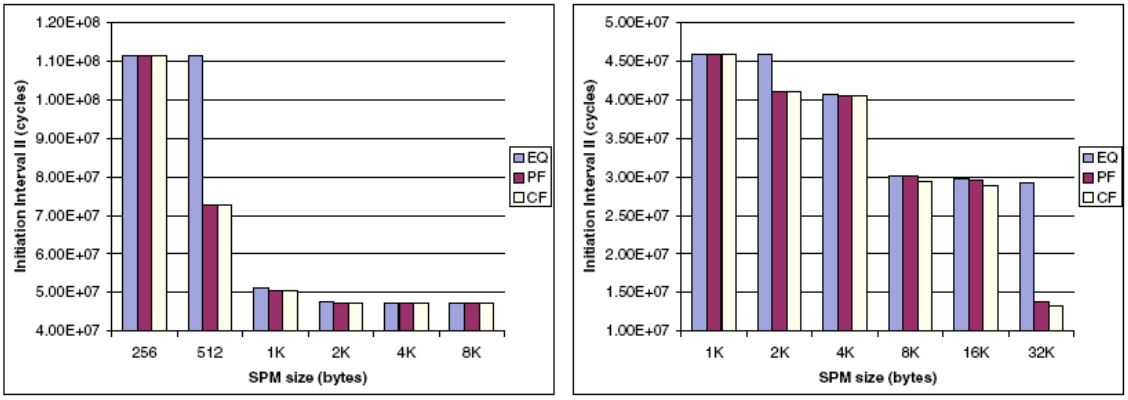


Figura 2.25: Resultados de eficiência da técnica programação linear inteira para particionamento de *scratchpad*, alocação de dados e escalonamento de tarefas. Retirado de (SUHENDRA, 2006).

2.9 Conclusões

Os trabalhos apresentados nesta seção mostram que vários trabalhos vem sendo desenvolvidos ao longo dos anos no contexto de hierarquias de memória em sistemas embarcados e MPSoCs. Entretanto, apenas alguns deles foram realizados considerando ambientes com NoC. É importante notar que o uso de NoCs torna o comportamento do sistema bem como o acesso à memória diferente dos casos com barramentos. Isso se deve, principalmente, ao fato de que NoCs oferecem paralelismo de comunicação e de maneira indireta oferecem uma maior escalabilidade e, assim, uma maior quantidade de elementos de processamento e memórias, o que torna ainda maior a necessidade de um mecanismo de comunicação (em função da organização de memória) eficiente. Dos trabalhos apresentados utilizando NoCs nem todos apresentam uma análise da eficiência

energética e nenhum deles apresenta uma análise comparativa de diferentes organizações de memória como é apresentada nesta dissertação. Além disso nenhum dos trabalhos menciona uma avaliação de organizações de memória em situações de migração de tarefas.

3 A PLATAFORMA VIRTUAL SIMPLE

Neste capítulo será apresentada a plataforma virtual SIMPLE. Nesta plataforma, originalmente contendo apenas um modelo de memória distribuída, foram implementados nesta dissertação os modelos de memória que são alvo deste trabalho. Neste capítulo serão detalhadas as implementações de cada um destes modelos.

3.1 SIMPLE

Com o objetivo de realizar estudos em ambientes multiprocessados baseados em NoC foi desenvolvida dentro do Laboratório de Sistemas Embarcados (LSE) da Universidade Federal do Rio Grande do Sul (UFRGS) uma plataforma virtual denominada SIMPLE (*SIMPLE Multiprocessor Platform Environment*) (BARCELOS, 2008). Esta plataforma foi descrita em systemC em nível de transações (TLM, do inglês, *Transaction Level Model*).

Este ambiente é composto por uma quantidade parametrizável de processadores FemtoJava (ITO 2001), interligados por uma rede de interconexões conhecida como SoCIN. Tanto o processador FemtoJava quanto a rede SoCIN são fruto de trabalhos realizados anteriormente pelo mesmo grupo e serão melhor detalhados mais adiante.

3.1.1 FemtoJava

Os processadores utilizados em SIMPLE são descrições em SystemC do processador FemtoJava multiciclo. O FemtoJava é um processador que executa código Java através de uma máquina de pilha que é compatível com a especificação da Máquina Virtual Java (JVM, do inglês *Java Virtual Machine*). Nesta versão multiciclo do processador FemtoJava, as instruções levam de 3 a 14 ciclos para serem completadas. FemtoJava adota arquitetura Harvard, de tal maneira que existe uma memória de dados e uma de instruções. Além disso, nesta versão multiciclo, a pilha de dados é mapeada na memória.

3.1.2 SoCIN

A NoC utilizada na ferramenta é uma versão em alto nível, também descrita em SystemC, da rede SoCIN (ZEFERINO, 2003). Nesta rede a comunicação é feita por meio de pacotes. Cada pacote é composto por *flits* (*Flow Control Unit*), já que SoCIN utiliza chaveamento de pacotes *wormhole*. Cada roteador possui cinco portas bi-direcionais (norte, sul, leste, oeste e local) nas quais há buffers somente na entrada. Além disso, a rede adota roteamento XY que elimina *deadlocks*.

3.2 Memória distribuída

O modelo de memória distribuída foi concebido originalmente no desenvolvimento da plataforma SIMPLE. Neste modelo, uma quantidade parametrizável de processadores é instanciada. Cada processador faz acesso localmente a uma memória de programa e uma memória de dados. Como apresentado na Figura 3.1, este modelo, assim como os modelos apresentados nas seções seguintes, utiliza uma rede-em-chip para conectar seus componentes. Os blocos marcados com "R" representam os roteadores desta rede-em-chip e os blocos "NI" são as interfaces de rede que conectam os processadores (marcado por "µP") aos roteadores. As interfaces de rede tem a função de atuar na camada de enlace, gerando pacotes que são enviados pela rede mediante uma solicitacao do processador. Outra funcao da interface de rede é receber os pacotes vindos do roteador para o processador. A interface identifica a carga útil do pacote (descartando o cabeçalho) e o envia ao processador. Como mencionado anteriormente, cada processador tem um módulo de memória local (marcado na figura como "MEM") conectado.

Este modelo faz uso de troca de mensagens explicitas entre os processadores. Originalmente na plataforma SIMPLE, o envio e recebimento destas mensagens era feito completamente por software. Estas funções realizavam o protocolo *handshake* necessário para enviar dados pela interface de rede. Entretanto estas funções levavam vários ciclos para enviarem uma mensagem e isto seria um fator que influenciaria os resultados dos experimentos. Assim, a solução adotada nesta dissertação foi implementar um módulo de DMA agregado ao processador. Este DMA recebe do processador apenas as informações necessárias para enviar a mensagem. Estas informações se resumem a: endereço de rede do destino, endereço de memória inicial do buffer de envio e, por fim, a quantidade de bytes a serem enviados a partir deste endereço de memória.

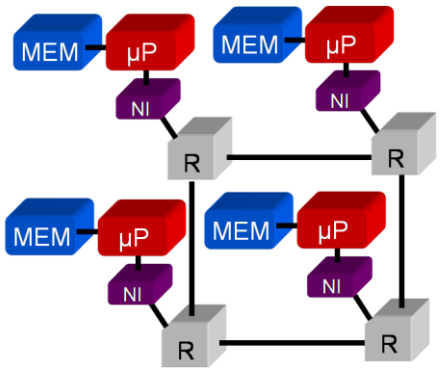


Figura 3.1: Modelo de memória distribuída em SIMPLE.

3.3 Memória compartilhada

O modelo de memória compartilhada implementado em SIMPLE consiste de uma memória de dados localizada em um nó da rede e acessada remotamente por todos os processadores do sistema. Neste modelo e nos outros modelos descritos a seguir, a memória de instruções foi mantida local ao processador. Esta decisão foi tomada devido ao fato de que o acesso aos dados possui uma natureza não-linear e, por isso, uma

análise do desempenho das organizações de memórias de dados se torna menos intuitivo e mais relevante.

Para diminuir o tempo médio de acesso à hierarquia de memória, foram desenvolvidas caches de dados privadas e locais para cada processador. As memórias cache implementadas são completamente associativas e o uso de seus algoritmos de substituição de blocos é parametrizável. Atualmente, existem implementações de algoritmos tradicionais como LRU (*Least Recently Used*), FIFO (*First In First Out*), FIFO *second chance*, LFU (*Least Frequently Used*) e aleatório. Outras características relativas à memória cache como tamanho e tamanho do bloco também são parametrizáveis. Uma solução de coerência de cache baseada em diretório é adotada. Esta solução de coerência de cache está descrita em (GIRÃO, 2007) e é resultado de trabalhos anteiores a esta dissertação. No entanto, fez-se necessária adaptações partindo da implementação original a fim de utilizar esta solução na plataforma SIMPLE

Devido a características da implementação do processador FemtoJava, a resposta do acesso à memória deve ser dada em um ciclo. Por isso, em SIMPLE adota-se uma solução de *clock-gating* para parar o processador sempre que o mesmo solicita uma leitura ou escrita. Este método foi implementado e utilizado com sucesso na dissertação de Rodrigo Bittencourt Motta (MOTTA, 2008). Uma das adaptações no modelo de coerência de cache adotado diz respeito à inclusão deste mecanismo de *clock-gating* na própria cache.

Outra preocupação neste modelo é o fato de que o FemtoJava multiciclo mapeia a pilha de dados na memória. Neste caso, utilizou-se uma pequena memória (diferente da memória principal) para armazenar a pilha. Esta memória encontra-se no nó do processador e um controlador, baseado no endereço solicitado, faz acesso à pilha ou à cache de dados.

Para realizar os pedidos de acesso à memória remota, as caches contam com um módulo gerenciador de comunicação chamado CaCoMa (*Cache Communication Manager*). Uma das funções deste módulo é o envio e recebimento de mensagens tanto de acesso a dados quanto de controle de coerência de cache. Outra adaptação do modelo de coerência de cache ocorreu neste módulo uma vez que se fez necessária a adoção do protocolo de envio e recebimento de dados para a NoC utilizada na plataforma SIMPLE.

O CaCoMa tem a função de traduzir os endereços lógicos em endereços físicos. Por isso, todas as operações de acesso à memória pela cache (em um eventual caso de *read miss*) são intermediadas por este módulo.

O modelo de comunicação neste ambiente é baseado em variáveis compartilhadas. A fim de manter a consistência dos dados foi necessário dar suporte a operações sobre *mutexes* como *down* e *up*. A primeira é utilizada para testar se a variável de *mutex* está liberada para poder acessar uma região crítica. A função *up* libera o *mutex*. Estas operações precisam ser atômicas e por isso a solução adotada foi mapear estas funções *down* e *up* em operações de *test-and-set* e *test-and-reset*, respectivamente.

A função de *down* implementada sinaliza para a cache que se faz necessária uma operação de *test-and-set* em um determinado endereço de memória. Então, a cache repassa esta informação para o CaCoMa que por sua vez envia um pacote com essas informações para o diretório. Um vez que o pacote chega ao diretório, este realiza um teste no endereço indicado e caso o valor seja zero ele muda o valor para um. Caso

contrário, não há mudanças. Um novo pacote de resposta é enviado com o valor após a operação e, se o *mutex* estiver travado, o CaCoMa envia uma nova solicitação para o mesmo endereço.

Já a função *up* realiza uma operação similar, porém não há teste. O valor no endereço indicado é modificado para o valor zero e então uma mensagem confirmando a operação é enviada pelo diretório ao CaCoMa.

Este modelo de memória compartilhada e espaço de endereçamento único está sujeito ao problema da coerência de cache. Neste caso foi adotada uma solução baseada em diretório por ser uma solução que não necessita de operações de *broadcast*, as quais não são suportadas pela NoC. Esta implementação será melhor detalhada a seguir.

3.3.1 Implementação da solução de coerência de cache baseada em diretório

Para centralizar as informações de um módulo de memória e manter a coerência das caches, foi implementado um módulo Diretório (marcado na figura como “DIR”) que se encontra associado à memória como mostra a Figura 3.2. Nesta figura também encontra-se representado o módulod e Cache de Dados (marcado como “D$”) que contém, além da própria cache, o CaCoMA). O Diretório, assim como o CaCoMa, também atua na camada de enlace de dados gerando pacotes que serão enviados pela rede bem como interpretando pacotes de solicitações oriundas dos outros nós. Este diretório implementado na plataforma é caracterizado como um diretório *Full-Map* uma vez que utiliza *n* bits por bloco (onde *n* é o número de processadores) para saber que cache detém que bloco, mais um bit para sinalizar o estado deste bloco.

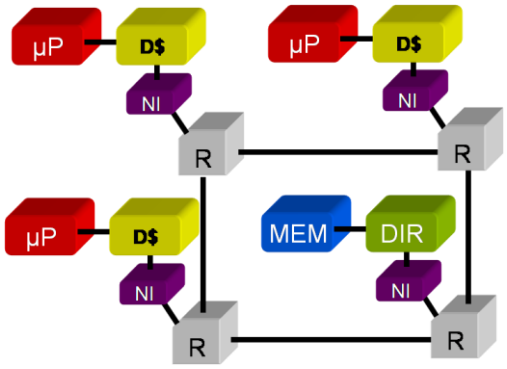


Figura 3.2: Memória compartilhada com coerência de cache baseada em diretório em SIMPLE.

Como dito anteriormente, o Diretório mantém informações sobre todos os blocos daquela memória cujas cópias estão nas caches espalhadas pelo sistema. Assim, ele é capaz de implementar a coerência entre elas através de operações de controle tais como *write-backs* forçados (requisições de atualização do bloco na memória) e invalidações de blocos. Para realizar estas operações de coerência, o Diretório mantém duas tabelas: a STA (*Status Table*) e a PTA (*Processor Table*).

A STA informa quais caches de processadores contêm uma cópia de um bloco da memória. Esta tabela contém *B* linhas e *P*+1 colunas. *B* é o número de blocos da memória, isto é, cada linha da tabela fornece informações de um bloco (linha 0 contém informações do bloco 0, linha 1 contém informações do bloco 1 e assim por diante). *P* é o número de processadores e cada coluna contém a informação de quais blocos estão na cache de cada processador. Dessa forma, a coluna 0 informa quais os blocos que estão

na cache do processador 0, a coluna 1 informa quais os blocos que estão na cache do processador 1 e assim por diante. Uma coluna extra indica se o bloco em questão está sujo ou não. Um bloco estar sujo significa que algum processador fez uma escrita neste bloco e, portanto, o bloco na memória não está atualizado. Esta situação acarreta na necessidade (caso uma outra cache solicite uma cópia deste bloco) da cache que contém a cópia suja neste bloco realizar uma operação de *write-back*, ou seja, enviar a cópia de volta do bloco para a memória. A Figura 3.3 mostra um exemplo de uma STA.

BLOCO	P0	P1	P2	P3	Sujo
0	1	1	1	0	0
1	0	1	0	0	1
2	0	0	1	0	1
3	0	0	1	0	0

Figura 3.3. Tabela STA.

Para que seja possível enviar pedidos como o *write-back* para as caches que contêm os blocos sujos, por exemplo, é necessário que o diretório saiba o endereço de NoC do processador. Para isso, o Diretório mantém a PTA que relaciona endereços de rede e o processador presente neste endereço. Um exemplo de uma tabela PTA está ilustrado na Figura 3.4.

Processador	Endereço
P0	0,0
P1	0,1
P2	1,0
P3	1,1

Figura 3.4. Tabela PTA.

Para cada solicitação oriunda de um processador do sistema, o diretório realiza modificações na STA e ocasionalmente realiza requisições de *write-back* e invalidações. A Figura 3.5 mostra um fluxograma que define as ações tomadas pelo diretório dependendo do estado do bloco solicitado para leitura (*read miss*). Caso o bloco esteja limpo, o diretório simplesmente atualiza a tabela STA indicando que mais um processador está compartilhando este bloco. Por fim o bloco é enviado para o processador solicitante. Por outro lado, se o bloco solicitado estiver sujo, o diretório envia uma solicitação de *write-back* sem invalidação para o processador que detém a cópia do bloco. Uma vez enviada a solicitação, o diretório aguarda a chegada do bloco para que este possa ser enviado ao processador solicitante (uma *flag* de atualização pendente é setada). O pedido de atualização enviado não inclui um pedido de invalidação pois o processador requisitante do bloco não intenciona modificá-lo (solicitação de leitura) e com isso ambos os processadores podem compartilhar este bloco.

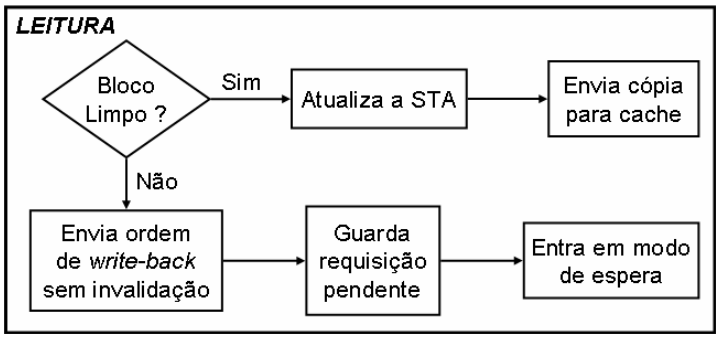


Figura 3.5: Operações do diretório em uma solicitação de leitura.

Em uma situação de escrita (*write miss*), o diretório novamente verifica se o bloco solicitado encontra-se limpo ou sujo. Caso ele esteja limpo, o diretório inicia o envio de *N* solicitações de invalidação do bloco para os *N* processadores que compartilham-no (já que somente um processador por vez pode ter posse, para escrita, de um bloco). Uma vez realizado o envio destas mensagens, ocorre a atualização da STA indicando que somente o processador solicitante terá o bloco e o estado deste é sujo. No caso de uma solicitação de escrita em um bloco que está sujo, o diretório envia uma solicitação de atualização desta vez com invalidação (de novo devido ao fato de que cada bloco só pode ser modificado por um processador por vez) para o processador que atualmente detém o bloco em sua cache. Feito isso, o diretório aguarda a chegada do bloco solicitado (novamente uma *flag* de atualização pendente é setada) e, quando isto ocorrer, o bloco é atualizado na memória e enviado ao processador requisitante. Este processo é descrito no fluxograma da Figura 3.6.

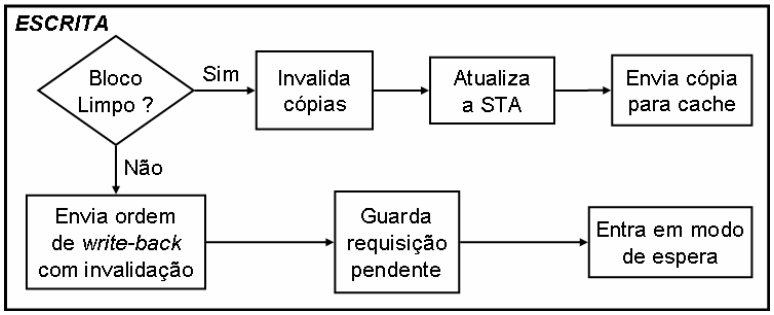


Figura 3.6: Operações do diretório em uma solicitação de escrita.

Uma outra solicitação possível nesta solução de coerência de cache é um pedido de permissão de escrita. As caches podem ter os blocos somente para leitura no momento que lhes é solicitado uma escrita no mesmo. Neste caso, a cache precisa enviar ao diretório um pedido para modificar o bloco que ela já tem. Esta operação é importante pois dependendo do estado do bloco (se sujo ou limpo) o diretório realiza operações diferentes tanto na leitura quanto na escrita. O fluxograma da Figura 3.7 representa as ações do diretório nesta situação.

Quando um pedido de permissão de escrita chega ao diretório é necessário verificar se a cache ainda possui o bloco. Esta situação é possível no caso em que uma cache envia o pedido de permissão de escrita antes de perceber a chegada de um pedido de invalidação oriundo do diretório devido a um pedido de escrita (que chegou ao diretório primeiro) feito por algum outro processador no sistema. Dessa forma, quando o diretório verifica que esta situação ocorreu, este ignora o pedido de permissão de escrita da cache. A cache, por sua vez, quando percebe que fez um pedido de permissão de

escrita e agora recebeu um pedido de invalidação, gera um novo pedido. Desta vez o pedido será de um bloco para escrita, o que acarreta na situação de *write miss* descrita anteriormente.

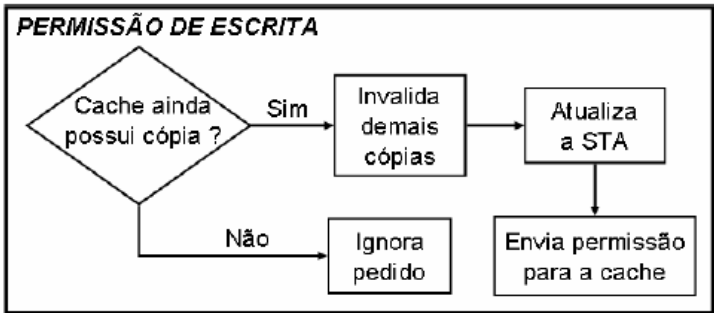


Figura 3.7: Operações do diretório em um pedido de permissão de escrita.

Como visto anteriormente, situações de pedidos de leitura e escrita (*read miss* e *write miss*) podem resultar em solicitações de *write-back* por parte do diretório. Quando isto é feito, o diretório passa a esperar o retorno destas solicitações e não atende a novas solicitações até que o *write-back* esperado seja recebido. Isto é feito para evitar que o diretório inicie o tratamento de uma nova solicitação sem garantir que o fim da solicitação anterior tenha mantido a coerência das caches do sistema. Entretanto, o diretório atende a solicitações à medida que elas chegam no buffer de recebimento. Assim, a mensagem de *write-back* que ele esteja esperando pode ficar presa no meio da fila de solicitações e assim criar uma situação de *deadlock* onde o diretório não processa nenhuma solicitação e os processadores ficam esperando a resposta de suas solicitações. Para resolver este problema, o diretório mantém dois buffers de requisições, um para mensagens de *write-back* e outro para as outras mensagens. O buffer de *write-backs* tem prioridade sobre o outro buffer e assim, o diretório trata primeiro todas as mensagens de *write-back.*

Logo, como ilustrado na Figura 3.8, ao receber uma mensagem de *write-back* o diretório verifica se existe uma requisição de *write-back* pendente (resultado de uma solicitação de leitura ou escrita). Em caso positivo, verifica se aquela mensagem diz respeito a esta pendência. Neste caso, a memória é atualizada e o diretório responde à solicitação que originou a necessidade de um *write-back.* Caso não seja o *write-back* esperado ou se simplesmente o diretório não estiver esperando *write-back* algum, o diretório atualiza a memória e a tabela STA.

