PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALOCAÇÃO DE TAREFAS E COMUNICAÇÃO ENTRE TAREFAS EM MPSOCS

por

CRISTIANE RAQUEL WOSZEZENKI

Prof. Dr. Fernando Gehm Moraes
Orientador

Dissertação de mestrado submetida como requisito parcial
à obtenção do grau de Mestre em Ciência da Computação.

Porto Alegre, Março de 2007.

PUCRS
Campus Central
Av. Ipiranga, 6681 - prédio 16 - CEP 90619-900
Porto Alegre - RS - Brasil
Fone: +55 (51) 3320-3544 - Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

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

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Alocação de Tarefas e Comunicação Entre Tarefas em MPSoCs***", apresentada por Cristiane Raquel Woszezenki, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas Embarcados e Sistemas Digitais, aprovada em 28/02/2007 pela Comissão Examinadora:

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

Prof. Dr. Avelino Fancisco Zorzo — PPGCC/PUCRS

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

Prof. Dr. Altamiro Amadeu Susin — UFRGS

Homologada em 16/04/07, conforme Ata No. 009 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P. 16 – sala 106 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@inf.pucrs.br
www.pucrs.br/facin/pos

Resumo

MPSoCs (do inglês, *Multiprocessor System On Chip*) constituem uma tendência no projeto de sistemas embarcados, pois possibilitam o melhor atendimento dos requisitos da aplicação. Isso se deve ao fato de que a arquitetura desses sistemas é composta por vários processadores, módulos de hardware dedicados, memória e meio de interconexão, fornecendo um maior poder computacional quando comparados a sistemas monoprocessados equivalentes. No entanto, estratégias que possibilitem o aproveitamento da capacidade de processamento destas arquiteturas precisam ser mais bem entendidas e exploradas. Para isso, é necessário dispor de infra-estruturas de hardware e software que habilitem gerenciar a execução de tarefas no MPSoC. A partir destas infra-estruturas deve ser possível, por exemplo, mapear tarefas dinamicamente nos processadores, balanceando a carga de trabalho do MPSoC através de estratégias de alocação dinâmica de tarefas.

O estado da arte da bibliografia no tema explora estratégias de alocação estática e dinâmica de tarefas sobre MPSoCs e avalia a viabilidade e eficiência das mesmas. Contudo, a necessidade de criação das infra-estruturas de hardware e software para viabilizar a exploração destas estratégias constitui-se um gargalo no avanço desta tecnologia. Adicionalmente, a maioria dos trabalhos utiliza plataformas modeladas em níveis muito abstratos de modelagem para avaliação das abordagens pesquisadas, reduzindo a confiabilidade dos resultados relatados.

A principal contribuição do presente trabalho é a proposta e implementação de uma plataforma MPSoC denominada HMPS (*Hermes Multiprocessor System*). HMPS conta com uma infra-estrutura de hardware e uma infra-estrutura de software, capazes de gerenciar a execução de tarefas no sistema. A plataforma HMPS é baseada em multiprocessamento homogêneo, e possui uma arquitetura de processadores mestre-escravo. A plataforma utiliza como meio de interconexão uma rede intra-chip (NoC) e possibilita que tarefas possam ser alocadas estática e/ou dinamicamente no sistema. Com isso, várias estratégias de alocação distintas podem ser implementadas e avaliadas. HMPS deverá ser um ponto de partida para vários trabalhos, contribuindo para a pesquisa na área de MPSoCs.

Este documento apresenta a proposta e a implementação da plataforma HMPS. Para a infra-estrutura de hardware utilizou-se a NoC HERMES, desenvolvida pelo grupo de pesquisa GAPH, e o processador de código aberto Plasma disponível no site *OpenCores*. Módulos de hardware foram desenvolvidos e alterações no código do Plasma foram realizadas, visando conectar o processador à NoC e realizar a alocação de tarefas na memória do processador. Para a infra-estrutura de software, foi desenvolvido um microkernel multitarefa que executa em cada processador escravo e a aplicação de alocação de tarefas que executa no processador mestre. São exploradas duas estratégias de alocação de tarefas: uma estática e uma dinâmica.

Palavras Chave: MPSoCs, multiprocessamento, multitarefa, NoC, alocação de tarefas.

Abstract

MPSoCs (*Multiprocessor System on Chip*) are an increasingly important trend in the design of embedded systems implemented as SoCs, since they facilitate the fulfillment of application requirements. This is because several processors, dedicated hardware modules memory blocks and interconnection media compose the architecture of such systems, making available a higher processing power when compared to equivalent monoprocessor systems. However, strategies to obtain the potential processing capacity offered by such architectures need to be better understood and explored. To enable evaluating such strategies, it is necessary to have available a hardware/software infrastructure capable to manage MPSoC tasks execution. From such an infrastructure, it should be possible, for example, to dynamically map tasks on processors, balancing the MPSoC workload through dynamic task allocation strategies.

The state of the art in the available literature explores static and dynamic task allocation strategies on MPSoCs and evaluates their viability and efficiency. Nonetheless, the need to create the hardware/software infrastructure to enable strategy exploration constitutes a bottleneck for the advance of this technology. Additionally, most works employ quite abstract models to evaluate the proposed approaches, reducing the reliability of the reported results.

The main contribution of the present work is the proposition and implementation of an MPSoC platform called HMPS (*Hermes Multiprocessor System*). HMPS offers a hardware/software infrastructure enabling to manage task execution in MPSoC systems. The HMPS platform is based on homogeneous multiprocessing, and has a master-slave architecture. The platform employs an network on chip (NoC) as interconnection media and allows that tasks be allocated either statically or dynamically. The platform allows several distinct allocation strategies to be implemented and evaluated at a quite detailed level of abstraction. HMPS is expected to be the starting point for several future works, contributing to the research on MPSoCs.

This document presents the proposition and implementation of the HMPS platform. For the hardware infrastructure, the platform employs the open source processor Plasma and the HERMES NoC, implemented by the GAPH Research Group. Some specific hardware modules were developed for the platform and some changes were made in the Plasma processor, with the goal of connecting the processor to the NoC and supporting task allocation at each processor. As for the software infrastructure, HMPS provides a multitasking microkernel executing in each slave processor and the task allocation application running on the master processor. Two task allocation strategies are available in HMPS: one static and one dynamic.

Keywords: MPSoC, multiprocessing, multitask, NoC, task allocation.

Agradecimentos

Não poderia ter chegado aqui sem que eu tivesse oportunidades: oportunidade de entrar no programa de mestrado e oportunidade de cruzar com pessoas que me ajudaram a crescer. Dessa forma, meu primeiro agradecimento é Àquele que criou essas oportunidades e que também me deu saúde para vivenciá-las: Deus.

Dentre as pessoas que me ajudaram a crescer, a primeira que eu devo citar é meu orientador, Moraes. Foi um pai quando precisei: me acolheu e me inseriu no seu seleto grupo de pesquisa. Um excelente orientador, sempre disponível a ajudar em qualquer momento. Fora isso, também sempre muito brincalhão, descontraindo “os ares”. Obrigada por tudo, Moraes!

De coração, agradeço também ao Prof. Avelino Zorzo, que era coordenador do Programa de Pós-Graduação no momento em que tomei a decisão de trocar de orientador. Teve uma atitude compreensiva e muito acolhedora, me apoiando e facilitando esse processo. Jamais vou esquecer disso!

Quero agradecer ao Prof. Ney Calazans que revisou o resumo e abstract deste trabalho. Agradecer também pelas conversas bem humoradas durante reuniões com meu orientador.

Tenho muito a agradecer ao Ismael, que fez parte deste trabalho como bolsista de IC e que me ajudou muito. Foram vários aprendizados em todas as vezes que trabalhamos juntos implementando e depurando o sistema. Além disso, descobri uma pessoa que, como eu, gosta muito de bala. Comprávamos muitas balas para comer enquanto trabalhávamos, contribuindo para uma engorda. Obrigada Ismael pela ajuda! (mas pelos kilos a mais você me paga!)

Contei também com a ajuda do Möller no início da dissertação, quando não tinha idéia de como implementar o que tinha de ser implementado; da Aline com suas dicas de Word e conversas sobre a vida, a profissão, o futuro, os namorados...

Quero agradecer à minha família, meus pais, Eduardo e Edite, e meus irmãos, Laerson, Darlon e Cristhiele que, mesmo estando longe, sempre estiveram muito presentes na minha vida e sempre me deram apoio e suporte. Obrigada por tudo! Vocês são os bens mais preciosos que eu tenho!

Agradeço também aos meus amigos, que sempre preenchem o coração... Gabriel, Márcio, Edson, Ewerson, Taís, Ost, David, Sérgio, Möller, Aline, Luciane, Silvano, Melissa, enfim... a todos que estão comigo de vez em quando e que agora não lembro o nome.

Por fim, quero agradecer ao meu namorado, Paulo, pelos bons fins de semana que vivemos durante esse período, fazendo com que a minha semana se tornasse mais alegre. Obrigada pelo amor, carinho e preocupação que você tem por mim.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ACG	*Architecture Characterization Graph*
APU	*Auxiliar Processor Unit*
BIU	*Bus Interface Unit*
BRAM	*Block RAM*
CAD	*Computer-Aided Design*
CP	*Constraint Programming*
CPU	*Cental Processing Unit*
DCT	*Discrete Cosine Transform*
DMA	*Direct Memory Access*
DSP	*Digital Signal Processor*
EDF	*Earliest Deadline First*
FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*
FPU	*Floating Point Unit*
FSB	*Front Side Bus*
GPRs	*General Purpose Registers*
IDCT	*Inverse Discrete Cosine Transform*
ILP	*Integer Linear Programming*
IQUANT	*Inverse Quantization Algorithm*
ISA	*Instruction Set Architecture*
ISO	*International Organization of Standardization*
IVLC	*Inverse Variable Length Coding*
LUT	*LookUp Table*
MDU	*Multiple/Divide Unit*
MIPS	*Microprocessor without Interlocked Pipeline Stages*
MMU	*Management Memory Unit*
MPSoC	*Multi Processor System On Chip*
MPU	*Microprocessor Units*
NI	*Network Interface*
NoC	*Network On Chip*
NORMA	*No Remote Memory Access*
OCP	*Open Core Protocol*
OSI	*Open System Interconnection*
PC	*Personal Computer*
PDA	*Personal Digital Assistant*
PE	*Processing Element*
QED	*Quantum Effect Devices*
QOD	*Quantidade, Ordem, Dependência*
RTEMS	*Real-Time Executive for Multiprocessors Systems*
SRAM	*Static Random Access Memory*
TCB	*Task Control Block*
VLSI	*Very Large-Scale Integration*

1 INTRODUÇÃO

Sistemas multiprocessados em *chip* (MPSoC, do inglês *Multiprocessor System-on-Chip*) são um dos recursos chave (uma das aplicações-chave) da tecnologia VLSI (do inglês *Very Large-Scale Integration*) e uma tendência no projeto de sistemas embarcados [TAN06]. Um MPSoC consiste de uma arquitetura composta por recursos heterogêneos, incluindo múltiplos processadores embarcados, módulos de hardware dedicados, memórias e um meio de interconexão [WOL04].

Um número crescente de aplicações embarcadas possui requisitos estritos de desempenho, consumo de potência e custo. Exemplos dessas aplicações são encontrados nas áreas de telecomunicações, multimídia, redes e entretenimento. O atendimento de cada um desses requisitos separadamente é uma tarefa difícil e a combinação deles constitui um problema extremamente desafiador. MPSoCs possibilitam a adaptação da arquitetura do sistema de forma a melhor atender os requisitos da aplicação: a atribuição de poder computacional onde é necessário possibilita atender a requisitos de desempenho; a remoção de elementos desnecessários reduz o custo e o consumo de potência. Isso mostra que MPSoCs não são apenas *chips* multiprocessadores, os quais apenas utilizam da vantagem da alta densidade de transistores para colocar mais processadores em um único *chip* e sem visar as necessidades de aplicações específicas. Ao contrário, MPSoCs são arquiteturas personalizadas que fazem um balanço entre as restrições da tecnologia VLSI com os requisitos da aplicação [JER05].

MPSoCs, além de núcleos, contêm um meio de interconexão. Existem diferentes meios de interconexão em MPSoCs: conexão ponto a ponto, barramento único, barramento hierárquico e redes intra-chip. O meio de interconexão mais tradicional é o barramento. Contudo, não são escaláveis além de algumas dezenas de núcleos [GUE00]. Uma moderna forma de interconexão de MPSoCs são as redes intra-chips, também denominadas NoCs (do inglês, *Network on Chip*) [WIN01][SGR01][BEN02][KUM02][MOR04]. Essas redes vêm sendo pesquisadas com o intuito de resolver problemas relacionados à comunicação de dados entre os componentes do sistema. Dentre esses problemas, encontra-se a baixa escalabilidade e a ausência de paralelismo, uma vez que a interconexão através de um barramento compartilhado permite que apenas uma comunicação possa ser estabelecida em um dado momento.

A motivação do presente trabalho reside na importância das arquiteturas MPSoCs para sistemas embarcados atuais e na necessidade de desenvolver arquiteturas escaláveis, uma vez que a tendência de MPSoCs é que eles sejam um “mar de processadores”, isto é, uma arquitetura constituída por uma grande quantidade de núcleos interconectados por uma NoC [HEN03]. Além disso, existe a necessidade do desenvolvimento de camadas de software básico para processadores embarcados que compõem o MPSoC, com a finalidade de gerenciar a execução das tarefas no sistema. Dessa forma, tarefas podem ser enviadas em tempo de execução para processadores que serão capazes de receber, armazenar, inicializar e executar as mesmas. Essa abordagem possibilita uma flexibilidade na distribuição da carga nos processadores do MPSoC, de forma que os

benefícios oferecidos por estas (e.g, paralelismo) possam ser melhor aproveitados.

Dado o contexto descrito nos parágrafos anteriores, este trabalho visa o desenvolvimento de uma infra-estrutura multiprocessada e multitarefa, que permita explorar a alocação dinâmica de tarefas. Os núcleos desta infra-estrutura são interconectados por uma NoC. Alocação dinâmica é o envio de novas tarefas a nodos escravos durante a execução do sistema. Esse envio pode ser mediante decisão de um nodo mestre ou a partir da requisição de novas tarefas por nodos escravos. Todos os nodos que compõem a infra-estrutura são todos processadores idênticos, constituindo assim uma plataforma com processamento homogêneo.

Dentro do contexto de alocação dinâmica está a *migração de tarefas*. Este mecanismo suspende a execução de uma tarefa em um nodo *x* e a envia para um nodo *y*, onde sua execução é retomada. A migração de tarefas é um tema bastante explorado na área de Sistemas Paralelos e Distribuídos. A aplicação desta abordagem em MPSoCs [BER06][OZT06][NGO06][NOL05] está sendo estudada de forma a empregar seus benefícios na área de sistemas embarcados. A migração de tarefas pode ser justificável para otimizar o desempenho do sistema, considerando critérios como, por exemplo, carga de trabalho nos processadores e a carga da comunicação nos enlaces da rede.

1.1 Arquitetura Conceitual

A Figura 1 apresenta a arquitetura conceitual desta dissertação: um MPSoC com processamento heterogêneo interligado por uma NoC, onde um nodo mestre deverá alocar as tarefas no MPSoC (nodos escravos). Um nodo escravo poderá ser um processador embarcado de propósito geral (CPU) ou um módulo de hardware reconfigurável [MOL05]. Dessa forma, as tarefas podem ser tanto de software (executarão em CPUs) ou de hardware (um *bistream* de reconfiguração parcial). Cada nodo escravo componente do sistema deverá ser capaz de receber uma tarefa, armazenar todas as informações referentes a esta tarefa em memória e executá-la. Para os processadores, faz-se necessário o desenvolvimento de uma camada de software, chamada *microkernel* (µkernel), que forneça os serviços básicos de um sistema operacional para o controle da execução das tarefas no ambiente multiprocessado e multitarefa. Para os módulos reconfiguráveis, faz-se necessária a inclusão de um controlador de configuração [CAR04] e infra-estrutura para conexão destes módulos.

O escopo da presente dissertação é o desenvolvimento da infra-estrutura de hardware e de software para a inclusão dos processadores em uma NoC. Desta forma, ao final deste trabalho será obtido um MPSoC com processamento homogêneo, ou seja, todos os núcleos são processadores de propósito geral idênticos. A inclusão de módulos de hardware reconfigurável ao sistema é tema de trabalhos futuros, já havendo diversos trabalhos desenvolvidos no grupo de pesquisa no qual a presente dissertação insere-se: [BRI04][CAR04][MOL05].

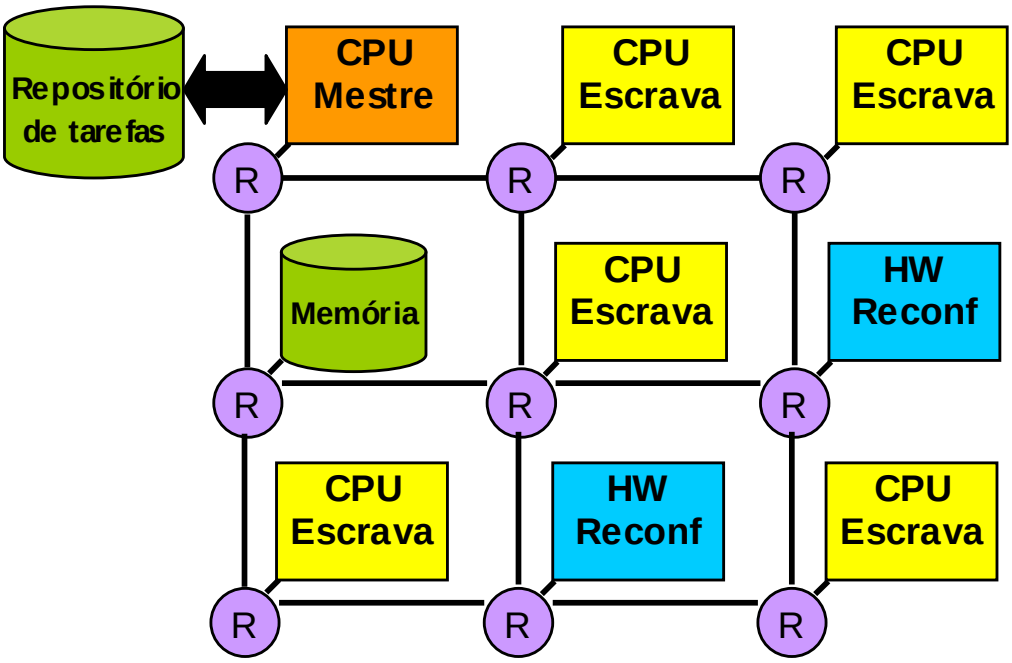


Figura 1 – Arquitetura conceitual da proposta. Um nodo mestre envia tarefas que estão inicialmente no repositório para CPUs escravas ou para módulos de hardware reconfigurável.

1.2 Objetivos

Os objetivos são divididos em duas classes: objetivos estratégicos e objetivos específicos.

Os objetivos estratégicos compreendem:

- Domínio da tecnologia de sistemas multiprocessadores em *chip* (MPSoC);
- Domínio da tecnologia de redes intra-chip (NoC).

Os objetivos específicos compreendem:

- Criação de um MPSoC homogêneo, composto por uma infra-estrutura de hardware e uma infra-estrutura de software;
  - Seleção de processador alvo a ser conectado na NoC HERMES;
  - Conexão do processador selecionado à HERMES, desenvolvendo-se a interface com a NoC (NI – Network Interface*)* e as camadas básicas de comunicação *(drivers)*;
  - Desenvolvimento de um *microkernel* multitarefa para ser executado em cada processador selecionado;
- Dotar o *microkernel* com serviços para receber e executar novas tarefas, serviço este denominado no contexto do presente trabalho, de *alocação de tarefas*.

1.3 Organização do documento

O Capítulo 2 apresenta o estado da arte em alocação de tarefas sobre MPSoCs. São abordados dois tipos de alocação de tarefas: alocação estática e alocação dinâmica. Ao final deste Capítulo, o trabalho é posicionado frente ao estado da arte.

O Capítulo 3 apresenta os conceitos básicos empregados no desenvolvimento deste trabalho. Estes conceitos são provenientes de várias áreas, caracterizando um trabalho multidisciplinar. São eles: redes intra-chip, processadores embarcados e sistemas operacionais.

O Capítulo 4 descreve a infra-estrutura de hardware utilizada. Fazem parte desta infra-estrutura a rede HERMES e o processador Plasma. As características de cada um são apresentadas. Ainda, são descritas as mudanças realizadas no processador e os mecanismos que foram implementados e agregados ao sistema de forma a atingir os objetivos almejados. A infra-estrutura de hardware gerada nesta etapa é a primeira contribuição deste trabalho.

O Capítulo 5 apresenta a infra-estrutura de software desenvolvida, constituindo a segunda contribuição deste trabalho. Esta infra-estrutura é composta pelo *microkernel* multitarefa que executa em cada processador escravo do MPSoC. São descritas as estruturas de dados utilizadas, a inicialização do sistema e os serviços implementados: tratamento de interrupções, escalonamento, comunicação entre tarefas, chamadas de sistema e *drivers* de comunicação.

O Capítulo 6 descreve as estratégias de alocação de tarefas empregadas neste trabalho. São utilizados os dois tipos de alocação: estática e dinâmica. É apresentado neste Capítulo o processador mestre da plataforma, que contém as estratégias de alocação, e uma interface com o repositório de tarefas. Este repositório contém o código objeto das tarefas que são transferidas para os processadores escravos.

O Capítulo 7 apresenta a validação do sistema: serviços do *microkernel* como escalonamento, comunicação entre tarefas; alocação estática e dinâmica de tarefas. Este Capítulo apresenta também a avaliação da plataforma mediante a execução de aplicações constituídas por tarefas cooperantes.

O Capítulo 8 encerra este documento com conclusões e trabalhos futuros.

2 ESTADO DA ARTE

A alocação de tarefas é um problema clássico em sistemas multiprocessados e recentemente têm recebido atenção de vários pesquisadores no âmbito de MPSoCs. Na literatura, a alocação de tarefas é também referenciada como mapeamento de tarefas. Alguns trabalhos utilizam alocação estática [RUG06][VIR04][HU04][MAR05], a qual é desenvolvida em tempo de projeto e não muda ao longo da execução do sistema. Outros utilizam a alocação dinâmica [BER06][OZT06][STR06][NGO06][NOL05], na qual emprega-se uma alocação inicial com migração de tarefas e/ou dados em tempo de execução.

2.1 Alocação Estática

A alocação estática é considerada atraente (do inglês, *attractive*) para sistemas nos quais a solução é computada uma única vez e é aplicada durante a vida inteira desses sistemas. É o caso de, por exemplo, sistemas dedicados a um algoritmo específico para processamento de sinal (e.g. filtro de imagens) e aplicações multimídia.

2.1.1 Ruggiero et al.

Ruggiero et al. [RUG06] apresentam um ambiente para alocação de tarefas, onde uma plataforma MPSoC é utilizada. O problema é dividido em dois subproblemas: alocação de tarefas em processadores e escalonamento de tarefas. O subproblema da alocação é resolvido com Programação Linear Inteira (ILP, do inglês *Integer Linear Programming*) e é assim definido: “dado um grafo de tarefas, o problema consiste em alocar *n* tarefas em *m* processadores, de forma que a quantidade total de memória alocada, para cada processador, não exceda o tamanho de cada memória local”. O subproblema de escalonamento é resolvido através de Programação por Restrições (CP, do inglês *Constraint Programming*). As soluções para ambos problemas interagem convergindo a uma solução ótima. Os grafos de tarefas são modelados como um *pipeline*, abordagem utilizada em certas aplicações multimídia, onde as tarefas ocorrem seqüencialmente no tempo. A arquitetura alvo utiliza memória distribuída com troca de mensagens. Os processadores são todos iguais (ARM7) interligados por barramento. Cada processador tem uma memória local e pode acessar memórias remotas. O sistema operacional utilizado é o RTEMS [RTE06]. Um estudo de caso demonstra uma maior eficiência computacional da abordagem que utiliza a interação de ILP com CP em relação às abordagens tradicionais (puramente ILP ou CP).

2.1.2 Virk e Madsen

Virk e Madsen [VIR04] apresentam um ambiente para modelagem em nível de sistema de plataformas MPSoCs heterogêneas, cujo meio de interconexão de núcleos é uma NoC. Este ambiente tem por objetivo desenvolver modelos para uma ampla classe de projetos e possui uma

biblioteca de modelos de componentes abstratos contendo características físicas, de desempenho e de potência. É necessária a modelagem do sistema e de todas as inter-relações entre os processadores, processos, interfaces físicas e interconexões.

Uma aplicação é composta de um conjunto de tarefas, cada uma das quais podendo ser decomposta em uma seqüência de segmentos. Cada tarefa possui parâmetros, como deadline, período, tempo de troca de contexto, entre outros.

A plataforma multiprocessada é modelada como uma coleção de PEs (do inglês, *Processing Elements*) e dispositivos interconectados por um canal de comunicação. Cada PE é modelado em termos dos serviços do SO fornecidos para as tarefas da aplicação. Três serviços são modelados: escalonamento, alocação de recursos e sincronização.

A NoC é modelada como um processador de comunicação e um evento de comunicação é modelado como uma tarefa de mensagem, que deve ser sincronizada, ter seus recursos alocados e escalonada. Cada implementação de NoC possui uma base de dados contendo informações de todos os seus recursos. O alocador da NoC tenta minimizar os conflitos de recursos. O escalonador da NoC executa as tarefas de mensagens de acordo com o requisito do serviço da rede. Além disso, ele tenta minimizar a ocupação dos recursos.

Para a validação do ambiente foi modelado um dispositivo multimídia portátil. A abstração do código da aplicação, bem como a extração dos parâmetros do grafo de tarefas e mapeamento das tarefas na NoC foram realizados manualmente. São utilizados 4 processadores heterogêneos cada um com sua memória local e todos interconectados por uma NoC com topologia toro.

2.1.3 Hu e Marculescu

Hu e Marculescu [HU04] apresentam um algoritmo para escalonamento de tarefas e comunicação em NoCs com núcleos heterogêneos, sob restrições de tempo real, minimizando o consumo de energia. O algoritmo automaticamente atribui tarefas a diferentes PEs e então escalona os mesmos. Os escalonamentos da comunicação e computação acontecem em paralelo.

Uma aplicação é descrita como um CTG (*Communication Task Graph*), onde as tarefas podem ter dependência de controle (uma não pode iniciar antes que outra termine) ou de dados (comunicação inter-tarefas durante execução). Uma arquitetura é descrita como um ACG (*Architecture Characterization Graph*), indicando nodos e conexões entre eles.

Dados os CTG e ACG, o problema consiste em decidir como escalonar as tarefas e as comunicações na arquitetura alvo. A solução para este problema tem impacto no consumo de energia do sistema, pois: (i) devido à heterogeneidade da arquitetura, atribuir a mesma tarefa para diferentes PEs pode levar a diferentes consumos de energia na computação; (ii) para diferentes alocações de tarefas, o volume de comunicação inter-tarefas e o caminho de roteamento pode mudar significativamente, causando diferentes consumos de energia na comunicação.

A arquitetura alvo é uma NoC com topologia malha 2D. Cada núcleo é composto por um PE e um roteador. Os PEs são heterogêneos, pois os mesmos podem ser processadores diferentes ou

módulos de hardware específico.

O escalonamento é não-preemptivo, estático e tenta minimizar o consumo de energia da aplicação. Para isso é preciso: (i) determinar quais tarefas executarão em quais PEs; (ii) em PEs que executam mais de uma tarefa, determinar qual é executada primeiro; (iii) determinar o tempo para todas as comunicações. Experimentos mostram que o algoritmo proposto, comparado com o algoritmo de escalonamento EDF (*Earliest Deadline First*), consome 55% e 39% menos de energia (duas categorias de benchmarks).

2.1.4 Marcon et al.

Marcon et al. [MAR05] propõem e avaliam um conjunto de modelos de computação voltados para a solução do problema de mapeamento de núcleos de propriedade intelectual sobre uma infra-estrutura de comunicação. Três modelos são propostos (CDM, CDCM e ECWM) e comparados, entre si e com três outros disponíveis na literatura (CWM, CTM e ACPM). Embora os modelos sejam genéricos, os estudos de caso restringem-se a infra-estruturas de comunicação do tipo rede intra-chip. Dada a diversidade de modelos de mapeamento, é proposto o metamodelo Quantidade, Ordem, Dependência (QOD), que relaciona modelos de mapeamento, usando os critérios expressos na denominação QOD. Considerando o alto grau de abstração dos modelos empregados, julga-se necessário prover uma conexão com níveis inferiores da hierarquia de projeto. Neste sentido, são propostos modelos de consumo de energia e tempo de comunicação para redes intra-chip. Visando demonstrar a validade dos modelos propostos, foram desenvolvidos métodos de uso destes na solução do problema de mapeamento. Estes métodos incluem algoritmos de mapeamento, estimativas de tempo de execução, consumo de energia e caminhos críticos em infra-estruturas de comunicação. É também proposto um *framework* denominado CAFES, que integra os métodos desenvolvidos e os modelos de mapeamento em algoritmos computacionais. É criado ainda um método que habilita a estimativa de consumo de energia para infra-estruturas de comunicação e sua implementação como uma ferramenta computacional.

2.2 Alocação Dinâmica

A alocação estática de tarefas não condiz com a realidade atual dos MPSoCs, onde estes são projetados para suportar diferentes aplicações e inclusive novas aplicações após o sistema ter sido fabricado. Como exemplo, citam-se telefones celulares 3G, que suportam inúmeros perfis de aplicações, e sistemas *set-top box* para televisão digital, com padrões para tratamento de imagem em constante evolução.

A alocação dinâmica de tarefas considera uma alocação inicial, que é ajustada ao longo da execução da aplicação. Esse ajuste é efetuado mediante o envio de novas tarefas a PEs. Ainda, a alocação dinâmica habilita a execução de ocorrer *migração de tarefas*. Este mecanismo suspende a execução de uma tarefa em um nodo *x* e envia esta tarefa para um nodo *y*, onde sua execução é retomada. A migração de tarefas têm sido largamente empregada na área de Sistemas Paralelos e

Distribuídos, de forma a melhorar o desempenho geral dos sistemas, distribuindo a carga de trabalho entre os PEs.

Pesquisas recentes têm explorado a migração de tarefas no contexto de MPSoCs com vários propósitos, como por exemplo: *(i)* diminuir o volume de comunicação entre as tarefas e conseqüentemente o consumo de energia total do sistema; *(ii)* balancear a carga de trabalho entre processadores, melhorando o desempenho do sistema; *(iii)* aumentar a tolerância a falhas de um sistema. Assim, os benefícios da migração de tarefas podem ser utilizados em MPSoCs de forma a melhor atender os requisitos de aplicações embarcadas.

2.2.1 Bertozzi et al.

A viabilidade da migração de tarefas em MPSoCs é estudada por Bertozzi et al. [BER06], onde é proposta uma infra-estrutura de software para gerenciamento de tarefas. O mecanismo de migração de tarefas proposto é baseado em *checkpoints* no código cuja estratégia é definida pelo usuário.

A arquitetura utilizada é um MPSoC composto por processadores ARM7 interligados por barramento, cada um com uma memória local. Uma cópia do sistema operacional μClinux executa em cada processador. Uma memória compartilhada é usada para comunicação entre processos e para manter a informação do estado global dos processos executando em cada processador (tabela de migração de processos). O sistema é organizado com um processador mestre e um número arbitrário de escravos. O mestre desempenha o controle de admissão de tarefas e a alocação inicial de tarefas nos processadores escravos.

Foi desenvolvido um middleware no nível de usuário sobre o μClinux, composto por um *daemon* e uma biblioteca de mensagens, fornecendo suporte à migração pelo uso de pontos de migração através de *checkpoint*s no código. O programador especifica explicitamente o estado mínimo da tarefa a ser salvo em cada ponto de migração. O middleware permite salvar o estado da tarefa e restaurá-lo no processador destino.

Um *daemon* mestre (que executa no processador mestre) implementa a estratégia de balanceamento de carga e controla os eventos de migração de forma centralizada. Um *daemon* escravo (que executa em cada processador escravo) é responsável por criar processos no processador local quando lhe for requisitado pelo mestre. Quando uma tarefa deve ser migrada, o *daemon* mestre ativa a entrada da tarefa selecionada na tabela de migração. Quando a tarefa atinge o ponto de migração, ela confirma a requisição de migração, desativando sua entrada na tabela de migração. A seguir, o *daemon* mestre requisita a criação de um novo processo no processador destino. As tarefas encontram-se replicadas em cada memória privada, evitando assim, a transferência de código. A Figura 2 expressa *T_migration*, o tempo gasto entre a chamada de um dado *checkpoint* e a restauração da tarefa no processador destino. *T_migration* é composto pelo: (*i*) tempo usado pela API de *checkpoint*s para verificar a requisição de migração, salvar o contexto e sair (*T_shutdown*); (*ii*) tempo gasto esperando o *daemon* ser ativado (*T_activation*); (*iii*) tempo necessário para restaurar a tarefa migrada (*T_reboot*).

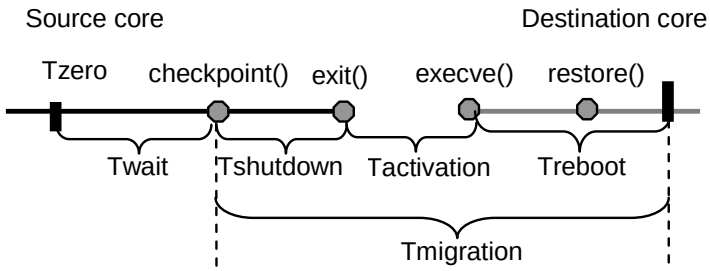


Figura 2 - Linha do tempo para o mecanismo de migração de [BER06].

Para medir o desempenho do sistema e caracterizar os custos de migração, diversos testes foram feitos. O *overhead* dessa abordagem foi caracterizado via simulação funcional e sua viabilidade no contexto de MPSoCs foi demonstrada.

2.2.2 Ozturt et al.

Ozturt et al. [OZT06] propõem um esquema de migração seletiva, o qual decide por migrar uma tarefa (código) ou um dado, de maneira a satisfazer um requisito de comunicação. A escolha entre as duas opções é feita em tempo de execução, baseada em estatísticas coletadas em tempo de compilação. O objetivo é reduzir o consumo de energia de um MPSoC durante a comunicação entre processadores.

A Figura 3 ilustra as duas opções de migração. A migração de dado é representada na Figura 3(a), onde o processador P_i envia um dado S ao processador P_j, o qual contém o procedimento Q que executa sobre S (Y é o dado de saída). A migração de tarefa é representada na Figura 3(b), onde P_j envia a tarefa para P_i e P_i devolve o resultado para P_j.

A abordagem de migração seletiva é composta por três componentes: *Profiling*, *Code Annotation* e *Selective Code/Data Migration*. O primeiro e o segundo são realizados em tempo de compilação. A técnica de *Profiling* coleta estatísticas de custos de energia para a migração de código e dado. Esses custos são calculados para cada unidade de migração de código e para cada unidade de migração de dado. *Code Annotation* indica a seqüência de comunicações envolvidas para um determinado código. O processo de *migração de dados ou código* é realizado em tempo de execução e baseia-se nos custos de energia calculados estaticamente. A abordagem proposta considera requisitos de comunicação e tenta tomar decisões globalmente ótimas (ou seja, considerando múltiplas comunicações e não apenas a comunicação corrente).

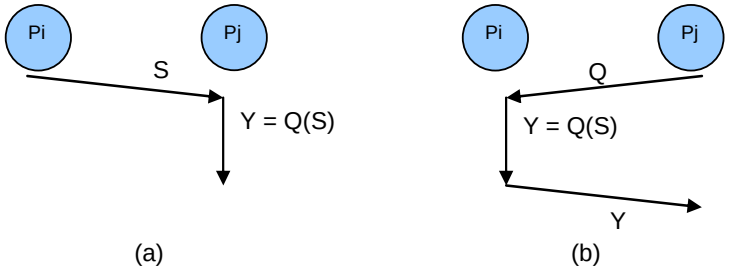


Figura 3 - (a) Migração de dado; (b) Migração de tarefa (código).

A arquitetura MPSoC utilizada no trabalho é composta por 8 processadores interligados por um barramento. Cada processador possui uma memória local de 32KB de capacidade. A comunicação entre os processadores é realizada através de troca de mensagens. A migração seletiva foi implementada e comparada com dois esquemas alternativos: um que sempre migra dado e outro que sempre migra código. Os resultados coletados indicam que a migração seletiva reduz o consumo de energia e o tempo de execução.

2.2.3 Streichert et al.

Em [STR06] é proposto um algoritmo de particionamento HW/SW para redes reconfiguráveis, o qual otimiza a alocação dinâmica de tarefas de forma que defeitos na rede possam ser tratados e o tráfego seja minimizado. Uma vez que a alocação estática não é tolerante a falhas e a redundância de nodos e canais possui alto custo, esta abordagem separa a funcionalidade do HW físico e realoca tarefas de nodos defeituosos em nodos sem defeitos, tornando-se tolerante a falhas. Para determinar a alocação de tarefas em recursos da rede, é usado o algoritmo de particionamento HW/SW executado durante a operação do sistema.

Nodos da rede são FPGAs (dispositivos reconfiguráveis) e CPUs, podendo assim atribuir tarefas de HW e SW para os mesmos, respectivamente. Os nodos são conectados ponto-a-ponto. A rede é composta por nodos computacionais C, sensores S, atuadores A e canais de comunicação. Tarefas são distinguidas por tarefas de sensor, de controlador e atuador. Sensores produzem dados que são processados por controladores e atuadores consomem dados. São atributos de uma tarefa necessários para o particionamento: tempo de execução, deadline, LUTs (área), células de memória, tamanho da migração e taxas de transmissão de dados.

O particionamento é executado em duas fases: (i) reparo rápido: quando um nodo da rede falha, as tarefas que estavam neste nodo são alocadas em outro. Réplicas de tarefas se tornam tarefas principais (originais). Novas rotas para comunicação são estabelecidas; (ii) otimização: esta fase tenta encontrar uma alocação de tarefas de forma que o tráfego de dados seja minimizado. De forma a tolerar outros defeitos do nodo, tarefas replicadas precisam ser criadas e alocadas.

A abordagem foi implementada na seguinte infra-estrutura: rede de 4 placas de FPGAs reconfiguráveis, sistema operacional microC-OS II. Um gerenciador de tarefas decide onde alocar as tarefas. Foi implementado um modelo comportamental dessa rede com diferentes cenários, variando o número de tarefas e o número de nodos. Os resultados mostram que o tráfego é reduzido em pelo menos 20%.

2.2.4 Ngouanga et al.

Ngouanga et al. [NGO06] exploram a alocação dinâmica de tarefas em uma arquitetura reconfigurável de grão grande. O trabalho considera uma plataforma (APACHES) composta por PEs homogêneos conectados por uma NoC. O número de nodos é parametrizável. Nodos de processamento (ou escravos) executam tarefas e são agrupados em clusters; nodos mestres ou controladores de rede gerenciam clusters e desempenham a alocação de tarefas nos mesmos. Há um

nodo mestre para cada cluster.

A Figura 4 apresenta o funcionamento da plataforma APACHES. Os nodos `M` e `I/O` são respectivamente o controlador de rede e o controlador de `I/O`. O restante são nodos escravos. Apenas o nodo mestre (`M`) é responsável por desenvolver o mapemanto de tarefas. Na Figura 4 (a), uma aplicação está executando na plataforma. Quando uma nova aplicação deve ser mapeada na rede, o sistema realiza seqüencialmente as seguintes operações:

1. Uma nova alocação é computada pelo nodo mestre;
2. Se a nova alocação implica realocar algumas tarefas de aplicações que já estão executando, o mestre move essas tarefas para suas novas localizações. Para isso é enviada uma requisição para cada PE que vai ter uma tarefa realocada (Figura 4(b)).
3. Os códigos das tarefas que foram realocadas e das tarefas da nova aplicação são enviados para os PEs aos quais elas foram atribuídas (Figura 4(c) e Figura 4(d)).
4. O sistema retoma a execução de tarefas interrompidas e inicia o processamento da nova aplicação mapeada na rede (Figura 4(e)).

