PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MODELO DE BALANCEAMENTO DE CARGA ATRAVÉS DE MIGRAÇÃO DE TAREFAS EM MPSOC'S DE TEMPO REAL

ALEXANDRA DA COSTA PINTO DE AGUIAR

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

Orientador: Prof. Dr. Fabiano Passuelo Hessel

Porto Alegre, Dezembro 2008

Dados Internacionais de Catalogação na Publicação (CIP)

A282m Aguiar, Alexandra da Costa Pinto de.
Modelo de balanceamento de carga através de migração de tarefas em MPSoC's de tempo real / Alexandra da Costa Pinto de Aguiar. – Porto Alegre, 2009.
89 f.

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

1. Informática. 2. Multiprocessadores. 3. Sistemas Distribuídos. 4. Migração de Tarefas. 5. Processamento de Tempo Real. 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 DISSERTAÇÃO DE MESTRADO

Dissertação intitulada **"Modelo de Balanceamento de Carga Através de Migração de Tarefas em MPSoC's de Tempo Real"**, apresentada por Alexandra da Costa Pinto de Aguiar, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas Embarcados e Sistemas Digitais, aprovada em 11/03/09 pela Comissão Examinadora:

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

Prof. Dr. Fernando Gehm Moraes - — PPGCC/PUCRS

Prof. Dr. Flávio Rech Wagner – — UFRGS

Homologada em..19.../..05.../..2009.., conforme Ata No..008/09.. pela Comissão Coordenadora.

Prof. Dr. Fernando Gehm Moraes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

Dedico este trabalho a minha família e a Deus.

Agradecimentos

Primeiramente a Deus e à minha família. Por tudo, simplesmente, tudo. O apoio, a alegria, a tristeza, o conforto, os xingamentos - tudo. Sempre presente quando preciso, sempre preocupados com minha integridade física e mental, sempre tão família. Difícil imaginar qualquer etapa da longa jornada que culmina neste trabalho sem a presença de vocês. Oliveira, Sonia, Samuel: vocês são tudo para mim e eu AMO VOCÊS! São a razão de absolutamente TUDO. Só tenho a agradecer a vocês.

Aos meus colegas que viraram amigos, aos que eram amigos e viraram colegas. A todos que, de alguma forma - na forma de riso, de conversa paralela, de um café no bar ou de ajuda direta mesmo, contribuíram para que o trabalho pudesse ser concluído com sucesso: Sérgio (ô), Luciano (bá), Andriele (é), Marcelo (capaz), Guindani (até tu), Laerte (hihi), Schonarth (eta), Samuka (ae), Carlos (@s) e mais uma galera do GSE e do GAPH (para não fazer injustiça com ninguém): muito obrigada pelo apoio de vocês. Aos que fizeram parte dessa caminhada pelos mais variados motivos: as malucas do apê (Dani, Fê, Maga e Su), Carine, Elis, Cahê, Serginho, Josué, Herrmann, Adilson e mais um mooooooooooooo-ooooooonte de amigos, primos, tios, manos e sobrinhas cujos nomes não me vêm na cabeça... desculpe! É o tardar da hora!

Aos meus cães rio pardenses (Thor, Coragem, Filotinho - o filho da Filó e Gurizinho) e ao meu bebê, Zé "taz" Bob - um cocker amável, simpático e extremamente ativo. Ele... que me fez levantar tantas vezes no meio da noite, limpar tantas coisas, dar tanto banho, comer tanto pêlo... mas que ainda assim, foi capaz de ficar comigo mesmo nas noites mal dormidas, acordar de madrugada sempre que necessário ou ficar sozinho nos momentos em que não pudia estar com ele. *Au* pra ti tb...

Aos diversos professores que fizeram parte da caminhada e, em especial, ao meu orientador Fabiano Hessel que soube guiar nos momentos de incerteza, apostar quando foi necessário e compreender quando foi mais necessário ainda. Muito obrigada pelo apoio e acolhimento nessa etapa tão importante. Aos eternos mestres, Rafael e Tatiana.. por tudo que vocês sabem que representam para mim :-)

Às universidades PUCRS e UNISC por fazerem parte do caminho e algumas marcas alternativas... como NET, Imortal Tricolor, Nescau, Nestle, Leite Moça, Garoto, Coca Cola, Guaraná Fruki e Skol (que foi?!)... que souberam dar o devido apoio nos momentos em que precisei de outras coisas... tudo isso com marca registrada :)

Aos meus 30 GB de MP3 que fizeram com que eu aumentasse meu conhecimento musical e pudesse, ao mesmo tempo, ter uma trilha sonora diversa o suficiente pra tudo o que eu sempre fiz... além de garantir minha sanidade mental quando o mundo à minha volta parecia pegar fogo.

Obrigada aos que fizeram, aos que fazem e boa sorte aos que ainda vão fazer parte desta história (Até a próxima... 4 anos, espero :-). *No one said it was easy, no one ever said it would be this hard* (The Scientist, Coldplay).

Resumo

Sistemas embarcados, em muitos casos, utilizam mais de um processador formando arquiteturas multiprocessadas homogêneas ou heterogêneas. Sistemas multiprocessados que sejam implementados em um único *chip* são denominados de MPSoC's. Assim como em sistemas multiprocessados de propósito geral, a utilização de técnicas de balanceamento de carga também pode trazer benefício no âmbito dos sistemas embarcados multiprocessados, uma vez que ajudam a distribuir de forma equilibrada as tarefas do sistema entre os diversos elementos de processamento existentes. Essa distribuição justa é um aspecto chave uma vez que pontos sobrecarregados devem ser evitados por apresentar, em geral, as maiores temperaturas do *chip*. Pontos superaquecidos de um *chip* podem ter mecanismos de falha acelerados e, por esse motivo, devem ser evitados. Além disso, técnicas dinâmicas de balanceamento de carga têm a possibilidade de lidar com a dinamicidade dos sistemas embarcados atuais, tais como equipamentos multimídia, onde o próprio usuário pode acrescentar tarefas ao sistema. Assim, este trabalho tem como objetivo propor um modelo de balanceamento de carga que utilize a técnica de migração de tarefas em um MPSoC que contemple, também, tarefas de tempo real. O modelo proposto utiliza gerenciadores locais e um gerenciador global e foi implementado sobre uma plataforma MPSoC real onde teve seu funcionamento validado, verificando-se uma diminuição na perda de *deadlines* bem como um equilíbrio maior do sistema ao longo de seu tempo de vida.

Palavras-chave: balanceamento de carga, migração de tarefas, MPSoC, tempo real

Abstract

Embedded systems, in many cases, use more than one processor producing either homogenous or heterogeneous multiprocessed architectures. Multiprocessed systems implemented in a single chip are known as MPSoC's. Similarly to what happens in general purpose multiprocessed systems, the use of load balancing techniques can also be positive in the multiprocessed embedded systems' area, since these techniques are helpful to distribute, in a more balanced manner, the tasks of the system among its several processing elements. The fair distribution provided by these techniques is a key aspect, since overloaded points must be avoided because they tend to present the highest chip temperature levels. These high temperature levels may also lead faster to permanent chip failure and must be avoided. Besides that, dynamic load balancing techniques are capable of dealing with the dynamic behavior presented in current embedded systems, such as multimedia equipment, where the user himself can add new tasks to the system. The main objective of this work is to discuss and present a novel load balancing model through the task migration technique in MPSoC's that contain real time tasks. The proposed model uses local and global managers and was implemented over a real MPSoC platform in which it was validated. There, it was possible to observe that deadline misses were decreased and the load balance of the system was reached throughout its life time.

Keywords: load balancing, task migration, MPSoC, real-time

Lista de Figuras

Lista de Abreviaturas

AA	Above Average
AV	Average
BA	Below Average
CMP	Chip Multiprocessor
CPU	Central Processing Unit
DVS	Dynamic Voltage Scaling
EDF	Earliest Deadline First
EP	Elemento de Processamento
E/S	Entrada/Saída
FCFS	First Come First Served
FPGA	Field Programmable Gate Array
GSE	Grupo de Sistemas Embarcados
HAL	Hardware Abstraction Layer
ISR	Interrupt Service Routine
ISS	Instruction Set Simulator
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
OL	OverLoaded
RM	Rate Monotonic
RTOS	Real-Time Operating System
RTS	Real-Time Systems
SE	Sistemas Embarcados
SO	Sistema operacional
SoC	System-on-Chip
TCB	Task Control Block

UART	Universal Asynchronous Receiver/Transmitter
UL	UnderLoaded
VHDL	Very-High-Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration
WCET	Worst Case Execution Time
XML	eXtensible Markup Language
XUP	Xilinx University Program

Sumário

1 Introdução

Atualmente, os Sistemas Embarcados (SE) são utilizados em uma vasta gama de produtos disponíveis no mercado estando presentes em eletrônicos, eletrodomésticos e em itens empregados na indústria automotiva. Esses sistemas caracterizam-se principalmente por atender aos requisitos específicos de uma determinada aplicação apresentando, na maioria dos casos, restrições de área e de consumo de energia. Além disso, sua presença no cotidiano das pessoas deve aumentar, uma vez que seus custos para o usuário final têm diminuído [1].

Em muitos desses sistemas estão presentes restrições temporais onde o tempo de resposta de uma determinada execução do sistema é tão importante quanto seu resultado lógico. Sistemas que apresentam esse tipo de comportamento são conhecidos como Sistemas de Tempo Real (do inglês, *Real-Time Systems* - RTS), e podem ser divididos, com base na rigorosidade das suas restrições de tempo, em duas grandes categorias: *hard real-time*, para as mais rigorosas, e *soft real-time*, para as mais brandas [2].

Por conseguinte, é comum que RTS's utilizem um Sistema Operacional (SO) específico, denominado de Sistema Operacional de Tempo Real (do inglês, *Real-Time Operating System* - RTOS) responsável por viabilizar seu gerenciamento. Além de prover funcionalidades comuns a SO's, como temporizadores e gerenciamento de interrupções e tarefas, uma das principais funções dos RTOS's é tentar garantir que a execução de uma determinada tarefa ocorra conforme a restrição temporal atribuída a ela [2]. Assim, o escalonador de tarefas de um RTOS não é responsável apenas por gerenciar a ordem de execução, mas também por assegurar o cumprimento das restrições temporais de um determinado conjunto de tarefas. Esse mecanismo é implementado com base em uma das diversas políticas de escalonamento existentes e que se adapte melhor ao tipo de aplicação alvo.

Além de poder apresentar restrições temporais, é desejável, por questões de custo e desempenho, que sistemas embarcados sejam implementados através da integração de todos os componentes necessários para sua execução em uma única pastilha (*chip*), formando um sistema comumente chamado de *System-on-Chip* (SoC). Um SoC permite a utilização de componentes heterogêneos, tais como CPU's (do inglês, *Central Processing Unit*), memórias e barramentos, entre outros. Além disso, é possível que um único SoC seja formado por mais de um elemento de processamento (EP). SoC's que empregam vários elementos de processamento em um único

chip são denominados de SoC multiprocessados ou MPSoC (do inglês, *Multiprocessor System-on-Chip*).

Os MPSoC's poderiam, erroneamente, ser comparados aos *chips* multiprocessados (do inglês, *Chip Multiprocessor* - CMP), uma vez que ambos possuem vários elementos de processamento [3]. Porém, CMP's são sistemas cuja abordagem de desenvolvimento integra vários processadores em um único *chip*, aproveitando a crescente densidade de transistores disponível no mercado [3]. MPSoC's, por sua vez, são arquiteturas customizadas que tentam equilibrar as vantagens oferecidas pela tecnologia VLSI (do inglês, *Very Large Scale of Integration*) com as necessidades da aplicação, que geralmente incluem baixo consumo de energia [3].

Dentro desse contexto, é importante observar que algumas características mais genéricas de sistemas multiprocessados computacionais de propósito geral podem ser observadas nos MPSoC's, entre as quais destaca-se a necessidade de um balanceamento de carga equilibrado entre os diversos elementos de processamento participantes do sistema. Essa característica já foi amplamente estudada em sistemas distribuídos [4], [5], [6] e, com o advento dos MPSoC's, volta a ser objeto de pesquisa, tanto no âmbito acadêmico quanto no industrial [7].

Assim, o objetivo deste trabalho é apresentar o desenvolvimento de um modelo para balanceamento de carga de um sistema MPSoC de tempo real, tornando sua carga o mais homogênea possível através de migrações dinâmicas de tarefas entre os diversos elementos de processamento envolvidos no processo. Sistemas cuja distribuição de carga seja homogênea possuem diversos benefícios, como a melhora da utilização dos elementos do sistema e a minimização dos atrasos de comunicação, além de ajudar no cumprimento dos *deadlines* em MPSoC's cujos elementos estejam inicialmente sobrecarregados.

1.1 Motivação

MPSoC's estão sendo cada vez mais empregados tanto na indústria como no meio acadêmico e a tendência é que ainda haja um grande crescimento na utilização desses sistemas [8]. Se por um lado o aumento contínuo do desempenho dos microprocessadores estimulou, ao longo dos anos, o surgimento de máquinas monoprocessadas cada vez mais poderosas, por outro lado, o fato de essa ser uma abordagem que normalmente acarreta em um consumo de energia crescente tem sido um limitador. Dessa forma, uma das soluções que vem sendo adotadas é a utilização de mais de um elemento de processamento com freqüência de operação mais baixa e menor consumo de energia, para conseguir manter o compromisso entre desempenho e consumo de energia.

O fato de MPSoC's terem um futuro promissor implica que novos problemas intrínsecos a esse tipo de sistema terão de ser investigados. Por se tratarem de sistemas multiprocessados, é possível uma comparação em alto nível com sistemas multiprocessados de propósito geral. Nessa comparação, um dos problemas comuns aos dois tipos de sistema é a questão do balanceamento de carga. Investigar antigas técnicas de balanceamento de carga como migração de tarefas, porém, dentro desse novo contexto, é um trabalho bastante desafiador, assim como verificar se outras abordagens mais recentes podem ou não cumprir o mesmo papel também o é.

Em muitos casos, o próprio usuário de dispositivos multimídia, como tocadores de música ou vídeo ou ainda de dispositivos de comunicação como celulares, tem a possibilidade de agregar novas aplicações com comportamento não previsto pelo projetista do equipamento. Assim, torna-se necessária a disponibilização de mecanismos que possam lidar, dinamicamente, com esse aumento e diversificação das cargas computacionais dos MPSoC's.

Através da utilização de técnicas de migração é possível distribuir a carga computacional de um sistema de forma mais homogênea, evitando pontos críticos sobrecarregados. Tipicamente, as temperaturas são as mais altas de todo o circuito onde há sobrecarga de execução, sendo que, pontos superaquecidos podem acelerar mecanismos de falha, tais como eletromigração, *stress migration* e *dielectric breakdown* [9]. Isso possibilita a ocorrência de falhas permanentes no dispositivo [10]. Portanto, é possível afirmar que o balanceamento de carga do sistema pode facilitar o gerenciamento das questões térmicas do *chip*, movendo a carga dos elementos de processamento mais quentes (com muito processamento) para aqueles mais frios (com pouco processamento) [11].

Além disso, é possível, através do balanceamento de carga com migrações de tarefas, prevenir-se o descumprimento de *deadline* de tarefas de tempo real *hard* principalmente quando iniciadas em um processador cuja carga de trabalho já representar seu limite máximo. Nesses casos, por exemplo, pode ocorrer a migração de tarefas para outros EP's, liberando espaço no EP requisitado pela tarefa de tempo real *hard*.

Mesmo com todos os potenciais benefícios que a utilização de mecanismos de migração de tarefas podem trazer ao projeto de um MPSoC, seu uso efetivo e em larga escala ainda não é realidade [7] e é dentro desse contexto que o presente trabalho está integrado.

1.2 Objetivo

O objetivo principal deste trabalho é apresentar, discutir e validar um modelo de balanceamento de carga através de migração de tarefas que seja aplicável em um MPSoC com tarefas de tempo real. A validação desse modelo ocorreu através de sua implementação em um *kernel* capaz de ser executado em processadores que possuam o conjunto de instruções MIPS I (como o Plasma [12], por exemplo). O modelo foi implementado como um módulo do *kernel* e a validação ocorreu de duas formas: prototipação em FPGA e execução em um ISS (do inglês, *Instruction Set Simulator*), o que possibilita o teste do modelo em um nível mais alto de abstração com um fator de erro conhecido. Além disso, o *kernel* oferece a geração de *logs* quando executado no ISS. Esses *logs*, referentes à execução do sistema, podem, então, ser visualizados em uma ferramenta gráfica desenvolvida especialmente para a depuração do sistema, onde se pode acompanhar a execução das tarefas nos diversos EP's ao longo do tempo.

Os resultados foram obtidos principalmente para analisar: i) a capacidade do modelo de, dinamicamente, balancear a carga do sistema; ii) a relação custo/benefício da utilização do modelo; iii) a capacidade do modelo em lidar com situações envolvendo tarefas de tempo real.

1.3 Organização do texto

Este trabalho está organizado da seguinte maneira: o próximo capítulo apresenta os conceitos principais acerca de balanceamento de carga para sistemas multiprocessados tanto de propósito geral quanto para sistemas embarcados seguido do capítulo de revisão bibliográfica dos trabalhos mais significativos em relação ao estudo proposto. Os modelos dos componentes necessários para a utilização do modelo proposto encontram-se no Capítulo 4 seguido do Capítulo 5 que apresenta, em detalhes, o modelo de balanceamento de carga através de migração de tarefas. Já o Capítulo 6 traz os resultados para os diversos casos de teste aplicados. Finalmente, o Capítulo 7 exibe as conclusões do trabalho e diretrizes para futuros estudos.

2 Balanceamento de carga

Este capítulo aborda os diversos aspectos envolvidos no balanceamento de carga de sistemas multiprocessados. É importante frisar que o conceito de balanceamento de carga pode ser aplicado tanto em sistemas multiprocessados de propósito geral quanto em sistemas multiprocessados de propósito específico, como os MPSoC's. Isso ocorre devido ao fato de ambas as abordagens possuírem em comum a possibilidade de divisão da carga de trabalho do sistema entre os diversos EP's que as compõe, apesar de divergirem sobre aspectos fundamentais de sua implementação em função, principalmente, da diferença de propósito existente entre elas. Assim, itens como conceituação, taxonomia e aspectos envolvendo a implementação de algoritmos para balanceamento de carga são explorados, embasando o posicionamento crítico do modelo proposto a ser apresentado ao longo do texto.

2.1 Conceito

Estudos relacionados ao balanceamento de carga vêm sendo realizados ao longo de muitos anos na computação distribuída e paralela de propósito geral [13], [14], [6], [5], [4]. O conceito propriamente dito de balanceamento de carga diz respeito à divisão da quantidade de trabalho entre os diversos nodos[1] participantes de um sistema. O objetivo principal é garantir que nenhum nodo (ou o menor número possível de nodos) esteja em uma condição de sobrecarga ou de subutilização. Segundo Sinha [13], as principais vantagens obtidas ao garantir o equilíbrio da carga de trabalho total do sistema são: (i) melhorar a utilização de todos os nodos; (ii) aumentar o desempenho total; (iii) minimizar atrasos de comunicação, uma vez que nodos em sobrecarga tendem a demorar mais tempo para atender e processar as requisições pendentes de comunicação.

Ainda de acordo com Sinha [13], abordagens que visam o equilíbrio do balanceamento de carga devem tentar garantir o desempenho total do sistema de acordo com alguma métrica específica sua. Nesse caso, pode-se considerar o desempenho a partir do *ponto de vista do*

[1] Neste capítulo nodo é sinônimo de elemento de processamento (EP).

usuário onde a métrica equivale ao tempo de resposta dos processos[2] envolvidos. Por outro lado, quando se considera o desempenho a partir do *ponto de vista dos recursos*, a métrica prioriza a vazão total do sistema. Uma vez que a maximização do uso de recursos é compatível com o aumento da vazão do sistema, basicamente todos os algoritmos de balanceamento de carga objetivam maximizar essa vazão.

2.2 Taxonomia

Os algoritmos de balanceamento de carga podem ser classificados em políticas globais ou locais [14]. As políticas locais preocupam-se individualmente com cada nodo do sistema e com seu escalonamento, enquanto que as globais levam em consideração questões de alocação de processos nos nodos disponíveis. Além disso, a taxonomia também prevê políticas estáticas e dinâmicas, divididas de acordo com o momento no qual são tomadas as decisões de escalonamento e alocação. A taxonomia completa está exibida na figura 1.

