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

DANIEL BARCELOS

Modelo de Migração de Tarefas para MPSoCs baseados em Redes-em-chip

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

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, março de 2008.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Barcelos, Daniel

Modelo de Migração de Tarefas para MPSoCs baseados em Redes-em-chip / Daniel Barcelos – Porto Alegre: Programa de Pós-Graduação em Computação, 2008.

92 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2008. Orientador: Flávio Rech Wagner.

1. Migração de Tarefas. 2. MPSoC 3. NoC. I. Wagner, Flávio Rech. II. Título.

AGRADECIMENTOS

Agradeço aos meus pais pelo apoio, à Bárbara pelo carinho, aos meus irmãos pela camaradagem; vocês todos significam muito para mim.

Agradeço aos amigos-colegas Mateus (*mbrutzig*), Kalile (*rkaandrade*), Girão (*ggbsilva)*, Paulista (*lgolob*), Paulo (*prmmeirelles*), Rodrigo (*rbmotta*) e Dalton (*dmcolombo*) pelas conversas e discussões, nem sempre acadêmicas é verdade. Aos doutorandos Eduardo Wenzel Brião e Elias Teodoro da Silva Jr. pelo exemplo na busca incessante do conhecimento.

Agradeço ao meu orientador, Flávio Rech Wagner, entre outras coisas, por mostrar os caminhos.

Finalmente, agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) por ter financiado esta pesquisa.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACK	*Acknowledgement*
API	*Application Programming Interface*
ARM	*Advanced RISC Machines*
ASIP	*Application Specific Instruction Set Processor*
BOP	*Begin of Packet*
CACO-PS	*Cycle-Accurate Configurable Power Simulator*
CAN	*Controller Area Network*
CISC	*Complex Instruction Set Computer*
CPI	*Cycles per Instruction*
CPU	*Central Processor Unit*
DMA	*Direct Memory Access*
DPM	*Dynamic Power Management*
DSP	*Digital Signal Processing*
DVS	*Dynamic Voltage Scaling*
EDF	*Earliest Deadline First*
EFT	*Energy Freezing Time*
EOP	*End of Packet*
FM	Freqüência Modulada
FPGA	*Field Programmable Gate Array*
IP	*Intellectual Property*
ISA	*Instruction Set Architecture*
JVM	*Java Virtual Machine*
LSE	Laboratório de Sistemas Embarcados
LSF	*Load Sharing Facility*
MMU	*Memory Management Unit*
MOSIX	*Multi Computer Operating System for Unix*
MP3	*MPEG Layer-3 Audio*

MPEG	*Moving Picture Experts Group*
MPSoC	*Multi-processor System on Chip*
NoC	*Network on Chip*
PC	*Program Counter*
PDA	*Personal Digital Assistant*
PE	*Processor Element*
PIC	*Position Independent Code*
PM	*Power Management*
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
ROM	*Read Only Memory*
RTC	*Real-Time Clock*
RTL	*Register Transfer Level*
RTSJ	*Real-Time Specification for Java*
SASHIMI	*System as Software and Hardware in Microcontrollers*
SIA	*Semiconductor Industry Association*
SO	Sistema Operacional
SoC	*System on Chip*
SP	*Stack Pointer*
TCP/IP	*Transmission Control Protocol/Internet Protocol*
TLM	*Transaction Level Model*
UFRGS	Universidade Federal do Rio Grande do Sul
UML	*Unified Modeling Language*
VAL	*New Value*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit Hardware*
VLIW	*Very Long Instruction Word*
VNI	*Virtual Network Interface*
YAFJS	*yet Another femtoJava Simulator*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Em relação a sistemas multiprocessados integrados em uma única pastilha (MPSoC), tanto a alocação dinâmica quanto a migração de tarefas são áreas de pesquisa recentes e abertas. Este artigo propõe uma organização de memória híbrida para sistemas com comunicação baseados em redes-em-chip, como maneira de minimizar a energia gasta durante a transferência de código decorrente de uma alocação ou migração de tarefa. É também introduzido um novo mecanismo de migração de tarefas, que, por sua vez, pode utilizar *check-pointing* ou outra técnica mais transparente.

O aumento do uso de sistemas multiprocessados na computação embarcada torna importante a avaliação de diferentes organizações de memória. Enquanto memórias distribuídas proporcionam acessos mais rápidos, memórias compartilhadas tornam possível o compartilhamento de dados sem a interferência dos processadores. Nos experimentos realizados, foi focada a redução da energia gasta na comunicação em um contexto onde uma migração de tarefas ou uma alocação dinâmica fosse necessária. Os resultados indicam que, considerando a migração do código, a solução proposta apresenta melhor eficiência do que soluções unicamente distribuídas ou compartilhadas. Foi também verificado que, em alguns casos, a estratégia híbrida reduz os tempos de migração. Na solução apresentada, o código pode ser transferido do nó onde a tarefa era originalmente executada ou de uma memória posicionada no centro da rede. A escolha entre as duas opções é feita em tempo de execução de uma maneira intuitiva, sendo a escolha baseada na distância entre os nós envolvidos na transferência. Os resultados indicam que a organização proposta reduz a energia de transferência de código em 24% e 10% em média, se comparada, respectivamente, a soluções utilizando somente memória global ou distribuída.

O modelo de migração de tarefas proposto é baseado na linguagem Java e na comunicação por troca de mensagens. Todo seu desenvolvimento se deu em *software*, não requerendo nenhuma modificação no sistema. O custo energético da migração foi então avaliado. Entende-se por custo energético a energia gasta nos processadores para envio e recebimento das mensagens e na estrutura de comunicação, uma *rede-em-chip*. Trabalhos já existentes não consideram o custo de migração, comparando apenas o arranjo inicial e final das tarefas no sistema. Este trabalho, entretanto, avalia todo o processo de migração. Através de experimentos, é estimado o tempo mínimo de execução da plataforma, como função do tamanho da tarefa e da distância entre os nós da rede, necessário para amortizar a energia gasta no processo de migração, considerando que os processadores utilizam a técnica de DVS para reduzir o consumo de acordo com suas cargas de processamento.

Palavras-Chave: migração de tarefas, sistemas embarcados, *redes-em-chip*, sistemas multiprocessados, sistemas distribuídos.

Task Migration Model for NoC-based MPSoCs

ABSTRACT

Regarding embedded Multi-processor Systems-on-Chip (MPSoCs), dynamic task allocation and task migration are still open research areas. This work proposes a hybrid memory organization for NoC-based systems as the way to minimize the energy spent during the code transfer when task migration or dynamic task allocation needs to be performed. It is also introduced a new flexible task migration mechanism, which can use check-pointing or a more transparent technique.

The increasing use of multi-processor architectures in embedded computing makes it important to evaluate different options for memory organization. While distributed memory allows faster accesses, a global memory makes possible the sharing of data without processor interference. In the experiments, it is targeted the communication energy reduction in a context where task migration or dynamic task allocation is required. Results indicate that the proposed hybrid memory organization presents better efficiency than distributed- or global-only organizations regarding code migration. It is also noticed that, in some cases, the hybrid strategy reduces the task migration times.

In the hybrid approach, the code can be transferred from the node where the task was originally running or from a memory positioned at the center of the system. The choice between the two options is done at runtime in a very intuitive way, based on the distance between the nodes involved on the transfer. Results are very encouraging and indicate that the proposed hybrid organization reduces the code transfer energy by 24% and 10% on average, as compared to global- and distributed-only memory organizations, respectively.

The proposed migration model is based on the Java language and on message passing communication method. It is mainly *software*-based, and does not require any system modification. The energy cost of the migration process is then evaluated, i.e., the energy spent on the sending and receiving cores and on the communication structure, a wormhole-based Network-on-Chip (NoC).

Previous works have compared system figures before and after task migration, while this study evaluates the whole migration process. Finally, it is derived the minimum execution time of the embedded system, as a function of the task size and of the distance between the cores on the NoC, that is required to amortize the energy spent on the migration process, considering that processors use Dynamic Voltage Scaling to reduce power consumption according to their current workloads.

Keywords: Task migration, embedded systems, network-on-chip, multi-processor systems, distributed systems.

1 INTRODUÇÃO

A cada dia temos mais sistemas computacionais à nossa volta. Não apenas computadores pessoais, mas diferentes dispositivos que contam com processadores e realizam as mais diversas tarefas. Atualmente pode-se encontrar processadores em assistentes digitais portáteis (PDA, do inglês, *Portable Digital Assistent*), telefones celulares, tocadores de música (MP3 Players), entre outros. Também as novas gerações de fornos de microondas, geladeiras e máquinas de lavar já contam, atualmente, com processadores no seu interior. A todos os dispositivos que possuem alguma forma de processador e que não são necessariamente computadores dá-se a denominação de *sistemas embarcados*. É difícil pensar hoje como seria a vida sem tais sistemas, pois, além de todos os equipamentos mencionados, eles se fazem presentes controlando funções do nosso automóvel, viabilizando o acesso ao transporte coletivo, através de sistemas de bilhetagem eletrônica, ou contabilizando as horas trabalhadas em sistemas ponto.

Contudo, seja por sofisticação ou necessidade, o mercado não pára de oferecer novos produtos diariamente. Alguns surgem como soluções necessárias, outros vêm resolver problemas que ainda não existem e muitos, ainda, não incorporam nenhuma nova funcionalidade apenas agregam funções, antes presentes em diferentes dispositivos, ou realizam as mesmas funções com um apelo mercadológico voltado ao *design* ou à marca do produto.

Com toda essa inovação, por meio de pesquisas, surgem novos desafios, quase que também diariamente. Atualmente, conforme mostrado na Figura 1.1, os desenvolvedores não são capazes de explorar toda a capacidade de integração disponível. Muitos sistemas embarcados são alimentados por baterias e o ritmo de evolução das mesmas é bastante baixo, como se pode também depreender da mesma figura. Assim, diferentes alternativas tiveram que ser desenvolvidas para maximizar a produtividade e a duração da bateria nos dispositivos portáteis. Tais alternativas passam por diferentes soluções para lidar com a crescente complexidade do *software* e para aumentar a capacidade de exploração das possibilidades de *hardware* sem aumentar demasiadamente o consumo dos dispositivos.

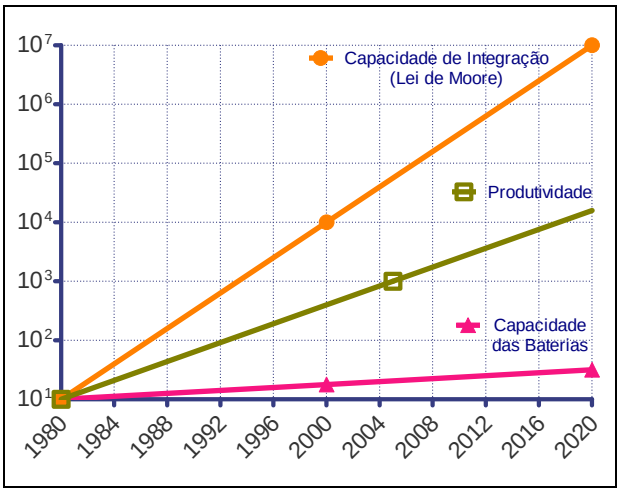

Figura 1.1: Hiato (*gap*) de produtividade (RABAEY, 2000)

No que diz respeito ao *software*, novas linguagens e metodologias foram desenvolvidas, de maneira a dar suporte ao desenvolvimento de aplicações cada vez mais complexas. Uma dessas linguagens é a linguagem Java, desenvolvida pela Sun. Totalmente orientada a objetos, a linguagem Java facilita o desenvolvimento em relação às linguagens C/C++. A facilidade de desenvolvimento, baseado em funções e bibliotecas pré-prontas, aliada a sua portabilidade, fizeram com que ela rapidamente ganhasse popularidade e se tornasse uma das linguagens mais usadas atualmente (Figura 1.2).

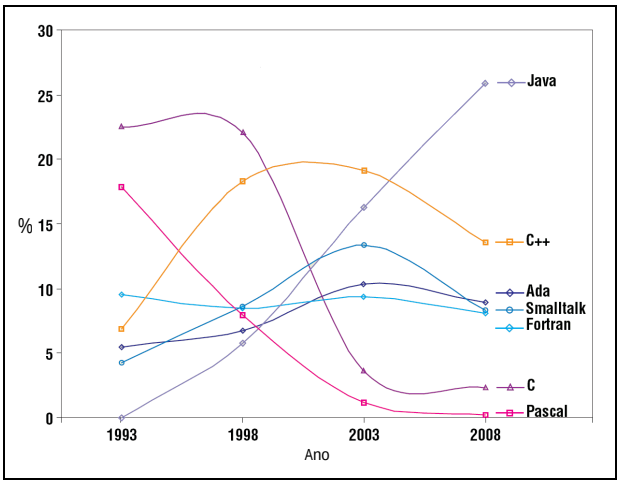

Figura 1.2: Curva de evolução do uso de linguagens de programação (CHEN, 2005)

Inicialmente, essa popularidade encontrou resistência entre os desenvolvedores de *software* do domínio dos sistemas embarcados, pois a linguagem Java era considerada lenta e muito pesada. Com o tempo e adaptações na linguagem, essa barreira vem sendo quebrada, muito em função do grande número de desenvolvedores presentes no mercado, o que diminui os custos de projeto, e do menor tempo de projeto possibilitado pela linguagem.

A linguagem Java está presente hoje em grande parte dos celulares – estimativas mencionam o percentual de 80% (LAWTON, 2002) – e *palmtops* do mercado, assim como em dispositivos de maior porte, como roteadores e multiplexadores. Em todos eles, porém, a linguagem Java ainda realiza funções secundárias, como jogos nos primeiros e gerência remota nos segundos.

Existem diferentes maneiras de se executar código Java. Uma delas é através de uma máquina virtual, que emula uma máquina Java em *software* em outra plataforma. É dessa maneira que costumeiramente se utiliza Java nos computadores de propósitos gerais. Outra forma é através do uso de um co-processador associado ao núcleo principal. Nesse caso, na ocasião da execução de um programa Java, ele é executado no co-processador e apenas as entradas e saídas do programa são intermediadas pelo processador principal. O referido co-processador pode implementar todos os *bytecodes* Java ou apenas um subconjunto deles, deixando os demais para serem realizados por rotinas de *software*, como no caso anterior. De qualquer maneira, executando o todo ou parte das instruções, esse método possibilita a aceleração da computação de programas escritos na linguagem Java. Há ainda a possibilidade de utilizar apenas um processador Java. Nessa solução apenas, programas Java são executados no processador e todo o processo de desenvolvimento acontece nessa linguagem.

Java também facilita o uso de UML (*Unified Modeling Language*), seja para modelagem do sistema ou na geração automática de código (OLIVEIRA, 2007). Além disso, existem pesquisas em curso visando gerar código Java validado via verificação formal através de outra linguagem, como Alloy (SPECHT, 2007). Com isso seria possível diminuir ainda mais o tempo de projeto, reduzindo os gastos com testes.

No que tange ao *hardware*, sistemas embarcados estão ficando cada vez mais complexos, possuindo diversas interfaces de entrada e saída, diferentes níveis de memória e até mesmo vários processadores ou elementos de processamento. A cada dia é possível colocar mais transistores em uma única pastilha e, com o aumento dessa capacidade de integração, que em breve chegará a um bilhão de transistores em um único *chip* (SIA, 2006), tem-se optado por colocar todos os componentes de um dispositivo em um único chip. A esses circuitos dá-se o nome de *SoC* (do inglês, *System-on-Chip*) (TAKAHASHI, 2001).

A maioria das metodologias de desenvolvimento de *hardware*, entre elas Bergamaschi (2000), afirma que o reuso de componentes possibilita lidar com a referida complexidade, além de reduzir os custos e o *time-to-market*[1]. Assim, tem-se então utilizado blocos pré-fabricados e verificados no desenvolvimento de sistemas maiores. Tais blocos denominam-se IPs (do inglês, *Intellectual Property*). Núcleos IP podem ser reutilizados de projetos anteriores ou comprados de empresas terceiras especializadas no desenvolvimento de um determinado tipo de componente. Finalmente, existem núcleos IP para executar as mais variadas tarefas de um sistema computacional, e eles podem corresponder a processadores, memórias, *hardware* dedicados, interfaces de protocolos de comunicação, entre outros.

Para conectar todos esses dispositivos presentes em um único *chip* a solução de barramento é atualmente a mais utilizada. Barramentos, entretanto, oferecem suporte limitado no que diz respeito à escalabilidade, reusabilidade, paralelismo e consumo de energia. No intuito de resolver tais problemas, diversas soluções foram propostas, entre elas as *redes-em-chip* (NoC, do inglês, *network-on-chip*) (BENINI, 2002).

Redes-em-chip, por sua vez, são fortemente inspiradas na aplicação de redes de computadores convencionais. Salvo a escala e a complexidade, conceitos aplicados

[1] *Time-to-market*: tempo que um produto leva para chegar ao mercado desde a fase de definição de requisitos.

durante décadas no âmbito das redes tradicionais vêm sendo trazidos para dentro dos circuitos integrados.

Finalmente, diferentes técnicas foram propostas e desenvolvidas para diminuir o consumo dos dispositivos portáteis sem afetar suas funcionalidades. Contudo, é verdade que a evolução da capacidade de integração diminuiu por si só o consumo dos transistores. Quanto menor a tecnologia, menor o consumo por transistor, por isso durante um bom tempo a questão da duração da bateria desses dispositivos foi relegada a um segundo plano.

À medida que novas funcionalidades foram agregadas aos sistemas embarcados, o consumo tornou-se um gargalo do desenvolvimento. Uma das primeiras soluções encontradas foi migrar funcionalidades antes existentes em *software* para o *hardware*. Dessa forma gasta-se menos energia ou melhora-se o desempenho, sendo que, por vezes, ocorrem os dois. Com o advento das tecnologias submicrônicas, o consumo estático dos circuitos aumentou consideravelmente, tornando onerosa a simples adição de mais *hardware* no circuito. Isso obrigou projetistas e pesquisadores a serem mais criativos a fim de maximizar a duração das baterias.

Uma vez identificado o problema, diferentes técnicas de gerenciamento de energia foram desenvolvidas. Tais técnicas podem ser implementadas em *software*, *hardware* ou ainda serem uma combinação de ambos. O desligamento de partes ociosas do circuito ou a diminuição da tensão de operação são duas técnicas que combinam *hardware* e *software* com o objetivo de diminuir o consumo.

Uma técnica que vem ganhando destaque nesse intuito é a migração de tarefas. Trata-se, resumidamente, de transferir a execução de uma tarefa de um processador para outro. Oriunda da área dos sistemas distribuídos, onde é usada principalmente com foco em desempenho e tolerância a falhas, o uso da funcionalidade de migração de tarefas em sistemas embarcados, com algumas modificações em relação aos sistemas distribuídos em função das restrições impostas, justifica-se porque esses dispositivos estão se aproximando dos computadores de propósitos gerais, sendo capazes de executar uma grande gama de aplicações, por vezes de diferentes domínios.

Um exemplo particularmente ilustrativo desse fato são, novamente, os celulares atuais, que possuem agenda, jogos e realizam um grande processamento multimídia. Logo, os projetistas não sabem previamente quais aplicações serão executadas no sistema, quando as mesmas serão disparadas e em qual combinação elas executarão concomitantemente. A migração de tarefas permite gerenciar essa variação dinâmica da carga de processamento do sistema, levando em conta alguma métrica ou um conjunto delas, concentrando as aplicações em um único nó da rede ou distribuindo-as de acordo com a necessidade. Embora a carga possa variar de maneira aleatória, mesmo sem conhecimento prévio das aplicações, é possível utilizar heurísticas que melhorem o desempenho do sistema, corrigindo a degradação do mesmo, diminuindo a comunicação no sistema, alocando tarefas que se comunicam muito entre si em um mesmo nó, entre outras. Em se tratando de sistemas distribuídos, diversas técnicas vêm sendo estudadas e aplicadas na migração de tarefas. Sistemas do tipo *cluster* e *grid* são exemplos de aplicações onde se faz necessário um gerenciamento das tarefas distribuídas ao longo do tempo e dos nós de processamento presentes nas redes. No entanto, ainda não há solução clara e definitiva para o problema da migração de tarefas em sistemas multiprocessados de uma única pastilha (MPSoCs).

A maioria dos trabalhos realizados abstrai a infra-estrutura necessária e se preocupa apenas em como distribuir as tarefas pela NoC, reduzindo o problema à alocação e ao escalonamento dos processos. Atualmente existem poucos estudos publicados no que diz respeito à melhor organização do sistema (topologia da rede, distribuição da memória – se local ou compartilhada, etc.) em casos onde será aplicada a técnica de migração de tarefas *intrachip*.

O presente estudo visa avaliar o custo energético da migração de tarefas em si e não os efeitos proporcionados pela aplicação de uma heurística específica da distribuição das tarefas no sistema. Para isso foi proposto um novo modelo de migração de tarefas, sendo que soluções já publicadas para o problema também são analisadas e discutidas. O modelo proposto se baseia no uso de processadores Java, da família femtoJava (ITO, 2001) (BECK, 2004), que são processadores de pilha, ou seja, ao invés de registradores, utilizam uma pilha de dados para executar as operações. Algumas características da linguagem Java são utilizadas como forma de obter melhores resultados, enquanto outras tiveram que ser contornadas para possibilitar a implementação do mecanismo.

O compromisso entre desempenho e energia consumida será levado sempre em consideração nas análises, uma vez que se trata da adoção de determinados métodos em sistemas embarcados, que, na sua extensa maioria, possuem a referida restrição de consumo.

Uma análise dessa natureza faz-se necessária porque mesmo requisitos desejáveis a um mecanismo de migração de tarefas em sistemas distribuídos ou a simples possibilidade de se transferir uma tarefa de processador podem vir a ser extremamente caros em plataformas embarcadas e, nesse caso, sua adoção torna-se proibitiva, inviabilizando a aplicação de heurísticas de alocação.

Este trabalho, especificamente, levará em consideração uma plataforma multiprocessada homogênea, na qual os processadores são interconectados por uma *rede-em-chip*. A adoção de tal plataforma deve-se principalmente ao fato de que, dessa maneira, aproxima-se a topologia do sistema estudado à daqueles em que se aplica a migração de tarefas originalmente, no domínio dos sistemas distribuídos.

1.1 Contribuições da Dissertação

A principal contribuição dessa dissertação é a avaliação em termos de energia de todo o processo de migração de tarefas. Há um detalhamento dos componentes responsáveis pelo gasto energético da operação. É ainda mostrada uma maneira de compensar o referido custo através do uso de técnicas de gerenciamento de energia.

Para atingir esse objetivo, diversas etapas foram cumpridas, sendo que essas correspondem a contribuições menores da presente pesquisa. Dentre elas, destaca-se:

- A proposição de um modelo de migração que tornou possível não só a avaliação de energia, mas também de desempenho.
- A implementação de um simulador que possibilitou a análise do modelo desenvolvido.
- O projeto, na linguagem SystemC, do processador femtoJava multiciclo.
- Uma versão comportamental do mecanismo de interrupção foi adicionada ao modelo existente de femtoJava *pipeline*.

- A análise de diferentes organizações de memória quando envolvidas no processo de migração de tarefas (BARCELOS, 2007).
- Um estudo sobre as relações de compromisso existentes na transposição do processo de migração de tarefas de sistemas distribuídos para sistemas embarcados.

1.2 Conclusões da Dissertação

Ao fim do trabalho, conclui-se que a migração de tarefas se apresenta como uma alternativa viável para a obtenção de melhor desempenho e eficiência energética em sistemas embarcados. A compensação em termos de desempenho, tendo em vista o custo da migração de tarefas, efetiva-se mais rapidamente do que a energética. Logo, para a obtenção de ganhos de energia é requerido que as tarefas depois de migradas executem por mais tempo e, com isso, que as migrações sejam mais esporádicas.

Isso se deve ao fato de que a distância entre os nós da rede não afeta, significativamente, o tempo de migração, enquanto que possui estratégica importância em termos de energia. Basicamente, o custo energético da migração é função do tamanho da tarefa migrada e da distância entre os nós envolvidos no processo. Com aumento da distância entre os nós envolvidos no processo, é possível que a consumo de energia causado pela migração, seja, majoritariamente, em função da comunicação, uma vez que o custo computacional do processo não depende da distância entre os nós.

Experimentos revelam, também, que a organização de memória adotada no sistema afeta o custo da migração. Para a plataforma estudada, verificou-se que o uso simultâneo tanto de memórias locais privativas, quanto globais compartilhadas, proporciona migrações menos onerosas em termos de consumo de energia.

1.3 Organização do Texto

O trabalho está organizado de maneira a facilitar a compreensão do leitor. O Capítulo 2 discute aspectos e técnicas utilizadas na migração de tarefas em sistemas distribuídos, apresenta o tema ao leitor e realiza uma breve revisão de conceitos.

O Capítulo 3 apresenta requisitos não funcionais desejáveis a um mecanismo de migração de tarefas. No Capítulo 4 é realizada uma discussão em cima de propostas já realizadas de mecanismos de migração de tarefas para sistemas embarcados. São levantadas suas virtudes e limitações, além de ser realizada uma análise crítica sobre os experimentos apresentados.

O Capítulo 5 descreve o modelo proposto em função da plataforma utilizada, enquanto o Capítulo 6 realiza sua validação através de experimentos. Finalmente, no Capítulo 7, são apresentadas as conclusões do presente estudo.

2 MIGRAÇÃO DE TAREFAS

A principal motivação para a utilização de migração de processos em sistemas distribuídos é a natureza dinâmica dos mesmos. Cada nó processador possui uma carga não conhecida *a priori* e essa pode variar de forma não predizível ao longo do tempo. Assim sendo, a alocação de um processo apenas no momento de sua criação não atende completamente os requisitos esperados pelos usuários. Uma estação livre no momento da criação de um processo pode, por exemplo, logo em seguida, tornar-se sobrecarregada devido ao *logon* local de um usuário na máquina.

A migração de tarefas na área de sistemas distribuídos é tema de pesquisa desde 1960. Já no que diz respeito à área de sistemas embarcados, a migração de tarefas se tornou objeto de estudo apenas com o advento das NoCs e dos *chips* multiprocessados.

O processo de migração de tarefas pode realizar-se de duas diferentes formas: preemptiva e não-preemptiva (SINHA, 1997). Migrações preemptivas ocorrem em tempo de execução, após o início da computação dos processos. Por algum motivo o sistema decide migrar a tarefa; é necessário, então, salvar o contexto da mesma e transferi-lo para o processador de destino.

Migrações não-preemptivas são aquelas que ocorrem antes do início do processamento de uma tarefa, operando da seguinte maneira: o nó mestre, responsável pelo sistema, precisa iniciar um novo processo; ele decide, seguindo alguma política, qual nó executará tal tarefa. A seguir, se for o caso, o código da aplicação é transferido pela rede e sua execução iniciada no nó escravo.

Alguns autores, como Coulouris (2005), não consideram isso um tipo de migração, reduzindo o problema a um caso de alocação de tarefas em sistemas multiprocessados. Já Tanenbaum (2002) denomina mobilidade forte e mobilidade fraca as formas de migração preemptiva e não-preemptiva, respectivamente.

Migrações não-preemptivas são mais simples, já que não envolvem transferência de dados, apenas de código, quando for o caso. O que existe é a troca de informações sobre o processo por parte dos módulos do sistema operacional existente em cada um dos nós envolvidos na operação.

Para realizar uma migração de maneira preemptiva, tanto o código, quanto os dados (variáveis e toda informação referente à execução da aplicação) e os recursos (arquivos, *devices*) necessitam ser migrados ou ter sua coerência mantida durante o processo. Por conta dessa necessidade, migrações preemptivas impõem um grande *overhead* ao sistema e precisam ser cuidadosamente realizadas a fim de lhe trazer vantagens.

Embora alguns autores diferenciem *migração de processos* de *migração de tarefas* em função do nível de abstração dos mesmos (PAINDAVEIVE, 1996), no presente estudo os termos serão tratados como sinônimos e se referirão a migração que se dá na camada que provê a funcionalidade (se sistema operacional, então processos; se máquina virtual Java, então tarefas, por exemplo). A Seção 2.2 abordará os diferentes níveis de suporte à migração. Como o trabalho privilegia aspectos conceituais, essa simplificação não implicará no comprometimento da compreensão do tema por parte do leitor.

A migração de tarefas envolve diversos aspectos funcionais, os quais são discutidos ao longo deste capítulo. As diferentes combinações possíveis desses aspectos resultam em modelos com diferentes propriedades não funcionais como desempenho, eficiência energética, etc. Aspectos não funcionais são ortogonais à parte algorítmica do sistema e são tema do Capítulo 3 desse trabalho.

2.1 Vantagens

A funcionalidade de migração de tarefas pode ser implementada em sistemas distribuídos para prover diversas vantagens aos usuários. A maioria das vantagens resulta em um ganho de desempenho, mas também pode ser obtido um sistema mais robusto com a sua utilização. A redução do tempo médio de resposta dos programas, o aumento da vazão computacional aparente do sistema, uma utilização mais efetiva dos recursos presentes, a redução do tráfego de comunicação pela rede e o aumento da confiabilidade do sistema, entre outros, são exemplos de benefícios acarretados pelo uso da técnica. Entre as vantagens mencionadas, pelo menos duas não são intuitivas e serão explanadas a seguir: *diminuição do tráfego da rede* e *aumento da confiabilidade do sistema*.

A diminuição do tráfego de comunicação da rede se dá ao agrupar tarefas que se comunicam muito seguidamente ou com grande volume de dados em um mesmo processador ou em processadores próximos. Para isso é necessário conhecer a natureza das aplicações, ou através de uma caracterização prévia ou através do monitoramento do tráfego da rede, e, no momento da alocação ou de uma realocação, colocar tais próximas entre si. Essa é uma função do gerenciador de carga (ver Seção 2.6).