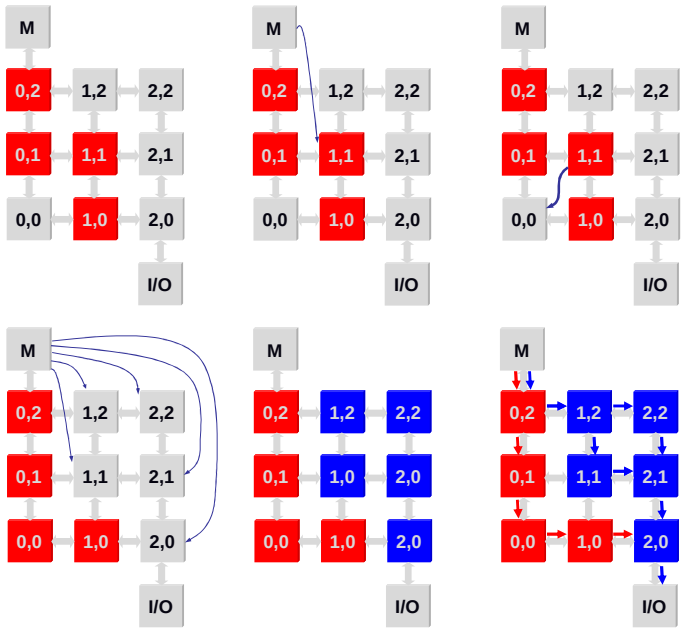


Figura 4 - Alocação de tarefas e roteamento de dados na plataforma APACHES.

Cada PE executa um *microkernel* multitarefa. São usados dois algoritmos de alocação de tarefas: *simulated annealing* e *force directed*. O primeiro é baseado na simulação de um fenômeno físico, o qual muda aleatoriamente a posição das tarefas na rede, estima custos e aceita ou rejeita mudanças de acordo com um critério de temperatura. O segundo calcula o posicionamento das tarefas, levando em conta a força resultante produzida pela atração entre tarefas comunicantes. Essa força de atração é proporcional ao volume de comunicação entre tarefas interligadas. Resultados mostram que a alocação dinâmica traz benefícios quando comparados com uma alocação estática ou aleatória.

2.2.5 Nollet et al.

Nollet et al. [NOL05] desenvolveram um esquema para o gerenciamento de recursos (computação e comunicação) em uma plataforma heterogênea multiprocessada cujos núcleos são interconectados por uma NoC. Estes núcleos são monotarefa e podem ser núcleos de hardware reconfigurável.

Um gerente (centralizado) é responsável por alocar a quantidade certa de recursos, de forma a acomodar tarefas de uma aplicação em tempo de execução. Isso basicamente significa: (1) alocar recursos de computação, (2) gerenciar recursos de comunicação na NoC e (3) tratar de várias questões relacionadas à migração de tarefas em tempo de execução na NoC.

A heurística de gerenciamento de recursos considera propriedades específicas do hardware reconfigurável para a alocação de tarefas. Quando ocorre um mapeamento falho ou quando os requisitos do usuário mudam, a heurística de gerenciamento de recursos pode utilizar a migração de tarefas. Como em [BER06], uma tarefa migra para outro PE apenas em determinados pontos no código (pontos de migração definidos pelo usuário). Um problema importante considerado na migração de tarefas é assegurar a consistência da comunicação durante o processo de migração. Esta questão é originada do fato de que, depois de receber uma requisição de migração, a quantidade de tempo e mensagens de entrada que uma tarefa requer até atingir o ponto de migração são desconhecidos. Isso significa que as tarefas produtoras de mensagens (i.e. pares comunicantes) devem manter o envio de mensagens até que a tarefa migrante informe que um ponto de migração foi atingido e que ela parou o consumo de mensagens. Contudo, pode haver mensagens armazenadas e não-processadas nos canais de comunicação entre tarefas produtoras e tarefas migrantes. Dessa forma, Nollet et al. apresentam dois mecanismos de migração de tarefas em NoCs, que asseguram a consistência de mensagens no processo de migração de tarefas.

A plataforma multiprocessada foi emulada interconectando um processador StrongARM (PE mestre) a um FPGA contendo PEs escravos. Um sistema operacional executa no PE mestre e contém a heurística de gerenciamento de recursos e mecanismo de migração de tarefas.

2.3 Posicionamento do Trabalho em Relação ao Estado da Arte

O objetivo deste trabalho é o desenvolvimento de uma plataforma multiprocessada e multitarefa em *chip* que permita a alocação dinâmica de tarefas sobre os processadores. Esta plataforma é um MPSoC interligado por uma NoC. Todos os nodos do MPSoC são processadores embarcados de propósito geral idênticos. Um nodo mestre aloca as tarefas nos nodos escravos e cada escravo componente do sistema é capaz de receber uma tarefa, armazenar todas as informações referentes a esta tarefa em memória e executá-la. Para tanto, foi implementado um *microkernel* que fornece os serviços básicos de um sistema operacional necessários para o controle da execução das tarefas no ambiente multiprocessado e multitarefa.

O nodo mestre contém as informações de alocação de todas as tarefas a serem executadas, relacionando um conjunto de tarefas por nodo escravo. Na inicialização do sistema, ele carrega as

tarefas de um repositório, enviando-as aos nodos escravos designados para executá-las. Como este envio de tarefas ocorre apenas na inicialização do sistema, denomina-se esta alocação como **alocação estática**. Esta alocação inicial pode ser realizada aleatoriamente, manualmente ou então utilizar um dos algoritmos de alocação estática apresentados aqui. O foco principal do trabalho é a **alocação dinâmica**, no qual o nodo mestre envia novas tarefas aos nodos escravos mediante requisição destes, durante a execução do sistema.

São contribuições deste trabalho: (i) definição e implementação de uma plataforma MPSoC, com processadores interconectados por uma NoC (contribuição de hardware); (ii) desenvolvimento de um *microkernel* para alocação estática e dinâmica de tarefas, permitindo um melhor uso dos processadores (contribuição de software).

A Tabela 1 resume as características dos trabalhos apresentados nesta Seção e posiciona o presente trabalho frente ao estado da arte. As características apresentadas na tabela incluem: processador utilizado, arquitetura de memória (distribuída ou centralizada), tipo de processamento (homogêneo ou heterogêneo), meio de interconexão, estratégia de alocação, sistema operacional e objetivo do trabalho.

Tabela 1 - Tabela comparativa dos trabalhos no estado da arte de mapeamento de tarefas.

Trabalho	Processador	Memória	Processamento	Interconexão	Alocação	SO	Objetivo
Ruggiero et al.	ARMv7	Distribuída	Homogêneo	Barramento	Estática	RTEMS	Alocar e escalonar tarefas em MPSoCs de forma a melhorar a eficiência da comunicação
Virk e Madsen	ND	Distribuída	Heterogêneo	NoC	Estática	RTOS abstrato	Formular um modelo para uma ampla classe de projetos
Hu e Marculescu	ND	ND	Heterogêneo	NoC	Estática	ND	Alocar e escalonar tarefas em MPSoCs heterogêneos, sob restrições de tempo real, minimizando o consumo de energia
Marcon et al.	Genérico	ND	Heterogêneo	NoC	Estática	ND	Mapear núcleos da NoC de forma a reduzir o consumo de energia e o tempo de execução.
Bertozzi et al.	ARM7	Distribuída	Homogêneo	Barramento	Dinâmica	uClinux	Provar a viabilidade da migração de tarefas em MPSoCs
Ozturk et al.	-	Distribuída	Homogêneo	Barramento	Dinâmica	ND	Reduzir consumo de energia durante a comunicação entre processadores
Streichert et al.	Placas FPGA e CPU RISC	ND	Heterogêneo	Ponto-a-ponto	Dinâmica	microC-OS II	Tratar defeitos em nodos/enlaces e minimizar o tráfego de dados nos enlaces
Ngouanga et al.	Plasma	Distribuída	Homogêneo	NoC	Dinâmica	*Microkernel* próprio	Explorar a alocação dinâmica de tarefas em uma matriz de PEs homogêneos
Nollet et al.	StrongARM e FPGA	Distribuída	Heterogêneo	NoC	Dinâmica	ND	Gerenciar recursos de comunicação e computação em uma NoC contendo núcleos de hardware reconfigurável
Woszezenki	Plasma	Distribuída	Homogêneo	NoC	Dinâmica	*Microkernel* próprio	Oferecer uma plataforma para explorar mecanismos de alocação dinâmica de tarefas.

3 COMPONENTES DO SISTEMA - CONCEITOS

Este trabalho inclui conceitos da área de Arquitetura de Computadores, Sistemas Digitais e Sistemas Operacionais, caracterizando assim sua constituição multidisciplinar. Este Capítulo apresenta os conceitos envolvidos neste trabalho: NoCs, Processadores Embarcados e Sistemas Operacionais.

3.1 Network on Chip - NoC

Uma maneira de minimizar os problemas oriundos da arquitetura de barramentos é através da utilização de redes intra-chip, também denominadas NoCs. Redes intra-chip utilizam conceitos originados da área de redes de computadores e de comunicação de dados, as quais empregam a organização da transferência de informação em camadas e protocolos [SGR01]. As principais características das NoCs que motivam o seu estudo são [BEN01][MOR04]: *(i)* confiabilidade e eficiência no gerenciamento de energia; *(ii)* escalabilidade da largura de banda em relação a arquiteturas de barramento *(iii)* reusabilidade; *(iv)* decisões de roteamento distribuídas.

A escalabilidade refere-se à capacidade de se interconectar núcleos adicionais de hardware, sem diminuir significativamente o desempenho global do sistema [OST03]. No caso do barramento, o acréscimo de novos núcleos implica aumentar o comprimento total deste, reduzindo o desempenho global do sistema. Em NoCs, o acréscimo de um elemento da rede leva ao aumento do número de canais de comunicação, melhorando de uma maneira global o seu desempenho [OST03].

Devido à multiplicidade de caminhos possíveis em NoCs, é possível explorar também o paralelismo na comunicação entre os núcleos, ou seja, pares de núcleos distintos podem realizar transações simultaneamente. Em barramento, apenas uma transação pode ser efetuada num dado momento, limitando assim a largura de banda utilizada por um módulo.

A reusabilidade de estruturas de comunicação é um conceito suportado tanto por estruturas de interligação por barramento como por rede. Esta característica diz respeito a se utilizar uma mesma estrutura de comunicação, sem modificá-la, em projetos distintos.

A interligação por barramento utiliza fios longos, ocasionando um maior consumo de energia que em NoCs. Estas, por sua vez, fazem uso eficiente do consumo de energia, uma vez que a comunicação é feita ponto a ponto, utilizando fios de tamanho reduzido, devido à pouca distância existente entre os módulos.

3.1.1 Nodos da rede

Uma rede é composta por nodos de processamento e nodos de chaveamento. A Figura 5 mostra uma rede em anel, uma topologia bastante simples e econômica. Cada nodo de chaveamento possui ligações para outros dois nodos de chaveamento vizinhos e para um nodo de processamento local [MEL05].

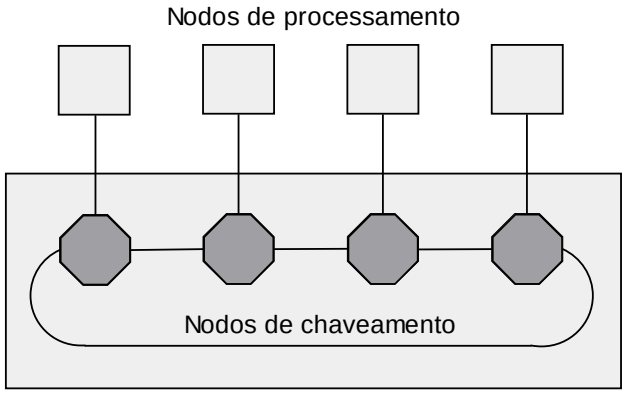


Figura 5 – Topologia em anel.

Os nodos de processamento (Figura 6(a)) contém um núcleo capaz de processar informação, podendo ser tanto um processador com memória local quanto um núcleo dedicado a uma tarefa específica, como por exemplo realizar a Transformada Discreta do Co-seno (DCT, do inglês, *Discrete Cosine Transform*) em um tratamento de imagem. Os nodos de chaveamento, denominados roteadores, (Figura 6(b)) realizam a transferência de mensagens entre os nodos de processamento. Em geral, eles possuem um núcleo de chaveamento (ou chave), uma lógica para roteamento e arbitragem (R&A) e portas de comunicação para outros nodos de chaveamento e, dependendo da topologia, para um nodo de processamento local. As portas de comunicação incluem canais de entrada e saída, os quais podem possuir, ou não, *buffer*s para o armazenamento temporário de informações [MEL05].

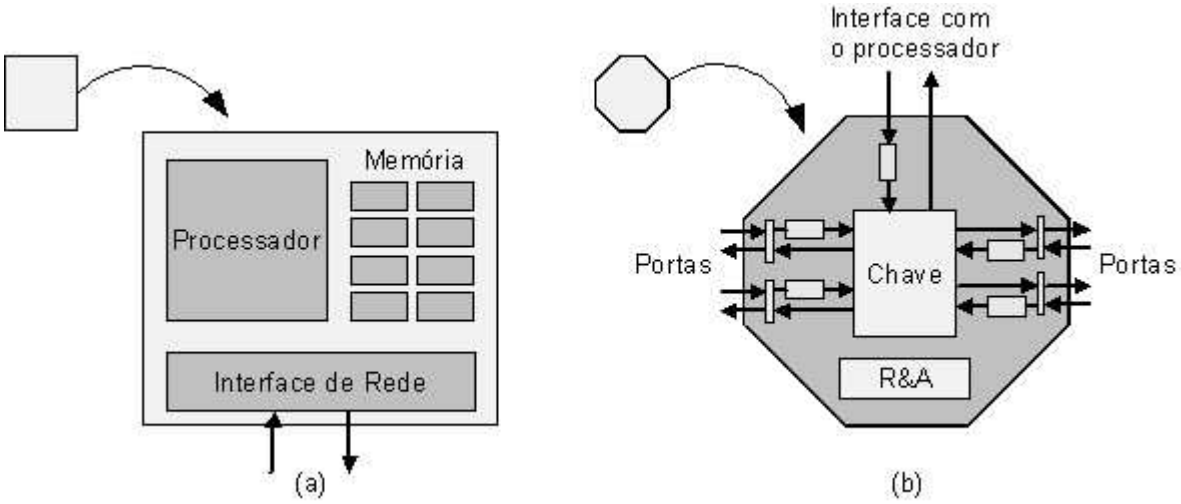


Figura 6 – Nodos: (a) de processamento; (b) de chaveamento.

3.1.2 Mensagens

As informações trocadas pelos nodos são organizadas sob a forma de mensagens, as quais, possuem, em geral, três partes: um cabeçalho (*header*), um corpo de dados (*payload*) e um terminador (*trailer*). No cabeçalho estão incluídas informações de roteamento e controle utilizadas pelos nodos de chaveamento para propagar a mensagem em direção ao nodo destino da comunicação. O terminador, por sua vez, inclui informações usadas para a detecção de erros e para

a sinalização do fim da mensagem.

Tipicamente, as mensagens são quebradas em pacotes para a transmissão. Um pacote é a menor informação que contém detalhes sobre o roteamento e seqüenciamento dos dados e mantém uma estrutura semelhante a de uma mensagem, com um cabeçalho, um corpo e um terminador. Um pacote é constituído por *flits*[1], cuja largura depende da largura física do canal.

3.1.3 Organização em camadas

As redes de interconexão para multiprocessadores empregam muitos conceitos empregados em redes de computadores. Um deles é a organização em camadas que encapsulam funções equivalentes àquelas definidas no modelo OSI (do inglês, *Open System Interconnection*), um padrão internacional de organização de redes de computadores proposto pela ISO (do inglês, *International Organization for Standardization*) [DAY83].

A arquitetura de uma rede que segue o modelo OSI é formada por níveis, interfaces e protocolos. Cada nível oferece um conjunto de serviços ao nível superior, usando funções realizadas no próprio nível e serviços disponíveis nos níveis inferiores. Os nodos de chaveamento de redes de interconexão que são estruturados em camadas hierárquicas implementam algumas das funções dos níveis inferiores (físico, enlace, rede) do modelo OSI, descritas abaixo [MEL05]:

- **Nível Físico**: realiza a transferência de dados em nível de bits através de um enlace.
- **Nível de enlace**: efetua a comunicação em nível de quadros (grupos de bits). Preocupa-se com o enquadramento dos dados e com a transferência desses quadros de forma confiável, realizando o tratamento de erros e o controle do fluxo de transferência de quadros.
- **Nível de rede**: faz a comunicação em nível de pacotes (grupos de quadros). Responsável pelo empacotamento das mensagens, roteamento dos pacotes entre a origem e o destino da mensagem, controle de congestionamento e contabilização de pacotes transferidos.

3.1.4 Topologias de redes de interconexão

Quanto à topologia, as redes de interconexão para multiprocessadores são definidas pela conexão de seus roteadores e podem ser agrupadas em duas classes principais: as redes diretas e as redes indiretas [ZEF03][MOR04].

Nas redes diretas, cada nodo de chaveamento possui um nodo de processamento associado, e esse par pode ser visto como um elemento único dentro da máquina, tipicamente referenciado pela palavra nodo, como mostra a Figura 7.

Topologias de redes diretas estritamente ortogonais mais utilizadas são a malha n-dimensional (Figura 8(a)), o toróide (Figura 8(b)) e o hipercubo (Figura 8(c)).

[1] *Flit* é a menor unidade de transferência de dados.

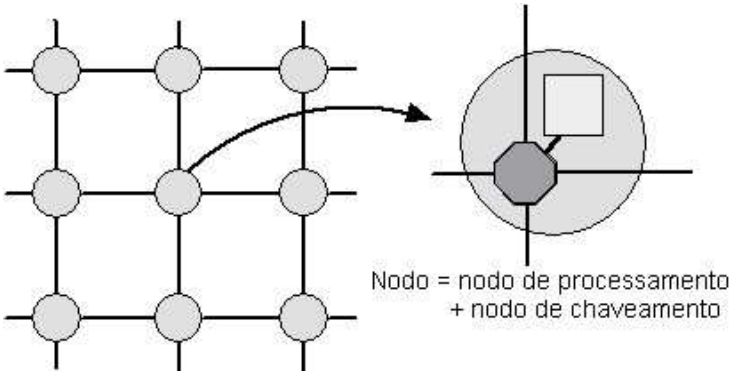


Figura 7 – Nodos de redes diretas.

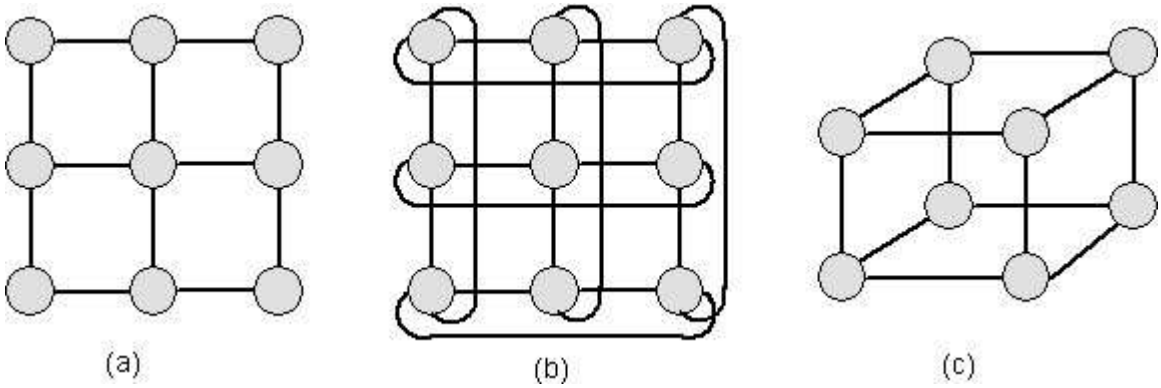


Figura 8 – (a) Malha 2D 3x3; (b) Toróide 2D 3x3; (c) Hipercubo 3D.

Nas redes indiretas os nodos de processamento possuem uma interface para uma rede de nodos de chaveamento baseados em chaves (roteadores). Cada roteador possui um conjunto de portas bidirecionais para ligações com outros roteadores e/ou nodos de processamento. Somente alguns roteadores possuem conexões para nodos de processamento e apenas esses podem servir de fonte ou destino de uma mensagem. A topologia dessa rede é definida pela estrutura de interconexão desses roteadores.

Duas topologias clássicas de redes indiretas se destacam: o crossbar e as redes multiestágio. Para conexão indireta de N nodos de processamento, o crossbar (Figura 9) é a topologia ideal, pois consiste de um único roteador NxN capaz de ligar qualquer entrada com qualquer saída. A desvantagem desta topologia é o custo, o qual cresce quadraticamente com o número de nodos de processamento.

3.2 Processadores Embarcados

MPSoCs são projetados tendo por base processadores de propósito geral (MPUs, do inglês, *Microprocessor Units*), processadores de sinais digitais (DSPs, do inglês, *Digital Signal Processor*) e co-processadores de propósito específico. Dessa forma, o entendimento de suas arquiteturas fornece um contexto para a escolha dos processadores adequados, em termos de desempenho, custo, potência e outros requisitos [JER04].

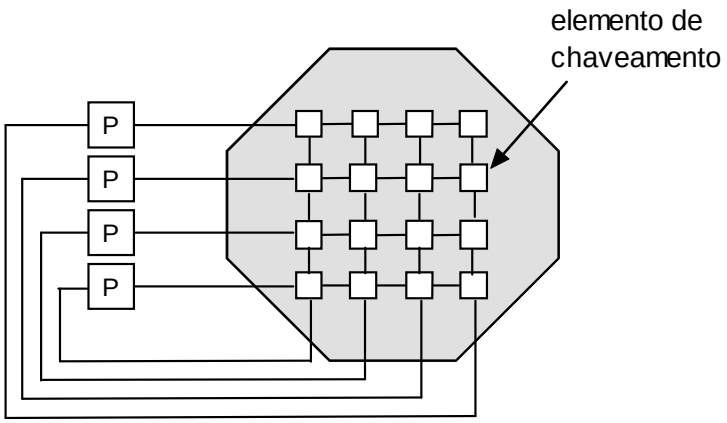


Figura 9 – Crossbar 4x4.

Processadores embarcados de propósito geral possuem aspectos diferentes de processadores de alto desempenho utilizados em PCs (do inglês, *Personal Computer*). Um PC deve suportar uma ampla gama de tipos de aplicação: ferramentas de produtividade (email, processadores de texto, apresentações), ferramentas de projeto (CAD, do inglês, *Computer-Aided Design*), jogos, multimídia e o sistema operacional em si. Por outro lado, os sistemas embarcados executam um conjunto específico de tarefas. Dessa forma, um processador de propósito geral embarcado, deve suportar diferentes aplicações embarcadas, cada uma das quais com um conjunto específico de tarefas [JER04].

A arquitetura do conjunto de instruções (ISA, do inglês, *Instruction-Set Architecture*) de um processador modela as funcionalidades deste e permite suporte eficiente para linguagens de alto nível, como C/C++. A micro-arquitetura (i.e., a organização do hardware) reflete a natureza genérica do ciclo busca/execução das instruções. Ela não é concebida para um algoritmo particular ou conjunto de algoritmos, como acontece em processadores de aplicação específica (e.g. processadores de rede e processadores gráficos). A arquitetura de processadores de aplicação específica possibilita explorar o paralelismo especificamente em nível de algoritmo (e.g. pacotes em processadores de rede ou retângulos em processadores gráficos) [JER04].

As próximas subseções apresentam alguns processadores embarcados com arquitetura RISC 32-bits. Neste trabalho, optou-se em utilizar esta classe de processadores por estes serem utilizados largamente em sistemas embarcados [TAN06], como por exemplo o processador ARM7.

3.2.1 ARM

Processadores ARM [ARM06] são utilizados, por exemplo, em telefones celulares e PDAs (do inglês, *Personal Digital Assistant*). Sua arquitetura tem evoluído e, atualmente, existem cinco famílias: ARM7, ARM9, ARM9E, ARM10 e ARM11. A sucessão de famílias representa mudanças na micro-arquitetura. Além disso, a ISA é freqüentemente melhorada, de forma a suportar novas funcionalidades. Essas melhorias funcionais incluem *Thumb*, um conjunto de instruções de 16 bits para código compacto; DSP, um conjunto de extensões aritméticas para aplicações DSP; e *Jazelle*, uma extensão para execução direta de *bytecode* Java. Outra melhoria da ISA é o suporte à unidade

de gerenciamento de memória (MMU, do inglês, *Memory Management Unit*).

3.2.2 IBM PowerPC 440

O processador PowerPC 440 da IBM [IBM06] foi concebido para uma variedade de aplicações embarcadas e integra um *pipeline* superescalar de sete estágios, com suporte à duas instruções por ciclo. Um diagrama de blocos do PowerPC 440 é apresentado na Figura 10. Esse processador possui:

- *caches* separadas para dados e para instruções;
- 32 registradores de propósito geral de 32 bits (GPRs, do inglês, *General Purpose Registers*);
- múltiplos *timers*, uma unidade de gerenciamento de memória (MMU, do inglês, *Management Memory Unit*), suporte à depuração e gerenciamento de potência;
- interface ao barramento *CoreConnect*;
- uma interface para *cache* L2 com até 256KB de SRAM, provendo capacidade adicional de memória;
- uma interface APU (do inglês, *Auxiliar Processor Unit*) que pode anexar uma unidade de ponto flutuante, um co-processador, um DSP ou outras funções para o *pipeline* de execução do processador.

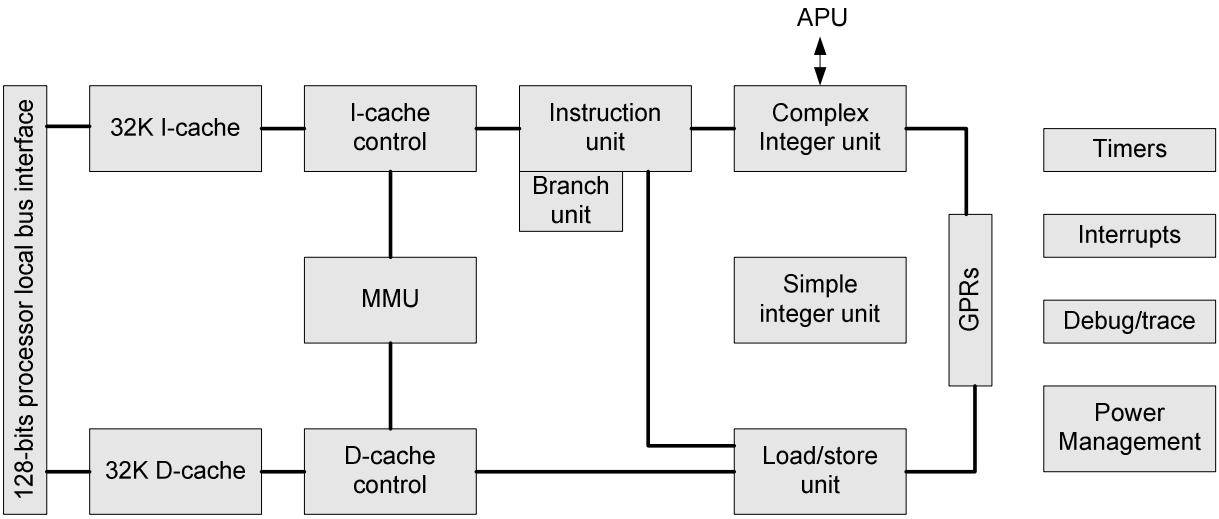


Figura 10 - Diagrama de blocos do processador PowerPC.

3.2.3 Processadores MIPS

As famílias de processadores MIPS, desenvolvidos pela MIPS Techonologies Inc. [MIP06], têm sido licenciadas e integradas nos projetos de várias companhias. A tecnologia oferecida pela MIPS tem sido usada para roteadores Cisco, modems à cabo e ADSL, *smartcards*, impressoras a *laser*, *set-top boxes*, *palmtops* e o *PlayStation2* da Sony Inc. A Tabela 2 apresenta as especificações das famílias dos processadores MIPS.

Tabela 2 – Características dos processadores MIPS.

Modelo	Freq. (MHZ)	Ano	Transistores (milhões)	Tamanho do *chip* (mm^2)	Pinos de I/O	Potência (W)	Dcache (KB)	Icache (KB)	Cache L2 (KB)
R2000	16.7	1985	0.11	--	--	--	32	64	Não
R3000	25	1988	0.11	66.12	145	4	64	64	Não
R4000	100	1991	1.35	213	179	15	8	8	1024
R4400	150	1992	2.3	186	179	15	16	16	1024
R4600	133	1994	2.2	77	179	4.6	16	16	512
R5000	180	1996	3.7	84	223	10	32	32	1024
R8000	90	1994	2.6	299	591	30	16	16	1024
R10000	200	1995	6.8	299	599	30	32	32	512
R12000	300	1998	6.9	204	600	20	32	32	1024
R14000	600	2001	7.2	204	527	17	32	32	2048
R16000	700	2002	--	--	--	20	64	64	4096
R16000A	800	2004	--	--	--	--	64	64	4096

A primeira CPU MIPS comercial foi o R2000, anunciado em 1985. Ele possui instruções de multiplicação e divisão e trinta e dois registradores de 32 bits de propósito geral. O R2000 também possui suporte a até quatro co-processadores, um desses podendo ser a Unidade de Ponto Flutuante (FPU, do inglês, *Floating Point Unit*) R2010 [WIK06].

O R3000 sucedeu o R2000 em 1988, adicionando caches de 32KB para dados e instruções, e provendo suporte à coerência de cache para multiprocessadores. O R3000 também incluiu uma MMU. Como o R2000, o R3000 pode ser conectado à FPU R3010.

A série R4000, lançada em 1991, estendeu o conjunto de instruções MIPS para uma arquitetura de 64 bits, absorveu a FPU, criando um sistema em único *chip*, operando até 100MHz. Com a introdução do R4000, logo apareceram versões melhoradas, incluindo o R4400 em 1993, o qual incluiu caches de 16KB e um controlador para outra cache externa (de nível 2) de 1MB.

Uma vez que a tecnologia MIPS pode ser licenciada, a companhia *Quantum Effect Devices* (QED), projetou o R4600, o R4700, o R4650 e o R5000. O R4600 e o R4700 foram usados em versões de baixo custo das estações de trabalho da companhia SGI (do inglês, *Silicon Graphics Incorporation*). O R4650 foi usado em *set-top boxes* para WebTV. A QED posteriormente projetou a família RM7000 e RM9000 de dispositivos para o mercado embarcado, como dispositivos de rede e impressoras a laser.

O R8000 (1994) foi o primeiro projeto MIPS superescalar, capaz de executar duas instruções aritméticas e duas operações de memória por ciclo de clock. O projeto foi distribuído sobre seis circuitos integrados: uma unidade de inteiros (com caches de 16KB para instruções e 16KB para dados), uma unidade de ponto-flutuante, duas para o controle dos *tags* das caches nível 2, uma para monitorar o barramento (para coerência de *cache*), e um controlador de cache.

Em 1995 o R10000 foi apresentado. Esse processador possui velocidade de relógio mais alta do que o R8000, tendo caches de dados e instruções de 32KB. Ele também é superescalar, mas sua maior inovação inclui execução das instruções fora de ordem.

Projetos recentes têm sido baseados no R10000. O R12000 utiliza processo de fabricação mais atual que o R10000, permitindo aumento da velocidade do relógio. O R14000 permite suporte à DDR SRAM na cache fora do *chip* e barramento de dados (FSB, do inglês, *Front-Side Bus*) mais rápido (200MHZ). As últimas versões compreendem o R16000 e o R16000A os quais apresentam velocidade de relógio mais alta, cache L1 (nível 1) adicional e um *chip* menor comparado com os anteriores.

MIPS32 24Kf

O MIPS32 24Kf é um processador sintetizável para aplicações embarcadas. Seu diagrama de blocos é apresentado na Figura 11. Possui uma arquitetura com *pipeline* de oito estágios. Sua MMU possui quatro entradas para instruções e oito para dados. Apresenta uma unidade de ponto flutuante (FPU) que suporta instruções de precisão simples e dupla e uma unidade de multiplicação/divisão (MDU, do inglês, *Multiple/Divide Unit*) de alto desempenho. O bloco CorExtend permite utilizar os registradores HI/LO na MDU. Caches de dados e instruções podem ser configuradas em 0, 16, 32 e 64KB. A unidade de interface com o barramento (BIU, do inglês, Bus Interface Unit) implementa o padrão OCP (*Open Core Protocol*) . Interfaces opcionais suportam blocos externos como co-processadores. O módulo EJTAG fornece suporte à debug.

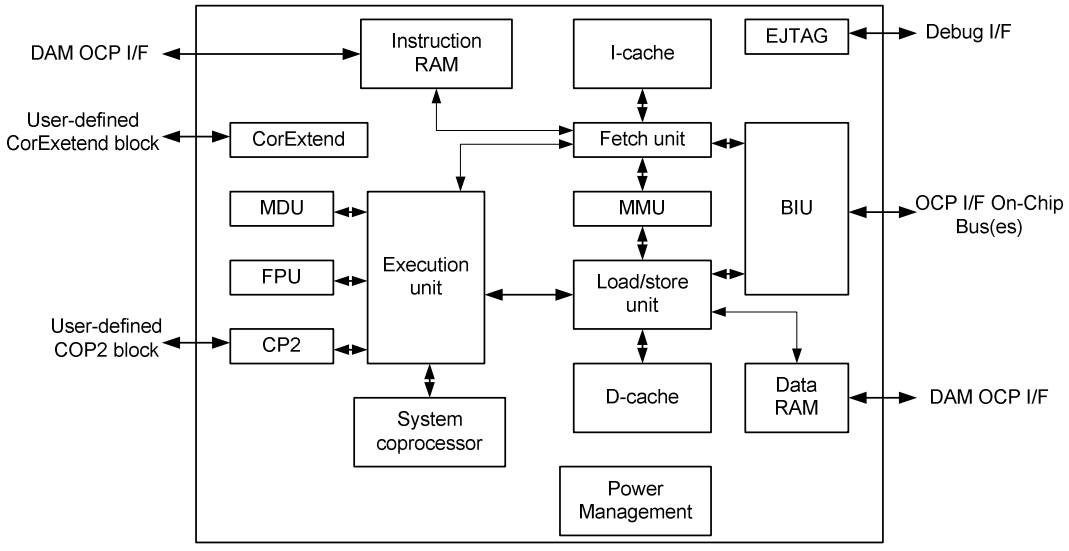


Figura 11 - Diagrama de blocos do MIPS32 24Kf.

3.3 Sistemas Operacionais Embarcados

Sistemas operacionais são programas intermediários responsáveis por controlar os recursos existentes em um computador (processadores, memórias, dispositivos de E/S, etc.), além de fornecer a base para o desenvolvimento de programas de aplicação [TAN97] [SIL95]. As aplicações

são compostas por tarefas, sendo uma tarefa um conjunto de instruções e dados com informações necessárias à sua correta execução em um processador. Além disso, os sistemas operacionais podem ser vistos como uma camada de software que provê um ambiente com uma interface mais simples e conveniente para o usuário.

Os sistemas operacionais dispõem de diversos tipos de serviços. No entanto, levando em consideração a maioria das implementações de sistemas operacionais existentes atualmente, pode-se dizer que os principais serviços implementados no kernel do sistema operacional são [TAN97] [SIL95]:

- **Escalonamento de tarefas**: este serviço é fundamental em qualquer sistema operacional, pois é necessário para gerenciar a utilização de recursos, bem como a ordem de execução das tarefas constituintes de sistema, o que é realizado de acordo com a política de escalonamento adotada.

- **Troca de contexto**: sempre que uma nova tarefa é escalonada, é necessário realizar uma troca de contexto no sistema, ou seja, o sistema operacional deve salvar as informações da tarefa que está executando e carregar as informações necessárias para a execução da nova tarefa. As trocas de contexto ocorrem em intervalos de tempo determinados, chamados *timeslice*.

- **Comunicação entre tarefas**: este serviço é responsável pelo gerenciamento da troca de informações entre tarefas cooperantes, ou seja, tarefas que necessitam de resultados de outra(s) tarefa(s) durante sua execução. Esta troca pode ser realizada através de memória compartilhada ou troca de mensagens.

- **Tratamento de interrupções**: interrupções são mecanismos através dos quais outros módulos (E/S, memória) podem interromper o processamento normal do processador, e sempre que isto acontece, o processador deve interromper sua execução e iniciar a execução de uma rotina específica a fim de tratar esta interrupção.

- **Gerenciamento de memória**: este serviço consiste na tarefa de conhecer as partes da memória que estão ou não em uso, alocar memória para as tarefas quando necessário e liberar memória quando estas estiverem com sua execução finalizada.

Como um subgrupo de sistemas operacionais, encontram-se os **sistemas operacionais embarcados**. Estes vêm tornando-se bastante populares, visto que implementam apenas as funcionalidades necessárias à aplicação que será executada. O tamanho de um sistema operacional embarcado (espaço de memória ocupado) tende a ser menor que o de um sistema operacional convencional, reduzindo o *kernel* a um *microkernel* (µkernel), o que é desejável para aplicações embarcadas, como por exemplo, aplicações para celulares. Assim como os sistemas operacionais convencionais, os sistemas operacionais embarcados oferecem mecanismos de escalonamento, tratamento de interrupções, gerenciamento de memória entre outros [WOL01]. Exemplos de sistemas operacionais embarcados são o µClinux [UCL06], o eCos [ECO06] e o EPOS [EPO06].

O µClinux é um sistema operacional derivado do Linux usado em microcontroladores que

não possuem unidades de gerenciamento de memória (MMUs). A grande contribuição do µClinux é o fato de possuir um código modular e pequeno (menos de 512KB para o *kernel* e menos de 900 KB para *kernel* mais ferramentas), reduzindo o consumo de energia e o uso de recursos computacionais. Contudo, não oferece bom suporte para multitarefas.

O eCos é um sistema operacional portável para arquiteturas 16, 32 e 64 bits e pode ser usado tanto em microprocessadores como microcontroladores. Sua biblioteca possui uma camada de abstração ao hardware, fornecendo suporte a diversas famílias de processadores. O eCos permite diversas políticas de escalonamento, mecanismo de interrupções, primitivas de sincronização e comunicação específica para a aplicação do usuário. Além disso, o eCos suporta gerenciamento de memória, tratamento de exceções, bibliotecas, *drivers*, etc. Ele possui ferramentas de configuração e construção, além de compiladores e simuladores da GNU.

O EPOS é um sistema operacional orientado a aplicação, i.e., se adapta automaticamente aos requisitos da aplicação que o usuário elabora. É concebido para aplicações dedicadas e portado para diversas arquiteturas de processadores.

A maioria dos sistemas operacionais embarcados existentes apresenta *(i)* consumo de memória elevado; *(ii)* serviços não necessários ao escopo deste trabalho (por exemplo, pilha TCP/IP, alocação dinâmica de memória) e *(iii)* elevada complexidade. Assim, optou-se por implementar um sistema operacional de tamanho reduzido contendo apenas os serviços necessários para alcançar os objetivos do trabalho (estes serviços são apresentados no Capítulo 5).

O tamanho do sistema operacional deve ser levado em conta, uma vez que a quantidade de memória disponível em sistemas embarcados é restrita. Neste trabalho, são utilizados 64KB de memória para cada processador (espaço no qual reside o sistema operacional e as tarefas). Cada BRAM (Block RAM) possui capacidade de 2KB, resultando em 32 BRAMs por processador. Uma vez que a plataforma criada utiliza 6 processadores, são necessárias 192 BRAMS. Para prototipar este sistema nas plataformas disponíveis no grupo GAPH, as quais contém dispositivos XC2VP30 (136 BRAMs), XC2V4000 (120 BRAMs) e XC4LX25 (72 BRAMs), será avaliado um MPSoC com 4 processadores.