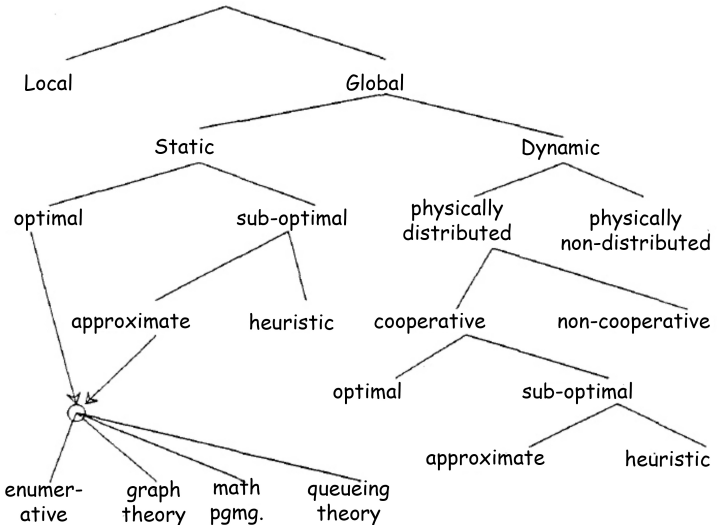

Figura 1 – Taxonomia para escalonamento distribuído de processos
Fonte:Adaptado de [14]

A taxonomia apresentada por Sinha [13] é semelhante a de Casavant [14] e ambas apresentam os algoritmos estáticos e os dinâmicos. O balanceamento de carga estático é um método pelo qual a carga de trabalho de cada EP é distribuída no início da execução do sistema. A principal vantagem desse método é a redução da comunicação entre o gerenciador de carga e os

[2]Neste capítulo, considera-se processo como sendo uma atividade computacional realizada por um nodo.

outros componentes do sistema. De forma controversa, a principal desvantagem é justamente a sua falta de capacidade de adaptação às características imprevisíveis que possam ocorrer em tempo de execução, degradando o desempenho da aplicação como um todo [15]. Entre os estáticos, Sinha [13] apresenta uma nova divisão, categorizando algoritmos determinísticos e probabilísticos, sendo que os do primeiro grupo usam informações sobre as propriedades dos nodos bem como as características dos processos para - de forma determinística - alocá-los nos nodos. Já o segundo grupo, usa informações de atributos estáticos do sistema, como número de nodos, capacidade de processamento de cada nodo, topologia de rede, entre outros para formular regras simples para a alocação dos processos.

Por outro lado, o balanceamento de carga dinâmico determina a distribuição da carga de trabalho em tempo de execução. Para tanto, é determinado um nodo mestre que realiza a alocação de novos processos para os membros do sistema dependendo de informações de carga recentemente coletadas. Uma vez que a distribuição da carga de trabalho é feita em tempo de execução, pode oferecer melhores resultados ao custo de um *overhead* associado em função das necessidades de comunicação [15], significando que se deve verificar a relação de compromisso entre desempenho e comunicação de forma a obter os melhores resultados [16].

Os algoritmos dinâmicos são ainda divididos em centralizados e distribuídos, de acordo com o responsável pela manutenção do balanceamento de carga no sistema. Na abordagem centralizada, informações acerca do estado do sistema são reunidas em um único nodo - denominado de servidor centralizado - que concentra todas as decisões sobre o escalonamento. As decisões são tomadas mais rapidamente uma vez que esse servidor tem conhecimento sobre a carga e as necessidades dos nodos, enquanto que na abordagem distribuída todos devem chegar a um consenso. Para que isso ocorra, os diversos nodos, periodicamente, enviam mensagens de atualização de sua situação para o servidor central. O principal problema da abordagem centralizada é o da confiabilidade, uma vez que o balanceamento é necessariamente dependente de um único nodo. De forma a amenizar esse problema, uma alternativa é manter diversos nodos responsáveis por supervisionar o nodo central e, em caso de falha, iniciar um novo servidor para não prejudicar o sistema [17].

Já a abordagem distribuída não se limita a utilizar apenas um nodo para realizar o escalonamento dos processos evitando o gargalo existente na abordagem anterior. Cada nodo possui um controlador local que é executado de forma assíncrona e concorrente com os demais participantes do sistema e cada um é responsável por realizar as decisões de escalonamento de um determinado conjunto de EP's. Por outro lado, essa abordagem, além de tornar cada EP mais complexo computacionalmente - cada nodo deve implementar a política de balanceamento de carga - ainda traz a desvantagem do aumento de comunicação, uma vez que as decisões per-

tinentes às migrações serão tomadas baseadas em informações de todos os nodos, tornando necessário diversas trocas de mensagens entre eles.

Assim, o modelo proposto neste trabalho é uma proposta *global* que, apesar de se utilizar de informações locais dos EP's, preocupa-se em manter o equilíbrio total do sistema de forma *dinâmica* sendo uma política *centralizada* que concentra em uma única estrutura - o gerenciador global - a coordenação de todo o esquema de balanceamento.

2.3 Desenvolvimento de algoritmos de balanceamento de carga

Ao realizar-se o desenvolvimento de um algoritmo de balanceamento de carga, Sinha [13] destaca os seguintes itens a ser observados:

- *política de estimativa de carga* que determina como estimar a carga de um nodo do sistema em particular;
- *política de transferência de processos* que determina se um processo deve ser executado localmente ou remotamente;
- *política de localização* que determina o nodo destino de um processo selecionado para transferência;
- *política para troca de informação do estado do sistema* que aponta como devem ser trocadas informações acerca da carga do sistema entre os nodos;

Esses itens são detalhados a seguir.

Política de estimativa de carga. Nesse caso, conforme Tanembaum [18], a melhor maneira para se verificar a carga de um nodo do sistema é através da taxa de utilização do EP. Essa taxa é definida como o número de ciclos executados de fato por cada processo computacional ou ainda pode ser medida através da utilização de um *timer* que periodicamente verifica o estado do EP. O modelo proposto também utiliza a taxa de utilização do EP como sendo a medida de estimativa de carga.

Política de transferência de processos. Essa política diz respeito à decisão acerca do volume de carga de um nodo, isto é, se está com sobrecarga ou se está sendo subutilizado. Usualmente, é empregado um único valor de disparo (do inglês, *threshold*) que demarca as áreas sobrecarregadas ou subutilizadas de um nodo. Esse valor pode ser pré-determinado (estático) ou definido em tempo de execução (dinâmico) através da carga média de todo o sistema. Outra decisão importante nesse passo do desenvolvimento do algoritmo de balanceamento está relacionada à

quantidade de valores de disparo. Ao se utilizar um único valor, dividem-se as possíveis cargas do nodo em sobrecarregada ou subutilizada. Por outro lado, pode-se optar por dois valores de disparo, onde existe um limite alto para o início da área de sobrecarga e um limite baixo para o início da área de subutilização, mantendo-se uma área onde a carga do sistema é considerada normal. Ambas as abordagens podem ser visualizadas na figura 2.

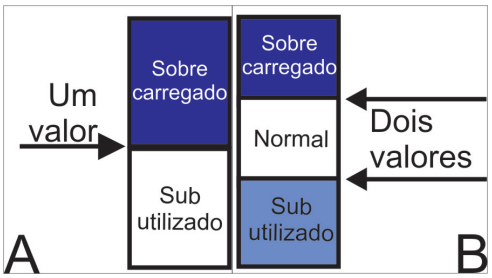

Figura 2 – Carga nas regiões com: (a) único valor de disparo e (b) dois valores de disparo
Fonte: Adaptado de [13]

No modelo proposto adota-se a abordagem com dois valores de disparo sendo que sua definição ocorre de forma dinâmica em função do comportamento do sistema. A maneira pela qual esses valores são calculados é definida no modelo e será descrita no Capítulo 5.

Política de localização. Essa política visa escolher o nodo destino para um processo que será transportado de um nodo para outro. Existem quatro políticas principais acerca da localização do destino:

1. *threshold*, onde se escolhe um destino aleatoriamente e executa-se um teste para verificar se a transferência do processo em questão colocará esse nodo sorteado em uma situação proibitiva ou não. Caso não seja possível a transferência para esse nodo, escolhe-se outro nodo, também de forma aleatória, e repete-se o procedimento;

2. *shortest*, onde *n* nodos são escolhidos de forma randômica e são ordenados de acordo com sua carga. O processo é, então, transferido para aquele com menor carga, a não ser que essa transferência coloque o nodo em questão em uma situação proibitiva, escolhendo-se então, o próximo nodo com a menor carga e assim, sucessivamente. O modelo proposto baseia-se nessa política de localização para encontrar o destino de uma migração e, além de considerar o aspecto carga, considera, também, os aspectos temporais e de consumo de energia do nodo que irá receber a migração;

3. *bidding*, que utiliza conceitos análogos aos da economia como compradores e vendedores. Cada nodo assume um papel: o gerente é aquele que tem processos que necessitam de um

local para executar enquanto que os contratantes são os nodos que podem assumir essa carga. A partir daí mensagens são enviadas para que o sistema entre em um consenso;

4. *pairing*, que visa criar diversos pares no sistema entre os quais a carga é trocada. Quando um nodo estiver sobrecarregado, seleciona outro para que seja seu par, de forma aleatória. Esse nodo sorteado pode ou não aceitar o convite. Ao aceitar o pedido, a migração de processos começa entre o par, a partir do nodo mais carregado para aquele com menor carga.

Política para troca de informação sobre o estado do sistema. Uma vez que o algoritmo de balanceamento de carga deve ter conhecimento acerca da carga de cada um dos nodos do sistema é necessário que seja adotada uma política que defina o modo pelo qual essa transferência de informações será realizada, entre as quais se pode citar:

1. *broadcast*[3] *periódico*, onde cada nodo envia seu estado para todos os participantes do sistema em intervalos pré-definidos. Esse método não é considerado vantajoso uma vez que gera muito tráfego na rede de comunicação além de mensagens inúteis daqueles nodos cujo estado não se modificou desde a última mensagem enviada. Por fim, a escalabilidade desse método também se mostra bastante pobre uma vez que um grande número de nodos no sistema acarreta em uma quantidade significativa de mensagens para a rede de comunicação;

2. *broadcast quando ocorre mudança de estado*, que visa evitar o problema da troca de mensagens inúteis existente no método de *broadcast* periódico uma vez que prevê o envio de mensagens somente quando ocorre uma modificação no estado do nodo transmissor. Nesse caso, entende-se por troca de estado toda vez que um processo chega a um nodo (seja do mundo externo ou de um outro nodo) ou deixa o nodo (seja porque foi migrado ou teve sua execução terminada). Um refinamento desse método é enviar mensagem não em qualquer mudança de estado, mas sim, quando o nodo troca seu estado de carga (sobrecarregado ou subutilizado). Essa alteração funciona melhor quando empregada em um sistema que utiliza dois valores de disparo na sua política de transferência de processos [13]. O modelo proposto baseia-se, em partes, nessa política de troca de informação sobre o estado do sistema. Inicialmente, todos os nodos enviam ao gerenciador global suas informações e, após essa etapa, sempre que cada monitor local verificar que sua própria carga mudou comparada à ultima informação enviada, faz novo envio ao gerenciador global;

[3] *broadcast* é um tipo de mensagem enviada de um nodo para todos os outros participantes do sistema.

3. *troca de informações sob demanda*, que se baseia na observação de que os nodos precisam saber sobre o estado dos outros nodos somente quando estiverem sobrecarregados ou subutilizados. No método, quando necessário (sobrecarregado ou subutilizado) um nodo pede para que todos os outros enviem para ele suas respectivas situações de carga. Ao receber uma mensagem desse tipo todos os nodos enviam informações a respeito de sua carga. Em uma possível melhoria somente os nodos que podem, de fato, cooperar com aquele que fez a requisição enviam informações a respeito de sua carga, ou seja, se o nodo que está fazendo a requisição está subutilizado, somente os sobrecarregados respondem-no;

4. *troca de informações por polling*[4], onde não existem mensagens de *broadcast*, uma vez que se tenta, através desse método, alcançar uma maior escalabilidade do que a existente nos métodos baseados em *broadcast*. Assim, quando um nodo precisa de cooperação seleciona, de forma aleatória, outros nodos e os pergunta, individualmente, acerca de suas situações de carga.

2.4 Gerenciamento dos processos

Em um sistema centralizado deve-se realizar o gerenciamento dos processos que precisam utilizar o EP através de mecanismos e políticas específicas. Similarmente, sistemas multiprocessados também necessitam de tal gerenciamento visando a divisão dos processos requisitantes entre os nodos existentes. Para satisfazer esse gerenciamento, são utilizados, principalmente, os seguintes conceitos:

- *alocação de processos*, que estuda as formas pelas quais um determinado processo deve ser designado a um EP;

- *migração de processos*, que estuda como um processo pode ser movido entre os diversos elementos de processamento do sistema;

- *uso de threads* para a obtenção de um paralelismo de grão fino com o objetivo de melhorar a utilização dos recursos computacionais existentes.

Dentre esses três conceitos, o foco deste trabalho encontra-se explicitamente no de migração de processos. Dessa maneira somente os itens que dizem respeito à migração de processos serão discutidos a seguir.

[4] *polling* é uma atividade síncrona de amostragem do estado de um determinado componente.

2.4.1 Migração de processos

A migração de processos é a realocação, de um determinado processo, de um local atual (nodo fonte) para um outro nodo (nodo destino). O fluxo de execução de uma migração está representado pela figura 3, onde se pode observar as operações realizadas tanto no nodo origem quanto no nodo destino ao longo do tempo.

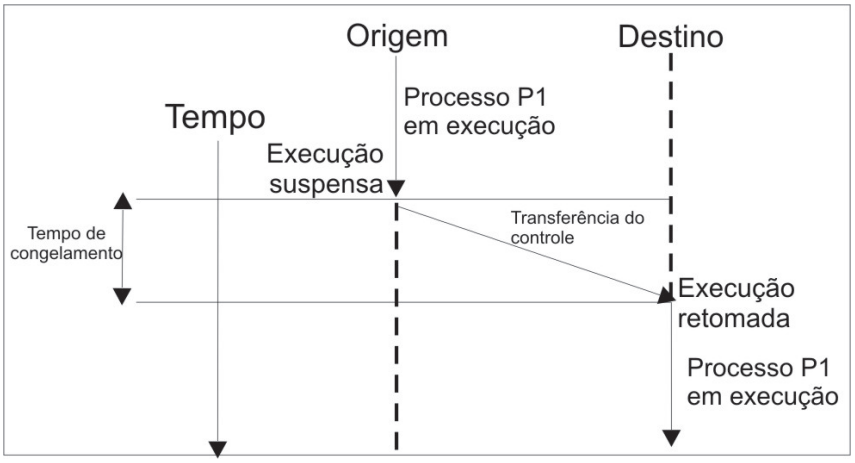

Figura 3 – Fluxo de execução de uma migração
Fonte:Adaptado de [13]

Em geral, a principal motivação para o emprego da migração de processos em sistemas distribuídos (de propósito geral ou embarcado) está relacionada à dinamicidade existente nas suas aplicações. Isso significa que, mesmo após uma alocação balanceada de sua carga inicial [5], não existe garantia de que, ao longo do tempo de vida do sistema, sua carga permanecerá com o balanceamento inicialmente provido pela alocação dos processos. Assim, a migração de processos é necessária quando, por diversos fatores, tais como o término prematuro de um processo ou a criação de um novo, a carga do sistema deixa de ter um balanceamento equilibrado ao longo de sua execução, necessitando de uma solução.

Adicionalmente, para que a migração de processos possa ocorrer, diversas atividades devem ser definidas e implementadas [13]. As atividades principais apontadas por Sinha [13] estão descritas a seguir.

[5] No modelo proposto utilizou-se uma alocação inicial estática, baseada na experiência do usuário.

Transferência do contexto dos processos

O processo de migração envolve a transferência de, basicamente, dois tipos de informações, do nodo origem para o nodo destino:

- *estado do EP*, que consiste no seu estado de execução (conteúdo dos registradores), informação de escalonamento, informação sobre a utilização da memória principal associada a esse processo, estados de operações de E/S entre outros;
- *espaço de endereçamento do EP*, que consiste do código, dados e pilha do programa.

Como, por vezes, a transferência do estado do EP não é viável, designa-se na literatura por transferência de contexto a troca de informações acerca do código e dados do processo. Nesse âmbito, existem três técnicas principais para migração e que são detalhadas a seguir.

1. *Cópia*. Técnica que congela o processo (do inglês, *total freezing*) durante a cópia do seu contexto para o nodo destino. Apesar de apresentar uma implementação bastante simplificada, aumenta o tempo de migração, o que não é desejado em sistemas que possuam restrições de tempo real, por exemplo. Essa técnica pode ser visualizada na figura 4.

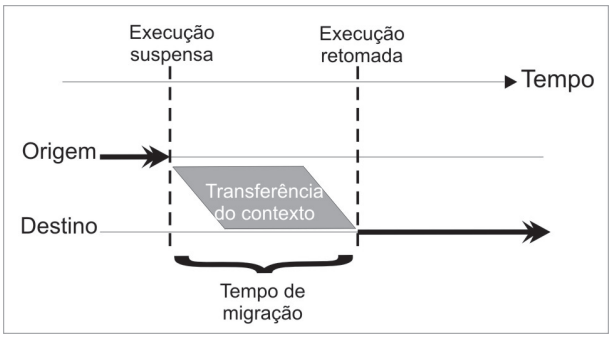

Figura 4 – Transferência do contexto de um processo por cópia
Fonte:Adaptado de [13]

2. *Pré-cópia*. Técnica que inicia a cópia do contexto ainda durante a execução do processo e somente quando essa cópia estiver pronta é que o processo é congelado, e apenas são atualizados eventuais itens que tenham tido seus valores alterados durante esse período. Essa técnica, visualizada na figura 5, torna-se uma opção atrativa para sistemas de tempo real, uma vez que diminui drasticamente o tempo de migração, ainda que sob a pena de uma implementação mais dificultosa.

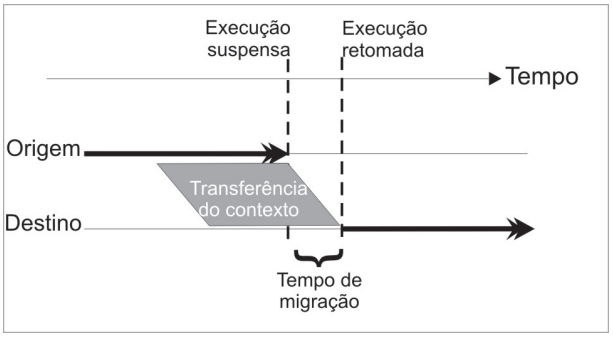

Figura 5 – Transferência do contexto de um processo por pré-cópia
Fonte:Adaptado de [13]

3. *Cópia sob demanda*. Técnica que, no momento da migração, apenas passa o processo para o nodo destino deixando todos os seus dados no nodo origem. Adicionalmente, sempre que o processo precisar de algum dado deve solicitá-lo ao antigo nodo de origem. Esse método possui similaridade com o sistema de memórias *cache* e baseia-se no princípio da localidade espacial para justificar seu funcionamento. Apesar de apresentar o menor *overhead* de migração, aumenta excessivamente o uso do meio de comunicação, além de apresentar problemas de confiabilidade devido a questões de consistência, não sendo considerado um método viável. Ainda assim, está representado pela figura 6.

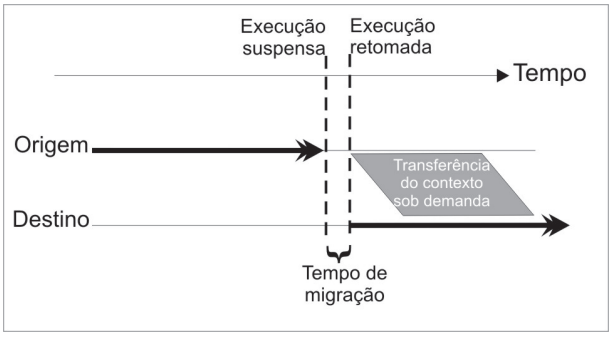

Figura 6 – Transferência do contexto de um processo por cópia sob demanda
Fonte:Adaptado de [13]

O modelo proposto prevê que a cópia total do contexto e dos dados do processo migratório sejam enviados ao destino no momento da migração. Apesar de ser uma opção que apresenta um *overhead* de migração alto, acredita-se que, dependendo do meio de interconexão utilizado e do tamanho das tarefas empregadas, esse *overhead* possa ser não proibitivo, como apresentado em [19]. Embora essa tenha sido a opção escolhida para o modelo proposto, em função de restrições encontradas durante a implementação da plataforma de validação, utilizou-se uma

proposta um pouco diferente: o código das tarefas encontra-se em todos os EP's e, durante uma migração, somente os dados são enviados no que é conhecido como código replicado, uma abordagem que, apesar do *overhead* de memória, também foi empregada no estudo de Mulas [20]. Para tomar a decisão acerca da política de cópia do contexto, foi realizado um estudo referente às possibilidades existentes. Entre elas, pode-se destacar a migração parcial (migra-se apenas os dados, com código replicado) ou a migração total (migra-se os dados e o código, com código relocável). Para que a migração total possa ser implementada, é necessário que o *kernel* utilizado suporte código relocável, sendo que o *kernel* disponibilizado na plataforma de validação não possui tal recurso. Assim, estudou-se a possibilidade de utilizar a migração parcial, onde o código encontra-se em uma memória única, centralizada ou replicado em cada EP, com memória distribuída. A abordagem centralizada tende a representar um gargalo muito grande além de exigir que o meio de conexão suporte tal tráfego. Por isso, apesar de não ser a opção mais otimizada - em função do desperdício de memória - optou-se pela migração parcial, com código replicado e memória distribuída.