Para o aumento da confiabilidade do sistema parte-se do pressuposto de que a estrutura de nós do mesmo é conhecida. Dessa maneira pode-se posicionar tarefas críticas em nós sabidamente seguros. Uma outra maneira de aumentar a confiabilidade é monitorar a degradação dos nós. Um nó que tem sua memória mais utilizada a cada instante pode estar com algum mau funcionamento e pode precisar ser reiniciado em breve. Assim, no ato dessa constatação, o sistema poderia migrar as tarefas mais importantes desse nó para outro, garantindo que o sistema seja minimamente prejudicado.

2.2 Requisitos

Para permitir a implementação da funcionalidade de migração de tarefas, um sistema deve prover a possibilidade de transferir o contexto de uma aplicação entre os nós da rede. Como contexto, entende-se desde os valores dos registradores internos, o espaço de endereçamento da aplicação (dados e instruções) e os recursos utilizados (arquivos, *devices*, conexões).

Ainda é necessário que o processo e os recursos utilizados sejam acessíveis através de um nome, independente de sua localização. Por exemplo, o processo A se comunica com o processo B que se encontra no nó 1 da rede. Caso o processo B migre para o nó 2, o processo A ainda deve ser capaz de encontrá-lo; para isso, normalmente, cada nó possui um tabela de tradução do nome dos processos para a sua localização.

O suporte aos requisitos mencionados pode ser fornecido pelo sistema operacional, por uma biblioteca de funções e/ou pela linguagem de programação.

Finalmente, é preciso que os estados ou dados associados a um processo que não possam ser utilizados após a migração possam ser destruídos. Um exemplo é o que fazer com o processo original após a migração: ele precisa ser destruído e todos os seus identificadores locais liberados.

2.2.1 Suporte do Sistema Operacional

Alguns sistemas operacionais distribuídos como MOSIX (BARAK, 1989) e Sprite (OUSTERHOUT, 1988) fornecem suporte nativo à migração de tarefas. A migração de tarefas quando suportada pelo sistema operacional se torna mais transparente e eficiente, porém o desempenho do sistema é penalizado. O gerenciamento de um sistema dessa natureza é mais complexo e nem todas as aplicações tiram vantagem da funcionalidade, ou as migrações podem ocorrer muito esporadicamente, de forma que o *overhead* não seja compensado.

2.2.2 Suporte de Biblioteca de Funções

Nem todos os sistemas operacionais fornecem suporte nativo à migração de tarefas, e por vezes a troca do sistema operacional utilizado é muito onerosa em termos de reprogramação e aprendizado. A alternativa, então, é realizar a migração em espaço de usuário através de uma biblioteca de funções. Tal biblioteca deve fornecer todo o ferramental necessário à migração e apenas aplicações que fizerem uso explícito das funções implementarão a funcionalidade; logo, a transparência do processo é prejudicada.

2.2.3 Suporte de Linguagem de Programação

Existem linguagens que fornecem suporte à migração de tarefas, trazendo embutidas uma série de funcionalidades que facilitam essa tarefa. Essa abordagem se aproxima da utilização de bibliotecas. Exemplos de linguagens que fornecem tal suporte são Tcl/Tk (OUSTERHOUT, 1994) e Java (GOSLING, 1996). Linguagens interpretadas são naturalmente destinadas à migração e, por isso, vêm sendo muito utilizadas no domínio de agentes móveis.

A linguagem Java, por exemplo, permite que todos os dados da aplicação (instâncias de objetos e variáveis) sejam serializados, ou seja, armazenados de forma adequada, para serem posteriormente utilizados. Esse procedimento pode ser realizado para reiniciar uma tarefa num momento posterior e nada impede que a tarefa seja inicializada em outro processador. Uma série de artigos (FÜNFROCKEN, 1998) (TRUYEN, 2000) aborda o tema da migração de tarefas utilizando Java. Como Java executa sobre uma camada de abstração chamada JVM (Java *Virtual Machine*) é possível realizar migrações transparentes. O suporte por meio de linguagens de programação não permite um alto desempenho na migração, mas não onera aplicações que não façam uso da funcionalidade.

Como, normalmente, uma aplicação não tem acesso a diversas tabelas do sistema operacional, a migração em modo usuário (através de bibliotecas ou linguagens de programação) faz uso do método de *checkpoint* (ponto de salvamento) para poder reiniciar a execução da tarefa no nó destino a partir do ponto onde a execução foi suspensa, ou próximo a ele.

2.2.3.1 *Sistema de Checkpoint*

O sistema de *checkpoint* tem seu funcionamento baseado na premissa de que não é necessário todo o histórico da computação de uma aplicação para restaurar (reiniciar), a partir de certo ponto, sua execução. Entende-se por ponto de execução tanto a questão espacial – em qual instrução encontra-se o programa –; quanto temporal – em qual iteração de um laço está o programa. Assume-se assim que uma aplicação pode ser expressa por uma função discreta e que sua saída no tempo n+1 pode ser determinada pelos valores de um número finito de variáveis da função no tempo n. Ou, conforme a equação

$$f[n+1] = g[f[n]],$$

que, através de uma transformação na função no tempo n, é possível obter a função no tempo n+1.

É necessário, portanto, descobrir que dados são necessários armazenar para permitir a restauração da execução no nó destino. Algumas bibliotecas (POWELL, 1983) realizam essa tarefa de maneira automática, armazenando um registro de suas primitivas que foram utilizadas pela aplicação. Contudo, esse método pode fazer com que informações desnecessárias sejam guardadas. Além disso, caso, por exemplo, sejam armazenadas as mensagens recebidas pela aplicação, ao se restaurar a mesma em um nó destino, pode ser necessário fazer com que os demais processos que com ela se comunicam sejam forçados a retroceder sua execução ao ponto da última mensagem a eles enviados. Isso exige uma gerência global do sistema, para permitir a sincronização dos processos, o que impõe uma grande sobrecarga ao sistema.

Outra abordagem possível é deixar com o usuário (programador) a responsabilidade de fornecer as informações necessárias para a correta restauração da aplicação. Embora, não transparente, esse método é muito mais eficaz, uma vez que a análise semântica das informações permite que apenas os dados de fato necessários sejam salvos.

Basicamente, então, o sistema de *checkpoint* consiste em salvar as informações relevantes no nó origem para, a partir delas, reiniciar a aplicação do ponto de salvamento no nó destino.

Obviamente, as informações não podem ser salvas a cada instrução executada, sob pena de comprometer o desempenho da aplicação. Porém, se os pontos de salvamento forem muito esparsos ao longo do código, ou muito processamento será perdido, pois a aplicação será interrompida e recomeçará a partir do último *checkpoint*, ou a latência da migração será muito elevada, pois ela só iniciará quando a aplicação atingir o próximo *checkpoint*. A forma como será penalizado o desempenho do sistema, nesse último caso, depende da forma de implementação do sistema de *checkpoint*.

Na implantação de *checkpoints* ao longo da aplicação existe uma relação de compromisso entre distância e desempenho. Se os pontos forem muito próximos, muito tempo será perdido salvando as informações necessárias para a migração; caso sejam muito distantes, muito tempo será perdido recalculando informações que poderiam ter

sido salvas. Caso a abordagem espere até que a aplicação atinja o próximo *checkpoint*, o tempo extra em que o processador esteve com excesso de carga, por exemplo, provocará, também, perda de desempenho. O ponto ótimo desta relação varia entre aplicações e, por isso, a abordagem de deixar para o usuário definir os pontos de migração acaba por ser mais eficiente.

Caso a migração seja de responsabilidade do sistema operacional, o problema de salvar as informações sensíveis à aplicação inexiste. Uma vez que, devido ao determinismo intrínseco dos processadores e ao acesso completo ao estado atual da máquina por parte do sistema operacional, todos as dados referentes ao programa estão disponíveis, basta transferi-los para a máquina destino. Por vezes esses dados também contêm informações irrelevantes, porém o desempenho superior das rotinas do sistema operacional frente às do usuário e a transparência proporcionada compensam o *overhead* causado.

Checkpoints ainda podem ser utilizados para prover uma maior tolerância a falhas no sistema. Caso as informações referentes ao processo sejam salvas em um lugar seguro a cada *checkpoint*, em um nó servidor, por exemplo, no caso de falha do nó que executava a aplicação, essa poderá ter sua execução restaurada a partir do último *checkpoint* em um outro nó da rede.

2.3 Algoritmo de Migração de Tarefas

Embora possa variar a maneira como ocorre a migração de tarefas nos diferentes sistemas, pode-se resumi-la em alguns passos gerais:

1. Negociação da migração: é necessário informar os nós envolvidos no processo em que a migração irá ocorrer.
2. Suspensão da execução: o processo é congelado e seu estado é definido como *migrando*. Aqui pode existir uma fila de migração, na qual os processos esperam para ser transferidos, tal como existe a fila de processos prontos para ser escalonados e de processos bloqueados, por exemplo.
3. Redirecionamento da comunicação: as mensagens enviadas ao processo que está sendo migrado precisam ser armazenadas até o fim da transferência da aplicação para o nó destino.
4. Extração do contexto: as informações referentes à aplicação precisam ser transferidas ao nó destino (ver Seção 2.4).
5. Criação do processo no nó destino: após a transferência de algumas informações sobre o processo (o código, por exemplo, quando for o caso), uma instância da aplicação já pode ser criada no nó destino.
6. Transferência do contexto: as informações salvas são enviadas ao nó destino.
7. Encaminhamento das mensagens: as mensagens anteriormente armazenadas são encaminhadas ao nó destino.
8. Reinício da execução: após as etapas anteriores, a computação do processo é retomada. Quando todas as informações forem transmitidas, a instância do processo no nó origem pode ser apagada.

As etapas explanadas não são necessariamente implantadas em todos os mecanismos de migração de tarefas e não ocorrem necessariamente nessa ordem. O momento da

suspensão da execução do processo no nó origem, por exemplo, pode variar de acordo com a implementação.

Duas etapas do algoritmo de migração, por serem fundamentais à consistência da execução do sistema, merecem especial detalhamento e serão explicadas nas próximas seções: a *transferência do contexto* e o *redirecionamento das mensagens*.

2.4 Métodos de Transferência de Contexto

O contexto de uma aplicação engloba os registradores do processador (PC, *Stack Pointer*, uso geral, etc.), os dados (variáveis), o código da aplicação e as informações a respeito dos recursos (arquivos, dispositivos) utilizados pelo processo. Já Fugetta (1998) divide o contexto em três partes: segmento de código, as instruções do programa; segmento de recursos, as referências aos recursos externos utilizados pelo processo; e o segmento de execução, estado do processo, registradores e dados privados.

A questão da migração dos recursos é complexa, pois nem sempre eles podem ser migrados ou estão acessíveis em outros nós. Além disso, em alguns casos, a migração dos recursos envolve uma transferência de dados não aceitável (migrar um banco de dados inteiro, por exemplo). Portanto, doravante, transferência de contexto refere-se apenas à migração dos dados e das instruções dos processos.

Diferentes técnicas podem ser utilizadas na transferência do contexto de um processo entre um nó e outro. Elas apresentam diferentes requisitos e diferentes desempenhos, sendo que o último pode variar de acordo com o tipo de aplicação a ser migrada. As principais técnicas de migração de contexto são apresentadas a seguir.

2.4.1 Cópia (*Total Freezing*)

Esse método congela a aplicação enquanto copia para o nó destino todo o contexto da aplicação. Embora de fácil implantação, esse método é lento, pode sobrecarregar a rede e, devido à referida lentidão, ocasionar perdas de *deadlines* em aplicações de tempo-real. Como vantagem, está o fato de que não restam informações residuais no nó de origem após a migração. São exemplos de sistemas que utilizam esse método: Sprite, Condor (LITZKOW, 1988) e LSF (ZHOU, 1994). Modelos de migração de tarefas baseados em *checkpoints* normalmente implementam essa técnica (NOLLET, 2005a) (BERTOZZI, 2006). A Figura 2.1 ilustra como ocorre a migração nesse caso, na mesma é possível observar que a execução da tarefa só é retomada após o término da transferência do contexto da tarefa.

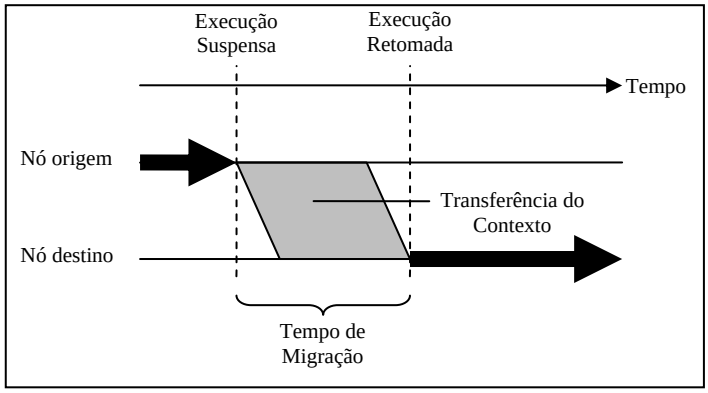

Figura 2.1: Transferência de contexto por cópia.

2.4.2 Pré-cópia

Nesse método o processo a ser migrado continua a ser executado no nó origem enquanto o contexto é migrado, conforme mostrado na Figura 2.2. Após a transferência, o processo é congelado e, além dos valores dos registradores internos, apenas os dados que foram modificados durante o processo de migração são reenviados ao nó destino. Embora essa técnica seja mais complexa e envolva uma maior transferência de dados através da rede, ela diminui o tempo em que a aplicação fica parada. Com isso, há um menor comprometimento quanto ao cumprimento dos *deadlines* de aplicações de tempo real. Esse método é utilizado no sistema operacional distribuído V-System (THEIMER, 1985).

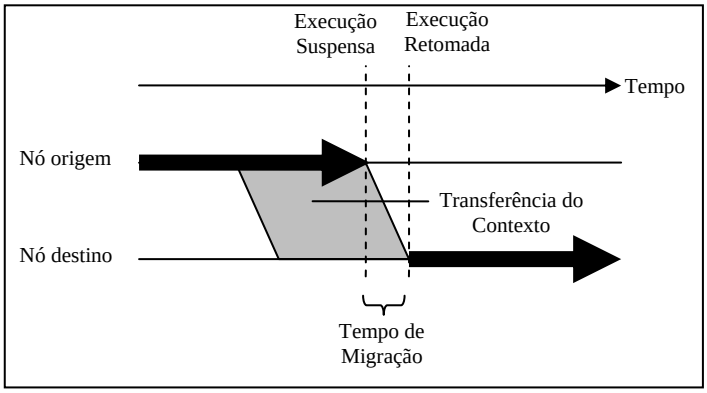

Figura 2.2: Transferência de contexto por pré-cópia.

2.4.3 Cópia sob Demanda (*Copy-On-Reference*)

Esse método, ilustrado na Figura 2.3, baseia-se na prerrogativa de que, apesar de uma tarefa endereçar uma grande quantidade de dados, apenas parte deles são utilizados durante sua execução. Assim não há transferência de dados no momento da migração de tarefas. Porém, quando o processo migrado necessitar de algum dado, este deverá ser buscado no nó origem causando uma grande latência no acesso ao mesmo. Uma analogia a esse método pode ser feita com o princípio de funcionamento das memórias *cache*. No primeiro acesso, os dados não estão na *cache* (nó destino) e precisam ser buscados na memória principal (nó origem); a partir daí, sempre que o dado for novamente necessário esse já estará presente e será acessado rapidamente. Adicionalmente, assim como memórias em *cache*, é possível assumir o princípio da localidade espacial sobre os dados e realizar a transferência de uma grande quantidade de *bytes* contíguos no momento do acesso a um determinado endereço de memória.

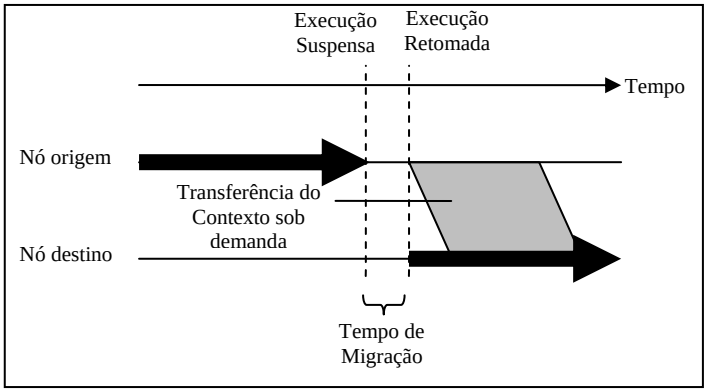

Figura 2.3: Transferência de contexto por cópia sob demanda.

Embora esse método possua um tempo de migração mínimo, o *overhead* causado pelo gerenciamento dos dados presentes/não-presentes e a questão da carga residual no nó origem tornam sua ampla adoção dificultada. Além disso, caso o nó origem pare de funcionar por algum motivo, o processo no nó destino terá a consistência de sua computação comprometida. Logo, esse método não proporciona confiabilidade.

2.4.4 Técnicas de Servidor

Técnicas de Servidor consistem em utilizar um nó seguro como servidor de dados, esperando-se que o nó servidor possua uma grande disponibilidade, e transferir-lhe, através de cópia ou pré-cópia, as informações do contexto do processo a partir do nó origem. No restabelecimento da execução do processo no nó destino, os dados seriam solicitados sob demanda ao nó servidor. Essa técnica garante que, rapidamente, qualquer dependência residual no nó origem será eliminada, mas força um aumento da utilização da rede, já que as informações são, eventualmente, transferidas duas vezes. A carga da rede fica desbalanceada, concentrando o tráfego nos links que levam ao nó servidor. Ainda, caso o nó servidor falhe, o processo de migração fica comprometido.

2.4.5 Sumário

A Tabela 2.1 apresenta uma comparação entre os diferentes tipos de transferência de contexto. Essa análise será particularmente útil para o estudo da aplicabilidade das técnicas em sistemas embarcados realizada no Capítulo 3.

Tabela 2.1: Custos de Métodos de Transferência de Contexto.

Método de Transferência de Contexto	**Tempo de Congelamento**	**Tempo Informações Residuais**	**Dependências Residuais**	***Tempo Inicial de Migração***
Cópia	Alto	Nenhum	Nenhuma	Alto
Pré-cópia	Baixo	Nenhum	Nenhuma	Alto
Cópia sob Demanda	Baixo	Alto	Sim	Baixo
Servidor	Variável	Nenhum (origem) Alto (servidor)	Nenhuma (origem) Sim (servidor)	Alto

2.5 Métodos de Redirecionamento de Mensagens

Ao se mover uma tarefa de um nó para outro, é necessário garantir que todas as mensagens pendentes, em transmissão durante o processo de migração e que serão futuramente enviadas, sejam entregues ao nó destino da tarefa. Sinha (1997) classifica as mensagens a serem redirecionadas em três categorias distintas:

- *Categoria 1:* Mensagens recebidas no nó origem após a suspensão do processo a ser migrado, mas antes dele ser reiniciado no nó destino;
- *Categoria 2*: Mensagens recebidas no nó origem após o reinício da execução da tarefa no nó destino;
- *Categoria 3:* Mensagens que serão enviadas ao processo após o reinício de sua execução no nó destino.

Para cada categoria de mensagens um tratamento deve ser realizado para garantir a correta computação da tarefa. Entretanto, não existe uma única forma de solucionar o problema das mensagens pendentes e, nas propostas dos diversos sistemas operacionais

distribuídos, diferentes técnicas de lidar com o problema foram propostas. As abordagens mais comuns para o problema serão explanadas a seguir.

A nomenclatura utilizada nessa seção merece ser enfatizada para possibilitar a correta compreensão dos mecanismos. Entende-se por *nó origem* o nó em que o processo encontra-se antes da sua migração; *nó destino,* por sua vez, trata-se do nó para onde o processo será migrado; *nó emissor* é o nó onde está a aplicação que envia mensagens ao processo migrado.

2.5.1 Reenvio de Mensagens

Este mecanismo foi utilizado originalmente no sistema V-System e adotado por Tanembaum (1990) no seu sistema Amoeba. O método consiste em forçar o reenvio das mensagens por parte do emissor. Nessa abordagem, as mensagens das categorias 1 e 2 retornam ao emissor com a informação (*status*) de destinatário não encontrado (*not deliverable*) ou são simplesmente descartadas. Assume-se que o emissor armazena uma cópia dos dados enviados e está preparado para realizar o reenvio.

No V-System mensagens do tipo 1 e 2 são descartadas e o nó emissor é informado de que não foi possível entregar as mensagens. O próprio sistema operacional, devido a sua arquitetura, garante o reenvio das mensagens até a recepção ser realizada com sucesso. Com isso, do ponto de vista da aplicação que enviou a mensagem, é garantida parte da transparência do processo de migração.

Já no sistema Amoeba, para mensagens da categoria 1 o nó origem envia uma resposta informando que o processo destino está congelado. O sistema tentará o reenvio até o momento em que as mensagens passarão para a categoria 2. Quando isso ocorre, a mensagem resposta é a de que o processo não é conhecido por parte do nó origem. O nó emissor então, através de uma mensagem de ***broadcast***[2], irá localizar o processo e reenviar a mensagem corretamente.

Após a descoberta da nova localização do processo migrado, o tratamento das mensagens da categoria 3 se dá de forma automática e sem dependências residuais no nó origem.

2.5.2 Nó Origem

Esse é o método utilizado no sistema distribuído Sprite e consiste em manter em uma tabela no nó origem a nova localização do processo após a migração. Se o processo migrar mais de uma vez, para nós distintos, o nó origem inicial (aquele no qual o processo foi criado) deverá ser informado para garantir a consistência do sistema. Mesmo após a migração, as mensagens continuam sendo entregues ao nó origem inicial, sendo ele o responsável pelo devido encaminhamento das mesmas.

O mecanismo de nó origem provoca um aumento na carga da rede, uma vez que as mensagens percorrem um caminho normalmente maior do que se fossem diretamente enviadas ao processo receptor. Mais, esse sistema é vulnerável do ponto de vista de falha em um nó, pois todos os processos por ele criados ficarão inacessíveis. Enquanto mensagens do tipo 3 são encaminhadas para o novo nó, mensagens das categorias 1 e 2 são armazenadas no nó origem até poderem ser recebidas pela tarefa migrada. Nota-se

[2] *Broadcast*: diz-se da mensagem que é enviada para todos os nós da rede.

que além do *overhead* de comunicação, parte do poder de processamento do nó origem será permanentemente desperdiçado com o encaminhamento de mensagens.

2.5.3 *Link Traversal*

Para mensagens da categoria 1 esse método utiliza uma fila de mensagens, que aguardam no nó origem para serem redirecionadas após o reinício da aplicação no nó destino. Para mensagens 2 e 3 o sistema armazena no nó origem um *link*, informando a nova localização do processo. Diferentemente do sistema de *nó origem*, caso o processo seja migrado novamente, um novo *link* será criado no nó destino (sob o ponto de vista da migração original) apontando para o novo destino da aplicação.

Assim, a mensagem percorre a rede e é redirecionada através de vários *links* até encontrar a aplicação destino da mensagem. Essa abordagem traz duas desvantagens, a primeira é sob o ponto de vista de desempenho e a segunda é a sua suscetibilidade a falhas, pois caso um nó deixe de operar, todos os processos por ele já iniciados – e que ainda não finalizados, obviamente – ficarão inacessíveis. Esse método foi utilizado no sistema DEMOS/MP (POWELL, 1983).

2.5.4 *Link Update*

Esse método estabelece canais virtuais entre os processos comunicantes. Assim, no ato de uma migração o nó origem informa, através dos canais virtuais referentes à aplicação, a nova localização do processo. Dessa forma, mensagens da categoria 3 são enviadas diretamente ao nó destino. Mensagens 1 e 2 são armazenadas no nó origem até o reinício da execução da aplicação, quando são, então, encaminhadas ao nó destino. Esse método é utilizado no sistema Charlotte (ARTSY, 1989).

Todos os métodos apresentados pressupõem que o ordenamento das mensagens é feito por uma camada superior (podendo ser responsabilidade da própria aplicação ou do sistema operacional), não sendo necessário garantir que as mensagens serão entregues na mesma ordem em que foram enviadas.

2.6 Gerenciador de Carga do Sistema

Para possibilitar uma migração de tarefas vantajosa, todos os recursos do sistema devem ser caracterizados. Cada nó deve possuir uma imagem (do inglês, *snapshot*) do estado do sistema. A freqüência da atualização dessa imagem é uma relação de compromisso entre a fidelidade do estado atual com o *overhead* de processamento e de carga na rede. Em sistemas dotados de um nó mestre, apenas esse conhece o estado geral do sistema, enquanto que os demais devem prover informações a ele sobre seus processos e recursos locais. À tarefa de coletar informações sobre a carga do sistema e, de acordo com alguma política, mantê-lo equilibrado dá-se o nome de *gerenciamento de carga do sistema*.

Na implementação de um gerenciador de carga duas questões são relevantes: que informações são a ele importantes e com que freqüência essas informações devem ser atualizadas. Para analisar a carga de cada nó do sistema, diversos dados podem ser utilizados, individualmente ou combinados entre si, como o número de tarefas na fila de execução, a carga média, a utilização do processador, a utilização das E/S, a quantidade de tempo livre do processador e a quantidade de memória livre no nó (SMITH, 1988).

Segundo Roush (1995), com base em análises empíricas, a melhor medida de carga é o número de tarefas na fila de execução. Isso se deve ao fato de que, embora todas as informações do sistema sejam dinâmicas, a freqüência de modificação dessa informação é mais lenta e, por isso, ela é menos suscetível a variações momentâneas que outros dados (como a utilização do processador, que pode aparentemente diminuir em função de uma operação de E/S bloqueante, por exemplo).

A freqüência com que as informações sobre o estado do sistema são atualizadas pode ser fixa ou variável. No primeiro caso, o nó mestre pode solicitar periodicamente aos demais nós que enviem informações referentes à sua carga. Ainda é possível adotar uma estratégia em que os nós escravos, sem solicitação, enviem também periodicamente as informações. No entanto, é necessário prever que poderá ocorrer uma defasagem entre as informações enviadas pelos diferentes nós. No segundo caso, a cada alteração na fila de execução local, os nós enviariam seu novo estado para o nó mestre ou para todos os demais nós, de acordo com a implementação. A atualização das informações, portanto, seria baseada na ocorrência de eventos no sistema

É necessário observar que, independentemente da estratégia adotada, centralizada (com nó mestre) ou distribuída, o estado conhecido do sistema não é totalmente consistente. Contudo, novamente segundo Roush, o balanceamento de carga pode, mesmo assim, ser realizado de maneira eficiente, ou seja, de forma a melhorar o desempenho do sistema. Observa-se, logo, que qualquer que seja o modelo de migração de tarefas utilizado – preemptivo ou não –, as informações sobre a carga do sistema são relevantes ou para permitir um remanejamento das tarefas ou para não sobrecarregar ainda mais um nó saturado. Finalmente, entre as abordagens centralizadas ou distribuídas de gerenciamento de carga, verifica-se um menor custo de comunicação e de processamento por parte da primeira e uma maior tolerância a falhas na segunda. Além disso, abordagens distribuídas podem sobrecarregar um nó anteriormente pouco utilizado caso várias tarefas sejam direcionadas a ele; logo, é preciso elaborar mecanismos que evitem que isso ocorra.

A partir do conhecimento sobre o estado do sistema, é necessário definir qual tarefa de um nó sobrecarregado será migrada. Milojicic (2000) e outros autores, de maneira unânime, afirmam que o tempo de execução de uma tarefa migrada deve ser longo, o suficiente, no mínimo, para amortizar o *overhead* causado pela migração. Cabrera (1986) afirma que é possível predizer, ou pelo menos prever, o tempo de vida de uma tarefa com base no tempo em que ela já está presente no sistema. Segundo o autor, 40% dos processos dobram o seu tempo de vida. Abordagens mais complexas, mas ainda baseadas no tempo de vida das aplicações, foram propostas em diversos sistemas distribuídos (HARCHOL-BARTER, 1997; SVENSSON, 1990; WANG, 1993 *apud*. MILOJICIC, 2000). Assim, para permitir que o gerenciador de carga seja mais efetivo na sua função, o tempo de vida de cada tarefa é outra informação relevante a ser enviada pelos nós a respeito de seu estado.

Em casos de gerenciamento distribuído, na implementação das políticas de migração, um nó sobrecarregado pode solicitar que uma de suas tarefas passe a ser executada por outro processador (esse último, normalmente, com menor carga). O nó sobrecarregado envia uma mensagem para toda a rede informando sua situação e aguarda a resposta de outro nó informando que pode dar início à migração. Caso o nó sobrecarregado tenha um bom conhecimento do estado do sistema, ele pode solicitar diretamente a um nó livre que receba alguma(s) de suas tarefas. A esse método dá-se diferentes nomes como *sender-initiated policy* ou *bidding algorithms*.

É possível, também, adotar uma estratégia em que os nós livres do sistema informem sua condição aos demais, solicitando tarefas. Essa política é interessante porque não onera ainda mais os processadores já sobrecarregados, obrigando-os a procurar por nós livres na rede. Novamente, se os processadores livres conhecerem o estado atual do sistema, eles podem solicitar a migração de uma tarefa diretamente a um nó de carga elevada (ao nó mais próximo, por exemplo). A bibliografia se refere a esse método tanto como *receiver-initiated policy*, quanto como *drafting algorithms*. A combinação de ambas as estratégias, *receiver-* e *sender-initiated*, também pode ser adotada, e procura explorar as vantagens de cada um dos métodos.

2.7 Migração de Processos em Sistemas Heterogêneos

Quando um processo é migrado em um ambiente homogêneo não existem problemas no que tange à interpretação dos dados e das instruções. Adicionalmente, o estado do processo pode ser mapeado um a um nos registradores do processador do nó destino.

No entanto, caso o ambiente seja heterogêneo (com nós de arquiteturas distintas), cada nó deverá ter a capacidade de interpretar o formato de dados e de instruções dos demais elementos processadores. Isso impõe um *overhead* ao desempenho do sistema e um maior esforço de programação, já que, para cada arquitetura diferente inserida no sistema, deverá haver um novo interpretador de formato. Em Sinha (1997) o número de interpretadores é formalizado, sendo de $n(n-1)$ interpretadores para um sistema com n tipos de processadores.