4 INFRA-ESTRUTURA DE HARDWARE

Este Capítulo apresenta a primeira contribuição deste trabalho: o desenvolvimento da *infra-estrutura de hardware* da plataforma MPSoC. Esta infra-estrutura possui os seguintes componentes, que são mostrados na Figura 12 e na Figura 13:

1. **NoC HERMES**: realiza a interconexão dos núcleos e o roteamento de pacotes entre os mesmos (Figura 12(a));
2. **Plasma**: nodo processador que executa a aplicação. Cada processador possui uma memória local, a qual não é acessível a outros processadores (Figura 12(b));
3. **NI (Network Interface)**: faz a interface entre o processador e a rede (Figura 13).
4. **DMA (Direct Memory Access)**: transfere para a memória do processador o código-objeto das tarefas enviadas por um nodo mestre (Figura 13).

Os dois últimos componentes, NI e DMA, encontram-se dentro do módulo Plasma.

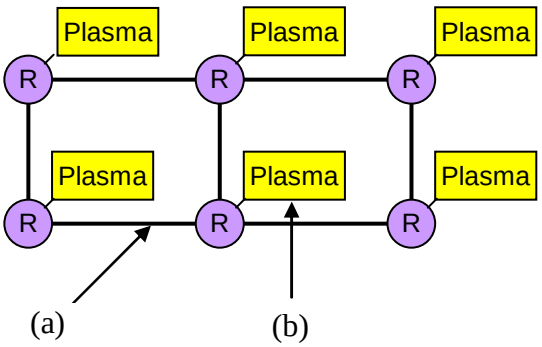


Figura 12 – (a) NoC HERMES; (b) processador Plasma.

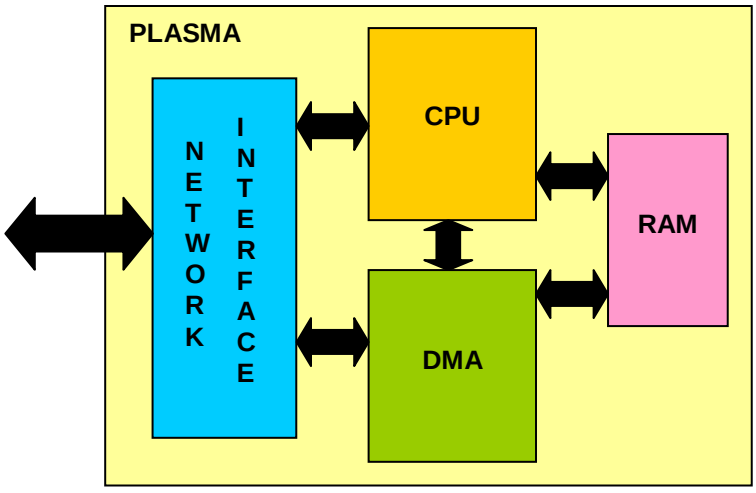


Figura 13 – Visão interna do nodo processador Plasma, contendo os núcleos: interface de rede (NI), a CPU (arquitetura MIPS), DMA e RAM.

A NoC HERMES [MOR04] e o processador Plasma [PLA06] são módulos IP não desenvolvidos no escopo do presente trabalho. A contribuição refere-se principalmente à integração dos diversos núcleos (processadores Plasma) à rede e no desenvolvimento dos módulos NI e DMA. As próximas seções discutem cada um dos componentes desta infra-estrutura.

4.1 NoC HERMES

Para a interconexão dos núcleos do MPSoC este trabalho utiliza a NoC HERMES, desenvolvida no grupo de pesquisa GAPH [GAP06]. A razão desta escolha é em virtude das vantagens que redes intra-chips proporcionam ao projeto de MPSoCs, bem como o domínio dessa tecnologia pelo grupo.

A NoC HERMES possui um mecanismo de comunicação denominado *chaveamento de pacotes*, no qual os pacotes são roteados individualmente entre os nodos sem o estabelecimento prévio de um caminho. Este mecanismo de comunicação requer o uso de um modo de roteamento para definir como os pacotes devem se mover através dos roteadores. A NoC HERMES utiliza o modo de roteamento *wormhole*, no qual um pacote é transmitido entre os roteadores em *flits*. Apenas o *flit* de cabeçalho possui a informação de roteamento. Assim, os *flits* restantes que compõe o pacote devem seguir o mesmo caminho reservado pelo cabeçalho [MOR04].

A NoC HERMES utiliza uma topologia malha, ilustrada na Figura 14. Essa topologia é justificada em função da facilidade de desenvolver o algoritmo de roteamento, inserir núcleos e gerar o leiaute do circuito [MEL05].

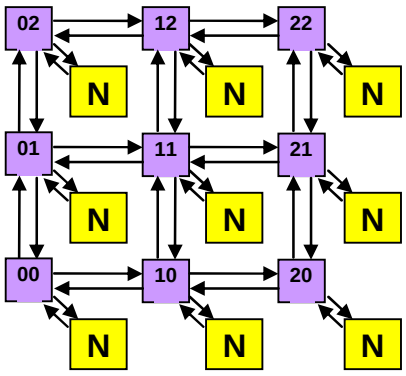


Figura 14 – Um exemplo de topologia malha para a NoC HERMES. N representa um núcleo e os endereços dos roteadores indicam a posição XY na rede.

A interface de comunicação entre roteadores Hermes vizinhos é apresentada na Figura 15. Os seguintes sinais compõem a porta de saída: (1) `Clock_tx`: sincroniza a transmissão de dados; (2) `Tx`: indica disponibilidade de dado; (3) `Lane_tx`: indica o canal virtual transmitindo dado; (4) `Data_out`: dado a ser transmitido; (5) `Credit_in`: informa disponibilidade de *buffer* no roteador vizinho, para cada canal virtual. O número de canais virtuais (`l` *lanes*) e a largura do barramento de dados (`n` bits) são parametrizáveis em função dos recursos de roteamento disponíveis e memória disponível para esquemas de buferização.

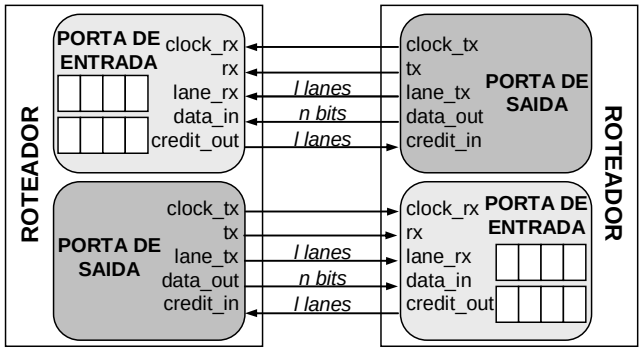


Figura 15 - Interfaces entre roteadores Hermes-VC.

O roteador Hermes possui uma lógica de controle de chaveamento centralizada e 5 portas bidirecionais: East, West, North, South e Local. A porta Local estabelece a comunicação entre o roteador e seu núcleo local. As demais portas ligam o roteador aos roteadores vizinhos. Cada porta unidirecional (entrada ou saída) do roteador corresponde a um canal físico. O modo de chaveamento *wormhole* adotado no roteador Hermes permite que cada canal físico seja multiplexado em n canais virtuais (VCs). A Figura 16 apresenta o roteador Hermes com dois VCs por canal físico.

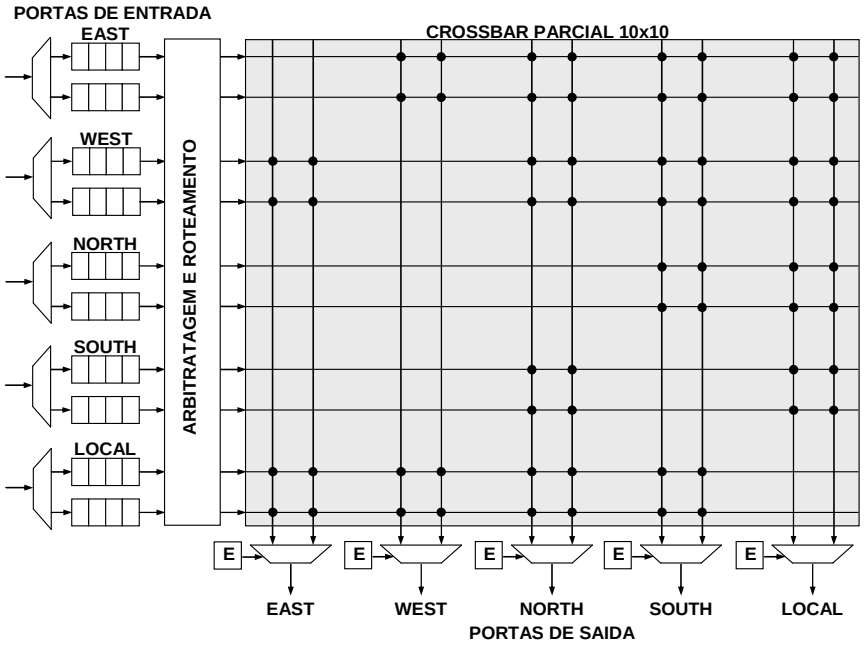


Figura 16 – Estrutura do roteador Hermes com dois canais virtuais. O módulo "E" na porta de saída representa o circuito que escalona uma dada porta de entrada para uma dada porta de saída.

A cada porta de entrada é adicionado um *buffer* para diminuir a perda de desempenho com o bloqueio de *flits*. A perda de desempenho acontece porque quando um *flit* é bloqueado em um dado roteador os *flits* seguintes do mesmo pacote também são bloqueados em outros roteadores. Com a inserção de um *buffer* o número de roteadores afetados pelo bloqueio dos *flits* diminui. O *buffer* inserido no roteador Hermes funciona como uma fila FIFO (First In First Out) circular, cuja profundidade p é parametrizável. Quando o canal físico é dividido em n VCs, uma fila FIFO de profundidade p/n é associada a cada VC. Por exemplo, o roteador apresentado na Figura 16 possui um espaço de armazenamento de 8 *flits* por porta de entrada, sendo multiplexado em dois VCs, com

cada *buffer* associado a cada VC com profundidade de 4 *flits* (8/2).

A porta de entrada é responsável por receber *flits* de pacotes e armazená-los no *buffer* do VC correto. A seleção do *buffer* é realizada através do sinal `lane_rx`, que informa a qual VC pertence o *flit* no canal físico. O sinal `lane_rx` possui `n` bits, onde `n` é o número de VCs. Após a seleção do *buffer* correto, o *flit* é armazenado e o número de créditos do VC (espaços livres para armazenamento) é decrementado. Quando o *flit* é transmitido pela porta de saída, o *flit* é removido do *buffer* e o número de créditos incrementado. A informação de crédito disponível é transmitida ao roteador vizinho através do sinal `credit_o`. O sinal `credit_o` também possui 1 bit para cada VC e é interpretado de forma análoga ao sinal `lane_rx`.

A lógica de controle de chaveamento implementa uma lógica de arbitragem e um algoritmo de roteamento. Quando um roteador recebe um *header flit*, a arbitragem é executada e se a requisição de roteamento do pacote é atendida, um algoritmo de roteamento é usado para conectar o *flit* da porta de entrada à correta porta de saída. Cada roteador deve ter um endereço único na rede. Para simplificar o roteamento na rede, este endereço é expresso nas coordenadas XY, onde X representa a posição horizontal e Y a posição vertical.

Um roteador pode ser requisitado para estabelecer até $nxm+1$ conexões simultaneamente, onde `n` representa o número de portas de entrada, `m` representa o número de VC e `1` representa a porta local. Uma lógica de arbitragem centralizada é usada para garantir acesso a um VC de entrada quando um ou mais VCs requerem simultaneamente uma conexão. A política de arbitragem Round-Robin é usada no roteador Hermes. Essa política utiliza um esquema de prioridades dinâmicas, proporcionando um serviço mais justo que a prioridade estática, evitando também *starvation*. A prioridade atribuída a um VC é determinada pelo último VC a ter uma requisição de acesso atendida.

No roteador Hermes, o algoritmo de roteamento XY é utilizado. O algoritmo XY compara o endereço do roteador atual (X_LY_L) com o endereço do roteador destino (X_TY_T) do pacote, armazenado no *header flit*. Os *flits* devem ser roteados para a porta Local quando o endereço X_LY_L do roteador atual é igual ao endereço X_TY_T do pacote. Se esse não for o caso, o endereço X_T é primeiro comparado ao endereço (horizontal) X_L. Os *flits* serão roteados para a porta East quando $X_L < X_T$, para West quando $X_L > X_T$ e se $X_L = X_T$ o *header flit* já está alinhado horizontalmente. Se esta última condição é a verdadeira, o endereço (vertical) Y_T é comparado ao endereço Y_L. Os *flits* serão roteados para a porta North quando $Y_L < Y_T$, para South quando $Y_L > Y_T$. Se a porta escolhida está ocupada, o *header flit* também como todos os *flits* subseqüentes do pacote serão bloqueados. No roteador Hermes com canais físicos multiplexados em VCs, uma porta (canal físico) é considerada ocupada quando todos os VCs estão ocupadas. No algoritmo de roteamento XY não é necessário estabelecer uma ordem entre os VCs, porque a restrição de percorrer primeiro a coordenada X e depois a coordenada Y é suficiente para evitar a ocorrência de *deadlock*.

Quando o algoritmo de roteamento encontra um canal de saída livre para uso, a conexão entre o VC de entrada e o VC de saída é estabelecido e a tabela de chaveamento é atualizada. A tabela de chaveamento é composta por três vetores: `in`, `out` e `free`. O vetor `in` conecta um VC

de entrada a um VC de saída. O vetor `out` conecta um VC de saída a um VC de entrada. O vetor `free` é responsável por modificar o estado do VC de saída de livre (1) para ocupado (0). Os vetores `in` e `out` são preenchidos por um identificador único construído pela combinação do número da porta e do número do VC.

Depois que todos os *flits* do pacote forem transmitidos, a conexão deve ser encerrada. Isto pode ser feito de dois modos diferentes: por um *trailer* ou usando um contador de *flits*. Um *trailer* requer um ou mais *flits* para serem usados como terminador de pacote e uma lógica adicional é necessária para detectar o *trailer*. Para simplificar o projeto, o roteador possui um contador para cada VC de entrada. O contador de um canal específico é inicializado quando o segundo *flit* do pacote é recebido, indicando o número de *flits* que compõem o *payload*. O contador é decrementado a cada *flit* transmitido com sucesso. Quando o valor do contador alcança zero, a posição do vetor free correspondente ao VC de saída vai para um (`free=1`), encerrando a conexão.

No contexto do presente trabalho a configuração da rede adotada é:

1. Topologia malha 3 *x* 2, suportando 6 nodos de processamento;
2. Largura de *flit* igual a 16 bits (optou-se por uma largura de *flit* inferior ao tamanho da palavra do processador (32 bits) para reduzir a área da rede);
3. Profundidade dos *buffers* igual a 8 *flits*;
4. Roteamento XY e arbitragem Round Robin;
5. Canais virtuais não são utilizados, simplificando o projeto do hardware. Trabalhos futuros, em redes maiores, poderão incluir canais virtuais para redução da congestão na rede.

4.2 Processador Plasma

O Plasma é um processador RISC de 32 bits com um subconjunto de instruções da arquitetura MIPS [HEN98]. Seu código (VHDL) é aberto, disponível através do *OpenCores* [OPE06]. O *pipeline* de instruções do Plasma contém três estágios: busca, decodificação e execução. Diferentemente da definição original do processador MIPS, a organização de memória do Plasma é Von Neumann e não Harvard. Além disso, o Plasma oferece suporte ao compilador C (`gcc`) e tratamento de interrupções. O diagrama da Figura 17 apresenta a interconexão do processador Plasma (`mlite_cpu`) a alguns periféricos que acompanham a distribuição do processador: interface serial (`uart`), memória para dados e programa (`ram`), contador de tempo para interrupção (`counter_reg`), portas de entrada e saída (`gpioA-reg` e `gpioO_reg`), máscara de interrupções (`irq_mask_reg`) e registrador de status das interrupções (`irq_status`).

Para atingir o objetivo pretendido neste trabalho, foram necessárias modificações no Plasma original. Estas modificações dizem respeito à criação de novos mecanismos, exclusão de módulos e registradores mapeados em memória, bem como inclusão de novos módulos e novos

registradores mapeados em memória. Estas mudanças são discutidas a seguir.

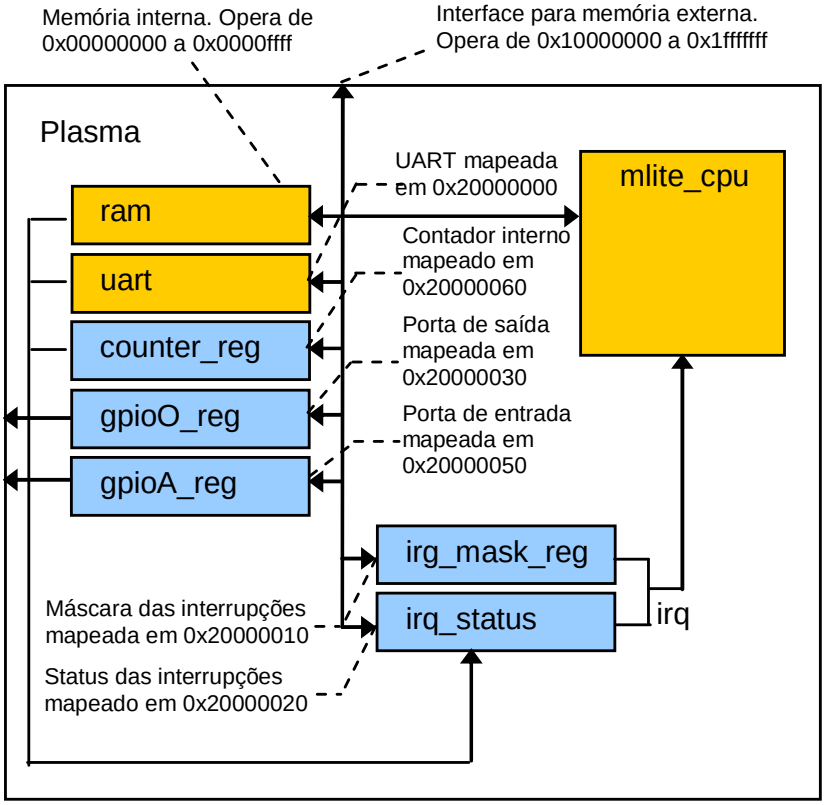


Figura 17 – Processador Plasma original (mlite_cpu e periféricos).

Para tornar possível a execução de múltiplas tarefas sobre a mesma CPU, foi criado um **mecanismo de paginação**. As mudanças realizadas afetam a geração de endereços. No Plasma original o endereço é composto por 13 bits, fazendo com que a memória tenha uma capacidade de 8192 bytes (2048 palavras de 32 bits). O tamanho desse endereço foi aumentado, no contexto deste trabalho, em 3 bits, totalizando em 65536 bytes de memória (16384 palavras de 32 bits). A memória é dividida em quatro páginas de 4096 palavras. Conforme mostra a Figura 18, os 2 bits mais significativos (14 e 15) do endereço indicam a página e o restante (0 a 13) indicam o deslocamento dentro da mesma.

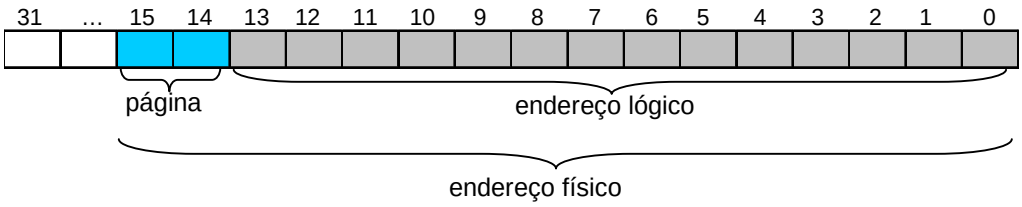


Figura 18 – Composição do endereço no Plasma modificado.

Ao banco de registradores do processador (entidade `Reg_bank`) foi adicionado um registrador, `page`, referente à página da memória. A configuração da página é realizada através da instrução `mtc0 reg,$10`. Nesta instrução, o endereço inicial de uma tarefa na memória (*offset*) contido no registrador `reg` é carregado para o registrador `$10` do CP0 (co-processador 0), que

corresponde ao registrador `page`.

O controlador de memória (entidade `Mem_ctrl`) gera um endereço (`mem_address_wop`[1]) que não contém a página, apenas o endereço lógico de deslocamento dentro da mesma. Dessa forma, o endereço físico gerado pela CPU é composto pela concatenação do endereço lógico (`mem_address_wop`, fornecido pelo controlador de memória) com a informação de página (`page`), como mostra a Figura 19.

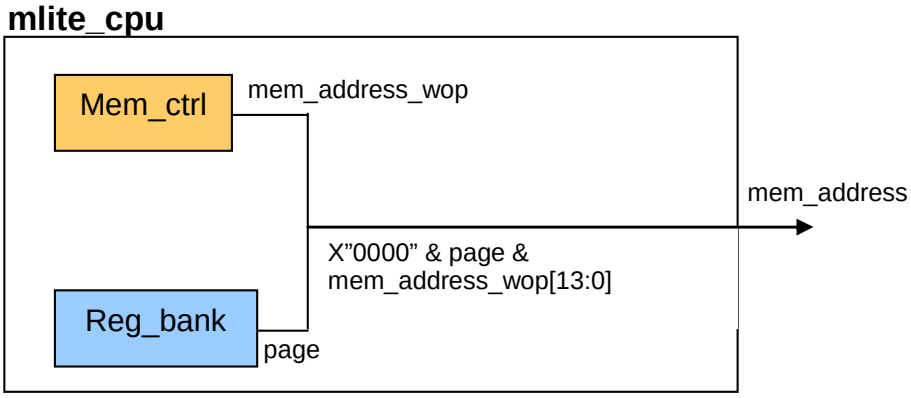


Figura 19 – Geração de endereços com adição da página.

Este mecanismo de paginação oferece segurança no acesso à memória evitando violação de endereços. Isso quer dizer que uma tarefa residente na página p_x nunca conseguirá acessar qualquer endereço na página p_y (sendo $x \neq y$), uma vez que todo endereço lógico gerado pela tarefa será concatenado com a página p_x.

O Plasma original não oferece suporte a interrupções de software. No entanto, a aplicação pode fazer chamadas de sistema, requisitando um serviço ao sistema operacional como, por exemplo, envio e recebimento de mensagens. Dessa forma, foi necessária a inclusão da instrução `syscall`, prevista na arquitetura MIPS, ao conjunto de instruções do Plasma. Esta instrução gera uma interrupção desviando o fluxo de execução do processador para um endereço pré-estabelecido, onde a chamada de sistema vai ser tratada. Chamadas de sistema são abordadas na Seção 5.7.

Foi também desenvolvido um módulo responsável por fazer a interface entre o processador e a NoC, a `Network Interface` (`NI`). Outro módulo, `DMA`, foi desenvolvido para transferir o código-objeto de tarefas que chegam na `NI` para a memória do processador. Estas duas entidades são detalhadas nas próximas subseções. A `uart`, contida no Plasma original (Figura 17), não se faz mais presente. A Figura 20 mostra o diagrama do módulo Plasma, com os respectivos componentes internos, após as modificações realizadas.

Ainda, foram excluídos os registradores mapeados em memória `gpio0_reg` e `gpioA_reg` (Figura 17) e foram acrescentados novos registradores destinados à comunicação entre as entidades Plasma, NI e DMA. São eles:

- Mlite_CPU↔NI : `Configuration`;
- Mlite_CPU ↔NI e DMA↔NI: `Status_Read`, `Status_Send`, `Read_Data`,

[1] wop – without page.

`Write_Data`, `Packet_ACK`, `Packet_NACK`, `Packet_END`;

- Mlite_CPU↔DMA: `Set_DMA_Size`, `Set_DMA_Address`, `Start_DMA`, `DMA_ACK`.

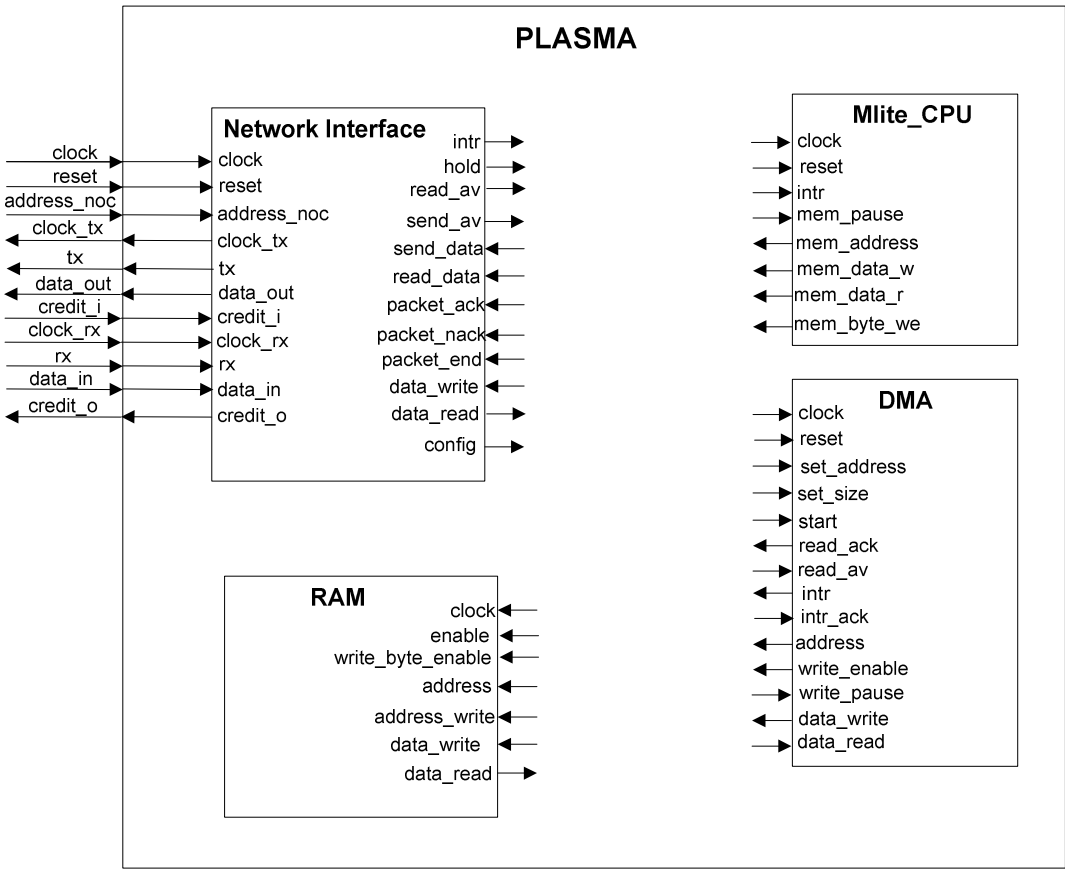


Figura 20 – Módulo PLASMA, contendo a CPU (Mlite_CPU), os módulos DMA, RAM e NI.

Visando facilitar a avaliação de desempenho do sistema, foi criado outro registrador mapeado em memória, chamado `Tick_Counter`. Este registrador acumula os ciclos de relógio durante a execução do sistema e pode ser lido pelo *microkernel* ou pela aplicação através de uma chamada de sistema (ver Seção 5.7). Com isso, pode-se medir o número de ciclos de relógio gastos para uma determinada operação do *microkernel* como, por exemplo, troca de contexto, escalonamento, ou para uma operação da aplicação.

Cada um dos registradores são abordados na seqüência do Capítulo,

4.3 Network Interface (NI)

A NI foi desenvolvida para realizar a interface entre o processador e a NoC HERMES. Ela é responsável por:

1. Enviar pacotes para a rede, segmentando os dados deste pacote em *flits*. Neste trabalho, a NI recebe do processador um pacote cujos dados possuem 32 bits de tamanho e os divide em segmentos de 16 bits (tamanho do *flit* da NoC);

2. Receber pacotes da rede armazenando-os em um *buffer*. Quando existir um pacote completo no

buffer ou quando o *buffer* estiver cheio, a NI interrompe o processador para que este receba os dados. Os segmentos de 16 bits referentes ao conteúdo do pacote recebido da rede são agrupados e repassados para o processador com palavras de 32 bits;

3. Repassar o código-objeto de tarefas recebido da rede, através do DMA, para a memória;
4. Informar ao processador (*microkernel*) qual a sua localização na rede (`netAddress`).

Um pacote que trafega na rede possui o seguinte formato:

`<target><size><payload>`

onde `target` indica o destino do pacote; `size` indica o tamanho, em *flits*, do conteúdo do pacote; `payload` indica o conteúdo do pacote. Os campos `target` e `size` possuem tamanho de 16 bits. O campo `payload` é constituído por:

`<service><service_parameters>`

onde `service` é o serviço solicitado e `service_parameters` são os parâmetros necessários a este serviço. O serviço reflete na ação a ser tomada pelo *microkernel* depois de ter recebido o pacote. Os serviços que um pacote pode carregar são descritos na Tabela 3.

Tabela 3 – Descrição dos serviços que um pacote carrega.

Serviço	Código	Descrição
REQUEST_MESSAGE	0x00000010	Requisição de uma mensagem.
DELIVER_MESSAGE	0x00000020	Entrega de uma mensagem previamente solicitada.
NO_MESSAGE	0x00000030	Aviso de que a mensagem solicitada não existe.
TASK_ALLOCATION	0x00000040	Alocação de tarefas: uma tarefa deve ser transferida pelo DMA para a memória do processador.
ALLOCATED_TASK	0x00000050	Aviso de que uma nova tarefa está alocada no sistema.
REQUEST_TASK	0x00000060	Requisição de uma tarefa.
TERMINATED_TASK	0x00000070	Aviso de que uma tarefa terminou a sua execução.
DEALLOCATED_TASK	0x00000080	Aviso de que uma tarefa terminou a sua execução e pode ser liberada.
FINISHED_ALLOCATION	0x00000090	Aviso que o nodo mestre terminou a alocação inicial das tarefas (alocação estática).

Para cada serviço existem diferentes parâmetros:

- `REQUEST_MESSAGE`:

`<source_processor><message_target><message_source>`

onde `source_processor` é o processador que está requisitando a mensagem; `message_target` é o identificador da tarefa que gerou o pedido de mensagem e `message_source` é o indentificador da tarefa fonte da mensagem.

- DELIVER_MESSAGE:

<source_processor><message_target><message_source>
<message_size><message>

onde source_processor é o processador que está entregando a mensagem; message_target é o identificador da tarefa que gerou o pedido de mensagem; message_source é o indentificador da tarefa fonte da mensagem; message_size é o tamanho da mensagem em palavras de 32 bits e message é a mensagem.

- NO_MESSAGE:

<source_processor><message_target><message_source>

onde source_processor é o processador que está entregando a resposta; message_target é o identificador da tarefa que gerou o pedido de mensagem e message_source é o identificador da tarefa fonte da mensagem.

- TASK_ALLOCATION:

<task_id><code_size><code>

onde task_id é o identificador da tarefa que deve ser alocada, code_size é o tamanho, em palavras de 32 bits, do código objeto da tarefa e code é o código objeto.

- ALLOCATED_TASK:

<processor><task_id>

onde processor é o processador onde a tarefa cujo identificador é task_id, foi alocada.

- REQUEST_TASK:

<processor><task_id>

onde processor é o processador que está requisitando uma tarefa e task_id é o identificador da tarefa requisitada.

- TERMINATED_TASK:

<task_id><processor>

onde task_id é o identificador da tarefa que terminou a execução e processor é o processador onde a tarefa está alocada.

- DEALLOCATED_TASK:

<task_id>

que significa o identificador da tarefa que está sendo liberada.

No serviço FINISHED_ALLOCATION o parâmetro utilizado é zero.

Todos os pacotes são montados pelos *drivers de comunicação* presentes no *microkernel* do

processador, fazendo parte da infra-estrutura de software da plataforma (ver Seção 5.8). Estes *drivers*, além de montar os pacotes, os enviam à NI e recebem pacotes da NI. A comunicação entre *drivers* e NI acontece através de escrita/leitura em/de registradores mapeados em memória. Estes registradores são descritos na Tabela 4.

Tabela 4 – Registradores mapeados em memória para a comunicação entre *drivers* e NI.

Registrador	Endereço	Descrição
`Status_Read`	`0x20000100`	*Driver* lê deste registrador para verificar se tem dado para leitura.
`Status_Send`	`0x20000110`	*Driver* lê deste registrador para verificar se pode enviar um dado à NI.
`Read_Data`	`0x20000120`	*Driver* lê deste registrador o dado esperado.
`Send_Data`	`0x20000130`	*Driver* escreve neste registrador para enviar um dado à NI.
`Packet_ACK`	`0x20000150`	*Driver* escreve neste registrador para informar à NI que pode receber o pacote.
`Packet_NACK`	`0x20000160`	*Driver* escreve neste registrador para informar à NI que não pode receber o pacote.
`Packet_END`	`0x20000170`	*Driver* escreve neste registrador para indicar à NI que terminou de receber o pacote.

Além destes registradores, existe ainda o registrador `Configuration`, cujo endereço é `0x20000140` e é lido pelo *microkernel* de forma a conhecer a localização do processador na rede (endereço de rede - `netAddress`).

A Figura 21 mostra os sinais da NI que fazem interface com a NoC e os sinais que fazem interface com a CPU. A Tabela 5 descreve cada um destes sinais.

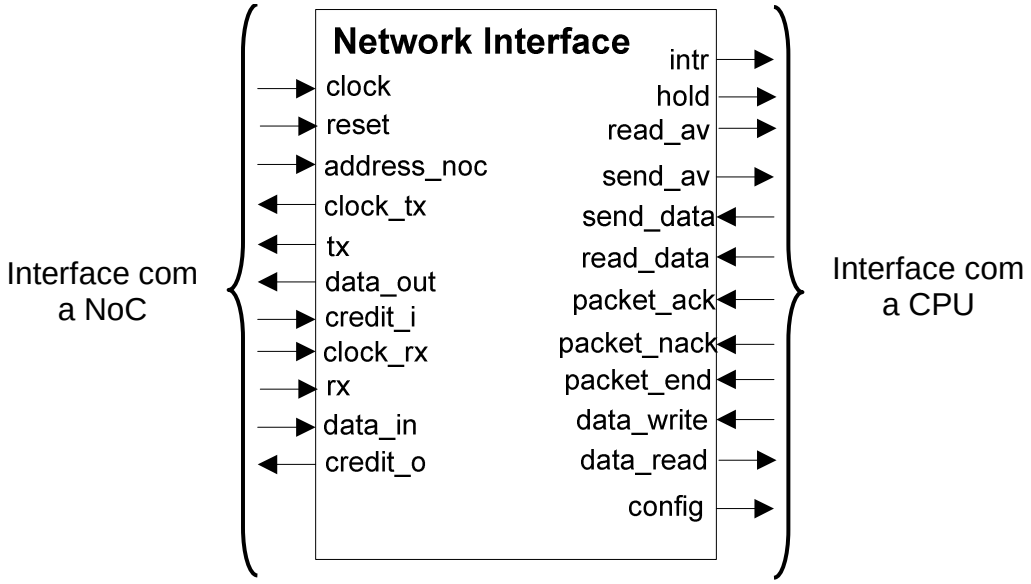


Figura 21 – Sinais de Network Interface.

Tabela 5 – Descrição dos sinais da NI que fazem interface com a NoC e com a CPU.

Sinal	Tipo	Descrição
Interface com a NoC		
address_noc	Entrada	Endereço do nodo na rede
clock_tx	Saída	Clock da linha de envio que coordena a transmissão (controle de fluxo é síncrono)
data_out	Saída	Dado enviado para a NoC
tx	Saída	Informa à NoC que tem dado a enviar
credit_i	Entrada	Indica se a NoC pode receber dados
clock_rx	Entrada	Clock da linha de recebimento
data_in	Entrada	Dado recebido da NoC
rx	Entrada	Indica se tem dado a receber da NoC
credit_o	Saída	Informa à NoC se pode receber dados
Interface com a CPU		
intr	Saída	Interrupção da CPU
hold	Saída	Informa à CPU o momento de começar a executar.
send_av	Saída	Informa à CPU se pode ser enviado um dado para a NoC (registrador Status_Send)
read_av	Saída	Informa à CPU se existe dado disponível para a leitura (registrador Status_Read)
send_data	Entrada	Indica quando o dado em data_write deve ser enviado.
read_data	Entrada	Indica quando o dado em data_read foi lido
packet_ack	Entrada	Indica que a CPU pode receber o pacote (registrador Packet_ACK)
packet_nack	Entrada	Indica que a CPU não pode receber o pacote (registrador Packet_NACK)
packet_end	Entrada	Indica que a CPU terminou de receber os dados de um pacote (registrador Packet_END)
data_write	Entrada	Dado recebido da CPU a ser enviado para a NoC (registrador write_data)
data_read	Saída	Dado recebido da NoC e repassado para a CPU (registrador read_data)
config	Saída	Informa à CPU seu endereço de rede (registrador CONFIGURATION)

4.3.1 Envio de pacotes para a NoC

Conforme já citado no início desta Seção, a NoC recebe e envia dados com tamanho de 16 bits e o processador escreve e recebe os dados com tamanho de 32 bits. Dessa forma, a NI deve

agrupar os dados provenientes da NoC repassando para o processador e dividir os dados provenientes do processador e repassar para a NoC.

A Figura 22 mostra como o *driver* de transmissão envia um dado para a NI. O dado encontra-se no registrador `$4` (`a0`). Os endereços dos registradores `Status_Send` e `Send_Data` são carregados para os registradores `$8` (`t0`) e `$9` (`t1`), respectivamente (linhas 4 e 5). Verifica-se se é possível enviar um dado à NI através da leitura de `$8`, que contém `Status_Send` (linha 8). Se o dado não pode ser enviado, é porque a NoC não pode receber dados. Então, o *driver* aguarda dentro do laço (linhas 7 a 10)[1] por um espaço no *buffer* para escrever o dado. Se o dado pode ser enviado, ele é escrito no endereço especificado pelo registrador `$9` (`Send_Data`) (linha 12).

```
1     .equ STATUS_SEND  ,0x20000110
2     .equ SEND_DATA    ,0x20000130
3
4      la   $8,STATUS_SEND
5      la   $9, SEND_DATA
6
7 request_busy_target:
8      lw   $13,0($8)
9      beq  $13,$0,request_busy_target
10     nop
11
12     sw   $4,0($9)
```

Figura 22 – Envio de um dado à NI.

O dado que a NI recebe do processador é armazenado em um *buffer* de 32 bits (`buffer_out`). O dado enviado para a NoC é a metade mais significativa deste *buffer*, como mostra a Figura 23(a). Para enviar a metade menos significativa, esta é deslocada para a metade mais significativa (Figura 23(b)).

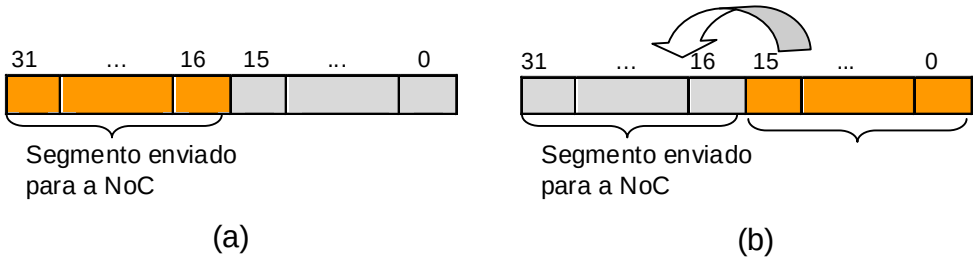


Figura 23 – Segmentação de um dado recebido do processador para o envio à NoC.

