PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SUPORTE PARA APLICAÇÕES DINÂMICAS EM SISTEMAS MULTIPROCESSADOS INTRA-CHIP HOMOGÊNEOS

SERGIO JOHANN FILHO

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Fabiano Passuelo Hessel

Porto Alegre
2012

J65s Johann Filho, Sergio
Suporte para aplicações dinâmicas em sistemas multiprocessados intra-chip homogêneos / Sergio Johann Filho. – Porto Alegre, 2012.
160 f.

Tese (Doutorado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Fabiano Passuelo Hessel.

1. Informática. 2. Multiprocessadores. 3. Arquitetura de Computador. I. Hessel, Fabiano Passuelo. II. Título.

CDD 004.35

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

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

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "Suporte para Aplicações Dinâmicas em Sistemas Multiprocessados Intra-Chip Homogêneos", apresentada por Sérgio Johann Filho, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, Sistemas Embarcados e Sistemas Digitais, aprovado em 2012 pela Comissão Examinadora:

Prof. Dr. Fabiano Passuelo Hessel - Orientador — PPGCC/PUCRS

Prof. Dr. Rubem Dutra Ribeiro Fagundes - — PPGEE/PUCRS

Prof. Dr. Rodolfo Jardim de Azevedo - — UNICAMP

Prof. Dr. César Augusto Missio Marcon – — PPGCC/PUCRS

Homologada em 31/07/2012, conforme Ata No. 15 pela Comissão Coordenadora.

Prof. Dr. Paulo Henrique Lemelle Fernandes
Coordenador.

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

DEDICATÓRIA

Dedico este trabalho à minha família e amigos.

"Any sufficiently advanced technology is indistinguishable from magic."
(Arthur C. Clarke)

AGRADECIMENTOS

Em primeiro lugar gostaria de agradecer aos meus pais Sérgio e Mada, aos meus irmãos Marcelo e Rodrigo e à minha namorada Camila, em conjunto com a sua família, pelo apoio, incentivo e carinho. Aos colegas do GSE meus sinceros agradecimentos pelas discussões produtivas, idéias e constante desenvolvimento do nosso sistema operacional Hellfire OS. Um muito obrigado a todos os meus amigos e ao meu orientador prof. Fabiano, pela confiança e apoio no desenvolvimento desta tese.

SUPORTE PARA APLICAÇÕES DINÂMICAS EM SISTEMAS MULTIPROCESSADOS INTRA-CHIP HOMOGÊNEOS

RESUMO

Sistemas MPSoC modernos fazem uso de recursos que eram disponibilizados apenas em computadores de propósito geral provendo mais funcionalidades para as aplicações. A evolução arquitetural possibilita que mais recursos sejam implementados nestes sistemas embarcados e determina um aumento na complexidade dos novos projetos de *hardware* e *software*.

Além do aumento da complexidade de projeto em sistemas MPSoC atuais, torna-se evidente a dificuldade na utilização eficiente dos recursos computacionais encontrados em tais plataformas. Assim como o determinismo e o tempo de resposta priorizado em muitos sistemas embarcados, a programabilidade de MPSoCs é muito relevante. Dessa forma, interfaces bem definidas de *software* ajudam o desenvolvedor a criar aplicações que utilizam de maneira otimizada os recursos computacionais encontrados nestes sistemas.

A maior parte das aplicações embarcadas são divididas em tarefas e estaticamente mapeadas a elementos de processamento em tempo de projeto, de forma a otimizar um conjunto de métricas pré-estabelecidas. No entanto, a natureza dinâmica de novas aplicações estabelece que estratégias eficientes de mapeamento dinâmico e migração de tarefas sejam implementadas. Neste contexto, esta tese apresenta um modelo para aplicações dinâmicas e gerenciamento distribuído destas em sistemas MPSoC homogêneos. O gerenciamento do sistema faz uso dos conceitos de migração de tarefas e restrições temporais, onde parâmetros de caracterização das tarefas são utilizados nas tomadas de decisão de escalonamento e otimização em tempo de execução.

Neste trabalho é utilizada uma arquitetura MPSoC homogênea, composta por elementos de processamento com memórias locais interconectados por uma NoC. Este ambiente permite a execução de aplicações gerenciadas por um sistema operacional distribuído que implementa o modelo proposto e oferece diversos serviços para o desenvolvimento e otimização de aplicações embarcadas. Muitos trabalhos na área fazem uso de um gerente centralizado para realizar a otimização do sistema em tempo de execução, no entanto tais soluções tendem a ser pouco escaláveis. Os resultados

obtidos mostram que o uso de gerentes distribuídos apresentam maior eficiência para sistemas com um grande número de elementos de processamento e tarefas, com redução nos tempos de estabilização do sistema e redução nas perdas de *deadline* para aplicações com restrições de tempo real.

Palavras Chave: Sistemas embarcados, Sistemas Operacionais, MPSoC, RTOS, Modelagem de aplicações, Mapeamento dinâmico, Migração de tarefas.

SUPPORT FOR DYNAMIC APPLICATIONS IN HOMOGENEOUS MULTI-PROCESSOR SYSTEMS-ON-CHIP

ABSTRACT

Modern MPSoC systems use resources previously available only in general purpose computers providing more functionalities for the applications. The architectural evolution enables more resources to be implemented on these embedded systems and determines an increased complexity of new hardware and software designs.

In addition to the increased design complexity of current MPSoC systems, it is evident the difficulty in efficient use of computational resources found on such platforms. As well as the determinism and response time prioritized in many embedded systems, the programmability of MPSoCs is very relevant. Thus, well-defined software interfaces help developers to create applications that utilize optimally the computational resources found in these systems.

Most embedded applications are divided into tasks and statically mapped to processing elements at design time, in order to optimize a set of pre-stablished metrics. However, the dynamic nature of new applications requires efficient strategies for the dynamic mapping and task migration to be implemented. In this context, this thesis presents a model for dynamic applications and distributed management of these in homogeneous MPSoC systems. The system management uses task migration concepts and timing constraints, where tasks characterization parameters' are used in scheduling decision making and optimization at runtime.

In this work we used a homogeneous MPSoC architecture, consisting of processing elements with a local memory interconected by a NoC. This environment allows the execution of applications managed by a distributed operating system that implements the proposed model and offers many services for the development and optimization of embedded applications. Many works in this field make use of a centralized manager to perform the system optimization at runtime, however such solutions tend to be not very scalable. Results show that the use of distributed managers present greater efficiency in systems with a large number of processing elements and tasks, with a reduction

in the system stabilization time and reduction of deadline misses for applications with realtime constraints.

Keywords: Embedded systems, Operating systems, MPSoC, RTOS, Application modelling, Dynamic mapping, Task migration.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ILP – Instruction Level Parallelism

TLP – Thread Level Parallelism

RTS – Real-Time System

RTOS – Real-Time Operating System

SOC – System-on-Chip

MPSOC – Multi Processor System-on-Chip

CMP – Chip Multi-Processor

ISP – Instruction Set Processor

DSP – Digital Signal Processor

FPGA – Field Programmable Gate Array

IP – Intellectual Property

ADL – Architecture Description Language

STG – Single Task Graph

MTG – Multi Task Graph

BF – Best Fit

WF – Worst Fit

MPEG – Motion Picture Experts Group

MWD – Multi-Window Display

VOPD – Video Object Plane Decoder

LEC-DN – Low Energy Consumption, Dependences Neighborhood

MPOS – Multi-Processor Operating System

MJPEG – Motion JPEG

PPN – Polyhedral Process Network

DVFS – Dynamic Voltage Frequency Scaling

NOC – Network-on-chip

MMU – Memory Management Unit

ATM – Application Task Model

ATG – Application Task Graph

ACCM – Architecture Communication and Computation Model

ACCG – Architecture Communication and Computation Graph

FCFS – First Come First Served

RR – Round Robin

RM – Rate Monotonic

DM – Deadline Monotonic

LLF – Least Laxity First

EDF – Earliest Deadline First

NN – Nearest Neighbor

FF – First Free

MMCL – Minimum Maximum Channel Load

MACL – Minimum Average Channel Load

PL – Path Load

BN – Best Neighbor

HAL – Hardware Abstraction Layer

API – Application Programming Interface

TCB – Task Control Block

PIC – Position Independent Code

GOT – Global Offset Table

GCC – Gnu Compiler Collection

RTL – Register Transfer Level

MIPS – Microprocessor without Interlocked Pipeline Stages

ASIC – Application Specific Integrated Circuit

DMA – Direct Memory Access

COW – Copy on Write

MJPEG – Motion Joint Photographic Experts Group

RGB – Red, Green, Blue

DCT – Discrete Cossine Transform

VLC – Variable Length Coder

VOPD – Video Object Plane Decoder

MPEG4 – Moving Pictures Experts Group, layer 4

SUMÁRIO

1. INTRODUÇÃO

Sistemas embarcados tradicionalmente fazem uso de um único elemento de processamento em conjunto com módulos de entrada e saída e memória integrados em um único *chip*, sendo esta organização arquitetural conhecida como sistema em um *chip* ou SoC. A complexidade, aliada aos requisitos de desempenho em aplicações embarcadas, no entanto, têm crescido substancialmente [82]. O emprego de multiprocessamento em um mesmo *chip* provê o poder computacional necessário para a execução de aplicações que demandam cada vez mais desempenho. Dessa forma, sistemas multiprocessados em um *chip* ou MPSoCs, são utilizados com o intuito de oferecer o poder de processamento necessário para a execução de aplicações reais complexas [30] aliado às necessidades da aplicação.

Os problemas enfrentados para a obtenção de aumento do paralelismo em nível de instrução (ILP) não são recentes, e muitas vezes como alternativa são utilizadas soluções que implementam o paralelismo em nível de *thread* (TLP) [5]. Integrando-se múltiplos elementos de processamento em um mesmo *chip*, torna-se possível a exploração de TLP, contornando-se dessa forma o gargalo no desempenho. Ao ser utilizado um grande número de elementos de processamento integrados, trabalhando em frequências reduzidas, é possível a obtenção de um alto desempenho do sistema e ao mesmo tempo uma redução significativa no consumo de energia, algo essencial para dispositivos que operam com baterias e que ainda assim precisam ter seu tempo de operação prolongado e peso reduzido.

A integração de diversos elementos de processamento em um único *chip* com o objetivo de executar aplicações paralelas demanda a utilização de infraestruturas de comunicação eficientes. Estas estruturas podem ser baseadas em barramentos, conexões ponto-a-ponto ou redes intra-chip [5]. Redes intra-chip (NoCs) são compostas por um conjunto de roteadores interconectados por canais de comunicação, e oferecem uma série de vantagens sobre as outras estruturas de interconexão tais como redução no consumo de energia e escalabilidade, o que torna o seu uso ideal para sistemas com grande número de elementos de processamento. NoCs fazem uso de conceitos como topologias, algoritmos de roteamento e técnicas de bufferização [68], sendo estes advindos de redes de computadores de propósito geral. Dessa forma, muitas das técnicas originadas em sistemas distribuídos podem ser diretamente aplicadas neste tipo de organização.

Um dos grandes problemas decorrentes do uso de sistemas multiprocessados está no aproveitamento efetivo de recursos. Para que os recursos sejam utilizados eficientemente, a aplicação precisa ser descrita como um conjunto de tarefas comunicantes distribuídas entre elementos de processamento e trabalhando em paralelo. O grande problema está no fato de boa parte das aplicações seguir um fluxo de execução serial e principalmente no fato de não existir suporte para a implementação de aplicações distribuídas de maneira a facilitar a programação em ambientes multiprocessados. Ainda quando existente o suporte para a programação em ambientes MPSoC, muitas vezes é necessário que seja feita a atribuição de tarefas à elementos de processamento em tempo de projeto sem a

possiblidade de modificação, o que pode não ser eficiente no caso de aplicações que são carregadas dinamicamente ou possuem algum tipo de interação com o usuário que consequentemente modifica o perfil de execução.

A aplicação composta por um conjunto de tarefas pode ser *mapeada* de maneira automatizada com o uso de métodos estáticos ou dinâmicos. Os métodos estáticos são utilizados em situações onde o conjunto de tarefas que compõem a aplicação é fixo [89]. Métodos dinâmicos são adequados a aplicações com um conjunto variável de tarefas ou tarefas com características variáveis. O objetivo de métodos estáticos e dinâmicos é encontrar o melhor posicionamento das tarefas na arquitetura de acordo com alguma métrica, como por exemplo redução do consumo de energia, redução no tráfego na rede e balanceamento de carga dos elementos de processamento.

Em tempo de execução, além do mapeamento dinâmico de tarefas métodos que utilizam o conceito de *migração de tarefas* podem ser aplicados com o intuito de melhorar o desempenho de determinada aplicação já previamente mapeada. A migração de tarefas trasfere estas de um elemento de processamento a outro, podendo ser gerenciada pelo desenvolvedor da aplicação ou de maneira automática.

1.1 Motivação

Sistemas embarcados multiprocessados estão presentes na maioria das aplicações que foram tradicionalmente gerenciadas por sistemas monoprocessados. A utilização de múltiplos elementos de processamento em um único *chip* introduz novos desafios devido ao aumento da complexidade arquitetural. Entre os principais desafios podem ser salientados a programabilidade, otimização e reuso de tais sistemas para aplicações com características dinâmicas.

Uma forma de lidar com tais fatores está em reduzir o esforço necessário para construir aplicações para MPSoCs. Sistemas operacionais de tempo real oferecem interfaces padronizadas e dessa forma os desenvolvedores podem utilizar ou pelo menos ter facilitado o acesso ao poder computacional disponível no *hardware*.

Conforme o poder computacional oferecido por novos MPSoCs aumenta, aplicações e funcionalidades podem ser incluídas em tempo de execução, algo pouco comum em sistemas embarcados multiprocessados do passado. Tipicamente, MPSoCs eram formados por poucos elementos de processamento de poder computacional mediano, onde a aplicação era completamente definida e tinha suas tarefas atribuídas a processadores em tempo de projeto. Algumas aplicações atuais, no entanto, possuem uma complexidade aumentada e apresentam uma carga variável aos elementos de processamento [46] assim como um perfil variável de tráfego na rede de interconexão, o que torna praticamente impossível definir em tempo de projeto a melhor forma de alocar tarefas a recursos. Outras aplicações, além de apresentarem características dinâmicas, possuem a necessidade de cumprir com restrições de tempo real especificadas em tempo de projeto. Dessa forma, mecanismos

eficientes que permitam adaptar recursos de *hardware* às necessidades da aplicação precisam ser implementados.

Arquiteturas MPSoC podem ser classificadas como heterogêneas e homogêneas. Soluções heterogêneas oferecem uma série de vantagens como redução no consumo de energia aliado a um bom compromisso de área em silício [42, 54, 64]. Entretanto, quando comparadas à soluções homogêneas, seu projeto e manutenção podem ser considerados complexos do ponto de vista de implementação. Dessa forma, soluções homogêneas permitem um projeto mais rápido e o reuso da mesma arquitetura para diferentes produtos [14]. Além disso, estas soluções permitem uma exploração do espaço de projeto de maneira simplificada, através do uso de ferramentas em alto nível, interfaces padronizadas e o uso de algoritmos mais simples para a realização da atividade de mapeamento de tarefas, algo desejável devido ao considerável aumento no número de elementos de processamento em MPSoCs nos últimos anos.

1.2 Objetivos e Originalidade

Objetiva-se com o desenvolvimento deste trabalho apresentar um modelo de tarefas e arquitetura multiprocessada que descreva aplicações dinâmicas de tempo real, e que suporte a otimização da aplicação com o uso de técnicas de migração de tarefas e mapeamento dinâmico. Os objetivos específicos deste trabalho incluem:

- Definição de um modelo de tarefas de tempo real e aspectos arquiteturais, o qual suporta aplicações com características dinâmicas;
- Implementação do modelo de tarefas proposto em um sistema operacional de tempo real;
- Definição de uma API para o sistema operacional com o intuito de facilitar o desenvolvimento de aplicações embarcadas em ambientes multiprocessados;
- Implementação de um mecanismo para a otimização do sistema em tempo de execução de maneira automática e distribuída, utilizando serviços de avaliação de estado, trocas de mensagem e migração de tarefas;
- Validação do modelo de tarefas proposto em uma ferramenta de simulação baseada em um protótipo em *hardware* o qual implementa o modelo arquitetural especificado;
- Descrição de aplicações de acordo com a API implementada e utilização destas aplicações para avaliação de resultados;

A originalidade deste trabalho consiste na proposta de um novo modelo de tarefas que incorpora parâmetros de tempo real na descrição de aplicações multiprocessadas dinâmicas em ambientes intra-chip. Diversos trabalhos que empregam o conceito de mapeamento dinâmico de tarefas não abordam questões como o contexto de execução de aplicações, sendo voltados basicamente a

aplicações de *streaming*. Ainda, não são levadas em consideração as restrições de tempo real, e devido a maneira demasiadamente simplificada na qual tarefas executam de acordo com os modelos encontrados na maioria dos trabalhos investigados, uma modificação no mapeamento ou inclusão de novas tarefas em tempo de execução pode desestabilizar a execução da aplicação. Outros trabalhos os quais empregam o conceito de migração de tarefas utilizam pontos de migração, o que torna a migração de tarefas não transparente ao desenvolvedor. Outro ponto é que a maioria dos trabalhos que utilizam algum mecanismo para a gerência automática de migração o fazem de maneira centralizada, tornando-se pouco escaláveis para arquiteturas MPSoC com um grande número de elementos de processamento.

1.3 Organização do Trabalho

O presente trabalho apresenta-se organizado em sete capítulos. No primeiro foi apresentada uma breve introdução sobre o tema a ser abordado. Além disso a motivação e os objetivos foram enumerados. O Capítulo 2 apresenta uma série de conceitos gerais sobre sistemas embarcados, complementado a introdução. No Capítulo 3 são apresentados diversos trabalhos relacionados sobre mapeamento e migração de tarefas e comparados ao modelo proposto. Ao final do Capítulo, um quadro comparativo de todos os trabalhos avaliados é apresentado, classificando os mesmos de acordo com critérios estabelecidos. O Capítulo 4 apresenta o modelo proposto para a descrição de aplicações dinâmicas de tempo real e a arquitetura proposta para a execução destas, abordando os parâmetros necessários para representar tarefas e como são escalonadas. Além disso são apresentados os protocolos de comunicação entre tarefas, migração de tarefas e gerência de migração. No Capítulo 5 é apresentada a implementação do modelo proposto na forma de um sistema operacional, focando principalmente no modelo de tarefas. Os parâmetros caracterizados utilizados como base de tempo de escalonamento são apresentados, assim como o escalonamento e comunicação entre tarefas. Neste Capítulo são abordados alguns detalhes de como é realizado o mapeamento, migração e gerência de tarefas, além da API do sistema operacional. Ao final do Capítulo, a implementação do modelo de arquitetura proposto é apresentada. O Capítulo 6 dedica-se aos experimentos realizados. Inicialmente, são abordados testes de desempenho e *overhead* do sistema operacional. A seguir, alguns testes do mecanismo de gerência de migração são realizados sobre aplicações sintéticas, comparando um modelo de migração centralizado ao modelo de migração distribuído proposto. Ao final do Capítulo, aplicações reais são modeladas de acordo com a proposta e posteriormente avaliadas. O Capítulo 7 relaciona algumas conclusões obtidas e aponta sugestões para trabalhos futuros relacionados ao tema proposto.

2. DEFINIÇÕES E CONCEITOS GERAIS

Este Capítulo apresenta conceitos gerais sobre sistemas embarcados, sendo estes essenciais para a compreensão do restante do texto. Serão abordados tópicos como sistemas de tempo real, sistemas multiprocessados, modelagem e mapeamento de tarefas que compõem uma aplicação.

2.1 Sistemas de Tempo Real

Em muitos sistemas embarcados estão presentes restrições temporais, aliadas as já presentes restrições que caracterizam este tipo de sistema como a vida útil de baterias, tamanho, imunidade a choques e desgastes entre outros. Algumas aplicações exigem tempo de resposta determinado, e a validade do resultado do processamento depende deste fator. Sistemas desta natureza são conhecidos como *sistemas de tempo real* ou RTSs. Segundo Farines [20], os sistemas de tempo real podem ser divididos em duas categorias distintas:

- *Hard Real-Time* - Possuem restrições estritas quanto ao tempo de resposta e variação deste. Sistemas *hard* devem sempre prover resultados no tempo determinado, ou a validade do próprio sistema torna-se nula. Outros sistemas ou vidas podem depender destes;

- *Soft Real-Time* - Possuem restrições quanto ao tempo de resposta, no entanto o sistema não é invalidado no caso de atrasos, apenas tem seu desempenho degradado.

Os sistemas operacionais de tempo real ou RTOSs são aqueles que controlam aplicações com algum tipo de restrição de tempo real. Desta forma, muitos RTSs são gerenciados por sistemas operacionais deste tipo. Esta classe de sistema operacional provê funcionalidades semelhantes a sistemas de propósito geral, como atendimento a interrupções, abstração de processos ou tarefas, comunicação entre processos ou tarefas e gerenciamento de memória. De acordo com Farines [20], o principal objetivo de um RTOS é garantir a execução de uma tarefa dentro do seu tempo determinado. O fator chave neste conceito é o *jitter* do sistema, que é a variação da periodicidade frente o real período de uma tarefa. Normalmente, um sistema operacional *hard real-time* possui menor e mais previsível *jitter* que um sistema operacional *soft real-time*. Desta forma, o objetivo no projeto de um sistema deste tipo não é desempenho em termos de *throughput*, e sim previsibilidade.

Sistemas operacionais que geralmente conseguem atender requisições temporais são ditos *soft real-time OS*, e aqueles que atendem suas requisições sempre e de maneira determinística são ditos *hard real-time OS*. Muitos são os sistemas operacionais pertencentes à categoria *real-time*, entre os quais destacam-se MicroC/OS-II [53], AMX [33], CMX [15], Integrity [23] e MQX [4].

2.2 Sistemas Embarcados Multiprocessados

Por questões de custo, tempo de projeto, desempenho e confiabilidade, é desejável que sistemas embarcados sejam implementados na forma de um único *chip*, dando origem aos chamados SoC. Comumente, SoCs são construídos como um agrupamento de componentes heterogêneos, como elementos de processamento, memórias, aceleradores (co-processadores) e meio de interconexão entre os componentes.

Uma especialização deste tipo de sistema embarcado deu origem aos sistemas que empregam mais de um elemento de processamento, os chamados MPSoC. Segundo Jerraya [30], sistemas MPSoC podem ser, de maneira equivocada, comparados a arquiteturas CMP. Ainda, CMPs são construídos com a integração de múltiplos elementos de processamento (normalmente do mesmo tipo) em um único *chip*, aproveitando a alta densidade de transistores oferecida por tecnologias de implementação de *hardware* atuais e não importando o consumo de energia (voltado a desempenho). MPSoCs, por outro lado, possuem arquiteturas customizadas e têm como objetivo utilizar a alta densidade de transistores aliada às necessidades de uma determinada aplicação. Estas necessidades podem ser determinadas pelo tempo de resposta, consumo de energia, tamanho, entre outros. Algumas das razões para o emprego de MPSoCs em projetos atuais está ligado a fatores como redução no tempo de projeto, simplificação do processo de verificação em sistemas complexos e extensão da programabilidade de plataformas.

MPSoCs são classificados como homogêneos ou heterogêneos. Sistemas que possuem mais de um tipo de elemento de processamento são heterogêneos e os com um único tipo são classificados como homogêneos. A heterogeneidade tem como objetivo fornecer componentes específicos a uma determinada aplicação, e esta deve ser aplicada conforme a necessidade. Muitas vezes, devido ao grande número de componentes presentes em um mesmo MPSoC opta-se por um menor nível de heterogeneidade, simplificando dessa forma o projeto de *hardware* e *software*.

2.3 Elementos de processamento

Elementos de processamento são unidades de execução implementadas em *hardware*, onde aplicações são executadas. Estas unidades de execução podem ser elementos de propósito geral, como processadores de conjuntos de instruções (ISPs) , elementos de processamento especializados (DSPs), estruturas implementadas em FPGA , núcleos de propriedade intelectual (IPs) dedicados e memórias especializadas [83].

2.4 Meios de interconexão

Para que as unidades de execução presentes em um *chip* possam trocar informações são necessárias estruturas de interconexão que realizam a comunicação entre estas. Estas estruturas podem ser barramentos multiponto, barramentos hierárquicos, conexões ponto a ponto e redes intra-chip [83]. Estes meios de interconexão diferenciam-se por sua organização física, desempenho, escalabilidade, consumo de energia, custo de implementação entre outros fatores.

Barramentos multiponto são estruturas comumente utilizadas para conectar elementos de processamento em um mesmo *chip*. São relativamente escaláveis, permitindo a conexão de dezenas de nodos, o que torna a implementação de *hardware* de interconexão relativamente simples e organizada. No entando, o meio é compartilhado entre os processadores e problemas como colisões de mensagens, aumento no consumo de energia e carga capacitiva tornam-se inevitáveis com o aumento do número de nodos. Uma forma encontrada para amenizar estes problemas é a implementação de barramentos de forma hierárquica. Essa solução, no entanto, apenas ameniza os problemas anteriormente citados.

Conexões ponto a ponto possuem um desempenho ótimo, uma vez que são linhas dedicadas entre elementos de processamento. Do ponto de vista de projeto são pouco escaláveis e custosas em termos de área em silício, o que limita o seu uso a poucos elementos de processamento.

Redes intra-chip são propostas como a solução para os problemas citados, e seu uso é crescente no projeto de MPSoCs com dezenas e mesmo centenas de elementos de processamento. NoCs são implementadas por roteadores e canais de comunicação entre estes, na forma de conexões ponto a ponto, e são altamente escaláveis além de prover um maior paralelismo comparado a barramentos. Os roteadores podem ser dispostos em diferentes topologias, e a estes são conectados elementos de processamento.

2.5 Abstração de Sistemas com Modelos em Alto Nivel

O aumento da complexidade e os desafios encontrados quando da especificação até a construção e manutenção de sistemas MPSoC tornam evidente a crescente utilização de descrições em alto nível de abstração[43]. Segundo Ost [68], a modelagem consiste em produzir uma descrição do sistema, ou parte deste, utilizando algum tipo de formalismo. Estas descrições são baseadas em uma especificação e podem estar na forma de representações gráficas ou linguagens.

Ainda, de acordo com Lehoczky [29], um modelo deve conter as mesmas propriedades da entidade modelada que são relevantes a uma determinada aplicação. Neste contexto, um modelo é uma simplificação de uma entidade, construído com o intuito de acelerar determinado processo, e sua utilidade depende da relevância da representação adotada para a resolução de um problema específico.

2.6 Exploraçao do Espaço de Projeto

Usualmente, o *co-design* inicia em nível de sistema, onde a divisão final do sistema *hardware* e *software* ainda nao foi completada. Após os passos iniciais, um modelo de verificação funcional pode ser feito através de mecanismos de co-simulação, os quais ajudam na validação e refinamento do sistema.

Restrições comuns de sistemas embarcados, como *time-to-market* exíguo e a alta complexidade de projetos atuais, implicam na utilização de simuladores em diferentes níveis de abstração por parte dos projetistas, desde simuladores de conjunto de instruções até descrições em nível de sistema, usualmente implementadas em ADL. Nestes simuladores, tanto modelos de *hardware* quanto de *software* são executados com o intuito de se realizar uma análise do comportamento do sistema em um estágio inicial de desenvolvimento.

Neste contexto, diversos trabalhos [21, 34, 73, 80] apresentam esforços no que diz respeito a modelagem de RTOS e aplicações em um alto nível de abstração. O maior problema nesta abordagem é que quando o projeto é refinado, apesar do modelo do RTOS poder ser traduzido automaticamente em *software*, normalmente um RTOS amplamente adotado é preferido, o que previne que resultados obtidos durante a modelagem sejam suficientemente precisos quando a implementação do *hardware* real estiver concluída.

Yoo [87] apresenta o problema de *equivalência de código*, o qual é definido como, se o código executado pelo sistema modelado for diferente do executado pela arquitetura alvo, então o comportamento do sistema pode ser diferente do que foi simulado. Assim, decisões erradas de projeto podem ser tomadas. Além disso, ao utilizar-se abordagens de simulação em alto nível de abstração, usualmente o comportamento de tempo do sistema não é levado em consideração ou o mesmo não é completamente preciso [77]. Simuladores onde o quesito de *timing* da arquitetura alvo não for replicado podem modelar o sistema embarcado paralelo imprecisamente.

2.7 Tarefas

No contexto deste trabalho, tarefas são unidades sequenciais e básicas de execução. Tarefas podem coexistir em um mesmo sistema, onde compartilham tempo de processamento se estiverem em um mesmo elemento de processamento ou executam paralelamente se estiverem distribuídas. Uma aplicação embarcada multiprocessada é composta por um conjunto de tarefas, que executam em um conjunto de elementos de processamento e implementam uma determinada funcionalidade.

Essencialmente, no contexto deste trabalho tarefas são uma funcionalidade implementada em *software*. Outros trabalhos [11] utilizam o conceito de tarefas em *hardware*, no entando este conceito não se aplica ao modelo aqui proposto.

2.8 Particionamento e Mapeamento de Tarefas

O particionamento das tarefas da aplicação compreende no agrupamento de tarefas. A atividade de mapeamento tem como objetivo realizar o posicionamento dos grupos de tarefas na arquitetura, utilizando para esse fim uma função de mapeamento. Sabe-se que o problema de mapeamento é NP-completo [70] e algoritmos como *Simulated Annealing*, *Tabu Search*, *Cutting Tree*, métodos gulosos entre outros podem ser utilizados como alternativa ao método exaustivo para a obtenção de resultados aceitáveis quanto a qualidade do mapeamento, além da vantagem de possuírem um tempo de computação reduzido. As métricas mais comuns para a realização do mapeamento são consumo de energia, latência, ocupação dos canais de comunicação, carga dos elementos de processamento e tempo de resposta da aplicação.

Uma proposta de classificação para o mapeamento de tarefas é apresentada por Mandelli [44]. Esta proposta classifica o mapeamento de acordo com quatro critérios, sendo estes (i) o momento em que é executado; (ii) o número de tarefas por elemento de processamento; (iii) a entidade de controle do mapeamento; e (iv) a arquitetura alvo. Assim, esta classificação pode ser detalhada como:

- Momento em que é executado

  Estático: realizado em tempo de projeto. As heurísticas utilizadas para a realização do mapeamento podem levar em consideração todos os aspectos do sistema, uma vez que o conjunto de tarefas já é definido. Entretanto, as heurísticas de mapeamento em tempo de projeto são computacionalmente complexas para lidarem com sistemas dinâmicos.

  Dinâmico: realizado em tempo de execução. Heurísticas podem utilizar informações definidas em tempo de projeto, além de algoritmos simples e rápidos para a realização do mapeamento de tarefas dinamicamente.

  Com reservas de recursos: heurísticas realizam a verificação de recursos antes de realizar o mapeamento.

  Sem reservas de recursos: heurísticas realizam o mapeamento de acordo com a necessidade. O sistema pode não ter recursos suficientes para mapear uma tarefa, sendo necessário esperar a liberação de recursos.

- Número de tarefas

  Monotarefa: uma única tarefa pode ser mapeada por elemento de processamento.

  Multitarefa: diversas tarefas podem ser mapeadas em um mesmo elemento de processamento.

- Entidade de controle

  Centralizado: um elemento de processamento é responsável pela gerência do mapeamento.

Distribuído: diversos elementos de processamento são responsáveis pela gerência. Pode ser dividido em regiões, onde um elemento é responsável pelo controle do mapeamento de outros na mesma região.

- Arquitetura alvo

  Homogênea: todos os elementos de processamento são idênticos.

  Heterogênea: diferentes tipos de elemento de processamento ou aceleradores são utilizados. Anteriormente ao mapeamento deve ser realizado o *binding* que consiste em definir quais tarefas podem ser executadas por elementos específicos.

2.9 Migração de Tarefas

A migração de tarefas consiste na relocação de tarefas de um elemento de processamento para outro em tempo de execução. Assim, o conceito de migração pode ser diferenciado do mapeamento, uma vez que em uma migração uma tarefa necessariamente precisa existir (estar mapeada). Para que este processo ocorra, torna-se necessário parar a execução da tarefa a ser migrada, salvar o contexto do elemento de processamento[1], realizar a cópia dos segmentos de código, dados e o contexto para o elemento de processamento destino, criar uma tarefa nova no destino, restaurar na tarefa criada o contexto copiado, reiniciar a tarefa e destruir a tarefa migrada no elemento de processamento de origem.

Existem essencialmente duas formas de serem implementados mecanismos de migração de tarefas de acordo com Nollet [64]. O primeiro, consiste na utilização de *pontos de migração* a serem definidos no código das tarefas. Este tipo de migração possui algumas vantagens, como simplicidade na implementação do mecanismo e determinismo. Assim, a invocação do processo de migração apenas irá ocorrer em momentos oportunos pois o mesmo é definido pelo desenvolvedor. Além disso, existe uma menor chance de ocorrerem perdas de *deadline* em sistemas de tempo real, uma vez que pontos críticos na execução podem ser tratados ou avaliados em tempo de projeto. A principal desvantagem dessa forma é a necessidade do programador estar envolvido na otimização, decidindo exatamente os pontos ideais de migração. No caso de aplicações realmente dinâmicas, a decisão sobre onde adicionar pontos de migração no código pode ser inviável, e é possível que a aplicação venha a piorar seu desempenho.

O segundo mecanismo consiste na implementação da migração de tarefas de maneira transparente. Nesta forma, o processo de migração é controlado por um sistema operacional, que deve decidir, com base em algum critério, relocar tarefas de um elemento de processamento a outro. As vantagens dessa forma de migração são a transparência do processo de migração em relação a aplicação (o programador não precisa definir pontos de migração) e adaptabilidade da

[1] O salvamento do contexto de um processador consiste em armazenar o conteúdo de registradores (que mantém o estado atual do processador) em uma estrutura de dados específica, com o intuito de restaurar o estado de execução em um momento oportuno.

aplicação em situações de modificação das características do sistema. A desvantagem dessa forma é sua complexidade de implementação, e também os cuidados que devem ser tomados para que não ocorram migrações em excesso e que sejam preservados os critérios de tempo real da aplicação.

3. TRABALHOS RELACIONADOS

O presente Capítulo apresenta alguns dos trabalhos relacionados encontrados na literatura. Tais trabalhos são classificados de acordo com o seu tipo de mapeamento, sendo este estático ou dinâmico. O conceito de migração de tarefas diverge em alguns pontos do conceito de mapeamento dinâmico e alguns trabalhos, incluindo este, foram classificados em uma categoria a parte.

3.1 Mapeamento Estático

Os trabalhos apresentados nesta Seção referem-se ao mapeamento estático realizado em sistemas MPSoC. Alguns trabalhos não detalham formalmente o modelo utilizado para a realização do mapeamento de tarefas, mas preocupam-se com a descrição da técnica utilizada para a realização dessa atividade. Naturalmente, a maioria dos trabalhos pode utilizar um modelo correspondente a algum dos apresentados anteriormente.

O trabalho apresentado por Mihal [55] utiliza o conceito de modelos de computação para descrever os requisitos da aplicação com relação a comunicação e princípios arquiteturais sobre NoCs, com o intuito de modelar aplicações paralelas em ambientes heterogêneos. Na abordagem utilizada, a aplicação é descrita em alto nível e particionada manualmente. O particionamento atribui as partes que compõem uma aplicação a elementos de processamento, e os mesmos são posteriormente mapeados na arquitetura. Neste trabalho não é apresentada nenhuma técnica automática para realização do mapeamento, o qual é feito manualmente.

Nos trabalhos apresentados por Lei [38, 37] é proposta uma solução para o problema de mapeamento, a qual faz a utilização de algoritmos genéticos. A aplicação é modelada através de dois diferentes grafos, chamados STG e MTG. Esses grafos são gerados automaticamente pela ferramenta TGFF[18]. No trabalho, os autores descrevem a arquitetura como uma NoC malha, e propõem um algoritmo que tem como objetivo a redução no tempo de execução da aplicação.

A proposta do trabalho de Rhee [75] consiste em lidar com o problema de mapeamento de elementos de processamento em uma NoC, com o objetivo de reduzir o consumo de energia e o congestionamento do meio de interconexão. Os autores propõem uma técnica que permite o mapeamento de múltiplos elementos de processamento a cada roteador da rede, reduzindo assim o número de saltos médio. As aplicações são modeladas como grafos que representam o volume de dados transmitidos entre elementos de processamento. Segundo os autores, reduções de até 81% no consumo de energia e 2.5% na largura de banda necessária são observadas, comparando essa técnica com outras que utilizam apenas o mapeamento de um elemento de processamento por roteador da rede.

Os trabalhos de Hu [27, 28] utilizam um grafo denominado CTG para descrever uma aplicação. Esta descrição baseia-se na caracterização da comunicação através de pesos atribuídos

a canais de comunicação, representados pela quantidade de dados transmitidos. Este modelo, além do volume e largura de banda necessária para a comunicação, incorpora informações referentes a computação, especificada detalhadamente através de seu tempo de execução, consumo de energia e tempo limite para execução. O objetivo dos autores é propor uma técnica com o intuito de reduzir o consumo de energia de uma determinada aplicação, e o algoritmo utilizado pelos autores para melhorar o tempo de mapeamento é divisão e conquista.

Murali [61] utiliza uma representação em forma de grafo de elementos de processamento para descrever a arquitetura. O objetivo dos autores é realizar o mapeamento de acordo com a largura de banda necessária a aplicação. Em seu trabalho, apresenta uma ferramenta que permite selecionar a topologia de NoC empregada. Além disto, realiza um estudo de duas alternativas para a estratégia de roteamento que pode ser um caminho único mínimo ou múltiplos caminhos, permitindo a divisão de tráfego na rede. O trabalho é estendido em [62], incorporando outras métricas ao modelo como latência e consumo de energia da aplicação. Em [60] são realizados estudos sobre um conjunto de aplicações, e diversas soluções de mapeamento são investigadas utilizando o método *Tabu Search*.

Manolache [45] apresenta uma proposta para a solução do problema de falhas temporárias em canais de comunicação utilizando para isso redundância de mensagens. No trabalho são apresentados modelos para a aplicação e seu mapeamento, para o *hardware* e para a comunicação. O modelo da aplicação é representado por um conjunto de grafos direcionados acíclicos. Cada tarefa é modelada em um grafo e possui parâmetros como período, tempo de execução e *deadline*. O mapeamento das tarefas é feito em tempo de projeto utilizando-se o algoritmo *Tabu Search*, e o caminho dos pacotes é definido em nível de aplicação. O objetivo do trabalho é garantir tolerância a falhas e reduzir a latência média para a entrega de mensagens na rede.

O trabalho apresentado por Srinivasan [84] aprofunda-se em uma técnica de minimização do consumo de energia no mapeamento de elementos de processamento, e utiliza critérios adicionais como largura de banda e latência. Os autores modelam o problema de mapeamento definindo elementos de processamento como vértices de um grafo direcionado, e arestas como a largura de banda e latência necessárias a aplicação em questão. Para solucionar o problema do mapeamento, definem uma função que permite atribuir elementos de processamento a *tiles* na rede. A rede possui topologia malha, e parâmetros como largura de banda e consumo de energia são definidos para cada roteador. Diversos algoritmos que implementam funções multimídia como MP3 e H.263 são utilizados para realização dos experimentos, e o método de mapeamento utilizado pelos autores é o algoritmo *Cutting Tree*.

O problema do mapemento de tarefas em uma NoC com topologia malha é apresentado por Marcon em [47]. No trabalho, o grafo CWG é utilizado para representar o volume de dados de comunicações entre elementos de processamento e o CDG para representar a ordenação de mensagens. Os autores utilizam a técnica *Simulated Annealing* e também um método exaustivo para encontrar o melhor mapeamento. Comparado ao trabalho apresentado por Hu [27], foram obtidas melhorias significativas na qualidade do mapeamento, sendo o consumo de energia (redução de 21%) e tempo de execução (redução de 41%) as métricas utilizadas. Em [48] é utilizado um

novo grafo, chamado CDCG, o qual detalha o instante de envio de cada pacote de dados na rede, e o trabalho apresenta novamente resultados significativos comparado a trabalhos anteriores. Um trabalho mais recente [50] apresenta outras técnicas para a realização do mapeamento, entre elas pode-se enumerar *Greedy Incremental*, *Largest Communication First* e o método *Tabu Search*.

Orsila [66] modela aplicações com o uso de grafos direcionados, que incorporam informações sobre a dependência entre tarefas, o volume de dados transferido e o tempo de computação. Neste trabalho é apresentado um método para otimizar os parâmetros do algoritmo *Simulated Annealing*, com o intuito de diminuir o tempo necessário para encontrar soluções adequadas ao problema de mapeamento. Como resultado, observou-se que diferentes métodos devem ser utilizados para realização das otimizações, divididas como locais e globais. Uma extensão do trabalho é realizada em [67], apresentando resultados comparativos com relação ao uso de memória para a execução dos algoritmos utilizados.

No trabalho apresentado por Mehran [52], os autores modelam aplicações através de grafos de elementos de processamento, que detalham suas dependências e necessidades com relação a banda de comunicação. Um algoritmo de mapeamento chamado *Spiral* é proposto. Esse algoritmo procura mapear elementos de processamento de maneira espiral em uma NoC. Nos estudos realizados, foram experimentados mapeamentos com dimensões da rede entre 3x3 e 6x6 *tiles*, e os resultados apresentaram ganhos significativos na qualidade do mapeamento ao comparar seu método a algoritmos genéticos.

3.2 Mapeamento Dinâmico

Os trabalhos apresentados nessa Seção referem-se ao mapeamento dinâmico realizado em diversos ambientes MPSoC. Novamente, alguns trabalhos não detalham formalmente o modelo utilizado para a descrição das tarefas a serem particionadas ou mapeadas, mas preocupam-se com a descrição da técnica utilizada para a realização dessa atividade.

Ngouganga [63] apresenta uma solução para o mapeamento em uma arquitetura MPSoC homogênea, utilizando elementos de processamento que executam um *microkernel* e encontram-se conectados em uma NoC. No trabalho desenvolvido, define-se um elemento de processamento mestre e múltiplos escravos, sendo o mestre responsável pelo mapeamento dinâmico das tarefas no sistema. Inicialmente, não existe um mapeamento estabelecido, dessa forma o sistema dinamicamente atribui tarefas a processadores. Dois algoritmos são utilizados no trabalho, sendo eles *Simulated Annealing* e força direcionada. O tempo para realização do mapeamento é ignorado neste trabalho. Segundo os autores, o algoritmo de força direcionada apresentou um tempo de execução menor que o *Simulated Annealing*, e resultados comparados a soluções que utilizam mapeamento aleatório apresentaram boa escalabilidade.

O trabalho desenvolvido por Wronski [86] utiliza os algoritmos BF e WF para a realização de mapeamento dinâmico em uma arquitetura MPSoC. Um modelo implementado em alto nível é

utilizado para simular os elementos de processamento, e o meio de interconexão utilizado é uma NoC descrita em nível RTL. Através da co-simulação os autores conseguem estimar o consumo de energia da aplicação para diferentes tipos de mapeamento. Cada tarefa do sistema é representada por um valor de tempo de execução, e um número de chaveamentos (atribuído aleatoriamente). Em todos os experimentos foram utilizadas aplicações sintéticas geradas a partir do gerador TGFF. Os resultados apontam grandes mudanças no consumo de energia da aplicação, dependendo do mapeamento empregado. Brião [8] estende este trabalho e utiliza outros algoritmos, como a combinação da técnica de clusterização linear com os algoritmos BF e WF.

Hölsenspies [25] apresenta um trabalho voltado a estimativas na qualidade de mapeamentos de aplicações com requisitos de tempo real em arquiteturas MPSoC heterogêneas. As aplicações são modeladas como grafos direcionados, constituídos por tarefas e caminhos de comunicação representados por filas. Neste trabalho, um dos elementos de processamento é utilizado como gerente, e executa um sistema operacional dedicado. A gerência baseia-se em informações obtidas em tempo de projeto (número de processadores, estimativas em geral, etc.) e em tempo de execução toma decisões quanto ao mapeamento com o objetivo de reduzir o consumo de energia e garantir qualidade de serviço. Em [26] são incorporados mais detalhes ao modelo e apresentados mecanismos de mapeamento para áreas reconfiguráveis da arquitetura.

Chou [13] utiliza um método que realiza o mapeamento de elementos de processamento em MPSoCs homogêneos. Uma NoC com topologia malha é utilizada como meio de interconexão e são utilizados dois caminhos separados para controle e dados de uma determinada aplicação. No trabalho, as aplicações são modeladas a partir de grafos que representam o volume de dados e largura de banda exigidos pela aplicação. Um elemento de processamento gerente é utilizado como controlador do mapeamento, e o mesmo deve em tempo de execução alocar elementos de processamento disponíveis no MPSoC e dinamicamente realocar elementos de processamento com o intuito de diminuir a fragmentação da aplicação[1]. Segundo os autores, seu método apresenta melhorias significativas com relação a mapeamento aleatório em consumo de energia (45%) e pouca penalidade na qualidade (21%) comparado a um mapeamento exaustivo que teria um tempo de execução infinitamente maior para o tamanho do MPSoC empregado (dimensão 7x7). Este trabalho é complementado em [14], onde os autores utilizam o conceito de perfis baseados no comportamento do usuário para realizar o mapeamento dinamicamente, avaliando a periodicidade e volume de dados das tarefas. Diferentes técnicas de mapeamento são avaliadas, sendo uma proposta em [13] que utiliza áreas contíguas para mapear tarefas de uma aplicação, e a outra utiliza regiões em formato geométrico. São utilizadas aplicações reais para a realização dos experimentos.

Mehran [51] realiza um estudo sobre a qualidade do mapeamento dinâmico em um sistema MPSoC formado por uma rede de dimensão 4x4. Segundo os autores, o comportamento de uma determinada aplicação varia dinamicamente durante seu tempo de vida, o que motiva a utilização de diferentes mapeamentos durante sua execução. O algoritmo Spiral [52] é utilizado novamente neste

[1]Conforme [13] uma aplicação é dita fragmentada ou não contígua quando tarefas altamente comunicantes encontram-se distantes (com relação ao número de saltos na rede), o que acarreta em uma execução não otimizada da aplicação.

trabalho, onde os autores introduzem o conceito de migração parcial e total de tarefas. Nos testes realizados com diversas aplicações sintéticas, em diferentes cenários, foi observado que a migração parcial obteve melhores resultados para maioria das aplicações, e o critério utilizado foi a redução no tempo de mapeamento. No algoritmo em espiral, tarefas que possuem grandes taxas de dados entre si são alocadas de maneira que fiquem próximas umas das outras.

Al Faruque [2] realiza a descrição de aplicações utilizando grafos de dependência entre tarefas e largura de banda necessária. O mapeamento das tarefas é realizado de maneira distribuída, utilizando gerentes locais para diferentes elementos de processamento dentro de um mesmo *cluster* virtual e gerentes globais para selecionar determinado *cluster* dinamicamente. Assim, um mecanismo é implementado para sincronizar gerentes locais e globais e realizar as tomadas de decisão de maneira distribuída utilizando técnicas de seleção de *cluster*, migração e re-clusterização. São apresentadas comparações com outros trabalhos que utilizam heurísticas de mapeamento semelhante porém empregando o modelo centralizado, e resultados positivos na abordagem distribuída são encontrados em MPSoCs de grandes dimensões, com tamanhos a partir de 12x12 até 64x64 nodos.

Wildermann [85] propõe uma técnica para a realizar o mapeamento dinâmico de tarefas em um ambiente MPSoC homogêneo, onde o objetivo é reduzir o consumo de energia de uma determinada aplicação e garantir o cumprimento de requisitos de execução. Aplicações são descritas por grafos, e o conceito de tarefas mestre / escravo é utilizado para reproduzir o comportamento de aplicações do tipo *streaming*. A heurística proposta procura reduzir sobrecargas de comunicação geradas por tarefas dinamicamente mapeadas e utiliza o conceito de autômatos celulares, onde células modificam sua ocupação de acordo com o estado dos vizinhos. Uma avaliação da heurística é apresentada, e métricas como tempo médio de processamento de uma aplicação, *deadlines* e sobrecarga de comunicação são utilizadas. Para comparações, é utilizada a heurística NN, e apesar desta possuir um baixo *overhead*, a heurística proposta mostra-se mais eficiente pois apresenta o melhor compromisso levando em consideração as métricas utilizadas.

Zipf [89] apresenta uma heurística para a realização do mapeamento dinâmico de tarefas em ambientes MPSoC homogêneos. A proposta consiste em organizar o algoritmo de gerência de mapeamento de maneira distribuída, de forma que as dimensões do sistema sejam escaláveis. No trabalho, tarefas podem ser mapeadas em tempo de execução de acordo com informações de outros nodos sobre a carga de processamento, tamanho das tarefas, requisitos de comunicação e contenções na rede. Para a obtenção de resultados os autores utilizam um protótipo em uma malha de dimensões 3x3. Segundo os autores, a solução proposta possui uma penalidade de 25% perante um algoritmo exato. Ainda, a qualidade do mapeamento em malhas de maiores dimensões, onde o algoritmo *Simulated Annealing* é utilizado como referência é reduzida em 30%. De acordo com a proposta, estas penalidades são razoáveis, uma vez que existe um baixo custo computacional e de comunicação e dessa forma a solução descentralizada torna-se viável para sistemas dinâmicos com grande número de processadores.

As heurísticas NN e BN propostas por Carvalho [12] são complementadas por Singh [82, 83] para incluir o suporte multitarefa ao mapeamento de aplicações dinâmicas. As aplicações são des-

critas em grafos em forma de árvore, onde uma tarefa inicial é utilizada para mapear outras. Estas tarefas iniciais (uma para cada aplicação) são dispostas em regiões centrais de grupos virtuais de elementos de processamento (clusters), e processadores são alocados de acordo com a necessidade da aplicação dentro destes grupos. Além disso, é apresentado em [82] o conceito de agrupamento de tarefas com o objetivo de alocar tarefas comunicantes em um mesmo elemento de processamento. Em [83] é utilizada uma variação para a alocação de tarefas em um mesmo elemento de processamento (agrupamento), levando-se em consideração o histórico de tarefas mapeadas em um dado elemento de processamento para tomar-se a decisão de mapeamento local (tarefa comunica-se com alguma previamente mapeada localmente) ou em outro nodo.

Carvalho [12, 11] propõe diversas heurísticas dinâmicas para o mapeamento de tarefas em MPSoCs heterogêneos com o objetivo de reduzir congestionamentos na rede. Para isso, pares de tarefas comunicantes são aproximadas, diminuindo o número de saltos entre estas e consequentemente os caminhos congestionados. Aplicações são modeladas por grafos que retratam tarefas e comunicações entre estas. Novas tarefas são mapeadas dinamicamente a partir de tarefas iniciais neste modelo. De acordo com o trabalho, o custo médio das soluções propostas comparadas ao método estático de referência é em torno de 16% com relação ao consumo de energia e volume de dados. Diversas aplicações reais são modeladas para a obtenção de resultados, e algoritmos como MPEG4, MWD, RBERG e VOPD são avaliados com as heurísticas propostas.

No trabalho de Schranzhofer [78] informações sobre o mapeamento da aplicação são definidas em tempo de projeto e incluídas em *templates* que são armazenados em tabelas. Em tempo de execução o sistema é avaliado por um mecanismo de gerência, e dinamicamente o mapeamento mais adequado é selecionado. De acordo com os resultados, o custo da solução proposta é bastante baixo e a redução do consumo de energia comparado a um trabalho publicado pelo mesmo autor anteriormente é em torno de 45%. O modelo de aplicação utilizado é formado por tarefas que possuem determinado uso de processador e consumo de energia e entre estas tarefas são definidas transações.

Braak [7] propõe a gerência de recursos em tempo de execução com o objetivo de prover um certo nível de flexibilidade e tolerância a falhas em sistemas MPSoC heterogêneos devido a imperfeições na fabricação de circuitos integrados e falhas decorrentes de desgastes. Inicialmente, é realizado um particionamento da aplicação em tarefas. Após, em tempo de execução a alocação de recursos é realizada em quatro fases: *binding*, *mapping*, *routing* e *validation*. Tarefas são descritas por grafos e possuem requisitos de tempo de processamento e comunicação. Os experimentos realizados demonstram que a solução é viável, e que o tempo de alocação por recurso é medido em dezenas de milissegundos. A ferramenta TGFF é utilizada para gerar diferentes cenários de aplicações sintéticas, sendo tais cenários variações entre conjuntos de tarefas altamente comunicantes e com alta utilização de tempo de processador.

Mandelli [44] apresenta novas heurísticas para a realização do mapeamento dinâmico de tarefas em MPSoCs homogêneos com o objetivo de reduzir o consumo de energia e explorar a dependência entre tarefas em tais ambientes. O trabalho de Carvalho [11] é utilizado como referência

para o modelo de aplicação empregado, que consiste em tarefas iniciais e tarefas mapeadas dinamicamente a partir destas. O mesmo trabalho é utilizado nos resultados obtidos, onde as heurísticas NN, BN, PL e BN são comparadas a outras propostas, como LEC-DN (mapeamento), Premap (clusterização) e Premap-DN (combinação entre métodos). Uma contribuição significativa é a extensão do modelo de tarefas e das heurísticas para incluir o suporte multi-tarefa, algo pouco explorado por trabalhos na área.