Uma forma de contornar o problema (MAGUIRE, 1988) é a utilização de uma representação externa para os dados em cada arquitetura. Dessa forma, o número de interpretadores seria reduzido para n em um sistema com n tipos de processador.

Quanto às instruções, pode-se fazer uso de mecanismos de tradução binária. Embora tais mecanismos provoquem um aumento do processamento dos nós, existem estudos (ALTMAN, 2001) que mostram a viabilidade da utilização dos mesmos. Apesar da alta quantidade de memória requerida para a tradução, através da especulação dinâmica do código executado pelos programas é possível obter performances próximas às alcançadas pela máquina nativa rodando a mesma aplicação.

Ainda no que tange às instruções, uma abordagem semelhante à de Maguire (1988) pode ser utilizada. Trata-se da utilização de um código intermediário, que é interpretado através de uma máquina virtual, ou compilado em tempo de execução pelos diversos processadores presentes no sistema. Java é um exemplo atual desse tipo de estratégia. Java já é a linguagem mais utilizada no desenvolvimento de agentes móveis (*mobile agents*), justamente por prover esse suporte adicional à mobilidade.

Uma alternativa à adoção de códigos intermediários é proposta por Dimitrov (1998). Ele propõe um modelo onde os nós podem ser heterogêneos desde que possuam uma versão do programa compilado para sua arquitetura. Para fugir do problema da transferência do segmento de execução (registradores), a migração pode ocorrer somente na ocasião do chamamento de uma sub-rotina ou método. Tais métodos podem utilizar apenas variáveis locais e parâmetros devem ser passados por meio da pilha de dados. Assim, a transferência do contexto se resume a transferir os dados das pilhas das aplicações. No retorno da sub-rotina, a execução volta ao nó original. Essa estratégia não é transparente e exige a modificação do código para ser suportada. Resumidamente,

as migrações no modelo tratam-se de migrações não preemptivas de granularidade fina, em nível de sub-rotina.

Uma evolução do modelo de Dimitrov é o sistema SNOW, proposto por Chanchio (2002). Nele o autor propõe uma visão lógica da estrutura de memória, de comunicação e do segmento de execução apenas durante a migração. Assim, é possível lidar com diferentes arquiteturas e organizações de sistema, com uma perda desprezível de desempenho durante a execução das tarefas. Após o mapeamento da estrutura lógica sobre a estrutura física do nó que executará a tarefa, nenhuma camada intermediária é necessária na execução das aplicações. Elas podem acessar diretamente os recursos da máquina. Entretanto, o processo de migração proposto ainda é oneroso frente aos aplicados em sistemas homogêneos e exige a utilização de um pré-compilador, o que não oferece suporte a aplicações com código legado.

Uma abordagem mais simples, e por vezes mais eficiente, é a utilização de processadores com diferentes organizações internas, porém com arquiteturas idênticas. Do ponto de vista do programador, os processadores são tidos como homogêneos, mas, internamente aos mesmos, as instruções são processadas de maneira distinta. Nesse caso, os processadores possuem o mesmo conjunto de instruções, mas a presença de estruturas especializadas para o tratamento de diferentes tipos de instruções é alternada entre os nós. Deste modo, em um sistema heterogêneo com um processador DSP (*Digital Signal Processing*), um com presença de *pipeline* e um VLIW (*Very Long Instruction Word*), por exemplo, as tarefas podem circular livremente, sendo executadas em qualquer nó.

Um exemplo de família de processadores com diferentes organizações e mesma ISA (*Instruction Set Architecture*) é o femtoJava, desenvolvido por Ito (2001). Trata-se de um processador Java que interpreta os *bytecodes* da linguagem diretamente em *hardware*. Como o femtoJava é um processador voltado para o mercado de sistemas embarcados, algumas adaptações foram feitas para permitir o acesso a portas de E/S e ele é capaz de executar apenas um subconjunto das instruções Java. Mais detalhes sobre esse processador são encontrados na Seção 5.1.1.

3 APLICABILIDADE

Aqui serão apresentados requisitos não funcionais de um mecanismo de migração de tarefas e será discutida a aplicabilidade das técnicas expostas no Capítulo 2 em sistemas embarcados multiprocessados (MPSoCs), em especial aqueles interconectados por meio de redes *intrachip* (NoCs).

3.1 Requisitos

Esta seção descreve características desejáveis de um modelo de migração de tarefas do ponto de vista de sistemas distribuídos, tais como: desempenho, transparência, tolerância a falhas e escalabilidade (CHANCHIO, 2002) (BUNGALE, 2003) e, no caso específico de sistemas embarcados, eficiência energética.

3.1.1 Desempenho

Se o aumento do desempenho do sistema como um todo é o maior objetivo da migração de tarefas, o processo de migração deve ser tão rápido quanto possível. Logo, o tempo de migração, no qual o processo fica congelado, deve ser minimizado. Nesse âmbito, diversas estratégias de transferência de informações entre o nó origem e o nó destino da tarefa foram desenvolvidas. Como exposto no Capítulo 2, as estratégias basicamente se diferenciam no que diz respeito ao momento em que os dados são migrados de um nó para outro, eles podem ser transferidos no momento da migração (*cópia* e *pré-cópia*) ou sob demanda (*lazy*). Enquanto transferências em tempo de migração aumentam o tempo no qual o processo permanece congelado, transferências sob demanda penalizam, posteriormente, o tempo de acesso aos dados e possuem um gerenciamento mais complexo.

3.1.2 Transparência

Segundo Sinha (1997) transparência é um importante requisito no que diz respeito ao suporte à migração de tarefas e dois níveis de transparência podem ser identificados: *nível de acesso a objetos* e *nível de chamada de sistema e comunicação entre processos*.

3.1.2.1 Nível de acesso a objetos

Trata-se de um requisito mínimo para o suporte à migração de tarefas. Se um sistema suporta transparência no nível de acesso a objetos, então objetos como arquivos e dispositivos podem ser acessados independentemente da sua localização na rede.

Obviamente esse nível de transparência requer a existência de um mecanismo de localização remota.

3.1.2.2 Nível de chamada de sistema e comunicação entre processos

A comunicação entre processos deve ser independente de localização, ou seja, um processo deve localizar o outro não a partir de seu nó processador, mas a partir de um nome. Dessa maneira é possível realizar migrações preemptivas. Caso esse recurso não esteja presente no sistema, após uma migração, o nó onde originalmente a tarefa estava localizada deverá garantir o redirecionamento de toda informação que possuir como destino tal tarefa, o que causaria um *overhead* no nó e na rede. Porém, mesmo em um sistema com esse nível de transparência, o nó original da tarefa é responsável pelo redirecionamento das mensagens ao nó destino enquanto a tarefa estiver em processo de migração, ou seja, para a correta consistência da aplicação, é necessário garantir que a mesma receba as mensagens que lhe foram enviadas.

3.1.3 Tolerância a Falhas

Toda abordagem que faz uso de um nó mestre em alguma função na implementação do mecanismo de migração de tarefas impõe ao sistema uma maior suscetibilidade a falhas. Por outro lado, o determinismo de mecanismos distribuídos existe, mas a validação dos mesmos, para todos os casos possíveis, é extremamente difícil e, em tempo de execução, podem ocorrer situações não previstas que levem ao funcionamento incorreto do sistema. Uma solução híbrida, contar com nós mestres redundantes, seria possível, mas, embora aparentemente óbvia, de acordo com as pesquisas realizadas, nunca foi proposta.

Cabe aqui uma ressalva: embora a funcionalidade de migração de tarefas forneça tolerância a falhas em função do mau funcionamento de um nó do sistema sob o ponto de vista das aplicações, é necessário, também, que o próprio mecanismo seja, por sua vez, igualmente tolerante a falhas. Desta maneira, durante todo o processo de migração, é interessante manter as informações já transferidas em ambos os nós envolvidos na migração (origem e destino) e não descartá-las à medida que forem enviadas. Isso garante que, caso ocorra uma falha durante a transferência, na transmissão ou no nó destino, a migração possa ser reiniciada sem maiores problemas, por outro caminho ou para outro nó.

Informações residuais de um processo em um nó são indesejadas porque no caso de falha a aplicação pode tornar-se inacessível ou passar a funcionar erroneamente. Logo, abordagens de transferência de contexto por demanda e redirecionamento de mensagens pelos métodos de *nó origem* e *link traversal* devem ser evitadas caso o objetivo seja desenvolver um mecanismo tolerante a falhas.

3.1.4 Escalabilidade

O bom funcionamento do mecanismo de migração, ou seja, o funcionamento eficiente, transparente e tolerante a falhas, deve se manter independentemente do número de nós do sistema. Mais que isso, o número de migrações necessárias no sistema não deve afetar o desempenho do mesmo, ou seja, dado um tempo de amortização desde a última migração, o sistema deve começar a operar de maneira mais eficiente.

3.1.5 Eficiência Energética

Eficiência energética é um requisito inerente a muitos sistemas embarcados. Dessa maneira não é sempre interessante a aplicações desse domínio obter o melhor desempenho possível. Uma migração de tarefas muito cara em termos energéticos, ou seja, se a transferência demandar muita energia, pode tornar-se pouco interessante em plataformas embarcadas.

Uma possibilidade é alterar o foco principal da funcionalidade, no lugar de procurar aumentar o desempenho ou a tolerância a falhas. Visa-se, então, garantir uma maior eficiência energética por parte do sistema, sem comprometer o seu funcionamento. O balanceamento de carga associado à aplicação de DVS (do inglês, *Dynamic Voltage Scaling*) (WEISER, 1994) ou DPM (do inglês, *Dynamic Power Management*) (BENINI, 2000) pode proporcionar ganhos energéticos ao sistema, mesmo que em um primeiro momento a migração provoque o aumento do consumo (WRONSKI, 2006).

A técnica de DVS consiste em diminuir a tensão de operação do sistema ou parte dele, consumindo assim menos energia. Por sua vez, DPM é o nome que se dá a técnica de desligar partes ociosas do sistema. Pode-se, ainda, utilizar as duas técnicas concomitantemente no mesmo sistema. Com o aumento percentual do consumo estático, em relação ao consumo total, nas novas tecnologias o uso de DPM vem se destacando, já que é mais efetivo na diminuição de consumo estático. Existem também, não obstante, diversos *chips* comerciais que fazem uso de DVS. Finalmente, DPM pode ser visto como um caso extremo de DVS, aquele em que a tensão é levada a zero. Um potencial problema a ser contornado com a aplicação de DPM é o fato de que os dados computados são perdidos, ou seja, não permanecem armazenados nos registradores, o que não ocorre com a aplicação de DVS.

Para aplicação de DVS em sistemas MPSoCs obter os melhores ganhos possíveis, distribui-se as tarefas o mais igualmente possível entre os nós, considerando a carga computacional de cada uma. Com isso é possível diminuir a tensão de operação de todos os nós. Dessa maneira têm-se maiores ganhos porque existe uma relação quadrática entre tensão e consumo de energia dinâmica. Diminuindo-se a tensão, baixa-se também a freqüência e se gasta mais tempo para realizar a mesma computação; para o mesmo intervalo de tempo, porém, se gasta ainda menos energia. A Figura 3.1 ilustra essa relação.

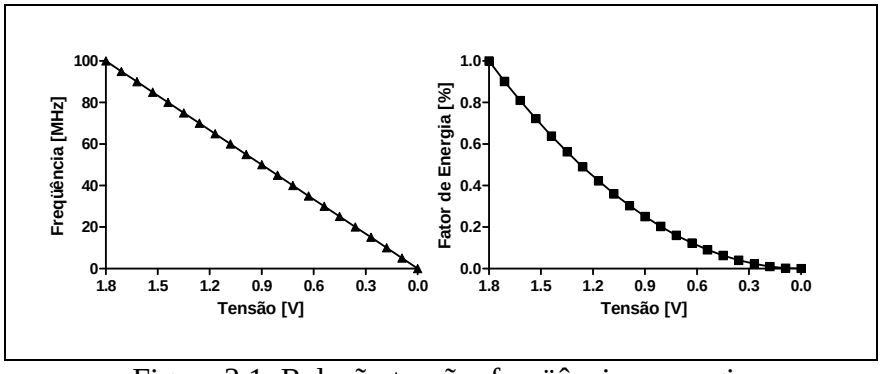

Figura 3.1: Relação tensão, freqüência e energia

Já no uso de DPM, associado à alocação de tarefas, deve-se concentrar ao máximo as tarefas em um mesmo nó, desde que não se comprometa o funcionamento do sistema, e desligar os nós ociosos.

Segundo Brião (2008), a escolha entre concentrar ou distribuir tarefas entre os processadores, para fins de economia de energia, depende da tecnologia utilizada e da quantidade de comunicação entre as tarefas.

Outra forma de economizar energia, sem o uso de DVS ou DPM, é alterar o gerenciador de carga de maneira que ele passe a considerar também a comunicação entre os processos. Desse modo processos comunicantes entre si, que provocam um grande volume de dados e, conseqüentemente, consumo na rede, são migrados para um mesmo processador.

3.2 Análise Crítica

Em sistemas embarcados o requisito chave é eficiência energética. Transparência pode ser relegada a um segundo plano, já que normalmente o usuário (programador) é um especialista do sistema e cada plataforma lançada, normalmente, exige que novos programas sejam desenvolvidos e compilados para ela. A reusabilidade está presente em nível de código fonte, e não em código nativo para os processadores presentes. Diante disso, alteração das aplicações a fim de implementar o suporte à migração não é um problema maior.

Boa parte dos sistemas embarcados atuais não provê tolerância a falhas. Aplicações comerciais que se destinam ao usuário doméstico, como MP3 *Players*, PDAs (*Personal Digital Assistant*) e câmeras digitais, não são de uso crítico e em prol da eficiência energética (duração da bateria do celular, por exemplo) os fabricantes não provêem tolerância a falhas em seus produtos. Logo, dificilmente um sistema contendo um processador de uso geral e um processador DSP continuará a responder adequadamente caso um dos dois falhe. A despeito de ser um requisito importante em diversos domínios – tais como sistemas presentes em automóveis e aplicações médicas –, aspectos de tolerância a falhas não foram abordados nesse trabalho. Uma vez que o foco da pesquisa se dá na concepção arquitetural de um mecanismo de migração de tarefas, a sua adoção como maneira de corrigir a degradação permanente de um nó da rede é deixada como um interessante tópico a ser abordado no futuro.

Já a escalabilidade, com o ritmo de aumento de integração atual, deve ser vista com atenção, pois o desenvolvimento de um mecanismo de migração não escalável pode levá-lo a tornar-se obsoleto rapidamente. Embora hoje uma NoC 4x4 de topologia grade seja suficiente para a maioria dos sistemas, em poucos anos poderão coexistir mais de 100 núcleos em um mesmo *chip*.

O desempenho do mecanismo deve ser o maior possível, desde que para isso não seja comprometida a eficiência energética do mesmo. Obviamente a migração de tarefas consumirá energia, entretanto, a idéia aqui é que o reposicionamento de uma ou mais tarefas possibilite uma posterior melhora no consumo.

Considerando todo o mencionado, verifica-se que, para a minimização do consumo de energia, o contexto transmitido de uma aplicação a ser migrada deve ser o menor possível. Em vista disso, a adoção de técnicas que utilizem *checkpoints* é indicada para mecanismos de migração embarcados, porém, a transparência oferecida por um suporte em nível de sistema operacional é bem-vinda desde que não comprometa o desempenho do mecanismo. Quanto às estratégias de migração de contexto, a opção deve ser feita entre as opções de cópia ou pré-cópia, pois a utilização de transferência sob demanda ou utilizando um servidor de dados onera o desempenho do sistema e da rede,

respectivamente. A primeira exige um suporte em cada nó para a identificação de dados não presentes e sua solicitação no nó origem. Já a segunda faz com que os dados sejam transmitidos duas vezes através da rede, uma vez do nó origem ao servidor e em seguida do servidor ao nó destino. Isso pode sobrecarregar os enlaces e aumenta o consumo de energia por parte dos *links* e roteadores.

No que tange ao redirecionamento de mensagens, novamente no intuito de diminuir a complexidade e o consumo, a estratégia ideal seria a perda das mensagens transmitidas com posterior reenvio das mesmas já que o armazenamento das mensagens no nó origem para posterior entrega causaria um gasto de energia em *buffers* adicionais. Uma estratégia a ser considerada e mensurada, embora comprometa a transparência, seria informar os processos comunicantes da migração da aplicação, fazendo com que o envio de mensagens seja suspenso momentaneamente até segunda ordem. Com isso, mensagens não seriam perdidas e, principalmente, não seriam enviadas, aliviando a rede e o consumo.

Relegando, então, transparência e tolerância a falhas, o gerenciamento centralizado da carga do sistema, com um nó mestre, proporciona redução de energia, já que apenas um nó executaria uma versão mais complexa do sistema operacional e os demais proveriam apenas funcionalidades básicas do mesmo.

4 ANÁLISE DO ESTADO DA ARTE

Este capítulo apresenta trabalhos sobre modelos de migração de tarefas já desenvolvidos para o domínio de sistemas embarcados. Fica clara a relação de herança de diversas características do domínio de sistemas distribuídos nos modelos propostos. Também é possível observar que as restrições energéticas afetam principalmente a transparência do processo de migração. Uma análise sobre cada um dos mecanismos é realizada após a apresentação dos mesmos e uma comparação entre eles é realizada na Seção 4.2. Finalmente, na Seção 4.3 é apresentada a plataforma que está sendo desenvolvida pelo Grupo de pesquisa.

4.1 Trabalhos Relacionados

4.1.1 Bertozzi

Em Bertozzi (2006) é proposto e simulado um modelo de migração de tarefas para MPSoCs. Esse é o primeiro trabalho divulgado que passa da fase de prova de conceitos e realiza medições. Os resultados foram obtidos através de simulações em um ambiente chamado MPARM (LOGHI, 2004).

A plataforma utilizada possui um número variável de processadores do tipo ARM v7 de 32 bits. Um processador é chamado mestre e é responsável pela gerência da migração das tarefas. Os demais são chamados escravos e não podem iniciar uma migração de forma autônoma. Cada processador possui uma memória privada e existe uma memória compartilhada através da qual é realizada a comunicação entre processos. Todos os elementos da plataforma são interconectados através de um barramento. O sistema operacional executado em cada um dos nós é do tipo ucLinux (UCLINUX, 2007).

A estratégia utilizada leva em consideração as restrições energéticas presentes em sistemas embarcados e procura criar um modelo de migração o mais leve possível. O modelo se baseia em *checkpoints* e é todo realizado em nível de aplicação, possuindo, então, baixo *overhead*, mas também baixa transparência.

Os resultados exibidos são animadores e mostram ganhos obtidos com a migração de tarefas. Basicamente, fica claro que, apesar do *overhead* imposto pela transferência do contexto pela rede, é mais vantajoso migrar tarefas do que sobrecarregar um determinado processador. Bertozzi mostra ainda a relação entre o tempo de amortização do impacto no desempenho causado pela migração de tarefas e o tamanho do contexto dos processos migrados.

Um resumo das características da plataforma utilizada é apresentado na Tabela 4.1; a topologia do sistema é apresentada na Figura 4.1.

Tabela 4.1: Especificação da Plataforma de Bertozzi (2006).

Meio de Comunicação	Barramento
Tipo de Processadores	Homogêneos
Processadores	ARM v7 32 bits
Sistema Operacional	uCLinux Kernel 2.6

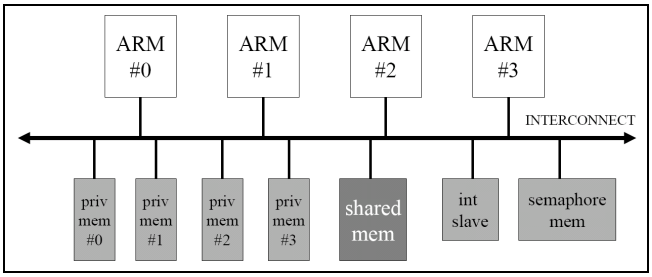

Figura 4.1: Topologia da Plataforma (BERTOZZI, 2006).

O *overhead* no desempenho causado pelo mecanismo de *checkpoints* é analisado e revela-se desprezível mesmo para aplicações com grande número de pontos de migração. As tarefas migradas são sintéticas e possuem contextos de diferentes tamanhos. Também é feita uma análise sobre uma aplicação real, de criptografia, e os autores do trabalho concluem que a baixa transparência do modelo torna-se uma vantagem. Vantagem essa justificada pelo fato de que se é o usuário que informa os pontos de migração, o mesmo pode escolher lugares em que o contexto da aplicação é menor, diminuindo o tempo de migração. Entretanto, a necessidade de explicitamente decidir os dados a serem migrados requer um grande conhecimento prévio das aplicações por parte dos programadores, logo, mesmo de posse de código fonte, a adaptação das aplicações para o modelo não é trivial. Nenhuma análise sobre a consistência da comunicação entre processos é apresentada. Finalmente, o autor também não faz uma análise sob o aspecto energético da solução desenvolvida.

4.1.1.1 Análise dos Resultados

Nesta subseção são apresentados os resultados obtidos por Bertozzi, resumidos anteriormente, e é feita uma análise dos mesmos.

A primeira análise feita mostra o impacto da utilização da técnica de *checkpoint* no desempenho das aplicações. A Tabela 4.2 mostra a penalidade sofrida no desempenho da aplicação em função da freqüência de pontos de salvamento. Embora os dados apresentados sejam função do tempo, as chamadas à API (*Application Programming Interface*) de migração estão espalhadas espacialmente no código, e não são acionadas periodicamente via interrupção ou mecanismo semelhante como seria possível pensar. O que, de fato, foi realizado, foi a medição do tempo necessário para a aplicação verificar se é necessário salvar seu contexto e, não o sendo, retomar seu fluxo normal de execução. O tempo encontrado foi de 2us.

A Tabela 4.3 mostra que o tempo de migração de uma tarefa no sistema é linear em relação ao tamanho do contexto das aplicações. São apresentados os tempos de

migração para processos com diferentes tamanho de contexto, embora não seja analisado o tempo de migração em função da carga de comunicação do sistema. Particularmente, no estudo de caso, as transferências de dados que ocorrem no barramento podem acontecer em virtude da migração de uma tarefa, ou devido ao acesso à memória compartilhada

Tabela 4.2: *Overhead* causado pelos *checkpoints*.

Freqüência de *Checkpoints*	**Impacto no Desempenho**
1 (1 call every ms)	0,2%
0,2 (1 call every 5 ms)	0,04%
0,1 (1 call every 10 ms)	0,02%
0,05 (1 call every 20 ms)	0,01%

Fonte: Bertozzi, 2006.

Tabela 4.3: *Overhead* do tempo de migração.

Tamanho do Contexto [Kb]	**Tempo [ms]**	**Tempo/Tamanho [ms/Kb]**
1	0,0565	0,0565
8	0,4471	0,0559
16	0,9231	0,0577
32	1,8131	0,0567
64	3,6136	0,0565

Fonte: Bertozzi, 2006.

A comparação realizada confronta um cenário desbalanceado contra o cenário onde as tarefas migrariam para obter um balanceamento de carga. O experimento utiliza apenas dois processadores e quatro tarefas. A Figura 4.2 mostra que após um determinado tempo de amortização, uma tarefa migrando de um cenário desbalanceado para um processador com menor ocupação consegue obter mais tempo de CPU[3] (*Central Processor Unit*) em um menor tempo de execução. O eixo das abscissas representa o tempo em que a aplicação esteve de fato utilizando o processador, enquanto as ordenadas apontam o passar do tempo. A curva ***standalone*** representa uma aplicação executando sozinha no processador, a curva ***2 tasks*** representa um processador com duas tarefas e a curva ***3 tasks***, por sua vez, representa um cenário com 3 tarefas em um único processador. As demais curvas mostram, para diferentes tamanhos de contexto, o impacto da migração em um cenário onde o processador A encontra-se inicialmente com 3 tarefas e uma dessas tarefas migra para o processador B, onde a princípio executava-se apenas uma tarefa. As migrações iniciam por volta dos 50ms (eixo y) e são concluídas em diferentes intervalos de tempo, por volta do momento em que a tarefa obtém 40ms de CPU (eixo x).

[3] Tempo de CPU: período de tempo em que uma tarefa é de fato executada pelo processador

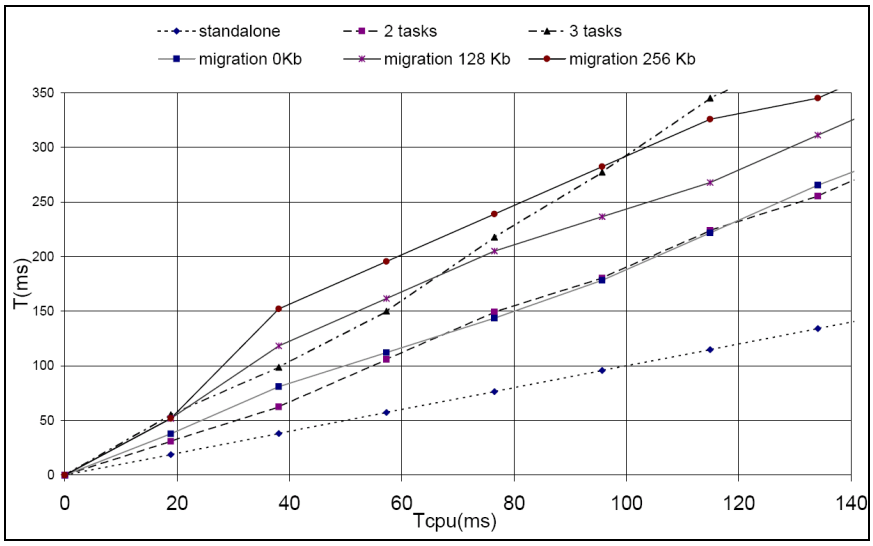

Figura 4.2: Tempo de CPU vs. Tempo de execução (BERTOZZI, 2006).

Bertozzi verifica ainda que o tempo de amortização da migração é função linear do tamanho do contexto da mesma, conforme mostra a Figura 4.3. Os pontos do gráfico representam os resultados empíricos obtidos e a reta os valores esperados, calculados analiticamente.

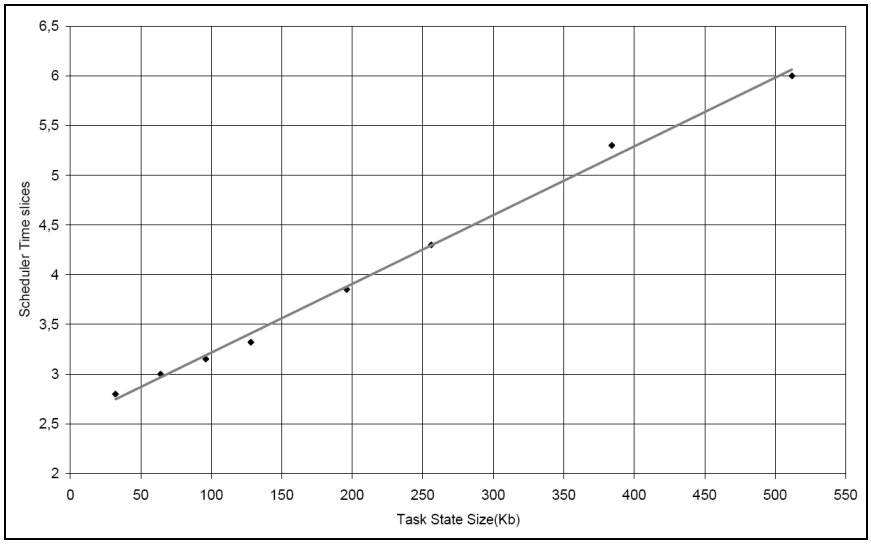

Figura 4.3: Tempo necessário para amortização da migração de tarefas (BERTOZZI, 2006).

Embora Bertozzi não faça uma análise sob o ponto de vista do consumo, o fato de obter uma mesma fatia de processamento em um menor tempo, operando à mesma freqüência, leva a crer que, aplicando migração, ocorra também uma economia de energia no sistema.

4.1.2 Nollet

Em seu trabalho Nollet (2005a) apresenta um exemplo de sistema que faz uso da facilidade de migração de tarefas e possui interconexão por meio de NoCs. A análise realizada é baseada em prototipação, contudo, não são apresentados muitos dados a respeito do desempenho, seja temporal ou energético. Um valor para o tempo perdido

na migração de uma tarefa é inferido apenas analiticamente, não sendo demonstrada nenhuma confirmação da correção do mesmo.

A plataforma heterogênea é emulada ligando um processador StrongARM de um PDA a um FPGA (*Field Programmable Gate Array*), onde se encontram os demais nós. Nesse trabalho, os nós no FPGA são reconfiguráveis em tempo de execução, e as tarefas podem ser executadas tanto em *software* quanto em *hardware*. Como foge do escopo do presente estudo, a forma como é feita a transição *software-hardware* das tarefas não será abordada, mas pode ser vista em Mignolet (2003). Um resumo das especificações da plataforma é apresentado na Tabela 4.4.

Tabela 4.4: Especificação da Plataforma de Nollet (2005a).

Meio de Comunicação	NoC
Tipo de Processadores	Heterogêneos
Processadores	StrongARM, SoftCore Microblaze
Sistema Operacional	*In-house*

O modelo de migração envolve a interação do sistema operacional (SO) e, por conta disso, é mais transparente que o proposto por Bertozzi (2006), mesmo tendo sido proposto anteriormente. Porém, apesar do suporte, o modelo é baseado em *checkpoints*, assim é necessário que o programador explicitamente introduza pontos de migração no seu código, o que, por sua vez, diminui a transparência do modelo.

Dois mecanismos de migração são propostos. Em ambos é garantida a consistência da comunicação entre diversos processos, sendo que cada nó possui uma *lookup table* através da qual é possível saber onde está sendo executada uma determinada aplicação.

