PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SERVIÇOS DE COMUNICAÇÃO DIFERENCIADOS EM SISTEMAS MULTIPROCESSADOS EM CHIP BASEADOS EM REDES INTRA-CHIP

EVERTON ALCEU CARARA

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação.

Orientador: Prof. Dr. Fernando Gehm Moraes

Porto Alegre
2011

Dados Internacionais de Catalogação na Publicação (CIP)

C261s Carara, Everton Alceu

Serviços de comunicação diferenciados em sistemas multiprocessados em chip baseados em redes intra-chip / Everton Alceu Carara. – Porto Alegre, 2011.

107 p.

Tese (Doutorado) – Fac. de Informática, PUCRS.
Orientador: Prof. Drª Fernando Gehm Moraes

1. Informática. 2. Multiprocessadores. 3. Serviços de Comunicação. 4. Redes de Computadores. I. Moraes, Fernando Gehm. II. Título.

CDD 004.35

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "Serviços de Comunicação Diferenciados em Sistemas Multiprocessados em Chip Baseados em Redes Intra-Chip", apresentada por Everton Alceu Carara, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, Sistemas Embarcados e Sistemas Digitais , aprovada em 25/08/2011 pela Comissão Examinadora:

Prof. Dr. Fernando Gehm Moraes – Orientador — PPGCC/PUCRS

Prof. Dr. Ney Laert Vilar Calazans – — PPGCC/PUCRS

Prof. Dr. Michel Robert — LIRMM – França

Prof. Dr. Marius Strum – — USP

Prof. Dr. Luigi Carro – — UFRGS

Homologada em 13/09/11..., conforme Ata No. 18..... pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

SERVIÇOS DE COMUNICAÇÃO DIFERENCIADOS EM SISTEMAS MULTIPROCESSADOS EM CHIP BASEADOS EM REDES INTRA-CHIP

RESUMO

Sistemas multiprocessados em chip (*MPSoCs - Multiprocessor Systems-on-Chip*) estão sendo considerados como provável padrão para implementar os sistemas embarcados futuros. O poder computacional destas plataformas possibilita a execução simultânea de diversas aplicações com diferentes requisitos. O emprego de redes intra-chip (*NoCs – Networks-on-Chip*) como infraestrutura de comunicação em tais plataformas é uma realidade em pesquisas acadêmicas e projetos industriais. NoCs são comumente vistas como alternativa aos tradicionais barramentos, oferecendo como principais vantagens escalabilidade e suporte a diversas comunicações em paralelo. Contudo, a motivação para o seu emprego em SoCs (*Systems-on-Chip*) vai além dessas vantagens óbvias, visto que NoCs podem suportar diversos serviços de comunicação com diferentes níveis de qualidade.

Visto que comumente as aplicações que executam em MPSoCs são compostas por diferentes tarefas comunicantes, o eficiente suporte à comunicação tem um papel fundamental no desempenho destas e é uma área estratégica no desenvolvimento de plataformas multiprocessadas. Muitos trabalhos têm sido conduzidos na última década nas áreas de NoCs e MPSoCs, entretanto poucos tratam a lacuna existente entre os níveis de rede (serviços de comunicação) e de aplicação em MPSoCs baseados em NoC. Este trabalho tem por objetivo a implementação de diferentes serviços de comunicação no nível de rede e a disponibilização destes no nível de aplicação, preenchendo assim a lacuna existente entre tais níveis através de uma melhor integração hardware/software.

A metodologia de projeto seguida neste trabalho parte da implementação de mecanismos específicos no nível da rede, os quais dão suporte a serviços de comunicação diferenciados. Tais serviços são expostos no nível de aplicação através de primitivas que compõem a API (*Application Programming Interface*) de comunicação. O propósito desta abordagem é oferecer ao desenvolvedor de aplicações meios, em software, para satisfazer os requisitos de comunicação das aplicações, especialmente daquelas com restrições temporais. As avaliações realizadas mostram o funcionamento e os benefícios obtidos através da utilização dos serviços implementados, além de apontar alguns cenários onde estes não se adequam tão bem.

Palavras chave: MPSoC, NoC, QoS, API e serviços de comunicação.

DIFFERENTIATED COMMUNICATION SERVICES IN MULTIPROCESSOR SYSTEMS-ON-CHIP BASED ON NETWORKS-ON-CHIP

ABSTRACT

Multiprocessor systems on chip (MPSoCs) are being considered as a probable standard for implementing future embedded systems. The computational power of these platforms allows the simultaneous execution of several applications with different requirements. The use of networks-on-chip (NoCs) as the communication infrastructure in these platforms is a reality in academic research and industrial projects, as well NoCs are considered an alternative to traditional buses, offering as major advantages scalability and support to multiple parallel communications. However, the motivation for their use in SoCs (Systems-on-Chip) goes beyond these evident advantages, once NoCs can support several communication services with different levels of quality.

Since applications running on MPSoCs are often composed by several communicating tasks, efficient support to communications has a key role on the performance of these and is a strategic area in the development of multiprocessed platforms. Many works have been conducted in the last decade in NoC and MPSoC areas, however few of these address the gap between the network (communication services) and application levels within NoC-based MPSoCs. This work aims at the implementation of different communication services at the network level and its availability at the application level, in this way bridging the gap between these levels by a better hardware/software integration.

The followed design methodology starts with the implementation of specific mechanisms at the network level to support differentiated communication services. These services are exposed at the application level through primitives that compose a communication API (Application Programming Interface). The purpose of the approach is to offer to application developers' software support to meet application communication requirements, especially to those applications with time constraints. The performed evaluations show the operation and the benefits obtained through the use of the implemented services, besides identifying some scenarios where these do not fit so well.

Keywords: MPSoC, NoC, QoS, API and communication services.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ADPCM	Adaptive Differential Pulse-Code Modulation
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
BE	Best Effort
CHAIN	CHip Area INterconnect.
CTG	Communication Task Graph
DMA	Direct Memory Access
DSP	Digital Signal Processor
DMEM	Data Memory
DyAD	Dynamic Adaptive Deterministic
EOP	End of Packet
FF	Flip-Flop
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FPMAC	Floating-point Multiply-accumulator
FSL	Fast Simplex Link
GALS	Globally Asynchronous Locally Synchronous
GAPH	Grupo de Apoio ao Projeto de Hardware
GPP	General Purpose Processor
GS	Guaranteed Service
GT	Guaranteed Throughput
HAMUM	Hamiltonian Adaptive Multicast Unicast Method
HeMPS	Hermes Multiprocessor System
IDN	I/O Dynamic Network
IMEM	Instruction Memory
IP	Intellectual Property
ISS	Instruction Set Simulator
JPEG	Joint Photographic Experts Group
LEC-DN	Low Energy Consumption – Dependences Neighborhood
LUT	Look up Table
MDN	Memory Dynamic Network
MJPEG	Motion JPEG
MPARM	Multiprocessor ARM
MMPI	Multiprocessor Message Passing Interface
MPI	Message Passing Interface
MPSoC	Multi Processor System on a Chip
MSI	Modified Shared Invalid
MWD	Multi-Window Displayer
µkernel	Microkernel

NePA	Networked Processor Array
NI	Network Interface
NoC	Network-on-Chip
NO-OPT	No-Optimized
NORMA	No Remote Memory Access
NPU	Network Processing Unit
NUMA	Non Uniform Memory Access
ocMPI	on-chip MPI
OmpSCR	OpenMP Source Code Repository
OpenMP	Open Multi-Processing
OPT	Optimized
PE	Processing Element
POOSL	Parallel Object-Oriented Specification Language
QoS	Quality-of-Service
RAM	Random Access Memory
ROM	Read-Only Memory
RTEMS	Real-Time Executive for Multiprocessor Systems
RTL	Register Transfer Level
SD	Synchronous Data Flow
SDM	Spatial Division Multiplexing
SDRAM	Synchronous Dynamic Random Access Memory
SNS	Smart Network Stack
SMP	Symmetric Multiprocessor
SIMPLE	Simple Multiprocessor Platform Environment
SoC	System on a Chip
SoCIN	System-on-Chip Interconnection Network
SOPC	System-on-a-Programmable-Chip
STN	Static Network
TDM	Time Division Multiplexing
TDN	Tile Dynamic Network
UART	Universal Synchronous Asynchronous Receiver Transmitter
UDN	User Dynamic Network
UMA	Uniform Memory Access
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
VLIW	Very Large Instruction Word
VLSI	Very Large Scale Integration
VOPD	Video Object Plane Decoder
XML	Extensible Markup Language

SUMÁRIO

1. INTRODUÇÃO

Para a próxima década (2010-2020), a lei de Moore prevê a integração de vários bilhões de transistores em um único chip. Entretanto, tornou-se claro que em termos de processamento a exploração do paralelismo no nível de instrução, com *pipelines* profundos e agressivas técnicas superescalares, juntamente com o uso da abundância de transistores para implementar *caches* chegou ao seu limite. O aumento do desempenho a partir de altas frequências está cada vez mais difícil devido aos problemas de dissipação de calor e ao elevado consumo de energia. A solução está em arquiteturas multiprocessadas, as quais apresentam uma melhor relação desempenho/consumo que arquiteturas monoprocessadas com desempenho semelhante, além de possibilitarem a exploração do paralelismo em níveis mais altos de abstração.

Para amortizar os custos de fabricação e os longos ciclos de desenvolvimento de projetos ASIC, os pesquisadores buscam soluções para satisfazer a crescente necessidade de programabilidade pós-fabricação, a qual é oferecida pelos processadores. O uso de vários processadores no projeto de sistemas embarcados e a sua integração em um único chip (*SoC - System-on-Chip*) deram origem a uma nova classe de sistemas chamada de MPSoC (*Multiprocessor System-on-Chip*). MPSoCs emergiram na última década como uma importante e auspiciosa classe de sistemas VLSI (*Very Large Scale Integration*). Essa classe é considerada a mais promissora na continuidade da exploração do alto nível de integração proporcionado pela tecnologia de semicondutores, satisfazendo restrições de desempenho e consumo de energia [CEN08]. Um MPSoC é um sistema integrado em um único chip que agrega múltiplos processadores como componentes principais [WOL08]. Em um nível mais alto de abstração, MPSoCs podem ser vistos como plataformas reconfiguráveis através da programação (software) dos seus processadores. O crescente interesse nessa nova classe de sistemas reside na sua capacidade de combinar alto desempenho e flexibilidade. Graças à sua programabilidade, um mesmo MPSoC pode ser empregado em diferentes produtos, reduzindo o *time-to-market* destes e estendendo a vida útil da plataforma (*time-in-market*). Essa reusabilidade aumenta o volume de produção dos chips e reduz o custo destes para o consumidor. O projeto de MPSoCs é uma atividade multidisciplinar que envolve infraestruturas de comunicação intra-chip, arquitetura de microprocessadores e memórias, modelos de programação, fluxos de co-simulação e metodologias para modelagem e exploração do espaço de projeto.

Quanto ao multiprocessamento, MPSoCs podem ser classificados em duas categorias.

- *Heterogêneos*: sistemas compostos por diferentes tipos de processadores (GPPs, DSPs, ASIPs, etc), aceleradores e periféricos.

- *Homogêneos*: sistemas compostos por uma unidade básica chamada de elemento de processamento (*PE – Processing Element*), a qual é instanciada múltiplas vezes. Comumente, os componentes básicos de um PE incluem um processador de propósito geral e uma memória local.

Arquiteturas heterogêneas são mais voltadas a aplicações com rígidas restrições de potência, consumo de energia e desempenho, sendo mais difíceis de serem programadas e menos flexíveis que arquiteturas homogêneas. A homogeneidade oriunda da replicação dos PEs tem como principais benefícios escalabilidade e tolerância a falhas, além de favorecer o *layout* do circuito e técnicas como mapeamento e migração de tarefas.

A disponibilidade de diversos recursos de processamento presente nos MPSoCs possibilita a exploração do paralelismo no nível de tarefas. Para tal fim, as aplicações desenvolvidas são compostas por diversas tarefas que podem ser executadas em paralelo, sendo estas mapeadas nos processadores da plataforma alvo. Além disso, várias aplicações podem executar simultaneamente, caracterizando o paralelismo no nível de aplicação. Em geral, aplicações são independentes e não exigem comunicação entre si. Entretanto, tarefas de uma mesma aplicação comunicam-se entre si para troca de dados e sincronização. Com o aumento do número de aplicações/tarefas executando simultaneamente, o volume de dados movimentados na plataforma intensifica-se a ponto de poder comprometer o desempenho geral do sistema. Em virtude da relação direta entre comunicação entre tarefas e desempenho da aplicação, é imprescindível que a infraestrutura de comunicação empregada suporte elevadas taxas de comunicação e alto grau de paralelismo. A solução natural para tal problema está nas redes de interconexão intra-chip (*NoCs – Networks-on-Chip*), as quais além de suprirem as necessidades relativas a largura de banda e paralelismo, proporcionam a escalabilidade necessária para o crescimento dos sistemas. Em [LEE07] é apresentada uma avaliação analítica e experimental onde os autores comparam várias infraestruturas de comunicação intra-chip (barramentos, NoCs e conexões ponto-a-ponto) e verificam que NoCs apresentam as menores diferenças entre atrasos estimados e atrasos reais obtidos após a extração do *layout*.

NoCs são basicamente compostas por roteadores e canais de comunicação ponto-a-ponto que interconectam núcleos de propriedade intelectual (*IP - Intellectual Property*) de um sistema integrado [ZEF03]. O roteador é o elemento principal, sendo responsável pela definição de rotas (roteamento), controle de fluxo, qualidade de serviço e garantia de entrega dos pacotes. Escalabilidade está inerentemente presente no conceito de NoCs, visto que a largura de banda agregada cresce junto com suas dimensões (e.g. adição de roteadores e enlaces). O modelo de comunicação utilizado é troca de mensagens transmitidas na forma de pacotes. Recentemente, diversas pesquisas têm sido conduzidas na área de NoCs a fim de propor diferentes *serviços de*

comunicação que estendem o serviço básico de troca de mensagens, tais como controle de prioridades no envio, estabelecimento de conexões (temporárias ou permanentes) e suporte à comunicação coletiva (*multicast/broadcast*). O principal objetivo desses serviços é proporcionar um tratamento diferenciado aos fluxos de comunicação com restrições temporais (e.g. latência e vazão), oferecendo diferentes níveis de qualidade (*QoS – Quality-of-Service*), a fim de prover meios para o atendimento de *deadlines*.

MPSoCs baseados em NoC são uma nova tendência em sistemas embarcados. A união dessas duas áreas segue o exemplo de sucesso dos *clusters* da área de processamento paralelo, onde processadores são interconectados por redes de alto desempenho. A partir dessa união, torna-se possível a criação de plataformas intra-chip escaláveis e programáveis com alto poder computacional, capazes de executar simultaneamente diversas aplicações. Todavia, a integração de várias aplicações sobre a mesma plataforma deve ser conduzida com cautela, uma vez que cada aplicação tem seus próprios requisitos e padrões de comunicação entre tarefas, os quais exigem da rede diferentes serviços de comunicação. *Portanto, fica evidente a necessidade de ampliar a programabilidade dos MPSoCs baseados em NoC a fim de disponibilizar os serviços de comunicação da rede no nível de software*. Essa abordagem visa estender a união arquitetural (MPSoCs e NoCs) até o nível de software, oferecendo um controle mais amplo sobre a plataforma. Isso favorece a exploração do espaço de projeto e facilita a otimização incremental das aplicações a partir de ciclos programação e avaliação.

O desenvolvimento de aplicações paralelas para plataformas multiprocessadas deve seguir um modelo de programação que determina como as tarefas se comunicam e se sincronizam. Tipicamente um modelo de programação consiste em primitivas incorporadas a uma linguagem de programação. A funcionalidade de tais primitivas pode ser implementada diretamente pela infraestrutura de comunicação ou a partir de serviços básicos oferecidos por esta (e.g. *multicast* a partir de vários *unicasts*). Os principais modelos de programação paralela são:

- *Troca de mensagens*: neste modelo, comunicação e sincronização são expressas *explicitamente* no código fonte das aplicações utilizando primitivas que implementam a troca de mensagens (e.g. *send* e *receive*). Esse modelo proporciona ao programador maior controle sobre a execução paralela, permitindo a eficiente exploração dos serviços de comunicação oferecidos pela plataforma, a fim de satisfazer os requisitos das aplicações. A eficiência da solução depende do *programador*. Plataformas com arquitetura de memória NORMA (*No Remote Memory Access*) empregam esse modelo de programação, cuja implementação mais conhecida é o MPI [WAL96].
- *Memória compartilhada*: neste modelo, a comunicação entre tarefas é expressa *implicitamente* no código fonte a partir do acesso a variáveis compartilhadas em memória. A sincronização entre elas depende da implementação de mecanismos de exclusão mútua (e.g. semáforos). Nesse modelo o paralelismo é explorado

principalmente pelo compilador (e.g. diretivas de compilação), facilitando a programação das aplicações. A eficiência da solução depende do *compilador*. Plataformas com arquitetura de memória UMA (*Uniform Memory Access*) e NUMA (*Non Uniform Memory Access*), tipicamente empregam esse modelo de programação, cuja implementação mais conhecida é o OpenMP [OPE97].

A melhor maneira de programar MPSoCs em relação a produtividade e qualidades dos resultados é atualmente um tópico de intensa discussão. O desenvolvimento de aplicações altamente otimizadas exige um grande esforço, necessitando de ajustes cuidadosos em termos de algoritmo, particionamento e mapeamento na plataforma alvo [LEU10].

1.1 Originalidade e objetivos da Tese

A originalidade do trabalho está na exposição dos serviços de comunicação da NoC no nível de software através de uma API (*Application Programming Interface*), preenchendo assim a lacuna existente entre os níveis de rede e de aplicação em MPSoCs baseados em NoC. A partir de primitivas de comunicação em software, o programador deve ser capaz de programar aplicações de maneira que elas atinjam seus requisitos. Essa abordagem visa dar autonomia às aplicações, permitindo que elas tenham controle sobre os recursos da rede de maneira distribuída e em tempo de execução.

O presente trabalho foi desenvolvido dentro do escopo de MPSoCs homogêneos com arquitetura de memória NORMA (troca de mensagens) e NoCs com topologia malha bidimensional e roteamento distribuído, tendo como base o MPSoC acadêmico HeMPS (Capítulo 3).

Os serviços oferecidos por um determinado módulo (hardware/software) são as funcionalidades que este implementa. O termo *serviço diferenciado* refere-se a uma extensão das funcionalidades básicas, visando oferecer um controle mais amplo sobre o módulo tornando-o mais flexível e personalizável. Os seguintes serviços de comunicação diferenciados foram adicionados à API do MPSoC HeMPS:

- *Baseado em prioridades*: a prioridade de uma mensagem está relacionada à alocação de recursos da rede para a sua transmissão. Uma mensagem com alta prioridade tem mais recursos disponíveis que uma mensagem com baixa prioridade;
- *Baseado em conexão*: antes da transmissão de um fluxo de mensagens, uma conexão é estabelecida entre um par origem/destino e os recursos da rede permanecem alocados durante toda a comunicação;
- *Roteamento diferenciado*: uma mensagem pode ser transmitida pela rede por um caminho determinístico ou adaptativo;
- *Comunicação coletiva (Multicast)*: uma mesma mensagem é transmitida para vários destinos de maneira escalável.

Os objetivos estratégicos da presente Tese incluem:

- Analisar o ganho de desempenho proporcionado pelos serviços implementados em relação ao serviço básico de comunicação;
- Explorar qualidade dos serviços de comunicação (QoS) a partir do nível de software, de maneira a satisfazer os requisitos de desempenho das aplicações;
- Minimizar a interferência entre aplicações que executam simultaneamente, a fim de tornar o desempenho destas independente da presença/ausência umas das outras;
- Dominar o projeto de MPSoCs baseados em NoC desde o nível de rede até o nível de aplicação.

Tais objetivos estratégicos dependem dos seguintes objetivos específicos:

- Nível de rede
    - Implementação de um mecanismo de alocação de recursos baseado em prioridades fixas;
    - Implementação do modo de chaveamento por circuito a fim de suportar conexões fim-a-fim;
    - Combinação entre roteamentos adaptativo e determinístico a fim de utilizar um ou outro, dependendo do fluxo;
    - Implementação de um algoritmo *multicast* a fim dar suporte à comunicação coletiva de maneira escalável.
- Nível de sistema operacional
    - Definição de primitivas de comunicação e parâmetros;
    - Implementação da API de comunicação;
    - Implementação de um escalonador de tarefas preemptivo baseado em prioridades.

1.2 Contribuição

Este trabalho deixa como contribuição para a comunidade de pesquisa em MPSoCs um *framework open source* (Apêndice B – HeMPS Generator) que possibilita a geração automatizada de MPSoCs baseados em NoC com suporte a diversos serviços de comunicação acessíveis via software, a fim de permitir ao projetista controlar o compartilhamento da arquitetura de interconexão em alto nível e atingir os requisitos das aplicações. Tal *framework* permite ainda a seleção de diversos parâmetros de hardware/software, mapeamento de tarefas e depuração no nível de aplicação. A plataforma gerada é completamente descrita em VHDL RTL sintetizável. Para fins de

simulação, o *framework* disponibiliza também, modelos SystemC com precisão de ciclo para processadores (ISS) e memórias.

1.3 Organização do texto

O restante do texto está organizado em mais seis capítulos:

- O Capítulo 2 revisa trabalhos relacionados ao tema da presente proposta de Tese, como MPSoCs baseados em NoCs e soluções que visam garantir os requisitos de desempenho das aplicações;
- O Capítulo 3 apresenta o MPSoC acadêmico HeMPS, o qual serve de base para a realização do presente trabalho e diversos outros dentro do grupo de pesquisa GAPH (Grupo de Apoio ao Projeto de Hardware);
- O Capítulo 4 descreve os serviços de comunicação baseados em prioridades e conexões. Tais serviços são explorados no escopo de características relevantes (e.g. QoS e *composability*) a sistemas que executam aplicações de tempo real;
- O Capítulo 5 propõe um serviço de comunicação com roteamento diferenciado, o qual combina as vantagens de algoritmos de roteamento determinístico e adaptativo a fim de assegurar a disponibilidade de um número maior de recursos de comunicação a tráfegos prioritários;
- O Capítulo 6 descreve o serviço de comunicação coletiva que tem como base o algoritmo *multicast dual-path*. Tal serviço oferece uma alternativa eficiente para o envio de mensagens com múltiplos destinos, além de reduzir o volume de tráfego na rede;
- O Capítulo 7 encerra esta Tese discutindo as conclusões obtidas e apresentando algumas ideias para trabalhos futuros.

Os Apêndices A e B apresentam, respectivamente, a lista das publicações realizadas durante o período do doutorado e o framework *HeMPS Generator*.

2. TRABALHOS RELACIONADOS

Diferentes arquiteturas de MPSoCs têm sido propostas recentemente, tanto no âmbito acadêmico como industrial, focando vários domínios de aplicação. No entanto, proporcionar meios a fim de que as aplicações tenham seus requisitos de desempenho satisfeitos ainda continua sendo um desafio. Este Capítulo é dividido em duas seções. Na Seção 2.1 são revisados MPSoCs baseados em NoCs, discutindo algumas de suas características como arquitetura e serviços/API de comunicação. Tais MPSoCs reúnem características que definem a tendência das futuras plataformas multiprocessadas. A Seção 2.2 apresenta várias propostas que têm por finalidade oferecer garantias de desempenho às aplicações a partir de mecanismos específicos agregados ao sistema. Cada uma das seções inclui em seu final considerações sobre os trabalhos revisados.

2.1 MPSoCs baseados em NoCs

2.1.1 Tile64™

O dispositivo Tile64 [TIL07] foi desenvolvido pela Tilera e pode ser visto como uma evolução do processador RAW [TAY02]. Trata-se de uma matriz homogênea bidimensional de 64 PEs interconectados pela NoC iMesh [WEN07]. Além dos PEs, a NoC conecta também controladores de memória e de entrada/saída. Cada PE contém um processador VLIW, *caches* L1 e L2, um DMA e um roteador. Um PE pode executar de maneira independente um sistema operacional completo ou vários PEs podem ser agrupados a fim suportar um sistema operacional multiprocessado como o SMP Linux. A família de dispositivos Tile-Gx é a mais recente e suporta até 100 PEs, no entanto segue a mesma arquitetura básica do Tile64 ilustrada na Figura 1.

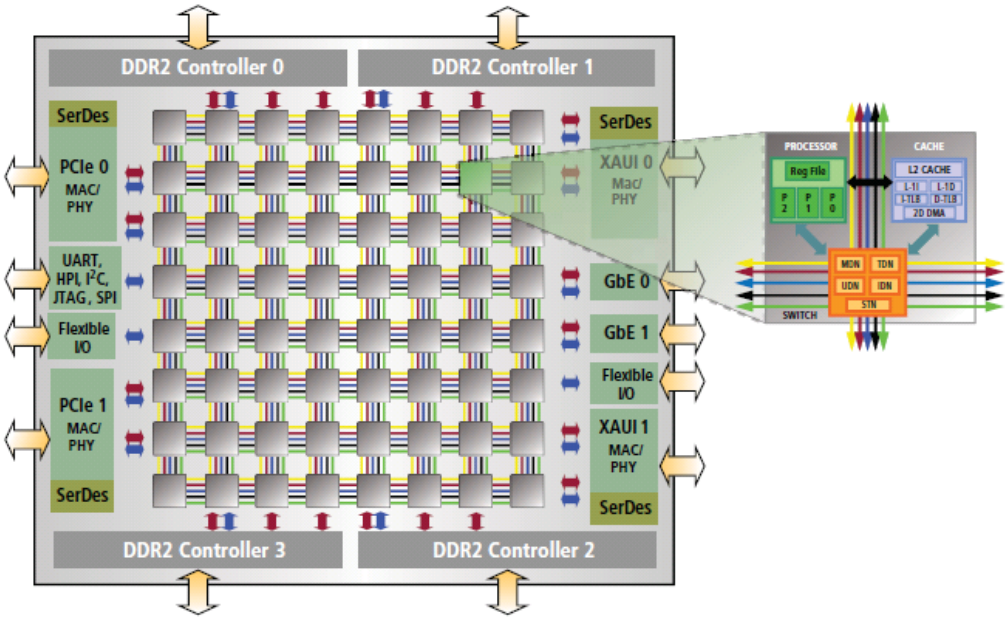


Figura 1 – Arquitetura do Tile64 e PE [TIL07].

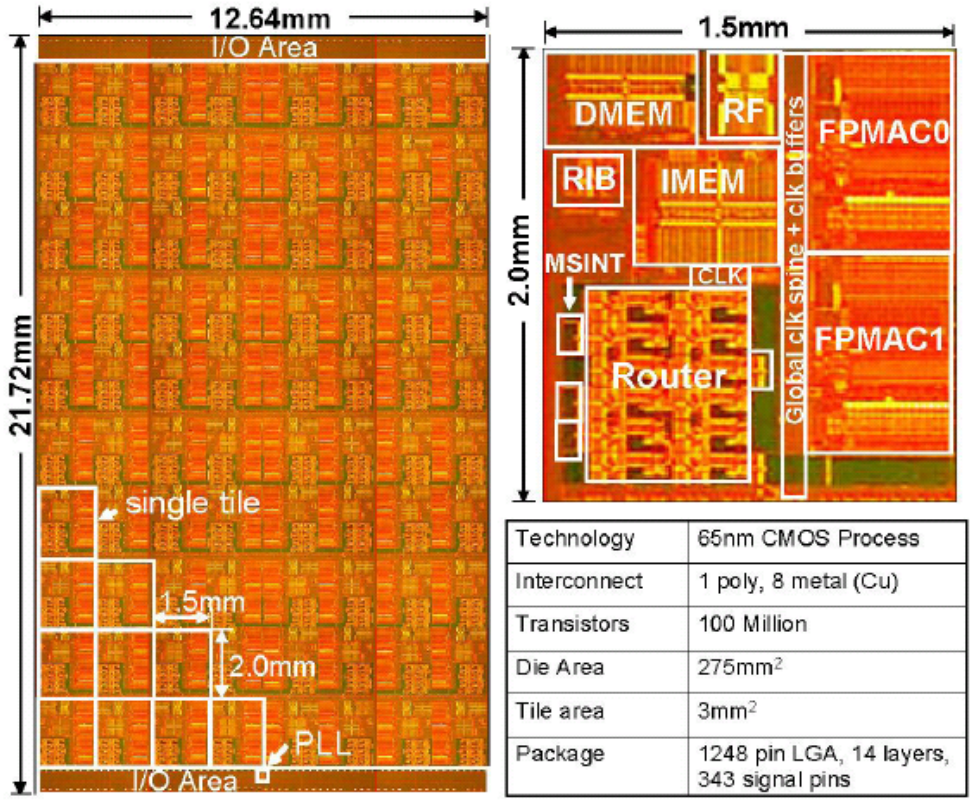
A NoC iMesh é composta por 5 redes malha bidimensionais: UDN (*User Dynamic Network*), IDN (*I/O Dynamic Network*), STN (*Static Network*), MDN (*Memory Dynamic Network*) e TDN (*Tile Dynamic Network*). As redes dinâmicas (UDN, IDN, MDM e TDN) implementam chaveamento por pacotes (*wormhole*) e a rede estática (STN) implementa chaveamento por circuito. As redes UDN, IDN e STN estão integradas no *pipeline* do processador, permitindo rápido acesso via registradores. Assim, qualquer instrução do processador pode ter como origem e/ou destino uma dessas redes. Essa abordagem não dissocia comunicação e computação dentro do PE, pois o processador é responsável pela injeção e retirada de pacotes nas redes, estando desta maneira sujeito à bloqueios devido à ausência de dados durante uma leitura ou ausência de espaço durante uma escrita.

A Tilera fornece ao programador a biblioteca *iLib*, a qual possui uma série de primitivas de comunicação implementadas sobre a rede UDN. Essa biblioteca é independente de sistema operacional e dá acesso direto à rede, devendo ser ligada a cada uma das aplicações durante o processo de compilação. A *iLib* implementa comunicação baseada em *sockets* e troca de mensagens. Através de *sockets* é possível criar conexões lógicas a partir da alocação de *buffers* (FIFO) no PE destino. Dois tipos de conexão lógica são suportadas; (*i*) *raw channels* e (*ii*) *buffered channels.* Utilizando *raw channels*, conexões lógicas são criadas alocando *buffers* de acesso rápido no nível da rede (integrados no *pipeline*), enquanto *buffered channels* aloca *buffers* na memória local. *Raw channels* oferece comunicação com baixa latência, entretanto o controle de fluxo fim-a-fim deve ser implementado pelo programador. *Buffered channels* implementa o protocolo de comunicação *rendezvous*, onde a origem da comunicação solicita permissão de envio e aguarda uma resposta do destino, antes de enviar dados (protocolo síncrono). A comunicação baseada em troca de mensagens é semelhante ao padrão MPI, suportando o envio de mensagens a qualquer momento para qualquer destino sem a necessidade de uma conexão. Resultados comparativos entre os tipos de comunicação suportados reportam uma vazão (bytes/ciclo) de 3,93 para *raw channels*, 1,95 para *buffered channels* e 1 para troca de mensagens, considerando enlaces de 4 bytes [TIL07].

2.1.2 Terascale

O projeto Terascale da Intel [VAN07][VAN08] apresenta um *array* de processamento composto por 80 *tiles* operando a uma frequência de 4GHz (Figura 2). Um *tile* consiste em um PE conectado a um roteador. Cada PE possui duas unidades independentes de ponto flutuante de precisão simples (FPMAC), memória de instruções (IMEM) e memória de dados (DMEM). A NoC empregada apresenta topologia malha, chaveamento por pacotes (*wormhole*) e interfaces assíncronas. Toda comunicação entre PEs é realizada através de troca de mensagens utilizando as instruções *send* e *receive*. O pico de desempenho atingido é 1.0 Tflops à 1V e 1.28 Tflops à 1.2V. A potência dissipada é estimada em 98W à 1V e 181W à 1.2V. O projeto possui em torno de 100 milhões de transistores. No nível de arquitetura, observa-se uma estrutura típica de MPSoC baseado

em NoC. Entretanto o fato dos PEs serem baseados em unidades de ponto flutuante e o baixo suporte a software o deixa a margem da tendência em sistemas embarcados.



Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm^2
Tile area	3mm^2
Package	1248 pin LGA, 14 layers, 343 signal pins

Figura 2 - *Layout* do chip e PE [VAN07].

2.1.3 SIMPLE

SIMPLE (*Simple Multiprocessor Platform Environment*) [SIL08] é uma plataforma virtual baseada em SystemC, usada para o desenvolvimento e validação de MPSoCs homogêneos com suporte a aplicações de tempo real. O sistema é baseado em PEs interconectados pela NoC SoCIN [ZEF03]. Cada PE contém um processador Java [ITO01], memória local, interface de rede, portas de entrada/saída e um relógio de tempo real. Os processadores implementam RTSJ (*Real-Time Specification for Java*) [BOL01], dessa maneira suportando *multithread* e aplicações de tempo real. A troca de mensagens entre *threads* é suportada pela COM-API, a qual permite o estabelecimento de canais de comunicação e a atribuição de diferentes prioridades às mensagens. Entretanto a NoC não dá suporte a serviços de comunicação com QoS, baseados em conexões ou prioridades. Como estudo de caso foi implementado um controlador de guindaste (*crane*) particionado em três *threads*. Cada uma é executada por um dos três processadores que compõem a instância da plataforma. Alguns tempos obtidos em relação ao processamento do envio e da recepção de dados variam, respectivamente, de 4800 e 3700 ciclos de *clock* para mensagens de 1 byte a 312100 e 388700 ciclos de *clock* para mensagens de 490 bytes. Estes valores não consideram a latência da NoC, apenas o tempo de processamento das primitivas de comunicação.

2.1.4 HS-Scale

A HS-Scale [ALM09] é um MPSoC que visa a auto-adaptabilidade do sistema em tempo de execução a partir da técnica de migração de tarefas. A migração de uma tarefa é disparada a partir do monitoramento da carga do processador ou da ocupação das filas de mensagens recebidas. Não há uma preocupação em relação a heurística utilizada para selecionar o processador destino da tarefa a ser migrada. Requisições de migração são enviadas aos processadores mais próximos e a tarefa é migrada para o primeiro que aceitar a requisição. Apenas o código da tarefa é migrado, restringindo a abordagem à migração apenas de tarefas modeladas como laços contendo a sequência recepção, processamento e envio de dados. A arquitetura do sistema é formada por uma matriz bidimensional homogênea de NPUs (*Network Processing Units*) que se comunicam através da NoC Hermes [MOR04]. Cada NPU inclui um processador Plasma [RHO01], memória local, interface de rede (*NI - Network Interface*), roteador (implícito no *Network Layer*), temporizador (*Timer*) e UART. A Figura 3 ilustra uma instância 4x4 da HS-Scale juntamente da estrutura da NPU.