A Figura 24 apresenta a máquina de transição de estados de NI que faz o envio de um pacote para a NoC. A máquina de estados inicia no estado `Starget`. Neste estado é enviado para a NoC o destino do pacote. Uma vez que o destino do pacote possui 16 bits, a metade menos significativa do dado recebido do processador é copiada para a metade mais significativa de `buffer_out`. O estado avança para `Ssize` se o processador deseja enviar mais um dado

[1] A instrução `nop` é importante, pois a arquitetura MIPS tem a característica *delayed-branch*, a qual executa as instruções após os saltos.

(`send_data=1`) e se a NoC pode receber o dado (`waiting_out=0`). Em `Ssize` é enviado para a NoC o tamanho, em *flits*, do pacote. Assim como o destino, o tamanho do pacote também possui 16 bits. Então, a metade menos significativa do dado recebido do processador é copiada para a metade mais significativa de `buffer_out`. O estado avança para `Spayload` se o processador deseja enviar mais um dado (`send_data=1`) e se a NoC pode receber o dado (`waiting_out=0`). Em `Spayload` é enviado todo o conteúdo do pacote para a NoC, cujos dados possuem 32 bits. Aqui, os dados recebidos do processador são armazenados inteiramente em `buffer_out` fazendo o deslocamento da metade menos significativa para a parte mais significativa.

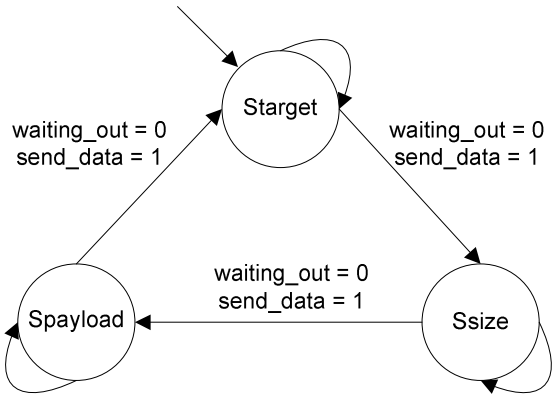


Figura 24 – Máquina de estados para o envio de pacotes à NoC.

A Figura 25 ilustra a segmentação de pacotes realizada pela NI. Na Figura 25(a) têm-se um pacote de requisição de mensagem montado pelo *driver*. Na Figura 25(b) têm-se o pacote segmentado em *flits* de 16 bits para o envio à NoC. Cada campo do `payload`, que possui tamanho de 32 bits é divido em dois *flits*.

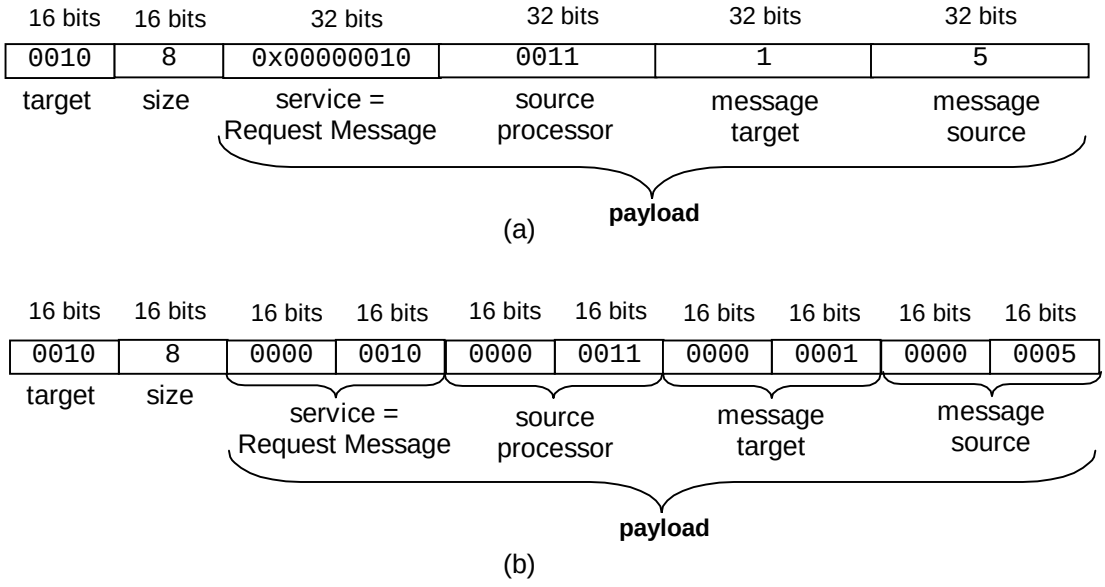


Figura 25 – Segmentação de pacotes. (a) pacote recebido do processador; (b) pacote enviado para a NoC.

4.3.2 Recebimento de pacotes da NoC

Os dados recebidos da NoC são armazenados em um *buffer*, que é composto por LUT_RAMs. Dois apontadores, `LAST` e `FIRST` no *buffer* indicam, respectivamente, a posição em que um dado recebido da NoC vai ser escrito e a posição do dado a ser lido pelo processador. A Figura 26 mostra o *buffer* com os apontadores. Dados recebidos da NoC são armazenados de 16 em 16 bits. Para repassar um dado ao processador, dois *flits* do *buffer* são agrupados.

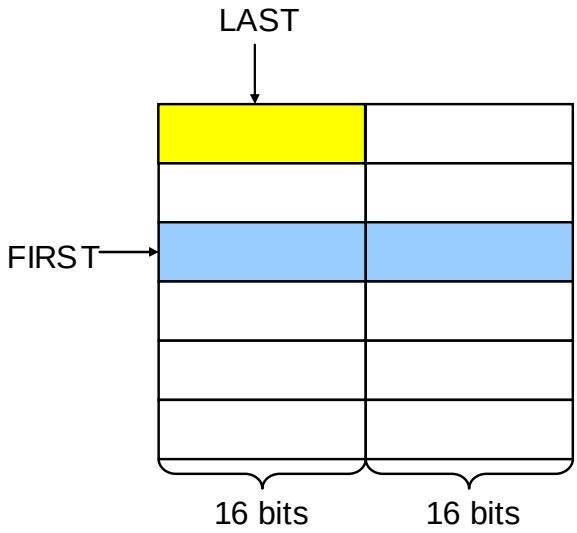


Figura 26 – *Buffer* onde são armazenados os dados provenientes da NoC.

A máquina de transição de estados de NI que faz o recebimento de um pacote é mostrada na Figura 27. O estado inicial é `Swait`, no qual espera-se receber o primeiro *flit* do pacote (correspondente ao destino). Se existem mais dados a serem recebidos (`rx=1`) e tem espaço no *buffer* para armazená-los (`tem_espaço=1`), o estado avança para `Ssize`. Em `Ssize`, o apontador `LAST` é incrementado e o tamanho do pacote é armazenado no *buffer* e também em um sinal (`size_in`) para o controle do recebimento. Novamente, se existem mais dados a serem recebidos (`rx=1`) e tem espaço no *buffer* para armazená-los (`tem_espaço=1`), o estado avança para `Swasting`. Neste estado, os *flits* do payload são armazenados no `buffer` e a cada flit armazenado, `LAST` é incrementado e `size_in` é decrementado. Quando o último flit do pacote deve ser armazenado no *buffer* (`size_in=1`), o estado passa a ser `Sending`. Em `Sending`, além de armazenar o último flit do pacote no *buffer*, `LAST` aponta para a próxima linha do *buffer*.

A Figura 28 mostra como o *driver* recebe um dado da NI. Os endereços dos registradores `Status_Read` e `Read_Data` são carregados para os registradores `$8` e `$9`, respectivamente (linhas 4 e 5). Verifica-se se o dado está disponível através da leitura do endereço especificado por `$8` (`Status_Read`) (linha 8). Se o dado não está disponível, é porque o *buffer* de NI que armazena os dados advindos da NoC está vazio. Então, o *driver* aguarda dentro do laço (linhas 7 a 10) pela chegada do dado no *buffer* para a leitura do mesmo. Se o dado está disponível, ele é lido do endereço contido no registrador `$9` (`Read_Data`)(linha 12).

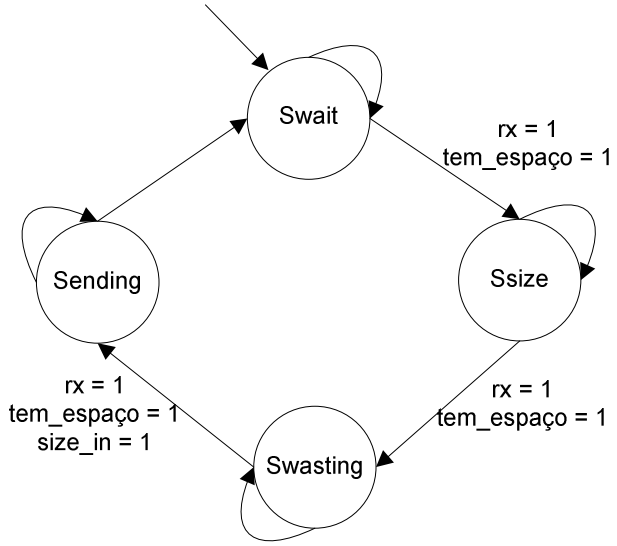


Figura 27 – Máquina de estados para o recebimento de pacotes da NoC.

```
1     .equ STATUS_READ  ,0x20000100
2     .equ READ_DATA    ,0x20000120
3
4     la    $8, STATUS_READ
5     la    $9, READ_DATA
6
7 deliver_source_processor:
8     lw    $12,0($8)
9     beqz  $12,deliver_source_processor
10    nop
11
12    lw    $13,0($9)
```

Figura 28 - Recebimento de um dado da NI.

4.4 Direct Memory Access (DMA)

O DMA foi desenvolvido para transferir o código objeto de uma tarefa para a memória do processador permitindo que este continue sua execução. O código objeto de uma tarefa é enviado pela rede para um determinado processador. O DMA é responsável por transferir este código objeto para a memória do processador.

Assim como qualquer outro pacote, quem recebe o código objeto da tarefa é a NI, armazenando-o no *buffer* de recebimento. Uma vez o pacote recebido, as seguintes operações são realizadas:

1. A NI interrompe a CPU informando a chegada de um pacote;
2. O *microkernel*, que executa na CPU e faz parte da infra-estrutura de software do sistema (Capítulo 5), interpretará o pedido de interrupção como nova tarefa a ser alocada. O *microkernel* obtém o identificador da tarefa e o tamanho do código objeto da mesma e verifica a

disponibilidade de página livre na memória. A CPU informa ao DMA o endereço da memória a partir do qual o código objeto deve ser transferido e o tamanho do código objeto;

3. O DMA faz acessos à NI para ler o código objeto e acessos à memória para escrever o mesmo. Quando o código já estiver alocado, o DMA interrompe a CPU informando que uma nova tarefa está na memória;
4. O *microkernel* faz as inicializações da tarefa, e partir disso a tarefa executará quando for escalonada.

Este mecanismo permite que o processador realize a execução das suas tarefas em paralelo com a recepção de novas tarefas. A comunicação entre o *microkernel* e o DMA ocorre através de registradores mapeados em memória. A Tabela 6 descreve estes registradores.

Tabela 6 – Registradores mapeados em memória para a comunicação entre CPU e DMA.

Registrador		Descrição
Set_DMA_Size	0x20000200	*Microkernel* escreve neste registrador para informar ao DMA o tamanho do código objeto a ser transferido para a memória
Set_DMA_Address	0x20000210	*Microkernel* escreve neste registrador para informar ao DMA o endereço da memória a partir do qual o código deve ser transferido.
Start_DMA	0x20000220	*Microkernel* escreve neste registrador para informar ao DMA que a transferência pode ser iniciada.
DMA_ACK	0x20000230	*Microkernel* escreve neste registrador para informar ao DMA que a interrupção foi aceita.

A Figura 29 mostra os sinais do DMA que fazem interface com a CPU, a NI e a memória. A Tabela 7 descreve cada um desses sinais.

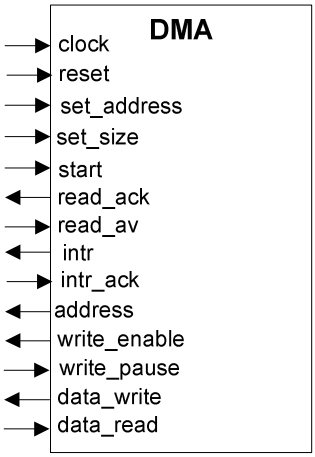


Figura 29 – Sinais do DMA.

Tabela 7 – Descrição dos sinais do DMA.

Sinal	Tipo	Descrição
set_address	Entrada	Indica o endereço da memória a partir do qual o código objeto deve ser transferido (registrador Set_DMA_Address)
set_size	Entrada	Indica o tamanho do código objeto a ser transferido (registrador Set_DMA_Size)
start	Entrada	Indica que a transferência deve ser iniciada (registrador Start_DMA)
read_ack	Saída	Informa a NI que recebeu um dado (um *flit*)
read_av	Entrada	Indica se tem dado disponível para leitura na NI.
intr	Saída	Interrompe o processador quando termina a transferência
intr_ack	Entrada	Indica que o processador já reconheceu a interrupção (registrador DMA_ACK)
address	Saída	Informa à memória o endereço em que o dado deve ser escrito
write_enable	Saída	Informa à memória que deseja escrever
write_pause	Entrada	Indica que não pode escrever na memória.
data_write	Saída	Informa a memória o dado a ser escrito
data_read	Entrada	Dado recebido da NI que vai ser enviado para a memória do processador.

O DMA comunica-se com a NI através dos registradores mapeados em memória que a CPU utiliza para comunicar-se com a NI (Status_Read e Read_Data).

A Figura 30 ilustra a máquina de transição de estados do DMA. O estado inicial é Swait, no qual são conhecidos o endereço da memória a partir do qual o código objeto vai ser transferido e o tamanho do código objeto. Também neste estado a interrupção é desativada, após a CPU informar que reconheceu a interrupção. Se a CPU informa ao DMA que pode iniciar a transferência (start=1), o estado avança para Scopy. Neste estado, cada dado referente ao código objeto é buscado na NI e escrito na memória do processador. A cada escrita, o endereço da memória é incrementado e o tamanho do código objeto é decrementado. Quando o tamanho do código objeto é zero (size=0), o estado avança para Send, onde a escrita na memória é desabilitada (write_enable=0) e a CPU é interrompida (interrupt=1). O estado passa a ser Swait novamente.

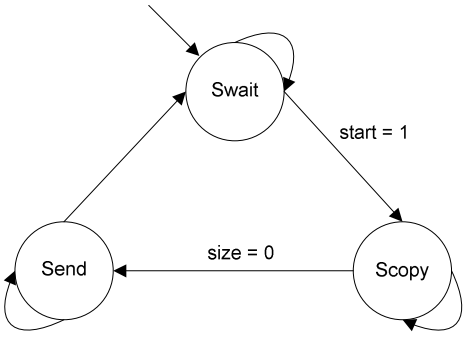


Figura 30 – Máquina de estados do DMA.

5 INFRA-ESTRUTURA DE SOFTWARE

Este Capítulo descreve os serviços do *microkernel* desenvolvido para o gerenciamento de tarefas na plataforma MPSoC. Cada processador escravo componente do sistema possui uma cópia deste *microkernel*. As principais funções do *microkernel* são o suporte à execução de múltiplas tarefas e a comunicação entre tarefas. Este *microkernel* corresponde à segunda contribuição deste trabalho.

Um sistema operacional multitarefa permite que várias tarefas compartilhem o uso de uma mesma CPU, ou seja, várias tarefas são executadas concorrentemente. Essa abordagem requer *(i)* gerenciamento de memória e proteção de memória, uma vez que várias tarefas compartilharão o mesmo espaço de armazenamento; *(ii)* escalonamento, visto que as tarefas irão concorrer pela mesma CPU e; *(iii)* mecanismos de comunicação entre as tarefas [SIL00].

Conforme mencionado no Capítulo 4, cada processador possui uma memória local. Esta memória é dividida em páginas. O *microkernel* reside na primeira página e as tarefas residem nas páginas subseqüentes. Dessa forma, a **gerência de memória** empregada neste trabalho, preocupa-se apenas em determinar a página na qual reside a tarefa que está executando. A geração de endereços físicos é realizada pelo mecanismo de paginação visto na Seção 4.2.

O **escalonamento** de tarefas é preemptivo. O algoritmo utilizado é o Round Robin, no qual as tarefas são escalonadas de forma circular e cada uma delas executa durante uma fatia de tempo (*timeslice*). Sempre que uma tarefa nova deve ser escalonada, o contexto da tarefa que estava executando deve ser salvo. Após o escalonamento, o contexto da nova tarefa é restaurado.

O acesso à memória é caracterizado como NORMA (No Remote Memory Access), ou seja, a memória local de um processador não é acessível a outro. Assim, a **comunicação entre tarefas** ocorre através de troca de mensagens. Para tanto, são utilizados *pipes* de mensagens. Cada tarefa possui um *pipe*, no qual são armazenadas todas as mensagens que esta tarefa envia a outras. Para enviar e receber mensagens, tarefas fazem **chamadas de sistema**.

Quando tarefas localizadas em processadores diferentes desejam se comunicar, é necessária a **comunicação entre processadores**. *Drivers de comunicação* montam e desmontam pacotes contendo as informações das mensagens que são enviadas através da rede de interconexão.

5.1 Estrutura do *microkernel*

A estrutura do *microkernel* multitarefa é mostrada na Figura 31. Ela é composta por diferentes níveis de serviços. No nível mais abaixo (Nível 1) encontra-se o serviço de inicialização do sistema (**Boot**), onde são inicializados os ponteiros para dados globais (gp) e para pilha (sp), a seção de dados estáticos e as estruturas de dados para gerenciamento das tarefas. Após a inicialização, o serviço de boot aciona o escalonador. No nível acima do nível de boot (Nível 2), encontram-se os ***drivers* de comunicação**. O último nível (Nível 3) é composto pelos serviços de

tratamento de interrupções, **escalonamento**, **comunicação entre tarefas** e **chamadas de sistema**.

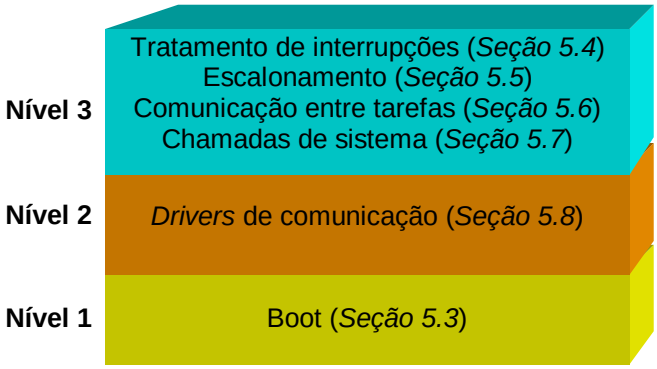


Figura 31 – Estrutura em níveis no *microkernel*.

Os serviços do *microkernel* multitarefa foram implementados parte em linguagem C e parte em linguagem de montagem (*assembly*). Os *drivers* de comunicação e operações de tratamento de interrupção, como salvamento e recuperação de contexto, são implementados em assembly devido ao acesso a registradores facilitado pela linguagem; estruturas de dados e funções como escalonamento, chamadas de sistema e comunicação entre tarefas são implementadas em linguagem C, pois é uma linguagem de alto nível facilitando a programação destas funções.

Conforme mencionado no início deste Capítulo, a gerência de memória empregada aqui preocupa-se apenas em determinar a página onde a execução acontece. Não existe o serviço de alocação dinâmica de memória, pois, além de aumentar o tamanho e complexidade do *microkernel*, este serviço não se faz necessário para atingir o objetivo pretendido neste trabalho.

O tamanho do *microkernel* desenvolvido é 6,6KB, utilizando 41% da página, que possui capacidade de 16KB.

5.2 Estruturas de dados para gerenciamento das tarefas

Antes de abordar os serviços que compõem o *microkernel*, é necessário conhecer as estruturas de dados utilizadas pelo mesmo para o gerenciamento das tarefas.

Para gerenciar a execução das tarefas, o *microkernel* mantém um **TCB** (*Task Control Block*) para cada tarefa, no qual estão contidos, entre outros dados, os valores de 30 registradores do Plasma. A Figura 32 apresenta, em linguagem C, a estrutura `TCB`. Os registradores salvos no TCB são os registradores temporários (`$t0` a `$t9`), registradores salvos (`$s0` a `$s8`), registradores de argumentos (`$a0` a `$a3`), registradores de retorno (`$v0` e `$v1`), endereço de retorno (`$ra`), o ponteiro para a pilha de dados (`$sp`), o ponteiro para dados globais (`$gp`) e os registradores utilizados nas operações de multiplicação e divisão (`$hi` e `$lo`). Além destes registradores, é armazenado, para cada tarefa, o contador de programa (`pc`), o `offset` que indica seu endereço inicial na memória, seu identificador (`id`) e seu `status`. O status de uma tarefa pode ser:

- Pronta (`READY`) – quando está pronta para executar;
- Executando (`RUNNING`) – quando está utilizando a CPU;

- Terminada (`TERMINATED`) – quando terminou sua execução;
- Esperando (`WAITING`) – quando requisita uma mensagem e aguarda a resposta;
- Livre (`FREE`) – quando o TCB da tarefa está livre e pode ser alocado;
- Alocando (`ALLOCATING`) – quando o TCB está sendo alocado.

```
typedef struct{
      unsigned int reg[30];
      unsigned int pc;
      unsigned int offset;
      unsigned int id;
      unsigned int status;
}TCB;
```

Figura 32 – Estrutura de um TCB.

É mantido também um *pipe* global de mensagens. Um *pipe* é uma área de memória destinada à comunicação entre tarefas. Nele ficam armazenadas (até serem consumidas) as mensagens que as tarefas enviam a outras tarefas. Associados ao *pipe*, estão `pipe_order`, que indica a ordem de chegada das mensagens no *pipe* e `pipe_ocupation` que indica quais posições do *pipe* estão ocupadas. O *pipe* possui capacidade para armazenar 10 mensagens.

Uma mensagem (`Message`) armazenada no *pipe* possui um tamanho (`length`), uma tarefa destino (`target`), uma tarefa fonte (`source`) e o conteúdo (`msg`). O conteúdo de uma mensagem possui tamanho máximo de 128 inteiros.

É mantida também, uma tabela chamada `task_location` que estabelece a localização de todas as tarefas do sistema: qual tarefa está localizada em qual processador. Esta tabela é consultada no momento de uma comunicação entre tarefas, conforme será explicado adiante, na Seção 5.6.

O *microkernel* gerencia a execução de até três tarefas: t_1, t_2 e t_3. Esta limitação é imposta apenas no contexto deste trabalho, por restrições de memória, sendo possível parametrizar o código de forma a suportar um número maior de tarefas. Existe ainda, uma tarefa inativa (`idle`). Na prática, a tarefa `idle` é apenas uma função localizada dentro do *microkernel*, que executa um laço infinito. Ela é um artifício utilizado para permitir que o *microkernel* fique aguardando por interrupções provenientes da NoC, quando não tem tarefas a serem escalonadas e executadas. Dessa forma, enquanto tarefa `idle` está executando, as interrupções permanecem habilitadas e o *microkernel* pode receber e enviar pacotes.

É mantido um vetor de TCBs com quatro posições: cada índice do vetor corresponde a uma tarefa. A tarefa `idle` está localizada no último índice do vetor.

5.3 Boot — Inicialização do Sistema Local

Cada processador escravo componente do MPSoC possui uma cópia do *microkernel*. Quando sua execução é iniciada, são atribuídos valores iniciais a algumas variáveis do sistema e

estruturas de dados.

Os registradores `$gp` e `$sp` do Plasma são inicializados, respectivamente, com o ponteiro para dados globais e ponteiro para a pilha referentes ao *microkernel*. Estes valores são conhecidos pelo *microkernel*, para que, sempre que este retome a execução do sistema (ocorrência de interrupções ou chamadas de sistema), ele possa reconfigurar os registradores `$gp` e `$sp`. Isto é necessário, pois antes de o fluxo de execução passar para o *microkernel*, uma tarefa pode estar sendo executada e, assim sendo, os registradores `$gp` e `$sp` possuem, respectivamente, o ponteiro para dados globais e ponteiro para a pilha da tarefa.

Inicialmente, nenhuma tarefa está alocada na CPU. Assim, o *microkernel* inicializa o vetor de TCBs, indicando que estão todos livres (`status=FREE`). A cada TCB é associada uma página da memória, atribuindo-o um `offset`, como mostra a Figura 33. Na primeira página encontra-se o *microkernel*.

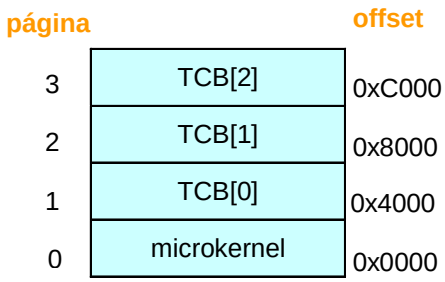


Figura 33 – Configuração da memória.

A tabela de localização de tarefas (`task_location`) é inicializada indicando que todas suas entradas estão livres. A estrutura `pipe_ocupation` também é inicializado indicando que todas as posições do mesmo estão desocupadas.

Os endereços das rotinas que tratam dos diferentes tipos de interrupção são armazenados em um vetor de ponteiros para funções (`ISR`). Estas rotinas são registradas, atribuindo seus respectivos endereços a uma posição deste vetor. As interrupções registradas são interrupções advindas da NoC, de um contador de *timeslice* e do DMA. Por fim, a máscara de interrupções é configurada para conter cada uma dessas interrupções. Este processo é explicado na próxima Subseção.

Após as inicializações, a tarefa `idle` é escalonada. Enquanto esta tarefa está em execução, o sistema aguarda por uma interrupção da NoC (chegada de pacotes) ou do DMA (nova tarefa na memória).

5.4 Tratamento de Interrupções

Um processador componente do MPSoC pode ser interrompido via *hardware* ou via *software*. Assim, uma interrupção pode ser proveniente:

1. da **NI**, para que ele receba pacotes advindos da NoC (*interrupção de hardware*);

2. do **contador de *timeslice***, informando que o *timeslice* de uma tarefa acabou e uma nova tarefa deve ser escalonada. *Timeslice* é uma fatia de tempo durante a qual uma tarefa é executada (*interrupção de hardware*);
3. do **DMA**, informando que o código objeto de uma nova tarefa já se encontra em sua memória local e que esta pode ser executada (*interrupção de hardware*);
4. da **aplicação**, através de primitivas que geram **chamadas de sistema** (*interrupção de software*).

No caso das interrupções de hardware, o processador mantém um registrador contendo a máscara das interrupções (`irq_mask_reg`) e outro registrador contendo o status das interrupções (`irq_status`), ou seja, quais as interrupções ativas. O registrador de *status* é composto pelos bits mostrados na Figura 34.

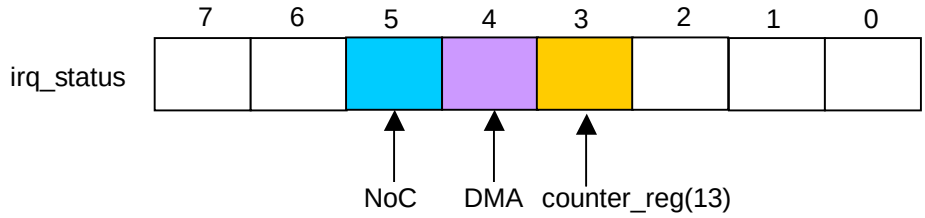


Figura 34 – Composição do registrador de *status* das interrupções.

Assim, cada fonte de interrupção de hardware contém uma máscara conhecida pelo *microkernel*. Essas fontes e suas respectivas máscaras são mostradas abaixo:

```
#define IRQ_COUNTER   0x08    (Máscara=00001000)
#define IRQ_DMA       0x10    (Máscara=00010000)
#define IRQ_NOC       0x20    (Máscara=00100000)
```

Na inicialização do sistema, o *microkernel* registra uma rotina de tratamento de interrupção para cada tipo de interrupção:

```
OS_InterruptRegister(IRQ_COUNTER, Scheduler);
OS_InterruptRegister(IRQ_NOC, DRV_Handler);
OS_InterruptRegister(IRQ_DMA, DMA_Handler);
```

Registrar significa armazenar em um vetor de ponteiros para função (`ISR`) o endereço da rotina responsável pelo tratamento de determinada interrupção, associando a posição neste vetor com a máscara. Assim, quando ocorrer uma interrupção cuja fonte é o contador de *timeslice*, por exemplo, e cuja máscara é definida por `IRQ_COUNTER`, a função `Scheduler` entrará em execução.

Após associar as rotinas de tratamento de interrupção, a máscara de interrupções é configurada, habilitando as três interrupções:

```
OS_InterruptMaskSet(IRQ_COUNTER18 | IRQ_NOC | IRQ_DMA);
```

Com isso, a máscara possui o valor `00111000`. O processador é interrompido quando a operação `AND` entre a máscara de interrupções e o *status* das interrupções possui um resultado diferente de `ZERO`. Quando isso ocorre, o fluxo de execução salta para um endereço fixo: o endereço `0x3C`, onde o tratamento é iniciado. Enquanto uma interrupção é tratada, não podem ocorrer novas interrupções. Assim, as interrupções são desabilitadas (via hardware) no início do tratamento e realibilitadas ao término (via software).

O tratamento de interrupções pode ser mais bem explicado em uma seqüência de passos, conforme a seguir.

1. O primeiro passo do tratamento de uma interrupção é o salvamento de contexto da tarefa que estava sendo executada. O *microkernel* mantém em uma variável global, `current`, o endereço do TCB da tarefa vigente. Neste TCB são salvos os registradores e o `pc` da tarefa.

2. Antes da interrupção, os registradores `$sp` e `$gp` do processador são referentes à tarefa em execução. Dessa forma, eles são configurados com valores referentes ao *microkernel*;

3. É chamada uma rotina (`OS_InterruptServiceRoutine`) responsável por verificar qual a causa da interrupção e, conseqüentemente, chamar a função designada a tratar a interrupção;

4. Após a interrupção tratada, o contexto da tarefa é restaurado. Se a interrupção foi causada pelo contador de *timeslice*, uma nova tarefa foi escalonada e começa a ser executada. Senão, a tarefa que estava executando antes de interrupção retoma sua execução.

A Figura 35 mostra uma interrupção advinda do contador de *timeslice*. Os sinais `irq_mask_reg` e `irq_status` indicam, respectivamente, a máscara de interrupções (quais interrupções estão sendo esperadas) e o *status* das interrupções (quais interrupções estão ativas). O sinal `intr_enable` indica quando as interrupções estão habilitadas. O sinal `intr_signal` indica quando o processador foi interrompido e `page` indica a página da memória em que se encontra o fluxo de execução. As legendas da Figura são explicadas a seguir.

1. O processador executa a tarefa que está na página `1` (`page=1`);

2. O registrador de *status* tem o seu 4º bit configurado em `1`;

3. A operação `AND` entre a máscara de interrupções e o registrador de *status* resulta em um valor diferente de zero (`00001000`);

4. Uma interrupção é gerada e novas interrupções são desabilitadas;

5. O *microkernel* entra em execução (`page=0`), salva o contexto da tarefa suspendida e verifica que a interrupção foi causada pelo contador de *timeslice*;

6. A rotina chamada para tratar essa interrupção é a rotina `Scheduler`, que faz o escalonamento de uma nova tarefa: a tarefa que se encontra na página `2` (`page=2`).

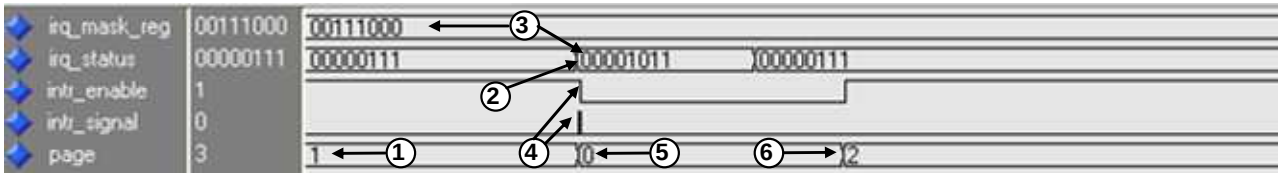


Figura 35 – Interrupção advinda do contador de *timeslice*.

Uma interrupção advinda da NoC é mostrada na Figura 36. As legendas da Figura são explicadas a seguir.

1. O processador executa a tarefa que está na página `3` (`page=3`);
2. O registrador de *status* tem o seu 6º bit configurado em `1`;
3. A operação `AND` entre a máscara de interrupções e o registrador de *status* resulta em um valor diferente de zero (`00100000`);
4. Uma interrupção é gerada e novas interrupções são desabilitadas;
5. O *microkernel* entra em execução (`page=0`), salva o contexto da tarefa suspendida e verifica que a interrupção foi causada pela chegada de pacotes pela NoC. A rotina chamada para tratar essa interrupção é a `DRV_Handler`, que trata o pacote;
6. O contexto da tarefa que foi suspendida é restaurado e ela volta a executar (`page=3`).

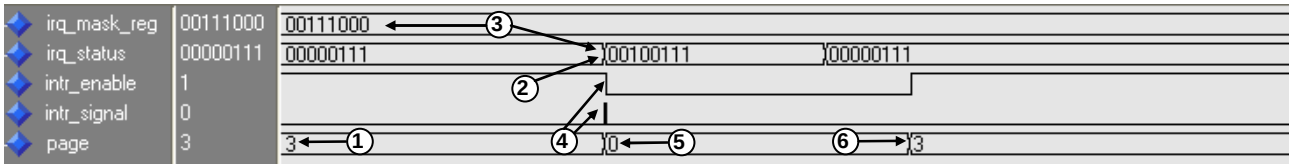


Figura 36 – Interrupção advinda da chegada de pacotes da NoC.

Da mesma forma que acontece nas interrupções de hardware, nas interrupções de software, o fluxo de execução também salta para um endereço fixo: o endereço `0x44` e tem as seguintes operações:

1. Salvamento parcial de contexto, ou seja, apenas alguns registradores são salvos: registradores de argumento (`$a0` a `$a4`), endereço de retorno (`$ra`), ponteiro para dados globais (`$gp`) e ponteiro para a pilha (`$sp`);
2. Configuração do `$sp` e `$gp` do *microkernel*;
3. Chamada da função `Syscall`, que trata a chamada de sistema;
4. Recuperação do contexto parcial da tarefa que causou a interrupção e retomada de sua execução;

O serviço de chamadas de sistema é abordado mais adiante, na Subseção 5.7.

5.5 Escalonamento

O escalonamento utilizado pelo *microkernel* é preemptivo e sem prioridades. A política de escalonamento é Round Robin [SIL00][TAN97], na qual as tarefas são escalonadas de maneira circular. A função de escalonamento é mostrada na Figura 37.

```
1 int Scheduler(){
2     int scheduled = 0;
3     int i;
4
5     if (current->status == RUNNING)
6           current->status = READY;
7
8     for (i=0; i<MAXLOCALTASKS; i++){
9           if (roundRobin == MAXLOCALTASKS-1)
10                roundRobin = 0;
11          else
12                roundRobin++;
13          current = &(tcbs[roundRobin]);
14          switch(current->status){
15                case READY:
16                      scheduled = 1;
17                      break;
18                default:
19                      break;
20          }
21          if (scheduled == 1){
22                current->status = RUNNING;
23
24                OS_InterruptMaskSet(IRQ_COUNTER);
25                MemoryWrite(COUNTER_REG, 0);
26                return 1;
27          }
28    }
29    current = &tcbs[MAXLOCALTASKS];
30    OS_InterruptMaskClear(IRQ_COUNTER18);
31
32    return 0;
33 }
```

Figura 37 – Função de escalonamento.

O *microkernel* mantém nas variáveis globais `current` e `rounRobin`, o endereço do TCB da tarefa que está executando e o índice do vetor de TCBs correspondente a esta tarefa, respectivamente. O `status` da tarefa que foi interrompida passa a ser `READY` (linha 6). O escalonador procura a próxima tarefa a ser executada: se `rounRobin` indica que a tarefa que estava executando é a última do vetor[1], então a tarefa a executar deve ser a primeira (linhas 9 e 10); senão, deve ser a subseqüente (linhas 11 e 12). A variável `current` passa a conter o endereço da tarefa escolhida (linha 13). Se ela está pronta para executar (`status=READY`) ela é escalonada (linhas 15 a 17). Depois da tarefa ser escalonada, seu status passa a ser `RUNNING`, as interrupções do contador de *timeslice* são habilitadas e este é reinicializado (linhas 21 a 26). Se nenhuma tarefa

[1] Neste caso, a última tarefa do vetor significa a tarefa que está no índice MAXLOCALTASKS-1, a qual corresponde a uma tarefa da aplicação. Na prática, a que última tarefa do vetor (a que se encontra no índice MAXLOCALTASKS), é a tarefa `idle`, que só é escalonada quando nenhuma tarefa da aplicação está pronta para executar.

da aplicação pôde ser escalonada, então, a tarefa `idle` entra em execução e interrupções advindas do contador de *timeslice* são desabilitadas (linhas 29 e 30).

A Figura 38 ilustra o escalonamento circular de três tarefas: t_1 que se encontra na página `1`, t_2 na página `2` e t_3 na página `3`.

1. A tarefa t_1 (`page=1`) executa durante seu *timeslice*;
2. Uma interrupção é gerada, novas interrupções são desabilitadas e o *microkernel* entra em execução (`page=0`);
3. O contexto da tarefa que foi suspendida, t_1, é salvo e o escalonador escalona a próxima tarefa: t_2. O contexto de t_2 é carregado e o fluxo de execução salta para a página `2`. As interrupções são habilitadas;
4. A tarefa t_2 executa durante seu *timeslice*. Uma interrupção é gerada, novas interrupções são desabilitadas e o *microkernel* entra em execução (`page=0`);
5. O contexto da tarefa que foi suspendida, t_2, é salvo e o escalonador escalona a próxima tarefa: t_3. O contexto de t_3 é carregado e o fluxo de execução salta para a página `3`. As interrupções são habilitadas.

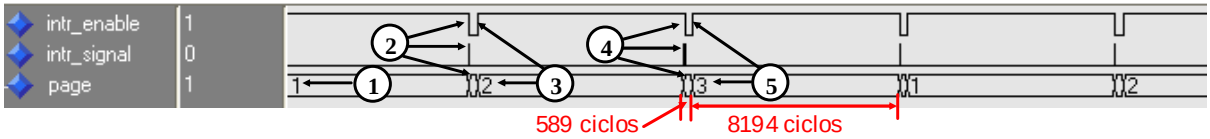


Figura 38 – Escalonamento de tarefas.

Uma troca de contexto totaliza 589 ciclos e uma tarefa executa durante 8194 ciclos (163,8 µs, para um relógio de 50MHZ). Sistemas operacionais em geral, utilizam um *timeslice* 10 vezes maior do que a troca de contexto [TAN97]. Aqui, o *timeslice* é parametrizável e é aproximadamente 14 vezes maior do que o tempo de troca de contexto, pois tem-se por objetivo reduzir a sobrecarga de processamento por parte do *microkernel*. A Tabela 8 mostra o tempo gasto para cada operação da troca de contexto: salvamento de contexto, escalonamento e recuperação de contexto.

Tabela 8 – Tempo gasto para as operações de troca de contexto.

Operação	Número de ciclos	Tempo (relógio=50MHZ)
Salvamento de contexto	90	1,8µs
Escalonamento	419	8,3µs
Recuperação de contexto	80	1,6µs
Total	**589**	**11,7µs**

Há casos em que o escalonador deve entrar em execução sem que ocorra uma interrupção do contador de *timeslice*. Isto acontece quando:

1. Uma tarefa termina sua execução (Seção 5.7);
2. Uma tarefa tenta receber uma mensagem que está em outro processador, gerando um pacote de requisição e aguardando sua resposta (Seção 5.6);
3. A tarefa `idle` está executando e uma nova tarefa é alocada na memória do processador (Seção 6.2).
4. A tarefa `idle` está executando e foi recebida a resposta de uma requisição de mensagem (Seção 5.6).