3.3 Migração de Tarefas

Nollet [65] apresenta um estudo que utiliza um gerenciamento centralizado de recursos em tempo de execução através do uso de uma heurística para migração de tarefas em um ambiente heterogêneo. O trabalho baseia-se em uma plataforma que conta com um processador *ARM* conectado a elementos de processamento escravos, interconectados por uma NoC malha 3x3. No trabalho, um sistema operacional centralizado realiza o gerenciamento e alocação dos recursos computacionais, e na aplicação são definidos pontos de migração que podem ser utilizados pelo sistema operacional. Os nodos escravos são implementados em uma FPGA e são reconfiguráveis em tempo de execução, o que permite a execução de tarefas tanto em *hardware* quanto em *software*. A arquitetura utilizada é apresentada por Mignolet [54]. O objetivo de utilizar migrações de tarefa no contexto do trabalho ocorre em virtude de modificações nas características da aplicação ou falhas durante o mapeamento de recursos. Em [64] o trabalho é complementado com o intuito de diminuir o tempo de reação encontrado nos mecanismos de migração previamente propostos. Os autores propõem uma técnica de reutilização de registradores de depuração encontrados no processador utilizado (*PowerPC 405*) com o intuito de diminuir o *overhead* inicial de migrações de tarefas.

O trabalho de Bertozzi [6] aborda migrações de tarefas em MPSoCs e propõe um mecanismo onde um sistema operacional (uCLinux modificado) centralizado realiza a gerência. Na aplicação, implementada seguindo um modelo mestre escravo, o usuário é responsável por definir pontos de migração. Tarefas mestre são responsáveis por realizar a admissão e alocação de recursos, objetivando o balanceamento de carga no sistema. Como resultados, os autores realizam uma série de comparações e concluem que o *overhead* gerado pelo mecanismo implementado é baixo se comparado a soluções distribuídas, que podem inviabilizar a utilização de migração de tarefas.

Ozturk [69] apresenta uma nova abordagem para o processo de migração, denominada de migração seletiva. Essa abordagem é descrita por três componentes principais: personalização, anotação de código e migração seletiva. As primeiras duas etapas são realizadas em tempo de projeto. Assim, a etapa de personalização realiza a coleta do custo energético da migração de fragmentos específicos tanto de código quanto de dados através da rede de comunicação. Os custos são anotados no código e utilizados durante a migração seletiva que acontece em tempo de execução. A arquitetura utilizada no trabalho é composta por oito elementos de processamento 32 kB de memória local, interligados por um barramento. Um conjunto de *benchmarks* criado pelos

autores é utilizado para a obtenção de resultados, sendo o consumo de energia e tempo de execução das aplicações as métricas utilizadas para a avaliação.

Carta [10] apresenta o algoritmo *MiGra* que tem como objetivo reduzir gradientes de temperatura em MPSoCs, utilizando migrações de tarefas. O mecanismo apresentado baseia-se em valores de temperatura do *chip* que são medidos em tempo de execução para tentar balancear sua temperatura sem contudo aumentar o consumo de energia. O algoritmo *MiGra*, diferentemente de outras técnicas que reagem quando são observadas altas temperaturas, tenta fazer com que as temperaturas dos elementos de processamento fiquem em uma média do sistema como um todo. Dessa forma, tarefas podem ser migradas mesmo de nodos considerados mais frios, ao contrário de outras técnicas que migram tarefas somente de nodos mais quentes. Ao término de uma migração, é observado o consumo total de energia do *chip*. Como resultado, são apresentadas comparações entre abordagens que realizam balanceamento de carga simples e balanceamento de carga que leva em consideração o consumo de energia.

Em Götz [22] é apresentado um fluxo para a realização da realocação dinâmica de tarefas híbridas, as quais podem executar tanto em *hardware* quanto em *software*. No modelo proposto, tarefas são representadas por um grafo de transição de estados e cada estado é denominado bloco de computação, que representa uma determinada operação de uma tarefa. Na descrição do grafo são verificados pontos de encontro entre as versões *software* e *hardware* da tarefa. Ainda na descrição são representados pontos de troca, onde pode ser realizada a realocação de tarefas. Nestes pontos existe somente um contexto que precisa ser salvo. O modelo é implementado em uma ferramenta que é capaz de gerar as versões de *hardware* e *software* da tarefa juntamente com um componente de gerenciamento da migração, relacionado com um sistema operacional.

Pittau [72] apresenta um estudo sobre uso da migração de tarefas e seu impacto em aplicações multimídia em MPSoCs. No trabalho são empregados elementos de processamento que possuem a funcionalidade que permite variar a frequência de operação, sendo estes interconectados por um barramento. Cada elemento de processamento possui uma memória local, onde são armazenados os dados das tarefas. O mecanismo de migração realiza a transferência destes dados para um módulo de memória compartilhado entre os elementos de processamento no evento de uma migração, com o intuito de reduzir o *overhead* quando comparado a uma solução que utiliza puramente trocas de mensagens.

A proposta de Brião [8] leva em consideração o *overhead* da migração de tarefas em um ambiente dinâmico e apresenta seu impacto em termos de consumo de energia, desempenho e restrições de tempo real no contexto de MPSoCs baseados em NoC. No trabalho, foi desenvolvida uma ferramenta capaz de simular o comportamento de sistemas baseados em NoC que executam tarefas geradas pela ferramenta TGFF, as quais são dinamicamente carregadas. A migração de tarefas é executada baseada em um modelo de cópia da tarefa, que consiste em migrar todo o contexto e código. Segundo os autores, a migração de tarefas pode ser utilizada em sistemas embarcados visto que apresenta ganhos em termos de desempenho e redução no consumo de energia envolvidos no sistema e ainda garantir o cumprimento de *deadlines* em sistemas *soft real-time*.

No trabalho de Mulas [59] que consiste em um complemento dos trabalhos de Carta [10] e Pittau [72], o processo de migração de tarefas é realizado por um *middleware* (MPOS) e o principal objetivo é obter uma política de balanceamento térmico e energético para aplicações de *streaming* em ambientes MPSoC levando em consideração o desvio padrão da temperatura dos elementos de processamento mais quentes e mais frios. Assim como no trabalho anterior, migrações são feitas através de um mecanismo denominado pelos autores por *replicação de tarefas*, que consiste em se manter uma cópia de cada tarefa em todos os processadores. Segundo os autores, manter uma cópia reduz o *overhead* natural de uma migração embora implique em maior área necessária em memória.

Marchesan [46] apresenta um mecanismo para a realização de migrações de tarefa em um MPSoC baseado em NoC. No trabalho, tarefas são especificadas com requisitos de computação e filas de comunicação e são gerenciadas por um sistema operacional que executa em cada nodo da arquitetura, fornecendo serviços de comunicação e migração de tarefas. No mecanismo implementado, são definidos pontos de migração nas chamadas às primitivas de comunicação, dessa forma migrações de tarefa apenas ocorrem quando tarefas comunicam-se. O objetivo do trabalho é apresentar a viabilidade da implementação de mecanismos de migração em plataformas altamente distribuídas e escaláveis, onde cada nodo da arquitetura é caracterizado por possuir uma memória local e filas de comunicação em *hardware*. Para a obtenção de resultados são utilizadas aplicações reais (DES e MJPEG), onde comparações entre decisões de migração baseadas em utilização de processador e localidade das tarefas são realizadas.

Shen [81] propõe a utilização de uma arquitetura MPSoC heterogênea configurável baseada em elementos de processamento que possuem o mesmo conjunto de instruções básico. A idéia é utilizar estes elementos de processamento em diferentes versões e realizar migrações de tarefas, sendo o conjunto de instruções extendido para a execução de tarefas específicas com o intuito de melhorar a execução destas e reduzir o consumo de energia do sistema. Os autores argumentam que um grande percentual da execução de aplicações utiliza apenas o conjunto de instruções básico, e que a heterogeneidade e homogeneidade da plataforma deve ser balanceada para que sejam obtidas melhorias com extensões dos elementos de processamento. Assim, tarefas podem ser migradas entre elementos de processamento apenas se o elemento destino possuir todas as instruções necessárias para a execução destas. O trabalho se concentra na infra-estrutura necessária para suportar o escalonamento de tarefas e migração entre elementos de processamento, e a aplicação utilizada no estudo de caso apresentado é o MJPEG.

Cuesta [16] explora os benefícios de técnicas de migração *thermal-aware* em sistemas MPSoC. Os autores propõem políticas para a redução da temperatura média do *chip* e gradientes de temperatura com pouco impacto no desempenho. No trabalho são propostas três políticas de migração baseadas em funções adaptáveis para três fatores distintos: média do desvio padrão de temperatura entre processadores, temperatura máxima do *chip* e gradiente termal entre os núcleos. Os experimentos realizados foram desenvolvidos em uma plataforma de emulação MPSoC (composta por um processador *PowerPC* mestre e outros oito RISC escravos interconectados por barramentos) e os mecanismos propostos integrados em um sistema operacional que realiza a migração de tarefas

e medições de temperatura. Segundo os autores, a redução na temperatura comparada a outros trabalhos relacionados é em torno de 30%.

Cannella [9] propõe a descrição de aplicações utilizando PPNs em um ambiente MPSoC homogêneo. Cada nodo possui uma memória local, e é interconectado a outros por uma NoC e filas de comunicação. No trabalho, um sistema operacional fornece serviços de comunicação e migração de tarefas. Migrações de tarefas são utilizadas para garantir determinado aspecto (modelado na aplicação) como qualidade de serviço, disponibilidade de recursos ou consumo de energia. Duas aplicações reais são avaliadas nos resultados (Sobel e MJPEG), e a plataforma como um todo é a amplamente discutida apresentando resultados de *overhead* para os protocolos de comunicação e migração de tarefas. Ainda, a aplicação MJPEG é explorada no intuito de variar as métricas de desempenho em tempo de execução, utilizando para isso a modelagem proposta e o mecanismo de migração.

3.4 Análise Comparativa e Questões em Aberto

As Tabelas 3.1 e 3.2 apresentam um resumo das principais características dos trabalhos relacionados ao mapeamento e migração de tarefas em sistemas embarcados dos últimos anos. Como pode ser observado, há uma grande quantidade de trabalhos abordando tanto o mapeamento estático quanto o mapeamento dinâmico e migração de tarefas, com um crescente interesse com relação a sistemas dinâmicos em trabalhos recentes [83, 11, 78, 7, 44, 16, 9].

Apesar das diversas vantagens na utilização de arquiteturas heterogêneas, a maioria dos trabalhos utiliza um modelo homogêneo. Grande parte dos autores argumenta que o espaço de projeto é mais facilmente explorável neste tipo de arquitetura, e que os modelos podem ser simplificados significativamente. Ainda, existe uma forte tendência a se transferir parte da complexidade na implementação de sistemas MPSoC para o *software*, uma vez que é possível desenvolver *chips* com centenas de núcleos com tecnologias de fabricação atuais. Assim, a replicação de núcleos com as mesmas características facilita o processo de desenvolvimento do *software* e do *hardware* e permite uma maior flexibilidade da arquitetura para diferentes aplicações. No presente trabalho será adotada uma arquitetura MPSoC homogênea com memórias distribuídas, uma vez que de acordo com trabalhos recentes esta é a abordagem mais adequada para um grande número de elementos de processamento.

Boa parte dos trabalhos utiliza reserva de recursos, que consiste em alocar espaço para todas as tarefas da aplicação mesmo que existam tarefas que não estejam executando no momento. A reserva de recursos pode ser utilizada para garantir QoS, entretanto as restrições impostas por este tipo de mecanismo são grandes: (i) subutilização dos recursos computacionais e; (ii) dimensionamento pessimista da arquitetura, pois podem ser necessários mais núcleos devido a reserva. Segundo Mandelli [44] a reserva de recursos pode aumentar o tempo de execução da aplicação pois é necessário que haja espaço para todas as tarefas de uma dada aplicação, e esta deixa de executar

até que isto aconteça. Em um sistema sem reservas, tarefas podem executar a medida em que for necessário se houver um recurso disponível, não sendo necessário o mapeamento completo da aplicação. Dessa forma, no presente trabalho será utilizada a abordagem sem reserva de recursos.

O emprego de mecanismos de mapeamento e migração que fornecem suporte multi-tarefa é crescente em trabalhos recentes [89, 83, 44, 46, 81, 16, 9]. A abordagem multi-tarefa possui uma série de vantagens, tais como redução do tráfego na rede de interconexão (tarefas podem ser mapeadas para um mesmo elemento de processamento), redução do tamanho da arquitetura MPSoC, uma vez que um mesmo nodo pode ser compartilhado por diversas tarefas e maior flexibilidade em aplicações dinâmicas. Muitas vezes, tarefas não possuem grande complexidade computacional e alocá-las em um nodo que não possui recursos multi-tarefa inevitavelmente subutiliza a arquitetura. Seguindo a idéia proposta em trabalhos recentes, será adotado neste trabalho a abordagem multi-tarefa. Este modelo vem ao encontro do modelo multi-programado que predomina em sistemas operacionais de propósito geral.

São três os modelos de gerenciamento para a realização do mapeamento dinâmico e migração de tarefas encontrados na literatura: (i) controlado pelo usuário; (ii) centralizado e; (iii) distribuído. A maioria dos trabalhos, com exceção de Al Faruque [2], Zipf [89], Marchesan [46] e Cannella [9] utilizam o conceito de gerenciamento centralizado. No modelo centralizado, um elemento de processamento mestre controla todos os outros presentes no sistema e possui uma visão global das características da aplicação. O modelo centralizado permite a implementação de mecanismos eficientes do ponto de vista da qualidade do mapeamento dinâmico, ao custo da sua escalabilidade. Novos sistemas que possuem um grande número de processadores e tarefas tornarão o trabalho do mestre cada vez maior, sobrecarregando-o. No modelo distribuído, a responsabilidade pelo mapeamento dinâmico é compartilhada entre todos os elementos de processamento [89]. Trocas de mensagem são necessárias para que sejam conhecidas características da aplicação e estado dos outros nodos, uma vez que não existe visão global do sistema. Dessa forma, as decisões relacionadas ao mapeamento dinâmico são realizadas pela interação entre nodos. Apesar da gerência distribuída do mapeamento dinâmico em MPSoCs ser pouco explorada até o presente momento, esta será a abordagem adotada pois acredita-se que no futuro o modelo centralizado será um gargalo. O modelo de gerência distribuído proposto é uma contribuição deste trabalho.

De maneira geral, tarefas de aplicações embarcadas podem ser facilmente representadas por grafos, sendo esta a representação utilizada pela maioria dos trabalhos revisados. Alguns trabalhos não possuem uma representação em alto nível [6, 69, 10, 16], e descrevem seu modelo como sendo parte da própria implementação. As principais características representadas nos modelos estudados referem-se ao volume de dados e largura de banda, uma vez que estes são de grande interesse em ambientes que empregam NoCs como meio de interconexão. O processamento das tarefas nos nodos é muitas vezes representado simplesmente por ciclos de relógio ou utilização do processador e não é frequentemente explorado nos modelos. Alguns trabalhos ignoram o conceito de *contexto de tarefa*, o que simplifica o modelo de mapeamento dinâmico e migração, mas restringe muito a aplicabilidade destes modelos em aplicações reais.

Outro conceito pouco utilizado é a implementação de algoritmos de escalonamento de tempo real em sistemas MPSoC e representação dos parâmetros das tarefas em um modelo em alto nível. A criação de uma representação em alto nível que incorpora parâmetros de tempo real é uma contribuição original deste trabalho, uma vez que nos trabalhos encontrados na literatura este conceito é pouco explorado. Com relação à programabilidade, muitas vezes a aplicação é modelada simplesmente como transferências de fluxos de dados, e as funcionalidades disponibilizadas restringem-se a primitivas de comunicação. O presente trabalho propõe um modelo de programação mais completo do ponto de vista da aplicação, onde um sistema operacional fornece suporte semelhante a sistemas de propósito geral e facilita o desenvolvimento de aplicações embarcadas.

A maior parte dos trabalhos revisados utilizam como função de custo o consumo de energia. Outros trabalhos consideram o tempo de execução da aplicação, utilização dos canais da rede e temperatura do *chip*, e indiretamente redução no consumo de energia de maneira proporcional. A redução no consumo de energia é de extrema relavância em ambientes MPSoC, no entanto outros problemas precisam ser atacados devido a flexibilidade de sistemas dinâmicos, que permitem a inclusão de novas funcionalidades em tempo de execução e possuem um perfil variável de execução. Assim, este trabalho utiliza como função de custo a utilização de processador e garantias de tempo real da aplicação, sendo esta uma abordagem pouco explorada nos trabalhos da área.

Segundo Zipf [89] a migração de tarefas diferencia-se do mapeamento dinâmico uma vez que no segundo novas tarefas podem ser mapeadas em tempo de execução. O autor deste trabalho discorda, pois o fato de existir um mecanismo que permita a migração de tarefas de um nodo a outro não invalida a possibilidade de existir outro mecanismo que permita a realização do mapeamento dinamicamente. Apesar do presente trabalho estar inserido na classificação *migração de tarefas*, foram modelados e descritos mecanismos para a realização do mapeamento dinâmico. O mecanismo de migração é uma contribuição significativa deste trabalho, sendo este o principal motivo da classificação utilizada.

3.5 Considerações Finais

Este Capítulo apresentou os principais trabalhos encontrados na literatura relacionados ao tema proposto. Dessa forma, o presente trabalho tem como intuito complementar o estado da arte, apresentando um modelo para a realização do mapeamento dinâmico e migração de tarefas em sistemas MPSoC homogêneos com grande número de nodos. Foram identificadas algumas limitações com relação aos modelos de tarefa, programabilidade e gerência destas em ambientes dinâmicos, e dessa forma espera-se contribuir para o avanço desta área de pesquisa.

Tabela 3.1 – Resumo comparativo entre os trabalhos sobre mapeamento

Autor	Tipo de Mapeamento	Arquitetura	Reserva de Recursos	Multi-tarefa	Gerenciamento	Modelo de Aplicação	Infra-estrutura de Comunicação	Algoritmo	Função Custo
Mihal [55] (2003)	Estático	Heterogênea	N/A	Sim	N/A	Modelo em UML em alto nível	Barramento	N/A	N/A
Lei [38, 37] (2003)	Estático	Heterogênea	N/A	Não	N/A	Volume de dados na comunicação (grafo)	NoC malha	Genético	Tempo de execução
Rhee [75] (2004)	Estático	Homogênea	N/A	Não	N/A	Volume de dados na comunicação e largura de banda (grafos)	NoC malha	Mapeamento manual	Ocupação dos canais e saltos entre roteadores
Hu [27, 28] (2004, 2005)	Estático	Homogênea	N/A	Sim	N/A	Volume de dados na comunicação e tempo de execução (grafos)	NoC malha	Divisão e conquista	Consumo de energia
Murali [61, 62, 60] (2004, 2006)	Estático	Homogênea	N/A	Não	N/A	Volume de dados na comunicação e largura de banda (grafos)	NoC malha e toro	Tabu Search	Largura de banda, atraso, consumo de energia e área
Manolache [45] (2005)	Estático	Homogênea	N/A	Sim	N/A	Período, tempo de execução e limite de tempo (grafos)	NoC malha	Tabu Search	Garantias de latência e consumo de energia
Srinivasan [84] (2005)	Estático	Homogênea	N/A	Não	N/A	Largura de banda e latência (grafos)	NoC malha	Cutting Tree	Latência e largura de banda
Marcon [47, 48, 50] (2005, 2007)	Estático	Homogênea	N/A	Não	N/A	Volume de dados, dependência e tempo de execução (grafos)	NoC malha	Força bruta, Simulated Annealing e outros	Consumo de energia e tempo de execução
Orsila [66, 67] (2006, 2007)	Estático	Homogênea	N/A	Sim	N/A	Volume de dados na comunicação, dependência e tempo de execução (grafos)	NoC malha	Simulated Annealing e outros	Tempo de execução
Mehran [52] (2007)	Estático	Homogênea	N/A	Não	N/A	Largura de banda e dependência (grafos)	NoC malha	Spiral	Ocupação dos canais de comunicação e saltos entre roteadores
Ngouanga [63] (2006)	Dinâmico	Homogênea	Sim	Não	Centralizado	Volume de dados na comunicação (grafo)	NoC malha	Simulated Annealing e força direcionada	Tempo de comunicação
Wronski [86] (2006)	Dinâmico	Homogênea	Sim	Não	Centralizado	Tempo de execução e número de chaveamentos (grafos)	NoC malha	Best Fit, Worst Fit e outros	Consumo de energia e tempo de execução
Hölsenspies [25, 26] (2007, 2008)	Dinâmico	Heterogênea	Sim	Não	Centralizado	Volume de dados na comunicação	NoC malha	Método iterativo	Consumo de energia e qualidade de serviço
Chou [13, 14] (2007, 2008)	Dinâmico	Homogênea	Sim	Não	Centralizado	Volume de dados na comunicação e largura de banda (grafos)	NoC malha	Seleção e mapeamento	Consumo de energia e fragmentação da aplicação
Mehran [51] (2008)	Dinâmico	Homogênea	Sim	Não	Centralizado	N/A	NoC malha	Spiral	Tempo de mapeamento e consumo de energia
Al Faruque [2] (2008)	Dinâmico	Heterogênea	Não	Não	Distribuído	Volume de dados e largura de banda (grafos)	NoC malha	Gerentes distribuídos	Tempo de mapeamento
Wildermann [85] (2009)	Dinâmico	Homogênea	Não	Não	Centralizado	Mestre / escravo com tempo de execução e volume de dados (grafos)	NoC malha	LD, NN, RT, RT+N	Tempo de execução, consumo de energia e latência
Zipf [89] (2009)	Dinâmico	Homogênea	Não	Sim	Distribuído	Tarefas com filas de comunicação (grafos)	NoC malha	Gerentes distribuídos	Utilização de processador, distância entre tarefas
Singh [82, 83] (2009, 2010)	Dinâmico	Heterogênea	Não	Sim	Centralizado	Tarefas comunicantes (grafos em árvore)	NoC malha	Agrupamento, NN-MT, BN-MT	Contenção, consumo de energia e volume de dados
Carvalho [12, 11] (2008, 2010)	Dinâmico	Heterogênea	Não	Não	Centralizado	Volume de dados e largura de banda (grafos)	NoC malha	FF, NN, PL, BN, MACL, MMCL	Contenção e volume de dados
Schranzhofer [78] (2010)	Dinâmico	Homogênea	Sim	Não	Centralizado	Tarefas com utilização de processador, consumo de energia e transações (grafos)	NoC malha	Compute Paths	Consumo de energia
Braak [7] (2010)	Dinâmico	Heterogênea	Sim	Não	Centralizado	Tempo de processamento e comunicação (grafos)	NoC malha	Divisão e conquista	Consumo de energia, fragmentação, balanceamento de carga, desgaste
Mandelli [44] (2011)	Dinâmico	Homogênea	Não	Sim	Centralizado	Volume de dados e largura de banda (grafos)	NoC malha	LEC-DN, Premap, Premap-DN	Consumo de energia

Tabela 3.2 – Resumo comparativo entre os trabalhos sobre migração

Autor	Arquitetura	Reserva de Recursos	Multi-tarefa	Gerenciamento	Modelo de Aplicação	Infra-estrutura de Comunicação	Algoritmo	Função Custo
Nollet [65, 64] (2005)	Heterogênea	Não	Sim	Controlado pelo usuário	Tarefas com QoS e volume de dados (grafos)	NoC malha	Pontos de migração (gerenciado pelo SO)	Modificação do perfil e falhas
Bertozzi [6] (2006)	Homogênea	Não	Sim	Centralizado	Tarefas mestre / escravas	NoC malha	Pontos de migração	Balanceamento de carga
Ozturk [69] (2006)	Homogênea	Não	Não	Centralizado	Tarefas de *hardware* e *software*	Barramento	Migração seletiva	Consumo de energia e tempo de execução
Carta [10] (2007)	Homogênea	Não	Sim	Centralizado	Tarefas com utilização de processador	NoC malha	MiGra	Temperatura, consumo de energia
Götz [22] (2007)	Heterogênea	Não	Sim	Controlado pelo usuário	Blocos de computação e pontos de encontro (grafos)	Barramento	Pontos de migração	N/A
Pittau [72] (2007)	Homogênea	Não	Sim	Controlado pelo usuário	Tarefas e filas de comunicação	Barramento	Pontos de migração	Consumo de energia
Brião [8] (2008)	Homogênea	Sim	Sim	Centralizado	Tempo de execução e número de chaveamentos (grafos)	NoC malha	Best Fit, Worst Fit e outros	Consumo de energia e tempo de execução
Mulas [59] (2008)	Homogênea	Não	Sim	Centralizado	Tarefas com utilização de processador	NoC malha	MiGra / DVFS (gerenciado pelo SO)	Temperatura, consumo de energia
Marchesan [46] (2009)	Homogênea	Sim	Sim	Distribuído	Tarefas com tempo de execução e filas de comunicação (grafos)	NoC malha	Serviço de melhoria, pontos de migração (gerenciado pelo SO)	Utilização de processador, distância entre tarefas
Shen [81] (2009)	Heterogênea	Não	Sim	Centralizado	Tarefas com tempo de processador e comunicação (grafos)	Barramento e NoC	First Match First Served	Tempo de execução
Cuesta [16] (2010)	Heterogênea	Não	Sim	Centralizado	N/A	Barramento	Pontos de migração (gerenciado pelo SO)	Temperatura
Cannella [9] (2011)	Homogênea	Não	Sim	Distribuído	Tarefas e filas de comunicação (PPNs)	NoC malha	Serviços do SO	QoS, consumo de energia
Proposta	Homogênea	Não	Sim	Distribuído	Tarefas de tempo real e melhor esforço caracterizadas, volume de dados (grafos)	NoC malha	Busca por espalhamento em situação de sobrecarga (gerenciado pelo SO)	Tempo real da aplicação (*deadlines*), utilização de processador, tempo de migração

4. MODELO PROPOSTO

Neste Capítulo é apresentado o modelo utilizado para representar características da aplicação e arquitetura MPSoC no contexto deste trabalho. Inicialmente a organização da arquitetura adotada é discutida, a definição do problema é apresentada e definições gerais são utilizadas como introdução ao modelo formal aqui descrito.

4.1 Organização da Arquitetura

O emprego de um mesmo tipo de elemento de processamento para a construção da arquitetura MPSoC caracteriza uma implementação homogênea e de diferentes tipos caracteriza uma implementação heterogênea. A utilização de elementos de processamento diferentes entre si, e específicos a determinadas tarefas garantem uma melhor utilização da área em silício e redução no consumo de energia [42, 54, 64], devido a especialização destes elementos para determinada aplicação. No entanto, a complexidade da organização do ponto de vista de projeto é aumentada em arquiteturas com dezenas ou mesmo centenas de processadores, em virtude da perda de regularidade. Além disso, o espaço de soluções para o mapeamento de tarefas é comprometido, uma vez que nem todos os elementos de processamento podem executar as mesmas operações [63]. Outro ponto seria a redução da modularidade (o que aumenta o tempo de projeto) e o aumento da complexidade na programação do *software* que executa em tais arquiteturas. A programabilidade de tais plataformas é um fator importante a ser avalidado, sendo este um dos maiores desafios enfrentados em arquiteturas com grande número de processadores.

Em relação ao meio de interconexão, diversos trabalhos [28, 61, 8, 13] citam NoCs como o principal meio de comunicação utilizado em arquiteturas MPSoC devido a sua escalabilidade e modularidade comparado a barramentos multiponto e conexões ponto a ponto. Os problemas enfrentados em arquiteturas baseadas em barramento tornam-se mais evidentes com o aumento do número de processadores, sendo estes o aumento de colisões de mensagens (o meio é compartilhado), limitações físicas como o aumento da carga capacitiva e redução na frequência de operação [47]. O emprego de NoCs contorna esses problemas pois o número de processadores é escalável do ponto de vista físico (são utilizados fios curtos e não compartilhados) e comunicações entre diferentes elementos de processamento podem ocorrer de forma paralela [88].

Um grande número de trabalhos utiliza arquiteturas MPSoC homogêneas por motivos práticos [14]. Entre as vantagens estão a diminuição do tempo de projeto, melhor modularização, maior facilidade de exploração do espaço de projeto utilizando ferramentas de alto nível, possibilidade do emprego de algoritmos mais simples para a realização do mapeamento de tarefas e reutilização da mesma arquitetura para diferentes produtos. A organização homogênea empregada neste trabalho é caracterizada por um conjunto de elementos de processamento idênticos conectados por uma NoC com topologia malha de duas dimensões.

4.2 Definições Gerais

Neste trabalho, são definidas descrições em alto nível que modelam características da arquitetura e da aplicação. O objetivo destas descrições é simplificar a definição dos processos de mapeamento e reconfiguração de aplicações multiprocessadas no contexto do ambiente MPSoC apresentado.

É possível representar separadamente a aplicação e a estrutura aonde a mesma irá executar. Assim, diferentes partes da plataforma podem ser enumeradas como pertencentes a duas principais categorias, sendo estas *arquitetura* e *aplicação*. A categoria *arquitetura* é constituída por:

- *Hardware*

  Elementos de processamento

  Meio de interconexão

- *Software*

  Sistema operacional

A categoria *aplicação* é constituída por um conjunto de tarefas, onde cada uma possui as propriedades a seguir. Basicamente, uma tarefa possui um tempo de computação e um tempo de comunicação. Tarefas periódicas têm definido seus parâmetros de intervalo entre execuções (ou invocações), o tempo que executam entre estes intervalos e o limite de tempo para o término de execução. Tarefas aperiódicas não possuem definição de intervalo de tempo entre suas execuções. No contexto deste trabalho, tarefas periódicas possuem parâmetros de tempo real (período, tempo de execução e *deadline*) e as aperiódicas são tarefas de melhor esforço ou eventos externos, como interrupções por exemplo. Não são definidas tarefas periódicas de melhor esforço no presente modelo, entretanto podem receber tal classificação tarefas que estejam perdendo *deadlines*. A comunicação entre tarefas é feita por trocas de mensagem ou memória compartilhada, e que no segundo caso pode ser encapsulada pelo sistema operacional (mantendo o modelo de trocas de mensagem) ou realizada diretamente por uma região de memória compartilhada e protegida por exclusão mútua. As propriedades de uma tarefa são resumidas em:

- Computação

  Periódicas

    Tarefas de tempo real - RT

    Melhor esforço - NRT

  Aperiódicas

    Eventos - RT

    Melhor esforço - NRT

- Comunicação
  - Trocas de mensagem
  - Memória compartilhada
    - Diretamente por exclusão mútua
    - Encapsulada pelo sistema operacional

A arquitetura de *hardware* é formada por uma série de nodos compostos por um elemento de processamento com uma interface de rede e um roteador da NoC. Em cada processador, uma instância de sistema operacional é executada (arquitetura de *software*), e cada sistema operacional gerencia uma ou mais tarefas da aplicação. As tarefas fazem uso de serviços fornecidos pelo sistema operacional. Os nodos conectados em rede, formando uma malha de duas dimensões compõem a arquitetura. A Figura 4.1 apresenta um exemplo de tal arquitetura, separando os elementos de *hardware* e *software*.

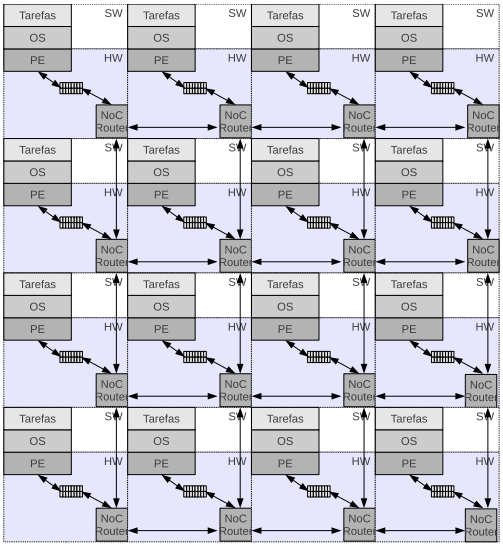


Figura 4.1 – Arquitetura MPSoC proposta

A aplicação é constituída por um conjunto de tarefas distribuídas entre nodos da rede. Estas tarefas podem realizar determinado processamento e comunicar entre si localmente ou remotamente de maneira transparente. Outras tarefas podem entrar, serem replicadas ou saírem do sistema durante o tempo de execução. Tarefas possuem parâmetros que serão detalhados adiante, e podem executar em um mesmo elemento de processamento, compartilhando recursos como tempo de processador e acesso ao meio de interconexão.

O modelo proposto faz a utilização de grafos para representar características tanto da arquitetura quanto da aplicação mapeada na mesma. Este modelo será apresentado na Seção 4.5. Apenas as tarefas necessárias para a inicialização da aplicação precisam estar mapeadas antes da execução do sistema, uma vez que mecanismos de mapeamento dinâmico e migração de tarefas fornecidos pelo sistema operacional podem ser utilizados para reconfigurar a aplicação em tempo de execução.

4.3 Definição do Problema

Diversos trabalhos [67, 47, 61] apresentam soluções para o mapemento de aplicações de maneira estática. Embora sejam estas soluções ideais para aplicações que possuam um comportamento conhecido em tempo de projeto, o mesmo não é verdadeiro para aplicações que variam suas características durante o tempo de execução. Outro caso seria o de sistemas que permitem a inclusão dinâmica de aplicações. Para estes, soluções que possibilitam a realização do mapeamento dinamicamente apresentam melhores resultados [51, 14]. A variação do perfil da aplicação durante sua execução pode acarretar em problemas como perdas dos *deadlines*, saturação dos canais na rede e até problemas térmicos. Dessa forma, mecanismos de reconfiguração dinâmica podem reduzir estes problemas e garantir, ou pelo menos melhorar, o tempo de resposta em sistemas desta natureza.

No contexto desse trabalho, é realizado um mapeamento de tarefas *iniciais* de maneira estática (em tempo de projeto). O sistema operacional que executa nos elementos de processamento provê serviços para mapear dinamicamente tarefas no sistema, através de primitivas para a criação e replicação de tarefas. Além disso, este monitora periodicamente as características das tarefas que estão em execução. Caso ocorram variações sensíveis em suas características, diferentes ações podem ser tomadas. Um mecanismo de gerência presente em cada elemento de processamento observa o perfil de execução local e em caso de necessidade procura migrar tarefas de acordo com os recursos disponíveis em nodos vizinhos. Antes de maiores detalhes, no entanto, serão apresentados os modelos propostos.

4.4 Modelo Proposto

4.4.1 Aplicação

O modelo de aplicação proposto é composto por um conjunto de tarefas que executam em elementos de processamento da arquitetura. Cada tarefa possui parâmetros individuais apresentados a seguir, além de seu código e dados que implementam sua funcionalidade. Uma tarefa é definida por uma n-upla $\tau_i =< id_i, uid_i, p_i, e_i, d_i, lc_i, pwr_i, cd_i, dt_i, ctx_i >$ sendo id_i sua identificação local, uid_i sua identificação única, p_i seu período; e_i o tempo de execução ou capacidade no período; d_i seu *deadline*; lc_i o volume de dados gerado pelo envio de mensagens; pwr_i o consumo de energia médio; cd_i o tamanho do segmento de código, dt_i o tamanho do segmento de dados e ctx_i o seu contexto.

Os parâmetros p_i, e_i e d_i são regidos por uma política de escalonamento de tempo real e descritos em uma unidade de tempo abstrata, denominada *tick*[1]. Algumas dessas políticas estão descritas em [40, 36, 39, 24, 3, 79]. Os parâmetros pwr_i, cd_i e dt_i são dependentes de tecnologia, compilador utilizado e algoritmo, e precisam ser posteriormente caracterizados. Cada tarefa τ_i é preemptiva e gera *jobs* a cada p_i *ticks*, sendo que estes *jobs* possuem um tempo de execução e_i que precisa ser completado antes do próximo período para que não ocorram perdas de *deadline*. O *overhead* em virtude de trocas de contexto e algoritmo de escalonamento são incluídos no tempo de execução da tarefa, afetando seu progresso de execução de maneira proporcional.

Cada tarefa, a qual executa de acordo com o modelo de tarefas, é atribuída a um único nodo, e cada nodo pode executar uma ou mais tarefas que compartilham tempo de processador de acordo com os parâmetros citados anteriormente. O conjunto de tarefas a ser executado nos nodos que compõem a arquitetura implementam a aplicação multiprocessada. Diversas aplicações podem executar em uma mesma arquitetura, compartilhando recursos. Como cada nodo possui um escalonador de tarefas local, são utilizadas duas identificações para uma mesma tarefa sendo id_i sua identificação local, que indica um índice em uma lista de tarefas e apenas é relevante no contexto do nodo em que determinada tarefa executa e uid_i sua identificação única ou global.

Tarefas podem não ter especificados seus parâmetros p_i, e_i e d_i, e neste caso são definidas como tarefas de melhor esforço. Este tipo de tarefa executa aperiodicamente, e somente quando houver tempo de processamento livre, ou seja, o tempo de processamento é inicialmente reservado às tarefas de tempo real. Tarefas de melhor esforço compartilham o tempo livre entre si, e são escalonadas com um algoritmo rotativo [74, 19], onde cada uma executa por um *tick*. Em virtude da reserva para as tarefas de tempo real, tarefas de melhor esforço podem ter postergada sua execução indefinidamente.

O acesso a recursos compartilhados (regiões críticas) são mutualmente exclusivos por natureza. Dessa forma, o acesso precisa ser serializado. Primitivas de sincronização como semáforos e *mutexes* devem ser utilizados para esse fim. Apesar das tarefas serem preemptivas no presente modelo, em virtude do compartilhamento de recursos tarefas podem ser bloqueadas por outras de menor prioridade (problema da *inversão de prioridade*). Este é um problema que deve ser resolvido em nível de aplicação ou com a utilização de mecanismos que fogem ao escopo do presente trabalho.

Definição. O modelo de tarefas proposto assume que: (i) as tarefas podem ser periódicas (tarefas de tempo real) ou de melhor esforço; (ii) o *deadline* relativo de uma tarefa de tempo real é igual ao seu período; (iii) as tarefas que compõem diferentes aplicações são independentes, ou seja, não possuem precedência; (iv) tarefas podem ser inicialmente mapeadas em um determinado elemento de processamento e posteriormente realocadas; (v) tarefas podem entrar e sair do sistema a qualquer momento.

[1] *Tick* é considerada a unidade mínima de escalonamento. Todos os parâmetros de tempo real são representados com *ticks*, e o próprio *tick* possui um valor de tempo determinado (1 ms por exemplo).

A comunicação entre tarefas é representada pela tupla $c_{ij} =< \tau_j, \omega_{ij} >$, onde τ_j é a tarefa alvo e ω_{ij} o volume de dados gerado especificamente para a tarefa alvo. Como a soma dos dados transmitidos por uma tarefa é composto por uma ou múltiplas comunicações (sendo esta uma lista de comunicações), o total de dados enviado pela tarefa geradora do tráfego é representado por lc_i, assumindo n como sendo o número de comunicações:

$$lc_i = \sum_{u=1}^{n} \omega_{ij}$$

O processo de realocação de tarefas a diferentes nodos em tempo de execução é referenciado como *migração* no contexto deste trabalho. Tarefas podem ser migradas de um nodo a outro desde que tenham sido configuradas como não fixas. Este tipo de informação é armazenada no contexto da tarefa.

4.4.2 Arquitetura

A arquitetura proposta é composta por elementos de processamento conectados através de filas a uma rede-intra-chip (NoC) com estrutura regular em uma malha de duas dimensões, roteamento determinístico XY e chaveamento de pacotes utilizando o método *wormhole*. Com o intuito de facilitar a validação e prototipação, além dos motivos anteriormente citados, será utilizada uma arquitetura homogênea composta por processadores simples[2]. O modelo, no entanto, é genérico o suficiente para incorporar parâmetros necessários para a representação de plataformas MPSoC heterogêneas. A modelagem de ambientes MPSoC heterogêneos, no entanto, foge ao escopo do presente trabalho.

As características de um nodo da arquitetura são representadas pela n-upla $\mu_m =< ct_m, f_m, l_m, nt_m, sp_m, o_m, ms_m, iq_m, oq_m, v_m >$ sendo ct_m o tipo de processador; f_m a frequência de operação em MHz; l_m a carga de processador resultante dos parâmetros das tarefas que executam no mesmo; nt_m o número de tarefas presentes; sp_m a política de escalonamento utilizada pelo sistema operacional; o_m o *overhead* (entre 0 e 1) do sistema operacional; ms_m o tamanho da memória de programa e dados (em *bytes*); iq_m o tamanho da fila de entrada da interface de rede, oq_m o tamanho da fila de saída da interface de rede (ambas as filas tem seu tamanho especificado em *flits*) e v_m o volume de dados enviado, onde $v_m = v_{m_1}...v_{m_l}$ é a soma de todos os volumes entre nodos m e n originados em m.

Para cada canal de comunicação[3] originado no nodo, seus parâmetros são representados pela tripla $v_{m_n} =< \mu_n, dv_{mn}, h_{mn} >$, sendo μ_n o processador alvo, dv_{mn} volume de dados transmitido e h_{mn} o número de saltos entre os nodos utilizando o roteamento XY. O volume de dados transmitido pelo nodo em um determinado canal de comunicação é constituido pela soma de lc_i de todas as tarefas de m que transmitem dados para um mesmo nodo n destino:

[2]Processadores que implementam um conjunto de instruções regular, não possuem *cache* nem mecanismos de gerência de memória em *hardware* como uma unidade MMU.

[3]Entende-se por canal de comunicação um caminho virtual estabelecido entre dois nodos, de forma que entre eles esteja ocorrendo troca de mensagens.

$$dv_{mn} = \sum_{u=1}^{r} lc_i$$

onde r é o último elemento de uma lista composta por tarefas de m que enviam dados para tarefas de n. Parâmetros como o nodo alvo e o número de saltos são caracterizados após a etapa de mapeamento. O volume total de dados em um enlace entre roteadores da rede é composto pela soma de todos os volumes de dados v_{m_n} que percorrem o enlace por estarem no caminho de comunicação, e este total é representado por tv_{kl}. Assim, a largura de banda bw_{kl} necessária em um enlace deve ser tal que $tv_{kl} \leq bw_{kl}$.

O mapeamento dos elementos de processamento a *tiles*[4] é feito de maneira estática no contexto deste trabalho, uma vez que os elementos de processamento são idênticos e o efeito do mapeamento e agrupamento de tarefas é realizado em nível de aplicação.

4.5 Representações em Alto Nível

Grande parte dos trabalhos na área utilizam grafos para descrever diferentes aspectos da aplicação e da arquitetura. Esta representação mostra-se dessa forma natural, e será utilizada para descrever o modelo proposto. As representações em alto nível apresentadas a seguir são descritas por grafos direcionados.

4.5.1 Aplicação - O Modelo ATM

No modelo de tarefas da aplicação, ou ATM uma aplicação é representada por um grafo $ATG = G(T, C)$ composto por um conjunto de tarefas. As tarefas são representadas por vértices τ_1 a τ_n e a comunicação entre as mesmas por arestas c_{ij}. Cada elemento de processamento pode executar múltiplas tarefas, o que permite que as mesmas se comuniquem tanto localmente como remotamente (realizando acesso ao meio de interconexão). Essas tarefas possuem parâmetros individuais, sendo esses identificação local e global, período, capacidade, *deadline*, lista de comunicações, consumo de energia e tamanho dos segmentos de código e dados e seu estado, descritos na Figura 4.2. Os parâmetros mandatórios descritos nos vértices são a identificação local e global, o período a capacidade e *deadline*. Cada tarefa é representada por um vértice e uma identificação da legenda é utilizada para descrever detalhadamenete seus parâmetros, incluindo a caracterização se necessário. Entre tarefas comunicantes existem arestas, onde é representada a direção da comunicação, bem como o volume de dados enviado pela tarefa de origem. O mapeamento inicial é apresentado pelo isolamento de tarefas em grupos, não sendo sua representação obrigatória. Esta representação também é utilizada para descrever o funcionamento dos protocolos de comunicação, mapeamento e migração propostos, omitindo-se detalhes como parâmetros das tarefas e volume de dados.

[4] Neste trabalho, *tile* é uma posição arbitrária na rede de interconexão onde é posicionado fisicamente um elemento de processamento.

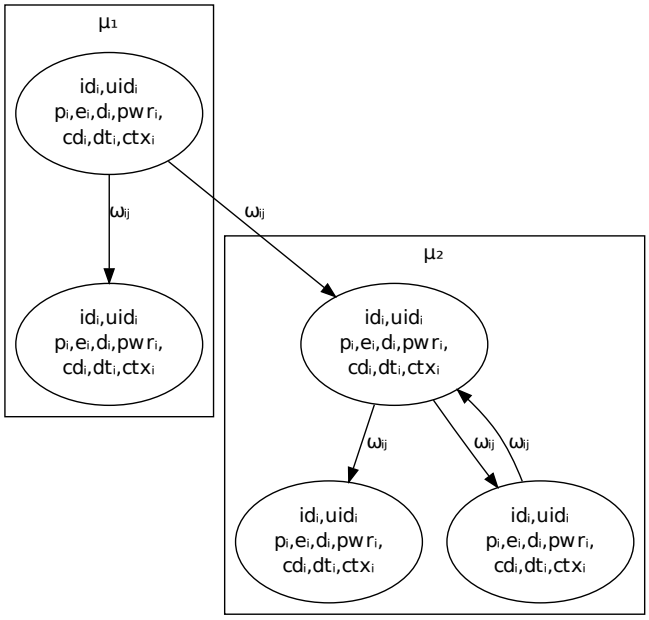


Figura 4.2 – Representação dos parâmetros do modelo ATM em um grafo ATG

O grafo de tarefas da aplicação, ou ATG é utilizado para representar o modelo, descrevendo tanto a aplicação como a comunicação existente entre tarefas, e parâmetros individuais de execução e caracterização de cada tarefa. Formalmente, o ATG é definido como:

Definição. Um grafo de tarefas da aplicação é um grafo dirigido representado pelo par $ATG = \langle T, C \rangle$, onde $T = \{\tau_1, \tau_2, ..., \tau_n\}$ é o conjunto de vértices e $C = \{c_{12}, c_{13}, ...c_{21}, c_{22}, ..., c_{2k}, c_{l1}, c_{l2}, ..., c_{ln-1}\}$ é o conjunto de arestas. O conjunto de vértices representa o processamento da aplicação e o conjunto de arestas representa as comunicações entre tarefas. Define-se id_i a identificação local, uid_i a identificação única, p_i o período da tarefa, e_i a capacidade da tarefa, onde $e_i \leq p_i$, d_i o *deadline*, onde $e_i \leq d_i$, pwr_i o consumo de energia, dt_i o tamanho do segmento de dados, cd_i o tamanho do segmento de código e ctx_i seu contexto. Sendo ξ_{ijq} a quantidade de dados da mensagem de índice $q(m_q)$ que são enviados da tarefa τ_i para a tarefa τ_j, e sendo k_{ij} o total de mensagens enviadas de τ_i para τ_j, então $\omega_{ij} = \sum_{q=1}^{k_{ij}} \xi_{ijq}$ é o volume de dados enviado de τ_i para τ_j. O volume de dados originado da tarefa i é $lc_i = \sum_{i=1}^{n} \omega_{ij}$, onde n é o número de comunicações da tarefa. Cada vértice $\tau_i \in T$ é uma tupla $\tau_i = \langle id_i, uid_i, p_i, e_i, d_i, lc_i, pwr_i, cd_i, dt_i, ctx_i \rangle$. Cada aresta $c_{ij} \in C$ é uma tripla $c_{ij} = \langle \tau_i, \tau_j, \omega_{ij} \rangle$, tal que $\tau_i \neq \tau_j$ e $\omega_{ij} > 0$.

A Figura 4.3 apresenta uma aplicação exemplo onde existem 8 mensagens trocadas entre 8 tarefas $T = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7, \tau_8\}$, com um volume de dados enviados entre tarefas sendo $\omega_{12} = 10$, $\omega_{28} = 7$, $\omega_{32} = 5$, $\omega_{34} = 10$, $\omega_{45} = 2$, $\omega_{46} = 2$, $\omega_{64} = 2$ e $\omega_{78} = 1$. Os parâmetros das tarefas são definidos como $\tau_i = \{id_i, uid_i, p_i, e_i, d_i, lc_i, pwr_i, cd_i, dt_i, ctx_i\}$, sendo $\tau_1 = \{t1, n1t1, 5, 1, 5, 10, 11, 123, 1000, ctx_i\}$, $\tau_2 = \{t2, n1t2, 12, 1, 12, 7, 11, 100, 1400, ctx_i\}$, $\tau_3 = \{t3, n1t3, 4, 1, 4, 15, 11, 250, 1200, ctx_i\}$, $\tau_4 = \{t1, n2t1, 5, 1, 5, 4, 11, 520, 1030, ctx_i\}$, $\tau_5 = \{t2, n2t2, 12, 1, 12, 0, 11, 50, 1750, ctx_i\}$, $\tau_6 = \{t3, n2t3, 4, 2, 4, 2, 11, 100, 1100, ctx_i\}$, $\tau_7 =$

$\{t1, n3t1, 5, 1, 5, 1, 11, 321, 2000, ctx_i\}$, $\tau_8 = \{t2, n3t2, 12, 1, 12, 0, 11, 140, 1500, ctx_i\}$. Importante observar que o parâmetro ctx_i não foi explicitado, e o parâmetro lc_i representa o total de volume de dados enviado pela tarefa em todas as suas comunicações.

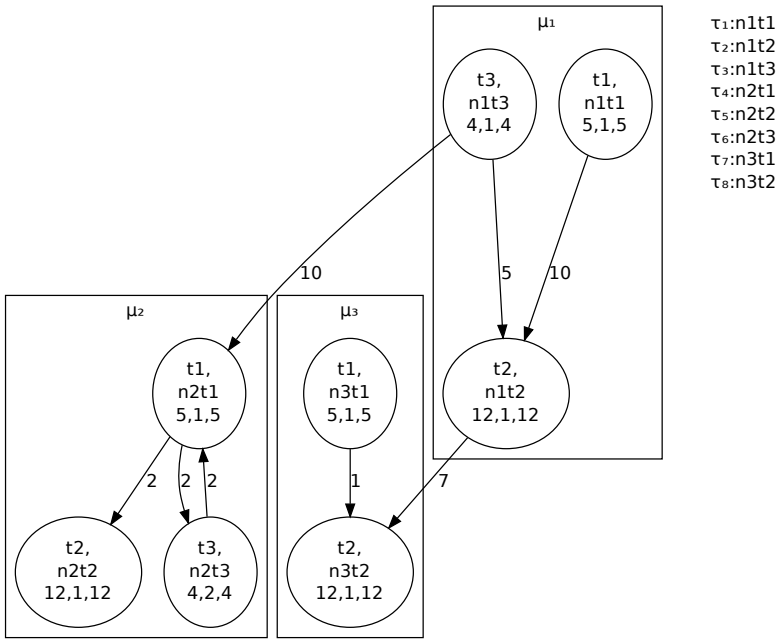


Figura 4.3 – ATG de uma aplicação MPSoC exemplo

4.5.2 Arquitetura - O Modelo ACCM

O modelo de comunicação e computação da arquitetura, ou ACCM é apresentado a seguir. Neste modelo, o grafo $ACCG = G(N, L)$ é utilizado para representar características da arquitetura de *hardware* e *software* e a topologia da rede de comunicação. O modelo apresenta as configurações dos nodos e sua posição, além do volume de dados que trafega entre os enlaces da rede.

Os nodos são representados no grafo ACCG por vértices μ_1 a μ_c e a comunicação por tv_{kl} formado pela soma do volume de dados v_m de todos os nodos que possuem seu tráfego direcionado pelo mesmo caminho. Nos vértices são representadas informações como carga, número de tarefas, política de escalonamento, tipo de processador empregado, frequência de operação, tamanho da memória e tamanho das filas entre elementos de processamento e roteadores e volume de dados total originado pelo nodo. Na Figura 4.4 são apresentados os parâmetros que caracterizam a configuração de nodos e a informação sobre o volume de dados.