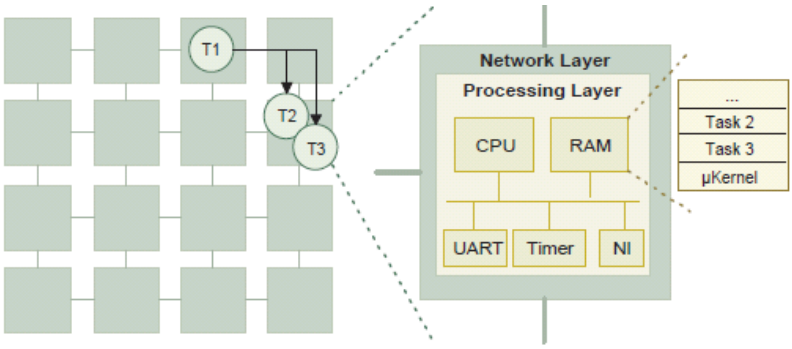


Figura 3 – Instância 4x4 da plataforma HS-Scale e estrutura da NPU [ALM09].

Cada NPU executa um *µkernel* (microkernel) preemptivo com capacidade de executar várias tarefas simultaneamente. O *µkernel* implementa também alguns mecanismos típicos de sistemas operacionais como alocação dinâmica de memória, semáforos e *mutex*, além de uma API básica de comunicação entre tarefas. Essa API é baseada no padrão MPI de troca de mensagens e possui apenas duas primitivas; (*i*) `MPI_Send()` e (*ii*) `MPI_Receive()`, as quais são utilizadas respectivamente para o envio e recepção de mensagens. O protocolo de comunicação empregado é *eager*, onde a origem da comunicação envia dados independente do estado do destino (protocolo assíncrono). Não há um acordo entre origem e destino antes do início da transmissão. Visto que a API é implementada no *µkernel*, as primitivas são invocadas a partir de chamadas de sistema. Essa abordagem evita que as tarefas tenham acesso direto ao hardware. Resultados obtidos a partir de uma instância 4x4 multi-FPGA (Figura 4) da plataforma mostram a elevação da vazão de um decodificador MJPEG quando as tarefas, inicialmente mapeadas na mesma NPU, são migradas uma a uma para as NPUs vizinhas [ALM09]. Resultados semelhantes são observados em [ALM10] a partir da simulação de uma instância 2x2 da plataforma executando a mesma aplicação.

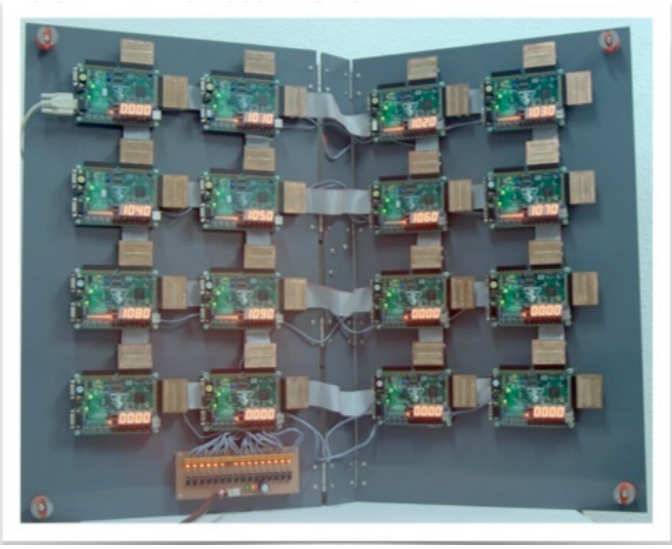


Figura 4 – Instância 4x4 multi-FPGA (Spartan 3) da plataforma HS-Scale.

2.1.5 NePA

NePA (*Networked Processor Array*) [LEE08] é uma matriz bidimensional homogênea de PEs baseados no processador OpenRISC [ERL01] (Figura 5). A plataforma inclui também um simulador escrito em SystemC e modelos RTL sintetizáveis (Verilog) dos componentes que compõem a arquitetura. Além de canais virtuais, a NoC replica os canais físicos no eixo Y (norte e sul) criando dois caminhos verticais disjuntos, visando a prevenção de *deadlock* em algoritmos de roteamento adaptativos. O sistema conta com NIs bem definidas que abstraem a complexidade da NoC e dissociam computação e comunicação graças a módulos DMA responsáveis pelo envio e recepção de pacotes. Essa plataforma não oferece uma API de comunicação de alto nível, sendo a troca de mensagens implementada através da leitura/escrita em registradores da NI mapeados em memória. Para demonstrar a utilidade do simulador, o *benchmark* SPLASH-2 [WOO95] foi executado sobre uma instância 7x7 do NePA, avaliando a potência média dissipada para diferentes mapeamentos.

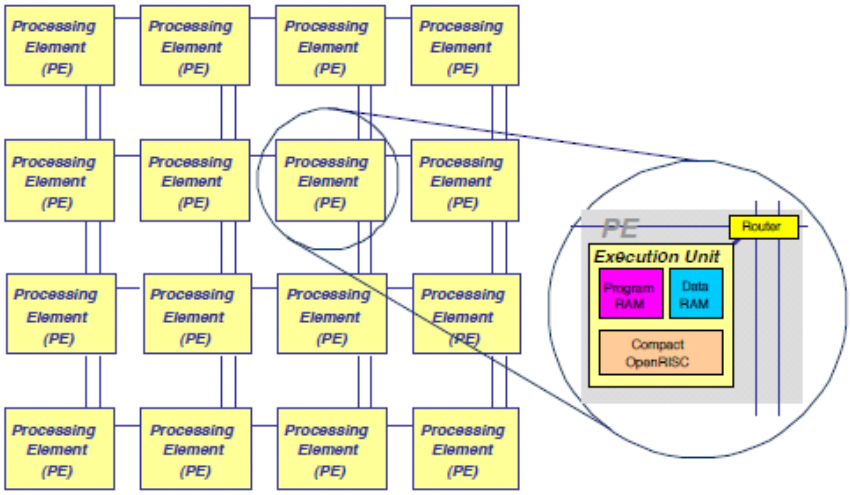


Figura 5 - Arquitetura 4x4 do NePA e do PE [LEE08].

2.1.6 Geração automatizada de MPSoCs baseados em NoCs

xENoC [JOV08] é um ambiente proposto para a geração automatizada de MPSoCs baseados em NoC. Ele é baseado na ferramenta *NoCWizard*, a qual permite a geração de NoCs (Verilog RTL) a partir da especificação de diversos parâmetros como topologia, controle de fluxo, modo de chaveamento e algoritmo de roteamento. O ambiente possui uma biblioteca de IPs contendo diferentes processadores e aceleradores, o que possibilita a geração de MPSoCs heterogêneos. O sistema é todo descrito em um arquivo XML (características da NoC, IPs e mapeamento) que é utilizado como entrada para as ferramentas de geração automatizada.

Além da infraestrutura de hardware, xENoC inclui também uma biblioteca para troca de mensagens e sincronização entre tarefas chamada ocMPI (*on-chip* MPI) [JOV09]. Essa biblioteca é uma versão embarcada do padrão MPI e é independente de sistema operacional. As primitivas MPI suportadas são listadas na Figura 6. Dependendo dos requisitos da aplicação, apenas um subconjunto das primitivas precisa ser incluído. A quantidade de memória necessária para armazenar a biblioteca pode variar de 4942 bytes (primitivas básicas) à 13258 bytes (conjunto completo). Visto que a NoC gerada suporta apenas transmissões *unicast*, serviços de comunicação coletiva como *broadcast* (`ocMPI_Broadcast()`) são implementados a partir das primitivas básicas `ocMPI_Send()` e `ocMPI_Recv()`. Experimentos pouco elucidativos sobre uma instância 2x2 da plataforma reportam *speed-ups* próximos do número de processadores, para aplicações com alto grau de independência de dados.

Types of MPI functions	Ported MPI functions
Management	ocMPI_Init(), ocMPI_Finalize(), ocMPI_Initialized(), ocMPI_Finalized(), ocMPI_Comm_size(), ocMPI_Comm_rank(), ocMPI_Get_processor_name(), ocMPI_Get_version()
Profiling	ocMPI_Wtick(), ocMPI_Wtime()
Point-to-point Communication	ocMPI_Send(), ocMPI_Recv(), ocMPI_SendRecv()
Advanced & Collective Communication	ocMPI_Broadcast(), ocMPI_Barrier() ocMPI_Gather(), ocMPI_Scatter(), ocMPI_Reduce(), ocMPI_Scan(), ocMPI_Exscan()

Figura 6 – Primitivas MPI suportas pela biblioteca ocMPI [MUR09].

Em [KUM07] é apresentado um fluxo integrado automatizado para a geração de MPSoCs voltados para FPGAs. A arquitetura é baseada em processadores Silicon Hive [SIL06] e na NoC Aethereal [GOO05]. O MPSoC é descrito em um arquivo de especificação do sistema, o qual serve como entrada para o fluxo. Ao final, é gerada uma descrição VHDL RTL do sistema e modelos de simulação para cada um dos seus componentes. A Figura 7 mostra um trecho do arquivo de especificação juntamente com a arquitetura correspondente. O *host* (Figura 7) atua como mestre do sistema, podendo ser um computador ou um processador embarcado. Além de depuração do sistema, ele tem como principais funções carregar os códigos objetos das tarefas nas memórias locais dos

processadores e configurar a NoC através do estabelecimento das conexões entre IPs comunicantes. O serviço de comunicação baseado em conexões implementado pela Aethereal permite a especificação dos parâmetros da conexão (e.g. largura de banda e latência máxima) em tempo de projeto. Visto que o *host* é responsável pelo estabelecimento das conexões, estas são transparentes aos IPs. A comunicação é realizada através de interfaces de entrada/saída mapeadas em memória. Para a validação do fluxo, dois sistemas foram gerados. O primeiro consistindo de três processadores (sendo um o *host*) conectados a um único roteador executando uma aplicação produtor/consumidor. O segundo tem como *host* um computador e a arquitetura é semelhante a da Figura 7, utilizando alguns recursos da placa de prototipação como memória e dispositivos de áudio/vídeo. Para o segundo sistema, várias topologias de rede foram utilizadas, desde um único roteador até uma malha 2x2.

```
System NoC_IOs
{
 node    core1  (noc.ni1.pi);
 node    core2  (noc.ni2.pi);
 node    core3  (noc.ni3.pi);
 node    extmem (noc.ni4_0.pi);
 node    video  (noc.ni4_1.pi);
 node    audio  (noc.ni4_2.pi);
 node    host;
 network noc (host.cfg, host.dat, core1.pi, core2.pi,
                                              core3.pi);
};
```

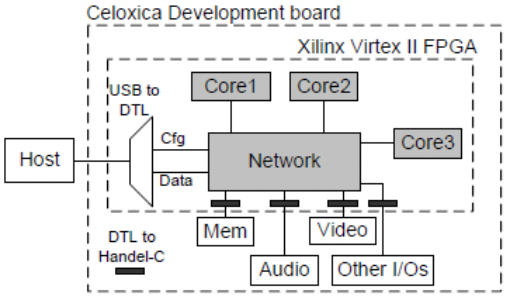


Figura 7 – Exemplo de sistema proposto em [KUM07], incluindo um trecho do arquivo de especificação e a arquitetura correspondente.

[SIN10] também apresenta um fluxo de projeto de MPSoCs baseados em NoCs voltado para FPGAs. Inicialmente é gerada a NoC utilizando a ferramenta *NoC Generator* [YAN10]. Essa ferramenta permite a geração de NoCs vazão garantida através da multiplexação espacial dos enlaces (*SDM – Spatial Division Multiplexing*). Uma vez gerada a NoC, PEs baseados no processador microblaze são conectados às interfaces de rede através de portas FSL *(Fast Simplex Link*), concluindo assim a geração da infraestrutura de hardware do MPSoC. A comunicação entre PEs é baseada em conexões, as quais são estabelecidas por um PE que controla a comunicação do MPSoC. As configurações das conexões são geradas em tempo de projeto pela ferramenta *NoC Generator* e dependem dos requisitos de comunicação das aplicações a serem executadas. O PE de controle armazena estas configurações e estabelece as conexões antes das aplicações iniciarem a execução. Experimentos realizados sobre uma instância 3x3 da plataforma avaliam apenas os tempos para o estabelecimento de conexões, os quais os autores afirmam ser muito baixos. Entretanto, o tempo para o estabelecimento de uma conexão entre dois PEs se mostra alto quando comparado à plataforma HeMPS (Capítulo 3). Os autores reportam um tempo de 2885 ciclos de *clock*, enquanto a plataforma HeMPS leva em torno de 500 ciclos de *clock*.

Em [ALO10] é apresentada uma metodologia que auxilia o projetista na construção de MPSoCs baseados em NoCs, desenvolvimento de aplicações e análise de

desempenho. Os PEs são criados utilizando a ferramenta *SOPC builder* da Altera, a qual permite a seleção de diversos periféricos em torno do processador NIOS II. A NoC utilizada com infraestrutura de comunicação entre os PEs é criada a partir da ferramenta *NoCMaker* [RUF09], a qual oferece ainda um ambiente de simulação onde é possível avaliar desempenho, consumo de energia e área. Como estudo de caso foi criado um MPSoC composto por quinze PEs escravos e um mestre interconectados por uma malha bidimensional. O PE mestre tem acesso exclusivo a 16MB de memória SDRAM fora de chip e é responsável por orquestrar o trabalho dos PEs escravos. A NoC utilizada implementa chaveamento por pacotes (*wormhole*) e controle de fluxo *handshake*, e oferece apenas o serviço básico de troca de mensagens. Para troca de mensagens entre tarefas é utilizada a biblioteca ocMPI. A escalabilidade da plataforma foi avaliada utilizando duas aplicações; (*i*) multiplicação de matrizes e (*ii*) detector de círculos. O detector de círculos apresentou um ganho de desempenho linear juntamente com aumento o número de processadores (1-15), enquanto a multiplicação de matrizes apresentou ganhos até 7 processadores (1-7).

2.1.7 Proposta de Murilo [MUR09]

Na Tese de Murilo a NoC xpipes [BER04] foi modificada a fim de oferecer serviços de comunicação baseados em prioridades e conexão. Um mecanismo de prioridades foi implementado no árbitro do roteador (centralizado) e tem como objetivo definir a ordem na qual são servidas as requisições vindas das portas de entrada. Quando duas ou mais portas de entrada requisitam uma mesma porta de saída, o árbitro serve primeiro aquela que contém o pacote com maior prioridade. A prioridade de um pacote pode variar de 0 à 7 (mais alta) e é inserida no seu cabeçalho. Essa abordagem oferece baixas garantias visto que as prioridades são verificadas apenas quando há requisições simultâneas para uma mesma porta de saída. O atendimento do árbitro está mais diretamente relacionado com a ordem das requisições, sendo as prioridades na realidade um mecanismo de desempate. O suporte à conexões físicas é baseado em um mecanismo de chaveamento por circuito implementado sobre chaveamento por pacotes. O estabelecimento de uma conexão é feito a partir de um pacote de abertura de conexão que é enviado da origem ao destino. O pacote de abertura de conexão reserva toda a largura de banda do caminho percorrido por toda duração da comunicação. Conexões podem ser unidirecionais ou bidirecionais. A conexão é desfeita por um pacote de fechamento de conexão enviado pela origem, o qual desaloca os recursos da rede à medida que vai sendo transmitido pela conexão.

Uma API foi desenvolvida para integrar os dois novos serviços de comunicação da rede no nível de software e permitir às aplicações explorarem QoS. Ela interage diretamente com a interface de rede, a qual é responsável por definir a prioridade no cabeçalho dos pacotes e gerenciar abertura/fechamento de conexões. A Figura 8 lista as primitivas que compõem a API. As primitivas `ni_open_channel()` e

`ni_close_channel()` são responsáveis respectivamente por abrir e fechar conexões através do envio de pacotes de controle. As primitivas `ni_send_priority_packet()` e `ni_recv_priority_packet()` servem para o envio de pacotes de dados utilizando os serviços de comunicação baseado em prioridades ou conexão.

```
// Set up an end-to-end circuit
// unidirectional or full duplex (i.e. write or R/W)
inline int ni_open_channel(uint32_t address, bool full_duplex);

// Tear down a circuit
// unidirectional or full duplex (i.e. write or R/W)
inline int ni_close_channel(uint32_t address, bool full_duplex);

// Write a packet with a certain level of priority to a specific address
inline int ni_send_priority_packet(uint32_t address, int data, int level);

// Receive a packet with a certain level of priority from an specific address
inline int ni_recv_priority_packet(uint32_t address, int *data, int level);
```

Figura 8 – API utilizada para suportar QoS em software [MUR09].

Sobre tais primitivas foi adicionada uma implementação da API OpenMP especialmente voltada para MPSoCs [MAR09] (Figura 9). Nessa abordagem o programador não acessa diretamente as primitivas de QoS (Figura 8), as quais são chamadas pela implementação do OpenMP. A eficiente exploração de QoS depende das ferramentas de suporte que geram código a partir da descrição das aplicações utilizando o OpenMP. A Figura 9 ilustra o modelo da plataforma virtual com arquitetura de memória compartilhada criada utilizando o ambiente de simulação MPARM [BEN05].

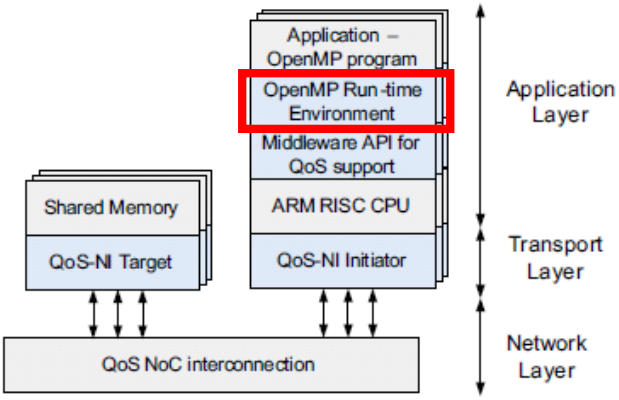


Figura 9 – Modelo de plataforma implementa, ilustrando as camadas que compõem o sistema [MUR09].

O MPARM possibilita a simulação de sistemas descritos em SystemC (com precisão de ciclo), os quais são criados a partir de uma livre combinação de PEs, memórias e infraestruturas de comunicação. O PE é composto por um ISS (*Instruction Set Simulator*) ARM, *cache* L1 e alguns periféricos como controlador de interrupções e UART. Em termos de infraestrutura de comunicação, o ambiente oferece barramentos, *crossbars* e NoCs. Como sistema operacional, são suportados o RTEMS [RTE10] e uma versão reduzida do µClinux [µCL10]. Uma instância 3x8 da plataforma contendo 8 PEs e dois bancos de memória compartilhada (16 módulos) foi avaliada utilizando variações do *benchmark Loop with dependencies*, o qual faz parte do OmpSCR (*OpenMP Source Code*

Repository) [DOR05]. A partir de diversos experimentos observou-se uma redução de até 66% no tempo de execução em relação à implementação tradicional do OpenMP, graças ao suporte a prioridades.

2.1.8 Proposta de Fu et al [FU10]

Este trabalho propõe uma interface para troca de mensagens em MPSoCs baseados em NoCs chamada de MMPI (*Multiprocessor Message Passing Interface*). Tal interface é compatível com o padrão MPI no nível de código fonte e seu objetivo é melhorar a eficiência da exploração do espaço de projeto em termos de mapeamento de aplicações. Um arquivo de mapeamento armazena a localização das tarefas no sistema e é lido através da função `MPI_Init()`. Essa abordagem assegura a independência do código fonte das aplicações em relação ao mapeamento de suas tarefas. Diferentes alternativas de mapeamento podem ser avaliadas alterando-se apenas o arquivo de mapeamento, entretanto somente mapeamento estático é suportado. A Figura 10 mostra a plataforma na qual a interface foi implementada e a arquitetura do PE. A implementação de tal plataforma é baseada no ambiente de simulação M5 [BIN06][YU10], o qual oferece modelos de hardware para criação e simulação de sistemas. Como no caso da biblioteca ocMPI oferecida pelo ambiente xENoC, padrões e comunicação mais complexos são implementados a partir das funções básicas `MPI_Send()` e `MPI_Recv()`. A quantidade memória necessária para suportar a implementação mínima da MMPI é 11KB, podendo variar de acordo com número de funções requisitadas.

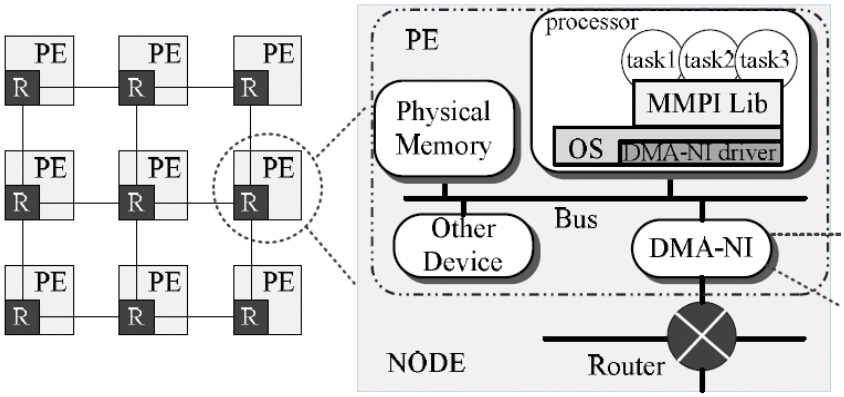


Figura 10 – Plataforma e arquitetura do PE [FU10].

2.1.9 Considerações

A Tabela 1 apresenta um quadro comparativo listando algumas características de cada um dos trabalhos revisados. A última linha da Tabela apresenta características do trabalho aqui proposto.

Uma característica comum em sistemas baseados em NoC é a utilização da topologia *malha*, cuja principal característica está relacionada à capacidade de adequar-se a aplicações que podem ser particionadas em várias tarefas (e.g. operações com

matrizes e processamento de imagens). Além de ser quase um consenso, a regularidade apresentada por esta topologia é bastante adequada para a construção de arquiteturas *homogêneas*, enquanto topologias irregulares (personalizadas) são tipicamente empregadas em arquiteturas heterogêneas projetadas para aplicações específicas.

A natureza inerentemente distribuída das NoCs favorece também o modelo de programação baseado em *troca de mensagens*, uma vez que qualquer comunicação entre IPs é realizada através de pacotes transmitidos de um ponto a outro da rede. Na realidade qualquer infraestrutura de comunicação realiza troca de mensagens entre os IPs conectados a ela. Mesmo em arquiteturas de memória compartilhada distribuída (NUMA) que implementem um modelo de programação baseado em memória compartilhada, a comunicação implícita é convertida em mensagens trocadas entre IPs e módulos de memória distribuídos. O custo dessa conversão é o preço pago pela expressão implícita do paralelismo na implementação das aplicações e pela abstração da memória distribuída, ambas as facilidades oferecidas pelo modelo de programação de memória compartilhada. Além do mais, aplicações paralelas são tipicamente modeladas a partir de grafos que explicitam a comunicação (e.g. *CTG-Communication Task Graph* e *SDF-Synchronous Data Flow*), onde os vértices representam tarefas e as arestas representam troca de mensagens.

Observa-se que a maioria dos trabalhos revisados está em um estágio inicial na integração entre MPSoCs e NoCs, como poder evidenciado principalmente no projeto Terascale, NePA, [KUM07] e [SIN10], os quais não oferecem nem mesmo uma API de comunicação básica implementada em alto nível. No geral as APIs comumente implementam *apenas o serviço básico de comunicação* composto por envio e recepção de mensagens (troca de mensagens), oferecendo pouco ou nenhum controle sobre os recursos da rede. Padrões de comunicação mais complexos são ordinariamente implementados em software a partir do serviço básico como em xENoC, [ALO10] e [FU10]. Da mesma maneira, *poucas das NoCs empregadas implementam serviços de comunicação com capacidade de oferecer algum tipo de garantia às aplicações* além da entrega das mensagens.

O desenvolvimento conjunto entre os serviços de comunicação das NoCs e o suporte em software a estes é uma *área estratégica fundamental* para a evolução dos MPSoCs e foco dessa Tese. A seguir são listados os serviços de comunicação abordados nesse trabalho relacionando-os com o estado da arte.

- Prioridades
    - [MUR09]: implementado no roteador como um mecanismo de desempate entre requisições simultâneas para uma mesma porta de saída;
    - Presente trabalho: implementado no roteador como um mecanismo de alocação de recursos baseado em prioridades.

- Conexão
    - Tile64: implementa conexão lógica a partir da alocação de *buffers* no PE destino;
    - [KUM07][SIN10]: implementam conexão física no nível da NoC, a qual é estabelecida entre IPs comunicantes por um PE central;
    - [MUR09]: implementa conexão física no nível da NoC, a qual é estabelecida por qualquer PE;
    - Presente trabalho: mesma idéia apresentada em [MUR09]. A diferença é que o OpenMP não dá acesso a API de comunicação.
- Roteamento diferenciado
    - Até o presente momento, nenhum trabalho revisado implementa tal serviço, sendo este, no conhecimento do autor, proposto pela primeira vez em [CAR10] e tratado nesta Tese.

Tabela 1 – Quadro comparativo das características dos trabalhos revisados

Trabalho	Arquitetura	Topologia	Modelo de programação	Serviços de comunicação	API de comunicação	Descrição
Tile64™ [TIL07]	Homogênea	Malha	Troca de mensagens	Conexão (lógica) e tráfegos isolados por diferentes redes	iLib	Física (ASIC)
Terascale [VAN08]	Homogênea	Malha	Troca de mensagens	Básico	Instruções *send/receive*	Física (ASIC)
SIMPLE [SIL08]	Homogênea	Malha	Troca de mensagens	Básico	COM-API	Comportamental (SystemC)
HS-Scale [ALM09]	Homogênea	Malha	Troca de mensagens	Básico	Implementada no*μkernel*	RTL (VHDL)
NePA [LEE08]	Homogênea	Malha	Troca de mensagens	Básico	Registradores mapeados em memória	RTL (Verilog)
xENoC [JOV08]	Homogênea/ Heterogênea	Malha/ Torus/ Spidergon	Troca de mensagens	Comunicação coletiva (SW)	ocMPI	RTL (Verilog)
[KUM07]	Homogênea/ Heterogênea	Malha/ Anel	Memória compartilhada	Conexão (física, estabelecida por PE central)	Entrada/Saída mapeada em memória	RTL (VHDL)
[SIN10]	Homogênea	Malha	Troca de mensagens	Conexão (física, estabelecida por PE central)	Leitura/escrita em portas FSL	RTL (VHDL)
[ALO10]	Homogênea	Malha	Troca de mensagens	Comunicação coletiva (SW)	ocMPI	RTL (Verilog)
[MUR09]	Homogênea	Malha	Memória compartilhada	Prioridades e conexão (física, estabelecida por qualquer PE)	OpenMP	Comportamental (SystemC)
[FU10]	Homogênea	Malha	Troca de mensagens	Comunicação coletiva (SW)	MMPI	Comportamental (SystemC)
Tese	**Homogênea**	**Malha**	**Troca de mensagens**	**Prioridades, conexão (física, estabelecida por qualquer PE), roteamento diferenciado e comunicação coletiva (HW)**	**Implementada no *μkernel***	**RTL (VHDL)**

- Comunicação coletiva
    - xENoC [ALO10][FU10]: implementam em software a partir do serviço básico de comunicação;
    - Presente trabalho: implementa em hardware, no nível da NoC, a partir do algoritmo *multicast dual-path*.

2.2 Serviços e estratégias para oferecer garantias de desempenho

2.2.1 GENESYS

O MPSoC GENESYS [OBE10] permite a criação de plataformas para diversos domínios de aplicação (e.g. aviação e automotivo) a partir de uma infraestrutura que oferece um conjunto de serviços básicos. Esses serviços são de 4 tipos: (*i*) sincronização; (*ii*) comunicação; (*iii*) configuração e (*iv*) controle de execução. Sobre estes, serviços de mais alto nível são implementados pelos IPs que compõem o sistema e personalizam a plataforma para um determinado domínio de aplicação. A definição dos serviços básicos originou-se a partir de discussões com indústrias de diferentes domínios de aplicação. O conjunto de serviços básicos é mantido mínimo, incluindo apenas aqueles considerados fundamentais para suportar domínios específicos de aplicações. A Figura 11 ilustra a estrutura dos serviços.

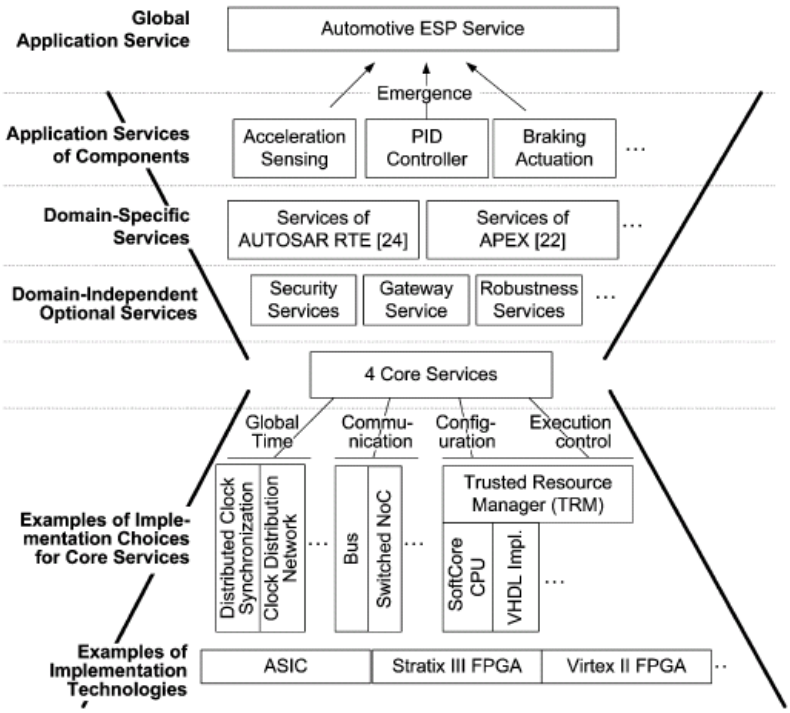


Figura 11 – Estruturas dos serviços [OBE10].

No centro da figura estão os serviços básicos (*core services*), para os quais existem diferentes opções de implementação. Por exemplo, os serviços de comunicação podem ser implementado por uma NoC ou um barramento. Acima dos serviços básicos pode-se observar um refinamento destes a fim de torná-los cada vez mais específicos,

dependendo da aplicação alvo. Um protótipo do GENESYS foi implementado utilizando um FPGA Stratix III para executar o simulador automotivo TORCS [ONI09]. Os resultados mostram a adequação do MPSoC para a implementação de sistemas com rígidas restrições de latência e largura de banda.

2.2.2 Proposta de Motakis et al [MOT11]

Este trabalho apresenta uma abordagem para acessar os serviços da Spidergon STNoC a partir do nível de software. A abordagem é baseada em uma camada de *drivers* sobre o hardware e uma biblioteca de funções (*libstnoc*) que implementa a API. A Spidergon STNoC oferece serviços relativos a gerenciamento de energia, roteamento, segurança e QoS. Tais serviços são acessíveis em tempo de execução e expostos ao nível de software através de registradores mapeados em memória. A Figura 12 ilustra a organização das camadas de software que compõem a abordagem. Através da ferramenta gráfica iNoC é possível gerar diferentes instâncias da Spidergon STNoC (descrição RTL) juntamente dos *drivers* (Linux), os quais podem ser inseridos no kernel ou compilados separadamente como módulos. Como estudo de caso foi utilizada a plataforma heterogênea Morpheus [ROS10] executando uma aplicação de detecção de movimentos em vídeo vigilância. A Figura 13 ilustra a arquitetura da plataforma. O processador ARM é o supervisor do sistema e configura comunicação entre os demais recursos computacionais através da *libstnoc*.

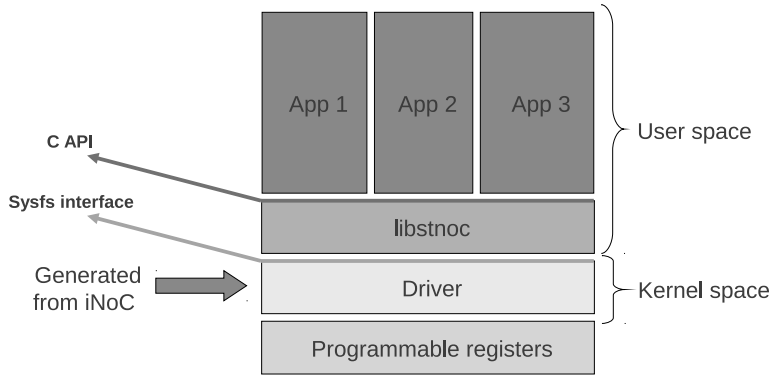


Figura 12 – Camadas de software da abordagem proposta [MOT11].

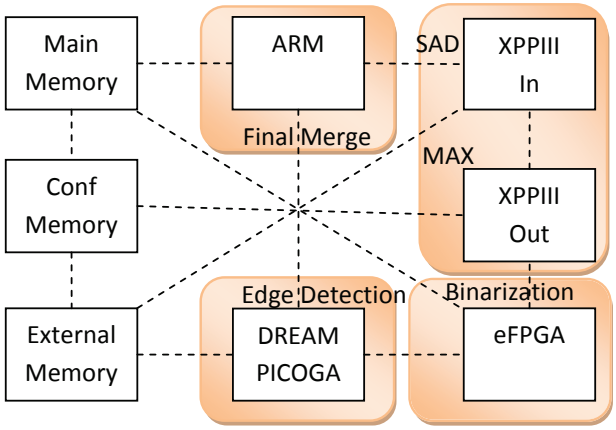


Figura 13 – Arquitetura da plataforma Morpheus [MOT11].

2.2.3 Proposta de Radulescu et al [RAD04]

Este trabalho apresenta uma interface de rede que oferece serviços garantidos e BE através de conexões. A NI oferece os serviços no nível de transporte do modelo de referência ISO-OSI devido a este ser o primeiro nível onde os serviços oferecidos são independentes da implementação da rede. Toda a complexidade da gerência dos serviços está implementada na NI, simplificando a NoC. Em tempo de projeto alguns parâmetros podem ser configurados como número de portas, tipo das portas (mestre ou escravo), número de conexões por portas e nível de serviço das portas. As conexões são unidirecionais com as garantias configuráveis, podendo ser *unicast* ou *multicast*. Elas são baseadas em TDM (*Time Division Multiplexing*), permitindo o compartilhamento de um mesmo enlace por diferentes conexões. Um IP central chamado de módulo de configuração é responsável por configurar todas as conexões de uma determinada aplicação antes desta iniciar sua execução. As interfaces de rede origem e destino de uma conexão são configuradas remotamente pelo módulo de configuração. Ele oferece um controle global sobre o estado das conexões e a disponibilidade de enlaces no sistema. No entanto, tal abordagem eleva o tempo para o estabelecimento de uma conexão, além de poder comprometer a escalabilidade do sistema.