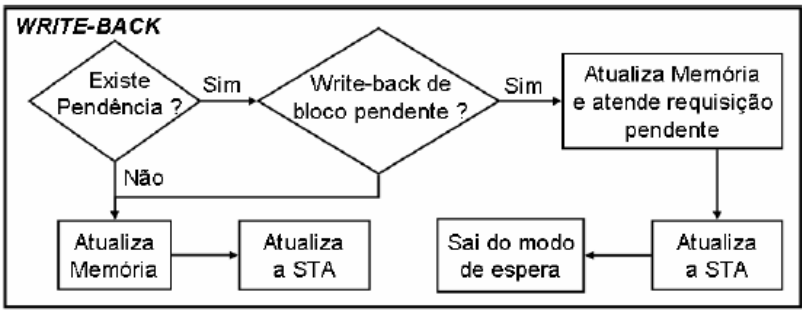


Figura 3.8: Operações do diretório na chegada de um write-back.

3.4 Memória compartilhada distribuída

Um modelo de memória compartilhada distribuída diz respeito a um modelo onde o espaço de endereçamento é compartilhado (todos os endereços são acessáveis por todos os processadores), porém os endereços estão distribuídos em memórias fisicamente distribuídas no sistema. O uso de barramentos tende a não aproveitar os benefícios desta abordagem devido à natureza sequencial do mecanismo de comunicação. NoCs por sua vez permitem que diferentes módulos de memória na rede possam ser acessados de maneira paralela e, com isso, tirar proveito deste modelo.

Na implementação de memória compartilhada distribuída em SIMPLE, diferentes módulos físicos de memória estão espalhados pelo sistema. Para um espaço de *x* endereços e *n* módulos de memória, cada módulo mapeará *x*/*n* endereços seqüenciais. Dessa maneira, se uma memória n_1 tem como último endereço *i* obrigatoriamente existe uma memória n_2 no sistema que tem como primeiro endereço *i*+1. Esta situação é mostrada na Figura 3.9 que ilustra um ambiente com duas memórias, ambas com 32 endereços (*x* igual a 64 e *n* igual a 2).

Este modelo é muito semelhante ao modelo de memória compartilhada. O fato de existir um espaço de endereçamento compartilhado e caches locais também acarreta no problema da coerência de cache. Assim, este modelo também faz uso de uma solução de coerência de cache baseada em diretório. Nesse caso cada módulo físico de memória está associado a um módulo diretório que realiza a manutenção da coerência de cache relativa à faixa de endereços compreendida neste módulo de memória.

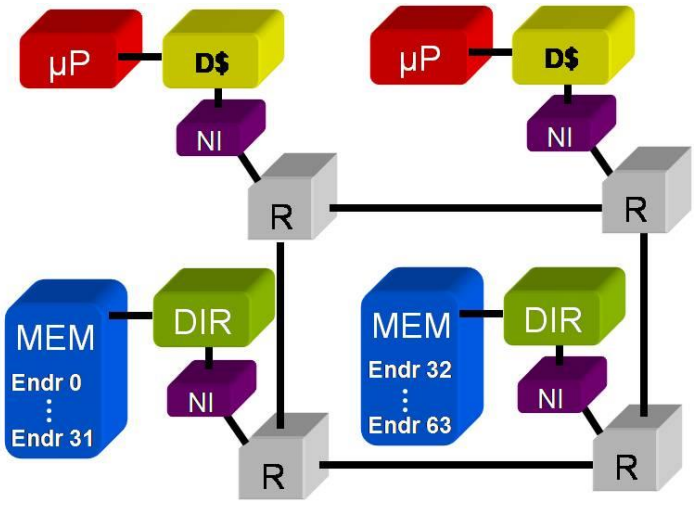


Figura 3.9: Memória compartilhada distribuída em SIMPLE.

Uma mudança com relação ao modelo de memória compartilha é que, para que as solicitações sejam enviadas ao diretório correto, a cache precisa descobrir a que módulo de memória (em termos de endereço de rede) pertence aquele endereço solicitado. A obtenção desta localização é feita a partir de uma tabela chamada de ATA (*Address Table*) que contém os endereços de NoC de todos os módulos de memória, bem como o último endereço lógico das mesmas. Assim, é possível saber em que posição e em que módulo de memória encontra-se o dado requisitado pela cache. A Figura 3.10 mostra um exemplo de uma tabela ATA.

Endereço Lógico	Endereço Físico
0x00000100	0,1
0x00000300	0,2
0x00000500	1,2

Figura 3.10. Tabela ATA.

Neste modelo a comunicação por variáveis compartilhadas é idêntica à utilizada no modelo de memória compartilhada

3.5 Memória nDMA

Os modelos de memória apresentados até agora são bastante conhecidos na literatura e difundidos em vários sistemas multiprocessados conhecidos. Esta seção introduz um novo modelo de memória baseado na combinação de um espaço de endereçamento distribuído (como no modelo de memória distribuída) e o uso de uma memória única e, de fato, fisicamente compartilhada na rede.

Neste modelo de memória a comunicação se dá por uma programação similar aos módulos de DMA. Os processadores da NoC enviam mensagens de programação do DMA para que possam ser feitas cópias de dados. Devido a esta semelhança, este novo modelo (ilustrado na Figura 3.11) foi denominado nDMA (NoC DMA). O objetivo deste tipo de comunicação é minimizar o tráfego de dados pela rede, fazendo uso do fato de que todas as memórias dos processadores estão localizadas no mesmo nó da rede.

Esta memória é composta por *NP* (onde *NP* diz respeito ao número de processadores no sistema) bancos de memória localizados em um único nó da rede e gerenciados por um único controlador de memória (identificado na Figura 3.11 como “MC”). Cada banco armazena os dados de um único processador. Para cada requisição que chega no nó da memória, o controlador é responsável por identificar o processador requisitante e então acessar o seu respectivo banco de memória.

Cada processador do sistema tem sua própria cache de dados. Com o uso de uma memória que suporta diferentes espaços de endereçamento não existe a necessidade de qualquer mecanismo de coerência de cache uma vez que não existem dados compartilhados. Esta característica já sugere uma redução no tráfego da rede se comparado com as outras memórias compartilhadas.

Uma outra consequência de ter espaços de endereçamento distintos é que o método de comunicação mais intuitivo a ser utilizado é o de troca de mensagens. Tipicamente, a troca de mensagens é implementada enviando dados de um processador (de sua própria memória, para ser mais preciso) para outro através do meio físico de comunicação. No entanto, neste modelo apresentado, todas as memórias estão centralizadas em um mesmo nó e por isso não há a necessidade de enviar dados. Consequentemente, o método de troca de mensagens precisa ser adaptado para que funcione em tal ambiente.

Basicamente, o processador que deseja enviar uma mensagem precisa enviar um pedido de cópia (*copy request*) para a memória (ao invés de enviá-lo para o processador destino) informando um endereço de origem e a quantidade de bytes a serem copiados a partir dele. No nó da memória, o controlador de memória identifica esta mensagem e

envia uma notificação ao processador de destino para informar a intenção de um processador em enviar dados para ele. O processador de destino responde a esta notificação com um o endereço de destino para esta comunicação em particular. Quando esta resposta chego no nó da memória, o controlador inicia a cópia dos dados do banco de memória do processador de origem para o banco de memória do processador de destino. A Figura 3.12 descreve estes passos para realizar a comunicação no modelo nDMA.

Considerando que cada processador tem sua própria cache privada, no momento de enviar um pedido de cópia, o processador de origem pode ter, em sua cache, blocos que estejam dentro da faixa de endereços que ele deseja enviar ao processador de destino. Neste caso, ocorre uma verificação destes blocos nesta cache privada. Caso algum deles tenha sido modificado é necessário que este seja atualizado na memória. Isto acarreta em um *write-back* forçado por parte da cache. Este processo é imperativo para que a transferencia dados entre os dois processadores seja feita utilizando-se blocos mais atuais.

De maneira similar, o processador de destino, ao ser notificado de que algum outro processador intenciona enviar-lhe dados, precisa realizar operações de *write-back* forçado. Este *write-back* forçado será feito caso ele tenha em sua cache blocos que contenham endereços que façam parte do buffer de recebimento. Esta operação de *write-back* deve ser realizada antes da cópia dos dados entre um processador e outro. Além disso, estes blocos precisam ser invalidados na cache de dados do processador de destino, já que serão modificados após a cópia dos dados.

Estes são os únicos momentos em que ocorre troca de dados pela rede para fins de comunicação.

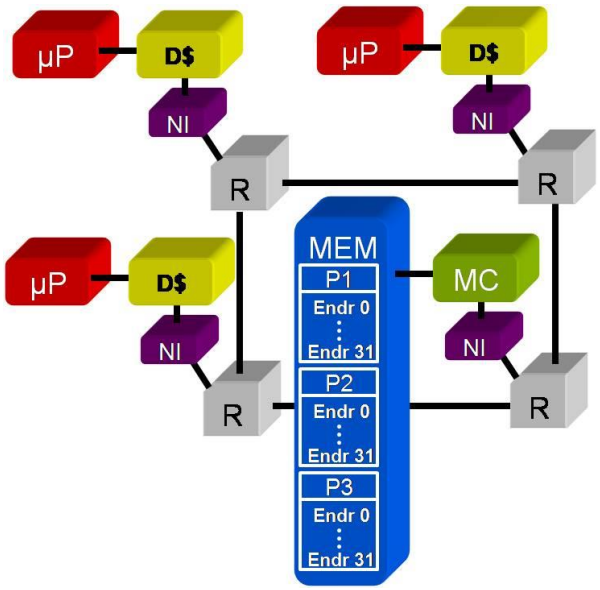


Figura 3.11: Memória nDMA em SIMPLE.

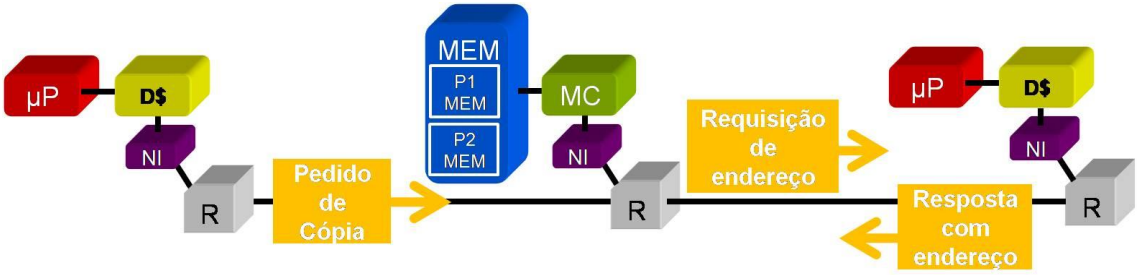


Figura 3.12: Modelo de troca de mensagens no modelo nDMA.

3.6 Gerador de Tráfego

O meio físico de comunicação de um sistema multiprocessado afeta diretamente o modelo de memória adotado. Isto se deve ao fato de que o modelo de memória leva a um modelo de comunicação da aplicação que, por sua vez, é influenciado pelo mecanismo físico de comunicação. Baseado nisso, foi realizado um estudo com o objetivo de emular um ambiente com alta latência de comunicação e quantificar seu impacto no sistema para diferentes modelos de memória.

Para emular tal ambiente, foi desenvolvido um gerador de tráfego sintético incorporado no módulo de interface de rede na plataforma SIMPLE. A idéia é criar um tráfego sintético para aumentar a latência do meio de comunicação e assim, emular a execução paralela de diversas aplicações que geram comunicação entre os diversos processadores.

O Gerador de Tráfego implementado está situado dentro da interface de rede de cada nó do sistema, como ilustrado na Figura 3.13, e uma máquina de estados coordena a inclusão de um pacote de tráfego no buffer de envio da interface de rede. Fazendo uso de um identificador no cabeçalho do pacote, o Gerador de Tráfego é capaz de analisar o buffer de recebimento a fim de excluir qualquer pacote de tráfego antes que ele seja processado pelo processador ao qual a interface de rede está associada.

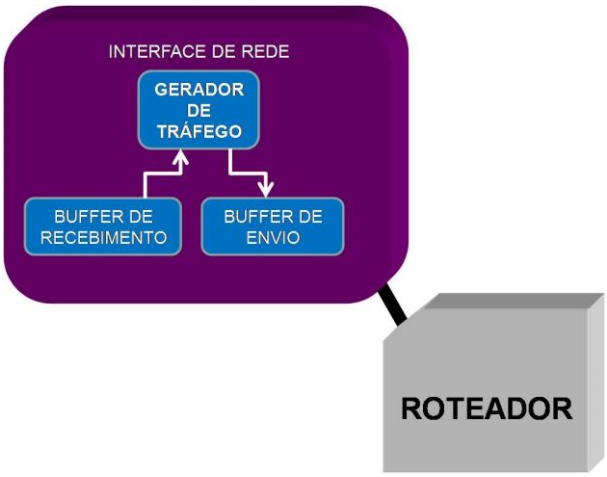


Figura 3.13. Gerador de Tráfego.

Este componente trabalha de maneira bastante simples. A cada período (previamente configurado), o Gerador de Tráfego cria um pacote de 10 bytes a ser enviado a um nó destino determinado através de uma sequência *round-robin* entre todos os roteadores do sistema, um por vez. O pacote de tráfego utilizado tem tamanho de 10 bytes para que a interface de rede possa enviá-lo entre um período de envio de tráfego sintético e outro. O objetivo da adoção desta abordagem round-robin é estressar o meio de comunciação de uma maneira uniforme de forma a analisar o impacto do throughput da rede-em-chip nos modelos de memória. É sabido que este comportamento é improvavél de acontecer em um cenário real. Entretanto o experimento tem a intenção apenas de testar os limites dos modelos estudados.

Com este sistema de envio de tráfego seguindo uma seqüência *round-robin*, a cada 10 ciclos, por exemplo, um roteador 0 no sistema envia um pacote de tráfego para o roteador 1 enquanto que o roteador 1 envia um pacote de tráfego para o roteador 2 e assim por diante. No próximo ciclo, o roteador 0 envia um pacote para o roteador 2 e o roteador 1 envia um pacote para o roteador 3 e assim por diante. Logo, a cada período, cada roteador envia um pacote de tráfego para um destino diferente de modo que o impacto causado pelo alto tráfego gerado seja distribuído por toda a rede.

Se o período é igual a 1, a cada ciclo o Gerador de Tráfego criaria pacotes de tráfego e assim, a rede ficaria ocupada 100% do tempo. Nos experimentos realizados neste trabalho foram utilizadas cargas de 10% e 20%, o que significa que a cada 10 ciclos e 5 ciclos um pacote de tráfego será criado, respectivamente. Estes resultados são apresentados na Seção 4.3.

O tráfego criado por este Gerador de Tráfego pode ser considerado uniformemente distribuído em termos espacial e temporal. A simplicidade deste gerador tem o único intuito de aumentar a latência de comunicação do sistema. No futuro pretende-se realizar experimentos emulando tráfego real obtido através de aplicações simuladas préviamente. Outros Geradores de Tráfego também podem vir a ser incorporados na plataforma virtual tais como Carara (2007) e Mahadevan (2005).

3.7 Suporte à migração de tarefas

O processo de migração de tarefas vem sendo utilizado desde muito tempo no contexto de sistemas distribuídos. O grande objetivo de migrar uma tarefa é obter ganhos de performance ao distribuir tarefas dentro do sistema de modo a ter uma distribuição, idealmente, homogênea entre os elementos de processamento.

O processo da tomada de decisão de quando deve ser efetuada uma migração das tarefas é conhecido como alocação de tarefas. Dá-se o nome de migração de tarefas ao processo de transferir a execução de uma tarefa de um processador para outro (TANENBAUM, 1992).

A fim de tornar possível a funcionalidade da migração de tarefas, é necessário que o sistema em questão dê suporte à transferência do contexto da aplicação. Este contexto é composto de conteúdos de registradores no momento da migração, o conteúdo da memória de dados e memória de programa, além dos recursos alocados para a tarefa a ser migrada tais como arquivos ou conexões.

Com a larga utilização de sistemas multiprocessados e também com o advento das NoCs, o conceito de migração de tarefas tornou-se alvo de trabalhos no contexto de sistemas embarcados. Entretanto, diferentemente do contexto de sistemas distribuídos,

em se tratando de sistemas embarcados, existe um requisito que se torna tão importante quanto o desempenho: eficiência energética. Dessa forma, uma migração de tarefas que consome muita energia pode não ser uma solução interessante.

Em sistemas embarcados, o balanceamento das cargas do sistema pode ser feito em prol de uma maior eficiência energética. Este balanceamento pode ser realizado adotando-se técnicas como o DVS (do inglês, *Dynamic Voltage Scaling*) (WEISER, 1994) ou DPM (do inglês, *Dynamic Power Management*) (BENINI, 2000). A técnica de DVS consiste em diminuir a voltagem do processador para que consuma menos energia enquanto que a técnica de DPM consiste em desligar partes do sistema que estejam ociosas.

Alguns poucos trabalhos foram desenvolvidos abordando migração de tarefas no contexto de MPSoCs baseados em NoC (NOLLET, 2005) (BARCELOS, 2007), entretanto nenhum realizou uma análise comparativa de modelos de migração de tarefas em diferentes organizações de memória.

Na dissertação de mestrado de Daniel Barcelos (BARCELOS, 2008) foi proposto um modelo de migração de tarefas bem como uma detalhada avaliação em termos de consumo de energia de todo o processo de migração incluindo um detalhamento dos componentes responsáveis pelo gasto energético. Neste mesmo trabalho foi demonstrado que existe um período de amortização para compensar o custo da migração. Esta compensação existe devido à distribuição das cargas no sistema fazendo com que todas as tarefas sejam executadas mais rapidamente. Porém esta compensação também pode ocorrer na forma de uma economia no consumo de energia utilizando a técnica DVS. Ao realizar a migração das tarefas, o tempo de execução das mesmas diminui (já que há um melhor balanceamento das cargas), assim, é possível, utilizando o DVS, diminuir a freqüência de operação dos processadores e assim economizar energia e ainda assim atender os requisitos temporais das tarefas em questão.

Estes experimentos de migração de tarefas foram realizados na plataforma SIMPLE utilizando o modelo de memória distribuída. Com as rotinas de migração de tarefas utilizadas neste trabalho foi possível dar suporte à migração de tarefas nos outros modelos de memória disponíveis na plataforma SIMPLE.

A seguir é descrito como a migração de tarefas é realizada em cada um dos modelos de memória.

- **Migração de tarefas no modelo de memória distribuído**

Como descrito anteriormente, o modelo de migração de tarefas adotado consiste em migrar o código da aplicação bem como os dados e a pilha. Cada um destes elementos é enviado de um nó para outro utilizando rotinas que lêem os dados da memória (de programa ou de dados) do femtoJava e os enviam utilizando a interface de rede. Rotinas similares recebem o conteúdo no nó de destino e escrevem os dados e o código nas respectivas memórias.

- **Migração de tarefas no modelo de memória compartilhada**

Devido ao fato de que a memória de dados é compartilhada, não é necessária a migração dos dados no processo de migração da tarefa no modelo de memória

compartilhada. Sendo assim, somente o código e a pilha são transferidos de um nó para outro da rede.

Como o mecanismo de coerência de cache baseado em diretório mantém uma tabela relacionando o endereço de NoC de cada processador do sistema, é necessário o envio de uma mensagem de controle no momento da migração para avisar ao diretório que a cache naquele nó já não possui aqueles dados ou uma mensagem de *write-back* atualizando os dados modificados na memória.

Com o objetivo de dar suporte ao envio tanto de mensagens de solicitação de dados à memória (efetuadas pela cache) quanto a mensagens de migração de tarefas geradas pelo próprio FemtoJava, fez-se necessária a inclusão de um árbitro que recebe ambos os pedidos de envio de dados (da cache e do processador) e os encaminha para a interface de rede.

- **Migração de tarefas no modelo de memória compartilhada distribuída**

No modelo de memória compartilhada distribuída o processo de migração de tarefas ocorre exatamente da mesma forma que no modelo de memória compartilhada já que em ambos os modelos a memória de dados é centralizada.

A mesma modificação realizada no modelo de memória compartilhada ao utilizar um árbitro para gerenciar o envio de mensagens de acesso à memória e migração de tarefas foi utilizada neste ambiente.

- **Migração de tarefas no modelo de memória nDMA**

No caso do modelo de memória nDMA, são realizados os envios da memória de programa bem como a pilha. No caso da memória de dados, já que a mesma se encontra no mesmo nó que todas as outras memórias do sistema, a transferência dos dados é feita localmente e em paralelo com o envio do código e da pilha, tornando o processo teoricamente mais rápido do que a migração no modelo de memória distribuída.

Neste modelo também foi utilizada a solução de um árbitro para o envio de mensagens de migração de tarefas como mencionado nos modelos anteriores. Além disso, a máquina de estados do controlador de memória foi modificado para que possa identificar a chegada de uma mensagem de migração de tarefas. Esta mensagem simples identifica a origem e destino dos dados, ou seja, que tarefa sai de que nó para que outro nó da rede. De posse desta informação o controlador realiza a cópia dos dados de um módulo de memória para outro. Toda esta operação é realizada localmente sem a necessidade de gerar tráfego na rede e com isso faz com que o envio do código da tarefa possa ser realizado concomitantemente.