O grafo de comunicação e computação da arquitetura, ou ACCG é utilizado para descrever o modelo, representando uma arquitetura como a comunicação existente entre nodos e parâmetros individuais de cada nodo. Formalmente, o ACCG é definido como:

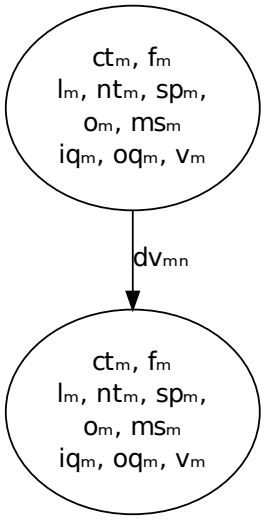


Figura 4.4 – Representação dos parâmetros do modelo ACCM em um grafo ACCG

Definição. Um grafo de comunicação e computação da arquitetura é um grafo dirigido representado pelo par $ACCG = \langle N, L \rangle$, onde $N = \{\mu_1, \mu_2, ..., \mu_c\}$ é o conjunto de vértices e $L = \{tv_{12}, tv_{13}, ...tv_{21}, tv_{22}, ..., tv_{2u}, tv_{k1}, tv_{k2}, ..., tv_{kl-1}\}$ é o conjunto de arestas. O conjunto de vértices representa os nodos da arquitetura e o conjunto de arestas representa os enlaces entre pares de nodo. Define-se ct_m o tipo de processador, f_m a frequência de operação, l_m a carga do nodo onde $0 \leq l_m$, nt_m o número de tarefas presentes no nodo onde $0 \leq nt_m$, sp_m a política de escalonamento empregada, o_m o *overhead* do sistema operacional onde $0 \leq o_m \leq 1$ e $o_m \in l_m$, ms_m o tamanho da memória de programa e dados, iq_m e oq_m o tamanho das filas e v_m o volume de dados enviado. Sendo lc_i o total de dados de uma tarefa que envia mensagens, e sendo r o total de tarefas que enviam mensagens de μ_m para μ_n, então $dv_{mn} = \sum_{u=1}^{r} lc_i$ é o total de dados enviados de μ_m para μ_n, onde $dv_{mn} \in v_m$. Cada vértice $\mu_m \in N$ é uma tupla $\mu_m = \langle ct_m, f_m, l_m, nt_m, sp_m, o_m, ms_m, iq_m, oq_m, v_m \rangle$. Cada aresta $tv_{kl} \in L$ é composta pela soma dos volumes de dados $v_{m_n} \in v_m$ que atravessam o enlace representado pela aresta.

A Figura 4.5 apresenta uma arquitetura exemplo composta por seis nodos dispostos em uma topologia malha 3x2. No exemplo, os volumes de dados que transitam nos enlaces são $tv_{12} = 200$, $tv_{21} = 50$ e $tv_{25} = 300$. A configuração dos nodos é $\mu_1 = \langle MIPS, 25, 0.5, 4, RM, 0.005, 32768, 64, 64, 200 \rangle$, $\mu_2 = \langle MIPS, 25, 0.6, 3, RM, 0.005, 32768, 64, 64, 200 \rangle$ e $\mu_5 = \langle MIPS, 25, 0.2, 4, RM, 0.005, 32768, 64, 64, 0 \rangle$. Apenas três dos seis nodos são utilizados. O volume total de dados no enlace tv_{25} é 300 e como $v_2 = 200$, supõem-se que 150 bytes do volume de dados enviado por μ_2 são direcionados ao nodo μ_5, e os outros 50 são enviados a μ_1. μ_1, por sua vez, envia 150 bytes a μ_5 e 50 bytes a μ_2.

Uma representação semelhante é apresentada por Murali [61, 62]. No presente trabalho, no entanto, outros aspectos da arquitetura são mostrados no grafo ao invés da largura de banda disponível entre os enlaces. Dessa forma, a semelhança entre as representações restringe-se ao formato da topologia.

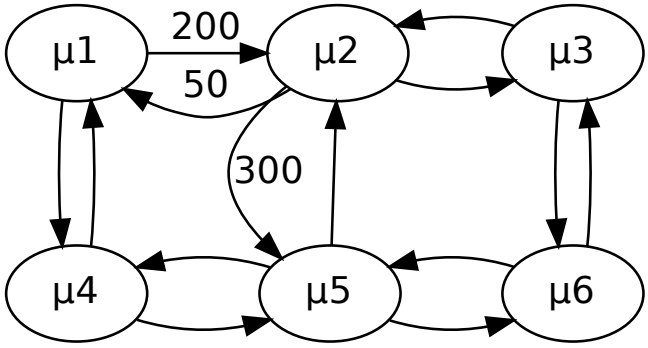


Figura 4.5 – ACCG de uma arquitetura MPSoC exemplo

4.6 Comunicação Entre Tarefas

O modelo proposto assume que o conjunto de tarefas que compõem a aplicação, assim como suas características, podem ser modificadas durante sua execução. Dessa forma, não é possível realizar a comunicação direta entre tarefas não fixas sem o conhecimento de sua localização. O processo de criação ou mapeamento de tarefas é gerenciado pelo sistema operacional, e novas tarefas podem entrar e deixar o sistema em tempo de execução, modificando o mapeamento inicial.

Para contornar este problema, assume-se uma tarefa de controle de comunicação em cada nodo, a qual possui uma identificação conhecida. Esta tarefa não pode ser removida nem migrada para outro nodo, pois é essencial para que o sistema de comunicação entre tarefas que migram entre nodos funcione adequadamente.

Inicialmente, uma tarefa que deseja se comunicar com outra utiliza a informação sobre a posição da tarefa destino baseando-se na informação conhecida sobre o mapeamento inicial, obtido em tempo de projeto. Esta informação é armazenada em uma referência local. Supõe-se que pelo menos uma das tarefas comunicantes seja não fixa neste caso. Antes de comunicar diretamente com a tarefa destino, no entanto, a tarefa de origem verifica se a tarefa destino encontra-se no mesmo processador. Neste caso, a comunicação é feita diretamente. Se o destino não for local, a tarefa envia uma solicitação à tarefa de controle do nodo remoto. Caso a tarefa destino esteja neste nodo, a tarefa de controle de comunicação responde com esta informação. Caso contrário, a tarefa controle responde informando para onde a outra foi migrada. Todas as tarefas que deixam um nodo são adicionadas a uma lista deste nodo, pesquisada a cada solicitação. Caso uma tarefa migrada anteriormente retorne ao nodo, esta é removida da lista. Com a informação sobre a nova posição da tarefa destino, a tarefa origem tenta se comunicar novamente, realizando uma pergunta à tarefa de controle do novo nodo. O processo se repete até que a tarefa de origem encontre a tarefa destino, e possa se comunicar diretamente com a mesma. A comunicação entre tarefas é gerenciada pelo sistema operacional. Tarefas comunicantes apenas realizam chamadas a primitivas *send()* e *receive()*, as quais tem sua complexidade abstraida em nível de aplicação. Maiores detalhes sobre as primitivas de comunicação serão apresentados no Capítulo 5.

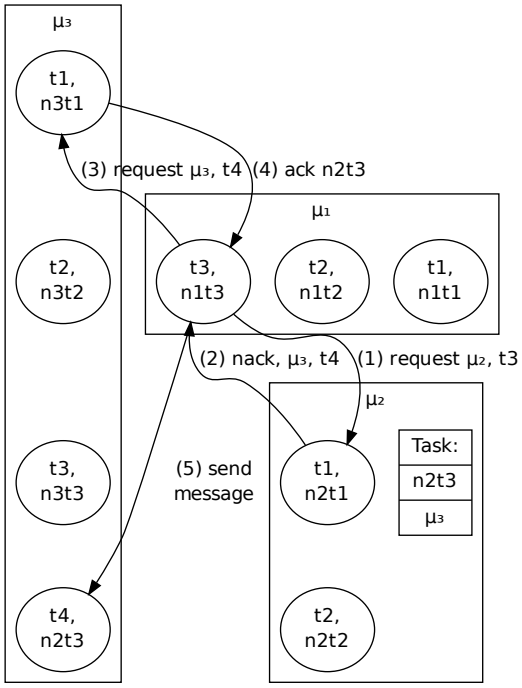


Figura 4.6 – Protocolo de comunicação entre tarefas

A Figura 4.6 apresenta o protocolo de comunicação. No exemplo, a tarefa $n1t3$ deseja comunicar com $n2t3$. Para isso, envia um *request* $\mu 2$*, t3* ao nodo $\mu 2$ (1). A tarefa de controle em $\mu 2$ verifica a lista de tarefas migradas, e responde *nack,* $\mu 3$*, t4* a $n1t3$ (2). A seguir, $n1t3$ solicita a comunicação com $n2t3$ à tarefa de controle de $\mu 3$ realizando um *request* $\mu 3$*, t4* (3). O controle verifica a lista de tarefas migradas, e como não encontra a tarefa solicitada responde a $n1t3$ com *ack, n2t3* (4). Ao receber a informação, $n1t3$ atualiza sua informação local sobre a tarefa remota e comunica-se diretamente com $n2t3$ (5). Caso venha a se comunicar com $n2t3$ novamente, a tarefa $n1t3$ realiza um *request* $\mu 3$*, t4* diretamente a $\mu 3$. A identificação local da tarefa é utilizada para que tarefas que executam em um mesmo nodo possam se comunicar diretamente, já a identificação global possibilita uma identificação única de qualquer tarefa do sistema, além de diferenciar tarefas migradas de outros nodos de tarefas originalmente mapeadas localmente.

Da mesma forma que tarefas migratórias são mantidas em uma lista, tarefas excluídas também são mantidas nesta, juntamente com a informação de exclusão. Caso ocorra uma solicitação à uma terefa excluída naquele nodo, a tarefa de controle informa a origem. Tarefas que iniciam a comunicação também podem ser migradas para outros nodos. Neste caso, o processo de comunicação acontece da mesma forma. Juntamente com a mensagem, é sempre enviada a informação sobre a origem da mesma, identificando a tarefa de origem no destino. Caso a tarefa que iniciou o processo de comunicação seja migrada, a tarefa destino automaticamente recebe esta informação na próxima comunicação.

Durante a execução do protocolo, ou seja, do início (solicitação da tarefa) até o envio propriamente dito, a tarefa de origem não pode ser migrada, permanecendo fixa no mesmo processador. Da mesma forma, ao receber uma requisição à uma tarefa local, a tarefa de controle não permite que a mesma seja migrada até que a comunicação tenha sido concluída. Detalhes sobre o

protocolo de migração serão apresentados na Seção 4.9. Caso as tarefas comunicantes sejam fixas, ou seja, foram configuradas para que não sejam migradas, a comunicação entre as mesmas é estabelecida diretamente, pois sua localização é conhecida já em tempo de projeto. Este mecanismo de comunicação direta é essencial para a implementação do protocolo de comunicação descrito nesta Seção.

A comunicação entre tarefas ocorre dentro do tempo de execução de cada tarefa, ou seja, assume-se que a comunicação possui tempo de processamento e faz parte da tarefa. Assim, uma tarefa pode ser preemptada a qualquer momento pelo sistema operacional durante sua comunicação e continuar o processo no seu próximo período. Tarefas de melhor esforço, as quais não possuem parâmetros de tempo real, executam sempre que houver tempo de processamento livre, e compartilham este tempo entre si.

4.7 Escalonamento de Tarefas

Na literatura são encontradas diversas propostas para a realização do escalonamento de tarefas em sistemas de tempo real. Entre os algoritmos mais utilizados estão o *First Come First Served* (FCFS) [79], *Round Robin* (RR) [74, 19], *Rate Monotonic* (RM)[40, 36], *Deadline Monotonic* (DM) [39], *Least Laxity First* (LLF) e *Earliest Deadline First* (EDF)[24, 3].

A política empregada no modelo proposto é o algoritmo *Rate Monotonic*, onde tarefas com períodos menores possuem prioridade sobre outras tarefas. Esta política, no entanto, não garante que todas as tarefas serão executadas obedecendo restrições temporais caso o conjunto não seja escalonável. Detalhes sobre o teste de escalonabilidade empregado são descritos na Seção 4.10. Outros detalhes sobre o mecanismo de escalonamento serão apresentados no Capítulo 5.

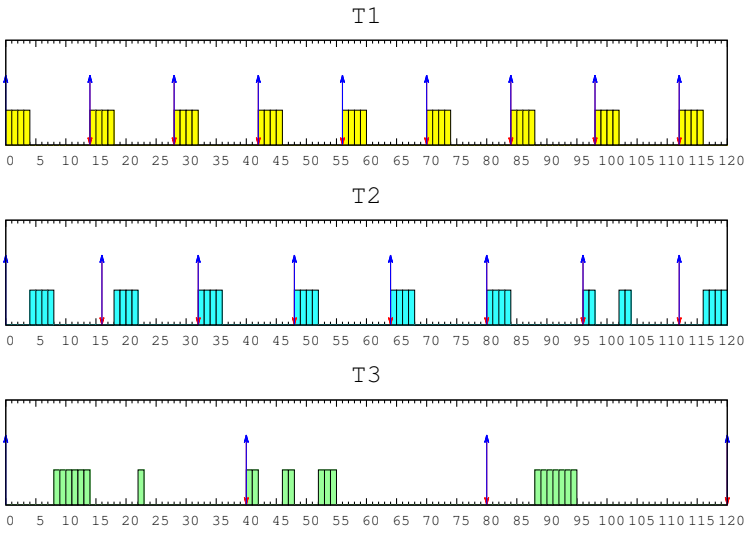


Figura 4.7 – Escalonamento de tarefas Rate Monotonic

Na Figura 4.7 é apresentado o escalonamento de três tarefas com os parâmetros p_i, e_i e d_i sendo respectivamente $\tau_1 = \langle 14, 4, 14 \rangle$, $\tau_2 = \langle 16, 4, 16 \rangle$ e $\tau_3 = \langle 40, 7, 40 \rangle$. As tarefas de maior

prioridade, no caso com menor período, sempre preemptam as de menor prioridade. A ocupação do processador para o exemplo apresentado é de 71% e de acordo com a política RM, este conjunto de três tarefas é escalonável.

4.8 Mapeamento Dinâmico

O mapeamento é uma função que define a posição das tarefas nos recursos da arquitetura. A qualidade do mapeamento influencia diretamente no desempenho da aplicação, pois congestionamentos e sobrecargas advindos de um mapeamento ruim comprometem a execução de suas tarefas, subutilizando recursos.

4.8.1 Mapeamento Estático Inicial

O mapeamento inicial das tarefas da aplicação é definido em tempo de projeto. Este mapeamento pode ser feito manualmente, ou automatizado com o intuito de maximizar a função de custo do mapeamento, onde o custo é definido como consumo de energia, tempo de processamento, ocupação dos canais na rede de comunicação, número de saltos entre nodos, entre outros. Neste trabalho, as tarefas iniciais são atribuídas a nodos manualmente, sendo necessário o conhecimento prévio do grafo que descreve a aplicação em seu estado inicial.

Segundo [11], o tempo de mapeamento estático não é crítico quando comparado ao tempo de mapemento realizado em tempo de execução. Assim, o tempo de mapeamento em sistemas dinâmicos deve ser considerado, uma vez que impacta diretamente no desempenho da aplicação. O problema do mapeamento é NP-completo e mesmo em tempo de projeto pode ser proibitivo, o que motiva utilização de heurísticas com o intuito de reduzir este tempo.

Nos sistemas que possuem um conjunto fixo de tarefas, técnicas de mapeamento estático são utilizadas com o intuito de otimizar o custo do mapeamento. Este mapeamento inicial é realizado sobre todas as tarefas do sistema, ou seja, as tarefas e os seus parâmetros são definidos em tempo de projeto. No modelo proposto, novas tarefas podem ser adicionadas, destruídas, replicadas ou terem seus parâmetros modificados em tempo de execução. Assim, técnicas de mapeamento estático tendem a ser pouco eficientes do ponto de vista de custo do mapeamento para aplicações dinâmicas, uma vez que este custo varia de acordo com o perfil da aplicação. Outro aspecto é o tempo de execução de heurísticas comumente utilizadas para a realização do mapeamento estático, que em cenários dinâmicos tornam-se impraticáveis.

O emprego de heurísticas estáticas para o mapeamento inicial não é invalidada pelo comportamento dinâmico da aplicação, pois no modelo proposto parte das tarefas podem ser confinadas a posições fixas. Além disso, não existe um conhecimento prévio sobre o perfil de execução da aplicação, e a melhor escolha é sem dúvidas tomar a melhor decisão possível sobre o mapeamento já em tempo de projeto, e otimizar a aplicação dinamicamente de acordo com as variações no perfil da

aplicação. Vale ressaltar que além de tarefas configuradas como fixas, as tarefas iniciais da aplicação necessariamente precisam ser mapeadas estaticamente.

4.8.2 Repositório Local de Tarefas

O trabalho aqui apresentado propõe além de instâncias distribuídas de um sistema operacional em cada nodo, a utilização de repositórios locais de tarefas. A idéia é, em tempo de projeto definir grupos de tarefas (particionamento) e associá-los a repositórios. Cada instância do sistema operacional é responsável pela gerência de seu próprio repositório, descentralizando o controle e permitindo dessa forma uma gerência de maneira distribuída.

O modelo proposto difere de trabalhos como [11, 44], os quais utilizam um único repositório em uma memória centralizada e um mecanismo de paginação que faz a utilização de código não relocável. O modelo proposto neste trabalho, no entanto, não possui a limitação imposta pelo tamanho de páginas ou código, o que permite uma melhor utilização da memória uma vez que esta é alocada em segmentos e dinamicamente. Uma única região de memória dividida em segmentos existe em cada nodo, e dessa forma a fragmentação interna é minimizada.

4.8.3 Mapeamento de Tarefas e Variação de Carga

Neste trabalho, o mapeamento dinâmico das tarefas é feito pela criação ou replicação de tarefas em um nodo da arquitetura. Ao ser mapeada (o que pode ser feito a partir de uma tarefa já mepada) uma tarefa é configurada com os parâmetros descritos na Seção 4.4 e mapeada localmente. Diferentemente de outros trabalhos propostos na literatura [78, 83, 11, 44], os quais utilizam diversas heurísticas para mapear tarefas dinamicamente, o presente trabalho aproxima o modelo de tarefas e seu mapeamento a conceitos de sistemas operacionais clássicos. O mecanismo de migração é utilizado para modificar esse mapeamento de tarefas em tempo de execução, obtendo uma funcionalidade semelhante ao mapeamento dinâmico empregado em trabalhos relacionados. Outros trabalhos como [64, 10, 59, 81, 46, 9] utilizam o conceito de migração de tarefas como proposto neste trabalho.

O evento de criação, replicação ou modificação de tarefas altera o cenário inicial sobre a utilização do sistema, usado como referência para a realização do mapeamento inicial. Algumas vezes, um nodo em questão não possui recursos disponíveis para mapear e executar uma tarefa e é criada uma situação de sobrecarga. O efeito da sobrecarga, considerando o modelo proposto, torna o conjunto de tarefas não escalonável e faz com que a aplicação perca seu critério de tempo real. Outra situação é a saturação e contenção dos canais de comunicação da rede originado de fluxos de dados de novas tarefas. Ainda, uma nova tarefa comunicante pode ser mapeada a uma grande distância em saltos na rede, o que faz com que seu fluxo de dados percorra muitos enlaces e ocupe recursos de maneira não otimizada.

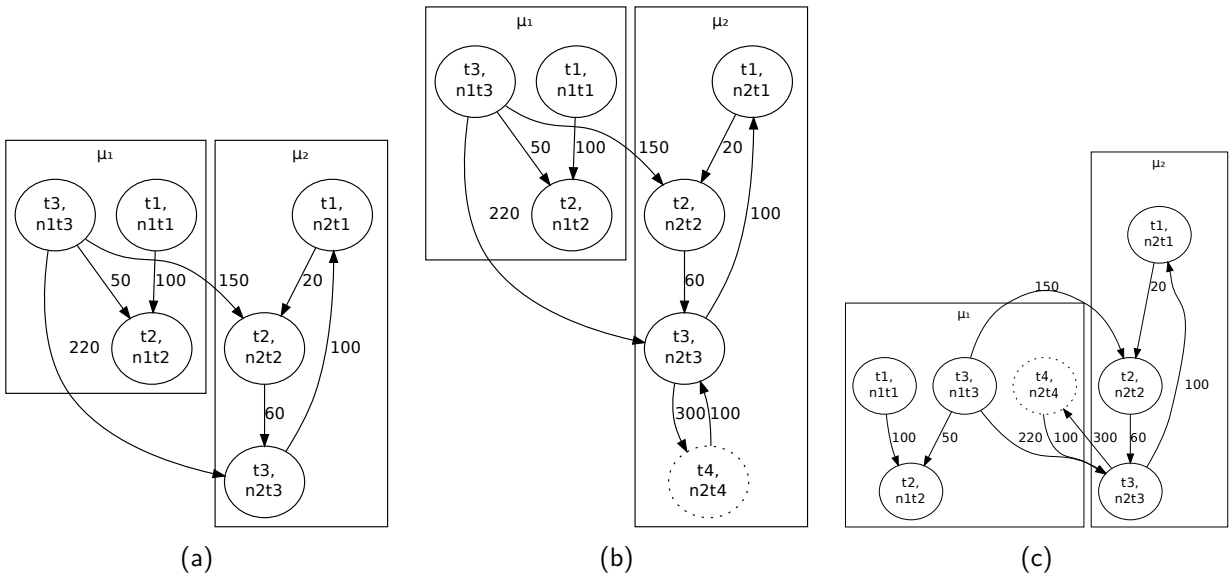


Figura 4.8: Mapeamento dinâmico. Em (a) o sistema encontra-se em seu mapeamento inicial. A tarefa *n2t3* cria uma nova tarefa (b), mapeando-a localmente em $\mu 2$. A tarefa mapeada *n2t4* é migrada para $\mu 1$ (c).

No exemplo apresentado na Figura 4.8, dois nodos $\mu 1$ e $\mu 2$ possuem três tarefas em seu mapeamento inicial. Os parâmetros especificados das tarefas são id_i, uid_i, p_i, e_i e d_i. O conjunto de tarefas é $\tau_1 = \{t1, n1t1, 5, 1, 5\}$, $\tau_2 = \{t2, n1t2, 11, 2, 11\}$, $\tau_3 = \{t3, n1t3, 5, 1, 5\}$, $\tau_4 = \{t1, n2t1, 4, 1, 4\}$, $\tau_5 = \{t2, n2t2, 3, 1, 3\}$ e $\tau_6 = \{t3, n2t3, 5, 1, 5\}$. A carga dos nodos é 58%[5] e 78% respectivamente nesta configuração (Figura 4.8(a)). Os volumes de dados são $\omega_{12} = 100$, $\omega_{32} = 50$, $\omega_{35} = 150$, $\omega_{36} = 220$, $\omega_{45} = 20$, $\omega_{56} = 60$ e $\omega_{64} = 100$. Em um certo momento (Figura 4.8(b)), a tarefa *n2t3* cria uma nova tarefa $\tau_7 = \{t4, n2t4, 5, 1, 5\}$, que possui o volume de dados $\omega_{76} = 100$ e gera outro volume $\omega_{67} = 300$ de *n2t3*. A carga do nodo $\mu 2$ sobe para 98% após o mapeamento da tarefa, deixando-o em uma situação de sobrecarga. O último passo (Figura 4.8(c)) consiste em migrar a tarefa *n2t4* para o nodo $\mu 1$, distribuindo o processamento. A carga dos nodos nesta configuração é de 78%.

São três as possíveis situações que incidem em uma situação de sobrecarga: (i) uma tarefa de tempo real já mapeada tem seus parâmetros alterados, aumentando sua prioridade e consequentemente seu uso de processador ou seu uso da rede; (ii) uma nova tarefa é mapeada, aumentando a carga de processamento ou uso da rede além do limite imposto e; (iii) uma tarefa já mapeada é replicada e é ultrapassado o limite de utilização da rede ou processamento.

4.9 Migração de Tarefas

O processo de migração de tarefas ocorre em diversas etapas, e pode ser realizado por um gerente de migração ou por uma tarefa qualquer local, desde que não seja ela mesma (uma tarefa

[5] O cálculo da carga (ou fator de utilização) é apresentado da Seção 4.10.

não pode migrar a si). A migração da tarefa acontece inicialmente em paralelo com sua execução, uma vez que a tarefa apenas é bloqueada pelo mecanismo de migração após o envio do seu código.

O controle de recursos de cada nodo é mantido por gerentes de migração distribuídos e é detalhado na Seção 4.10. Dessa forma, o processo de migração por si só não realiza nenhum tipo de controle de recursos como utilização de processador e ocupação dos canais, e apenas falha caso o nodo destino não consiga mapear a tarefa migrada por não possuir memória livre ou a tarefa possuir comunicações pendentes não resolvidas dentro de um tempo pré-determinado. O protocolo de troca de mensagens automaticamente resolve as dependências entre tarefas comunicantes após migrações, pois garante que exista uma referência local atualizada sobre determinada tarefa antes que uma mensagem propriamente dita seja enviada.

Tarefas comunicantes podem possuir um atraso no processo de migração até que comunicações pendentes sejam resolvidas. Neste caso, o mecanismo de migração libera a tarefa a ser migrada para que esta conclua comunicações pendentes e não permite que novas comunicações sejam realizadas, postergando respostas a solicitações para trocas de mensagem para esta tarefa.

A migração de tarefas consiste nos seguintes passos, apresentados na Figura 4.9, onde os passos 1 a 4 são descritos em 4.9(a), os passos 5 a 9 em 4.9(b) e o passo 10 em 4.9(c): (1) o código da tarefa é enviado ao nodo destino; (2) a tarefa que está sendo migrada é bloqueada; (3) os dados da pilha são enviados; (4) o contexto da tarefa é enviado; (5) o código é relocado no nodo destino; (6) é realizado o mapeamento no nodo destino; (7) a tarefa tem seu contexto restaurado; (8) o nodo destino responde sobre o estado da migração, juntamente com a identificação local da nova tarefa; (9) a tarefa é excluída no nodo origem; (10) a identificação da tarefa no nodo destino é adicionada a uma lista de tarefas migradas local.

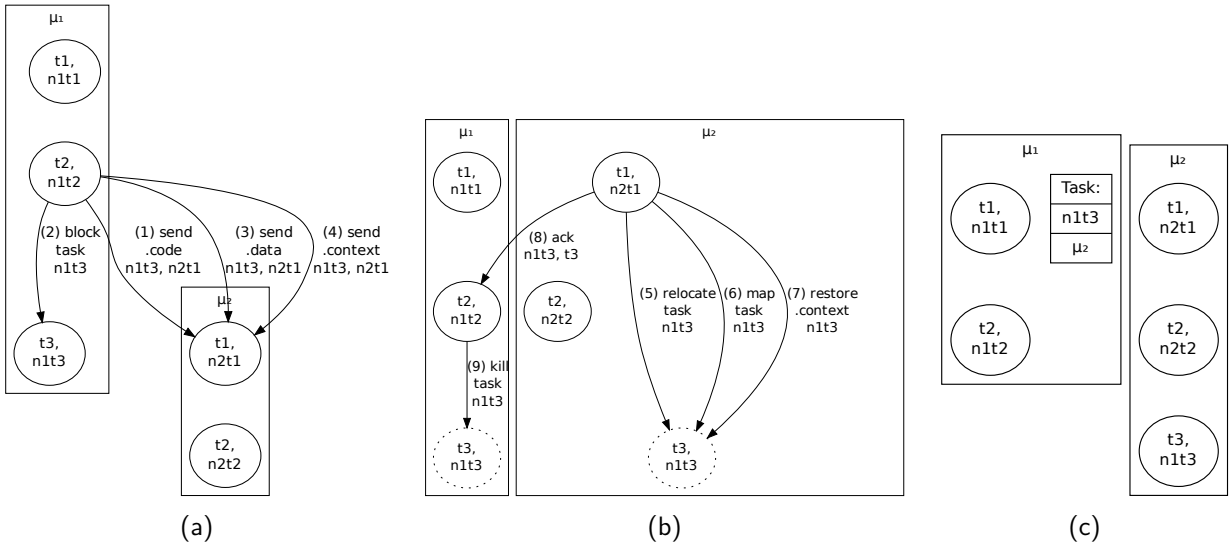


Figura 4.9: Migração de tarefas. Em (a) a tarefa a ser migrada *n1t3* tem seu código enviado ao nodo destino $\mu 2$ e é bloqueada. A tarefa *n2t1* reloca e mapeia a tarefa *n1t3*, avisando a origem sobre o estado, que exclui a tarefa (b). Uma entrada é adicionada a lista de tarefas migradas em $\mu 1$ (c)

A tarefa de controle de comunicações é responsável por receber, mapear e restaurar o contexto da tarefa migrada, além de notificar a origem sobre o estado da operação. Esta tarefa também é responsável por manter atualizada a lista de tarefas migradas, atender a solicitações de mensagem e resolver conflitos no caso de comunicações pendentes. Como citado anteriormente, esta tarefa possui uma posição fixa, e não pode ser migrada.

São realizadas quatro trocas de mensagem entre nodos para a execução do protocolo, e outras mensagens internas a cada nodo são feitas por memória compartilhada e encapsuladas pelo sistema operacional. As trocas de mensagem compreendem o envio do código, dados e contexto, e recebimento do estado da migração (sucesso ou falha) envidado pelo destino. O tempo para a realização do processo não é desprezível, uma vez que a tarefa a ser migrada fica bloqueada durante a execução do protocolo. Este tempo de migração varia de acordo com a tarefa, a quantidade de dados armazenado na pilha da mesma e seu número de comunicações. No nodo destino o processo de migração ocorre de acordo com os parâmetros da tarefa de controle de comunicações, executando no seu tempo e não interferindo no atual conjunto de tarefas que executa no nodo. No nodo origem, o protocolo executa dentro do tempo da tarefa que invoca a migração, e caso esta tarefa seja o gerente, o restante das tarefas continua executando normalmente.

4.10 Gerência de Migração

A gerência de migração é utilizada para controlar a forma como o sistema deve proceder com relação a modificações em seu perfil de execução. O mecanismo de gerência é responsável por avaliar o estado do sistema e realizar decisões de migração. Algumas das funções de custo utilizadas em trabalhos da literatura são: utilização dos elementos de processamento; ocupação dos canais da rede; número de saltos; consumo de energia; entre outros.

Neste trabalho propõe-se a utilização de uma abordagem distribuída, isto é, cada nodo possui um gerente de migração local. Este gerente local possui os mesmos atribuitos de uma tarefa de tempo real e comunica-se com outros gerentes diretamente.

A Figura 4.10 apresenta um exemplo que emprega gerentes distribuídos. No exemplo, uma malha 3x2 é utilizada, onde a tarefa de controle de comunicação (T1), a gerência de migração local (T2) e as tarefas de usuário executam no mesmo ambiente. Ainda, ambas compartilham os serviços fornecidos pela camada inferior, composta pelo sistema operacional, elementos de processamento e rede de interconexão.

4.10.1 Definições

São definidos dois estados para cada nodo da arquitetura, onde o mesmo pode encontrar-se em situação normal, ou sobrecarregado. A definição sobre qual estado o nodo encontra-se é obtida pela taxa de utilização de processador u_i, pelo limite de utilização de processador ul_i, pelo

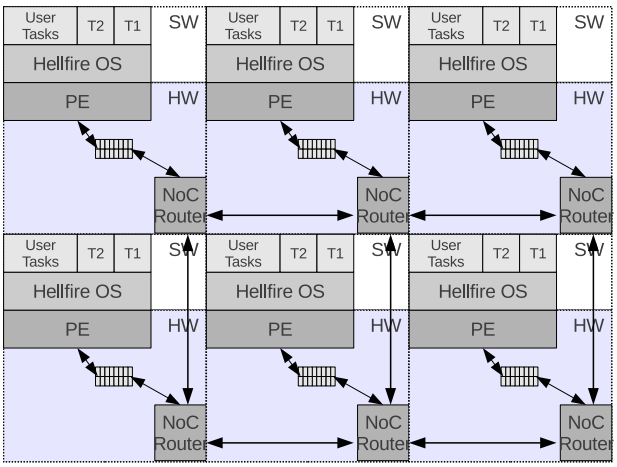


Figura 4.10 – Gerentes de migração distribuídos

fator de carga de processador l_i, pelo limite de carga de processador ll_i, pelo volume de dados tv_i e pela largura de banda disponível no enlace fbw_i. Os valores limite são dependentes da política de escalonamento utilizada no nodo e da vazão provida pelo roteador. Sendo η o número de tarefas presentes no nodo e j o índice de determinada tarefa, representa-se respectivamente a taxa de utilização e o fator de carga do nodo:

$$u_i = \sum_{j=1}^{\eta} \frac{e_j}{p_j}$$
$$l_i = \sum_{j=1}^{\eta} \frac{e_j}{d_j}$$

Definição. Um determinado nodo é considerado sobrecarregado quando $u_i > ul_i$ ou $tv_i > fbw_i$.

No modelo de tarefas proposto, é assumido que os parâmetros p_i e d_i da política de escalonamento são iguais e dessa forma $u_i = l_i$. O fator de utilização, no entanto, pode ser avaliado independentemente da política de escalonamento como indicador do estado (normal ou sobrecarregado) do nodo. Cada política de escalonamento possui um limite de utilização específico, e esse limite pode variar ainda de acordo com o número de tarefas presentes no sistema. Para o algoritmo RM[40, 36, 39], o qual possui prioridade fixa, esse limite é definido por:

$$\sum_{j=1}^{\eta} \frac{e_j}{p_j} \leq \eta(2^{1/\eta} - 1)$$

O processo de gerência e migração possui um custo que está relacionado às decisões de migração (algoritmo de gerência), transferência do código, parada da tarefa a ser migrada, transferência de seus dados, reinicio da tarefa no nodo destino e resposta ao nodo origem. O custo da gerência de migração é representado por ψ:

$$\psi = \rho + \sigma + \upsilon$$

Onde ρ representa o tempo para a execução do algoritmo de gerência no nodo origem e trocas de mensagem com nodos vizinhos (candidatos), σ o tempo para a transferência dos segmentos de dados, código e contexto da tarefa e υ o tempo para o reinício da tarefa e resposta à origem. O custo do processo deve ser avaliado como sendo parte do tempo de execução da tarefa a ser migrada de forma que $e_i + \psi_i < p_i$ para que os *deadlines* da tarefa sejam respeitados. Além disso, torna-se necessária a avaliação da taxa de utilização no nodo destino tal que $u_k < ul_k$ para que todos os *deadlines* sejam respeitados, onde:

$$u_k = \sum_{j=1}^{\eta_k} \frac{e_j}{p_j} + \frac{e_i+\psi}{p_i}$$

Além dos parâmetros citados, deve ser observada a quantidade de memória disponível no nodo destino para que este possa receber a tarefa a ser migrada e que a largura de banda disponível comporte a tarefa tal que $v_i < fbw_k$. A quantidade de memória disponível é especificada por fm_k, e é definida por:

$$fm_k = m_k - (\sum_{i=1}^{\eta}(cd_i + dt_i) + cd_{os} + dt_{os})$$

Ou seja, a memória disponível fm_k é obtida a partir da subtração entre o valor do tamanho de memória total e a soma de todos os segmentos de dados e código de todas as tarefas presentes no elemento de processamento em conjunto com o tamanho dos segmentos do sistema operacional.

Caso a tarefa possua uma quantidade de dados e código maior que a memória disponível no elemento de processamento destino, a migração não poderá ocorrer. Sendo i a tarefa a ser migrada e k o processador destino, o teste resume-se a:

$$cd_i + dt_i \leq fm_k$$

A largura de banda disponível no roteador destino pode ser apenas aproximada, uma vez que é difícil realizar o cálculo de todos os fluxos de dados que trafegam pelo roteador em questão. Para avaliar esta condição, faz-se necessário calcular a largura de banda disponível no nodo destino:

$$fbw_k = bw_k - tv_k$$

A largura de banda disponível deve ser tal que $fbw_k \gg v_i$, para que a tarefa possa migrar para o nodo destino sem prejudicar a comunicação das tarefas remotas do nodo e de outros que possuam comunicações que utilizam enlaces vizinhos e fazem parte do caminho de comunicação de v_i a partir do nodo destino.

4.10.2 Gerentes Distribuídos

Conforme citado anteriormente, cada nodo possui um gerente de migração local no modelo proposto. A solução proposta tem como objetivo melhorar o estado global do sistema de maneira

progressiva, sem ter no entanto um custo elevado do ponto de vista computacional (solução *greedy incremental*).

Algumas das decisões realizadas pelo gerente, como a escolha da tarefa a ser migrada e o nodo alvo da migração podem ser realizadas de acordo com diversas heurísticas. A escolha da tarefa a migrar pode levar em consideração parâmetros como utilização do processador pela tarefa, número de comunicações, volume de dados, número de saltos ou simplesmente uma escolha randômica.

Outras decisões são a busca por candidatos de migração e a escolha do nodo alvo. Entre as heurísticas utilizadas para realizar a seleção estão os algoritmos *Nearest Neighbor* (NN), *First Free* (FF) e heurísticas que avaliam o volume de dados e contenções na rede [11] como o *Minimum Maximum Channel Load* (MMCL), *Minimum Average Channel Load* (MACL), *Path Load* (PL) e *Best Neighbor* (BN). As funções de custo podem levar em consideração diferentes fatores como a carga de processamento livre, memória disponível, proximidade em saltos na rede, ocupação dos canais e alcançabilidade.

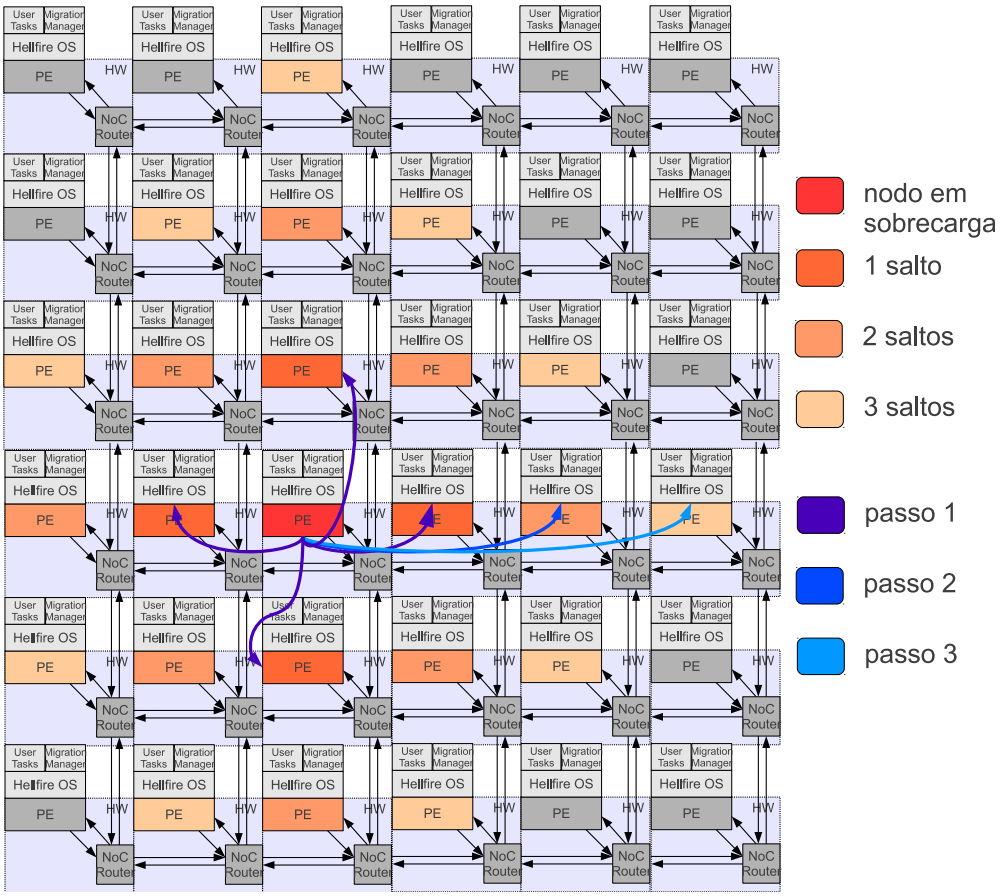


Figura 4.11 – Heurística para a seleção de alvos para migração de tarefas

A Figura 4.11 apresenta um exemplo, onde um nodo sobrecarregado utiliza um método semelhante as heurísticas NN e BN para realizar a busca por candidatos. O método proposto utiliza a busca por vizinhos por espalhamento. Neste método, são enviadas mensagens para todos os vizinhos a um mesmo número de saltos de uma só vez, ao contrário de outras heurísticas propostas. O número de vizinhos cresce a medida que o número de saltos aumenta, desde que os limites da

rede não tenham sido atingidos, quando mais uma vez o seu número diminui progressivamente. No exemplo apresentado, que consiste de uma malha com dimensões 6x6, um elemento de processamento encontra-se em sobrecarga. O primeiro passo do algoritmo consiste em realizar a busca por recursos nos 4 vizinhos que encontram-se a um salto de distância. Não havendo recurso disponível, é realizada uma busca em 8 vizinhos que estão a dois saltos sendo este o segundo passo. No terceiro passo, é realizada uma busca em 10 vizinhos, e os limites da rede começam a ser atingidos a partir deste ponto.

A diferença desta estratégia comparada as estratégias NN e BN é que todos os vizinhos que estiverem a uma mesma distância em saltos são consultados de uma única vez e com suas respostas, o gerente local decide qual o melhor candidato levando em consideração não apenas a ocupação dos canais (pelo número de saltos), mas também a carga de processamento e a memória disponível. Outro ponto é que a heurística é executada em qualquer elemento de processamento sobrecarregado, e não a partir de um mestre. Além disso, o algoritmo é executado para que posteriormente seja realizada uma migração de tarefa para outro nodo, ao invés de mapear uma nova tarefa por medida de necessidade de executar tal tarefa, como é o caso de outros trabalhos [52, 51, 83, 11, 44] os quais utilizam a heurística para a realização do mapeamento dinâmico de tarefas. As solicitações realizadas aos nodos candidatos são feitas diretamente a seus gerentes locais. Estes respondem com informações como carga do nodo, número de tarefas, política de escalonamento, tráfego gerado e memória livre. Com base nas informações de todos os candidados desta distância (um salto, por exemplo), o gerente de migração decide pela melhor opção entre todos os recursos disponíveis no passo corrente do algoritmo. Caso não exista recurso disponível, novas solicitações são realizadas aos próximos candidados, até os limites da rede. Caso não seja encontrado recurso disponível, o nodo permanece em sobrecarga e não apto a realizar nova busca por um tempo não determinado neste momento.

Todas as funções que realizam escolhas levam em consideração os limites impostos pelas definições anteriormente apresentadas. A única excessão refere-se ao custo da migração ψ. O custo de migração define o tempo no qual a tarefa a ser migrada não irá executar. Este custo pode ser amortizado a longo prazo após a migração para tarefas que estão perdendo *deadlines*, uma vez que manter estas tarefas executando em nodos sobrecarregados faz com que estas deixem de executar de acordo com seus parâmetros de tempo real.

4.11 Considerações Finais

Este Capítulo apresentou os modelos de aplicação, arquitetura e protocolos para comunicação, mapeamento e migração de tarefas no ambiente em questão. Estes modelos e protocolos foram utilizados como base para a implementação do sistema operacional Hellfire OS, apresentado no Capítulo 5.

5. IMPLEMENTAÇÃO DO MODELO

A seguir serão apresentadas as implementações dos modelos de tarefa, gerenciamento de aplicação e arquitetura propostos no Capítulo 4. Estas implementações foram utilizadas para validar o modelo proposto com a obtenção de resultados.

5.1 Características Gerais

Neste trabalho foi desenvolvido um sistema operacional de tempo real (HellFire OS[1]) portável e totalmente preemptivo, baseado em uma arquitetura *kernel* monolítico, porém modular. Este sistema operacional implementa o modelo de tarefas descrito no Capítulo 4, e atualmente possui algumas ferramentas para o desenvolvimento e simulação de aplicações embarcadas de tempo real. O sistema operacional pode ser configurado de acordo com a aplicação a ser executada, e parâmetros como o número máximo de tarefas no sistema, tamanho de pilha das tarefas, tamanho da memória *heap* (pode ser alocada dinamicamente), política de escalonamento, opções para *debug*, *logging* e migração de tarefas podem ser customizados. Essa customização permite que o tamanho da imagem binária final[1] do sistema operacional seja otimizada, tornando possível a execução do sistema em arquiteturas com tamanho de memória reduzido. Algumas das funcionalidades disponibilizadas ao desenvolvedor incluem:

- Sistema operacional preemptivo (tarefas *podem* opcionalmente cooperar);
- Gerenciamento dinâmico de tarefas (adicionar, remover, bloquear, resumir, alterar parâmetros, *fork()*);
- Chamadas de sistema (informações sobre *deadlines*, uso de processador, memória, energia, parâmetros de tarefas, tempos de trocas de contexto);
- Diferentes políticas de escalonamento para tarefas com prioridade fixa e dinâmica;
- Exclusão mútua e semáforos;
- Mailboxes;
- Alocação, liberação e gerência dinâmica de memória;
- Verificações de integridade do sistema de forma automática;
- LibC customizada;

[1] A imagem binária final do sistema é composta pelo sistema operacional e repositório local de tarefas que executam no mesmo. Esta imagem é carregada na memória de um nodo, permitindo que após a inicialização o sistema operacional execute as tarefas.

- Biblioteca para emulação de ponto flutuante com precisão simples (com funcionalidades adicionais como conversões, cálculos de raiz quadrada e funções trigonométricas);
- Comunicação entre tarefas por trocas de mensagem ou memória compartilhada (primitivas bloqueantes e não bloqueantes);
- Migração de tarefas;
- Gerência de migração;

Periféricos são acessados através de entrada e saída mapeada em memória ou por portas de entrada e saída. O mapa de periféricos pode ser configurado na *camada de abstração de hardware* (HAL) para uma solução específica de *hardware*, o que facilita a portabilidade do sistema operacional para outras arquiteturas. Atualmente, existem portes para as arquiteturas MIPS (multiprocessador), x86 e ARM (monoprocessador).

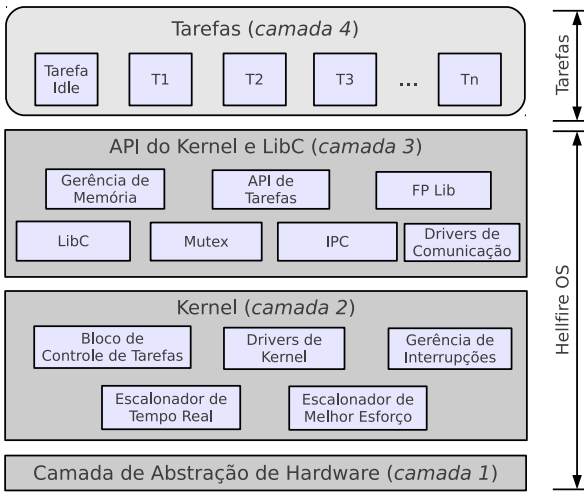


Figura 5.1 – Estrutura em camadas do sistema operacional Hellfire OS

A Figura 5.1 apresenta a estrutura do sistema operacional. Todas as funções dependentes de arquitetura são implementadas na HAL (camada 1). O *kernel* do sistema é implementado sobre esta camada (camada 2). Alguns *device drivers* de baixo nível são implementados nesta camada, onde possuem acesso privilegiado a estruturas internas do sistema e ao *hardware*. Uma biblioteca reduzida de funções padrão da linguagem C (LibC), assim como a API do sistema operacional são implementadas sobre o *kernel* (camada 3). Tanto as tarefas quanto o sistema operacional compartilham a biblioteca padrão, o que permite redução na utilização de memória. As tarefas de usuário são implementadas na camada 4, e utilizam da API disponibilizada. Nesta camada também são implementados os *device drivers* que executam em nível de usuário (como o controle de comunicação e gerência de migração), que possuem os mesmos parâmetros de tarefas de usuário, ou seja, são regidos pela mesma política de escalonamento.

Rotinas de tratamento de interrupção, salvamento e recuperação de contexto são dependentes de arquitetura e dessa forma foram escritas em linguagem de máquina. Essas rotinas fazem

parte da camada de abstração de *hardware*. Apenas uma parte desta camada é descrita em linguagem de máquina, sendo o restante de todo o *software* descrito em linguagem C. É importante salientar que esta camada pode ser facilmente portada para outras arquiteturas, devido a modularidade do sistema operacional. Os endereços dos periféricos acessíveis por *software* são ilustrados da Figura 5.2. A definição destes endereços faz parte da HAL específica para a implementação dos nodos utilizados na arquitetura, que consistem em processadores Plasma [76] modificados. Detalhes sobre a organização interna dos nodos serão apresentados na Seção 5.8.

```
/*********** hardware memory addesses ***********/
#define MISC_BASE             0x20000000
#define UART_WRITE            0x20000000
#define UART_READ             0x20000000
#define IRQ_MASK              0x20000010
#define IRQ_STATUS            0x20000020
#define GPIO0_OUT             0x20000030
#define GPIO1_OUT             0x20000040
#define GPIOA_IN              0x20000050
#define COUNTER_REG           0x20000060
#define NOC_READ              0x20000070     /*READ*/
#define NOC_WRITE             0x20000080     /*WRITE*/
#define NOC_STATUS            0x20000090     /*STATUS*/
#define FREQUENCY_REG         0x200000A0
#define TICK_TIME_REG         0x200000B0
#define ENERGY_MONITOR        0x200000C0
#define OUT_FACILITY          0x200000D0     /* simulator only */
#define LOG_FACILITY          0x200000E0     /* simulator only */
#define EXIT_TRAP             0x200000F0     /* simulator only */

/*********** interrupt bits **************/
#define IRQ_UART_READ_AVAILABLE         0x01
#define IRQ_UART_WRITE_AVAILABLE        0x02
#define IRQ_COUNTER18_NOT               0x04
#define IRQ_COUNTER18                   0x08
#define IRQ_GPIO30_NOT                  0x10
#define IRQ_GPIO31_NOT                  0x20
#define IRQ_GPIO30                      0x40
#define IRQ_GPIO31                      0x80
#define IRQ_NOC_READ                    0x100
```

Figura 5.2 – Endereço dos periféricos acessíveis por *software*

Um fluxo de execução básico do sistema é apresentado na Figura 5.3. Este fluxo não apresenta estados onde tarefas são mapeadas e excluídas durante a execução entre outros detalhes, por questões de simplicidade.

O fluxo de execução é descrito a seguir. Estruturas de dados do sistema operacional (e *hardware* específico) são inicializadas. Após esta inicialização, os tratadores de interrupção são registrados e habilitados. Neste momento, tarefas iniciais são adicionadas ao sistema e a execução é iniciada. O sistema fica em espera até o acontecimento de um evento de interrupção. Neste momento a rotina de serviço de interrupções é chamada, o contexto básico do processador é salvo e um tratador para a interrupção é invocado de acordo com a origem da interrupção. Em um evento de *timer*, o tratador de interrupções para escalonamento é chamado, o contexto da tarefa é salvo e o escalonador é invocado. Após o escalonamento, o contexto da tarefa escolhida é restaurado, e sua execução é continuada. Se durante a execução de uma tarefa, a mesma abrir mão de sua fatia de tempo de processador (modo cooperativo), o tratador de interrupções para escalonamento é chamado diretamente. Outras interrupções são tratadas da mesma maneira que interrupções de *timer*, entretanto não ocorre reescalonamento de tarefas. Após a execução do *kernel driver*, a rotina de serviço de interrupções restaura o contexto da tarefa interrompida e sua execução é continuada.

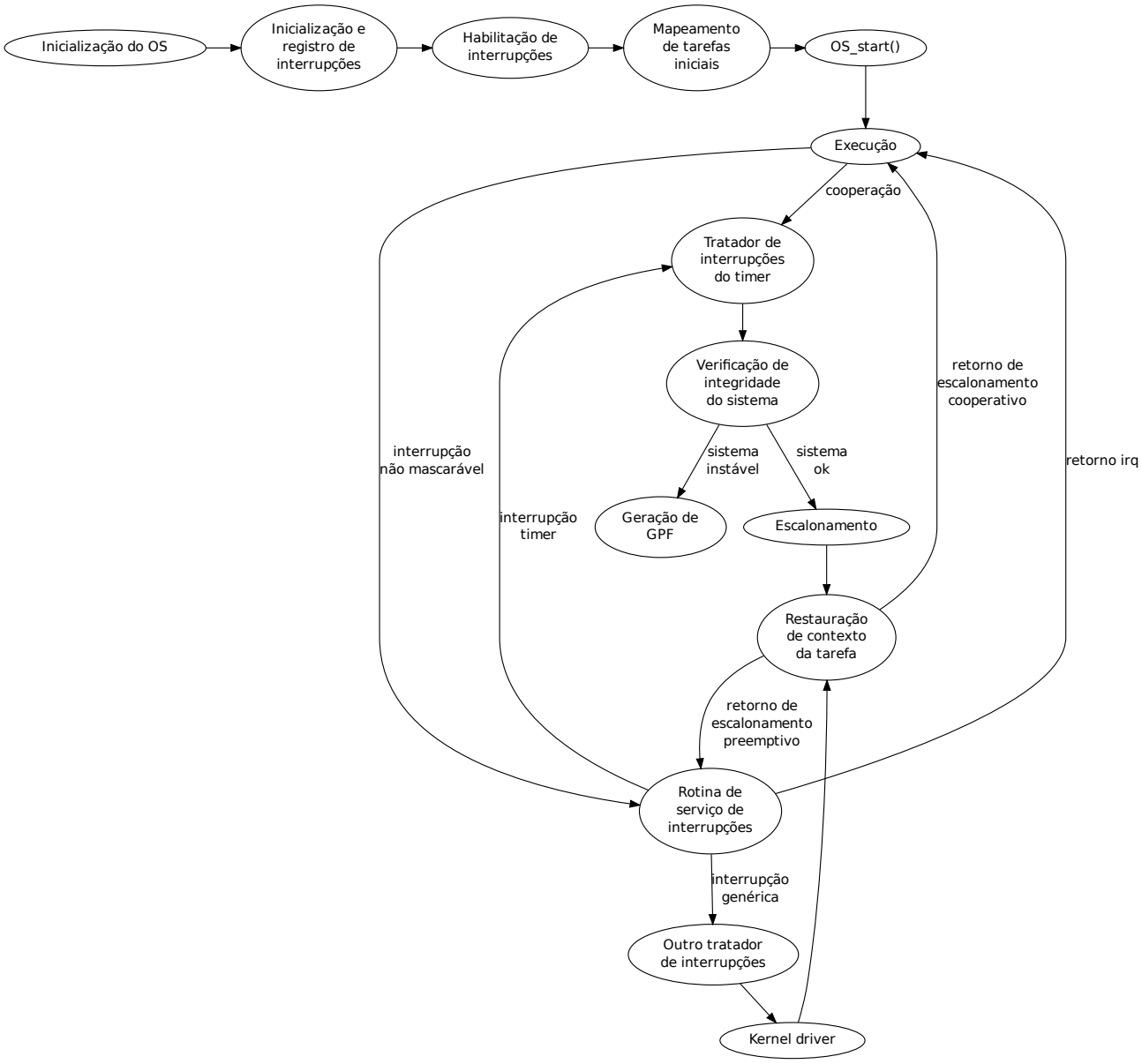


Figura 5.3 – Fluxo de execução simplificado do sistema operacional Hellfire OS

5.2 Medidas de Escalonamento

De acordo com [35], um sistema operacional de tempo real não é apenas definido por seu comportamento, ou seja sua política de escalonamento, mas também por suas propriedades temporais, as quais impactam na evolução da execução de um conjunto de tarefas. Neste trabalho, a implementação do modelo de tarefas replica estas propriedades, e origina o valor do parâmetro chamado *overhead* do sistema operacional.