2.2.4 Proposta de Agarwal et al [AGA06]

Este trabalho apresenta uma interface de rede que suporta um serviço de comunicação baseado em prioridades. O objetivo é oferecer diferentes níveis de QoS a fim de atingir os requisitos de aplicações de tempo real. A Figura 14 ilustra a arquitetura da interface de rede.

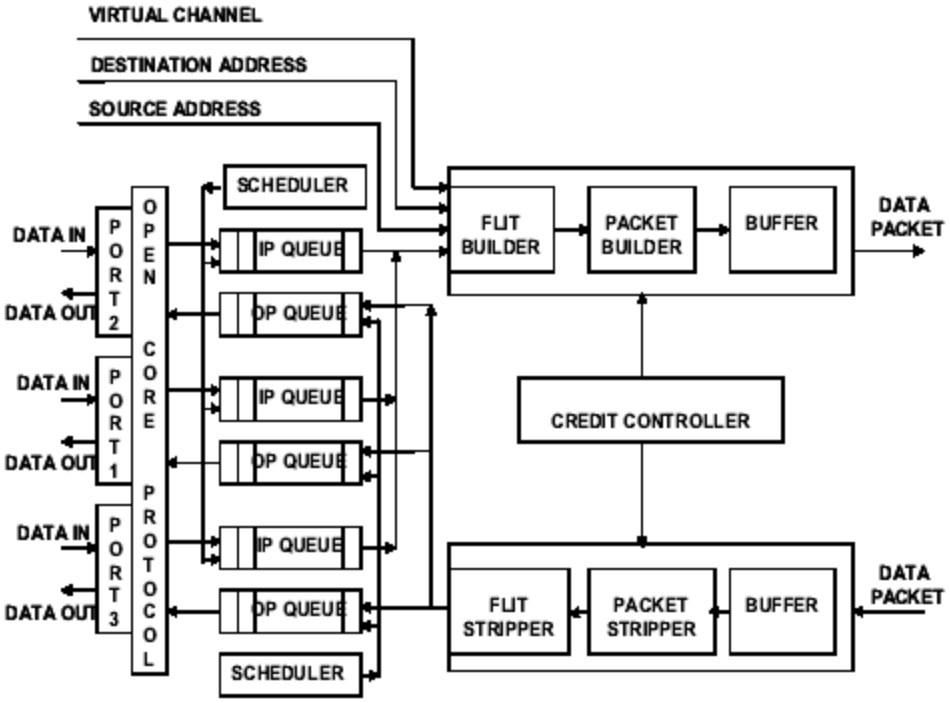


Figura 14 – Arquitetura da interface de rede proposta em [AGA06].

Os dados gerados pelo IP são injetados em uma das três portas bidirecionais (*port 1, port 2 e port 3*). Cada porta corresponde a um nível de prioridade e tem uma fila associada (*queue*) que armazena os dados oriundos do IP. Baseado nas prioridades das portas, o módulo *scheduler* seleciona uma das filas para injetar os pacotes na rede. As prioridades e a política de escalonamento podem ser definidas pelo usuário. Os módulos *flit/packet builder* e *flit/packet stripper* são responsáveis, respectivamente, pelo empacotamento e desempacotamento dos dados. O módulo *credit controller* é responsável pelo controle de fluxo fim-a-fim. Essa interface de rede implementa na realidade um controle de acesso ao meio físico, visto que o serviço de prioridades oferecido se mostra independente da implementação arquitetura de interconexão.

2.2.5 SpiNNaker

O MPSoCs SpiNNaker [YAN09] implementa um controle de admissão adaptativo com o propósito de proporcionar um serviço de comunicação com garantias sobre uma infraestrutura de comunicação assíncrona chamada CHAIN (*CHip Area INterconnect*) [BAI02]. Tal controle de admissão garante limites de latência no acesso a uma memória SDRAM externa e é implementado localmente em cada um dos IPs do sistema, regulando dinamicamente a taxa de injeção. A ideia é manter a CHAIN operando abaixo do ponto de saturação, pois assim a largura de banda agregada tende a ser igualmente compartilhada pelos IPs. O controle de admissão é baseado em um convencional laço de realimentação fechado. A referência do laço indica a latência máxima para obter-se uma resposta da SDRAM e a saída é a taxa de injeção. Resultados obtidos em um cenário com 5 IPs acessando uma SDRAM mostram uma redução da latência média e máxima de 28% e 47,5%, respectivamente, utilizando o controle de admissão proposto.

2.2.6 Proposta de Ahmad et al [AHM06]

Este trabalho descreve uma NoC dinamicamente reconfigurável proposta para MPSoCs adaptativos. A particularidade dessa NoC é a capacidade alterar dinamicamente algumas de suas características juntamente com a mudança nos requisitos de comunicação das aplicações. Sua inteligência está implementada na pilha de protocolos chamada de *Smart Network Stack* (SNS), a qual toma decisões sobre roteamento, modo de chaveamento e tamanho de pacote, dependendo dos dados a serem enviados. As novas características da NoC são incluídas no cabeçalho dos pacotes e processados pelos roteadores. Alguns experimentos foram conduzidos sobre uma topologia torus 4x4 utilizando o simulador de redes NS-2 [NS2]. Entretanto, pela falta de resultados comparativos com uma versão da NoC que não implementa a SNS, os resultados obtidos não deixam claro os ganhos da NoC proposta.

2.2.7 CoMPSoC

O CoMPSoC [HAN09] explora o serviço de comunicação baseado em conexões da NoC Aethereal com o intuito de evitar a interferência entre os fluxos das aplicações que executam simultaneamente sobre a plataforma. O grau de independência do desempenho de uma aplicação, independente da presença ou ausência de outras aplicações na plataforma é chamado de *composability* [KUM08]. Essa propriedade visa garantir que os requisitos das aplicações são atingidos em diferentes cenários. Diferente de arquiteturas BE, o fluxo de projeto do CoMPSoC tem um papel relevante no que diz respeito a *composability*, movendo a complexidade da alocação de recursos em tempo de execução para os tempos de projeto e compilação. Isto impõem restrições que não permitem ao CoMPSoC carregar aplicações dinamicamente, limitando-o a criação de sistemas estaticamente alocados. Como estudo de caso, um decodificador JPEG e um processador de áudio são executados sobre uma instância da plataforma em FPGA composta por três processadores, um codificador de áudio e um controlador de vídeo interconectados pela NoC Aethereal. Os resultados mostram que ambas as aplicações atingem seus requisitos quando executam simultaneamente. Entretanto, não há nenhuma comparação entre o desempenho das aplicações executando sozinhas na plataforma e juntas. Tal comparação poderia evidenciar a efetividade do isolamento entre as aplicações.

2.2.8 Proposta de Kumar et al [KUM08]

Este trabalho apresenta uma abordagem dinâmica para garantir *composability*. O trabalho propõe um gerenciador de recursos central que é responsável por controlar a interferência entre as aplicações, de maneira que elas consigam manter seus requisitos de desempenho. O gerenciador de recursos monitora o desempenho das aplicações através de mensagens de controle oriundas destas. Quando o gerenciador de recursos detecta que uma aplicação X está operando abaixo do requisito devido à interferência de uma aplicação Y, ele suspende a execução da aplicação Y até que a aplicação X recupere seu desempenho. O tempo de reação do sistema depende da frequência das mensagens de controle enviadas pelas aplicações. Um balanço deve ser feito entre o tempo de reação e a carga do sistema, visto que uma alta frequência de mensagens resulta em uma reação mais rápida ao custo da elevação do tráfego. Um estudo de caso envolvendo um decodificador JPEG e um decodificador H263 foi desenvolvido sobre um modelo POOSL (*Parallel Object- Oriented Specification Language*) [THE07] da plataforma. Os resultados mostram que sem o gerenciador de recursos, o decodificador JPEG atinge uma vazão além da mínima e o decodificador H263 fica com a vazão abaixo da desejada. Quando o gerenciador de recursos é empregado, ambas as aplicações atingem os requisitos mínimos de vazão, ainda que abaixo da vazão atingida quando estas executam sozinhas na plataforma. É importante ressaltar que os resultados obtidos a partir do modelo POOSL não consideram a contenção na infraestrutura de

comunicação, a qual pode ser considerada a principal fonte de interferências entre aplicações. Os autores argumentam que a concorrência pelos recursos computacionais já é suficiente para demonstrar a complexidade do problema.

2.2.9 Proposta de Burgio et al [BUR10]

Este trabalho propõe uma nova solução dinâmica e baseada em software para o problema de dimensionamento de *slots* de acesso a barramentos. O escalonamento de *slots* considerado é baseado em uma roda periódica com um *slot* de acesso para cada mestre do sistema. A Figura 15 ilustra a estrutura do sistema. Sempre que a carga de trabalho de um processador muda durante a execução do sistema, ele pode fazer uma requisição para aumentar seu *slot* de acesso ao barramento. As requisições são feitas a um processador mestre (*assigner*), o qual é responsável por coletar todas as requisições, computar os novos *slots* e configurar o árbitro do barramento. O protocolo de requisição é implementado como uma caixa de mensagens na *cache* L2. Os processadores escrevem na caixa de mensagens usando a primitiva `req_mailbox_write()` e o processador mestre lê as mensagens usando a primitiva `req_mailbox_read()`. Tão logo o processador mestre serve as requisições, ele responde aos processadores utilizando a primitiva `res_mailbox_write()` e estes lêem a resposta utilizando a primitiva `res_mailbox_read()`.

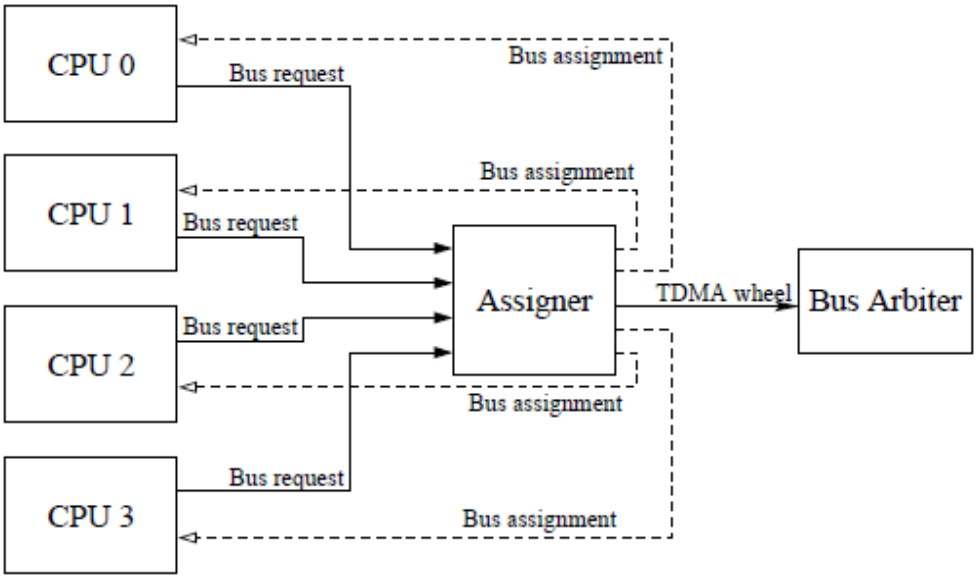


Figura 15 – Estrutura do sistema [BUR10].

O árbitro é programado pelo processador mestre através de uma API específica, a qual proporciona acesso direto aos registradores de configuração do árbitro. A Figura 16 ilustra a configuração dinâmica do arbitro através das primitivas da API. Uma vez que todas as requisições de largura de banda foram tratadas através da primitiva `set_tdma_time_wheel_slot()`, a nova roda de acesso ao barramento é carregada através da primitiva `load_tdm_time_slot()`.

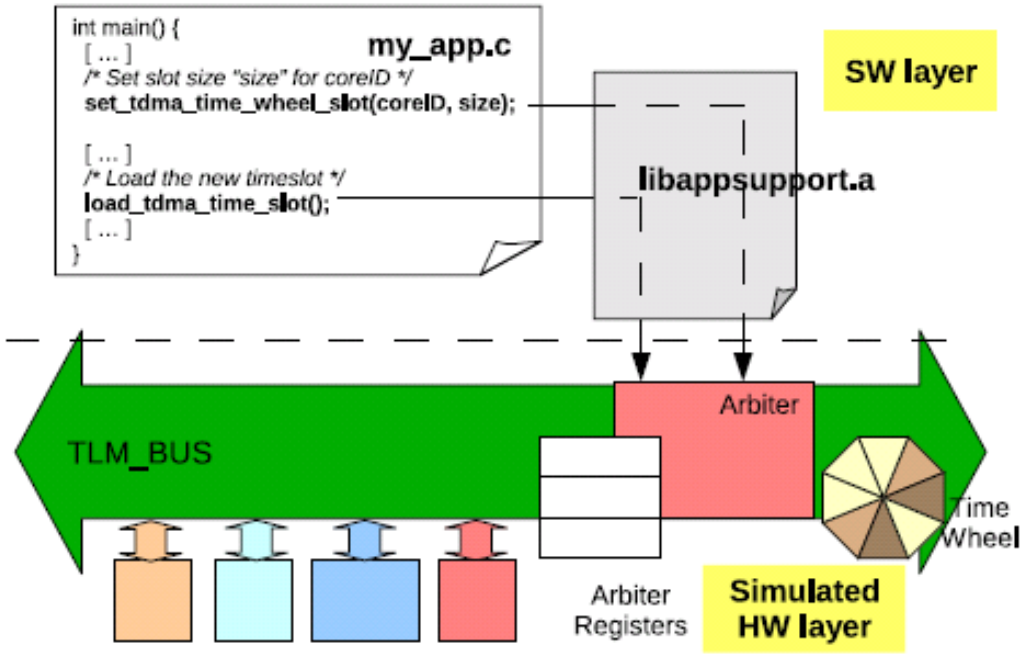


Figura 16 – Configuração do árbitro [BUR10].

2.2.10 Considerações

A Tabela 2 apresenta um resumo das soluções investigadas para garantir os requisitos de desempenho das aplicações. A última linha da Tabela resume a ideia do trabalho aqui proposto.

Em geral, observa-se soluções para oferecer garantias implementadas apenas no nível de hardware, sem o correspondente suporte no nível de software. As soluções propostas são invisíveis do ponto de vista da aplicação e não dão ao projetista a possibilidade gerenciar os fluxos. Alguns trabalhos nem mesmo suportam serviços de comunicação diferenciados no nível de interconexão e contornam esta deficiência através de abordagens como controle de admissão. Outra abordagem observada e que vai na contramão do paradigma de sistemas multiprocessados é o gerenciamento centralizado da infraestrutura de comunicação, também observada em algumas propostas de MPSoCs baseados em NoCs.

Tabela 2 – Resumo das soluções investigadas.

Trabalho	Proposta
GENESYS [OBE10]	Oferece uma infraestrutura de serviços básicos de 4 tipos (sincronização, comunicação, configuração e controle de execução), a partir dos quais serviços de mais alto nível são implementados a fim de personalizar a plataforma para um determinado domínio de aplicação.
[MOT11]	Abordagem para expor os serviços da Spidergon STNoC (gerenciamento de energia, roteamento, segurança e QoS) no nível de software a partir de registradores mapeados em memória acessíveis através da biblioteca *libstnoc*.
[RAD04]	Apresenta uma interface de rede que oferece serviços garantidos e BE através de conexões (TDM), as quais são estabelecidas por um IP central.
[AGA06]	Apresenta uma interface de rede que oferece um serviço de comunicação baseado em prioridades a partir de diferentes portas. Cada porta corresponde a uma prioridade e tem uma fila associada.
SpiNNaker [YAN09]	Propõe um controle de admissão de tráfego adaptativo, a fim de manter a infraestrutura de comunicação operando abaixo do ponto de saturação. O controle é distribuído e baseado em um laço de realimentação fechado que regula a taxa de injeção dos IPs.
[AHM06]	Descreve uma NoC que altera dinamicamente algumas de suas características (roteamento, modo de chaveamento e tamanho de pacote) a fim de adaptar-se a mudanças nos requisitos de comunicação das aplicações.
CoMPSoC [HAN09]	Explora o serviço de comunicação baseado em conexões da NoC Aethereal com o intuito de garantir a independência do desempenho de aplicações que executam simultaneamente sobre a plataforma (*Composability*).
[KUM08]	Propõe um gerenciador central de recursos que controla a interferência entre aplicações que executam simultaneamente sobre a plataforma.
[BUR10]	Oferece uma API de alto nível para controlar dinamicamente o compartilhamento do barramento entre IPs.
Esta Tese	**Desenvolvimento de serviços de comunicação a partir da implementação de mecanismos específicos no nível da NoC e o suporte a estes em software através de uma API de comunicação de alto nível.**

3. HEMPS – MPSOC DE REFERÊNCIA

Este Capítulo descreve o MPSoC acadêmico HeMPS, o qual serve de base para a realização do presente trabalho e diversos outros dentro do grupo de pesquisa GAPH. Todos os serviços de comunicação diferenciados propostos foram implementados e avaliados sobre essa plataforma. Ela pode ser dividia em três partes principais; (*i*) infraestrutura de hardware (arquitetura), (*ii*) microkernel (middleware) e (*iii*) aplicações (software). Cada uma dessas partes será descrita em três seções distintas (3.1 - 3.3). A seção 3.4 encerra o capítulo apresentando as camadas que compõem o sistema.

3.1 Infraestrutura de hardware

HeMPS (*Hermes Multiprocessor System*) [CAR09a] é um MPSoC homogêneo baseado no processador Plasma e na NoC Hermes. Instâncias deste MPSoC são criadas a partir da interconexão de Plasma-IPs (PEs) através da NoC Hermes (infraestrutura de comunicação). Além dos componentes de computação e comunicação, há também uma memória externa chamada repositório de tarefas (*Task Repository*), a qual armazena as tarefas das aplicações a serem executadas. A Figura 17 ilustra uma instância 2x3 do MPSoC mostrando seus principais componentes de hardware. A seguir estes são descritos nas subseções seguintes.

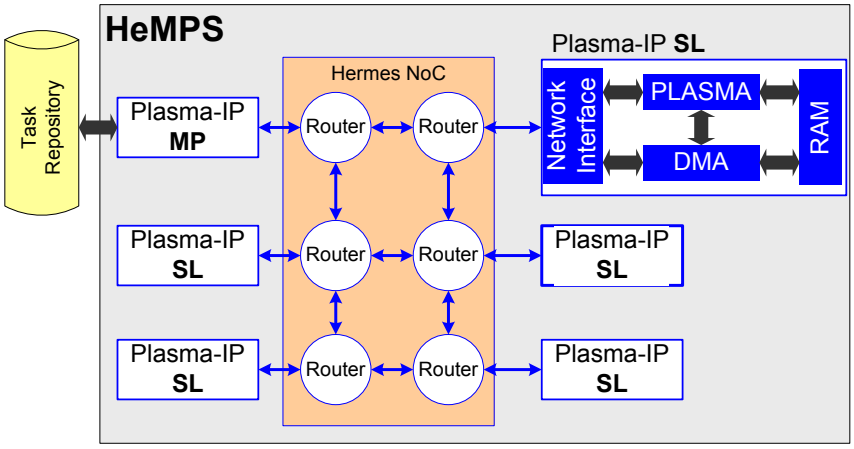


Figura 17 – Instância 2x3 do MPSoC HeMPS[CAR09a].

3.1.1 Plasma-IP

Os Plasma-IPs são os PEs do sistema e podem ser de dois tipos; (*i*) mestre (Plasma-IP MP), responsável pelo gerenciamento dos recursos de computação e (*ii*) escravo (Plasma-IP SL), realiza a computação das aplicações. A gerência dos recursos de computação (Plasma-IP SL) é centralizada, realizada por um único Plasma-IP MP, o qual não computa aplicações. O Plasma-IP é constituído pelos seguintes componentes:

- *Processador Plasma*: processador RISC de 32 bits que implementa um subconjunto de instruções da arquitetura MIPS. Diferentemente do MIPS original, o Plasma

apresenta uma organização de memória Von Neumann. O processador oferece ainda suporte à linguagem C através do compilador *gcc*;

- *Memória privada* (RAM): armazena o microkernel executado pelo processador Plasma. No caso do Plasma-IP SL, além do microkernel, a memória armazena também tarefas das aplicações;
- *Interface de rede* (*Network Interface*): realiza a interface entre o Plasma e a NoC Hermes. É responsável pelo envio e recebimento de pacotes na rede;
- *DMA* (*Direct Memory Access*): desenvolvido para auxiliar o Plasma na troca de mensagens, possibilitando ao processador continuar a computação enquanto o DMA recebe/transmite pacotes;

Para atingir alto desempenho nos PEs, a arquitetura do Plasma-IP visa a separação entre computação e comunicação. A interface de rede e o DMA são responsáveis por enviar e receber pacotes (comunicação) enquanto o processador Plasma executa as tarefas (computação). A memória local (*scratch pad*) é uma RAM dupla porta que permite acesso simultâneo pelo processador e pelo DMA.

3.1.2 NoC Hermes

A NoC Hermes [MOR04] emprega uma topologia malha 2D, chaveamento por pacotes (*wormhole*) e controle de fluxo baseado em crédito. Os roteadores têm *buffers* de entrada, uma lógica de controle (*Switch Control* – arbitragem e roteamento) compartilhada por todas as portas, um *crossbar* interno e até cinco portas bidirecionais (norte, sul, leste, oeste e local). A Figura 18 ilustra uma instância 3x3 da NoC Hermes juntamente da arquitetura interna do roteador.

A porta local estabelece a comunicação entre o roteador e seu IP, sendo as demais portas utilizadas para conectar o roteador aos roteadores vizinhos. A arbitragem das requisições oriundas das portas de entrada é realizada utilizando escalonamento *round-robin* e os pacotes são roteados utilizando o algoritmo de roteamento XY.

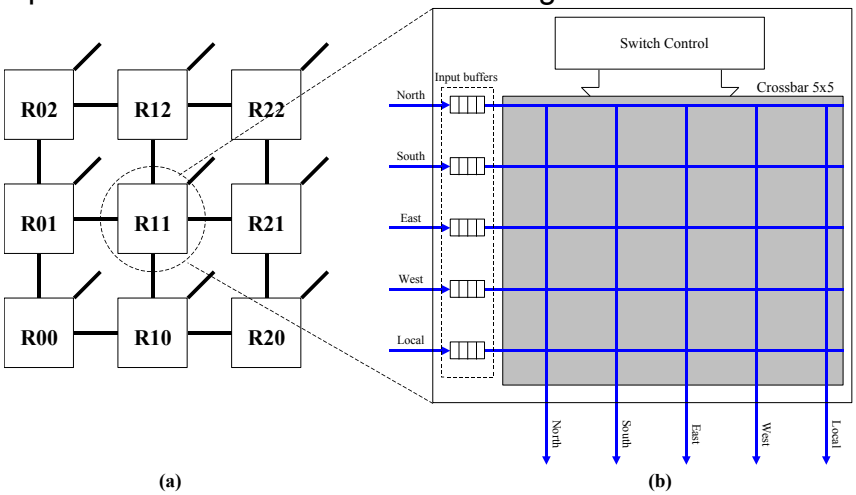


Figura 18 – (a) instância 3x3 da NoC Hermes; (b) arquitetura interna do roteador [CAR08a].

3.1.3 Repositório de tarefas

As aplicações que executam sobre a plataforma são particionadas em diversas tarefas que se comunicam através de troca de mensagens. O repositório de tarefas é uma memória externa ao MPSoC, de grande capacidade em relação às memórias privadas, a qual armazena o código-objeto das tarefas que executarão no sistema. O Plasma-IP MP é o único a ter acesso ao repositório de tarefas (conexão direta) e é responsável por mapear e alocar as tarefas nos Plasma-IP SL. Um conjunto inicial de aplicações é armazenado no repositório em tempo de projeto, no momento da geração da plataforma. Durante a execução do sistema essa memória opera como uma memória ROM, sob o ponto de vista do MPSoC (Plasma-IP MP), podendo entretanto receber novas aplicações dinamicamente, em tempo de execução, através de um *host* externo.

3.2 Microkernel

Cada processador escravo executa um microkernel que suporta a execução de múltiplas tarefas e a comunicação entre estas. A memória local do Plasma-IP SL é dividida em páginas (tamanho fixo e iguais), das quais as primeiras são alocadas pelo microkernel e as demais pelas tarefas (uma por página). O escalonamento de tarefas implementado é o *round-robin* sem prioridades. O microkernel possui uma tabela de tarefas com a localização de todas as tarefas alocadas no sistema (locais e remotas).

As páginas são protegidas pelo microkernel e toda comunicação entre tarefas é realizada através de troca de mensagens. A comunicação é suportada através de um *pipe* global de mensagens e as primitivas `Send()` e `Receive()`, as quais constituem a API de comunicação da plataforma HeMPS. Tal *pipe* é implementado como um vetor na área de memória do microkernel, o qual armazena mensagens enviadas pelas tarefas locais. As primitivas de comunicação são implementadas pelo microkernel e invocadas pelas tarefas através de chamadas de sistemas. Essa abordagem mantém o sistema estruturado em camadas (Seção 3.4), impossibilitando às aplicações o acesso direto à infraestrutura de comunicação.

O modelo de computação que garante a sincronização entre tarefas é baseado em redes de processos de Kahn (*KPN - Kahn Process Networks*) [KAH74]. KPN é um modelo de computação distribuído onde canais de comunicação baseados em FIFOs infinitas (*pipes*) conectam os processos uns aos outros, formando uma rede. Esse modelo se baseia no princípio fundamental de que a leitura do canal de comunicação deve ser bloqueante e a escrita deve ser não-bloqueante.

O protocolo de comunicação adotado é o *read request* [BAG08], o qual garante o controle de fluxo fim-a-fim e o ordenamento de mensagens. Quando uma tarefa qualquer chama a primitiva `Send()`, a mensagem é armazenada no *pipe* local e a computação continua, caracterizando uma escrita não-bloqueante. Ao chamar a primitiva `Receive()`, a mensagem pode ser lida do *pipe* local (comunicação local) ou de um *pipe* remoto

(comunicação remota). Se as tarefas comunicantes estão alocadas no mesmo PE, a mensagem é lida do *pipe* local. Caso contrário, o microkernel envia uma requisição de mensagem para o PE onde a mensagem está armazenada, a tarefa entra em estado de espera (*wait*) e uma nova tarefa é escalonada. Quando a mensagem requisitada é recebida, o microkernel a armazena na página da tarefa destino e muda seu estado para pronta (*ready*). A Figura 19 ilustra a comunicação remota entre tarefas. Na Figura 19(a) é assumido que a tarefa *t2* armazenou uma mensagem no *pipe* (*global_pipe*) endereçada à tarefa *t5* (Send(&msg,t5)), a qual requisita uma mensagem à tarefa *t2* (Receive(&msg,t2)). Na Figura 19(b) a mensagem requisitada (*msg*) é transmitida e armazenada na página da tarefa *t5*. O microkernel garante o ordenamento das mensagens numerando-as à medida que são armazenadas no *pipe*. Note que as primitivas `Send()` e `Receive()` abstraem a localização da tarefa destino/origem da mensagem. É tarefa do microkernel obter a localização das tarefa (tabela de tarefas) durante o processamento das primitivas de comunicação. Essa abordagem permite que o código das aplicações seja independente do mapeamento.

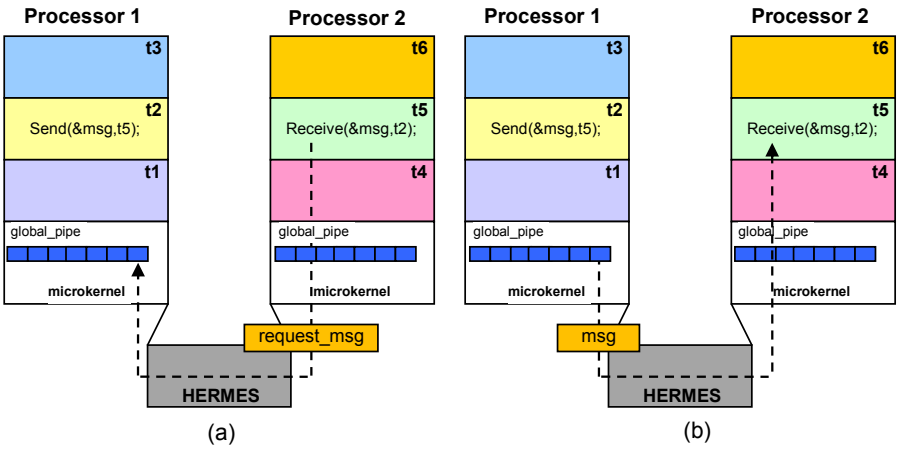


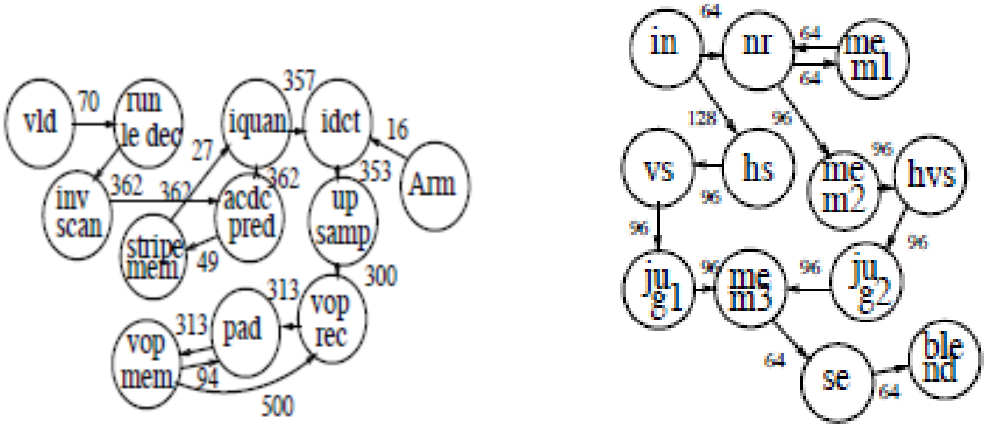
Figura 19 – Comunicação remota entre tarefas [CAR09a].

Esse microkernel foi desenvolvido especialmente para a plataforma HeMPS a partir do sistema operacional disponível na distribuição do processador Plasma. Ele é praticamente todo escrito na linguagem C, tendo apenas o chaveamento de contexto e a inicialização do segmento de dados globais e estáticos (*.bss*) escritos em *assembly*. Suas principais funcionalidades são o suporte a multitarefa, a API de comunicação e o tratamento de interrupções (hardware/software), o que resulta um tamanho reduzido de aproximadamente 15KB, típico de sistemas embarcados.

3.3 Aplicações

As aplicações paralelas desenvolvidas para a plataforma HeMPS são particionadas em tarefas descritas em linguagem C. Cada tarefa tem sua própria função `main()`, podendo ser vista como um processo pesado. Tais aplicações podem ser representadas a partir de grafos dirigidos onde os vértices representam as tarefas e as arestas representam o fluxo de dados. A Figura 20 ilustra duas aplicações de

processamento de vídeo representadas a partir de grafos. Neste exemplo, os vértices dos grafos estão assinalados com a quantidades de dados transferidos entre tarefas em MB/s. Em termos de software, o fluxo de dados representados pelas arestas é implementado a partir das primitivas `Send()` e `Receive()`.



(a) VOPD **(b) MWD**

Figura 20 – Aplicações representadas a partir de grafos dirigidos. (a) *Video Object Plane Decoder*; (b) *Multi-Window Displayer* [JAL04].

Várias aplicações reais têm sido desenvolvidas para a plataforma HeMPS como decodificadores JPEG, MJPEG e ADPCM, multiplicador de matrizes e resolvedor de equações. Além disso, uma abordagem comum é a modelagem de aplicações a partir de *traces* de execução e grafos como os da Figura 20. Nessa abordagem cada tarefa é tipicamente descrita como um laço contendo a sequência recepção, tempo de processamento e envio de dados.

3.3.1 Carga de trabalho dinâmica

Aplicações que executam em MPSoCs possuem muitas vezes uma carga dinâmica, onde as tarefas que as compõem são alocadas sob demanda. Isso implica um número variável de tarefas executando simultaneamente, o qual pode exceder a capacidade dos recursos de processamento disponíveis. Para lidar com esta questão, a plataforma HeMPS suporta carga de trabalho dinâmica, onde somente um subconjunto de tarefas é inicialmente alocado no sistema (mapeamento estático). As demais tarefas são armazenadas no repositório e alocadas sob demanda (mapeamento dinâmico).

A carga de trabalho dinâmica ocorre através da alocação de tarefas sob demanda durante a execução das aplicações. O desenvolvedor de uma aplicação define o subconjunto de tarefas necessário para o início da execução. Tipicamente aquelas que iniciam o fluxo e dados (e.g. *vld* e *arm* na Figura 20(a)). O evento que dispara a alocação de uma nova tarefa é a execução da primitiva `Send()`. Cada vez que uma tarefa chama a primitiva `Send()`, o microkernel verifica na tabela de tarefas se a tarefa destino da mensagem está alocada no sistema. Se a tarefa está alocada, a mensagem é armazenada no *pipe*. Caso contrário, o microkernel primeiro envia ao processador mestre

uma requisição de tarefa e em seguida armazena a mensagem no *pipe*. Esse processo é transparente ao nível de tarefa e é ilustrado na Figura 21.

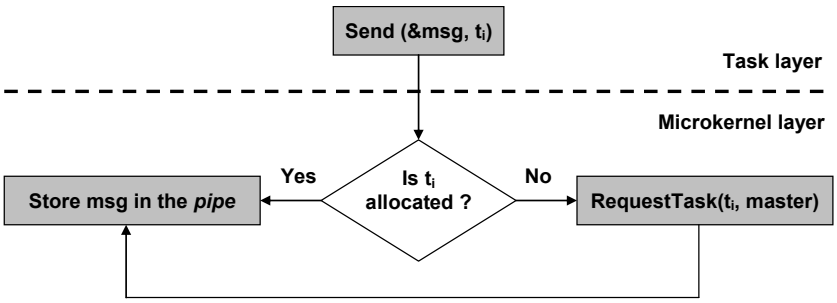


Figura 21 – Processo de alocação de tarefas sob demanda [CAR09a].

Ao receber uma mensagem de requisição de tarefa, o processador mestre procura a tarefa no repositório e a transmite a um processador escravo que tenha uma página disponível. Concluída a transmissão, o processador mestre notifica todos os escravos sobre a localização da nova tarefa alocada.

3.3.2 Mapeamento

A plataforma HeMPS suporta dois tipos de mapeamento; (*i*) estático e (*ii*) dinâmico. O mapeamento estático é realizado em tempo de projeto, onde é definido em qual PE uma determinada tarefa deve ser executada. Todas as tarefas de uma aplicação podem ser mapeadas estaticamente ou apenas um subconjunto destas, sendo as demais mapeadas dinamicamente pelo processador mestre.