No primeiro mecanismo, após o sistema operacional decidir realizar a migração de uma determinada tarefa, ele aguarda que a mesma atinja um *checkpoint* no seu fluxo de execução. Quando um processo chega ao ponto de migração e existe a solicitação de que migre, ele envia uma mensagem às demais aplicações com as quais realiza comunicação, informando que entrará em processo de migração. As aplicações devem informar da ciência da migração e, a partir desse momento, o processo pode começar a migrar. O sistema operacional então instancia a tarefa no nó destino, copiando todo contexto da aplicação, e faz com que o nó origem redirecione ao nó destino todas as mensagens, inclusive as não processadas. A cada mensagem informando o reconhecimento da migração da tarefa, o sistema operacional atualiza a *lookup-table* do nó emissor da mesma, para não ser mais necessário redirecionar mensagens desse nó. Após todos os nós passarem essa informação, o sistema operacional libera os recursos relativos à tarefa migrada no nó de origem. Uma ilustração do funcionamento desse mecanismo é mostrada na Figura 4.4.

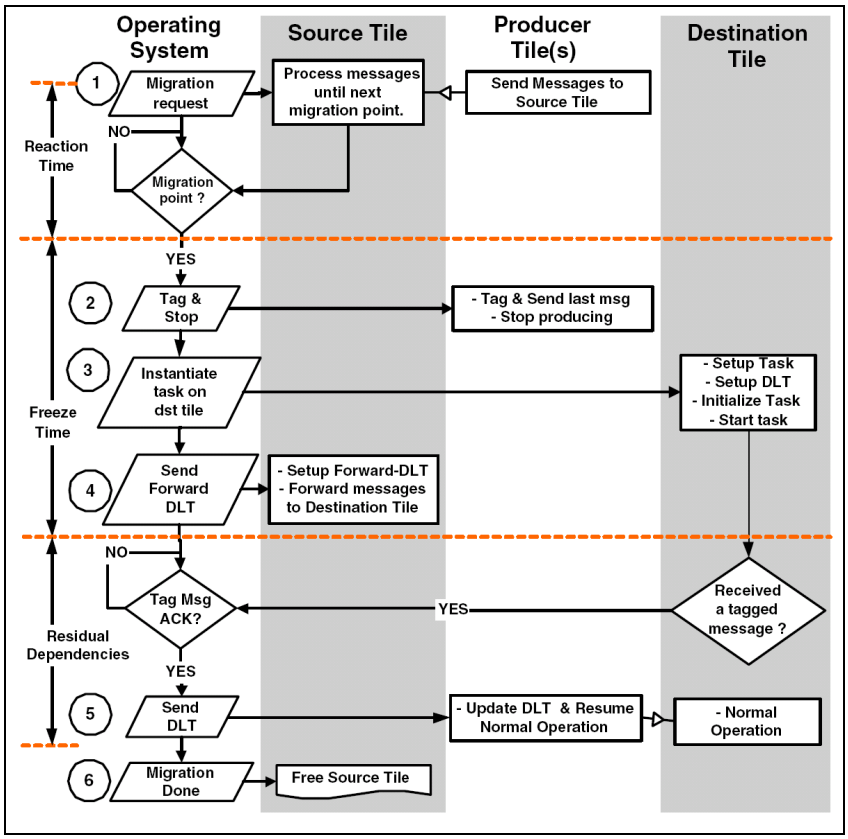

Figura 4.4: Mecanismo de Migração (NOLLET, 2005a)

No segundo mecanismo, não há migração de contexto ou de mensagens, pois se trata de uma abordagem *dataflow*. As aplicações produtoras de dados apenas são informadas de que uma tarefa migrará e param de produzir dados. A tarefa a ser migrada, quando atinge um ponto independente de estado (sem contexto), é então transferida para o nó destino. Logo após, o sistema operacional informa todas as tarefas envolvidas que podem reiniciar o envio de mensagens.

A abordagem de solicitar o cancelamento do envio de mensagens é necessária devido à baixa capacidade de armazenamento das filas presentes em cada nó do sistema. Essa estratégia diminui a transparência do processo de migração, porém proporciona uma maior economia de energia.

Não são apresentados resultados qualitativos ou quantitativos da modelagem proposta para nenhuma métrica (seja desempenho, escalabilidade ou potência) e nenhum estudo de caso é explorado. A contribuição do artigo está na implementação de um mecanismo de migração de tarefas e não na comprovação de sua eficiência.

Em Nollet (2005b) uma forma de diminuir o *overhead* causado pela inserção de *checkpoints* é proposta baseando-se em registradores de depuração. Registradores de depuração, quando presentes, provocam uma interrupção no processamento quando o PC (*Program Counter*) atinge um dos endereços presentes nesses registradores. Dessa maneira, embora não contribua com a transparência do processo de migração, este se torna mais rápido, já que conta com um suporte de *hardware*. Basicamente, no início da aplicação, a mesma informa ao sistema operacional os seus pontos de migração e esses são mapeados nos registradores. A partir de então, ao atingir cada um desses pontos, a tarefa é interrompida e o sistema operacional verifica se há necessidade de realizar uma

migração. Nenhuma análise é feita caso o número de pontos de migração seja maior que o número de registradores de depuração.

Já em Nollet (2004) é proposto um sistema operacional distribuído especialmente desenvolvido para sistemas com meio de comunicação baseado em NoCs. Segundo suas análises, através do gerenciamento adequado dos recursos da rede, a vazão da NoC pode melhorar em até cinqüenta por cento, aumentando conseqüentemente o desempenho do sistema como um todo. O sistema operacional utiliza duas NoCs que interligam todos os nós do sistema: uma é responsável pela transmissão de dados das aplicações através da rede e na outra trafegam informações referentes ao controle do sistema, gerenciamento da carga, monitoramento da taxa de injeção de dados na rede e atualização das rotas dos pacotes (o roteamento nesse caso é adaptativo, baseado em *lookup tables*). Apesar do *overhead* de área e energia causado pela utilização das duas NoCs, a estratégia garante que pacotes de controle não precisarão disputar banda com os pacotes de dados. A arquitetura suportada é do tipo mestre/escravo, ou seja, embora a duplicação das redes ofereça tolerância a falhas, o sistema centralizado peca nesse aspecto.

4.1.3 Ozturk

Ozturk (2006) propõem uma estratégia de migração seletiva, a qual decide migrar uma tarefa (código) ou dado, de maneira a satisfazer um requisito de comunicação em termos de largura de banda. A escolha entre as duas opções é realizada dinamicamente (em tempo de execução), baseada em estatísticas coletadas estaticamente (em tempo de compilação). O objetivo deste trabalho é reduzir o consumo de energia de um MPSoC durante a comunicação entre os processadores.

A Figura 4.5 ilustra as duas opções de migração. A migração de dado é representada na Figura 4.5a, onde o processador *Pi* envia um dado *S* ao processador *Pj*, o qual contém o procedimento *Q* que executa sobre *S* (*Y* é o dado de saída). A migração de tarefa é representada na Figura 4.5b, onde *Pj* envia a tarefa *Q* para *Pi* e este devolve o resultado para *Pj*.

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

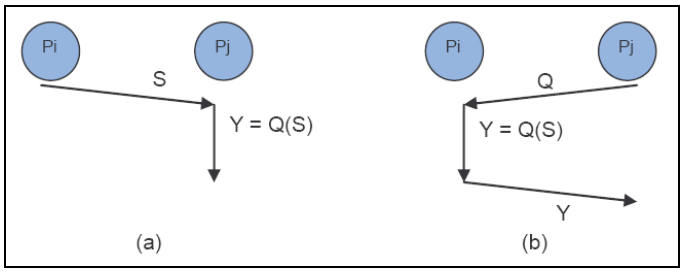

Figura 4.5: (a) Migração de dados; (b) migração de tarefa (código).

A arquitetura MPSoC utilizada no trabalho é composta de 8 processadores interligados por um barramento. Cada processador possui memória local de 32kB de capacidade. A comunicação entre os processadores é realizada através de troca de mensagens. A Tabela 4.5 resume essas informações. A migração seletiva foi implementada e comparada com duas estratégias alternativas: uma estratégia em que sempre acontece migração de dados e outra que sempre migra código. Os resultados coletados indicam que a migração seletiva reduz o consumo de energia e o tempo de execução.

Tabela 4.5: Especificação da Plataforma de Ozturk (2006).

Meio de Comunicação	Barramento
Tipo de Processadores	Homogêneos
Processadores	Não informado, *pipeline* de 5 estágios
Sistema Operacional	Não informado

4.1.4 Pittau

Em seu trabalho, Pittau (2007) avalia o uso da migração de tarefas e seu impacto no desempenho de aplicações de tempo-real *soft*. Ele demonstra, para processadores capazes de operar em diferentes freqüências, que pode ser viável em termos de energia migrar tarefas para diferentes processadores. O modelo de Pittau se difere dos demais no que diz respeito à migração dos dados. Cada processador trabalha com os dados em uma memória local e, na ocasião de uma migração, esses dados são copiados para uma memória compartilhada para em seguida serem transferidos para a memória privada do processador destino da tarefa. Essa abordagem é interessante em sistemas interconectados por barramento porque evita o *overhead* causado por um mecanismo de troca de mensagens, uma vez que os dados são simplesmentes logicamente realocados na memória. A arquitetura utilizada por Pittau pode ser visualizada na Figura 4.6; um resumo das características da plataforma é apresentado na Tabela 4.6.

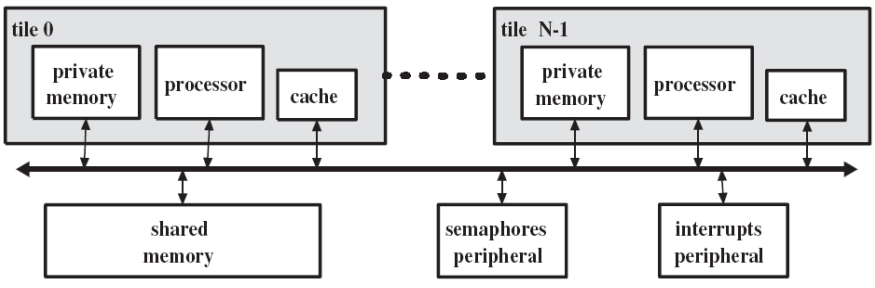

Figura 4.6: Topologia da plataforma (PITTAU, 2007)

Tabela 4.6: Especificação da Plataforma de Pittau (2007).

Meio de Comunicação	Barramento
Tipo de Processadores	Homogêneos
Processadores	Não informado
Sistema Operacional	uCLinux Kernel 2.4

O estudo de caso do trabalho é uma aplicação que implementa um rádio digital FM. A aplicação é dividida em tarefas menores e todos os mapeamentos possíveis que satisfaçam as condições de tempo-real são avaliados. A partir desses dados é implementado um mecanismo que mantém as tarefas distribuídas da maneira mais econômica possível na plataforma de acordo com a taxa de transmissão (*bitrate*) desejada pelo usuário. Quanto maior a taxa, maior a freqüência e/ou mais processadores são utilizados pela aplicação.

Embora avalie a viabilidade da aplicação da migração de tarefas em um sistema embarcado com uma aplicação real, Pittau não explicita os custos da migração de tarefas, estando interessado apenas nos estados final e inicial da distribuição da aplicação na plataforma. O uso de uma memória compartilhada para a cópia dos dados é uma estratégia interessante em sistemas que utilizam barramento; contudo, em sistemas baseados em NoC, a utilização de um espaço de memória compartilhada deve ser feita através de uma camada de *software* e *hardware* que trabalha sobre o mecanismo de troca de mensagens. Uma solução dessa natureza aumenta o custo da transmissão dos dados embora facilite a comunicação e não cause efeitos de sincronização, uma vez que os dados não são escritos ao mesmo tempo pelos processadores.

4.2 Análise Crítica

Dos modelos apresentados, apenas o de Nollet (2005a) utiliza NoC como meio de interconexão, sendo que os demais utilizam barramento. Outros aspectos dos modelos propostos também diferem, o que dificulta a comparação entre os mesmos. Enquanto Bertozzi (2006), Nollet e Pittau (2007) detalham o mecanismo de migração, o foco de Ozturk (2006) é a heurística de migração aplicada.

No que diz respeito à organização de memória das soluções apresentadas, Nollet e Ozturk utilizam apenas memórias privadas, em uma abordagem puramente distribuída. Já Bertozzi e Pittau possuem memória compartilhadas em seus sistemas. Apenas Pittau não utiliza troca de mensagens para transferir o contexto das tarefas de um nó para outro. Mesmo Bertozzi, com seu modelo de memória compartilhada, utiliza troca de mensagens através de *mailboxes* localizadas na memória global do sistema.

Em nenhum dos modelos propostos foi feita uma análise sobre a escalabilidade, ou seja, a influência do número de nós sobre o efeito da funcionalidade de migração. Espera-se, porém, que, com o aumento do número de nós no sistema, abordagens que utilizam redes-em-chip sejam menos suscetíveis a gargalos de comunicação do que aquelas que utilizam barramento.

Considerando o que foi anteriormente exposto, uma comparação justa no que tange ao desempenho dos modelos não pode ser feita, uma vez que eles diferem em nível arquitetural, ou seja, não possuem o mesmo tipo de meio de comunicação e organização de memória. Além disso, mesmo que tenham sido apresentados tempos de migração, os

processadores utilizados não são os mesmos, logo, não é possível fazer uma comparação precisa.

Algo que não é realizado em nenhum dos trabalhos mencionados é o levantamento do custo em termos de consumo de energia das migrações realizadas. Mesmo Ozturk, que realiza o levantamento energético da utilização de suas heurísticas, não detalha o consumo da migração em si, dando ênfase ao custo das alocações em si. Mesmo nas suas comparações, ele não considera o custo computacional da migração, apenas a energia gasta em comunicação.

O presente trabalho se propõe a detalhar o custo energético do modelo de migração desenvolvido. Além disso, é realizado também o levantamento do desempenho do mecanismo. Considera-se que ambos os levantamentos se fazem necessários para a completa caracterização de um modelo de migração de tarefas. Procurou-se assim preencher uma lacuna existente nos modelos existentes. O custo computacional e energético do processo de migração precisa ser levado em consideração na aplicação de heurísticas que se propõem tanto a melhorar o desempenho quanto o consumo do sistema. Como definido por Bertozzi, o intervalo mínimo entre migrações em um sistema depende do tempo de amortização do processo. No momento em que o sistema se encontra com uma nova e mais eficiente distribuição das tarefas, é necessário permanecer com esse arranjo de tarefas imóvel por um período mínimo de tempo, mais precisamente, até que o custo da migração em termos de desempenho seja compensado.

O mesmo ocorre quando o foco é a energia consumida pelo sistema. É necessário considerar o custo energético da migração e permanecer em um mesmo estado, com o mesmo arranjo de tarefas, por um determinado período de tempo. No presente trabalho esse levantamento é realizado e verifica-se que o tempo de amortização em termos de energia é, significativamente, maior do que aquele quando se considera apenas o desempenho.

4.3 Plataforma LSE

O Laboratório de Sistemas Embarcados (LSE) da Universidade de Federal do Rio Grande do Sul (UFRGS) vem desenvolvendo uma plataforma multiprocessada embarcada que agregue as últimas novidades tecnológicas e científicas. Além dos requisitos desejáveis já mencionados no âmbito dos sistemas embarcados, a plataforma deverá ser adaptativa e robusta. Diferentes trabalhos de pesquisa do grupo vão ao encontro desse anseio, sendo que parte da plataforma já foi implementada e vem servindo de apoio aos estudos realizados no laboratório.

Os processadores utilizados serão do tipo femtoJava (vide Seção 5.1.1) e, com a constatação de Nollet (2005a) de que é viável implementar migração de tarefas sobre MPSoCs baseados em NoCs, foi adotada como meio de comunicação a rede SoCIN (ZEFERINO, 2004) (vide Seção 5.1.2).

O desenvolvimento de um *middleware* adaptativo baseado em Java é tema de pesquisa de Elias T. da Silva Jr. (SILVA JR., 2007). Tal *middleware* provê transparência no acesso a objetos, tanto no que tange a sua localização, quanto a sua natureza física de implementação (*software* ou *hardware*). Sob o ponto de vista do programador, o acesso se dá de maneira indistinta, por meio da utilização de métodos fornecidos através de uma API. São previstas duas formas de comunicação no *middleware*, através de memória compartilhada e de troca de mensagens. Uma

biblioteca, denominada *APICom*, fornece esse suporte para a comunicação através da implementação de um modelo de comunicação baseado em camadas. Uma vez implementadas primitivas de troca de mensagens, será possível realizar migrações de tarefas no sistema e, ainda, de migrações de objetos. Hoje já é possível abstrair a localização e alterar a natureza física de alguns dos objetos (do escalonador de tarefas, por exemplo) em tempo de execução, mas ainda não é possível migrar objetos em *software* de um nó para outro.

O desenvolvimento do *middleware* originalmente utilizava a plataforma de simulação CACO-PS (BECK, 2003a) na validação dos modelos. Uma vez validados são então desenvolvidos blocos VHDL dos objetos a serem implementados em *hardware*. A verificação final é realizada através da síntese e execução da plataforma em um FPGA.

Por sua vez, em seu trabalho Eduardo W. Brião (BRIÃO, 2007) (BRIÃO, 2008) propõe a exploração do espaço de projeto de plataformas embarcadas de forma dinâmica, assumindo que certas especificidades do comportamento dos sistemas só são conhecidas em tempo de execução. Assim a configuração da plataforma em termos de elementos de processamento e de localização dos processos poderá ser alterada ao longo da execução dos programas. As políticas de balanceamento de carga poderão ser alteradas de acordo com o cenário atual da plataforma, variando de acordo com a carga da bateria, por exemplo. Os trabalhos vêm sendo desenvolvidos sobre o simulador Serpens (WRONSKI, 2006). Esse simulador implementa em nível TLM (*Transaction Level Model*) modelos de processadores femtoJava e da rede SoCIN.

O modelo de migração de tarefas adotado (em *software*) será baseado nos modelos para migração de sistemas homogêneos, já que os processadores femtoJava possuem a mesma arquitetura para qualquer uma das implementações. Pelo mencionado, o mecanismo poderá ser dotado de um nível de transparência maior, não exigindo a utilização de pré-compiladores ou compiladores específicos. Até o momento (2008) a adoção de duas redes fisicamente separadas, uma de controle e outra de transferência de dados, como proposto por Nollet (2005a), não havia sido cogitada e estudos deverão ser feitos para avaliar a viabilidade energética dessa estratégia. Na migração do contexto espera-se que o fato de se utilizar um processador a pilha permita a transferência de mais carga útil em um menor número de *bytes*, já que as informações relevantes aos processos encontram-se na pilha do processador e não espalhadas na memória. Desse modo, as migrações transferirão um menor número de *bytes* e poderão ser amortizadas mais rapidamente, já que o tempo de amortização é função do tamanho do contexto, conforme demonstra Bertozzi (2006). Como em Nollet (2005a), o modelo de migração de objetos deverá suportar a alteração da natureza física dos mesmos; assim, além dos núcleos de femtoJava, haverá a presença de elementos reconfiguráveis no sistema. Uma abordagem semelhante a de Mignolet (2003) poderá ser utilizada.

Como produto final espera-se obter uma plataforma flexível, que possa ser utilizada em diferentes domínios de aplicação com alterações apenas através de reconfigurações em tempo de execução. Com isso, eventuais produtos desenvolvidos sobre a plataforma conseguiriam atingir um menor *time-to-market*, através da reutilização de componentes (de *software* e *hardware* reconfigurável) previamente desenvolvidos, e um menor preço, já que a escala de produção seria maior.

5 MODELO DE MIGRAÇÃO DE TAREFAS

5.1 Plataforma

Esta seção versará sobre as versões reais dos componentes modelados nos diferentes simuladores aqui descritos. Os componentes utilizados foram desenvolvidos em outros projetos do grupo e foram escolhidos para serem aqui utilizados pela facilidade de acesso aos códigos-fonte e de suporte. Existem modelos descritos em diferentes níveis de abstração dos processadores e da *rede-em-chip* apresentados. Curiosamente, foram implementados primeiramente modelos de mais baixo nível, como VHDL, para ambos, para só depois haver descrições mais abstratas, como CACO-PS e SystemC. Logo, embora não tenham sido desenvolvidas seguindo o fluxo de projeto ideal, a existência de descrições em diferentes níveis de abstração permite a exploração do espaço de projeto no nível de detalhamento desejado. Em etapas inicias, por exemplo, é possível avaliar a partir de modelos mais abstratos e menos precisos a tendência de comportamento do sistema para, em seguida, partir para modelos detalhados, mais acurados que permitam um levantamento preciso das características de desempenho e consumo dos mesmos. Neste capítulo, mais especificamente, serão apresentados o processador e a *rede-em-chip* utilizados na plataforma: o processador FemtoJava e a rede SoCIN, respectivamente.

5.1.1 Processador FemtoJava

No projeto do modelo de migração de tarefas, para a plataforma de simulação, foram utilizados processadores Java, do tipo FemtoJava, desenvolvidos localmente pelo grupo, que executam *bytecodes* diretamente em *hardware*. O estudo de arquiteturas Java é motivado pelo aumento de sua popularidade entre desenvolvedores de sistemas embarcados, principalmente devido a seu suporte à portabilidade. Os processadores utilizadas neste trabalho implementam apenas um subconjunto das instruções Java. Eles também não dão suporte à alocação dinâmica de objetos e não fazem uso de *garbage collector*. Conseqüentemente, todos os objetos são estaticamente alocados nas aplicações desenvolvidas.

O modelo utilizado é baseado no processador apresentado por Ito (2001). Ele possui uma arquitetura multiciclo, sendo que as instruções levam de 3 a 14 ciclos para serem processadas e sua organização de memória é do tipo *Harvard*, separando a memória de dados da memória de instruções. Numa definição mais ampla, no entanto, ele pode ser considerado uma máquina CISC (*Complex Instruction Set Computer*), já que suas

instruções possuem um tamanho variado e são por vezes complexas. Adicionalmente, todas as instruções são executadas utilizando uma pilha de dados, tal qual realizado pela JVM. Existem, no entanto, outros tipos de processadores femtoJava, entre eles, uma versão *pipeline* e uma VLIW com palavra de tamanho variável (BECK, 2004).

5.1.2 Rede SoCIN

A rede SoCIN foi originalmente apresentada em Zeferino (2004) como uma solução minimalista e parametrizável de *rede-em-chip*. Ela possui roteamento XY e chaveamento do tipo *wormhole* (DALLY, 1986), seu suporte original se limitava a redes do tipo grelha (*mesh*), mas é possível estendê-la para redes toróide sem alterações de *hardware*. Finalmente, o controle de fluxo é dado por meio de protocolos de negociação (*handshake*) e a arbitragem é realizada através de uma fila circular (*round-robin*).

O elemento principal da rede SoCIN é o roteador, denominado RASoC. Além da possibilidade de parametrização, o foco do desenvolvimento foi um baixo custo, tanto em termos de área quanto de consumo de energia. O roteador é composto pelos *buffers*, controladores de roteamento (R) e árbitros (A). Os canais do roteador são unidirecionais e cada porta dele possui dois deles – uma de entrada e outro de saída, possibilitando assim comunicação *full-duplex*. As portas, por padrão, são cinco: norte (N), sul (S), leste (E), oeste (W) e local (L); entretanto, dependendo da localização do roteador na rede é possível remover uma ou duas delas economizando ainda mais área e energia.

A arquitetura do roteador é implementada de maneira distribuída. Os controladores de roteamento operam de maneira paralela, lendo os pacotes dos *buffers* de entrada – não existem *buffers* de saída. Cada porta de entrada possui um controlador de roteamento e cada porta de saída possui um árbitro. Os controladores podem rotear os pacotes para uma mesma saída simultaneamente, cabe então ao árbitro da porta definir quem terá acesso à saída através da política de fila circular.

5.1.3 SASHIMI

O SASHIMI (*System as Software and Hardware in Microcontrollers*) (ITO, 2001) é um ambiente para desenvolvimento de plataformas embarcadas baseado nos processadores femtoJava. A partir da descrição em linguagem Java de uma aplicação, o *framework* é capaz de gerar automaticamente um processador otimizado para a mesma, além de alguns periféricos. A ferramenta permite uma rápida exploração do espaço de projeto permitindo ao projetista escolher se funções mais complexas, como divisão e multiplicação, serão implementadas em *hardware* ou *software*. É possível também escolher o tipo de processador femtoJava que será gerado (multiciclo ou *pipeline*) e a largura do barramento de dados (8, 16 ou 32 *bits*).

O SASHIMI recebe como entrada um programa Java compilado, no formato de um arquivo *.class*. A ferramenta, então, faz algumas alterações nesse arquivo a fim de torná-lo compatível com o subconjunto de instruções suportadas pelos processadores. Mais especificamente, instruções complexas são traduzidas para uma seqüência de instruções mais simples. Ao fim do fluxo do *framework* são geradas descrições VHDL do processador escolhido, das memórias de instruções e de dados e dos demais periféricos, além dos arquivos *.class* adaptados. A análise realizada pela ferramenta possibilita que ela gere processadores otimizados para a aplicação em questão. Dessa maneira tem-se a geração de ASIPs (*Application Specific Instruction Set Processor*) que suportam apenas o conjunto de instruções utilizado pela aplicação.

O SASHIMI conta também com uma biblioteca de classes, acessadas por meio de uma API, que implementam funções úteis ao projetista de sistemas embarcados. Através da biblioteca é possível, por exemplo, acessar as portas do processador e a memória diretamente, além de configurar os dispositivos mapeados em memória. A biblioteca também conta com escalonadores de tarefas e fornece acesso a todos os periféricos já desenvolvidos para a família de processadores como por exemplo *timers*, controlador CAN, interface serial, controle das interrupções, multiplicadores, divisores, entre outros. O *middleware* mencionado na Seção 4.3 e a API de comunicação por ele utilizada também fazem parte da biblioteca do ambiente de desenvolvimento.

5.2 Simuladores

Nesta seção serão apresentados os simuladores que, de alguma forma, foram utilizados no trabalho de pesquisa. Todos os simuladores aqui apresentados foram desenvolvidos pelo próprio grupo de pesquisa (*in-house*).

5.2.1 CACO-PS

Com o objetivo de obter-se medidas precisas de energia foi desenvolvido o simulador CACO-PS (*Cycle-Accurate COnfigurable Power Simulator*). Com precisão de tempo de relógio, tanto em relação ao desempenho quanto à energia, o simulador fornece uma análise acurada dos componentes simulados. Para a análise do consumo, o simulador baseia-se no número médio de chaveamento de *gates* de um componente. Soma-se a isso, a possibilidade de descrição estrutural ou comportamental de qualquer arquitetura, uma vez que novos componentes podem ser adicionados ao sistema.

O simulador CACO-PS foi desenvolvido na linguagem C, procurando-se obter alto desempenho na velocidade das simulações. A partir de uma biblioteca de componentes descritos também na linguagem C, é possível instanciar diferentes arquiteturas através da combinação desses componentes. A descrição da arquitetura é lida uma única vez no início da execução da simulação. Com isso obtém-se simulações com velocidade razoável, já que o comportamento dos componentes é simulado através da execução de código binário. O simulador CACO-PS já possui descritas diversas organizações de processadores femtoJava mas, originalmente, não fornece suporte à execução de sistemas multiprocessados. Não existem, também, descrições de meios de comunicação, sejam eles barramentos ou redes-em-chip.

Uma versão estendida do simulador foi criada para dar suporte a multiprocessamento. Essa versão executa várias instâncias do simulador que se comunicam por meio de *sockets*, ou, mais detalhadamente, comunicam-se através do protocolo TCP/IP. Com isso tornou-se possível a instanciação de diversos processadores femtoJava, e experimentos utilizando o protocolo CAN (do inglês, *Controller Area Network*) foram realizados. Contudo, o tempo de simulação começou a ter como gargalo a velocidade da comunicação TCP/IP, onerando ainda mais o desempenho do simulador.

O modelo de energia de cada componente é descrito através de uma função que recebe como parâmetros as suas entradas e saídas. Tal função deve retornar o consumo em termos de chaveamento de *gates*. Não existe forma rígida de como se dará o cálculo de chaveamento de *gates*, porém o autor recomenda a utilização dos valores de entrada e saída do componente a fim de depreender-se a taxa de bits chaveados e conseqüentemente de *gates*. Esse método é recomendado porque o chaveamento de

transistores é a fonte de dissipação da potência dinâmica dos componentes. Assim, para cada componente, sabendo-se a taxa de chaveamento, isto é, a quantidade de transições de suas entradas, pode-se calcular a quantidade de potência dinâmica dissipada, dada pela fórmula (RABAEY, 1996):

$$P_{dyn} = \alpha \, . \, C \, . \, f \, . \, V_{dd}^{2}$$

onde α é a taxa de chaveamento, C a capacitância de *gate* e f e V_{dd} a freqüência e a tensão de operação, respectivamente.

O simulador não dá suporte, entre os componentes básicos descritos na biblioteca, ao cálculo da potência estática, fato esse justificado em função de que, à época de seu desenvolvimento, a potência dinâmica ainda era responsável pela quase totalidade do consumo (KIM, 2003). Mais detalhes sobre o simulador CACO-PS podem ser encontrados em Beck (2003a) (2003b).

5.2.2 Serpens

A necessidade de simular tarefas e comunicação concomitantemente, associada à utilização de grafos de tarefas como descrição das mesmas e a possibilidade de avaliação de técnicas de gerenciamento de energia fez surgir o simulador Serpens. O simulador possibilita o uso de gerenciamento de energia (PM, do inglês, *Power Management*) e de regulagem dinâmica da tensão (DVS, do inglês, *Dynamic Voltage Scaling*), com isso é possível avaliar heurísticas de balanceamento de carga. O modelo do simulador é semelhante àquele utilizado em Hu (2004) a despeito de uma maior complexidade. De fato, foram adicionados custos de serviços do sistema operacional e o consumo das tarefas é função de tensão de operação dos núcleos no Serpens.