Redirecionamento de mensagens dos processos

Outra preocupação existente acerca da migração de processos em sistemas distribuídos diz respeito à necessidade de garantia de que todas as mensagens pendentes sejam entregues corretamente ao processo destino, independente de sua localização no sistema.

Para isso, Sinha [13] categoriza as mensagens em três tipos distintos:

1. *tipo 1*: mensagens que tenham sido recebidas no nodo origem após a execução do processo ter sido parada nesse nodo mas ainda não tiver sido iniciada no nodo destino;

2. *tipo 2*: mensagens que tenham sido recebidas no nodo origem cuja execução já tenha sido iniciada no nodo destino;

3. *tipo 3*: mensagens que tenham sido enviadas de qualquer outro nodo após a execução do processo migratório já ter sido iniciada no nodo destino.

Para cada um desses tipos existem diferentes técnicas que propõem soluções diversas para o problema e que são detalhadas a seguir.

1. *Reenvio de mensagem*. Mecanismo bastante simplificado que requer que o nodo transmissor tenha maneiras de reenviar a mensagem e de detectar a falha durante o envio. Aplicado em sistemas como o V-System [21] e Amoeba [22], trata as mensagens dos três tipos com pequenas diferenças de implementação em relação a maneira pela qual a falha é detectada e o novo destino descoberto.

2. *Mecanismo de nodo origem*. Nesse mecanismo assume-se que o descritor do processo mantém a informação acerca do seu nodo origem e, também, que o nodo origem de um processo sempre saiba sua atual localização. Adicionalmente, as mensagens são sempre enviadas para os nodos de origem dos processos e cabe a esses nodos fazer o redirecionamento. Apesar de ter sido utilizado nos sistemas AIX's TCF [23] e Sprite [24], esse método apresenta duas grandes desvantagens. A principal está ligada à questão da confiabilidade, uma vez que, se o nodo origem de um processo posteriormente migrado falhar, o processo migrado não receberá mais as mensagens a ele destinadas. Outro grande problema é que o processo migrado continua a ser uma carga para o seu nodo de origem.

3. *Ligações transversais*. Esse mecanismo prevê soluções distintas dependendo do tipo de mensagem. Para as mensagens do tipo 1, prevê a manutenção de uma fila que guarda as mensagens enviadas até que o processo seja iniciado no nodo destino. Já as mensagens dos tipos 2 e 3 devem utilizar uma ligação (do inglês, *link*) criada no nodo origem e que informa o novo destino do processo. É importante destacar que essa abordagem diferencia-se da de nodo origem uma vez que, no caso de uma nova migração, a ligação deverá ser criada na nova localização do processo. Como muitas ligações transversais podem ser necessárias para que se encontre um processo, qualquer falha em uma dessas ligações torna inviável sua utilização. Além dessa desvantagem, da confiabilidade, o desempenho também é apontado por Sinha [13] como sendo um problema. Esse mecanismo foi empregado no sistema DEMOS/MP [25].

4. *Atualização de caminho*. Utilizado pelo sistema Charlotte [26], esse mecanismo emprega o conceito de canais de comunicação virtuais ou independentes de localização. Assim, durante a migração de um determinado processo, o nodo origem envia mensagens de atualização de caminho (do inglês, *link update*) para todos os controladores dos processos que se comunicam com o processo migratório. Em função desse mecanismo, as mensagens do tipo 3 são enviadas diretamente ao nodo destino enquanto que mensagens dos tipos 1 e 2 ficam armazenadas no nodo origem até que o processo seja iniciado no nodo destino quando são, então, encaminhadas.

No caso do modelo proposto, o redirecionamento é feito da seguinte maneira: quando um nodo recebe uma ordem de migração, faz um *broadcast* informando a todos os elementos do sistema o novo destino de uma determinada tarefa. Para que isso seja possível, cada nodo mantém uma tabela de roteamento com informações acerca das tarefas com as quais se comunica e seus destinos. Ainda, cada tarefa possui uma identificação única em todo o sistema, o que facilita a implementação de tal mecanismo.

Finalmente, com base nos conceitos aqui apresentados juntamente com a definição dos componentes do sistema utilizado para validar o trabalho, apresentada no Capítulo 4, será possível um melhor entendimento do modelo proposto no Capítulo 5, principalmente após a verificação da análise do estado da arte, apresentada no próximo capítulo.

3 Análise do estado da arte

Neste capítulo está exposta uma revisão literária que detalha pesquisas relacionadas com a área de balanceamento de carga e de migração de tarefas na computação embarcada. Apesar de se tratarem de trabalhos específicos para a computação embarcada é notória a interferência que os sistemas multiprocessados de propósito geral têm sobre as técnicas apresentadas. Ao longo da apresentação dos trabalhos relacionados, é feita uma análise crítica onde se apresenta a posição da pesquisa proposta com relação aos trabalhos correlatos.

3.1 Nollet

Em Nollet [27] é apresentado um estudo que trata do gerenciamento centralizado de recursos e em tempo de execução através do uso de uma heurística para o gerenciamento de uma NoC e de migração de tarefas. O trabalho baseia-se em uma plataforma que conta com um processador *StrongARM* de um PDA conectado a um FPGA que contém os EPs escravos, interconectados por uma NoC 3x3. Nesse sistema, existe um SO centralizado que faz todo o gerenciamento da alocação dos recursos computacionais através de descritores de EPs. Além disso, os nós que estão no FPGA são reconfiguráveis em tempo de execução e o mecanismo que permite a execução das tarefas tanto em *hardware* quanto em *software* não faz parte do escopo do presente trabalho e está detalhado em Mignolet [28]. Nesse trabalho, a migração de tarefas é utilizada em duas situações principais: caso haja uma modificação dos requisitos de usuário (como a troca de resolução durante uma aplicação de vídeo, por exemplo) ou ocorra uma falha no mapeamento de recursos. São apresentados dois mecanismos de migração de tarefas, denominados de geral e de *pipeline*.

No primeiro método, existem pontos específicos no código responsáveis por indicar que o nodo origem pode migrar e que o nodo destino pode receber a migração. Para garantir a consistência de mensagens, o nodo origem deve, ao atingir um ponto de migração, sinalizar essa situação para seu SO que, então, se encarrega de pedir às tarefas produtoras uma suspensão temporária do envio de mensagens. Após essa suspensão, a tarefa é migrada e iniciada no nodo destino. Em relação à consistência das mensagens, o SO encarrega-se de inicializar - no

nodo origem - uma tabela que contém o novo destino das mensagens além de encaminhar as mensagens a esse novo destino. As tabelas de todos os produtores vão sendo atualizadas na medida em que o destino vai recebendo mensagens desses produtores.

Já o segundo método de migração baseia-se em algoritmos que usem a abordagem *pipeline* como, por exemplo, descompressão de vídeo, e que contenham pontos sem estado (do inglês, *stateless points*), ou seja, que em certos momentos recebam no seu *pipeline* informações novas e independentes. Isso faz com que o mecanismo de migração possa mover múltiplas tarefas *pipelined* de uma única vez sem preocupar-se sobre a transferência do estado da tarefa.

Por fim, são apresentadas questões relacionadas ao desempenho do mecanismo de migração através da medição de, por exemplo, seu tempo de reação. Esse tempo é aquele levado entre a requisição da migração e sua efetiva realização, que ocorre quando o ponto pré-definido de migração é atingido e varia de acordo com os mecanismos empregados.

Como continuação desse trabalho, Nollet [29] apresenta um estudo visando, justamente, diminuir o tempo de reação encontrado nos mecanismos de migração previamente propostos. Nessa pesquisa, os autores propõem uma técnica de reuso de registradores de depuração do próprio processador para diminuir o *overhead* inicial de uma migração de tarefas em um MPSoC heterogêneo. Cabe ao SO verificar (de maneira não explorada pelos autores) a carga do sistema e notificar as tarefas que devem ser migradas. O foco do trabalho está em uma técnica que visa minimizar o tempo entre a tomada de decisão do SO em relação à migração e o tempo em que, efetivamente, a migração começa a ser realizada.

Na implementação, os autores utilizaram o processador *PowerPC 405* [30], presente nos FPGA's *Virtex-II Pro* da *Xilinx*. Nesse sistema, como pode ser observado na figura 7, após o início de uma tarefa (1) um tratador de migrações é registrado junto ao SO (2). Esse tratador será o responsável por coletar o estado lógico da tarefa, após essa atingir um ponto de migração. Além disso, todos os endereços de pontos de migração são registrados no SO (3). Nesse ponto, a tarefa inicia sua execução. Na ausência de uma requisição de migração, não há nenhum *overhead* em tempo de execução (4). O SO mantém os endereços de pontos de migração em estruturas criadas especificamente para esse fim. A cada troca de contexto o SO atualiza um dos registradores de depuração (IAC, do inglês *Instruction Address Compare*), e, quando o gerenciador de um determinado recurso decide migrar, ativa esses registradores (5) (6). Assim que a tarefa atinge uma instrução definida em um ponto de migração, uma interrupção de *hardware* é gerada (7) para ativar o tratador de sinal de migração (8).

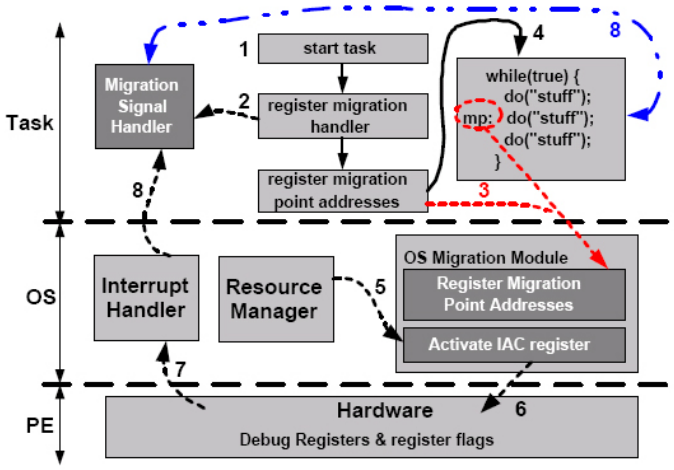

Figura 7 – Migração de tarefas através de registradores de depuração
Fonte: Adaptado de [29]

Com relação aos trabalhos de Nollet, ambos concentram-se em técnicas específicas para migração. Além de serem dependentes de uma determinada arquitetura, os autores não exploram em nenhum momento as razões pelas quais deve-se realizar uma migração. Apesar disso, apontam itens que devem ser levados em consideração como o tempo de reação de uma migração, que não deve ser alto a ponto de tornar proibitivo o uso de tal mecanismo. Além disso, apresenta uma preocupação com o tratamento das mensagens enviadas às tarefas migratórias, o que também foi empregado no modelo proposto. Por fim, apresenta pontos de migração no código que apesar de potencialmente apresentar melhor desempenho, faz com que o programador tenha que explicitamente - e de forma estática - definir onde e quando as migrações devem ocorrer. No modelo proposto, tem-se uma idéia diferente, uma vez que se tenta atacar o problema do balanceamento de uma forma dinâmica e, assim, dinamicamente, ordenar migrações aos diversos EP's envolvidos.

3.2 Bertozzi

O trabalho de Bertozzi et al. [7] trata explicitamente da migração de tarefas em MPSoC's e propõe um esquema gerenciado pelo usuário baseado em verificação de pontos no código, além de suporte de *middleware* em nível de usuário. A implementação realizada pelos autores foi organizada seguindo um modelo mestre/escravo, onde o processo mestre realiza controles de admissão e uma alocação inicial dos processos, almejando uma justa distribuição de carga.

Também foi utilizado um esquema de escalonamento centralizado, mais viável segundo os autores. O sistema implementado utiliza o SO μClinux [31] como base para o desenvolvimento de um *middleware* que tem suporte à passagem de mensagem, que provê o término e a invocação remota de tarefas. O processo mestre é quem coordena as migrações que serão realizadas e, como foi implementado de forma centralizada, mantém uma tabela com todos os dados pertinentes à migração, bem como, uma área que contém as informações sobre trocas de contexto. Em relação ao código, a proposta requer que sejam definidos pontos possíveis de migração. É necessariamente nesses pontos que a migração pode ser realizada sendo que ao chegar a um ponto desses é que a tarefa sinaliza às outras que está apta a migrar. Cabe então ao SO decidir se e onde aquela tarefa deve executar. Os resultados apresentados pelos autores são bastante focados em medições de um *overhead* gerado tanto pela migração em si, quanto pelo *deamon* de migração que fica executando no sistema à procura de processadores ociosos.

O estudo realizado por Bertozzi, assim como o de Nollet, possui pontos de migração no código, opondo-se a idéia do modelo aqui proposto. Apesar disso, Bertozzi propõe uma solução que, assim como o presente modelo, é centralizada na coordenação das migrações por entender que se trata de uma abordagem mais viável para MPSoC's. Por fim, os resultados de Bertozzi são focados em medições do *overhead* gerado pela migração sendo que esse cálculo é imprescindível a qualquer trabalho que envolva migração de tarefas, uma vez que, apesar dos potenciais benefícios de sua utilização, um *overhead* muito alto pode tornar sua utilização proibitiva.

3.3 Ozturk

Em Ozturk [32] uma abordagem de migração, denominada de migração seletiva, é detalhada. Essa abordagem é composta por três componentes principais: personalização (do inglês, *profiling*), anotação de código (do inglês, *code annotation*) e migração seletiva. As duas primeiras partes são realizadas em tempo de compilação sendo que a personalização realiza a coleta do custo energético da migração de certos fragmentos tanto de código quanto de dados através da rede de comunicação. Esses custos são anotados no código e auxiliam a migração seletiva que acontece em tempo de execução.

Apesar disso, a decisão de quando uma migração deve ocorrer não é explorada pelos autores enquanto que a validação do sistema utilizou um barramento como meio de comunicação (apesar desse meio não ser restritivo). Os resultados foram analisados com base em um conjunto de *benchmarks* desenvolvido pelos autores. Essas aplicações são utilizadas como entrada na pla-

taforma de simulação cuja saída exibe uma estimativa do consumo de energia e da quantidade de ciclos necessários para a execução. A arquitetura utilizada no trabalho é composta por 8 processadores com 32 kB de memória local, interligados por um barramento responsável pela troca de mensagens que é empregada como meio de comunicação entre os processadores.

A união entre abordagens de tempo de compilação e tempo de execução é promissora principalmente quando se trata de consumo de energia. O modelo proposto prevê no seu modelo de tarefas informações acerca do consumo de energia ao longo do tempo de vida dessa tarefa. Esses valores de consumo de energia podem ser estimados e ter seus valores anotados para ser utilizados em tempo de execução pelo modelo de balanceamento.

3.4 Barcelos e Brião

O trabalho de Barcelos [33] propõe uma abordagem de organização híbrida de memória (centralizada e distribuída) focando em uma migração de tarefas que consuma menos energia. Para isso, dependendo dos nodos origem e destino, o dado a ser migrado é enviado diretamente da origem ao destino ou o destino recebe o dado a partir de uma memória centralizada. O método baseia-se em informações relativas à energia gasta para a transferência dos dados (seja do nodo origem, seja da memória centralizada) e utiliza aquela que consome menos energia. A arquitetura empregada é formada por processadores interconectados por uma NoC e seu simulador foi desenvolvido em SystemC. Os resultados mostram que a abordagem de memória híbrida apresenta uma redução no consumo de energia de 24% e 10%, quando comparada às abordagens puramente globais e distribuídas, respectivamente.

No mesmo grupo de pesquisa, Brião [19] propõe um trabalho que leva em consideração o *overhead* da migração de tarefas em um ambiente dinâmico e mostra seu impacto em termos de energia, desempenho e restrições de tempo real no contexto de MPSoC's baseados em NoC's. Nesse trabalho, uma ferramenta foi desenvolvida em *SystemC* capaz de simular o comportamento de sistemas baseados em NoC's que executam tarefas geradas pelo TGFF [34] (do inglês, *Task Graph For Free*) as quais são dinamicamente carregadas. A migração de tarefas é executada baseada em um modelo de cópia. Esse método é bastante simples e possui um alto *overhead* associado, uma vez que todo o contexto (código, dados, pilha e conteúdos dos registradores internos) é migrado. Apesar disso, segundo os autores, a migração de tarefas pode ser utilizada em sistemas embarcados baseados em NoC's visto que apresenta ganhos em termos de desempenho e redução no consumo de energia envolvidos no sistema. Esses ganhos são suficientes para garantir o cumprimento de *deadlines* de sistemas *soft real-time*.

O estudo de Barcelos é bastante interessante pois propõe uma organização híbrida na memória sendo que o modelo proposto não levou em consideração diferentes possibilidades para essa situação. Apesar disso o foco no trabalho de Barcelos é um pouco diferente no que tange questões arquiteturais enquanto que o presente modelo abstrai essas questões focando apenas no balanceamento do sistema. Já com relação à pesquisa de Brião o presente modelo também adota o modelo de cópia do contexto, embora por restrições de implementação, a replicação de código tenha sido empregada. Além disso, Brião também propõe levar em consideração o *overhead* da migração com relação a diversos aspectos muito embora não se concentre nas razões que levam uma migração a ser necessária.

3.5 Carta, Pittau e Mulas

A pesquisa de Carta [35] apresenta o algoritmo *MiGra* que serve para reduzir gradientes de temperatura em MPSoCs através de migração de tarefas. Esse trabalho baseia-se em valores de temperatura do *chip*, que são medidos em tempo de execução, para balancear sua temperatura sem causar impactos negativos no consumo de energia. O algoritmo MiGra, diferentemente de várias técnicas que reagem quando percebem níveis alarmantes de temperatura, tenta fazer com que as temperaturas dos processadores não ultrapassem uma certa distância da média de temperatura do sistema como um todo. Isso faz com que as tarefas sejam migradas mesmo de processadores mais frios, ao contrário de outras técnicas que sempre migram tarefas somente de núcleos mais quentes. Além disso, o algoritmo leva em consideração o consumo de energia total do *chip* após a migração a ser realizada (são feitas estimativas).

Os autores apresentam diferentes abordagens para a escolha dos conjuntos de tarefas a serem migrados. Com base nessas três versões do algoritmo são analisados os resultados da comparação com algoritmos que realizam balanceamento de carga simples e balanceamento de carga que levam em consideração o consumo de energia.

No estudo de Pittau [36] o uso da migração de tarefas e seu impacto em aplicações multimídia para multiprocessadores embarcados. Nesse trabalho, são empregados processadores que permitem o uso de diferentes freqüências de operação, além de se provar a viabilidade da migração de tarefas entre esses processadores, interconectados por um barramento. No modelo de migração empregado cada processador possui uma memória local que contém os dados das tarefas. Na ocorrência de uma migração, esses dados são transferidos dessa memória para uma memória compartilhada entre todos os processadores, evitando o *overhead* de um sistema composto por troca de mensagens.

Já o trabalho de Mulas [20] utiliza a migração através de um *middleware* e o objetivo principal é obter uma política de balanceamento térmico para aplicações de *streaming* em arquiteturas multiprocessadas. Devido a restrições de arquitetura, a migração é feita através do que é denominado pelos autores de *replicação de tarefas*, que consiste em se manter uma réplica de cada tarefa em todos os processadores o que evita o *overhead* natural de uma migração embora implique em maior área necessária em memória.

O trabalho de Carta apresenta um item inovador com relação aos trabalhos já expostos: leva em consideração valores reais da temperatura do *chip* como sendo de disparo para a realização de migrações. Além disso, apresenta a idéia de que é preciso - pelo menos em termos de temperatura - manter-se um nível médio em todo o sistema, não tratando somente os casos de excesso de temperatura. Essa abordagem de manter o sistema equilibrado num todo também foi empregada no modelo proposto embora não se leve em consideração as questões de temperatura do *chip*, pois isso requer uma plataforma específica que ofereça esse tipo de informação. No trabalho de Pittau é utilizada a técnica de migração de tarefas para aplicações específicas de multimídia e, assim como o presente trabalho, também utiliza um barramento para fazer a interconexão dos EP's. Por fim, o trabalho de Mulas utiliza a questão térmica como justificativa e, também devido a restrições de arquitetura, utiliza uma réplica do código da tarefa em todos os processadores assim como o presente trabalho.

3.6 Coskun

O trabalho de Coskun [11], propõe um sistema cujo escalonamento de tarefas leva em consideração a temperatura real do MPSoC. Nessa pesquisa, os autores mostram o desenvolvimento e a avaliação de políticas dinâmicas de escalonamento em nível de SO, que possuem um *overhead* de desempenho desprezível. Os autores trabalham em nível de sistema operacional e são citados diversos outros escalonadores que possuem suporte ao balanceamento de carga. Esses outros escalonadores, porém, não tratam de itens importantes, no projeto de sistemas embarcados, tais como a temperatura e o consumo de energia. Pontos superaquecidos em um *chip* podem acelerar diversos mecanismos de erro, como a eletromigração, o que não é desejado. Esse trabalho propõe um escalonamento que leve em consideração um mapa térmico de todo o MPSoC na hora de decidir a ordem pela qual as tarefas serão executadas. A implementação foi realizada em um *UltraSPARC T1*, da *Sun Microsystems*, que contém 8 núcleos e memória, unidades de E/S e comunicação. Como essa é uma técnica que necessita da medição da temperatura de um *chip*, alguns itens mais específicos devem ser conhecidos, tais como a tecnologia de fabricação.