O mapeamento dinâmico é caracterizado pela carga de trabalho dinâmica. O algoritmo utilizado para selecionar o PE onde uma nova tarefa deve ser alocada é baseado na heurística LEC-DN (*Low Energy Consumption – Dependences Neighborhood*) [MAN11a]. Essa heurística tem como principal objetivo a redução na energia de comunicação. O PE selecionado pelo algoritmo é o mais próximo (em número de *hops*) dos PEs onde já estão alocadas tarefas com as quais a nova tarefa deve se comunicar.

Decisões de mapeamento podem influenciar drasticamente o desempenho do sistema, uma vez que a má distribuição das tarefas sobre os PEs pode intensificar o tráfego na NoC, acarretando interferência entre fluxos de diferentes aplicações. Por isso, diversas heurísticas visam alocar tarefas comunicantes em PEs próximos (poucos *hops* de distância), de maneira que diferentes aplicações ocupem diferentes regiões da plataforma, reduzindo assim o número de enlaces compartilhados. Entretanto, a natureza dinâmica de plataformas como a HeMPS, onde novas aplicações entram e outras deixam o sistema durante a execução, dá margem à fragmentação devido à dispersão dos PEs. A Figura 22 ilustra um exemplo de sistema fragmentado, considerando uma instância 3x5 da plataforma HeMPS. Cada cor representa uma aplicação. O mapeamento destas foi realizado dinamicamente utilizando a heurística proposta em [MAN11b]. Observa-se uma pequena fragmentação das aplicações vermelha e verde.

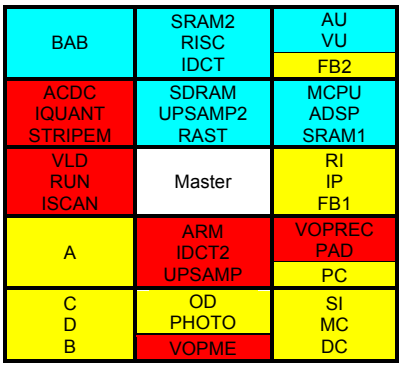


Figura 22 - Exemplo de sistema fragmentado. As aplicações verde e vermelha apresentam tarefas dispersas nos sistema. [MAN11b].

Para resolver esse problema, uma das possibilidades é a utilização da técnica de migração de tarefas (fora do escopo da presente Tese). Essa técnica se faria útil para aproximar tarefas comunicantes, migrando uma tarefa que se comunica com outra de um PE distante para um mais próximo [ALM10]. Entretanto, sua eficiência depende de PEs disponíveis que habilitem uma redistribuição de tarefas. Caso o sistema esteja fragmentado e com praticamente todos os PEs alocados, essa técnica não induz a melhora de desempenho desejada.

Por outro lado, as colisões entre fluxos de diferentes aplicações, devido à fragmentação do sistema, podem ser minimizadas a partir de serviços de comunicação que ofereçam algum tipo de garantia. Tais serviços possibilitam a reserva de recursos de comunicação à aplicações com requisitos de desempenho, deixando os demais recursos livres para serem compartilhados entre as outras aplicações. Dessa maneira, mesmo que as tarefas comunicantes estejam dispersas na plataforma, o fluxo de dados se mantém contínuo, evitando o aumento da latência devido à bloqueios oriundos de regiões congestionadas.

3.4 Camadas do sistema

Tipicamente sistemas computacionais são divididos em camadas com diferentes níveis de abstração a fim de gerenciar sua complexidade. Cada camada tem uma funcionalidade específica e se comunica com as camadas adjacentes. Dessa maneira, cada camada se serve dos serviços oferecidos pelas camadas inferiores e fornece serviços às camadas superiores. A Figura 23 ilustra as diferentes camadas que compõem a plataforma HeMPS e as entidades correspondentes. Notar que as camadas *Task*, *OS* e *Transport* são replicadas nas várias instâncias do Plasma-IP.

A seguir são descritas cada uma das camadas.

- Camada de aplicação (*Applications*): contém as aplicações a serem mapeadas no sistema. Cada aplicação é representa por um grafo dirigido podendo ter suas tarefas distribuídas em vários PEs;

- Camada de tarefa (*Task*): contém a descrição das aplicações a partir de tarefas que se comunicam usando as primitivas implementadas pela API;
- Camada de sistema (*OS*): é composta por um conjunto de *drivers* responsáveis pelo carregamento de tarefas, gerenciamento da memória local e do DMA, empacotamento/desempacotamento de mensagens. Além disso, realiza o escalonamento de tarefas (*round-robin*) e implementa a API;
- Camada de transporte (*Transport*): executa a injeção/recepção de pacotes e controle de fluxo;
- Camada de rede (*Network*): responsável pela transmissão de pacotes sem perda de dados.

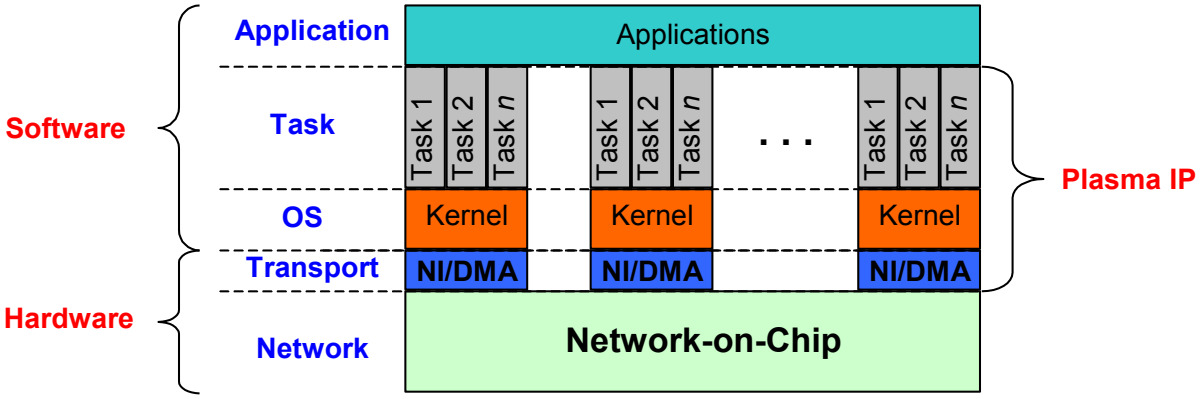


Figura 23 - Camadas da plataforma HeMPS e suas entidades [CAR09b].

O MPSoC HeMPS segue claramente a tendência dos futuros sistemas embarcados multiprocessados (Tabela 1), apresentando uma arquitetura estruturada e suporte básico em software para o desenvolvimento de aplicações paralelas. Esse sistema será utilizado como base para a implementação dos serviços de comunicação diferenciados propostos nessa Tese. A implementação de tais serviços segue uma abordagem *bottom-up*, partindo-se da implementação de mecanismos específicos no nível de rede e criando suporte a estes até o nível de software.

4. SERVIÇOS DE COMUNICAÇÃO BASEADOS EM PRIORIDADES E CONEXÕES

Uma vez revisada uma série de trabalhos relacionados, posicionando a presente Tese em relação ao estado da arte (Capítulo 2), e apresentada a arquitetura do MPSoC de referência (Capítulo 3), este Capítulo apresenta a primeira contribuição desta Tese, a qual compreende os serviços de comunicação baseados em prioridades e conexões. Tais serviços foram implementados e avaliados sobre as plataformas HeMPS e HS-Scale (Seção 2.1.4). A plataforma HeMPS é a plataforma alvo deste trabalho, no entanto, a utilização da plataforma HS-Scale neste Capítulo deve-se ao período de doutorado sanduíche realizado no laboratório LIRMM (Montpellier/França), durante o qual esses serviços vinham sendo desenvolvidos.

Os serviços de comunicação implementados pelas NoCs podem ser de dois tipos: melhor esforço (*BE – Best-Effort*) ou serviço garantido (*GS – Guaranteed Service*). Serviços de comunicação do tipo BE garantem a entrega de todos os pacotes entre um par origem/destino, contudo não proporcionam nenhum tipo de garantia em relação a métricas de desempenho como vazão ou latência. Esse tipo de serviço atribui a mesma prioridade a todos os pacotes, impossibilitando um tratamento diferenciado para fluxos de aplicações com algum tipo de restrição temporal. O termo QoS refere-se à capacidade de uma rede de distinguir fluxos e tratá-los de maneira diferenciada, proporcionando diferentes níveis de qualidade. Portanto, serviços de comunicação do tipo BE são inadequados para satisfazer requisitos de QoS de aplicações com rígidas restrições temporais.

Para atender a requisitos de desempenho, a rede comumente implementa mecanismos específicos em sua arquitetura. Este Capítulo apresenta dois mecanismos que fornecem, respectivamente, suporte aos serviços de comunicação baseado em prioridades e conexões: (*i*) alocação de recursos baseado em prioridades fixas e (*ii*) chaveamento por circuito. O serviço de comunicação baseado em prioridades proporciona garantias flexíveis aos fluxos de aplicações onde certas variações no desempenho da comunicação não são relevantes. Já o serviço de comunicação baseado em conexões proporciona garantias rígidas a partir da reserva exclusiva de recursos da rede por todo o tempo da comunicação. Conexões garantem uma comunicação fim-a-fim livre de qualquer tipo de interferência por parte de outras comunicações que ocorrem em paralelo na rede. Tais serviços de comunicação proporcionam ao projetista acesso aos mecanismos que controlam a alocação de recursos da NoC.

A partir de serviços de comunicação que oferecem algum tipo de garantia pode-se explorar QoS a fim de evitar a interferência entre os fluxos de aplicações que executam simultaneamente sobre uma mesma plataforma, caracterizando *composability* no nível da rede. *Composability* é uma propriedade que visa garantir a satisfação dos requisitos das

aplicações independente das demais aplicações executando simultaneamente no sistema. Visto que aplicações com diferentes requisitos de desempenho podem executar simultaneamente, *composability* deve ser garantida àquelas com restrições temporais (e.g. aplicações de tempo real). *Composability* é frequentemente atingida em tempo de projeto ou através de políticas de escalonamento específicas em sistemas operacionais de tempo real. Na indústria automotiva e aeroespacial, por exemplo, *composability* é frequentemente atingida não compartilhando recursos (computação/comunicação) entre as aplicações [HAN09].

No presente trabalho, *composability* é atingida no nível da rede a partir da exploração da QoS por meio dos serviços de comunicação propostos. Entretanto, no nível de processamento, o compartilhamento dos PEs por diferentes tarefas pode acarretar interferência entre aplicações. Para minimizar essa interferência e atingir *composability* também no nível de processamento, no final deste Capítulo é apresentado um escalonamento preemptivo com prioridades, o qual permite que tarefas pertencentes a aplicações com restrições temporais tenham acesso privilegiado ao processador.

4.1 Mecanismos de prioridades e chaveamento por circuito na NoC Hermes

Originalmente a NoC Hermes emprega uma topologia malha bidimensional construída a partir de roteadores com até cinco portas bidirecionais. Os roteadores localizados nas bordas da malha têm menos portas que aqueles localizados entre as bordas. Das cinco possíveis portas, quatro (norte, sul, leste e oeste) são utilizadas para a conexão com roteadores vizinhos e uma (local) conecta o roteador ao IP. Os roteadores possuem *buffers* de entrada para o armazenamento temporário de *flits*, um *crossbar* que conecta as portas de entrada às portas de saída e um módulo chamado de *Switch Control*, responsável pela arbitragem das portas de entrada, roteamento e controle do *crossbar* (Figura 18).

A principal modificação feita na arquitetura do roteador para suportar um serviço de comunicação baseado em prioridades foi a duplicação dos canais físicos (portas bidirecionais) [CAR09b]. O roteador resultante suporta até dez portas bidirecionais. A partir dessa nova arquitetura um mecanismo de prioridades pode ser usado na alocação dos canais físicos. A Figura 24 ilustra a arquitetura do novo roteador. Os dois canais físicos em uma mesma direção são numerados de 0 a 1. O mesmo vale para porta local.

A replicação dos canais físicos é uma abordagem que vem se popularizando recentemente [GIL10][YOO10][KAK11]. Tal abordagem é empregada principalmente como uma alternativa a canais virtuais. A implementação de ambas abordagens implica custos de área relacionados aos *buffers* de entrada e ao *crossbar*. No entanto, a abordagem de canais virtuais necessita ainda um custo extra relativo à multiplexação dos canais físicos (TDM), o que torna seu custo total em área superior à replicação dos canais físicos, considerando o número de canais virtuais e físicos iguais [CAR07][CAR08b]. Além

disso, a largura de banda agregada do roteador é diretamente proporcional ao fator de replicação dos canais físicos, ao passo que o aumento do número de canais virtuais não altera a largura de banda. A potência dissipada por ambas as abordagens é similar [YOO10].

Canais virtuais foram introduzidos por Dally e Seitz [DAL87], visando resolver o problema de *deadlock* em redes que implementam *wormhole* e não visando desempenho, apesar de contribuírem para uma melhor utilização dos canais físicos. Essa abordagem é uma herança das redes de computadores, onde há uma limitação significativa no número de conexões que conectam dois elementos comunicantes. Nesse caso a multiplexação do meio físico torna-se a alternativa mais adequada. Por outro lado, dentro de um chip, a abundância de espaço disponível para conexões favorece a replicação dos canais físicos.

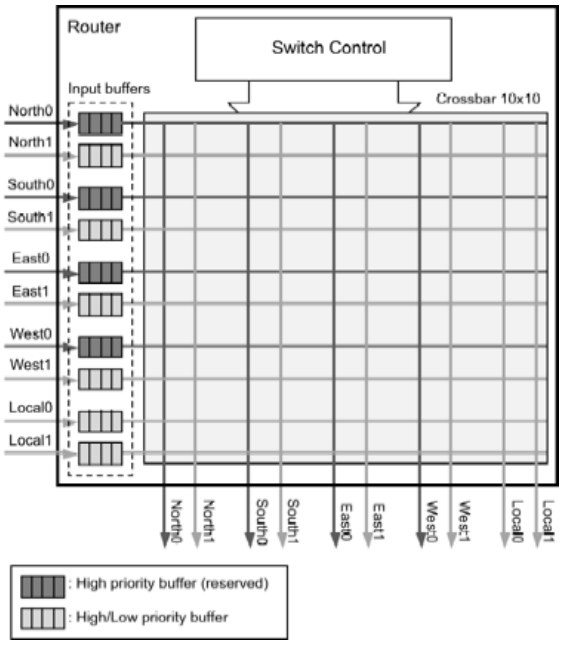


Figura 24 – Arquitetura do roteador com dez portas bidirecionais [CAR09b].

O mecanismo de prioridades implementado nesse roteador é baseado em prioridades fixas. Duas classes de tráfego são distinguidas pela NoC: (*i*) pacotes com prioridade alta e (*ii*) pacotes com prioridade baixa. Um canal físico (canal 0) é reservado para transmitir exclusivamente pacotes com prioridade alta, ao passo que o outro canal (canal 1) pode transmitir pacotes com prioridade alta ou baixa. O compartilhamento de um dos canais físicos (canal 1) entre as duas classes de tráfego aumenta o suporte da rede a pacotes com prioridade alta, pois possibilita a transmissão de dois fluxos de alta prioridade simultaneamente na mesma direção. O mecanismo de prioridades oferece um serviço de comunicação diferenciado a tráfegos de alta prioridade através da reserva virtual de recursos (recursos em cinza escuro na Figura 24). Todavia, quando mais de dois fluxos de alta prioridade competem por um mesmo caminho, a interferência entre eles nessa implementação é inevitável. De fato, NoCs que empregam algum tipo de mecanismo de prioridades tendem a atuar como NoCs BE à medida que o tráfego de alta

prioridade se intensifica [MEL06]. Esse mecanismo explora a tolerância de algumas aplicações à variações modestas no desempenho da rede, onde a perda de alguns *deadlines* não é um problema (e.g. *soft real time*).

O serviço de comunicação baseado em conexões é suportado a partir do modo de chaveamento por circuito. Esse chaveamento coexiste juntamente com o chaveamento por pacotes, de maneira que a NoC Hermes suporta simultaneamente ambos os chaveamentos. Uma conexão física é estabelecida entre um único par origem/destino e os recursos da rede permanecem alocados durante todo o tempo da comunicação. As conexões são unidirecionais e estabelecidas/encerradas pela origem da comunicação através de pacotes de controle (dois *flits*).

Essa abordagem de chaveamento por circuito é a mais simples que pode ser implementada, considerando a arquitetura da NoC Hermes. Ela exige alterações mínimas na arquitetura do roteador e apresenta um baixo custo em área. Tal simplicidade deve-se ao fato de que o chaveamento por circuito foi implementado sobre o chaveamento por pacotes. No chaveamento por pacotes, os *flits* de *payload* de um pacote comum seguem o caminho alocado pelo *flit* de cabeçalho. Esse caminho permanece alocado até o último *flit* de *payload* ser transmitido. Na abordagem de chaveamento por circuito implementada, o pacote de controle que estabelece uma conexão é o *flit* de cabeçalho de um pacote comum e as mensagens transmitidas pela conexão representam o *payload* desse pacote. O pacote de controle que desaloca a conexão representa o último *flit* de *payload* de um pacote comum.

Toda a largura de banda do caminho entre origem e destino é alocada pela conexão, permitindo às aplicações atingir a máxima vazão possível sem qualquer tipo de interferência proveniente de outras comunicações. Visto que a alocação total da largura de banda pode subutilizar os recursos quando a vazão das aplicações é baixa, conexões são restritas somente ao canal 0. Assim o canal 1 está sempre disponível para transmitir pacotes usando chaveamento por pacotes.

As portas locais do roteador (local 0 e local 1) são utilizadas pelo PE dependendo do serviço de comunicação. A porta local 0 é utilizada pelo serviço baseado em conexões enquanto a porta local 1 serve o serviço baseado em prioridades. Pacotes injetados na NoC pela porta local 0 são transmitidos a partir do modo de chaveamento por circuito, enquanto os pacotes injetados na porta local 1 são transmitidos utilizando chaveamento por pacotes. Um PE pode manter uma conexão através da porta local 0, enquanto envia pacotes pela porta local 1 utilizando chaveamento por pacotes. Visto que a porta local 0 permite a conexão entre apenas dois PEs, o objetivo dessa abordagem é possibilitar aos PEs conectados comunicarem-se com outros PEs utilizando chaveamento por pacotes através da porta local 1, mantendo a conexão estabelecida.

A NoC diferencia os pacotes injetados a partir de campos específicos no cabeçalho. Quando um pacote chega a um roteador, o módulo *Switch Control* (Figura 24)

extrai informações do cabeçalho para executar o algoritmo de roteamento e a alocação/desalocação dos canais físicos. A Figura 25 ilustra a estrutura do pacote. O primeiro *flit* é o cabeçalho do pacote e os demais que o seguem compõem o *payload*. O último *flit* do pacote é sinalizado por um sinal chamado *eop* (*end-of-packet*). O *flit* de cabeçalho contém os seguintes campos:

- *Service* (4 bits): Tipo do pacote (e.g. estabelecimento/encerramento de conexão e modo de chaveamento);
- *Unused* (3 bits): bits não utilizados atualmente;
- *P* (1 bit): prioridade do pacote (‘0’: alta; ‘1’: baixa);
- *Target* (8 bits): indica o endereço do roteador destino do pacote.

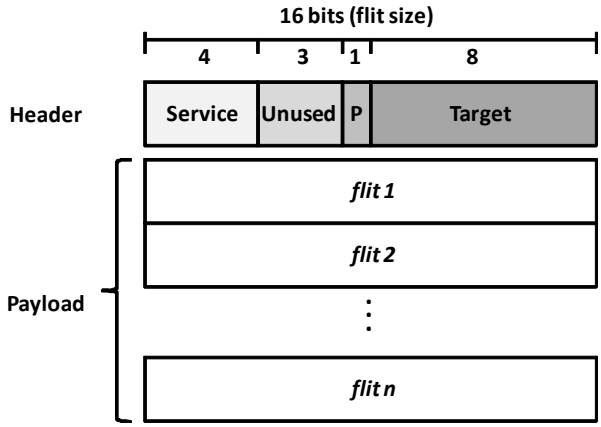


Figura 25 – Estrutura do pacote da NoC Hermes para suportar serviços de comunicação baseado em prioridades e conexões [CAR11].

4.2 Integração dos mecanismos no nível de software

Ambos os mecanismos (alocação de recursos baseada em prioridades e chaveamento por circuito) foram integrados nas APIs de comunicação das plataformas HeMPS e HS-Scale, dando origem aos *serviços de comunicação baseados em prioridades e conexões*. Esses serviços de alto nível permitem o gerenciamento de recursos da NoC Hermes em software. Em ambas as plataformas, o sistema operacional (microkernel) é responsável pela ligação entre a API (nível de tarefas) e os mecanismos da NoC (nível de rede). Esta abordagem possibilita ao programador explorar QoS em alto nível a partir do código fonte das tarefas que compõem as aplicações. Ambas as plataformas dão suporte à programação em alto nível através da linguagem C.

As APIs de comunicação das plataformas HeMPS e HS-Scale implementam duas primitivas básicas de comunicação entre tarefas: (*i*) `Send()/MPISend()` e (*ii*) `Receive()/MPIRcv()`, respectivamente. Elas oferecem um serviço de comunicação básico de troca de mensagens do tipo BE. Para suportar os novos mecanismos implementados na NoC Hermes, um novo parâmetro foi adicionado às primitivas `Send()` e `MPISend()` e duas novas primitivas foram incluídas na API. O parâmetro *priority* foi adicionado às primitivas `Send()` e `MPISend()` a fim de dar suporte ao serviço de

comunicação baseado em prioridades. As novas primitivas `Connect()` e `Close()` dão suporte ao serviço de comunicação baseado em conexões. Os protótipos das primitivas aparecem na Listagem 1. O parâmetro *target*, presente nas primitivas, refere-se à tarefa destino. A localização das tarefas (mapeamento) é transparente ao programador e é responsabilidade do sistema operacional localizar as tarefas baseado em tabelas de mapeamento. Essa abordagem oferece um nível mais alto de abstração ao programador durante o desenvolvimento das aplicações.

Listagem 1 – Primitivas para suporte aos serviços de comunicação baseados em prioridades e conexão.

```
/* Envia uma mensagem especificando a prioridade (HeMPS)*/
void Send(Message *msg, int target, int priority);

/* Envia uma mensagem especificando a prioridade (HS-Scale) */
void MPISend(int target, int fifo, void *data, int size, int priority);

/* Estabelecimento de conexão (HeMPS e HS-Scale) */
void Connect(int target);

/* Encerramento de conexão (HeMPS e HS-Scale) */
void Close();
```

O parâmetro *priority* pode assumir três valores definidos como constantes: (*i*) HIGH; (*ii*) LOW e (*iii*) GT. Os dois primeiros valores referem-se à prioridade da mensagem a ser enviada e dão suporte ao serviço de comunicação baseado em prioridades. A prioridade da mensagem é especificada no nível de tarefa, desce toda a pilha de protocolo até o nível de rede e seta o bit de prioridade (*P* - Figura 25) no cabeçalho do(s) pacote(s) que compõem a mensagem. Esse processo efetivamente faz a ligação entre os níveis de tarefa e de rede. O valor GT (*Guaranteed Throughput*) indica que a mensagem deve ser enviada pela conexão previamente estabelecida. Neste caso, o valor GT não seta o bit de prioridade no cabeçalho, mas o campo de serviço (*Service* - Figura 25). Uma conexão entre um par origem/destino é estabelecida pelo PE origem da comunicação através da primitiva `Connect()`, a qual cria um pacote de estabelecimento de conexão e o envia ao PE destino. Esse pacote vai alocando recursos da rede (canal 0) pelos quais é transmitido a fim de criar a conexão. Uma vez estabelecida a conexão, as mensagens são enviadas através desta por meio das primitivas `Send()/MPISend()` atribuindo o parâmetro *priority* com o valor GT. Ao fim da comunicação, a conexão é encerrada pelo PE origem através da primitiva `Close()`, a qual cria um pacote de encerramento de conexão e o envia ao PE destino. Esse pacote é transmitido pela conexão e vai desalocando os recursos da rede conforme avança em direção ao PE destino.

Visto que uma conexão aloca toda a largura de banda do caminho entre origem e destino, cada roteador suporta apenas uma conexão com a porta local 0. Portanto, apenas uma tarefa por PE pode estabelecer uma conexão. Esta conexão permanece ativa até a chamada da primitiva `Close()`, mesmo que a tarefa que estabeleceu a conexão não esteja escalonada. O roteador não é capaz de preemptar a conexão quando a tarefa origem da comunicação perde o processador. Se outra tarefa no mesmo PE

tentar estabelecer uma conexão, ela ficará bloqueada até que a conexão seja encerrada. O mesmo vale para uma tarefa que já tem uma conexão estabelecida, ao tentar estabelecer uma segunda conexão. Essa limitação habilita somente uma tarefa por PE comunicar-se utilizando o serviço de conexões, enquanto as demais devem utilizar o serviço de prioridades. Portanto, deve-se evitar alocar no mesmo PE tarefas que se servem do serviço de comunicação baseado em conexões. Isso deve ser incluído como uma restrição na heurística de mapeamento.

4.3 Avaliação

Esta Seção apresenta os resultados obtidos a partir de simulações das plataformas HeMPS e HS-Scale. Ambas as plataformas estão totalmente descritas em VHDL RTL sintetizável e possuem modelos SystemC com precisão de ciclo do processador Plasma e sua memória privada para fins de aceleração da simulação. Os fluxos de perturbação utilizados nos experimentos podem ser caracterizados como diversos tráfegos comuns em MPSoCs como atualização de blocos de *cache*, *debug*, carga e migração de tarefas.

4.3.1 HeMPS

Esta Seção avalia o serviço de comunicação baseado em prioridades sobre a plataforma HeMPS. O objetivo é verificar o impacto que o controle priorizado sobre os recursos da rede tem na vazão dos fluxos. Dois cenários foram avaliados em uma instância 4x4 da plataforma. O primeiro cenário simula um MPSoC homogêneo, onde todos os fluxos são gerados por Plasma-IPs. O segundo cenário simula um MPSoC heterogêneo com geradores de tráfego perturbando os fluxos gerados pelos Plasma-IPs.

4.3.1.1 MPSoC homogêneo

O objetivo deste primeiro experimento é mostrar o comportamento da NoC conforme o número de fluxos com alta prioridade vai aumentando. A Figura 26 ilustra a distribuição espacial dos fluxos utilizados no experimento, segundo o algoritmo de roteamento Hamiltoniano. Esse algoritmo é detalhado no Capítulo 5 (Seção 5.1.1). F1 e F2 são fluxos que exigem QoS, enquanto os demais (F3, F4, F5 e F6) são fluxos de perturbação. Um fluxo é composto por rajadas de pacotes (524 *flits*/pacote) intercaladas por tempos ociosos. A taxa de injeção dos fluxos F1 e F2 é 30% da largura de banda do enlace e os fluxos de perturbação têm uma taxa de injeção média igual a 18,5%. Essas taxas de injeção garantem a saturação de regiões onde há competição de recursos, como entre os roteadores 5 e 6, visando uma clara demonstração do funcionamento do serviço de comunicação baseado em prioridades. Todos os fluxos são gerados por aplicações sintéticas executando sobre os processadores Plasma.

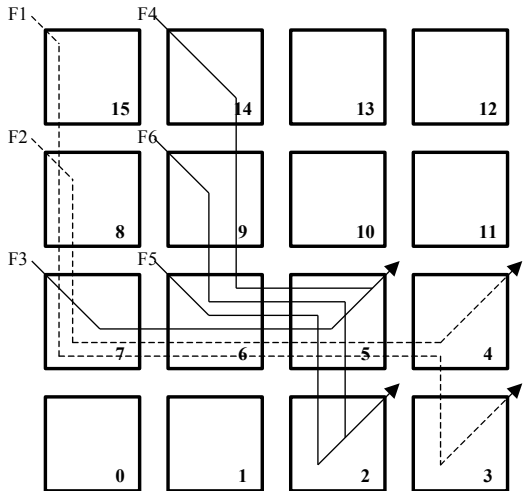


Figura 26 – Distribuição espacial dos fluxos [CAR09b].

A Figura 27 apresenta as vazões dos fluxos observadas nos destinos 2, 3, 4 e 5 (Figura 26) conforme o número de fluxos com alta prioridade cresce (*Number of QoS flows*). Visto que cada um dos pares de fluxos F3-F4 e F5-F6 tem o mesmo destino, a vazão de cada par foi calculada como a vazão total no destino dividida por dois, pois os fluxos têm a mesma taxa de injeção. Inicialmente, somente os fluxos que exigem QoS (F1 e F2) enviam pacotes com alta prioridade e a vazão de ambos é próxima da taxa de injeção (30%). O serviço de comunicação baseado em prioridades proporciona um isolamento parcial entre fluxos de alta e baixa prioridade. Pode-se observar que os fluxos de perturbação quase não interferem na vazão dos fluxos que exigem QoS, apesar da grande concorrência pelos canais físicos entre os roteadores 5 e 6 (Figura 26 – cinco fluxos concorrem pelos dois canais físicos). A partir dessa situação, conforme aumenta o número de fluxos de perturbação que passa a enviar pacotes de alta prioridade, a vazão dos fluxos F1 e F2 decai. Quando todos os seis fluxos (F1, F2, F3, F4, F5 e F6) passam a enviar pacotes de alta prioridade, a NoC começa a operar como uma NoC BE. Isso ocorre porque todos os fluxos são tratados igualmente visto que possuem à mesma prioridade. Nessa condição, as vazões dos fluxos F1 e F2 reduziram de 29,91% e 28,83% para 21,95% e 20.91%, respectivamente.

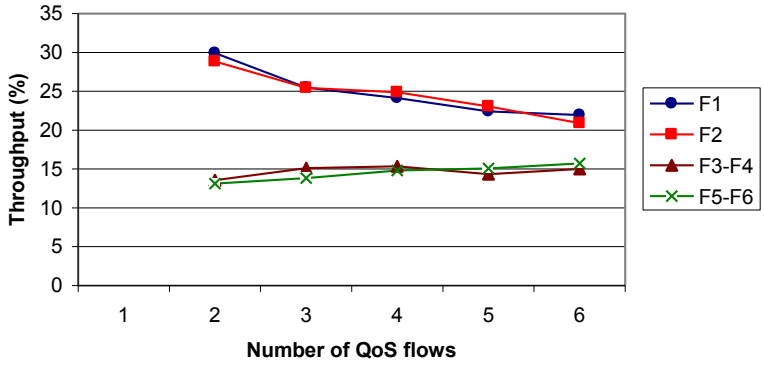


Figura 27 – Deterioração dos fluxos F1 e F2 [CAR09b].

Esse experimento mostra que:

- NoCs que oferecem apenas serviços de comunicação do tipo BE subutilizam os recursos da rede por falta de controle sobre estes;
- Quando o número de fluxos de alta prioridade não excede o suporte à QoS projetada, o serviço de comunicação baseado em prioridades é capaz de oferecer garantias flexíveis aos fluxos que exigem QoS. Nesse experimento, a NoC dá garantias a até dois fluxos concorrendo por caminhos em comuns;
- Abundância de recursos e alta largura de banda não são suficientes para oferecer garantias aos fluxos. Garantias dependem do controle sobre os recursos da NoC a partir de mecanismos especiais adicionados à sua arquitetura;
- Mesmo não excedendo a largura de banda total da NoC, os fluxos podem causar interferências entre si.

4.3.1.2 MPSoC heterogêneo

O segundo experimento tem por objetivo mostrar que o serviço de comunicação baseado em prioridades pode eficientemente proporcionar garantias de vazão mesmo em situações onde os tráfegos de perturbação têm altas taxas de injeção. A Figura 28 ilustra a distribuição espacial dos fluxos. F1 é um fluxo que exige QoS e é gerado por uma processador Plasma, enquanto os fluxos de perturbação F2 e F3 são gerados por geradores de tráfego. A taxa de injeção do fluxo F1 é 30% da largura de banda do enlace e os fluxos F2 e F3 tem uma taxa de injeção igual a 100%. Os geradores dos tráfego emulam IPs especializados com altas taxas de injeção. Ainda que uma taxa de 100% possa parecer irreal, ela pode ser facilmente obtida se os IPs operam em frequências superiores à NoC.

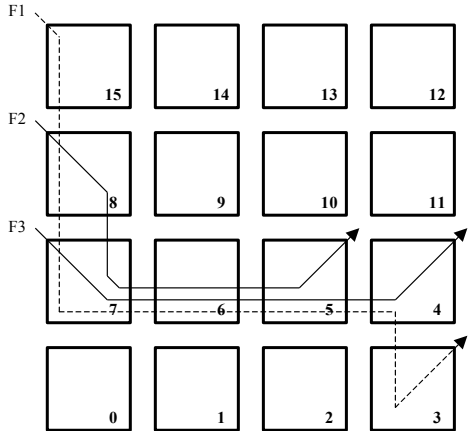


Figura 28 – Distribuição espacial dos fluxos [CAR09b].

Neste cenário, considerando uma NoC BE com canais físicos duplicados, a vazão do fluxo F1 observada no destino 3 é igual a 10,5%, aproximadamente 1/3 da taxa de injeção. Esse baixo desempenho deve-se a dois motivos: (*i*) a taxa de injeção dos fluxos F2 e F3 é muito alta, o que dificulta o acesso do fluxo F1 aos recursos compartilhados; (*ii*) todos fluxos são tratados igualmente pela NoC, independente dos seus requisitos.

Considerando uma NoC com suporte a mecanismo de prioridades e definindo a prioridade do fluxo F1 como alta (via API), enquanto a prioridades dos fluxos F2 e F3 é definida como baixa, F1 atinge uma vazão igual a 29,8% da largura de banda do enlace. O mecanismo de prioridades cria uma reserva virtual, onde certos recursos (canal 0) estão disponíveis exclusivamente para fluxos de alta prioridade.

4.3.2 HS-Scale

Esta Seção utiliza os serviços de comunicação baseados em prioridades e conexões a fim de atingir *composability* sobre a plataforma HS-Scale. O objetivo é garantir os requisitos de desempenho da aplicação alvo quando esta é mapeada na plataforma juntamente com outras aplicações que concorrem por recursos de comunicação em comum. Os experimentos foram realizados em uma instância 4x4 da plataforma HS-Scale misturando aplicações reais e sintéticas. A aplicação real (alvo) é um decodificador áudio/vídeo com restrições temporais composto por sete tarefas. A Figura 29 ilustra o grafo de tarefas do decodificador. O *pipeline* de vídeo é executado por um decodificador MJPEG particionado em três tarefas (MJ1, MJ2 e MJ3) e o *pipeline* de áudio é composto por um decodificador ADPCM (AD) e um filtro do tipo FIR. Uma tarefa inicial chamada SPLIT demultiplexa os *streams* compactados (áudio/vídeo) e os envia aos respectivos *pipelines*, enquanto a tarefa JOIN sincroniza os *streams* descompactados. A vazão mínima requerida pela aplicação é 30 *frames*/s (vídeo) e 32000 amostras/s (áudio estéreo) sincronizados. As aplicações sintéticas não têm restrições temporais e apenas emulam acesso a dispositivos de saída e memórias, os quais são emulados por tarefas em software.