Nestes quatro casos, uma variável global, `needTaskScheduling`, é configurada para `1` indicando que uma nova tarefa deve ser escalonada. Esta variável é sempre configurada para `0` antes do tratamento de uma interrupção (de hardware ou software) e verificado após do tratamento, pois é apenas durante o tratamento que ela é configurada para `1`, ou seja, apenas um evento que causou uma interrupção pode ocasionar um escalonamento.

5.6 Comunicação Entre Tarefas

Tarefas podem cooperar em tempo de execução, trocando informações entre si. A comunicação entre tarefas ocorre através de *pipes*. Segundo Tanenbaun [TAN97], um ***pipe*** é um canal de comunicação no qual mensagens são consumidas na ordem em que são armazenadas no mesmo. No contexto deste trabalho, um *pipe* é uma área de memória pertencente ao *microkernel* reservada para troca de mensagens entre tarefas, onde as mensagens são armazenadas de forma ordenada e consumidas de acordo com a ordem.

A Figura 39 apresenta três formas de implementar a comunicação entre tarefas através de *pipes*. Na primeira (Figura 39(a)), é mantido um *pipe* único, no qual são armazenadas **todas as mensagens de tarefas locais enviadas para quaisquer tarefas**. Na segunda (Figura 39(b)), cada tarefa possui um *pipe* exclusivo associado a ela, no qual ficam armazenadas **todas as mensagens recebidas de outras tarefas**. Dessa forma, se a tarefa t_3 enviar uma mensagem (`msg`) para t_2, a mensagem será armazenada no *pipe* de t_2 (`pipe2`). Quando t_2 desejar receber esta mensagem, ela será buscada no `pipe2`. Assim como na segunda forma de implementação da comunicação, na terceira (Figura 39(c)), cada tarefa possui um *pipe* exclusivo associado a ela. Contudo, neste *pipe* ficam armazenadas **as mensagens que ela envia a outras tarefas**. Assim, se a tarefa t_3 enviar uma mensagem (`msg`) para t_2, a mensagem será armazenada no *pipe* de t_3 (`pipe3`). Quando t_2 desejar receber esta mensagem, ela será buscada no `pipe3`.

A abordagem utilizada neste trabalho é a de *pipe* global. Contudo, antes disso, foram implementadas e testadas as outras duas abordagens, nas quais foram encontrados problemas conforme descrito a seguir.

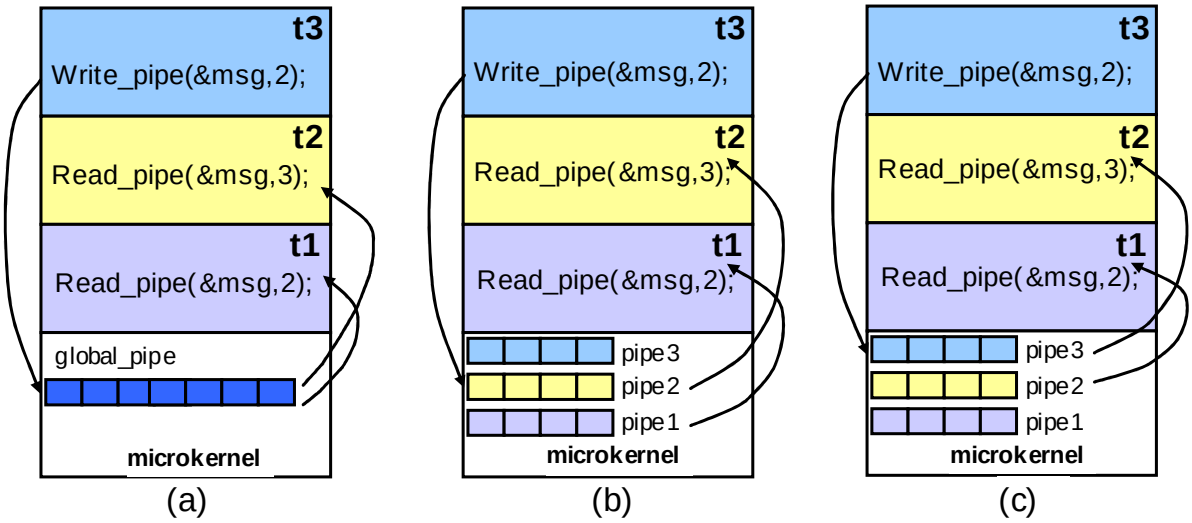


Figura 39 – Formas de implementação de *pipes*: (a) um *pipe* único para todas as tarefas; (b) um *pipe* exclusivo para mensagens recebidas; (c) um *pipe* exclusivo para mensagens enviadas.

Na abordagem da Figura 39(b), se o *pipe* de uma tarefa estiver cheio, o processador não pode mais receber mensagens e a rede pode ficar bloqueada. A Figura 40 ilustra esta situação. Suponha que duas tarefas t_1 e t_2, respectivamente em P_8 e P_4 enviem, cada uma, uma mensagem à t_3 em P_6. A tarefa t_3 aguarda as mensagens nesta ordem: primeiro de t_1 e depois de t_2. Suponha então, que a mensagem de t_2 chegue primeiro (Figura 40(a)) e o *pipe* de t_3 fique cheio. Quando a mensagem de t_1 chegar, não vai haver espaço no *pipe* (Figura 40(b)) e, uma vez que t_3 não consegue consumir a mensagem esperada (mensagem de t_1), a rede vai ficar bloqueada. Assim, esta forma de comunicação torna o sistema suscetível a situações de bloqueio facilmente.

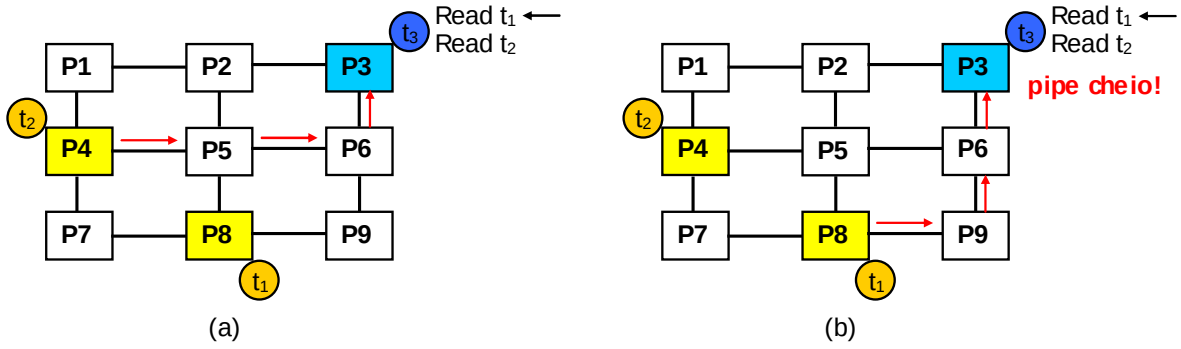


Figura 40 – Situação de bloqueio na comunicação entre tarefas com escrita de mensagens no *pipe* da tarefa destino.

Este problema de bloqueio pode ser resolvido com a abordagem da Figura 39(c), no qual uma mensagem é escrita no *pipe* da tarefa fonte e só enviada através da rede mediante requisição pela tarefa destino. Quando a mensagem requisitada não está disponível, é enviado um pacote de controle indicando que não existe mensagem ainda. A desvantagem desta abordagem é o uso não otimizado da área de memória dos *pipes*, no processo de escrita de mensagens: uma tarefa pode ser altamente cooperante escrevendo muitas mensagens, enquanto outra raramente escreve mensagens.

Dessa forma, o *pipe* global otimiza o uso da área de memória destinado à comunicação entre tarefas e evita também situações de bloqueio na rede, uma vez que, da mesma forma que na abordagem da Figura 39(c), as mensagens são enviadas através da rede mediante requisições.

As tarefas se comunicam através de duas primitivas. Para o envio de uma mensagem, uma tarefa utiliza a primitiva:

```
WritePipe(&mensagem, id_destino)
```

onde `&mensagem` especifica o endereço lógico (dentro da página onde está a tarefa) em que está armazenada a mensagem e `id_destino` é o identificador da tarefa para a qual a mensagem está sendo enviada.

Para o recebimento de uma mensagem é utilizada a primitiva:

```
ReadPipe(&mensagem, id_fonte)
```

onde `&mensagem` especifica o endereço lógico (dentro da página onde está a tarefa) em que a mensagem será armazenada e `id_fonte` é o identificador da tarefa que enviou a mensagem.

A comunicação pode acontecer entre tarefas que residem em processadores diferentes, como mostra a Figura 41. Quando uma tarefa t_5, no processador `Proc2`, deseja receber uma mensagem de uma tarefa t_2, o *microkernel* verifica qual a localização de t_2 (`task_location`, citado na Seção 5.2). Se t_2 encontra-se no processador local, o *microkernel* copia a mensagem do *pipe* para a página de t_5, como mostra a Figura 39(a). Neste exemplo, t_2 encontra-se em um processador remoto, `Proc1`. Dessa forma, a comunicação ocorre em uma seqüência de passos:

1. O *microkernel* em `Proc2` monta um pacote de requisição (`request_msg`) e envia a `Proc1`, requisitando uma mensagem de t_2 para t_5 (Figura 41(a));

2. A tarefa t_5 é colocada em espera (`status=WAITING`) e uma nova tarefa é escalonada em `Proc2`;

3. O *microkernel* em `Proc1` recebe a requisição e verifica no *pipe* se existe uma mensagem para t_5. Se sim, o pacote contendo as informações e o conteúdo da mensagem é enviado a `Proc2`, como mostra a Figura 41(b). Se não, é enviado um pacote informando que não há mensagem (`no_message`);

4. Quando `Proc2` recebe a resposta da requisição, t_5 passa a ter status `READY` e pode ser novamente escalonada para continuar sua execução. Se a resposta contiver a mensagem esperada, ela é copiada para o endereço especificado por `&msg`, na primitiva de comunicação. Este endereço é localizado dentro da página da tarefa. Se a resposta for `no_message`, t_5 pode tentar novamente receber a mensagem. Para isso, o recebimento da mensagem, na aplicação, deve ser implementado em um laço: `while(!ReadPipe(&msg,2));`

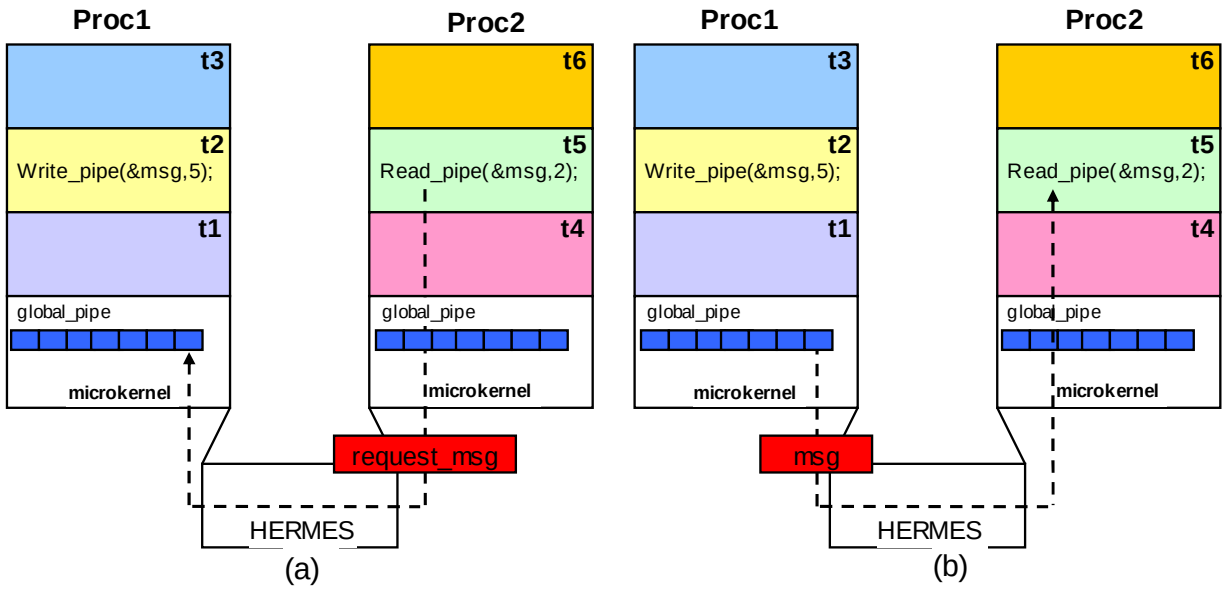


Figura 41 – Comunicação entre tarefas de diferentes processadores. O *microkernel* monta um pacote contendo as informações da mensagem e o envia através da rede.

É importante salientar que uma tarefa entra em estado de espera apenas quando ela faz `ReadPipe` de uma mensagem cuja tarefa está em um processador remoto. Ou seja, no momento em que ela requisita uma mensagem e deve aguardar o pacote de resposta. Caso contrário, isto é, quando a mensagem é de uma tarefa local, não é preciso aguardar resposta. A primitiva chamada retorna o resultado da operação e a tarefa decide se quer tentar novamente executar `ReadPipe`.

Assim como a primitiva `ReadPipe` pode não ser concluída com sucesso, ou seja, quando a mensagem esperada não está disponível, o envio de mensagens também pode falhar. Isso acontece quando o *pipe* está cheio e, portanto, não há mais espaço para novas mensagens. Dessa forma, o envio de uma mensagem também pode ser implementado dentro de um laço: `while(!WritePipe(&msg,5));`, garantindo que, em algum momento, a mensagem vai ser escrita no *pipe* para posterior leitura.

As mensagens são ordenadas no momento em que são armazenadas nos *pipes*. Para cada mensagem é associado um número inteiro indicando sua ordem (para isso é utilizado o vetor `pipe_order` (visto na Seção 5.2). No envio de uma mensagem, o *pipe* é percorrido verificando se já existem mensagens da tarefa fonte para a tarefa destino. Se afirmativo, é verificado qual a maior ordem existente. A nova mensagem é armazenada no *pipe* indicando em `pipe_order` que sua ordem é a maior ordem encontrada mais 1. Quando uma tarefa desejar receber uma mensagem, a mensagem repassada a ela será a que tiver menor ordem.

As primitivas de comunicação entre tarefas (`WritePipe` e `ReadPipe`) ocasionam chamadas de sistema, de forma que o *microkernel* assume o controle, gerenciando a leitura e escrita nos *pipes*, bem como a leitura e escrita de mensagens em endereços de memória de diferentes páginas. As chamadas de sistema são tratadas a seguir.

5.7 Chamadas de Sistema

Uma chamada de sistema é uma interrupção gerada pelo software com intuito de requisitar um serviço do sistema operacional [SIL00]. Diversos sistemas operacionais possuem chamadas de sistema para diferentes propósitos, entre eles, gerenciamento de processos, gerenciamento de arquivos, proteção, gerenciamento de tempo, realizar operações de E/S, entre outros [TAN97]. Neste trabalho, chamadas de sistemas são utilizadas para realizar a comunicação entre tarefas, para terminar a execução de uma tarefa e para informar a uma tarefa o valor do contador de ciclos de relógio (`Tick_Counter`).

Na prática, as primitivas de comunicação `WritePipe` e `ReadPipe` são definidas como sendo uma função denominada `SystemCall`, que recebe como primeiro parâmetro um inteiro especificando o serviço (`1` para `WritePipe` e `2 para ReadPipe`). Além desse primeiro parâmetro, ela recebe os outros dois argumentos advindos da primitiva, como mostra a Figura 42. A primitiva `GetTick` é também definida como sendo a função `SystemCall` com o parâmetro `3`.

```
#define WritePipe(msg, target) SystemCall(1,(unsigned int*)msg,target)

#define ReadPipe(msg, source) SystemCall(2,(unsigned int*)msg,source)

#define GetTick() SystemCall(3)
```

Figura 42 – Definições das primitivas de comunicação através da função SystemCall.

A Figura 43 mostra a função `SystemCall`. Ela é implementada em assembly e gera a chamada de sistema através da instrução `syscall` (linha 5), logo após retornando à execução da tarefa (linha 7).

```
1   globl SystemCall
2   .ent SystemCall
3  SystemCall:
4    .set noreorder
5    syscall
6    nop
7    jr $31
8    nop
9    .set reorder
10  .end SystemCall
```

Figura 43 – Função `SystemCall` em Assembly gera a interrupção de software através da instrução `syscall`.

Conforme já visto na Seção 4.2, a instrução `syscall` foi acrescentada ao conjunto de instruções do processador Plasma, pois na sua distribuição original, esta instrução não era reconhecida. Também, foi visto na Seção 5.4 que, quando a aplicação executa uma primitiva que gera uma chamada de sistema: (*i*) o contexto da tarefa é salvo parcialmente; (*ii*) é chamada uma função, em C, denominada `Syscall` responsável por realizar a chamada de sistema retornando o resultado da operação para a tarefa; (*iii*) o contexto parcial da tarefa é recuperado e; (*iv*) a tarefa retoma sua execução. A rotina implementada em C, responsável por tratar a chamada de sistema é

mostrada na Figura 44.

```
1 int Syscall(int service, unsigned int* msg, int taskID){
2       switch(service){
3               case WRITEPIPE:
4                       return WritePipe(msg, taskID);
5                       break;
6               case READPIPE:
7                       return ReadPipe(msg, taskID);
8                       break;
9               case EXIT:
10                      (explicado na Seção 6.1.3)
11                      break;
12              case GETTICK:
13                      return MemoryRead(TICK_COUNTER);
14                      break;
15      }
16      return 0;
17 }
```

Figura 44 – Função em C que trata uma chamada de sistema.

Uma chamada de sistema espera 3 argumentos (linha 1). O primeiro indica o serviço desejado: `1` para `WritePipe`; `2` para `ReadPipe`; `0` para `Exit` e `3` para `GetTick`. `Exit` não é uma primitiva utilizada pela tarefa. Na prática, quando a tarefa executa um `return [value]`, uma chamada de sistema é gerada passando como argumento o valor `0`. O procedimento tomado para este serviço é discutido na Seção 6.1.3.

Se o serviço desejado for `GetTick`, dois argumentos seguintes são ignorados e o valor do relógio é informado à tarefa (linhas 12 a 14). Se o serviço solicitado for `WritePipe`, os dois parâmetros seguintes possuem, respectivamente, o endereço lógico no qual se encontra a mensagem e o identificador da tarefa destino (linhas 3 a 5). Se for `ReadPipe`, os dois parâmetros seguintes possuem, respectivamente, o endereço lógico para onde a mensagem vai ser copiada e o identificador da tarefa fonte (linhas 6 a 8). Nestes dois últimos serviços é retornado `0` à tarefa para indicar que a operação não pôde ser efetuada e `1` para indicar que foi bem sucedida.

5.8 *Drivers* de Comunicação

Conforme visto na Seção 4.3 os *drivers* fazem o envio de pacotes para a NoC e o recebimento de pacotes da NoC. Foram vistos também os serviços que um pacote carrega e o formato do pacote para cada tipo de serviço. Para cada serviço, existe um tratamento. Dessa forma, a Figura 45 apresenta a função que é chamada quando acontece uma interrupção proveniente da chegada de pacotes pela NoC. Esta função faz chamadas aos *drivers* de comunicação necessários para o tratamento dos pacotes. Todos os *drivers* são implementados em assembly.

Nesta seção são discutidos apenas os *drivers* de comunicação para os serviços de troca de mensagens. Os *drivers* de comunicação para os serviços de alocação de tarefas são objetos de discussão do Capítulo 6.

```
1 void DRV_Handler() {
2     int service, size, i;
3
4     DRV_ReadService(&service);
5
6     switch (service)
7     {
8           case REQUEST_MESSAGE:
9                 DRV_DeliverMessage(netAddress);
10                break;
11          case DELIVER_MESSAGE:
12                DRV_ReadMessage();
13                if (current == &tcbs[MAXLOCALTASKS]);
14                      needTaskScheduling = 1;
15                break;
16          case NO_MESSAGE:
17                DRV_NoMessage();
18                if (current == &tcbs[MAXLOCALTASKS]);
19                      needTaskScheduling = 1;
20                break;
21          case TASK_ALLOCATION:
22                ...
23                break;
24          case ALLOCATED_TASK:
25                ...
26          case DEALLOCATED_TASK:
27                ...
28          case FINISHED_ALLOCATION:
29                ...
30    }
31 }
```

Capítulo 6

Figura 45 – Função `DRV_Handler` que faz o tratamento de interrupções da NoC.

Quando chega um pacote pela NoC, a primeira informação a se saber é o serviço que este pacote carrega. Assim, é chamado um *driver* (`DRV_ReadService`) que lê da NI o serviço (linha 4). Se o serviço for `REQUEST_MESSAGE`, é chamado o *driver* `DRV_DeliverMessage`, responsável por fazer a devolução da mensagem requisitada, passando o endereço de rede do processador local (`netAddress`) como parâmetro para a montagem do pacote (linha 9). Este *driver* lê da NI o processador fonte, a tarefa destino e a tarefa fonte da mensagem. Com base nestas duas últimas informações, a mensagem é procurada no *pipe*. Se ela existe, o pacote com a mensagem é montado como segue:

```
<target_processor><size><service=DELIVER_MESSAGE>
<source_processor=netAddress><message_target><message_source>
```

Se a mensagem não existe, é enviado um pacote com o serviço NO_MESSAGE, indicando que a mensagem não existe. O pacote é montado da seguinte forma:

```
<target_processor><size=8><service=NO_MESSAGE>
<source_processor=netAddress><message_target><message_source>
```

Se o serviço que o pacote carrega é `DELIVER_MESSAGE`, é chamado o *driver* `DRV_ReadMessage` (linha 12), responsável por ler a mensagem e transferí-la para a aplicação. Este *driver* lê da NI o processador fonte, a tarefa destino e a tarefa fonte da mensagem. O endereço

da memória para onde a mensagem deve ser copiada é buscado no TCB da tarefa: registrador `$a1`, que contém o primeiro parâmetro da chamada de sistema que a tarefa realizou (`ReadPipe(&msg, source_id)`). Este endereço encontra-se na página onde a tarefa que fez a chamada `ReadPipe` está alocada. Então, o *driver* lê da NI o tamanho e o conteúdo da mensagem copiando estas informações para o endereço especificado pela tarefa. O *driver* retorna `1` para a tarefa indicando que a chamada `ReadPipe` foi concluída com sucesso e o `status` da tarefa é configurado para `READY`.

Uma tarefa que faz uma chamada `ReadPipe` requisitando uma mensagem que está em um processador remoto é colocado em estado de espera (`status=waiting`), não sendo mais escalonada até receber a resposta da requisição. Dessa forma, após receber a resposta, é verificado se a tarefa que estava executando no momento da chegada desta resposta era a tarefa `idle` (linha 13). Se sim, a variável `needTaskScheduling` indica que é preciso escalonar uma nova tarefa (linha 14).

Se o serviço for `NO_MESSAGE`, é chamado o *driver* `DRV_NoMessage` (linha 16). Este *driver* lê da NI o processador fonte, a tarefa destino e a tarefa fonte da mensagem. O *driver* retorna `0` para a tarefa indicando que a chamada `ReadPipe` não foi concluída com êxito e o `status` da tarefa passa a ser `READY`. O serviço `NO_MESSAGE`, assim como o `DELIVER_MESSAGE` é uma resposta do serviço `REQUEST_MESSAGE`. Este, por sua vez, gerado pela chamada `ReadPipe` (com mensagem remota). Desta forma, assim como acontece com `DELIVER_MESSAGE`, após receber a resposta, é verificado se a tarefa que estava executando antes de chegar esta resposta é a tarefa `idle` (linha 18). Se sim, a variável `needTaskScheduling` indica que é preciso escalonar uma nova tarefa (linha 19).

6 ALOCAÇÃO DE TAREFAS

Este Capítulo apresenta a estratégia de alocação de tarefas implementada neste trabalho. Foram desenvolvidas duas estratégias de alocação: alocação estática e alocação dinâmica. Este Capítulo apresenta inicialmente o nodo mestre, o qual implementa o processo de alocação. O nodo mestre possui uma interface com um repositório de tarefas, responsável pelo armazenamento do código executável de todas as tarefas que devem ser executadas no sistema. A Seção 6.2 apresenta como as tarefas são recebidas e alocadas nos nodos escravos. A arquitetura é apresentada de forma simplificada na Figura 46.

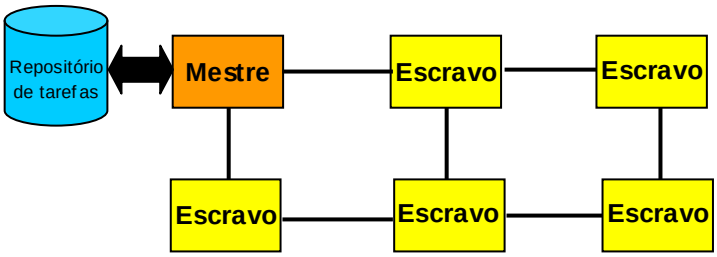


Figura 46 – MPSoC com um nodo mestre.

6.1 Nodo Mestre

Diferentemente dos nodos escravos, o nodo mestre executa apenas a aplicação de alocação. A *aplicação de alocação* de tarefas compreende estruturas de dados necessárias para gerenciar a alocação, *drivers* de comunicação com escravos e tratamento de interrupções provenientes da NoC. O mestre possui ainda uma interface com um repositório de tarefas, o qual é uma memória externa com códigos objetos de tarefas a serem alocadas no MPSoC.

Trabalhos futuros compreendem tornar a aplicação de alocação uma tarefa genérica, permitindo que ela compartilhe tempo de execução com as demais aplicações do sistema.

6.1.1 Repositório de tarefas

O repositório de tarefas fisicamente é uma memória de grande capacidade, externa ao sistema MPSoC. Ele está conectado diretamente ao mestre, e não à rede, visando reduzir o tráfego na rede e aumentar o desempenho do sistema. A interface entre o processador mestre e o repositório de tarefas é composta pelos sinais `address`, que indica o endereço de leitura na memória e `data_read`, que indica o dado lido. Conforme mostra a Figura 17 (página 46), no Plasma, a memória externa possui endereçamento de `0x10000000` a `0x1fffffff`.

A Figura 47 mostra a estrutura do repositório de tarefas, com duas tarefas (t_1 e t_2). Para cada tarefa armazenada no repositório são conhecidas as seguintes informações: identificador (`id`), tamanho do código objeto da tarefa (`size`) e endereço inicial do código objeto

(initial_address). Estas informações estão nos primeiros endereços do repositório, constituindo um cabeçalho. A partir deste cabeçalho encontram-se os códigos objetos das tarefas. O número de tarefas presentes no repositório é uma informação contida na aplicação de alocação.

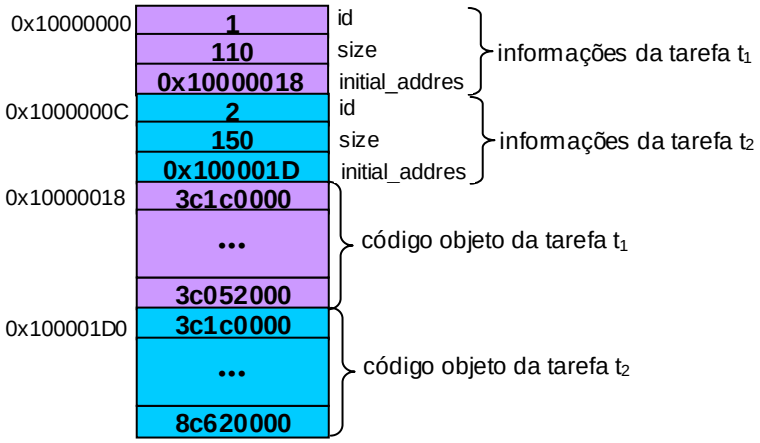


Figura 47 – Estrutura do repositório de tarefas.

6.1.2 Alocação estática

A plataforma MPSoC implementada possui 5 processadores escravos, como mostra a Figura 46. O nodo mestre mantém em um vetor, denominado slaves, os endereços destes processadores escravos na rede, como mostrado na Tabela 9.

Tabela 9 – Endereços de rede para cada escravo.

Escravo	Endereço na rede
SLAVE1	0x00000000
SLAVE2	0x00000010
SLAVE3	0x00000020
SLAVE4	0x00000011
SLAVE5	0x00000021

Associado ao vetor slaves, utiliza-se uma estrutura denominada free_pages, a qual indica quantas páginas livres cada escravo possui. Inicialmente, todos têm 3 páginas livres. Sempre que uma tarefa for alocada em um escravo S, este tem o número de páginas livres decrementada.

Uma tabela, static_allocation, contém as informações de alocação estática. Nela, os campos task e slave indicam, respectivamente, que a tarefa task deve ser alocada no processador slave. A tabela é criada como mostra a Figura 48. As chamadas à função InsertStaticAllocation inserem três entradas na tabela indicando, respectivamente, que a tarefa com identificador 1 deve ser alocada no escravo SLAVE1 (linha 1); a tarefa com identificador 2 deve ser alocada no escravo SLAVE2 (linha 4) e; a tarefa com identificador 3 deve ser alocada no escravo SLAVE3 (linha 7). Após atribuir uma tarefa a um escravo S, a função OccupiedPage, decrementa o número de páginas livres de S (linhas 2, 5 e 8).

```
1 InsertStaticAllocation(1, SLAVE1);
2 OccupiedPage(SLAVE1);
3
4 InsertStaticAllocation (2, SLAVE2);
5 OccupiedPage(SLAVE2);
6
7 InsertStaticAllocation (3, SLAVE3);
8 OccupiedPage(SLAVE3);
```

Figura 48 – Criação da tabela de alocação estática, `static_allocation`.

A definição das tarefas iniciais do sistema requer a compilação da *aplicação de alocação* para cada nova aplicação. Esta recompilação pode ser evitada se a lista de tarefas iniciais também for incluída no repositório de tarefas.

A função de alocação estática, `TasksAllocation`, é mostrada na Figura 49. O mestre utiliza um ponteiro para uma estrutura `TaskPackage` (linha 3) de forma a percorrer o cabeçalho de informações das tarefas no repositório. `TaskPackage` possui a mesma estrutura de cabeçalho utilizada no repositório: `id`, `size` e `initial_address`. De forma a acessar a memória externa, é atribuído a este ponteiro o endereço `0x10000000` (linha 4).

A tabela `static_allocation` é percorrida de forma a alocar todas as tarefas nela contidas (linha 6). A variável `slave` indica o escravo no qual a tarefa vai ser alocada (linha 7). O cabeçalho de informações no repositório é percorrido (linha 8) e quando a tarefa a ser alocada é encontrada (linha 9), é chamado o *driver* `DRV_AllocationTask`, passando como parâmetros o escravo (`slave`) onde a tarefa deve ser alocada e o endereço para a estrutura com as informações da tarefa (`&task[k]`) (linha 10). Este *driver* lê as informações do repositório montando e enviando um pacote com os seguintes campos:

`<`**`target`**`=slave><`**`size`**`=6+(task[k].size*2)><`**`service`**`=TASK_ALLOCATION>`

`<`**`task_id`**`=task[k].id><`**`code_size`**`=task[k].size><`**`code`**`=conteúdo do intervalo [task[k].initial_address, task[k].initial_address + task[k].size]>`

Após alocar a tarefa, o mestre deve informar aos outros escravos que uma nova tarefa está alocada no sistema. Então, é chamado o *driver* `DRV_AllocatedTask` passando como parâmetros o escravo onde a tarefa foi alocada (`slave`), o identificador da tarefa (`task[k].id`) e o escravo que está sendo informado (`slaves[j]`) (linha 13). Dado que a rede não possui serviço de *multicast*, esta operação é realizada por várias transmissões *unicast*. Este *driver* monta e envia um pacote com os seguintes campos:

`<`**`target`**`=slaves[j]><`**`size`**`=6><`**`service`**`=ALLOCATED_TASK>`

`<`**`processor`**`=slave><`**`task_id`**`=task[k].id>`

Após realizar a alocação estática, o mestre executa o *driver* `DRV_FinishedAllocation`, que monta um pacote com o serviço `FINISHED_ALLOCATION` e envia a todos os escravos, indicando que a alocação estática foi concluída. Então, ele habilita as

interrupções provenientes da NoC de forma a aguardar pacotes de comunicação dos escravos, para realizar a alocação dinâmica de tarefas.

```
1 void TasksAllocation(){
2   int i, j, slave, k;
3   TaskPackage* task;
4   task = (TaskPackage*)0x10000000;
5
6   for(i = 0; static_allocation[i].task != -1; i++){
7     slave = static_allocation[i].slave;
8     for(k = 0; k < MAXGLOBALTASKS; k++){
9         if(static_allocation[i].task == task[k].id){
10          DRV_AllocationTask(slave, &task[k]);
11          for(j=0; j<MAXPROCESSORS; j++){
12              if(slaves[j] != slave)
13                DRV_AllocatedTask(slave, task[k].id, slaves[j]);
14          }
15          break;
16        }
17     }
18  }
19}
```

Figura 49 – Função de alocação estática do nodo mestre, `TasksAllocation`.

6.1.3 Alocação dinâmica

A alocação dinâmica de tarefas compreende o envio de uma tarefa (t_i) a um nodo escravo mediante requisição de outra tarefa (t_j). Ou seja, o nodo mestre realiza a alocação dinâmica quando uma tarefa que está executando (t_j) requisita a alocação de outra tarefa que se encontra no repositório (t_i). É importante destacar que as requisições de alocação são, na prática, transparentes a t_j, sendo requisitadas pelo *microkernel* quando t_j tentar enviar uma mensagem a t_i e esta não estiver alocada no sistema.

O nodo mestre, após a execução da alocação estática, aguarda interrupções provenientes da rede com solicitação de serviços. Os serviços contidos nos pacotes podem ser: *(i)* requisição de alocação de tarefa; *(ii)* notificação de término de execução de tarefa.

O fluxograma da Figura 50 ilustra quando tarefas são requisitadas, no lado do nodo escravo. Quando a tarefa t_j tenta enviar uma mensagem à tarefa t_i (`WritePipe(&msg,`t_i`)`) (1), o *microkernel* do nodo escravo verifica na tabela de localização de tarefas (`task_location`) se t_i está alocada. Se afirmativo, a chamada de sistema é concluída com sucesso (2). Senão, verifica-se se a alocação estática já está terminada. Se afirmativo, t_i é requisitada chamando o *driver* `DRV_RequestTask` (3). Senão, armazena-se a requisição em uma tabela de requisições de tarefas, `requestTask` (4). Esta tabela contém os campos `requested` e `requesting`, que correspondem, respectivamente, ao identificador da tarefa requisitada e o identificador da tarefa que está requisitando. Nos dois últimos casos (3 e 4), a tarefa que está requisitando, é colocada em espera (`status=WAITING`).

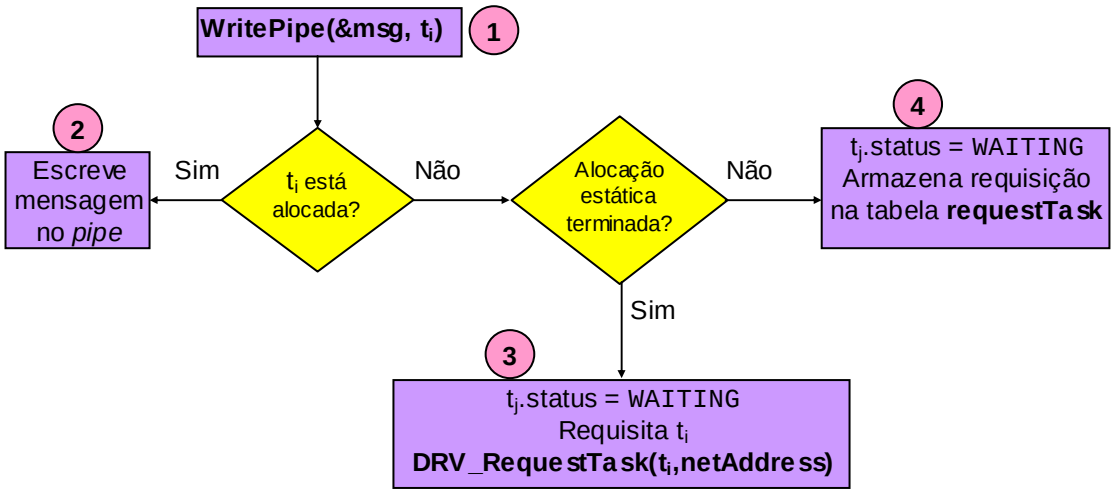


Figura 50 – Requisição de uma tarefa por parte do nodo escravo.

Enquanto o processo de alocação estática não termina, tarefas que já foram alocadas podem solicitar tarefas que ainda deverão ser alocadas. Dessa forma, as requisições são armazenadas na tabela `requestTask` e na medida que as tarefas requisitadas são alocadas, as tarefas que fizeram as requisições são desbloqueadas e podem ser escalonadas novamente. Suponha que, no caso da Figura anterior, a requisição de t_j por t_i tenha sido armazenada em requestTask (4). A Figura 51, mostra o procedimento tomado quando t_i é alocada no sistema (1). A tarefa é inserida na tabela de localização de tarefas, `task_location` (2). Se existe requisição de t_i na tabela `requestTask`, esta requisição vai ser removida e t_j vai ser desbloqueada (`status=READY`) (3);

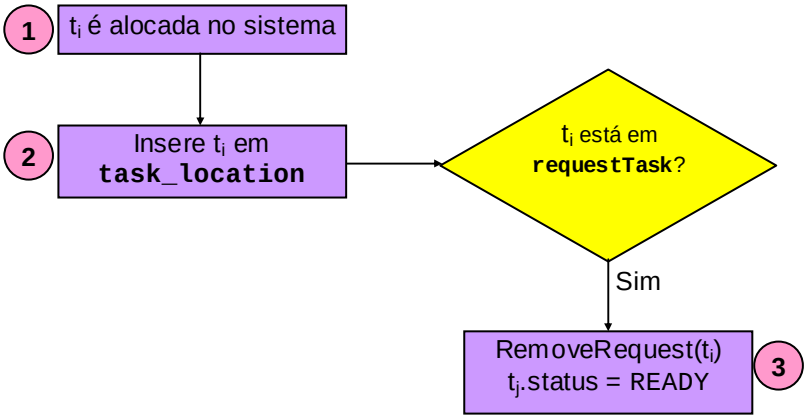


Figura 51 – Desbloqueio da tarefa t_j, cuja requisição por t_i estava na tabela de requisições de tarefas, `requestTask`.

Quando o processo de alocação estática termina, verifica-se, em `requestTask`, se ainda existem requisições pendentes. Se sim elas deverão ser alocadas. Este tratamento é explicado na Seção 6.2.

Quando uma tarefa deve ser requisitada, o *driver* `DRV_RequestTask` monta e envia um pacote com os seguintes campos:

<**target**=MASTER><**size**=6><**service**=REQUEST_TASK>

<**processor**=netAddress><**taskID**=t_i>

O fluxograma da Figura 52 ilustra o procedimento realizado quando uma tarefa t_i termina sua execução. Uma chamada de sistema é gerada com o serviço EXIT (1). Se não existem mensagens no *pipe* escritas por t_i, o status da tarefa passa a ser FREE, liberando o TCB que ela está ocupando; t_i é removida da tabela tasks_location; é chamado o *driver* DRV_TerminatedTask, que informa ao mestre que a tarefa t_i terminou sua execução e pode ser liberada (2). Se ainda existem mensagens de t_i no *pipe*, o status desta tarefa passa a ser TERMINATED, a variável needTaskScheduling é configurada para 1 (3) e só após todas as mensagens de t_i serem consumidas, é informado ao mestre que a tarefa pode ser liberada. Isso se deve ao fato de que, quando uma tarefa é liberada, ela é removida da tabela de localização de tarefas de todos os escravos. Se tiver alguma tarefa tentando ler uma mensagem da tarefa liberada, esta não vai ser encontrada na tabela.