A linguagem de programação escolhida para a implementação do simulador foi SystemC em nível de transações (TLM, do inglês, *Transaction Level Model*), que oferece a abstração requerida e os mecanismos de sincronização de modelos de *hardware* necessários. SystemC vem se consolidando como uma linguagem para a descrição de sistemas de *hardware*, em função de seu suporte ao modelamento de atrasos, sinais, etc. Originalmente foram implementados uma estrutura de comunicação (um modelo da *rede-em-chip* SoCIN) e um processador (com custos baseados no femtoJava *pipeline*). O processador é responsável pelo processamento dos grafos de tarefas que forem nele alocados.

O modelo de energia dos componentes descritos é heterogêneo. O consumo de potência dinâmica dos roteadores e *links* da rede é calculado através da biblioteca Orion (WANG, 2002), a mesma forma utilizada no simulador XPipes (JALABERT, 2004). O simulador implementa um estimador de energia para a chave *crossbar* e para os *buffers* do roteador, sendo os últimos responsáveis por cerca de 90% do consumo do roteador. Os componentes descritos no roteador, além dos dois mencionados, contemplam ainda o árbitro do roteador. Considerando os mesmos e, de maneira similar à Ye (2002), a energia gasta para transferir um *phit* de dados entre dois roteadores é definida como:

$$E_{phit} = E_{wrt} + E_{arb} + E_{read} + E_{xb} + E_{link} \qquad [\mathrm{J}]$$

onde E_{wrt}, E_{arb}, E_{read}, E_{xb}, e E_{link} representam, respectivamente, a energia gasta na escrita de um *phit* no buffer de entrada, seleção do canal (árbitro), leitura do *phit* do buffer, chave *crossbar* e canal de saída.

O consume estático das memórias é estimado utilizando o modelo de Butts (2000):

$$E_{static} = \frac{V_{dd} \cdot I_{leak} \cdot k_{design} \cdot 6N_{bits}}{f} h \qquad [J]$$

onde V_{dd} é a tensão de operação, I_{leak} corresponde à corrente de fuga dos *gates* da célula de memória , a constante k_{design} é um parâmetro de tecnologia, N_{bits} é o número de *bits* na memória, η é o número de ciclos simulados e, finalmente, f é a freqüência de operação.

Originalmente, o simulador Serpens não considerava os custos de alocação das tarefas. As tarefas surgiam nos nós como se já estivessem presentes, mas apenas bloqueadas. Para os trabalhos envolvendo migração de tarefas o custo de alocação foi adicionado. Pressupôs-se que as tarefas encontravam-se em outro nó e passou-se a avaliar o custo de movimentar a tarefa através da rede. Mais detalhes sobre o simulador Serpens podem ser encontrados em Wronski (2006).

5.2.3 YAFJS

YAFJS é a sigla para a expressão em inglês *yet Another femtoJava Simulator*, que em um tradução livre significa, “mais um simulador de processadores femtoJava”, numa alusão às diferentes ferramentas que o grupo já possui. O simulador foi desenvolvido na linguagem SystemC. O projeto procurou reutilizar códigos de modelos de componente já existentes, descritos para os outros simuladores do grupo. Atualmente o simulador suporta a rede SoCIN e os processadores femtoJava Multiciclo e *Pipeline*.

O simulador foi desenvolvido de forma a ser rápido e escalável. A simulação dos processadores atualmente implementados, derivados do CACO-PS, ocorre em nível de ciclo de relógio, ou seja, é precisa, porém complexa. Já a NoC, derivada do simulador Serpens, como mencionado, foi desenvolvida em nível TLM, possuindo uma velocidade de simulação superior, embora seja menos fiel ao modelo original. A única topologia suportada atualmente pelo simulador é a do tipo grelha. A sua extensão para demais topologias requereria alterações no projeto do roteador. Apenas a alteração das conexões entre os enlaces da rede não seria suficiente para a obtenção de uma nova topologia funcional. A conexão dos enlaces dos roteadores laterais, por exemplo, a fim de formar uma topologia do tipo toróide, exigiria uma adaptação no protocolo XY implementado pela NoC.

5.2.3.1 Diferenças entre o modelo SoCIN implementado e o real

A principal diferença entre o modelo implentado e o especificado em Zeferino (2004) é a forma como se dá o endereçamento dos nós. No projeto original, o endereçamento se dá utilizando um *bit* para especificar o sentido do deslocamento e mais n bits indicando o número de *hops* a ser percorrido. A analogia com a representação sinal + magnitude de números binários, nesse caso, é perfeita. Já no modelo implementado, o endereçamento é em complemento de dois, onde os valores representam as coordenadas (x, y) do processador destino na rede.

Embora haja diferenças conceituais na forma de acesso aos dados do modelo TLM em relação à rede real, as métricas de energia e atraso utilizadas na simulação foram calibradas de acordo com o modelo real. O tamanho de um *flit* é parametrizável e pode adotar qualquer valor múltiplo de 8.

5.2.3.2 *Diferenças entre o modelo femtoJava implementado e o real*

Por ter sido descrito em nível RTL, o modelo do processador é mais fiel ao real do que a NoC. O modelo utilizado, derivado do CACO-PS, não suportava as instruções de acesso aos dispositivos de I/O mapeados em memória. Para corrigir essa deficiência, alterações foram realizadas no projeto original. O modelo, entretanto, oferece suporte a interrupções e a operação em modos de economia de energia.

5.2.3.3 *VNI - A Interface Virtual de Rede*

Para permitir a comunicação entre o processador e a NoC uma interface de rede foi criada. No entanto, como estão em níveis de abstração distintos, o mapeamento um a um dos valores utilizados pelo processador e pela NoC não pode ser realizado. Esse mapeamento foi, então, feito através de uma interface de rede virtual (VNI) que não modela atrasos. Trata-se de um *transactor* descrito em SystemC. A interface é responsável por ler os dados da porta do processador e passá-los para o formato de pacote utilizado pela NoC. Na visão do processador os pacotes são iguais aos que seriam utilizados em uma implementação física, já para a NoC são vistos como estrutura de dados complexa. O mapeamento realizado é apresentado na Figura 5.1. Os campos *msg* e *number* da estrutura não podem ser acessados por programas Java desenvolvidos para o processador, assim são utilizados apenas na depuração do sistema. O campo *payload* é composto por três bytes, cada um representando um *phit* de dados. BOP (*begin of packet*) e EOP (*end of packet*) são mapeados um a um, porém para o processador são *bits*, enquanto na estrutura são valores do tipo *bool*. Os bits *ack* e *val* não são modelados pelo nível TLM da NoC. Já os atrasos e o consumo são modelados utilizando como base a estrutura real da rede.

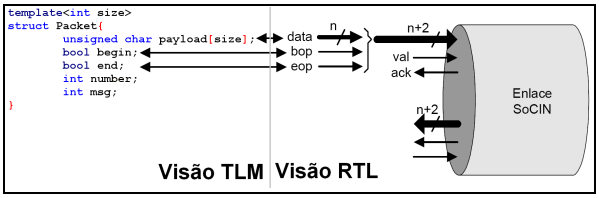

Figura 5.1: Mapeamento entre Níveis de Descrição

5.2.3.4 *Modelo de Portas*

Na implementação para o modelo de processador *pipeline* utilizou-se o mesmo endereço de porta do processador para escrita (envio) e leitura (recepção) de dados. Para o processador multiciclo, no entanto, optou-se por mapear entrada e saída em portas diferentes. Essa última escolha foi feita após averiguação, a fim de assegurar a compatibilidade entre o código gerado para o simulador e para a execução na plataforma prototipada. O mapeamento em uma única porta, de toda maneira, seria apenas uma visão lógica do processo, sendo que em uma implementação física os pinos de entrada e saída seriam distintos. Isso ocorre porque os canais da NoC são unidirecionais. No entanto, uma implementação diferente também poderia ser realizada e, nesse caso, a comunicação seria do tipo *half-duplex*, diminuindo a vazão da rede. Como a porta de comunicação do processador é mapeada em memória, uma implementação real poderia utilizar o sinal de leitura e escrita da memória, como controle de um buffer *tri-state*, para acessá-la. Obter-se-ia assim um mapa de memória diferente para leitura e escrita. Uma possível implementação desse mecanismo é mostrada na Figura 5.2.

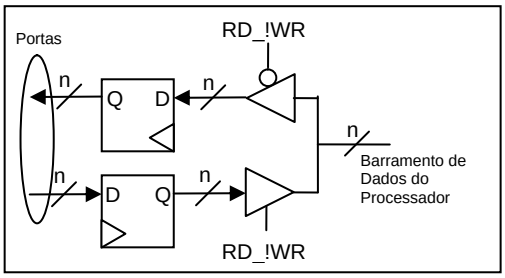

Figura 5.2: Possível implementação da Porta de E/S.

5.2.3.5 Protocolo de Comunicação

Para evitar a perda de dados durante o envio/recepção, foi criado um protocolo de comunicação. Esse protocolo permite que o femtoJava insira/retire mensagens da rede. Ele é baseado em *hand shaking*. Duas *flags* são utilizadas no protocolo, *acknowledgement* (ACK) e *new value* (VAL). A primeira verifica se o byte anterior foi consumido e a segunda informa da existência de um novo dado na porta.

A Figura 5.3a apresenta o fluxograma do protocolo para o envio de uma mensagem. Inicialmente o processador testa se o dado anterior foi consumido (ACK=0). Em caso afirmativo, ele escreve o novo dado na porta e informa o envio (VAL=1). Finalmente ele aguarda que a rede consuma o dado (ACK=1) e desliga o bit de envio (VAL=0).

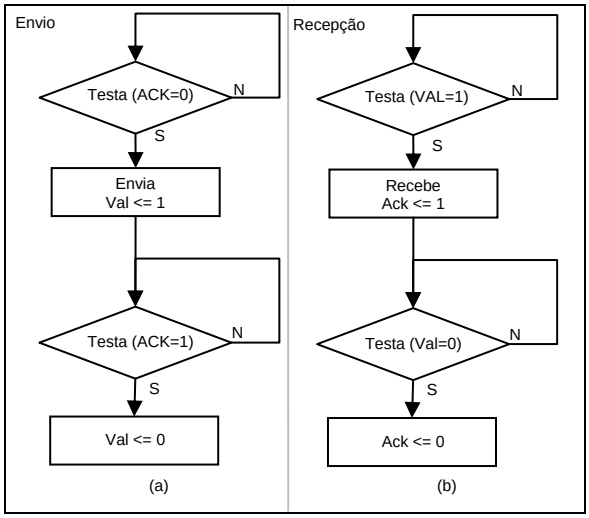

Figura 5.3: Fluxograma do Protocolo de Comunicação (a) Envio (b) Recepção.

A Figura 5.3b apresenta o fluxograma do protocolo para o recebimento de uma mensagem. Caso exista um novo dado na porta (VAL=1), então o dado é recebido e é informada a leitura do mesmo (ACK=1). Espera-se a confirmação do *acknowledgement* (VAL=0) e a seguir permite-se o envio de um novo dado. Nota-se que os protocolos de envio e recepção funcionam em conjunto. Para o envio de dados o *bit* 15 da porta de comunicação é utilizado como VAL e ACK, respectivamente na leitura e escrita da porta. Para recepção é utilizado o *bit* 14. Esse protocolo é compatível com o projeto da rede SoCIN, de modo que aplicações que operem com sucesso no simulador poderiam ser diretamente inseridas em um protótipo real do sistema.

5.2.3.6 Primitivas de Comunicação

Foi desenvolvida uma API para permitir a comunicação através da NoC. A API foi implementada na linguagem Java e possui quatro primitivas. Duas primitivas são do

tipo *Send*, e permitem enviar mensagens pela rede. As demais são do tipo *Receive*, e, por sua vez, permitem receber dados provenientes dos enlaces locais. A variação entre as primitivas de mesmo tipo está na forma de implementação, pois tanto s*ends* quanto r*eceives* podem ser, ou não, bloqueantes. Funções bloqueantes retornam ao usuário apenas quando a tarefa é concluída. Já funções não bloqueantes, caso não seja possível completar a operação solicitada, retornam imediatamente um código de erro. Adicionalmente, funções não bloqueantes exigem que a aplicação esteja apta a reconhecer e agir corretamente na ocorrência de um erro de envio ou recepção.

Normalmente, na utilização de primitivas de comunicação, faz-se uso de *sends* não bloqueantes e de *receives* bloqueantes. A primitiva bloqueante de recepção pode ser utilizada através de *polling*. Essa técnica, entretanto, não permite que o processador realize outra tarefa enquanto espera pelo recebimento de uma mensagem. Uma alternativa é fazer uso de interrupções, assim a cada vez que um pacote é recebido pela rede é disparada uma interrupção, desviando o fluxo de execução do processador para que esse possa tratar a mensagem recém chegada. Para situações em que apenas uma tarefa é executada no processador, a utilização de *polling* não prejudica a aplicação sob o aspecto do desempenho. Contudo, em termos energéticos, a utilização de interrupções tende a ser mais econômica por permitir que o processador aguarde por uma mensagem em estado de *sleep*. As figuras 5.4 e 5.5 apresentam o código das primitivas bloqueantes, de envio e recepção, respectivamente, da API implementada. A implementação das primitivas não bloqueantes é semelhante.

```
public static void send(int x_src, int y_src, int x_dest, int y_dest, int data)
{
    while((FemtoJavaIO.read(4) & 0x8000) != 0);
    FemtoJavaIO.write(0x8000 | x_dest, 8);
    while((FemtoJavaIO.read(4) & 0x8000) == 0);
    FemtoJavaIO.write(0x0, 8);

    while((FemtoJavaIO.read(4) & 0x8000) != 0);
    FemtoJavaIO.write(0x8000 | y_dest, 8);
    while((FemtoJavaIO.read(4) & 0x8000) == 0);
    FemtoJavaIO.write(0x0, 8);

    while((FemtoJavaIO.read(4) & 0x8000) != 0);
    FemtoJavaIO.write(0x8000 | x_src, 8);
    while((FemtoJavaIO.read(4) & 0x8000) == 0);
    FemtoJavaIO.write(0x0, 8);

    while((FemtoJavaIO.read(4) & 0x8000) != 0);
    FemtoJavaIO.write(0x8000 | y_src, 8);
    while((FemtoJavaIO.read(4) & 0x8000) == 0);
    FemtoJavaIO.write(0x0, 8);

    while((FemtoJavaIO.read(4) & 0x8000) != 0);
    FemtoJavaIO.write(0x8000 | data, 8);
    while((FemtoJavaIO.read(4) & 0x8000) == 0);
    FemtoJavaIO.write(0x0, 8);
}
```

Figura 5.4: Código da primitiva de envio bloqueante.

```
public static int receive()
{
    data_in = FemtoJavaIO.read(4);
    while((data_in & 0x4000) == 0)
        data_in = FemtoJavaIO.read(4);
    // Sobe o ACK
    FemtoJavaIO.write(data_in | 0x4000, 8);
    // Espera baixar o VAL da interface de rede
    while((FemtoJavaIO.read(4) & 0x4000) != 0);
    // Baixa o ACK
    FemtoJavaIO.write(data_in & 0xbfff, 8);
    return data_in;
}
```

Figura 5.5: Código da primitiva de recepção bloqueante.

Futuramente essas funções poderão ser incorporadas à API do próprio SASHIMI. Como se pode observar na figura, os *sends* e *receives* fazem uso das funções de baixo nível *read* e *write* da biblioteca *femtoJavaIO*. Esses métodos permitem, respectivamente, a leitura e a escrita das portas do microprocessador. São cinco os parâmetros das funções de envio (*send*). O par [x_src, y_src] representa o endereço do nó emissor da mensagem, e analogamente, o par [x_dest, y_dest] representa o endereço do nó destino da mensagem. No campo *data* informa-se o dado a ser transferido. Caso mais de um *byte* precise ser enviado, será necessário utilizar mais de uma mensagem para transferir a informação ao nó destino.

Funções que permitam o envio de mais de um *byte* poderão ser implementadas com base nas funcionalidades das primitivas já existentes. O valor de retorno de um *send* não bloqueante será 0 em caso de envio com sucesso e 1 no caso de falha. O valor de retorno de um *send* bloqueante deve ser ignorado. Por sua vez, a função *receive* não possui parâmetros e o valor de retorno é o próprio byte recebido. No caso de falha de um *receive* não bloqueante, será retornado o valor 0xFFFF. Não foi implementado suporte a *timeouts* nas primitivas bloqueantes.

5.2.3.7 Modelo de Energia

Quanto à NoC, uma vez que o modelo deriva do simulador Serpens a mesma maneira de estimar a energia é utilizada, através da biblioteca Orion. Para o processador, optou-se por uma abordagem diferente.

A partir da descrição em linguagem VHDL, utilizando a ferramenta *PowerCompiler* da *Synopsys*, verificou-se o consumo médio de cada instrução. Foi possível notar que o consumo das instruções varia principalmente com o número de ciclos da mesma e se ela realiza ou não acessos à memória. O impacto das entradas não é significativo no todo do consumo. A partir dos dados obtidos, o consumo dos processadores é estimado através do monitoramento das instruções executadas. Tem-se assim, um modelo de processador com precisão em nível de instrução para o consumo e de ciclo para o desempenho. Os dados são disponibilizados de maneira que é possível visualizar o consumo de cada método do programa Java executado e o consumo total até um determinado instante de tempo.

Tanto no que tange a *rede-em-chip* quanto os processadores, os parâmetros foram ajustados para corresponder ao consumo de circuitos fabricados com a tecnologia TSMC 0,18um (TSMC, 2007).

5.3 Proposta

Uma vez levantadas as características desejáveis e necessárias de um mecanismo de migração de tarefas para sistemas embarcados partiu-se para o projeto do mesmo. Optou-se por desenvolvê-lo prioritariamente através de rotinas de *software*, não apenas devido à flexibilidade fornecida, mas principalmente porque, levantados os custos do mesmo, sabe-se que a transição de qualquer ação do mecanismo para *hardware* traria benefícios de desempenho e energia na execução da tarefa. Obviamente não se sabe de que ordem de grandeza seria esse ganho, contudo, desenvolvendo o sistema em *software* tem-se uma curva *lower bound* de desempenho e energia, conhece-se assim o pior caso e podem-se realizar as melhorias necessárias de acordo com os requisitos do sistema. Com todo o mecanismo desenvolvido em *software*, uma vez calculados os tempos de amortização de desempenho e energia, sabe-se que qualquer esforço de programação acarretará em melhorias ao sistema.

O mecanismo de migração desenvolvido obedece ao algoritmo apresentado na Seção 2.3, embora não implemente todas as etapas. O foco do desenvolvimento foi o custo energético do mesmo, contudo, ao contrário dos demais mecanismos para sistemas embarcados já apresentados (conforme exposto no Capítulo 4), optou-se, como decisão de projeto, por fornecer uma maior transparência ao modelo, aumentando o nível de abstração com que o programador lida com o mesmo. O aumento da transparência impõe uma perda de desempenho e em termos energéticos ao sistema (Seção 3.1.2). Entretanto, a definição de pontos de migração e dos dados a serem migrados aumenta o tempo de projeto e pode acarretar comportamentos inesperados caso esses não sejam escolhidos adequadamente. Considerou-se, também, que a programação de aplicações que explorem a possibilidade da utilização de diversos processadores concomitantemente já é, por si só, um desafio suficiente para o desenvolvedor (vide a dificuldade em encontrar *benchmarks* adequados para o domínio de sistemas embarcados paralelos e distribuídos). Tirou-se assim, mais uma responsabilidade do programador, deixando-a a cargo do sistema operacional ou do *middleware* da plataforma.

A flexibilidade, visto o desenvolvimento realizado em *software*, também foi um fator considerado no desenvolvimento do mecanismo de migração de tarefas objeto de trabalho do presente estudo. Em vista disso, embora não encorajado, é possível utilizar técnicas de *check-pointing* no mecanismo. A migração de tarefas pode ser iniciada pelo sistema operacional ou pela aplicação, dessa segunda maneira é o programador que define em que pontos a migração ocorrerá. É possível associar a isso uma análise de carga do sistema, realizando a migração somente se necessária. Caso a migração seja iniciada pelo sistema, ele próprio, em função de alguma métrica, definirá quando executar a migração. Não foram implementadas heurísticas de balanceamento de carga, que visem um aumento do desempenho ou diminuição do consumo do sistema; tampouco os serviços que dariam acesso às informações necessárias para a aplicação das mesmas. Esses trabalhos são objetos de estudo de outros membros do Grupo e estão fora do escopo dessa dissertação. Muito embora o autor tenha dado suporte a outras pesquisas desenvolvendo os algoritmos de *bin-packing* explorados por Brião (2007) em linguagem Java, tais algoritmos não serão aqui apresentados.

5.3.1 Arquitetura

Embora relativamente genérico, o modelo desenvolvido utiliza certas características dos processadores e da rede a fim de aumentar o desempenho da migração de tarefas.

Logo, apesar de ser desenvolvido em *software* o mecanismo leva em consideração características do *hardware* utilizado.

A migração de tarefas divide-se em três etapas principais: a migração do código, a migração dos objetos e a migração dos dados da pilha do processador.

O código fica armazenado na memória de instruções, organizado seqüencialmente e orientado a byte. A cópia é realizada byte a byte, a partir de uma interface, denominada *MigrateInterface*, implementada para fornecer acesso direto à memória, uma vez que a linguagem Java nativamente abstrai tudo o que diz respeito às entradas e saídas do sistema. O cabeçalho das funções implementadas com o intuito de prover acesso direto às memórias é apresentado na Figura 5.6.

```
public interface MigrateInterface {

    public int readMEM(int array, int index);
    public void writeMEM(int array, int index, int integer);

}
```

Figura 5.6: Cabeçalho das funções de entrada e saída na memória

Ambas as funções apresentadas utilizam um sistema de endereçamento de base mais deslocamento, sendo o valor da base que define se serão copiados dados ou código. Respectivamente, as funções *readMEM* e *writeMEM* são responsáveis por ler e escrever nas memórias. Durante a síntese, pós-compilação, as funções dessa interface são traduzidas para *bytecodes*. Notadamente elas não possuem código Java específico, sendo de fato apenas uma API que fornece acesso a métodos descritos em linguagem nativa. Uma vez os parâmetros carregados na pilha, as funções de leitura e escrita são traduzidas, nessa ordem, para as instruções *iaload* e *iastore*. Para fins de simulação, uma biblioteca denominada *FemtoJavaMigrate* implementa a funcionalidade dos métodos da interface. Logo, a codificação é realizada utilizando os métodos dessa biblioteca.

Adicionalmente, para se ter acesso ao código, foi necessário modificar o mapa de memória do processador já que não havia zona de endereçamento reservada para o acesso à memória de instruções. Originalmente, a memória de instruções só era acessada pelo processador na busca de instruções, não podendo ser acessada como memória de dados. A memória de instruções foi mapeada a partir do endereço 0x10000, logo após o fim da memória de dados, que, dada a origem 16 bits do processador femtoJava, endereça apenas 64kB endereços. O novo mapa de memória é mostrado na Figura 5.7. Na prática, a memória de instruções e a área de endereçamento na memória de dados atuam como sombras de memória uma da outra.

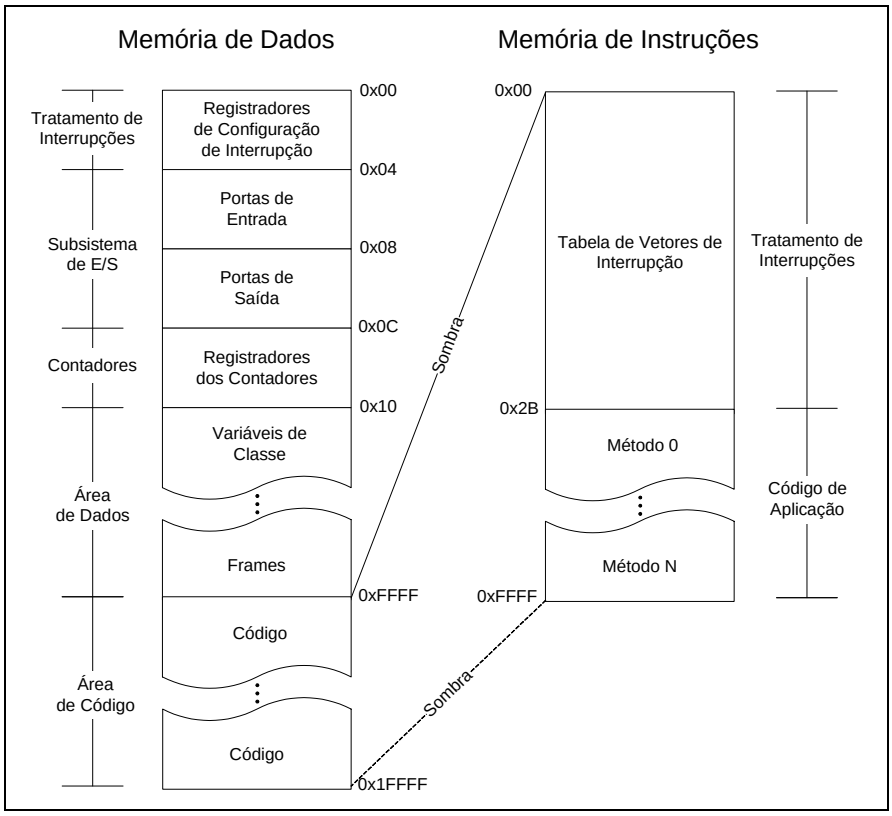

Figura 5.7: Novo mapa de memória de dados dos processadores

A alteração realizada tornou a memória de instruções acessível para leitura e escrita, o que provocou mudanças na sua organização. A memória de instruções deixou de ser uma memória ROM (*Read Only Memory*, do inglês, memória somente leitura) e passou a ser uma memória RAM (*Random Access Memory*, do inglês, memória de acesso aleatório) com duas portas para a leitura e uma para a escrita. A segunda porta de leitura não é estritamente necessária, uma vez que no processador multiciclo não há acessos à memória de instruções sem ser no estágio de busca das mesmas e nesse estágio não há tratamento de dados. a utilização de uma nova porta, entretanto, facilita a lógica de controle e deixa pronta a organização de memória para a utilização eventual de um processador ***pipeline***. A nova organização, porém, foi validada apenas na descrição em SystemC, não tendo sido feita a implementação em VHDL da solução.

Apesar das alterações possibilitarem o tratamento do código como dados comuns restou algo que pode ser considerado uma inconsistência no sistema: enquanto os dados são orientados a palavra (4 bytes), o código é orientado a byte. Com isso, embora tenham o mesmo número de endereços, a zona de dados contém 256kB e a de código apenas 64kB. Operações de escrita que tentarem escrever mais do que um byte na área de código terão seu valor truncado para 8 bits. Nenhum estudo de adequação dessa situação foi realizado, visto que na prática o fato não se apresenta como um problema físico de implementação.

Os objetos encontram-se na memória de dados, distribuídos aleatoriamente, sendo cada um deles composto por um identificador de tipo, atributos e referências para objetos filho. Os métodos dos objetos encontram-se na memória de instruções e são copiados na etapa anterior. Como o processador femtoJava não possui nenhum tipo de gerenciamento de memória, é necessário que o programador defina em tempo de projeto (pós-compilação) quais trechos da memória de dados necessitam ser copiados. Outra

solução possível é copiar toda a memória de dados, ou, mais eficientemente, copiar apenas a parte utilizada da memória de dados. Seja qual for o método escolhido pelo programador, o acesso à memória será realizado através das funções ***readMEM*** e ***writeMEM*** apresentadas, utilizando como base o endereço 0x0.

A pilha armazena todas as variáveis de método, os endereços de retorno e realiza a função dos registradores dos processadores convencionais, armazenando os dados temporários utilizados em operações aritméticas, lógicas, etc. Nos processadores femtoJava a pilha inicia no fim da zona de endereçamento de dados (0xFFFF) e cresce em direção ao início da mesma conforme a Figura 5.8. A posição atual da pilha pode ser obtida pela instrução ***save_ctx*** que retorna um valor inteiro contendo o endereço atual do ***Stack Pointer*** (SP, do inglês, ponteiro de pilha) do processador. Com isso não é necessário copiar nenhum dado adicional, pois são conhecidos os endereços inicial e final da pilha de dados. A instrução ***save_ctx*** não existe originalmente na linguagem Java, contudo, a última possui ***bytecodes*** livres; isso foi pensado para que programadores pudessem justamente estender o conjunto de instruções conforme necessidades específicas, tal como foi realizado.

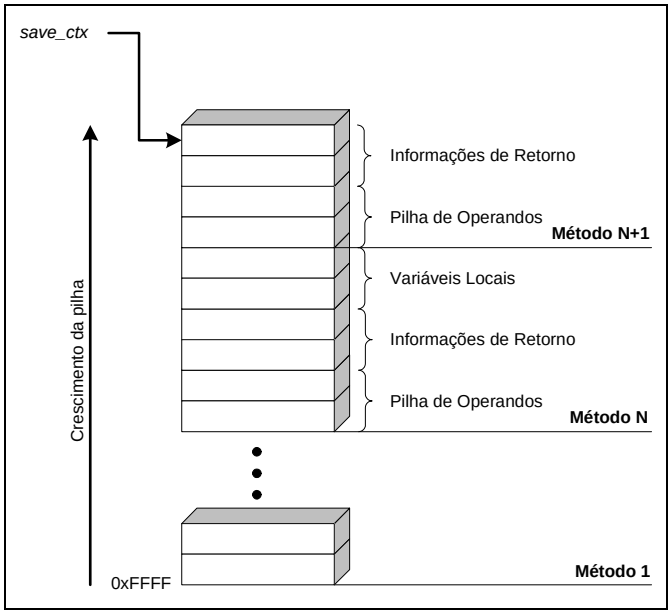

Figura 5.8: Pilha de Operandos

Todo o exposto até o momento nesta seção refere-se ao nó que está enviando os dados. Mesmo que análoga, a recepção da tarefa possui algumas diferenças. Tanto dados, quanto código são escritos na memória através da função ***writeMEM*** e o SP é restaurado pela função ***restore_ctx***, que armazena no registrador a posição atual da pilha.