Portanto, é válido destacar que esse MPSoC foi produzido em uma tecnologia de 90nm. A figura 8 ilustra o MPSoC utilizado como exemplo.

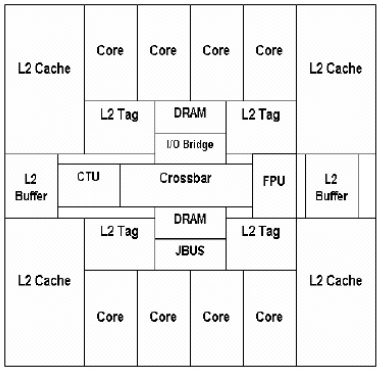

Figura 8 – Planta-baixa do MPSoC
Fonte: Adaptado de [11]

Finalmente, em relação ao presente trabalho é importante observar que o estudo desenvolvido por [11] apresenta uma aplicação para a migração de tarefas e para quaisquer outras técnicas que visem manter o equilíbrio da carga dos nodos do sistema de forma dinâmica. A migração em si, realizada no trabalho, não está detalhada e, portanto, não pode ser explorada. Apesar disso, a idéia de empregar técnicas de balanceamento de carga para evitar a ocorrência de defeitos provenientes de superaquecimento em *chips* é bastante promissora e tende a ser, cada vez mais, objeto de pesquisa tanto no meio acadêmico quanto na indústria.

3.7 Götz

O trabalho de Götz [37] destaca um fluxo para realizar a realocação dinâmica entre tarefas híbridas, que podem executar tanto em *hardware* quanto em *software*. Essas tarefas são representadas por um grafo de transição de estados onde cada estado é chamado de bloco de computação, o qual representa uma determinada operação de uma tarefa. No grafo existem *pontos de encontro* (*matching points*) entre as versões compiladas e sintetizadas (CPU e FPGA, respectivamente) da tarefa. Da mesma forma, são representados *pontos de troca* (*switching points*) nos quais se pode realizar a realocação de tarefas. Nesses pontos existe somente um contexto que precisa ser salvo. Um exemplo desse grafo pode ser observado na figura 9. Através dessa representação a ferramenta é capaz de gerar as versões de *hardware* e *software* da tarefa juntamente com um componente de gerenciamento da migração, relacionado com o sistema operacional.

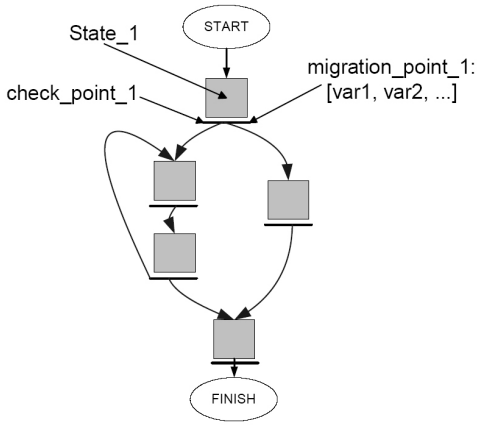

Figura 9 – Grafo de transição de estado de uma tarefa híbrida realocável
Fonte: Adaptado de [37]

Apesar de mostrar as funcionalidades da ferramenta, os detalhes de como o contexto é salvo ou dos critérios que levam à migração não são explorados no artigo. Esse estudo mostra-se interessante para o presente trabalho uma vez que trabalha com realocação dinâmica de tarefas e utiliza, apesar de não detalhado no artigo, um mecanismo exclusivo para a tomada de decisão em relação à migração ou não de uma tarefa.

3.8 Zheng

O trabalho de Zheng [38] mostra um algoritmo de escalonamento com requisito de eficiência energética baseada em migração de tarefas voltada para tarefas periódicas de tempo real. A abordagem contém uma parte *off-line* que é responsável por designar as tarefas nos processadores. As tarefas podem ser colocadas em um ou em dois processadores. As designadas para um processador são denominadas de *tarefas fixas*. Aquelas que residem em dois processadores são as *tarefas de migração*. É nessa fase que as tarefas de migração têm seus destinos (no caso de uma migração) pré-estabelecidos. Não são explorados no artigo os motivos que levam o sistema a decidir se uma tarefa é ou não de migração. Apesar disso, os resultados são encorajadores e apontam que o escalonamento com migração de tarefas restrita tem o melhor desempenho energético quando comparado a sistemas que não possuem migração ou possuem migração arbitrária, contribuindo para o modelo proposto no sentido de que restringe os momentos em que as migrações ocorrem.

4 Modelo dos componentes do sistema

O modelo proposto de balanceamento de carga através de migração de tarefas assume que existam algumas estruturas com funcionalidades e características básicas para que sua execução seja possível. Dentre essas estruturas podem ser citadas: um meio de interconexão e elementos de processamento que contenham um *kernel* sobre o qual o modelo possa ser implementado, bem como um determinado modelo de tarefas. Assim, este capítulo aborda a descrição desses itens e, ao seu final, descreve a plataforma de validação utilizada e adaptada a este trabalho de acordo com os modelos descritos. Entre essas adaptações, a própria política de gerenciador global e monitores locais teve de ser implementada na forma de *drivers* do *kernel* empregado. Ainda, a adoção de memória distribuída com código replicado também foi uma decisão que ocorreu em função de limitações dessa plataforma de validação e das necessidades do modelo proposto, como apresentado na Seção 2.4.1.

4.1 Modelo dos componentes do sistema

Nesta seção estão descritos os detalhes sobre os modelos dos componentes do sistema necessários para que o modelo proposto possa ser executado.

4.1.1 Temporização do sistema

Para a temporização do sistema assume-se o conceito de *ticks*, onde cada *tick* equivale a uma unidade de tempo. É importante destacar que através da utilização do conceito de *ticks*, o sistema torna-se escalável, flexível e genérico uma vez que é possível adaptá-lo a diferentes processadores com diferentes freqüências de *clock*. A temporização do sistema será provida pelo EP utilizado no trabalho.

4.1.2 Modelo de Tarefa

Este trabalho considera uma tarefa (do inglês, *task*) como sendo uma unidade de computação indivisível sendo que um conjunto de tarefas T consiste em *n* tarefas periódicas independentes ($T = \{\tau_1, \tau_2 ... \tau_n\}$) e cada tarefa τ_i é definida pela tupla $(id_i, r_i, C_i, D_i, P_i)$, onde id_i é o identificador da tarefa, r_i é o tempo de liberação ou *release* da tarefa, C_i é o pior caso de tempo de execução (do inglês, *worst case execution time*, WCET) da tarefa, D_i é o *deadline* da tarefa e P_i seu período.

Cada vez que uma tarefa encontra-se pronta para execução, ela envia uma requisição para iniciar sua execução. Caso a tarefa seja periódica, os tempos de liberação sucessivos ocorrem no instante $r_k = r_0 + kP$, onde r_0 é o primeiro tempo de liberação da tarefa, k é o instante de liberação que se deseja saber e P o período. Assim, para saber o tempo de liberação r_3, deve somar ao tempo de liberação inicial r_0 o valor P do período multiplicado pelo valor de k, nesse caso, *3*. Os *deadlines* absolutos sucessivos são $d_k = r_k + D$. O modelo de tarefa está representado na figura 10 onde pode-se observar os tempos de liberação bem como os diversos *deadlines*.

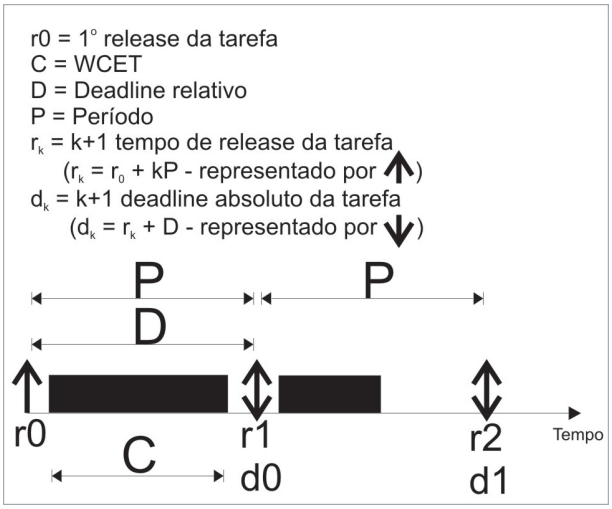

Figura 10 – Modelo de tarefa periódica

A partir desse modelo de tarefa é possível calcular a utilização u_i de cada tarefa τ_i sobre o EP como sendo ($u_i = \frac{C_i}{P_i}$). Assim a utilização *U* do EP é dada pelo somatório das utilizações de seu conjunto de tarefas: $U = \sum_{i=1}^{n} \frac{C_i}{P_i}$. Também se pode verificar o fator de carga que cada tarefa τ_i exerce sobre um EP como sendo ($ch_i = \frac{C_i}{D_i}$). O somatório dos fatores de carga de cada tarefa τ_i do conjunto de tarefas T representa o fator de carga CH do EP: $CH = \sum_{i=1}^{n} \frac{C_i}{D_i}$.

Além dos itens contemplados no modelo de tarefa, outros parâmetros devem ser definidos para a utilização do modelo proposto neste trabalho. O primeiro deles é o identificador id_i da tarefa, utilizado pelo sistema. Esse identificador, gerado pelo *kernel* de um EP, é utilizado entre os gerenciadores do balanceamento de carga durante a troca de informações sobre as tarefas do sistema, conforme será tratado no Capítulo 5.

Outro parâmetro importante a ser definido é o que diz respeito ao consumo energético e_i da tarefa, que pode ter seu valor definido de acordo com ferramentas de estimativa de consumo de energia, como a proposta por [39]. Por fim, uma tarefa τ_i possui uma utilização de memória m_i ao longo do tempo que, neste trabalho, também deve ser definida para que os gerenciadores propostos possam fazer suas verificações.

Além disso, é possível a utilização de tarefas aperiódicas como sendo um caso especial de tarefas periódicas, onde não existe período P_i. Assim, o modelo de tarefa aperiódica utilizado no trabalho é (id_i, r_i, C_i, D_i) cujos parâmetros têm significados similares aos empregados pelas tarefas periódicas.

Ainda, considera-se que tarefas periódicas têm período P_i com mesmo valor de *deadline* D_i ($P_i = D_i$) e se prevê, no sistema, a utilização de tarefas de melhor esforço como sendo as que não possuem um valor de *deadline* definido. Por fim, os parâmetros temporais das tarefas são expressos em unidades de *ticks*.

4.1.3 *Kernel*

O trabalho utiliza um *kernel* que possibilita o gerenciamento das tarefas do sistema respeitando restrições de tempo-real. Para isso, o *kernel* baseia-se na noção temporal de *ticks* e faz o escalonamento das diversas tarefas do sistema através de um módulo escalonador. É importante destacar que o modelo proposto prevê a utilização de um *gerenciador local ou de um global* (definido no Capítulo 5) que deve ser um sub-módulo do *kernel*.

4.1.4 Elemento de processamento - EP

O EP empregado no trabalho possui um conjunto de tarefas T que são executadas sobre o *kernel* a ser utilizado. Seu fator de utilização pode ser definido por $U = \sum_{i=1}^{n} \frac{C_i}{P_i}$ e seu fator de carga definido por $CH = \sum_{i=1}^{n} \frac{C_i}{D_i}$, cujos parâmetros foram explorados anteriormente. Um EP também apresenta um consumo de energia E ao longo do tempo, que é equivalente ao gasto

energético e_s mínimo utilizado para a manutenção do sistema somado ao total dos consumos de energia de cada tarefa τ_i do seu conjunto T de tarefas: $E = e_s + \sum_{i=1}^{n} e_i$.

O EP em questão possui uma memória privada cuja utilização M é dada pela soma entre a utilização de memória do sistema, a constante m_s, e do somatório dos fatores de utilização de memória de todas as tarefas τ_i do conjunto T, $\sum_{i=1}^{n} m_i$. Assim, a utilização de memória do EP é representado: $M = m_s + \sum_{i=1}^{n} m_i$.

Na figura 11 pode ser visualizado um EP e sua interação com outros componentes, tais como *kernel* e conjunto de tarefas.

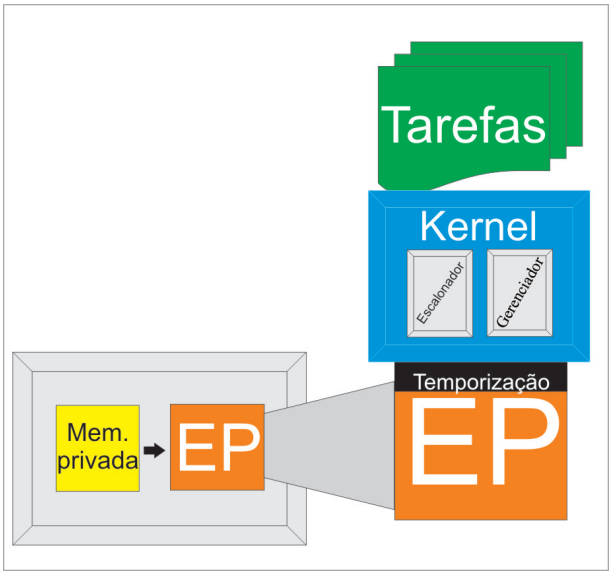

Figura 11 – Elemento de processamento abstrato empregado no trabalho

Por fim, cabe ao EP prover a noção de temporização do sistema através do conceito de *ticks*, conforme previamente apresentado.

4.1.5 Meio de interconexão

Neste trabalho assume-se que os componentes estejam interligados através de um meio de interconexão que possua o tempo de transmissão s_{ij} de uma mensagem entre dois componentes (i e j) dado em função da quantidade de dados transmitidos. A relação da quantidade de dados transferidos entre componentes com o tempo gasto para sua comunicação apresenta um comportamento linear.

Além disso, o meio de interconexão deve ser capaz de prover informações acerca do consumo de energia ϵ_{ij} gasto durante essa comunicação. Assim, utilizando o mesmo conceito

de quantidade de dados transferidos entre componentes pode-se dizer que seu comportamento também é linear, quando não existem colisões.

4.1.6 Arquitetura do sistema

Após a completa descrição de todos os componentes do sistema, pode-se exemplificar através da figura 12, a união desses componentes como sendo a arquitetura empregada para validar o modelo de balanceamento de carga proposto neste trabalho.

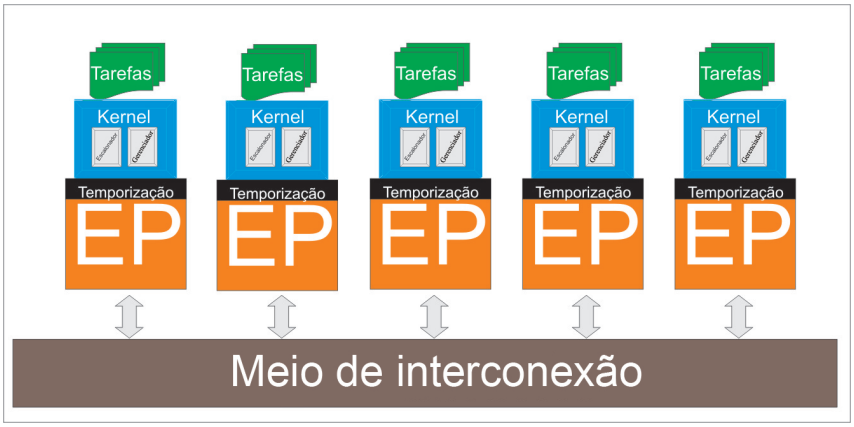

Figura 12 – Arquitetura do sistema

4.2 Plataforma de validação

A plataforma utilizada, baseada no estudo de [39], é composta por microprocessadores MIPS, por um meio de interconexão e por módulos de memória. O número de processadores utilizados na plataforma está relacionado ao desempenho necessário para a execução da validação do modelo proposto. Essa plataforma pode ser tanto simulada como implementada como um protótipo em um dispositivo FPGA. No caso da implementação em FPGA existe uma limitação natural quanto ao número de processadores devido a área (tamanho) do dispositivo disponível. É importante destacar que a escolha de alguns componentes arquiteturais, como o processador MIPS e a utilização de um barramento como meio de interconexão, tem sua justificativa apontada no estudo de [39] e não fazem parte do escopo da presente pesquisa. A seguir estão expostos detalhes das estruturas empregadas neste estudo.

4.2.1 *Kernel*

O *kernel* inicialmente desenvolvido em [39] cuja implementação vem sido continuada no Grupo de Sistemas Embarcados (GSE) da PUCRS é um micro-kernel composto de diversos módulos. Diversos itens do kernel podem ser configurados, tais como: o número máximo de tarefas em um processador, o tamanho da pilha de cada tarefa, o tamanho da área de *heap* do processador, tamanhos das filas de comunicação, tamanho dos pacotes de comunicação, prazo para expiração dos pacotes, política de escalonamento e o uso ou não de comunicação dirigida a interrupções. O objetivo ao permitir tal nível de configurabilidade é capacitar a otimização do tamanho da imagem final do *kernel* tornando possível sua execução mesmo em arquiteturas que possuam restrições de memória. Os periféricos são acessados via E/S mapeada em memória cuja configuração para uma solução específica pode ser realizada na camada de abstração de *hardware* (do inglês, *Hardware Abstraction Layer* - HAL).

A estrutura do *kernel* empregado está ilustrada na figura 13 onde se pode visualizar a camada HAL que contém todas as funções e definições específicas do *hardware*. O micro-kernel é, então, implementado sobre essa camada. Funções padrão da linguagem *C*, bem como a API do *kernel* estão implementadas no topo do kernel tornando disponíveis a camada de *drivers* e de aplicações de usuário. Outras facilidades do *kernel* como *driver* de migração, gerência de memória e exclusão mútua são implementadas no kernel e a aplicação encontra-se no topo da pilha.

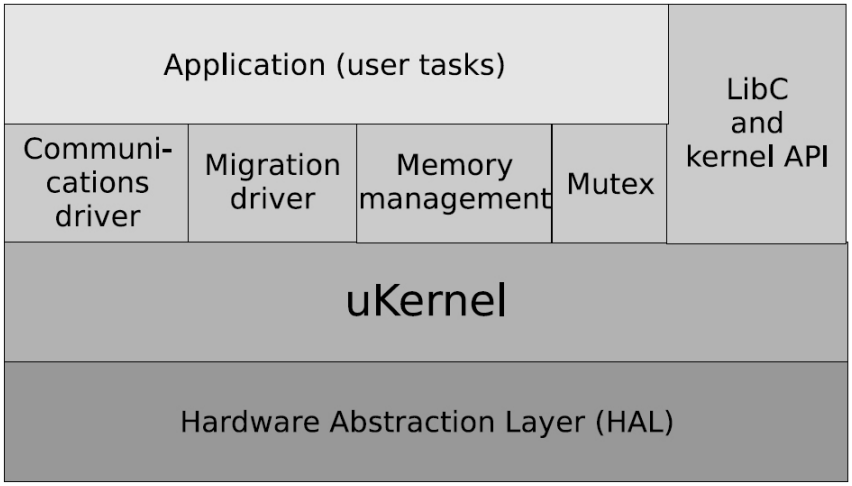

Figura 13 – Estrutura do *kernel* utilizado

As únicas partes do *kernel* dependentes de *hardware* são a rotina de tratamento de interrupção, o processo de inicialização das interrupções e as funções de restauração e salvamento

de contexto. Essas funções foram escritas em *assembly* (assim como parte da HAL) embora possam ser portadas para outras arquiteturas.

Após as etapas de compilação e ligação (do inglês, *linking*), um único binário é gerado contendo o *kernel* e a aplicação. Esse binário pode então ser transferido aos processadores de acordo com suas funções no sistema. No ambiente multiprocessado utilizado neste trabalho, cada processador contém sua própria imagem do kernel.

Com relação ao funcionamento do *kernel* é possível observar um fluxo básico de execução do sistema na figura 14. Com o intuito de facilitar a compreensão do fluxo, não são levadas em consideração, para esse exemplo, questões dinâmicas envolvendo criação e destruição de tarefas durante a execução da aplicação. O fluxo consiste, basicamente, em:

- *inicialização do SO*, onde estruturas de dados são configuradas e inicializadas. Além disso, as funções das tarefas são registradas e adicionadas ao *kernel*;

- *execução do SO*, onde interrupções são registradas e inicializadas e, então, habilitadas. Caso uma interrupção ocorra, o tratador de interrupções é acionado e um temporizador de interrupção, denominado de *dispatcher ISR* é executado, o contexto salvo e o escalonador chamado. Após o escalonamento, o contexto da tarefa escolhida para execução é restaurado. Caso a tarefa libere o processador por conta própria (*yields*), o *dispatcher ISR* é chamado. Caso uma interrupção ocorra, o tratador de interrupções é acionado. Outras interruções são tratadas da mesma maneira que as de temporização com a exceção de que não há necessidade de reescalonamento.