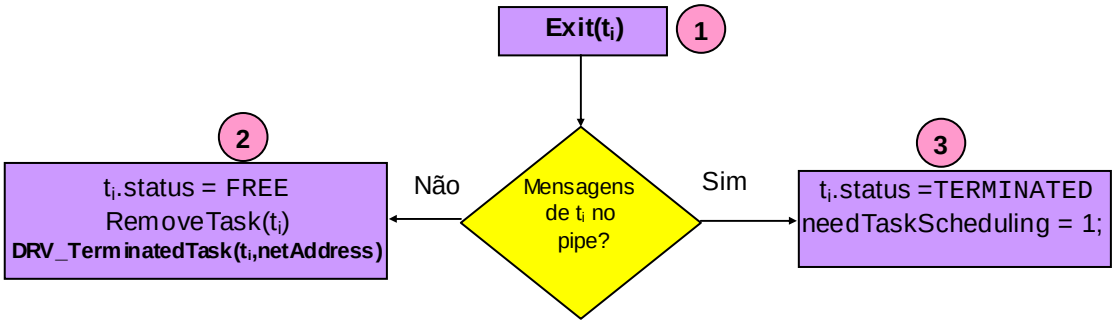


Figura 52 - Término de uma tarefa no nodo escravo.

Para informar ao mestre que uma tarefa está terminada e pode ser liberada, o *driver* DRV_TerminatedTask monta e envia um pacote com os seguintes campos:

<**target**=MASTER><**size**=6><**service**=TERMINATED_TASK>

<**task_id**=t_i><**processor**=netAddress>

O nodo mestre aguarda interrupções provenientes da NoC, geradas pela recepção dos pacotes REQUEST_TASK e TERMINATED_TASK, descritos acima. Quando uma interrupção é gerada, o contexto do mestre é salvo na pilha e o fluxo de execução salta para a função que trata a interrupção. Esta função é mostrada na Figura 53.

O serviço que o pacote carrega é lido da NI pelo *driver* DRV_ReadService e armazenado no endereço do parâmetro service (linha 7). Se o serviço for REQUEST_TASK, é chamado o *driver* DRV_RequestTask, que lê da NI o identificador da tarefa requisitada e o escravo que está requisitando (linha 10). O cabeçalho de informações do repositório de tarefas é percorrido e quando a tarefa requisitada for encontrada (linha 12), procura-se um escravo com página livre para alocar a tarefa (linha 13). Então, é chamado o *driver* DRV_AllocationTask

(abordado em 6.1.2) (linha 14). Após alocar a tarefa, é enviado um pacote a todos os outros escravos informando que uma nova tarefa foi alocada (linha 18).

Se o serviço for TERMINATED_TASK, é chamado o *driver* DRV_TerminatedTask que lê da NI o identificador da tarefa que terminou a execução e o escravo onde ela está alocada (linha 25). Então, o *driver* DRV_DeallocatedTask monta e envia um pacote para outros escravos informando que uma tarefa deve ser liberada (linha 28). Este pacote contém os seguintes campos:

<**target**=slaves[j]><**size**=4><**service**=DEALLOCATED_TASK>

<**task_id**=task[k].id>

```
1 void DRV_Handler() {
2   int service, size, i, taskID, j;
3   unsigned int slave;
4   TaskPackage* task;
5   task = (TaskPackage*)0x10000000;
6
7   DRV_ReadService(&service, &size);
8   switch (service){
9      case REQUEST_TASK:
10     DRV_RequestTask(&taskID, &slave);
11        for(i = 0; i < MAXGLOBALTASKS; i++){
12           if(taskID == task[i].id){
13              slave = SearchSlaveAvailable();
14              DRV_AllocationTask(slave, &task[i]);
15              OccupiedPage(slave);
16              for(j=0; j<MAXPROCESSORS; j++){
17                 if(slaves[j] != slave)
18                    DRV_AllocatedTask(slave, task[i].id, slaves[j]);
19              }
20              break;
21           }
22        }
23     break;
24     case TERMINATED_TASK:
25        DRV_TerminatedTask(&taskID,&slave);
26        for(j=0; j<MAXPROCESSORS; j++){
27           if(slaves[j] != slave)
28              DRV_DeallocatedTask(taskID,slaves[j]);
29        }
30     break;
31   }
32 }
```

Figura 53 – Função DRV_Handler, que trata as interrupções no nodo mestre.

Uma importante função do código da Figura 53 é a SearchSlaveAvailable (linha 13). No contexto do presente trabalho, a distribuição de carga procura uniformizar o número de tarefas por processador. Sugere-se para trabalhos futuros incluir funções mais elaboradas de distribuição de carga, que levem em conta parâmetros como volume de comunicação entre as tarefas, posicionamento das tarefas na rede de forma a minimizar o congestionamento na rede, o tempo de execução das tarefas, dentre outros.

6.2 Nodo escravo

Conforme explicado na Seção 5.3, os nodos escravos, quando são inicializados, contém em

sua memória apenas o *microkernel*, executando a tarefa `idle`. Dessa forma, as tarefas são transferidas para a memória do processador, através da rede, em dois momentos: recepção das tarefas iniciais (podendo não haver tarefas inicias designadas para o nodo escravo), caracterizando a alocação estática; e recepção de novas tarefas em tempo de execução, caracterizando a alocação dinâmica.

A Figura 54 complementa a Figura 45, apresenta no Capítulo 5, com os serviços executados pelo *microkernel*. Quatro serviços foram incluídos, para permitir a alocação e liberação de tarefas. Estes serviços são executados mediante uma interrupção causada pela recepção de um pacote oriundo da rede.

```
1 void DRV_Handler() {
2   int service, size, i, task;
3
4   DRV_ReadService(&service, &size);
5
6   switch (service)
7   {
8      case REQUEST_MESSAGE:
9      ...
10     case DELIVER_MESSAGE:        } Capítulo 5
11     ...
12     case NO_MESSAGE:
13     ...
14     case TASK_ALLOCATION:
15            for(i=0; i<MAXLOCALTASKS; i++)
16                   if(tcbs[i].status == FREE)
17                          break;
18            tcbs[i].status = ALLOCATING;
19            tcbs[i].pc = 0x0;
20            allocatingTCB = &tcbs[i];
21            OS_InterruptMaskClear(IRQ_NOC);
22            DRV_StartAllocation(tcbs[i].offset);
23            break;
24     case ALLOCATED_TASK:
25            DRV_AllocatedTask();
26            break;
27     case DEALLOCATED_TASK:
28            DRV_DeallocatedTask(&task);
29            RemoveTask(task);
30            break;
31     case FINISHED_ALLOCATION:
32            finishedAllocation = 1;
33            for(i = 0; i < MAX_REQUEST_TASK; i++){
34               if(requestTask[i].requesting != -1)
35                   DRV_RequestTask(requestTask[i].requested, netAddress);
36            }
37            break;
38     }
39 }
```

Figura 54 – Função `DRV_Handler` do escravo, com o tratamento de alocação de tarefas, complementando a Figura 45.

Se o serviço que o pacote carrega é `TASK_ALLOCATION`, significa que o código objeto de uma tarefa está sendo transferido do repositório para a NI e deve ser alocado na memória do

processador. Dessa forma, para alocar a tarefa, é procurado um TCB livre (linhas 15 a 17)[1]. O `status` deste TCB passa a ser `ALLOCATING` (linha 18) e o `pc` é configurado para `0` (linha 19). Uma variável global, `allocatingTCB` contém o endereço do TCB que está sendo utilizado para alocar a tarefa (linha 20). As interrupções provenientes da NoC são desabilitadas (linha 21). O *driver* `DRV_StartAllocation` é chamado passando como parâmetro o endereço (página) a partir do qual a tarefa vai ser alocada (linha 22). Este *driver* lê da NI o identificador da tarefa armazenando-o no TCB referenciado por `allocatingTCB` e o tamanho do código objeto da tarefa. Então, ele informa ao DMA o tamanho do código objeto da tarefa (escrevendo no registrador `SET_DMA_SIZE`), o endereço da memória a partir do qual o código deve ser transferido (escrevendo no registrador `SET_DMA_ADDRESS`) e ativa o DMA (escrevendo no registrador `START_DMA`).

Após ter recebido e tratado um pacote com o serviço `TASK_ALLOCATION`, a CPU do escravo continua sua execução em paralelo com o DMA, que realiza a transferência do código objeto para a memória. O controlador de DMA interrompe a CPU quando a transferência estiver concluída. A função que trata a interrupção do DMA é mostrada na Figura 55. A tarefa cujo código acaba de ser transferido para a memória é colocada na tabela `task_location` (linha 3). A tarefa, que está ocupando o TCB referenciado por `allocatingTCB`, passa a ter status `READY` (linha 4). O *microkernel* avisa ao DMA que a interrupção foi aceita (linha 5). Se a tarefa que estava executando antes da interrupção era a tarefa `idle`, `needTaskScheduling` é colocada em `1` indicando a necessidade do escalonamento de uma nova tarefa (linhas 7 e 8). As interrupções provenientes da NoC são habilitadas (linha 10).

```
1 void DMA_Handler(){
2
3     InsertTaskLoc(allocatingTCB->id,netAddress);
4     allocatingTCB->status = READY;
5     MemoryWrite(DMA_ACK,1);
6
7     if (current == &tcbs[MAXLOCALTASKS])
8           needTaskScheduling = 1;
9
10    OS_InterruptMaskSet(IRQ_NOC);
11}
```

Figura 55 - Função `DMA_Handler` que trata a interrupção advinda do DMA.

Se o serviço é `ALLOCATED_TASK`, significa que uma tarefa foi alocada no sistema. Para tratar este pacote, é chamado o *driver* `DRV_AllocatedTask` (linha 25 da Figura 54). Este *driver* lê da NI o endereço do processador no qual a tarefa foi alocada e o identificador da tarefa. Estes dois dados são inseridos na tabela de localização de tarefas (`task_location`).

Se serviço é `DEALLOCATED_TASK`, significa que uma tarefa deve ser liberada. Dessa

[1] Sempre haverá retorno de TCB livre, pois o mestre mantém o controle de páginas livres em todos os processadores escravos.

forma, é chamado o *driver* `DVR_DeallocatedTask` (linha 28 da Figura 54), que lê da NI o identificador da tarefa. Em seguida, a tarefa é removida da tabela `task_location` (linha 29 da Figura 54).

Se o serviço é `FINISHED_ALLOCATION`, significa que o nodo mestre terminou a alocação estática. Assim, a variável `finishedAllocation` é configurada em 1 (linha 32 da Figura 54) e verifica-se na tabela `requestTask` se existe alguma requisição de tarefa pendente. Se afirmativo, o *driver* `DRV_RequestTask` é chamado passando como parâmetros a tarefa requisitada e o endereço do escravo que está solicitando (linha 35 da Figura 54).

A validação dos procedimentos de alocação de tarefas, estática e dinâmica, é apresentada no Capítulo seguinte.

7 RESULTADOS

Este Capítulo está estruturado como segue. Inicialmente apresenta-se a validação do *microkernel*, ilustrando-se por simulação funcional cada serviço implementado. A Seção seguinte apresenta a utilização do sistema para a execução de uma aplicação paralela simples, M*erge Sort*, procurando-se mostrar o impacto do posicionamento das tarefas no desempenho da aplicação. A terceira parte do Capítulo apresenta os resultados da execução de aplicações paralelas sobre a plataforma. A quarta e última parte apresenta parte da codificação MPEG, composta por 5 tarefas: envio de dados, IVLC (Inverse Variable Length Coding), IQUANT (Inverse Quantization Algorithm), IDCT (Inverse Discrete Cosine Transform) e recepção dos dados. Os resultados serão apresentados na forma da taxa de recepção dos dados pela última tarefa. Por MPEG ser uma aplicação tempo real, é importante a avaliação da vazão que o sistema pode fornecer.

7.1 Validação do *Microkernel*

Os serviços avaliados nesta Seção compreendem: escalonamento de tarefas, comunicação entre tarefas na mesma CPU, comunicação entre tarefas posicionadas em CPUs distintas, e a alocação (estática e dinâmica) de tarefas.

7.1.1 Escalonamento de tarefas

A Figura 56 apresenta a alocação inicial de três tarefas, t_1, t_2 e t_3 em um dado processador. São mostrados os sinais da NI , CPU (o processador) e DMA. Na NI, `data_in` indica os dados recebidos da NoC e `interrupt_plasma` indica que a NI está desejando interromper o processador para receber pacotes. Na CPU, `page` indica a página da memória da qual o código objeto é executado. O código objeto do *microkernel* encontra-se na página 0. Nas páginas 1, 2 e 3 residem códigos objetos de tarefas. Os sinais sinal `intr_enable` e `intr_signal` indicam, respectivamente, quando as interrupções estão habilitadas e quando o processador é interrompido. No DMA, `start` representa quando o DMA começa a transferência do código objeto de uma tarefa para a memória do processador. O sinal `interrupt` indica quando o DMA comunica que o código já está completamente da memória.

A seguir, são mostrados os passos do processo de alocação inicial de tarefas em um processador. Os números abaixo têm correspondência com os números da Figura 56.

1. A CPU inicia executando o *microkernel* (`page=0`) que, após completar o boot, chama a tarefa `idle`, habilitando as interrupções.

2. O mestre envia um pacote para o processador com o serviço `TASK_ALLOCATION`, indicando que uma tarefa (t_1) será alocada em sua memória.A NI recebe o pacote e interrompe a CPU.

3. As interrupções são desabilitadas. O *microkernel* trata a interrupção lendo o pacote da NI e

verificando que uma tarefa deve ser transferida para a memória. Após o tratamento da interrupção, a CPU sinaliza o DMA para que este comece o processo de transferência.

4. O *microkernel* habilita as interrupções e continua sua execução enquanto o DMA faz a transferência.
5. O DMA termina a transferência dos dados e interrompe a CPU.
6. O *microkernel* desabilita as interrupções, faz a inicialização da tarefa e a escalona para executar.
7. As interrupções são habilitadas e t_1 entra em execução (`page=1`). A região destacada na Figura mostra que a tarefa t_1 é executada por um perído de tempo muito pequeno, pois um pedido de interrupção é atendido.
8. Outro pacote indicando nova tarefa (t_2) é recebido pela NI que interrompe a CPU.
9. Da mesma forma que antes, a CPU sinaliza o DMA para iniciar a transferência.
10. A CPU retoma a execução da tarefa t_1.
11. O DMA interrompe a CPU indicando que existe uma nova tarefa na memória.
12. O mesmo processo se repete para t_3. Mesmo com a chegada de novas tarefas, t_1 é executada até completar seu *timeslice*.
13. Após ter completado a alocação inicial, o mestre envia um pacote ao processador com o serviço `FINISHED_ALLOCATION`. A NI recebe este pacote e interrompe a CPU.

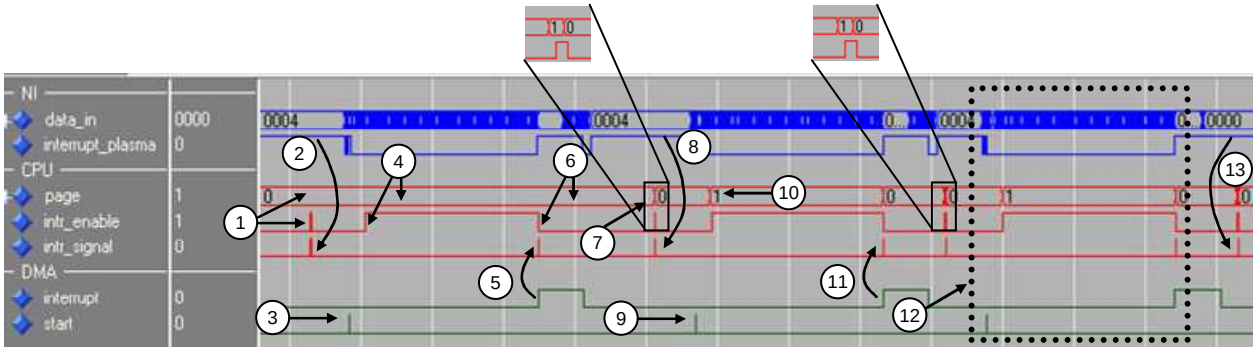


Figura 56 – Alocação inicial das tarefas t_1, t_2 e t_3 no processador.

A seqüência de operações descritas acima demostram a correta operação dos mecanismos de interrupção, DMA e alocação estática de tarefas.

Uma vez as tarefas alocadas na CPU, o *microkernel* passa a escaloná-las. A Figura 57 mostra a continuação da simulação apresentada na Figura 56, apresentando o escalonamento de tarefas. Agora t_1, t_2 e t_3 concorrem pelo uso da CPU. Os eventos apresentados na Figura 57 são descritos como segue:

1. t_1 termina seu *timeslice* (`page=1`).
2. Uma interrupção é gerada pelo contador de *timeslice*.
3. As interrupções são desablitadas. O *microkernel* entra em execução, salva o contexto de t_1, escalona t_2 e carrega o contexto desta nos registradores.
4. As interrupções são habilitadas. O fluxo de execução é desviado para página 2, onde t_2 está alocada.
5. t_2 termina seu *timeslice* e uma nova interrupção é gerada.
6. As interrupções são desabilitadas. O *microkernel* entra em execução, salva o contexto de t_2, escalona t_3 e carrega o contexto desta nos registradores.
7. As interrupções são habilitadas. O fluxo de execução é desviado para a página 3, onde t_3 está alocada.

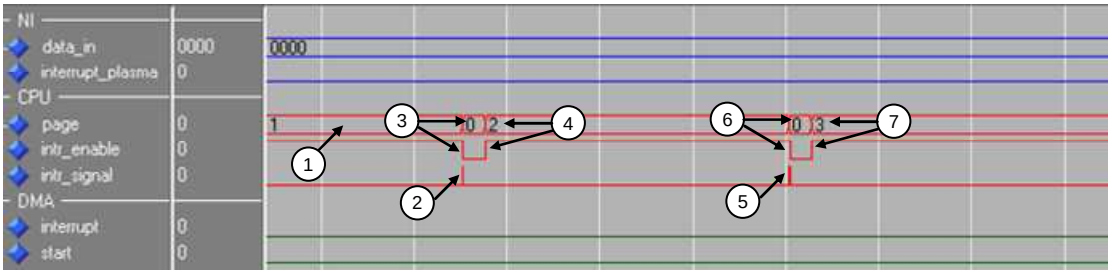


Figura 57 – Troca de contexto entre t_1, t_2 e t_3.

Quando uma tarefa termina sua execução, ela deve ser removida da lista de tarefas a serem escalonadas. A Figura 58 mostra o término da tarefa t_2. O sinal `intr_signal` representa quando acontece uma chamada de sistema.

1. t_2 faz uma chamada de sistema comunicando ao *microkernel* que terminou sua execução.
2. O *microkernel* entra em execução, atribui `TERMINATED` para o status de t_2 e escalona a próxima tarefa, t_3.
3. A partir deste momento, apenas t_1 e t_3 concorrem pelo uso da CPU.

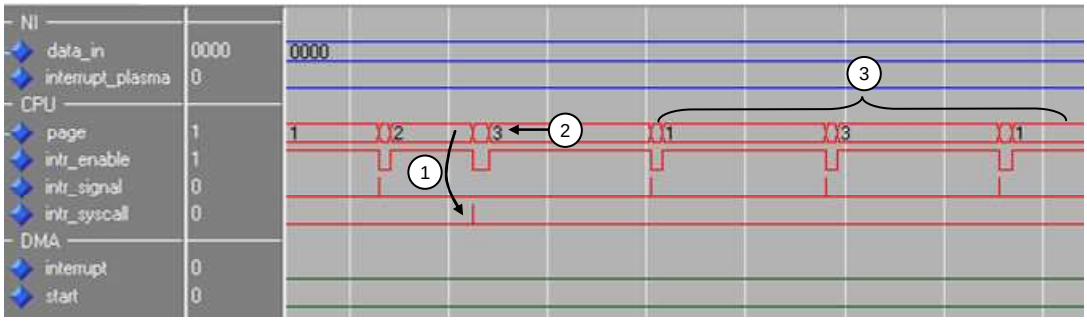


Figura 58 – Término da tarefa t_2.

7.1.2 Comunicação entre tarefas na mesma CPU

A Figura 59 mostra a comunicação entre duas tarefas, t_1 e t_2, que residem na mesma CPU. São mostrados os sinais do *microkernel* (`page` e `intr_syscall`), uma parte da área de memória de t_1 (`Tarefa 1`) correspondente à mensagem a ser enviada e uma parte da área de memória de t_2 (`Tarefa 2`) correspondente à mensagem que vai ser recebida. Os passos do processo de comunicação entre tarefas na mesma CPU são apresentados a seguir.

1. t_1 executa e monta uma mensagem cujo tamanho (`length`) é 3 e cujo conteúdo é 1 (`msg[0]`), 2 (`msg[1]`) e 3 (`msg[2]`). t_1 faz uma chamada de sistema (`WritePipe(&msg,2)`), enviando a mensagem para t_2.
2. O *microkernel* entra em execução e copia a mensagem para o *pipe*.
3. O *timeslice* de t_1 termina e t_2 é escalonada.
4. t_2 faz uma chamada de sistema (`ReadPipe(&msg,1)`) pedindo a leitura da mensagem de t_1.
5. O *microkernel* copia a mensagem do *pipe* para a área de memória de t_2.
6. t_2 volta a executar.

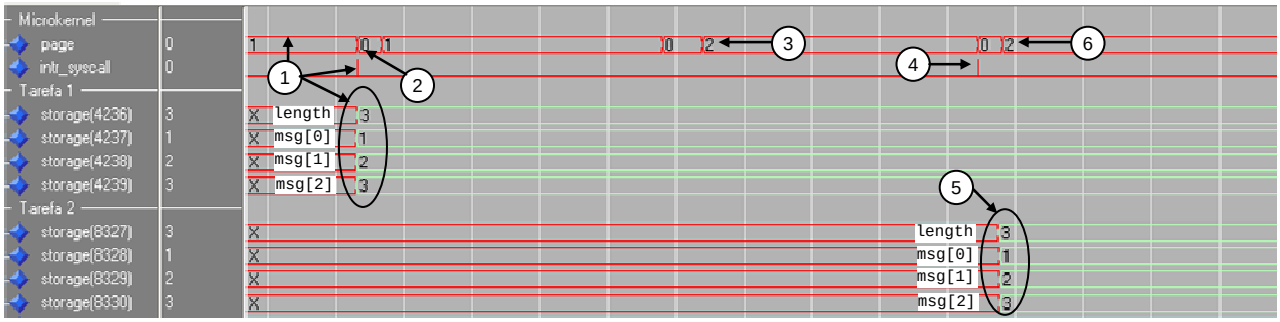


Figura 59 – Comunicação entre t_1 e t_2 que estão alocadas no mesmo processador.

A Figura 60 apresenta os tempos, em ciclos de relógio, para a escrita e a leitura de uma mensagem de tamanho 3. A escrita da mensagem no *pipe* gasta 357 ciclos. Este número compreende o intervalo entre a chamada de sistema executada por t_1 (`WritePipe`) e o término do tratamento desta chamada pelo *microkernel*. As operações executadas neste intervalo são: (*i*) salvamento parcial de contexto de t_1; (*ii*) verificação do serviço resquisitado; (*iii*) escrita da mensagem no *pipe*; (*iv*) restauração do contexto parcial de t_1. A leitura da mensagem gasta 365 ciclos. Este número compreende o intervalo entre a chamada de sistema executada por t_2 (`ReadPipe`) e o término do tratamento desta chamada pelo *microkernel*. As operações executadas neste intervalo são: (i) salvamento parcial de contexto de t_2; (*ii*) verificação do serviço resquisitado; (*iii*) leitura da mensagem no *pipe* e escrita da mesma na página de t_2; (*iv*) restauração do contexto parcial de t_2.

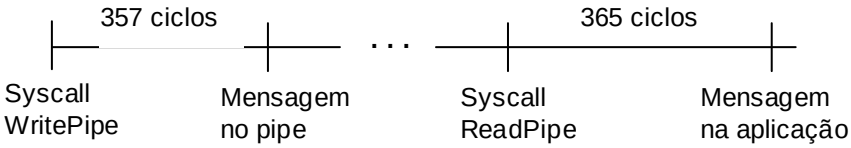


Figura 60 - Tempos, em ciclos de relógio, para a escrita e a leitura de uma mensagem de tamanho 3, no mesmo processador.

A Tabela 10 e apresenta o número de ciclos de relógio gastos para as operações `WritePipe` e `ReadPipe`, com diferentes tamanhos de mensagem. Os gráficos da Figura 61 e da Figura 62 mostram que o crescimento é linear em função do tamanho da mensagem.

Tabela 10 – Número de ciclos de relógio para as operações `WritePipe` e `ReadPipe` com diferentes tamanhos de mensagem.

Tamanho da mensagem	Número de ciclos (`WritePipe`)	Número de ciclos (`ReadPipe`)
3	357	365
30	573	573
60	813	821
100	1133	1141
128	1357	1365

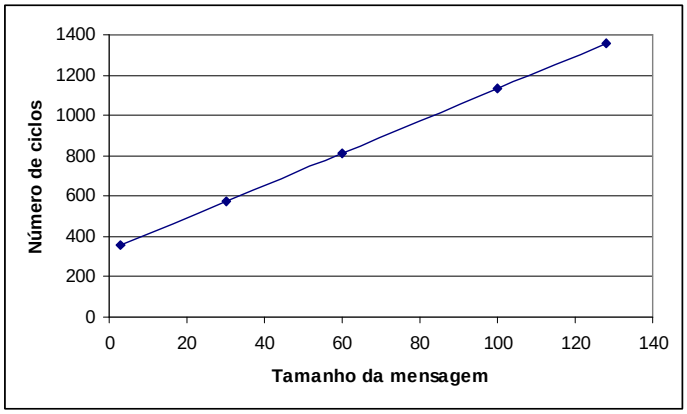


Figura 61 – Gráfico para a operação `WritePipe` em função do tamanho da mensagem.

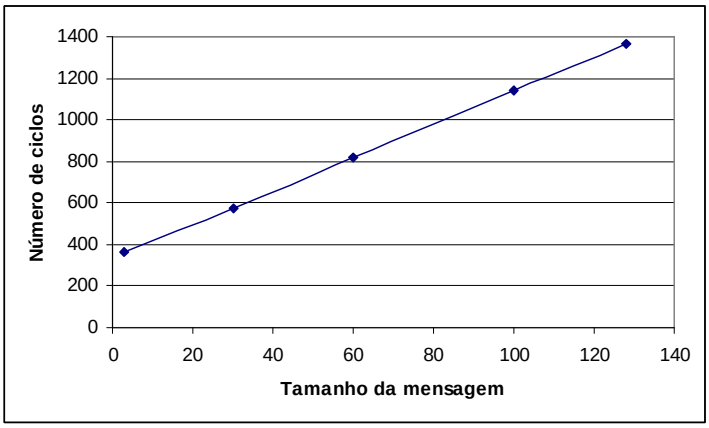


Figura 62 - Gráfico para a operação `ReadPipe` em função do tamanho da mensagem.

7.1.3 Comunicação entre tarefas em CPUs distintas

A Figura 63 apresenta a comunicação entre duas tarefas que residem em processadores diferentes. A tarefa t_1. reside em PROC1 e a tarefa t_2, em PROC2. Ambas tarefas estão alocadas na página 1 da memória de cada processador. Os passos para a comunicação são descritos a seguir.

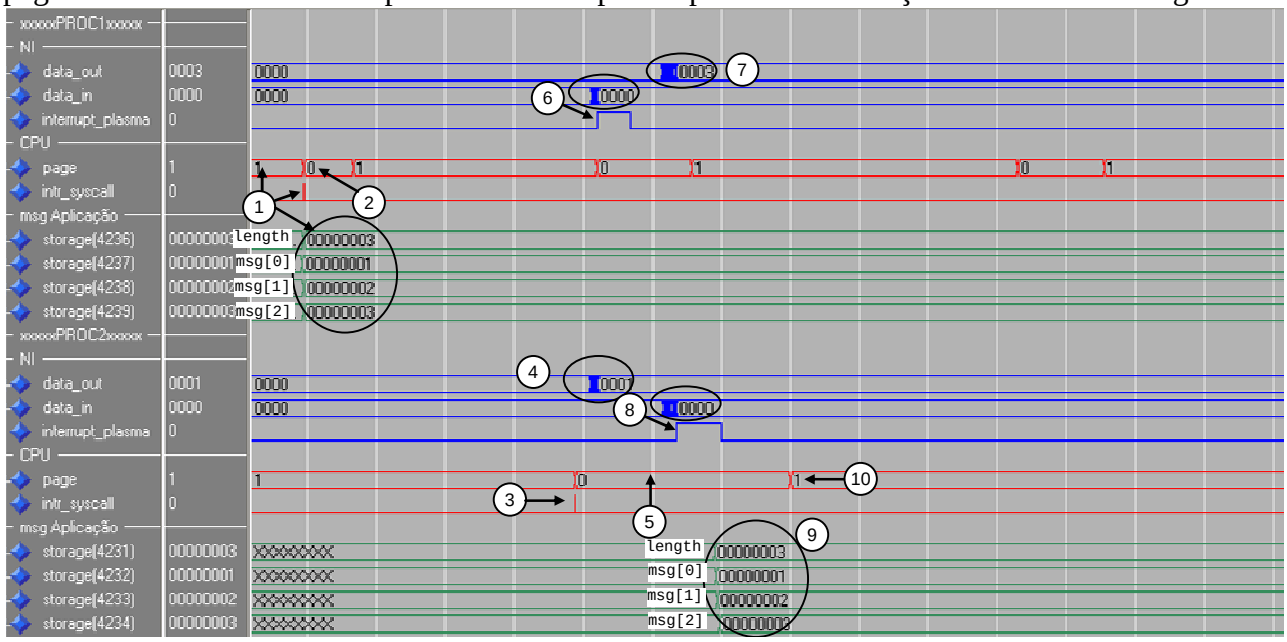


Figura 63 – Comunicação entre tarefas em CPUs distintas.

1. Em PROC1, t_1 executa e monta uma mensagem cujo tamanho (`length`) é 3 e cujo conteúdo é 1 (`msg[0]`), 2 (`msg[1]`) e 3(`msg[2]`). t_1 faz uma chamada de sistema (`WritePipe(&msg,2)`), enviando a mensagem para t_2.
2. O *microkernel* entra em execução e copia a mensagem para o *pipe* do *microkernel*. Não há envio de dados para t_2.
3. Posteriormente, t_2 (em PROC2) executa e faz uma chamada de sistema pedindo a leitura da mensagem de t_1 (`ReadPipe(&msg,1)`) . Este evento é totalmente assíncrono em relação à geração da mensagem por parte da tarefa t_1.
4. O *microkernel* verifica que t_2 está alocada em PROC1, monta um pacote de requisição de mensagem e envia este pacote ao processador PROC1.
5. t_2 é colocada em estado de espera (`status=waiting`) e, uma vez que não tem mais tarefas para escalonar, o *microkernel* chama a tarefa `idle` e aguarda a resposta da requisição.
6. O pacote de requisição chega em PROC1 e a NI interrompe a CPU.
7. O *microkernel* monta a mensagem de resposta e envia a PROC2.
8. A mensagem de resposta chega em PROC2 e a NI interrompe a CPU.

9. O *microkernel* copia a mensagem para a área de memória de t_2.

10. t_2 volta a executar.

O tempo para a escrita no *pipe* é o mesmo da comunicação entre tarefas na mesma CPU, pois a escrita é sempre local. No entanto o tempo de leitura é superior, pois envolve um número superior de passos. A Figura 64 apresenta os intervalos de tempo envolvidos nesta comunicação.

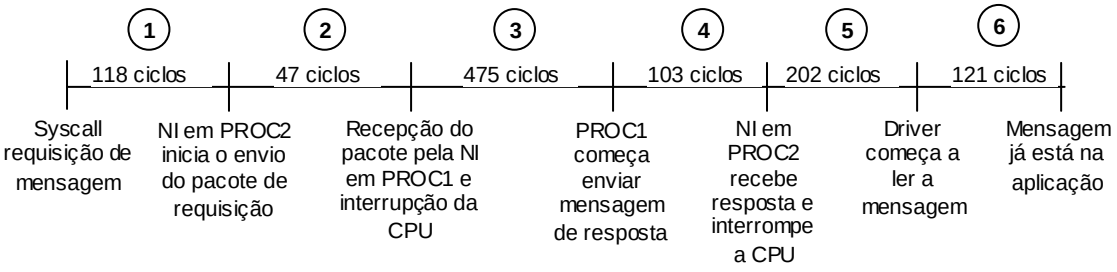


Figura 64 - Análise do tempo para leitura de mensagem armazenada em outro processador.

O tempo total para a operação de *read*, com tamanho da mensagem igual a 3 e *hops*=1, é igual a 1066 ciclos de relógio (somatório dos intervalos 1 a 6 na Figura 64). Cada intervalo é detalhado a seguir:

- **1º intervalo** – Invariável. Corresponde ao processamento realizado desde a chamada de sistema `ReadPipe` pela tarefa t_2, até o início do envio do pacote de requisição à `PROC1`.

- **2º intervalo** – Corresponde a latência de rede e varia de acordo com o número de saltos na mesma. Neste exemplo, o número de saltos é mínimo — 1 *hop*: `PROC1` é vizinho de `PROC2`. O número de operações de arbitragem e roteamento envolvidas é igual à distância dos roteadores mais um, pois o roteador destino deve encaminhar o pacote para a porta local. Logo, no presente caso, *hops*=1, temos 2 operações de arbitragem e roteamento.

- **3º intervalo** – Invariável. Corresponde ao processamento de `PROC1` para tratar o pacote de requisição e iniciar o envio da mensagem de resposta.

- **4º intervalo** – Corresponde à latência de rede. Varia de acordo com o número de saltos na rede e com o tamanho da mensagem. Neste exemplo, a mensagem possui tamanho 3 gastando 103 ciclos neste intervalo. É importante ressaltar que a NI interrompe o processador quando: (*i*) possui um pacote completo no *buffer* da NI ou; (*ii*) quando o *buffer* da NI estiver cheio. Dessa forma, para mensagens superiores ao tamanho do *buffer* da NI (profundidade igual a 16 palavras de 32 bits) a interrupção do processador pela NI vai acontecer sempre na segunda condição (*buffer* cheio) gastando 200 ciclos de relógio. Este número leva em consideração a distância mínima entre fonte e destino na rede, devendo-se somar saltos adicionais para distâncias maiores.

- **5º intervalo** – Invariável. Corresponde ao processamento de PROC2 para identificar o serviço carregado pelo pacote e chamar o *driver* responsável por ler a mensagem da NI e copiá-la para a área de memória de t_2.

- **6º intervalo** – Corresponde ao tempo de processamento do *driver* para ler a mensagem de NI

transferi-la para a área de memória de t_2. O número de ciclos gastos neste intervalo varia de acordo com o tamanho da mensagem. A Tabela 11 apresenta essa variação e o gráfico da Figura 65 mostra que o crescimento é linear (condição ideal, sem congestionamento na rede).

Tabela 11 – Número de ciclos de relógio gastos para transferir mensagens (de diferentes tamanhos) da NI para a área de memória da aplicação (intervalo 6 da Figura 64).

Tamanho da mensagem	Número de ciclos
3	121
30	499
60	919
100	1479
128	1871

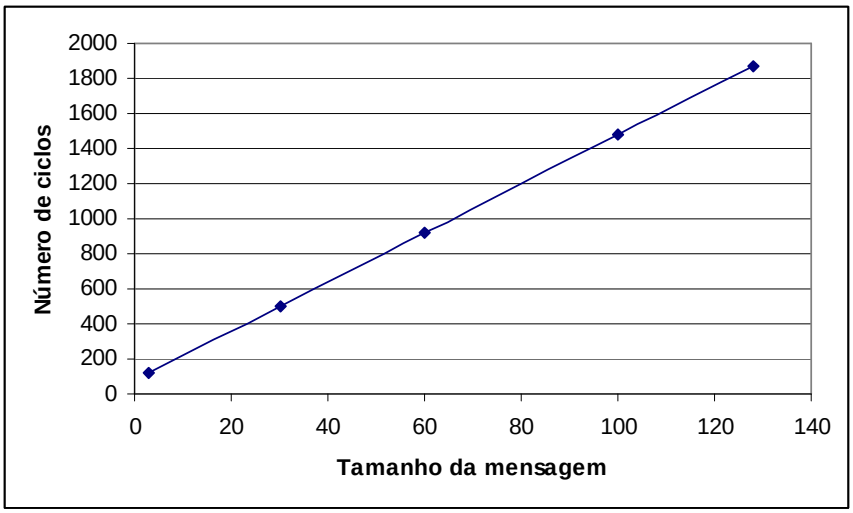


Figura 65 – Gráfico para a operação de transferência de mensagens (com tamanho variado) da NI para a área de memória da aplicação.

A comunicação remota é, em média, três vezes mais lenta que a comunicação local (comparando-se o valor 1066 com 365). O projetista das aplicações que executarão no sistema deverá considerar este tempo no momento do posicionamento das tarefas. Uma aplicação cujas tarefas são altamente cooperantes possui melhor desempenho quando estas tarefas são alocadas no mesmo processador ou em processadores vizinhos.

7.1.4 Alocação dinâmica de tarefas

O último serviço a ser validado é a alocação dinâmica de tarefas. As simulações abaixo apresentam a execução das tarefas t_1 e t_2, não havendo a tarefa t_3 no sistema. No momento que a tarefa t_2 tentar enviar uma mensagem para t_3, t_3 será alocada dinamicamente no mesmo processador.

A Figura 66 apresenta a alocação dinâmica da tarefa t_3 que é solicitada por t_2. São

apresentados alguns sinais do DMA não foram mostrados na Figura 56, que trata da alocação estática. Estes sinais são: `set_size`, que indica que o DMA está recebendo o tamanho do código objeto; `set_address`, que indica que o DMA está recebendo o endereço da memória a partir do qual o código deve ser transferido; `data_read` significa cada dado lido da NI; `data_write` é cada dado escrito na memória e; `address_write` significa o endereço da memória onde está sendo escrito um dado do código objeto.

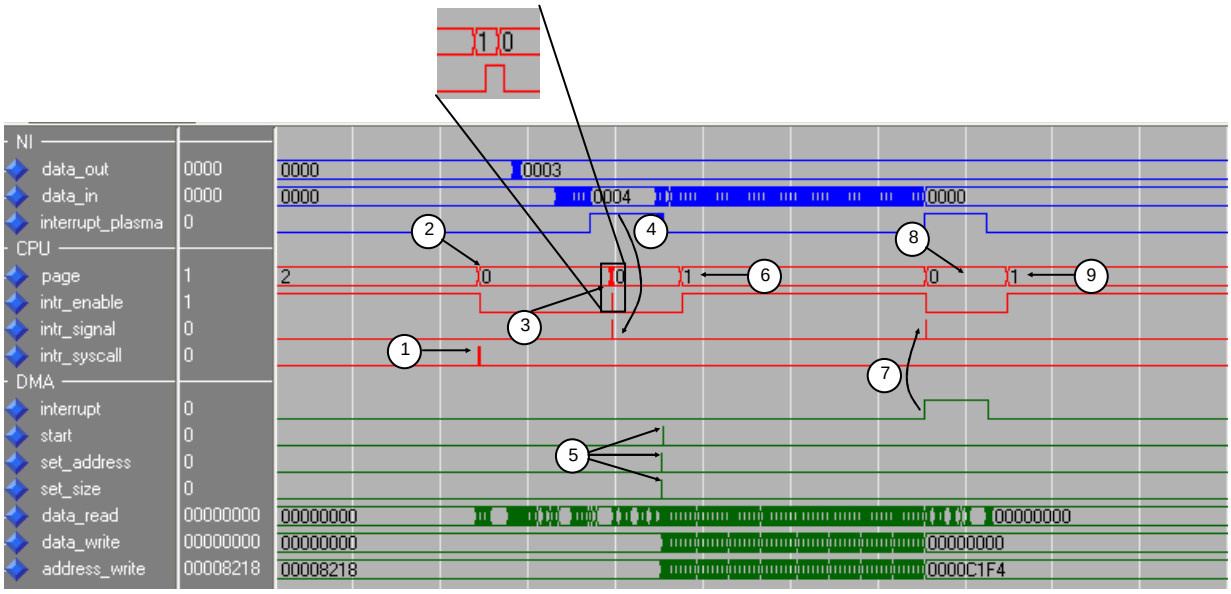


Figura 66 – Alocação dinâmica de uma tarefa.