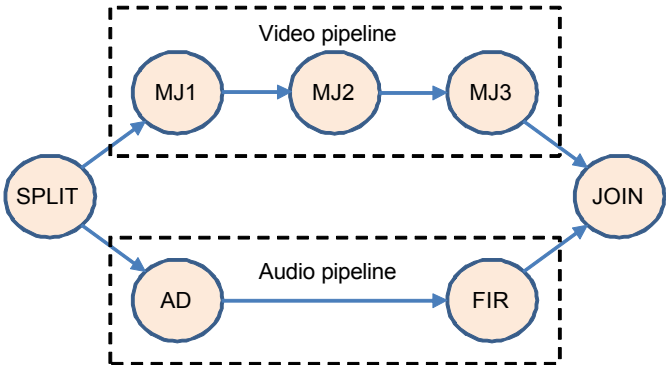


Figura 29 – Grafo de tarefas do decodificador áudio/vídeo [CAR11].

Inicialmente, o decodificador é mapeado sozinho na plataforma (Figura 30) e as tarefas se comunicam utilizando o serviço de comunicação baseado em prioridades. A vazão obtida é igual a 31,13 *frames*/s e é usada como referência. Visto que neste experimento o *pipeline* de vídeo produz um volume de dados significativamente maior que o *pipeline* de áudio, no decorrer do texto a vazão da aplicação é expressa apenas em *frames*/s, no entanto ela inclui também as amostras de áudio, pois a vazão é mensurada na tarefa JOIN.

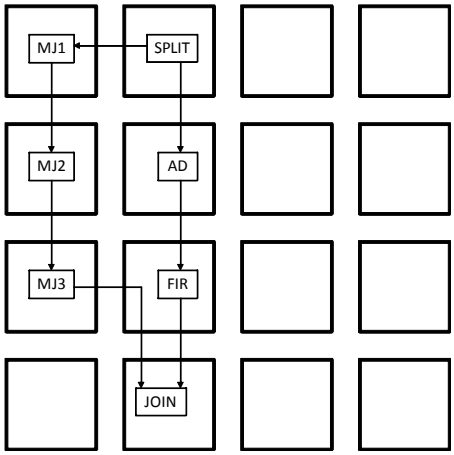


Figura 30 – Mapeamento ótimo do decodificador áudio/vídeo [CAR11].

O mapeamento ótimo mostrado na Figura 30 é comumente obtido durante a inicialização do sistema, quando todos os recursos estão disponíveis. Em sistemas onde pode haver carga dinâmica de aplicações, como *smart phones* e *tablets*, elas são frequentemente alocadas e removidas da plataforma, resultando na dispersão (fragmentação) dos recursos disponíveis (Seção 3.3.2 - Figura 22). Consequentemente, um mapeamento ótimo é cada vez mais difícil de ser atingido em tempo de execução, a menos que o sistema suporte algum tipo de remapeamento dinâmico (e.g migração de tarefas). As novas aplicações alocadas tendem a compartilhar os recursos do sistema com as aplicações já alocadas. A Figura 31 ilustra uma situação onde não foi possível atingir um mapeamento ótimo do decodificador devido à dispersão dos recursos. Quatro novas aplicações foram adicionadas ao sistema, cada uma com duas tarefas: (*i*) T1→MEM, (*ii*) T2→MEM, (*iii*) T3→OUT e (*iv*) T4→OUT. O mapeamento resultante é suscetível à interferência entre as aplicações devido à concorrências pelos recursos da rede.

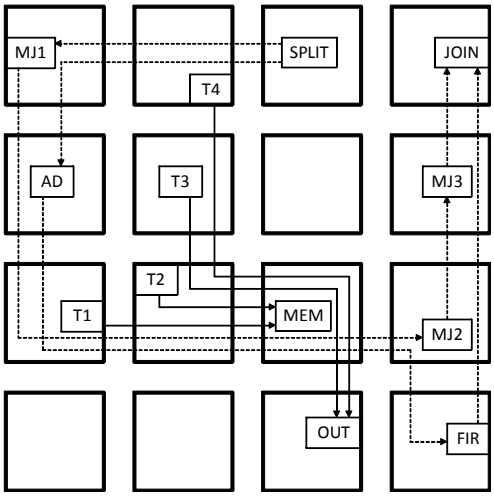


Figura 31 – Decodificador áudio/vídeo perturbado pelas novas aplicações [CAR11].

A Tabela 3 apresenta a vazão do decodificador variando o número de aplicações com fluxos de alta prioridade no mapeamento apresentado na Figura 31 para seis diferentes cenários. Quando somente os fluxos do decodificador têm alta prioridade

(cenário S1), a vazão obtida é apenas 1,6% menor que a de referência, mantendo-se acima dos 30 *frames*/s requeridos pela aplicação. O serviço de comunicação baseado em prioridades efetivamente garante aos fluxos da aplicação alvo acesso privilegiado aos recursos da rede, evitando interferências por parte dos demais fluxos. No entanto, as limitações desse tipo serviço tornam-se evidentes na proporção em que o número de fluxos de alta prioridade concorrendo por recursos de comunicação em comum aumenta. Isso pode ser observado na degradação da vazão nos cenários S2 a S5, quando as tarefas T4, T3, T1 e T2 passam uma a uma a enviar pacotes com alta prioridade. A vazão é reduzida em 52% quando os fluxos de todas aplicação tem alta prioridade (cenário S5). Nessa situação, a NoC passa a operar em modo BE e sua alta largura de banda não é suficiente para garantir os requisitos da aplicação alvo. A partir desse cenário, onde o serviço de prioridades já não oferece mais garantias para atingir o requisito mínimo de 30 *frames*/s, o serviço de conexões é empregado. Visto que o fluxo de dados do *pipeline* de vídeo tem uma taxa mais elevada que o de áudio, ele foi escolhido para usar o serviço de conexões. Este último cenário (S6) é mostrado na última linha da Tabela 3, onde somente as tarefas do decodificador de vídeo comunicam-se através de conexões, enquanto as demais se mantém enviando pacotes de alta prioridade. O serviço de comunicação baseado em conexões estendeu as garantias oferecidas pelo sistema e ainda elevou a vazão da aplicação alvo em 3,5% em relação à vazão de referência. Os resultados obtidos nos cenários S1 e S6 mostram a eficiência dos serviços de comunicação propostos no gerenciamento dos fluxos a partir do nível de software.

Tabela 3 – Resultados de vazão para 6 diferentes cenários correspondentes ao mapeamento da Figura 31 [CAR11].

	Flows			Throughput
	Low Priority	**High Priority**	**GT Connection**	
S1	T1,T2,T3,T4	Audio, Video	-	30.62 frames/s
S2	T1,T2,T3	Audio, Video, T4	-	23.8 frames/s
S3	T1,T2	Audio,Video,T3,T4	-	16.76 frames/s
S4	T2	Audio, Video, T1,T3,T4	-	16.08 frames/s
S5	-	Audio, Video, T1,T2,T3,T4	-	14.81 frames/s
S6	-	Audio,T1,T2,T3,T4	Video	32.24 frames/s

A transmissão de um determinado fluxo através da NoC pode modificar a sua taxa de injeção, introduzindo valores variáveis de latência e resultando na perda de certos *deadlines* no IP destino. Essa variação instantânea da latência é chamada de *jitter* e deve ser minimizada em aplicações com restrições temporais. Em uma transmissão sem *jitter*, os pacotes de um determinado fluxo são transmitidos dentro de intervalos de tempo constantes, como mostra a Figura 32(a). Uma discrepância na latência dos pacotes altera

o intervalo de tempo no qual estes são transmitidos, caracterizando o *jitter* (Figura 32(b)).

A Figura 33 e a Figura 34 mostram o *jitter* do *pipeline* de vídeo correspondente aos cenários de referência (Figura 30) e S1 a S6 (Tabela 3). O eixo X representa o intervalo de tempo entre os blocos decodificados (*i* na Figura 32) que chegam à tarefa JOIN e o eixo Y representa a quantidade de blocos decodificados que chegam em cada intervalo de tempo.

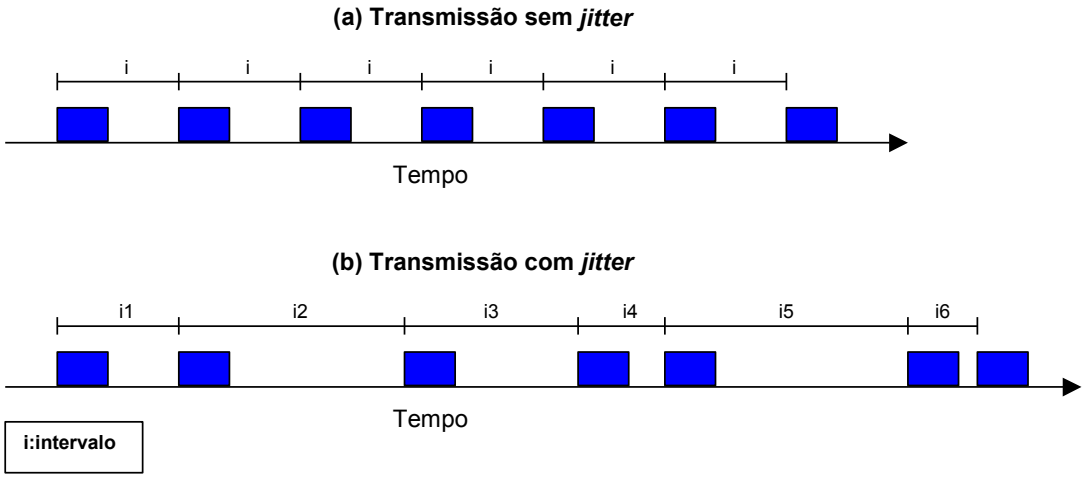


Figura 32 – Exemplos de transmissões sem e com *jitter*.

A Figura 33 mostra o *jitter* para os cenários; referência (Figura 30 – mapeamento ótimo); alta prioridade para os fluxos das tarefas do decodificador áudio/vídeo (cenário S1); e conexões para os fluxos das tarefas do *pipeline* de vídeo (cenário S6). Nesses três cenários, a maior parte dos blocos decodificados chega à tarefa JOIN dentro dos intervalos (média e desvio padrão): 191 ± 55, 192 ± 59, 189 ± 57 kilo ciclos de *clock* para os cenários de referência, S1 e S6 respectivamente. Além de vazões semelhantes, a similaridade entre as três curvas mostra que os serviços propostos garantem *jitter* equivalente mesmo na presença de fluxos de perturbação.

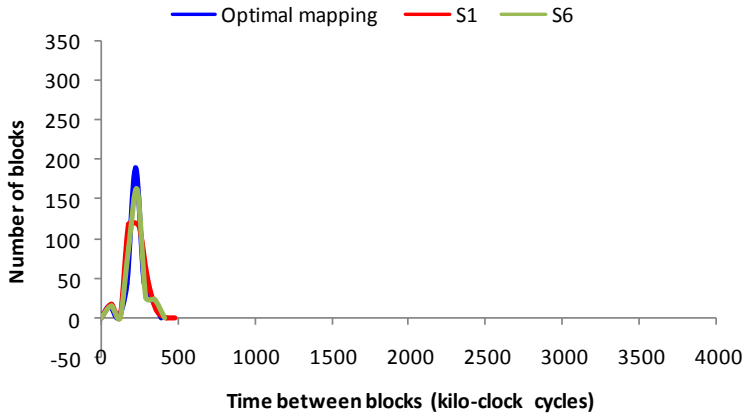


Figura 33 – *Jitter* para os cenários onde o requisito da aplicação é garantido [CAR11].

A Figura 34 mostra o *jitter* para os cenários S2 a S5. Os blocos decodificados chegam à tarefa JOIN dentro dos intervalos: 254 ± 212, 416 ± 1042, 366 ± 1119, 382 ± 1132 kilo ciclos de *clock* para cada cenário. Os fluxos de perturbação gerados pelas

tarefas T1 a T4 interferem nos fluxos do decodificador elevando o tempo médio entre os blocos e o *jitter*. Estes são responsáveis pela degradação da vazão observada na Tabela 3.

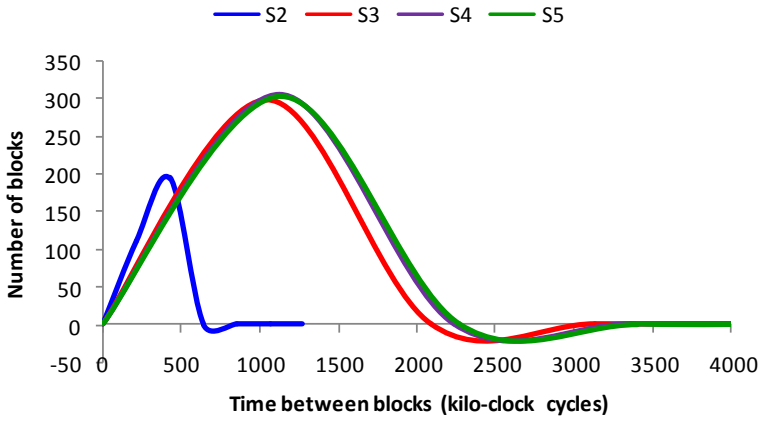


Figura 34 – *Jitter* para os cenários onde o requisito da aplicação não é atingido [CAR11].

A Figura 35 mostra a área, medida em LUTs e FFs, para os principais componentes da NPU: (*i*) PE, contendo o processador Plasma, interface de rede e memória local; (*ii*) roteador da NoC. A área de uma NPU mapeada no dispositivo Virtex5 LX330 sem suporte aos serviços de comunicação propostos é igual a 4016/1997 LUTs/FFs. A versão da NPU suportando tais serviços tem uma área igual a 5652/2384 LUTs/FFs, tendo um acréscimo de LUTs e FFs de 40,74%/19,38%, respectivamente. Esse acréscimo de área é proveniente do aumento na área do roteador (165%), devido à duplicação dos canais físicos, que por sua vez dobra sua largura de banda. O impacto da área do roteador na área da NPU pode ser reduzido se um processador mais complexo que Plasma for utilizado. O CoMPSoC [HAN09], por exemplo, reporta uma área de 57882 LUTs para um MPSoC com 3 processadores, SRAM e dispositivos de entrada/saída para áudio/vídeo.

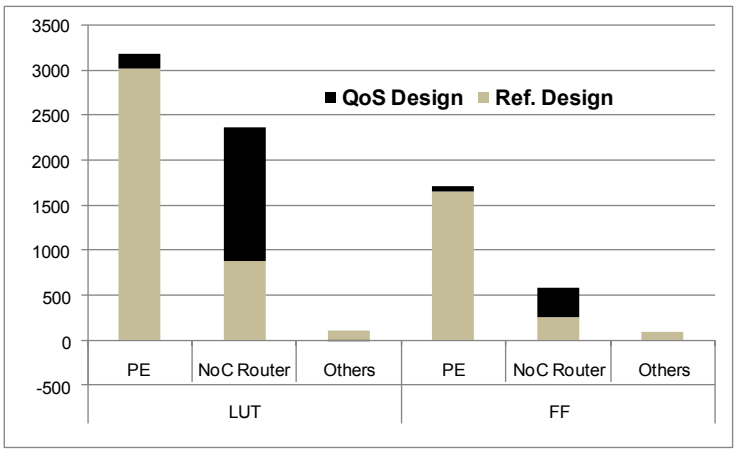


Figura 35 – Número de LUTs e FFs para os principais componentes da NPU. As barras pretas representam o acréscimo de área para suportar os serviços de comunicação. Dispositivo: Virtex5 LX330.

4.4 Escalonamento preemptivo com prioridades

A partir dos serviços de comunicação baseados em prioridades e conexões pode-se evitar a interferência entre os fluxos das aplicações graças ao controle sobre os recursos da NoC que estes proporcionam. Assim é possível atingir *composability* no nível da rede. Entretanto, quando diferentes tarefas compartilham o mesmo processador, o tempo de processamento é dividido entre elas e isso pode reduzir o desempenho das aplicações levando-as à perda de *deadlines*. Para controlar a interferência entre tarefas alocadas no mesmo PE, pode-se empregar um mecanismo de prioridades no algoritmo de escalonamento. A ideia é aumentar a prioridade das tarefas pertencentes a aplicações com restrições temporais. Obviamente, quando tarefas de aplicações com restrições temporais compartilham o mesmo PE, tal solução pode ser ineficiente, devendo-se evitar essa situação durante o mapeamento.

Como dito anteriormente, o microkernel da plataforma HeMPS (Seção 3.2) escalona as tarefas utilizando *round-robin*. Nesse escalonamento o processador é compartilhado igualmente pelas tarefas. Elas são colocadas em uma lista circular que é percorrida regularmente, alocando o processador para cada uma por um intervalo de tempo fixo chamado *time slice*. Ao final do *time slice*, a tarefa em execução é preemptada e uma nova tarefa é escalonada.

O mecanismo de prioridades, aqui incluído no *round-robin*, permite ao desenvolvedor indicar, em tempo de projeto, a prioridade e o *time slice* de cada tarefa. Nessa abordagem, uma tarefa com prioridade mais alta que a tarefa em execução pode preemptá-la. A tarefa preemptada tem o restante do seu *time slice* armazenado, o qual é restaurado quando ela for escalonada novamente. A Figura 36 ilustra um exemplo do funcionamento do algoritmo *round-robin* incrementado com prioridades e *time slice* definidos por tarefa.

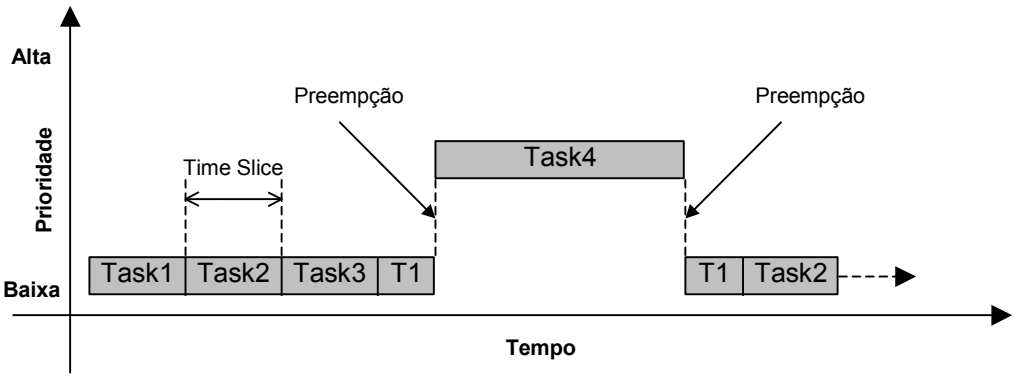


Figura 36 – *Round-robin* incrementado com prioridades e *time slice* por tarefa [LI03].

No exemplo da Figura 36 o processador é compartilhado por quatro tarefas, três com prioridade e *time slice* iguais (*Task1*, *Task2* e *Task3*), e uma com prioridade mais alta e *time slice* maior (*Task4*). Note que a tarefa *Task1* é escalonada novamente após a tarefa *Task3* devido ao fato de que a tarefa *Task4* estava em estado de espera (*wait*) no momento do escalonamento (e.g. após uma chamada de `Receive()`). No instante em

que tarefa *Task4* está pronta para executar (*ready*) ela preempta a tarefa em execução (*Task1*), a qual volta a executar o restante do seu *time slice* após a tarefa *Task4* terminar o seu ou entrar em estado de espera novamente.

4.4.1 Avaliação

Para avaliar o mecanismo de prioridades implementado sobre o escalonamento *round-robin*, utilizou-se novamente como aplicação alvo o decodificador áudio/vídeo (Figura 29). Como perturbação, utilizou-se um multiplicador de matrizes que implementa o algoritmo de Fox [FOX87]. Tal multiplicador é composto por dez tarefas (TM, T00, T01, T02, T10, T11, T12, T20, T21 e T22). Mais detalhes sobre essa aplicação serão apresentados no Capítulo 6. A Figura 37 ilustra o mapeamento das duas aplicações sobre uma instância 3x4 da plataforma HeMPS. Note que todas as tarefas do decodificador compartilham os PEs com tarefas do multiplicador de matrizes. Os fluxos das duas aplicações (omitidos da figura) são isolados no nível da rede através do serviço de comunicação baseado em prioridades.

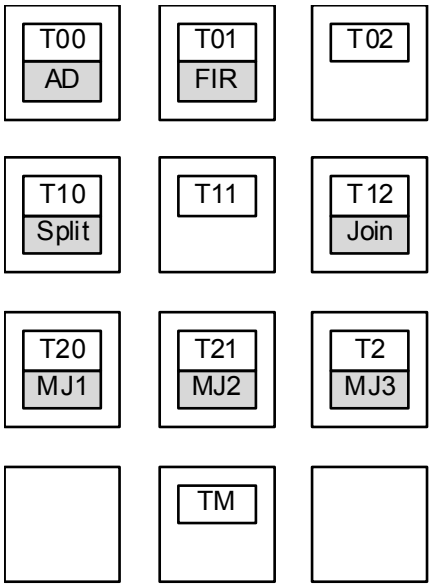


Figura 37 – Mapeamento do decodificador áudio/vídeo e do multiplicador de matrizes sobre uma instância 3x4 da plataforma HeMPS.

Nesse experimento, a aplicação alvo decodifica 40 *frames* (incluindo áudio) enquanto o multiplicador realiza 180 multiplicações de matrizes 33x33. A Tabela 4 apresenta o tempo de execução da aplicação alvo para decodificar 40 *frames* e a vazão, variando a concorrência por recursos de computação (PEs). Inicialmente, o decodificador é mapeado sozinho na plataforma com suas tarefas dispostas na mesma configuração da Figura 37 (cenário S1), correspondendo a um mapeamento ótimo (livre de concorrência por recursos). No cenário S2, as tarefas da aplicação alvo compartilham os PEs com as tarefas do multiplicador de matrizes. Nesse cenário todas as tarefas de ambas as aplicações têm a mesma prioridade e o mesmo *time slice*. Logo, o escalonamento corresponde a um *roud-robin* simples, alocando o processador igualmente para as tarefas

de ambas as aplicações. Tal compartilhamento gera uma interferência das tarefas do multiplicador de matrizes sobre as da aplicação alvo, aumentando o tempo de execução desta última em 41,8% e reduzindo a vazão em 30%. Esse cenário é análogo ao cenário S5 da Tabela 3, onde a NoC opera em modo BE, tratando igualmente todos os fluxos. A fim de reduzir tal interferência, a prioridade das tarefas da aplicação alvo foram aumentadas em relação às tarefas do multiplicador de matrizes (cenário S3). Assim, as tarefas da aplicação alvo podem preemptar as tarefas do multiplicador de matrizes sempre que estiverem prontas para executar. Em relação ao cenário S1, no cenário S3 o tempo de execução da aplicação alvo aumentou 2,6% e a vazão reduziu 4,6%. Para minimizar a interferência entre as aplicações, no cenário S4 o *time slice* das tarefas da aplicação alvo é o dobro do *time slice* das tarefas do multiplicador de matrizes. Como resultado, o isolamento entre as aplicações é quase total e o desempenho da aplicação alvo é muito semelhante ao obtido quando esta executa sozinha sobre a plataforma.

Tabela 4 – Desempenho da aplicação alvo variando a concorrência por recursos de computação (PEs). Os valores de vazão superiores ao experimento anterior (Tabela 3), devem-se à utilização da plataforma HeMPS, a qual possui um melhor desempenho na comunicação entre as tarefas.

	Prioridade das tarefas (0 é a mais alta)		*Time slice* das tarefas (ciclos de *clock*)		Tempo de execução	Vazão
	Decodificador	Multiplicador	Decodificador	Multiplicador		
S1	100	-	16384	-	930 ms	43 frames/s
S2	100	100	16384	16384	1319 ms	30 frames/s
S3	90	100	16384	16384	955 ms	41 frames/s
S4	90	100	32768	16384	931 ms	43 frames/s

A Figura 38 mostra o intervalo de tempo entre *frames* decodificados pela aplicação alvo em cada um dos 4 cenários da Tabela 4. Nos cenários onde a aplicação alvo executa sozinha sobre a plataforma (S1) e onde ela está isolada (S3 e S4), observa-se que um *frame* é decodificado a cada ±2300000 ciclos de *clock*. No cenário S2, onde a aplicação alvo sofre uma interferência do multiplicador de matrizes, inicialmente observa-se que um *frame* é decodificado a cada ±3700000 ciclos de *clock*. Após um certo tempo tem-se um *frame* é decodificado a cada ±2300000 ciclos de *clock*. Essa variação evidencia o momento em que multiplicador terminou de multiplicar as 180 matrizes e foi desalocado, deixando a aplicação alvo executar sozinha sobre a plataforma.

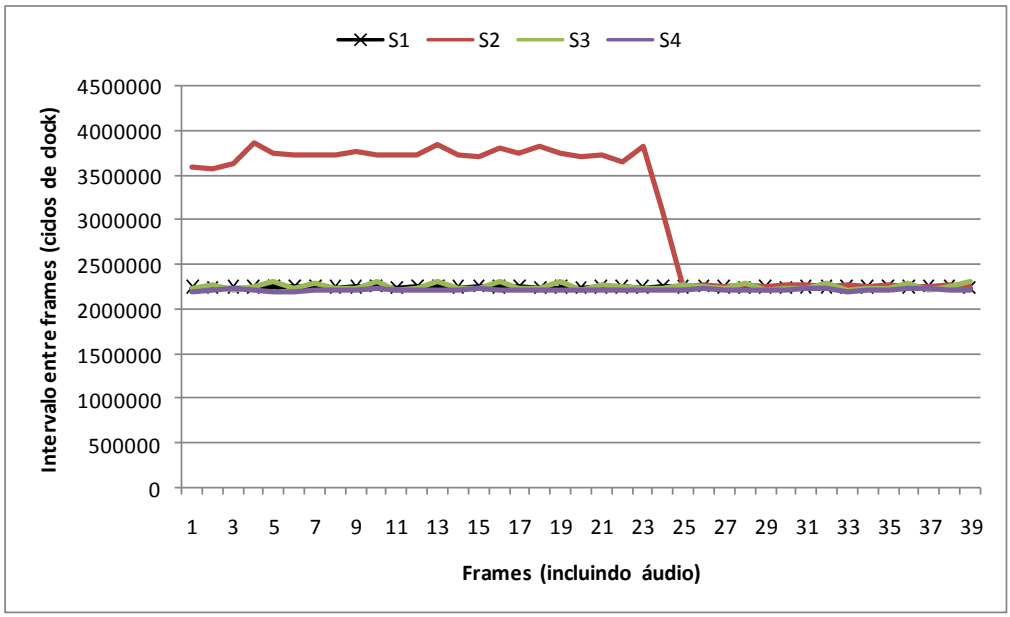


Figura 38 – Intervalo de tempo entre *frames* decodificados.

4.5 Considerações

A grande disponibilidade de recursos (e.g. *buffers*, canais físicos, largura de banda) na implementação de NoCs deve ser gerenciada de maneira inteligente a partir de mecanismos específicos que ofereçam controle sobre a alocação destes. Esses mecanismos dão suporte à criação de serviços de comunicação diferenciados que devem ser utilizados com a finalidade de combinar, da melhor maneira possível, a alocação de recursos de rede com os requisitos de comunicação das aplicações. Em MPSoCs baseados em NoCs, o acesso a tais serviços através da API de comunicação aumenta a programabilidade do sistema oferecendo um controle mais amplo sobre a plataforma.

O controle sobre os recursos da NoC oferecido pelos serviços de comunicação baseados em prioridades e conexões possibilita a minimização da interferência de fluxos BE sobre os fluxos de aplicações com restrições temporais. Entretanto em ambientes multitarefa, tal interferência deve ser considerada também no nível de processamento. Portanto, o tempo de processamento dos PE deve também ser gerenciado através do escalonamento de tarefas, a fim de possibilitar a combinação entre a alocação de recursos de comunicação e processamento.

5. SERVIÇO DE COMUNICAÇÃO COM ROTEAMENTO DIFERENCIADO

Este Capítulo apresenta a segunda contribuição desta Tese, o serviço de comunicação com roteamento diferenciado. O termo diferenciado refere-se à possibilidade que o serviço oferece de aplicar um algoritmo de roteamento adaptativo ou determinístico a uma determinada mensagem. Após uma revisão sobre o estado da arte, acredita-se que esse serviço é uma contribuição original tanto na área de NoCs quanto de MPSoCs.

Os algoritmos de roteamento podem ser classificados, segundo a definição do(s) caminho(s) entre um par origem/destino, como *determinísticos* ou *adaptativos*. Um algoritmo determinístico fornece sempre o mesmo caminho entre um determinado par origem/destino, pois tal caminho é definido unicamente pelas posições relativas do par. Por outro lado, dependendo do grau adaptividade, um algoritmo adaptativo pode fornecer mais de um caminho entre um par origem/destino. Tipicamente o caminho é definido em função do estado instantâneo da rede (roteamento distribuído), incluindo pontos de congestionamento e falhas nos enlaces e/ou roteadores. Entretanto, ele pode ser também definido pela origem (roteamento na origem) a partir de um conjunto de caminhos pré-definido. A principal vantagem de um algoritmo determinístico é a sua simplicidade, além de prover baixa latência quando a rede não está congestionada. Em contrapartida, um algoritmo adaptativo está apto a evitar canais congestionados usando caminhos alternativos, adequando-se melhor a redes com tráfego intenso. A capacidade de adaptação de um algoritmo de roteamento aumenta a chance dos pacotes trafegarem em locais distantes de pontos quentes da rede (*hot-spots*).

Quanto a adaptatividade, o algoritmo pode ser *completamente adaptativo* ou *parcialmente adaptativo*. A diferença entre eles é que este último impõe restrições de roteamento que limitam sua busca por caminhos alternativos. De acordo com a minimalidade, um algoritmo de roteamento adaptativo pode ser classificado como *mínimo* ou *não-mínimo*. Um algoritmo mínimo considera somente os menores caminhos entre um determinado par origem/destino. Esta obrigatoriedade não ocorre em um algoritmo não-mínimo, que pode adotar caminhos de qualquer comprimento. Essa característica habilita uma busca por caminhos alternativos mais ampla que um algoritmo mínimo.

Este Capítulo propõe um esquema de roteamento orientado a fluxo que permite rotear fluxos de pacotes de diferentes classes através de diferentes versões do algoritmo de roteamento. Esse esquema visa unir as vantagens dos algoritmos de roteamento determinístico e adaptativo com o propósito de criar um serviço de comunicação com roteamento diferenciado. Visto que um algoritmo adaptativo pode fornecer vários caminhos entre um par origem/destino, ele oferece um serviço de comunicação com uma qualidade superior a um algoritmo determinístico, pois além de estar apto a evitar bloqueios, ele reduz a contenção, ainda que possa acarretar aumento da latência devido

ao uso de um caminho mais longo, no caso de algoritmos não-mínimos. A partir do esquema proposto, a NoC passa a suportar simultaneamente os dois tipos de algoritmos. A ideia é utilizar o algoritmo adaptativo no roteamento de mensagens com restrições temporais flexíveis, enquanto as demais são roteadas deterministicamente.

5.1 Roteamento orientado a fluxo

O roteamento orientado a fluxo é um esquema que pode ser implementado tendo como base qualquer algoritmo de roteamento adaptativo. A condição básica é que exista uma versão determinística do algoritmo adaptativo selecionado. Pode ser provado que esta versão sempre existe fixando um único caminho para cada par origem/destino a partir do conjunto de caminhos fornecidos pelo algoritmo adaptativo. Visto que um algoritmo adaptativo oferece caminhos alternativos, ele pode ser aplicado a fluxos de alta prioridade, enquanto fluxos de baixa prioridade são roteados usando a versão determinística do mesmo algoritmo. Os roteadores são responsáveis por selecionar a versão do algoritmo a ser aplicada para cada pacote durante a transmissão.

Durante a execução do roteamento pelo roteador, uma questão importante em algoritmos adaptativos é a política de seleção da porta de saída dentre as retornadas pelo algoritmo. Uma política comum é basear a decisão no nível de congestionamento dos roteadores vizinhos. No entanto, tal abordagem não garante um caminho livre de congestionamento, nem mesmo o caminho menos congestionado, pois se trata de uma informação local e instantânea, a qual pode orientar o roteamento para áreas congestionadas que não são localmente visíveis. Para reduzir o custo de área e manter a implementação o mais simples possível, o esquema proposto não adota detecção de congestionamento para selecionar a porta de saída. Quando o algoritmo retorna mais de uma porta de saída disponível, a porta selecionada é aquela que leva a um dos caminhos mais curtos. O custo de área do esquema proposto é muito baixo, menos de 1% da área do roteador.

O roteamento orientado a fluxo pode ser implementado tendo como base algoritmos adaptativos conhecidos como *odd-even* [CHI00] ou aqueles baseados no modelo *turn model* (e.g. *west first* e *north last*) [GLA94]. Este trabalho implementa o roteamento orientado a fluxo sobre a NoC Hermes tendo como base o algoritmo de roteamento Hamiltoniano [LIN94], o qual substitui o algoritmo XY (Seção 3.1.2). O algoritmo Hamiltoniano foi escolhido devido à simplicidade em obter-se uma versão determinística a partir da versão adaptativa, além de servir de base para a implementação de algoritmos *multicast* (e.g. *dual-path* e *multipath*). Em [EBR10], uma versão mínima adaptativa desse algoritmo chamada HAMUM (*Hamiltonian Adaptive Multicast Unicast Method*) foi comparada com os algoritmos XY, *odd-even* e o esquema DyAD [HU04]. O HAMUM apresentou um desempenho superior em termos de latência média sobre malhas de dimensões 8x8 e 14x14, considerando uma distribuição de tráfego com um *hot-spot*.

5.1.1 Algoritmo de roteamento Hamiltoniano

Um caminho Hamiltoniano é um caminho acíclico no qual é possível atingir todos os nodos de um grafo passando apenas uma vez em cada nodo [HAR72]. Sobre uma malha bidimensional podem ser definidos vários caminhos Hamiltonianos. A Figura 39 ilustra dois caminhos Hamiltonianos definidos sobre uma rede malha 5x5 com apenas um canal físico (bidirecional) entre roteadores vizinhos. Os enlaces sólidos identificam um caminho Hamiltoniano que parte do roteador 0 e vai até o roteador 24, ao passo que os enlaces tracejados identificam um caminho Hamiltoniano reverso. Os roteadores foram identificados utilizando rótulos que variam de *0* a *N-1*, sendo *N* o número de roteadores. Os rótulos crescem da esquerda para a direita em linhas pares e da direita para a esquerda em linhas ímpares. A partir dos caminhos Hamiltonianos definidos, a rede pode ser dividida em duas sub-redes *disjuntas* e *acíclicas*. Uma sub-rede contém os enlaces que vão de um rótulo menor para um maior e a outra contém os enlaces que vão de um rótulo maior para um menor. Os caminhos seguem a ordem crescente ou decrescente dos rótulos. Os enlaces pontilhados, os quais não fazem parte dos caminhos Hamiltonianos, podem ser usados para reduzir o comprimento das rotas [BOP98].