5.2.1 Base de Tempo

No presente trabalho, a base de tempo é provida por um contador em *hardware*[2] de 32 *bits*, que opera na mesma frequência do elemento de processamento. Esta base de tempo é referenciada como *tick*, e corresponde às unidades das medidas utilizadas nas definições dos parâmetros de tarefa descritos na Seção 4.4. Pode-se selecionar um sinal apropriado deste contador em *hardware*, e a partir deste sinal obter a geração de interrupções de *timer*, as quais ocorrem em uma inversão lógica do sinal selecionado. De acordo com o sinal e frequência de operação, podem ser obtidos diferentes períodos de *tick*.

O período de *tick* é calculado de acordo com a fórmula, onde *a* é o sinal desejado do contador e *freq* é a frequência de operação do elemento de processamento, em *hertz*:

$$period = \frac{2^a}{freq}$$

Diferentes frequências de operação e seleção de sinais do contador definem um grande conjunto de valores para o tempo do *tick*, permitindo ao desenvolvedor a escolha da granularidade de escalonamento adequada a uma determinada aplicação. A Tabela 5.1 apresenta valores para tempos de *tick*, variando-se o sinal selecionado do contador (*bit*) e a frequência de operação. Por exemplo, ao selecionar o décimo quinto *bit* menos significativo do contador, temos os tempos apresentados na segunda coluna da tabela, para cada frequência. Pode ser observado que a seleção de sinais com *bits* de maior significância torna maior o tempo de *tick*, uma vez que estes são invertidos no contador em *hardware* com menor frequência.

A Tabela 5.2 enumera a quantidade de interrupções de *timer* por segundo, de acordo com a frequência de operação e sinal selecionado do contador. Observa-se que a 100MHz e com o sinal 15 selecionado, são geradas 3125 interrupções por segundo, o que equivale, em um algoritmo de escalonamento que não reescalona a tarefa recém preemptada ao mesmo número de trocas de contexto.

Os valores de 25MHz para a frequência de operação e sinal 18 do contador foram utilizados como padrão, o que corresponde ao período de 10.48ms entre interrupções. Assim, são realizadas

[2] Detalhes sobre a arquitetura empregada estão descritos na Seção 5.8.

Tabela 5.1 – Valores para tempos de *tick*

Frequência de operação (MHz)	15	16	17	18	19	20	21
25	1.31ms	2.62ms	5.24ms	10.48ms	20.97ms	41.94ms	83.88ms
33	0.99ms	1.98ms	3.97ms	7.94ms	15.88ms	31.77ms	63.55ms
50	0.65ms	1.31ms	2.62ms	5.24ms	10.48ms	20.97ms	41.94ms
66	0.49ms	0.99ms	1.98ms	3.97ms	7.94ms	15.88ms	31.77ms
100	0.32ms	0.65ms	1.31ms	2.62ms	5.24ms	10.48ms	20.97ms

Tabela 5.2 – Número de trocas de contexto

Frequência de operação (MHz)	15	16	17	18	19	20	21
25	763.36	381.68	190.84	95.42	47.69	23.84	11.92
33	1010.1	505.05	251.89	125.94	62.97	31.48	15.74
50	1538.46	763.36	381.68	190.84	95.42	47.69	23.84
66	2040.82	1010.1	505.05	251.89	125.94	62.97	31.48
100	3125	1538.46	763.36	381.68	190.84	95.42	47.69

aproximadamente 95 chamadas ao escalonador por segundo. Estes valores são equivalentes ao protótipo em *hardware*, e foram escolhidos como um compromisso entre o tempo de resposta do sistema operacional, facilidade na prototipação e *overhead* em virtude das trocas de contexto.

5.2.2 *Overhead* do Sistema Operacional

O sistema operacional Hellfire OS provê a chamada de sistema[3] *OS_LastContextSwitchTime()* que retorna o tempo gasto em trocas de contexto em ciclos. Esta chamada utiliza o contador em *hardware* para efetuar a medição, sendo portanto independente das ferramentas de *software*. Tendo-se o tempo gasto em trocas de contexto (dependente da política de escalonamento, implementação e compilador utilizado), o número de interrupções de *timer* por segundo (*ticks*) e a frequência de operação, o *overhead* pode ser calculado por:

$$overhead = \frac{tps \times csl}{freq}$$

Onde *overhead* é expresso por um número entre 0 e 1, *tps* é o número de *ticks* por segundo, *csl* é a latência (ou tempo) das trocas de contexto e *freq* é a frequência de operação, em *hertz*. O tempo de uma troca de contexto é despendido sempre que ocorrer uma interrupção de *timer*. Assim, esse custo incide sempre sobre o progresso das tarefas, uma vez que as fatias de tempo de processador são distribuídas de acordo com a política de escalonamento empregada, e o *overhead* é absorvido a cada *tick*.

[3] A API do sistema operacional Hellfire OS é apresentada na Seção 5.6.

Como exemplo, a uma frequência de operação de 25MHz e um período entre interrupções de 10.48ms, uma tarefa escalonada executa por aproximadamente 262000 ciclos (supondo que no período em questão a tarefa não realiza chamada por reescalonemento e a mesma seja preemptada após o término do *tick*). Se for considerada uma latência de 1500 ciclos do sistema operacional[4], é observado um *overhead* de aproximadamente 0.57%.

Interrupções do *timer* são utilizadas para a geração de *ticks* do sistema. O seu período deve ser bem balanceado, de forma que uma fatia de tempo muito longa pode tornar o sistema pouco responsivo (e pode não honrar as restrições de tempo real) e uma fatia de tempo muito curta pode aumentar o *overhead* do sistema operacional.

5.3 Implementação do Modelo de Tarefas

Uma tarefa é definida pelos parâmetros descritos na Seção 4.4. Essencialmente, o sistema operacional descrito implementa tarefas periódicas (primitiva *OS_AddPeriodicTask()*) e aperiódicas (primitiva *OS_AddTask()*). Assim, os parâmetros essenciais para o mapeamento de tarefas periódicas no sistema são a identificação, o período, o tempo de execução e o *deadline*. O comportamento de uma tarefa é definido como um bloco de código em linguagem C, implementado por uma função do tipo *void*, ou seja, uma função que não recebe parâmetros nem retorna valores. Uma tarefa pode ser entendida como uma função que itera infinitamente, mas pode ser interrompida a qualquer momento pelo sistema operacional (a tarefa é preemptada) e ter sua execução continuada posteriormente. O escalonamento de tarefas pode utilizar diversas políticas, conforme descrito na Seção 4.7 e na implementação atual foi utilizada a política RM para tarefas de tempo real [40, 36].

Uma troca de contexto ocorre apenas por interrupção de *hardware* (que pode ser mascarada), ou a tarefa desiste da execução voluntariamente (primitiva *OS_TaskYield()*), permitindo que o sistema operacional escolha outra tarefa para execução. A Figura 5.4 apresenta um exemplo de implementação, mostrando de maneira geral como uma tarefa é organizada. Variáveis locais são declaradas no corpo da tarefa, e armazenadas em sua pilha. O código de inicialização é um segmento de código que executa apenas uma vez, não sendo seu uso mandatório (pode ser utilizado, no entanto, para inicializar estruturas de dados da tarefa). O verdadeiro código da tarefa executa em um laço infinito.

Cada tarefa do sistema encontra-se em um dos seguintes estados: *pronta*, *executando*, *bloqueada*, *esperando* e *não executou ainda*. A tarefa é considerada *pronta* quando a mesma foi preemptada pelo sistema operacional ou realizou pedido de reescalonamento voluntariamente. Neste estado, a tarefa encontra-se na fila de escalonamento. No estado *executando* a tarefa encontra-se em execução, e esta acabou de ser escalonada. A tarefa encontra-se no estado *bloqueada* quando está pronta para executar, no entanto foi removida da fila de escalonamento (voluntariamente ou não).

[4]Valor estimado, obtido por testes realizados no sistema operacional Hellfire OS compilado com GCC 4.6.0 e executando a política Rate Monotonic com 10 tarefas. A latência depende de fatores como compilador, arquitetura, política de escalonamento e sua implementação e número de tarefas.

```
void Task(void){
    /* (variáveis alocadas na pilha) */
    unsigned int i,j;

    /* (código de inicialização) */
    j = 10;

    while(1){
        */ (código da tarefa) */
        for(i=0 ; i<j ; i++){
            printf("\nHello World! %d", i);
        }
    }
}
```

Figura 5.4 – Corpo da descrição de uma tarefa exemplo

No estado *esperando* a tarefa está em espera em um semáforo, e não pode progredir sua execução até que outra tarefa incremente o mesmo até o ponto em que ela seja liberada. Inicialmente, todas as tarefas encontram-se no estado *não executou ainda*. Após a primeira execução, se não ficar presa em um semáforo ou bloqueada uma determinada tarefa é mantida no estado *pronta* até que seja escalonada novamente. Caso não exista tarefa a ser escalonada, uma tarefa especial adicionada na inicialização do sistema chamada *idle task* é escalonada, não podendo esta ser bloqueada ou excluída. Apenas são executadas tarefas que estiverem na fila de escalonamento. Os possíveis estados de uma tarefa são apresentados na Figura 5.5.

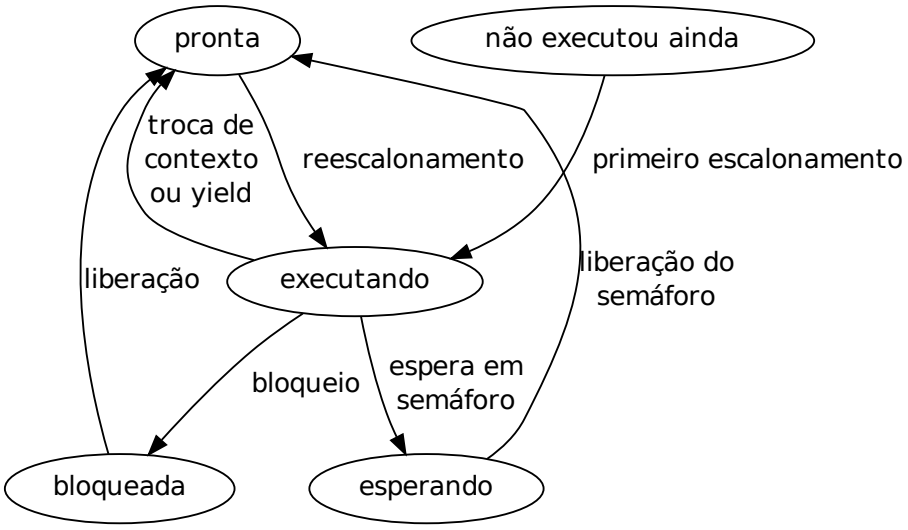


Figura 5.5 – Estados das tarefas

Para garantir a execução de tempo real do sistema, tarefas não podem desabilitar interrupções. Mascarar uma interrupção do *timer*, mesmo que por um curto espaço de tempo, pode fazer com que o *kernel* perca a interrupção e o escalonamento perca sua validade de tempo real.

Todas as informações que dizem respeito a tarefas são armazenadas em uma estrutura especial denominada TCB ou bloco de controle de tarefa. Nesta estrutura, o sistema operacional

mantém todas as propriedades das tarefas: sua identificação, descrição, estado de escalonamento, informações de progresso, período, tempo de execução, *deadline*, utilização do processador e memória, contexto da tarefa, ponteiros de uso geral (região de memória da pilha, por exemplo) e informações sobre transmissão de dados.

5.3.1 Escalonamento de Tarefas

O mecanismo de escalonamento foi implementado em dois níveis. A cada *tick*, o escalonador de tarefas periódicas (primeiro nível) é executado, e tarefas de tempo real são tratadas de acordo com a política RM representada no Algoritmo 5.6. Caso não existam tarefas de tempo real a serem escalonadas (o escalonador de tempo real retorna 0), o escalonador de tarefas de melhor esforço (segundo nível) é executado. Este escalonador, apresentado no Algoritmo 5.7, escolhe entre as tarefas de melhor esforço de acordo com um algoritmo circular. Assim, as tarefas periódicas possuem precedência sobre as aperiódicas de melhor esforço. Apenas as tarefas de tempo real são consideradas no momento em que é avaliada a carga do processador, uma vez que estas obrigatoriamente devem executar. Tarefas de melhor esforço são executadas apenas se houver tempo de processador livre, e desta forma não incidem em carga extra.

Entre as tarefas de melhor esforço encontra-se uma tarefa de sistema, denominada *idle task*. Esta tarefa possui a identificação de número zero, e tem um papel fundamental para o funcionamento do sistema operacional. A *idle task* não pode ser excluída, bloqueada ou migrada. Na Figura 5.8 é apresentado um exemplo do funcionamento do escalonamento em dois níveis. As tarefas 1, 2 e 3 são tarefas de tempo real, e possuem os parâmetros $\tau_i = \{p_i, e_i, d_i\}$ definidos como $\tau_1 = \{4, 1, 4\}$, $\tau_2 = \{6, 2, 6\}$ e $\tau_3 = \{8, 1, 8\}$. As tarefas 0 (*idle task*), 4, 5, 6 e 7 são tarefas de melhor esforço. Neste exemplo a carga do elemento de processamento é 70%. Se for levado em consideração um tempo de *tick* de 10.48ms e uma frequência de 25MHz, o tempo de execução das tarefas τ_1, τ_2 e τ_3 seria aproximadamente 262000, 524000 e 262000 ciclos respectivamente. Ainda, a tarefa τ_1 executa a cada 42ms (4 *ticks*), τ_2 a cada 63ms (6 *ticks*) e τ_3 a cada 84ms (8 *ticks*).

Logicamente, uma tarefa pode abrir mão de sua fatia de processador a qualquer momento, e não executar até completar seu *tick*. Esse tipo de situação ocorre quando uma tarefa deve ser executada em um período determinado (a cada 100ms por exemplo) e completa seu trabalho rapidamente (em 10000 ciclos por exemplo). O restante do tempo de tick pode ser utilizado por outras tarefas, e dessa forma a tarefa voluntariamente pede por reescalonamento. Na atual implementação, no momento em que uma tarefa abre mão de tempo de processador, o escalonador de melhor esforço é executado.

Tarefas aperiódicas de tempo real, também conhecidas como eventos, possuem maior prioridade que todas as outras do sistema, e são tratadas de maneira especial. Este tipo de tarefa não depende de outras, e ocorre em situações especiais como eventos externos (por exemplo, interrupções da interface de interconexão). Ao ocorrer um evento externo, uma parte do contexto da tarefa

```
1. j ← 65535
2. schedule ← 0
3. for all i = 1, ...max_tasks do
4.    task ← tasks[i]
5.    if task and task.period > 0 then
6.       if task.status = READY or task.status = NOT_RUN then
7.          if task.period < j and task.capacity_counter > 0 then
8.             j ← task.period
9.             schedule ← i
10.         end if
11.         task.priority ← task.priority − 1
12.         if task.priority = 0 then
13.            task.next_tick_count ← task.next_tick_count + task.period
14.            task.priority ← task.period
15.            if task.capacity_counter > 0 then
16.               task.deadline_misses ← task.deadline_misses + 1
17.               task.capacity_counter ← task.capacity
18.            end if
19.         end if
20.      end if
21.   end if
22. end for
23. if schedule = 0 then
24.   return 0
25. else
26.   task.capacity_counter ← task.capacity_counter − 1
27.   return schedule
28. end if
```

Figura 5.6 – Escalonamento Rate Monotonic

```
1. while TRUE do
2.    if i < max_tasks then
3.       i ← i + 1
4.    else
5.       i ← 0
6.    end if
7.    if task and task.period > 0 then
8.       if task.status = READY or task.status = NOT_RUN then
9.          return i
10.      end if
11.   end if
12. end while
```

Figura 5.7 – Escalonamento circular

corrente (seja ela uma tarefa de tempo real ou não) é salvo na pilha da mesma, e um tratador (ou *driver*) é executado. Ao término da execução do evento, o contexto da tarefa interrompida é

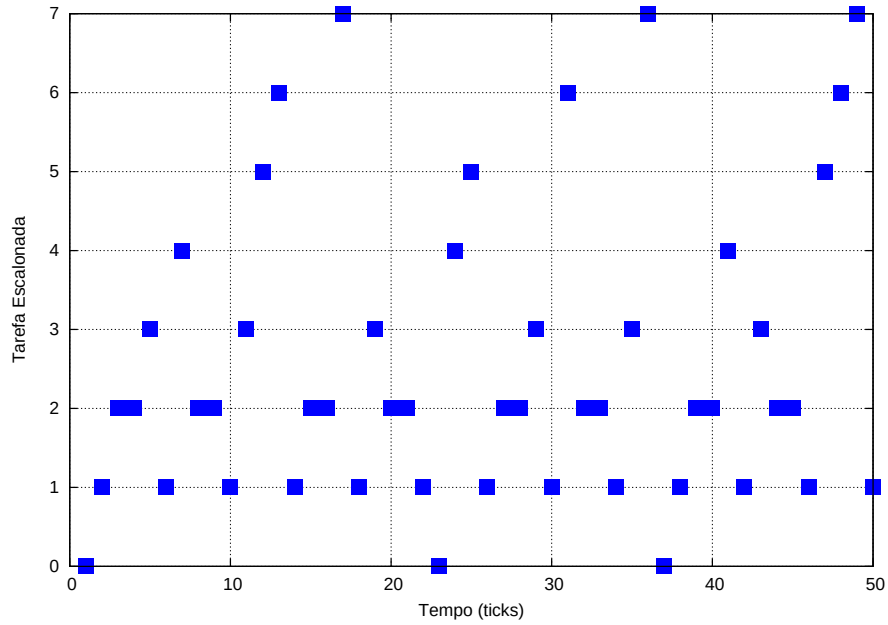


Figura 5.8 – Escalonamento de tarefas em dois níveis

restaurado. Vale ressaltar que este mecanismo apenas preempta temporariamente a tarefa corrente, pois a mesma é continuada após o término do evento. Caso uma tarefa periódica de tempo real não possa ser interrompida em determinados momentos por eventos aperiódicos, o desenvolvedor pode mascarar interrupções de determinado dispositivo (exceto o *timer*).

5.3.2 Comunicação Entre Tarefas

A comunicação entre tarefas é realizada por dois modelos diferentes no sistema Hellfire OS. O primeiro é o modelo de comunicação por memória compartilhada, adequado para tarefas que executam no mesmo nodo. O outro modelo é comunicação por trocas de mensagens, adequado para tarefas que executam em nodos diferentes.

Estes dois modelos diferem em sua perspectiva de programação. Comunicação por memória compartilhada pode ser implementada pela proteção de uma estrutura de dados global compartilhada. Esta estrutura pode ser de qualquer tipo, como por exemplo uma *struct* em linguagem de programação C. A proteção é feita com o uso de primitivas para exclusão mútua ou semáforos [71]. Esta proteção precisa ser utilizada para evitar que mais de uma tarefa acesse a mesma estrutura de dados concomitantemente[5], corrompendo os dados. Primitivas como *OS_MsgSend()*, *OS_MsgRecv()*, *OS_MboxSend()*, *OS_MboxRecv()* e *OS_MboxAccept()* encapsulam este modelo e abstraem sua implementação como trocas de mensagem. Apenas tarefas fixas e em um mesmo nodo podem utilizar tais primitivas, uma vez que a memória local é utilizada para realizar a comunicação. Nestas primitivas, não são utilizados os *drivers* de comunicação da NoC.

[5]Diz-se que tarefas acessam dados concomitantemente quando determinada tarefa modifica uma estrutura de dados (mas não completa a modificação) e ocorre uma troca de contexto, sendo que a tarefa escalonada também acessa a estrutura, corrompendo dados (em uma escrita) ou lendo dados corrompidos.

A comunicação por trocas de mensagens é implementada com o uso de primitivas específicas do sistema operacional, tais como *OS_Send()*, *OS_Receive()* e *OS_UniqueIDSend()* que podem enviar e receber qualquer tipo de dado. O programador é responsável por alocar *buffers* em nível de aplicação e especificar a identificação única da tarefa a receber os dados em um envio. A tarefa destino automaticamente identifica a tarefa fonte em um recebimento de dados.

As primitivas de comunicação por trocas de mansagem seguem o modelo produtor / consumidor. Cada tarefa possui uma fila circular local de recepção com tamanho configurável, contendo pacotes que podem ser retirados em ordem pela primitiva adequada. Se a fila estiver vazia, a tarefa fica bloqueada na primitiva de recebimento (no caso de uma primitiva bloqueante) ou é mantida na primitiva até que ocorra um *timeout*, especificado na aplicação. Da mesma forma, uma tarefa que envia dados a outra pode ficar bloqueada na primitiva de envio no caso de contenções na rede. Caso a tarefa receptora não possua mais espaço na fila de recepção, pacotes subsequentes são descartados. O descarte de pacotes foi utilizado pois existe uma única fila em *hardware* por nodo, e normalmente existem diversas tarefas em cada nodo. Se uma tarefa não está tratando os pacotes recebidos, apenas os seus são descartados, não comprometendo a recepção de mensagens de outras tarefas. A inserção de pacotes na fila de recepção, assim como bloqueio e liberação de tarefas é gerenciado por *drivers* do sistema operacional. Um controle de fluxo pode ser implementado em nível de aplicação em primitivas de comunicação direta entre tarefas (primitivas *OS_Send()* e *OS_Receive()*) e é implementado automaticamente no caso do protocolo de comunicação por identificação única (*OS_UniqueIDSend()*).

A Figura 5.9 apresenta o caminho efetuado pelos dados durante uma troca de mensagem entre duas tarefas de nodos distintos. Inicialmente, a primitiva de envio da origem encapsula a mensagem contida em um espaço de memória em nível de aplicação em pacotes, preenchendo a fila de saída da tarefa (1). Desta fila é retirado um pacote, o qual é copiado para a fila de saída em *hardware* (2). A interface de rede é sinalizada, e o pacote é enviado pela rede (3). Ao chegar ao nodo destino preenchendo a fila de entrada, é gerada uma interrupção e o sistema operacional retira o pacote desta fila e o decodifica, encaminhando este para a fila da tarefa destino (4). Posteriormente a mensagem é desencapsulada e copiada para um espaço de memória em nível de aplicação (5).

A implementação das primitivas de comunicação por trocas de mensagem foi realizada em dois níveis. As primitivas de comunicação de alto nível, expostas na API do sistema, são responsáveis por encapsular e desencapsular mensagens em pacotes de dados, tomando conta de detalhes como *padding* e sequenciamento. Internamente, *drivers* do sistema são responsáveis por realizar a transferência de pacotes entre nodos origem e destino. Estes *drivers* trabalham com pacotes de tamanho fixo, e sinalizam a interface de comunicação da rede durante o envio de dados para filas em *hardware*, e recebem um sinal da interface durante o recebimento de dados, quando retiram dados da fila de recebimento em *hardware* e copiam estes dados para filas circulares de pacotes de cada tarefa.

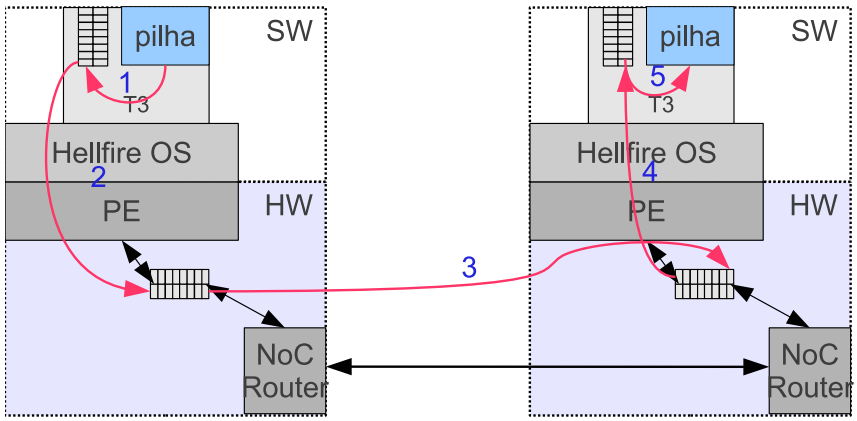


Figura 5.9 – Comunicação entre tarefas, filas de *software* e *hardware*

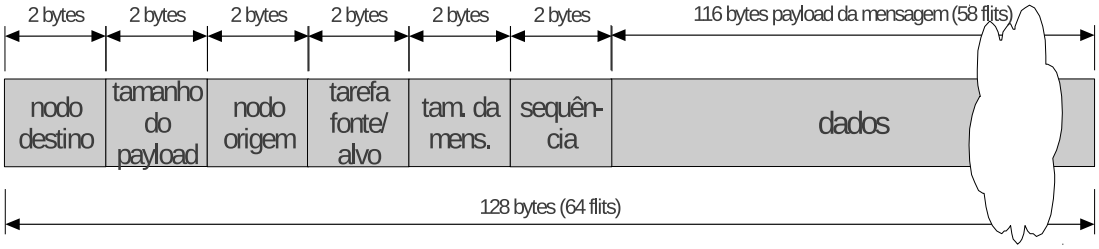


Figura 5.10 – Formato do pacote de dados

O formato dos pacotes de dados é apresentado na Figura 5.10. Os pacotes possuem um cabeçalho, contendo o endereço do roteador destino e o tamanho do *payload*[6]. Após o cabeçalho são apresentados os dados a serem transferidos. Estes dados são compostos por uma identificação do nodo origem, uma identificação da tarefa origem e tarefa destino, o tamanho da mensagem (pode ser maior que o tamanho do pacote) um número de sequência e por fim os dados que serão colocados na fila de *software* da tarefa. Desta forma, os dados contidos após o *payload* serão gerenciados pelo *driver* de comunicação do sistema operacional, e seu conteúdo não é relevante para o meio de interconexão.

Este formato de pacote acarreta em um *overhead* de comunicação em torno de 9.37% (pacote de 64 flits de 16 bits) devido aos cabeçalhos. O tamanho de pacote pode ser modificado, e para isso tanto o tamanho das filas em *hardware* quanto a configuração do sistema operacional precisam ser as mesmas. Este tamanho foi definido como padrão após uma série de testes, e escolhido devido ao seu compromisso entre tamanho das filas, tempo de processamento de pacotes e *overhead* devido ao *padding*. A Figura 5.11 apresenta o desempenho de pico na transferência de dados em nível de aplicação para diferentes tamanhos de fila em *hardware* empregados. Nos testes foi assumida uma situação ideal com o envio de uma única mensagem de cada tamanho, onde tarefas de tempo real de envio e recebimento ocupam toda a capacidade de processamento em dois nodos vizinhos para realizar a comunicação.

[6]Carga útil de dados do pacote.

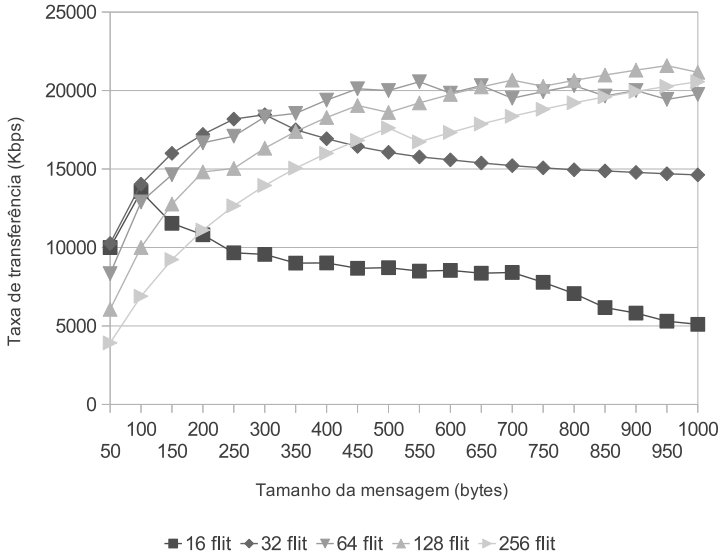


Figura 5.11: Tamanho das filas em *hardware*, desempenho de pico em trocas de mensagem entre dois nodos

As primitivas básicas para o envio e recebimento de mensagens assumem uma identificação fixa das tarefas envolvidas na comunicação. Conforme o modelo apresentado anteriormente na Seção 4.6, tarefas podem ser configuradas como não fixas, e o mecanismo para permitir a comunicação entre este tipo de tarefa foi apresentado na mesma Seção. Para tarefas não fixas é utilizada uma primitiva específica de envio (*OS_UniqueIDSend()*), sendo a primitiva de recebimento a mesma (*OS_Receive()*). A gerência do mecanismo é feita pelo sistema operacional, onde uma tarefa de controle organiza a comunicação. O controle de comunicação sincroniza o recebimento e atualiza tabelas com informação sobre migrações, para que requisições de primitivas *OS_UniqueIDSend()* sejam atendidas e tarefas que invocam tal primitiva tenham atualizada a informação de comunicação localmente. As primitivas de comunicação são apresentadas na Seção 5.6.

5.4 Alocação Dinâmica de Memória

Um mecanismo de alocação dinâmica de memória foi implementado para ser utilizado tanto pelo sistema operacional (alocação dos espaços de pilha das tarefas, filas de comunicação, estruturas de controle) quanto pelas tarefas da aplicação. Este mecanismo é implementado pelas primitivas *OS_Malloc()*, *OS_Free()*, *OS_Calloc()* e *OS_Realloc()*, e é genérico o suficiente para ser utilizado em diferentes arquiteturas. Tarefas que alocam memória dinamicamente devem ser configuradas como fixas, sendo esta uma limitação da atual implementação e discutida nos Trabalhos Futuros.

O alocador baseia-se em uma única região estática de memória em cada nodo, tendo seu tamanho definido em tempo de projeto. Para alocar um segmento de memória, o alocador percorre uma lista encadeada de ponteiros, onde são definidos o início e fim de cada segmento. A lista é

percorrida em ordem, e caso exista espaço contíguo entre segmentos já alocados para armazenar o espaço requerido (método *First Fit*), é feito o ajuste de ponteiros e retornada a posição inicial da região alocada. Durante a alocação também é feita a compactação do *heap*. A compactação consiste em unir segmentos contíguos anteriormente alocados e já liberados de memória em um único segmento contíguo memória livre, diminuindo-se a fragmentação.

5.5 Mapeamento de Tarefas no Sistema Hellfire OS

Esta Seção apresenta como é realizada a atividade de mapeamento no sistema operacional Hellfire OS. São abordados o mapeamento das tarefas iniciais e o suporte do sistema operacional para a realização de mapeamento dinâmico e migrações de tarefa. Os gerentes de migração são discutidos no final da Seção.

5.5.1 Mapeamento Inicial

Neste momento, o mapeamento das tarefas iniciais é realizado manualmente, isto é, o desenvolvedor é responsável por descrever a aplicação e definir os grupos de tarefa (particionamento), e a posição dos grupos nos respectivos nodos. Estas definições são feitas no código fonte da aplicação, como apresentado na Figura 5.12. No exemplo, seis tarefas iniciais são mapeadas em dois elementos de processamento. Não foi especificada a quantidade de nodos que compõem a arquitetura, sendo este um parâmetro de configuração do sistema operacional e da arquitetura em tempo de projeto. Apenas uma parte da descrição da aplicação onde é realizado o mapeamento está sendo apresentada, sendo a implementação das tarefas não relevante no exemplo.

```
void ApplicationMain(void){
        // tarefa T_i = <t_i, n_CPU_ID t_i, p_i, e_i, d_i>
        // tarefas T0, T1 e T2 são mapeadas durante a inicialização
#if CPU_ID==0
        // tarefa T3 = <t3, n0t3, 50, 1, 50>
        OS_AddPeriodicTask(task1, 50,1,50, "task one", 1024, 1, TASK_CAN_MIGRATE);
        // tarefa T4 = <t4, n0t4, 5, 1, 5>
        OS_AddPeriodicTask(task2, 5,1,5, "task two", 1500, 1, TASK_CAN_MIGRATE);
        // tarefa T5 = <t5, n0t5, 5, 1, 5>
        OS_AddPeriodicTask(task3, 5,1,5, "task three", 1050, 1, TASK_CANNOT_MIGRATE);
        // tarefa T6 = <t6, n0t6>
        OS_AddTask(task4, "task four", 1024, 1, TASK_CANNOT_MIGRATE);
#endif
#if CPU_ID==1
        // tarefa T3 = <t3, n1t3, 5, 1, 5>
        OS_AddPeriodicTask(task1, 5,1,5, "task one", 2048, 1, TASK_CAN_MIGRATE);
        // tarefa T4 = <t4, n1t4, 5, 1, 5>
        OS_AddPeriodicTask(task2, 5,1,5, "task two", 2048, 1, TASK_CAN_MIGRATE);
#endif

        OS_Start();
}
```

Figura 5.12 – Mapeamento de tarefas iniciais

A identificação da tarefa no contexto de execução (por exemplo $t3$) é gerenciada pelo sistema operacional, assim como sua identificação global (por exemplo $n0t3$). Dessa forma, os parâmetros passados às primitivas de mapeamento de tarefas consistem em um ponteiro para a função que implementa sua funcionalidade[7], parâmetros de execução (período, capacidade e *deadline*), um texto de identificação, tamanho da pilha, valor de consumo energético arbitrário e definição sobre a possibilidade de migração ou não da tarefa (fixa ou não fixa).

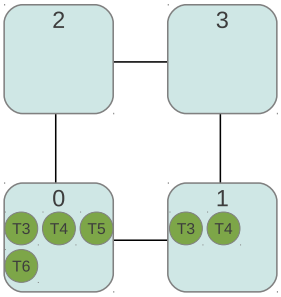


Figura 5.13 – Mapeamento de tarefas iniciais em uma malha 2x2

A Figura 5.13 descreve este mapeamento de uma maneira simplificada, representando a posição das tarefas nos nodos da arquitetura em uma malha 2x2. Não estão representadas no exemplo as tarefas do sistema operacional, sendo estas a tarefa 0 (*idle*), tarefa 1 (controle de comunicações) e tarefa 2 (gerente de migração).

5.5.2 Repositório Local de Tarefas

Em conjunto com o mapeamento das tarefas iniciais, são definidos os conteúdos dos repositórios locais de cada nodo. Estes repositórios possuem a implementação das tarefas, e são organizados de maneira distribuída. Assim como o mapeamento das tarefas iniciais, o conteúdo dos repositórios é especificado manualmente, e pode ser realizado com o uso de diretivas de pré-processador ou arquivos de código fonte separados.

Na Figura 5.14 é apresentada a organização do repositório local de tarefas. Cada instância de sistema operacional possui seu próprio repositório, que pode ser modificado dinamicamente. Durante inicialização do sistema, o repositório local é formado por tarefas iniciais e tarefas locais dinâmicas, ou seja, que não foram mapeadas. Qualquer tarefa presente no repositório local pode ser mapeada por outra tarefa ou durante a inicialização pelo sistema operacional. Caso venha a ser excluída, uma tarefa local permanece no repositório. A modificação dinâmica dos repositórios locais compreende na relocação de tarefas e faz parte do mecanismo de migração, apresentado na Seção 5.5.4.

Cada nodo possui uma única memória local e o código do sistema operacional, assim como o repositório de tarefas local, é mantido nesta memória. Tarefas armazenadas no repositório local

[7] Implementação do código da tarefa. Esta função pode realizar chamadas a outras funções recursivamente.

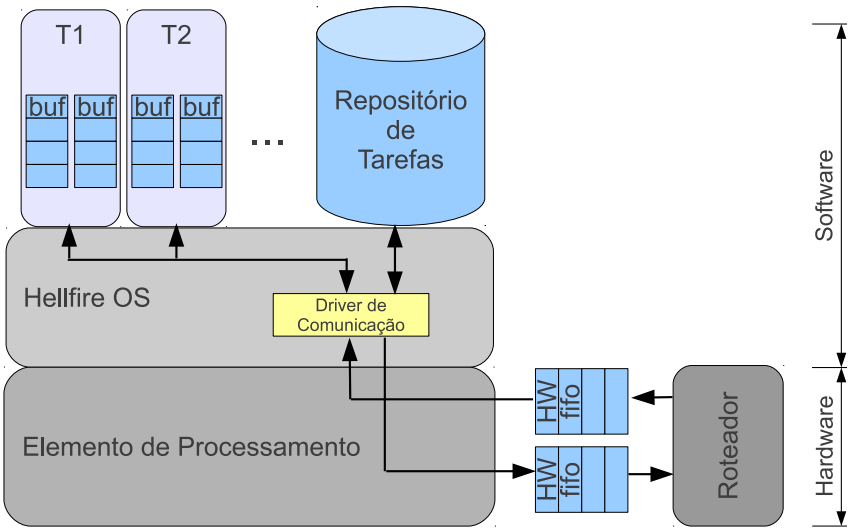


Figura 5.14 – Repositório local de tarefas

original não podem ser removidas do mesmo. No entanto, novas tarefas de repositórios remotos podem ser adicionadas na área de memória dinâmica, e removidas posteriormente. A Figura 5.15 apresenta de maneira simplificada a disposição dos elementos de *software* na memória de um nodo da arquitetura. Na área estática, encontram-se juntamente com o o código do sistema operacional as tarefas locais. A região de dados (pilha) das tarefas locais é alocada dinamicamente, pois a alocação é feita em tempo de execução e com tamanho que varia entre cada tarefa. A região de memória dinâmica, além de servir para alocar estruturas de dados dinamicamente para as tarefas locais e o sistema operacional, armazena o código das tarefas que migraram para este nodo, ou seja, que fazem parte do repositório local.

5.5.3 Mapeamento Dinâmico

O mapeamento dinâmico de tarefas é realizado pela adição de novas tarefas (primitivas *OS_Add PeriodicTask()*, *OS_AddTask()* e *OS_Fork()*) em tempo de execução. Este mapeamento é realizado por outras tarefas, e pode alterar as características de carga do sistema. Logicamente, a migração de tarefas ou a exclusão destas possui o efeito inverso, liberando recursos localmente.

Na Figura 5.16 é apresentado um exemplo onde é realizado o mapeamento dinâmico com o uso da primitiva *OS_Fork()*. Basicamente, uma única tarefa da aplicação é mapeada inicialmente. Esta tarefa realiza a própria replicação após certo ponto em sua execução, e uma outra tarefa com as mesmas características é mapeada dinamicamente variando a carga do sistema. A implementação do exemplo é apresentada na primeira parte da Figura, onde é ilustrado o o uso da API do sistema operacional.

Nas Figuras 5.17 e 5.18 é apresentado um exemplo do escalonamento das tarefas de uma aplicação sintética que utiliza mapeamento dinâmico e a carga resultante de sua execução. Esta

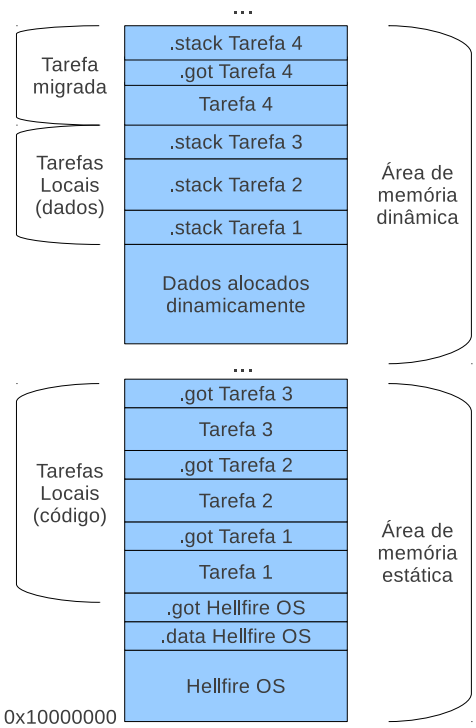
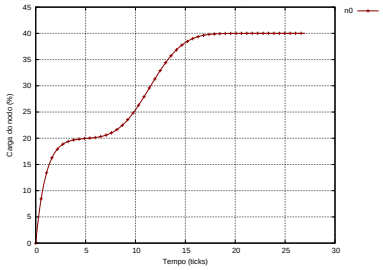


Figura 5.15 – Disposição dos elementos na memória de um nodo

```
#include <prototypes.h>

void task(void){
        int32 i, val,c=0;
        uint8 forked=0, locked;
        uint16 period, capacity, deadline;

        while(1){
                OS_TaskParameters(OS_CurrentTaskId(), &period, &capacity, &deadline, &locked);
                printf("\nTask %d, c == %d period = %d, capacity = %d, deadline = %d",
OS_CurrentTaskId(), c, period, capacity, deadline);
                c++;
                if ((forked == 0) && (c > 500)){
                        forked = 1;
                        val = OS_Fork();

                        if (val > -1){
                                printf("\nOS_Fork() succeded!");
                        }else{
                                printf("\nOS_Fork() failed!");
                                for(;;);
                        }
                }
        }
}

void ApplicationMain(void){
        OS_AddPeriodicTask(task, 10,2,10, "fork", 1024, 1, TASK_CAN_MIGRATE);

        OS_Start();
}
```



(a) (b)

Figura 5.16 – Variação de carga (b) após o mapeamento dinâmico de uma tarefa (a)

aplicação consiste de um conjunto inicial de cinco tarefas, sendo a tarefa 1 a única de tempo real e tendo seus parâmetros $\tau_i = \{id_i, p_i, e_i, d_i\}$ definidos como $\tau_1 = \{t1, 4, 1, 4\}$. Após aproximadamente 60 *ticks*, a tarefa 2 mapeia dinamicamente a tarefa $\tau_5 = \{t5, 6, 3, 6\}$ e após mais 40 *ticks* a tarefa $\tau_6 = \{t6, 8, 1, 8\}$. Observa-se que a tarefa τ_5 executa até aproximadamente o *tick* 120, onde esta é excluída. Após sua exclusão a utilização do elemento de processamento é reduzida, e as tarefas de melhor esforço passam a executar com maior frequência.

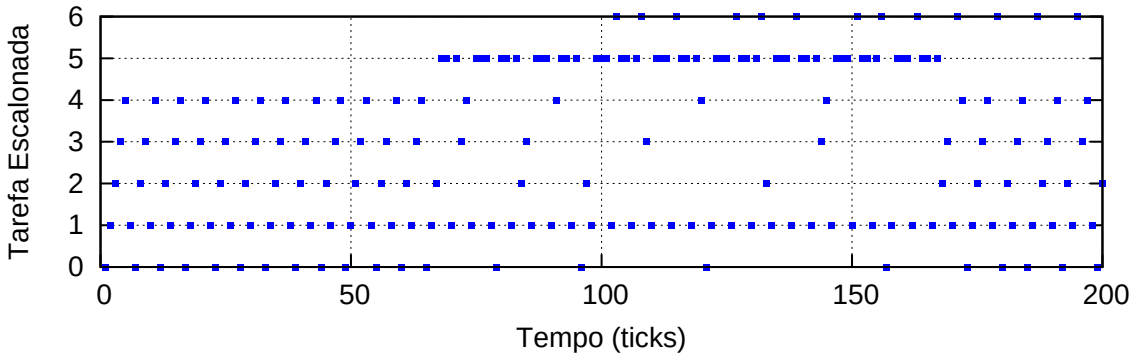


Figura 5.17 – Escalonamento de tarefas e mapeamento dinâmico

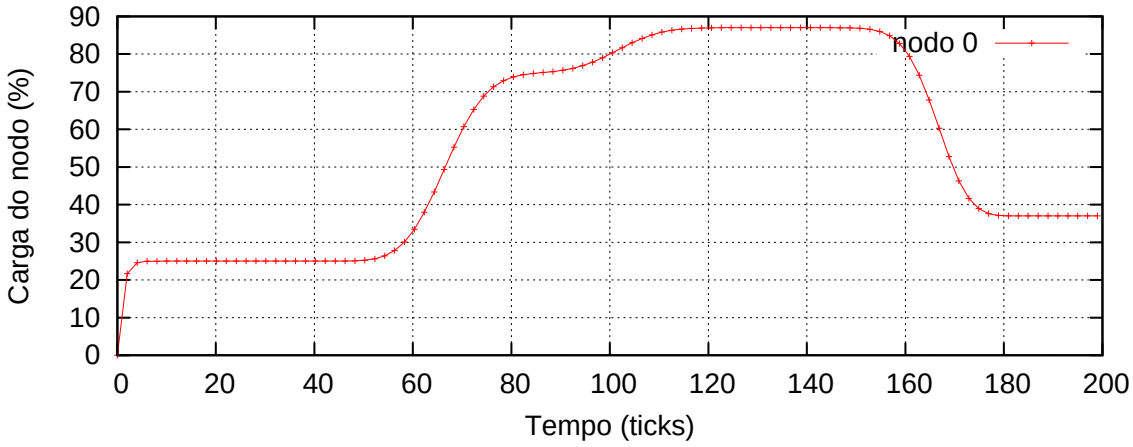


Figura 5.18 – Carga de processamento e mapeamento dinâmico

A tarefa 5 possui uma utilização de processamento de 50%, e em virtude disso as tarefas de melhor esforço perdem prioridade, como pode ser observado pelo seu perfil de execução na Figura 5.17. A tarefa 1, por possuir a maior prioridade de acordo com a política RM, não tem seu perfil de execução modificado durante toda a execução da aplicação. As tarefas 5 e 6 possuem um *jitter* variável, entretanto após o término da tarefa 5, a tarefa 6 passa a executar sem variações. Esse *jitter* é aceitável, uma vez que as tarefas cumprem com seus *deadlines* dentro do período estabelecido. A carga máxima de processamento é de 87% e nenhum *deadline* é perdido durante a execução.

5.5.4 Migração de Tarefas

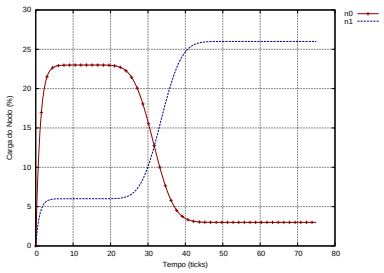
O sistema operacional atualmente possui uma primitiva que permite a migração explícita de tarefas de um nodo para outro. A primitiva *OS_TaskMigrate()* realiza esta função, e aceita como parâmetros a identificação de uma tarefa local e o nodo destino da tarefa.

Todas as tarefas que forem mapeadas no sistema podem ser migradas, desde que tenham sido previamente configuradas como não fixas com a opção *TASK_CAN_MIGRATE*. Tarefas de sistema, tarefas fixas, assim como a tarefa *idle task* e *drivers* são configurados com a opção *TASK_CANNOT_MIGRATE*, e a primitiva de migração é impedida de migrar tais tarefas.

Na Figura 5.19 é apresentado um exemplo de uso da primitiva de migração. No exemplo, existem duas tarefas da aplicação atribuídas ao nodo 0. Após executarem por um tempo determinado pelo algoritmo, a tarefa *migration* executa a primitiva de migração, transferindo a tarefa *i am alive* para o nodo 1.

```
#include <prototypes.h>

#if CPU_ID==0
void task(void){
        uint32 i, val,c=0,l;
        uint8 mig=0;

        while(1){
                c++;
                printf("\nhohohohoho!!!");
                if ((mig == 0) && (c == 1000)){
                        mig = 1;
                        l = MemoryRead(COUNTER_REG);
                        val = OS_TaskMigrate(3, CORE(1));     // migrate task 3 to CPU 1
                        l = MemoryRead(COUNTER_REG) - l;
                        if (val == 0){
                                printf("\nOS_TaskMigrate() succeeded, time: %d cycles", l);
                        }else{
                                printf("\nOS_TaskMigrate() failed!");
                                for(;;);
                        }
                }
        }
}

void task2(void){
        uint32 counter=0,i;
        uint8 buf[100];

        while(1){
                for(i=0;i<sizeof(buf);i++)
                        buf[i] = random()%256;
                printf("\nI AM ALIVE!!! >>> %d %d", counter++, buf[0]);
        }
}
#endif
#if CPU_ID==1
void task3(void){
        while(1){
                printf("?");
        }
}
#endif

void ApplicationMain(void){
#if CPU_ID==0
        OS_AddPeriodicTask(task, 30,1,30, "migration", 2048, 1, TASK_CANNOT_MIGRATE);
        OS_AddPeriodicTask(task2, 5,1,5, "i am alive", 4096, 1, TASK_CAN_MIGRATE);
#endif
#if CPU_ID==1
        OS_AddPeriodicTask(task3, 15,1,15, "dummy", 1024, 1, 1);
#endif

        OS_Start();
}
```

(a)



(b)

Figura 5.19: Migração manual de tarefas na aplicação (a) e variação de carga dos elementos de processamento origem (n0) e destino (n1) (b)

A primitiva de migração além de realizar a transferência de uma tarefa para outro nodo insere em uma lista a identificação global da tarefa migrada. Caso seja enviada uma mensagem para a tarefa migrada, a tarefa de controle de comunicação responde para onde a tarefa foi migrada,

de forma que a tarefa do nodo origem possa descobrir o novo destino. O protocolo é descrito anteriormente na Seção 4.6. Caso a tarefa seja transferida para um nodo onde já esteve, sua entrada na lista de migração é removida.

Mecanismos para realizar a migração de tarefas em ambientes que possuem uma unidade de gerência de memória (MMU) são mais simples de serem implementados, uma vez que não é necessário nenhum cuidado adicional ao ser transferido o código de um nodo para outro. Uma unidade MMU é utilizada para realizar a tradução de endereços lógicos para físicos, permitindo a carga de código de tarefas em posições arbitrárias da memória.

Por questões de simplicidade do *hardware*, optou-se por não utilizar uma unidade de gerência de memória neste trabalho e sim um mecanismo parcialmente suportado pelo compilador. Este mecanismo, consiste na emissão de código relocável (PIC) em conjunto com uma tabela específica, que deve ser gerenciada pelo sistema operacional. A tabela gerada pelo compilador, e mantida pelo sistema operacional é chamada GOT, e nela são armazenados endereços físicos para saltos e acessos à memória. O compilador, dessa forma, realiza a geração de código com endereços relativos à tabela GOT. Por exemplo, uma tarefa compilada para o endereço de memória 0x00000000 é relocada para o endereço 0x00000A00 após o processo de migração. Para que o código possa executar adequadamente, é necessário apenas realizar a modificação nas entradas da tabela GOT, acresentando 0xA00 aos valores antigos. O trabalho de Marchesan [46] utiliza um sistema semelhante de relocação, no entanto este não possui qualquer relação com o trabalho aqui apresentado.

Para que este mecanismo funcione, o sistema operacional e as tarefas da aplicação precisam ser compilados com *flags* específicas e após a geração do código binário, este precisa ser processado e os valores iniciais das tabelas GOT de cada tarefa definidos. O processo de migração de tarefas transfere para um nodo destino, além do contexto e dados da pilha da tarefa, o seu código e sua GOT que faz parte do contexto da tarefa. Ao receber uma tarefa, o mecanismo de migração no destino atualiza as entradas na GOT da tarefa para os endereços físicos da memória.

5.5.5 Gerentes Distribuídos para Migração de Tarefas

A implementação dos gerentes de migração é apresentada a seguir. Cada gerente é uma tarefa de tempo real, e possui uma identificação local fixa. Os parâmetros desta tarefa podem ser modificados em tempo de projeto com o intuito de melhorar o tempo de estabilização do sistema em caso de sobrecargas. Sendo esta uma tarefa de tempo real, um percentual de carga de processamento é reservado para a gerência. Cada nodo possui um gerente local, e em conjunto trabalham para reduzir perdas de *deadline* em situações de sobrecarga. Os gerentes de migração podem ser adicionados opcionalmente ao sistema, sendo este um parâmetro de configuração do sistema operacional.