As etapas envolvidas na alocação dinâmica de tarefa compreendem:

1. A tarefa t_2 faz uma chamada de sistema desejando enviar uma mensagem para t_3 (`while(!WritePipe(&msg,3))`).
2. O *microkernel* verifica que t_3 não está alocada. Segue-se então o envio de um pacote para o mestre com o serviço `REQUEST_TASK` solicitando a alocação desta tarefa (a transmissão do pacote deve ser observada nas transições que ocorrem no sinal `data_out`).
3. t_2 entra em estado de espera e t_1 é escalonada (em destaque na Figura).
4. O mestre envia um pacote ao processador com o serviço `TASK_ALLOCATION` e o código objeto de t_3. A NI recebe o pacote e interrompe a CPU.
5. O *microkernel* verifica que uma nova tarefa deve ser alocada. Então, envia ao DMA o tamanho do código objeto da tarefa e o endereço a partir do qual o código deve ser escrito e sinaliza para que este inicie a transferência da tarefa para a memória.
6. A tarefa t_1 continua a executar enquanto o DMA realiza a transferência.
7. O DMA termina a transferência e interrompe a CPU.

8. O *microkernel* faz as inicializações da tarefa.

9. t_1 volta a executar até completar seu *timeslice*.

A Figura 67 apresenta a continuação do processo descrito na Figura 66, mostrando que, após ser alocada, t_3 passa a ser escalonada.

1. Processo descrito na Figura 66.

2. t_2 é escalonada e refaz a chamada de sistema enviando a mensagem para t_3.

3. t_2 executa até completar seu *timeslice*.

4. t_3 é escalonada e executa a chamada de sistema requisitando a leitura da mensagem de t_2 (`ReadPipe(&msg,2)`).

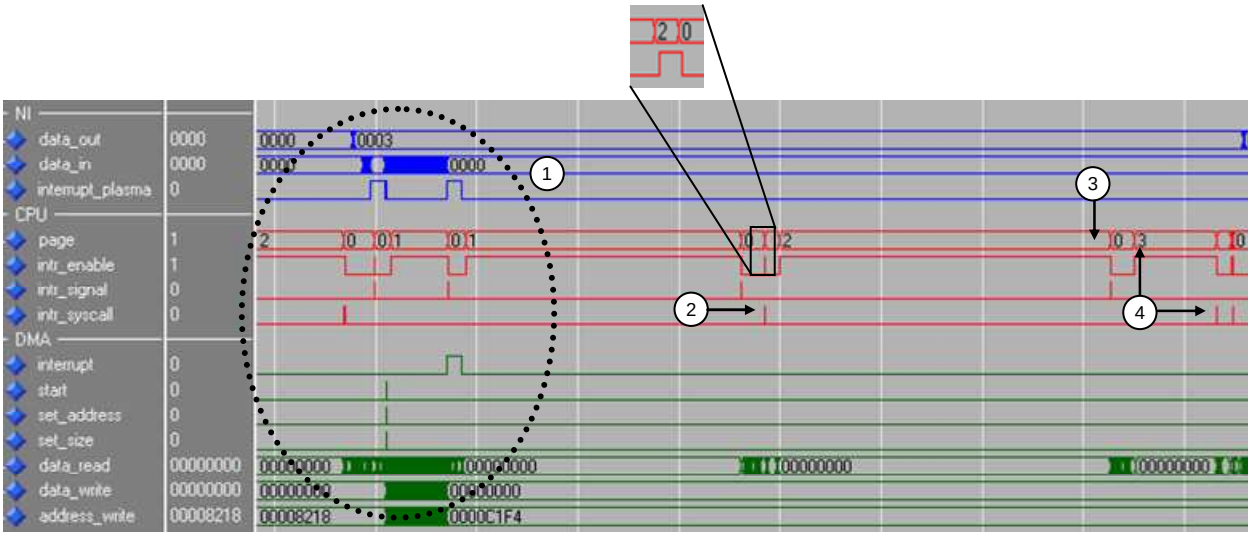


Figura 67 – Após ser alocada, t_3 é escalonada.

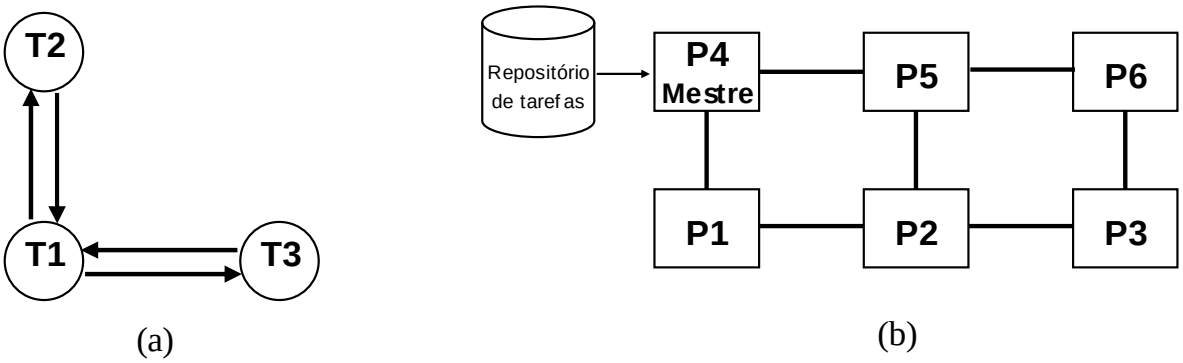
O processo de alocação dinâmica interfere minimamente no desempenho dos processadores, pois a transferência dos códigos objetos é feita por DMA. O número de ciclos de relógio gastos para a alocação de uma tarefa `t`, cujo tamanho é 3,19KB, é 11.700. A alocação corresponde ao intervalo entre o início da transmissão da tarefa pelo mestre e interrupção da CPU escrava pelo DMA, momento em que a transferência do código da tarefa foi completada. Porém, a tarefa que requisitou a alocação tem seu desempenho reduzido, pois a mesma é bloqueada no momento da chamada de sistema (`WritePipe`) que gerou a requisição de alocação. Justifica-se esta ação, pois as sucessivas escritas no *pipe* para uma tarefa ainda não alocada preencheriam o *pipe*, impedindo que as outras tarefas no processador comuniquem-se com tarefas já alocadas. A decisão de reduzir o desempenho localmente (bloquear a tarefa que requisitou a alocação) evita a degradação global de desempenho (o não bloqueio da tarefa que requisitou a alocação poderia bloquear inúmeras outras tarefas).

O projetista deve avaliar o volume de comunicação entre as tarefas, a fim de decidir por

políticas de alocação estática (quando tarefas comunicam-se com muita freqüência) ou alocação dinâmica (pouca comunicação entre as tarefas).

7.2 Aplicação Merge Sort

O objetivo deste experimento é ilustrar a operação do sistema MPSoC, através de um aplicação paralela simples, *Merge Sort*. A aplicação é descrita por um grafo, ilustrado na Figura 68(a), sendo função do mestre do sistema alocar as tarefas. A tarefa t_1 tem por função apenas enviar os elementos a serem ordenados para as tarefas t_2 e t_3, e depois receber e unir (operação *merge*) os resultados parciais. As tarefas t_2 e t_3 realizam o ordenamento (algoritmo *bubble sort*) dos dados recebidos. As três tarefas são alocadas estaticamente.



(a) (b)

Figura 68 – (a) Grafo da aplicação merge sort com três tarefas comunicantes; (b) Posicionamento dos diferentes processadores na rede.

A Tabela 12 apresenta o tempo de execução para ordenar um vetor de 400 elementos, utilizando-se diferentes posicionamentos para as tarefas.

Os resultados para os sistemas monoprocessado (mapeamento M1, tempo total igual a 1.232.349 ciclos de relógio) e multiprocessado (mapeamento M4, tempo total igual a 574.643 ciclos de relógio), correspondem ao esperado, ou seja, redução do tempo de processamento (*speed-up* igual a 2,14 – divisão entre $T_{end}t_1$ de M1 por $T_{end}t_1$ de M4).

Tabela 12 – Tempos de execução (em ciclos de relógio) para um vetor de 400 posições.

Mapeamento				$T_{start}t_1$	$T_{start}t_2$	$T_{end}t_2$	$T_{start}t_3$	$T_{end}t_3$	$T_{end}t_1$	Observação
	t_1	t_2	t_3							
M1	P1	P1	P1	13.553	18.171	1.052.672	33.606	1.081.752	1.232.349	Sistema Monoprocessado
M2	P1	P6	P6	13.226	18.116	819.462	26.371	838.330	987.167	Tarefas de ordenação parcial juntas
M3	P1	P1	P6	13.585	17.804	782.797	22.745	325.295	941.676	Ordenações parciais separadas
M4	P1	P3	P6	13.210	17.583	417.903	22.513	310.145	574.643	Tarefas Distribuídas

Esperar-se-ia tempos semelhantes para os mapeamentos M1 e M2 e para M3 e M4. Nos mapeamentos M1 e M2 as ordenações parciais ocorrem no mesmo processador, caracterizando uma operação seqüencial. Para os mapeamentos M3 e M4 as ordenações parciais ocorrem em processadores distintos, com a conseqüente operação em paralelo dos algoritmos *bubble sort*.

Para a análise destes resultados é preciso compreender como ocorre a comunicação entre as tarefas. A Figura 69 ilustra parte do código da tarefa t_1. Como pode ser observado na Figura, t_1 envia os dados para as tarefas de ordenação (linhas 1 a 4) e depois aguarda a recepção dos dados.

```
          // inicializa vetores

1)        while(!WritePipe(&msg1,2));  //envia msg1 para tarefa2
2)        while(!WritePipe(&msg2,2));  //envia msg1 para tarefa2

3)        while(!WritePipe(&msg3,3));  //envia msg1 para tarefa3
4)        while(!WritePipe(&msg4,3));  //envia msg1 para tarefa3

5)        puts("vetores dist.");       // debug


6)        while(!ReadPipe(&msg1,2)     //aguarda vetor ordenado da tarefa 2
7)        while(!ReadPipe(&msg2,2))    //aguarda vetor ordenado da tarefa 2
8)        while(!ReadPipe(&msg2,3))    //aguarda vetor ordenado da tarefa 3
9)        while(!ReadPipe(&msg2,3))    //aguarda vetor ordenado da tarefa 3

          // realiza o merge
```

Figura 69 – Código parcial da tarefa t_1.

Observando os tempos de execução dos mapeamentos M3 e M4, o resultado esperado seria um desempenho do mapeamento M3 próximo ao desempenho de M4, pois as rotinas de ordenação estão em processadores distintos (P1 e P6, apesar de P1 estar compartilhado entre t_1 e t_2). Neste mapeamento, M3, toda vez que a tarefa t_1 é escalonada, há uma tentativa para leitura de dados da tarefa t_2 (linha 6 da Figura 69). Uma vez que t_2 ainda não concluiu, a tentativa de leitura apenas consome ciclos, não realizando trabalho efetivo. Dado que t_2 conclui em 782.797 ($T_{end}t_2$), percebe-se uma alta sobrecarga do *microkernel* devido a leituras que não retornam dados.

Analisando a Tabela 12, pode-se inferir os seguintes tempos (aproximados):

- T_i - tempo de inicialização (alocação estática das tarefas): 20.000 ciclos de relógio
- T_s - tempo de ordenação parcial: 300.000 ciclos de relógio ($T_{end}t_3$ - $T_{start}t_3$, nos mapeamentos M3 e M4).
- T_m - tempo de *merge* dos resultados: 150.000 ciclos de relógio (($T_{end}t_1$ – max($T_{start}t_2$, $T_{start}t_3$), em todos os mapeamentos).

De posse destes tempos, pode-se calcular o tempo ideal de execução para os mapeamentos seqüencial e paralelo, como ilustrado na Tabela 13.

Tabela 13 - Tempos total (T_{total}) e ideal (T_{ideal}) de execução para os mapeamentos M1 e M4.

Mapeamento				T_{total}	T_{ideal}	Diferença	Observação
	t_1	t_2	t_3				
M1	P1	P1	P1	1.232.349	760.000	472.349	$T_{total} = T_i + 2 * T_s + T_m$
M4	P1	P3	P6	574.643	470.000	104.643	$T_{total} = T_i + T_s + T_m$

A diferença entre os tempos total e ideal é explicada por dois fatores:

a) Leituras sem sucesso geram chamadas de sistema sobrecarregando a rede e/ou o processador. A forma utilizada nos experimentos para minimizar esta sobrecarga foi adicionar um atraso entre cada solicitação de leitura:

```
while(!ReadPipe(&msg2,3)) {
    for(k=0;k<DELAY;k++);    // DELAY IGUAL A 200
}
```

A solução efetiva para minimizar a sobrecarga é realizar a leitura de forma bloqueante, não escalonando a tarefa novamente enquanto não forem enviados dados à esta tarefa. Esta otimização está entre as atividades a serem realizadas em trabalhos futuros.

b) No caso do mapeamento M1 também há uma sobrecarga devido ao escalonamento da tarefa t_1, quando esta requer um dado de t_2. Tornando a leitura bloqueante este escalonamento não ocorrerá enquanto não forem enviados dados à t_2.

A Figura 70 detalha a execução das tarefas, através do sinal page, no mapeamento paralelo, M3. Nesta Figura deve-se observar:

- O número de ciclos de relógio reduzido para configurar as tarefas (T_i).
- A sobrecarga pelas operações de leitura sem sucesso. Este tempo pode ser observado entre o término da tarefa t_3 e o início de execução do *merge*, correspondendo a 110.000 ciclos de relógio (sobrecarga apresentada na Tabela 13).

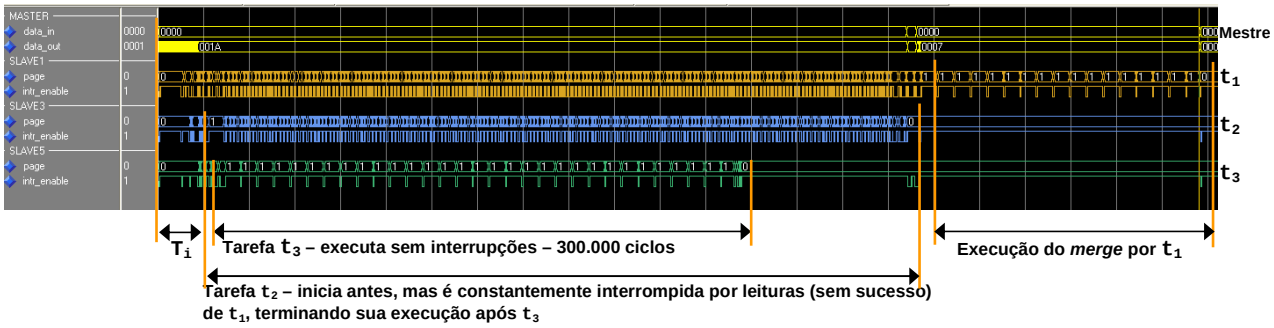


Figura 70 – Execução das tarefas t_1, t_2 e t_3 com a sobrecarga no processadores em que cada uma executa.

Aumentando-se o tamanho do vetor a ordenar para 1000 posições, o *speed-up* manteve-se aproximadamente igual (2,31).

Tabela 14 – Tempos de execução (em ciclos de relógio) para um vetor de 1000 posições.

Mapeamento				$T_{start}t_1$	$T_{start}t_2$	$T_{end}t_2$	$T_{start}t_3$	$T_{end}t_3$	$T_{end}t_1$	Observação
	t_1	t_2	t_3							
M1	P1	P1	P1	25.855	32.367	6.309.072	47.802	6.395.299	6.772.274	Sistema Monoprocessado
M2	P1	P6	P6	25.528	32.312	5.030.808	42.457	5.088.672	5.456.931	Tarefas de ordenação parcial juntas
M3	P1	P1	P6	25.887	32.000	4.640.494	38.831	1.790.426	5.039.047	Ordenações parciais separadas
M4	P1	P3	P6	25.512	31.779	2.540.956	38.177	1.783.306	2.929.408	Tarefas Distribuídas

Este experimento permitiu validar a plataforma MPSoC para uma aplicação multi-tarefa, com diferentes mapeamentos. A transferência de mensagens através da rede tem baixo impacto no desempenho, dado o elevado tempo de processamento das tarefas. Este experimento também mostrou que o mecanismo de comunicação entre tarefas pode ser otimizado, tornando a leitura bloqueante.

7.3 Aplicações Paralelas

O objetivo deste experimento é validar a correta execução de aplicações paralelas com tarefas comunicantes sob a plataforma. São utilizadas três aplicações, sendo duas a aplicação *Merge Sort*, apresentada na Seção anterior. A terceira é uma aplicação simples, composta por 4 tarefas. O grafo desta aplicação é mostrado na Figura 71. As tarefas A e B enviam, respectivamente, 3 e 4 vetores de 30 elementos para a tarefa C. Esta, por sua vez, repassa estes vetores para a tarefa D.

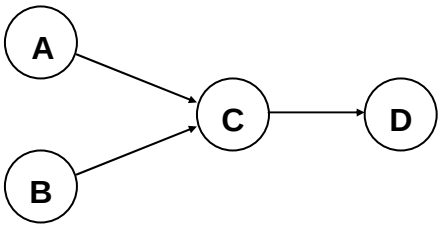


Figura 71 – Grafo da terceira aplicação utilizada no experimento.

Foram utilizados três diferentes mapeamentos, mostrados na Figura 72. As tarefas do primeiro *Merge Sort* são referenciadas por A_{M1}, B_{M1} e C_{M1}. As tarefas do segundo *Merge Sort* são referenciadas por A_{M2}, B_{M2} e C_{M2}. Por fim, as tarefas da terceira aplicação são referenciadas por A_C, B_C, C_C e D_C. O mapeamento M1 caracteriza-se pelo compartilhamento dos recursos pelas duas aplicações *Merge Sort*; o mapeamento M2 caracteriza-se pela independência de recursos para as aplicações *Merge Sort*; o mapeamento M3 compartilha parcialmente as tarefas do *Merge Sort*. A terceira aplicação é utilizada como "ruído" sobre as aplicações *Merge Sort.*

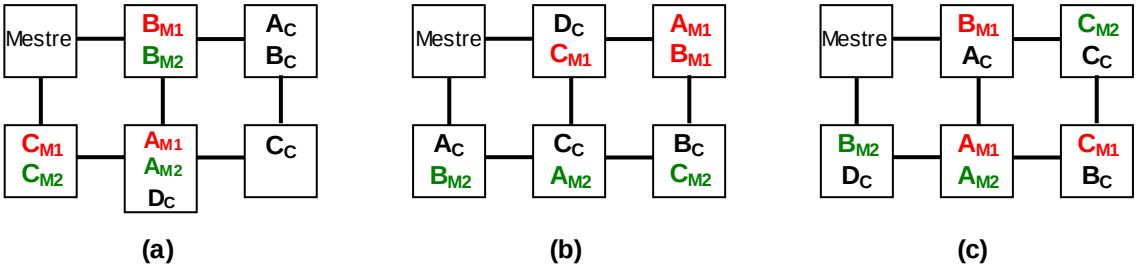


Figura 72 – Mapeamentos utilizados para execução de três aplicações paralelas. (a) M1; (b) M2; (c) M3.

A Tabela 15 apresenta os resultados da execução das três aplicações com os diferentes mapeamentos. Os resultados correspodem ao número de ciclos de relógio gastos para a execução de cada aplicação. No mapeamento `M1`, as duas aplicações *Merge Sort* possuem o mesmo mapeamento, concorrendo pelos mesmos recursos. Dessa forma, seus resultados são praticamente iguais (`1.056.667` e `1.065.460`). A terceira aplicação possui um tempo de vida curto, comparado às aplicações *Merge Sort* (`194.301`).

Tabela 15 – Número de ciclos gastos na execução das três aplicações com diferentes mapeamentos.

Mapeamento	*Merge Sort* 1	*Merge Sort* 2	Terceira aplicação
M1 (Figura 72(a))	1.056.667	1.065.460	194.301
M2 (Figura 72(b))	973.541	600.171	232.513
M3 (Figura 72(c))	880.908	858.306	253.690

No mapeamento `M2`, o segundo *Merge Sort* teve seu tempo de execução reduzido, quase igual ao tempo mínimo observado na Tabela 12 (574.643), pois as tarefas A_C, B_C e C_C da terceira aplicação, com as quais ele compartilha processamento, terminam sua execução deixando os processadores disponíveis exclusivamente para a execução das tarefas do segundo *Merge Sort* (A_{M2}, B_{M2} e C_{M2}). A terceira aplicação teve seu desempenho degradado: no mapeamento `M1` seu tempo de execução foi de `194.301` ciclos de relógio; no mapeamento `M2`, foi de `232.513`. Isso se deve ao fato de que, no `M1`, a tarefa B_C compartilha o processamento com A_C, e ambas terminam cedo. Além disso, a tarefa C_C executa sozinha no processador onde está alocada. Já em `M2`, todas as tarefas da terceira aplicação compartilham processamento com tarefas das aplicações *Merge Sort*, as quais possuem um tempo de vida maior e, conseqüentemente, continuarão a utilizar o processador enquanto a terceira aplicação não termina sua execução.

Ainda no mapeamento `M2`, o primeiro *Merge Sort* teve seu desempenho melhorado em relação ao mapeamento `M1`. A razão disso é que, em `M1`, ele compartilha os mesmos recursos com o segundo *Merge Sort*. Em `M2`, as tarefas A_{M1} e B_{M1} compartilham o mesmo processador, e a tarefa C_{M1} compartilha o processador com a tarefa C_C, da terceira aplicação, que termina sua execução mais cedo.

No mapeamento `M3`, as aplicações *Merge Sort* tiverem tempos praticamente iguais (`880.908` e `858.306`), pois as tarefas `B` (B_{M1} e B_{M2}) e `C` (C_{M1} e C_{M2}) compartilham recursos com

tarefas da terceira aplicação que, mais uma vez, terminam sua execução cedo fazendo com que as tarefas B e C das aplicações *Merge Sort* utilizem o processador com exclusividade. A alocação das tarefas A (A_{M1} e A_{M2}) no mesmo processador, contribui também para os resultados próximos.

A simulação de cada experimento gera um relatório para cada processador. Este relatório contém os resultados da execução de cada tarefa alocada no processador. Através destes relatórios, pôde-se obsrevar a correta execução das tarefas.

7.4 Codificação parcial MPEG

Esta Seção apresenta a execução de parte da codificação MPEG sob a plataforma implementada. A aplicação é composta por 5 tarefas, mostradas na Figura 73: t_1 envia a t_2 8 vetores, cada um composto por 128 bytes; t_2 recebe os vetores, executa o algoritmo IVLC e envia os resultados à t_3; t_3 recebe os vetores de t_2, executa o algoritmo IQUANT e envia os resultados à t_4; t_4 recebe os vetores de t_3, executa o algoritmo IDCT e envia os resultados à t_5.

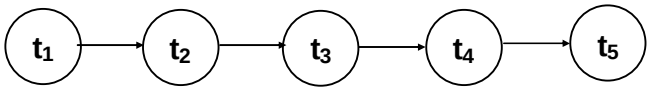


Figura 73 – Grafo da aplicação MPEG.

Uma vez que MPEG é uma aplicação de tempo real, é importante a avaliação da vazão que o sistema pode fornecer. Dessa forma, os resultados desta Seção são apresentados na forma da taxa de recepção dos dados pela última tarefa (t_5). Ou seja, é conhecido o instante de tempo T_i, no qual a tarefa t_5 recebe o primeiro byte de dados (primeiro byte do primeiro vetor) e o instante de tempo T_f, no qual t_5 recebe o último byte de dados (último byte do oitavo vetor). Conhecendo estes tempos pode-se calcular a vazão do sistema, para a recepção de dados por t_5. A Tabela 16 apresenta estes instantes de tempo para três diferentes mapeamentos. No mapeamento M1, as tarefas t_1, t_3 e t_5 compartilham o processador P1 e as tarefas t_2 e t_4 compartilham o processador P2. Para este mapeamento, é utilizada a alocação estática de tarefas. Nos mapeamentos M2 e M3 todas as tarefas estão distribuídas em 5 processadores. Em M2 utiliza-se alocação estática de tarefas e em M3 utiliza-se alocação dinâmica.

Tabela 16 – Instantes de tempo T_i e T_j para recepção de dados da tarefa t_5.

Mapeamento							$T_i - t_5$	$T_f - t_5$
	t_1	t_2	t_3	t_4	t_5	Estratégia		
M1	P1	P2	P1	P2	P1	Estática	25.785.530ns	38.561.690ns
M2	P1	P2	P3	P4	P5	Estática	1.320.760ns	5.957.910ns
M3	P1	P2	P3	P4	P5	Dinâmica	1.450.380ns	5.811.990ns

A Tabela 17 apresenta a vazão do sistema para os três mapeamentos. Os dados que chegam na tarefa t_5 totalizam em 1024 bytes (8 vetores de 128 bytes). Dessa forma, a vazão é calculada pela expressão: 1024/ ($T_i - T_f$). O mapeamento M1 apresentou menor vazão (85,3KB/s), pois as

cinco tarefas da aplicação concorrem pelo uso de dois processadores. Dessa forma, os resultados demoram mais tempo para chegar na tarefa t_5. Os mapeametos M2 e M3 obtiveram vazões equivalentes à quase o triplo da vazão obtida no mapeamento M1. Isso se deve ao fato de que as tarefas executam em paralelo e não concorrem por recursos.

Tabela 17 – Vazão do sistema para a recepção de dados da tarefa t_5.

Mapeamento	Vazão
M1	85,3KB/s
M2	222,6KB/s
M3	238,1KB/s

A Figura 74 apresenta o processo de alocação das tarefas para o mapeamento M1. Na Figura, o sinal data_in representa os dados provenientes da NoC que chegam no processador. O sinal intr_signal representa uma interrupção do processador. Observa-se na linha do tempo, a alocação seqüencial das cinco tarefas: t_1 em SLAVE1, t_2 em SLAVE2, t_3 em SLAVE1, t_4 em SLAVE2 e t_5 em SLAVE1. Observa-se ainda, que a tarefa t_1 não é interrompida no início de sua execução, pois a tarefa t_2 ainda não foi alocada e, portanto, não faz requisições de mensagem à t_1.

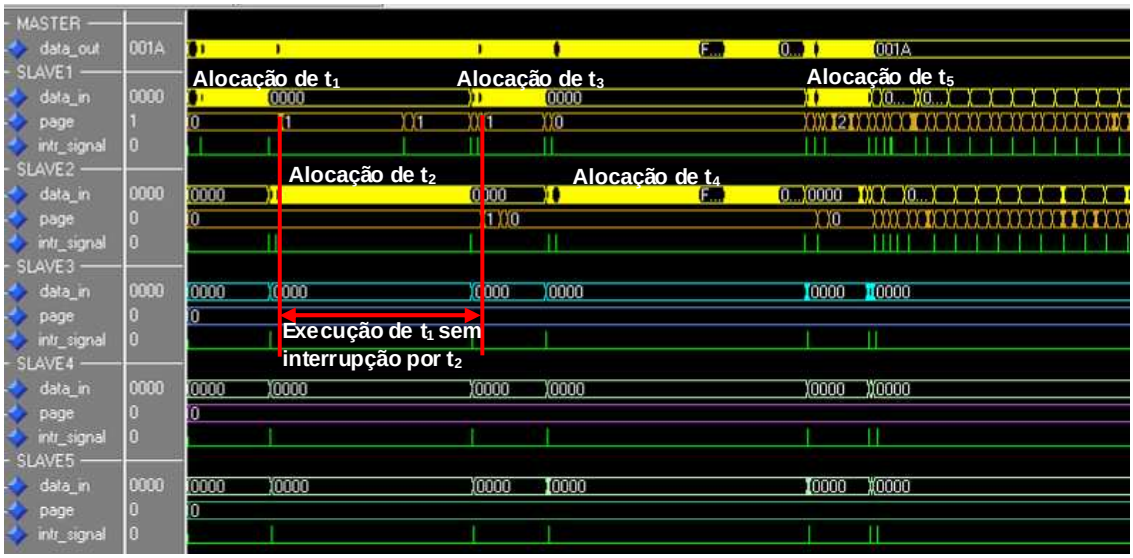


Figura 74 – Alocação de tarefas para o mapeamento M1, com alocação estática.

O término da execução das tarefas para o mapeamento M1 é mostrado na Figura 75. Observa-se a conclusão de todas as tarefas em SLAVE2 (page=0). Em SLAVE1, a tarefa t_5 (page=3) termina sua execução fazendo a impressão dos resultados.

A Figura 76 apresenta o processo de alocação das tarefas para o mapeamento M2. Observa-se na linha do tempo, a alocação seqüencial das cinco tarefas: t_1 em SLAVE1, t_2 em SLAVE2, t_3 em SLAVE3, t_4 em SLAVE4 e t_5 em SLAVE5. Neste mapeamento, a tarefa t_2 é interrompida várias vezes, desde o início de sua execução, pela tarefa t_3 que faz requisições de mensagens ainda não disponíveis (legenda 1 da Figura).

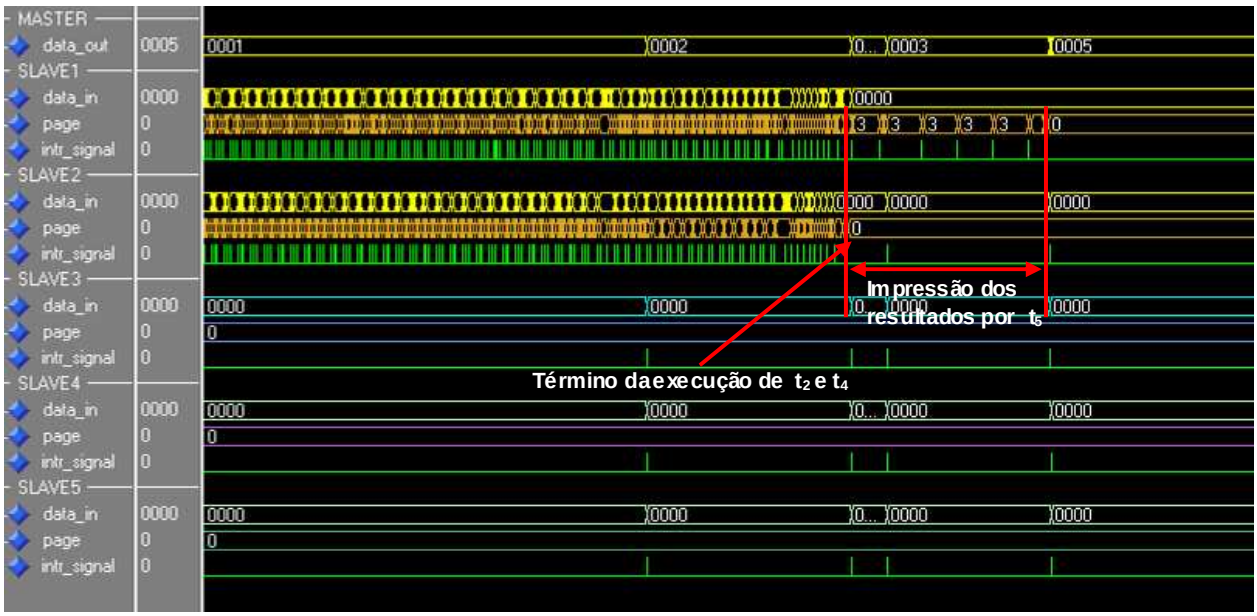


Figura 75 – Término da aplicação para o mapeamento M1.

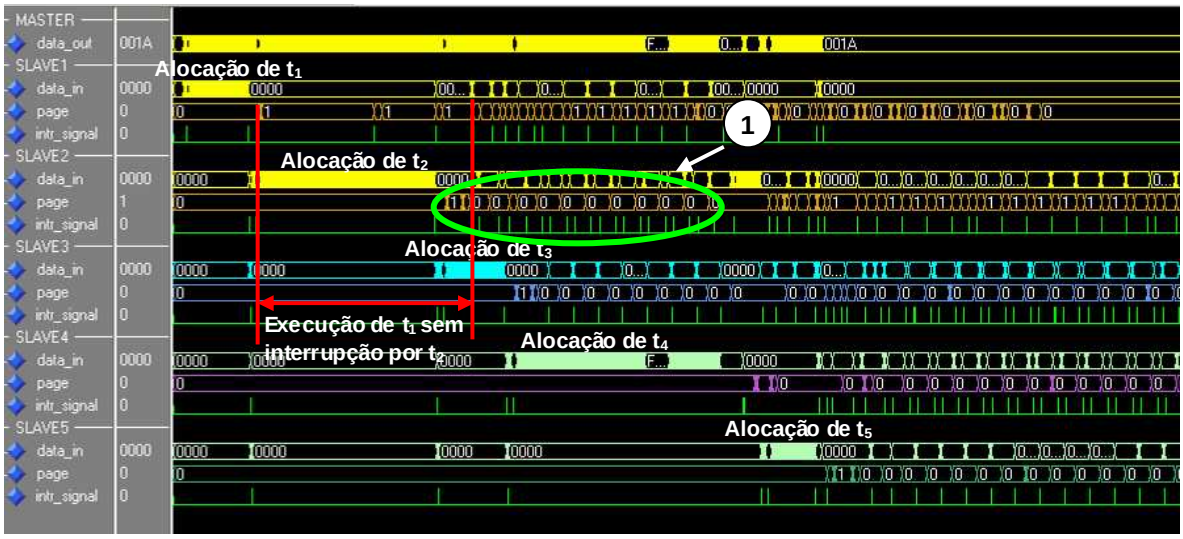


Figura 76 – Alocação de tarefas no mapeamento M2, com alocação estática.

O término das tarefas no mapeamento M2 é mostrado na Figura 77. A tarefa t_1 finaliza sua execução bastante tempo antes das outras tarefas, não aparecendo na Figura. Observa-se o término seqüencial das tarefas. A última tarefa a finalizar a execução é a tarefa t_5 em SLAVE5 (page=1), com a impressão dos resultados.

A Figura 78 apresenta o processo de alocação das tarefas para M3. Este mapeamento é igual ao utilizado em M2, contudo, a alocação é dinâmica. Isto significa que: a tarefa t_2 será alocada apenas quando for requisitada por t_1; a tarefa t_3 será alocada apenas quando for requisitada por t_2; a tarefa t_4 será alocada apenas quando for requisitada por t_3; por fim, a tarefa t_5 será alocada apenas quando for requisitada por t_4. Diferentemente do mapeamento M2, no qual a tarefa t_2 é interrompida sucessivas vezes pela tarefa t_3, no mapeamento M3, t_2 executa um tempo sem interrupções (legenda 1 da Figura) e requisita t_3. Após t_3 ser alocada e iniciar sua execução, a t_2 passa a ser interrompida (legenda 2 da Figura). O término das tarefas para o mapeamento M3 é o

mesmo para M2, mostrado na Figura 77.

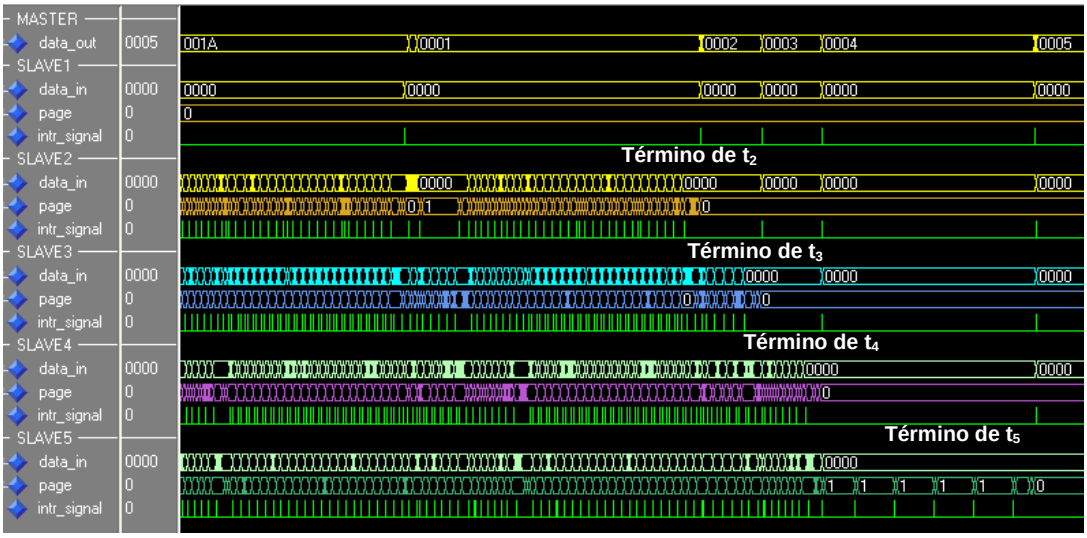


Figura 77 – Término da aplicação MPEG, para o mapeamento M2.

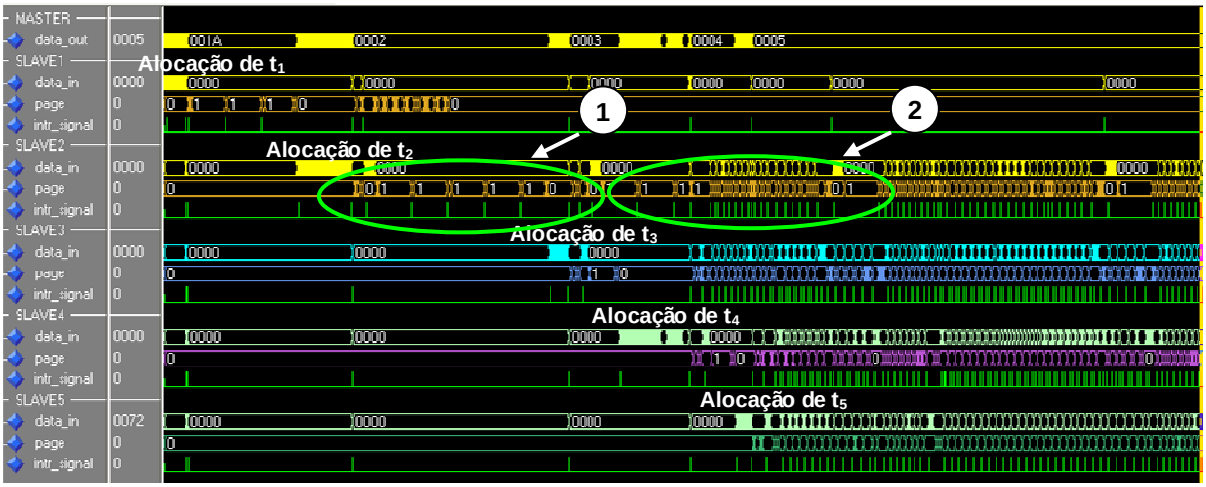


Figura 78 – Alocação de tarefas no Mapeamento M3, com alocação dinâmica.

8 CONCLUSÕES

Sistemas multiprocessados em *chip*, MPSoC, são uma tendência no projeto de sistemas embarcados. Entretanto, não há plataformas completas disponíveis em domínio público para realizar pesquisas no tema. Freqüentemente, os temas são tratados de forma isolada. Por exemplo, há inúmeros trabalhos situados dentro dos temas de rede intra-chip, processadores embarcados e sistemas operacionais embarcados. Contudo, trabalhos contendo a integração de todo o sistema e avaliando seu desempenho, são escassos.

A contribuição maior desta Dissertação foi o desenvolvimento de uma plataforma MPSoC completa, multiprocessada e multitarefa, com interconexão por rede intra-chip. Esta plataforma será de domínio público, disponível aos grupos de pesquisa que trabalham nas áreas de NoC, MPSoC, sistemas operacionais embarcados, entre outros.