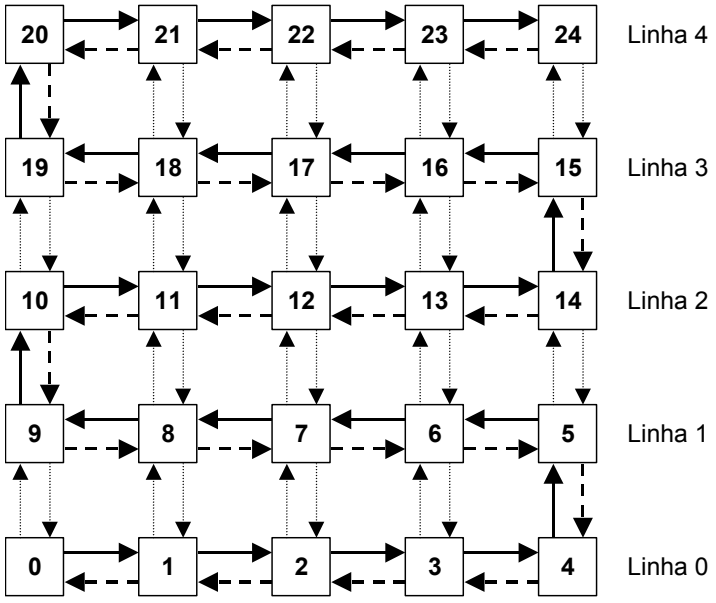


Figura 39 - Caminhos Hamiltonianos definidos sobre uma rede malha 5x5. Enlaces sólidos identificam um caminho que parte do roteador 0 e vai até o roteador 24, ao passo que os enlaces tracejados identificam o caminho Hamiltoniano reverso. Os enlaces pontilhados não fazem parte dos caminhos Hamiltoniano

A versão não-mínima parcialmente adaptativa do algoritmo de roteamento Hamiltoniano funciona como segue. Um pacote em um roteador com rótulo menor que o destino (origem < destino) é encaminhado para qualquer roteador vizinho cujo rótulo seja maior que o roteador local e menor/igual ao destino (local < vizinho ≤ destino). Considere, por exemplo, o roteador 12 como origem e o roteador 22 como destino. Alguns dos possíveis caminhos tomados pelo pacote são: {12, 17, 22} (caminho mínimo), {12, 13, 16, 17, 22}, {12, 13, 14, 15, 16, 17, 22}, {12, 17, 18, 21, 22} e {12, 13, 14, 15, 16, 17, 18, 19,

20, 21, 22} (caminho mais longo). De maneira análoga, quando um pacote está em um roteador com rótulo maior que o destino (origem > destino), ele é encaminhado para qualquer roteador vizinho cujo rótulo seja menor que o roteador local e maior/igual ao destino (local > vizinho ≥ destino). Considere, por exemplo, o roteador 13 como origem e o roteador 7 como destino. Os possíveis caminhos tomados pelo pacote são: {13, 12, 7} (caminho mínimo), {13, 12, 11, 8, 7} e {13, 12, 11, 10, 9, 8, 7} (caminho mais longo).

Para criar uma versão mínima determinística do algoritmo a partir da versão não-mínima parcialmente adaptativa, a condição de encaminhamento pode ser restringida para “encaminhar o pacote para o maior/menor vizinho, cujo rótulo seja menor/maior que o destino” (dependendo dos rótulos origem e destino). Nos exemplos 12→22 e 13→7, os caminhos tomados são respectivamente {12, 17, 22} e {13, 12, 7}.

Ambas as versões do algoritmo de roteamento Hamiltoniano são livres de *deadlock* uma vez que os pacotes são roteados sobre subredes disjuntas e acíclicas [LIN94]. De maneira semelhante ao algoritmo de roteamento *odd-even*, o qual proíbe algumas mudanças de direção em colunas ímpares e pares da malha, o algoritmo Hamiltoniano proíbe algumas mudanças de direção em linhas ímpares e pares, a fim de evitar ciclos que podem acarretar *deadlock*. Considerando os caminhos Hamiltonianos definidos na Figura 39, as mudanças de direção norte/oeste e sul/leste são proibidas em linhas pares da malha, e as mudanças de direção norte/leste e sul/oeste são proibidas em linhas impares.

Neste trabalho, o algoritmo de roteamento Hamiltoniano é utilizado simultaneamente nas duas versões apresentadas (*i*) não-mínima parcialmente adaptativa e (*ii*) mínima determinística, referidas no restante do texto apenas como adaptativa e determinística, respectivamente. Para permitir aos roteadores decidirem a versão do algoritmo a ser aplicada, um bit disponível no cabeçalho do pacote (campo *unused* - Figura 25) é definido como bit de roteamento. Durante a transmissão de um pacote, o bit de roteamento é verificado pelos roteadores e então a versão correspondente do algoritmo de roteamento é executada. Esse processo se repete em cada roteador percorrido pelo pacote (roteamento distribuído).

5.2 Integração do roteamento orientado a fluxo no nível de software

O controle sobre qual versão do algoritmo de roteamento deve ser utilizada foi integrada à API de comunicação da plataforma HeMPS através do parâmetro *priority* da primitiva `Send()` (Listagem 1). Assim, a plataforma passa a dispor de um serviço de comunicação com roteamento diferenciado, cuja versão do algoritmo a ser utilizada depende apenas do valor do parâmetro *priority*. Durante o processamento da primitiva `Send()`, o valor desse parâmetro é utilizado para definir o valor do bit de roteamento no cabeçalho do pacote que carrega a mensagem. Definindo o parâmetro *priority* como HIGH, indica que a versão adaptativa do algoritmo de roteamento deve ser aplicada. Ao

utilizar o valor LOW, os pacotes são definidos para serem roteados deterministicamente.

Uma questão a ser tratada ao utilizar um algoritmo de roteamento adaptativo é o ordenamento das mensagens, pois estas podem tomar caminhos diferentes e chegarem ao destino em uma ordem diferente da qual foram enviadas. A premissa para a ocorrência de um desordenamento é a transmissão simultânea de duas ou mais mensagens de uma origem para um mesmo destino. A NoC Hermes não implementa nenhum tipo mecanismo que garanta o ordenamento das mensagens. Entretanto, na plataforma HeMPS, esse ordenamento é assegurado em software pelo protocolo de comunicação *read request* implementado pelo microkernel (Seção 3.2). Esse protocolo garante que um PE origem somente envia uma mensagem *n* depois que o PE destino recebeu a mensagem *n-1*. Visto que a primitiva `Receive()` é bloqueante, é impossível requisitar uma segunda mensagem antes de ter recebido a primeira. Isto garante que nunca há mais de uma mensagem sendo transmitida para um mesmo destino em um determinado instante, portanto o ordenamento é assegurado.

5.3 Avaliação

Esta Seção apresenta dois experimentos utilizados para avaliar o serviço de comunicação com roteamento diferenciado. O primeiro experimento foi realizado utilizando tráfego sintético gerado por geradores de tráfego conectados à NoC Hermes. Neste caso o ordenamento dos pacotes não é tratado. O objetivo é avaliar o serviço de comunicação sob diferentes intensidades de tráfego. O segundo experimento foi realizado utilizando tráfego gerado por um decodificador MJPEG executando sobre a plataforma HeMPS (ordenamento dos pacotes assegurado). As métricas de desempenho utilizadas são latência, vazão e *jitter*. A topologia utilizada é malha 5x5 com canais físicos simples (Figura 39) e chaveamento por pacotes.

5.3.1 NoC Hermes com geradores de tráfego

Duas distribuições espaciais de tráfego foram avaliadas: (*i*) *hot-spot* e (*ii*) complemento. A Figura 40 apresenta a distribuição de tráfego *hot-spot* utilizada. Dois *hot-spots* ocorrem nos roteadores 12 e 17. A linha tracejada corresponde ao fluxo avaliado (2→22) e as linhas sólidas correspondem aos fluxos de perturbação. Os caminhos tomados pelos fluxos na Figura 40 consideram a versão determinística do algoritmo Hamiltoniano.

O fluxo avaliado (2→22) tem uma taxa de injeção *fixa* de 30% da largura de banda do enlace e os fluxos de perturbação têm sua taxa de injeção variando de 5% a 50%. O fluxo 2→22 é avaliado em três cenários de roteamento:

1. *Determinístico*: todos os fluxos são roteados utilizando a versão determinística do algoritmo de roteamento Hamiltoniano.
2. *Adaptativo*: todos os fluxos são roteados utilizando a versão adaptativa do algoritmo

de roteamento Hamiltoniano.

3. *Orientado a fluxo*: somente o fluxo 2→22 é roteado usando a versão adaptativa do algoritmo de roteamento Hamiltoniano, enquanto os demais são roteados deterministicamente.

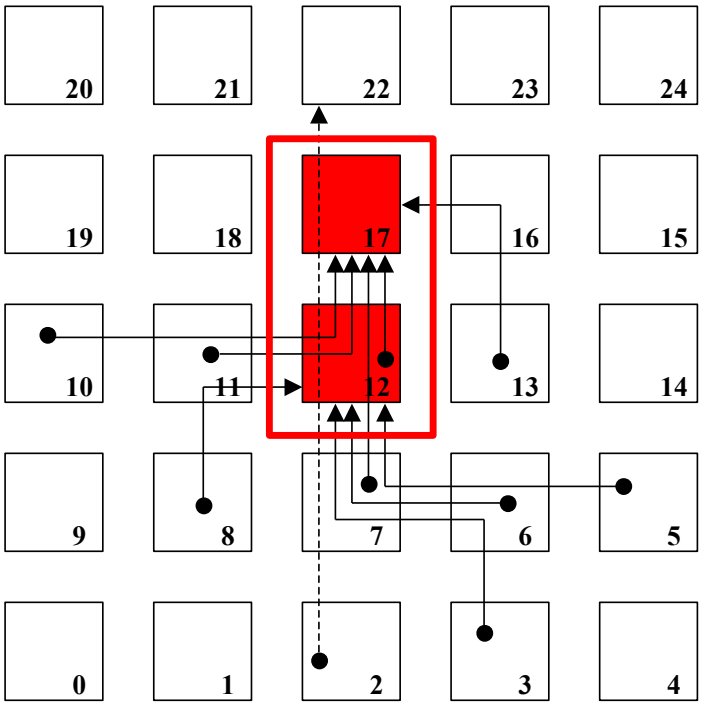


Figura 40 – Distribuição de tráfego *hot-spot*.

Os gráficos a seguir mostram os resultados de latência, vazão e *jitter* obtidos para o fluxo 2→22. As curvas *Flow* correspondem aos resultados obtidos roteando adaptativamente somente o fluxo 2→22. Os resultados de *jitter* foram obtidos considerando a taxa de injeção dos fluxos de perturbação igual a 20% da largura de banda do enlace. Para outras taxas de injeção moderadas (abaixo do ponto de saturação da rede) o comportamento é similar.

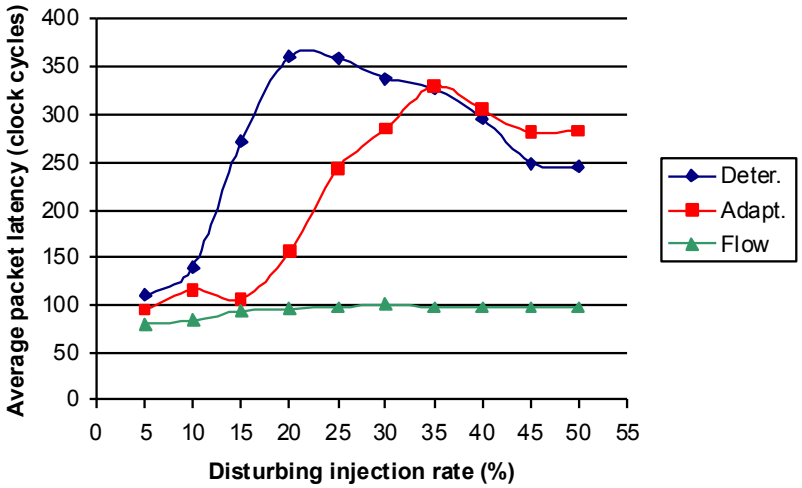


Figura 41 – Latência média do fluxo 2→22 [CAR10].

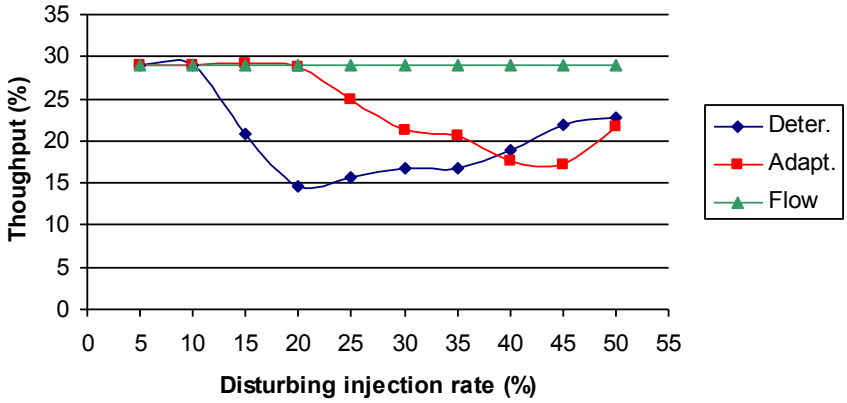


Figura 42 – Vazão média do fluxo 2→22 [CAR10].

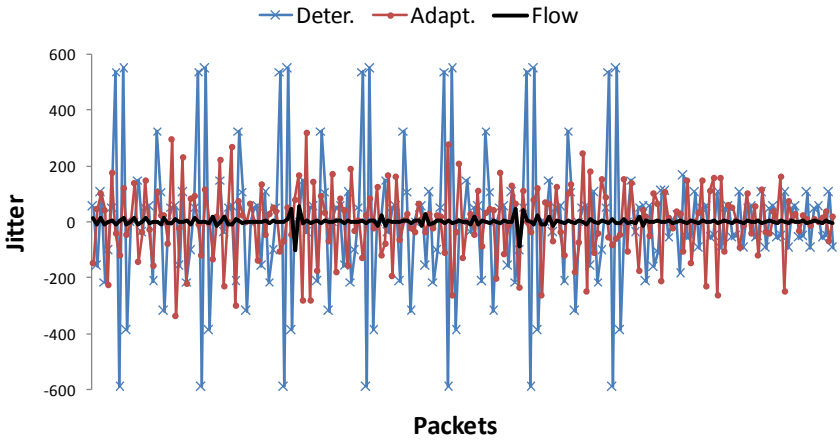


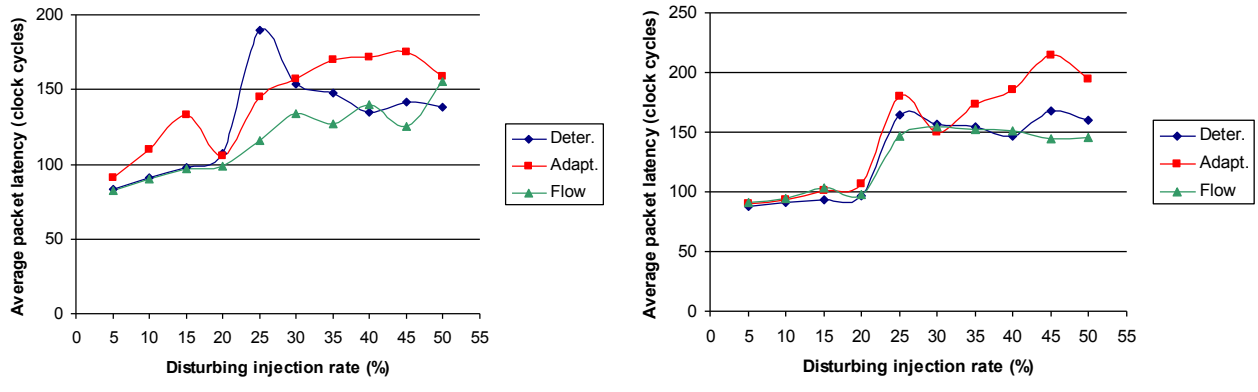
Figura 43 – *Jitter* do fluxo 2→22 em ciclos de *clock* [CAR10].

Observar a latência e a vazão do fluxo 2→22 (Figura 41 e Figura 42) nos cenários onde todos fluxos são roteados utilizando a mesma versão do algoritmo de roteamento (adaptativa ou determinística). A adaptatividade melhora o desempenho do roteamento Hamiltoniano, apresentando um bom desempenho quando a taxa de injeção dos fluxos de perturbação é baixa. Entretanto, sob tráfegos mais intensos, o desempenho da versão adaptativa decai como a determinística. Note que a partir de certas taxas de injeção dos fluxos de perturbação, os valores médios de latência e vazão do fluxo 2→22 começam, respectivamente, a diminuir e aumentar. Isso ocorre porque quanto maior a taxa de injeção dos fluxos de perturbação, mais cedo os geradores de tráfego terminam de enviar todos os pacotes desses fluxos (500 pacotes). Logo, em taxas de injeção mais altas, eles não perturbam o fluxo 2→22 durante toda a transmissão dos seus pacotes.

Quando o esquema de roteamento orientado a fluxo é empregado, a latência e a vazão do fluxo 2→22 não são significativamente afetadas pelos fluxos de perturbação. Graças à combinação proporcionada por esse esquema, o algoritmo determinístico limita os caminhos disponíveis para os fluxos de perturbação, deixando mais caminhos livres para serem explorados pelo algoritmo adaptativo empregado pelo fluxo 2→22. Por exemplo, quando a porta norte do roteador 7 (Figura 40) não está disponível, o fluxo 2→22 pode tomar o caminho {2, 7, 8, 11, 18, 21, 22}, o qual tem um tráfego de menor

intensidade. Quando todos os fluxos são roteados deterministicamente, o fluxo 2→22 não está apto a evitar a área de *hot-spot* e seu desempenho é severamente afetado. Uma queda de desempenho semelhante pode ser observada quando todos os fluxos são roteados adaptativamente. Neste caso, quando a porta norte do roteador 7 não está disponível, os fluxos de perturbação estão habilitados a explorar caminhos alternativos usando a porta oeste do roteador 7, que por sua vez podem perturbar o fluxo 2→22, visto que agora este último tem de compartilhar os recursos da NoC com os demais fluxos. O mesmo comportamento é observado quando o *jitter* é avaliado, como mostra a Figura 43. O roteamento adaptativo atenua o *jitter*, comparado com o roteamento determinístico, mas não o elimina. O roteamento orientado a fluxo praticamente remove todo o *jitter* do fluxo 2→22, com quase todos pacotes tendo a mesma latência. Note que nesse cenário uma versão *mínima* adaptativa do algoritmo de roteamento Hamiltoniano não ofereceria caminhos alternativos ao fluxo 2→22, visto que a menor distância entre dois pontos é uma linha reta.

Os bons resultados obtidos com o cenário *hot-spot* devem-se ao fato de que existem áreas livres de congestionamento na rede e o algoritmo adaptativo consegue encontrar caminhos alternativos nestas áreas. Uma situação bem diferente surge em um cenário com distribuição de tráfego complemento, onde a carga está igualmente distribuída na rede. Os gráficos da Figura 44 mostram a latência média dos fluxos 6→18 e 1→23 obtidos com um taxa de injeção de 20% da largura de banda do enlace, enquanto a taxa de injeção dos fluxos de perturbação variam de 5% a 50%. Mesmo limitando a disponibilidade de caminhos para os demais fluxos através da versão determinística do algoritmo, o roteamento orientado a fluxo apresentou pouco ganho em relação a todos os fluxos sendo roteados igualmente. Este experimento destaca a limitação dos algoritmos adaptativos na busca de caminhos alternativos em situações onde o tráfego está bem distribuído na rede, como no caso de uma distribuição de tráfego complemento.



Fluxo 6→18 **Fluxo 1→23**

Figura 44 – Latência média dos fluxos 6→18 e 1→23, considerando uma distribuição de tráfego complemento [CAR10].

5.3.2 Serviço de comunicação com roteamento diferenciado na plataforma HeMPS

A aplicação alvo neste experimento é um decodificador MJPEG particionado em nove tarefas (*Start*, IVLC1, IVLC2, IVLC3, IVLC4, IVLC5, IQUANT, IDCT e *Print*), as quais comunicam-se como um *pipeline*. A tarefa *Start* envia continuamente blocos compactados para a tarefa IVLC1, a qual efetivamente inicia a decodificação. O mapeamento e os fluxos da aplicação sobre a plataforma HeMPS são ilustrados na Figura 45. Os Plasma-IPs SL conectados aos roteadores 3, 4, 5, 6 e 7 executam somente tarefas de *debug* do sistema, gerando mensagens para o Plasma-IP MP (*Master*) que correspondem a fluxos de perturbação. Esse cenário caracteriza o Plasma-IP MP como um *hot-spot*, o qual perturba a comunicação entre as tarefas IVLC2 e IVLC3. Devido ao alto tempo de computação das tarefas em software (computação intensiva), a taxa de injeção gerada por elas é inferior a 3% da largura de banda dos enlaces da NoC.

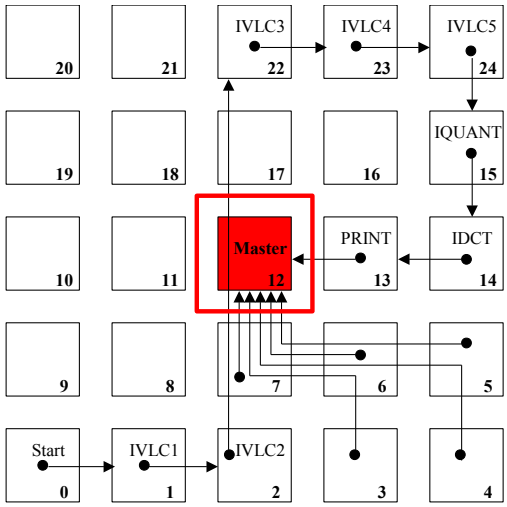


Figura 45 – Mapeamento das tarefas da aplicação MJPEG e fluxos de perturbação [CAR10].

Os mesmos três cenários de roteamento da Seção 5.3.1 são repetidos: (*i*) todos os fluxos roteados deterministicamente; (*ii*) todos fluxos roteados adaptativamente; e (*iii*) somente os fluxos do decodificador MJPEG roteados adaptativamente. A versão do algoritmo de roteamento a ser empregada sobre os fluxos de mensagens é definida no código fonte de cada tarefa através do parâmetro *priority* da primitiva Send() (Seção 5.2). A Tabela 5 apresenta os resultados médios de latência e vazão para todos os fluxos gerados pelas tarefas do decodificador MJPEG considerando os três cenários de roteamento. Os resultados de latência estão em ciclos de *clock* e a vazão corresponde a uma percentagem da largura de banda total do enlace. O tempo total (penúltima linha da tabela) corresponde ao tempo gasto para terminar a execução da aplicação em ciclos de *clock*. A última linha da tabela mostra o aumento no tempo total de execução em relação à aplicação executando sozinha na plataforma (sem perturbações). Comparando o roteamento determinístico e adaptativo, observa-se que a latência média dos pacotes do fluxo IVLC2→IVLC3 é próxima, enquanto o tempo total de execução é significativamente diferente. No cenário determinístico, devido à alta concorrência pela porta norte do

roteador 7, a tarefa IVLC2 é contida e seus pacotes só começam a ser entregues à tarefa IVLC3 depois que algumas tarefas de *debug* terminam e o tráfefo intenso no *hot-spot* diminui. Devido a tal contenção, o tempo total de execução aumenta, entretanto, a latência média dos pacotes permanece baixa porque a maioria destes só começa a ser injetada na rede após a redução do *hot-spot*. O roteamento orientado a fluxo consegue desviar do *hot-spot*, mantendo a latência e a vazão do fluxo IVLC2→IVLC3 semelhantes aos demais e com um aumento no tempo total de execução de apenas 1,3%. Esses resultados mostram que o roteamento orientado a fluxo favorece *composability*, visto que o desempenho obtido é próximo ao da aplicação executando sozinha sobre a plataforma.

Tabela 5 – Resultados médios de latência e vazão para as tarefas do decodificador MJPEG executando o cenário da Figura 45 [CAR10].

	Routing					
	Deterministic		Adaptive		Flow Oriented	
	Latency	Throughput	Latency	Throughput	Latency	Throughput
Start → IVLC1	477.57	2.37%	479.6	2.64%	477.4	2.7%
IVLC1 → IVLC2	349.5	1.23%	349.4	1.28%	349.5	1.41%
IVLC2 → IVLC3	1621.2	1.23%	1668.66	1.28%	377	1.41%
IVLC3 → IVLC4	350	1.23%	349.9	1.28%	349	1.41%
IVLC4 → IVLC5	349.5	1.23%	349.4	1.28%	350	1.41%
IVLC5 → IQUANT	349	1.23%	349	1.28%	349	1.41%
IQUANT → IDCT	349.5	1.23%	349.5	1.28%	349.5	1.41%
IDCT → PRINT	350	1.23%	350	1.28%	350	1.41%
Total time (cycles)	3,932,709		2,209,991		2,078,633	
Execution time overhead	91.66 %		7.7 %		1.3 %	

A Figura 46 apresenta os resultados de *jitter* da aplicação. Como observado anteriormente, em um cenário de *hot-spot* o roteamento orientado a fluxo praticamente elimina o *jitter*. Uma análise superficial poderia assumir que devido às baixas taxas de injeção de tarefas executando em software, a interferência entre seus fluxos seria mínima. Entretanto, este experimento demonstra que em um MPSoC real a interferência entre os fluxos das aplicações pode afetar significativamente métricas de desempenho importantes em aplicações com restrições de QoS.

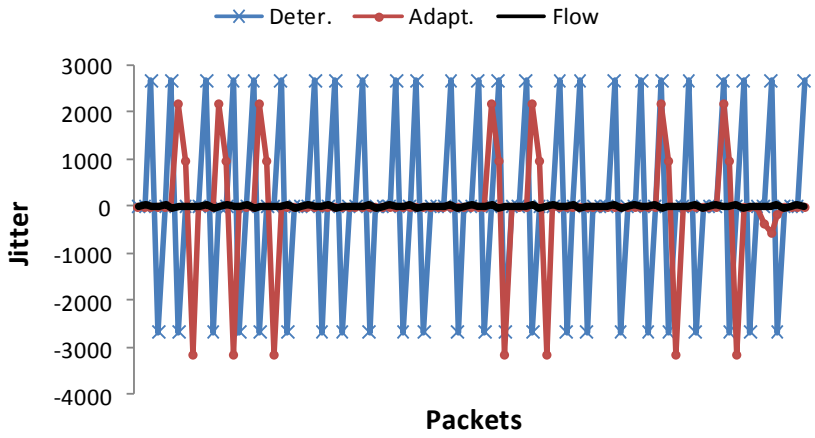


Figura 46 – *Jitter* do decodificador MJPEG em ciclos de *clock* [CAR10].

6. SERVIÇO DE COMUNICAÇÃO COLETIVA (*MULTICAST*)

Este Capítulo apresenta a terceira contribuição desta Tese, a comunicação coletiva baseada em *multicast*. Esse tipo de comunicação é o resultado da demanda de operações que envolvem dados compartilhados entre tarefas e que precisam ser transmitidos e recebidos através de um modelo de troca de mensagem [DUA02]. Dependendo da aplicação, a programação paralela pode fazer uso de diversos padrões de comunicação coletiva como *gather*, *reduce* e *scatter* [WAL96]. Entretanto, o padrão mais utilizado e que recebe maior atenção da comunidade de pesquisa é o *multicast*. O *broadcast* também é muito utilizado e pode ser tratado como um caso especial do *multicast*.

Visto que a infraestrutura de comunicação dos MPSoCs vem rapidamente mudando de barramentos para NoCs, alguns serviços oferecidos pelos barramentos também devem estar disponíveis em NoCs, em especial o serviço de comunicação coletiva. Esse serviço é responsável pela execução eficiente de diversas aplicações paralelas como algoritmos de pesquisa, busca em grafos e operações com matrizes (inversão e multiplicação por exemplo). Ele também é empregado na implementação de diferentes protocolos como controle e configuração de rede, sincronização e coerência de *cache*. Apesar de arquiteturas de interconexão baseadas em barramentos não serem escaláveis e proporcionarem baixo suporte a comunicações paralelas, elas nativamente suportam comunicação coletiva. Em NoCs com topologia malha, a implementação eficiente desse tipo de comunicação depende de algoritmos especiais de *multicast*. Comumente, o suporte a mensagens *multicast* é implementado de maneira não-escalável, enviando separadamente uma cópia da mensagem para cada destino. Tal solução aumenta o volume do tráfego na rede e consequentemente o consumo de energia à medida que aumenta o número de destinos das mensagens *multicast*. Algoritmos *multicast* para redes com topologia malha têm sido estudados e propostos amplamente para arquiteturas paralelas.

Este Capítulo apresenta a implementação do algoritmo *multicast dual-path* na NoC Hermes com a finalidade de dar suporte em hardware ao serviço de comunicação coletiva na plataforma HeMPS. Tal algoritmo permite a transmissão de mensagens *multicast* de maneira escalável, pois o número de cópias enviadas é limitado pelo número máximo de conjuntos nos quais os destinos são agrupados.

6.1 Algoritmo *multicast dual-path* na NoC Hermes

O algoritmo *dual-path* foi originalmente proposto por Lin et al. em [LIN94] para multicomputadores e tem como base o algoritmo de roteamento Hamiltoniano (Seção 5.1.1). Esse algoritmo é simples e bem conhecido na área de multicomputadores, além de possuir uma descrição clara e concisa que facilita sua implementação em NoCs com

topologia malha. A origem da mensagem *multicast* deve dividir o conjunto de destinos em dois subconjuntos, um contendo os destinos com rótulos maiores que a origem e outro contendo os menores. Em seguida, uma cópia da mensagem é enviada separadamente para cada subconjunto. Caso todos os rótulos do conjunto de destinos sejam maiores ou menores que a origem, apenas uma cópia de mensagem é enviada. A escalabilidade da transmissão da mensagem *multicast* é garantida pelo número máximo de cópias enviadas. O algoritmo *dual-path* garante que são enviadas no máximo duas cópias da mensagem, independente do número de destinos.

Para suportar a transmissão *multicast*, o cabeçalho dos pacotes da NoC Hermes foi estendido de maneira a incluir vários destinos, como ilustra a Figura 47. Cada *flit* de destino no cabeçalho tem o campo *service* (Figura 25) indicando que se trata de um destino do pacote. O fim do cabeçalho é indicado pelo campo *service* do último *flit* de destino. O cabeçalho de pacotes *multicast* tem tamanho variado, dependendo do número de destinos.

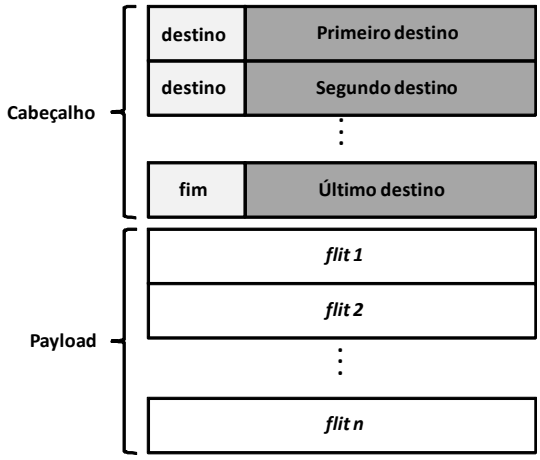


Figura 47 – Estrutura do pacote da NoC Hermes para suportar vários destinos.

A cópia da mensagem enviada ao subconjunto contendo os destinos maiores que a origem deve ter no cabeçalho os destinos ordenados em ordem crescente, enquanto a cópia enviada ao subconjunto contendo os destinos menores que a origem deve ter no cabeçalho os destinos ordenados em ordem decrescente. As cópias são entregues aos destinos na ordem em que estes aparecem no cabeçalho. A Figura 48 ilustra o formato dos pacotes contendo as cópias da mensagem a ser transmitida, considerando o roteador origem 6 e o conjunto de destinos {1, 3, 5, 8, 10, 12, 14}.

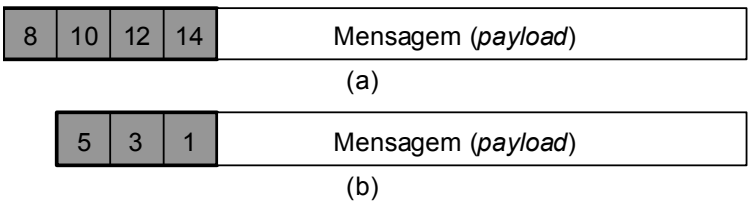


Figura 48 – Formato dos pacotes *multicast* contendo as cópias da mensagem. (a) Pacote enviado ao subconjunto contendo os destinos maiores que a origem. (b) Pacote enviado ao subconjunto contendo os destinos menores que a origem.

O caminho tomado pelos pacotes *multicast* é definido pelo algoritmo de roteamento Hamiltoniano, o qual define também o caminho tomado pelos pacotes *unicast*. Durante o roteamento dos pacotes *multicast*, o algoritmo Hamiltoniano é executado baseado no primeiro destino contido no cabeçalho, de maneira análoga a pacotes *unicast*. Quando o pacote chega ao primeiro destino contido no cabeçalho, a porta local do roteador é alocada. Em seguida, o roteador remove esse destino do cabeçalho e o roteamento é executado novamente baseado no próximo destino. Nessa segunda execução do roteamento, uma porta em direção ao próximo destino é alocada. Esse processo se repete nos demais roteadores do caminho até o último destino contido no cabeçalho. Em cada roteador destino, exceto o último, o pacote é encaminhado simultaneamente para as duas portas alocadas. O avanço do pacote nesses roteadores depende da disponibilidade simultânea do IP (porta local) e do roteador vizinho (porta norte, sul, leste ou oeste). A Figura 49(a) e a Figura 49(b) ilustram o processo de roteamento considerando os pacotes *multicast* apresentados na Figura 48 e o roteador 6 como origem. As figuras ilustram apenas os cabeçalhos, entretanto o *payload* está anexado a estes. Visto que a transmissão de pacotes *multicast* demanda a alocação de muitos recursos, a versão não-mínima parcialmente adaptativa do algoritmo de roteamento Hamiltoniano é empregada (Seção 5.1.1). A ideia é oferecer o máximo de recursos disponíveis a estes pacotes para que eles sejam transmitidos o mais rápido possível, com o intuito de evitar a sobrecarga intensificação do tráfego na rede.