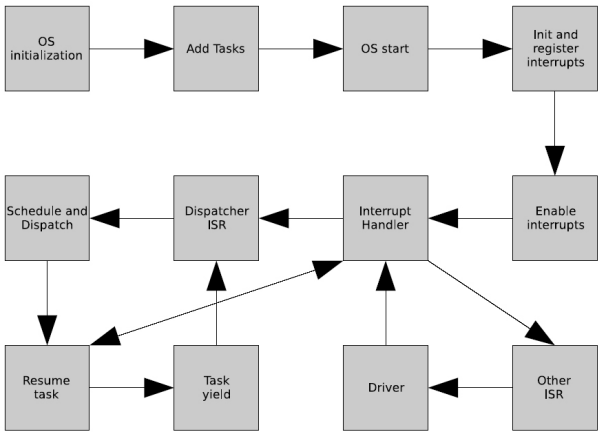

Figura 14 – Fluxo de execução do *kernel* utilizado

4.2.2 Modelo da tarefa

No *kernel* utilizado, uma tarefa é definida como um bloco de código que coexite com outros recursos do sistema assim como com outras tarefas. Pode ser considerada o bloco de execução básico e é escalonada de acordo com a política de escalonamento vigente. Uma tarefa pode ser melhor compreendida como sendo uma função que itera continuamente ignorando o fato de que será preempatada e reescalonada posteriormente.

Uma tarefa é preemptada somente quando uma interrupção de *hardware* ocorre (temporização, comunicação ou outro evento) ou se a própria tarefa decide ceder o processador permitindo que o *kernel* escolha outra tarefa para ser executada.

A máquina de estados utilizada pelo *kernel* implementado prevê três situações nas quais uma tarefa pode encontrar-se: *pronta*, *executando* ou *bloqueada*. Uma tarefa é considerada *pronta para execução* quando foi preemptada pelo *kernel* ou ainda não foi executada. Nesse estado, a tarefa encontra-se na fila de escalonamento e aguarda que o escalonador a escolha para execução. Já uma tarefa está no estado de *executando* quando o *dispatcher* do *kernel* restaura seu contexto e a tarefa assume controle do processador. Já o estado *bloqueado* é utilizado em situações especiais, como para evitar o problema da inversão de prioridade [40] em semáforos ou para manter um *driver* de dispositivo bloqueado.

Para prover execução de tempo real, aplicações que se encontram no espaço do usuário (tarefas) não têm permissão para desabilitar interrupções uma vez que, ao desabilitar interrupções, mesmo as interrupções referentes aos temporizadores não serão tratadas e, então, a resposta de tempo real pode ser comprometida.

As interrupções de temporizadores são utilizadas para gerar a noção de tempo para o sistema através de *ticks*. O período de cada *tick* deve ser bem balanceado já que fatias de tempo muito longas tornam o sistema com menor capacidade de responder prontamente (e não são considerados de tempo real) enquanto que fatias de tempo muito curtas aumentam o *overhead* gerado pelas trocas de contexto. Uma fatia de tempo de 10.48ms (aproximadamente) foi empregada, fatia essa que se encontra dentro dos limites comerciais da definição de *kernel hard real time*, cujo período máximo gira em torno dos 20ms. Essa fatia de tempo foi obtida através da utilização de interrupções geradas pelo *hardware*. Para o controle de geração de interrupção de *timer*, o processador Plasma emprega um registrador de 32 bits interno à sua arquitetura. Esse registrador tem seu valor incrementado a cada ciclo de *clock* e pode ser acessado através de leituras na memória.

Nesse caso, para a plataforma empregada, uma interrupção é gerada quando o *bit* 18 desse registrador - que funciona, também, como um *timer* - atinge o estado 1. O processador Plasma

é prototipado para executar a uma freqüência de operação de 25MHz, o que serve de base para o cálculo do período e freqüência da interrupção de *timer*. Quando o bit 18 entra no estado 1, permanece assim por 2^{18} ciclos, quando entra no estado 0, de mesma duração. O número de ciclos de um período é, então, definido pela soma desses dois estados: $2^{18} + 2^{18} = 2^{19}$. O cálculo em tempo do período em milisegundos é obtido da seguinte forma:

$$periodo = \frac{ciclos\ de\ periodo}{\frac{clock}{1000}}$$

$$periodo = \frac{2^{19}}{25000}$$

$$periodo = 20.97152ms^{1}$$

A utilização do *bit* 18 deu-se em função de esse apresentar o período mais próximo do desejado: de *hard real time* mas que não representasse um *overhead* muito alto de *kernel*. É importante destacar que quanto menor o período, mais vezes o *kernel* vai ser executado, o que nem sempre é um fator positivo, uma vez que as tarefas serão interrompidas a todo momento, representando um *overhead* desnecessário.

Finalmente, todas as informações que dizem respeito às tarefas estão armazenadas em uma estrutura de dados denominada de bloco de controle da tarefa (do inglês, *task control block* - TCB). Nessa estrutura, o *kernel* mantém todas as propriedades da tarefa, tais como identificador, descritor, estado da tarefa, número de *ticks* já executados, período, prazo de execução (deadline), contexto, ponteiro para seu ponto de partida, pilha, filas de comunicação e informação sobre os pacotes de comunicação. É importante destacar que os itens descritos no modelo de tarefa descrito anteriormente são mapeados, na implementação real, nas diversas propriedades existentes na TCB.

Política de escalonamento

Existem diversas políticas de escalonamento que têm sido empregadas com sucesso em diversos sistemas operacionais ao longo dos anos. Dentre as mais utilizadas podem ser citadas: *first come first served* (FCFS) [41], *round robin scheduling* [42], [43], *rate monotonic* (RM) [44], [45], [46] e o algoritmo *earliest deadline first* (EDF) [47], [48].

[1] obtido quando a interrupção utilizada é consultada somente na borda de subida do *clock*. A implementação empregada modifica a máscara de interrupção a cada iteração da rotina. Em um primeiro momento, a máscara torna-se sensível à borda de subida e, durante o tratamento da interrupção, a mesma é modificada para ficar sensível à borda de descida. O inverso acontece na borda de descida. Dessa forma, o período possui valor correspondente à metade de 20.97152 ms, ou seja, de aproximadamente 10,48 ms.

Neste trabalho, a necessidade por um algoritmo que tivesse as características de ser leve, preemptivo, justo e de tempo real fez com que fosse adotado um esquema de *round robin* com prioridades, onde tarefas com menores períodos têm prioridade sobre outras tarefas. Apesar disso, sempre há a garantia de que todas as tarefas serão executadas de um modo cíclico. Em vários aspectos, o algoritmo adotado se assemelha ao escalonamento gerado pela política *rate monotonic*.

As políticas de escalonamento para o *kernel* são implementadas diretamente no *dispatcher*, que é o módulo responsável por selecionar uma nova tarefa que esteja apta para executar, além de salvar o contexto da tarefa que está deixando a execução bem como restaurar o contexto da tarefa selecionada para o processamento.

Comunicação inter-tarefa

Em um ambiente multi programado sempre há a necessidade de comunicação entre as tarefas paralelas que executam no sistema. Na implementação de *kernel* utilizada neste trabalho, duas abordagens foram empregadas com o intuito de prover comunicação entre as tarefas.

Primeiramente, em um único processador, as comunicações são tratadas utilizando memória compartilhada. Um bloco compartilhado da memória é visível para mais de uma tarefa e através de primitivas de exclusão mútua, como *mutexes* e semáforos a comunicação pode ser feita e a integridade dos dados garantida.

Já no caso de comunicação entre múltiplos processadores, as comunicações pertinentes às tarefas são tratadas de forma diferenciada, uma vez que é adotado o paradigma da troca de mensagens. As primitivas básicas que permitem são as de *envio* e *recebimento* de pacotes. Nesse caso, cada tarefa possui uma fila privada destinada às questões da comunicação e que contém todas as mensagens recebidas.

Migração de tarefas

Para que o modelo de balanceamento de carga fosse possível de ser implementado, tornou-se necessário o desenvolvimento de um mecanismo de migração de tarefas para o *kernel* empregado. Como pôde ser visto no capítulo de trabalhos correlatos, diversos estudos concentram seus esforços em definir a melhor técnica para casos específicos de migração de tarefas. No caso do presente trabalho, definiu-se um mecanismo pelo qual a migração de tarefas ocorre e, mesmo estando ciente de que o mecanismo empregado pode e deve afetar o desempenho final do modelo proposto, a intenção deste trabalho é validá-lo em uma arquitetura real não necessitando ser essa a melhor opção disponível. Dessa forma, estudou-se uma forma simples, porém

eficiente, de se possibilitar a migração de tarefas no *kernel* empregado.

É importante destacar que, por si só, o processo de migração tende a ter impacto negativo no desempenho do sistema, uma vez que aumenta o *overhead* com relação ao seu tempo de resposta. Com o intuito de diminuir as penalidades de desempenho utiliza-se, neste trabalho, o conceito de migração parcial. Nesse caso, apenas os dados das tarefas são migrados de um processador para o outro. A principal restrição dessa abordagem é que deve haver replicação de código entre os processadores.

Para lidar com o processo de migração, um *driver* foi implementado na forma de uma tarefa com prioridade alta estando inicialmente bloqueado. Quando um processador recebe uma mensagem de migração, o *kernel* cria uma nova tarefa, recebe e restaura os dados da tarefa migrada, recalcula o *stack pointer* entre outros itens necessários e reinicia a nova tarefa. Quando este processo está completo, o *driver* de migração bloqueia a si mesmo evitando ser reescalonado em momento inapropriado.

Finalmente, o mecanismo de migração implementado garante que uma tarefa não possa migrar a si mesma. Dessa forma, para que uma migração ocorra uma outra tarefa deve, explicitamente, invocar a chamada de migração de tarefas para que a migração possa, de fato, ocorrer. Nesse caso, a tarefa que invoca o mecanismo de migração corresponde ao gerenciador local do modelo proposto, que apenas atende às requisições vindas do gerenciador global, como será explanado no próximo capítulo.

4.2.3 Processador Plasma

Os elementos de processamento utilizados neste trabalho são, na verdade, processadores Plasma [12]. Originalmente criado em [12], esse processador é um *soft-core*[2] descrito em VHDL e que implementa parte do conjunto de instruções MIPS. A freqüência de operação do processador empregado é de 25 MHz.

4.2.4 Barramento de interconexão

De acordo com a plataforma inicialmente criada em [39], o meio de interconexão escolhido para o MPSoC desenvolvido foi um barramento, descrito em VHDL, simulado e prototipado

[2]Descrição RTL em alto nível de um módulo de *hardware*, que pode ser visualizada, modificada ou adaptada para um determinado fim.

com sucesso. Com relação a interface entre o barramento, denominado de *HotWire bus*, e o processador Plasma, existe um adaptador que implementa a lógica de amarração (do inglês, *wrapping*) dos registradores de E/S dos processadores e do barramento, cuja largura de dados é de 20 *bits*. Além disso, o adaptador realiza o endereçamento dos processadores garantindo que dados não pertencentes a um determinado destino sejam descartados.

O barramento é formado por 4 *bits* de endereços e 16 *bits* de dados e apenas uma única palavra de dados é entregue ao barramento após a arbitragem. Leituras e escritas no barramento são realizadas de acordo com o *software* implementado em cada processador. Cabe ao *kernel* prover tais primitivas através da utilização de *drivers* que possuem acesso de baixo nível ao barramento. Na tabela 1 podem ser vistas, resumidamente, as principais características do barramento empregado.

Tabela 1 – Características do barramento

Largura da palavra	20 *bits* (parametrizável)
Número de portas	4 (parametrizável)
Tamanho das filas	8 *bytes* (nas portas de saída, parametrizável)
Arbitragem	algoritmo rotativo, sem prioridade
Latência	2 ciclos por palavra, após arbitragem
Freqüência de operação	25MHz (protótipo)
Ocupação em área	8372 portas lógicas
Throughput	100Mb/s (8 *bits* de dados @ 25MHz)
Protocolo de E/S	*handshake*

4.2.5 Arquitetura

A arquitetura final utilizada para a validação do trabalho está exibida na figura 15 e corresponde a um MPSoC homogêneo executado a 25 MHz, composto de quatro processadores *MIPS-like* e que se comunicam através de um barramento dedicado.

Cada um dos quatro processadores é composto por um núcleo, duas memórias (memória de *boot* e uma memória externa, onde são executadas as aplicações e o *kernel*) e uma UART, utilizada como E/S genérica através de um terminal, bem como empregada para carregar os binários nos processadores correspondentes. A arquitetura completa foi descrita em VHDL, simulada e prototipada em uma placa *Xilinx XUP* [49]

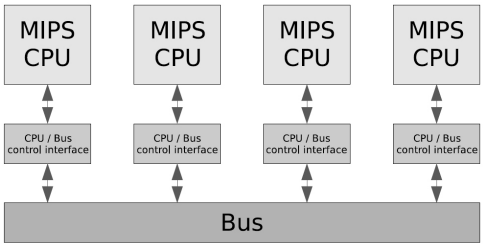

Figura 15 – Arquitetura do MPSoC de validação

4.2.6 Ferramenta para depuração gráfica

Além da plataforma descrita, foi desenvolvida, para este trabalho, uma ferramenta gráfica com o intuito de facilitar a depuração do sistema. Essa ferramenta, implementada na linguagem Java, aceita uma entrada descrita em XML e apresenta a execução gráfica do sistema. Para que isso fosse possível, uma funcionalidade de *log* foi adicionada ao *kernel* empregado com o intuito de armazenar informações pertinentes à execução do sistema.

Esse *log* contém informações pertinentes à execução do sistema na granularidade de *ticks*. A cada *tick* sabe-se exatamente o que está ocorrendo no sistema. Cada processador tem sua informação registrada sendo que esses registros são unificados para que a utilização da ferramenta gráfica seja possível. As informações sobre as tarefas concernem, basicamente, àquelas descritas no modelo de tarefa, como identificador da tarefa, *deadline*, carga, consumo energético entre outros. Na figura 16 pode ser visualizado um exemplo de tela cuja entrada foi alimentada por uma saída do *kernel*.

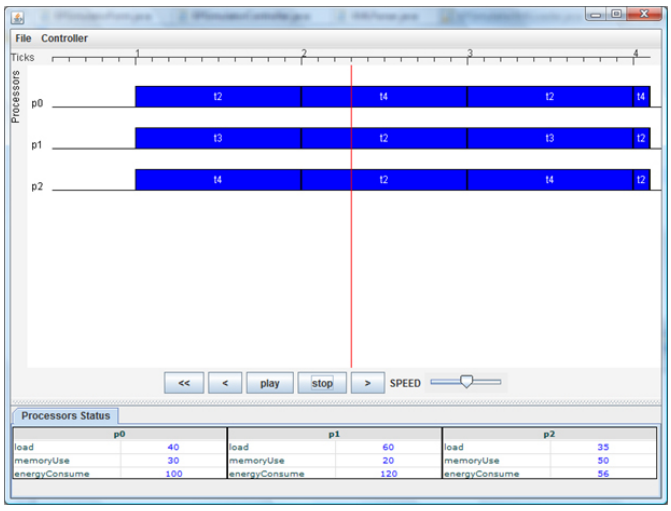

Figura 16 – Ferramenta para depuração do sistema

Além dessas informações, itens referentes às migrações também são armazenados no registro para que na ferramenta gráfica possa se visualizar o momento em que uma migração ocorre. Entre essas informações encontram-se origem e destino da migração e tempo estimado para essa comunicação. Na tela principal da ferramenta é possível visualizar-se ainda a situação de cada elemento de processamento com relação à carga atual e, através de cores, identificar se estão em uma situação normal, de sobrecarga ou de subutilização. Assim, o *kernel* empregado gera uma saída que é, posteriormente, carregada na ferramenta de depuração para melhor visualização do sistema.

5 Modelo de balanceamento de carga através de migração de tarefas

O modelo proposto visa permitir que o sistema, de maneira dinâmica, tenha seu comportamento verificado em relação à carga e, através da migração de tarefas, tenha sua situação normalizada ao longo de seu tempo de vida como um todo. Para isso, possui duas estruturas principais: o gerenciador global e o monitor local. No sistema existe apenas um gerenciador global responsável pelo monitoramento de seu estado geral. Já os monitores locais estão presentes em cada EP e os monitoram para, então, comunicar-se com o gerenciador global que tomará as decisões pertinentes. Nas próximas seções são descritos os detalhes envolvendo essas duas estruturas além de suas funcionalidades internas e interação com os demais componentes do sistema.

5.1 Monitor local

Cada EP do sistema possui um módulo em seu *kernel* denominado de monitor local. O monitor local consulta uma lista com as tarefas que executam no EP e, eventualmente envia essa lista para o gerenciador global. Além disso, o monitor local pode, também, receber do gerenciador global uma ordem de migração de tarefas. Nesse caso, o gerenciador global envia qual tarefa deve ser migrada e qual o EP destino da migração. Quando há uma migração, cabe ao gerenciador local enviar as tarefas designadas ao EP destino da migração através do meio de interconexão utilizado.

A simplicidade do monitor local tem como principais objetivos: (i) concentrar em somente uma estrutura a tomada de decisão acerca das decisões envolvendo balanceamento de carga e (ii) não sobrecarregar um EP que possui as tarefas críticas do sistema. Assim, todas as decisões que exigem maior poder computacional para serem tomadas são feitas diretamente no gerenciador global que apenas repassa os resultados pertinentes aos monitores locais envolvidos em possíveis migrações.

Em termos de implementação, o monitor local foi desenvolvido como sendo um *driver* do *kernel* utilizado. Dessa forma, entra como uma tarefa no sistema contendo uma prioridade

bastante baixa com o objetivo de não interferir na execução das outras tarefas do sistema. O fato de o gerenciador ser implementado como uma tarefa não influencia na decisão do gerenciador global pelos seguintes motivos: (i) uma tarefa não pode migrar a si mesma; (ii) o gerenciador local envia somente informações das demais tarefas do sistema, não contando ele próprio como uma tarefa apta a ser migrada.

5.2 Gerenciador global

O modelo proposto contém uma estrutura centralizada responsável pelo gerenciamento das questões de balanceamento de carga do sistema. Essa estrutura, denominada de *gerenciador global*, realiza o monitoramento dos EP's do sistema e, com base nos dados coletados, decide se migrações de tarefas entre os EP's são necessárias. Além disso, é responsável por coordenar as migrações em vários aspectos, como origem, destino e conteúdo.

Assim como o monitor local, o gerenciador global é caracterizado por uma camada de *software* executada sobre o *kernel* de um elemento de processamento caracterizados ao longo da Seção 4.2. Para desempenhar suas funcionalidades da maneira correta, o gerenciador global requer, também, o uso de uma memória para armazenar as informações acerca do estado do sistema.

Com relação à implementação do gerenciador global, um elemento de processamento dedicado foi empregado. Embora a utilização de um EP dedicado evidencie um certo desperdício de área é importante lembrar que lógicas de controle quando implementadas diretamente em *hardware* podem possuir um custo de área elevado além da manutenção desse módulo ser mais cara em termos de dificuldade e tempo. Além disso, existe uma tendência de que a maior parte das lógicas de controle sejam, quando possível, implementadas em *software* uma vez que essa abordagem demonstra uma flexibilidade maior quando comparada à implementação de *hardware*. Ainda, essa tendência também traz um ganho na manutenibilidade do sistema que é beneficiada porque projetistas devem preocupar-se somente com a aplicação em si não levando em consideração questões específicas de um módulo de *hardware*.

A seguir são apresentados os detalhes envolvendo o gerenciador global, suas funcionalidades e interação com outros elementos do sistema.

5.2.1 Monitoramento do sistema e definição da situação dos EP's

Na primeira execução do sistema o gerenciador global inicia suas atividades em um estado de espera. Nesse estado, ele aguarda que cada monitor local envie as informações correspondentes de seus elementos de processamento. Após essa comunicação inicial, os monitores locais devem enviar novas informações para o gerenciador global somente quando suas cargas alterarem de forma significativa. Isso faz com que, além de manter o gerenciador global atualizado sobre a carga de seu EP, não existam comunicações desnecessárias no sistema.

O gerenciador global recebe de cada monitor local uma listagem η contendo informações sobre todas as tarefas que estejam na fila de tarefas prontas do EP correspondente. Sobre cada tarefa τ_n, deve-se enviar as informações contidas no seu modelo $(id_i, r_i, C_i, D_i, P_i)$, além do consumo energético e_i, da utilização da memória m_i e do fator de carga CH_i da tarefa τ_n sobre o EP. Além dessas informações que concernem cada tarefa, o monitor local envia o estado de carga total e o consumo energético total do elemento de processamento por ele monitorado.

Com base nessas informações o gerenciador global tem condições de verificar o estado do sistema em relação à carga, ao consumo de energia e à utilização de memória além de classificar cada EP, de acordo com sua situação, em uma das seguintes categorias:

1. *UL* - subutilizado, (do inglês, *UnderLoaded* - UL)

2. *BA* - abaixo da média do sistema, (do inglês, *Below Average* - BA)

3. *AV* - na média do sistema, (do inglês, *Average* - AV)

4. *AA* - acima da média do sistema, (do inglês, *Above Average* - AA)

5. *OL* - sobrecarregado (do inglês, *OverLoaded* - OL).