4 RESULTADOS EXPERIMENTAIS

4.1 Aplicações e paralelismo aplicado

Para obter os resultados apresentados nesta seção, quatro aplicações foram implementadas: um algoritmo de estimação de movimento, uma multiplicação de matrizes, um codificador JPEG e um algoritmo de ordenação conhecido como Mergesort.

O primeiro algoritmo, cada processador busca um macrobloco (um subconjunto de uma imagem) em uma parte diferente (embora de mesmo tamanho) do quadro de referência. Nas simulações, um macrobloco de 8x8 pixels e uma imagem de formato QCIF (176x144 pixels) foram utilizados.

No segundo algoritmo foi paralelizado de tal maneira que cada processador realiza a multiplicação de um subconjunto igual de linhas da matriz A por todas as colunas da matriz B. Cada matriz utilizada nas simulações tem 32 x 32 elementos.

Um codificador JPEG pode ser visto como um algoritmo de três passos. Os primeiros dois passos (DCT 2-D e Quantização) podem ser realizados em paralelo para diferentes partes da imagem. Entretanto, o terceiro passo (Codificação de entropia) para ser realizado de maneira correta precisa como entrada a imagem como um todo (resultado da aplicação dos dois primeiros passos). Baseado nisso, a implementação paralela deste codificador JPEG divide uma imagem de 32x16 pixels de tamanho em oito blocos de 8x8 e cada processador é responsável por executar os primeiros dois passos em uma quantidade igual desses 8 blocos. No fim destes passos, cada processador envia os blocos resultantes para um processador mestre o qual realiza o passo final com a imagem completa.

O Mergesort é um algoritmo de ordenação que utiliza o conceito de dividir-para-conquistar. O procedimento utilizado é dividir no meio os elementos a serem ordenados e aplicar esta operação recursivamente até obter subsequências de tamanho 1 e a partir daí unir as subsequências ordenando-as (*merge*). A Figura 4.1 ilustra como funciona o algoritmo. Os passos de dividir-para-conquistar utilizados pelo Mergesort são:

- Dividir: Dividir os dados em subsequências pequenas;
- Conquistar: Classificar as duas metades recursivamente aplicando o Mergesort;
- Combinar: Juntar as duas metades em um único conjunto já classificado.

A versão paralela do algoritmo Mergesort utilizada nas simulações consiste em dividir o vetor igualmente entre os processadores do sistema e cada um deles ordenar sua parte do vetor. Ao fim de todas as ordenações, um processador é responsável por ordenar as subseqüências ordenadas por cada um dos outros processadores (inclusive a

subseqüência ordenada por ele mesmo), gerando assim o vetor original ordenado. O algoritmo paralelo utilizado nas simulações ordena um vetor de 2000 números inteiros.

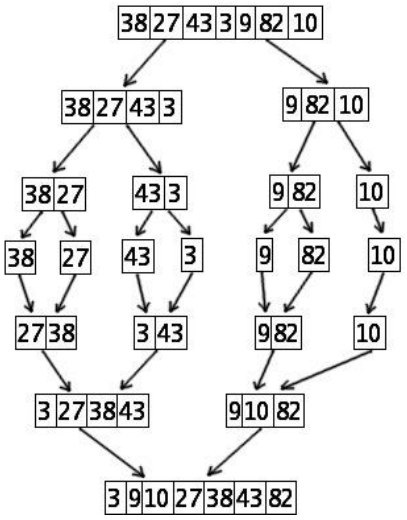


Figura 4.1. Funcionamento do Mergesort.

Para cada uma das aplicações foi modelado um comportamento paralelo como descrito. Nas versões dos algoritmos para o modelo de memória distribuída foi utilizado um modelo Mestre/Escravo. Na Estimação de Movimento, por exemplo, um processador executando o algoritmo Mestre divide igualmente, de acordo com o número de elementos de processamento (para cada caso com 2, 4 ou 8 processadores), o quadro de referência e envia estes dados para cada Escravo, além do macrobloco a ser buscado no quadro de referência. No fim da execução, o Escravo envia para o Mestre as coordenadas de onde foi encontrado o macrobloco dentro do quadro de referência. Os outros algoritmos para o modelo de memória distribuída foram implementados de maneira similar utilizando o modelo Mestre/Escravo, no qual o Mestre divide igualmente os dados e os envia para os Escravos e estes enviam o resultado de sua computação ao Mestre.

Nas versões dos algoritmos para memória compartilhada e compartilhada distribuída foram utilizados *mutexes* cujas funções *down* e *up* realizam uma solicitação de *test-and-set* e *test-and-reset* respectivamente para a cache. No caso da Multiplicação de Matrizes, por exemplo, cada processador executa o mesmo código e no inicio da execução realiza acesso a uma variável que retorna um identificador único (variável essa protegida por um *mutex*). Baseado neste identificador, o processador realiza a computação sobre parte dos dados. No caso da Multiplicação de Matrizes, o dado é uma quantidade de linhas da matriz A e todas as colunas da matriz B para que sejam geradas linhas da matriz C. O acesso às colunas da matriz B é protegida por um *mutex* bem como a matriz C. Os outros algoritmos implementados para os modelos de memória compartilhada e compartilhada distribuída seguem este mesmo modelo de programação com *mutexes*.

O modelo de memória nDMA adota o mesmo modelo Mestre/Escravo utilizado no modelo de memória distribuída. Entretanto, não é realizado o envio dos dados, apenas são enviadas mensagens para que os dados sejam copiados para os respectivos Escravos.

As tabelas abaixo apresentam o tamanho médio (considerando 2, 4 e 8 processadores), em bytes, da memória de programa e de dados das aplicações para cada modelo de memória.

Tabela 4.1. Tamanho das aplicações no modelo de memória distribuída.

Aplicação	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória de dados	110612	21392	27172	319268
Memória de programa	28261	5098	65294	24404

Tabela 4.2. Tamanho das aplicações nos modelos de memória compartilhada e compartilhada distribuída.

Aplicação	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória de dados	103904	17968	27388	305312
Memória de programa	18280	4492	59572	16220

Tabela 4.3. Tamanho das aplicações no modelo de memória nDMA.

Aplicação	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória de dados	109232	20540	27496	315937
Memória de programa	22208	5035	64423	23208

Considerando as entradas de dados de cada aplicação descrita acima, a Figura 4.2 abaixo mostra um gráfico da carga de comunicação aplicada no sistema em cada uma das aplicações e para quantidades diferentes de processadores. Tomando este gráfico como base, é esperado, por exemplo, que o algoritmo de estimação de movimento gere uma quantidade maior de troca de dados.

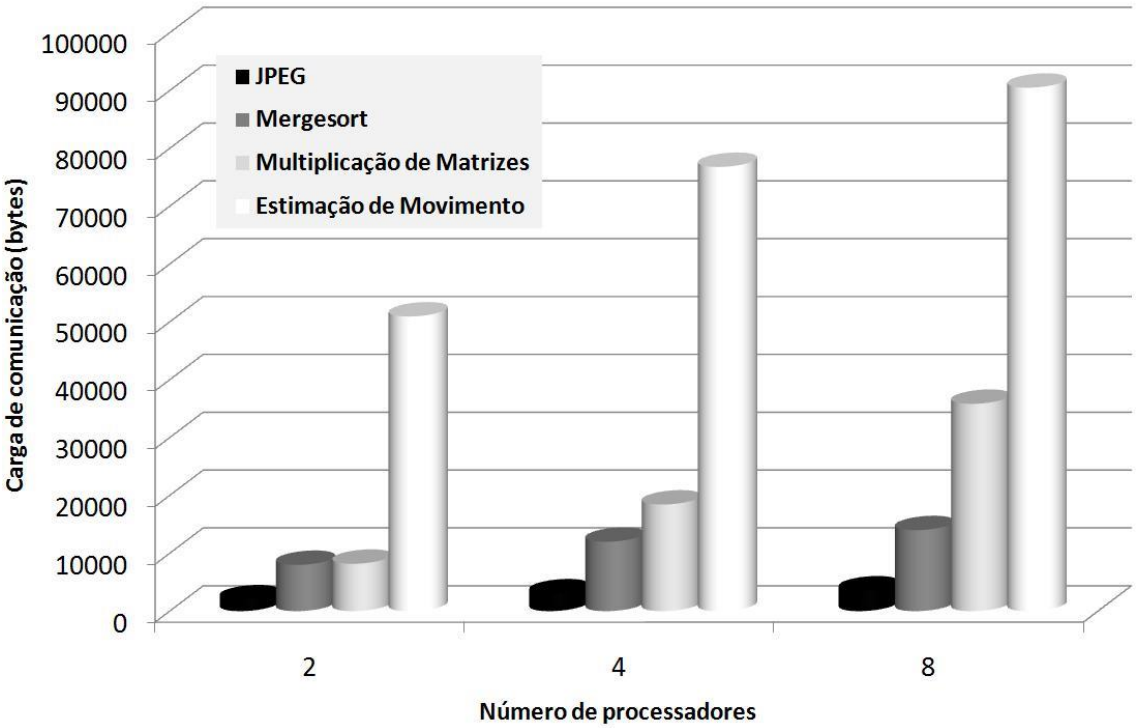


Figura 4.2: Carga de comunicação das aplicações.

Os experimentos apresentados nesta seção tiveram o objetivo de medir três características: performance, consumo total de energia e tráfego na rede.

Os valores de performance obtidos se referem ao número de ciclos necessários até que o ultimo processador terminasse sua execução.

O consumo de energia foi obtido considerando quatro aspectos:

- A energia da rede (incluindo buffers, árbitros, *crossbar* e *links*);
- Energia dos módulos de memória;
- Energia das caches;
- Energia de todos os processadores utilizados.

A energia da NoC foi calculada baseando-se na biblioteca Orion (WANG, 2002), enquanto que a energia das memórias e das caches foi calculada utilizando a ferramenta Cacti (WILTON, 1996). A tecnologia considerada para a obtenção destes consumos de energia foi 0,18µm. Todos os valores de energia considerados aqui dizem respeito à energia dinâmica do sistema.

No que diz respeito ao tráfego da NoC, foi medido o número total de bytes transferidos entre todos os módulos (processadores e memórias) durante a simulação.

A Tabela 4.4 abaixo apresenta as configurações de diferentes componentes do sistema com os as quais foram realizadas as simulações. É importante notar que estas simulações foram realizadas em um simulador deterministico com precisão de ciclo. Isso significa que, para uma mesma configuração (numero de processadores, tamanho de caches, aplicação e outras caracteristicas como as listadas na tabela abaixo), todo o comportamento do sistema não se altera quando executado mais de uma vez. Por consequencia não se faz necessária a execução de várias instancias para determinar um

caso médio uma vez que todos os resultados seriam iguais. Também é importante ressaltar que a precisão dos resultados é consequencia da precisão das bibliotecas Orion e Cacti citadas anteriormente. Para avaliar a precisao dos resultados seria necessario realizar uma comparação das configurações propostas neste trabalho com uma plataforma real que reproduzisse com fidelidade tais ambiente. Devido a indisponibilidade de tal plataforma, esta analise não faz parte do escopo deste trabalho.

Tabela 4.4. Características dos componentes utilizados nos experimentos.

Característica	**Valores**
Número de Processadores	2, 4 e 8
Frequência dos Processadores	100MhZ
Tamanho das Caches	256, 512 e 1024 bytes (exceto para a aplicação Multiplicação de Matrizes que utilizou caches de tamanho 1024, 2048 e 4096 bytes)
Tamanho do bloco da Cache	32 bytes (8 palavras)
Associatividade da Cache	Associativa por conjunto (*2-way associative*)
Política de escrita da Cache	Write-back
Roteamento da NoC	XY
Chaveamento da NoC	*Wormhole*
Topologia da NoC	Grelha (*mesh*)
Arbitragem da NoC	*Round-robin*
Frequência de operação da NoC	100MhZ

Quando utilizando 2 processadores a NoC encontra-se no formato 1x2 no modelo de memória distribuída, 1x3 nos modelos de memória compartilhada e nDMA e 2x2 no caso do modelo de memória compartilhada distribuída devido à adoção de duas memórias remotas. No caso de 4 processadores, a distribuição da NoC fica 1x4 no modelo de memória distribuída, 1x5 nos modelos de memória compartilhada e nDMA e 2x4 no caso do modelo de memória compartilhada distribuída, pelo uso de quatro memórias remotas. Em uma situação com 8 processadores, a NoC apresenta-se na configuração 1x8 no modelo de memória distribuída, 3x3 nos modelos de memória compartilhada e nDMA e 4x4 no caso do modelo de memória compartilhada distribuída, como resultado da adoção de 8 memórias remotas.

4.2 Experimentos com modelos de memória

Nesta seção são apresentados resultados gerados pela simulação dos benchmarks apresentados previamente. Foi considerado um ambiente onde apenas uma aplicação está sendo executada e sem nenhum tráfego adicional na rede.

4.2.1 Resultados quanto à performance

As Figuras 4.3, 4.4, 4.5 e 4.6 apresentam os resultados de performance para a Estimação de movimento, Multiplicação de matrizes, JPEG e Mergesort, respectivamente.

A memória nDMA apresenta resultados muito similares se comparada com as outras duas memórias compartilhadas. Para a aplicação JPEG, a nDMA apresenta resultados ainda melhores na maioria dos casos. Considerando as outras aplicações, a memória compartilhada distribuída apresenta uma ligeira vantagem no desempenho, especialmente em situações com 8 processadores. No caso da aplicação Mergesort, a memória distribuída apresenta um desempenho bem melhor do que os outros modelos e este desempenho é ainda maior conforme o número de processadores no sistema aumenta. Este melhor desempenho é causado pelo acesso paralelo à memória pelos vários processadores. Entretanto esta vantagem não aumenta proporcionalmente de acordo com o número de memórias devido ao fato de que nem todos os módulos de memória tem a mesma taxa de acesso. Logo, fica claro que algumas memórias são mais acessadas que outras diminuindo, assim, a vantagem deste modelo. Em suma, todas as três organizações de memória compartilhada apresentam resultados de performance similares entre si e seguem um mesmo padrão ao aumentar a performance conforme o número de processadores ou tamanho de cache aumenta.

Para os algoritmos de multiplicação de matrizes, JPEG e Mergesort, o modelo de memória distribuída apresenta melhores resultados do que todos os outros modelos. Quanto à Estimação de Movimento, o gráfico da Figura 4.4 indica que a memória distribuída não apresenta resultados melhores que as outras organizações. Isto é devido ao fato de que este algoritmo requer várias trocas de dados (resultando em longas trocas de mensagens) entre os processadores, como foi mostrado na Figura 4.2. Esta característica não parece afetar as outras organizações pelo simples fato de que, no caso das memórias compartilhada e compartilhada distribuída, não há troca explicita de dados e, para a memória nDMA, os dados têm que trafegar pela NoC e, assim, a latência da rede é evitada.

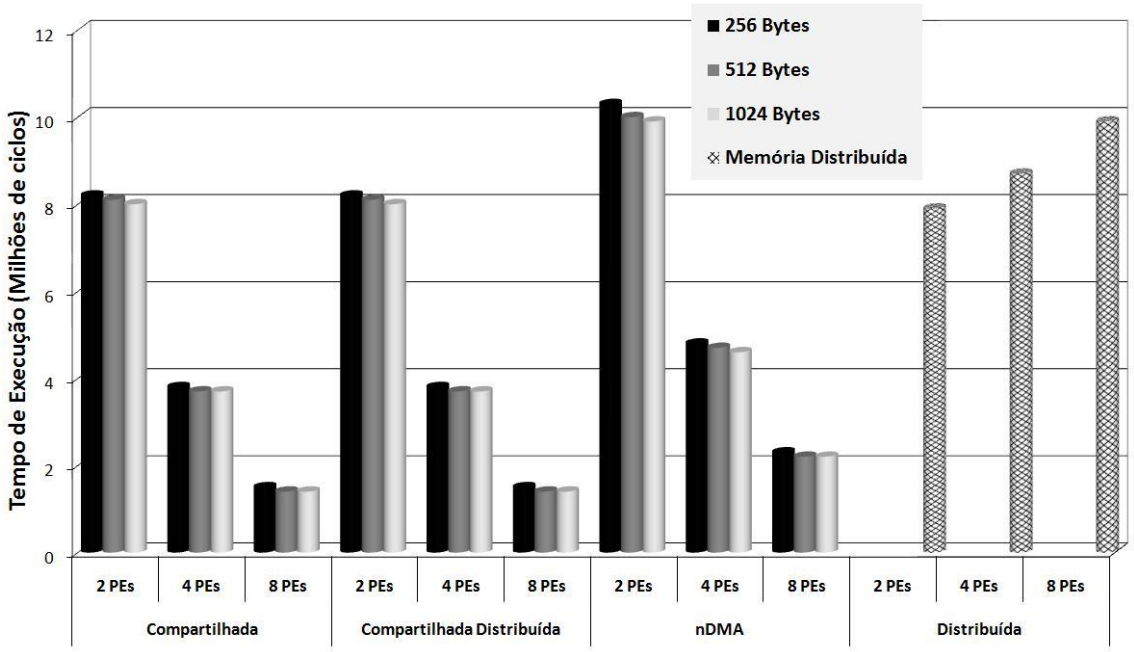


Figura 4.3: Performance na aplicação de Estimação de movimento.

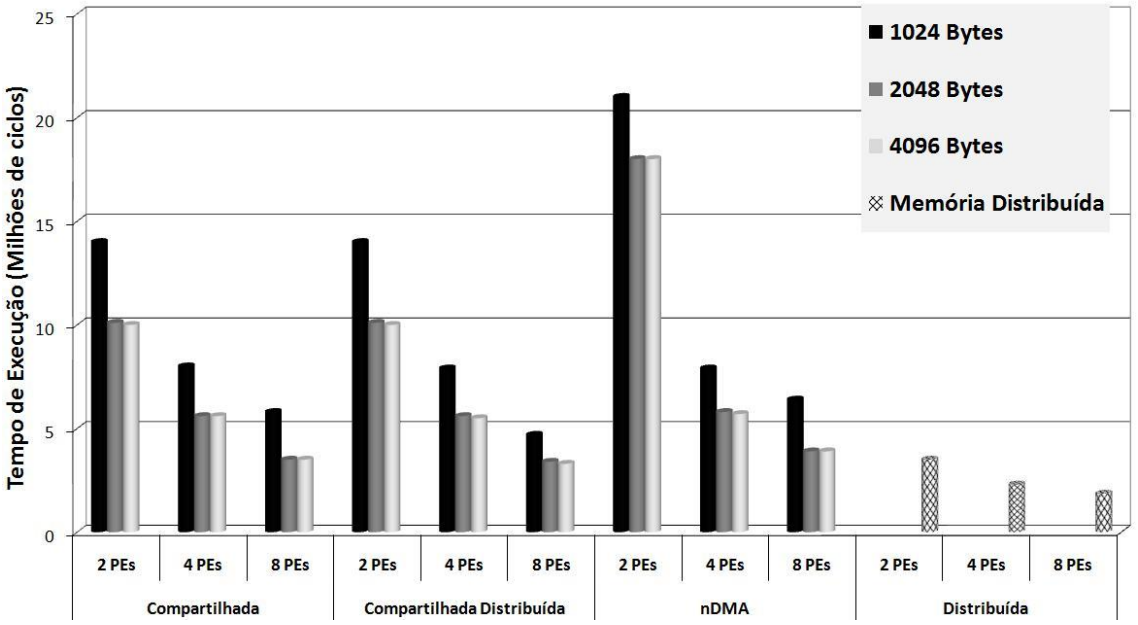


Figura 4.4: Performance na aplicação de Multiplicação de matrizes.

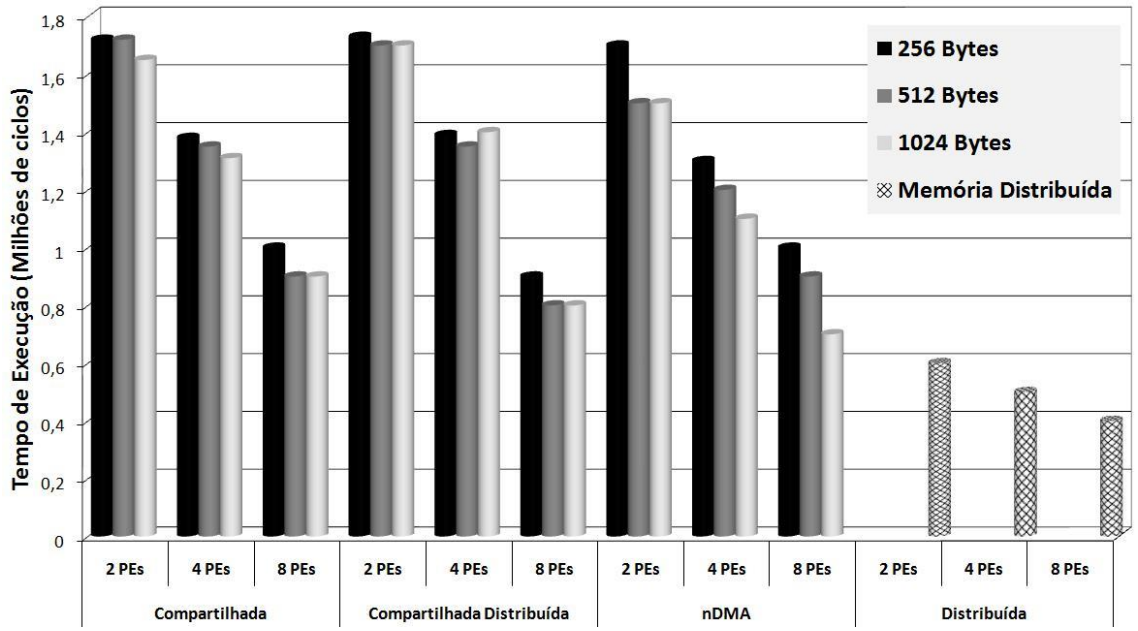


Figura 4.5: Performance na aplicação JPEG.