O algoritmo de gerência possui o seguinte funcionamento, e é apresentado no Algoritmo 5.20. Inicialmente, é verificada a presença de solicitações de informações sobre o nodo local por

parte de gerentes de outros nodos. Caso existam solicitações, responder a estas com informações locais e remover estes nodos da pesquisa por candidatos em um primeiro momento (nesta iteração do algoritmo). Juntamente com as solicitações, podem existir respostas atrasadas de outros nodos, em decorrência a perguntas feitas pelo gerente local anteriormente. Ignora-se estas respostas. A seguir, são verificadas as perdas de *deadline* do conjunto de tarefas e a carga do sistema, e caso a situação seja considerada normal (não houve um aumento nas perdas de *deadline* e a carga do sistema esteja abaixo do limite) o algoritmo é terminado nesta iteração. Caso contrário, o nodo é considerado sobrecarregado e é feita uma lista de tarefas que podem ser migradas. Desta lista, são removidas tarefas fixas e tarefas bloqueadas e a melhor tarefa é escolhida para ser migrada. Se não houver tarefa alguma para ser migrada, esta iteração do algoritmo é terminada, e o gerente deve esperar por algum tempo antes de executar novamente. Se houver pelo menos uma tarefa apta a ser migrada, é realizada uma pesquisa pelo melhor nodo candidato a receber a tarefa utilizando o algoritmo de espalhamento proposto. Para isto é composta uma lista com os candidatos, a qual é formada por respostas de recursos disponíveis em cada candidato. O melhor candidato é escolhido e é realizada a migração. Caso ocorra alguma falha, o próximo candidato é escolhido, e é realizada uma nova tentativa. Se não houver candidato após a falha, o gerente deve esperar por algum tempo antes de executar novamente. Se a migração for completada com sucesso, o gerente deve esperar por um tempo pequeno antes de iniciar uma nova iteração. Este tempo é calculado com base na identificação do nodo, e difere para cada um com o objetivo de evitar ao máximo situações de *deadlock*.

A escolha da melhor tarefa a migrar é realizada pela função *PickBestTask()*, e esta melhor tarefa é definida de acordo com o critério de escolha adotado. Na atual implementação, uma tarefa não fixa é escolhida aleatóriamente. A busca em espalhamento por nodos canditados a receberem a migração é realizada em *QueryOtherNodes()*. Dentre os nodos procurados, o melhor nodo é escolhido pela função *PickBestNode()* que leva em consideração os custos sobre tempo de processamento livre e memória disponível. O melhor nodo nesta iteração do algoritmo é escolhido como alvo, e a tarefa é migrada para este. Caso não exista um nodo com recursos suficientes para receber a tarefa, uma nova iteração do algoritmo por espalhamento é realizada, aumentando-se em um o número de saltos.

5.6 API do Sistema Hellfire OS

Nesta Seção é apresentada a API do sistema operacional Hellfire OS. A interface do sistema é relativamente simples e em conjunto com as bibliotecas implementadas fornece os serviços necessários para o desenvolvimento de aplicações embarcadas de tempo real. A API consiste em 6 classes de chamadas de sistema: gerenciamento de tarefas, informações do sistema, exclusão mútua, gerenciamento de memória, primitivas de comunicação e migração de tarefas. Esta API é apresentada na Tabela 5.3.

```
1. while TRUE do
2.    nodeslist ← AllNodes()
3.    while m ← Receive() do
4.       if m = "query" then
5.          Send(m.source_cpu, localinfo)
6.          RemoveFromNodesList(m.source_cpu)
7.       end if
8.    end while
9.    if ProcessorLoad() > local_ul and DeadLineMisses() > lastdeadlinemisses then
10.      for all i = 1, ...max_tasks do
11.         if TaskState(i) ≠ FIXED and TaskState(i) ≠ BLOCKED then
12.            AddToTaskList(i)
13.         end if
14.      end for
15.      if TaskListElements(tasklist) = 0 then
16.         Yield()
17.      else
18.         task ← PickBestTask(tasklist)
19.         while NodesToQuery(nodeslist) do
20.            nodes ← QueryOtherNodes(nodeslist)
21.            node ← PickBestNode(nodes)
22.            if node then
23.               MigrateTask(task, node)
24.               Yield()
25.               break
26.            else
27.               Yield()
28.            end if
29.         end while
30.      end if
31.   end if
32.   lastdeadlinemisses ← DeadLineMisses()
33. end while
```

Figura 5.20 – Gerente de migração local

5.7 Toolchain

Para o desenvolvimento de aplicações e do sistema operacional foi construído um conjunto de ferramentas para o ambiente Linux, baseado na coleção de compiladores GCC versão 4.6.0. Os fontes do compilador foram modificados com o intuito de evitar a geração de instruções com acesso desalinhado à memória. As ferramentas tem como arquitetura alvo o conjunto de instruções MIPS I, e incluem:

- compilador cruzado (*mips-elf-gcc*);
- montador de linguagem de máquina (*mips-elf-as*);

Tabela 5.3 – API do sistema operacional Hellfire OS

Chamada de sistema	Formato	Descrição
OS_TaskParameters()	int32 OS_TaskParameters(uint8 id, uint16 *period, uint16 *capacity, uint16 *deadline, uint8 *locked);	Ler parâmetros
OS_ChangeTaskParameters()	int32 OS_ChangeTaskParameters(uint8 id, uint16 *period, uint16 *capacity, uint16 *deadline, uint16 *locked);	Modificar parâmetros
OS_TaskTicks()	uint32 OS_TaskTicks(uint8 id);	Número de *ticks* executados
OS_TaskDeadlineMisses()	uint32 OS_TaskDeadlineMisses(uint8 id);	Perdas de *deadline* da tarefa
OS_TaskCpuUsage()	uint32 OS_TaskCpuUsage(uint8 id);	Utilização de processador da tarefa
OS_TaskEnergyUsage()	uint32 OS_TaskEnergyUsage(uint8 id);	Utilização de energia da tarefa
OS_TaskMemoryUsage()	uint32 OS_TaskMemoryUsage(uint8 id);	Utilização de memória da tarefa
OS_LastTickTime()	uint32 OS_LastTickTime(uint8 id);	Tempo do último *tick* do sistema
OS_LastContextSwitchTime()	uint32 OS_LastContextSwitchTime(void);	Tempo da última troca de contexto
OS_CurrentTaskId()	uint8 OS_CurrentTaskId(void);	Identificação da tarefa corrente
OS_CurrentCpuId()	uint8 OS_CurrentCpuId(void);	Identificação do processador
OS_CurrentCpuFrequency()	uint32 OS_CurrentCpuFrequency(void);	Frequência de operação do processador
OS_NTasks()	uint8 OS_NTasks(void);	Número de tarefas
OS_TaskSetSuperPeriod()	uint32 OS_TaskSetSuperPeriod(void);	Super período do conjunto de tarefas
OS_TaskSetDeadlineMisses()	uint32 OS_TaskSetDeadlineMisses(void);	Total de perdas de *deadline*
OS_BlockTask()	int32 OS_BlockTask(uint8 id);	Bloquear tarefa
OS_ResumeTask()	int32 OS_ResumeTask(uint8 id);	Reiniciar tarefa
OS_KillTask()	int32 OS_KillTask(uint8 id);	Excluir tarefa
OS_AddTask()	int32 OS_AddTask(void (*task)(), int8 description[], uint32 stack_size, uint32 energy_t, uint8 locked);	Adicionar tarefa de melhor esforço
OS_AddPeriodicTask()	int32 OS_AddPeriodicTask(void (*task)(), uint16 period, uint16 capacity, uint16 deadline, int8 description[], uint32 stack_size, uint32 energy_t, uint8 locked);	Adicionar tarefa de tempo real
OS_Fork()	int32 OS_Fork(void);	Realizar cópia da tarefa corrente
OS_CpuUsage()	uint32 OS_CpuUsage(void);	Utilização total do processador
OS_EnergyUsage()	uint32 OS_EnergyUsage(void);	Utilização total de energia
OS_MemoryUsage()	uint32 OS_MemoryUsage(void);	Utilização total de memória
OS_FreeMemory()	uint32 OS_FreeMemory(void);	Memória livre
OS_TaskYield()	void OS_TaskYield(void);	Chamar por escalonamento voluntariamente
OS_Start()	void OS_Start(void);	Iniciar o sistema operacional
OS_EnterRegion()	void OS_EnterRegion(mutex *m);	Entrada em região crítica
OS_LeaveRegion()	void OS_LeaveRegion(mutex *m);	Saída de região crítica
OS_SemInit()	void OS_SemInit(semaphore *s, int32 value);	Inicializar semáforo
OS_SemWait()	void OS_SemWait(semaphore *s);	Espera em semáforo
OS_SemPost()	void OS_SemPost(semaphore *s);	Sinaliza semáforo
OS_Free()	void OS_Free(void *ptr);	Libera região de memória
OS_Malloc()	void *OS_Malloc(uint32 size);	Aloca região de memória
OS_Calloc()	void *OS_Calloc(uint32 qty, uint32 type_size);	Aloca e limpa região de memória
OS_Realloc()	void *OS_Realloc(void *ptr, uint32 size);	Realoca região de memória
OS_MboxInit()	void OS_MboxInit(mailbox *mbox, uint8 n_waiting_tasks);	Inicializa *mailbox*
OS_MboxSend()	void OS_MboxSend(mailbox *mbox, void *msg);	Envia referência de dados para uma *mailbox*
OS_MboxRecv()	void *OS_MboxRecv(mailbox *mbox);	Recebe referência de dados em uma *mailbox*
OS_MboxAccept()	void *OS_MboxAccept(mailbox *mbox);	Recebe referência de dados em uma *mailbox* (não bloqueante)
OS_MsgSend()	int32 OS_MsgSend(uint8 target_id, uint8 buf[], uint16 size);	Envia mensagem (apenas memória compartilhada)
OS_MsgRecv()	int32 OS_MsgRecv(uint8 *source_id, uint8 buf[], uint16 *size);	Recebe mensagem (apenas memória compartilhada)
OS_Send()	int32 OS_Send(uint16 target_cpu, uint8 target_id, uint8 buf[], uint16 size);	Envia mensagem
OS_Receive()	int32 OS_Receive(uint16 *source_cpu, uint8 *source_id, uint8 buf[], uint16 *size);	Recebe mensagem
OS_NB_Send()	int32 OS_NB_Send(uint16 target_cpu, uint8 target_id, uint8 buf[], uint16 size, uint32 timeout);	Envia mensagem (não bloqueante)
OS_NB_Receive()	int32 OS_NB_Receive(uint16 *source_cpu, uint8 *source_id, uint8 buf[], uint16 *size, uint32 timeout);	Recebe mensagem (não bloqueante)
OS_PacketsSent()	uint32 OS_PacketsSent(uint8 id);	Pacotes enviados
OS_PacketsReceived()	uint32 OS_PacketsReceived(uint8 id);	Pacotes recebidos
OS_PacketsQueued()	uint8 OS_PacketsQueued(uint8 id);	Pacotes na fila de recepção em *software*
OS_PacketsLost()	uint32 OS_PacketsLost(uint8 id);	Pacotes perdidos
OS_PacketsSeqError()	uint32 OS_PacketsSeqError(uint8 id);	Erros de sequenciamento de pacotes
OS_TaskMigrate()	int32 OS_TaskMigrate(uint8 source_id, uint16 target_cpu);	Migrar tarefa
OS_UniqueIDInit()	int32 OS_UniqueIDInit(unique_id *unique_id, uint16 target_cpu, uint8 target_id);	Inicializar identificação única de tarefa
OS_UniqueIDInfo()	int32 OS_UniqueIDInfo(unique_id *unique_id, uint16 *target_cpu, uint8 *target_id);	Retornar identificação única de tarefa
OS_UniqueIDSend()	int32 OS_UniqueIDSend(unique_id *uid, uint8 msg_buf[], uint16 size, uint32 timeout);	Enviar mensagem utilizando identificação única

- *linker* (*mips-elf-ld*);
- ferramentas para manipulação de binários (*mips-elf-objdump, mips-elf-readelf, mips-elf-objcopy*).

Em conjunto com as ferramentas GNU são utilizados *scripts* com o intuito de automatizar diversos processos, como compilação, ligação, manipulação de binários, geração *hexdumps* e configuração do sistema operacional.

Para a construção de uma imagem binária a ser carregada em cada elemento de processamento, são realizados alguns passos: (i) montagem, compilação e customização do sistema operacional; (ii) compilação da aplicação; (iii) criação de uma imagem ELF contendo a aplicação e

sistema operacional; (iv) criação de uma imagem binária final, utilizando ferramentas para a manipulação. A criação da imagem final é necessária para que a mesma possa ser diretamente carregada na memória de determinado nodo.

5.8 Arquitetura MPSoC

A arquitetura é composta por um conjunto homogêneo de elementos de processamento, que se comunicam por meio de uma rede utilizando chaveamento por pacotes. O conjunto composto por um elemento de processamento, filas, um roteador e lógica de controle para a interface de rede implementa uma unidade de processamento, ou nodo. O elemento de processamento contido em cada nodo possui uma memória local, e executa uma instância do sistema operacional Hellfire OS. Cada instância do sistema operacional, com o suporte adicional do *hardware*, possui os recursos para a execução de aplicações multiprocessadas distribuídas e de tempo real.

Os elementos de processamento implementam o conjunto de instruções MIPS I [57] e possuem um *pipeline* de dois estágios. Entre os recursos disponíveis em cada elemento estão uma memória local, um contador (*timer*), uma controladora de interrupções e uma interface de comunicação serial. Nenhum recurso avançado como gerenciamento de memória (MMU) ou *cache* foram implementados. O processador Plasma [76] foi modificado para incluir acesso e controle às filas dispostas entre ele e o roteador da rede e utilizado como elemento de processamento.

O roteador contido em cada nodo implementa o repasse de pacotes pela rede. Este roteador possui cinco portas de rede (quatro portas conectadas a outros roteadores e uma porta local), por onde o repasse de pacotes acontece segundo o algoritmo XY determinístico. No momento em que os pacotes chegam ao destino, estes são direcionados para a porta local, onde estão dispostas filas.

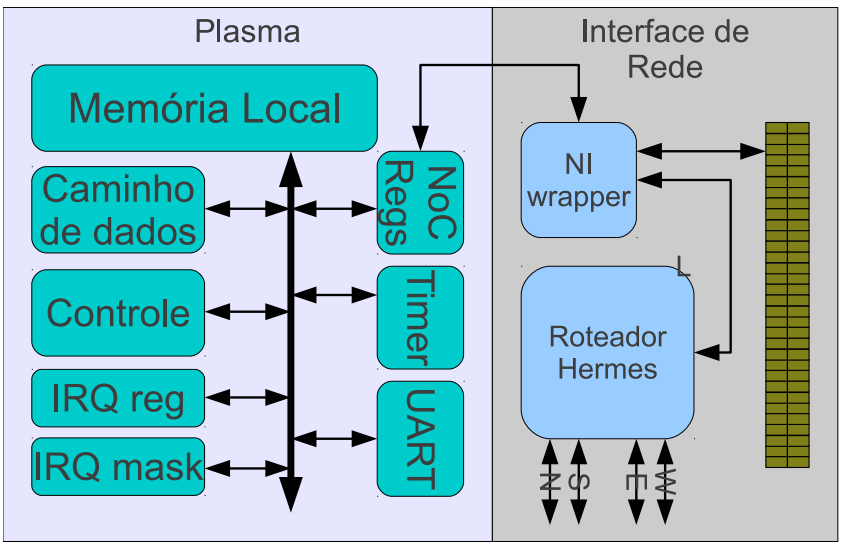


Figura 5.21 – Componentes de um nodo

A Figura 5.21 detalha os componentes que compõem um nodo. As filas são acessadas pelo elemento de processamento através de registradores mapeados em memória. Para ler os dados da fila de entrada, por exemplo, o *driver* de comunicação do sistema operacional realiza uma série de leituras sequenciais em um endereço específico. O *wrapper* implementado entre o elemento de processamento e as filas encarrega-se de efetuar a sinalização, e desta forma um *flit* pode ser copiado da fila em *hardware* para a memória local em dois ciclos [8]. No momento em que a fila de entrada de dados (sentido rede para o elemento de processamento) estiver cheia, uma interrupção é gerada pelo *wrapper*. O sistema operacional trata a interrupção e executa o *driver* de comunicação, que direciona o pacote para a fila em *software* da tarefa destino. No momento em que a fila de saída de dados (sentido elemento de processamento para a rede) estiver cheia, é gerado um sinal no *wrapper* e o pacote é encaminhado pela rede até que chegue à fila da interface de rede do nodo destino.

A arquitetura proposta foi implementada em três diferentes protótipos e uma ferramenta de simulação, a qual foi utilizada para a avaliação de resultados uma vez que não foi possível neste momento a construção de um protótipo com um grande número de nodos. No primeiro protótipo foi implementado um único nodo, com o intuito de validar o funcionamento básico da interface de rede. No segundo, quatro nodos em uma malha 2x2 foram utilizados para testes iniciais da interface de rede e *drivers* do sistema operacional. O último protótipo construído possui seis nodos em uma malha com topologia 3x2. Os protótipos foram utilizados para calibrar a ferramenta de simulação, previamente caracterizada. Nos nodos foram configurados roteadores Hermes [58] com filas internas de 16 *flits* e filas da interface de rede com 64 *flits* e cada elemento de processamento possui 32kB de memória local.

5.8.1 Ferramenta de Simulação

N-MIPS é uma ferramenta de simulação que oferece um ambiente MPSoC com um conjunto de até centenas de nodos interconectados em uma rede com topologia malha. Os elementos de processamento são implementados como simuladores do conjunto de instruções (ou ISSs) da arquitetura MIPS e a rede intra-chip como uma malha 2D organizada como um modelo de rede composto por roteadores Hermes. O tamanho da memória local dos elementos de processamento assim como o número de elementos e as dimensões da malha são configuráveis. Na atual implementação foi configurada uma memória de 1MB por nodo e dessa forma um grande número de tarefas pode executar em cada elemento de processamento, o que permite a verificação de cenários complexos de aplicação.

Esta ferramenta implementa um sistema MPSoC completo em linguagem C, e simula a execução de *software* utilizando precisão em nível de ciclo. Ao utilizar informações anotadas de *hardware* com relação à latências e potência, simulações rápidas de sistemas MPSoC complexos podem ser realizadas, mantendo no entanto informações detalhadas sobre desempenho, comportamento e

[8] Na arquitetura do elemento de processamento em questão, um acesso à memória é realizado por instruções *load* / *store*, e sua latência é de dois ciclos devido ao acesso à memória, que é compartilhada entre dados e instruções.

consumo de energia. As anotações do *hardware* foram feitas utilizando a metodologia apresentada em [31, 32]. Ambos trabalhos apresentam um modelo de classificação de instruções da arquitetura, que além de simplificar o modelo de energia e tornar possível simulações em alto nível de abstração, consegue manter precisão aceitável.

Para que a ferramenta representasse as reais características da arquitetura MPSoC proposta, a interface de rede e o roteador Hermes foram modelados e incluídos na ferramenta em [17] e [41]. Este roteador possui características como arbitragem rotativa, chaveamento por pacotes *wormhole*, armazenamento na entrada, controle de fluxo *handshake* e algoritmo de roteamento XY determinístico na implementação utilizada. Após a arbitragem, a qual possui um atraso em torno de sete ciclos por roteador conforme observado em simulações HDL e também caracterizado por Ost [68], a cada dois ciclos é repassado um *flit* de 16 *bits* por salto na rede com o controle de fluxo *handshake*. Após formado um caminho entre roteadores origem e destino, a comunicação ocorre na forma de *pipeline*. Dessa forma, é interessante que seja transferida uma quantidade significativa de dados a cada envio de pacote, de forma que as perdas geradas pelo tempo de arbitragem sejam amortizadas. A latência em ciclos para cada pacote é descrita como:

$$L_{packet} = (9 \times N_{hops}) + (2 \times N_{flits})$$

Congestionamentos acontecem quando uma determinada porta nos roteadores intermediários está no mesmo caminho de mais de uma transferência simultânea. Neste caso, a primeira transferência precisa terminar para que a porta fique livre para outra transferência. Este comportamento está implementado no simulador, assim uma boa aproximação da real latência do *hardware* em situações de congestionamento é representada.

```
#define NOC_COLUMN(core_n)      ((core_n) % NOC_WIDTH)
#define NOC_LINE(core_n)        ((core_n) / NOC_WIDTH)
#define CORE(core_n)        (0x0000|((NOC_COLUMN(core_n)<<4)|NOC_LINE(core_n)))
```

Figura 5.22 – Macros para o endereçamento de nodos

O sistema operacional utiliza endereçamento sequencial entre os elementos de processamento (nodo 0 a *n*). Dessa forma, torna-se necessário converter um número de elemento de processamento para o endereço do nodo na rede (16 bits). Em nível de aplicação, é utilizada a definição *CORE* que realiza esta conversão, levando em consideração as dimensões da rede. A implementação da definição é apresentada na Figura 5.22. O mesmo sistema de endereçamento é utilizado tanto na ferramenta de simulação quanto no protótipo, e segue o modelo de endereçamento de uma rede Hermes. O número máximo teórico de elementos de processamento suportados pelo simulador é de 256, e as dimensões limite para a malha de interconexão são de 16x16 nodos.

A posição e endereçamento de cada nodo em uma malha 4x4 é ilustrado na Figura 5.23. Neste sistema adotado, cada elemento de processamento possui uma posição fixa na rede de interconexão. Como os elementos de processamento são todos idênticos, o mapeamento das tarefas

é suficiente para explorar o espaço de projeto das aplicações. Em ambientes heterogêneos outras alternativas tornam-se possíveis, como o mapeamento dos elementos de processamento a posições arbitrárias da rede. Este tipo de mapeamento permite a otimização da aplicação em outro nível, no entanto este foge ao escopo do presente trabalho.

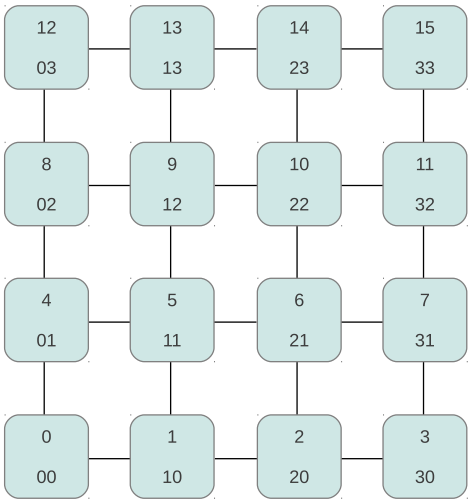


Figura 5.23 – Endereçamento de nodos na rede de interconexão

Além dos periféricos implementados no protótipo real, a ferramenta inclui um sistema de *log* de informações para cada elemento de processamento. O sistema de *log* é acessado pelo sistema operacional por endereços mapeados em memória. Assim, esta funcionalidade permite que o sistema operacional exporte informações de execução das aplicações em tempo real. A cada *tick*, informações detalhadas sobre cada tarefa são armazenadas em um arquivo, que pode ser posteriormente utilizado para realizar a extração de perfis de execução, geração de estatísticas de simulação e gráficos com o escalonamento das tarefas, volume de comunicação entre outros. Outras saídas geradas pelo simulador incluem os dispositivos de entrada e saída padrão como UARTs, terminais de alta velocidade e informações sobre as instruções executadas em cada ISS.

5.9 Considerações Finais

O modelo de aplicações proposto neste trabalho foi implementado em um RTOS e o modelo de arquitetura foi prototipado e adaptado a um simulador implementado anteriormente. Estas implementações têm como objetivo representar e validar os modelos propostos, uma vez que estes por serem originais não puderam ser retratados em sistemas já existentes. O sistema operacional é totalmente preemptivo, altamente configurável, possui bibliotecas padrão customizadas, temporizadores, semáforos, alocação dinâmica de memória, funcionalidades de depuração, suporte a diversas políticas de escalonamento, *drivers* de comunicação, mapeamento dinâmico de tarefas, migração de tarefas e serviços de gerência de migração distribuídos. A arquitetura foi inicialmente modelada em RTL e então caracterizada e implementada em um simulador compatível com o conjunto de

instruções MIPS I em nível de ciclo [31, 32]. Elementos de processamento, filas de comunicação em *hardware* e roteadores da rede de comunicação são todos emulados no simulador, e o número de processadores, tamanho das filas, assim como as dimensões da malha e tamanho das filas internas aos roteadores da rede são configuráveis.

6. RESULTADOS

A primeira parte deste Capítulo apresenta a avaliação de diversos componentes do sistema operacional, com o intuito de validar a implementação frente o modelo proposto. Na segunda parte são avaliadas aplicações sintéticas e reais, explorando as vantagens da abordagem de gerência de migração distribuída para conjuntos de tarefas que executam conforme o modelo. As métricas utilizadas referem-se ao *overhead* do sistema operacional, utilização dos elementos de processamento, número de migrações realizadas, tempo de estabilização do sistema e perdas de *deadline* da aplicação.

Em todos os testes realizados, foi considerada uma frequência de relógio de 25MHz (a não ser quando especificada outra frequência) e um tempo de *tick* de 10.48ms. A configuração das filas de entrada dos roteadores é de largura de 16 *bits* por *flit* com um tamanho de 16 *flits*, filas de entrada e saída da interface de rede entre os elementos de processamento e os roteadores com 64 *flits* cada e filas de *software* com capacidade para 32 pacotes para cada tarefa. Como definido anteriormente, o tamanho dos pacotes é mesmo das filas da interface de rede, sendo a configuração das filas de *hardware* a mesma para o sistema operacional e para a arquitetura. As dimensões das malhas utilizadas nos testes são 3x2, 4x4 e 6x5 nodos. As tarefas de controle de comunicação são aperiódicas e os gerentes de migração possuem os parâmetros de tempo real $\tau_i = \{id_i, p_i, e_i, d_i\}$ definidos como $\tau_2 = \{t2, 10, 1, 10\}$, ou seja, uma utilização (ou carga) de 10% de processamento e invocação destes a cada 104ms. O limite de utilização dos nodos foi definido de acordo com o valor conservador de 69% de utilização (segundo a política RM) e é apenas maior para casos onde uma única tarefa da aplicação é mantida em determinado nodo. Para a realização de medidas relacionadas a contagem de ciclos de relógio, foi utilizado um contador em *hardware* (emulado no simulador) acessível ao sistema operacional por um registrador mapeado em memória.

Tanto as aplicações sintéticas quanto as aplicações reais foram implementadas em linguagem C e descritas a partir de representações que utilizam o grafo ATG proposto. As aplicações iniciais utilizadas para extrair informações sobre o desempenho do sistema operacional com relação ao escalonamento de tarefas, primitivas de uso geral e trocas de mansagem não foram detalhadas por serem triviais.

Como plataforma de simulação foi utilizada a ferramenta N-MIPS descrita brevemente na Seção 5.8. Esta ferramenta possui recursos para a extração de informações de execução de aplicações e trabalha em conjunto com o sistema operacional para este fim. Os resultados gerados neste Capítulo foram extraídos diretamente desta ferramenta.

6.1 Desempenho do Sistema Operacional

6.1.1 Trocas de Contexto

O primeiro teste realizado refere-se ao desempenho do sistema operacional com relação às trocas de contexto. Para a geração dos diferentes cenários, foram utilizadas três frequências de operação distintas: 25MHz (protótipo e ferramenta de simulação), 100MHz (frequência de operação típica para sistemas com um grande número de elementos de processamento implementados em ASIC, por exemplo) e 500MHz (sistemas embarcados modernos e de alto desempenho).

Na Tabela 6.1 é apresentado o desempenho do sistema operacional com relação às trocas de contexto em diferentes situações. Foram criados cenários onde são executadas de 5 a 70 tarefas em um mesmo elemento de processamento. Nos testes realizados observou-se que os parâmetros das tarefas, assim como a variação do conjunto de tarefas entre periódicas e aperiódicas (tempo real, melhor esforço ou híbrido) não influencia no tempo de execução do escalonador do sistema operacional. Este comportamento deve-se ao fato do escalonamento ser realizado em dois níveis, ou seja, o algoritmo RM é executado e caso não existam tarefas de tempo real a serem executadas, o escalonador de melhor esforço é executado. Tarefas de qualquer tipo podem ser mapeadas ou excluídas a qualquer instante, e por isso os dois níveis de escalonamento são realizados. Independente das características do conjunto de tarefas, o tempo das trocas de contexto aumenta linearmente de acordo com o número de tarefas como pode ser observado nos resultados.

Tabela 6.1 – Tempo e *Overhead* das trocas de contexto em função do número de tarefas

Frequência		25MHz		100MHz		500MHz	
Número de Tarefas	Ciclos	Tempo	*Overhead*	Tempo	*Overhead*	Tempo	*Overhead*
5	1013	40us	0.38%	10us	0.09%	2us	0.01%
10	1563	62us	0.59%	16us	0.15%	3us	0.03%
15	2113	84us	0.80%	21us	0.20%	4us	0.04%
20	2663	106us	1.01%	27us	0.25%	5us	0.05%
25	3215	128us	1.22%	32us	0.30%	6us	0.06%
30	3765	150us	1.43%	37us	0.36%	7us	0.07%
35	4315	172us	1.64%	43us	0.41%	9us	0.08%
40	4865	194us	1.85%	49us	0.46%	10us	0.09%
45	5415	216us	2.06%	54us	0.51%	11us	0.10%
50	5965	238us	2.27%	60us	0.57%	12us	0.11%
55	6513	260us	2.48%	65us	0.62%	13us	0.12%
60	7063	282us	2.69%	71us	0.67%	14us	0.13%
65	7613	304us	2.90%	76us	0.72%	15us	0.14%
70	8165	326us	3.11%	82us	0.78%	16us	0.15%

A frequência de operação dos elementos de processamento influencia diretamente no *overhead*, uma vez que o tempo de *tick* é mantido constante. Considerando o pior caso de de-

sempenho, com uma frequência de operação de 25MHz e 70 tarefas em um nodo, o tempo de trocas de contexto é relativamente baixo e representa em torno de 3% do tempo total de execução. A 100MHz, um conjunto formado por 50 tarefas pode ser escalonado em torno de 60us, o que permite que o sistema operacional seja utilizado em aplicações de tempo real com um grande número de tarefas. Devido ao baixo *overhead*, para um número reduzido de tarefas é possível desconsiderar a influência do sistema operacional na execução das mesmas independente da frequência de operação utilizada. Na implementação, o tempo de escalonamento avança sobre o tempo de execução da tarefa rescém escalonada, fazendo com que esta tenha a execução do próximo *job* reduzida em exatamente o número de ciclos utilizado pelo sistema operacional para salvar os registradores na pilha da tarefa preemptada, executar o escalonador de tarefas e restaurar o contexto da tarefa corrente. Idealmente, no momento da ocorrência de uma interrupção de *timer* a próxima tarefa deveria começar a executar. Sabe-se que é impossível não existir um custo para o salvamento de contexto, escalonamento e restauração (em uma implementação puramente *software*), mas é possível reduzir ao máximo o tempo de execução do sistema operacional criando-se um escalonamento de tarefas próximo do ideal, que é o caso da atual implementação.

6.1.2 Primitivas de Uso Geral

Algumas primitivas e eventos do sistema operacional foram medidos com relação a sua latência. Assim como o tempo das trocas de contexto, o tempo de execução destas primitivas é considerado como parte do tempo de execução das tarefas que as invocam. A execução da rotina de interrupção da interface de rede ocorre no tempo de execução da tarefa corrente no elemento de processamento que recebe um pacote, sendo a melhoria do seu desempenho ou alternativas a este comportamento um trabalho futuro. Alternativas incluem postergação da interrupção até o término do *job* corrente, replicação de filas em *hardware* e outros mecanismos auxiliares como módulos DMA.

Na Tabela 6.2 são apresentados os valores medidos. O tamanho de pilha utilizado nas tarefas é de 2kB, sendo que este tamanho influencia no tempo de replicação das tarefas (é necessário realizar a cópia do contexto destas). São consideradas 10 tarefas nas medições, e este número influencia no tempo de trocas de contexto como observado anteriormente. A latência das primitivas para adicionar (*OS_AddTask()*, *OS_AddPeriodicTask*), bloquear (*OS_BlockTask()*), continuar a execução (*OS_ResumeTask()*) e excluir tarefas (*OS_KillTask()*) é constante e a primitiva de alocação e inicialização de memória *OS_Calloc()* é influenciada pela quantidade de memória requerida, sendo sua complexidade linear. A alocação de memória com a primitiva *OS_Malloc()* é pouco influenciada pela quantidade de memória requerida, uma vez que apenas ponteiros são manipulados nas estruturas de dados no alocador de memória implementado.

Em todos os casos analisados, uma única tarefa realiza chamadas às primitivas do sistema. Todas as primitivas que possuem algum tipo de dependência de dados são isoladas por exclusão mútua. Na arquitetura utilizada uma única memória é compartilhada entre dados e instruções, tendo os

Tabela 6.2 – *Overhead* de primitivas ou eventos

Syscall / Evento	Ciclos	Tempo
Troca de contexto (10 tarefas)	1563	62us
ISR da rede (64 *flits*)	1079	43us
OS_AddPeriodicTask()	2842	114us
OS_BlockTask()	88	<4us
OS_ResumeTask()	84	<4us
OS_Fork()	49107	1964us
OS_KillTask()	3087	123us
OS_ChangeTaskParameters()	122	5us
OS_CurrentTaskId()	9	<1us
OS_Malloc() (4kB)	754	30us
OS_Free() (4kB)	727	29us
OS_Calloc() (4kB)	21302	852us
OS_Realloc() (4kB para 8kB)	10760	430us

acessos à dados uma latência de 2 ciclos devido a esta característica. Os acessos à memória em uma alocação como no caso da primitiva *OS_Calloc()* são mais custosos que apenas a manipulação de algumas estruturas por esta razão. A primitiva *OS_Fork()* é relativamente custosa por razão similar. Internamente, esta primitiva executa uma chamada à *OS_AddTask()* ou *OS_AddPeriodicTask()* (dependendo do tipo de tarefa replicada). O maior custo ocorre no momento em que o contexto e a pilha da tarefa são copiados, sendo este o motivo de um maior tempo de execução da primitiva *OS_Fork()* frente outras. Na arquitetura dos nodos não existe uma unidade MMU, e torna-se impossível a implementação da técnica COW, restando a cópia total como alternativa, uma vez que a tarefa pai não pode ser bloqueada até o término da tarefa filha, nem ter seus dados modificados. Para o contexto de aplicação do sistema operacional proposto (aplicações embarcadas de tempo real) este modelo de cópia é válido, uma vez que o tamanho das tarefas é bastante reduzido, o que diminui a penalidade da primitiva *OS_Fork()*.

6.1.3 Primitivas de Comunicação

O desempenho das primitivas de comunicação medido refere-se a uma taxa de transferência constante obtida em nível de aplicação. Foi utilizado um controle de fluxo para evitar a saturação das filas em *software*, além de reduzir ao máximo a influência da execução da rotina de tratamento de interrupção da rede em tarefas não envolvidas com a comunicação. Desta forma, interrupções sucessivas em virtude de um grande número de mensagens ou mensagens formadas por uma grande quantidade de pacotes ocorrem dentro do tempo de execução da tarefa destino.

Para a realização dos testes, foram utilizados dois nodos vizinhos em uma malha 3x2 enviando-se 500 mensagens para a obtenção de cada ponto na Figura 6.1. O tamanho das mensagens é de 50 a 1000 bytes com intervalos de 50 bytes entre cada teste. O número de pacotes gerado

na rede foi de 500 a 4500 por teste, com um volume de dados entre 25000 e 500000 bytes. Em virtude do desempenho da rede de interconexão ser muito maior que as taxas alcançadas em nível de aplicação, a distância entre nodos não influenciou nos testes. Em redes com um grande número de nodos alguns dos enlaces são compartilhados em transferências entre múltiplos elementos de processamento, e as contenções geradas podem influenciar no desempenho de comunicação de acordo com a distância entre os nodos com relação à taxa de transferência e latência.

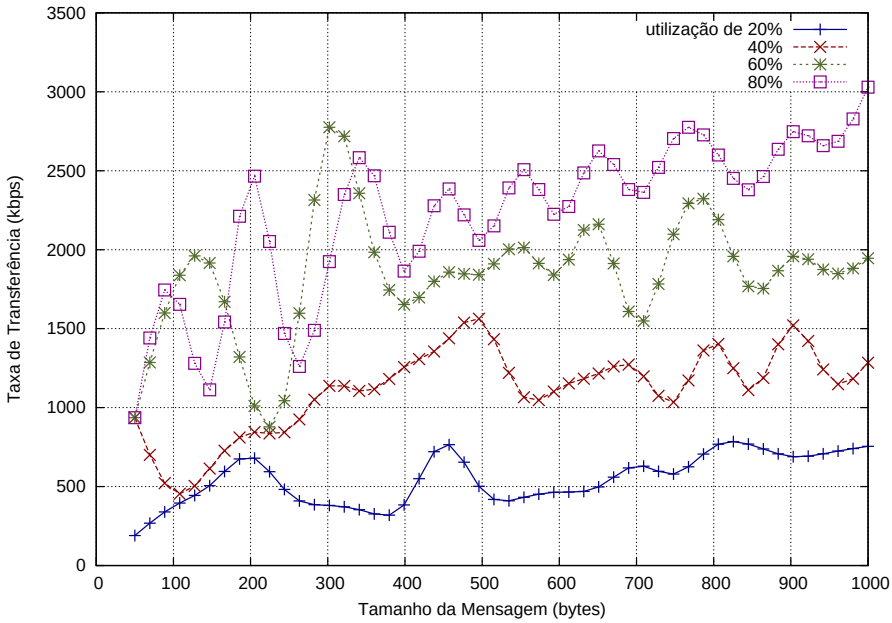


Figura 6.1: Desempenho das primitivas de comunicação em nível de aplicação como função da utilização de processador

A taxa de transferência obtida com o uso das primitivas de comunicação em nível de aplicação foi medida variando-se os parâmetros da tarefa de origem das mensagens. Foram criados cenários com utilização de processamento de 20%, 40%, 60% e 80%. Em todos os casos, a tarefa receptora da comunicação possui uma utilização de 80%. Nos testes realizados, é evidente a influência da utilização das tarefas no desempenho obtido pela primitiva de envio *OS_Send()*. Outro detalhe observado é a variação de desempenho em virtude do tamanho das mensagens para cada um dos cenários. Esta variação ocorre principalmente em virtude do *padding* realizado pela primitiva *OS_Send()* e remoção do mesmo na primitiva *OS_Receive()*. A taxa de transferência média obtida foi de 543 Kbps para 20% de utilização, 1120 Kbps para 40%, 1827 Kbps para 60% e 2212 Kbps para 80% de utilização. Sem controle de fluxo, as taxas de transferência dobram (não é necessária uma confirmação para cada mensagem) e a transferência de mensagens maiores que 1000 bytes também aumenta substancialmente o desempenho. Não foram utilizadas mensagens maiores devido ao tempo despendido para realizar a simulação em virtude do volume de dados e mensagens transferidas (500 mensagens por teste).

O desempenho de pico obtido (transferência de uma única mensagem de um nodo a outro, com ocupação total de processamento, sem controle de fluxo) é maior que os números apresentados, pois não é necessário que ocorra sincronização entre os *drivers* de envio e recebimento dos nodos

envolvidos na transferência. O desempenho de pico das primitivas de comunicação foi utilizado para projetar o tamanho das filas de *hardware* e *software*, sendo este apresentado na Seção 5.3. Utilizações entre 20% e 80% foram escolhidas em virtude da taxa mínima de transferência esperada e tempo de processamento livre para outras tarefas do sistema executarem normalmente.

6.1.4 Migração de Tarefas

O último aspecto analisado nesta Seção refere-se ao desempenho obtido pela primitiva *OS_Task Migrate()* em diferentes cenários, variando-se o tamanho da tarefa migrada e a carga do nodo destino. O tempo de migração inclui a cópia do contexto da tarefa a ser migrada, cópia dos segmentos de código e dados, tempo de restauração do contexto da tarefa no destino e resposta da operação, contendo a identificação da tarefa no nodo remoto ou uma falha.

Nos testes realizados, o tamanho total das tarefas migradas foi variado entre 1kB e 4kB e a influência da carga do nodo destino no tempo de migração foi avaliado para situações com 20% (carga baixa), 50% (carga média) e 80% (carga alta). Em todas as situações, os nodos destino possuem três tarefas de sistema em execução além de outras cinco de uma aplicação qualquer. Os parâmetros destas tarefas foram modificados para gerar as três situações de carga avaliadas. A invocação direta da primitiva de migração pela aplicação permite que uma tarefa de melhor esforço ou tempo real seja migrada para um nodo altamente carregado (acima do limite de utilização da política RM). No entanto, migrações realizadas pelos gerentes de migração evitam tais situações realizando a avaliação de carga antes de efeturem migrações. O caso de alta carga foi avaliado para demonstrar uma situação que excede os limites de carga de acordo com o modelo proposto.

Na Tabela 6.3 são apresentados os cenários de teste. Como pode ser observado, o tempo de migração aumenta linearmente de acordo com o tamanho das tarefas. Em situações onde nodos destino estão pouco carregados (20%), a migração de uma tarefa de 1kB é realizada em torno de 3ms. Uma tarefa de 4kB é migrada na mesma situação de carga em 9.5ms, ou seja, em um tempo inferior a um *tick* do sistema. Neste caso é possível realizar a migração de tarefas sem chance de haver perdas de *deadline*, pois o processo ocorre atomicamente dentro do *tick* da tarefa que invocou a migração ou do gerente de migração. A taxa de transferência de dados ao nodo destino no último caso apresentado ocorre a 3367 Kbps (média em virtude do protocolo de migração), uma vez que o destino possui memória em filas com o tamanho suficiente para receber toda a tarefa, evitando um controle de fluxo e transferindo uma quantidade de informação significativa em um curto espaço de tempo.

Situações intermediárias de carga (50%) podem existir em diversos elementos de processamento em sistemas com um grande número de tarefas. Ainda nesta situação de carga, nos testes foram realizadas migrações com tempos entre 17.49ms e 19.17ms. Comparando-se estes casos a nodos com pouca utilização, o tempo de migração basicamente é o dobro do pior dos casos avaliados

Tabela 6.3: Tempos de migração em função do tamanho da tarefa (código + dados) e carga do nodo destino

Carga	20%		50%		80%	
Tamanho da Tarefa	Ciclos	Tempo	Ciclos	Tempo	Ciclos	Tempo
1kB	84002	3.36ms	437398	17.49ms	700037	28.00ms
1.5kB	111307	4.45ms	444183	17.76ms	706993	28.27ms
2kB	135122	5.40ms	450955	18.04ms	713879	28.55ms
2.5kB	162628	6.50ms	457683	18.31ms	720835	28.83ms
3kB	186443	7.46ms	464512	18.58ms	727721	29.10ms
3.5kB	210315	8.41ms	472491	18.90ms	734664	29.38ms
4kB	237563	9.50ms	479447	19.18ms	741620	29.66ms

anteriormente. Este tempo de migração, no entanto, é pouco influenciado pelo tamanho das tarefas migradas em proporção ao tempo total da migração.

Nodos destino altamente carregados (80%) fazem com que o processo de migração se torne mais lento que em outras situações, com tempos variando entre 28ms e 29.66ms. Este atraso deve-se principalmente a redução de prioridade da tarefa de controle de comunicação. Com 80% de carga, além da carga da tarefa que está sendo migrada, existe uma menor probabilidade da tarefa de controle de comunicação ser executada, e dessa forma o processo é postergado e a tarefa a ser migrada é mantida bloqueada, o que pode fazer com que esta perca *deadlines*.

O processo de migração ocorre dentro do tempo de execução da tarefa que invoca a primitiva. No entanto, o processo pode exceder o tempo de execução desta tarefa, sendo continuado apenas no próximo período. Uma tarefa pode realizar uma migração que leve mais de um *tick* sem que ocorram perdas de *deadline* em virtude da migração se esta possuir parâmetros de execução suficientes (período e capacidade) e a tarefa que está sendo migrada não possuir uma prioridade maior que esta, de acordo com a política RM. Perdas de *deadline* em tarefas que estão sendo migradas não podem ser facilmente calculadas com antecedência, pois são dependentes dos parâmetros do conjunto de tarefas em execução.

6.2 Gerentes de Migração em Aplicações Sintéticas

Nesta Seção serão apresentados quatro cenários que demonstram o comportamento do mecanismo de gerência de migração de tarefas distribuído frente um gerente centralizado. Em todos os casos foi mantida uma carga inicial idêntica em ambas abordagens para a obtenção dos resultados. Com a abordagem distribuída, cada nodo possui um gerente de migração independente que mantém apenas a informação de carga local, e com a abordagem centralizada um dos nodos possui um gerente de migração centralizado e os outros possuem uma tarefa escrava que atende às solicitações de migração deste gerente. O gerente de migração é posicionado em um nodo próximo ao centro da malha em todos os casos, e corresponde aos nodos 1 (malha 3x2), 5 (malha 4x4) e 14 (malha 6x5). Dessa forma, os gerentes locais comunicam-se com o centralizado informando a carga

inicial do nodo (para que o gerente mantenha esta informação localmente) no início da execução e enviam mensagens a este a cada mudança de estado de carga (aumento ou diminuição da ocupação, número de tarefas e memória disponível). O gerente centralizado envia mensagens de migração após a tomada de decisão às tarefas escravas, que realizam a migração da tarefa escolhida pelo gerente para um nodo destino também escolhido pelo gerente.

A implementação do gerente centralizado leva em consideração o mesmo critério de seleção do alvo de migração dos gerentes distribuídos, ou seja, o algoritmo de busca por espalhamento a partir do ponto de sobrecarga. A diferença é que nem o gerente nem nodos afetados por uma sobrecarga enviam mensagens de solicitação aos nodos vizinhos (como é o caso no uso de gerentes distribuídos), pois a informação é mantida no gerente sempre atualizada. Devido a diferenças no perfil de execução das tarefas em nodos afetados, a ordem das migrações pode divergir entre as duas abordagens, assim como as tarefas escolhidas, os destinos e número de migrações. Por possuir uma visão global do sistema, o gerente centralizado possui a possibilidade de realizar migrações utilizando diferentes critérios, como selecionar regiões menos populadas da malha ou outros. Este tipo de critério não foi utilizado, pois a idéia do algoritmo de espalhamento é justamente manter próximas as tarefas originalmente mapeadas em um mesmo nodo, evitando a fragmentação da aplicação e consequentemente reduzindo a ocupação de enlaces no caminho de comunicação.

Nos testes realizados, foram criadas situações onde múltiplos nodos encontram-se sobrecarregados com o objetivo de estressar a eficiência das duas abordagens. Além disso, os algoritmos de gerência de migração apenas são ativados a partir do *tick* 100, para que todas as tarefas das aplicações exemplo executem pelo menos uma vez e dessa forma os gerentes tenham a opção de escolher qualquer uma do particionamento como candidata a migração[1]. Nestes cenários, existem diversos elementos de processamento em sobrecarga e que possuem tarefas que estão perdendo *deadlines*. Em todos os testes realizados, as aplicações foram simuladas por 5000ms (5 segundos). Todas as tarefas foram configuradas com um tamanho de pilha de 4kB, e devido à simplicidade de implementação destas (modelos de tarefa), o tamanho do código é bastante reduzido, variando entre 50 e 300 bytes. A medida de tempo utilizada nos exemplos é *ticks*, uma vez que os algoritmos de gerência executam e sincronizam com base em parâmetros de tempo real que utilizam esta unidade. Dessa forma, ao reduzir-se o tempo de *tick*, reduz-se o tempo de execução dos algoritmos. Vale ressaltar que as trocas de mensagens entre gerentes de migração e suas decisões ocorrem dentro do tempo de execução destes, e desta forma a execução das tarefas que compõem a aplicação não é afetada.

[1] As tarefas candidatas a migração devem ser configuradas na aplicação como não fixas. Além disso, para que possam ser migradas seu contexto precisa ser inicializado, o que acontece após a execução do primeiro *job* (a tarefa encontra-se no estado *pronto*).

6.2.1 Aplicação 1, MPSoC 3x2

A primeira aplicação sintética é apresentada na Figura 6.2, a qual possui suas tarefas iniciais mapeadas em um MPSoC com dimensões 3x2, juntamente com tarefas de controle de comunicação e gerentes de migração. O número inicial de tarefas da aplicação é 18 e os nodos 2 e 3 encontram-se em sobrecarga. Os parâmetros $\tau_i = \{p_i, e_i, d_i\}$ das tarefas que encontram-se nestes nodos definem uma carga de 89% e 94% respectivamente[2]. Em tempo de execução (*tick* 50) é mapeada mais uma tarefa no nodo 0, aumentando sua carga para 76%.

Esta aplicação possui um alto grau de comunicação e as tarefas mapeadas em cada nodo possuem parâmetros pouco semelhantes. No exemplo, é ilustrada a situação de sobrecarga em uma malha de pequenas dimensões onde uma aplicação complexa está executando e inicialmente dois elementos de processamento encontram-se em sobrecarga. Após o mapeamento de mais uma tarefa no nodo 0, são três os elementos de processamento em sobrecarga.

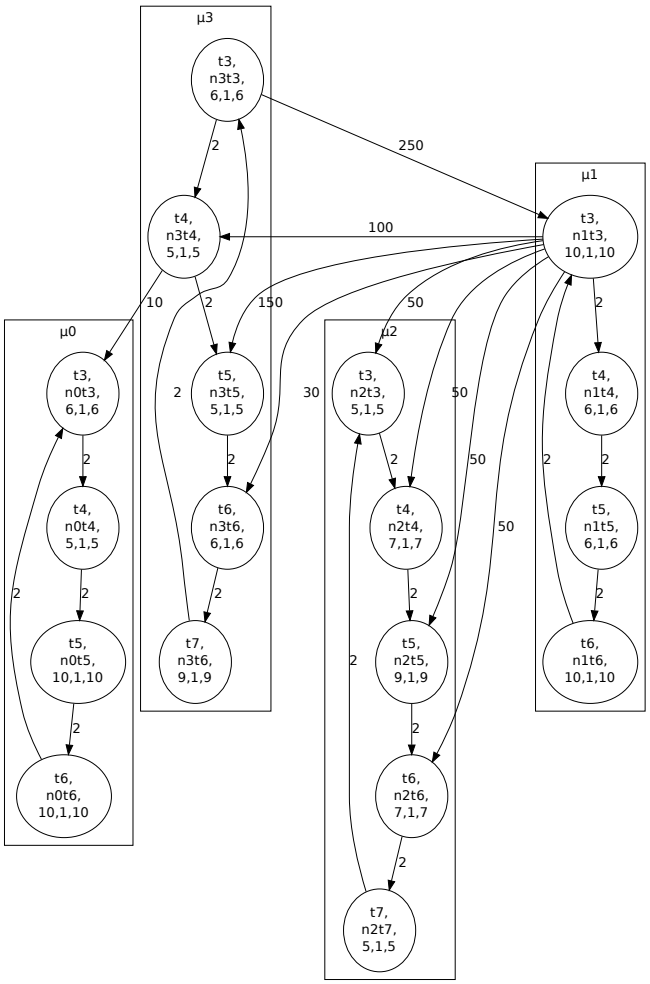


Figura 6.2 – ATG das tarefas iniciais da aplicação sintética 1

[2]O cálculo de carga é apresentado na Seção 4.10. As tarefas possuem uma utilização definida pela divisão de seus parâmetros de ocupação e_i por seus períodos p_i, e a carga do nodo é definida pela soma das utilizações de todas as tarefas da aplicação e do sistema operacional. Para o nodo 2 a utilização é definida como $u_2 = \frac{1}{10}+\frac{1}{5}+\frac{1}{7}+\frac{1}{9}+\frac{1}{7}+\frac{1}{5} \approx 0.89$ e para o nodo 3 como $u_3 = \frac{1}{10}+\frac{1}{6}+\frac{1}{5}+\frac{1}{5}+\frac{1}{6}+\frac{1}{9} \approx 0.94$.

Na Figura 6.3 é apresentado o perfil de carga da aplicação em duas situações distintas, onde um gerente de migração centralizado é responsável por estabilizar a carga (Figura 6.3(a)) e outra onde gerentes distribuídos são utilizados (Figura 6.3(b)). Como pode ser observado, os gerentes distribuídos conseguem estabilizar a carga dos nodos em menor tempo. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre o *tick* 100 e 275. O algoritmo de gerência troca mensagens, toma decisões e realiza 5 migrações em 175 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre o *tick* 100 e 190, onde 5 migrações são realizadas em 90 *ticks*.