Para cada aspecto considerado - carga e consumo de energia - deve-se calcular a média do sistema: $\frac{\sum_{i=1}^{n}(x)}{n}$, onde n é o numero de EP's, i representa cada EP e x representa o valor obtido para o aspecto considerado. Com esse cálculo obtêm-se os valores: da média AV_l de carga e da média AV_p de consumo de energia.

Com o intuito de simplificar a classificação dos EP's serão considerados somente três estados possíveis:

1. *UL* - subutilizado, (do inglês, *UnderLoaded* - UL)

2. *AV* - normal,

3. *OL* - sobrecarregado (do inglês, *OverLoaded* - OL).

Além da média, devem ser estipulados os valores que definirão o intervalo da normalidade (para cada aspecto), compreendido pela categoria AV, apresentada anteriormente. Existe um limite baixo (Δ_u) que definirá a diferença entre AV e UU e um limite alto (Δ_o) que definirá a diferença entre AV e OL.

O cálculo que define o valor desses limites irá depender do comportamento do sistema, ou seja, depende do valor de média AV_x obtido. Assim, foram definidas três situações distintas para o cálculo dos limites baixo (Δ_u) e alto (Δ_o) que determinam o intervalo de normalidade do sistema:

1. caso a média do sistema esteja muito alta ($AV > 80\%$(% da utilização máxima do sistema)):
   $\Delta_o = AV$
   $\Delta_u = 20\%$(da utilização máxima do sistema)

2. caso a média do sistema esteja muito baixa ($AV < 20\%$(% da utilização máxima do sistema)):
   $\Delta_o = 80\%$(da utilização máxima do sistema) $\Delta_u = AV$

3. caso a média do sistema esteja intermediária ($20\% \leq AV \leq 80\%$(% da utilização máxima do sistema)): $\Delta_o = 80\%$(da utilização máxima do sistema) $\Delta_u = 20\%$(da utilização máxima do sistema)

Os valores considerados como a base alta e a base baixa para a definição dos Δ_o e Δ_u, 80% e 20%, respectivamente, podem ser alterados pelo usuário do modelo de forma individual para cada aspecto:

- Δ_{uch}, limite inferior para o aspecto de carga;
- Δ_{up}, limite inferior para o aspecto de consumo de energia;
- Δ_{och}, limite superior para o aspecto de carga;
- Δ_{op}, limite superior para o aspecto de consumo de energia;

Esses valores interferem na maneira pela qual o sistema calculará, dinamicamente, os valores que determinam os estados de cada EP. Dessa forma, sua alteração, apesar de estática,

tem conseqüências que podem ser observadas dinamicamente durante a execução do sistema. Como padrão, o modelo considera a mesma base para todos os aspectos. Embora o aspecto consumo de energia possua limite inferior (para que os EP's possam ser categorizados) somente é considerado problemático o estado de OL.

A figura 17 exibe as categorias nas quais um EP pode ser enquadrado em relação a sua situação de carga e de consumo de energia. Nessa figura pode-se observar, além das categorias, os limites baixo Δ_u e alto Δ_o que definem o intervalo da normalidade.

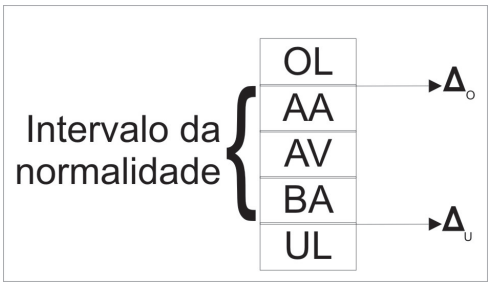

Figura 17 – Categorias de um EP (em relação a sua situação de carga e de consumo de energia)

A implementação dessa parte do gerenciador consiste em avaliar as informações enviadas pelos monitores locais e armazená-las em estruturas de dados pertinentes. Após essa etapa e baseando-se nos valores referenciais para cada aspecto, são calculados os valores de Δ_o e Δ_u que definem, então, o que será considerado normal, sobrecarregado ou subutilizado. A última etapa consiste em analisar as informações referentes a cada EP para os aspectos considerados (carga e consumo de energia) e categorizá-los diante dos intervalos determinados previamente. Uma vez que a categorização esteja pronta, o gerenciador global pode desempenhar a sua funcionalidade responsável por garantir o balanceamento do sistema.

5.2.2 Balanceamento

Baseado na análise da situação de cada EP, o gerenciador global deve tomar as decisões pertinentes para que o maior número possível de EP's alcançe o intervalo da normalidade, nos aspectos considerados. Como pôde ser observado, o gerenciador global tem conhecimento acerca da situação de cada EP do sistema e, por isso, irá definir:

- se é necessário migrar tarefas;
- qual deve ser o EP origem de uma migração;

- qual tarefa deve ser migrada
- qual deve ser o EP destino de uma migração.

O modelo proposto visa migrar apenas uma tarefa por vez quando isso for necessário. Essa abordagem poderia ser substituída por uma na qual várias tarefas ou um conjunto de tarefas fosse migrado de uma única vez. Apesar disso, a primeira abordagem foi adotada por se mostrar vantajosa em diversos aspectos:

- o tempo de procura da tarefa a ser migrada é menor: ao invés de buscar um conjunto que satisfaça uma série de regras, como requisitos de carga e temporais, deve-se encontrar apenas uma tarefa que se enquadre melhor ao perfil desejado;
- o modelo respeita a dinamicidade intrínseca dos sistemas embarcados uma vez que nesses sistemas normalmente as tarefas possuem tempos de execução bastante curtos. Assim, é maior a chance de o gerenciador global trabalhar com informações que reflitam fielmente o estado atual do sistema uma vez que seu tempo de execução é minimizado por escolher apenas uma tarefa;
- normalmente, nos meios de interconexão, tanto o tempo gasto para uma comunicação quanto seu consumo energético variam em função da quantidade de dados trafegados. Assim, quando diversas tarefas são enviadas de uma única vez, além do tempo de envio ser maior, o consumo energético também é maior, o que pode invalidar os possíveis benefícios alcançados pela migração de tarefas;
- apesar de não garantir que, em uma única interação, sejam resolvidos os problemas referentes ao excesso ou falta de carga de um determinado EP, o modelo proposto garante que o sistema permanecerá em equilíbrio a maior parte de seu tempo de vida.

Dependendo da situação do sistema, as ações pertinentes serão tomadas. Dentre as categorias apresentadas, são consideradas problemáticas apenas duas: subutilizado (UL) e sobrecarregado (OL). Assim, o gerenciador global irá considerar a realização de migrações de tarefas somente se existir um ou mais EP's que estejam UL ou OL (ou ainda se a situação do sistema estiver híbrida, com EP's em UL e OL).

Após verificar que existem EP's em situação de OL ou UL, deve-se selecionar os candidatos para as migrações de tarefas pertinentes. Esses candidatos devem ser capazes de suportar a nova carga sem reflexos negativos no seu consumo de energia. A seguir, os detalhes que envolvem o tratamento das situações de sobrecarga e de subutilização são apresentados.

Sobrecarga

A sobrecarga é caracterizada quando o EP ultrapassa o limite superior Δ_{och}. Esse EP em sobrecarga é dito como sendo o EP *origem* da migração e o modelo proposto, além de selecionar a tarefa a ser migrada, prevê a descoberta de um *destino* para essa migração, considerando os outros EP's como sendo candidatos.

Primeiramente, para os casos onde há sobrecarga de um EP, deve-se definir qual a carga a ser transferida que o faz entrar no intervalo de normalidade. O valor a ser atingido, em caso de sobrecarga, é de 70% da utilização máxima de todo o sistema, ou seja, a carga a transferir CH_t é igual à carga desse EP CH_i diminuída de 70% da utilização máxima do sistema: $CH_t = CH_i - 70\%$.

A seleção da tarefa a ser migrada deve respeitar o limite de carga a transferir CH_t. Assim, analisando a listagem η das tarefas do EP em OL, deve-se ordená-la, primeiramente pelo tipo da tarefa: melhor esforço e periódica de tempo real (ordenadas pelo período, de forma decrescente). Esse ordenamento é imprenscindível porque são as tarefas de melhor esforço aquelas que não possuem restrições temporais facilitando o processo de migração. Após esse ordenamento inicial, deve-se novamente ordenar a listagem η de forma descrescente, de acordo com a carga CH_i de cada tarefa e verificar qual dessas tarefas mais se aproxima do valor desejado de carga a transferir CH_t. Optou-se por esse método por ser uma opção com baixo custo computacional apesar de, possivelmente, não apresentar os melhores resultados na busca da tarefa ideal para a migração quando comparado a soluções heurísticas, por exemplo (mais sofisticadas e com maior custo computacional). Assim, define-se uma tarefa τ_t que deverá ser transferida para o EP selecionado entre os candidatos disponíveis para receber a migração.

A seleção dos candidatos que irão receber a carga excedente CH_t requer que os EP's do sistema sejam ordenados de acordo com sua categoria em UL e AV. EP's em OL não são considerados candidatos para a transferência de tarefas com outros EP's em OL. Após a ordenação, um teste define quais EP's são candidatos para receber a carga excedente CH_t: verifica-se se a carga excedente da tarefa que está sendo transferida CH_t somada à carga atual do candidato y, CH_y é menor que o limite superior Δ_{och} estipulado ($CH_t + CH_y < \Delta_{och}$).

Uma vez obtendo um conjunto de EP's capazes de receber a carga a ser trasnferida de um EP em OL, deve-se verificar o aspecto consumo de energia desse conjunto C de candidatos. Assim, o consumo de energia E_y atual de um candidato y acrescido do consumo estimado da tarefa a ser transferida τ_t não pode ultrapassar o limite superior Δ_{oe}.

Para o caso de uma tarefa τ_t de melhor esforço, deve-se selecionar qualquer candidato que tenha passado nos testes anteriores. Para listagens de tarefas τ_t que possuam tarefas de tempo real, o tempo de comunicação entre o EP origem e o candidato y à receber a migração deve ser

avaliado. Para isso, deve-se verificar se nenhuma tarefa de tempo real terá *deadline* descumprido em caso de migração para o candidato y. Caso a tarefa de tempo real não possua nenhum candidato que consiga atender sua requisição, então deve ser escolhida uma nova tarefa que possa ser migrada para aliviar a carga do processador em OL. Se mesmo após verificar-se todas as tarefas do EP problemático não for possível encontrar um candidato para receber migração, deve-se considerar que esse conjunto está vazio e que não há nenhum candidato para migração.

Caso não exista nenhum candidato que satisfaça as condições para migração de pelo menos uma tarefa do EP em situação de OL, o gerenciador global armazena essa requisição em uma área de requisições pendentes e, na próxima iteração para verificar a situação do sistema, analisa a possibilidade de atender tais pendências.

É importante salientar que, caso o EP em sobrecarga não estiver garantindo mais o cumprimento de *deadline* - isso ocorre quando a taxa de utilização de sua CPU ultrapassa sua capacidade máxima de 100% - a seleção dos candidatos à migração sofre uma alteração: verifica-se a possibilidade desse candidato receber uma determinada carga do EP em OL caso essa carga respeite a taxa de utilização máxima de 100% do EP destino da migração. Isso significa que, para esses casos especiais onde *deadlines* podem estar perdidos, é preferível que outros EP's entrem em sobrecarga desde que permaneçam abaixo da sua capacidade máxima.

Os testes devem ser feitos para todos os elementos em sobrecarga, sendo que, se houver mais de um elemento em sobrecarga, deve-se atendê-los por ordem decrescente de sobrecarga (do mais sobrecarregado para o menos sobrecarregado). É importante frisar que o gerenciador global não irá atender todas as requisições de uma única vez: ele irá ordenar uma determinada migração, verificar o estado resultante do sistema, para então, se necessário, ordenar mais migrações.

Subutilização

Quando houver EP's em UL (e não houver outros em OL) esses EP's são considerados os *destinos* de uma migração. Nesse caso, o modelo prevê a forma pela qual os candidatos à *origem* são selecionados.

No caso de subutilização do sistema uma técnica semelhante àquela empregada no tratamento dos casos de sobrecarga deve ser adotada: deve-se definir a carga que o EP subutilizado precisa receber para que ingresse no intervalo de normalidade. O valor a ser atingido, em caso de subutilização, é de 25% da utilização máxima de todo o sistema, ou seja, a carga a receber L_r é igual à 25% da utilização máxima do sistema diminuída da carga CH_n desse EP: $CH_r = 25\% - CH_n$.

Após definir a carga a receber, a seleção dos candidatos ocorre de maneira semelhante à seleção dos candidatos nos casos de sobrecarga. Inicialmente, ordena-se os EP's restantes de acordo com sua categoria em OL e AV. EP's em UL não são considerados candidatos para cederem tarefas a outros EP's em UL. Após essa ordenação, define-se quais EP's são candidatos para enviar a carga CH_r necessária pelo EP em UL, através de um teste: verifica-se se a carga atual do candidato y, CH_y diminuída da carga CH_r a ser transferida para o EP em UL respeita o limite inferior Δ_{uch} previamente definido ($CH_y - CH_r > \Delta_{uch}$).

Dentre os candidatos que passarem nesse teste, verifica-se aquele que possui menor tempo de comunicação com o EP destino da migração. Desse EP deve-se selecionar uma tarefa que possa ser enviada para o EP em UL. A seleção da tarefa é idêntica aquela encontrada na situação de sobrecarga, explanada anteriormente. Assim, também se deve verificar a restrição temporal imposta por eventuais tarefas de tempo real. Caso não seja possível enviar essa carga devido às restrições temporais, deve-se escolher outra tarefa até que seja possível encontrar um candidato apto a receber a migração. Caso não haja candidatos, uma nova estratégia deve ser adotada: deve-se verificar se é possível distribuir as tarefas desse elemento em subutilização nos outros EP's (sem colocá-los em situação proibitiva) até que não existam mais tarefas nesse elemento subutilizado. A seleção dos candidatos a receber a carga desse processador é feita de maneira semelhante àquela observada nas situações de sobrecarga. O gerenciador global ativa uma *flag* impondo que a cada iteração onde não houver sobrecarga a ser tratada, esse elemento deve ceder suas tarefas até que possa ter sua freqüência de operação reduzida ou até mesmo ser completamente desligado. Apesar de a plataforma de validação não oferecer tal recurso, essa estratégia é principalmente interessante quando se tem disponível mecanismos de DVS (do inglês, *Dynamic Voltage Scaling*) que permitem que essas operações sejam realizadas com o intuito de poupar energia.

Para o tratamento das questões de subutilização para o aspecto de consumo de energia um tratamento semelhante é encontrado. Conforme pôde ser visto pelos trabalhos de Coskun [11] e Carta [35] é importante manter-se um balanceamento do sistema como um todo, evitando tanto situações de sobrecarga quanto de subutilização já que os gradientes de temperatura gerados por essa diferença podem ser tão prejudiciais quanto a situação de desequilíbrio por si só [20].

5.2.3 Comunicação com os monitores locais

Este modelo utiliza troca de mensagens para a comunicação entre o gerenciador global e os monitores locais. Assim, o gerenciador global deve, periodicamente, receber uma mensagem

de cada um dos monitores locais contendo uma listagem η com seu conjunto de tarefas e informações pertinentes. O monitor local, envia essas informações, entra em um estado de espera e somente as envia novamente quando for verificada uma disparidade com relação à última listagem enviada. O gerenciador global define então acerca da necessidade de migração de tarefas e envia para os EP's origem de migração uma listagem τ_t com a tarefa a ser transferida e o EP destino da migração em questão.

A mensagem periódica enviada pelos monitores locais é imprescindível para que o gerenciador global tome suas decisões com base em uma situação não defasada do sistema. Além dessa mensagem, o gerenciador global envia outra mensagem somente no caso da necessidade de uma migração de tarefas.

5.2.4 Fluxo de execução

Através da união das duas estruturas, gerenciador global e monitores locais, pode-se definir um fluxo de interação entre elas. O diagrama da figura 18 apresenta a seqüência de interação entre as estruturas: gerenciador global e monitor local (no diagrama apresenta apenas um monitor local, apesar de no sistema real existir um monitor local para cada EP).

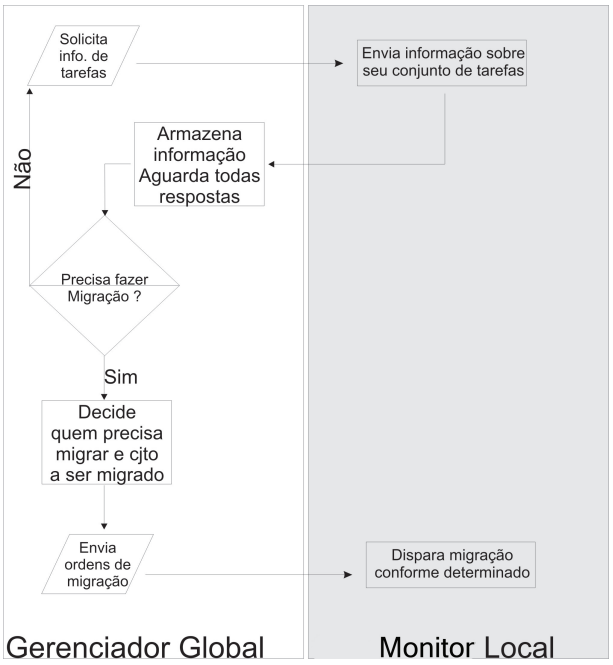

Figura 18 – Diagrama de seqüência mostrando interação entre o gerenciador global e um monitor local

5.2.5 Visão geral do sistema

Uma vez tendo definida a arquitetura na qual o modelo será executado, pode-se mapear o modelo proposto sobre a arquitetura definida. Esse mapeamento, coloca monitores locais nos escalonadores dos diversos EP exceto no EP dedicado para ser o mestre do sistema, que conterá em seu escalonador o gerenciador global. A visão geral do sistema pode ser visualizada na figura 19.

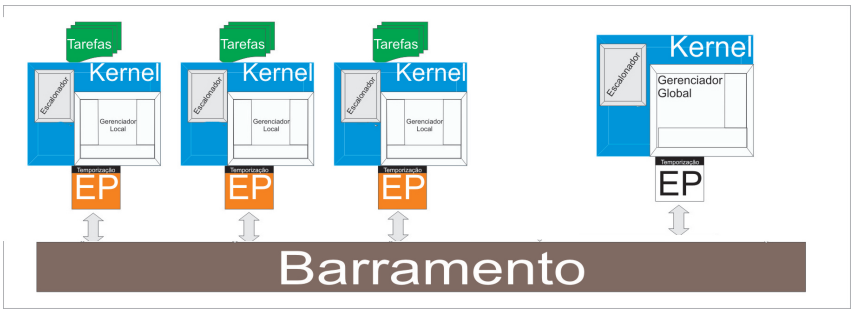

Figura 19 – Visão geral do modelo de balanceamento de carga proposto mapeado na arquitetura descrita

5.2.6 Análise crítica

O modelo de balanceamento de carga proposto visa, através de migrações de tarefas, manter o equilíbrio de um MPSoC de tempo real. O modelo considera o tempo de migração e analisa se *deadlines* de tarefas de tempo real serão perdidos. Além disso, o modelo expressa as condições pelas quais as tarefas que serão migradas devem ser escolhidas bem como define o método pelo qual o candidato a migração será definido.

O objetivo principal do modelo concerne na manutenção do equilíbrio do sistema em termos de carga e de consumo de energia ao longo do seu tempo de vida fazendo com que todas as vantagens de um sistema balanceado, explanadas nos capítulos anteriores, possam ser observadas. Adicionalmente, o modelo proposto possui algumas restrições em termos de informações que deve possuir sobre a arquitetura no qual será executado. Além dessas restrições, tais como conhecimento sobre o consumo energético e o tempo de envio de mensagens do meio de comunicação utilizado, o modelo pode ser adaptado para execução em qualquer plataforma que possua uma camada de *software* sobre a qual o modelo possa ser implementado. Essa flexibilização vai ao encontro com os estudos propostos em [8] e [50] que dissertam justamente sobre

o quão eficaz é a abordagem de se utilizar camadas de *software* que possuam manutenção e desenvolvimento facilitados quando comparada às abordagens puramente de *hardware*.

Em termos de componentes arquiteturais exigidos para a execução do modelo existem os elementos de processamento, interconectados uns aos outros, e conectados a um gerenciador global de carga do sistema, cuja implementação ocorre sobre um EP dedicado do sistema. Esses elementos de processamento possuem monitores locais que têm a capacidade de realizar as funções descritas anteriormente. Além disso, as tarefas que são executadas no sistema devem obedecer ao modelo apresentado para que possam ser levadas em consideração situações de tempo real, por exemplo, além de facilitar o gerenciamento energético do sistema.

É importante destacar que a maneira pela qual a plataforma de execução é implementada possui impacto considerável na avaliação real do sistema, porém, essa flexibilidade permite que diversas soluções de implementação sejam empregadas e testadas e que, de maneira relativamente eficaz, possa se alcançar o balanceamento de carga e energético do sistema através da utilização de um modelo que não se atém a uma tecnologia em específico.

Por fim, questões referentes à flexibilidade ainda têm de ser melhor analisadas. Para a plataforma de validação empregada observou-se uma escalabilidade satisfatória, uma vez que não ocorreram situações de colisão de mensagens no barramento utilizado. Apesar disso, a escalabilidade do modelo em si deve ser analisada de maneira separada, através de sua implementação em plataformas com outros meios de comunicação e maiores quantidades de elementos de processamento.