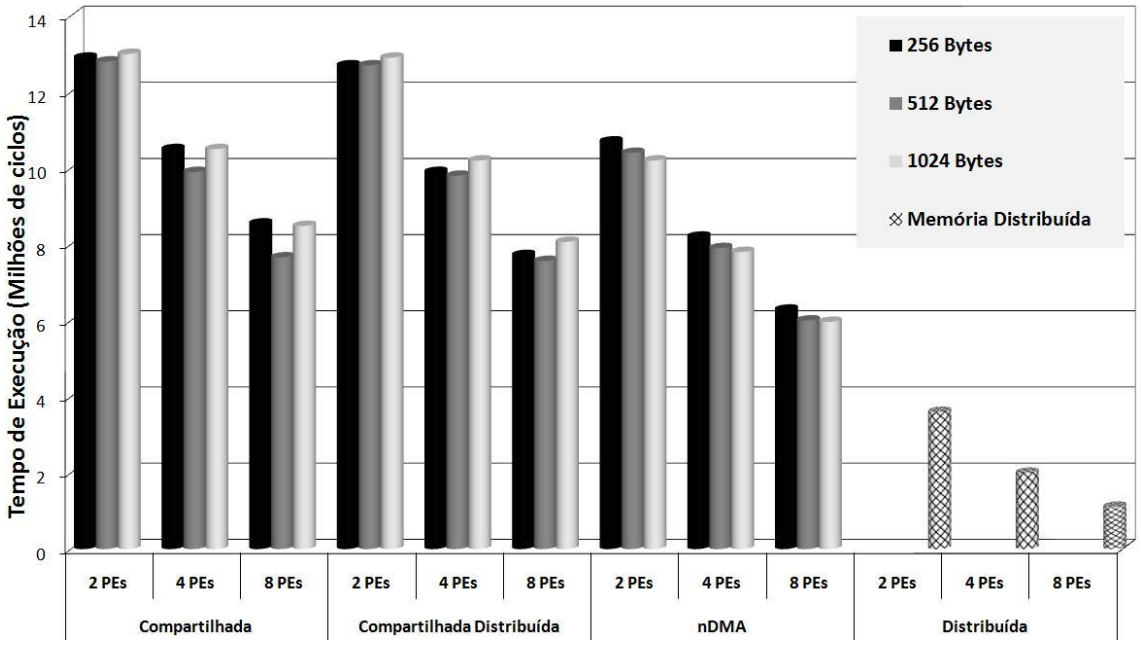


Figura 4.6. Performance na aplicação Mergesort.

A Tabela 4.5 abaixo apresenta os valores médios para cada aplicação em cada modelo de memória. Estes valores são uma média dos resultados para 2, 4 e 8 processadores para os valores das maiores caches (normalizando o resultado de acordo com os melhores resultados de cada modelo de memória). Estes resultados mostram que no caso da Estimação de Movimento (uma aplicação com grande carga de comunicação) a memória distribuída obteve os piores resultados.

Tabela 4.5. Valores médios de performance.

Modelo de Memória	**Valor médio para caches maiores (Em milhões de ciclos)**			
	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória Distribuída	8,83	2,55	0,5	2,2
Memória Compartilhada	4,36	6,36	1,28	10,6
Memória Compartilhada Distribuída	4,36	6,26	1,3	10,4
Memória nDMA	5,56	9,2	1,1	8

4.2.2 Resultados quanto ao consumo de energia

No que diz respeito à energia, as Figuras 4.7, 4.8 e 4.9 e 4.10 apresentam o consumo para as quatro aplicações. Para organização de memória existem cinco colunas representando o tamanho das caches (de 64 a 1024 bytes nas simulações da Estimação de movimento, JPEG e Mergesort e de 256 a 4096 bytes para a Multiplicação de matrizes).

Analisando a diferença de consumo de energia entre a solução nDMA e as outras duas organizações de memória compartilhada, é possível ver que o consumo de energia da memória em si é muito diferente na Estimação de movimento para a nDMA se comparada com os outros dois modelos com memória compartilhada. Além disso, a cache não apresenta um consumo de energia significativo para nenhum dos três modelos de memória compartilhada. Este comportamento sugere que a cache não está fazendo um bom uso do princípio da localidade para esta aplicação em particular e, por isso, o processador acessa pouco a cache e muito a memória. Por consequência o custo energético para a leitura na memória nDMA é *NP* vezes maior (onde *NP* é o número de processadores no sistema) do que o custo nas memórias compartilhada e compartilhada distribuída.

Quanto ao codificador JPEG, o grande diferencial entre a abordagem nDMA e as outras duas memórias compartilhadas é a energia gasta pela NoC. Por causa do modelo de comunicação da memória nDMA, o consumo de energia da NoC tende a ser menor devido à baixa quantidade de informação trafegando pela rede conforme o número de processadores aumenta. Entretanto, se o número de processadores é baixo (dois, por exemplo), o modelo nDMA demonstra piores resultados. Nos modelos de memória compartilhada e compartilhada distribuída esta comunicação é implícita no acesso à memória, e, também devido ao *overhead* causado pela manutenção da coerência de cache, a NoC aparece como um consumidor de energia significativo.

Como retratado nos resultados, no modelo de memória distribuída o consumo do processador é o maior responsável pelo consumo energético do sistema como um todo. O consumo da NoC e por vezes também da memória é negligenciável neste ambiente. A energia da memória é mais significativa na simulação da Estimação de movimento devido a grande quantidade de dados a serem manipulados.

Nos resultados de JPEG e Estimação de movimento, é possível ver um padrão similar, exceto pela energia da cache, a qual é negligenciável. Nos resultados da Estimação de movimento, a memória distribuída apresenta piores resultados do que no caso do JPEG e Multiplicação de matrizes. Isso é devido aos resultados de performance discutidos anteriormente. No caso da aplicação Mergesort, no modelo de memória distribuída, percebe-se que o consumo de energia diminui linearmente conforme o número de processadores aumenta. Além disso, nesta aplicação, a energia consumida pelo acesso a memória é muito menor do que o consumo do processador. Isso se deve à natureza da aplicação que tende a realizar várias comparações para um mesmo conjunto de dados. Nesta aplicação a memória nDMA apresenta uma eficiência energética maior do que as memórias compartilhada e compartilhada distribuída de forma mais visível do que nas outras aplicações.

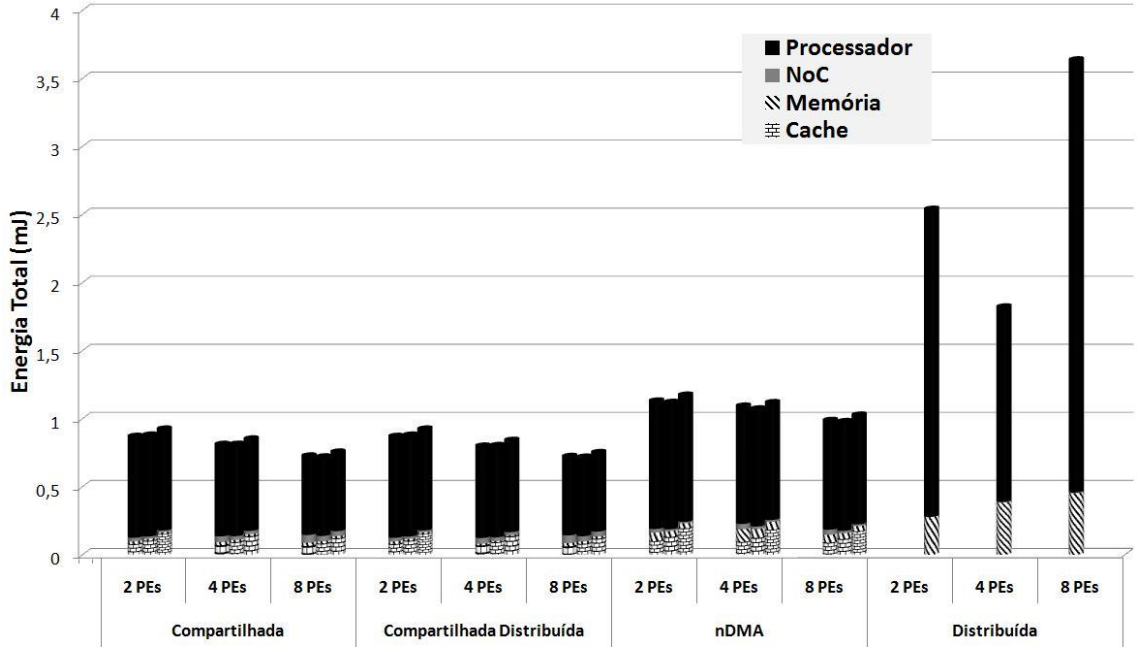


Figura 4.7: Consumo de energia na aplicação de Estimação de movimento.

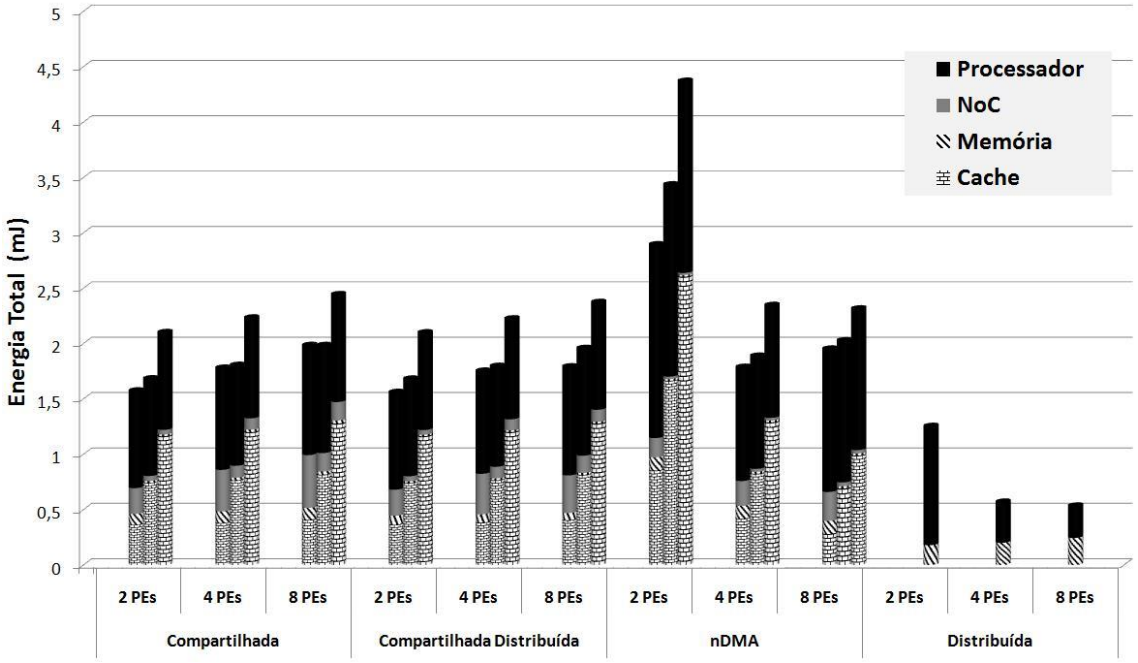


Figura 4.8: Consumo de energia na aplicação de Multiplicação de matrizes.

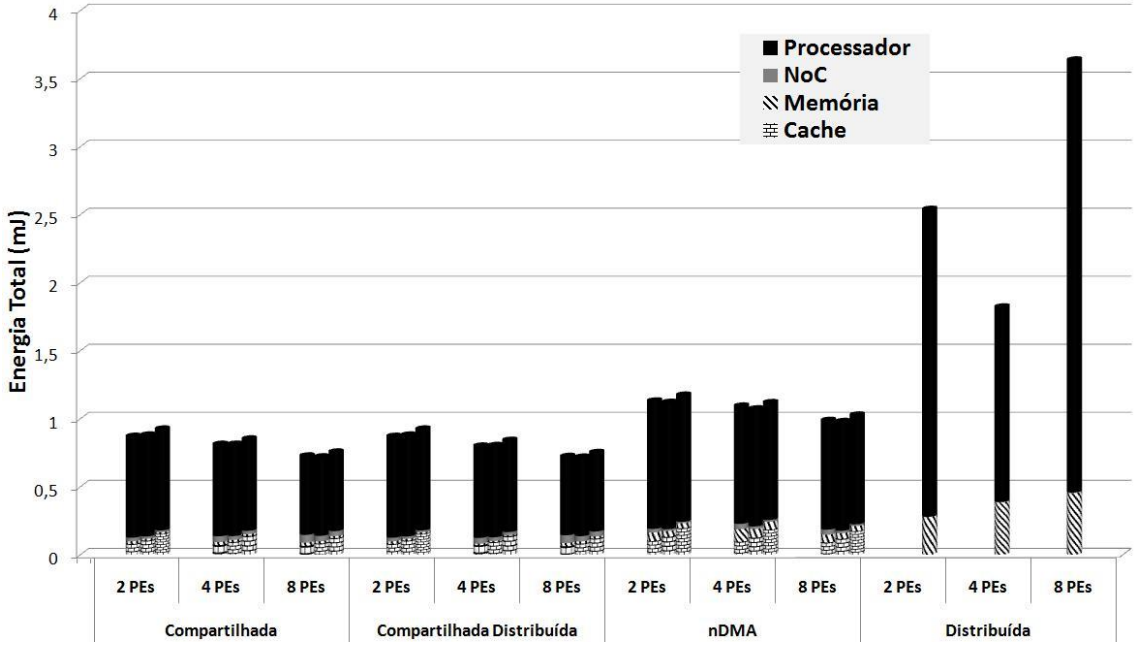


Figura 4.9: Consumo de energia na aplicação JPEG.

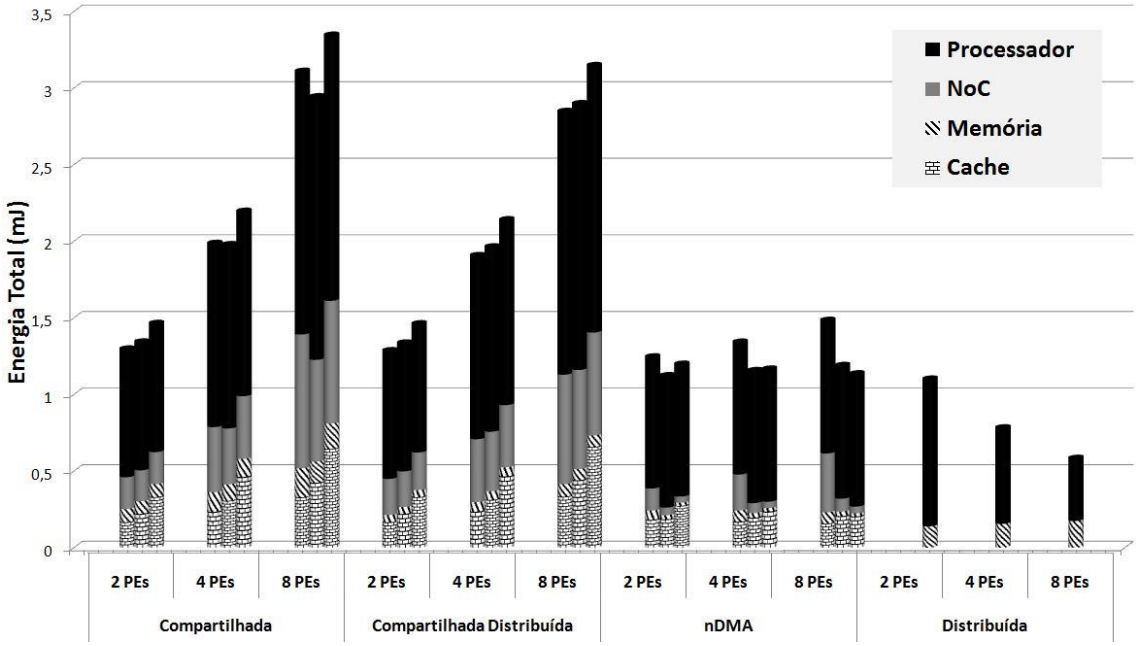


Figura 4.10. Consumo de energia na aplicação Mergesort.

Levando em consideração que estes experimentos utilizam um processador multiciclo, a significância da energia da cache poderia ser maior quando considerado um processador com pipeline. Imaginando um pipeline de cinco estágios, o número de ciclos destas execuções poderiam ser 4 vezes menor e o consumo de energia diminuiria dramaticamente. Neste caso, a energia do processador diminuiria cerca de 80%. Uma vez que o processador representa entre 45% e 65% de toda a energia do sistema para memórias compartilhadas, com um processador com pipeline a energia de todo o sistema cairia entre 36% e 52%. Para este cenário, a energia consumida pela cache seria a maior responsável pelo consumo de energia como um todo na simulação de

Multiplicação de matrizes, enquanto que nos casos do JPEG e Estimação de movimento a NoC seria o componente chave considerando energia.

A Tabela 4.6 apresenta os valores médios de consumo de energia para as aplicações utilizadas nos experimentos. Seguindo o mesmo padrão da Tabela 4.5, esta tabela mostra que, para a aplicação Estimação de Movimento, a memória distribuída apresenta piores resultados e, conforme a carga de comunicação da aplicação diminui (até o caso da aplicação JPEG), o desempenho deste modelo de memória é melhor. O modelo nDMA apresenta resultados melhores do que a memória distribuída em casos de grande carga de comunicação. Entretanto, o resultado das memórias compartilhada e compartilhada distribuída é melhor do que os da nDMA nesses mesmos casos. É importante resultar que, devido a um numero limitado de aplicações nos experimentos, estes resultados apenas sugerem esta co-relação entre a carga de comunicação da aplicacao e os modelos de memória apresentados. Em trabalhos futuros pretende-se investigar esta relação fazendo uso de mais aplicações com maiores cargas de comunicação

Tabela 4.6. Valores médios de consumo de energia.

Modelo de Memória	**Valor médio para caches maiores (mJ)**			
	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória Distribuída	2666	785	103	813
Memória Compartilhada	844	2259	180	2335
Memória Compartilhada Distribuída	839	2232	179	2250
Memória nDMA	1528	3010	166	1166

4.2.3 Resultados quanto ao tráfego na rede

O tráfego na rede é apresentado nas Figuras 4.11, 4.12, 4.13 e 4.14 para as aplicações Estimação de Movimento, Multiplicação de Matrizes, JPEG e Mergesort, respectivamente. Também neste caso existem cinco colunas por grupo, cada uma representando um tamanho de cache.

Considerando o tráfego na NoC, é possível ver que o padrão está relacionado ao tamanho da cache. Conforme o tamanho da cache aumenta, o tráfego diminui. As memórias compartilhada e compartilhada distribuída parecem ter tráfegos na rede muito similares. Contudo, o modelo nDMA apresenta menor tráfego na NoC do que as duas para a maioria dos casos. Para a aplicação JPEG esta melhora tem um valor médio de 45% e no melhor 88%. Este tráfego menor é um resultado direto do modelo de comunicação adotado para a memória nDMA. Além disso, os baixos resultados de tráfego para a organização nDMA mostram que os *write-backs* forçados necessários para a coerência dos dados transferidos na comunicação não representam um *overhead* significativo.

O resultado de maior destaque nos gráficos é o baixo tráfego gerado pelo modelo de memória distribuída se comparado com os outros. Porém, mais uma vez este resultado é dependente da carga de comunicação da aplicação. A coerência de cache e

principalmente o acesso à memória são os principais fatores desta diferença. Por outro lado, o único tráfego gerado no modelo de memória distribuída é com o propósito de realizar a comunicação entre os processadores e não para acesso a dados já que estes encontram-se em memórias privadas locais.

Estes resultados de tráfego são muito importantes para medir o uso da rede em tais modelos de memória. Para um ambiente de memória distribuída, o paralelismo na comunicação de uma NoC pode não levar a qualquer tipo de melhora no desempenho, dependendo do modelo de comunicação adotado. Por outro lado, como as memórias compartilhadas frequentemente precisam acessar memórias localizadas em outros nós, a eficiência do mecanismo de comunicação para realizar esta operação impacta diretamente nos resultados de performance e consumo de energia.

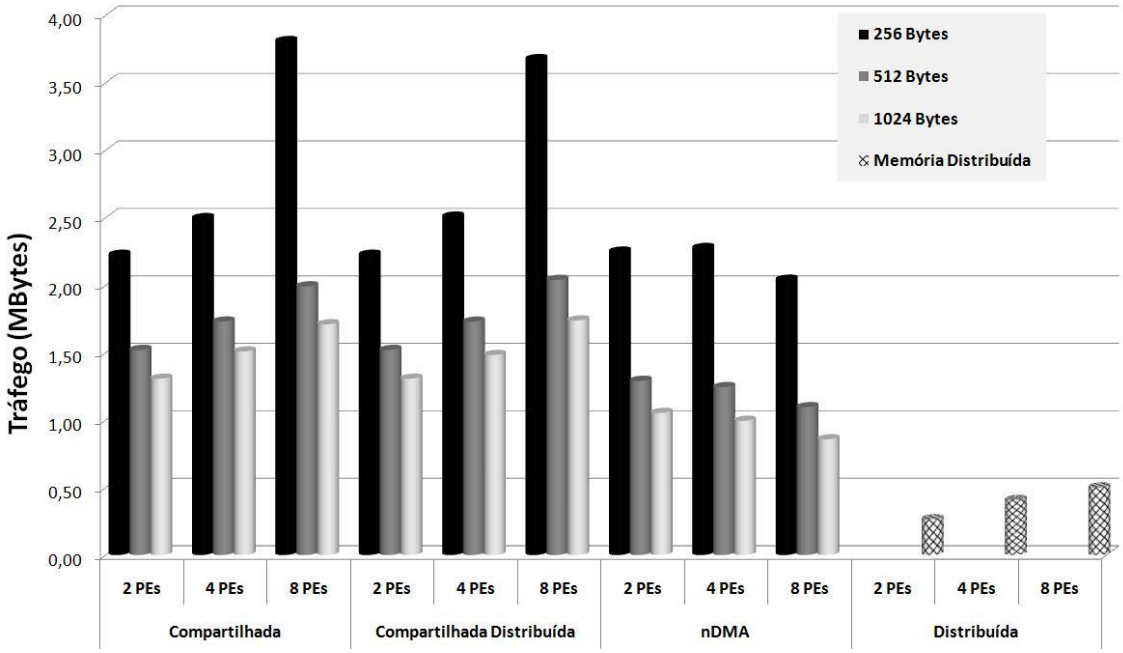


Figura 4.11: Tráfego na NoC na aplicação de Estimação de movimento.