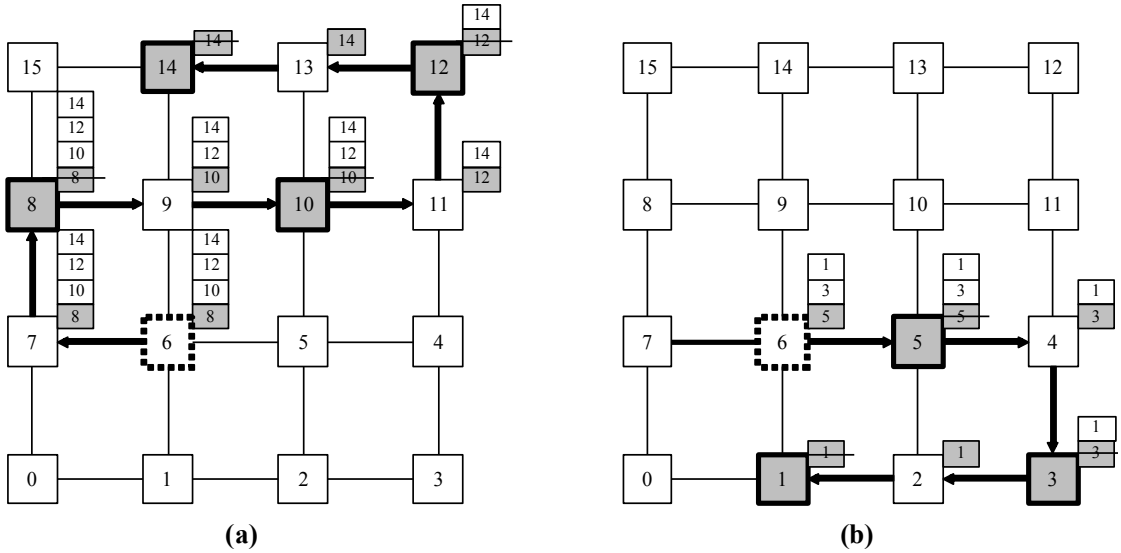


Figura 49 – Processo de roteamento de pacotes *multicast*. (a) Pacote sendo transmitido para o subconjunto dos destinos maiores que a origem. (b) Pacote sendo transmitido para o subconjunto dos destinos menores que a origem.

6.2 Integração do *multicast dual-path* no nível de software

No desenvolvimento de aplicações paralelas, a comunicação coletiva simplifica a programação, facilita a implementação de esquemas eficientes de comunicação e reflete

o agrupamento conceitual de tarefas. O interesse no uso de comunicação coletiva não é novidade e a sua importância pode ser evidenciada pela sua inclusão em bibliotecas como CCL [BAL95], no padrão MPI e no suporte a linguagens paralelas [LI91]. O serviço de comunicação coletiva foi adicionado à API da plataforma HeMPS através da primitiva `Multicast()`, a qual oferece acesso ao algoritmo *multicast dual-path* implementado na NoC Hermes. Essa primitiva possibilita uma comunicação envolvendo uma tarefa origem e várias tarefas destinos através de *uma única* chamada de sistema. As tarefas destino recebem a mensagem *multicast* através da primitiva `Receive()`, de maneira análoga à recepção de mensagens *unicast*. O protótipo da primitiva `Multicast()` e seus parâmetros são:

`void Multicast(Message *msg, int *target_list, int targets)`, onde:

- `Message *msg`: mensagem a ser enviada;
- `int *target_list`: conjunto de destinos da mensagem;
- `int targets`: número total de destinos.

Toda a preparação das cópias da mensagem como localização dos PEs onde as tarefas destino estão alocadas, divisão e ordenamento dos PEs destinos é realizada pelo microkernel durante o processamento da primitiva `Multicast()`. Caso alguma tarefa destino não esteja alocada no sistema durante o processamento da primitiva, o microkernel solicita a alocação ao processador mestre. A comunicação *multicast* também segue o protocolo de comunicação *read request* (Seção 3.2). Portanto, o PE origem do *multicast* só inicia a transmissão após receber a requisição da mensagem de todas as tarefas destino.

A integração do *multicast dual-path* na plataforma HeMPS apresenta uma limitação em relação ao número de tarefas destinatárias de uma mensagem *multicast*. Esta limitação deve-se ao fato de que o microkernel utiliza uma mesma estrutura de dado (struct da linguagem C) para descrever mensagens *unicast* e *multicast*. Antes da integração, esse descritor de mensagem já era utilizado em diversas funções do microkernel no tratamento de mensagens *unicast*. No descritor há um campo de 32 bits utilizado para indicar a(s) tarefa(s) destino da mensagem. No caso de mensagens *unicast* esse campo interpretado como um valor inteiro que indica a tarefa destino. No caso de mensagens *multicast* esse mesmo campo indica as tarefas destino a partir de uma codificação, onde cada bit corresponde ao identificador de uma tarefa do sistema. A limitação oriunda do compartilhamento do descritor de mensagens é que uma mensagem *multicast* pode ter no máximo 32 tarefas destino, sendo que o identificador destas deve estar dentro do intervalo 0 a 31. Tal abordagem foi escolhida com o intuito de minimizar as mudanças na estrutura do microkernel e maximizar a reutilização de código (e.g. primitiva `Receive()`), simplificando a integração do *multicast dual-path* no nível de software. Essa limitação é imposta pelo software, uma vez que a NoC suporta um número

arbitrário de IPs destino. A Figura 50 ilustra um exemplo da codificação utilizada no campo de destino do descritor de uma mensagem *multicast* destinada a dez tarefas.

Identificador da tarefa	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Destinos setados em '1'	1	0	0	0	0	1	0	1	0	0	0	1	0	0	1	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0

Figura 50 – Campo de 32 bits indicando como destino da mensagem *multicast* as tarefas 2, 4, 8, 10, 12, 16, 19, 23, 25 e 30.

6.3 Avaliação

A comunicação coletiva foi avaliada sobre diferentes instâncias da plataforma HeMPS. Os experimentos realizados visam comparar o algoritmo *multicast dual-path* com a trivial implementação de *multicast* baseada no envio de múltiplas mensagens *unicast*. As avaliações foram feitas utilizando um protocolo de coerência de *cache* e duas diferentes aplicações. A primeira avaliação foi feita a partir do protocolo de coerência de *cache* MSI. A segunda avaliação foi feita utilizando uma aplicação produtor/consumidores e a última foi realizada utilizando um multiplicador de matrizes distribuído que implementa o algoritmo de Fox.

6.3.1 Protocolo de coerência de cache [CHA11]

Esta seção avalia os ganhos obtidos a partir da utilização de mensagens *multicast* no protocolo de coerência de *cache* MSI (*Modified Shared Invalid*). Para a realização desta avaliação, uma *cache* de dados L1 foi adicionada ao Plasma-IP SL, e um IP de memória *cache* de dados L2 foi adicionado ao MPSoC, como mostra a Figura 51. A *cache* L1 é privada ao passo que a *cache* L2 armazena os blocos de memória compartilhados pelos PEs.

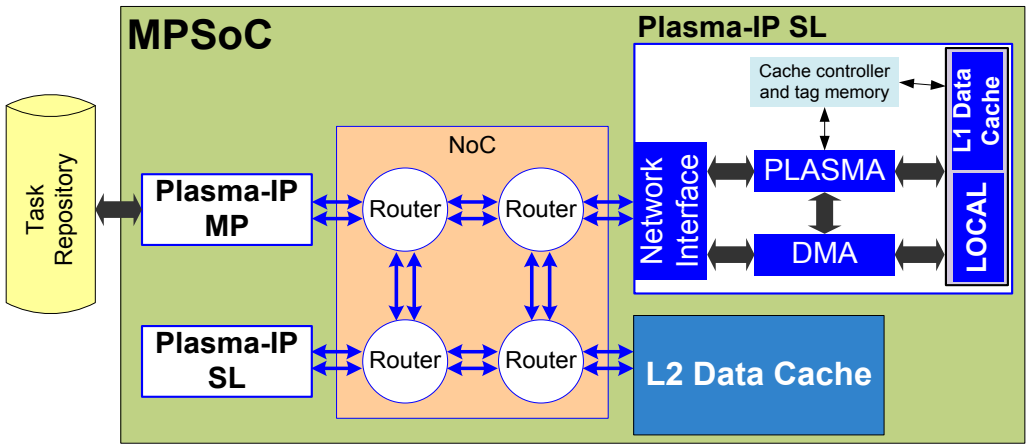


Figura 51 – Instância 2x2 da plataforma HeMPS com uma hierarquia de memória *cache* de dados em dois níveis (L1 e L2) [CHA11].

O MSI é um protocolo de coerência de *cache* baseado em diretório. Segundo esse protocolo um bloco compartilhado pode estar em um dos três possíveis estados; (*i*) *Modified*, uma cópia do bloco foi modificada em alguma *cache* L1, portanto, a *cache* L2

não contém uma entrada válida desse bloco, (*ii*) *Shared*, zero ou mais *caches* L1 podem conter uma cópia idêntica de um bloco que está armazenado na *cache* L2 e (*iii*) *Invalid*, o bloco não é válido. Esse protocolo foi implementado parte em hardware (*cache controller* – Plasma-IP SL) e parte em software (microkernel). As principais funções do módulo *cache controller* são detectar *miss/hit* e tratar operações de leitura/escrita. O microkernel realiza a substituição de blocos e operações de *write-back*.

A seguir são avaliados os ganhos obtidos a partir da utilização de mensagens *multicast* na implementação das operações de (*i*) invalidação de bloco e (*ii*) *write-back*, sobre uma instância 5x5 da plataforma HeMPS. A versão do MSI que implementa essas operações utilizando mensagens *multicast* é chamada de OPT (*optimized*), ao passo que a versão suportando apenas mensagens *unicast* é chamada de NO-OPT (*no-optimized*). Ambas as versões foram comparadas em termos de tempo e consumo de energia gastos nas comunicações entre PEs e *cache* L2 durante a execução das operações.

Para a avaliação de consumo de energia é adotado o modelo baseado em volume proposto por Hu et al. [HU03]. A equação 1 é utilizada para computar o consumo médio de energia no envio de um bit, em uma transmissão fim-a-fim, entre dois pontos da NoC.

$$E_{bit}^{hops} = n_{hops} * E_{S_{bit}} + (n_{hops} - 1) * E_{L_{bit}} \quad (1)$$

Onde: E_{Sbit} (20.58 pJ/flit), E_{Lbit} (2.84 pJ/flit) e n_{hops} representam, respectivamente, o consumo de energia em um roteador, nos fios de interconexão e o número de roteadores por onde o bit passou. O modelo de energia foi calibrado usando a tecnologia ST/IBM CMOS 65 nm a 1V, adotando *clock-gating*, 100 MHz de frequência e taxa de injeção de 10% da largura de banda dos enlaces. A ferramenta *PrimePower* gera os valores de potência e energia usados na equação 1.

6.3.1.1 Invalidação de bloco

Antes de modificar um bloco compartilhado, o PE requisita à *cache* L2 acesso exclusivo ao bloco. Ao receber essa requisição, a *cache* L2 envia uma mensagem de invalidação a cada um dos PEs que compartilha tal bloco, com o intuito de evitar incoerência. Após o processo de invalidação, a *cache* L2 autoriza o PE a modificar o bloco. Esta seção avalia os ganhos obtidos a partir da utilização de mensagens *multicast* na operação de invalidação variando o número de PEs que compartilham o bloco a ser invalidado. A Tabela 6 mostra o tempo em ciclos de *clock* e a energia consumida no processo de invalidação, considerando um bloco compartilhado por 3, 5 e 8 PEs. Os ganhos obtidos a partir do uso de mensagens *multicast* (OPT) não são crescentes em relação ao aumento do número de PEs devido aos diferentes mapeamentos utilizados em cada um dos experimentos. Isso implica o envio de uma ou duas cópias da mensagem de invalidação, dependendo da localização dos PEs em relação à origem *cache* L2 (Seção 6.1), bem como o comprimento dos caminhos tomados.

Tabela 6 – Tempo e energia consumida no processo de invalidação variando o número de PEs que compartilham o bloco a ser invalidado [CHA11].

	Protocol	**3 PEs**	**5 PEs**	**8 PEs**
Clock Cycles	NO-OPT	141	154	147
	OPT	129	127	129
OPT gain vs NO-OPT		**8.51%**	**17.53%**	**12.24%**
Energy (pJ)	NO-OPT	1635	2584	3798
	OPT	685	2073	2916
OPT Gain vs NO-OPT		**58.07%**	**19.76%**	**23.20%**

6.3.1.2 Write-back

Após um acesso de leitura à *cache* L1 resultar em um *miss*, o PE tem de enviar à *cache* L2 uma requisição de leitura a um bloco modificado. Ao receber essa requisição, a *cache* L2 envia uma requisição de *write-back* ao PE que mantém o acesso exclusivo ao bloco requisitado. Esse PE então executa o *write-back* enviando uma cópia do bloco à *cache* L2 e outra cópia ao PE que requisitou a leitura. Esta seção avalia os ganhos obtidos na requisição de leitura a partir da utilização de mensagens *multicast* na operação de *write-back*. Os experimentos realizados variam a distância em *hops* entre o PE que requisita a leitura de um bloco e a *cache* L2. Figura 52 apresenta os resultados de consumo de energia. A utilização de mensagens *multicast* (OPT) reduziu em média 12% o consumo de energia. Entretanto, a utilização de múltiplas mensagens *unicast* (NO-OPT) apresentou um desempenho pouco superior em termos de tempo (em média 30 ciclos de *clock* mais rápida). Isso pode ocorrer quando o custo de preparação da mensagem *multicast* não é compensado devido ao baixo número de destinos, dois nesse caso (L2 e PE que mantém o bloco). Além disso, o caminho tomado pela mensagem *multicast* pode ser muito longo em relação às *unicasts*, visto que ela utiliza um único caminho para atingir todos destinos.

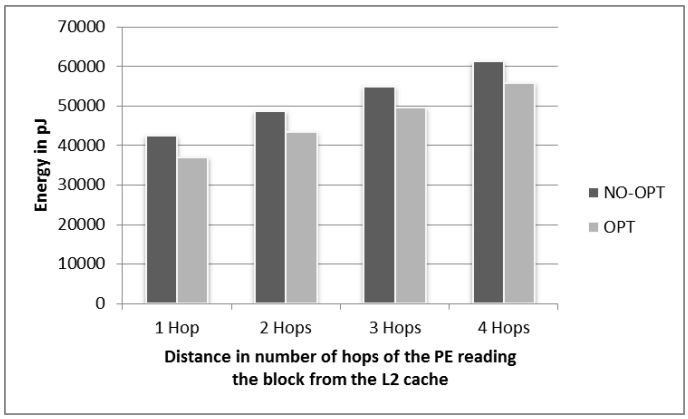


Figura 52 – Energia consumida durante a leitura de um bloco modificado variando a distância entre o PE leitor e a *cache* L2 [CHA11].

6.3.2 Produtor/Consumidores

Essa aplicação experimental consiste em um produtor que envia uma mensagem *multicast* e em seguida espera uma resposta (*acknowledge*) de cada consumidor [SUP99]. Os consumidores (destinos) recebem a mensagem e em seguida enviam a resposta ao produtor. A Listagem 2 mostra o núcleo do código do produtor. Dois tempos são capturados: (*i*) *t_Send*, corresponde ao tempo necessário para o produtor processar o envio da mensagem e (*ii*) *t_Ack*, corresponde ao tempo necessário para todos consumidores receberem a mensagem e confirmarem a recepção. Os experimentos foram realizados sobre uma instância 5x5 da plataforma HeMPS, fixando o produtor no centro da malha e variando o número de consumidores ao seu redor, implicando sempre o envio de duas cópias da mensagem.

Listagem 2 – Código do produtor.

```
t0 = GetTick();     /* Gets the initial time */

#ifdef DUAL_PATH    /* Sends the message using the dual-path algorithm */
Multicast(&msg,target_list,TARGETS);    /* High priority message by default
#else               /* Sends an individual message to each target */
for(i=0; i<TARGETS; i++)
       Send(&msg,target_list[i],HIGH);
#endif

t_Send = GetTick(); /* Gets the time to complete the sending */

for(i=0; i<TARGETS; i++)   /* Receives the acknowledgements */
Receive(&msg,target_list[i]);

t_Ack = GetTick();  /* Gets the time after receive all acknowledgements */
```

O primeiro experimento avalia os tempos *t_Send* e *t_Ack* em função do tamanho da mensagem. O tamanho da mensagem varia de 32 bytes a 512 bytes e o número de destinos foi fixado em 23. A Figura 53 ilustra os resultados obtidos, onde *DP* corresponde ao algoritmo *dual-path* e *Uni* corresponde ao envio de múltiplas mensagens *unicast*. Para mensagens pequenas (até 128 bytes), as duas implementações apresentam um desempenho muito próximo. Isto ocorre porque o processamento da primitiva `Multicast()` é mais complexo que o da primitiva `Send()` devido à preparação das cópias e este custo passa a ser compensado a partir de mensagens maiores (192 bytes). Visto que na implementação baseada em *unicast* a primitiva `Send()` é chamada 22 vezes a mais que a primitiva `Multicast()`, observa-se claramente um maior impacto no desempenho conforme o tamanho da mensagem aumenta.

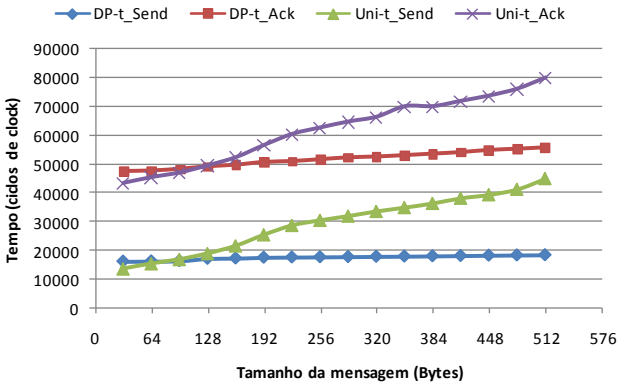


Figura 53 – *Multicast* em função do tamanho da mensagem.

O segundo experimento avalia os tempos *t_Send* e *t_Ack* em função do número de destinos para três tamanhos de mensagem: (*i*) pequeno (32 bytes); (*ii*) médio (256 bytes) e (*iii*) grande (512 bytes). O número de destinos varia de 2 a 23. As figuras a seguir ilustram os resultados obtidos para os três diferentes tamanhos de mensagem. A Figura 54 corrobora que para mensagens pequenas, ambas as implementações apresentam resultados próximos, independente do número de destinos. No entanto, o tráfego gerado pelo algoritmo *dual-path* é menor, acarretando menor consumo de energia [CAR08a][CHA11]. Já a Figura 55 e a Figura 56 mostram o crescente ganho de desempenho do algoritmo *dual-path* em relação à implementação baseada em *unicast* para mensagens maiores.

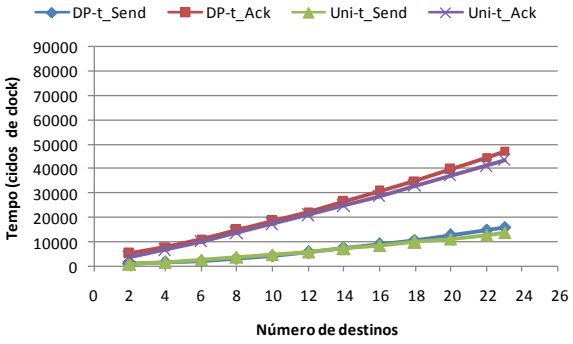


Figura 54 – *Multicast* em função do número de destinos. Mensagem de tamanho pequeno (32 bytes).

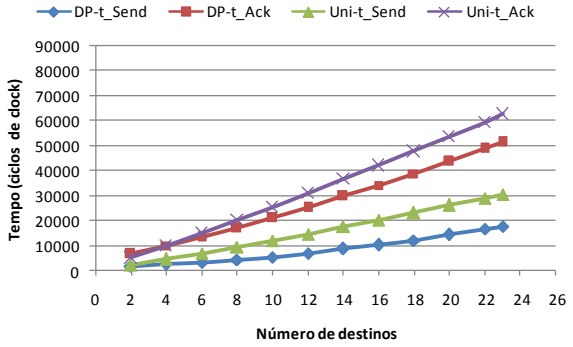


Figura 55 - *Multicast* em função do número de destinos. Mensagem de tamanho médio (128 bytes).

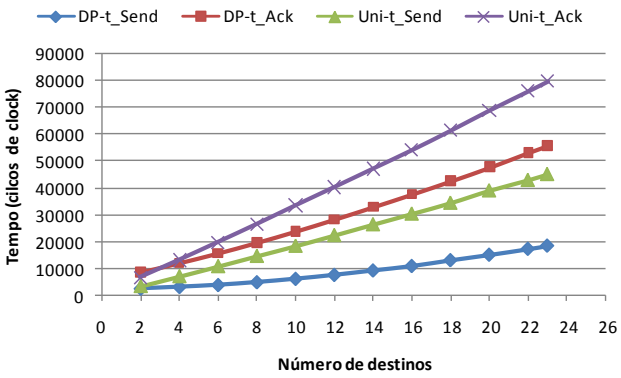


Figura 56 - *Multicast* em função do número de destinos. Mensagem de tamanho grande (512 bytes).

6.3.3 Multiplicação de matrizes distribuída

O multiplicador de matrizes distribuído é uma implementação do algoritmo de Fox [FOX87]. Esse algoritmo foi desenvolvido para realizar a multiplicação de matrizes de maneira distribuída em multicomputadores. O algoritmo assume matrizes de ordem n e um número de tarefas t onde a raiz quadrada de t divide n ($n \bmod \sqrt{t} = 0$). As matrizes a serem multiplicadas são particionadas em submatrizes de ordem $n/\sqrt{t}$ e atribuídas às tarefas. O particionamento segue um padrão de tabuleiro de damas. A Figura 57 ilustra duas matrizes (A e B) de ordem 12 (n) a serem multiplicadas, particionadas entre 9 tarefas (t).

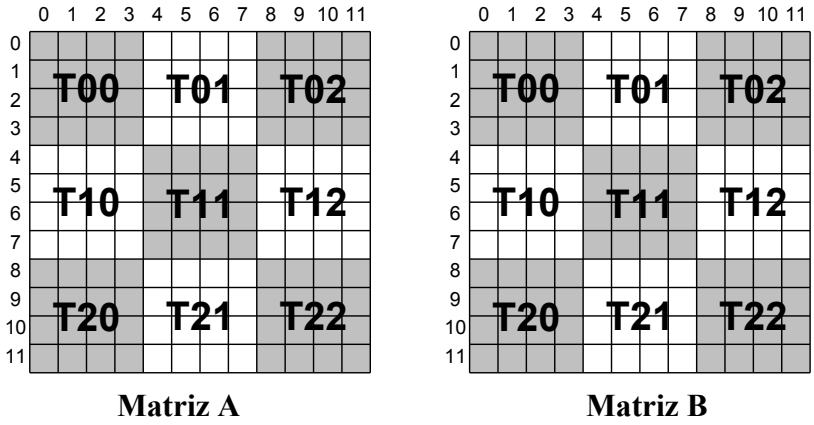


Figura 57 – Matrizes A e B particionadas entre 9 tarefas (T00, T01, T02, T10, T11, T12, T20 T21 e T22).

As tarefas podem ser vistas como um *grid* virtual de ordem $\sqrt{t}$, onde cada uma é responsável pela multiplicação das suas submatrizes de A e B. O algoritmo é executado em $\sqrt{t}$ iterações. A Figura 58 ilustra o padrão de comunicação entre as tarefas a cada iteração do algoritmo, considerando o *grid* virtual de tarefas 3x3 formado pelo particionamento das matrizes apresentado na Figura 57. A cada iteração é feito um *broadcast* de uma submatriz de A em cada uma das linhas do *grid* e em seguida cada tarefa envia a sua submatriz de B para vizinha ao norte (as tarefas na borda norte enviam

para as tarefas na borda sul na mesma coluna). As tarefas origem do *broadcast* em cada linha variam a cada iteração. Inicialmente o *broadcast* é feito pelas tarefas na diagonal principal e a cada iteração a diagonal é deslocada (com rotação) para a direita, indicando as novas tarefas origem.

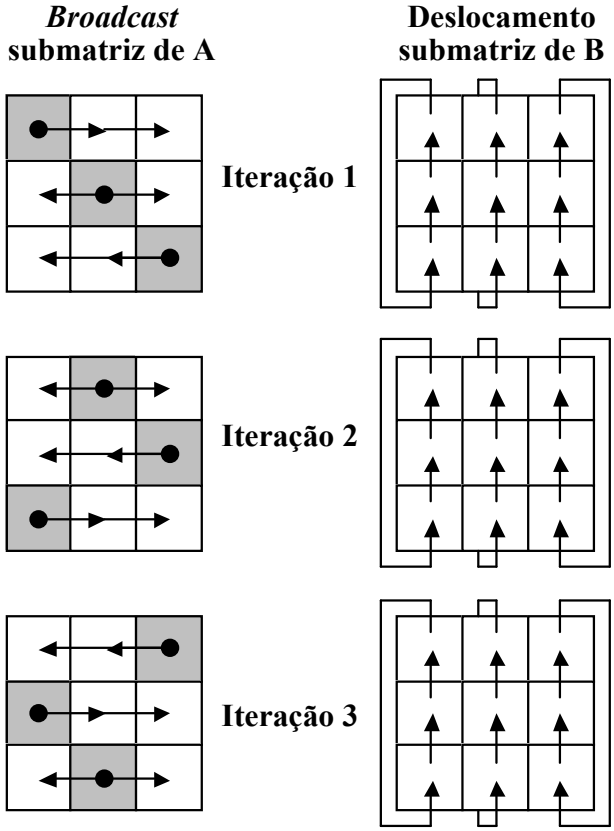


Figura 58 – Padrão de comunicação entre as tarefas a cada iteração do algoritmo de Fox.

O algoritmo de Fox foi implementado sobre a plataforma HeMPS em duas versões. Uma utiliza o *multicast dual-path* para fazer o *broadcast* da submatriz de A enquanto a outra o faz a partir de múltiplas mensagens *unicast*. Foram realizadas multiplicações de matrizes de ordem 33, 44 e 55, utilizando *grids* de tarefas de ordem 3, 4 e 5, respectivamente. Os *grids* de tarefas foram mapeados sobre instâncias da plataforma HeMPS de mesma ordem. Em todos experimentos cada tarefa é responsável pela multiplicação de submatrizes de ordem 11. A Tabela 7 apresenta os tempos em ciclos de *clock* para a conclusão das multiplicações e os ganhos obtidos a partir do emprego do *multicast dual-path*. No caso do *grid* 3x3, a versão com múltiplos *unicasts* tem um desempenho superior, pois o número de destinos do *broadcast* é pequeno (dois destinos por linha). A medida que o *grid* aumenta, e consequentemente o número de destinos do *broadcast*, o *multicast dual-path* vai apresentando ganhos crescentes. Devido à limitação discutida na seção 6.2, não foi possível realizar experimentos com *grids* de tarefas maiores. Entretanto, visto que a submatriz difundida em *broadcast* pode ser considerada uma mensagem grande (484 bytes), os ganhos proporcionados pelo *multicast dual-path* em *grids* maiores devem seguir os resultados da Figura 56.

Tabela 7 – Tempos em ciclos de *clock* para a realização das multiplicações das matrizes e ganhos obtidos com o *multicast dual-path*.

Matrizes	*Grid*	*Dual-path*	*Unicast*	Ganho
33x33	**3x3**	307639	305393	-0,73%
44x44	**4x4**	460570	463855	+0,7%
55x55	**5x5**	661087	669499	+1,25%

7. CONCLUSÃO

Este trabalho originou-se a partir da união natural de dois tópicos de pesquisa estratégicos que direcionam a evolução dos sistemas embarcados; redes e sistemas multiprocessados intra-chip (NoCs e MPSoCs). Visto que essas duas áreas evoluíram separadamente por anos, a integração deve considerar o melhor dos dois mundos; serviços de comunicação (NoCs) e programabilidade (MPSoCs). Revisando o estado da arte ficou claro que o potencial oferecido pelas NoCs não vinha sendo explorado adequadamente em níveis mais altos de abstração e que elas têm sido empregadas, em vários trabalhos, como meras infraestruturas para troca de mensagens. A estrutura nativamente distribuída das NoCs e as características intrínsecas ao paradigma oferecem aos projetistas a possibilidade de criar sistemas multiprocessados de grande poder computacional. A utilização de processadores de propósito geral como IPs (PEs) confere ao sistema flexibilidade e capacidade de personalização, o que garantem sua reutilização como um todo na implementação das mais variadas aplicações. Portanto, a característica mais relevante dos MPSoCs é sem dúvida a programabilidade, a qual foi explorada nessa Tese a partir dos serviços de comunicação diferenciados. A implementação de mecanismos específicos no nível da rede e o suporte a estes no nível de software a partir de uma API de comunicação consolidaram a integração entre NoCs e MPSoCs.

Os mecanismos implementados na NoC que dão suporte aos serviços de comunicação diferenciados são controlados pelo cabeçalho dos pacotes. Campos específicos no cabeçalho de um pacote expõem tais mecanismos a níveis superiores (IP). Diferente de diversos trabalhos relacionados, essa abordagem permite o acesso aos mecanismo sem a necessidade de interfaces de rede específicas, as quais os expõem através de registradores mapeados em memória. As interfaces de rede a serem utilizadas com esta NoC podem simplesmente fazer o papel de uma camada de adaptação entre as interfaces de comunicação ou entre as frequências de operação da rede e do IP no caso de MPSoCs GALS. A NoC em si pode ser vista como um IP de comunicação *stand alone* pronto para ser empregado em qualquer projeto, dando ao projetista a liberdade de criar suas próprias interfaces de rede, uma vez que diferentes IPs possuem diferentes interfaces.

A metodologia de integração proposta nesta Tese foi implementada em software no microkernel (HeMPS e HS-Scale). A ideia é capturar o serviço de comunicação a ser utilizado pela aplicação a partir das primitivas da API, e durante o processamento destas, o microkernel configura o cabeçalho do pacote de acordo com o serviço. Tal metodologia é simples e de baixo custo em termos de memória, uma vez que é implementada em software. A mesma metodologia pode ser implementada em uma biblioteca de funções específica, caso não haja um sistema operacional ou este não possua código aberto. Além disso, essa metodologia pode ser implementada por qualquer processador

utilizando uma linguagem de programação de alto nível (e.g. C ou C++). O microkernel tomado como ponto de partida para a realização deste trabalho tinha um tamanho de 15KB. Após a inclusão do suporte aos serviços de comunicação diferenciados e o escalonamento de tarefas baseado em prioridades o tamanho atual é 22KB. Em termos percentuais, pode-se dizer que o aumento de 46,6% (7KB) é considerável, entretanto o microkernel como um todo continua com um tamanho reduzido e adequado a sistemas embarcados.

Em termos de hardware, o maior impacto foi relativo à duplicação dos canais físicos para suportar o serviço de comunicação baseado em prioridades. Os demais serviços de comunicação implementados não exigem a adição/replicação de módulos hardware específicos à arquitetura do roteador. Os mecanismos que dão suporte a eles são implementados a partir de pequenas alterações nos módulos já existentes na arquitetura do roteador, acarretando baixo custo de área. Caso o roteador alvo já tenha os canais físicos replicados (e.g. iMesh – Seção 2.1.1), a implementação de um mecanismo de alocação baseado em prioridades também tem baixo custo. Além de aumentar a largura de banda agregada do roteador, múltiplos canais físicos podem ser explorados também para tolerância a falhas.

As avaliações realizadas mostraram a eficiência dos serviços de comunicação implementados e os benefícios obtidos controlando-os em software a partir da API. Tal controle oferece ao programador meios para explorar o espaço de projeto em busca dos melhores resultados, a fim de satisfazer os requisitos das aplicações. Tipicamente, os serviços de comunicação implementados proporcionam ganhos em situações de tráfego mais elevado onde há concorrência por recursos da NoC. Isso mostra que em situações onde a carga da rede é baixa, esta pode operar em uma frequência inferior à dos IPs, com o propósito de elevar o tráfego do ponto de vista da rede. Assim ela passa a consumir menos energia e os serviços de comunicação podem ser utilizados eficientemente. Os serviços de comunicação foram avaliados separadamente, entretanto, todos estão disponíveis na API de comunicação da plataforma HeMPS, podendo ser utilizados concomitantemente pelo programador das aplicações.

Os resultados obtidos a partir de simulações RTL agregam valor a estes, uma vez que o comportamento simulado é muito próximo do apresentado por uma implementação física. Apesar das simulações terem considerado instâncias pequenas de MPSoC (e.g. 4x4 e 5x5), os serviços de comunicação devem se comportar de maneira semelhante em instância maiores, visto que estes tem baixa dependência em relação as dimensões do sistema. Além do mais, MPSoCs de grandes dimensões podem ser divididos em regiões formadas por sub-MPSoCs com dimensões semelhantes às utilizadas neste trabalho.

Um significativo passo foi dado em relação à versão da plataforma HeMPS utilizada como ponto de partida para a realização deste trabalho. Além das implementações acrescentadas ao longo da Tese, muitos *bugs* foram resolvidos,

chegando-se a um nível mínimo de estabilidade do sistema que habilita a sua distribuição, de maneira que outros grupos de pesquisa além do GAPH possam ajudar a enriquecê-la e torná-la ainda mais estável, desenvolvendo novas aplicações e relatando eventuais *bugs*. A implementação de aplicações a partir de algoritmos paralelos (troca de mensagens) ou grafos de tarefas utilizando a API não apresenta grandes dificuldades. A plataforma HeMPS é uma fonte de conhecimento que envolve conceitos de diversas áreas da engenharia e da computação. Durante o período do doutorado foi possível exercitar a abrangência multidisciplinar envolvida no projeto de MPSoCs em vários níveis de abstração como:

- Programação VHDL e redes de computadores, durante o desenvolvimento de novos serviços comunicação na NoC Hermes;
- Sistemas operacionais, durante a implementação de novas primitivas de comunicação e escalonamento baseado em prioridades no microkernel da plataforma HeMPS;
- Interface hardware/software, durante o desenvolvimento de novos *drivers*;
- Programação SystemC e arquitetura de computadores, durante o desenvolvimento de modelos abstratos de processadores e memórias, a fim de acelerar a simulação da plataforma HeMPS;
- Programação paralela e arquiteturas paralelas durante o desenvolvimento de aplicações;
- Programação orientada a objetos, durante o desenvolvimento do *framework* de geração e depuração da plataforma HeMPS (Apêndice B – HeMPS Generator).