Com isso, tem-se a migração de uma tarefa. No entanto, tal como descrito, cada processador poderia contar com apenas uma única tarefa. Essa abordagem é interessante para o levantamento de resultados, pois já permitiu avaliar os custos do processo, embora praticamente não possua maior aplicação prática. Não possui aplicação porque não faz sentido migrar uma tarefa que roda sozinha em um processador para outro processador ocioso, pois não haverá nenhum tipo de amortização. Isso, evidentemente, caso os dois processadores sejam idênticos. Para processadores diferentes, mas com o mesmo conjunto de instruções, tal abordagem ainda pode encontrar alguma aplicação, executando tarefas ora em um ora em outro núcleo. É mais comum, porém, encontrar

processadores que executam diversas aplicações e realizar balanceamento de carga através de migrações de tarefas sobre esse tipo de ambiente. Mesmo no domínio dos sistemas embarcados, sistemas multitarefa estão se tornando corriqueiros. Assim sendo, implementou-se uma solução a ser utilizada em sistemas multitarefa que será objeto de análise da próxima seção.

5.3.2 Modelo

A família de processadores não dá suporte à multitarefa em *hardware*. Não existe implementação de memória virtual, de paginação ou mesmo de *threads* em *hardware*. A solução encontrada por Wehrmeister (2005), na sua proposta de API de tempo-real, foi o desenvolvimento de *threads* em *software*. Tais *threads*, denominadas *RealtimeThreads*, foram definidas pela RTSJ (*Real-Time Specification for Java*, do inglês, Especificação de Tempo-Real para Java) e são escalonadas pela JVM respeitando requisitos temporais. Por se tratar de uma linguagem orientada a objetos, a *RealtimeThread* é uma classe a partir da qual se declaram objetos para, esses sim, serem escalonados pela máquina Java.

A granularidade do modelo de migração de tarefas passou então a ser de *threads*. Logo, a partir de agora, ao se referir a uma tarefa está se referindo a uma *thread* Java. Solucionado o problema de fornecer suporte multitarefa, através da emulação em *software* de *multithread*, e mantida a arquitetura do modelo de migração apresentada anteriormente, partiu-se para a implantação do modelo.

Cada processador pode possuir, em tempo de execução, um número arbitrário de tarefas, cujo número máximo é conhecido, já que não existe alocação dinâmica de memória. Existem versões da API de tempo-real que dão suporte a até 32 tarefas. As tarefas são executadas no processador por meio de um escalonador, acionado pelo *timer* do núcleo. O escalonador é responsável por definir qual tarefa ganhará o processador, de acordo com os requisitos das mesmas. A partir de uma classe básica, denominada *Scheduler*, diferentes versões de escalonador já foram desenvolvidas, tais como um escalonador de prioridades e um escalonador EDF (*Earliest Deadline First*).

Cada *thread* possui uma área de código, objetos e pilha independentes. A pilha original torna-se assim a pilha do sistema, aquela utilizada pelo escalonador, enquanto as demais distam 2kB uma da outra conforme a Figura 5.9. Conforme descrito, a instrução *save_ctx* salva a posição atual da pilha, enquanto *restore_ctx* restaura o SP. Como o último valor na pilha é o endereço de retorno da interrupção que chamou o escalonador, para escalonar uma *thread* novamente basta reposicionar o SP de maneira adequada e efetuar um retorno de método através da instrução *return*. Todo esse processo fica mascarado para o programador, em função da implementação com alto nível de abstração da classe *RealtimeThread*, da qual pode-se encontrar mais detalhes em Wehrmeister (2005).

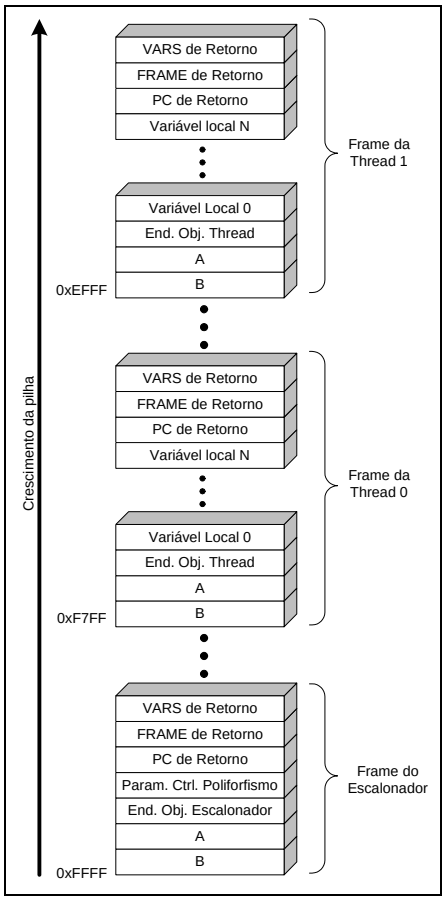

Figura 5.9: Organização das pilhas do sistema e das *threads*

A classe *RealtimeThread* foi estendida, dando origem à classe *RealtimeThreadEx* (de *extended*, do inglês, estendida). A nova classe herda todas as características da original e implementa um novo método que fornece o suporte necessário à implementação da migração de tarefas. Tal método, denominado *addToMigrateQueue*, adiciona a *thread* a uma fila de tarefas que estão esperando para serem migradas. Para fornecer suporte à migração de tarefas, os escalonadores também teriam que ser estendidos, sendo que a herança inerente à programação de objetos em Java garantiria o funcionamento original dos mesmos. Os métodos, propriedades e estruturas necessárias seriam adicionados. No entanto, optou-se por desenvolver um novo escalonador, que comprova o método, ficando a extensão dos demais como trabalhos futuros.

A classe *GenericScheduler* é uma extensão da classe padrão *Scheduler* e implementa adicionalmente o método *addToMigrateQueue* tornando o escalonador capaz de gerenciar a fila de migração. É declarada, ainda, uma lista de tarefas, onde ficam as referências para as *threads* a serem migradas, de maneira análoga à lista de escalonamento, e um contador para que se saiba quantas tarefas encontram-se na lista.

Adicionalmente, a classe *GenericScheduler* implementa uma funcionalidade ainda não presente na biblioteca de funções pré-existentes, um escalonador baseado em fatias de tempo (*timeslice)*. A não existência de tal funcionalidade deve-se mais ao fato do desenvolvimento dos recursos até o momento ser voltado para tempo-real e tarefas periódicas, e não para aplicações de propósitos gerais, do que à necessidade de um grande esforço de programação no intuito de implantá-la. O escalonador utiliza uma fila circular (*round-robin*) em que cada tarefa permanece um tempo definido (*timeslice*)

sendo processada. Tal tempo é definido na criação do objeto, através de um parâmetro do método construtor. A Figura 5.10 ilustra o diagrama de classes das classes implementadas e as relações entre as mesmas. Os métodos *isFeasible* e *runScheduler* devem ser obrigatoriamente implementados e se referem, respectivamente, ao teste se uma nova tarefa pode ser escalonada e ao comportamento do escalonador propriamente dito. Para o caso da classe *GenericScheduler*, o teste retorna verdadeiro desde que não se tenha atingido o número máximo de tarefas.

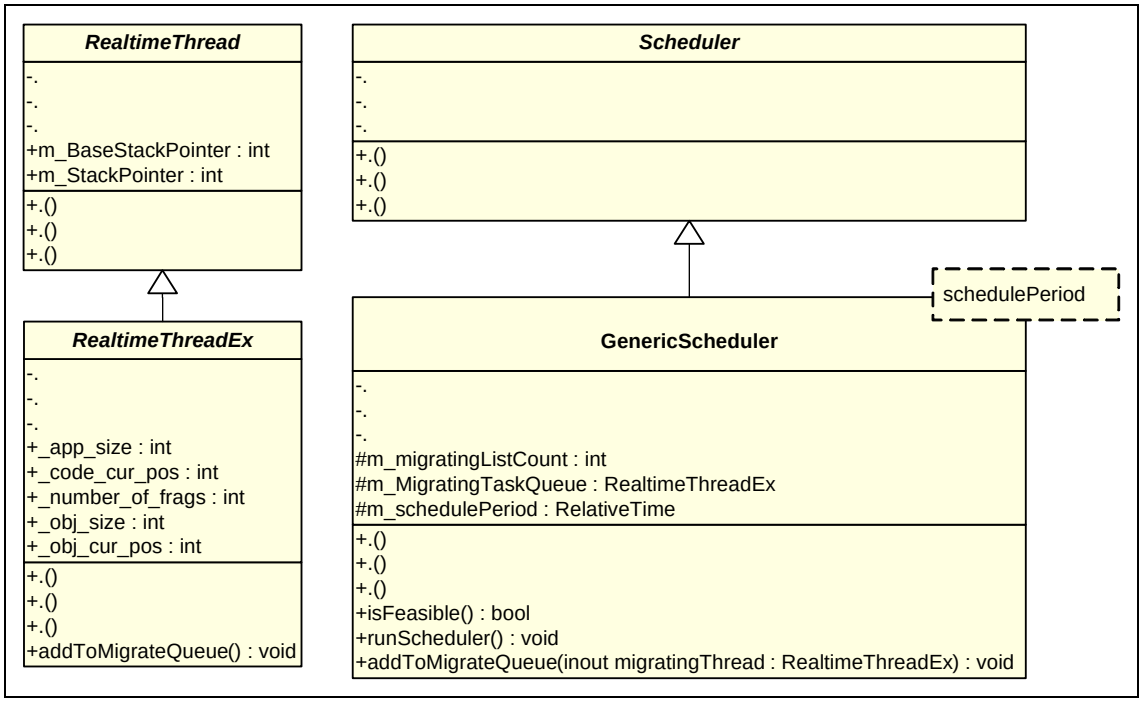

Figura 5.10: Diagrama de classes (*RealtimethreadEx* e *GenericScheduler*).

Uma vez implementadas as classes, o programador deve estender a classe *RealtimeThreadEx* e então definir o comportamento da tarefa, tal qual era realizado na classe *RealtimeThread* como definido por Wehrmeister (2005). Um objeto da classe implementada deve então ser instanciado e adicionado à fila de escalonamento do *GenericScheduler*.

Na ocasião de uma migração de tarefa, a mesma deve ser removida da fila de escalonamento através do método original da classe *RealtimeThread removeFromFeasibility*. A suspensão da execução da tarefa pode ser realizada antes da cópia do código, em uma abordagem do tipo cópia-total, ou após a cópia do mesmo, aproximando-se de uma solução do tipo pré-cópia. Transferências sob-demanda não foram implementadas, pois exigiriam mecanismos de gerência de memória; da mesma maneira, soluções do tipo servidor ainda exigiriam uma zona de memória compartilhada, não existente na plataforma utilizada. A implementação de memória compartilhada na plataforma é um dos trabalhos atuais do Grupo. Soluções próximas de servidor poderão ser realizadas a partir de então e poderão ser exploradas no futuro.

Os atributos *_app_size*, *_code_cur_pos*, *_number_of_frags*, *_obj_size*, *_obj_cur_pos* da classe *RealtimeThreadEx* contêm informações relevantes à migração de tarefas, correspondendo, respectivamente, ao tamanho do código da tarefa, à posição inicial do código na memória de instruções, ao número de fragmentos de memória não contínuos onde encontram-se objetos sensíveis à *thread* e ao tamanho e posição dos objetos na

memória de dados. Os parâmetros *_obj_size* e *_obj_cur_pos* são vetores de inteiros com dimensão igual ao valor de *_number_of_frags*.

A implementação se dá de tal forma porque, como já referido, os objetos não necessariamente encontram-se continuamente armazenados na memória. Os valores dessas propriedades são ajustados após a compilação da aplicação, através da análise do código *assembly* gerado. Atualmente esse processo se dá de forma manual, mas deverá ser inserido dentro do fluxo SASHIMI. As informações referentes à pilha são obtidas em tempo de execução, já que o seu tamanho varia ao longo da computação da tarefa. Os parâmetros *m_BaseStackPointer* e *m_StackPointer*, originariamente já presentes na classe *RealtimeThread*, contêm as informações sobre a pilha. O primeiro parâmetro informa a posição inicial da pilha e o segundo a posição atual da mesma, sendo que a partir da subtração desses valores é possível obter o tamanho da pilha. O parâmetro *m_BaseStackPointer* é preenchido no início da execução da *thread* e o *m_StackPointer* armazena a informação retornada pela instrução *save_ctx*, através do método *saveContext*, da classe *RealtimeThread*. Tal informação é atualizada toda vez que a tarefa é removida da execução pelo escalonador.

Para a transferência das tarefas quando em forma de *threads*, alguns serviços foram implementados. Tais serviços podem ser vistos como parte de um pequeno núcleo de sistema operacional distribuído ou de um *middleware*. Tais serviços devem estar presentes em todos os nós que desejem enviar ou receber tarefas. Na prática, todos os nós devem possuí-los, já que com os recursos do mecanismo de migração de tarefas é possível também a alocação dinâmica de tarefas. Se um nó executa sempre as mesmas tarefas e tais tarefas foram definidas em tempo de projeto, então não é requerido que os serviços estejam nele presentes, economizando memória, tanto de dados quanto de instruções.

Para o recebimento de uma tarefa, os serviços atuam atrelados à interrupção serial, responsável pelo recebimento das mensagens provenientes da rede. São três os serviços de recebimento: de código, do objeto e da pilha. Os três serviços possuem uma estrutura semelhante: após a definição do serviço a ser utilizado (através de uma mensagem com um identificador numérico de 1 byte), o primeiro campo define o número de bytes ou palavras a serem gravados na memória e o campo seguinte a posição inicial da gravação; ambos os campos são inteiros de 2 bytes. Em seguida, pode-se então transferir a carga útil do serviço. A cópia do código é orientada a byte, enquanto que a dos objetos e da pilha a palavras de 4 bytes. Como as primitivas de comunicação são também orientadas a byte, é necessário para os objetos e para a pilha remontar os dados antes de escrevê-los na posição correta. O *byte* mais significativo de cada palavra é enviado (e recebido, conseqüentemente) primeiro, passa-se então pelos demais *bytes*, até o último, o menos significativo da palavra. Como explicitado, o tamanho da carga útil pode ser dado em bytes, para o código, ou palavras, para os objetos e para a pilha. Assim, para o código se o campo de tamanho possuir valor N serão transmitidos N bytes, já para os objetos e a pilha serão transmitidos 4N bytes. Após a gravação de cada byte ou palavra a posição de gravação é incrementada em uma unidade, como esperado.

O primeiro serviço a ser chamado, normalmente, é o serviço da transferência do código, uma vez que entre cópia-total e pré-cópia é preferível usar o segundo método já que ele não impõe nenhuma sobrecarga (*overhead*) ao sistema. A cada byte recebido, o código é gravado na memória de instruções (*on-the-fly*) e os contadores de controle atualizados. Ao fim da carga útil, o sistema passa a esperar a requisição de um novo serviço.

O segundo serviço invocado, envolvido na migração de tarefas, diz respeito à transferência dos objetos relacionados à mesma. Esse serviço é invocado um número arbitrário de vezes, de acordo com o número de segmentos que compõem os objetos da tarefa. Da maneira como foi implementado, o objeto ***RealtimeThreadEx***, ou seja, a tarefa propriamente dita ou o objeto topo, precisa ser o último a ser transferido a fim de que o ponteiro de referência para o mesmo, que também é usado como identificador da tarefa, seja atribuído corretamente.

O último serviço trata da transferência da pilha. Os campos de tamanho e posição inicial do serviço servem para ajustar os atributos ***m_BaseStackPointer*** e ***m_StackPointer*** da ***thread***. Após a cópia na memória dos dados da pilha, a tarefa é automaticamente adicionada à fila de escalonamento e reiniciada. Adicionalmente, caso o processador esteja ocioso, o ***timer*** que controla o escalonador local é disparado novamente. O código final do serviço, isto é, das operações descritas anteriormente neste parágrafo, é mostrado na Figura 5.11.

```
// Após receber todos os dados e o código referente a thread, coloca-a na fila de escalonamento,
// A thread é identificada pela posição de seu objeto na memória RAM (_obj_cur_pos).

((RealtimeThreadEx) FemtoJavaMigrate.getThread(_obj_cur_pos[0])).m_BaseStackPointer = _stack_cur_pos + _stack_size;
((RealtimeThreadEx) FemtoJavaMigrate.getThread(_obj_cur_pos[0])).m_StackPointer = _stack_cur_pos;
((RealtimeThreadEx) FemtoJavaMigrate.getThread(_obj_cur_pos[0])).addToFeasibility();
((RealtimeThreadEx) FemtoJavaMigrate.getThread(_obj_cur_pos[0])).start();
if(sched.isAllTasksFinished())        // Evita ativar o timer duas vezes e garante que se não houver nenhuma tarefa
            sched.setupTimer();       // ele ligará o escalonador
```

Figura 5.11: Código de reinicialização da tarefa no nó destino

O método ***getThread*** da biblioteca ***FemtoJavaMigrate*** mostrado na figura anterior foi criado devido ao fato da linguagem Java ser fortemente tipada. A posição inicial do objeto é também a referência à tarefa, contudo todos os dados recebidos da rede pela rotina de tratamento de interrupção são tratados como inteiros. Assim, é preciso converter o tipo da variável em questão, que mesmo contendo o valor tratado não pode ser utilizado como valor referência da tarefa. A linguagem Java não permite uma conversão (***cast***) automática entre os tipos ***int*** e ***RealtimeThreadEx***. Em vista disso, foi criado um método para converter o tipo ***int*** para o tipo ***Object***, sendo que esse último pode ser tratado como uma ***thread***. Na hora da tradução para o código nativo Java, o método ***getThread*** não é traduzido, simplesmente o valor inteiro passado como parâmetro é deixado no topo de pilha, sendo conseqüentemente usado como valor de retorno do método. Essa abordagem foi necessária, embora vá contra os princípios da linguagem Java, apesar disso, o programador do sistema opera em um nível alto de abstração, não se envolvendo com esse tipo de construção. Após a transferência da pilha, a migração está concluída e o sistema passa a operar com a nova configuração de alocação de tarefas.

Foram testadas duas maneiras de implementar a migração de tarefas: uma delas usando o próprio escalonador do sistema e a outra utilizando um ***thread*** dedicada ao processo de migração. O primeiro método garante maior previsibilidade, uma vez que o escalonador não é preemptado e sempre conclui as migrações da fila antes de escalonar a próxima tarefa. Caso o meio garanta a entrega das mensagens relativas à migração (o que não era o caso nos experimentos) e utilize-se o escalonador como gerente do processo, pode-se dizer que este é um sistema de migração de tarefas de tempo-real. Com o uso de uma ***thread*** para o processo, ganha-se em flexibilidade mas a migração de tarefas é normalmente mais demorada, uma vez que, normalmente, são necessários diversos ciclos de execução da ***thread*** para migrar uma única tarefa. A principal vantagem em se utilizar uma ***thread*** é o fato do sistema não ficar bloqueado na

migração e, no caso da utilização de um escalonador de tempo-real, a garantia de não se perder nenhum *deadline* (desde que, claro, a totalidade das tarefas seja escalonável). Para o uso da *thread* como gerente das migrações foi necessário tornar pública a lista de tarefas a serem migradas, sendo essa inicialmente protegida e acessível apenas pelo escalonador. A fila de tarefas é sempre mantida pelo escalonador, uma vez que as tarefas não sabem qual *thread* é responsável pelo escalonamento.

Uma característica inerente ao modelo é o respeito ao escalonamento em tempo-real das tarefas, no caso da utilização de um escalonador desenvolvido para esse fim. Isso significa que, se uma tarefa tem período de 100ms e a migração ocorre em um tempo menor que esse, a tarefa no novo nó será escalonada em tempo hábil para que não perca o seu *deadline*. Obviamente, ambos os nós devem utilizar escalonadores de tempo-real, mas não é necessário que operem com o mesmo *clock*, apenas o relógio de tempo-real (RTC, do inglês, *Real-time Clock*) de cada nó deve ter sido disparado ao mesmo tempo.

5.3.3 Limitações

O modelo desenvolvido, não obstante, possui algumas limitações. O gerenciamento de recursos não se faz presente, uma vez que a plataforma não oferece suporte a arquivos, *pipes* e outros conceitos como esses. Mesmo se uma camada realiza a abstração da localização dos objetos na memória local, trata-se de um sistema distribuído. Assim, durante o processo de migração a consistência das mensagens precisa ser garantida, recurso esse que ainda não foi implementado.

Além disso, a família de processadores femtoJava não possui unidades de gerenciamento de memória (MMU, do inglês, *Memory Management Unit*), logo, é preciso administrar manualmente as cópias de memória, ajustando as referências a objetos e métodos ou, no caso dos métodos, utilizando um código independente de posição (PIC, do inglês, *position independent code*). A utilização de um PIC é inviabilizada pelo fato de que, tal qual na JVM, nos processadores femtoJava todas chamadas de sub-rotinas, assim como a referência a objetos, são feitas pelas suas localizações absolutas. Até o presente momento nenhum mecanismo para contornar esse problema foi desenvolvido; conseqüentemente, é necessário garantir manualmente que toda a área de memória ocupada pela *thread* (tanto de instruções quanto de dados) esteja livre no nó destino para poder ser usada pela nova tarefa. Outra alternativa é fazer com que as tarefas existam em todos os nós. Nesse caso, copiar-se-ia apenas os objetos e a pilha durante a migração e esses valores sobrescreveriam as áreas originais. Segundo Pittau (2007), essa é uma limitação aceitável, uma vez que boa parte dos sistemas operacionais embarcados não suporta código relocável. Finalmente, qualquer referência a variáveis globais também não é suportada, mas essa não é uma limitação séria, uma vez que o paradigma da programação orientada a objetos desencoraja tal prática (o uso de variáveis globais).

Das limitações apontadas duas merecem algumas considerações extras: a realocação de código e a consistência de mensagens. Alternativas ao uso de PIC, no que tange o código, passam pela alteração do *hardware* com a implementação de uma pequena MMU, ou, simplesmente, de um registrador de base, que seria usado em todas as referências à memória. Outra solução seria a análise do código no momento da cópia; contudo, apenas as chamadas a sub-rotinas seriam facilmente identificáveis, uma vez que as instruções de invocação de métodos (*invokevirtual* e *invokestatic*) são bem definidas. Quanto às referências a objetos, não há como saber, através da análise do código, se trata-se de uma referência ou de uma manipulação numérica simplesmente. O

mesmo ocorre com a parte dos dados (objetos e pilhas): não há identificação para referências ou mesmo alguma diferenciação do que são valores e o que são ponteiros.

Quanto à consistência de mensagens, estuda-se a implantação de um sistema semelhante ao de Nollet (2005a), tal qual descrito na Seção 4.1.2, associado à API de comunicação apresentada em (SILVA JR., 2007). Com o sistema de consistência de mensagens seria garantida a entrega correta daquelas que fossem recebidas durante a migração e o uso da API forneceria a abstração desejada no envio e recebimento das mesmas. Atualmente, cada *thread* é mapeada para uma porta de comunicação na API, em um esquema semelhante ao TCP/IP, onde um endereço é formado pelo conjunto das informações que identificam o nó e a *thread* propriamente dita. Originalmente essa informação era fixa, mas está sendo desenvolvido um serviço que atualize os dados dessa tabela em tempo de execução.

6 ESTUDOS DE CASO

Este capítulo apresenta três experimentos. O primeiro foi desenvolvido ainda na fase de escolha da plataforma que seria utilizada e demonstra uma interessante maneira de tornar migrações de tarefas menos onerosas e mais facilmente amortizadas tanto em termos de energia quanto de tempo. Nesse experimento foram avaliadas três diferentes organizações de memória, sempre considerando redes-em-chip como meio de comunicação.

Os dois outros experimentos buscam a validar o modelo desenvolvido. Um dos experimentos avalia o impacto em termos de desempenho do modelo. Em função do tamanho da tarefa é verificado o tempo de migração e são identificadas e analisadas as parcelas de tempo gastas em computação e em comunicação durante uma migração de tarefas.

O último experimento avalia a viabilidade do modelo em termos de energia. A partir da reprodução do experimento de Bertozzi (2006) — apresentado na Seção 4.1.1 são levantados os custos em termos de energia do mecanismo. Finalmente, é feita uma especulação de como e em quanto tempo esse gasto energético seria compensado.

A metodologia de cada experimento é explicada nas suas respectivas seções, entretanto, todos os experimentos pressupõem a mesma plataforma. A Figura 6.1 mostra uma visão abstrata da arquitetura simulada. Ela é composta de n x n nós interconectados pela *rede-em-chip* com configuração de grelha. Cada nó (também denominado PE, do inglês, *processor element*) é composto por um processador e uma memória local não-compartilhada (a partir de agora denominada *privada*). Essa memória pode ser vista como uma pequena memória *scratchpad* onde o código e os dados residem durante a execução. Alternativamente, um nó pode ser composto apenas por uma memória e seu respectivo controlador, que é implementado na forma de um DMA (*Direct Memory Access*) e realiza a interface entre a memória e a rede.

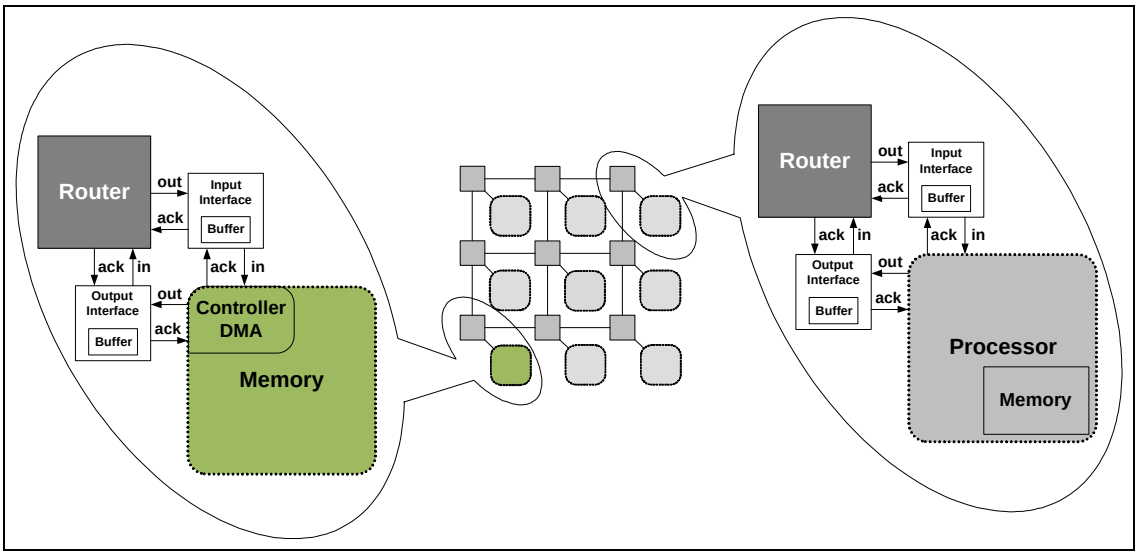

Figura 6.1: Plataforma baseada em *rede-em-chip*

A comunicação entre dois PEs é realizada através da troca explícita de mensagens. O PE onde a tarefa estava originalmente executando lê o código da tarefa a partir da sua memória local e o encaminha para o nó destino através da rede. Do ponto de vista dos demais PEs do sistema, o nó memória é um nó comum no sistema. Uma vez que os PEs são homogêneos (exceto os nós de memória), uma tarefa pode ser executada em qualquer processador.

6.1 Análise de Organizações de Memória

Primeiramente foi estimada – via simulação – a energia gasta na leitura de 1kB de dados de uma memória. O tamanho da memória utilizada foi de 64kB, sendo o tamanho de palavra adotado igual a 8 bits, o tempo de leitura de 10ns (100MHz) e foi assumido que a memória encontra-se desligada antes e após as operações de leitura. Depois de ligada, a memória leva 1us para estar pronta para uso, assim como demora outro 1us para ser desligada, consumindo, durante esses períodos, a mesma energia que consome durante as leituras. Foi medido, como esperado, que a leitura de 1kB leva 12,24us e consome aproximadamente 7,55uJ.

Foram simulados envio e recebimento de 1kB de dados através da rede e obteve-se a energia gasta nessa operação em função da distância dos nós da rede envolvidos na transferência. Considerando que um nó está localizado em uma coordenada (X, Y) na rede, a distância entre dois nós é dada pela Distância de Manhattan das coordenadas de origem e destino dos elementos envolvidos na operação:

$$D = |x\_dest - x\_orig| + |y\_dest - y\_orig| \quad \text{[hop]}$$

Na equação apresentada, x_orig e y_orig são, respectivamente, as coordenadas X e Y do nó que envia os dados e, de maneira análoga, x_dest e y_dest são as coordenadas do nó receptor.

Uma vez que as memórias locais dos processadores estão sempre ligadas, apenas o consumo devido à comunicação deve ser considerado. Três cenários, com duas diferentes organizações de memória, foram considerados: memória distribuída, onde as

transferências se dão entre os nós de processadores; memória global, com transferências entre a memória global e os nós de processamento destino; e uma segunda solução com memória global, onde a memória é dividida em páginas de 4kB e é possível ligar apenas a página envolvida na transmissão. Para as soluções com memória global, o tamanho total da memória é de 64kB e ela está posicionada no centro da rede.

A Figura 6.2 mostra a energia gasta em função da distância dos nós para os três cenários. Como apresentado, o cenário com memória distribuída, com as transmissões nó a nó, possui um custo energético 1,8 e 13 vezes menor que as soluções com paginação e memória global simples, respectivamente. É importante notar que a diferença entre as duas soluções mais eficientes é aproximadamente igual à energia gasta para transferir 1kB de código através de dois nós da rede. Em vista disso, transferências a partir da memória central, quando essa estiver pelo menos três nós mais próxima do nó destino do que o nó de processamento que também possui o código, serão mais eficientes, e essa é a chave da solução híbrida apresentada a seguir.