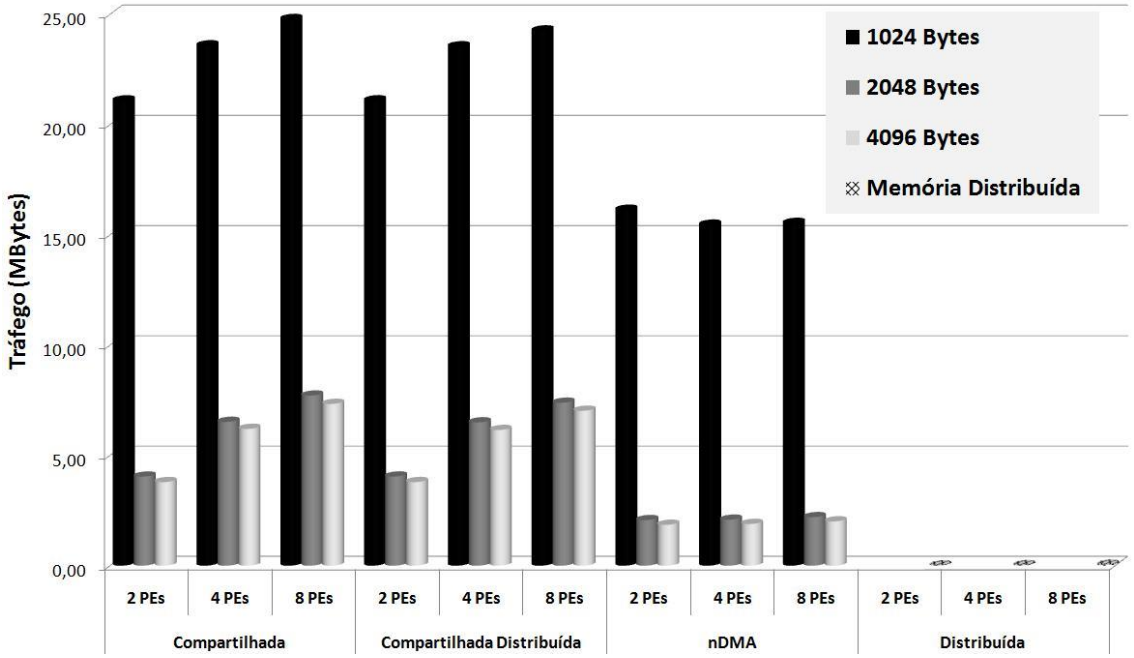


Figura 4.12: Tráfego na NoC na aplicação de Multiplicação de matrizes.

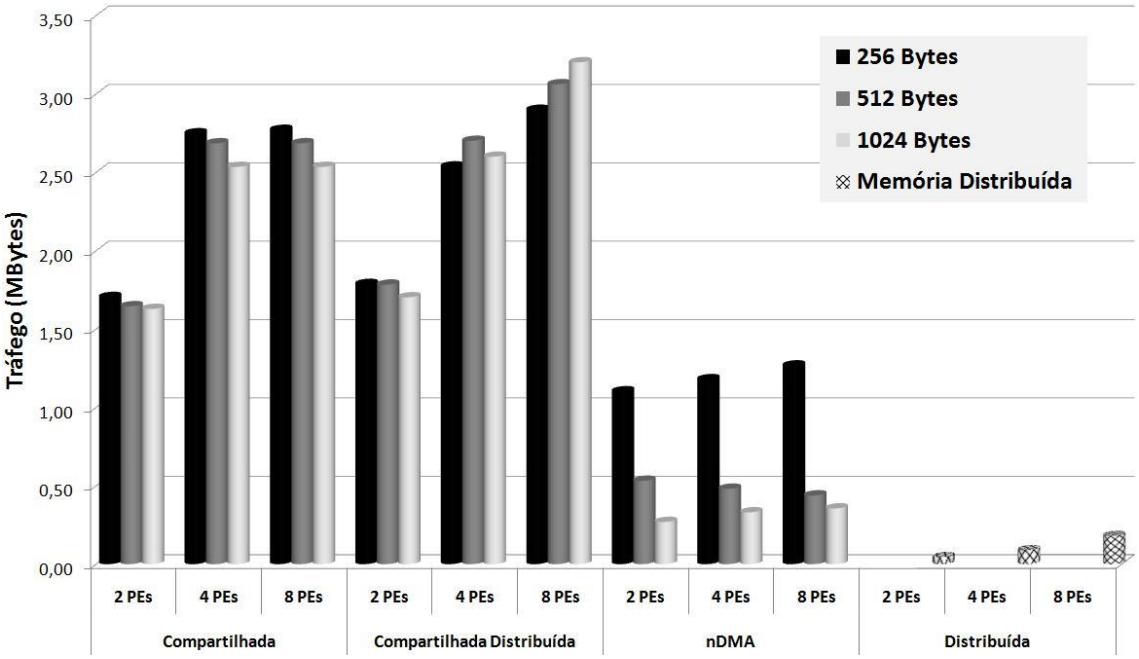


Figura 4.13: Tráfego na NoC na aplicação JPEG.

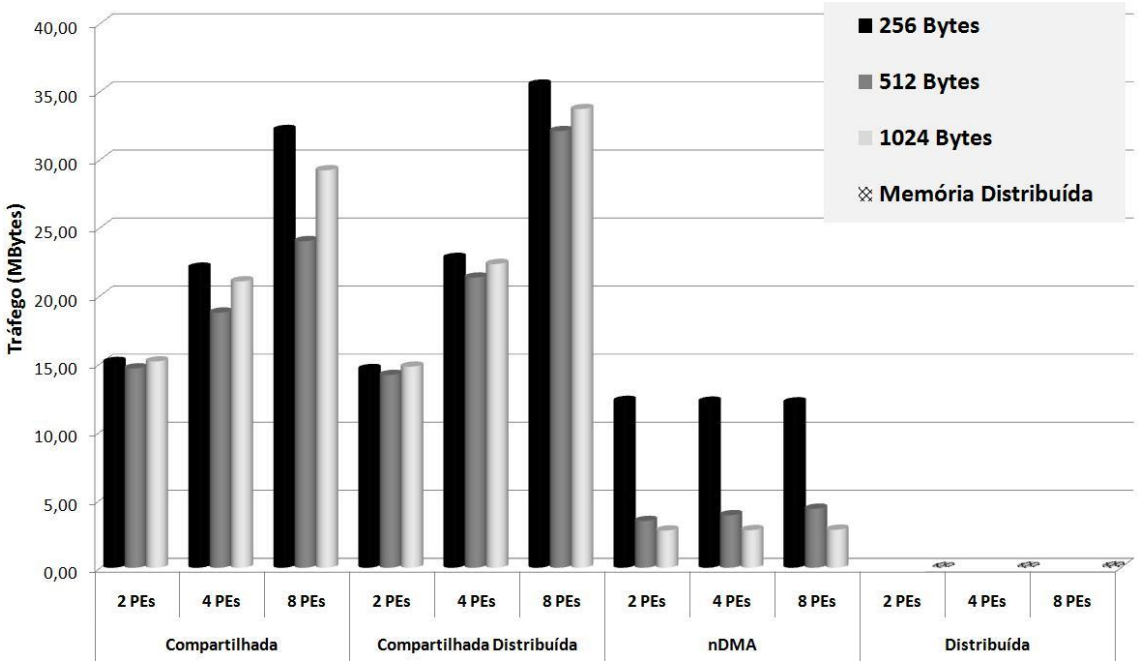


Figura 4.14. Tráfego na NoC na aplicação Mergesort.

A Tabela 4.7 apresenta os valores médios de tráfego na rede sumarizando os valores apresentados nesta seção. Como os resultados apresentados anteriormente mostram, o modelo de memória distribuída apresenta um baixo tráfego gerado na rede. Entretanto, em situações com grande carga de comunicação, este tráfego gerado tende a aumentar. Como previsto o modelo nDMA apresenta resultados de tráfego mais baixos do que os modelos de memória compartilhada, entretanto não demonstra melhores resultados do que o modelo de memória distribuída.

Tabela 4.7. Valores médios de tráfego na NoC.

Modelo de Memória	**Valor médio para caches maiores (MBytes)**			
	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória Distribuída	0,39	0,02	0,1	0,06
Memória Compartilhada	1,51	5,77	2,23	21,28
Memória Compartilhada Distribuída	1,51	5,65	2,5	23,6
Memória nDMA	0,9	1,91	1,1	2,76

4.2.4 Sumário dos resultados

A Tabela 4.8 sumariza os resultados apresentados nesta seção. De maneira geral, a memória distribuída apresentou melhores resultados em todos os aspectos apresentados. Entretanto quando a carga de comunicação em uma aplicação é grande, os experimentos sugerem que este modelo de memória apresenta piores resultados. Os modelos de memória compartilhada e compartilhada distribuída apresentaram resultados piores do que o modelo de memória distribuída e melhores do que o modelo nDMA, com poucas exceções. No caso do tráfego gerado na rede o modelo de memória nDMA apresentou melhores resultados do que os modelos de memória compartilhada e compartilhada

distribuída e os experimentos sugerem que, em situações de grande carga de comunicação, este modelo apresenta melhores resultados do que os outros modelos.

Tabela 4.8. Resultados experimentais

	Performance	**Consumo de energia**	**Tráfego na rede**
Memória distribuída	Melhores resultados do que os outros modelos, em quase todos os casos. Exceção é feita quando a carga de comunicação aumenta.	Melhores resultados como conseqüência direta da performance.	Melhores resultados quando a carga de comunicação é pequena.
Memória compartilhada	Resultados similares às outras memórias compartilhadas, porém desempenho pouco pior que a memória compartilhada distribuída.	Apresenta resultados intermediários entre a memória compartilhada distribuída e a nDMA. A taxa de *miss* da cache determina se o maior consumo é da cache ou NoC.	Piores resultados de tráfego.
Memória compartilhada distribuída	Melhores resultados das memórias compartilhadas.	Mostra melhores resultados que nDMA e memória compartilhada. Segue o mesmo padrão da memória compartilhada quanto à taxa de *misses*.	Resultados bastante similares aos da memória compartilhada.
nDMA	Desempenho pouco pior que o das outras memórias.	O consumo de energia da memória apresenta-se como principal componente no consumo de energia. Neste caso, o desempenho da cache é um fator decisivo.	Melhores resultados quando a carga de comunicação é grande.

4.3 Experimentos com simulação de alta latência

Esta seção apresenta os resultados considerando performance e consumo de energia dinâmica para as quatro aplicações descritas anteriormente numa situação de alta latência da NoC. É importante notar que os resultados são considerados analisando o quanto a performance diminui e o consumo de energia dinâmica aumenta quando a carga de tráfego aumenta gerando assim uma maior latência. Comparações entre valores absolutos do número de ciclos de execução e consumo de energia das diferentes organizações de memória não são considerados, pois dizem respeito ao desempenho intrínseco de cada modelo de memória para cada aplicação. Este tipo de análise já foi feito anteriormente como descrito na Seção 4.2. Em vez disso, o objetivo aqui é investigar quão tolerantes são os modelos de memória considerando um ambiente com alta latência de comunicação. Assim, todos os experimentos medem quão grande é a redução de performance e o aumento do consumo de energia em todos os modelos de memória quando a carga sintética injetada na rede na rede sobe de 10% para 20%.

A Tabela 4.9 mostra o aumento do tempo de execução das aplicações em situações sem nenhum tráfego sintético e com 10% e 20% de tráfego. Estes resultados dizem respeito a uma média dos tempos de execução para o caso de 8 processadores. Em alguns casos, onde o impacto do tráfego sintético é maior, o aumento chega a 100%

(casos de memória compartilhada e compartilhada distribuída). Isso mostra o quão grande é a latência gerada pela injeção de somente 10% e 20% de tráfego sintético. Portanto, o uso de 10% e 20% de tráfego sintético demonstra ser suficiente para realizar esta análise de latência sobre os modelos de memória. Os resultados completos destes experimentos serão apresentados a seguir.

Tabela 4.9. Tempo médio de execução em situações sem nenhum tráfego sintético e com 10% e 20% de tráfego.

Aplicação	**Estimação de Movimento**			**Multiplicação de Matrizes**			**JPEG**			**Mergesort**		
Memória Distribuída	9,9	10,4	10,6	1,9	2	2,4	0,4	0,5	0,6	1,1	1,2	1,4
Memória Compartilhada	1,43	3,2	6,3	4,2	20	40	1	3,9	12,1	8,2	25	51,4
Memória Compartilhada Distribuída	1,43	3,9	6,7	3,8	19,8	40	0,9	9,1	15,3	7,8	19	67,7
Memória nDMA	2,2	3,1	5,1	4,7	14,4	27	0,8	1,7	2,8	6,1	8,2	11,9

4.3.1 Resultados quanto à performance

Analisando o gráfico da Figura 4.15 percebe-se que o aumento do tráfego levando a um aumento da latência, quase não interfere na performance do modelo de memória distribuída para a aplicação de Estimação de Movimento. Este comportamento é uma conseqüência direta do baixo uso da NoC por este modelo de memória e por isso a baixa interferência em situações de maior latência de comunicação. A memória distribuída apresenta 2% de redução de performance, em média, quando a carga de tráfego aumenta de 10% para 20%.

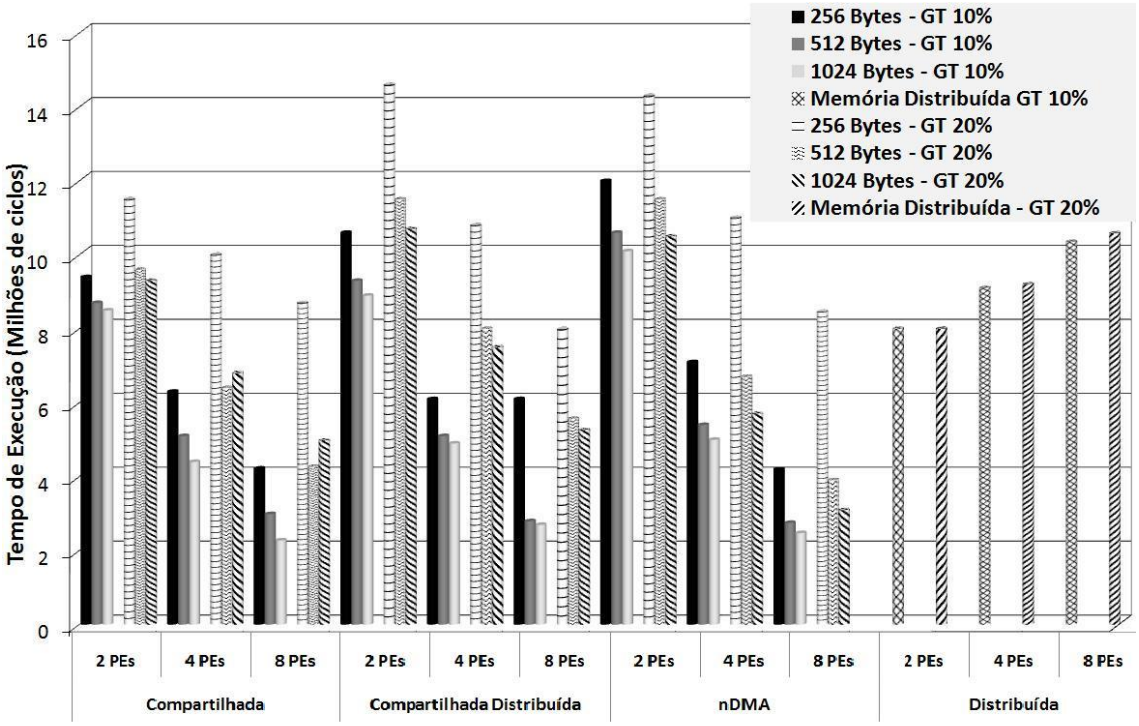


Figura 4.15. Resultados de performance para a Estimação de Movimento.

Por outro lado, as memórias compartilhada e compartilhada distribuída apresentam grande sensibilidade ao aumento da latência da NoC. Em alguns casos, o impacto gerado pelo uso de tráfego sintético chega a 100% de aumento quando a carga aumenta de 10% para 20% quando utilizando 8 processadores com memória compartilhada. Em média, a performance diminui 29% e 33% para as memórias compartilhada e compartilhada distribuída, respectivamente. A explicação para este comportamento está no fato de que memórias compartilhada e compartilhada distribuída possuem alta dependência da NoC, não apenas para fazer acesso aos dados da aplicação, mas também devido ao uso da memória compartilhada para comunicação. Além disso, o uso de uma solução de coerência de cache leva a uma maior taxa de *misses* devido ao uso de procedimentos de invalidação.

No caso da memória nDMA, o tráfego sintético apresenta uma menor influência, se comparado com os resultados apresentados pelas memórias compartilhada e compartilhada distribuída, especialmente quando o número de processadores aumenta. Os resultados mostram que a performance diminui 22%, em média, neste modelo de memória.

A razão para uma maior tolerância a maiores latências no modelo de memória nDMA é o fato de que seu modelo de comunicação trabalha enviando apenas pequenas mensagens de controle para o nó da memória. Ademais, a taxa de *misses* das caches neste caso é menor do que nos outros modelos de memória já que não se faz necessária a adoção de um mecanismo de coerência de cache e assim não existem mensagens de invalidação.

No caso da aplicação JPEG, os resultados de performance apresentados na Figura 4.16 mostram um comportamento bastante similar ao da aplicação Estimação de Movimento. Uma baixa influência do aumento da latência aparece nos resultados do modelo de memória distribuída e, em média, a performance é reduzida em 19%. Quanto

às memórias compartilhada e compartilhada distribuída, o aumento da latência causa uma redução média de performance de 44% e 55%, respectivamente. A memória nDMA, por sua vez, é menos influenciada pelo tráfego sintético e sua performance diminui 33% em média.

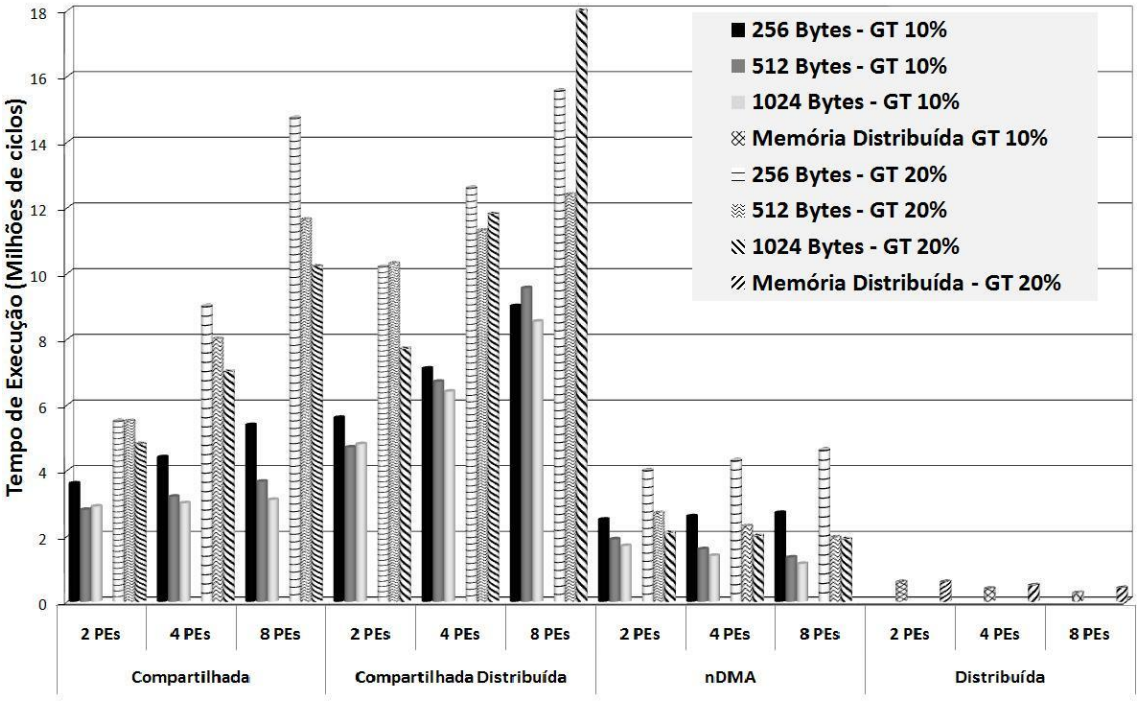


Figura 4.16. Resultados de performance para o JPEG.

Estes resultados mostram que, na aplicação JPEG, um aumento da latência da NoC causa maior impacto do que na aplicação Estimação de Movimento, considerando todos os modelos de memória. Levando em conta que a Estimação de Movimento é uma aplicação que demanda uma carga de comunicação muito maior do que a aplicação JPEG (como apresentado no gráfico da Figura 4.2), estes experimentos sugerem que, uma vez que a quantidade de comunicação gerada pela Estimação de Movimento já é muito grande, o aumento da latência causa um menor impacto. Este tipo de comportamento afeta todos os modelos de memória, porém em níveis diferentes.

Assim como nos experimentos com as aplicações JPEG e Estimação de Movimento, a Multiplicação de Matrizes (Figura 4.17) apresenta uma alta tolerância com relação ao aumento da latência no caso do modelo de memória distribuída, onde a performance diminui 4% em média.

Da mesma maneira como nas outras aplicações, as memórias compartilhada e compartilhada distribuída apresentaram baixa tolerância à situação de maior latência e suas performances diminuíram, em média, 40% e 48%, respectivamente.