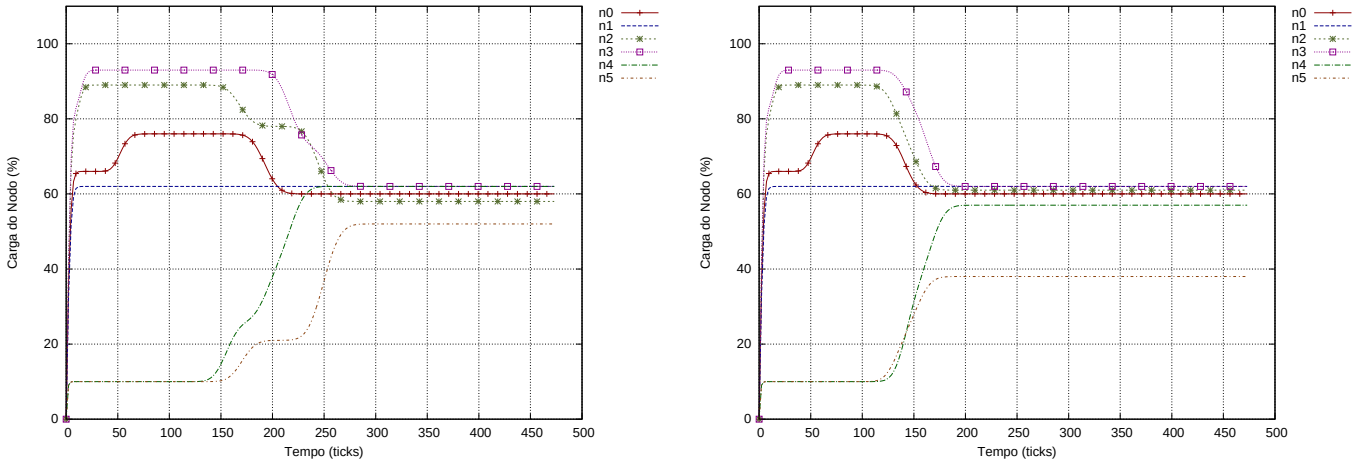


Figura 6.3: Aplicação sintética 1, utilização dos elementos de processamento em um MPSoC 3x2: gerente centralizado (a) e gerentes distribuídos (b)

Após as migrações as cargas dos nodos afetados 2, 3 e 0 são de 58%, 62% e 60% respectivamente no primeiro caso (gerente centralizado) e 61%, 62% e 60% no segundo caso. Os nodos 4 e 5 recebem as tarefas dos nodos afetados, sendo sua carga final 62% e 52% no primeiro caso e 57% e 38% no segundo.

A Figura 6.4 apresenta o mapeamento inicial (Figura 6.4(a)) e após (Figura 6.4(b)) a estabilização do sistema com gerentes distribuídos. Tarefas de diferentes nodos são identificadas por cores. Como pode ser observado, o nodo 2 (roxo) tem suas tarefas t_6 e t_4 migradas para o nodo 5, que encontra-se a um salto de distância na rede. O nodo 3 (amarelo) tem suas tarefas t_7 e t_4 migradas para o nodo 4 que também está posicionado a um salto de distância. Nenhum vizinho a um salto de distância (nodo 1 ou 3) apto a receber tarefas é encontrado pelo gerente de migração do nodo 0 (verde), o que faz com que a busca por espalhamento continue. A dois saltos, o nodo 4 possui tempo de processamento livre para receber e executar a tarefa t_3.

O perfil de tráfego gerado pela aplicação, tarefa de controle de comunicação, gerentes de migração e migrações de tarefas é apresentado na Figura 6.5. Para cada nodo é representado o volume total de dados, sendo este volume composto pelo total dos volumes gerados por todas as tarefas (referenciado como v_m no modelo). Este total inclui não só o volume de dados enviado a

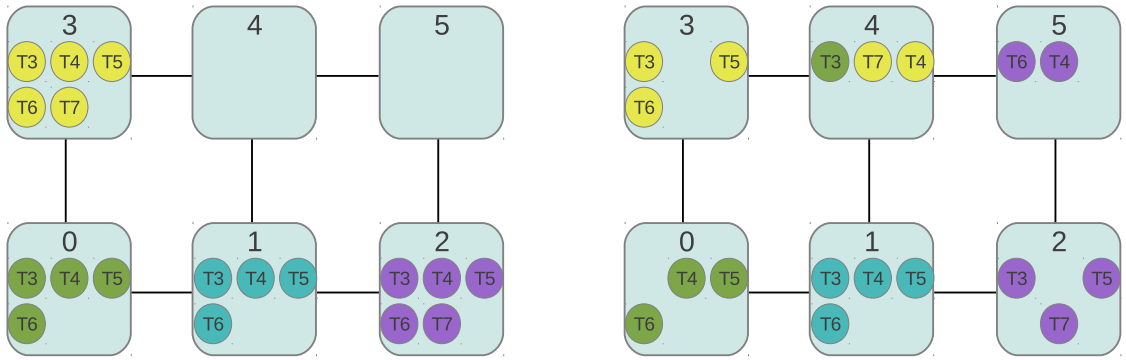


Figura 6.4: Aplicação sintética 1, gerentes distribuídos: mapeamento inicial até o *tick* 100 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

outros nodos, mas também o volume de dados trocado entre tarefas em um mesmo nodo. Maiores detalhes sobre a composição do tráfego gerado por cada nodo são apresentados na Seção 4.4.

As transferências que geram os maiores picos representam as migrações de tarefa. Comparando-se a Figura 6.5(a) e a Figura 6.5(b) é possível observar que as transferências relacionadas a migrações acontecem aproximadamente entre os *ticks* 160 e 260 no primeiro caso e em um espaço de tempo menor, aproximadamente entre os *ticks* 130 e 165 (estão mais próximas umas das outras) no segundo caso com gerentes distribuídos. Além disso, no segundo caso é observada a ocorrência de duas migrações em paralelo, representada pela sobreposição de tranferências na representação utilizada (*tick* 140, Figura 6.5(b)).

Nem todas as migrações são concretizadas em todos os casos, pois a modificação do perfil de carga do nodo destino, assim como trocas de mensagem entre o gerente centralizado e tarefas escravas durante uma migração pode fazer com que esta não seja possível em determinado instante (Figura 6.5(a), vê-se 6 transferências grandes, e apenas 5 migrações efetivadas). Este tipo de situação pode ocorrer quando existe um número significativo de nodos (para as dimensões do MPSoC) em sobrecarga exatamente no mesmo instante de tempo. Para evitar situações de sobrecarga adicionais, o processo de migração é cancelado e realizado posteriormente. O tráfego gerado pelas tarefas que compõem a aplicação neste exemplo varia entre 100 Kbps e 700 Kbps aproximadamente e atinge mais de 2000 Kbps em alguns momentos durante migrações.

Um resumo dos experimentos relacionados à primeira aplicação é apresentado na Tabela 6.4. Caso fosse mantida a situação inicial (nodos sobrecarregados e sem gerência de migração), ao término do tempo de execução seriam perdidos 16 *deadlines*. Com o mecanismo de gerência centralizado, as perdas de *deadline* são reduzidas para 8, e são realizadas 5 migrações. Com gerentes distribuídos, as perdas de *deadline* são reduzidas para 5, uma melhoria de 37.50% em comparação ao mecanismo centralizado e são realizadas 5 migrações. O tempo de estabilização do sistema com gerentes distribuídos apresenta uma melhoria de 48.57% para esta aplicação.

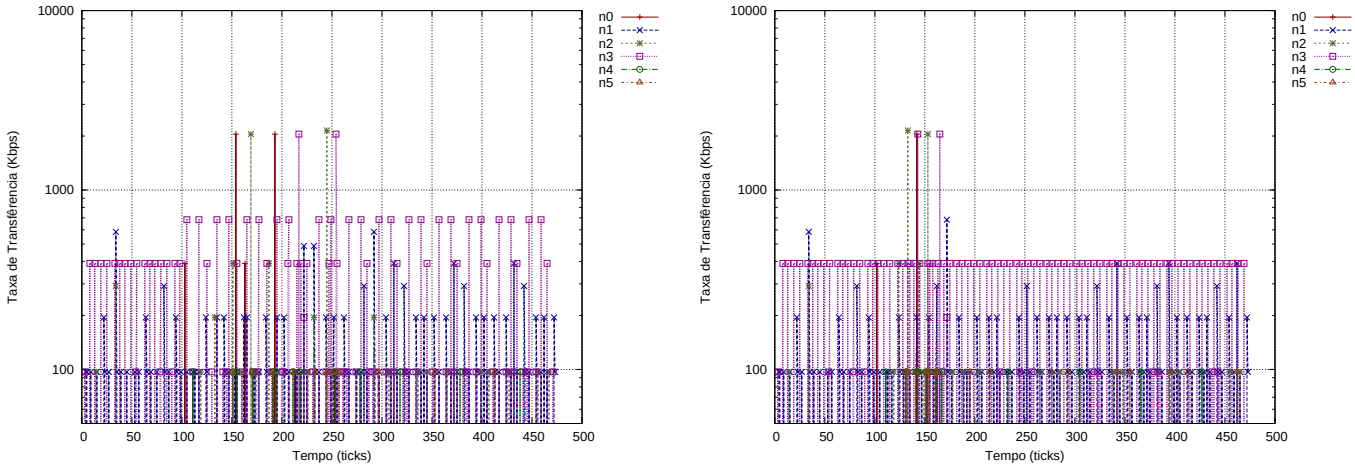


Figura 6.5: Aplicação sintética 1, perfil de tráfego gerado em um MPSoC 3x2: gerente centralizado (a) e gerentes distribuídos (b)

Tabela 6.4 – Resumo dos experimentos, Aplicação 1

Gerência	Migrações	Tempo de estabilização	Perdas de *deadline*
Sem gerência	0	-	16
Centralizada	5	175	8
Distribuída	5	90	5
Melhoria		48.57%	37.50%

6.2.2 Aplicação 2, MPSoC 4x4

A segunda aplicação sintética é apresentada na Figura 6.6. Esta aplicação possui suas tarefas iniciais mapeadas em um MPSoC com dimensões 4x4, juntamente com tarefas de controle de comunicação e gerentes de migração. O número de tarefas inicial da aplicação é 36 e os nodos 2, 3, 10 e 11 encontram-se em sobrecarga, com 89%, 94%, 89% e 95% de carga respectivamente. Em tempo de execução (*tick* 60) são mapeadas mais duas tarefas, sendo uma no nodo 0 e outra no nodo 8, aumentando carga destes para 76%.

Esta aplicação, assim como a anterior, possui um alto grau de comunicação e como característica as tarefas mapeadas em cada nodo possuem parâmetros pouco semelhantes. No exemplo é ilustrada a situação de sobrecarga em uma malha de dimensões médias onde uma aplicação complexa está executando e quatro elementos de processamento encontram-se em situação de sobrecarga inicialmente. Após o mapeamento dinâmico de tarefas nos nodos 0 e 8, são seis os elementos de processamento em sobrecarga.

Na Figura 6.7 é apresentado o perfil de carga da aplicação no MPSoC em duas situações distintas, onde um gerente de migração centralizado é responsável por estabilizar a carga (Figura 6.7(a)) e outra onde gerentes distribuídos são utilizados (Figura 6.7(b)). É observado que os

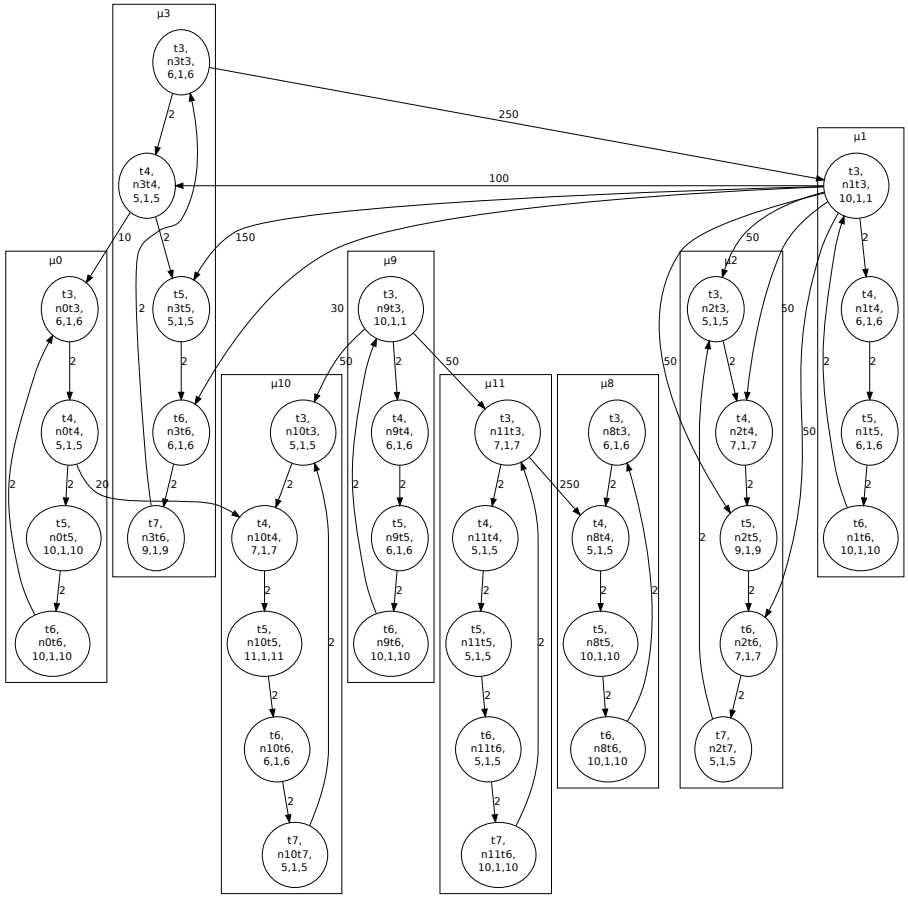
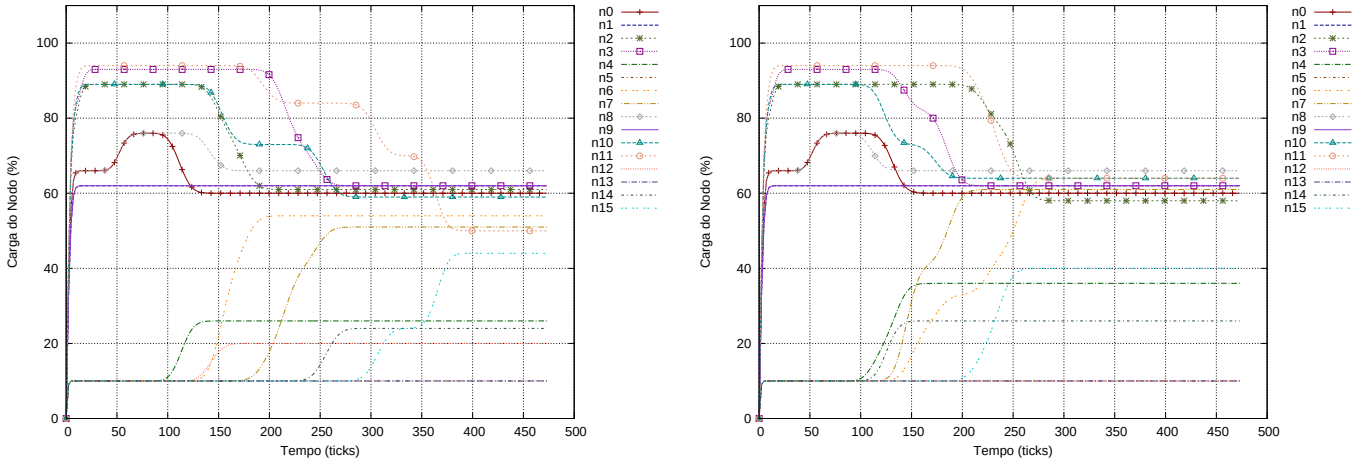


Figura 6.6 – ATG das tarefas iniciais da aplicação sintética 2

gerentes distribuídos conseguem estabilizar a carga dos nodos em menor tempo novamente. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 375, e assim o algoritmo de gerência troca mensagens, toma decisões e realiza 11 migrações em 275 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 260, onde 10 migrações são realizadas em 160 *ticks*.

As cargas dos nodos afetados 2, 3, 10, 11, 0 e 8 são de 62%, 63%, 58%, 45%, 60% e 66% após as migrações realizadas pelo gerente centralizado (Figura 6.7(a)). Os nodos 4, 6, 7, 12, 14 e 15 recebem tarefas dos nodos afetados, tendo a carga final de 28%, 55%, 50%, 20%, 23% e 43% respectivamente. Gerentes distribuídos (Figura 6.7(b)) realizam a estabilização da carga dos nodos afetados deixando-os com 58%, 62%, 64%, 64%, 60% e 66% de utilização respectivamente. Os nodos 4, 6, 7, 14 e 15 recebem as tarefas migradas neste caso, ficando com 37%, 64%, 60%, 25% e 40% de carga respectivamente.

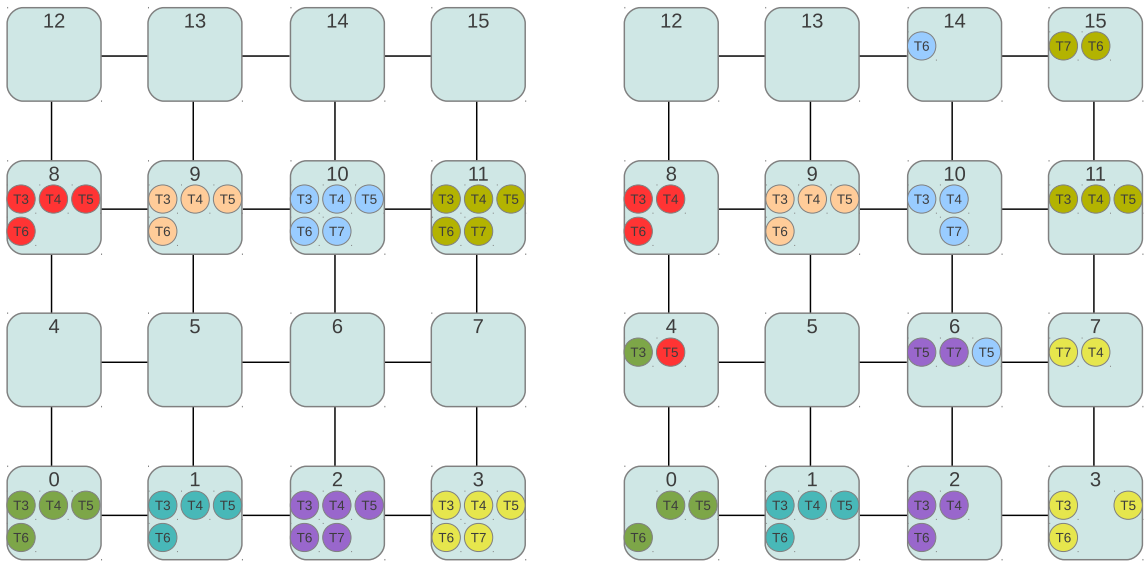
A Figura 6.8 apresenta o mapeamento inicial (Figura 6.8(a)) e após a estabilização do sistema pelos gerentes distribuídos (Figura 6.8(b)). Como pode ser observado, o nodo 2 (roxo) tem suas tarefas t_5 e t_7 migradas para o nodo 6, que encontra-se a um salto de distância na rede. O nodo 3 (amarelo) tem suas tarefas t_7 e t_4 migradas para o nodo 7. O nodo 10 (azul) tem sua tarefa t_6 migrada para o nodo 14 e t_5 para o nodo 6. As tarefas t_7 e t_6 do nodo 11 (oliva) são migradas



(a) (b)

Figura 6.7: Aplicação sintética 2, utilização dos elementos de processamento em um MPSoC 4x4: gerente centralizado (a) e gerentes distribuídos (b)

para o nodo 15 e as tarefas t_3 do nodo 0 (verde) e t_5 do nodo 8 (vermelho) são migradas para o nodo 4, que encontra-se a um salto de distância de ambos.



(a) (b)

Figura 6.8: Aplicação sintética 2, gerentes distribuídos: mapeamento inicial até o *tick* 100 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

O perfil de tráfego gerado na rede de interconexão é apresentado na Figura 6.9. Novamente, as transferências que geram os maiores picos representam as migrações de tarefa. Comparando-se as Figuras 6.9(a) e 6.9(b) é possível observar que as transferências relacionadas a migrações acontecem em um espaço de tempo menor com gerentes distribuídos. Estas transferências acontecem entre os *ticks* 115 e 365 com o mecanismo de gerência centralizado e entre os *ticks* 105 e 255 com o mecanismo de gerência distribuído. O tráfego gerado pelas tarefas que compõem a aplicação

varia entre 100 Kbps e 600 Kbps aproximadamente e atinge mais de 2000 Kbps durante migrações de tarefas.

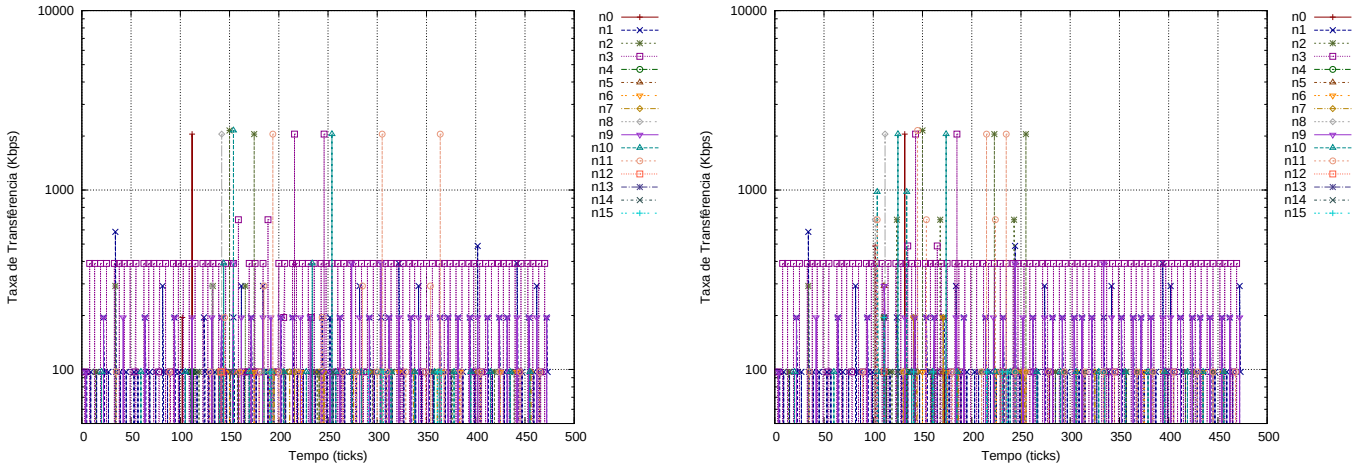


Figura 6.9: Aplicação sintética 2, perfil de tráfego gerado em um MPSoC 4x4: gerente centralizado (a) e gerentes distribuídos (b)

A Tabela 6.5 apresenta um resumo dos experimentos relacionados à segunda aplicação. Sem gerência de migração, ao término do tempo de execução seriam perdidos 68 *deadlines* no total. Com o mecanismo de gerência centralizado, as perdas de *deadline* são reduzidas para 14, e 11 migrações são realizadas. Com gerentes distribuídos, as perdas de *deadline* são reduzidas para 6, o que representa uma melhoria de 57.14% em comparação ao mecanismo centralizado sendo realizadas 10 migrações. Nesta aplicação, a estabilização do sistema ocorre com uma redução significativa de tempo ao serem utilizados gerentes distribuídos, com uma melhoria de 41.81%.

Tabela 6.5 – Resumo dos experimentos, Aplicação 2

Gerência	Migrações	Tempo de estabilização	Perdas de *deadline*
Sem gerência	0	-	68
Centralizada	11	275	14
Distribuída	10	160	6
Melhoria		41.81%	57.14%

6.2.3 Aplicação 3, MPSoCs 3x2, 4x4 e 6x5

A terceira aplicação sintética é apresentada na Figura 6.10. Para este experimento uma abordagem diferente foi utilizada, e consiste em replicar instâncias da mesma aplicação em MPSoCs com dimensões 3x2, 4x4 e 6x5. Para estes MPSoCs são alocadas 3, 4 e 7 instâncias desta mesma aplicação respectivamente, ou seja 12, 16 e 28 tarefas. Este experimento ilustra situações onde

diversas aplicações independentes executam em um mesmo ambiente. No exemplo, cada instância da aplicação é mapeada em um único elemento de processamento sendo sua carga total 100%, composta pela aplicação (90%) e o gerente de migração (gerente distribuído, gerente centralizado ou tarefa de migração).

O exemplo aqui ilustrado apresenta uma situação prática de aplicação do modelo proposto, onde apenas é definido um elemento de processamento onde é realizado o mapeamento de todas as tarefas iniciais de uma aplicação. Por exemplo, um usuário inicia um vídeo em uma plataforma multiprocessada hipotética e o decodificador (representado pela aplicação) possui tarefas que são alocadas a um único elemento de processamento inicialmente. Devido às necessidades de processamento da aplicação, algumas destas tarefas devem ser realocadas a outros elementos de processamento para que a qualidade de execução esperada seja alcançada (são eliminadas as perdas de *deadline*, e por consequência, eliminam-se as perdas de *frame*).

As instâncias de cada aplicação não possuem dependências entre si, e dessa forma o mecanismo de gerência pode explorar o conceito de localidade, que se adapta naturalmente ao modelo de gerentes distribuídos. Para cada instância, são definidas 4 tarefas, onde as tarefas t_4 e t_5 recebem dados e dessa forma dependem da tarefa t_3 e a tarefa t_6 recebe dados de t_4 e t_5. Para esta aplicação, são definidas 4 comunicações, com volumes de dados que variam entre 64 e 128 bytes.

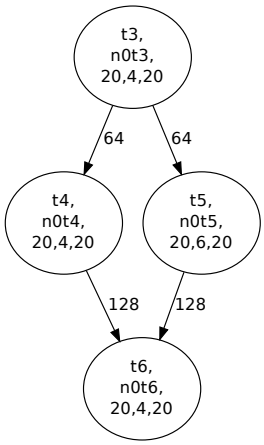


Figura 6.10 – ATG das tarefas iniciais da aplicação sintética 3

Aplicação 3, MPSoC 3x2

A primeira versão da terceira aplicação utiliza uma malha com dimensões 3x2, onde os nodos 0, 3 e 5 possuem uma instância da aplicação cada e encontram-se em sobrecarga. Na Figura 6.11 são apresentados os perfis de carga ao longo do tempo onde em um caso é utilizado um gerente centralizado (Figura 6.11(a)) e em outro gerentes distribuídos (Figura 6.11(b)). Comparado às aplicações anteriores, é evidente uma redução nos tempos de estabilização da carga nas duas abordagens de gerência. Esta redução deve-se ao fato de uma menor complexidade da aplicação utilizada e menor dependência de comunicação entre tarefas, o que simplifica as decisões dos gerentes.

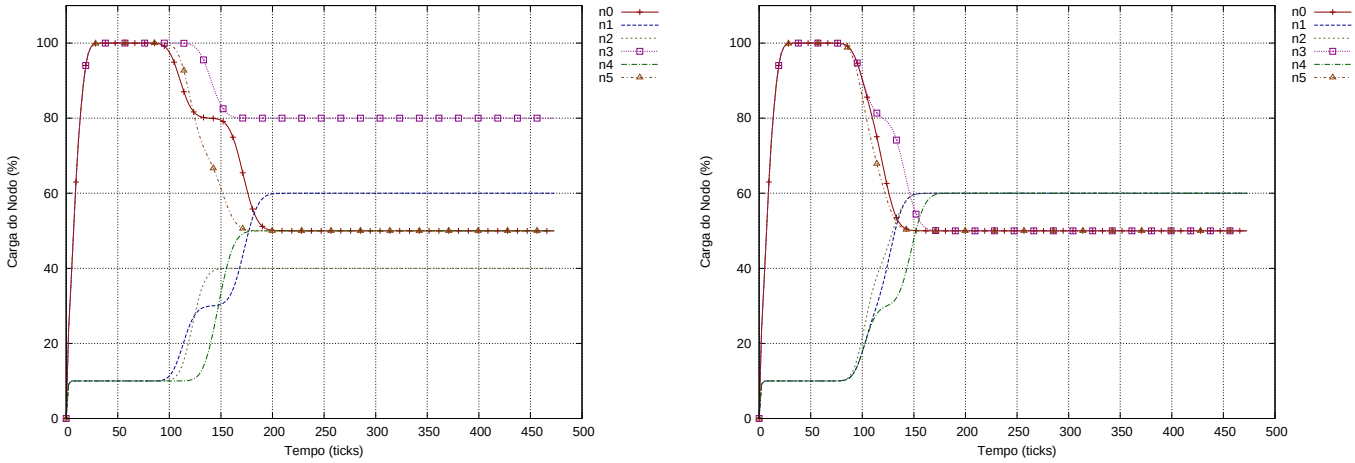


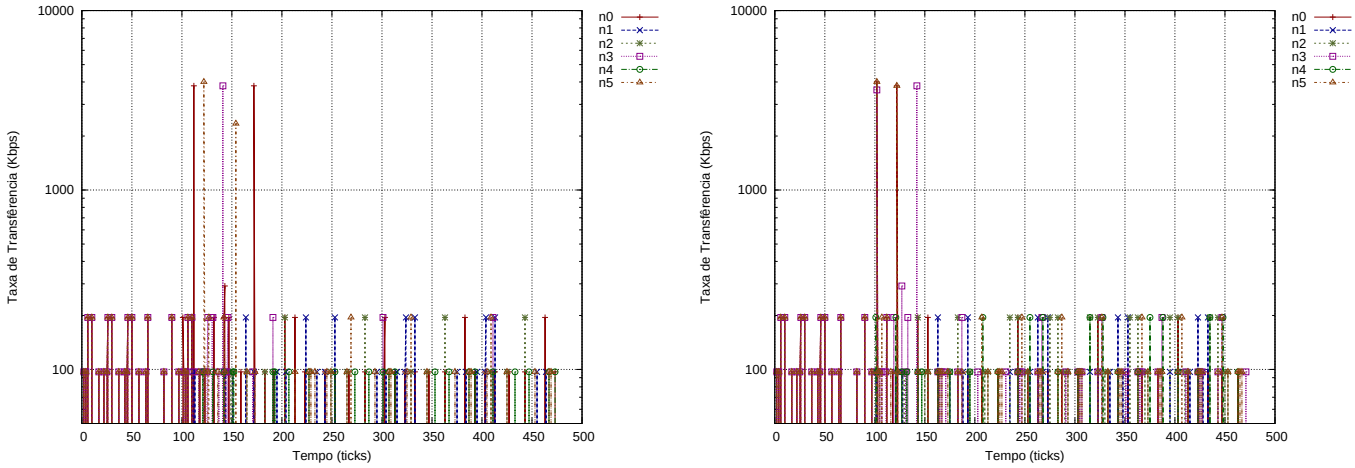
Figura 6.11: Aplicação sintética 3, utilização dos elementos de processamento em um MPSoC 3x2: gerente centralizado (a) e gerentes distribuídos (b)

Observa-se que os gerentes distribuídos conseguem estabilizar a carga dos nodos em menor tempo. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 200, ou seja em 100 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 175, ou seja em 75 *ticks*. No primeiro caso, são realizadas 5 migrações e no segundo 6 migrações.

O perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência é apresentado na Figura 6.12 para o caso centralizado (Figura 6.12(a)) e distribuído (Figura 6.12(b)). O volume de dados gerado mantêm-se entre 100 Kbps e 200 Kbps (característica da aplicação) e durante migrações atinge a taxa de 4000 Kbps. Após as migrações de tarefa, pode-se observar uma mudança no perfil de tráfego com relação à origem das mensagens. Inicialmente (até o *tick* 100), os únicos nodos que originam tráfego (interno) são os nodos 0, 3 e 5, que possuem as aplicações e encontram-se em sobrecarga. Após a migração de tarefas, os nodos 1, 2 e 4 também passam a originar tráfego, fato este que pode ser observado a partir das primeiras migrações na Figura 6.12(b) após o *tick* 100. Parte do tráfego originalmente direcionado internamente aos nodos passa a ser direcionado para a rede de interconexão, e tarefas migradas que possuem comunicações com tarefas dos nodos origem também passam a transmitir seu tráfego pela rede.

Aplicação 3, MPSoC 4x4

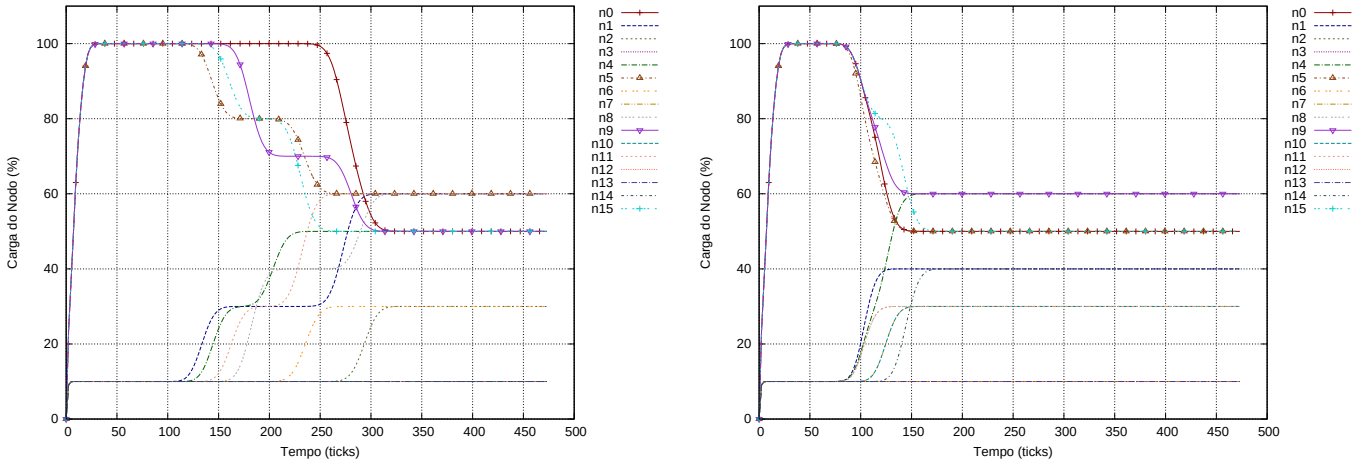
Na segunda versão da terceira aplicação é utilizada uma malha com dimensões 4x4, estando quatro elementos de processamento em situação de sobrecarga (nodos 0, 5, 9 e 15). Na Figura 6.13 são apresentados os perfis de carga ao longo do tempo utilizando-se um gerente centralizado (Figura 6.13(a)) e gerentes distribuídos (Figura 6.13(b)).



(a) (b)

Figura 6.12: Aplicação sintética 3, perfil de tráfego gerado em um MPSoC 3x2: gerente centralizado (a) e gerentes distribuídos (b)

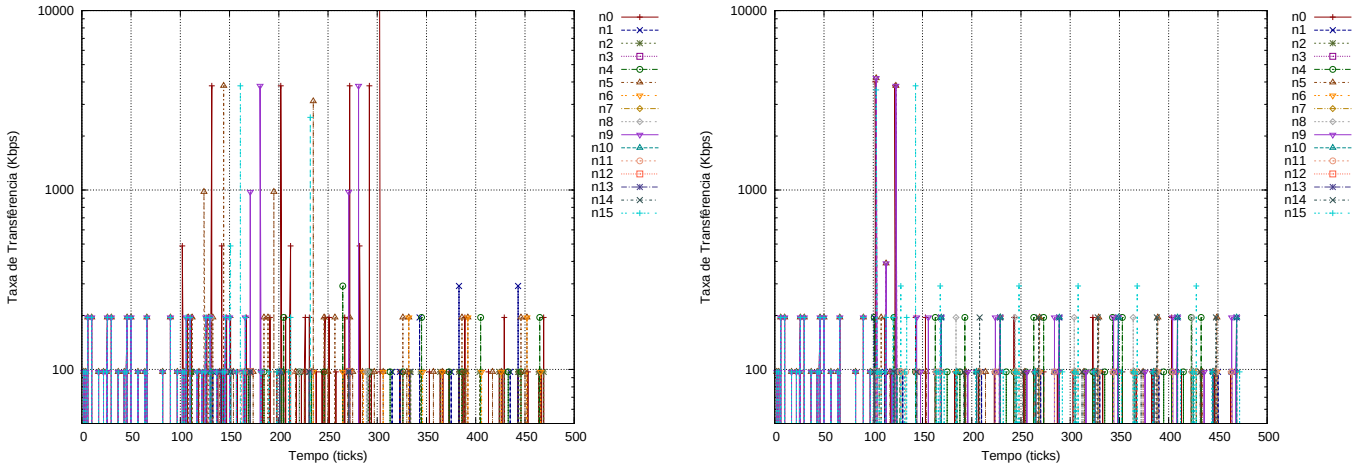
Observa-se que os gerentes distribuídos conseguem estabilizar a carga dos nodos em menor tempo. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 310, ou seja em 210 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 165, ou seja em 65 *ticks*. Nos dois casos são realizadas 8 migrações.



(a) (b)

Figura 6.13: Aplicação sintética 3, utilização dos elementos de processamento em um MPSoC 4x4: gerente centralizado (a) e gerentes distribuídos (b)

O perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência é apresentado na Figura 6.14 para o caso centralizado (Figura 6.14(a)) e distribuído (Figura 6.14(b)). O tráfego gerado na rede pelas tarefas que compõem a aplicação mantêm-se entre 100 Kbps e 200 Kbps e durante migrações atinge a taxa de 4000 Kbps. Após as migrações, o tráfego gerado na rede passa a variar entre 100 Kbps e 300 Kbps, em virtude da redução de perdas de *deadline*. Assim como no experimento anterior, a partir do *tick* 100 observa-se uma mudança no perfil de tráfego da aplicação em virtude das migrações. Inicialmente, apenas



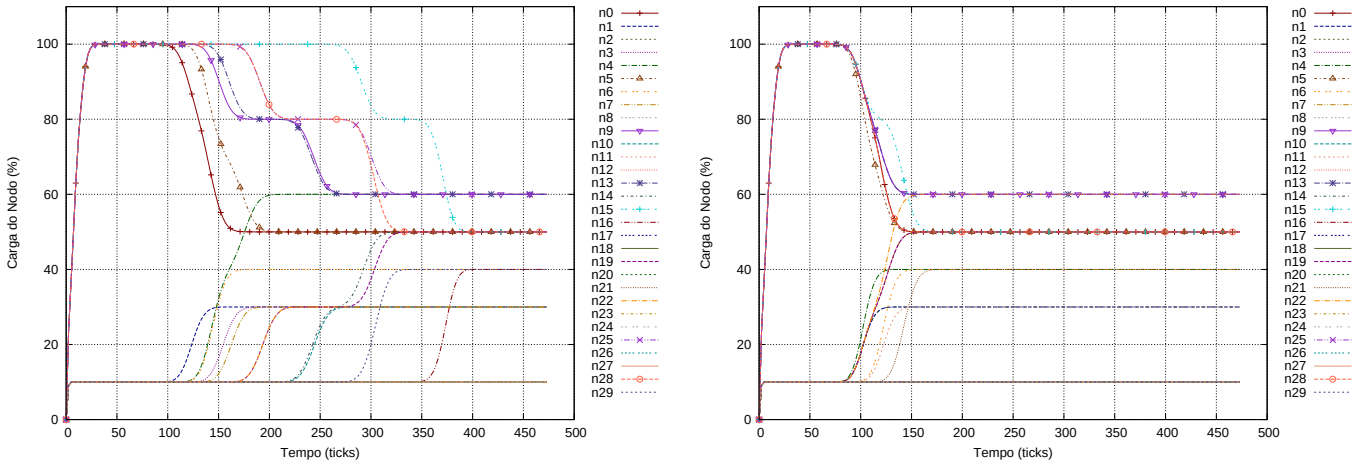
(a) (b)

Figura 6.14: Aplicação sintética 3, perfil de tráfego gerado em um MPSoC 4x4: gerente centralizado (a) e gerentes distribuídos (b)

os nodos 0, 5, 9 e 15 compõem a geração de tráfego, pois é nestes nodos que estão alocadas as aplicações. Em virtude das migrações de tarefa, outros nodos passam a transmitir dados, e o tráfego antes direcionado apenas internamente aos nodos passa pela rede de interconexão devido às comunicações entre tarefas migradas e tarefas residentes nos nodos de origem.

Aplicação 3, MPSoC 6x5

A terceira versão da terceira aplicação utiliza uma malha com dimensões 6x5, e os nodos 0, 5, 9, 13, 15, 25 e 28 encontram-se em sobrecarga. Na Figura 6.15 são apresentados os perfis de carga ao longo do tempo utilizando-se um gerente centralizado (Figura 6.15(a)) e gerentes distribuídos (Figura 6.15(b)).

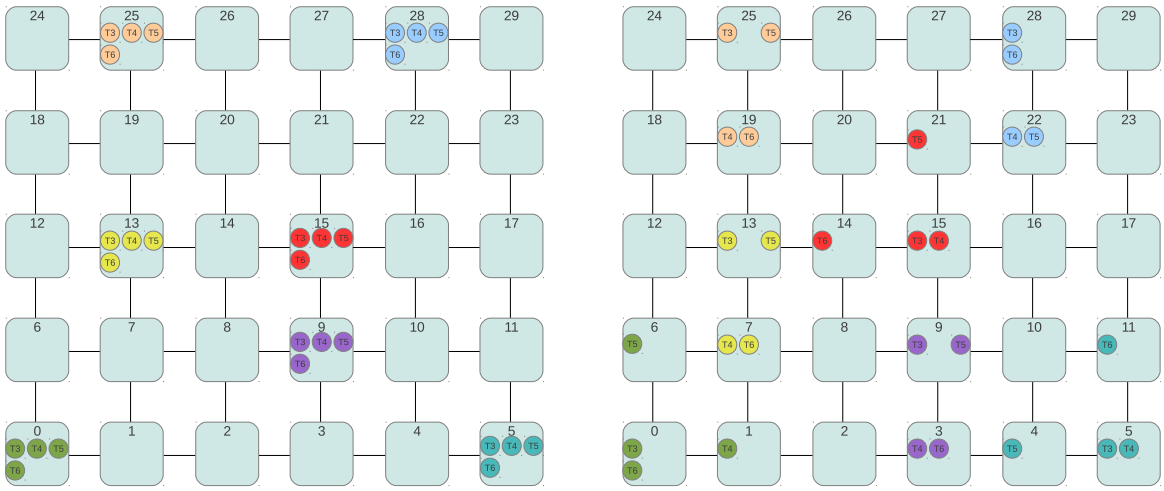


(a) (b)

Figura 6.15: Aplicação sintética 3, utilização dos elementos de processamento em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

Observa-se que os gerentes distribuídos conseguem estabilizar a carga dos nodos em menor tempo. Com malhas desta dimensão, mesmo aplicações simples como a ilustrada neste exemplo são beneficiadas com o uso de gerentes distribuídos devido à sua escalabilidade. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 390, ou seja em 290 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 165, ou seja em 65 *ticks*, praticamente o mesmo tempo dos experimentos anteriores com esta aplicação, independente do tamanho da malha. Nos dois casos são realizadas 14 migrações.

A Figura 6.16 apresenta o mapeamento inicial (Figura 6.16(a)) e após a estabilização do sistema com os gerentes distribuídos (Figura 6.16(b)). Observa-se que as tarefas t_4 e t_5 do nodo 0 (verde) são migradas para os nodos 1 e 6 respectivamente. As tarefas t_6 e t_5 do nodo 5 (azul marinho) são migradas para os nodos 11 e 4 respectivamente e as tarefas t_4 e t_6 do nodo 9 (roxo) são migradas para o nodo 3. As tarefas t_4 e t_6 do nodo 13 (amarelo) são migradas para o nodo 7 e as tarefas t_6 e t_5 do nodo 15 (vermelho) são migradas para os nodos 14 e 21 respectivamente. O nodo 25 (rosa) tem suas tarefas t_4 e t_6 migradas para o nodo 19 e as tarefas t_4 e t_5 do nodo 28 (azul claro) são migradas para o nodo 22. Inicialmente 23 nodos encontram-se desocupados, e após as migrações apenas 13 nodos não executam nenhuma tarefa da aplicação.



(a) (b)

Figura 6.16: Aplicação sintética 3, gerentes distribuídos: mapeamento inicial até o *tick* 100 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

A Figura 6.17 apresenta o perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência para o caso centralizado (Figura 6.17(a)) e distribuído (Figura 6.17(b)). O volume de dados gerado pela aplicação mantêm-se entre 100 Kbps e 200 Kbps inicialmente e durante migrações atinge a taxa de 4000 Kbps. Algumas tarefas perdem *deadlines*, e após a estabilização o volume de dados da aplicação mantêm-se entre 100 Kbps e 300 Kbps. Pode-se observar uma mudança no perfil de tráfego com relação à origem das mensagens e volume de dados após as migrações de tarefas.

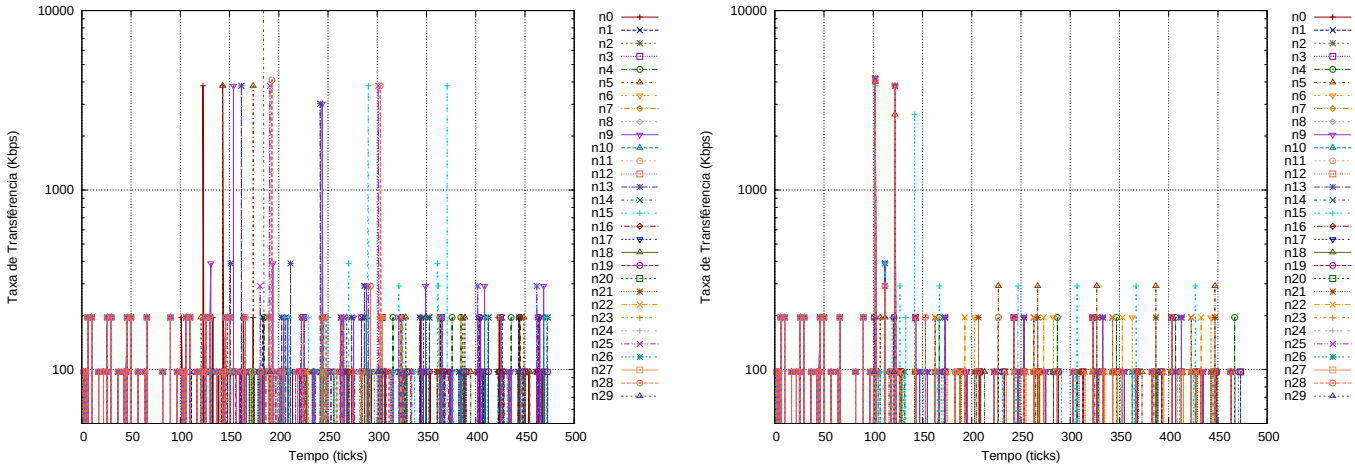


Figura 6.17: Aplicação sintética 3, perfil de tráfego gerado em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

Inicialmente (até o *tick* 100), os únicos nodos que originam tráfego (interno) são os nodos 0, 5, 9, 13, 15, 25 e 28 sendo estes os que possuem as aplicações e encontram-se em sobrecarga. Após a migração de tarefas, todos os nodos que recebem tarefas também passam a originar tráfego, fato este que pode ser observado a partir das primeiras migrações na Figura 6.17(b) após o *tick* 100. Parte do tráfego originalmente direcionado internamente aos nodos passa a ser direcionado para a rede de interconexão.

Na Tabela 6.6 é apresentado um resumo dos experimentos relacionados à terceira aplicação. Sem gerência de migração, ao término do tempo de execução seriam perdidos 36 *deadlines* no primeiro experimento (malha 3x2), 48 no segundo (malha 4x4) e 84 no terceiro (malha 6x5). Com o mecanismo de gerência centralizado, as perdas de *deadline* são reduzidas para 10, 20 e 33 nos respectivos experimentos. Utilizando-se o mecanismo de gerência distribuído, as perdas de *deadline* são reduzidas para 9 (melhoria de 10%) no primeiro experimento, 12 (melhoria de 40%) no segundo e 24 (melhoria de 27.27%) no terceiro caso. O tempo de estabilização é menor com a abordagem distribuída, e os ganhos em comparação a abordagem centralizada são de 25% (malha 3x2), 69.04% (malha 4x4) e 56,66% (malha 6x5).

Tabela 6.6 – Resumo dos experimentos, Aplicação 3

	MPSoC 3x2,	3 instâncias		MPSoC 4x4,	4 instâncias		MPSoC 6x5,	7 instâncias	
Gerência	Migrações	Tempo de estabilização	Perdas de *deadline*	Migrações	Tempo de estabilização	Perdas de *deadline*	Migrações	Tempo de estabilização	Perdas de *deadline*
Sem gerência	0	-	36	0	-	48	0	-	84
Centralizada	5	100	10	8	210	20	14	290	33
Distribuída	6	75	9	8	65	12	14	65	24
Melhoria		25%	10%		69.04%	40%		56.66%	27.27%

6.2.4 Aplicação 4, MPSoCs 3x2, 4x4 e 6x5

A quarta aplicação sintética é apresentada na Figura 6.18. Uma abordagem semelhante à aplicação anterior foi utilizada, e consiste em replicar instâncias da mesma aplicação em MPSoCs com dimensões 3x2, 4x4 e 6x5. Para estes MPSoCs são novamente alocadas 3, 4 e 7 instâncias desta aplicação, ou seja 27, 36 e 63 tarefas. No exemplo, cada instância da aplicação é mapeada em um único elemento de processamento sendo sua carga total 110%, composta pela aplicação (100%) e o gerente de migração (gerente distribuído, gerente centralizado ou tarefa de migração).

O exemplo aqui ilustrado apresenta uma situação onde uma aplicação relativamente complexa é mapeada em um único elemento de processamento, e diversas migrações devem ser realizadas para que esta possa executar de acordo com seus parâmetros de tempo real.

As instâncias de cada aplicação não possuem dependências entre si. Tarefas da mesma aplicação, no entanto, possuem maior dependência que na aplicação anterior. Para cada instância, são definidas 9 tarefas, onde as tarefas t_4, t_5, t_6, t_7 recebem dados da tarefa t_3, t_8 recebe dados de t_4, t_9 recebe dados das tarefas t_3, t_4 e t_5, t_{10} recebe dados de t_4, t_5, t_6 e t_7 e t_{11} recebe dados de t_8, t_9 e t_{10}. Para esta aplicação, são definidas 15 comunicações, com volumes de dados que variam entre 64 e 1280 bytes.

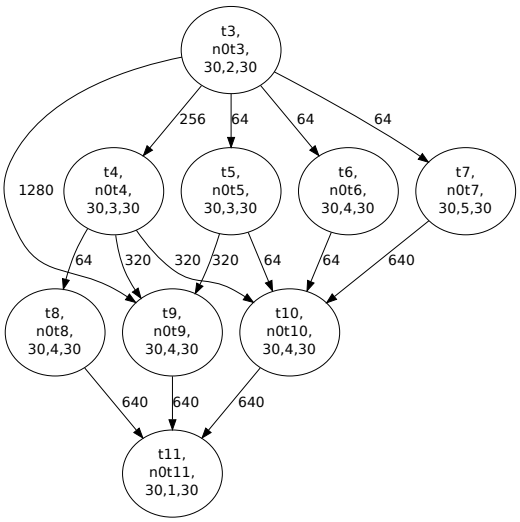


Figura 6.18 – ATG das tarefas iniciais da aplicação sintética 4

Aplicação 4, MPSoC 3x2

A primeira versão da quarta aplicação utiliza uma malha com dimensões 3x2, onde os nodos 0, 3 e 5 possuem uma instância da aplicação cada e encontram-se em sobrecarga, assim como na aplicação anterior. Na Figura 6.19 são apresentados os perfis de carga ao longo do tempo onde em um caso é utilizado um gerente centralizado (Figura 6.19(a)) e em outro gerentes distribuídos (Figura 6.19(b)). Comparado à aplicação anterior, esta apresenta um aumento nos tempos de estabilização da carga nas duas abordagens de gerência. O aumento deve-se ao fato de uma maior

complexidade da aplicação em virtude do número de tarefas e dependências de comunicação, uma vez que migrações não podem ser realizadas no instante em que tarefas envolvidas no processo estiverem realizando trocas de mensagem.

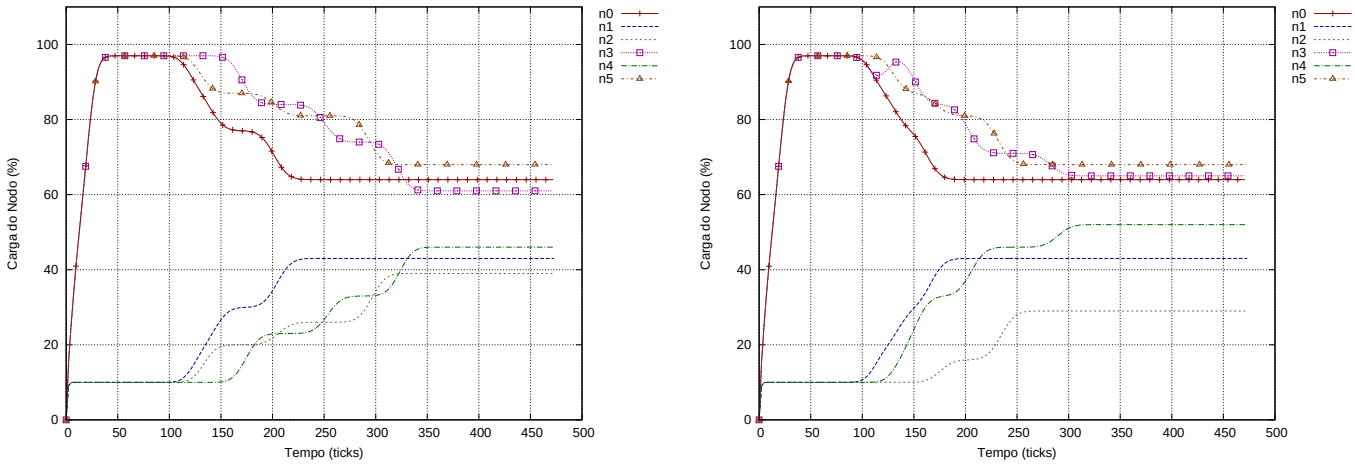


Figura 6.19: Aplicação sintética 4, utilização dos elementos de processamento em um MPSoC 3x2: gerente centralizado (a) e gerentes distribuídos (b)

Para este exemplo, gerentes distribuídos conseguem estabilizar a carga dos nodos em menor tempo apesar de terem o desempenho bastante comprometido. O número de dependências em cada aplicação, em conjunto com um número significativo de elementos de processamento sobrecarregados ao mesmo tempo restringe o paralelismo do algoritmo de gerência distribuído, e ele passa a ter um desempenho próximo ao do gerente centralizado. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 340, ou seja em 240 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 305, ou seja em 205 *ticks*. Em ambos os casos são realizadas 9 migrações.