O trabalho iniciou a partir das descrições VHDL da rede Hermes e do processador Plasma. O desenvolvimento concomitante do hardware para integrar os componentes principais e do software para suportar as aplicações concorrentes resultaram nas seguintes contribuições: (*i*) de hardware: arquitetura do nodo de processamento – composto pela interface de rede, DMA, processador; (*ii*) de software: desenvolvimento do *microkernel* para suporte a aplicações concorrentes e alocação estática e dinâmica de tarefas.

A plataforma MPSoC desenvolvida foi descrita em VHDL sintetizável (nível RTL), e validada por simulação VHDL. Apesar do tempo de simulação ser elevado, esta estratégia permite: (*i*) simular a arquitetura que realmente executará no hardware, não sendo um modelo abstrato, como muitas vezes encontra-se na literatura; (*ii*) avaliar com precisão, no nível de ciclo de relógio, o desempenho dos mecanismos empregados na comunicação, transferência de dados, chaveamento de contexto, entre outros.

8.1 Trabalhos Futuros

Enumera-se abaixo um conjunto de trabalhos que podem ser executados na seqüência do desenvolvimento desta plataforma MPSoC.

1. **Prototipação do sistema.** Para a execução de tarefas mais complexas, o tempo de simulação torna-se muito elevado. Visando avaliar o desempenho do sistema para aplicações complexas, deve-se prototipar o mesmo em FPGA.

2. **Leitura bloqueante.** A leitura não bloqueante de mensagens empregada neste trabalho faz com que, no caso de mensagem inexistente, muitos pacotes de requisição e resposta negativa (`no_message`) sejam gerados e enviados pela rede. Além de congestionar a rede, muito processamento da CPU é gasto para tratar estes pacotes, degradando o desempenho do sistema. Visando solucionar este problema, a leitura de mensagens deverá ser bloqueante, fazendo com

que uma tarefa seja suspensa no momento em que ela requisita uma mensagem e volte a ser escalonada somente após a mensagem ser recebida.

3. **Canais virtuais.** A NoC utilizada para realizar a interconexão dos módulos não possui canais virtuais, pois simplifica o projeto do hardware. Contudo, em redes maiores, haverá congestão nos canais da rede. Dado que já existe implementação da Hermes com canais virtuais, pode-se empregar esta rede para redução da congestão da rede.

4. **MPSoC heterogêneo.** A plataforma desenvolvida neste trabalho é um MPSoC com processamento homogêneo, ou seja, todos os núcleos são processadores embarcados de propósito geral idênticos (processador Plasma). Como trabalho futuro está a inserção de módulos de hardware reconfigurável na rede, tornando o processamento heterogêneo. No repositório, além de tarefas de software, existirão *bistreams* de reconfiguração parcial. Para tanto, faz-se necessária a inclusão de um controlador de configuração e infra-estrutura para conexão destes módulos.

5. **Mecanismos de avaliação de carga.** Visando distribuir a carga de trabalho do sistema, devem ser implementados mecanismos de avaliação de carga. Estes mecanismos podem considerar a carga de trabalho local de processadores e o tráfego nos canais da rede.

6. **Tornar a aplicação de alocação uma tarefa genérica.** Na plataforma desenvolvida, um nodo mestre é designado a executar a aplicação de alocação de tarefas. Trabalhos futuros compreendem tornar a aplicação de alocação uma tarefa genérica, permitindo que ela compartilhe tempo de execução com as demais aplicações do sistema. Além disso, sugere-se incluir na aplicação de alocação, **funções mais elaboradas de distribuição de carga**, que levem em conta parâmetros como volume de comunicação entre as tarefas, posicionamento das tarefas na rede de forma a minimizar o congestionamento na rede, o tempo de execução das tarefas, consumo de energia, dentre outros.

7. A **migração de tarefas** é um mecanismo a ser explorado dentro do contexto da alocação dinâmica. Este mecanismo suspende a execução de uma tarefa em um nodo *x* e a envia para um nodo *y*, onde sua execução é retomada. A migração de tarefas é um tema bastante explorado na área de Sistemas Paralelos e Distribuídos. Contudo, a aplicação desta abordagem em MPSoCs está sendo estudada de forma a empregar seus benefícios na área de sistemas embarcados. A migração de tarefas pode ser justificável para otimizar o desempenho do sistema, considerando critérios como, por exemplo, carga de trabalho nos processadores e a carga da comunicação nos enlaces da rede.

8. **Coerência de *cache***. Uma vez que tarefas paralelas podem trabalhar sobre os mesmos dados, é

necessária a funcionalidade de coerência de *cache*. Com isso, diferentes tarefas poderão compartilhar dados, fazendo com que as alterações realizadas por uma tarefa sejam conhecidas por todas as outras.

9. **Gerência de Memória**. A gerência de memória empregada neste trabalho dispõe apenas de um mecanismo de paginação, o qual faz a geração de endereços físicos de acordo com a página onde se encontra o fluxo de execução. Sugere-se incluir a funcionalidade de alocação e liberação dinâmica de memória.
10. Uma **interface gráfica com o usuário** está sendo desenvolvida de forma a facilitar o projeto e execução de aplicações sobre a plataforma. Até o presente momento, esta interface possibilita carregar diferentes aplicações e alocar cada uma de suas tarefas escolhendo os nodos escravos. As dimensões da rede são fixas (3x2) bem como o número de processadores (6). Sugere-se permitir ao usuário criar uma plataforma com uma rede de tamanho parametrizável, escolhendo o núcleo a ser conectado em cada nodo da rede: um processador ou um módulo de hardware reconfigurável.

REFERÊNCIAS BIBLIOGRÁFICAS

[ARM06] ARM, Capturado em: http://www.arm.com/. Acesso em 03/11/06.

[BEN01] BENINI, L.; De MICHELI, G. "**Powering Networks on Chip**". International Symposium on System Synthesis, 2001, pp. 33 –38.

[BEN02] BENINI, L.; De MICHELI, G. "**Networks On Chip: A New SoC Paradigm**". Computer, v. 35(1), 2002, pp. 70-78.

[BER06] BERTOZZI, S.; ACQUAVIVA, A.; BERTOZZI, D.; POGGIALI, A. "**Supporting task migration in multi-processor systems-on-chip: a feasibility study**". In: Design, Automation and Test in Europe (DATE'06), 2006, pp. 15-20.

[BRI04] BRIÃO, E.; "**Reconfiguração Parcial e Dinâmica para Núcleos de Propriedade Intelectual**". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, FACIN PUCRS, 2004, 147 p.

[CAR04] CARVALHO, E.; "**RSCM – Controlador de Configurações para Sistemas de Hardware Reconfigurável**". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, FACIN PUCRS, 2004, 153 p.

[DAY83] DAY, J.; ZIMMERMMAN, H. "**The OSI reference model**". Proceedings of IEEE, vol. 71, 1983, pp. 1334-1340.

[ECO06] eCos. Capturado em http://ecos.sourceware.org/. Acesso em 01/07/06.

[EPO06] EPOS – Embedded Parallel Operating System. Capturado em http://epos.lisha.ufsc.br/, Julho 2006.

[GAP06] GAPH – Hardware Design Support Group. Disponível em: http://www.inf.pucrs.br/~gaph.

[GUE00] GUERRIER, P.; GREINER, A., "**A Generic Architecture for On-Chip Packet-Switched Interconnections"**. In: Design, Automation and Test in Europe (DATE'00), 2000, pp. 250-256.

[HEN03] HENKEL, J. "**Closing the SoC Design Gap**". Computer v.36(9), Setembro 2003, pp. 119-121.

[HEN98] HENESSY, L.; PATTERSON, D. "**Computer Organization and Design : the Hardware/Software Interface**". San Francisco, CA: Morgan Kaufmann, 1998, 896 p.

[HU04] HU, J.; MARCULESCU, R. "**Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints**". In: Design, Automation and Test in Europe (DATE'04), 2004, pp 10234.

[IBM06] IBM. Capturado em: http://www.ibm.com/br/. Acesso em 29/10/06.

[JER04] JERRAYA, A.; WOLF, W. "**Multiprocessors Systems-on-Chips**". Morgan Kaufman Publishers, 2004, 608 p.

[JER05] JERRAYA, A.; TENHUNEN, H.; WOLF, W. "Guest Editors' **Introduction: Multiprocessors Systems-on-Chips**". Computer v.38(7), Julho 2005, pp. 36-40.

[KUM02] KUMAR, S.; JANTSCH, A.; MILLBERG, M.; OBERG, J.; SOININEN, J.; FORSELL, M.; TIENSYRJA, K. "**A Network on Chip Architecture and Design Methodology**". In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02), 2002, pp. 105-112.

[MAR05] MARCON, C.; CALAZANS, N.; MORAES, F.; SUSIN, A.; REIS, I.; HESSEL, F. "**Exploring**

NoC Mapping Strategies: An Energy and Timing Aware Technique". In: Design, Automation and Test in Europe (DATE'05), 2005, pp.502-507.

[MEL05] MELLO, A.; MÖLLER, L. CALAZANS, N.; MORAES, F. "**MultiNoC: A Multiprocessing System Enabled by a Network on Chip**". In: Design, Automation and Test in Europe (DATE '05), 2005, pp. 234-239.

[MIP06] MIPS Technologies Inc. Capturado em: http://www.mips.com/. Acesso em 20/10/06.

[MOL05] MÖLLER, L. "**Sistemas Dinamicamente Reconfiguráveis com Comunicação via Redes Intra-chip**". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2005, 156 p.

[MOR04] MORAES, F.; CALAZANS, N.; MELLO, A.; MÖLLER, L.; OST, L. "**HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip**". Integration the VLSI Journal, Amsterdam, v. 38(1), 2004, p. 69-93.

[NGO06] NGOUANGA, A. ; SASSATELI, G. ; TORRES, L. ; GIL, T. ; SOARES, A. B. ; SUSIN, A. A. "**A contextual resources use: a proof of concept through the APACHES platform**". In: IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS06), 2006, pp.42-47.

[NOL05] NOLLET, V.; AVASARE, P.; MIGNOLET, J-Y.; VERKEST, D. "**Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles**". In: Design, Automation and Test in Europe Conference (DATE'05), 2005, pp. 234-239.

[OPE06] OPENCORES. Capturado em http://www.opencores.org/, Julho 2006.

[OST03] OST, L. C. "**Redes Intra-Chip Parametrizáveis com Interface Padrão para Síntese em Hardware**". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2003, 102p.

[OZT06] OZTURK, O.; KANDEMIR, M.; SON, S. W.; KARAKOY, M. "**Selective code/data migration for reducing communication energy in embedded MpSoC architectures**". In: 16th ACM Great Lakes Symposium on VLSI (GLSVLSI '06), 2006, pp. 386-391.

[PLA06] PLASMA processor. Capturado em: http://www.opencores.org/projects.cgi/web/mips/overview, Julho 2006.

[RTE06] RTEMS. Capturado em http://www.rtems.com, Julho 2006.

[RUG06] RUGGIERO, M.; GUERRI, A.; BERTOZZI, D.; POLETTI, F.; MILANO, M. "**Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip**" In: Design, Automation and Test in Europe (DATE'06), 2006, pp. 3-8.

[SGR01] SGROI, M.; SHEETS, M.; MIHAL, A.; KEUTZER, K.; MALIK, S.; RABAEY, J.; SANGIOVANNI-VINCENTELLI, A. "**Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design**". In: Design Automation Conference (DAC'01), 2001, pp. 667-672.

[SIL00] SILBERSCHATZ, A. "**Applied Operating System Concepts**". New York, NY: John Wiley & Sons, 2000, 840 p.

[SIL95] SILBERSCHATZ, A; GALVIN P. B. "**Operating System Concepts**". Addison-Wesley, 1995, 780 p.

[STR06] STREICHERT, T.; HAUBELT, C.; TEICH, J. ; "**Dynamic Task Binding of**

Hardware/Software Reconfigurable Networks". In: Simpósio Brasileiro de Concepção de Circiutos Integrados (SBCCI'06), 2006.

[TAN06] TANURHAN, Y. "**Processors and FPGAs Quo Vadis?**". Computer, v. 39(11), 2006, pp. 108-110.

[TAN97] TANENBAUM, A. "**Operating Systems: Design and Implementation**". New Jersey: Prentice-Hall, 1997, 939 p.

[UCL06] uCLINUX – Embedded Linux/Microcontroller Project. Capturado em: http://www.uclinux.org, Julho 2006.

[VIR04] VIRK, K.; MADSEN, J. "**A system-level multiprocessor system-on-chip modelling framework**". In: International Symposium on System-on-Chip (ISSoC), 2004, pp. 81-84.

[WIK06] WIKPEDIA. "**MIPS architecture**". Capturado em http://en.wikipedia.org/wiki/MIPS_architecture. Acesso em 10/11/06.

[WIN01] WINGARD, D. "**MicroNetwork-based Integration for SoCs**". In: Design Automation Conference (DAC'01), 2001, pp. 673-677.

[WOL01] WOLF, W. "**Computer as Components: Principles of Embedded Computing System Design**". Academic Press, 2001.

[WOL04] WOLF, W. "**The Future of Multiprocessors Systems-on-Chips**". In: Design Automation Conference (DAC'04), 2004, pp. 681-685.

[ZEF03] ZEFERINO, C. A. "**Redes-em-Chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho**" Tese de Doutorado, PPGC-UFRGS, 2003, 194p.

ANEXO A - DESCRIÇÃO DA ENTIDADE PLASMA

```
-- TITLE: Plasma (CPU core with memory)
-- FILENAME: plasma.vhd
--
-- Memory Map:
--   0x00000000 - 0x0000ffff   Internal RAM (16KB)
--   0x10000000 - 0x000fffff   External RAM (1MB)
--   Access all Misc registers with 32-bit accesses
--   0x20000000  Uart Write (will pause CPU if busy)
--   0x20000000  Uart Read
--   0x20000010  IRQ Mask
--   0x20000020  IRQ Status
--   0x20000030
--   0x20000050
--   0x20000060  Counter

--   0x20000100 - Wrapper Status Reading
--   0x20000110 - Wrapper Status Sending
--   0x20000120 - Wrapper Read Data
--   0x20000130 - Wrapper Write Data
--   0x20000140 - Wrapper Configuration
--   0x20000150 - Wrapper Packet ACK
--   0x20000160 - Wrapper Packet NACK
--   0x20000170 - Wrapper Packet END

--   0x20000200 - Set DMA Size
--   0x20000210 - Set DMA Address
--   0x20000220 - Start DMA
--   0x20000230 - DMA ACK

--   0x20000300 - Tick Counter

--   IRQ bits:
--      7
--      6
--      5   NoC
--      4   DMA
--      3   Counter(18)
--      2  ^Counter(18)
--      1  ^UartWriteBusy
--      0   UartDataAvailable
------------------------------------------------------------------------

library ieee;
use work.mlite_pack.all;
use work.HermesPackage.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library unisim;
use unisim.vcomponents.all;

entity plasma is
   generic (
      memory_type      : string := "XILINX_X16"; --"DUAL_PORT_" "ALTERA_LPM";
      log_file         : string := "UNUSED";
      processor_type   : string := "");
   port (
      clock             : in  std_logic;
      reset             : in  std_logic;

      -- NoC Interface
      address_noc       : in  regmetadeflit;
      clock_tx          : out std_logic;
      tx                : out std_logic;
      data_out          : out regflit;
      credit_i          : in  std_logic;
      clock_rx          : in  std_logic;
      rx                : in  std_logic;
      data_in           : in  regflit;
      credit_o          : out std_logic;

      -- External Memory
      address           : out std_logic_vector(31 downto 2);
      data_write        : out std_logic_vector(31 downto 0);
      data_read         : in  std_logic_vector(31 downto 0);
      write_byte_enable : out std_logic_vector(3 downto 0);
      mem_pause_in      : in  std_logic);
end;


architecture plasma of plasma is
   signal address_reg         : std_logic_vector(31 downto 2);
   signal data_write_reg      : std_logic_vector(31 downto 0);
   <OUTROS SINAIS OMITIDOS – REQUISITAR FONTES À AUTORA DO TRABALHO>
```

```
   signal dma_data_read             : std_logic_vector(31 downto 0);
   signal dma_enable_internal_ram   : std_logic;


begin

   u1_cpu: mlite_cpu
      generic map (memory_type    => memory_type)
      port map (
         clk          => clock,
         reset_in     => reset,
         intr_in      => irq,

         mem_address  => cpu_mem_address,
         mem_data_w   => cpu_mem_data_write,
         mem_data_r   => cpu_mem_data_read,
         mem_byte_we  => cpu_mem_write_byte_enable,
         mem_pause    => cpu_mem_pause,
         current_page => current_page);

   u2_ram: ram
      generic map (memory_type => memory_type,
                   processor_type => processor_type)
      port map (
         clk              => clock,
         enable           => enable_internal_ram,
         write_byte_enable => dma_cpu_wb_enable,
         address          => cpu_mem_address(31 downto 2),
         address_write    => dma_cpu_address(31 downto 2),
         data_write       => dma_cpu_data_write,
         data_read        => data_read_ram);

   u3_uart: uart   -- UTILIZADA PARA FINS DE DEBUG
      generic map (log_file => log_file)
      port map(
         clk          => clock,
         reset        => reset,
         enable_read  => enable_uart_read,
         enable_write => enable_uart_write,
         data_in      => data_write_reg(7 downto 0),
         data_out     => data_read_uart,
         uart_read    => uart_read,
         uart_write   => uart_write,
         busy_write   => uart_write_busy,
         data_avail   => uart_data_avail);

   u4_noc: noc_interface
      port map(
         clock       => clock,
         reset       => reset,

         address     => address_noc,
         clock_tx    => clock_tx,
         tx          => tx,
         data_out    => data_out,
         credit_i    => credit_i,
         clock_rx    => clock_rx,
         rx          => rx,
         data_in     => data_in,
         credit_o    => credit_o,

         hold        => plasma_hold,
         send_av     => noc_send_av,
         read_av     => noc_read_av,
         intr        => noc_intr,
         send_data   => noc_send_data,
         read_data   => noc_read_data,
         packet_ack  => noc_packet_ack,
         packet_nack => noc_packet_nack,
         packet_end  => noc_packet_end,
         data_write  => noc_data_write,
         data_read   => noc_data_read,
         config      => noc_config
         );

   u5_dma: dma
      port map(
         clock          => clock,
         reset          => reset,
         set_address    => dma_set_address,
         set_size       => dma_set_size,
         start          => dma_start,
         read_ack       => dma_read_ack,
         read_av        => dma_read_av,
         intr           => dma_intr,
         intr_ack       => dma_intr_ack,
         address        => dma_mem_address,
         write_enable   => dma_write_enable,
         write_pause    => dma_write_pause,
         data_write     => dma_data_write,
         data_read      => dma_data_read
```

```
   );

write_byte_enable <= write_byte_enable_reg;
data_write <= data_write_reg;
address <= address_reg;

irq_status <= "00" & noc_intr & dma_intr &
              counter_reg(13) & not counter_reg(13) &
              not uart_write_busy & uart_data_avail;

irq <= '1' when (irq_status and irq_mask_reg) /= ZERO(7 downto 0) else '0';

cpu_mem_pause       <= plasma_hold or (uart_write_busy and enable_uart and write_enable);
cpu_read_data       <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "010010" and not write_enable
        = '1' else '0';

write_enable        <= '1' when write_byte_enable_reg /= "0000" else '0';
enable_misc         <= '1' when address_reg(30 downto 28) = "010" else '0';
enable_uart         <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "000000" else '0';
enable_uart_read    <= enable_uart and not write_enable;
enable_uart_write   <= enable_uart and write_enable;


noc_packet_ack <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "010101" and write_enable = '1' else
        '0';
noc_packet_nack <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "010110" and write_enable = '1'
        else '0';
noc_packet_end <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "010111" and write_enable = '1' else
        '0';
noc_send_data <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "010011" and write_enable = '1' else
        '0';
noc_read_data <= '1' when dma_read_ack = '1' or cpu_read_data = '1' else '0';
noc_data_write <= data_write_reg;

dma_read_av <= noc_read_av;
dma_data_read <= noc_data_read when dma_read_ack = '1' else data_write_reg;

dma_set_size <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "100000" and write_enable = '1' else
        '0';
dma_set_address <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "100001" and write_enable = '1'
        else '0';
dma_start <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "100010" and write_enable = '1' else '0';
dma_intr_ack <= '1' when enable_misc = '1' and address_reg(9 downto 4) = "100011" and write_enable = '1' else
        '0';

cpu_enable_internal_ram <= '1' when cpu_mem_address(30 downto 28) = "000" else '0';
dma_enable_internal_ram <= '1' when dma_mem_address(30 downto 28) = "000" else '0';
enable_internal_ram <= '1' when cpu_enable_internal_ram = '1' or dma_enable_internal_ram = '1' else '0';
dma_write_pause <= '1' when cpu_mem_write_byte_enable /= "0000" and cpu_enable_internal_ram = '1' else '0';
dma_cpu_address <= cpu_mem_address when cpu_mem_write_byte_enable /= "0000" and cpu_enable_internal_ram = '1'
        else dma_mem_address;
dma_cpu_data_write <= cpu_mem_data_write when cpu_mem_write_byte_enable /= "0000" and cpu_enable_internal_ram =
        '1' else dma_data_write;
dma_cpu_wb_enable <= cpu_mem_write_byte_enable when cpu_mem_write_byte_enable /= "0000" and
        cpu_enable_internal_ram = '1' else "1111" when dma_write_enable = '1' else "0000";

misc_proc: process(clock, reset, cpu_mem_address, address_reg, enable_misc,
                   data_read_ram, data_read_uart, cpu_mem_pause, data_read,
                   irq_mask_reg, irq_status,  write_enable,
                   counter_reg, cpu_mem_data_write, data_write_reg)
begin
   case address_reg(30 downto 28) is
      when "000" =>      --internal RAM
           cpu_mem_data_read <= data_read_ram;
      when "001" =>      --external RAM
           cpu_mem_data_read <= data_read;
      when "010" =>      --misc
         case address_reg(9 downto 4) is
         when "000000" =>        --uart
            cpu_mem_data_read <= ZERO(31 downto 8) & data_read_uart;
         when "000001" =>        --irq_mask
            cpu_mem_data_read <= ZERO(31 downto 8) & irq_mask_reg;
         when "000010" =>        --irq_status
            cpu_mem_data_read <= ZERO(31 downto 8) & irq_status;
         when "000110" =>        --counter
            cpu_mem_data_read <= counter_reg;
         when "010000" =>        --noc_read_av
            cpu_mem_data_read <= ZERO(31 downto 1) & noc_read_av;
         when "010001" =>        --noc_send_av
            cpu_mem_data_read <= ZERO(31 downto 1) & noc_send_av;
         when "010010" =>        --noc_read_data
            cpu_mem_data_read <= noc_data_read;
         when "010100" =>        --noc_config
            cpu_mem_data_read <= noc_config;
         when "110000" =>        --tickcounter
            cpu_mem_data_read <= tick_counter;
         when others =>
            cpu_mem_data_read <= ZERO;
         end case;
      when others =>
         cpu_mem_data_read <= ZERO;
```

```
      end case;

      if reset = '1' then
         address_reg <= ZERO(31 downto 2);
         data_write_reg <= ZERO;
         write_byte_enable_reg <= ZERO(3 downto 0);
         irq_mask_reg <= ZERO(7 downto 0);
         counter_reg <= ZERO;
         tick_counter <= ZERO;
      elsif rising_edge(clock) then
         if cpu_mem_pause = '0' then
            address_reg <= cpu_mem_address(31 downto 2);
            data_write_reg <= cpu_mem_data_write;
            write_byte_enable_reg <= cpu_mem_write_byte_enable;
            if enable_misc = '1' and write_enable = '1' then
               if address_reg(9 downto 4) = "000001" then
                  irq_mask_reg <= data_write_reg(7 downto 0);
               end if;
            end if;
         end if;

         -- reinitialize counter
         if cpu_mem_pause = '0' and
            enable_misc = '1' and
            write_enable = '1' and
            address_reg(9 downto 4) = "000110" then
            counter_reg <= data_write_reg;
         elsif current_page /= "00" then
            counter_reg <= bv_inc(counter_reg);
         end if;

         tick_counter <= tick_counter + 1;

      end if;
   end process;
end;
```

ANEXO B - INSTRUÇÕES DE USO DA PLATAFORMA

Este anexo descreve o processo de criação de aplicações e de alocação das mesmas na plataforma desenvolvida. Para automatizar o processo de alocação de tarefas, foi desenvolvida uma ferramenta chamada HMPSEditor[1] que faz a interface gráfica com o usuário. A plataforma desenvolvida é composta por 6 processadores: 1 processador mestre e 5 processadores escravos. A Figura 79 mostra a disposição e os endereços de cada processador na rede.

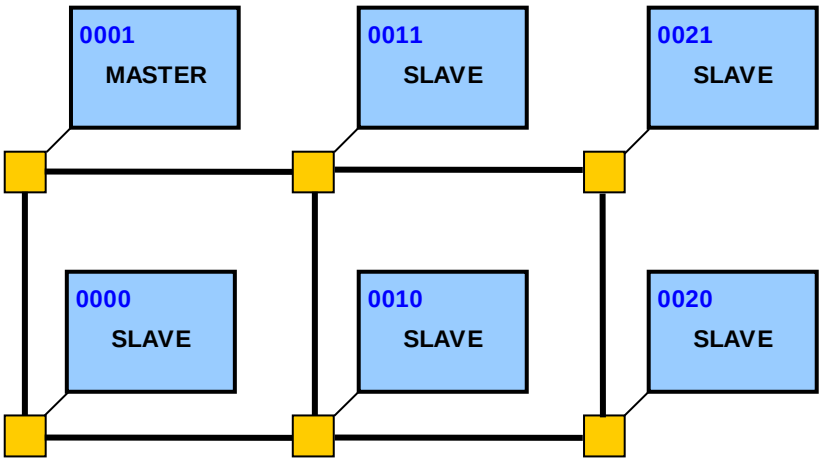


Figura 79 – Plataforma MPSoC e endereço de cada processador.

Para utilizar a plataforma, é necessária a ferramenta de simulação *Modelsim* e ambiente *cygwin*. As Seções seguintes descrevem o projeto, o processo de criação de aplicações e o processo de alocação de tarefas.

1) Organização do projeto

A Tabela 18 mostra a organização do projeto, que une hardware e software.

Tabela 18 – Organização do projeto da plataforma MPSoC.

Diretório	Conteúdo	Descrição
filter	*filter.exe*	Ferramenta que, a partir dos arquivos de resultados gerados na simulação, separa, em diferentes arquivos, os resultados de cada tarefa e do *microkernel* de cada processador.
simulation	Diretório p**lasma**	Contém os arquivos fontes do processador plasma, juntamente com NI e DMA.
	Diretório **noc16**	Contém os arquivos fontes da NoC.
	Diretório **repository**	Contem o arquivo fonte do repositório de tarefas.
	tb16.vhd	*Testbench* do projeto.
	code_master.txt	Aplicação de alocação que executa no mestre.
	code_slave.txt	*Microkernel* que executa em cada escravo
	output_master.txt	Relatório da execução do mestre.
	output_slave_1.txt	Relatório da execução do escravo 1.

[1] HMPS é uma sugestão para o nome da plataforma, que significa Hermes MultiProcessing System.

	output_slave_2.txt	Relatório da execução do escravo 2.
	output_slave_3.txt	Relatório da execução do escravo 3.
	output_slave_4.txt	Relatório da execução do escravo 4.
	output_slave_5.txt	Relatório da execução do escravo 5.
	compila.do	Arquivo de compilação do projeto do hardware.
software	Diretório **applications**	Contém aplicações, divididas em tarefas.
	Diretório **build**	Utilizado pela ferramenta de interface com o usuário para gerar arquivos de *include* com identificadores de tarefas, identificadores de processadores e informações de alocação.
	Diretório **include**	Arquivos de include utilizados pelas tarefas e pelo *microkernel*.
	Diretório **kernel**	Contém o *microkernel* dos processadores escravos e do processador mestre (aplicação de alocação)
tools	Diretório **HMPSEditor**	Contém o aplicativo de interface gráfica com o usuário que automatiza o processo de alocação de tarefas na plataforma
	bintohex.exe	Ferramenta utilizada para a geração de código objeto.
	convert.exe	Ferramenta utilizada para a geração de código objeto.
	rom_loader.exe	Ferramenta que carrega os códigos objetos das tarefas para o repositório.
	tracehex.exe	Ferramenta utilizada para a geração de código objeto.

2) Desenvolvendo aplicações

As aplicações são desenvolvidas em linguagem C e ficam organizadas no diretório **applications**. Cada aplicação possui um diretório contendo as tarefas. Cada tarefa também deve estar dentro de um diretório. As tarefas são nomeadas com letras do alfabeto. Dessa forma, os diretórios das tarefas devem ser assim nomeados: `taskA`, `taskB`, `taskC` e assim por diante. Dentro de cada diretório deve conter o arquivo fonte da tarefa, chamado *task.c*.

Para enviar (`WritePipe`) e receber (`ReadPipe`) mensagens, utiliza-se como identificadores das tarefas: `TASK`**`A`**, `TASK`**`B`**, `TASK`**`C`** e assim por diante.

3) Alocando aplicações

A Figura 80 apresenta a interface da ferramenta HMPSEditor. Ela oferece opção para parametrizar a dimensão da NoC (`NoC Size`) e o número máximo de tarefas por processador (`Max Local Tasks`). Contudo, esta funcionalidade ainda não está implementada e as dimensões da NoC estão fixas em 3x2 e o número máximo de tarefas por processador em 3.

A ferramenta permite listar as aplicações e as tarefas que as constituem (`Applications`). Uma vez que as dimensões da NoC são fixas, o número de processadores também é fixo: 6 processadores. A interface permite escolher qual dos processadores é mestre. Contudo, esta funcionalidade ainda não está implementada. A localização do mestre na rede está fixada em `0001`.

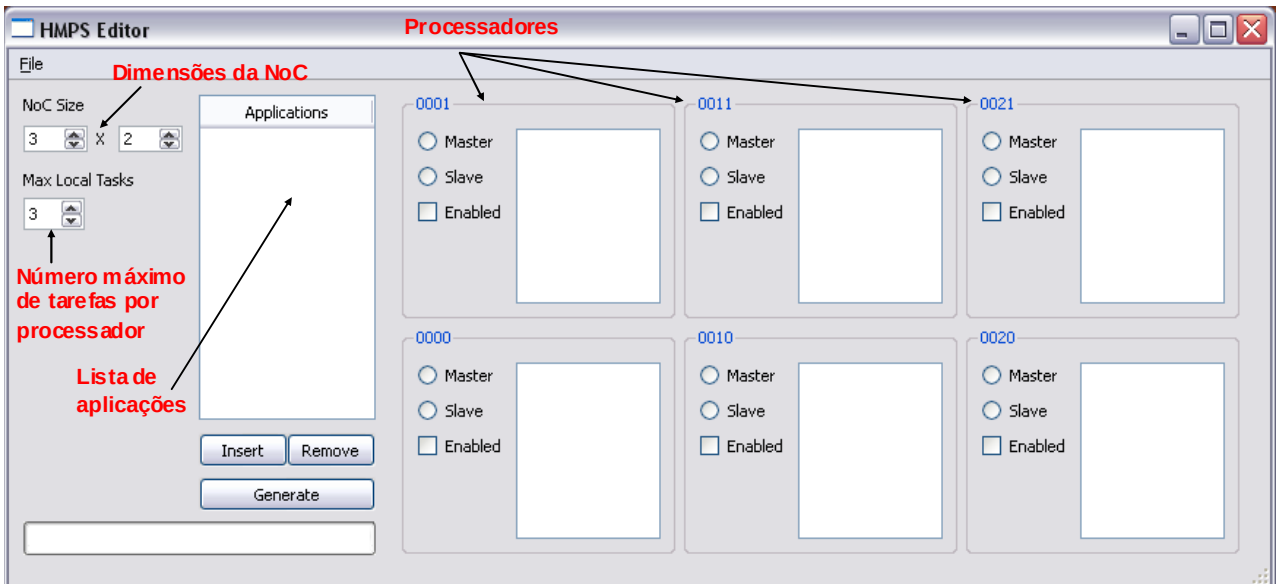


Figura 80 – Interface HMPSEditor.

Clicando no botão `Insert`, pode-se escolher as aplicações a serem carregadas para a interface (conforme a organização do projeto mostrado na Tabela 18, as aplicações encontram-se em **software/applications**). Elas vão ser listadas na área referenciada por `Applications`, como mostra a Figura 81. Neste exemplo, foram carregadas duas aplicações (`communication` e `merge_400_a`) constituídas por 4 e 3 tarefas respectivamente.

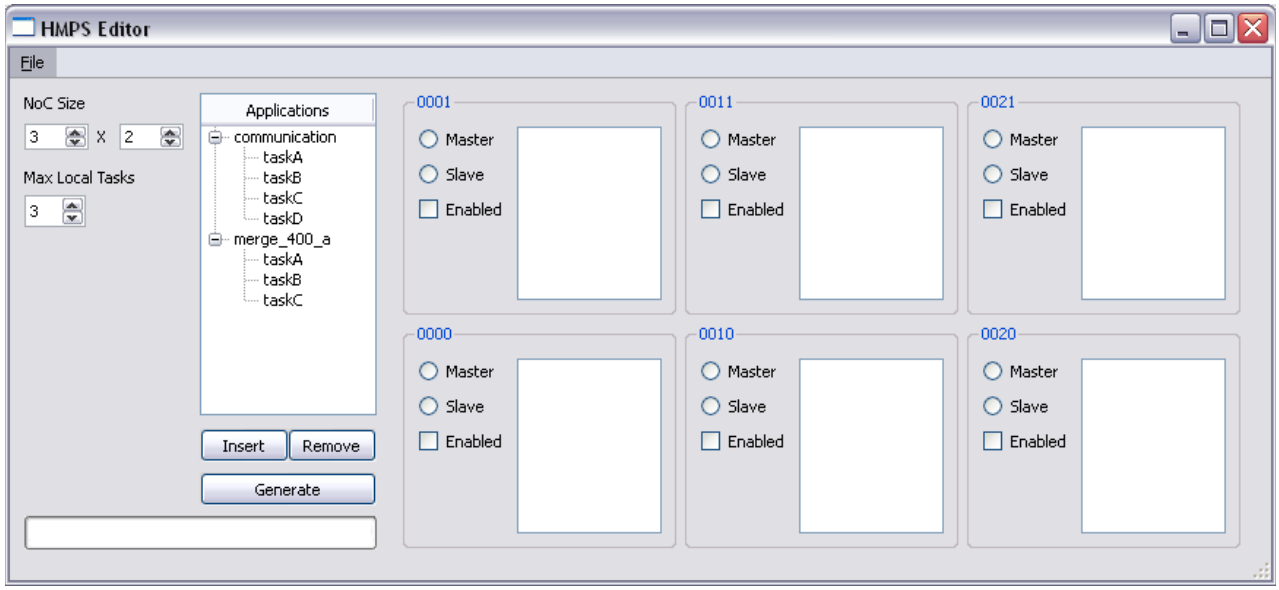


Figura 81 – Lista de aplicações carregadas na interface.

Para alocar uma tarefa, basta clicar sobre a mesma e arrastá-la sobre o escravo no qual deseja-se alocá-la. A Figura 82 mostra a distribuição das tarefas no MPSoC. As tarefas que são arrastadas sobre processadores escravos são alocadas estaticamente. Para carregar tarefas no repositório, sem que sejam alocadas estaticamente, deve-se arrastá-las sobre o processador mestre.

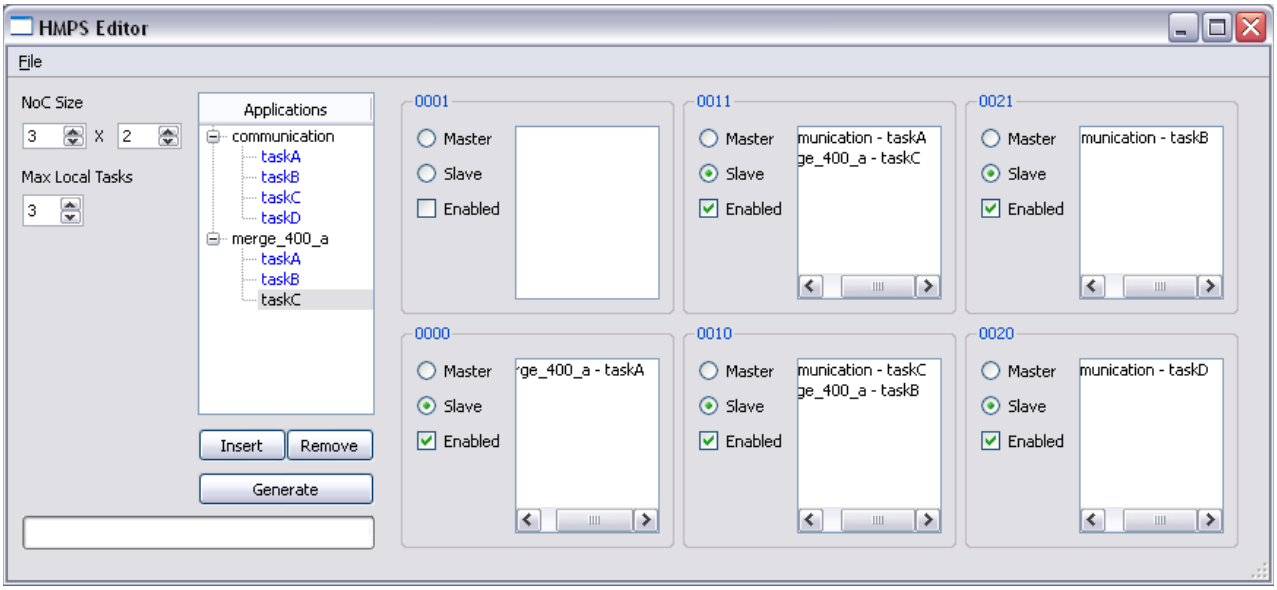


Figura 82 – Tarefas alocadas nos processadores escravos.

Após alocas as tarefas, clica-se no botão `Generate`. Com isso, a ferramenta gera: (*i*) os arquivos de inclusão utilizados pelas tarefas das aplicações, que definem os identificadores numéricos das tarefas; (*ii*) o arquivo de inclusão utilizado pelo mestre, que contém os endereços dos processadores escravos e as informações de alocação estática de tarefas; (*iii*) o arquivo *makefile* que faz o processo de compilação das tarefas e da aplicação de alocação do mestre e a criação do repositório de tarefas.

3) Compilando o projeto gerado

O arquivo *makefile* gerado pela ferramenta de interface encontra-se no diretório **software/build**. Utilizando o *shell* do *cygwin* e estando neste diretório, o projeto de software deve ser compilado através do comando:

```
make all -B
```

O projeto de hardware deve ser compilado a partir da ferramenta de simulação *modelsim*. É preciso estabelecer o diretório onde encontra-se a simulação. Para isso, deve-se acessaro menu *File* → *Change Directory* e escolher o diretório **simulation** deste projeto. Então, no modelsim, digita-se o comando:

```
do compila.do
```

Uma forma de onda contendo sinais de todos os processadores da plataforma pode ser carregada através do comando:

```
do wave_all_cpus.do
```

Para executar a simulação utiliza-se o comando `run <time>`, onde `time` é o tempo de duração da simulação.

Os resultados da simulação são registrados nos arquivos *output_master.txt*, *output_slave_1.txt*, *output_slave_2.txt*, *output_slave_3.txt*, *output_slave_4.txt* e *output_slave_5.txt*.

Cada arquivo contém as informações das operações executadas pelo processador. Estas operações incluem operações do *microkernel* e das tarefas locais.

4) Filtrando os resultados

O diretório **filter** contém a ferramenta, *filter.exe*, que separa os resultados da execução de cada processador em diferentes arquivos: um arquivo para cada tarefa e outro para o *microkernel*.

Os arquivos de resultados devem ser copiados para o diretório **filter**. Estando neste diretório (s*hell* do *cygwin*), a ferramenta deve ser executada passando como parâmetro os arquivos desejados:

```
./filter.exe output_slave_1.txt output_slave_2.txt ...
```