Para a memória nDMA os resultados foram melhores do que os das memórias compartilhadas e pior do que o da memória distribuída. A performance é reduzida em 33%, em média neste modelo.

Para a aplicação Mergesort, o gráfico da Figura 4.18 mostra que as memórias compartilhada e compartilhada distribuída apresentam resultados semelhantes (40% e 49% de perda de desempenho em média) enquanto que a memória nDMA apresenta

resultados melhores quando comparados os desempenhos no caso de 10% e 20% de carga na rede. Em média a memória nDMA perde 33% de desempenho nesta situação. Novamente, a memória distribuída apresentou melhores resultados uma vez que o aumento da latência na rede causou apenas 6% de perda de desempenho, em média.

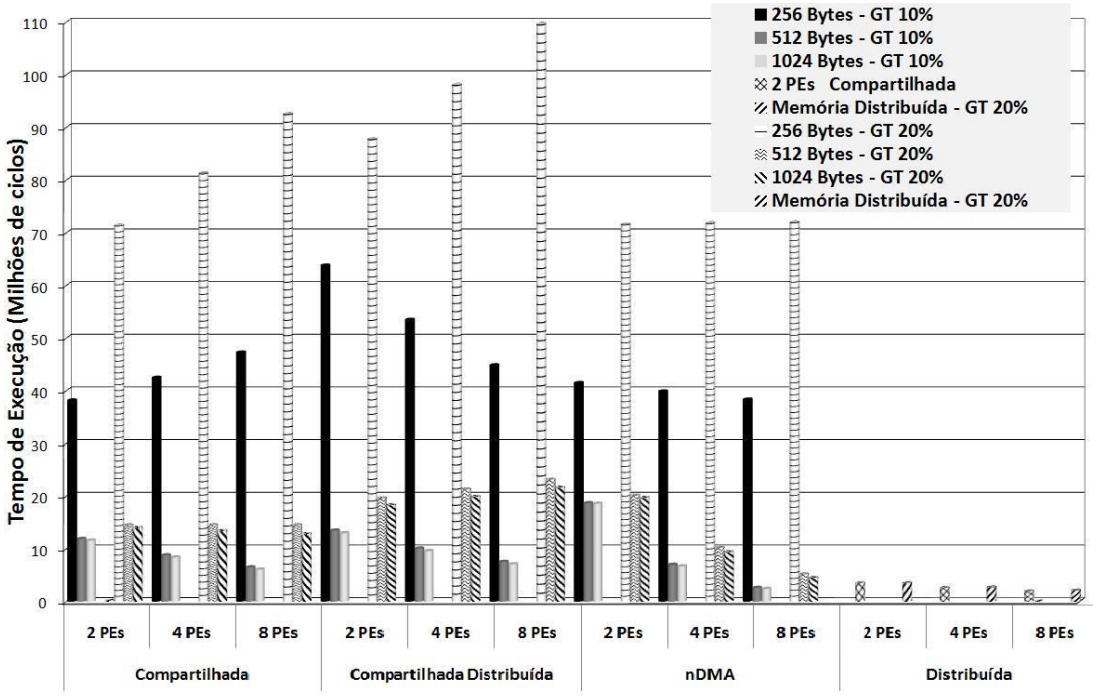


Figura 4.17. Resultados de performance para a Multiplicação de Matrizes.

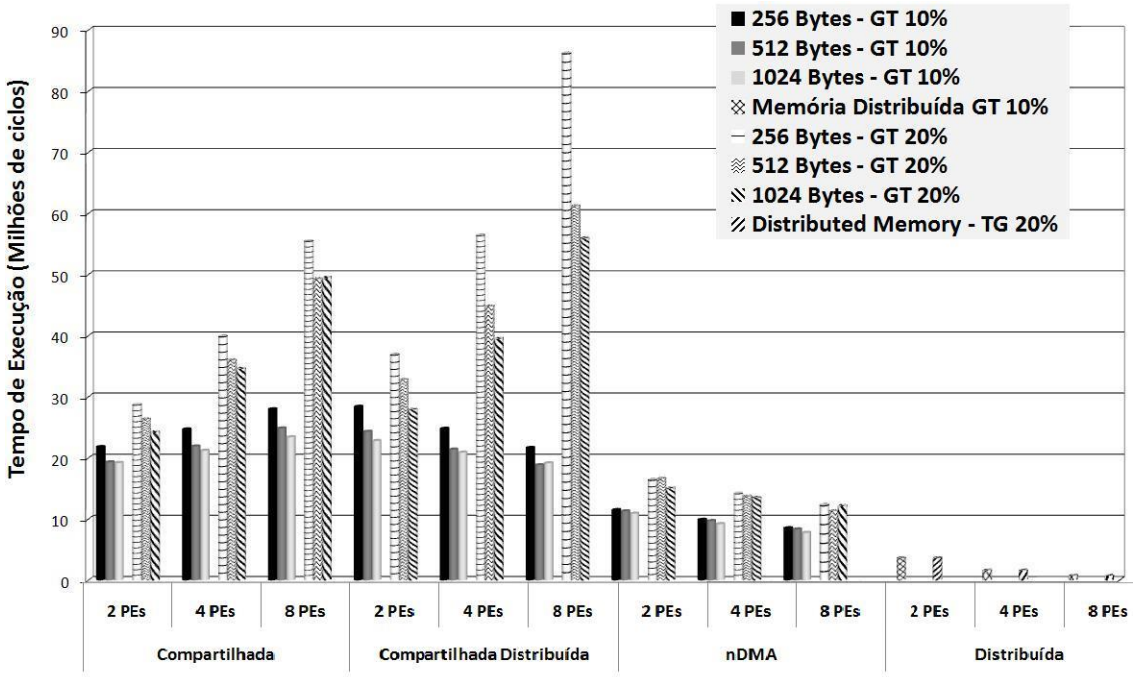


Figura 4.18. Resultados de performance para o Mergesort.

Em um primeiro momento, os experimentos sugerem que a memória distribuída é o modelo de memória mais robusto em um cenário de NoC com altas latências de comunicação. Embora o modelo nDMA apresente resultados consistentes, a necessidade de realizar um acesso a uma memória remota aparenta ser um gargalo para este modelo

em situações de alta latência. Memórias compartilhada e compartilhada distribuída parecem ser modelos mais vulneráveis a altas latências uma vez que eles apresentam os piores resultados para todas as aplicações. Este resultado se deve, em maior parte, ao frequente acesso à memória e o uso recorrente de mensagens de controle de coerência de cache.

Apesar de estes resultados isolados indicarem que o modelo de memória distribuída apresenta uma alta tolerância a altas latências na NoC, é importante observar estes mesmos resultados considerando a carga de comunicação das aplicações.

Tabela 4.10. Impacto do aumento de latência na performance.

Aplicação	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória Distribuída	2%	4%	19%	6%
Memória Compartilhada	29%	40%	44%	40%
Memória Compartilhada Distribuída	33%	48%	55%	49%
Memória nDMA	22%	33%	33%	33%

Nota: Os valores na tabela indicam o quanto a performance foi reduzida quando a carga na rede aumentou de 10% para 20%.

Considerando os resultados resumidos na Tabela 4.2, é possível perceber como a maior latência impacta na performance. Embora o modelo de memória distribuída apresente os melhores resultados individuais quando comparados com os outros modelos de memória, é muito claro o quanto a latência impacta na performance conforme a carga de comunicação da aplicação sendo executada aumenta. Na aplicação de estimação de movimento, a maior latência apenas afeta os resultados da memória distribuída em 2%. Entretanto quando uma aplicação com menor carga de comunicação é executada, como a aplicação JPEG, o impacto do aumento da latência chega a 19%. Os modelos de memória compartilhada e compartilhada distribuída aparentam sofrer menos devido a esta variação da carga de comunicação. De maneira similar, o modelo de memória nDMA sofre ainda menos já que varia de 22% para 33%.

Estes resultados levam à conclusão de que uma alta latência da NoC afeta mais o modelo de memória distribuída do que os outros modelos conforme a carga de comunicação da aplicação diminui. Além disso, o modelo nDMA apresentou resultados com menor variação conforme esta carga de comunicação diminui.

4.3.2 Resultados quanto ao consumo de energia

Os resultados de consumo de energia apresentados nesta seção mostram que o aumento da latência da rede impacta com maior veemência a própria NoC. Outros componentes como a memória e a cache não são afetados devido ao fato de que, embora a alta latência ao acessar uma memória exista, o número de acessos a ela e às caches não muda e, por isso, o consumo de energia dinâmica não muda.

Para a Estimação de Movimento, o consumo de energia no modelo de memória distribuída aumenta em apenas 1% em média, quando a carga sintética injetada na rede aumenta de 10% para 20%. Isto é uma consequência direta da independência deste

modelo com relação à NoC (e, consequentemente, sua independência com relação à alta latência) e segue o mesmo padrão mostrado nos resultados de performance.

As memórias compartilhada e compartilhada distribuída apresentam um aumento médio de 25% e 44% no consumo de energia, respectivamente, enquanto que o consumo da memória nDMA aumenta apenas 15% em média. Estes resultados são consistentes com os resultados de performance destes modelos de memória. Os resultados completos de consumo de energia para os modelos de memória com a aplicação Estimação de Movimento são apresentados na Figura 4.19.

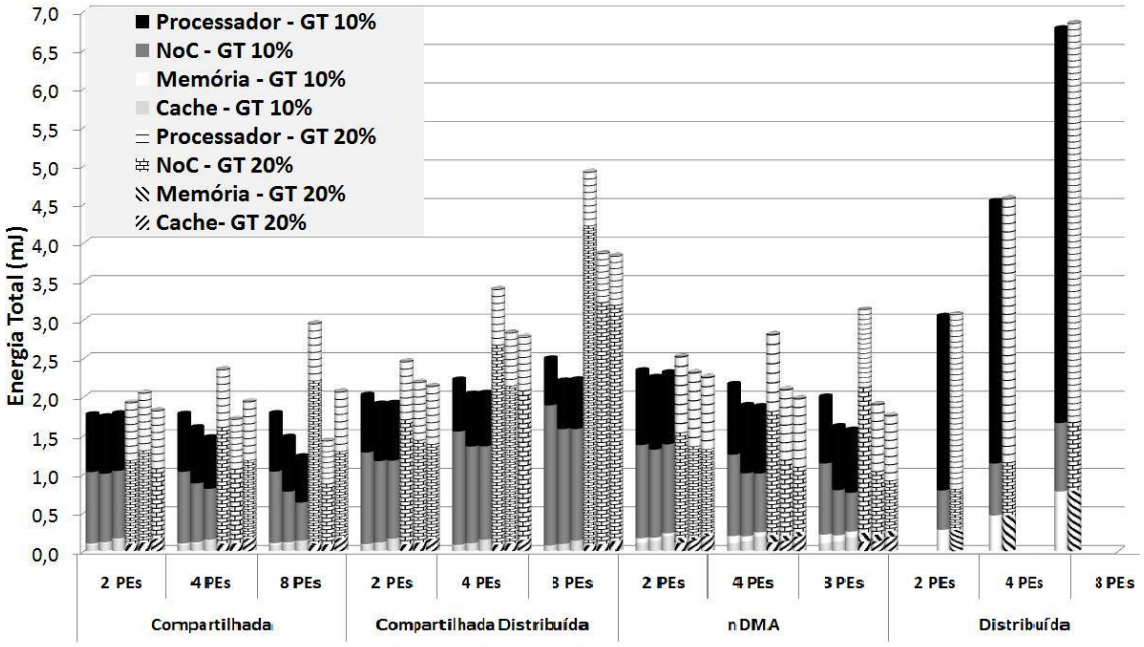


Figura 4.19. Resultados de consumo de energia para a Estimação de Movimento.

A Figura 4.20 apresenta os resultados da aplicação JPEG e também se mostram coerentes com os resultados de performance para esta mesma aplicação. A memória distribuída apresenta 32% de aumento de consumo de energia, enquanto que as memórias compartilhada, compartilhada distribuída e nDMA se mostram 67%, 79% e 24%, respectivamente, menos eficientes energeticamente.

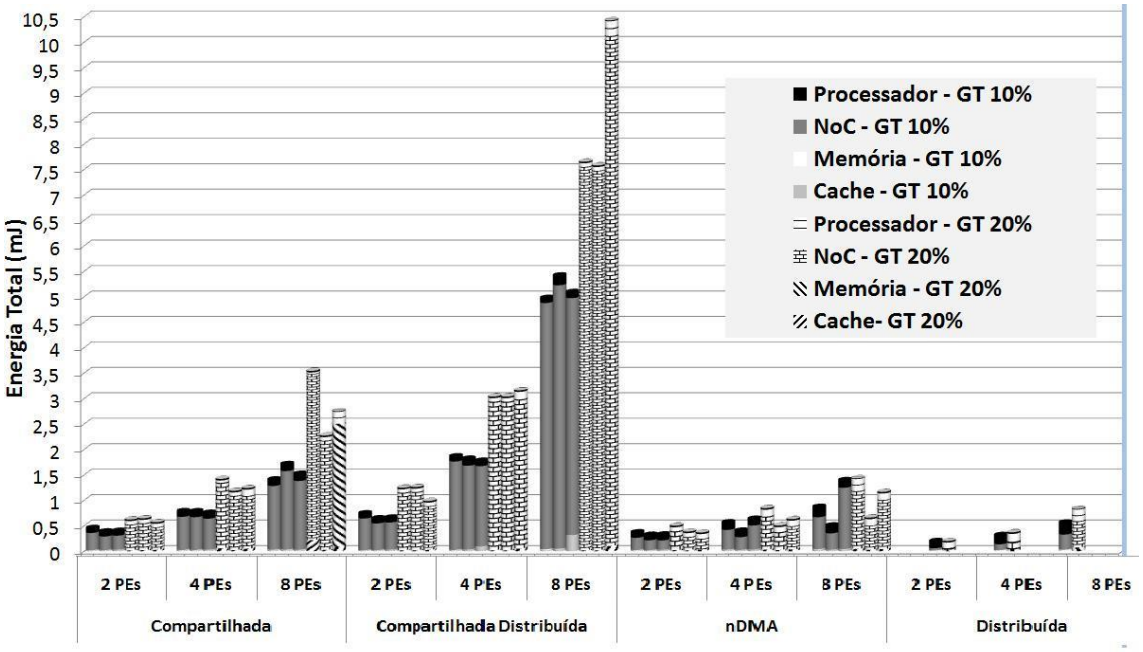


Figura 4.20. Resultados de consumo de energia para o JPEG.

Como ilustrado na Figura 4.21, o mesmo padrão também está presente nos resultados da Multiplicação de Matrizes. A memória distribuída demonstra 10% de aumento de consumo de energia em média. Por outro lado, as memórias compartilhada, compartilhada distribuída e nDMA apresentam um aumento médio de consumo de energia em 50%, 70% e 24%, respectivamente.

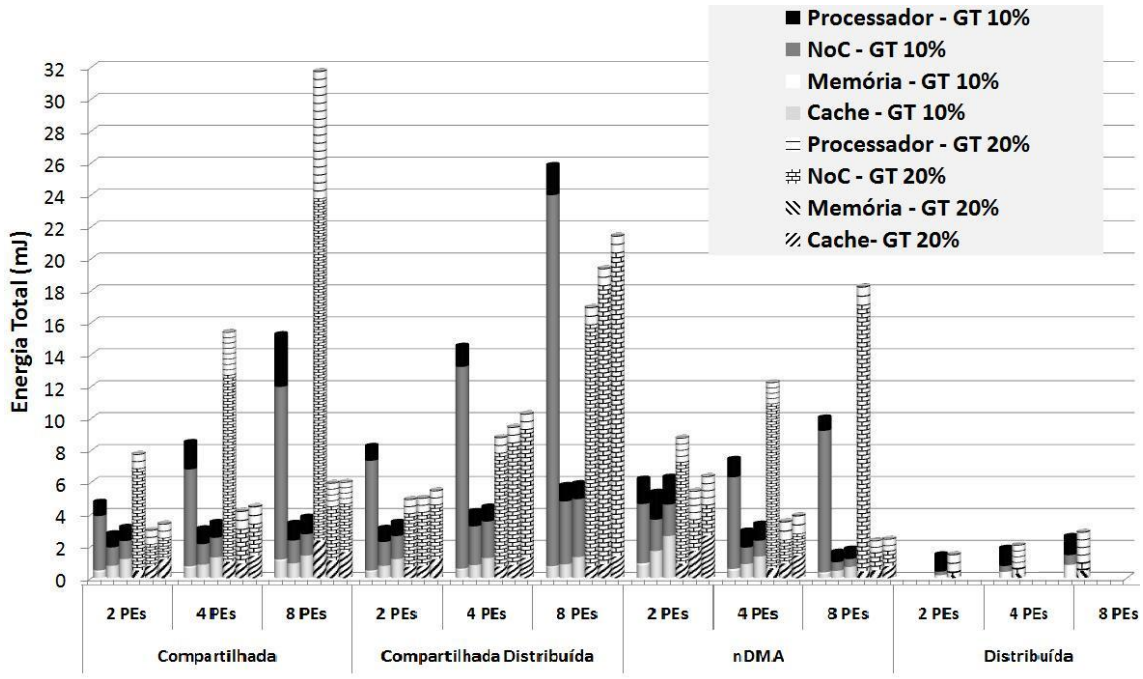


Figura 4.21. Resultados de consumo de energia para a Multiplicação de Matrizes.

Na Figura 4.22, os resultados de consumo de energia para a aplicação Mergesort mostram que o modelo de memória compartilhado distribuído sofreu um aumento expressivo no consumo de energia, em especial no caso com 8 processadores. Em média, os modelos de memória compartilhada e compartilhada distribuída tiveram um aumento no consumo de energia de 65% e 77%, respectivamente. O modelo de memória nDMA sofreu poucas variações e o aumento foi de 24%, em média. De maneira mais suave ainda, o modelo de memória distribuída sofreu pouco aumento no consumo de energia devido à alta latência. Seu aumento, em média, foi de 6%.

Novamente, o padrão apresentado nos resultados de performance aparece nos resultados de consumo de energia. Estes resultados sugerem que aplicações com necessidades de grandes cargas de comunicação aparentam ter uma redução menor no consumo de energia conforme a latência da rede aumenta.

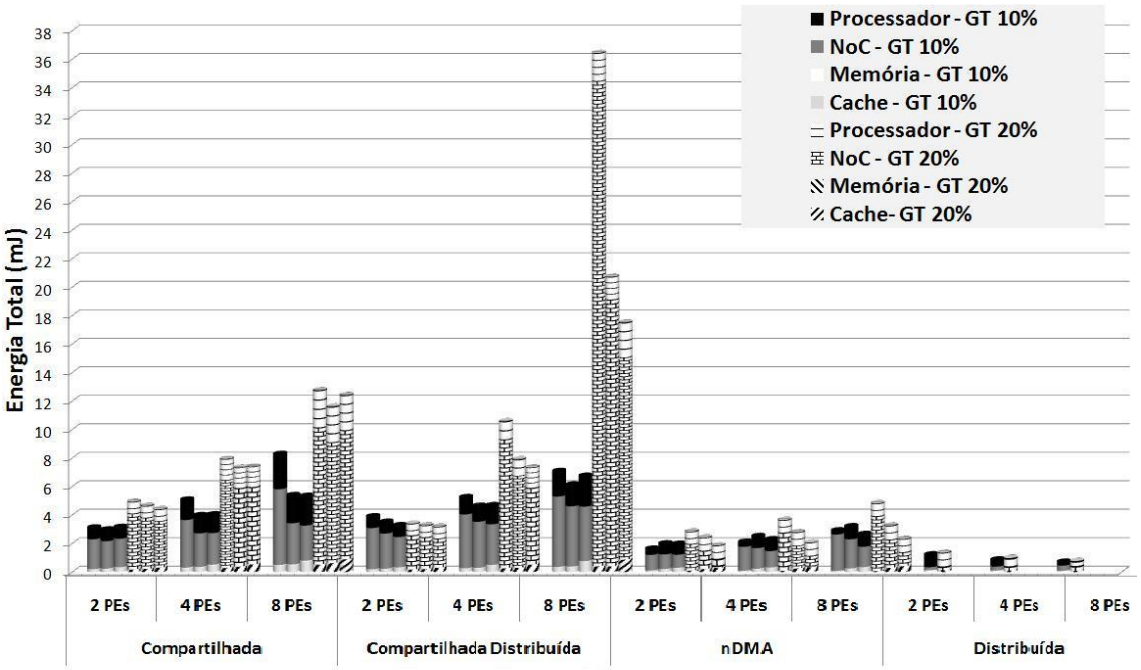


Figura 4.22. Resultados de consumo de energia para a Mergesort.

Assim como nos resultados de performance, é possível ver que uma alta latência impacta no consumo de energia dos modelos de memória de maneira diferente para cada um deles dependendo da carga de comunicação de cada aplicação. Os resultados da Tabela 4.3 mostram que o modelo de memória distribuída tem um maior aumento de consumo de energia em uma situação de alta latência quando a carga de comunicação da aplicação é baixa. Mais uma vez, as memórias compartilhada e compartilhada distribuída parecem sofrer mais do que qualquer outro modelo de memória enquanto que o modelo nDMA apresenta uma degradação suave no que diz respeito ao consumo de energia conforme sejam executadas aplicações com carga de comunicação mais baixa.

Tabela 4.11. Impacto da alta latência no consumo de energia dinâmica.

Aplicação	Estimação de Movimento	Multiplicação de Matrizes	JPEG	Mergesort
Memória Distribuída	1%	10%	32%	14%
Memória Compartilhada	25%	50%	67%	65%
Memória Compartilhada Distribuída	44%	70%	79%	77%
Memória nDMA	15%	24%	24%	24%

Nota: Os valores na tabela indicam o quanto a performance foi reduzida quando a carga na rede aumentou de 10% para 20%.