Na Figura Figura 6.20 é apresentado o perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência para o caso centralizado (Figura 6.20(a)) e distribuído (Figura 6.20(b)). O volume de dados gerado da aplicação mantêm-se entre 100 Kbps e 600 Kbps e durante migrações atinge a taxa de 4000 Kbps. Após as migrações de tarefa, pode-se observar uma mudança no perfil de tráfego com relação à origem das mensagens e um aumento do volume gerado pela aplicação que passa a variar entre 100 e 1200 Kbps em virtude da redução de perdas de *deadline*. Após a migração de tarefas, os nodos 1, 2 e 4 também passam a originar tráfego, fato este que pode ser observado a partir das primeiras migrações na Figura 6.20(b) após o *tick* 100. Parte do tráfego originalmente direcionado internamente aos nodos passa a ser direcionado para a rede de interconexão, e dessa forma tarefas migradas as quais possuem comunicações com tarefas dos nodos origem passam a transmitir seu tráfego pela rede.

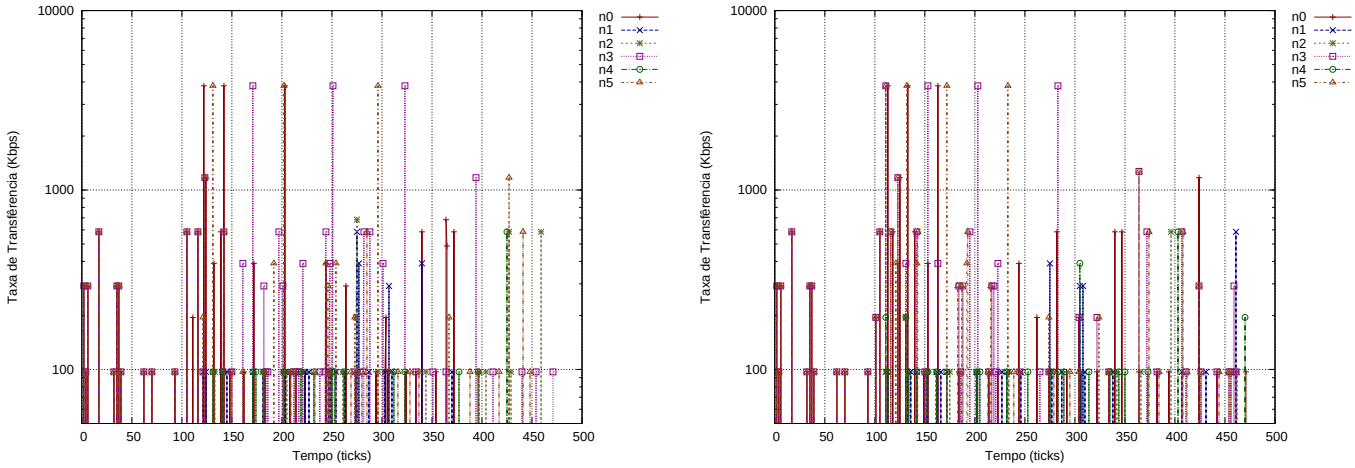


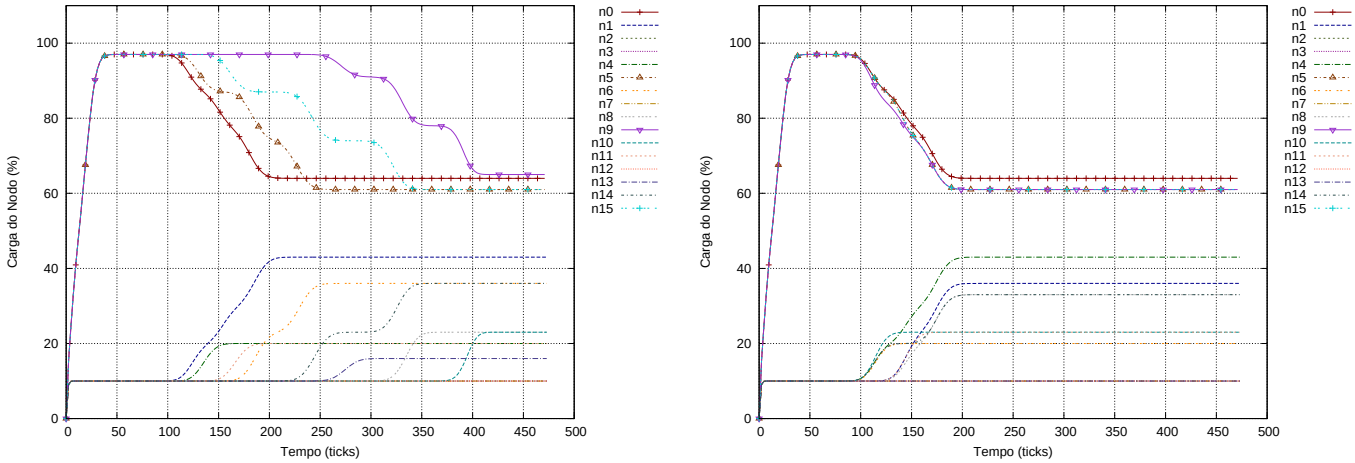
Figura 6.20: Aplicação sintética 4, perfil de tráfego gerado em um MPSoC 3x2: gerente centralizado (a) e gerentes distribuídos (b)

Aplicação 4, MPSoC 4x4

Na segunda versão da quarta aplicação é utilizada uma malha com dimensões 4x4, estando quatro elementos de processamento em situação de sobrecarga (nodos 0, 5, 9 e 15). Na Figura 6.21 são apresentados os perfis de carga ao longo do tempo utilizando-se um gerente centralizado (Figura 6.21(a)) e gerentes distribuídos (Figura 6.21(b)).

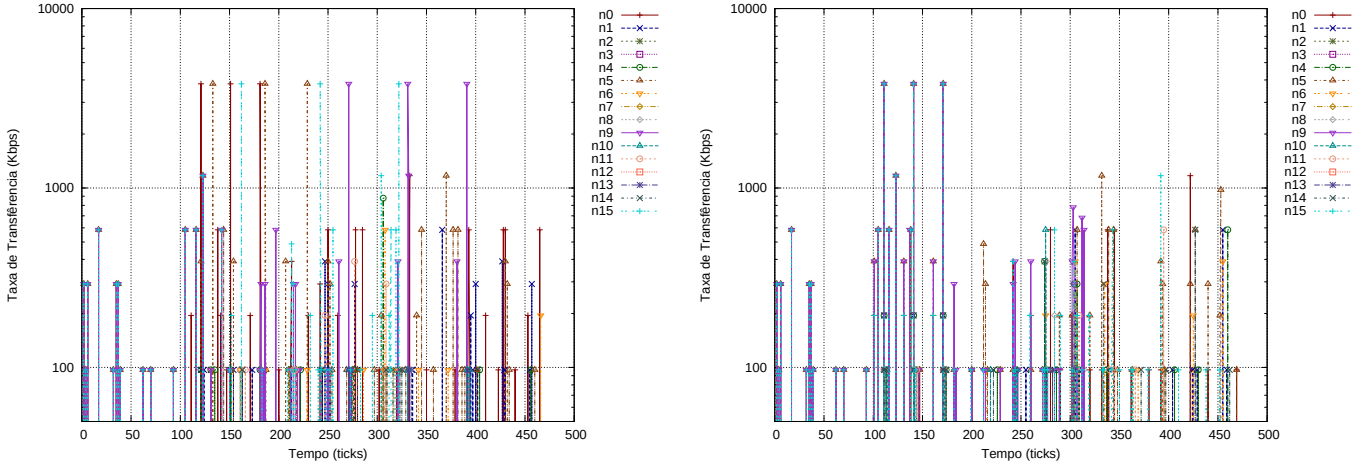
O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 410, ou seja em 310 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 195, ou seja em 95 *ticks* e mais uma vez é observada uma redução no tempo de estabilização. Nos dois casos são realizadas 12 migrações. Existem diversos elementos de processamento com carga baixa, e desta forma o algoritmo de gerência distribuído consegue establizar o sistema em tempo reduzido.

O perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência é apresentado na Figura 6.22 para o caso centralizado (Figura 6.22(a)) e distribuído (Figura 6.22(b)). O tráfego gerado na rede pelas tarefas que compõem a aplicação mantêm-se entre 100 Kbps e 600 Kbps e durante migrações atinge a taxa de 4000 Kbps. Após as migrações, o tráfego gerado na rede passa a variar entre 100 Kbps e 1200 Kbps. A partir do *tick* 100 observa-se uma mudança no perfil de tráfego da aplicação em virtude das migrações. No início da execução, apenas os nodos 0, 5, 9 e 15 realizam a geração de tráfego, pois é nestes que estão alocadas as aplicações. Em virtude das migrações de tarefa, outros nodos passam a transmitir dados, e o tráfego antes direcionado apenas internamente aos nodos passa pela rede de interconexão devido às comunicações entre tarefas migradas e tarefas residentes nos nodos de origem.



(a) (b)

Figura 6.21: Aplicação sintética 4, utilização dos elementos de processamento em um MPSoC 4x4: gerente centralizado (a) e gerentes distribuídos (b)



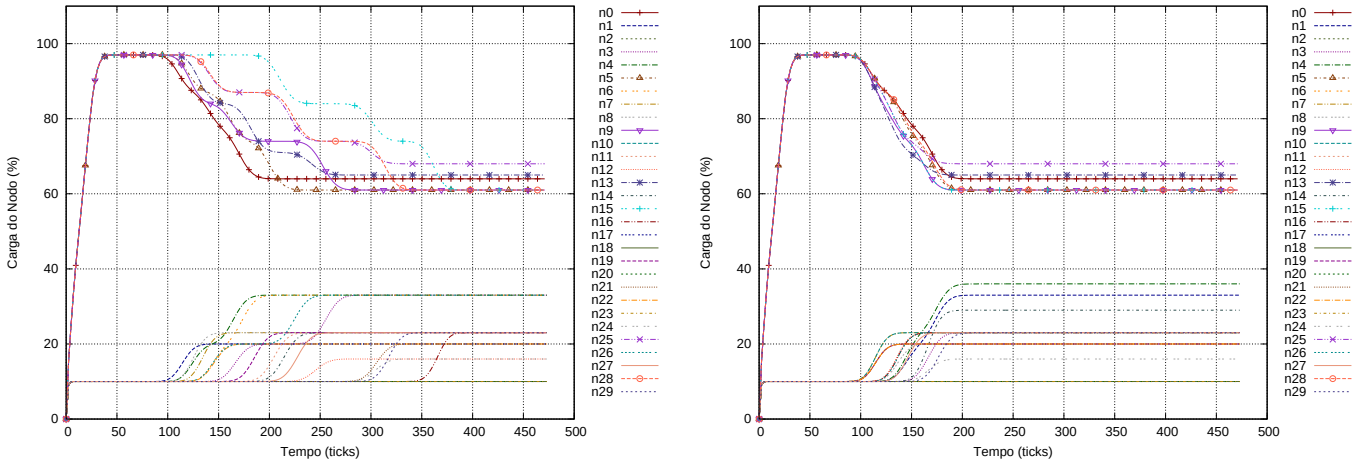
(a) (b)

Figura 6.22: Aplicação sintética 4, perfil de tráfego gerado em um MPSoC 4x4: gerente centralizado (a) e gerentes distribuídos (b)

Aplicação 4, MPSoC 6x5

A terceira versão da quarta aplicação faz uso de uma malha com dimensões 6x5, onde os nodos 0, 5, 9, 13, 15, 25 e 28 encontram-se em sobrecarga. Na Figura 6.23 são apresentados os perfis de carga ao longo do tempo com um único gerente centralizado (Figura 6.23(a)) e com gerentes distribuídos (Figura 6.23(b)).

Com malhas desta dimensão, aplicações relativamente complexas como a ilustrada neste exemplo são beneficiadas com o uso de gerentes distribuídos mesmo com muitas dependências entre tarefas. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 375, ou seja em 275 *ticks*. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 190, ou seja em 90 *ticks*, o mesmo tempo do experimento anterior com esta mesma aplicação. Nos dois casos são realizadas 20 migrações.

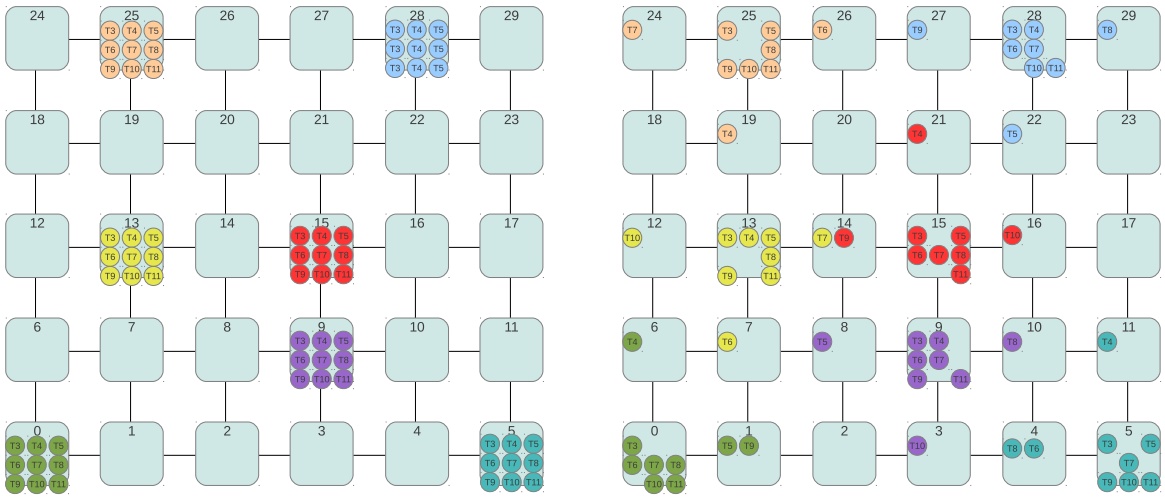


(a) (b)

Figura 6.23: Aplicação sintética 4, utilização dos elementos de processamento em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

O mapeamento inicial da aplicação é apresentado na Figura 6.24(a), e o mapeamento final obtido pelos gerentes distribuídos na Figura 6.24(b). Neste cenário, as tarefas t_5 e t_9 do nodo 0 (verde) são migradas para o nodo 1 e t_4 para o nodo 6. As tarefas t_8 e t_6 do nodo 5 (azul marinho) são migradas para o nodo 4 e t_4 para o nodo 11. O nodo 9 (roxo) tem suas tarefas t_{10}, t_5 e t_8 migradas para os nodos 3, 8 e 10 respectivamente enquanto o nodo 13 (amarelo) tem suas tarefas t_6, t_{10} e t_7 migradas para os nodos 7, 12 e 14. As tarefas t_9, t_{10} e t_4 do nodo 15 (vermelho) são migradas para 14, 16 e 21 enquanto t_4, t_7 e t_6 do nodo 25 (rosa) são migradas para os nodos 19, 24 e 26 respectivamente. O nodo 28 (azul claro) tem as tarefas t_5, t_9 e t_8 migradas para os nodos 22, 27 e 29. Anterior às migrações, 23 nodos estão desocupados (sem tarefas da aplicação) e após as migrações, apenas 5 nodos encontram-se nesta situação.

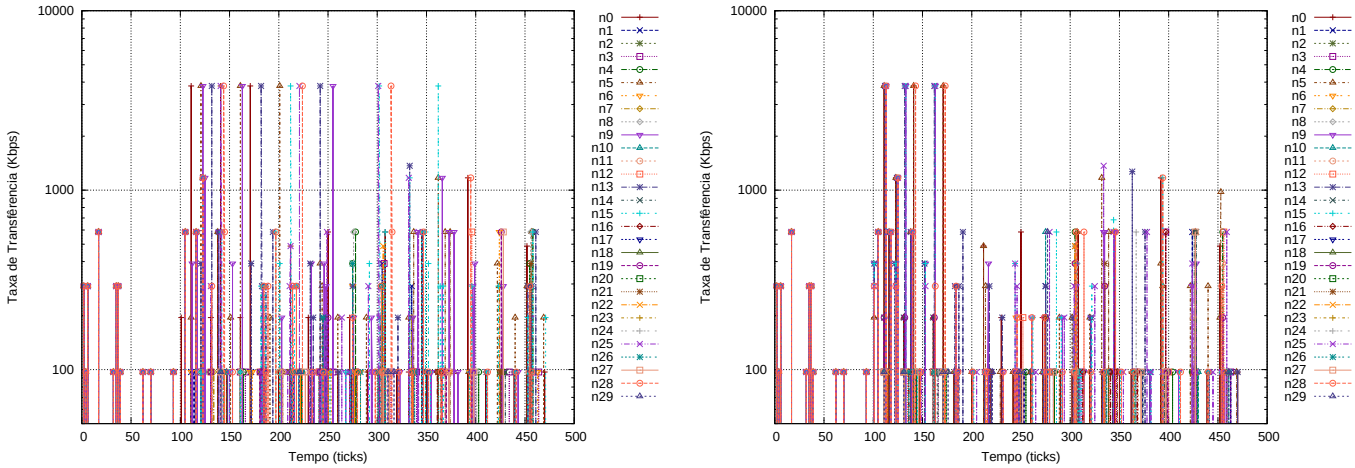
A Figura 6.25 apresenta o perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência para o mecanismo centralizado (Figura 6.25(a)) e distribuído (Figura 6.25(b)). O volume de dados gerado pela aplicação mantêm-se entre 100 Kbps



(a) (b)

Figura 6.24: Aplicação sintética 4, gerentes distribuídos: mapeamento inicial até o *tick* 100 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

e 600 Kbps inicialmente e durante migrações atinge a taxa de 4000 Kbps. Algumas tarefas perdem *deadlines*, e podem ser observados grandes espaços entre transferências (antes das migrações).



(a) (b)

Figura 6.25: Aplicação sintética 4, perfil de tráfego gerado em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

Após a estabilização o volume de dados da aplicação mantêm-se entre 100 Kbps e 1500 Kbps, com o maior número de transferências em torno de 600 kbps. Pode-se observar uma mudança no perfil de tráfego com relação à origem das mensagens e volume de dados após as migrações de tarefas. Inicialmente (até o *tick* 100), os únicos nodos que originam tráfego (interno) são os nodos 0, 5, 9, 13, 15, 25 e 28 sendo estes os que possuem as aplicações e encontram-se em sobrecarga. Após a migração de tarefas, todos os nodos que recebem tarefas também passam a originar tráfego, fato este que pode ser observado a partir das primeiras migrações na Figura 6.25(b) após o *tick*

100. Parte do tráfego originalmente direcionado internamente aos nodos passa a ser direcionado para a rede de interconexão.

Na Tabela 6.7 é apresentado um resumo dos experimentos relacionados à quarta aplicação. Sem gerência de migração, ao término do tempo de execução seriam perdidos 24 *deadlines* no primeiro experimento (malha 3x2), 32 no segundo (malha 4x4) e 56 no terceiro (malha 6x5). Com o mecanismo de gerência centralizado, as perdas de *deadline* são reduzidas para 8, 13 e 18 nos respectivos experimentos. Utilizando-se o mecanismo de gerência distribuído, as perdas de *deadline* são reduzidas para 8 no primeiro experimento (sem melhoria), 13 (melhoria de 38.46%) no segundo e 12 (melhoria de 33.33%) no terceiro experimento. O tempo de estabilização é menor com a abordagem distribuída, e os ganhos em comparação a abordagem centralizada são de 14.58% (malha 3x2), 69.35% (malha 4x4) e 67.27% (malha 6x5).

Tabela 6.7 – Resumo dos experimentos, Aplicação 4

	MPSoC 3x2,	3 instâncias		MPSoC 4x4,	4 instâncias		MPSoC 6x5,	7 instâncias	
Gerência	Migrações	Tempo de estabilização	Perdas de *deadline*	Migrações	Tempo de estabilização	Perdas de *deadline*	Migrações	Tempo de estabilização	Perdas de *deadline*
Sem gerência	0	-	24	0	-	32	0	-	56
Centralizada	9	240	8	12	310	13	20	275	18
Distribuída	9	205	8	12	95	8	20	90	12
Melhoria		14.58%	0%	0	69.35%	38.46%		67.27%	33.33%

6.3 Avaliação de Aplicações Reais

Esta Seção apresenta aplicações reais descritas de acordo com o modelo proposto. Para estas aplicações, são ilustradas situações onde o desempenho da mesma pode ser beneficiado com o uso de gerentes de migração distribuídos. Aplicações que possuem requisitos de execução de suas tarefas definidos em tempo de projeto devem ser otimizadas dinamicamente caso não estejam cumprindo com estes requisitos. Este tipo de situação ocorre caso as necessidades de processamento da aplicação com relação ao tempo de resposta e capacidade não sejam possíveis para todas as tarefas da aplicação com o mapeamento inicial, que pode ser não otimizado. Nos testes realizados, a aplicação é mapeada inicialmente em um único nodo e passa a executar de acordo com seus parâmetros de tempo real a medida que tarefas são migradas para outros nodos com recursos disponíveis.

6.3.1 MJPEG

Na aplicação MJPEG um *pipeline* para a de compressão de vídeo é ilustrado, e o grafo que descreve o fluxo de compressão de 16 blocos de 8x8 *pixels* (32x32 *pixels* de um *frame* de vídeo) desta aplicação é apresentado na Figura 6.26. A tarefa t_3 implementa a funcionalidade do bloco RGB2YUV, que tem como finalidade trasformar as três componentes que formam um *pixel* do espaço de cor RGB para o espaço de cor YUV (luminância e duas crominâncias). O

bloco DCT é implementado na tarefa t_4, e realiza a transformada do cossendo sobre os *pixels* do bloco, obtendo-se coeficientes que podem ser posteriormente comprimidos. A quantização destes coeficientes é realizada na tarefa t_5 a qual representa o bloco QUANT. Este bloco tem como objetivo diminuir a precisão dos coeficientes, tornando-os altamente redundantes, o que facilita o processo de compressão. A compressão do conjunto de *pixels* é realizada pela tarefa t_6 que implementa a funcionalidade do bloco VLC do codificador.

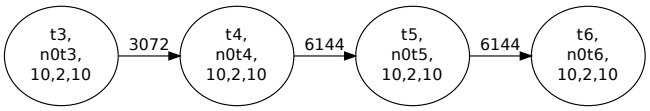


Figura 6.26 – ATG das tarefas da aplicação MJPEG

Cada bloco possui requisitos como quantidade de dados transmitidos e tempo de processamento, previamente caracterizados. O bloco RGB2YUV (tarefa t_3) possui um tempo de processamento de 88000 ciclos transmite 3072 bytes ao próximo estágio. O bloco DCT (tarefa t_4) utiliza 86352 ciclos, transmitindo 6144 bytes ao próximo bloco. O próximo estágio, representado pelo bloco QUANT (tarefa t_5) possui um tempo de processamento de 104144 ciclos e transmite 6144 bytes ao estágio VLC, que comprime o fluxo de dados em 260496 ciclos. Para a caracterização, os tempos de processamento de cada bloco foram obtidos no trabalho de Ngouanga[63] e foi utilizado um *tick* de tempo de execução (262000 ciclos) para transmitir cada mensagem entre blocos devido à sincronização necessária para a identificação global das tarefas, além de um *tick* adicional de processamento para todas as tarefas, uma vez que nenhuma possui tempo de processamento maior que este valor. No exemplo aqui apresentado, cada tarefa da aplicação foi configurada para executar a cada 10 *ticks* (104ms), e as tarefas dependem do resultado das anteriores.

Na implementação atual, a primeira tarefa apenas é liberada ao término da última em execuções sucessivas, tornando serial a execução de cada *pipeline* por fins de simplicidade e para evitar a saturação das filas em *software*, que possuem um tamanho limitado. Como cada tarefa da aplicação executa a cada 104ms espera-se que todas completem seu trabalho em aproximadamente 416ms para a obtenção de uma taxa de 2.4 *frames* por segundo, com 80% de utilização de tempo de processamento de um nodo pela aplicação (4 tarefas com os parâmetros $\tau_i = \{p_i, e_i, d_i\}$ definidos como $\tau_i = \{10, 2, 10\}$). Com duas instâncias da aplicação, é esperada uma taxa de 4.8 *frames* por segundo, uma vez que o processamento de 16 blocos de 8x8 *pixels* é totalmente independente nesta implementação.

A Figura 6.27(a) apresenta o mapeamento inicial da aplicação com duas instâncias do *pipeline* de processamento de vídeo alocados em um MPSoC 3x2. Inicialmente, as instâncias são alocadas nos nodos 0 e 5, os quais encontram-se com 90% de utilização cada (aplicação e gerente de migração do sistema operacional). Nesta configuração, uma iteração de cada *pipeline* executa em aproximadamente 20973678 ciclos, e juntos alcançam uma taxa de 2.38 *frames* por segundo. Esta configuração é mantida até o *tick* 300 (Figura 6.28(a)), onde os gerentes de migração distribuídos são ativados e realizam 4 migrações até a estabilização do sistema e obtenção do mapeamento final (Figura 6.27(b)), que ocorre em torno de 90 *ticks*. Observa-se que no mapeamento inicial (*tick* 0 a

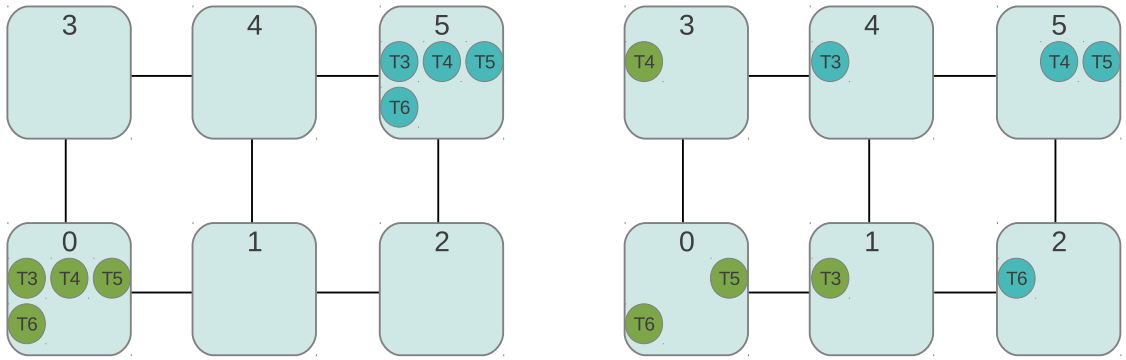


Figura 6.27: Aplicação MJPEG, gerentes distribuídos: mapeamento inicial até o *tick* 300 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

300) são executadas 8 iterações do algoritmo (Figura 6.28(b), transferências dos nodos 0 e 5 estão sobrepostas), ou aproximadamente 8 iterações em 3 segundos.

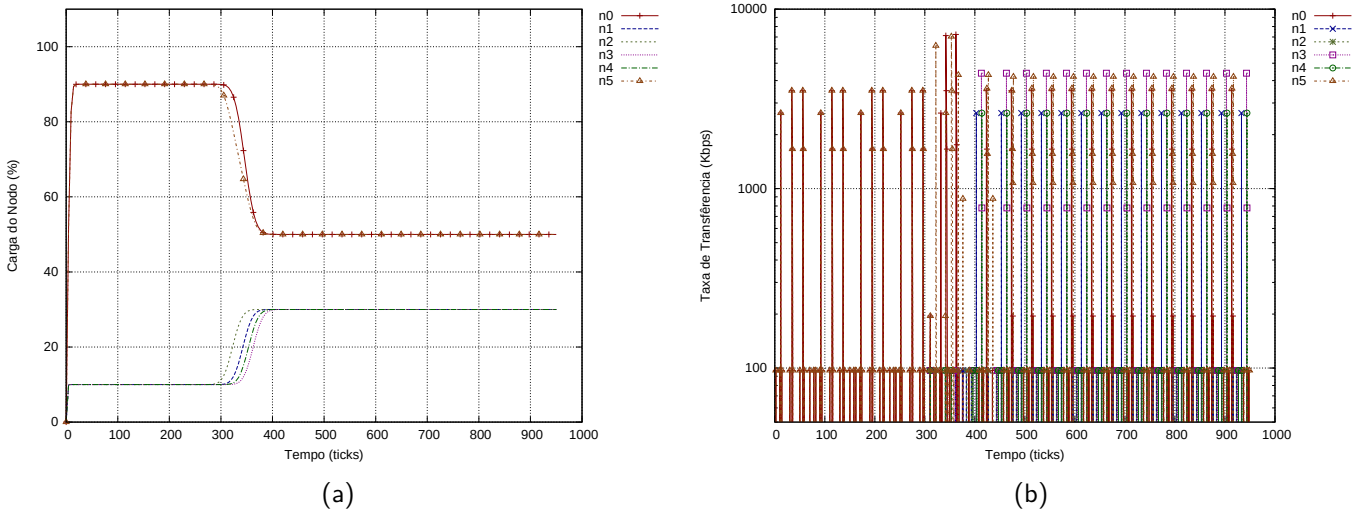


Figura 6.28 – Aplicação MJPEG, gerentes distribuídos: carga dos nodos (a) e perfil de tráfego (b)

O sistema encontra-se completamente estabilizado após as migrações a partir do *tick* 400 e executa em torno de 26 iterações em 5.5 segundos, alcançando o desempenho de aproximado de 10483633 ciclos por iteração do *pipeline* ou uma taxa de 4.77 *frames* por segundo, sendo este o valor esperado em virtude dos parâmetros de configuração e caracterização em tempo de projeto.

Um outro cenário de teste foi criado, onde além da aplicação MJPEG existem outras tarefas nos nodos previamente disponíveis. Estas tarefas compõem uma aplicação hipotética, que é apresentada na Figura 6.29. Esta aplicação também possui um intenso fluxo de comunicação entre suas tarefas além de uma utilização significativa de processamento. O objetivo deste cenário é demonstrar a possibilidade de migrar tarefas durante a execução de uma aplicação para otimizar o perfil de execução desta definido em tempo de projeto, mesmo em um ambiente onde os recursos são compartilhados com outras tarefas.

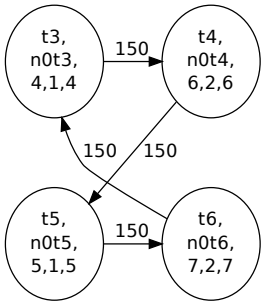
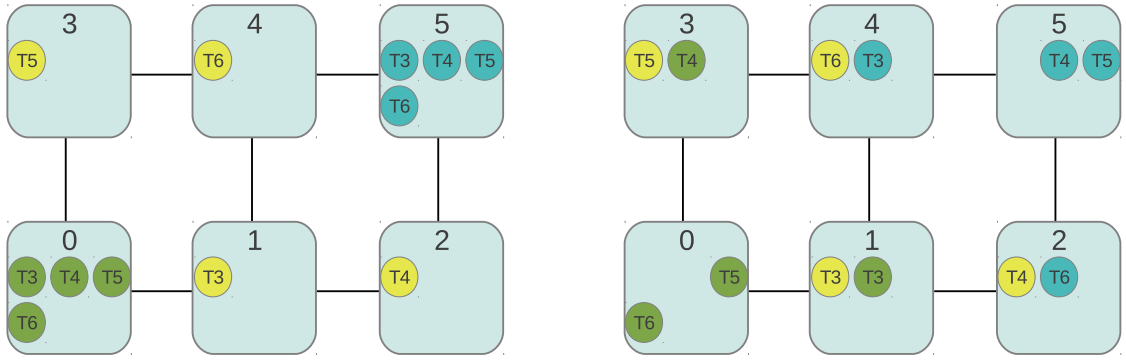


Figura 6.29: ATG das tarefas da aplicação hipotética que executa em conjunto com a aplicação MJPEG

Na Figura 6.30 é apresentado o mapeamento da aplicação hipotética (nodos 1, 2, 3 e 4) e da aplicação MJPEG que é o mesmo do caso anterior onde apenas esta aplicação era alocada. As tarefas da aplicação hipotética possuem seus próprios parâmetros de tempo real e seu volume de dados, além de compartilhar os canais de comunicação com a aplicação MJPEG. O mapeamento inicial é ilustrado na Figura 6.30(a) e o mapeamento final após as migrações na Figura 6.30(b).

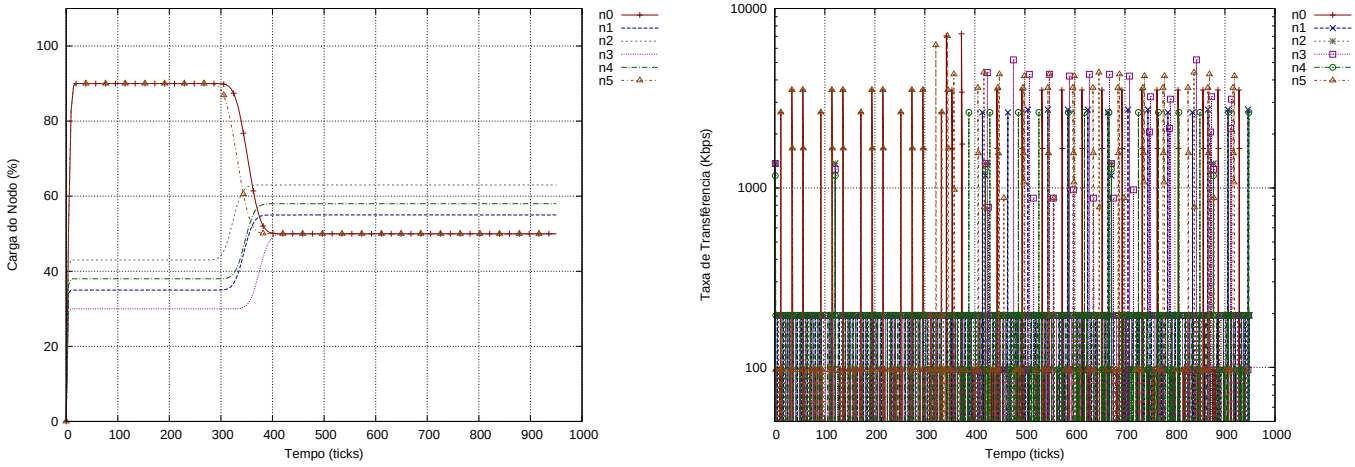


(a) (b)

Figura 6.30: Aplicação MJPEG em conjunto com outra hipotética, gerentes distribuídos: mapeamento inicial até o *tick* 300 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

A configuração do sistema e o desempenho da aplicação MJPEG mantida até o *tick* 300 (Figura 6.31(b)) é a mesma para este caso e o anterior, uma vez que as comunicações entre as tarefas da aplicação ocorrem no mesmo nodo, e não existem outras tarefas executando nestes. Após as migrações, que ocorrem em aproximadamente 95 *ticks* (Figura 6.31(a)), o perfil de tráfego da aplicação é modificado. A penalidade no tempo de estabilização em virtude da existência de outra aplicação previamente mapeada no mesmo MPSoC é de 5.26%. Esta modificação no perfil de tráfego frente o caso anterior pode ser atribuída ao compartilhamento dos canais de comunicação e das filas da interface de rede por tarefas da aplicação hipotética e da aplicação MJPEG, além dos parâmetros das tarefas que já executam nos nodos 1, 2, 3 e 4, que possuem prioridade maior que as tarefas da aplicação MJPEG. Um dos *pipelines* da aplicação MJPEG (inicialmente alocado no nodo 0) mantém o desempenho de aproximadamente 10483579 ciclos após as migrações, enquanto

outro (inicialmente alocado no nodo 5) passa a ter variável seu desempenho entre 7862252 ciclos e 13105096 ciclos. Esta variação, no entanto, não influencia no desempenho final da aplicação MJPEG, que mantém uma taxa de 4.77 *frames* por segundo mesmo com outra aplicação executando em paralelo no sistema.



(a) (b)

Figura 6.31: Aplicação MJPEG em conjunto com outra hipotética, gerentes distribuídos: carga dos nodos (a) e perfil de tráfego (b)

6.3.2 VOPD

A aplicação VOPD simula o comportamento de diversos módulos que compõem um decodificador VOP. Esta aplicação é descrita por Murali[61], onde são apresentadas as taxas de transferência de dados entre cada módulo. O atraso em virtude do processamento de cada parte não é especificado no trabalho de Murali, e desta forma os parâmetros de caracterização foram definidos de maneira semelhante à aplicação anterior. Cada tarefa representa um módulo da aplicação, e todas as tarefas foram configuradas para executarem a cada 104ms. Novamente são utilizados 2 *ticks* de tempo de execução (ou capacidade) por tarefa, onde um é utilizado para a transferência de dados e outro para processamento. O grafo desta aplicação caracterizada é apresentado na Figura 6.32, e este é composto por 17 tarefas.

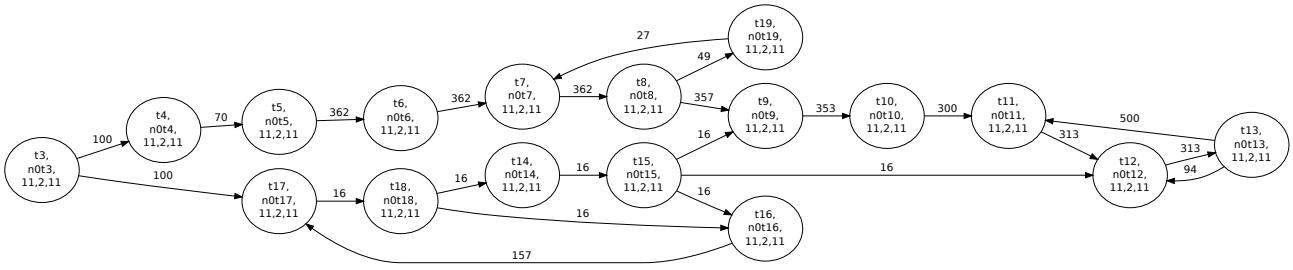


Figura 6.32 – ATG das tarefas da aplicação VOPD

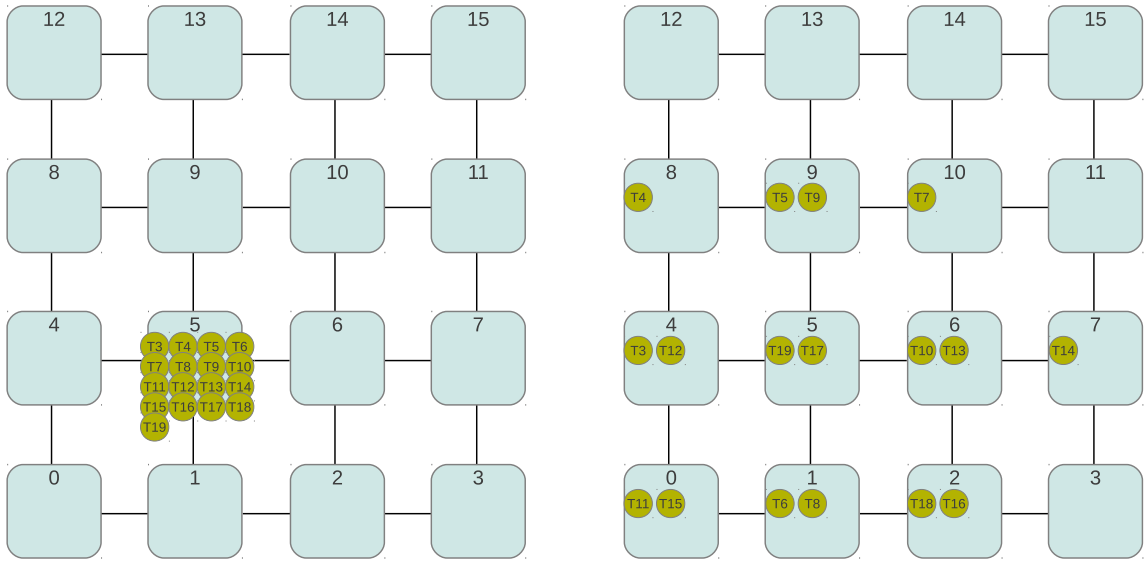
Os módulos da aplicação correspondem às seguintes tarefas: *demux* (t_3), *variable length decoder* (t_4), *run length decoder* (t_5), *inverse scan* (t_6), *AC/DC prediction* (t_7), *inverse quantizer* (t_8), *inverse cossine transform* (t_9), *up sample* (t_{10}), *VOP reconstruction* (t_{11}), *padding* (t_{12}), *VOP memory* (t_{13}), *up sample 2* (t_{14}), *reference memory* (t_{15}), *downsample and context calculation* (t_{16}), *arithmetic decoder* (t_{17}), *memory* (t_{18}) e *stripe memory* (t_{19}). As tarefas t_4 (*variable length decoder*) e t_{17} (*arithmetic decoder*) definem o início de caminhos independentes dentro do *pipeline*. Novamente, por questões de simplicidade e com o intuito de evitar a saturação das filas em *software* devido a seu tamanho limitado, cada iteração sucessiva de execução do *pipeline* ocorre após o término da anterior. Todas as tarefas da aplicação executam a cada 104ms, e em um pior caso onde cada tarefa dependesse do resultado da tarefa anterior o término de uma iteração deveria ocorrer em até 1781ms. Obviamente este não é o caso da aplicação, tendo em vista os dois caminhos paralelos do *pipeline*. O tempo de uma iteração não pode ser reduzido pela metade, no entanto, devido a alguns ciclos existentes no fluxo de execução da aplicação.

Todas as tarefas da aplicação alocadas a um único nodo correspondem a 340% de utilização com os parâmetros configurados (seriam necessários 3.4 nodos com 100% do processamento dedicado à aplicação). Cada tarefa da aplicação utiliza 20% de tempo de processador, e dessa forma não é possível alocar mais do que duas tarefas por nodo (40% das tarefas e 10% do gerente de migração), sem exceder os limites de utilização (três tarefas com 20% de utilização e o gerente de migração corresponderiam a 70% de carga).

Os mapeamentos inicial e final da aplicação VOPD caracterizada com os parâmetros descritos em um MPSoC 4x4 são apresentados na Figura 6.33. Neste cenário, todas as tarefas da aplicação são mapeadas no nodo 5 (Figura 6.33(a)). Este encontra-se em sobrecarga, e o gerente de migração realiza 15 migrações para nodos vizinhos utilizando o algoritmo de espalhamento. Como previsto, são alocadas no máximo duas tarefas por nodo devido aos limites de utilização (Figura 6.33(b)).

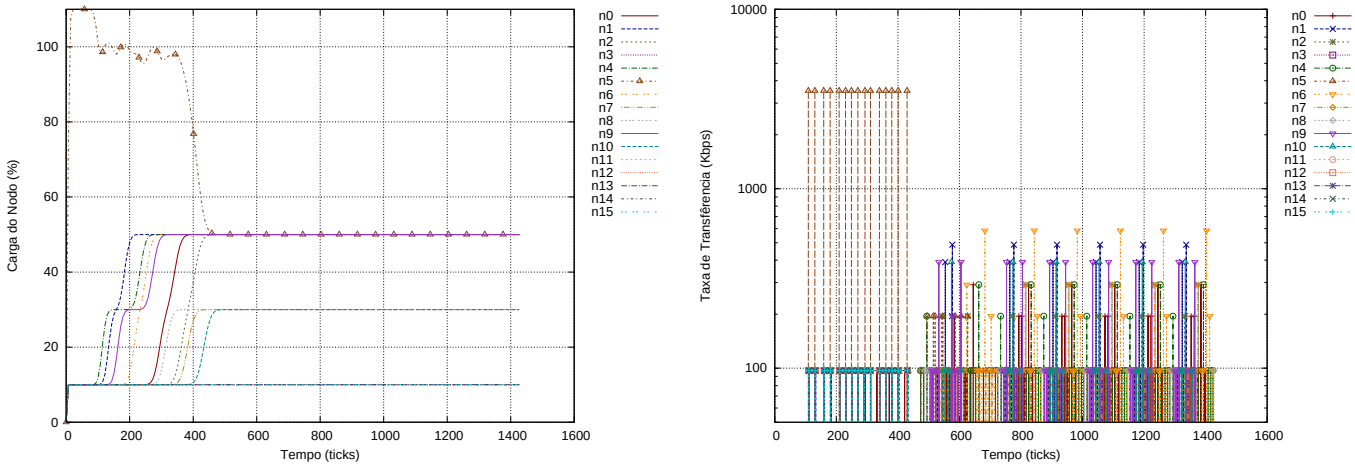
O posicionamento das tarefas nos nodos é dependente dos parâmetros de caracterização. Se o limite de utilização fosse maior que 69%, mais tarefas poderiam ser alocadas em um mesmo nodo respeitando as restrições de tempo real com os parâmetros utilizados. Mais tarefas poderiam ser alocadas também no caso da utilização de cada tarefa ser menor. Se as tarefas fossem configuradas com parâmetros de tempo real mais estritos (por exemplo, com período de 52ms e capacidade de 2 *ticks*), apenas uma tarefa da aplicação poderia ser alocada por nodo.

Uma situação interessante ocorre na execução da aplicação com o mapeamento inicial. A carga do nodo 5 é tão alta que apenas algumas tarefas executam inicialmente, e o *pipeline* de decodificação VOP é mantido bloqueado. O restante das tarefas, devido às perdas de *deadline* não chegam a executar nenhuma vez, e dessa forma o sistema operacional sequer conta sua ocupação. À medida que tarefas são migradas, momentaneamente a carga do nodo 5 cai e rapidamente sobe pois tarefas com a execução anteriormente bloqueadas passam a executar, apesar das perdas de *deadline*.



(a) (b)

Figura 6.33: Aplicação VOPD, gerentes distribuídos: mapeamento inicial até o *tick* 100 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)



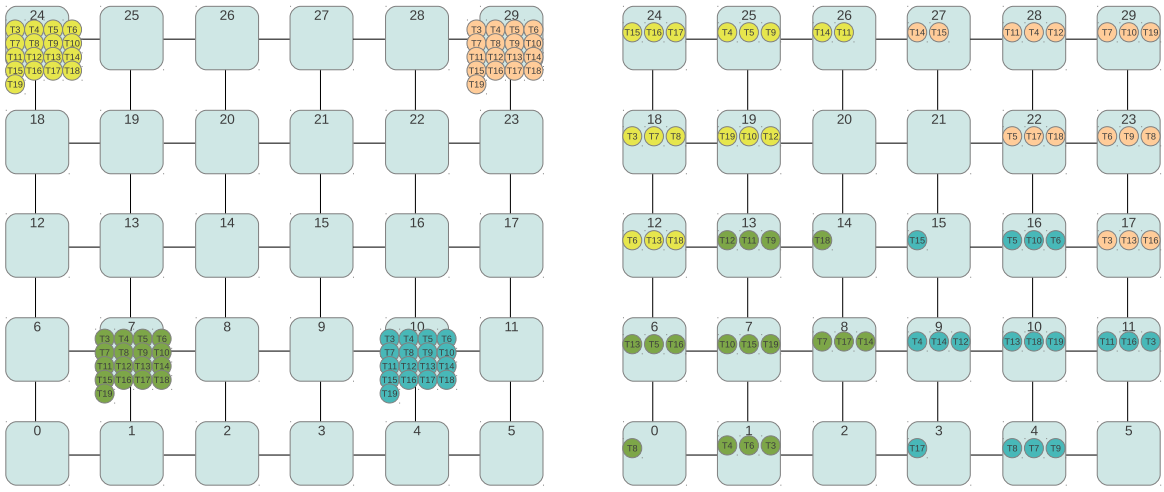
(a) (b)

Figura 6.34 – Aplicação VOPD, gerentes distribuídos: carga dos nodos (a) e perfil de tráfego (b)

Esta situação pode ser observada na Figura 6.34(a) do *tick* 100 (início do algoritmo de gerência) até o ponto em que todas as tarefas são migradas, o que ocorre no *tick* 450. A partir do *tick* 450 a aplicação tem todas as suas tarefas mapeadas definitivamente, e uma iteração do *pipeline* é executada em aproximadamente 36697866 ciclos, ou 1468ms, um valor muito próximo ao esperado como pior caso (1781ms, não considerando o paralelismo de execução do fluxo nem os ciclos no grafo). As transferências de dados entre as tarefas são ilustradas na Figura 6.34(b).

O segundo cenário criado para a aplicação VOPD faz uso de um MPSoC 6x5, onde quatro instâncias do *pipeline* de decodificação VOP são alocados. Todas as tarefas de um mesmo *pipeline* são inicialmente alocadas em um único nodo. Neste cenário, a configuração de todas as tarefas da aplicação é semelhante ao cenário anterior, no entanto os parâmetros $\tau_i = \{p_i, e_i, d_i\}$ de cada tarefa foram definidos como $\tau_i = \{11, 2, 11\}$. Desta forma, cada tarefa da aplicação executa a cada 115ms e como a utilização destas é um pouco menor (18.18%), até três tarefas desta aplicação podem ser alocadas a cada elemento de processamento em conjunto com as tarefas do sistema operacional, totalizando uma carga de 65%. Todas as tarefas da aplicação executam a cada 115ms, e em um pior caso onde cada tarefa dependesse do resultado da tarefa anterior o término de uma iteração de cada instância do *pipeline* deveria ocorrer em até 1960ms. As 17 tarefas de cada instância (68 tarefas da aplicação no total) correspondem a 309% de utilização em cada um dos nodos.

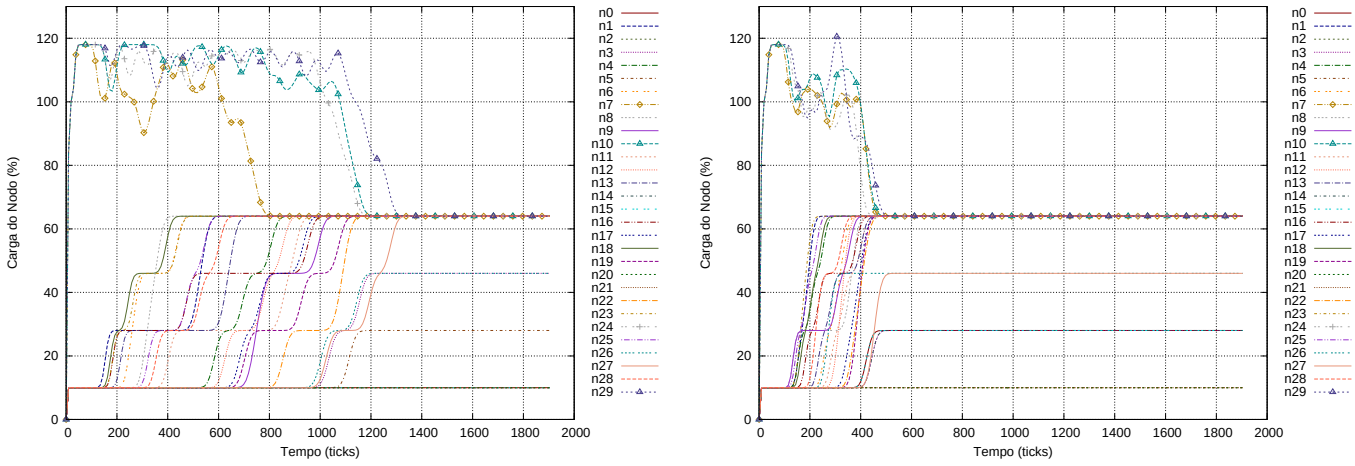
Os mapeamentos inicial e final da aplicação VOPD caracterizada com os parâmetros descritos em um MPSoC 6x5 são apresentados na Figura 6.35. Neste cenário, são mapeados *pipelines* independentes nos nodos 7, 10, 24 e 29 (Figura 6.35(a)). Estes encontram-se em sobrecarga, e os gerentes de migração realizam 56 migrações para nodos vizinhos utilizando o algoritmo de espalhamento. Como previsto, são alocadas no máximo três tarefas por nodo devido aos limites de utilização e configuração das tarefas (Figura 6.35(b)).



(a) (b)

Figura 6.35: Aplicação VOPD, gerentes distribuídos em um MPSoC 6x5: mapeamento inicial até o *tick* 100 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

A Figura 6.36 apresenta a carga do MPSoC ao longo do tempo para dois casos distintos, onde no primeiro um gerente de migração centralizado é utilizado (Figura 6.36(a)) e no segundo gerentes distribuídos. Esta aplicação, que consiste em um grande número de tarefas e um MPSoC de grandes dimensões deixa clara uma maior eficiência de gerentes distribuídos. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 1300 ou seja em 1200 *ticks*, aproximadamente 12.576 s. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 500 ou seja em 400 *ticks*, aproximadamente 4.192 s. Nos dois casos são realizadas 56 migrações.