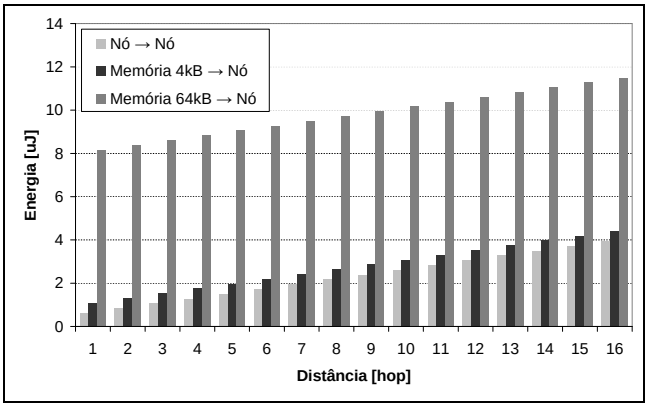

Figura 6.2: Energia gasta na transferência de 1kB de código

É importante ressaltar também que a média das distâncias entre os nós envolvidos nas transferências de código não são as mesmas para as soluções com diferentes organizações de memória, já que com uma memória global todas as transferências seriam realizadas a partir de um nó posicionado no centro da NoC; ao contrário, com a memória distribuída o nó de origem do código pode variar.

A Figura 6.3a mostra uma tabela, representando os nós da rede, com as distâncias entre os nós e a memória central em uma rede grelha. Os nós mais próximos são mais claros que os mais distantes, sendo que o valor exibido no interior de cada quadro é a distância de cada nó à memória. Quando todas as migrações de código se originam da memória global, o elemento processador no qual estava sendo executada a tarefa não importa, uma vez que a energia gasta pelo pacote que solicita o código à memória é desprezível. Assim sendo, os resultados independem da política de migração, podendo ela ser do tipo ***sender-*** ou ***receiver-initiated*** como definido em [Sinha, 1997].

A Figura 6.3b mostra um exemplo de uma possível tabela de distâncias para uma solução distribuída. O nó origem, aquele que executava a tarefa originalmente, está localizado em (0, 0) e pode haver distâncias de até 16 nós em uma rede grelha 9x9. Na solução distribuída para uma NoC NxN existem N^2 tabelas de distância diferentes, variando de acordo com os nós de origem e destino das transferências.

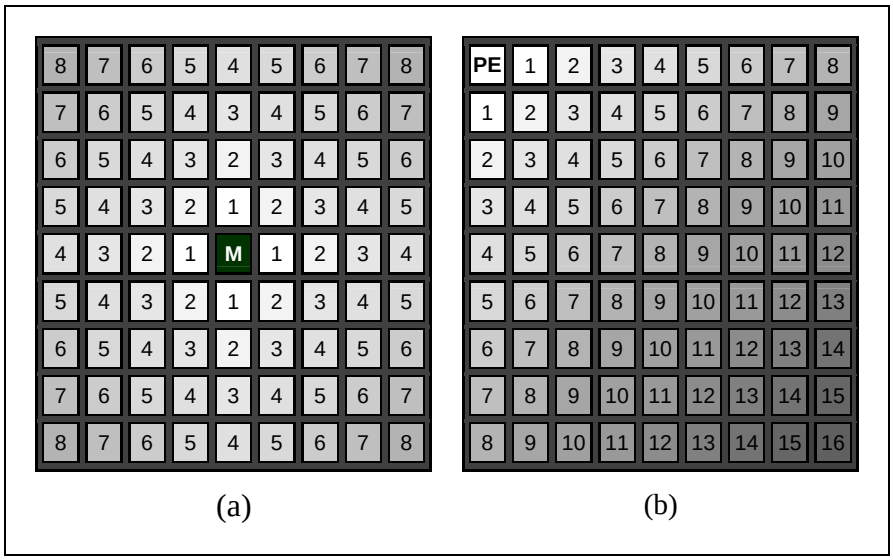

Figura 6.3: Distância dos nós à memória (a) e entre os nós na solução distribuída (b) em uma rede grelha 9x9.

Assumindo que cada nó possui a mesma possibilidade de receber uma tarefa e, ainda, para a solução distribuída, de enviar uma tarefa, a Tabela 6.1 mostra para as duas soluções desenvolvidas a distância média entre os nós comunicantes, considerando diferentes tamanhos de NoC. A distância média entre os nós na abordagem com memória global é 25% menor em relação à distância média na solução distribuída.

Tabela 6.1: Distância média para diferentes tamanhos de NoC.

	Distância Média [hop]	
Noc	**Global**	**Distribuído**
3x3	1.333	1.778
5x5	2.400	3.200
7x7	3.429	4.571
9x9	4.444	5.926

Considerando a distância média, o cenário com solução distribuída passa a ser aproximadamente 1,3 e 10 vezes mais eficiente em termos energéticos que as soluções com paginação e apenas com memória global, respectivamente, como mostrado na Figura 6.4.

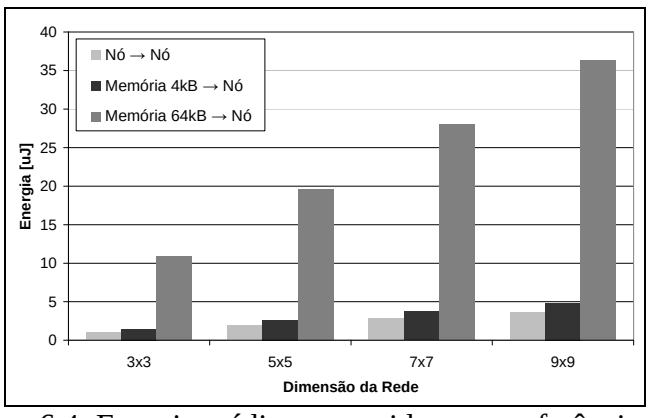

Figura 6.4: Energia média consumida na transferência de 1kB.

6.1.1 Solução Híbrida

Os resultados apresentados indicam que a distância média entre os nós envolvidos na transferência do código possui um grande impacto nos números de energia final. Assim sendo, uma solução híbrida que pretende reduzir a distância média é proposta. Essa solução também utiliza um nó central de memória, mas a transferência do código é realizada entre o nó destino e o emissor mais econômico, que pode ser tanto a memória global quanto o nó que executava a tarefa originalmente. Ressalta-se que nem sempre o sistema escolherá o nó mais próximo do destino para realizar a transferência. A memória global será escolhida como o nó que enviará o código apenas se a sua distância em relação ao nó destino for 3 ou mais nós menor do que a distância entre o processador que executava a tarefa anteriormente e o nó destino.

A Figura 6.5 ilustra a origem escolhida e a respectiva distância calculada para a transferência do código de uma tarefa executada originalmente junto ao nó localizado em (0, 0). Se o destino da tarefa é um dos nós claros, a transferência se dará a partir do nó origem. Caso contrário, o código será oriundo da memória global. Esta distribuição (nós claros – nós escuros) varia de acordo com o nó origem e o tamanho da rede. Com o aumento da dimensão da NoC, o percentual de transferências provenientes da memória global também aumenta. Mas, para os casos no qual o nó origem se localiza próximo à memória global é possível que essa nunca venha a ser usada. Especificamente, para usar a memória global em alguma transferência é necessário que o nó origem esteja localizado a uma distância maior que 2 nós da primeira.

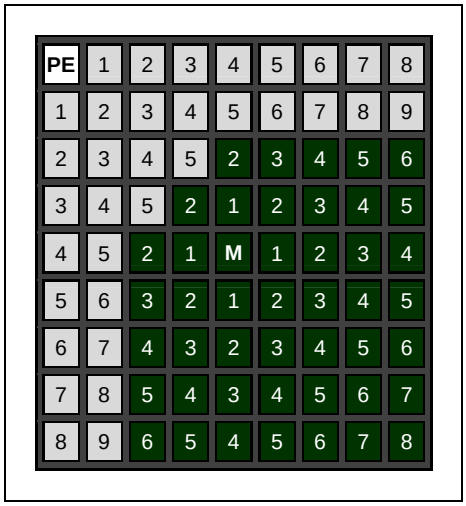

Figura 6.5: Exemplo de distância média para a solução híbrida

Com menores distâncias entre os nós obtém-se transferências energeticamente mais econômicas, como mostrado na Figura 6.6. A solução com memória global, sem paginação, foi omitida para permitir uma melhor visualização dos resultados. A economia média de energia obtida com a utilização da organização de memória híbrida proposta é de aproximadamente 24% e 10% em relação às opções com apenas memória global paginada e memória distribuída, respectivamente.

Uma vez que para uma rede 3x3 a memória global nunca seria utilizada (em outras palavras, a solução híbrida funciona tal qual a distribuída), essa dimensão de rede foi desprezada nos cálculos. Com uma menor distância entre os nós, também obtém-se um tempo de transferência menor. Esse tempo varia com as dimensões da rede e as distâncias médias, que são, em nós, para a solução híbrida iguais a *1,7*, *2,6*, *3,9* e *4,2* para redes de dimensões de 3x3, 5x5, 7x7 e 9x9, respectivamente. Considerando esse

fato, a solução híbrida possui um tempo de transferência menor para NoCs com dimensões de 9x9 ou maiores. A economia de tempo para esses sistemas, para os tamanhos de rede utilizados, obtida com a utilização da solução híbrida, calculada baseada nas distâncias médias, varia entre 1% e 6% sobre a solução com memória global paginada e entre 25% a 29% sobre a solução que utiliza apenas memória distribuída.

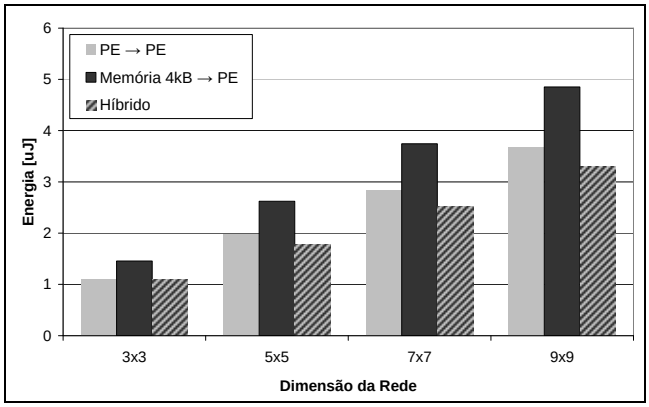

Figura 6.6: Energia média consumida na transferência de 1kB de dados.

O impacto da solução proposta é pequeno frente à energia total consumida pelo sistema, uma vez que migrações e novas alocações de tarefas ocorrem apenas eventualmente. O que se pretende, porém, é encontrar soluções para tornar o processo de migração de tarefas menos oneroso – tanto em termos de desempenho, quanto de energia consumida – e mais facilmente amortizável. Tendo isso em mente, ressalta-se que, para migrações que ocorrem em pontos de código sem contexto (*stateless points*) (NOLLETa, 2005), o código pode representar quase 100% dos dados transferidos, o que justifica a realização do presente experimento no objetivo de encontrar uma forma de amortizar o tempo e a energia consumidos no processo.

6.2 Validação do Modelo de Migração de Tarefas

Este segundo experimento foi realizado com dois objetivos: primeiro, validar o modelo desenvolvido, ou seja, testá-lo e atestar que o mesmo se comporta da maneira esperada; segundo, verificar suas características de desempenho. A comparação será realizada com tempos ideais, nos quais é considerado apenas o tempo de transferência e não há nenhum *overhead* de processamento, e com os dados disponibilizados por outros trabalhos da área.

Não será possível realizar uma comparação precisa com os modelos de Bertozzi (2006), Nollet (2005a) e Ozturk (2006), pois esses trabalhos não disponibilizam dados suficientes para uma comparação justa. Aqueles que apresentam os tempos de migração não informam qual a freqüência de operação dos processadores o que torna inviável a análise nesse aspecto.

O primeiro resultado apresenta o crescimento do tempo de migração em função da distância entre o nó receptor (origem) e emissor (destino) da tarefa. Nesse experimento todos os processadores operavam à freqüência de 100MHz, enquanto os roteadores da rede trabalhavam a 5MHz. Essas freqüências foram ajustadas a fim de garantir que não houvesse contenção por parte de um único processador, ou seja, que ele não fosse capaz

de injetar dados na rede mais rápido do que essa poderia consumir. Optou-se, no entanto por deixar a rede pouco ociosa, operando perto do limite de sua disponibilidade.

Em todos os experimentos, o tamanho de *flit* utilizado foi de 3 *phits*, sendo que o *phit* possui largura de 10 *bits* (com carga útil de 8). Um pacote pode ter tantos *flits* quanto desejar. O tamanho de todos os *buffers* utilizados é equivalente a 1 *flit*. Os pacotes utilizando as primitivas propostas ocupam 2 *flits*. O tempo de processamento de uma primitiva de envio é de, no mínimo, 422 ciclos, considerando a interface de rede ociosa. Não há tempo máximo determinado, assim como as primitivas bloqueantes não possuem *timeout* e as não bloqueantes retornam um erro caso a interface de rede esteja ocupada. Considerando o número de ciclos e o tamanho dos pacotes de migração a freqüência mínima de operação da rede para não haver contenção com os processadores operando a 100MHz deveria ser de cerca de 1,43MHz. Contudo esse valor assume que apenas um processador injetará dados em um determinado roteador, o que não é sempre verdade. Uma freqüência razoável de operação seria obtida considerando que há dados nas cinco entradas do roteador e todos desejam utilizar a mesma saída, mesmo que utilizando roteamento XY isso não seja possível. Dessa maneira uma freqüência adequada seria 25 vezes maior que 1,43MHz, ou seja, 35,75MHz. Esses dados, entretanto, são especulativos, é preciso analisar cada conjunto de aplicações e suas restrições de tempo-real para definir a melhor freqüência de fato para cada sistema.

A Figura 6.7 apresenta o tempo de migração de uma tarefa de 0kB de contexto. Na prática não existe tarefa com tal tamanho de contexto, sendo então utilizada a menor tarefa possível, contabilizando cerca de 400 bytes de carga útil (64 de dados de objeto, 92 de dados da pilha e 222 de código). Observa-se que o tempo de migração varia muito pouco em função da distância, cerca de 400ns/hop, o tempo necessário para o pacote transpor um roteador. Essa pequena variação ocorre porque o processo ocorre como um *pipeline*, ou seja, assim que um pacote é injetado na rede, outro já está sendo gerado e quando o primeiro passar para o próximo roteador o último será injetado e assim sucessivamente. Dessa maneira, depreende-se que a variação do tempo de migração é constante para qualquer tamanho de tarefa e, quanto maior a tarefa, menos significativa será a contribuição da distância entre os nós no tempo de migração.

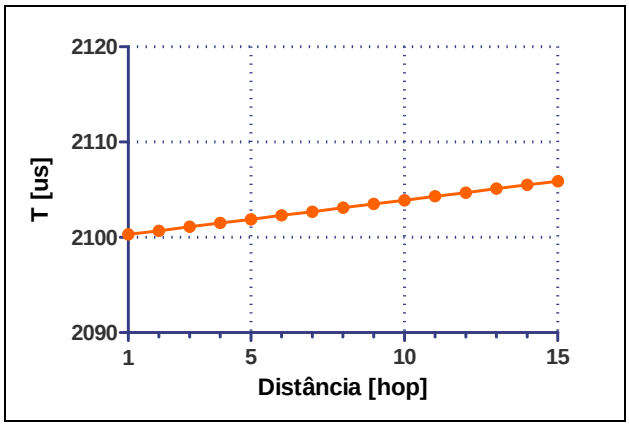

Figura 6.7: Tempo de migração como função da distância entre os nós

Os tempos de migração para tarefas de 8kB e 16kB de contexto são, respectivamente, 47,939ms e 94,146ms, para migrações entre nós a 1 hop de distância entre si. Observa-se com isso que o crescimento do tempo de migração é função linear do tamanho da tarefa, assim como era esperado.

Considerando tempos ideais, com uma rede sempre disponível, operando a uma freqüência de 1,43MHz, mas sendo injetado um *phit* por ciclo de roteador na mesma, o tempo de migração para as tarefas do parágrafo anterior, distando o nó origem e destino o mesmo número de hops, seria de 34ms e 64ms, respectivamente. Ao investigar-se a origem dessa variação de tempo, verificou-se que se o processador injetasse dois pacotes a cada 422 ciclos de fato os tempos seriam aproximadamente os mesmos. Apesar disso, ao retornar da primitiva, existe certo processamento para disponibilizar na memória os próximos dados dos pacotes, onerando o desempenho. Com isso, tem-se, em média 538 ciclos entre cada primitiva de envio, o que permitiria à rede operar a 1,12MHz.

Já na comparação com o trabalho de Pittau (2007), os cerca de 9,5 milhões de ciclos aqui necessários para migrar uma tarefa de 16kB parecem muito expressivos frente aos 156 mil ciclos por ele apresentados. Contudo, duas considerações devem ser feitas. Em primeiro lugar, no seu estudo o autor utiliza processadores do tipo *Microblaze* (XILINX, 2007) que possui um *pipeline* de cinco estágios, fazendo com que o número de ciclos por instrução (CPI, do inglês, *Cycle per Instruction*) desse processador seja próximo de 1,25; já para o processador femtoJava Multiciclo, as instruções levam de 3 a 14 ciclos para executar e seu CPI médio fica em 6,65.

Em segundo lugar, e principalmente, como o sistema de Pittau utiliza barramento e uma memória compartilhada para transferir os dados, apenas uma operação de escrita e uma de leitura precisam ser realizadas para transferir um dado. Assim, não há empacotamento de mensagens, o que onera muito menos o desempenho do sistema.

Os tempos apresentados por Bertozzi (2006), que também utiliza troca de mensagens, por sua vez, são cerca de 10 vezes menores que o do mecanismo aqui proposto, contudo o autor também utiliza processadores com *pipeline* (da família ARM), memória compartilhada e não informa a freqüência de operação dos mesmos, tornando inviável uma comparação e análise mais profunda.

6.3 Experimento de Bertozzi

Este experimento foi modelado de maneira a reproduzir o estudo apresentado na Seção 4.1.1. Aqui os processadores operam a 100MHz e os roteadores a 5MHz. De maneira similar, foi avaliado o tempo necessário para uma tarefa obter certa quantia de tempo de CPU em quatro cenários.

No primeiro cenário, a tarefa executa sozinha; no segundo, a tarefa roda no mesmo processador junto a outra tarefa; no terceiro, a tarefa divide o núcleo processante com mais duas tarefas e; finalmente, no quarto cenário, é utilizada migração de tarefas para mover uma tarefa que compartilhava um processador com mais duas tarefas para outro que originalmente executava apenas uma tarefa. Para o último cenário, o experimento foi realizado com três tamanhos de contexto: 0kB, 8kB e 16kB. O contexto aqui se refere somente aos dados, já que o código também é transmitido, sendo, porém, muito pequeno (cerca de 400 bytes). O código é compacto da referida maneira devido ao fato de se tratarem de tarefas de teste que apenas escrevem seu respectivo identificador, para fins de depuração, em uma porta do processador.

Nos experimentos aqui apresentados, salvo indicação contrária, a distância entre os nós emissor e receptor das tarefas é de um *hop*. Também foi considerado que, imediatamente antes do início do processo de migração de tarefas, as mesmas são

paradas no nó emissor e reiniciadas logo após a conclusão do processo. Com isso foi possível acelerar a velocidade de migração de tarefas. É importante notar que essa não é uma limitação do mecanismo, mas, sim, uma decisão de projeto para os experimentos. Os resultados são mostrados na Figura 6.8.

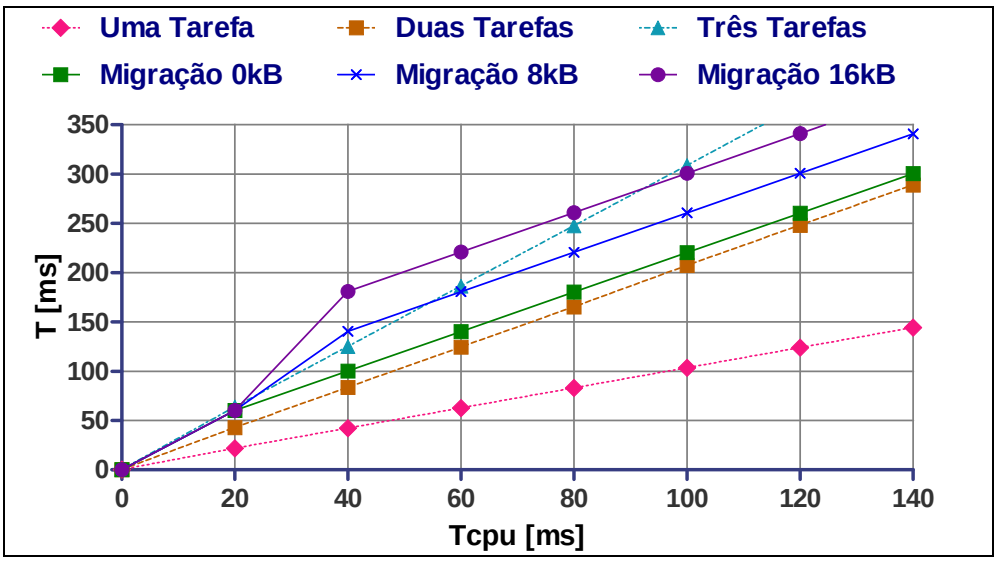

Figura 6.8: Tempo de CPU vs. Tempo de Execução

Na figura, o eixo das abscissas (eixo *x*) representa o tempo efetivo de CPU, ou seja, o tempo que o processador se dedicou a executar, especificamente, a tarefa analisada em cada cenário. Em todos os casos, a tarefa analisada é aquela iniciada por último pelo escalonador *round-robin* local. Por sua vez, o eixo das ordenadas (eixo *y*) indica o tempo de execução, isto é, indica o passar do tempo. Neste experimento, o processo de migração – quando realizado – se inicia 60ms após o início da execução das tarefas. É possível notar que, com a migração, depois de um período de amortização, que ocorre aproximadamente aos 275ms (eixo *y*) para a tarefa com contexto de 16kB, por exemplo, a tarefa começa a obter mais tempo de CPU no mesmo tempo de execução quando comparada à situação do terceiro cenário (curva *Três Tarefas*). A amortização ocorre aproximadamente aos 180ms para a tarefa com contexto de 8kB, e para a tarefa de contexto 0kB a amortização é praticamente instantânea. Nesse gráfico, quanto menor a derivada da curva, mais rapidamente um tarefa realiza a sua computação.

Quando a tarefa é executada sozinha em um processador (curva *Uma Tarefa*), ela obtém mais tempo de processador em um menor tempo de execução, assim como o esperado. Pode-se observar que quando o contexto é pequeno, como na situação da migração da tarefa de 0kB de contexto, o tempo de amortização é pequeno e nesse cenário o tempo que a tarefa leva para obter um determinada quantidade de CPU é próximo ao do cenário inicialmente balanceado (curva *Duas Tarefas*). Na prática as duas curvas nunca se cruzam, tornado-se, após de um tempo, paralelas. As curvas envolvendo migrações de tarefas de contextos maiores também se tornam paralelas à do cenário balanceado.

Mesmo sendo os resultados análogos àqueles apresentados em Bertozzi (2006) – reproduzidos na Seção 4.1.1 –, existem algumas diferenças que merecem ser explicadas. As tarefas aqui apresentadas são menores, mas o tempo de migração é praticamente o mesmo. Isso ocorre porque o processador FemtoJava multiciclo é bastante lento – se

fosse utilizada a versão *pipeline* do mesmo processador, a transferência seria realizada cerca de cinco vezes mais rápido, por exemplo. Ressalta-se que o gargalo da migração de tarefas aqui apresentada é o processador e não a *rede-em-chip*. É importante notar também que Bertozzi não informa a freqüência de utilização dos processadores. Sabe-se que, de fato, a *rede-em-chip* possui baixa utilização, uma vez que são necessários cerca de 500 ciclos de processador para injetar um pacote na rede.

A Figura 6.9 mostra os resultados relativos à energia dinâmica consumida pelos dois nós envolvidos no processo de migração de tarefas e para os três diferentes tamanhos de contexto das mesmas. A energia da rede para cada nó corresponde ao roteador (incluindo os *buffers*) e os links de saída que conectam o processador à rede e a rede entre si. O eixo *y*, para todos os gráficos, indica a energia gasta, enquanto o eixo *x* representa o tempo de execução. Note que o eixo *y* da Figura 6.8 corresponde ao eixo *x* aqui.

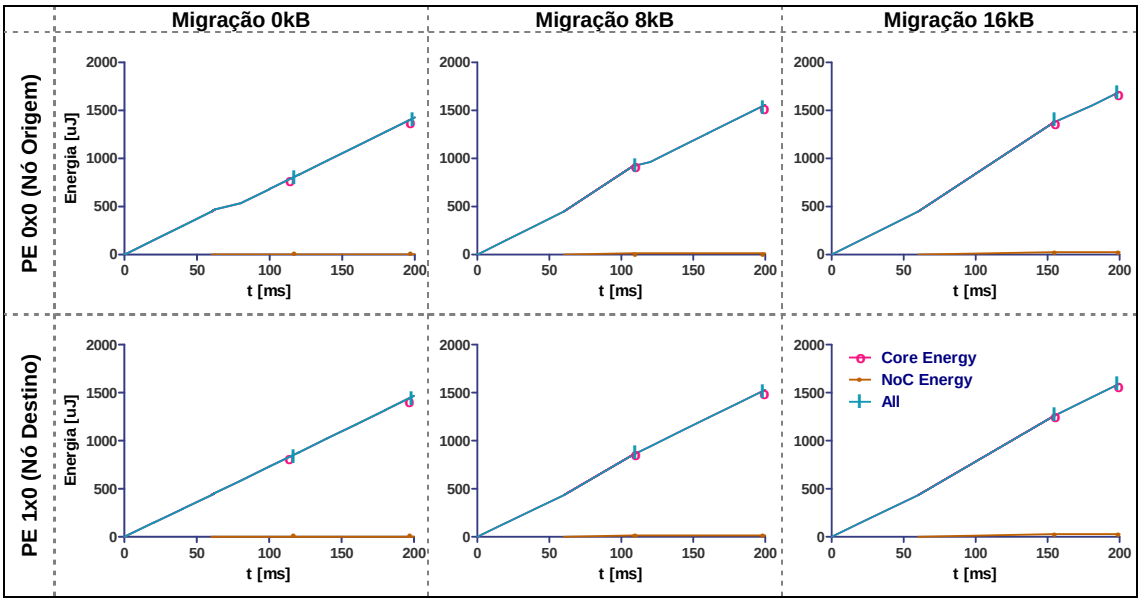

Figura 6.9: Energia gasta ao longo do tempo de execução

É possível notar que para tarefas de contexto de 0kB a energia da rede é desprezível, ao passo que para os outros tamanhos de tarefas a rede começa a ter importância significativa no consumo total do sistema. Também é possível perceber que a dissipação de potência (a derivada da curva *Energia do Núcleo*) durante o processo de migração é maior do que enquanto o sistema simplesmente escalona tarefas. Isso ocorre porque as primitivas de comunicação, assim como a API de migração, utilizam um subconjunto energeticamente oneroso de instruções do processador. O processo de migração é caro em termos de energia porque nesse período uma grande quantidade de operações de I/O é realizada.

A Figura 6.10 mostra a energia gasta no sistema em cada um dos seus principais componentes. A energia de *comunicação* corresponde à energia gasta no processo de migração. Para a rede, ela corresponde a toda a energia, uma vez que as tarefas não se comunicam entre si. Já para os processadores ela está relacionada às instruções usadas para executar as ações do processo de migração. Por sua vez, a energia de *execução* significa a energia gasta pela computação das tarefas e pelo núcleo de sistema operacional. Através desses resultados é possível notar que o custo de migração é significativamente maior do que simplesmente a energia gasta na rede. Além disso, o custo de migração aumenta linearmente com o tamanho da tarefa.

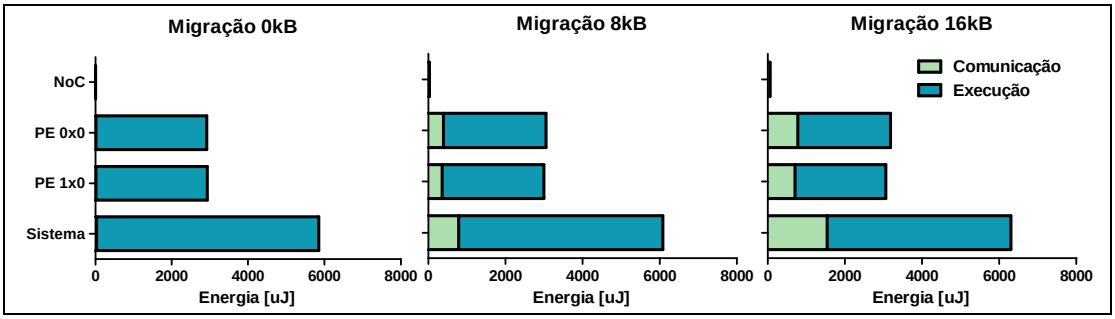

Figura 6.10: Energia detalhada.

Levando-se em consideração o desempenho, depois da migração de tarefas, se bem realizada, é necessário apenas aguardar certo período de tempo para começar a obter ganhos; no entanto, caso a métrica considerada seja energia, a obtenção de vantagens não é tão simples. Para obter ganhos de energia é necessário utilizar alguma técnica de gerenciamento de energia, como DVS, por exemplo.

Na simulação da aplicação de DVS, assumiu-se que os processadores possuem quatro tensões de operação e que a correta execução da tarefa não é afetada pela diminuição da freqüência. Isto quer dizer, por exemplo, falando-se de tarefas periódicas de tempo-real, que as mesmas realizam a computação necessária sem perda de *deadlines*. Cada tarefa impõe a mesma carga computacional ao sistema, que equivale a 25% da capacidade do processador quando este executa na sua velocidade máxima. Essa carga inclui todas as interações com o sistema operacional. Os níveis de operação correspondem às freqüências de 100, 75, 50 e 25MHz. Foi desenvolvido um modelo analítico para determinar o tempo durante o qual o sistema deveria manter a mesma configuração a ponto de começar a obter ganhos em termos de energia, que foi denominado *energy freezing time* (EFT). Em outras palavras, o EFT é o tempo necessário para o sistema amortizar completamente o consumo de energia do processo de migração e começar a prover ganhos energéticos em relação ao cenário sem migração que possui três tarefas em um nó e apenas uma em outro.