7.1 Trabalhos futuros

As garantias flexíveis oferecidas por serviços de comunicação como o baseado em prioridades (Capítulo 4) ou com roteamento diferenciado (Capítulo 5) podem ser severamente afetadas quando muitas aplicações compartilham os mesmos serviços simultaneamente. Tal situação foi evidenciada no experimento com o decodificador áudio/vídeo apresentado na Seção 4.3.2 (Tabela 3). Em um cenário onde todas as aplicações transmitiam fluxos de alta prioridade, foi necessário mudar o serviço de comunicação do *pipeline* de vídeo para um baseado em conexão, a fim de que o decodificador atingisse novamente a vazão mínima exigida. Tal mudança foi simples, entretanto, realizada manualmente a partir de uma alteração mínima no código fonte da aplicação.

Em sistemas dinâmicos onde a todo momento aplicações são alocadas/desalocadas, certos serviços de comunicação diferenciados são necessários apenas quando há concorrência pelos recursos da infra-estrutura de comunicação. Neste caso seria desejável o próprio sistema realizar a monitoração do desempenho das aplicações e dinamicamente adaptar o serviço de comunicação de maneira autônoma,

baseado apenas nas restrições especificadas pelo programador em tempo de projeto. Tais restrições podem ser obtidas através de um traçado de perfil (*profiling*) da aplicação onde são estabelecidos limites de desempenho, como vazão, latência e *jitter*. Dependendo do mapeamento de uma aplicação na plataforma, o seu desempenho geral pode ser afetado em decorrência da deterioração da comunicação de apenas algumas tarefas, devido às dependências entre estas. Portando, uma possível abordagem é realizar uma monitoração entre pares de tarefas comunicantes e ajustar o serviço de comunicação localmente, ao invés de interferir globalmente em toda a aplicação. Tal monitoração e ajuste do serviço de comunicação podem ser implementados no microkernel como trabalhos futuros.

REFERÊNCIAS BIBLIOGRÁFICAS

[AGA06] Agarwal, A.; Mustafa, M.; Pandya, A. S. "*QoS Driven Network-on-Chip Design for Real Time Systems*". In: Canadian Conference on Electrical and Computer Engineering (CCECE'06), 2006, pp. 1291 – 1295.

[AHM06] Ahmad, B.; Erdogan, A. T.; Khawam S. "*Architecture of a Dynamically Reconfigurable NoC for Adaptive Reconfitgurable MPSoC*". In: Adaptive Hardware and Systems (AHS'06), 2006, pp. 405-411.

[ALM09] Almeida, G. M. et al. "*An Adaptive Message Passing MPSoC Framework*". International Journal of Reconfigurable Computing, vol. 2009, 20p, 2009.

[ALM10] Almeida, G. M. et al. "*Evaluating the Impact of Task Migration in Multi-Processor Systems-on-Chip*". In: Symposium on Integrated Circuits and Systems (SBCCI'10), 2010, pp. 74-78.

[ALO10] Alonso, E. F. et al. "*A NoC-based Multi-{soft}core with 16 cores*". In: International Conference on Electronics, Circuits and Systems (ICECS'10), 2010, pp. 259-252.

[BAG08] Bagherzadeh, N.; Matsuura, M. "*Performance Impact of Task-to-Task Communication Protocol in Network-on-Chip*". In: International Conference on Information Technology: New Generations (ITNG'08), 2008, pp. 1101-1106.

[BAI02] Bainbridge, J.; Furber, S. "*Chain: a Delay-insensitive Chip Area Interconnect*". IEEE Micro, v.22(5), 2002, pp. 16-23.

[BAL95] Bala, V. et al. "*CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers*". IEEE Transactions on Parallel and Distributed Systems, v.6(2), 1995, pp. 154-164.

[BEN05] Benini, L. et al. "*MPARM: Exploring the Multi-Processor SoC Design Space with SystemC*". The Journal of VLSI Signal Processing, v.41(2), 2005, pp. 169–182.

[BER04] Bertozzi, D.; Benini, L. "*Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip*". Circuits and Systems Magazine, v.4(2), 2004, pp. 18–31.

[BIN06] Binkert, N. L. et al. "The *M5 Simulator: Modeling Networked Systems*". IEEE Micro, v.26(4), 2006, pp. 52-60.

[BOL01] Bolleta, G. et al. "*The Real-Time Specification for Java*". Capturado em: www.rtsj.org, 2010.

[BOP98] Boppana, R. V.; Chalasani, S.; Raghavendra, C. S. *"Resource Deadlocks and Performance of Wormhole Multicast Routing Algorithms"*. IEEE Transactions on Parallel and Distributed Systems, v.9(8), 1998, pp. 535-549.

[BUR10] Burgio, P. et al. "*Adaptive TDMA Bus Allocation and Elastic Scheduling: A Unified Approach for Enhancing Robustness in Multi-core RT Systems"*. In: International Conference on Computer Design (ICCD'10), 2010, pp. 187-194.

[CAR07] Carara, E.; Calazans, N.; Moraes, F.; "*Router Architecture for High-Performance NoCs*". In: Symposium on Integrated Circuits and Systems (SBCCI'07), 2007, pp. 111-116.

[CAR08a] Carara, E. "*Estratégias para Otimização de Desempenho em Redes Intra-Chip - Implementação e Avaliação sobre a Rede Hermes*". Dissertação de Mestrado, FACIN, Pontifícia Universidade Católica do Rio Grande do Sul, 2008, 89p.

[CAR08b] Carara, E.; Calazans, N.; Moraes, F.; "*A New Router Architecture for High-Performance Intrachip Networks*". Journal of Integrated Circuits and Systems, v.3(1), 2008, pp. 23-31.

[CAR09a] Carara, E. et al. "*HeMPS - A Framework for NoC-Based MPSoC Generation*". In: International Symposium on Circuits and Systems (ISCAS'09), 2009, pp. 1345–1348.

[CAR09b] Carara, E.; Calazans, N.; Moraes, F. "*Managing QoS Flows at Task Level in NoC-based MPSoCs*". In: International Conference on Very Large System Integration (VLSI-SoC'2009), 2009.

[CAR10] Carara, E.; Moraes, F. "*Flow Oriented Routing for NoCs*". In: International SoC Conference (SOCC'10), 2010, pp. 367-370.

[CAR11] Carara, E.; Almeida, G. M.; Sassatelli, G.; Moraes, F. "*Achieving Composability in NoC-Based MPSoCs through QoS Management at Software Level*". In: Design, Automation and Test in Europe (DATE'11), 2011, pp. 407-412.

[CEN08] Ceng, J. et al. "*MAPS: An Integrated Framework for MPSoC Application Parallelization*". In: Design Automation Conference (DAC'08), 2008, pp. 754-759.

[CHA11] Chaves, T. M.; Carara, E. A.; Moraes, F. G. "*Energy-Efficient Cache Coherence Protocol for NoC-based MPSoCs*". In: Symposium on Integrated Circuits and Systems (SBCCI'11), 2011, pp. 215-220.

[CHI00] Chiu, G. *"The Odd-Even Turn Model for Adaptive Routing"*. IEEE Transactions on Parallel and Distributed Systems, v.7(11), 2000, pp. 729-738.

[DAL87] Dally, W. J.; Seitz, C. L. *"Deadlock-Free Message Routing in Multiprocessors Interconnection Networks"*. IEEE Transactions on Computers v.36(5), 1987, pp.547-553.

[DOR05] Dorta, A.; Rodriguez, C.; Sande, F. "*The OpenMP Source Code Repository*". In: 13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05), 2005, pp. 244–250.

[DUA02] Duato, J.; Yalamanchili, S.; Ni, L."*Interconnection Networks*". Morgan Kaufmann, 2002, 624p.

[EBR10] Ebrahimi, M.; Daneshtalab, M.; Liljeberg, P.; Tenhunen, H. "*HAMUM - A Novel Routing Protocol for Unicast and Multicast Traffic in MPSoCs*". In: 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'10), 2010, pp. 525 – 532.

[ERL01] Erlandsson, M. "*Openrisc1000*". Capturado em: http://opencores.org/project,or1k, 2010.

[FOX87] Fox, G. C.; Otto, S. W.; Hey, A. J. G. "*Matrix Algorithms on a Hypercube I: Matrix Multiplication*". Parallel Computing, v.4(1), 1987, pp. 17–31.

[FU10] Fu, F. et al. "*MMPI: A Flexible and Efficient Multiprocessor Message Passing Interface for NoC-Based MPSoC*". In: International SoC Conference (SOCC'10), 2010, pp. 359-362.

[GIL10] Gilabert, F.; Gómez, M.E.; Medardoni, S.; Bertozzi, D. "*Improved Utilization of NoC Channel Bandwidth by Switch Replication for Cost-Effective Multi-Processor Systems-on-Chip*". In: International Symposium on Networks-on-Chip (NOCS'10), 2010, pp. 165 – 172.

[GLA94] Glass, C. J.; Ni, L. M. "*The Turn Model for Adaptive Routing*". Journal of the Association for Computing Machinery, v.41(5), 1994, pp. 874-902.

[GOO05] Goossens, K.; et al. "*Æthereal Network-on-Chip: Concepts, Architectures, and Implementations*". IEEE Design & Test of Computers, v.22(5), 2005, pp. 414-421.

[HAN09] Hansson, A.; Goossens, K.; Bekooij, M.; Huisken, J. "*CoMPSoC: A Template for Composable and Predictable Multi-Processor System on Chips*". ACM Transactions on Design Automation of Electronic Systems, v.14(1), 2009, pp. 1-24.

[HAR72] Harary, F. *"Graph Theory"*. Addison-Wesley, 1972, 274p.

[HU03] Hu, J; et al. "*Energy-aware Mapping for Tile-based NoC Architectures under Performance Constraints*". In: Asia and South Pacific Design Automation Conference (ASP-DAC'03), 2003, pp. 233-239.

[HU04] Hu, J.; Marculescu, R. "*DyAD - Smart Routing for Networks-on-Chip*". In: Design Automation Conference (DAC'04), 2004, pp. 260-263.

[ITO01] Ito, S. A.; Carro, L.; Jacobi, R. P. "*Making Java Work for Microcontroller Applications*". IEEE Design & Test of Computers, v.18(5), 2001, pp. 100-110.

[JAL04] Jalabert, A.; Murali, S.; Benini, L.; De Micheli, G.; *"×pipesCompiler: A Tool for Instantiating Application Specific Networks-on-Chip*". In: Design, Automation and Test in Europe (DATE'04), 2004, pp. 884 – 889.

[JOV08] Joven, J.; Carrabina, J. et al *"xENOC – An eXperimental Network-on-Chip Enviroment for Parallel Distributed Computing on NoC-based MPSoC Archtectures*". In: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'08), 2008, pp. 141-148.

[JOV09] Joven, J. "*A Lightweight MPI-based Programming Model and its HW Support for NoC-based MPSoCs*". In: PhD Forum Design, Automation and Test in Europe (DATE'09), 2009.

[KAH74] G. Kahn. "*The Semantics of a Simple Language for Parallel Programming*". In: Proceedings of IFIP Congress on Information Processing, 1974, pp. 471-475.

[KAK11] Kakoee, M. R.; Bertacco, V.; Benini, L. "*ReliNoC: A Reliable Network for Priority-Based on-Chip Communication*". In: Design, Automation and Test in Europe (DATE'11), 2011, pp. 491-496.

[KUM07] Kumar, A. et al. "*An FPGA Design Flow for Reconfigurable Network-Based Multi-Processor Systems on Chip*". In: Design, Automation and Test in Europe & Exhibition, (DATE'07), 2007, pp. 117-122.

[KUM08] Kumar, K.; Mesman, B.; Theelen, B.; Corporaal, H.; Ha, H. "*Analyzing Composability of Applications on MPSoC Platforms"*. Journal of Systems Architecture: The EUROMICRO Journal, v.54(3-4), 2008, pp. 369-383.

[LEE07] Lee, H. G. et al. “*On-chip Communication Architecture Exploration: A Quantitative Evaluation of Point-to-point, Bus, and Network-on-Chip Approaches*”. ACM Transactions on Design Automation of Electronic Systems, v.12(3), 2007, pp. 1-20.

[LEE08] Lee, S.; Bagherzadeh, N. “*NePA: Networked Processor Array for High Performance Computing*”. In: Architecture of Computing Systems (ARCS’08), 2008, 6p.

[LEU10] Leupers, R. et al. “*Cool MPSoC Programming*”. In: Design, Automation and Test in Europe & Exhibition (DATE’10), 2010, pp. 1488-1493.

[LI91] Li, J.; Chen, M. “*Compiling Communication-Efficient Programs for Massively Parallel Machines*”. IEEE Transactions on Parallel and Distributed Systems, v.2(3), 1991, pp. 361-375.

[LI03] Li, Q.; Yao, C. “*Real-Time Concepts for Embedded Systems*”. CPM Books, 2003, 294p.

[LIN94] Lin, X.; McKinley, P. K.; Ni, L. M. “*Deadlock-free Multicast Wormhole Routing in 2-D Mesh Multicomputers*”. IEEE Transactions on Parallel and Distributed Systems, v.5(8), 1994, pp. 793-804.

[MAN11a] Mandelli, M. et al. “*Energy-Aware Dynamic Task Mapping for NoC-based MPSoCs*”. In: International Symposium on Circuits and Systems (ISCAS’11), 2011, pp. 1676-1679.

[MAN11b] Mandelli, M.; Amory, A.; Ost, L.; Moraes, F. “*Multi-Task Dynamic Mapping onto NoC-based MPSoCs*”. In: Symposium on Integrated Circuits and Systems (SBCCI'11), 2011, pp. 191-196.

[MAR09] Marongiu, A.; Benini, L. “*Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy*”. In: Design, Automation and Test in Europe & Exhibition (DATE’09), 2009, pp. 809–814.

[µCL10] “*The uClinux Embedded Linux/Microcontroller Project*”. Capturado em: http://www.uclinux.org, 2010.

[MEL06] Mello, A.; Tedesco, L.; Calazans, N.; Moraes, F. "*Evaluation of Current QoS Mechanisms in Networks on Chip*". In: International Symposium on System-on-Chip (SOC’06), 2006, 4p.

[MOO65] Moore, G. “*Cramming More Components Onto Integrated Circuits*”. Electronics, vol. 38(8), 1965, pp. 82-85.

[MOR04] Moraes, F. et al. “*HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip*”. Integration the VLSI Journal, v.38(1), 2004, pp. 69-93.

[MOT11] Motakis, A.; Kornaros, G.; Coppola, M. “*Dynamic Resource Management in Modern Multicore SoCs by Exposing NoC Services*”. In: International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC’11), 2011, 6p.

[MUR09] Murilo, J. J.; “*HW-SW Components for Parallel Embedded Computing on NoC-Based MPSoCS*”. Tese de Doutorado, Universitat Autonòma de Barcelona, 2009, 218p.

[NI93] Ni, L. M.; McKinley, P. K. “*A Survey of Wormhole Routing Techniques in Direct Networks”*. IEEE Computer Magazine, v.26(2), 1993, pp. 62-76.

[NS2] "*The network simulator - ns-2"*. Capturado em: http://www.isi.edu/nsnam/ns, 2010.

[OBE10] Obermaisser, R.; Kopetz, H.; Paukovits, C. "*A Cross-Domain Multiprocessor System-on-a-Chip for Embedded Real-Time Systems*". IEEE Transactions on Industrial Informatics, v.6(4), 2010, pp. 548-567.

[ONI09] Onieva, E. et al. "*A Modular Parametric Architecture for the TORCS Racing Engine*". In: IEEE Symposium on Computational Intelligence and Games (CIG'09), 2009, pp. 256–262.

[OPE97] "*The OpenMP API Specification for Parallel Programming*". Capturado em: http://www.openmp.org, 2010.

[RAD04] Radulescu, A. et al. "*An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration*". In: Design, Automation and Test in Europe (DATE'04), 2004, pp. 878-883.

[RAJ07] Rajeshwar, K.; Roozeboom, F. "*Gordon E. Moore and His Legacy: Four Decades and* Counting". The Electrochemical Society Interface, 2007, pp. 11-13.

[RHO01] Rhoads, S. "*Plasma - Most MIPS I(TM) Opcodes*". Capturado em: http://www.opencores.org/project,plasma, 2010.

[ROS10] Rossi, D. et al. "*A Heterogeneous Digital Signal Processor for Dynamically Reconfigurable Computing*". IEEE Journal of Solid-State Circuits, v.45(8), 2010, pp. 1615–1626.

[RTE10] "*Real-Time Executive for Multiprocessor Systems (RTEMS)*". Capturado em: http://www.rtems.com, 2010.

[RUF09] Rufas, D. C. et al. "*NoCMaker: A Cross-Platform Open-Source Design Space Exploration Tool for Networks-on-Chip*". In: 4th Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC'09), 2009.

[SIL06] "*Silicon Hive"*. http://www.silicon-hive.com, 2006.

[SIL08] Silva, E.; Barcelos, D.; Wagner, F.; Pereira, C. "*A Virtual Platform for Multiprocessor Real-Time Embedded Systems"*. In: International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'08), 2008, 6p.

[SIN10] Singh, A. K.; Kumar, A.; Srikanthan, T.; Ha, Y. "*Mapping Real-life Applications on Run-time Reconfigurable NoC-based MPSoC on FPGA*". In: The 2010 International Conference on Field-Programmable Technology (FPT'10), 2010, pp. 365-368.

[SUP99] Supinski, B. R.; Karonis, N. T. "*Accurately Measuring MPI Broadcasts in a Computational Grid*". In: Proceedings of the 8th International Symposium on High Performance Distributed Computing (HPDC'99), 1999, pp. 29-37.

[TAY02] Taylor, M. B. tt al. "*The Raw Microprocessor: a Computational Fabric for Software Circuits and General-Purpose Programs*". IEEE Micro, v.2(22), 2002, pp. 25–35.

[THE07] Theelen, B. et al. "*Software/Hardware Engineering with the Parallel Object-Oriented Specification Language*". In: 5th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '07), 2007, pp. 139–148.

[TIL07] Tilera Corporation. "*TILE64^{TM} Processor*". Product Brief Description, 2007.

[VAN07] Vangal, S. et al. "*An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS*". In: IEEE International Solid-State Circuits Conference (ISSCC'07), 2007. pp. 5-7.

[VAN08] Vangal, S. et al. "*An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS*". IEEE Journal of Solid-State Circuits, v.43(1), 2008. pp. 29-41.

[WAL96] Walker, D.; Dongarra, J. "*MPI: A Standard Message Passing Interface*". Supercomputer, v.12, 1996, pp. 56-68.

[WEN07] Wentzlaff, D. et al. "*On-Chip Interconnection Architecture of the Tile Processor*". IEEE Micro, v.27(5), 2007, pp. 15-31.

[WOL08] Wolf, W.; Jerraya, A.; Martin, G. "*Multiprocessor System-on-Chip (MPSoC) Technology*". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, v.27(10), 2008, pp. 1701–1713.

[WOO95] Woo, S. C.; Ohara, M.; Torrie, E.; Singh, J. P.; Gupta, A. "*The SPLASH-2 Programs: Characterization and Methodological Considerations*". In: International Symposium on Computer Architecture (ISCA'95), 1995, pp. 24-36.

[YAN09] Yang, S.; Furber, S. B.; Plana, L. A. "*Adaptive Admission Control on the SpiNNaker MPSoC*". In: 22nd IEEE International SOC Conference (SOCC'09), 2009, pp. 243-246.

[YAN10] Yang, Z. J. et al. "*An Area-efficient Dynamically Reconfigurable Spatial Division Multiplexing Network-on-Chip with Static Throughput Guarentee*". In: The 2010 International Conference on Field-Programmable Technology (FPT'10), 2010, pp. 389 – 392.

[YOO10] Yoon, Y. J.; Concer, N.; Petracca, M.; Carloni, L. "*Virtual Channels vs. Multiple Physical Networks: A Comparative Analysis*". In: Design Automation Conference (DAC'10), 2010, pp. 162-165.

[YU10] Yu, M. et al. "*A Fast Timing-Accurate MPSoC HW/SW Co-Simulation Platform Based on a Novel Synchonization Scheme*". In: International MultiConference of Engineers and Computer Scientists (IMECS'10), 2010, pp. 1396-1400.

[ZEF03] Zeferino, C. A.; Susin, A. A. *"SoCIN: a Parametric and Scalable Network-on-Chip*". In: Symposium on Integrated Circuits and Systems (SBCCI'03), 2003, pp. 169-174.

APÊNDICE A – PUBLICAÇÕES

A Tabela 8 apresenta o conjunto das publicações realizadas durante o período do doutorado (2008-2011), juntamente da respectiva classificação no Qualis da CAPES (março/2010). A coluna *descrição* relaciona a publicação com o texto da presente Tese, quando se aplica, ou o tema principal da publicação. As publicações 3, 5 e 7 são resultantes do período de sanduíche realizado no laboratório LIRMM (Montpellier/França). A exceção da publicação 6 (pôster), todas as demais foram foram publicadas como artigos completos.

Tabela 8 - Publicações realizadas durante o período do doutorado (2008-2011).

	Publicação	Descrição
1	CHAVES,T.; CARARA,E.; MORAES,F. *Energy-efficient Cache Coherence Protocol for NoC-based MPSoCs* In: **SBCCI**, 2011 (Qualis B3)	Avaliação da comunicação coletiva no Capítulo 6
2	CHAVES,T.; CARARA,E.; MORAES,F. *Exploiting Multicast Messages in Cache-Coherence Protocols for NoC-based MPSoCs* In: **ReCoSoC**, 2011.	Utilização de *multicast* no protocolo de coerência de cache na HeMPS
3	CARARA,E.; ALMEIDA,G.; SASSATELLI,G.; MORAES,F. *Achieving Composability in NoC-Based MPSoCs Through QoS Management at Software Level*. In: **DATE**, 2011 (Qualis B1).	Capítulo 4
4	MANDELLI,M.; OST,L.; CARARA,E.; GUINDANI,G.; ROSA,T.; MEDEIROS,G.; MORAES,F. *Energy-Aware Dynamic Task Mapping for NoC-based MPSoCs*. In: **ISCAS**, 2011 (Qualis A2).	Mapeamento dinâmico implementado na HeMPS
5	ALMEIDA,G.; BUSSEUIL,R.; CARARA,E.; HERBERT,N.; VARYANI,S.; SASSATELLI,G.; BENOIT,P.; TORRES,L.; MORAES,F. *Predictive Dynamic Frequency Scaling for Multi-Processor Systems-on-Chip*. In: **ISCAS**, 2011 (Qualis A2).	Suporte a DFS implementado na HS-Scale
6	CARARA,E.; MORAES,F. *Flow Oriented Routing for NoCs*. In: **SOCC**, 2010 (Qualis B1).	Capítulo 5
7	ALMEIDA,G.; VARYANI,S.; BUSSEUIL,R.; SASSATELLI,G.; TORRES,L.; BENOIT,P.; CARARA,E.; MORAES,F. *Evaluating the Impact of Task Migration in Multi-Processor Systems-on-Chip*. In: **SBCCI**, 2010 (Qualis B3).	Migração de tarefas implementado na HS-Scale
8	CARARA,E.; OLIVEIRA,R.; CALAZANS,N.; MORAES,F. *HeMPS - A Framework for NoC-Based MPSoC Generation*. In: **ISCAS**, 2009 (Qualis A2).	Descrição da plataforma HeMPS, Seção 3
9	CARARA,E.; CALAZANS,N.; MORAES,F. *Managing QoS Flows at Task Level in NoC-Based MPSoCs*. In: **VLSI-SoC**, 2009 (Qualis B3).	Capítulo 4
10	CARARA,E.; MORAES,F. *Deadlock-Free Multicast Routing Algorithm for Wormhole-Switched Mesh Networks-on-Chip*. In: **ISVLSI**, 2008 (Qualis B4).	Capítulo 6
11	CARARA,E.; CALAZANS,N.; MORAES,F. *A New Router Architecture for High-Performance Intrachip Networks*. **JICS**. Journal of Integrated Circuits and Systems, v. 3, 2008.	Base para o roteador utilizado no trabalho

12	MORAES,F.; CARARA,E.; PIGATTO,D.; CALAZANS,N. *MOTIM an Industrial Application Using NOCs.* In: **SBCCI**, 2008 (Qualis B3).	Aplicação industrial implementada utilizando o roteador proposto na publicação 11

APÊNDICE B – HEMPS GENERATOR

HeMPS Generator é o nome do *framework* utilizado para geração automatizada da plataforma HeMPS e depuração das aplicações. O *framework* é baseado em uma interface gráfica amigável e intuitiva. A Figura 59 ilustra a janela principal do *HeMPS Generator*.

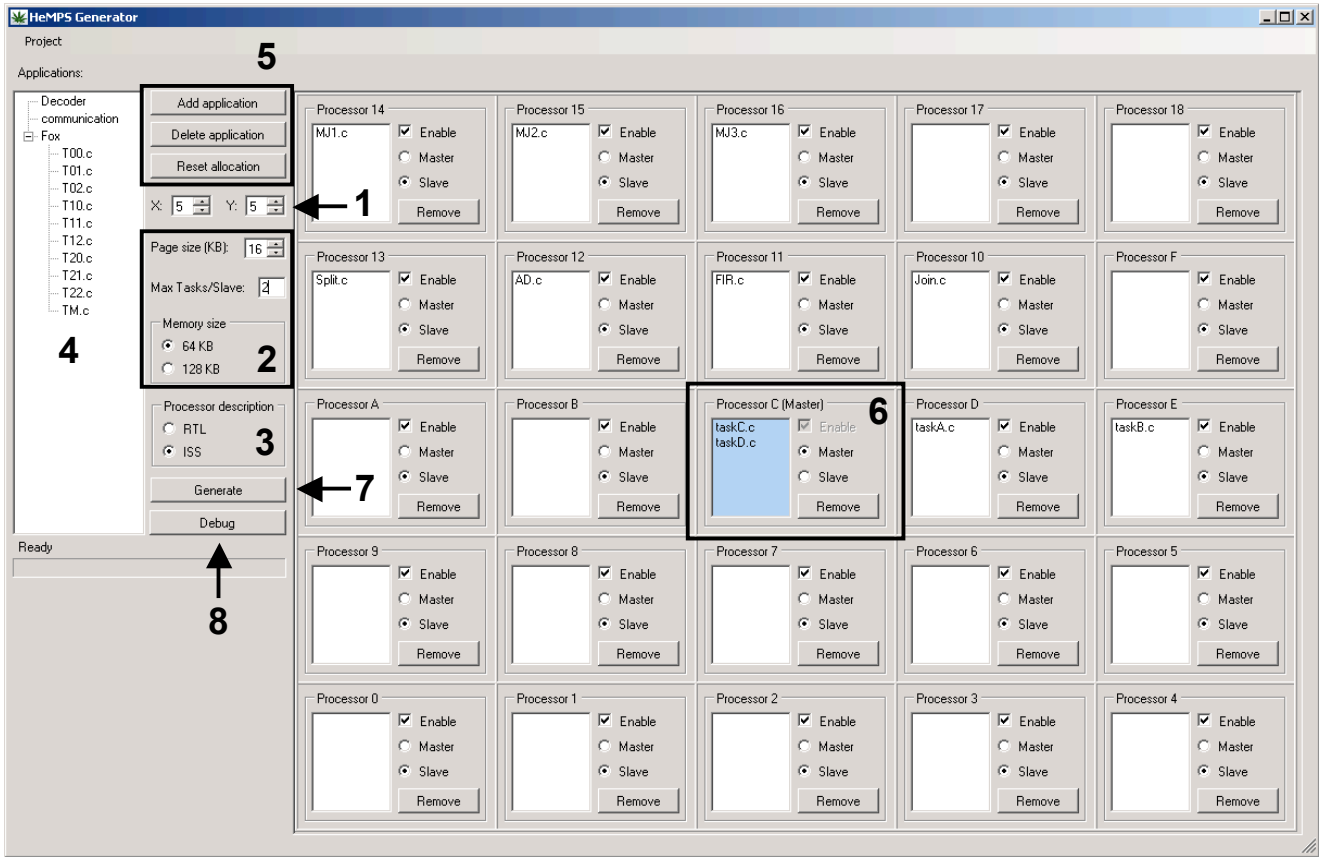


Figura 59 – Janela principal do *framework HeMPS Generator*.

Essa janela habilita a personalização da plataforma a partir de diversos parâmetros. Primeiramente, o número de PEs da plataforma pode ser definido através das dimensões *X* e *Y* (1), as quais também representam as dimensões da NoC Hermes a ser utilizada. Depois, pode-se definir o tamanho de página e tamanho total da memória privada dos PEs (2). O número máximo de tarefas executadas pelos PEs é uma função dos tamanhos de página e da memória privada, sendo que uma página comporta apenas uma tarefa. Para acelerar a simulação da plataforma, processadores e memórias privadas podem ser modelados, respectivamente, a partir de ISSs e modelos C/SystemC, ambos com precisão de ciclo (3).

O painel à esquerda na janela principal (4) contém as aplicações a serem executadas na plataforma. Cada aplicação é uma raiz, cujas folhas são as tarefas que as compõem. Uma nova aplicação pode ser adicionada ou removida do painel através dos botões *Add application* e *Delete application* (5), respectivamente. O mapeamento estático é determinado arrastando uma tarefa do painel de aplicações até o processador escravo

onde ela deve executar. As tarefas arrastadas até o processador mestre (6) são mapeadas e alocadas dinamicamente sob demanda durante a execução do sistema. Qualquer PE pode ser o mestre do sistema, sendo este definido através do *radio button Master*. Um duplo clique em uma tarefa arrastada até um PE abre uma nova janela com as propriedades da tarefa (Figura 60). Atualmente é possível definir a prioridade de escalonamento e o *time slice* de cada tarefa (Seção 4.4). Ao pressionar o botão *Reset allocation* (5), todas as tarefas arrastadas até os PEs voltam para o painel de aplicações juntamente das suas respectivas aplicações.

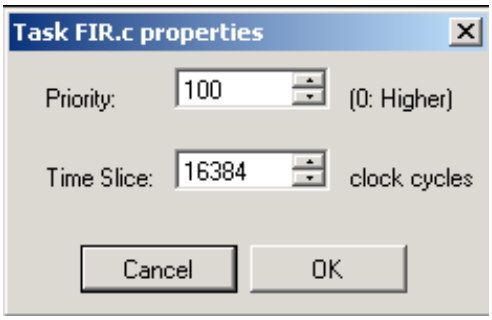


Figura 60 – Janela de propriedades da tarefa.

O botão *Generate* (7) dispara a geração automatizada do sistema. Primeiramente o hardware da plataforma é gerado segundo os parâmetros definidos na janela principal. Em seguida são compilados os códigos das tarefas e do microkernel. Por fim, os códigos das tarefas são incluídos no repositório enquanto o microkernel é incluído nas memórias privadas dos PEs. A plataforma gerada pode ser totalmente descrita em VHDL RTL, ou uma mistura de VHDL e C/SystemC, para fins de simulação (3).

Após a geração, pode-se simular a plataforma resultante através de um simulador. No grupo de pesquisa GAPH o simulador utilizado é o *ModelSim* (Mentor). Uma vez concluída a simulação, o resultado da execução das aplicações pode ser observada a partir da janela de depuração (Figura 61), a qual é aberta pelo botão *Debug* (8). Nessa janela, cada painel corresponde a um PE do sistema. Dentro de cada painel tem-se uma aba para cada tarefa executada no PE. Tais abas equivalem a um terminal que mostra o resultado da execução das tarefas.

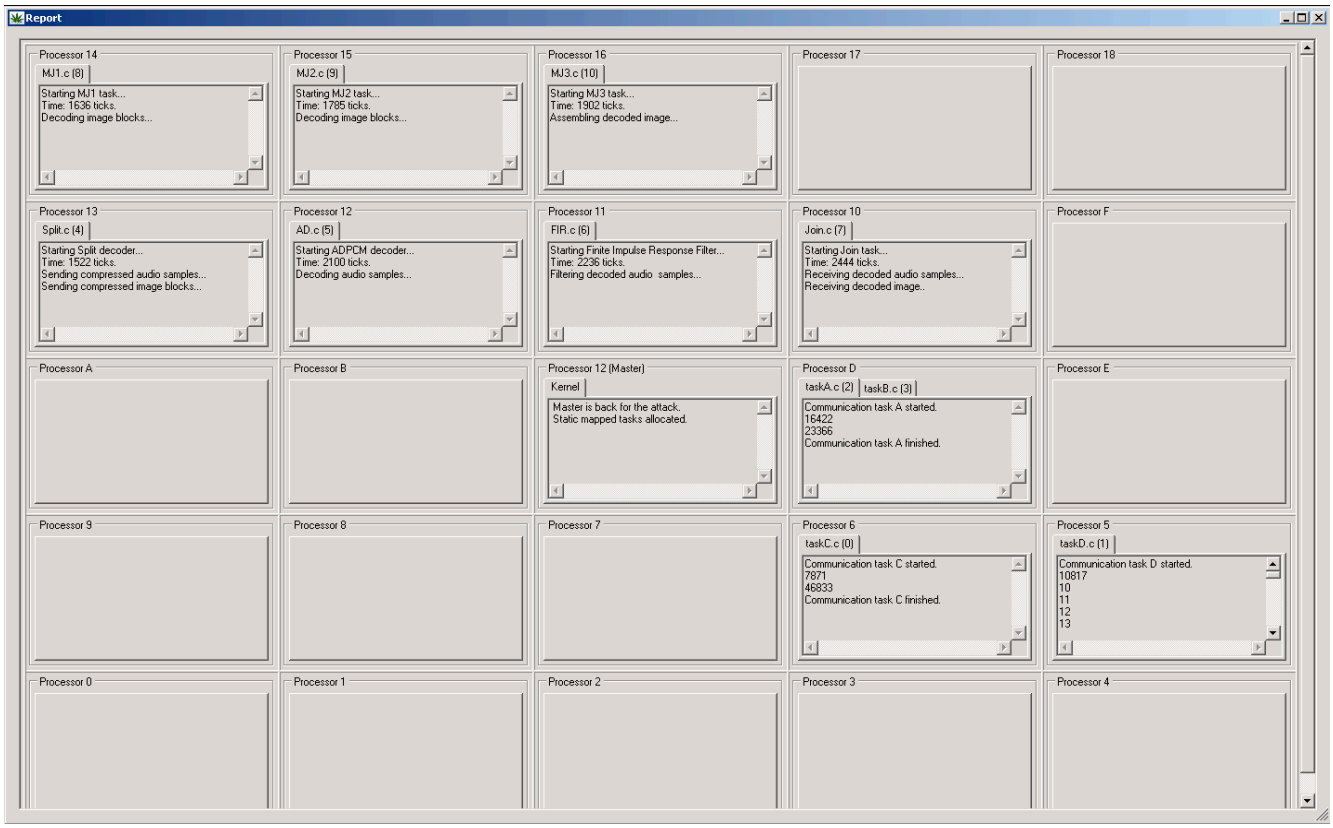


Figura 61 – Janela de depuração.