6 Estudos de caso

Este capítulo apresenta os estudos de caso realizados com o objetivo de validar o modelo e provar sua eficiência. Cada um desses cenários prevê uma situação inicial diferenciada em termos de carga e mostra o que ocorre no sistema ao se empregar o modelo de migração proposto.

A metodologia dos experimentos é basicamente a mesma, estando descrita ao longo do texto. A plataforma, única para todos os cenários apresentados e descrita na Seção 4.2, é composta por quatro processadores Plasma interconectados por um barramento sendo que cada processador executa um *kernel* sobre o qual o modelo proposto foi implementado. É importante destacar que, uma vez que se trata de um MPSoC homogêneo, uma tarefa pode ser executada em qualquer processador do sistema.

6.1 Metodologia de validação

O modelo proposto foi implementado sobre o micro *kernel* descrito na Seção 4.2. Nesse *kernel*, os monitores locais foram implementados como um *driver* do sistema, cuja prioridade foi definida como sendo baixa o suficiente para que não influenciasse negativamente na execução das outras tarefas críticas do sistema porém alta o suficiente para que pudesse ser executado com uma freqüência satisfatória. As primitivas de migração de tarefas disponibilizadas pelo *kernel* foram empregadas para que a efetiva migração de tarefas pudesse ocorrer.

Já o gerenciador global, foi implementado como sendo uma aplicação e executado sem a presença de outras tarefas no elemento de processamento designado para sua execução. Além disso, o algoritmo que o descreve consiste em uma série de passos responsáveis pelas diversas verificações detalhadas na descrição de seu modelo.

O *kernel* utilizado como sendo a base para a implementação do modelo pode ser tanto executado diretamente em um protótipo em *hardware* quanto em um ambiente de simulação do tipo ISS (do inglês, *Instruction-Set Simulator*) sem que sejam necessárias alterações ou modificações na sua estrutura. O único item que é alterado diz respeito à geração de um registro de informações que foi utilizado para analisar os estudos de caso propostos. Esse registro corresponde a escritas em memória que, no simulador, é apenas exibido na tela ao longo da execução.

Já no protótipo em *hardware* essa funcionalidade é desabilitada e somente as informações pertinentes às aplicações são exibidas na tela através da utilização de uma porta serial UART.

6.2 Estudo de caso 1: maioria dos EP's em situação de subutilização

O objetivo do primeiro estudo de caso é verificar o comportamento do modelo perante uma situação onde a maioria ou todos os elementos de processamento do sistema encontram-se em subutilização. Ao se observar o modelo de migração, é proposto que somente se tratrão os casos onde há subutilização caso não exista nenhum outro elemento em sobrecarga. Essa decisão foi tomada principalmente porque são as situações de sobrecarga aquelas cujas conseqüências potencialmente causam mais danos à execução do sistema.

Adicionalmente, o modelo proposto prevê que os casos mais severos de subutilização possam ser resolvidos através da retirada das tarefas desse EP problemático e reposicionando-as nos diversos EP's restantes. Em um sistema que permita que a freqüência do processador possa ser diminuída em tempo de execução através de técnicas como DVS (do inglês, *Dynamic Voltage Scaling*) ou que um EP possa ser até desligado e, posteriormente, se necessário, religado, é possível se aproveitar dessa característica do modelo proposto sendo que uma economia de energia considerável pode ser obtida nesses casos. Trabalhos com os de [51], [52] e [53] ilustram as vantagens que podem ser alcançadas através de técnicas semelhantes. A grande diferença é que, no modelo proposto, esse fato está atrelado à carga do sistema como um todo e tende a potencializar eventuais ganhos. Apesar dos benefícios que possam vir a ser obtidos com tais técnicas é importante destacar que a plataforma de validação utilizada não possui nenhum tipo de implementação nesse sentido e, assim, não se pode comprovar a real eficácia desse método.

Apesar disso, uma *flag* de sinalização em cada processador foi utilizada contendo a informação referente ao seu possível desligamento ou não. A ativação dessa *flag* está diretamente ligada às ordens provenientes do gerenciador global. Assim, cabe a essa estrutura verificar se existe a possibilidade de redistribuir as tarefas do EP que pretende-se efetuar o desligamento e, após ordenar as migrações pertinentes, enviar uma mensagem ordenando o possível desligamento.

No estudo de caso utilizado para ilustrar essa situação, todos os elementos de processamento estão em situação de subutilização. É importante destacar que, por questões de implementação do *kernel*, existe um certo atraso para que a mensuração de carga do sistema possa ser confiável: é necessário que as informações estejam estáveis. Isso ocorre porque em um período inicial as tarefas estão sendo adicionadas e os diversos serviços do *kernel* estão sendo inicializados podendo comprometer o resultado. Para contornar esse problema o gerenciador global

foi implementado de forma a ter suas atividades iniciadas após o período de estabilidade do sistema.

Na figura 20 está ilustrado um gráfico que contém o comportamento dos três elementos de processamento utilizados no trabalho e não se contabiliza, nesse caso, o EP no qual o gerenciador global está alocado. No gráfico, cujo modelo será empregado nos três estudos de caso, o eixo das ordenadas encontram-se os valores de carga sendo que está destacado, quando possível, o intervalo da normalidade adotado enquanto que no eixo das abscissas está o tempo representado em milisegundos.

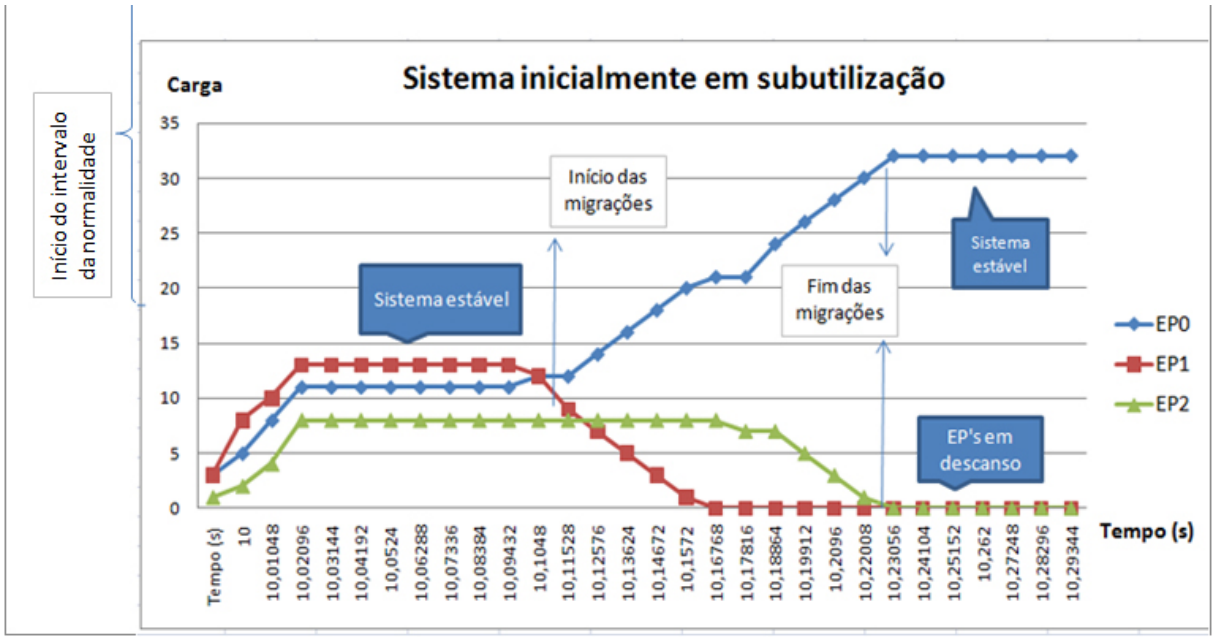

Figura 20 – Estudo de caso 1: maioria dos EP's em situação de subutilização

No gráfico é possível observar o comportamento do sistema ao longo do tempo. Inicialmente, um breve período de adaptação do *kernel* utilizado onde os valores de carga não se encontram estáveis. Após um certo período, o sistema fica estável e o gerenciador global começa a ordenar as migrações pertinentes. O gerenciador verifica que, apesar de existirem três elementos de processamento em situação de subutilização, é possível transferir a carga do elemento que possui menos carga (EP 2) para aquele que possui maior carga (EP 0). Assim, diversas migrações ocorrem até que EP 0 assume a carga de EP 2 que pode então ser desligado ou ter sua freqüência reduzida com o objetivo de, principalmente, reduzir o consumo de energia. Após essa série de iterações, o gerenciador global percebe que mesmo tendo ordenado o desligamento de um EP e elevado o outro para a condição de normalidade ainda existe um elemento de processamento em subutilização. O gerenciador passa a ordenar migrações do EP 1 para o

EP 0 que pode assumir toda a carga de EP 1 e ainda assim não ficar em uma situação proibitiva de sobrecarga. Com a ordem de desligamento do EP 1, o gerenciador verifica que o sistema agora se encontra estável e que migrações não são mais necessárias. Nesse estudo de caso, as cinco tarefas utilizadas possuíam baixa prioridade (entre 80 e 130[1]) e sua funcionalidade era um cáluclo de CRC.

6.3 Estudo de caso 2: maioria dos EP's dentro do intervalo da normalidade

No segundo estudo de caso apresentado objetiva-se validar o comportamento do modelo proposto em uma situação inicialmente mais estável do que aquela apresentada na seção anterior. Neste exemplo, apenas um dos elementos de processamento encontra-se em uma situação considerada proibitiva enquanto que os outros dois EP's estão situados dentro do intervalo da normalidade. Como pôde ser observado ao longo da descrição do modelo proposto, o gerenciador global deve, primeiramente, verificar os elementos de processamento que se encontram em sobrecarga priorizando-os frente aos que estão em subutilização. É importante destacar que, além das questões térmicas levantadas pelos trabalhos de [11], [35] e [20] e que podem acelerar a ocorrência de falhas permanentes no dispositivo, os EP's em sobrecarga excessiva - cujas tarefas estejam ocupando mais do que a capacidade máxima da CPU (mais do que 100% de utilização) - não podem mais oferecer garantias perante o cumprimento de *deadlines* das tarefas de tempo real do sistema. Isso significa que, ao ultrapassar os 100% de utilização de uma CPU, *deadlines* estão sendo perdidos e, com isso, todas as conseqüências negativas associadas a esse fato podem ocorrer.

A figura 21 contém um gráfico que ilustra a situação criada para este estudo de caso, que conta com dois EP's em situação de normalidade (EP 1 e EP 2) enquanto que o EP 0 está extrapolando a utilização máxima de 100% da CPU. Essa CPU não oferece mais garantias quanto ao cumprimento de seus *deadlines* e deve ser atendida com máxima prioridade no sistema. Assim, após se atingir a estabilidade do sistema, conforme detalhado na seção anterior, o gerenciador global entra em ação para tentar resolver a situação problemática do EP 0. A decisão tomada pelo gerenciador global é de, inicialmente, retirar um pouco da carga excessiva encontrada no EP 0 e colocá-la no processador que tenha menor carga: EP 2. Esse EP, por sua vez, também possui limitações com relação à quantidade de carga que pode assumir, sendo que, a partir do

[1] No *kernel* empregado quanto maior o valor absoluto da prioridade de uma tarefa, menor prioridade ela tem, ou seja, ela será executada menos vezes do que uma tarefa de maior prioridade (e de valor absoluto maior).

momento em que migrações do EP 0 fossem colocar o EP 2 em uma situação proibitiva, o gerenciador global analisa a possibilidade de transferir o restante da sobrecarga para o elemento de processamento restante: EP 1. Assim, após terminar as migrações entre os EP's 0 e 2, é a vez do EP 1 receber a carga ainda excedente de EP 0. Ao final da série de migrações os três EP's encontram-se dentro do intervalo da normalidade e os *deadlines* podem, novamente, ser garantidos. Nesse estudo de caso, as sete tarefas utilizadas possuíam prioridade média (entre 10 e 50) e sua funcionalidade era um cáluclo de CRC.

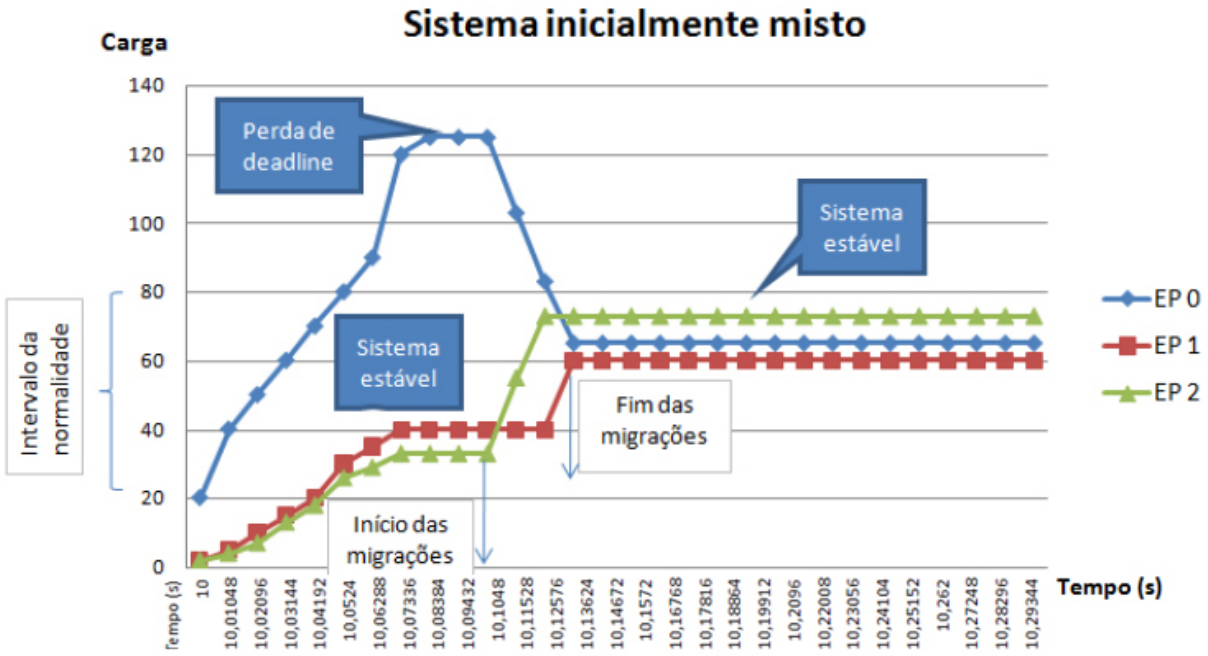

Figura 21 – Estudo de caso 2: maioria dos EP's dentro do intervalo da normalidade

6.4 Estudo de caso 3: maioria dos EP's em situação de sobrecarga

O último estudo de caso proposto visa apresentar o comportamento do modelo de balanceamento em uma situação onde a maioria dos elementos já se encontra em uma situação de sobrecarga. Assim como discutido na seção anterior, é importante frisar que além dos malefícios citados ao longo do trabalho relacionados com a sobrecarga, o fato de um processador estar com uma carga superior à sua capacidade máxima o torna incapaz de garantir quaisquer cumprimento de *deadlines*.

Nesse estudo de caso, cujo gráfico encontra-se ilustrado na figura 22, dois elementos de processamento estão em sobrecarga: EP 0 e o EP 2, sendo que o EP 0 está extrapolando sua capacidade máxima oferencendo riscos ao cumprimento dos *deadlines* existentes em seu conjunto de tarefas. O outro EP, EP 1, está em uma situação normal, porém bastante próxima à faixa inicial da sobrecarga. Nesse caso, o gerenciador global irá atender, novamente o processador que se encontra na situação mais grave de sobrecarga, ou seja, irá atender primeiramente aquele cujos *deadlines* estão sendo perdidos. O primeiro candidato para migraçao é o processador que está com menor carga, nesse caso, o EP 1. O principal problema nesse ponto é que o gerenciador global deve ordenar migrações que não coloquem o EP destino da migração em uma situação proibitiva. Assim, ao verificar que uma quantidade x de carga, que deve ser transferida do EP 0 para o EP 1, coloca o EP destino em uma situação de sobrecarga, o gerenciador global não ordenaria nenhuma migração.

Apesar disso, como o modelo proposto deve levar em consideração as questões temporais existentes nos EP's, ao verificar que mesmo o EP com menor carga não pode atender o EP que está em sobrecarga uma segunda verificação é feita: caso o EP cuja carga está se tentando transferir (no caso, o EP 0) estiver acima de sua capacidade máxima de 100% de utilização de CPU, ou seja, possibilitando a perda de *deadlines* durante sua execução, o gerenciador global - ao invés de verificar se o EP destino ultrapassará a zona da normalidade - certifica-se apenas de que o EP destino não ultrapasse os 100% de utilização máxima a qual ele tem direito. Isso significa que o modelo proposto sacrifica eventuais elementos de processamento em uma situação de normalidade (como é o caso do EP 1) em prol de ajudar o sistema a garantir o cumprimento dos *deadlines*. No sentido de satisfazer essas restrições, as migrações que são ordenadas pelo gerenciador global colocam todos os elementos em sobrecarga (antes de sua execução havia apenas dois nessa situação), porém garante que os *deadlines* existentes sejam cumpridos dentro do algoritmo de escalonamento local que cada EP utiliza. Nesse estudo de caso, as 8 tarefas utilizadas possuíam alta e média prioridade (2 tarefas com prioridade abaixo de 9 e as outras entre entre 10 e 40[2]) e sua funcionalidade era um cáluclo de CRC.

6.5 Análise crítica

Foi demonstrado através de três cenários com configurações diferentes que o modelo proposto de balanceamento de carga consegue atingir seu objetivo principal que é, justamente,

[2]No *kernel* empregado, tarefas com prioridade variando entre 1 e 9 são consideradas *hard real time*; entre 10 e 39, *soft real time*; entre 40 e 127, *best effort* e entre 128 e 255, *low priority*

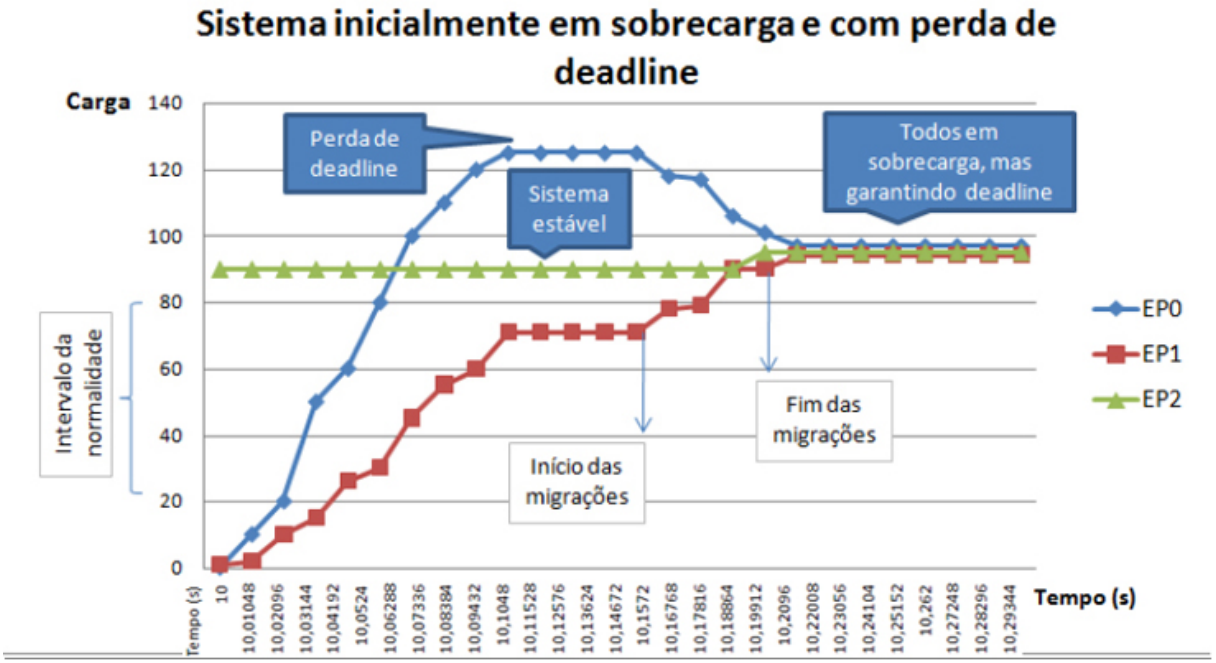

Figura 22 – Estudo de caso 3: maioria dos EP's em situação de sobrecarga

manter o sistema em uma situação de equilíbrio. O modelo proposto comporta-se diferentemente para cada situação encontrada, buscando atingir os seguintes objetivos:

- manter o equilíbrio do sistema;
- evitar ou pelo menos diminuir a perda de *deadlines* de tarefas (aumentar ao máximo a garantia de que *deadlines* serão cumpridos);
- economizar energia quando técnicas pertintentes estiverem disponíveis sobre a plataforma na qual o modelo será implementado.