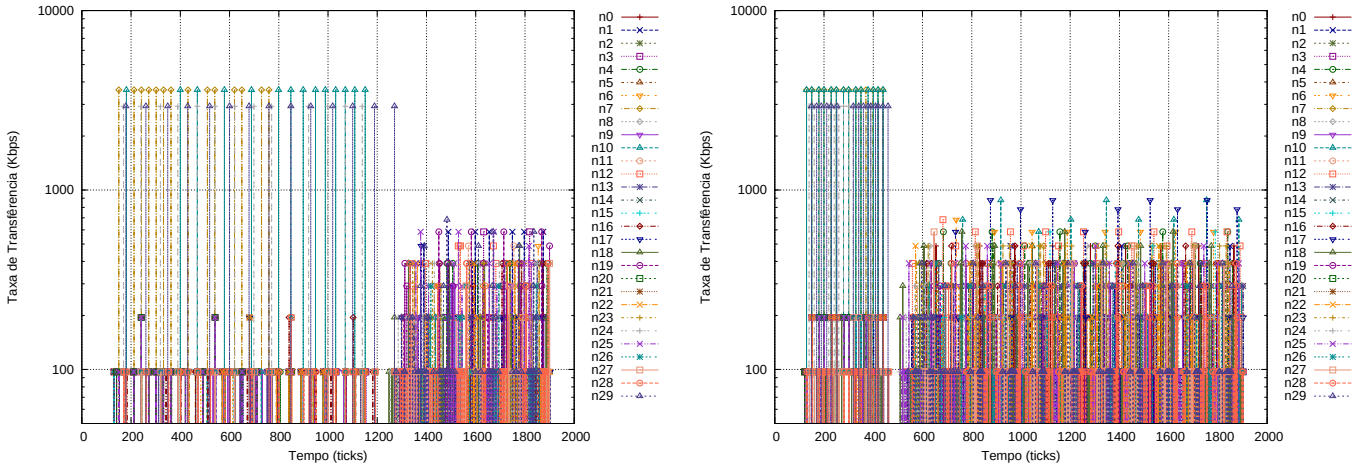
(a) (b)

Figura 6.36: Aplicação VOPD, utilização dos elementos de processamento em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

A partir do *tick* 500 (Figura 6.36(b)) a aplicação tem todas as suas tarefas mapeadas definitivamente, e uma iteração de cada um dos *pipelines* é executada em aproximadamente 40367871 ciclos, ou 1614ms, um valor muito próximo ao esperado como pior caso (1960ms, não considerando o paralelismo de execução do fluxo nem os ciclos no grafo).

Na Figura 6.37 é apresentado o perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência para o mecanismo centralizado (Figura 6.37(a)) e distribuído (Figura 6.37(b)). Inicialmente, em virtude das perdas de *deadline*, a aplicação basicamente não realiza a transferência de dados. Após a estabilização, o volume de dados da aplicação mantêm-se entre 100 kbps e 700 kbps (gerente centralizado) e entre 100 Kbps e 900 Kbps (gerentes distribuídos). A diferença deve-se ao posicionamento das tarefas no MPSoC, que não é o mesmo para os dois casos. Observa-se que a aplicação apenas começa a executar após as migrações de tarefa, que ocorrem em menor tempo no caso dos gerentes distribuídos.

Um resumo de experimentos realizados com a aplicação VOPD em um MPSoC 6x5 é apresentado na Tabela 6.8. Sem gerência de migração, ao término do tempo de execução (20 s) seriam perdidos 8604 *deadlines*. Com o mecanismo de gerência centralizado, as perdas de *deadline* são reduzidas para 2459 e utilizando-se o mecanismo de gerência distribuído, as perdas mantêm-se em 1080, o que representa uma melhoria de 56.07%. O tempo de estabilização é menor com a



(a) (b)

Figura 6.37: Aplicação VOPD, perfil de tráfego gerado em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

abordagem distribuída, e o ganho em comparação a abordagem centralizada para esta aplicação é de 66.66%.

Tabela 6.8 – Resumo dos experimentos, Aplicação VOPD

Gerência	Migrações	Tempo de estabilização	Perdas de *deadline*
Sem gerência	0	-	8604
Centralizada	56	1200	2459
Distribuída	56	400	1080
Melhoria		66.66%	56.07%

6.3.3 MPEG4

O comportamento de módulos de *hardware* e *software* de um codificador MPEG4 descrito por Milojevic[56] é simulado nesta aplicação. Novamente, apenas as taxas de transferência de dados são especificados para cada módulo, e dessa forma modelou-se o processamento de maneira semelhante às aplicações anteriores. Nesta aplicação, cada tarefa representa um módulo, e todas as tarefas foram configuradas para executarem a cada 115ms. São utilizados 2 *ticks* de tempo de execução por tarefa, onde um é reservado para a transferência de dados e outro para o processamento. Esta aplicação é composta por 18 tarefas, e é descrita na Figura 6.38.

Os módulos da aplicação correspondem às seguintes tarefas: *SRAM New Frame* (t_3), *Input Control* (t_4), *FIFO Current MBL* (t_5), *FIFO New MBL* (t_6), *ME* (t_7), *FIFO MV* (t_8), *MC* (t_9), *Error Block* (t_{10}), *Comp Block* (t_{11}), *Texture Coding* (t_{12}), *Texture Block* (t_{13}), *Texture Update* (t_{14}), *SRAM RecFrame* (t_{15}), *Copy Controller* (t_{16}), *Search Area* (t_{17}), *YUV Buffer* (t_{18}), *Quantised MBL* (t_{19}) e *VLC* (t_{20}). Por questões de simplicidade e com o intuito de evitar a saturação das filas

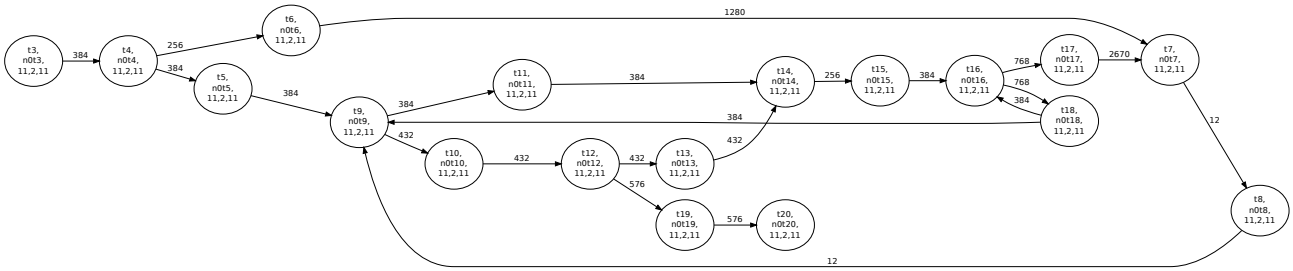
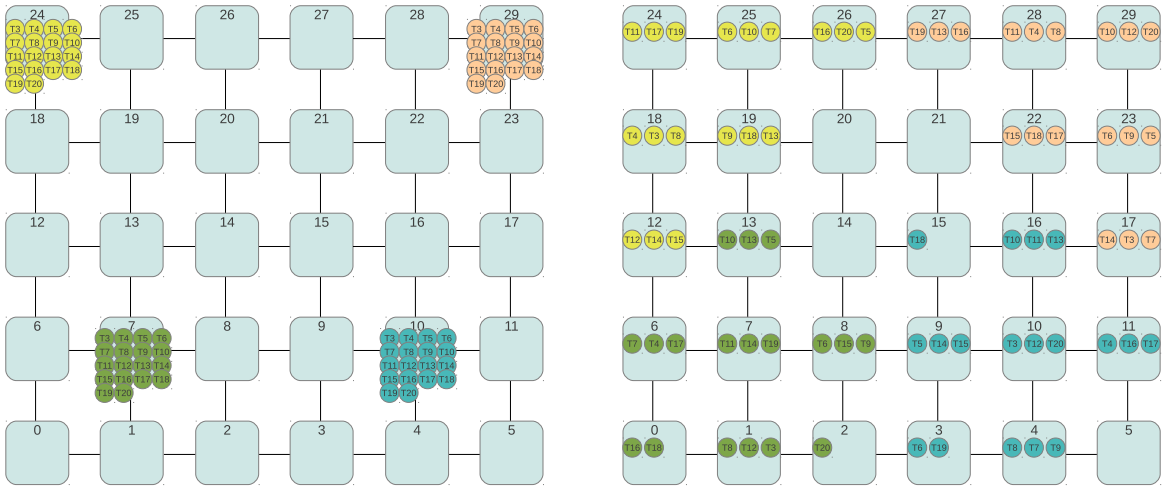


Figura 6.38 – ATG das tarefas da aplicação MPEG4

em *software* devido a seu tamanho limitado, cada iteração sucessiva de execução do *pipeline* ocorre após o término da anterior.

Para esta aplicação são utilizadas quatro instâncias do *pipeline* de codificação MPEG4 em um MPSoC 6x5. Todas as tarefas de um mesmo *pipeline* são inicialmente alocadas em um único nodo. Cada tarefa da aplicação possui uma utilização de 18.18%, e desta forma até três tarefas podem ser alocadas a cada elemento de processamento em conjunto com as tarefas do sistema operacional, totalizando uma carga de 65%. As tarefas da aplicação executam a cada 115ms, e em um pior caso onde cada tarefa dependesse do resultado da tarefa anterior o término de uma iteração de cada instância do *pipeline* deveria ocorrer em até 2075ms (não considerando os ciclos nem o paralelismo no *pipeline*). As 18 tarefas de cada instância (72 tarefas da aplicação no total) correspondem a 327% de utilização em cada um dos nodos ocupados.

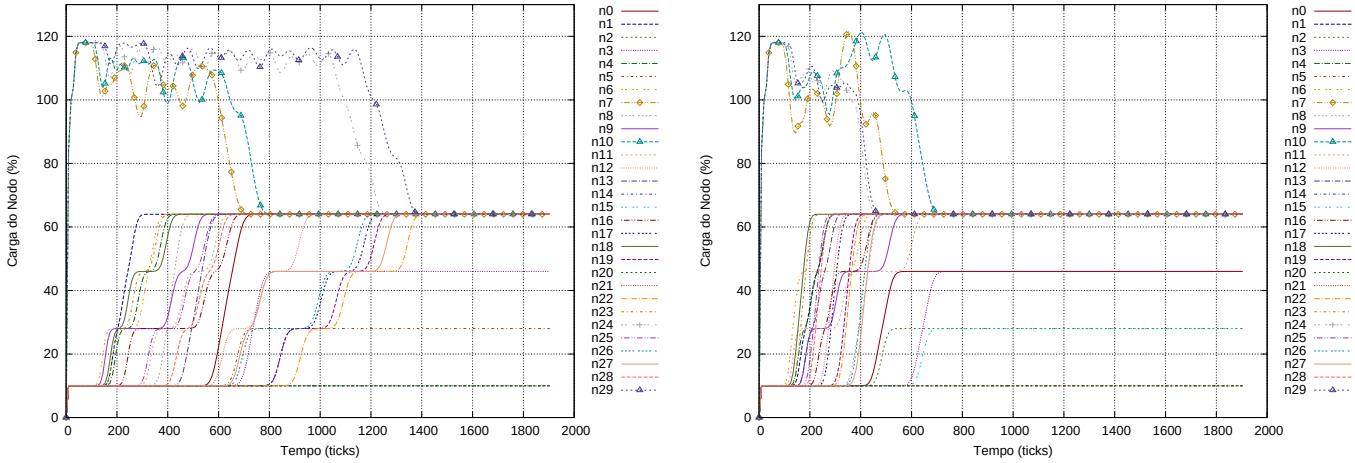


(a) (b)

Figura 6.39: Aplicação MPEG4, gerentes distribuídos em um MPSoC 6x5: mapeamento inicial até o *tick* 100 (início do algoritmo) (a); mapeamento final (estabilização do sistema) (b)

Os mapeamentos inicial e final da aplicação MPEG4 caracterizada com os parâmetros descritos em um MPSoC 6x5 são apresentados na Figura 6.39. Neste cenário, são mapeados *pipelines* independentes nos nodos 7, 10, 24 e 29 (Figura 6.39(a)). Estes encontram-se em sobrecarga, e os gerentes de migração realizam 60 migrações para nodos vizinhos utilizando o algoritmo de

espalhamento. São alocadas no máximo três tarefas por nodo devido aos limites de utilização e configuração das tarefas (Figura 6.39(b)).



(a) (b)

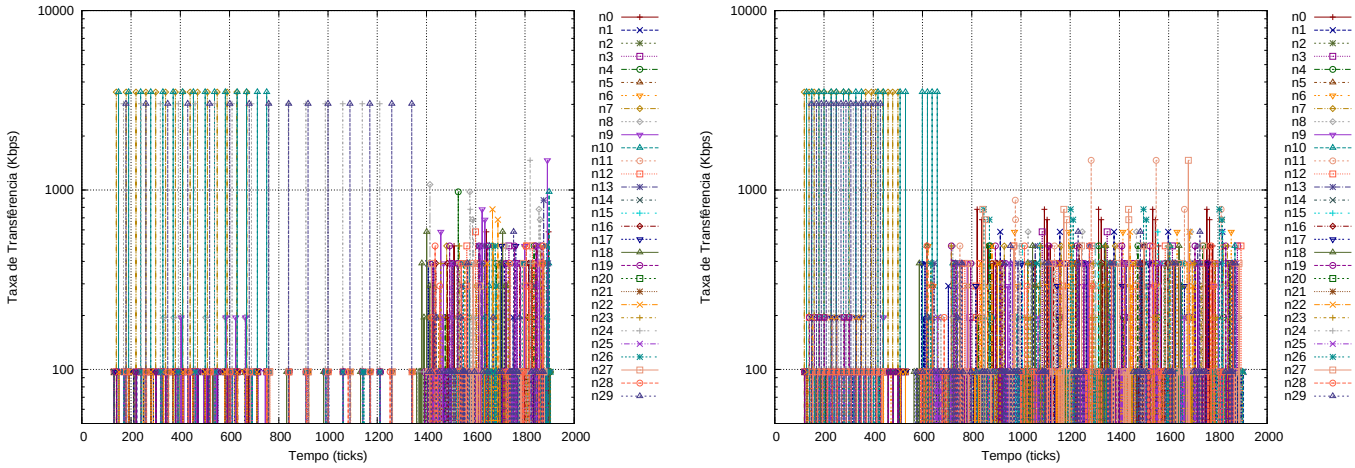
Figura 6.40: Aplicação MPEG4, utilização dos elementos de processamento em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

A Figura 6.40 apresenta a carga do MPSoC ao longo do tempo para os casos onde um gerente de migração centralizado é utilizado (Figura 6.40(a)) e outro onde são utilizados gerentes distribuídos. Esta aplicação, que consiste em um grande número de tarefas e um MPSoC de grandes dimensões apresenta novamente uma maior eficiência dos gerentes distribuídos. O tempo de estabilização no primeiro cenário (gerente centralizado) acontece entre os *ticks* 100 e 1380 ou seja em 1280 *ticks*, aproximadamente 13.414 s. No segundo cenário (gerentes distribuídos) a estabilização ocorre entre os *ticks* 100 e 700 ou seja em 600 *ticks*, aproximadamente 6.288 s. Nos dois casos são realizadas 60 migrações.

A partir do *tick* 700 (Figura 6.40(b)) a aplicação tem todas as suas tarefas mapeadas definitivamente, e uma iteração de cada um dos *pipelines* é executada em aproximadamente 57669496 ciclos, ou 2306ms.

Na Figura 6.41 é apresentado o perfil de tráfego gerado pela aplicação e em decorrência das migrações e trocas de mensagem de gerência para o mecanismo centralizado (Figura 6.41(a)) e distribuído (Figura 6.41(b)). Inicialmente, em virtude das perdas de *deadline*, a aplicação basicamente não realiza a transferência de dados. Após a estabilização, o volume de dados da aplicação mantêm-se entre 100 kbps e 1500 kbps.

Um resumo de experimentos realizados com a aplicação MPEG4 em um MPSoC 6x5 é apresentado na Tabela 6.9. Sem gerência de migração, ao término do tempo de execução (20 s) seriam perdidos 9292 *deadlines*. Com o mecanismo de gerência centralizado, as perdas de *deadline* são reduzidas para 2383 e utilizando-se o mecanismo de gerência distribuído, as perdas mantêm-se em 1288, o que representa uma melhoria de 45.95%. O tempo de estabilização é menor com a abordagem distribuída, e o ganho em comparação a abordagem centralizada para esta aplicação é de 53.12%.



(a) (b)

Figura 6.41: Aplicação MPEG4, perfil de tráfego gerado em um MPSoC 6x5: gerente centralizado (a) e gerentes distribuídos (b)

Tabela 6.9 – Resumo dos experimentos, Aplicação MPEG4

Gerência	Migrações	Tempo de estabilização	Perdas de *deadline*
Sem gerência	0	-	9292
Centralizada	60	1280	2383
Distribuída	60	600	1288
Melhoria		53.12%	45.95%

6.4 Resumo dos experimentos

Na Tabela 6.10 é apresentado um resumo dos experimentos realizados com relação a gerência de migração. Nos casos apresentados, foram ilustradas situações onde múltiplos nodos encontram-se em sobrecarga em diferentes malhas e em diferentes cenários. Buscou-se nestes experimentos destacar as vantagens da utilização de gerentes distribuídos neste tipo de aplicação.

Tabela 6.10 – Experimentos sobre gerentes de migração, múltiplos nodos em sobrecarga

Aplicação	Tamanho do MPSoC	Tarefas	Tempo de Simulação	Migrações		Tempo de estabilização			Perdas de *deadline*		
				GC	GD	GC	GD	Melhoria	GC	GD	Melhoria
Aplicação 1	3x2	18+1	5 s	5	5	175	90	48.57%	8	5	37.50%
Aplicação 2	4x4	36+2	5 s	11	10	275	160	41.81%	14	6	57.14%
Aplicação 3	3x2	12	5 s	5	6	100	75	25.00%	10	9	10.00%
Aplicação 3	4x4	16	5 s	8	8	210	65	69.04%	20	12	40.00%
Aplicação 3	6x5	28	5 s	14	14	290	65	56.66%	33	24	27.27%
Aplicação 4	3x2	27	5 s	9	9	240	205	14.58%	8	8	0.00%
Aplicação 4	4x4	36	5 s	12	12	310	90	69.35%	13	8	38.46%
Aplicação 4	6x5	63	5 s	20	20	275	90	67.27%	18	12	33.33%
MJPEG	3x2	8	10 s	4	4	115	90	21.73%	32	29	9.37%
VOPD	6x5	68	20 s	56	56	1200	400	66.66%	2459	1080	56.07%
MPEG4	6x5	72	20 s	60	60	1280	600	53.12%	2383	1288	45.95%

Experimentos onde apenas um nodo encontra-se em sobrecarga são apresentados na Tabela 6.11. A primeira e segunda aplicações (Aplicação 2 e Aplicação 3) possuem o nodo 9 em sobrecarga, a aplicação MJPEG o nodo 0 e as aplicações VOPD e MPEG4 o nodo 5. Observa-se que neste tipo

de situação, a abordagem distribuída não apresenta uma melhoria significativa sobre a abordagem centralizada para aplicações com um pequeno número de tarefas. As aplicações VOPD e MPEG4, por terem um número significativo de tarefas e diversas dependências entre estas, possuem um melhor desempenho com a abordagem distribuída. Nestes casos, o número de mensagens necessário para sincronizar o gerente centralizado com as tarefas escravas aumenta o tempo de estabilização, e consequentemente influencia nas perdas de *deadline*.

Tabela 6.11 – Experimentos sobre gerentes de migração, um nodo em sobrecarga

Aplicação	Tamanho do MPSoC	Tarefas	Tempo de Simulação	Migrações		Tempo de estabilização			Perdas de *deadline*		
				GC	GD	GC	GD	Melhoria	GC	GD	Melhoria
Aplicação 3	6x5	4	5 s	2	2	65	65	0.00%	2	2	0.00%
Aplicação 4	6x5	9	5 s	3	3	95	90	5.26%	3	3	0.00%
MJPEG	3x2	4	10 s	2	2	45	45	0.00%	0	0	0.00%
VOPD	4x4	17	16 s	15	15	505	350	30.69%	361	311	13.85%
MPEG4	4x4	18	16 s	15	15	520	385	25.96%	363	271	25.34%

6.5 Discussão Sobre os Resultados

Métodos gulosos podem ser eficientemente utilizados para a resolução de alocação de recursos em ambientes MPSoC com um grande número de elementos de processamento ou tarefas. O modelo proposto faz uso do conceito multitarefa, e dessa forma o problema de migração e mapeamento de tarefas torna-se ainda mais complexo quando comparado a modelos monotarefa. Algoritmos comumente utilizados para a resolução do problema de mapeamento de tarefas de forma estática, tais como o *Simulated Annealing*, tendem a ter altos tempos de computação e mesmo com otimizações tornam-se proibitivos para uso em aplicações dinâmicas.

As desvantagens de métodos gulosos estão relacionadas principalmente ao fato da não existência de uma solução ótima de mapeamento, pois busca-se otimizar a solução progressivamente e não de maneira global. Além disso é impossível prever a alocação de recursos de aplicações dinâmicas em tempo de projeto. Este tipo de solução, no entanto, possui uma complexidade computacional reduzida e pode ser aplicada em sistemas com um grande número de tarefas sem ocasionar em longos tempos de execução nas tomadas de decisão.

Com relação aos experimentos realizados, foram apresentadas duas soluções para o gerenciamento de migrações sendo uma delas voltada ao modelo de gerência centralizado, e outra ao modelo de gerência distribuído. O modelo centralizado ilustrado nos resultados é apenas uma das formas de se implementar tal solução. A implementação adotada para o mecanismo centralizado faz uso de um gerente mestre e tarefas escravas em outros nodos da arquitetura, e foi feita desta forma pois adapta-se aos modelos de tarefa e arquitetura propostos. A principal desvantagem encontrada no uso de um gerente centralizado está relacionada às trocas de mensagem necessárias para manter atualizada a informação de todos os nodos. O desempenho desta solução é agravado com o aumento do número de nodos e em virtude da modificação do perfil de execução das tarefas, devido à modificação dos seus parâmetros ou a entrada e saída de tarefas no sistema. Dessa forma, a solução de gerência de migração centralizada tende a ser pouco escalável neste tipo de sistema.

Gerentes distribuídos mostram-se mais eficientes para um grande número de tarefas e elementos de processamento, uma vez que trocas de mensagem entre gerentes tornam-se necessárias apenas em situações de sobrecarga, ou seja, em momentos onde há a necessidade de serem realizadas migrações de tarefa. Além disso, por serem independentes, diversas migrações podem ocorrer em paralelo em áreas distindas do sistema MPSoC sem serem afetadas pelo número de elementos de processamento ou tarefas. O mecanismo de gerência distribuído perde eficiência no momento em que são necessárias muitas iterações do algoritmo devido à não disponibilidade de recursos em nodos próximos. Este tipo de situação é mais clara em MPSoCs altamente sobrecarregados, como é o caso das aplicações sintéticas 3 e 4 com malhas de dimensões 3x2, onde 50% dos elementos de processamento encontram-se sobrecarregados exatamente ao mesmo tempo. Foi observado que com apenas um único nodo em sobrecarga, ambas abordagens possuem um desempenho semelhante tanto em tempo de estabilização quanto em perdas de *deadline* para aplicações com um número reduzido de tarefas. A medida que o número de nodos em sobrecarga aumenta, e a medida em que existam nodos com carga livre na vizinhança dos pontos de sobrecarga, a abordagem distribuída mostra-se mais eficiente. Em virtude das observações realizadas, para a abordagem distribuída não importa o número de nodos em um MPSoC, mas sim o quão próximo recursos de processamento livre podem ser encontrados. Ao estarem próximos (distância em saltos) os elementos de processamento com carga livre suficiente para receberem tarefas de um nodo sobrecarregado, menos mensagens de gerência são necessárias e dessa forma o algoritmo termina em tempo reduzido.

Acredita-se que o número de elementos de processamento em sistemas MPSoC tende a crescer, e dessa forma será mais comum que ocorram situações onde mais de um nodo encontra-se em sobrecarga em sistemas dinâmicos. Apesar disso, devido ao grande número de elementos de processamento é provável que existam recursos disponíveis neste mesmo MPSoC para executar a aplicação, bastando para isso modificar o seu mapeamento.

O modelo proposto assume a execução de tarefas utilizando o conceito de tempo real, e dessa forma supõe-se que devam ser mantidos os critérios de ordem de execução, melhoria na execução da aplicação em casos de sobrecarga e redução nas perdas de *deadline* e por isso o modelo de gerentes distribuído mostra-se mais adequado que o modelo centralizado, pois nos casos avaliados a estabilização do sistema ocorre em menor tempo.

7. CONCLUSÃO E TRABALHOS FUTUROS

Neste Capítulo são apresentadas inicialmente as contribuições do presente trabalho e publicações relacionadas a este. Após, as conclusões são expostas e pontos relacionados a trabalhos futuros são enumerados.

7.1 Contribuições

As contribuições relacionadas ao presente trabalho podem ser descritas como:

- **Revisão do estado da arte** - A primeira contribuição do presente trabalho refere-se à investigação de trabalhos relacionados com o tema proposto e classificação destes. São investigados trabalhos sobre o mapeamento de tarefas (estático e dinâmico) e migração de tarefas, e a partir de sua classificação foi possível observar as lacunas existentes no estado da arte.
- **Definição de um modelo de tarefas e arquitetura** - A segunda contribuição diz respeito à definição de um modelo de tarefas com parâmetros de tempo real e organização MPSoC para a execução de aplicações multitarefa. O modelo de tarefas apresentado ataca pontos pouco explorados por outros trabalhos, como tempo real e caracterização das tarefas que compõem a aplicação. Com relação ao modelo de arquitetura, foi proposta uma solução homogênea.
- **Gerentes distribuídos** - A terceira contribuição refere-se à definição de mecanismos para a implementação de gerentes distribuídos. A gerência de migração não restringe-se apenas às tomadas de decisão sobre o destino das tarefas a serem migradas, mas também aos protocolos de controle de comunicação e processo de migração de maneira coordenada. Foi apresentado o protocolo para controle de comunicação para tarefas com identificação global, o protocolo utilizado pela primitiva de migração para a realização do processo e o algoritmo de espalhamento para a seleção de alvos de migração utilizado pelos gerentes. Uma característica única relacionada ao modelo de tarefas e gerentes de migração é a migração de tarefas sem a necessidade de definição de pontos de migração.
- **Implementação de um sistema operacional preemptivo de tempo real e gerentes de migração distribuídos** - A quarta contribuição deste trabalho diz respeito à implementação dos modelos propostos em um sistema operacional (Hellfire OS). Um conjunto extenso de algoritmos foi implementado em módulos, e a união destes módulos permitiu a construção de um *kernel* com suporte a uma grande quantidade de serviços para aplicações de tempo real. Os principais módulos podem ser enumerados como: (i) camada de abstração de *hardware*; (ii) *kernel*; (iii) escalonador de tarefas multinível; (iv) alocador de memória; (v) biblioteca padrão ANSI C e biblioteca de ponto flutuante; (vi) mecanismos de exclusão mútua; (vii) subsistema de trocas de mensagem e *drivers* para NoC; (viii) gerentes de migração.

- **Extensão de uma ferramenta de simulação** - A quinta contribuição refere-se à extensão da ferramenta de simulação N-MIPS para adaptar-se ao modelo de arquitetura proposto. Esta ferramenta utiliza anotações do *hardware* para emular o comportamento de um ambiente MPSoC completo baseado em NoC além de estimar o tempo de execução de aplicações multiprocessadas.

7.2 Publicações

- MAGALHÃES, Felipe ; LONGHI, Oliver; JOHANN, Sergio F.; AGUIAR, Alexandra; HESSEL, Fabiano P. . **NoC-based Platform for Embedded Software Design: An Extension of the Hellfire Framework**. In: International Symposium on Quality Electronic Design (ISQED), 2012, Santa Clara, USA. International Symposium on Quality Electronic Design (ISQED), 2012. v. 1. p. 1-7.

- ANTUNES, Eduardo B. ; SOARES, Matheus ; JOHANN, Sergio F.; AGUIAR, Alexandra ; Sartori, Marcos ; HESSEL, Fabiano P. ; MARCON, César A. M. . **Partitioning and Dynamic Mapping Evaluation for Energy Consumption Minimization on NoC-Based MPSoC**. In: International Symposium on Quality Electronic Design (ISQED), 2012, Santa Clara, USA. International Symposium on Quality Electronic Design (ISQED), 2012. v. 1. p. 1-7.

- ANTUNES, Eduardo B. ; AGUIAR, Alexandra ; JOHANN, Sergio F. ; Sartori, Marcos ; HESSEL, Fabiano P. ; MARCON, César A. M. . **Partitioning and Mapping on NoC-Based MPSoC: An Energy Consumption Saving Approach**. In: Workshop on Network on Chip Architectures (NoCArC), 2011, Porto Alegre. International Symposium on Microarchitectures (Micro-44), 2011. v. 1. p. 1-6.

- AGUIAR, Alexandra. ; JOHANN, Sergio F.; MAGALHAES, Felipe. ; CASAGRANDE, Thiago. ; HESSEL, Fabiano. . **Hellfire: A Design Framework for Critical Embedded Systems' Applications**. In: IEEE International Symposium on Quality Electronic Design, 2010, San Jose. ISQED - Proceedings of the 11th IEEE International Symposium on Quality Electronic Design. Los Alamitos : IEEE, 2010. v. 1. p. 730-737.

- JOHANN, Sergio F.; AGUIAR, Alexandra. ; MARCON, César A. M.; HESSEL, Fabiano. . **High-Level Estimation of Execution Time and Energy Consumption for Fast Homogeneous MPSoCs Prototyping**. In: IEEE/IFIP International Symposium on Rapid System Prototyping, 2008, Monterey, USA. IEEE/IFIP International Symposium on Rapid System Prototyping (RSP). Los Alamnitos : IEEE Computer Society, 2008. v. 1. p. 27-33.

7.3 Conclusões

Sistemas que fazem uso de arquiteturas heterogêneas apresentam uma série de vantagens, como especialização dos elementos de processamento para determinado domínio e redução no consumo de energia. Apesar destas vantagens, foi identificado que a maioria dos trabalhos encontrados na literatura utilizam o modelo homogêneo. As principais justificativas para a contínua utilização deste modelo são: (i) transferência da complexidade para o *software*; (ii) facilidade no reuso de componentes de *hardware* e redução no tempo de projeto; (iii) flexibilidade da arquitetura para diferentes aplicações.

Com relação ao tipo de mapeamento empregado, existe uma grande quantidade de trabalhos na literatura que abordam tanto o mapeamento estático quando o mapeamento dinâmico e migração de tarefas. Sistemas dinâmicos, no entanto, têm recebido atenção especial nos últimos anos, algo que pode ser observado em virtude do crescente número de trabalhos [83, 11, 78, 7, 44, 16, 9] relacionados a esta abordagem.

Muitos dos trabalhos encontrados na literatura [65, 63, 6, 14, 51, 85, 78, 11, 83, 44], utilizam o conceito de gerente centralizado para realizar migrações ou o mapeamento de tarefas. Entre as vantagens desta abordagem estão a facilidade de manter o estado global do sistema, menor complexidade de implementação e possibilidade de avaliar o estado de todo o sistema. As desvantagens são a concentração de todo o fluxo de mensagens de gerência para uma única região da rede, ponto único de falha e impossiblidade de realizar diversas migrações ou mapeamentos de maneira paralela. Ainda, com o aumento do número de processadores em arquiteturas MPSoC atuais, a utilização de um único gerente tende a ser pouco escalável [63, 2, 89].

A principal proposta do presente trabalho é contribuir para a área de mapeamento dinâmico distribuído e migração de tarefas em sistemas MPSoC homogêneos. Neste trabalho, um modelo de tarefas e gerentes distribuídos foi proposto. Tal modelo emprega conceitos de tempo real que mostra-se relevante para um conjunto grande de aplicações embarcadas. Além disso, o gerenciamento distribuído também proposto objetiva a redução no tempo de migrações em sistemas com um grande número de elementos de processamento e tarefas, sendo crescente o emprego este tipo de sistema em aplicações atuais.

7.4 Trabalhos Futuros

Nesta Seção são enumeradas algumas sugestões para trabalhos futuros.

- Integração do framework CAFES[49] ao framework de desenvolvimento de *software* do Hellfire OS [1]. Parâmetros de caracterização das tarefas podem ser convertidos para ocupação de processamento (já existente no CAFES) de maneira relativamente fácil utilizando políticas de

escalonamento estáticas. Dessa forma, o particionamento e mapeamento inicial podem ser realizados de maneira automatizada.

- Implementação de heurísticas para a redução do número de migrações de tarefas. Podem ser levadas em consideração características dos nodos vizinhos e escolhas relacionadas à seleção das tarefas a serem migradas.

- Implementação de heurísticas que levem em consideração as dependências entre tarefas com o objetivo de aproximar tarefas comunicantes, reduzindo o uso dos canais de comunicação, saturação dos canais e consumo de energia. Estas heurísticas poderiam melhorar a abordagem do algoritmo de espalhamento que realiza estas otimizações porém de maneira ingênua.

- Migração de tarefas que alocam memória dinamicamente em nível de aplicação. Poderia-se manter uma lista de alocações, ou ponteiros para regiões de memória alocadas por determinada tarefa. Com base nesta lista, alocar no nodo destino da migração as regiões de memória com os mesmos tamanhos, e durante a migração copiar cada uma destas para o nodo destino, da mesma forma como é atualmente feito com a pilha das tarefas.

- Redefinição dos modelos de energia atualmente implementados no simulador N-MIPS e integração destes modelos no sistema operacional. Será necessário caracterizar a arquitetura para diferentes tecnologias.

- Inclusão de módulos DMA entre os elementos de processamento e interface de rede, com o intuito de reduzir o *overhead* do sistema operacional em trocas de mensagem.

- Extensão da ferramenta de simulação para incluir o conceito de GALS, além de outros tipos de elemento de processamento.

- Investigação de mecanismos de migração de tarefas e mapeamento dinâmico em sistemas heterogêneos ou heterogêneos com um ISA que pode ser extendido. Elementos de processamento com um conjunto reduzido de registradores também é uma possibilidade. Atualmente, já é possível realizar a compilação da aplicação e do sistema operacional para um conjunto reduzido de registradores, e esta aplicação e sistema operacional executam normalmente em uma versão completa do elemento de processamento, ou em uma versão com a metade do número de registradores (com uma perda de desempenho relativamente baixa).

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Aguiar, A.; Johann, S. F.; Magalhaes, F. G.; Casagrande, T. D.; Hessel, F. "Hellfire: A design framework for critical embedded systems' applications". In: ISQED, 2010, pp. 730–737.

[2] Al Faruque, M.; Krist, R.; Henkel, J. "Adam: Run-time agent-based distributed application mapping for on-chip communication". In: Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE, 2008, pp. 760 –765.

[3] Andrews, M. "Probabilistic end-to-end delay bounds for earliest deadline first scheduling". In: INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, 2000, pp. 603–612.

[4] ARC, I. "Arc mqx rtos", capturado em: http://www.arc.com/ software/mqx/index.html, Outubro 2008.

[5] Benini, L.; Bertozzi, D. "Network-on-chip architectures and design methods", *Computers and Digital Techniques, IEEE Proceedings*, vol. 152–2, mar 2005, pp. 261–272.

[6] Bertozzi, S.; Acquaviva, A.; Bertozzi, D.; Poggiali, A. "Supporting task migration in multi-processor systems-on-chip: a feasibility study". In: Proceedings of the conference on Design, automation and test in Europe: Proceedings, 2006, pp. 15–20.

[7] Braak, T. D.; Hölzenspies, P. K. F.; Kuper, J.; Hurink, J. L.; Smit, G. J. M. "Run-time spatial resource management for real-time applications on heterogeneous mpsocs". In: Proceedings of the Conference on Design, Automation and Test in Europe, 2010, pp. 357–362.

[8] Brião, E. W.; Barcelos, D.; Wagner, F. R. "Dynamic task allocation strategies in mpsoc for soft real-time applications". In: Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 1386–1389.

[9] Cannella, E.; Derin, O.; Meloni, P.; Tuveri, G.; Stefanov, T. "Adaptivity support for mpsocs based on process migration in polyhedral process networks", *VLSI Design*, vol. 2012–Article ID 987209, February 2012, pp. 22–37, special issue on Application-Driven Design of Processor, Memory, and Communication Architectures for MPSoCs.

[10] Carta, S.; Acquaviva, A.; Del Valle, P. G.; Pittau, M.; Atienza, D.; Rincon, F.; De Micheli, G.; Benini, L.; Mendias, J. M. "Multi-Processor Operating System Emulation Framework with Thermal Feedback for Systems-on-Chip". In: 17th ACM Great Lakes Symposium on VLSI (GLSVLSI), 2007, pp. 311–316.

[11] Carvalho, E. L.; Calazans, N. L. V.; Moraes, F. G. "Dynamic task mapping for mpsocs", *IEEE Design and Test of Computers*, vol. 27, 2010, pp. 26–35.

[12] Carvalho, E. L. S. "Mapeamento Dinâmico de Tarefas em MPSoCs Heterogêneos Baseados em NoC", Tese de Doutorado, Pontifícia Universidade Católica do Rio Grande do Sul, PUCRS, Brasil., 2009.

[13] Chou, C.-L.; Marculescu, R. "Incremental run-time application mapping for homogeneous nocs with multiple voltage levels". In: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, 2007, pp. 161–166.

[14] Chou, C.-L.; Marculescu, R. "User-aware dynamic task allocation in networks-on-chip". In: Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 1232–1237.

[15] CMX, S. I. "Cmx-rtx real-time multi-tasking operating system", capturado em: http://www.cmx.com/rtx.htm, Outubro 2008.

[16] Cuesta, D.; Ayala, J. L.; Hidalgo, J. I.; Atienza, D.; Acquaviva, A.; Macii, E. "Adaptive task migration policies for thermal control in mpsocs". In: Proceedings of the 2010 IEEE Annual Symposium on VLSI, 2010, pp. 110–115.

[17] de Magalhães, F. G. "Extensão da Plataforma Hellfire para Utilização de Redes Intra-chip", Trabalho de conclusão, 2010.

[18] Dick, R. P.; Rhodes, D. L.; Wolf, W. "Tgff: task graphs for free". In: Proceedings of the 6th international workshop on Hardware/software codesign, 1998, pp. 97–101.

[19] Faisstnauer, C.; Schmalstieg, D.; Purgathofer, W. "Priority round-robin scheduling for very large virtual environments". In: IEEE Virtual Reality Conference 2000, 2000, pp. 135–142.

[20] Farines, J.-M.; da Silva Fraga, J.; de Oliveira, R. S. "Sistemas de Tempo Real". São Paulo-SP: Second Escola de Computação, IME-USP, 2000.

[21] Gerstlauer, A.; Yu, H.; Gajski, D. "Rtos modeling for system level design". In: Design, Automation and Test in Europe Conference and Exhibition, 2003, 2003, pp. 130–135.

[22] Götz, M.; Xie, T.; Dittmann, F. "Dynamic relocation of hybrid tasks: A complete design flow." In: Sassatelli, G.; Glesner, M.; Bobda, C.; Benoit, P. (Editores), ReCoSoC, 2007, pp. 31–38.

[23] Green Hills Software, I. "Integrity real-time operating system", capturado em: http://www.ghs.com/products/rtos/integrity.html, Outubro 2008.

[24] Hesselink, W. H.; Tol, R. M. "Formal feasibility conditions for earliest deadline first scheduling", Relatório Técnico, 1994.

[25] Holzenspies, P.; Smit, G.; Kuper, J. "Mapping streaming applications on a reconfigurable mpsoc platform at run-time". In: Proceedings of the International Symposium on System-on-Chip (SoC 2007), 2007, pp. 74–77.

[26] Hölzenspies, P. K. F.; Hurink, J. L.; Kuper, J.; Smit, G. J. M. "Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc)". In: Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 212–217.

[27] Hu, J.; Marculescu, R. "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints". In: Proceedings of the conference on Design, automation and test in Europe - Volume 1, 2004, pp. 10–23.

[28] Hu, J.; Marculescu, R. "Energy- and performance-aware mapping for regular noc architectures", *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 24–4, 2005, pp. 551–562.

[29] Jantsch, A. "Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation". San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[30] Jerraya, A.; Tenhunen, H.; Wolf, W. "Multiprocessor systems-on-chips", *Computer*, vol. 38–Issue 7, July 2005, pp. 36– 40.

[31] Johann, S. F. "Estimativa de Desempenho de Software e Consumo de Energia em MPSoCs", Dissertação de Mestrado, 2008.

[32] Johann, S. F.; Aguiar, A.; Marcon, C. A. M.; Hessel, F. P. "High-level estimation of execution time and energy consumption for fast homogeneous mpsocs prototyping". In: RSP '08: Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, 2008, pp. 27–33.

[33] KADAK, P. L. "Amx real time operating system", capturado em: http://www.kadak.com/rtos/rtos.htm, Outubro 2008.

[34] Le Moigne, R.; Pasquier, O.; Calvez, J.-P. "A generic rtos model for real-time systems simulation with systemc". In: Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, 2004, pp. 82–87 Vol.3.

[35] Le Moigne, R.; Pasquier, O.; Calvez, J.-P. "A generic rtos model for real-time systems simulation with systemc". In: DATE '04: Proceedings of the conference on Design, automation and test in Europe, 2004, pp. 76–82.

[36] Lehoczky, J.; Sha, L.; Ding, Y. "The rate monotonic scheduling algorithm: exact characterization and average case behaviour", *IEEE Real-Time Systems Symposium*, 1989, pp. 166–171.

[37] Lei, T.; Kumar, S. "Algorithms and tools for network on chip based system design". In: Proceedings of the 16th symposium on Integrated circuits and systems design, 2003, pp. 163–.

[38] Lei, T.; Kumar, S. "A two-step genetic algorithm for mapping task graphs to a network on chip architecture". In: Proceedings of the Euromicro Symposium on Digital Systems Design, 2003, pp. 180–186.

[39] Leung, J. Y.; Whitehead, J. "On the complexity of fixed-priority scheduling of periodic, real-time tasks", *Performance Evaluation*, vol. 2–4, 1982.

[40] Liu, C. L.; Layland, J. "Scheduling algorithms for multiprogramming in a hard real-time environment", *Journal of the ACM*, vol. 20–1, 1973, pp. 46–61.

[41] Longhi, O. B. "Incrementando o Simulador do Ambiente HellfireFW para Suportar Sistemas NoC", Trabalho de conclusão, 2010.

[42] Lyonnard, D.; Yoo, S.; Baghdadi, A.; Jerraya, A. "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip". In: Design Automation Conference, 2001. Proceedings, 2001, pp. 518 – 523.

[43] Määttä, S.; Indrusiak, L. S.; Ost, L.; Möller, L.; Nurmi, J.; Glesner, M.; Moraes, F. "Validation of executable application models mapped onto network-on-chip platforms". In: SIES, 2008, pp. 118–125.

[44] Mandelli, M.; Amory, A.; Ost, L.; Moraes, F. G. "Multi-task dynamic mapping onto noc-based mpsocs". In: Proceedings of the 24th symposium on Integrated circuits and systems design, 2011, pp. 191–196.

[45] Manolache, S.; Eles, P.; Peng, Z. "Fault and energy-aware communication mapping with guaranteed latency for applications implemented on noc". In: Proceedings of the 42nd annual Design Automation Conference, 2005, pp. 266–269.

[46] Marchesan Almeida, G.; Sassatelli, G.; Benoit, P. "An adaptive message passing mpsoc framework", *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1–21.

[47] Marcon, C.; Borin, A.; Susin, A.; Carro, L.; Wagner, F. "Time and energy efficient mapping of embedded applications onto nocs". In: Proceedings of the 2005 Asia and South Pacific Design Automation Conference, 2005, pp. 33–38.

[48] Marcon, C.; Calazans, N.; Moraes, F.; Susin, A.; Reis, I.; Hessel, F. "Exploring noc mapping strategies: An energy and timing aware technique". In: Proceedings of the conference on Design, Automation and Test in Europe - Volume 1, 2005, pp. 502–507.

[49] Marcon, C.; Calazans, N.; Moreno, E.; Moraes, F.; Hessel, F.; Susin, A. "Cafes: A framework for intrachip application modeling and communication architecture design", *J. Parallel Distrib. Comput.*, vol. 71–5, Maio 2011, pp. 714–728.

[50] Marcon, C. A.; Moreno, E. I.; Calazans, N. L. V.; Moraes, F. G. "Evaluation of algorithms for low energy mapping onto nocs", *2007 IEEE International Symposium on Circuits and Systems*, 2007, pp. 389–392.

[51] Mehran, A.; Khademzadeh, A.; Saeidi, S. "Dsm: A heuristic dynamic spiral mapping algorithm for network on chip", *IEICE Electronics Express*, vol. 5–13, 2008, pp. 464–471.

[52] Mehran, A.; Saeidi, S.; Khademzadeh, A.; Afzali-Kusha, A. "Spiral: A heuristic mapping algorithm for network on chip", *IEICE Electronics Express*, vol. 4–15, 2007, pp. 478–484.

[53] Micrium. "Microc/os-ii kernel overview", capturado em: http://www.micrium.com/products/rtos/kernel/rtos.html, Outubro 2008.

[54] Mignolet, J.-Y.; Nollet, V.; Coene, P.; Verkest, D.; Vernalde, S.; Lauwereins, R. "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip". In: DATE '03: Proceedings of the conference on Design, Automation and Test in Europe, 2003, pp. 10–16.

[55] Mihal, A.; Keutzer, K. "Mapping concurrent applications onto architectural platforms". Hingham, MA, USA: Kluwer Academic Publishers, 2003, pp. 39–59.

[56] Milojevic, D.; Montperrus, L.; Verkest, D. "Power dissipation of the network-on-chip in a system-on-chip for mpeg-4 video encoding", *2007 IEEE Asian SolidState Circuits Conference*, 2007, pp. 392–395.

[57] MIPS, T. "Mips32 architecture for programmers, vol ii: The mips32 instruction set". Capturado em: http://www.mips.com/products/product-materials/processor/mips-architecture/, 2011.

[58] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: an infrastructure for low area overhead packet-switching networks on chip", *Integr. VLSI J.*, vol. 38–1, 2004, pp. 69–93.

[59] Mulas, F.; Pittau, M.; Buttu, M.; Carta, S.; Acquaviva, A.; Benini, L.; Atienza, D.; De Micheli, G. "Thermal balancing policy for streaming computing on multiprocessor architectures", *Design, Automation and Test in Europe, 2008. DATE '08*, March 2008, pp. 734–739.

[60] Murali, S.; Coenen, M.; Radulescu, A.; Goossens, K.; De Micheli, G. "Mapping and configuration methods for multi-use-case networks on chips". In: Proceedings of the 2006 Asia and South Pacific Design Automation Conference, 2006, pp. 146–151.

[61] Murali, S.; De Micheli, G. "Bandwidth-constrained mapping of cores onto noc architectures". In: Proceedings of the conference on Design, automation and test in Europe, 2004, pp. 20–26.

[62] Murali, S.; De Micheli, G. "Sunmap: a tool for automatic topology selection and generation for nocs". In: Proceedings of the 41st annual Design Automation Conference, 2004, pp. 914–919.

[63] Ngouanga, A.; Sassatelli, G.; Torres, L.; Gil, T.; Soares, A.; Susin, A. "A contextual resources use: a proof of concept through the apaches' platform", *Design and Diagnostics of Electronic Circuits and Systems*, vol. 0, 2006, pp. 42–47.

[64] Nollet, V.; Avasare, P.; Mignolet, J.-Y.; Verkest, D. "Low cost task migration initiation in a heterogeneous mp-soc". In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, 2005, pp. 252–253.

[65] Nollet, V.; Marescaux, T.; Avasare, P.; Mignolet, J.-Y. "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles". In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, 2005, pp. 234–239.

[66] Orsila, H.; Kangas, T.; Salminen, E.; Hamalainen, T. D. "Parameterizing simulated annealing for distributing task graphs on multiprocessor socs, international symposium on system-on-chip 2006", Relatório Técnico, Tampere University of Technology (TUT).

[67] Orsila, H.; Kangas, T.; Salminen, E.; Hamalainen, T. D.; Hannikainen, M. "Automated memory-aware application distribution for multi-processor system-on-chips", *J. Syst. Archit.*, vol. 53, November 2007, pp. 795–815.

[68] Ost, L. C. "Abstract Models of NoC-Based MPSoCs for Design Space Exploration", Tese de Doutorado, Pontifícia Universidade Católica do Rio Grande do Sul, PUCRS, Brasil., 2010.

[69] Ozturk, O.; Kandemir, M.; Son, S. W.; Karakoy, M. "Selective code/data migration for reducing communication energy in embedded mpsoc architectures". In: Proceedings of the 16th ACM Great Lakes symposium on VLSI, 2006, pp. 386–391.

[70] Park, K. "A heuristic approach to task assignment optimization in distributed systems", *IEEE International Conference on Computational Cybernetics and Simulation*, vol. 2–1, 1997, pp. 1838 – 1842.

[71] Peterson, G. L. "Myths about the mutual exclusion problem", *Information Precessing Letters*, vol. 12–3, 1981, pp. 115–116.

[72] Pittau, M.; Alimonda, A.; Carta, S.; Acquaviva, A. "Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation". In: ESTImedia, 2007, pp. 59–64.

[73] Posadas, H.; Ádamez, J.; Sánchez, P.; Villar, E.; Blasco, F. "Posix modeling in systemc". In: Proceedings of the 2006 Asia and South Pacific Design Automation Conference, 2006, pp. 485–490.

[74] Rasmussen, R. V.; Rasmussen, R. V.; Trick, M. A.; Trick, M. A. "Round robin scheduling - a survey", Relatório Técnico, European Journal of Operational Research, 2006.

[75] Rhee, C.-E.; Jeong, H.-Y.; Ha, S. "Many-to-many core-switch mapping in 2-d mesh noc architectures". In: Proceedings of the IEEE International Conference on Computer Design, 2004, pp. 438–443.

[76] Rhoads, S. "Plasma - most mips i(tm) opcodes". Capturado em http://opencores.org/, 2011.

[77] Schirner, G.; Dömer, R. "Introducing preemptive scheduling in abstract rtos models using result oriented modeling". In: DATE '08: Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 122–127.

[78] Schranzhofer, A.; Chen, J.-J.; Santinelli, L.; Thiele, L. "Dynamic and adaptive allocation of applications on mpsoc platforms". In: Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific, 2010, pp. 885 –890.

[79] Schwiegelshohn, U.; Yahyapour, R. "Analysis of first-come-first-serve parallel job scheduling". In: In Proceedings of the 9th SIAM Symposium on Discrete Algorithms, 1998, pp. 629–638.

[80] Shaout, A.; Mattar, K.; Elkateeb, A. "An ideal api for rtos modeling at the system abstraction level". In: Mechatronics and Its Applications, 2008. ISMA 2008. 5th International Symposium on, 2008, pp. 1–6.

[81] Shen, H.; Petrot, F. "Novel task migration framework on configurable heterogeneous mpsoc platforms". In: Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific, 2009, pp. 733–738.

[82] Singh, A.; Jigang, W.; Prakash, A.; Srikanthan, T. "Efficient heuristics for minimizing communication overhead in noc-based heterogeneous mpsoc platforms". In: Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on, 2009, pp. 55 –60.

[83] Singh, A. K.; Srikanthan, T.; Kumar, A.; Jigang, W. "Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms", *J. Syst. Archit.*, vol. 56, July 2010, pp. 242–255.

[84] Srinivasan, K.; Chatha, K. S. "A technique for low energy mapping and routing in network-on-chip architectures". In: Proceedings of the 2005 international symposium on Low power electronics and design, 2005, pp. 387–392.

[85] Wildermann, S.; Ziermann, T.; Teich, J. "Run-time mapping of adaptive applications onto homogeneous NoC-based reconfigurable architectures". In: The 2009 International Conference on Field-Programmable Technology (FPT'09), 2009, pp. 514–517.

[86] Wronski, F.; Brião, E. W.; Wagner, F. R. "Evaluating energy-aware task allocation strategies for mpsocs". In: Kleinjohann, B.; Kleinjohann, L.; Machado, R. J.; Pereira, C. E.; Thiagarajan, P. S. (Editores), DIPES, 2006, pp. 215–224.

[87] Yoo, S.; Nicolescu, G.; Gauthier, L.; Jerraya, A. "Automatic generation of fast timed simulation models for operating systems in soc design". In: Proceedings of the conference on Design, automation and test in Europe, 2002, pp. 620–625.

[88] Zeferino, C. A. "Arquiteturas e Modelos para Avaliação de Área e Desempenho", Tese de Doutorado, Universidade Federal do Rio Grande do Sul, UFRGS, Brasil., 2003.

[89] Zipf, P.; Sassatelli, G.; Utlu, N.; Saint-Jean, N.; Benoit, P.; Glesner, M. "A decentralised task mapping approach for homogeneous multiprocessor network-on-chips", *Int. J. Reconfig. Comput.*, vol. 2009, 2009, pp. 1–14.