4.3.3 Sumário dos resultados quanto a latência

Nesta seção forma apresentados resultados relativos a experimentos com os diferentes modelos de memória em situações com diferentes níveis de latência na rede. Os resultados de performance a principio demonstram que o modelo de memória distribuída apresenta resultados melhores em todos os modelos. Entretanto uma análise mais profunda indica que os resultados do modelo distribuído tendem a piorar em aplicações com baixa carga de comunicação. Por outro lado, o modelo de memória nDMA apresenta maior tolerância a latências maiores conforme a carga de comunicação de uma determinada aplicação é alta.

Resultados similares são apresentados no que diz respeito ao consumo de energia. Nestes resultados percebe-se a grande vulnerabilidade dos modelos de memória compartilhada e compartilhada distribuída devido à grande dependência da rede não só no que diz respeito ao acesso aos dados residentes em uma memória remota como também no que diz respeito à comunicação utilizando variáveis compartilhadas.

4.4 Análise do impacto da migração de tarefas

Esta seção apresenta os resultados dos experimentos de migração de tarefas considerando performance e consumo de energia dinâmica. Estes experimentos têm por objetivo quantificar o custo do processo de migração de tarefas em cada um dos modelos de memória. Cada modelo de memória exige o envio de código e/ou dados e pilha dependendo do caso, como explicado anteriormente na Seção 3.7.1. Estes experimentos consideram o pior caso, que seria a criação das *N* tarefas em um nó, onde *N* representa o número de processadores para o qual aquela tarefa foi descrita, e essas tarefas são migradas para *N* nós da rede.

Cada coluna de cada modelo de memória representado nos gráficos representa, respectivamente, situações de migração para 2, 4 e 8 nós.

4.4.1 Resultados quanto à performance

De acordo com os resultados de performance apresentados nas Figuras 4.23, 4.24, 4.25 e 4.26, o modelo de memória distribuída apresenta piores resultados em todos os casos. Este resultado já era esperado como discutido na Seção 3.7 devido ao fato de que

neste modelo de memória existe a necessidade do envio do código da aplicação, da pilha de dados e também da memória de dados. Este último não precisa ser copiado nos casos da memória compartilhada e compartilhada distribuída e no caso da memória nDMA a cópia pode ser feita em paralelo com a migração do código e pilha devido ao fato de que todas as memórias do sistema estão presentes em um mesmo nó da rede o que também torna a cópia mais rápida.

Entretanto, estes não são os únicos fatores responsáveis pelo pior desempenho da memória distribuída. Outro fator responsável por este desempenho é o tamanho do código que no caso da memória distribuída é normalmente maior do que os outros modelos de memória devido ao fato de que a comunicação precisa ser completamente explicita, de modo que o programador precisa descrever a cópia dos dados para buffers de envio e leitura dos dados de buffers de recebimento bem como chamadas das próprias funções de envio e recebimento. Este procedimento ocorre em menor intensidade no caso do modelo de memória nDMA que, apesar de não necessitar do envio da memória de dados, trabalha com o envio explicito de mensagens de controle para comunicação entre os elementos de processamento. Por outro lado, os modelos de memória compartilhada e compartilhada distribuída apresentam os melhores resultados em todos os casos. Isto se deve não só ao fato de que somente é necessária a cópia do código da aplicação e da pilha de dados, mas também devido à simplicidade do código no modelo de memória compartilhada.

Apesar dos melhores resultados dos modelos de memória compartilhada e compartilhada distribuída, é importante notar que não foi levada em consideração, nestes experimentos, a migração do conteúdo das caches e por isso é de se esperar uma perda de desempenho durante os primeiros ciclos de execução após a migração nestes casos devido ao uma considerável quantidade de *misses* compulsórios da cache. Este caso não foi estudado uma vez que a migração ocorre no momento da criação da tarefa e por isso as caches praticamente não contêm informação relevante para aquela tarefa. Entretanto, se for levada em consideração uma migração durante a execução da tarefa este caso de *misses* compulsórios na cache ocorrerá. Esta é uma situação cujo resultado depende do ponto no tempo dentro da execução da tarefa no qual seja decidido efetuar a migração. Esta escolha é em função do algoritmo de alocação de tarefas e não foi considerado nestes experimentos que avaliam somente o mecanismo de migração de tarefas em si.

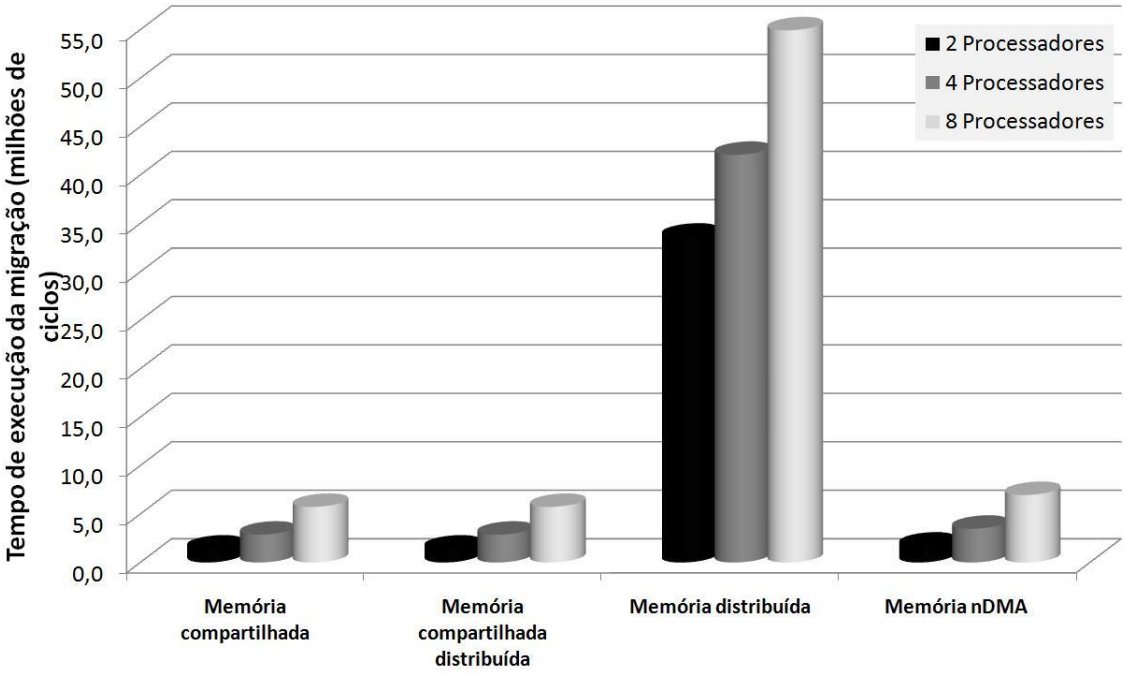


Figura 4.23. Performance da migração da Estimação de Movimento.

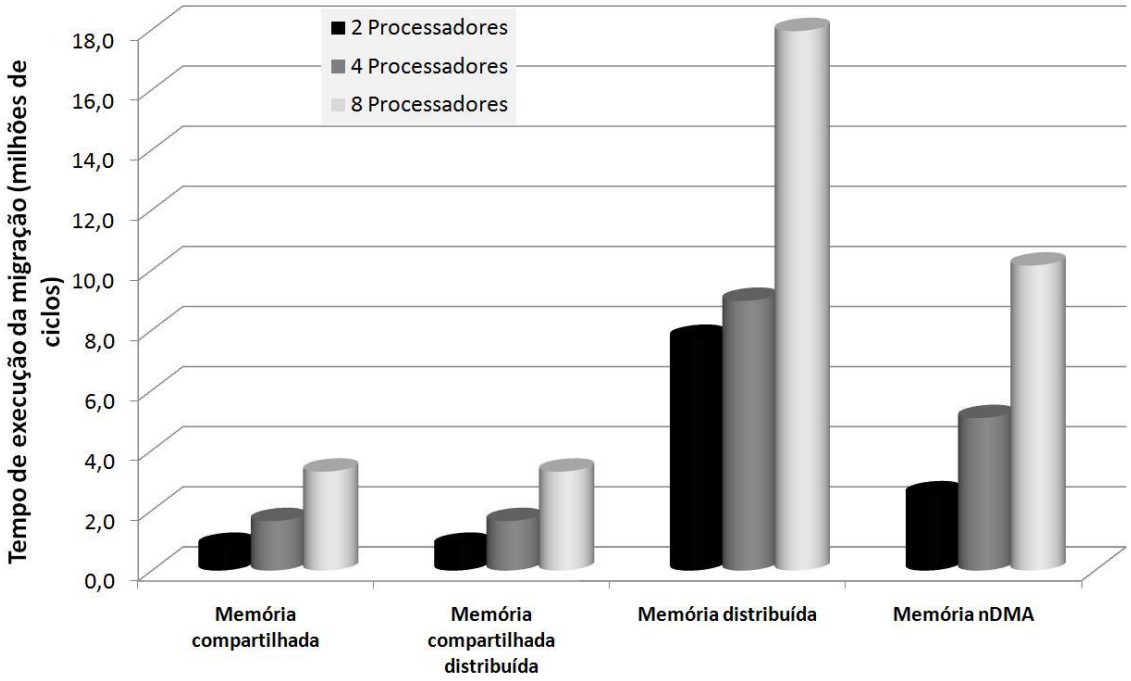


Figura 4.24. Performance da migração da Multiplicação de Matrizes.

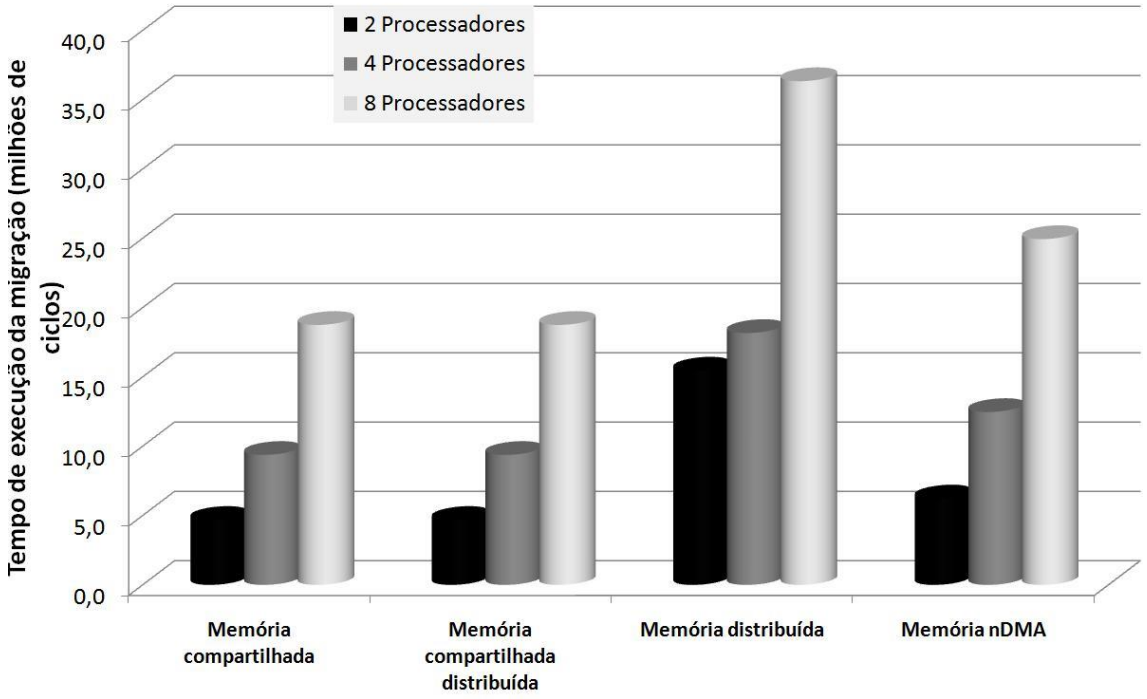


Figura 4.25. Performance da migração do JPEG.

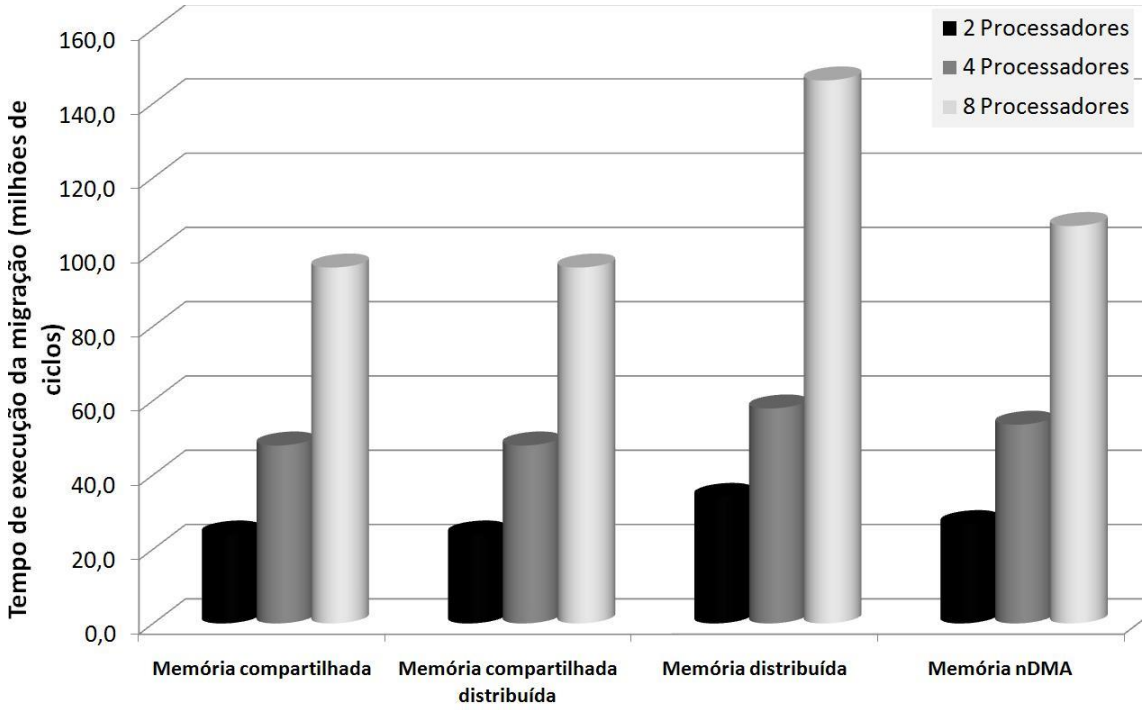


Figura 4.26. Performance da migração do Mergesort.

4.4.2 Resultados quanto ao consumo de energia

No que diz respeito ao consumo de energia dinâmica, os resultados ilustrados nas Figuras 4.27, 4.28, 4.29 e 4.30 mostram o mesmo padrão apresentado nos resultados de performance.

Dentre os componentes do sistema, percebe-se que a NoC apresenta um baixo consumo de energia durante o processo de migração se comparado com o consumo de energia dos processadores e da memória. Em particular, o processo de migração aparenta ser muito custoso para o processador que precisa executar todo o processo de envio e recebimento das mensagens bem como executar as rotinas de leitura e escrita dos dados nas memórias de programa e de dados (no caso da memória distribuída).

Mais uma vez os resultados apontam para um pior resultado por parte da memória distribuída pelos fatores discutidos na seção anterior. As memórias compartilhada e compartilhada distribuída apresentam os melhores resultados dado o menor tamanho de código a ser lido e escrito nas memórias (diminuindo assim o numero de acessos) o que também diminui o tempo de execução do processador realizando esta tarefa, o que leva a um menor consumo de energia.

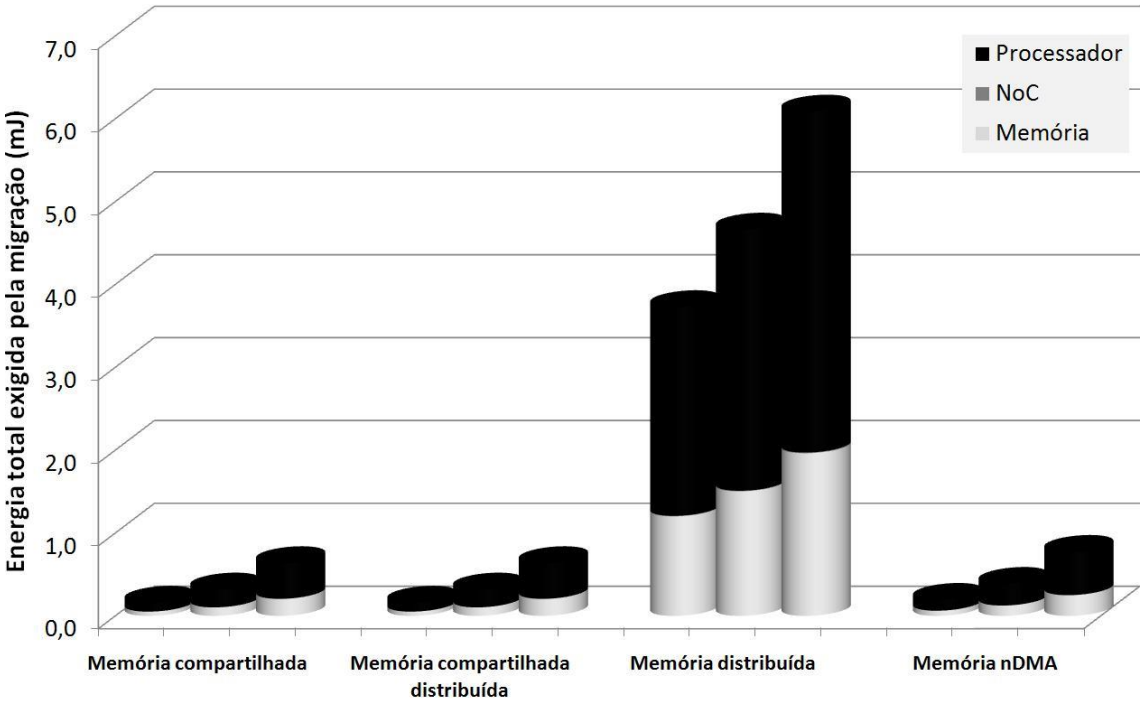


Figura 4.27. Consumo de energia da migração da Estimação de Movimento.

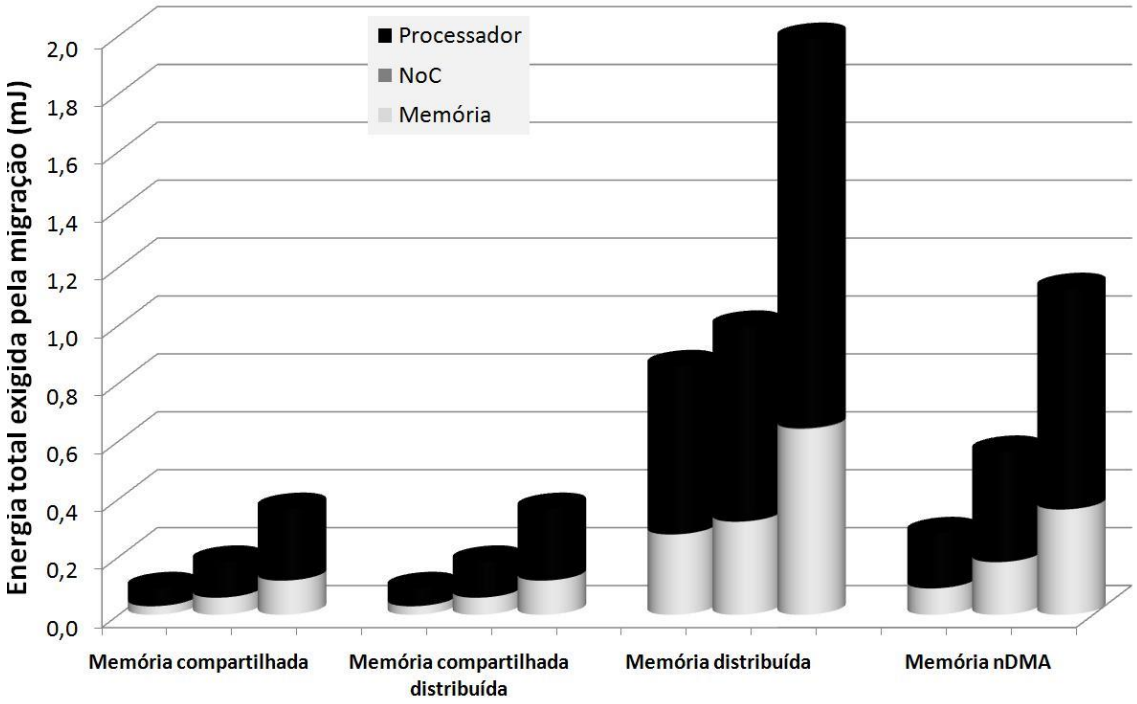


Figura 4.28. Consumo de energia da migração da Multiplicação de Matrizes.