Os resultados são bastante animadores mas ainda itens como o tempo e, principalmente, o consumo de energia gasto efetivamente para se alcançar as migrações devem ter seus valores medidos. O objetivo de mensurações mais precisas com relação a esses itens visam comprovar que os benefícios atingidos pela utilização de migração de tarefas não sejam penalizados de forma tal que a inviabilize. As migrações de tarefas na plataforma de validação utilizada possuem um tempo que pode ser calculado como segue. Na configuração utilizada da plataforma, o barramento de comunicação gasta em torno de 15000 ciclos de relógio para transmitir 128 *bytes* de dados (correspondente ao tamanho do pacote empregado). Sendo que o barramento opera a velocidade de 25 MHz, leva-se em torno de 0.6 ms para a transmissão de um único pacote. Com essas informações é possível a obtenção do *throughput* do sistema, de 213KB/s. Considerando

que o tamanho do contexto a ser transferido de cada tarefa é de 1700 *bytes* (para o estudo de caso aplicado), são necessários 14 pacotes para sua transmissão completa. Como cada pacote necessita de em torno de 15000 ciclos para ser processado, serão gastos para a transmissão do contexto da tarefa 210000 ciclos de relógio. Verificando o tempo gasto por esses 210000 ciclos a um *clock* de 25MHz, tem-se que o tempo aproximado de migração é de 8,4ms, ou seja, inferior ao valor de um *tick* da plataforma de validação utilizada, que é de 10,48ms. Apesar disso, analisando-se isoladamente o tempo de transmissão de 128 *bytes* de 15000 ciclos pode ser considerado extremamente alto. Essa desvantagem ocorreu principalmente devido à falta de DMA (do inglês, *Direct Memory Access*) na plataforma de validação utilizada bem como à falta de otimização dos *drivers* de comunicação existentes.

Finalmente, é possível afirmar que além do tempo de migração relativamente baixo alcançado pela plataforma de validação empregada, o modelo proposto é capaz de garantir que *deadlines* sejam cumpridos em determinadas situações de sobrecarga excessiva. Além disso, é flexível para ter sua validação realizada em diversas plataformas podendo ser então, confrontado com outros trabalhos salientado-se que é necessário que qualquer plataforma de validação que seja empregada deve oferecer primitivas de comunicação e de migração de tarefas, bem como outras estruturas que viabilizem a implementação do modelo proposto.

7 Conclusão

O estudo proposto apresentou um modelo de balanceamento de carga para MPSoC's considerando tarefas de tempo real. Esse modelo apresenta uma estrutura centralizada, denominada de *gerenciador global* que se comunica com diversas estruturas do tipo *monitor local*.

Os monitores locais são responsáveis por enviar, para o gerenciador global, as informações pertinentes aos elementos de processamento nos quais se encontram e cabe ao gerenciador global determinar se migrações de tarefas são necessárias, qual deve ser o EP origem de uma migração, qual deve sero EP destino dessa migração e que tarefa deve ser enviada. O gerenciador global tenta evitar situações de subutilização e recomenda, em certos casos, que um determinado EP seja desligado ou tenha sua freqüência de operação diminuída através de técnicas como DVS. Além disso, tenta evitar situações de sobrecarga onde tenta retirar EP's dessa situação sem, no entanto, sobrecarregar o EP destino da migração. A excessão ocorre quando o EP origem possui mais de 100% de utilização, caracterizando uma situação propícia para que *deadlines* de tarefas de tempo real sejam perdidos. Nesse caso, o gerenciador global infringe o *intervalo da normalidade* e tenta fazer com que o sistema, apesar de totalmente sobrecarregado, não perca *deadlines*.

No sentido de validar o modelo, três cenários de estudo de caso foram propostos. O primeiro deles, visa apresentar o comportamento do modelo perante uma situação onde a maioria ou todos os EP's encontram-se em subutilização. Foi possível observar que o gerenciador global percebeu que podia reunir diversas tarefas em um único processador recomendando que os outros dois fossem desligados ou tivessem suas freqüências diminuídas.

Já o segundo caso de teste, uma situação onde a maioria dos EP's encontra-se dentro do intervalo da normalidade foi exibida. Nesse caso, existia apenas um EP em situação proibitiva sendo que, de maneira correta, o gerenciador global ordenou que migrações ocorressem para os EP's menos sobrecarregaos fazendo com que todos os EP's adentrassem a faixa da normalidade estabelecida.

O último estudo de caso visou exibir o comportamento do modelo de balanceamento frente a uma situação onde todos os elementos ou a maioria deles estivessem em uma situação de sobrecarga. Nesse exemplo, existia um EP cujos *deadlines* não estavam mais sendo garantidos, uma vez que sua taxa de utilização era superior à sua capacidade máxima de 100%. Nesse

caso, o gerenciador global - após perceber que não poderia transferir nenhuma tarefa para os outros EP's pois os colocaria em uma situação de sobrecarga acima do intervalo da normalidade - verificou que o EP problemático estava com uma utilização acima de 100%. Nesse caso, o gerenciador global opta por não respeitar seu próprio intervalo da normalidade para tentar garantir que os *deadlines* do EP problemático pudessem voltar a ser cumpridos. Para que isso fosse possível, transferiu tarefas para os outros EP's que ultrapassaram a faixa de normalidade sem, porém, ultrapassar sua capacidade máxima de execução.

O modelo proposto se mostrou válido e foi verificado em uma plataforma MPSoC formada por quatro processadores Plasma interconectados por um barramento. A plataforma, desenvolvida inicialmente em [39], pôde ser prototipada em um FPGA além de contar com uma ferramenta ISS de simulação de seu comportamento, também, inicialmente desenvolvida em [39]. A implementação real do modelo proposto ocorreu em um *kernel* cuja execução é possível tanto na plataforma de *hardware* quanto na de simulação sem alterações significativas. Uma ferramenta gráfica de depuração foi utilizada com o objetivo de facilitar a visualização do escalonamento e das migrações ocorridas entre os diversos EP's. Apesar de ter sido validado nessa plataforma, o modelo proposto é independente dela e pode ser validado em outras plataformas, futuramente.

Além disso, uma revisão bibliográfica foi realizada com o intuito de explorar os diversos trabalhos relacionados existentes. Uma breve análise crítica de cada um desses trabalhos com relação ao modelo proposto pôde ser apreciada ao longo de sua descrição. Também realizou-se uma pesquisa no sentido de oferecer ao leitor um embasamento teórico mais aprofundado sobre as questões de balanceamento de carga em sistemas multiprocessados. Apesar de ter propósitos e restrições que os diferenciam, os sistemas multiprocessados tanto embarcados quanto os de propósito geral apresentam a necessidade de um balanceamento de carga, justificando a necessidade de tal embasamento para uma melhor compreensão do trabalho apresentado.

7.1 Trabalhos futuros

É importante destacar que a pesquisa realizada, não obstante, deixa margem para que diversos trabalhos futuros possam ser realizados. Inicialmente, a verificação mais precisa com relação às perdas de *deadlines* pode ser calculada através da inclusão de funcionalidades no *kernel* empregado para que se ofereça ao modelo medidas mais precisas. Ainda no sentido de melhorar o desempenho, pode-se realizar medições relacionadas ao consumo de energia utilizado para que as migrações fossem possível bem como verificar se seu *overhead* não a torna proibitiva.

A implementação de uma plataforma com técnicas de DVS, por exemplo, possibilitaria que o modelo proposto tivesse comprovada a eficácia de sugerir que um processador tenha sua freqüência de operação diminuída ou que seja até mesmo desligado em caso de subutilização severa.

Um parâmetro que não está incluído no modelo proposto diz respeito à freqüência de comunicação entre tarefas. Esse parâmetro pode ter significativo impacto no desempenho atingido pelo sistema, uma vez que o modelo proposto não o leva em consideração sendo que pode acarretar em um aumento significativo na utilização do meio de interconexão dos elementos.

Finalmente, é interessante a avaliação do modelo propoto em outras plataformas. Deve-se verificar quais componentes arquiteturais beneficiam ou prejudicam o desempenho do sistema como um todo, bem como o funcionamento do algoritmo possui qualquer mudança significativa.

Referências

[1] CARRO, L.; WAGNER, F. Sistemas Computacionais Embarcados. In: XXII Jornadas de Atualização em Informática, JAI'03., 2003, Campinas, SP, Brasil. *XXIII Congresso da Sociedade Brasileira de Computação*. 2003.

[2] FARINES, J.; FRAGA, J.; OLIVEIRA, R. *Sistemas de Tempo Real*. 2. ed. São Paulo-SP: Second Escola de Computação, IME-USP, 2000.

[3] JERRAYA, A.; TENHUNEN, H.; WOLF, W. Multiprocessor Systems-on-Chips. *IEEE Computer Society Press - Computer*, Los Alamitos, CA, USA, v. 38, n. 7, p. 36– 40, July 2005.

[4] MELLO, R.; SENGER, L. Modelo de migração baseado na avaliação da carga e tempo de vida de processos em ambientes heterogêneos. *IEEE Latin America Transactions*, Los Alamitos, CA, USA, v. 4, n. 5, p. 370–375, Sept. 2006.

[5] CHANG, H.; OLDHAM, W. Dynamic Task Allocation Models for Large Distributed Computing Systems. *IEEE Transactions on Parallel Distributed Systems*, Piscataway, NJ, USA, v. 6, n. 12, p. 1301–1315, Jan. 1995.

[6] SUEN, T.; WONG, J. Efficient Task Migration Algorithm for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, Los Alamitos, CA, USA, v. 3, n. 4, p. 488–499, July 1992.

[7] BERTOZZI, S.; ACQUAVIVA, A.; BERTOZZI, D.; POGGIALI, A. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In: Conference on Design, Automation and Test in Europe, DATE'06., 2006, Munich, Germany. Washington, DC, USA: IEEE Computer Society, 2006. p. 1–6.

[8] JERRAYA, A.; FRANZA, O.; LEVY, M.; NAKAYA, M.; PAULIN, P.; RAMACHER, U.; TALLA, D.; WOLF, W. Roundtable: Envisioning the Future for Multiprocessor SoC. *IEEE Design & Test of Computers*, Los Alamitos, CA, USA, v. 24, n. 2, p. 174–183, Mar. 2007.

[9] COSKUN, A.; ROSING, T.; GROSS, K. Temperature Management in Multiprocessor SoCs Using Online Learning. In: Annual Conference on Design Automation, DAC'08., 2008, Anaheim, CA, USA. San Diego, California, USA: ACM Press, 2008. p. 890 – 893.

[10] COUNCIL, J. E. D. E. Failure Mechanisms and Models for Semiconductor Devices. www.jedec.org/download/ search/jep122C.pdf. Acesso em 03 de março de 2008.

[11] COSKUN, A.; ROSING, T.; WHISNANT, K. Temperature Aware Task Scheduling in MPSoCs. In: Conference on Design, Automation and Test in Europe, DATE'07., 2007, Nice, France. Washington, DC, USA: IEEE Computer Society, 2007. p. 1–6.

[12] CORES, O. Plasma most MIPS I(TM) opcodes. Disponível em http://www.opencores.org.uk/projects.cgi/web/mips/. Acesso em 01 de dezembro de 2008.

[13] SINHA, P. *Distributed Operating Systems*: Concepts and Design. 1. ed. Wiley-IEEE Press, 1996.

[14] CASAVANT, T.; KUHL, J. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, Los Alamitos, CA, USA, v. 14, n. 2, p. 151–154, Feb. 1988.

[15] ABUBAKAR, A.; RASHID, H.; USMAN, A. Evaluation of Load Balancing Strategies. In: National Conference on Emerging Technologies, NCET'04., 2004. USA: IEEE, ACM, Szabist, 2004. p. 67–70.

[16] DANDAMUDI, S. Performance Impact of Scheduling Discipline on Adaptive Load Sharing in Homogeneous Distributed System. In: Fifteenth International Conference on Distributed Computing Systems, DCS'95., 1995, Washington, DC, USA. Washington, DC, USA: IEEE Computer Society, 1995. p. 484.

[17] THEIMER, M.; LANTZ, K. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, Piscataway, NJ, USA, v. 15, n. 11, p. 1444–1458, Nov. 1989.

[18] TANENBAUM, A. *Distributed operating systems*: . 1. ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[19] BRIãO, E.; BARCELOS, D.; WAGNER, F. Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications. In: Conference on Design, Automation and Test in Europe, DATE'08.

[20] MULAS, F.; PITTAU, M.; BUTTU, M.; CARTA, S.; BENINI, A. A. L.; ATIENZA, D.; DEMICHELI, G. In: .

[21] CHERITON, D. The V distributed system. *Communications of the ACM*, New York, NY, USA, v. 31, n. 3, p. 314–333, Mar. 1988.

[22] MULLENDER, S.; VAN ROSSUM, G.; TANANBAUM, A.; VAN RENESSE, R.; VAN STAVEREN, H. Amoeba: a distributed operating system for the 1990s. *IEEE Computer Society Press - Computer*, Los Alamitos, CA, USA, v. 23, n. 5, p. 44–53, May 1990.

[23] WALKER, B.; MATHEWS, R. Process Migration in AIX's Transparent Computing Facility (TCF). *IEEE Technical Committee on Operating Systems Newsletter*, v. 3, n. 1, p. 5–7, Jan. 1989.

[24] DOUGLIS, F. *Transparent process migration in the Sprite operating system*. 1991. Tese (Doutorado em Física) - University of California at Berkeley, Berkeley, CA, USA, 1991.

[25] POWELL, M.; MILLER, B. Process Migration in DEMOS/MP. In: Association for Computing Machinery Symposium on Operating System Principles, SOSP'83., 1983, New York, NY, USA. New York, NY, USA: ACM Press, 1983. p. 110–119.

[26] ARTSY, Y.; FINKEL, R. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer Society Press - Computer*, Los Alamitos, CA, USA, v. 22, n. 9, p. 47–56, Jan. 1989.

[27] NOLLET, V.; MARESCAUX, T.; AVASARE, P.; MIGNOLET, J.-Y. Centralized Run-Time Resource Management in a Network-on-Chip Containing Reconfigurable Hardware Tiles. In: Conference on Design, Automation and Test in Europe, DATE'05., 2005, Washington, DC, USA. Washington, DC, USA: IEEE Computer Society, 2005. p. 234–239.

[28] 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: .

[29] NOLLET, V.; AVASARE, P.; MIGNOLET, J.; VERKEST, D. Low Cost Task Migration Initiation in a Heterogeneous MPSoC. In: Conference on Design, Automation and Test in Europe, DATE'05., 2005, Washington, DC, USA. New York, NY, USA: IEEE Computer Society, 2005. p. 252–253.

[30] XILINX. PowerPC User Guide. Disponível em http://direct.xilinx.com/bvdocs/userguides/ug011.pdf. Acesso em 18 de março de 2009.

[31] UCLINUX. uClinux – Embedded Linux Microcontroller Project. Disponível em http://www.uclinux.org/. Acesso em 10 de março de 2009.

[32] OZTURK, O.; KANDEMIR, M.; SON, S.; KARAKOY, M. Selective Code/Data Migration for Reducing Communication Energy in Embedded MPSoC Architectures. In: ACM Great Lakes symposium on VLSI, GLSVLSI'06., 2006, Philadelphia, PA, USA. New York, NY, USA: ACM Press, 2006. p. 386–391.

[33] BARCELOS, D.; BRIãO, E.; WAGNER, F. A hybrid memory organization to enhance task migration and dynamic task allocation in NoC-based MPSoCs. In: Annual Conference on Integrated Circuits and Systems Design, SBCCI'07., 2007, Copacabana, Rio de Janeiro. New York, NY, USA: ACM Press, 2007. p. 282–287.

[34] DICK, R.; RHODES, D.; WOLF, W. TGFF: Task Graphs For Free. In: Sixth International Workshop on Hardware/Software Codesign, CODES/CASHE'98., 1998, Seattle, WA. New York, NY, USA: ACM Press, 1998. p. 97–101.

[35] CARTA, S.; ACQUAVIVA, A.; DELVALLE, P.; PITTAU, M.; ATIENZA, D.; RINCON, F.; DEMICHELI, G.; BENINI, L.; MENDIAS, J. Multi-Processor Operating System Emulation Framework with Thermal Feedback for Systems-on-Chip. In: ACM Great Lakes Symposium on VLSI, GLSVLSI'07., 2007, Stresa - Lago Maggiore, Italy. New York, NY, USA: ACM Press, 2007. p. 311–316.

[36] PITTAU, M.; ALIMONDA, A.; CARTA, S.; ACQUAVIVA, A. Impact of Task Migration on Streaming Multimedia for Embedded Multiprocessors: A Quantitative Evaluation. In: Embedded Systems for Real-Time Multimedia, ESTImedia'07., 2007. New York, NY, USA: IEEE/ACM/IFIP, 2007. p. 59–64.

[37] GöTZ, M.; XIE, T.; DITTMANN, F. Dynamic Relocation of Hybrid Tasks: A Complete Design Flow. In: Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC'07., 2007. Editors SASSATELLI, G.; GLESNER, M.; BOBDA, C.; BENOIT, P. Montpellier, França: Univ. Montpellier II.

[38] ZHENG, L. A Task Migration Constrained Energy-Efficient Scheduling Algorithm for Multiprocessor Real-time Systems. In: International Conference on Wireless Communications, Networking and Mobile Computing, WiCom'07., 2007. Los Alamitos, CA, USA: IEEE Computer Society, 2007. p. 3055–3058.

[39] FILHO, S.; AGUIAR, A.; MARCON, C.; HESSEL, F. High-Level Estimation of Execution Time and Energy Consumption for Fast Homogeneous MPSoCs Prototyping. In: Ninteenth IEEE/IFIP International Symposium on Rapid System Prototyping, RSP'08., 2008, Washington, DC, USA. Washington, DC, USA: IEEE Computer Society, 2008. p. 27–33.

[40] SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. Priority inheritance protocols: an approach to real-time synchonization. *IEEE Transactions on Computers*, Washington, DC, USA, v. 39, n. 9, p. 1175–1185, Jan. 1990.

[41] SCHWIEGELSHOHN, U.; YAHYAPOUR, R. Analysis of First-Come-First-Serve Parallel Job Scheduling. In: Ninth Annual ACM-SIAM Symposium on Discrete algorithms, SODA'98., 1998, San Francisco, California, United States. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1998. p. 629–638.

[42] RASMUSSEN, R.; TRICK, M. Round Robin Scheduling - A Survey. Technical report, European Journal of Operational Research, 2006.

[43] FAISSTNAUER, C.; SCHMALSTIEG, D.; PURGATHOFER, W. Priority Round-Robin Scheduling for Very Large Virtual Environments. In: IEEE Virtual Reality 2000 Conference, VR'00., 2000. Washington, DC, USA: IEEE Computer Society, 2000. p. 135–142.

[44] LIU, C.; LAYLAND, J. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, Los Alamitos, CA, USA, v. 20, n. 1, p. 46–61, Jan. 1973.

[45] LEHOCZKY, J.; SHA, L.; DING, Y. The rate monotonic scheduling algorithm: exact characterization and average case behaviour. In: IEEE Real-Time Systems Symposium, RTSS'89., 1989. Los Alamitos, CA, USA: IEEE Computer Society, 1989. p. 166–171.

[46] LEUNG, J.; WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, v. 2, n. 4, p. 237–250, 1982.

[47] HESSELINK, W.; TOL, R. Formal feasibility conditions for earliest deadline first scheduling. Technical report, Department of Computer Science of Rijksuniversiteit Groningen, 1994.

[48] ANDREWS, M. Probabilistic End-to-End Delay Bounds for Earliest Deadline First Scheduling. In: Ninteenth Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM'00., 2000, Tel Aviv, Israel. New York, NY, USA: IEEE Computer Society, 2000. p. 603–612 vol.2.

[49] XILINX. Virtex-II Pro Datasheet. Disponível por www em: <http://www.xilinx.com/publications/products/v2pro/ds_pdf/ds083.htm>. Acesso em 15 de dezembro de 2008.

[50] HARITAN, E.; YAGI, H.; WOLF, W.; HATTORI, T.; PAULIN, P.; NOHL, A.; WINGARD, D.; MULLER, M. Multicore design is the challenge! What is the solution? In: Annual Design Automation Conference, DAC'08., 2008. New York, NY, USA: ACM Press, 2008. p. 128–130.

[51] CAO, Z.; FOO, B.; LEIA, H.; SCHAAR, M. Optimality and improvement of dynamic voltage scaling algorithms for multimedia applications. In: Annual Conference on Design Automation, DAC'08., 2008, San Francisco, CA, USA. New York, NY, USA: ACM Press, 2008. p. 179–184.

[52] CHANG, P.; WU, I.; SHANN, J.; CHUNG, C. ETAHM: An energy-aware task allocation algorithm for heterogeneous multiprocessor. In: Annual Conference on Design Automation, DAC'08., 2008, San Francisco, CA, USA. New York, NY, USA: ACM Press, 2008. p. 776–779.

[53] QU, G. What is the limit of energy saving by dynamic voltage scaling? In: International Conference on Computer Aided Design, ICCAD'01., 2001, San Jose, California. Piscataway, NJ, USA: IEEE Press, 2001. p. 560–563.