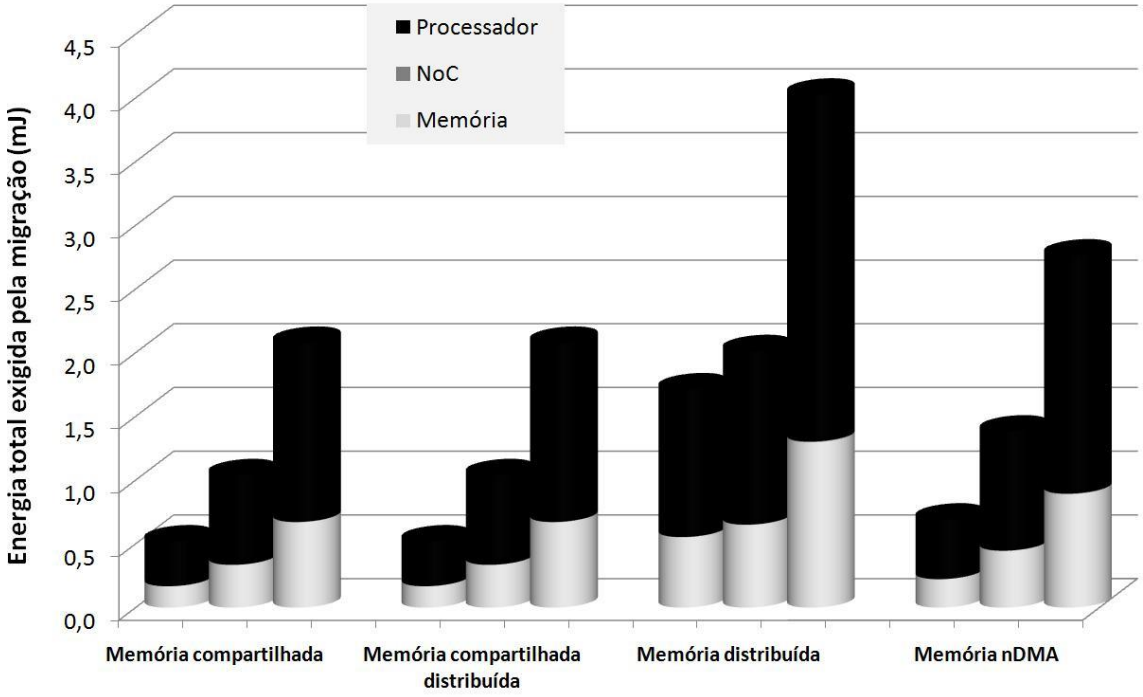


Figura 4.29. Consumo de energia da migração do JPEG.

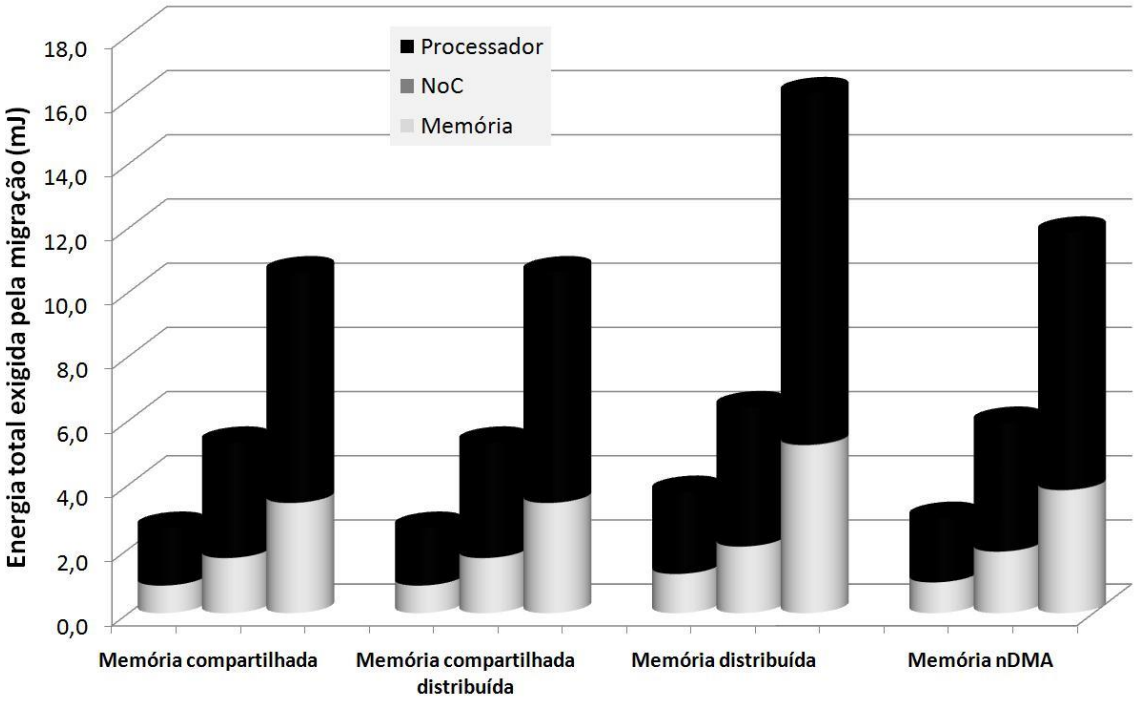


Figura 4.30. Consumo de energia da migração do Mergesort.

4.5 Sumário dos experimentos

Os experimentos apresentados neste capitulo se dividem em três aspectos:

- Análise dos modelos de memória executando o conjunto de aplicações sugerido.
- Análise dos modelos de memória em situações de alta latência.
- Análise da execução da migração de tarefas em cada modelo de memória.

No primeiro aspecto conclui-se que o modelo de memória distribuída apresenta melhores resultados de performance, exceto em casos onde a carga de comunicação da aplicação é grande. Nestes casos os modelos de memória compartilhada e compartilhada distribuída apresentam melhores resultados. O modelo nDMA também apresenta resultados melhores do que o modelo de memória distribuída nestes casos, porém em casos de baixa carga de comunicação apresenta piores resultados. Os resultados de consumo de energia tendem a seguir o mesmo padrão dos resultados de performance, devido ao fato de que um menor tempo de execução tende a apresentar um menor consumo de energia.

No que diz respeito aos experimentos em situações de alta latência percebe-se que o modelo de memória distribuída apresenta piores resultados de performance considerando a perda de desempenho conforme o aumento da latência na comunicação em aplicações com baixa carga de comunicação. De maneira simétrica, o modelo nDMA apresenta melhores resultados conforme a latência aumenta em aplicações de baixa comunicação. Isto é resultado do modelo de comunicação da memória nDMA que tende a utilizar mensagens menores na troca de dados entre os elementos de

processamento. Os resultados de consumo de energia apresentam um padrão de comportamente similar aos resultados de performance.

Por fim, nos experimentos considerando a execução de uma migração de tarefas, os modelos de memória compartilhada e compartilhada distribuída apresentam melhores resultados devido à natureza intríseca destes modelos que dispensa o envio de dados das aplicações pela rede. O modelo de memória distribuída apresenta os piores resultados devido ao fato de que tanto código quanto dados precisam ser migrados.

5 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou um estudo sobre o impacto da hierarquia de memória em MPSoCs baseados em NoC. Para realizar esta avaliação foram implementadas diversas organizações de memória em uma plataforma virtual para simulação de MPSoCs chamada SIMPLE. SIMPLE é parametrizável em vários aspectos arquiteturais e foi desenvolvida em SystemC. Foram implementados um modelo de memória compartilhada, um modelo de memória compartilhada distribuída bem como um novo modelo de memória fisicamente compartilhado porém com diferentes espaços de endereçamento chamado de nDMA.

Foram realizados experimentos com um conjunto de benchmarks com níveis diferentes de carga de comunicação, isto é, níveis diferente de quantidade de informação a ser transmitida entre os elementos de processamento. Os experimentos tiveram por objetivo medir performance, consumo de energia dinâmica e tráfego gerado na rede.

A partir dos resultados de performance foi possível concluir que a memória distribuída leva alguma vantagem sobre os outros modelos em casos onde a carga de comunicação é pequena. Quando isso não é verdade, os outros modelos de memória apresentam um melhor resultado. Considerando performance nos modelos de memória compartilhada, a conclusão é que eles apresentam resultados muito similares com ligeira vantagem, na maioria dos casos, para a memória compartilhada distribuída, devido à possibilidade de acesso à memória em paralelo, o que não é possível em outros modelos. Todavia, a memória nDMA apresentou resultados competitivos e até mesmo melhores em alguns casos.

Quanto aos resultados de energia, foi mostrado que o modelo de memória distribuída tem o processador como o principal componente consumidor de energia e segue a mesma característica de dependência da carga de comunicação apresentada nos resultados de performance. A organização de memória compartilhada apresenta a cache como a principal responsável pelo consumo de energia nos casos onde o princípio da localidade é bem explorado. Caso não seja, a NoC e a memória principal são os principais responsáveis. No modelo nDMA, a NoC não demonstra tal relevância devido ao baixo tráfego gerado na rede.

No que diz respeito ao tráfego na NoC, duas principais conclusões são possíveis:

- A memória nDMA apresenta considerável melhoria quando comparada com as outras memórias compartilhadas. Esta melhoria chega a 45% em média para algumas aplicações.

- A memória distribuída apresenta muito menos tráfego do que os outros modelos. Entretanto este resultado é extremamente dependente da carga de comunicação como apresentado nos resultados de performance e consumo de energia.

Diante destes resultados experimentais, a organização de memória nDMA, desenvolvida no escopo deste trabalho, é eficiente o suficiente no que diz respeito à performance e consumo de energia e se mostra uma melhor solução considerando tráfego na NoC se comparada com outros modelos de memória compartilhada. Quando comparada com o modelo de memória distribuída, os experimentos sugerem que nDMA apresenta melhores resultados quando a carga de comunicação aumenta caracterizando-a como um modelo de memória robusto.

Em um segundo momento foram realizados experimentos visando medir a tolerância dos modelos de memória em situações de alta latência na NoC. Para realizar estes experimentos foi desenvolvido um gerador de tráfego para ser utilizado nos experimentos de modo a emular alta latência.

Os experimentos sugerem que o modelo de memória distribuída apresenta alta tolerância à latência de comunicação na maioria das situações, especialmente em situações onde a carga de comunicação é alta. O modelo nDMA também apresentou tolerância em alguns casos e mostrou melhores resultados em situações onde a carga de comunicação é baixa. O alto nível de comunicação requerido pelos modelos de memória compartilhada e compartilhada distribuída leva a piores resultados conforme o tráfego sintético aumenta. Isto é devido ao fato de que a comunicação por meio da NoC é requerida não somente para acessar os dados da aplicação, mas também para realizar comunicação entre as tarefas por meio de uma memória remota.

No que diz respeito ao consumo de energia dinâmica, é possível concluir que o componente do sistema mais afetado pela alta latência é a própria NoC, enquanto que outros componentes tais como a memória e as caches não são afetadas devido ao fato de que o número de acessos não muda com a alta latência e por isso o consumo de energia dinâmica também não muda.

Embora o modelo de memória distribuída apresente melhores resultados na maioria das situações, existe um padrão que mostra que, para aplicações com baixa carga de comunicação, este modelo sofre mais com as altas latências da rede. Por outro lado, o modelo de memória nDMA apresenta resultados com baixa degradação conforme a carga de comunicação de uma aplicação é baixa.

Outro conjunto de experimentos teve por objetivo analisar a performance das diferentes organizações de memória no que diz respeito à migração de tarefas. Como resultado de trabalhos anteriores, a plataforma SIMPLE possui suporte para migração de tarefas (memória de dados, código e pilha) utilizando o modelo de memória distribuída. Para realizar os experimentos comparativos foram realizadas adaptações nos modelos de memória implementados anteriormente de modo que fosse possível utilizar as mesmas rotinas de migração de tarefas utilizadas em um contexto de memória distribuída.

Os resultados experimentais sugerem que o modelo de memória distribuída é o modelo no qual a migração de tarefas é mais onerosa. Estes resultados experimentais já eram esperados devido ao simples fato de que a memória distribuída necessita enviar os dados da tarefa de maneira explicita pela rede, o que não é verdade para os outros modelos. No caso da memória compartilhada e compartilhada distribuída, o fato de ter

uma memória global limita a quantidade de informação a ser transferida em uma migração à memória de programa e a pilha. No caso do modelo nDMA a situação é bem parecida. A diferença se dá pelo fato de que a cópia da memória de dados é feita localmente no nó da memória, o que a torna um processo mais rápido e independente da rede. Outro fator que fez com que a memória compartilhada e compartilhada distribuída obtivessem melhores resultados foi o fato de que o código das aplicações nestes casos é mais simples do que no caso da memória distribuída, onde toda a comunicação deve ser explicitada.

Trabalhos futuros incluem a realização de experimentos com mais aplicações que façam uso de cargas de comunicação ainda maiores a fim de comprovar a tendência de que o desempenho de acordo com o modelo de memória é dependente da quantidade de dados a serem transferidos durante a comunicação de uma determinada aplicação.

O gerador de tráfego utilizado neste trabalho cria um tráfego que é uniformemente distribuído tanto no tempo quanto no espaço. Como trabalhos futuros, estuda-se a utilização de geradores de tráfego mais complexos (tais como os apresentados em Carara (2007) e Mahadevan (2005)) de modo a avaliar a possível influência do modelo de tráfego nos resultados.

Também estão incluídos nos trabalhos futuros experimentos com alocação de tarefas para as diferentes organizações de memória.

Um ambiente complexo com vários sub-sistemas e várias aplicações abre espaço para a adoção de uma metodologia de projeto que tenha por objetivo prover uma eficiente comunicação interna a estes sub-sistemas e também entre eles. Dada a conclusão desta dissertação de que certas aplicações obtêm resultados distintos para cada modelo de memória, um outro trabalho futuro seria a modelagem de uma hierarquia de memória híbrida formada por diferentes hierarquias para cada sub-sistema. O uso de diferentes hierarquias de memória almejaria prover o melhor modelo de comunicação e acesso a dados para cada sub-sistema e sua respectiva aplicação.

A complexidade deste ambiente traria à tona problemas já conhecidos dos sistemas multiprocessados. Entretanto estes problemas tomariam dimensões ainda maiores. Um exemplo é o problema da coerência de cache, que neste caso se tornaria mais complexo devido à necessidade de manter coerência de dados entre os sub-sistemas. Além disso, o uso de diversos e diferentes modelos de memória levaria à necessidade da coerência de memória prevista por Dutt (2008). Esta tendência é conseqüência do uso de memórias híbridas, o que faria com que houvesse a necessidade de manter a coerência de dados entre os diferentes modelos de memória presentes em um mesmo sistema.

REFERÊNCIAS

ABRAHAM, S.; MAHLKE, S. Automatic and efficient evaluation of memory hierarchies for embedded systems. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 1999, Haifa. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 1999. p. 114–125.

BANAKAR, R. ET AL. Scratchpad Memory: a Design Alternative for Cache On-Chip Memory in Embedded Systems. In: INTERNATIONAL SYMPOSIUM ON HARDWARE/SOFTWARE CODESIGN, 2002, Estes Park. P**roceedings…** New York, NY: ACM, 2002. p. 73–78.

BARCELOS, D.; BRIÃO, E. W.; WAGNER, F. R. A Hybrid Memory Organization to Enhance Task Migration and Dynamic Task Allocation in NoC-based MPSoCs. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2007, Rio de Janeiro. **Proceedings…** New York, NY: ACM, 2007. p. 282-287.

BARCELOS, D. **Um modelo de migração de tarefas para MPSoCs baseados em redes-em-chip**. 2008. 92p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BARTIC, T. A. *et al*. Topology adaptive network-on-chip design and implementation. **Computer and Digital Tecniques**, v. 152, n. 4, p.467-472, Jul. 2005.

BENINI, L. *et al.* A Survey of Design Techniques for System-level Dynamic Power Management. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Princeton, NJ, v. 8, n. 3, p. 299-316, June 2000.

BENINI, L.; DE MICHELI, G. Networks on chips: a new SoC paradigm. **IEEE Computer**, [S.l.], v. 35, n. 1, p. 70-78, Jan. 2002.

BRIÈRE, M. et al. **Heterogeneous modeling of an optical network-on-chip with SystemC**. In: IEEE INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, 16. , 2005. p. 10-16.

CARARA, E.; MELLO, A.; MORAES, F. Communication Models in Networks-on-Chip. In: INTERNATIONAL WORKSHOP ON RAPID SYSTEM PROTOTYPING, 2007, Porto Alegre. **Proceedings…** Los Alamitos, CA: IEEE Computer Society 2007. pp.57-60.

CHING, D.; SCHAUMONT, P.; VERBAUWHEDE, I. Integrated modeling and generation of a reconfigurable network-on-chip. In: PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 18. 2004. **Proceedings...** [S.l.:s.n.], 2004.

DE MASSAS, P. G.; PETROT, F. Comparison of memory write policies for NoC based Multicore Cache Coherent Systems. In: DESIGN AUTOMATION AND TEST

IN EUROPE DATE, 2008, Munique, Los Alamitos, CA: IEEE Computer Society 2008. p. 997-1002.

DUTT, N. Memory-aware NoC Exploration and Design. In: DESIGN, AUTOMATION AND TEST IN EUROPE DATE, 2008, Munique, **Proceedings...** Los Alamitos, CA: IEEE, 2008. v. 1, p. 1128-1129.

FORSELL, M. A scalable high-performance computing solution for networks onchips. **IEEE Computer**, [S.l.], v. 22, n. 5, p. 46-55, Set. 2002.

GIRÃO, G. ET AL. Cache coherency communication cost in a NoC-based MPSoC platform, In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2007, Rio de Janeiro. **Proceedings…** New York, NY: ACM, 2007. p. 288-293.

HISER, J.D.; DAVIDSON, J.W.; WHALLEY, D.B. Fast, Accurate Design Space Exploration of Embedded Systems Memory Configurations. In: SYMPOSIUM ON APPLIED COMPUTING, Seoul, 2007. **Proceedings...** New York, NY: ACM, 2007 699-706.

HWANG, K. **Advanced Computer Architecture: Parallelism, Scalability, Programmability**. [S.l.]: McGraw-Hill, 1993.

ITO, S.A.; CARRO, L.; JACOBI, R.P. Making Java work for microcontroller applications. **Design & Test of Computers, IEEE** Volume 18, n. 5, p.100-110.

JERGER, N.E.; PEH, L-S.; LIPASTI, M., Virtual Tree Coherence: Leveraging Regions and In-Network Multicast Trees for Scalable Cache Coherence In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE MICRO, Lake Como, 2008. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2008.

KIM, D.; KIM, M.; SOBELMAN, G.E. DCOS: Cache Embedded Switch Architecture for Distributed Shared Memory Multiprocessor SoCs. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, Island of Kos, 2006. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2006. pp. 979-982.

LEE, H. G. *et al*. On-Chip Communication Architechture Exploration: a quantitative exploration of Point-to-Point, Bus and Network-on-chip Architectures, **ACM Transactions on Design Automation of Eletronic Systems**, Vol. 12 , pp. 21-40, 2007.

LOGHI, M.; PONCINO, M. Exploring energy/performance tradeoffs in shared memory MPSoC: Snoop-based cache coherence vs. software solutions. In: DESIGN AUTOMATION AND TEST IN EUROPE DATE, Munique, 2005. **Proceedings…** Washington, DC: IEEE Computer Society, 2005. pp. 508–513.

MAHADEVAN, S. *et al*. A Network Traffic Generator Model for Fast Network-on-Chip Simulation, In: DESIGN AUTOMATION AND TEST IN EUROPE, DATE, 2005. **Proceedings…** Washington, DC : IEEE Computer Society, 2005. pp. 780-785.

MARESCAUX, T.; BROCKMEYER, E.; CORPORAAL, H. The impact of higher communication Layers on NoC supported MPSoCs. In: INTERNATIONAL SYMPOSIUM ON NETWORKS-ON-CHIP, Princeton, 2007. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2007. p. 107-116.

MARWEDEL, P. **Embedded System Design.** Boston: Kluwer Academic Publishers, 2003.

MILENKOVIC, A.; MILENKOVIC, M.; BARNES, N. A performance evaluation of memory hierarchy in embedded systems. In: SOUTHEASTERN SYMPOSIUM ON SYSTEM THEORY, 2003. **Proceedings...** [S.n: S.l] 2003, p 427-431.

MONCHIERO, M. Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors**.** In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS: ARCHITECTURES, MODELING AND SIMULATION, 2006. **Procedings…** [S.n: S.l] 2006, p. 144-151.

MOTTA, R. B. **Uso de Clock Gating para Compartilhamento de Memória em CMPs Heterogêneos.** 2008. 106p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

NOLLET, V. *et al.* Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In: DESIGN AUTOMATION AND TESTE IN EUROPE, DATE, 2005. **Proceedings…** Washington, DC : IEEE Computer Society, 2005. p. 234-239.

PATTERSON, D. A.; Hennessy, J. L. Computer Architecture: A Quantitative Approach. São Francisco: Morgan Kaufmann, 2003.

PETROT, F.; GREINER, A.; GOMEZ, P. On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN, 2006, Dubrovnik, **Proceedings…** Washington, DC: IEEE Computer Society 2006, p. 53-60.

SHIUE, W.; CHAKRABARTI, C. Memory exploration for low power, embedded systems. **Journal of VLSI Signal Processing**, vol. 29, p. 167–178, 2001.

STEINKE, S. ET AL. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In: PROCEEDINGS OF DESIGN AUTOMATION AND TEST IN EUROPE, p. 409-415, 2002.

SUHENDRA, V.; RAGHAVAN, C.; MITRA, T. Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In: PROCEEDINGS OF THE 2006 INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS, p. 401-410, 2006.

TANENBAUM, A. S. **Modern Operating Systems**. 2nd ed. Englewood Cliffs: Prentice Hall, 1992.

VERMA, M.; WERMEYER, L.; MARWEDEL, P. Cache-aware Scratchpad Allocation Algorithm. In PROCEEDINGS OF DESIGN AUTOMATION AND TEST IN EUROPE, p. 1264-1269, 2004.

WANG, H. Orion: A Power-performance Simulator for Interconnection Networks. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 2002. **Proceedings...** New York: ACM, 2002. p. 294-305.

WEISER, M. *et al.* Scheduling for Reduced CPU Energy. In: CONFERENCE ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, 1994, Monterrey, California. **Proceedings…** Berkeley, CA: USENIX Association, 1994. p.13-23.

WILTON, S.; JOUPPI, N. Cacti: an enhanced access a cycle time model, **IEEE Journal of Solid State Circuits**, Vol. 31, No. 5, p.677-688, 1996.

ZAHRAN, M. M. On cache memory hierarchy for Chip-Multiprocessor**. SIGARCH Computer Architecture News**, vol. 31, p. 39-48, 2003.

ZEFERINO, C. A. ET AL. A study on communication issues for systems-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2002. **Proceedings…** [S.n.] Porto Alegre, p. 121-126, Setembro 2002.

ZEFERINO, C.; SUSIN, A. SoCIN: a parametric and scalable network-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2003, São Paulo. **Proceedings…** Los Alamitos, CA: IEEE Computer Society 2003. pp.169-174.