Foram comparados cenários com e sem migração, utilizando-se DVS em ambos, para uma comparação justa. A energia gasta antes do início da migração não é levada em consideração, dado que a distribuição das tarefas era a mesma para ambos os cenários. Antes da migração, um nó possui três tarefas e o outro apenas uma. O processo irá balancear a carga, deixando os dois nós com duas tarefas cada. Durante a migração, ambos os núcleos operam à máxima velocidade, com a finalidade de torná-la o mais rápida possível. Depois do processo, ambos os núcleos, no cenário com migração, começam a operar a 50MHz, uma vez que cada um conta com duas tarefas. Para o cenário sem migração, as freqüências de operação, são, durante todo o tempo, 75 e 25MHz para os processadores com três e uma tarefa, respectivamente.

Migrações de tarefas devem ocorrer esporadicamente, mas deseja-se saber qual é o intervalo mínimo entre duas migrações. O EFT é função do tamanho do contexto da tarefa, da distância entre os nós envolvidos no processo e da eficiência do mecanismo de migração. A Figura 6.11 apresenta o EFT (eixo das ordenadas) como função do tamanho do contexto da tarefa (abscissas) para dez diferentes distâncias entre os nós. Conforme esperado, o EFT cresce linearmente com o tamanho do contexto da tarefa e

com a distância entre os nós. Para o pior caso, o tempo de amortização é de cerca de 4500ms, o que é aceitável e significa que uma migração leva aproximadamente cinco segundos para ser amortizada e é esse o tempo de espera necessário entre duas migrações consecutivas. Esse custo é considerado aceitável, porque, mesmo se os processos de sistema possuem um curto tempo de vida e não executam tempo o suficiente para justificar uma migração (MILOJICIC, 2000), aplicações de usuário não o são. Uma ligação telefônica é normalmente mais longa do que cinco segundos, e costuma-se ouvir música em dispositivos portáteis, por exemplo, por um período ainda mais longo.

A Figura 6.11 também mostra a vantagem da utilização de aplicações sem contexto (*stateless applications*), em que o código está presente em todos os nós (como serviços do sistema operacional, por exemplo). Para esse tipo de tarefa, o *overhead* de migração é praticamente nulo e o seu tempo de amortização é bastante curto (~45ms para distâncias de um *hop*). Mesmo assim, existem algumas desvantagens na utilização de aplicações sem contexto, como, por exemplo, o fato de não possuírem em todos os pontos de execução contexto nulo. Na realidade, existem pontos de execução nos quais nenhum dado precisa ser transferido; logo, na prática, a migração deve ocorrer nesses pontos. Com isso, o processo perde em transparência, uma vez que o programador deve explicitamente definir os pontos de migração, já que o sistema operacional não conhece o comportamento da aplicação.

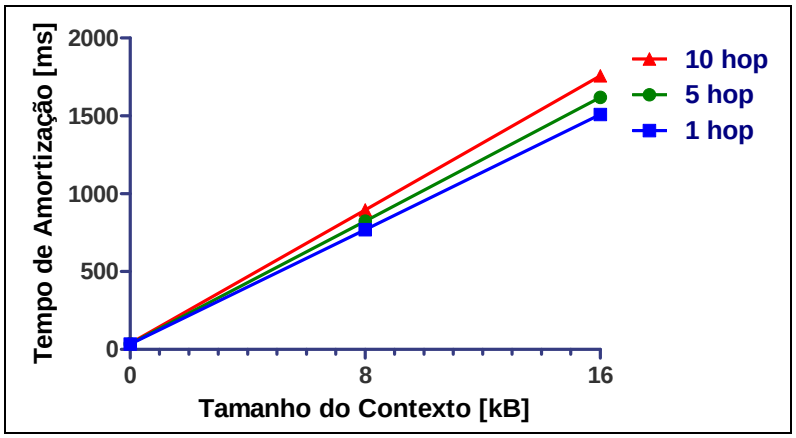

Figura 6.11: Tempo de amortização energética (EFT)

6.4 Análise Crítica

Foi demonstrado que a organização de memória, em MPSoCs, afeta sensivelmente o custo da migração de tarefas. Com isso a exploração de diferentes organizações de memória ganha importância em sistemas onde a funcionalidade poderá ser aplicada. Para a plataforma estudada, um sistema híbrido de memória, com memórias locais e globais – distribuída e compartilhada, respectivamente – proporciona ganhos em termos de energia gasta na migração de tarefas. Os ganhos são animadores pois indicam que a solução híbrida proposta reduz em 24% e 10% em média a energia gasta na transferência do código da tarefa, quando comparado com soluções que utilizam apenas memória compartilhada ou distribuída, respectivamente. No quesito tempo de migração, a organização híbrida proporciona, eventualmente, ganhos de até 6% e 29%, quando comparada às mesmas soluções.

Uma análise mais profunda, no entanto, deve ser realizada a fim de explorar novas organizações. A utilização de memórias *cache* precisa ser avaliada e é necessário verificar a eficiência das soluções com aplicações reais, que explorem as possibilidades fornecidas por um sistemas multiprocessado interconectado por um *rede-em-chip*.

Este trabalho avaliou também o custo energético da aplicação da funcionalidade de migração de tarefas em um sistema específico, com memória distribuída e comunicação baseada em redes-em-chip. Como demonstrado pelos experimentos, a energia consumida no processo é, basicamente, função do tamanho da tarefa e da distância entre os nós envolvidos na transferência. Para redes com baixa taxa de utilização, o tempo de migração, entretanto, varia muito mais em função do tamanho das tarefas do que com a distância entre nós. Isso ocorre porque o processo de migração se dá como um fluxo contínuo e a tarefa migra como um *pipeline* através do sistema. Foi também demonstrado que o *overhead* energético é imposto, principalmente, pela execução dos processadores para distâncias pequenas, porém o custo de comunicação – na *rede-em-chip* – se torna bastante importante com o aumento da distância entre os nós. A aplicação de técnicas de gerenciamento de energia pode, no entanto, tornar viável a aplicação da migração de tarefas em termos energéticos, sendo que a amortização energética ocorre em um tempo maior do que a amortização em termos de desempenho, para uma mesma tarefa.

Muito embora os tempos de amortização tenham sido considerado satisfatórios – de 45ms a 4500ms para energia e de até 190ms para desempenho, nos experimentos realizados – melhorias podem e devem ser realizadas no mecanismo de migração. Uma dessas melhorias é o estudo da adoção de um mecanismo de DMA para realizar as transferências, isso diminuirá a parcela dos processadores no custo energético da migração, entretanto, é preciso avaliar a adição de um bloco de *hardware* para a realização de uma atividade esporádica, como a migração de tarefas. Melhorias no tempo de migração devem também ser realizadas. Os tempos atuais, de 2ms a 94ms, restringem o período de tarefas de tempo-real. Tarefas com tempo de migração maior que seu período de execução, ou, em última análise, que sua folga entre execuções, não podem ser migradas, sob pena de perderem *deadlines* e comprometerem o sistema. Nesse caso, o uso de DMA também pode proporcionar ganhos ao injetar mais seguidamente pacotes na rede. Outras soluções seriam a adoção de processadores mais rápidos, embora essa seja uma solução da plataforma e não do modelo de migração, ou a utilização de primitivas de comunicação mais eficientes.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho avaliou em termos energéticos a viabilidade da utilização da técnica de migração de tarefas em sistemas embarcados. Mais especificamente, o gasto de energia no processo de migração foi mensurado e discriminado. Confirmaram-se ainda experimentos já realizados no que diz respeito ao tempo de amortização de uma migração. Através da aplicação da técnica de DVS verificou-se que é também possível obter a amortização energética do processo e obter ganhos com o mesmo. Foi dado início ao estudo do impacto da organização de memória em sistemas capazes de efetuar migração de tarefas. Finalmente, uma revisão de conceitos a respeito do tema e uma análise da aplicabilidade da técnica são deixadas como legado para trabalhos futuros.

O primeiro experimento mostrou que a exploração de alternativas de organizações de memória pode trazer benefícios, tornando mais leve o processo de migração. Efetivamente, para uma determinada plataforma, foi mostrado que há diferenças de gasto de energia dependendo de onde se origina o código da tarefa migrada. Para o sistema estudado, é energeticamente mais econômico migrar o código a partir um processador próximo em detrimento à utilização de uma memória à mesma distância.

O segundo experimento revelou que o tempo de migração não depende, significativamente, da distância entre os nós emissor e receptor da tarefa, muito embora o terceiro experimento tenha verificado que em termos energéticos, a distância entre os nós agrega um sensível custo à operação de migração. Algumas condições de contorno para a validade dos resultados obtidos são a utilização da organização proposta e a baixa carga de dados da rede, o que possibilita que a distância entre os nós tenha pouca influência no tempo de migração.

Já o terceiro experimento permitiu averiguar também que é possível obter amortização do processo de migração de tarefas em termos de energia através da utilização de uma técnica de gerenciamento de energia conhecida como DVS.

O simulador desenvolvido representou grande parte do esforço despendido na pesquisa, mas como resultado, o grupo conta agora com uma ferramenta de simulação mais rápida e de precisão semelhante às anteriores. Pela primeira vez o grupo conta com um simulador de vários processadores e com um modelo de *rede-em-chip* para simulação em um nível mais abstrato que o permitido pela linguagem VHDL (*VHSIC hardware description language*). Salienta-se que em última instância o simulador pode ser visto como a transformação do simulador CACO-PS para uma biblioteca na linguagem de descrição SystemC, como proposto por Beck (2004).

A averiguação mais importante dessa pesquisa, como já mencionado, é a quantificação do custo energético do processo de migração de tarefas. Com isso, é possível agora afirmar que a aplicação de técnicas de alocação de tarefas em sistemas embarcados multiprocessados de uma única pastilha pode trazer benefícios quanto ao consumo do sistema, ou seja, pode fazer o sistema operar de forma satisfatória, mas gastando globalmente, em termos de espaço e tempo, menos energia.

Assim como este trabalho surgiu a partir de pesquisas anteriores do grupo de pesquisa, novos objetos de estudo foram deixados em aberto e poderão ser explorados futuramente. Salienta-se que o simulador desenvolvido já é utilizado por um mestrando e três doutorandos do grupo em suas respectivas pesquisas. A próxima seção aborda com maior riqueza de detalhes alguns dos possíveis trabalhos que podem vir a ser realizados.

7.1 Trabalhos Atuais e Futuros

7.1.1 Implementação e validação de heurísticas de alocação

Já são objeto de estudo do grupo heurísticas de alocação de tarefas. Contudo, tais pesquisas são conduzidas em um nível maior de abstração, utilizando o simulador Serpens. O refinamento dos resultados, através do desenvolvimento de algoritmos Java para as políticas de alocação, é o próximo passo natural do projeto. Após a avaliação energética dessas heurísticas, as mais efetivas poderão ser implementadas em nível de protótipo.

7.1.2 Expansões no simulador YAFJS

Embora bastante genérico, melhorias no simulador podem e devem ser realizadas. Uma extensão já desenvolvida é o suporte à memória compartilhada no simulador, com um ou mais nós possuindo apenas uma memória que pode ser acessada por diferentes processadores localizados em outros nós. Enquanto esperam pelos dados os processadores aguardam congelados, através do uso de *clock gating* no relógio do processador.

Outra melhoria a ser desenvolvida no próximo semestre é a agregação de memórias *cache* aos processadores. Deverão ser implementadas *caches* com diferentes políticas de substituição e exploradas técnicas de coerência das mesmas no simulador. Possivelmente não serão necessárias adaptações no modelo do processador para a utilização de *caches.*

A conclusão do desenvolvimento do modelo de femtoJava *pipeline* no simulador, especificamente a finalização do mecanismo de interrupção, possibilitará a simulação de plataformas com nós heterogêneos, o que abrirá novas possibilidades de exploração do espaço de projeto e gerará novos resultados para as técnicas desenvolvidas no grupo.

7.1.3 Inclusão do mecanismo de migração junto ao *middleware* adaptativo

O mecanismo de migração já estava sendo adaptado para operar como um serviço do *middleware* adaptativo desenvolvido no Grupo (SILVA JR., 2007). Essa adaptação passa inicialmente, por transpor o mecanismo e adotar a API de comunicação como forma de transmissão e não mais primitivas de baixo nível. Assim a migração se tornará ainda mais transparente, mas será necessário levantar seus custos novamente. Os estudos preliminares e a modelagem do mecanismo do sistema se dirigem para fazer

com que a migração seja vista como um evento do sistema, ficando a sua implementação mais orientada a objetos e intuitiva ainda. Possivelmente a parte de recepção de uma tarefa passará para uma *thread* e sairá da rotina de interrupção, o que a tornará mais próxima às implementações de mecanismos de sistemas distribuídos comerciais.

7.1.4 Exploração da organização de memória em MPSoCs baseados em NoC

O simulador e suas extensões possibilitarão a exploração do espaço de projeto no que tange à distribuição e aos tipos de memória de uma plataforma embarcada. Estudos dessa natureza já foram realizados considerando-se barramentos como meio de comunicação. Entretanto, tratando-se de redes-em-chip é preciso averiguar se as mesmas características se mantêm, levantando os custos do uso de cada organização de memória possível. Caso contrário, poderá se definir organizações de memórias ideais para diferentes tipos de aplicação.

7.1.5 Desenvolvimento de novos modelos de migração

Com novas organizações de memórias, será preciso averiguar qual a melhor maneira de migrar tarefas em cada uma. Um estudo dessa natureza buscaria a generalização do modelo aqui proposto, levando em consideração um grau de liberdade que estava fixo, a plataforma em si.

O impacto do uso apenas de memórias compartilhadas, a associação de diferentes níveis de *cache* e a distribuição heterogênea de memórias pelo sistema são fatores que podem determinar alterações no modelo de migração de tarefas. Um exemplo prático é o fato de que, ao se utilizar memória compartilhada, não será mais necessário utilizar troca de mensagens para o envio e recebimento do código.

O custo dessas alterações será levantado e será possível a comparação das novas soluções de migração propostas com aquelas desenvolvidas para sistemas com estruturas de memória semelhante, mas interconectadas por barramentos.

REFERÊNCIAS

ALTMAN, E. R. *et al.* Advances and Future Challenges in Binary Translation and Optimization. **Proceedings of the IEEE**, Piscataway, v.89, n. 11, p. 1710-1722, Nov. 2001.

ARTSY, Y.; FINKEL, R. Designing a Process Migration Facility. **Computer**, Los Alamitos, CA, v. 22, n. 9, p. 47-56, Sept. 1989.

BARAK, A.; WHEELER, R. MOSIX: An Integrated Multi-processor UNIX. In: WINTER USENIX CONFERENCE, 1989, San Diego, CA. **Proceedings…** Berkeley, CA: USENIX Association, 1989. p. 101-112.

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

BECK FILHO, A. C. S. *et al.* CACO-PS: a general purpose cycle-accurate configurable power simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2003, São Paulo. **Proceedings...** Los Alamitos, CA: IEEE Computer Society, 2003. p. 349–354.

BECK FILHO, A. C. S. *et al.* General Purpose Compiled-Code Power Simulator. In: SIMPÓSIO REGIONAL DE MICROELETRÔNICA, SIM, 18., 2003**,** Novo Hamburgo, RS. **Proceedings...** Novo Hamburgo: Feevale, 2003. p. 133-136.

BECK, A. C. S. **Uso da Técnica VLIW para Aumento de Performance e Redução do Consumo de Potência em Sistemas Embarcados Baseados em Java**. 2004. 126p. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS Porto Alegre, BR-RS.

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

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

BERGAMASCHI, R.; LEE, W. Designing System-on-Chip using Cores. In: DESIGN AUTOMATION CONFERENCE, DAC, 37., 2000. **Proceedings…** New York: ACM, 2000. p. 420-425

BERTOZZI, S. *et al.* Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2006. **Proceedings...** [S.l.]: IEEE, 2006. v. 1, p. 1-6.

BRIÃO, E. W.; BARCELOS, D.; WRONSKI, F.; WAGNER, F. R. Impact of Task Migration in NoC-based MPSoCs for Soft Real-time Applications. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, IFIP-SOC, 2007, Atlanta, USA. **Proceedings…** New York: IEEE, 2007. p.296-299.

BRIÃO, E. W.; BARCELOS, D.; WAGNER, F. R. Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications. Trabalho aceito para Design Automation and Test in Europe, DATE, 2008, Munich, Germany.

BUNGALE, P. P. *et al.* An Approach to Heterogeneous Process State Capture/Recovery to Achieve Minimum Performance *Overhead* During Normal Execution. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, IPDPS, 2003. **Proceedings…** Washington, DC: IEEE Computer Society, 2003. p. 104.

BUTTS, J. A.; SOHI G. S. A Static Power Model for Architects. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 2000, Monterey, California. **Proceedings…** New York: ACM Press, 2000. p. 191-201.

CABRERA, L. The Influence of Workload on Load Balancing Strategies. In: WINTER USENIX CONFERENCE, 1986. **Proceedings…** Berkeley, CA: USENIX Association, 1986. p. 446-458.

CHANCHIO, K.; XIAN-HE S. SNOW: *Software* Systems for Process Migration in High-Performance, Heterogeneous Distributed Environments. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING WORKSHOPS, 2002. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2002. p. 589-596.

CHEN, Y. *et al.* An Empirical Study of Programming Language Trends. **IEEE Computer**, New York, v. 22, n. 3, p. 72-79, May 2005.

COULOURIS, G. *et al.* **Distributed systems**: concepts and design. 4th ed. Harlow: Addison-Wesley, 2005. 927 p.

DALLY, W. J.; SEITZ, C. L. The Torus Routing Chip. **Journal of Distributed Computing**, [S.l.], v.1, n.3, p.187-196, Oct. 1986.

DIMITROV, B.; REGO, V. Arachne: A portable threds system supporting migrant threads on heterogeneous network fams. **IEEE Trans. Par. Distr. Syst.**, Los Alamitos, v. 9, n. 5, p. 459-469, May 1998.

FUGGETTA, A. Understanding Code Mobility. **IEEE Trans. Softw. Eng.**, Los Alamitos, v. 24, n. 5, p. 342-361, May 1998.

FÜNFROCKEN, S. Transparent Migration of Java-Based Mobile Agents. In: INTERNATIONAL WORKSHOP ON MOBILE AGENTS, 1998. **Proceedings...** Berlin: Springer-Verlag, 1998. p. 26-37 (Lecture Notes in Computer Science, v. 1477).

GOSLING, J. *et al.* **The Java Language Specification**. Reading, MA: Addison Wesley, 1996.

HU, J.; MARCULESCU R. Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints. DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, 2004. **Proceedings...** [S.l.]: IEEE Computer Society, 2004. 234-239 p.

ITO, S. A. *et al.* Making Java work for microcontroller applications**. IEEE Design & Test of Computers**, Los Alamitos, v. 18, n. 5, p.100-110, Sept./Oct. 2001.

JALABERT, A. *et al.* XpipesCompiler: A Tool for Instantiating Application Specific Networks on Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, DATE, 2004. **Proceedings...** Washington, DC: IEEE Computer Society, 2004. v. 2, p. 884-889.

KIM, N. S. *et al.* Leakage Current: Moore's Law Meets Static Power. **Computer**, Los Alamitos, v.36, n.12, p. 68-75, Dec. 2003.

LAWTON, G. Moving Java into Mobile Phones. **Computer**, Los Alamitos, v.35, n.6, p. 17-20, 2002.

LITZKOW, M. *et al*. Condor – A Hunter of Idle Workstations. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 1988, San Jose, CA. **Proceedings…** Washington, DC: IEEE Computer Society, 1988. p. 104-111.

LOGHI M.; PONCINO M.; BENINI L. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In: GREAT LAKES SYMPOSIUM ON VLSI, GLSVLSI, 2004, Boston, MA. **Proceedings…** New York, NY: ACM Press, 2004. p. 406-410.

MAGUIRE JUNIOR, G. Q.; SMITH, J. M. Process Migration: Effects on Scientific Computation. **SIGPLAN Notices**, New York, v. 23, n. 3, p. 102-106, 1988.

MIGNOLET, J-Y. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In: DESIGN AUTOMATION AND TESTE IN EUROPE, DATE, 2003. **Proceedings…** Washington, DC : IEEE Computer Society, 2003. p. 986-992.

MILOJICIC, D. S. *et al.* Process Migration Survey. **ACM Computing Surveys**, New York, v. 32, n. 3, Sept. 2000.

NOLLET V. *et al.* Operating-System Controlled Network on Chip**.** In: DESIGN AUTOMATION CONFERENCE, DAC, 2004, San Diego, CA. **Proceedings…** New York: ACM, 2004. p. 256-259.

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

NOLLET, V. *et al.* Low Cost Task Migration Initiation in a Heterogeneous MP-SoC. In: DESIGN AUTOMATION AND TESTE IN EUROPE, DATE, 2005. **Proceedings…** Washington, DC : IEEE Computer Society, 2005. p. 252-253.

OLIVEIRA, M. F. S. *et al.* Model driven engineering for MPSOC design space exploration. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2007, Rio de Janeiro. **Proceedings…** New York: ACM, 2007. p. 81-86.

OUSTERHOUT, J. K. *et al.* The Sprite Network Operating System. **Computer**, Los Alamitos, CA, v. 21, n. 2, p. 23-26, Feb. 1988.

OUSTERHOUT, J. K. **Tcl and the Tk toolkit**. Reading, MA: Addison-Wesley, 1994.

OZTURK O. *et al.* Selective Code/Data Migration for Reducing Communication Energy in Embedded MpSoC Architectures. In: GREAT LAKES SYMPOSIUM ON VLSI, GLSVLSI, 2006, Philadelphia, PA. **Proceedings…** New York: ACM, 2006. p. 386-391.

PAINDAVEINE, Y.; MILOJICIC, D.S. Process vs. task migration. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, HICSS, 1996. **Proceedings…** Washington, DC: IEEE Computer Society, 1996. p. 636-645.

PITTAU M. *et al.* Impact of Task Migration on Streaming Multimedia for Embedded Multi-processors: A Quantitative Evaluation. In: EMBEDDED SYSTEMS FOR REAL-TIME MULTIMEDIA, ESTIMedia, 2007. **Proceedings...** [S.l.: s.n], 2007. p. 59-64.

POWELL, M.; MILLER, B. Process Migration in DEMOS/MP. In: SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1983. **Proceedings...** Berkeley, CA: University of Californa at Berkeley, 1983. p. 110-119.

RABAEY, J. M. **Digital Integrated Circuits**: A Design Perspective. Upper Saddle River: Prentice Hall, 1996.

RABAEY, J. M. Silicon Platforms for the Next Generation Systems: What does Reconfigurable *Hardware* Play? In: INT. WORKSHOP FIELD PROGRAMMABLE LOGIC AND APPLICATIONS, 2000. **Proceedings…** London: Springer-Verlag, 2000. p. 277-285.

ROUSH, E.T. **The freeze free algorithm for process migration**. Urbana: Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1995.

SEMICONDUCTOR Industry Association. International Technology Roadmap for Semiconductors: 2007 Edition. Disponível em: <http://www.itrs.net/Links/2007ITRS/Home2007.htm>. Acesso em: 02 abr. 2008.

SILVA, E. T. *et al.* Hardware support in a middleware for distributed and real-time embedded applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2006, Ouro Preto. **Proceedings…** New York: ACM, 2006. p. 149-154.

SINHA, P. K. **Distributed operating systems**: concepts and design. New York: IEEE Computer Society, 1997.

SMITH, J. M. A Survey of Process Migration Mechanisms. **Operating Systems Review**, New York, v. 22, n. 3, p. 28-40, July 1988.

SPECHT, E. *et al.* Analysis of the use of declarative languages for enhanced embedded system software development. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2007, Rio de Janeiro. **Proceedings…** New York: ACM, 2007. p. 324-329.

TANENBAUM, A. S. *et al.* Experiences with the Amoeba Distributed Operating System. **Communic. ACM**, New York, v. 33, n. 12, p. 46-63, Dec. 1990.

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

THEIMER, M.; LANTZ, K.; CHERITON, D. Preemptable Remote Execution Facilities for the V-System. In: SYMPOSIUM ON OPERATION SYSTEMS PRINCIPLES, 1985, Washington, DC. **Proceedings…** New York: ACM, 1985. p 2-12.

TRUYEN, E. *et al.* Portable Support for Transparent Thread Migration in Java. In: INTERNATION SYMPOSIUM ON AGENT SYSTEMS AND APPLICATIONS, ASA, 2.; INTERNATION SYMPOSIUM ON MOBILE AGENTS, MA, 4., 2000, Zurich, Switzerland. **Proceedings…** Berlin: Springer, 2000. p. 29-43. (Lecture Notes in Computer Science, v. 1882).

TAIWAN Semiconductor Manufacturer Company. Disponível em: <http://www.tsmc.com.tw>. Acesso em: 02 abr. 2008.

UCLINUX. Disponível em: <http://www.uclinux.org>. Acesso em: 13 fev. 2008.

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

WEHRMEISTER, M. A. **Framework orientado a objetos para projeto de *hardware* e *software* embarcados para sistemas tempo-real**. 2005. 104 p. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre, BR-RS.

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

WRONSKI, F. *et al.* Evaluating Energy-Aware Task Allocation Strategies for MPSoCs. In: CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, 2006, Braga, Portugal. **Proceedings…** Boston: Springer, 2007**.** p. 215-224.

XILINX. **Microblaze Processor Reference Guide**. Disponível em: http://www.xilinx.com/support/documentation/sw_manuals/edk92i_mb_ref_guide.pd. Acesso em: 02 abr. 2008.

YE, T. *et al.* Analysis of Power Consumption on Switch Fabrics in Network Routers. In: DESIGN AUTOMATION CONFERECE, DAC, 2002, New Orleans. **Proceedings…** New York: ACM, 2002. p. 524-529.

ZEFERINO, C. A. *et al.* RASoC: a router soft-core for networks-on-chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, 2004. **Proceedings…** Washington, DC: IEEE Computer Society, 2004. p. 198-203.

ZHOU, S. *et al.* Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. **Software – Practive and Experience**, New York, v. 23, n. 12, p. 1305-1336, Dec. 1993.

ANEXO A NÚMERO DE LINHAS DE CÓDIGO

Arquivos de Cabeçalho

Nome do Arquivo	Linhas
./Arbiter.h	40
./Blocked_queue.h	27
./Buffer.h	62
./common.h	13
./cores/fjmulticiclo/adder.h	23
./cores/fjmulticiclo/addsub.h	27
./cores/fjmulticiclo/alu32.h	30
./cores/fjmulticiclo/alu32_branch.h	23
./cores/fjmulticiclo/and_2p.h	23
./cores/fjmulticiclo/clock5div.h	27
./cores/fjmulticiclo/common.h	61
./cores/fjmulticiclo/CSerialCom/SerialCom.h	87
./cores/fjmulticiclo/dbus_arb1.h	35
./cores/fjmulticiclo/dbus_arb10.h	28
./cores/fjmulticiclo/dbus_arb11.h	22
./cores/fjmulticiclo/dbus_arb12.h	22
./cores/fjmulticiclo/dbus_arb13.h	22
./cores/fjmulticiclo/dbus_arb19.h	23
./cores/fjmulticiclo/dbus_arb2.h	27
./cores/fjmulticiclo/dbus_arb28.h	23
./cores/fjmulticiclo/dbus_arb29.h	23
./cores/fjmulticiclo/dbus_arb3.h	22
./cores/fjmulticiclo/dbus_arb30.h	23
./cores/fjmulticiclo/dbus_arb31.h	23
./cores/fjmulticiclo/dbus_arb4.h	22
./cores/fjmulticiclo/dbus_arb5.h	22
./cores/fjmulticiclo/dbus_arb6.h	22
./cores/fjmulticiclo/dbus_arb7.h	22
./cores/fjmulticiclo/dbus_arb8.h	22
./cores/fjmulticiclo/dbus_arb9.h	22
./cores/fjmulticiclo/dffa.h	28
./cores/fjmulticiclo/dfs.h	41
./cores/fjmulticiclo/femtoJava.h	2035
./cores/fjmulticiclo/femtojava_defs.h	67
./cores/fjmulticiclo/fsm.h	175
./cores/fjmulticiclo/gen_0.h	22
./cores/fjmulticiclo/gen_1.h	22
./cores/fjmulticiclo/gen_cons.h	28
./cores/fjmulticiclo/gen_cons32.h	28
./cores/fjmulticiclo/gen_m2.h	22
./cores/fjmulticiclo/get_2_bits.h	22
./cores/fjmulticiclo/get_bits_16_9.h	22
./cores/fjmulticiclo/get_bits_8_1.h	22
./cores/fjmulticiclo/get_bit_0.h	22
./cores/fjmulticiclo/get_bit_26.h	22
./cores/fjmulticiclo/get_bit_27.h	22
./cores/fjmulticiclo/get_bit_28.h	22
./cores/fjmulticiclo/get_bit_29.h	22
./cores/fjmulticiclo/get_bit_32.h	28
./cores/fjmulticiclo/hwti.h	128
./cores/fjmulticiclo/hwtul.h	159
./cores/fjmulticiclo/iconsctrl.h	28
./cores/fjmulticiclo/imm_cat.h	23
./cores/fjmulticiclo/imm_log_h.h	27
./cores/fjmulticiclo/imm_signal_ext.h	26
./cores/fjmulticiclo/Interrupt_Control.h	71
./cores/fjmulticiclo/maw_mux2_1.h	39
./cores/fjmulticiclo/mux2_1.h	27
./cores/fjmulticiclo/mux4_1.h	30
./cores/fjmulticiclo/mux8_1.h	35
./cores/fjmulticiclo/not1b.h	22
./cores/fjmulticiclo/or_2p.h	23
./cores/fjmulticiclo/pass.h	22
./cores/fjmulticiclo/ram32.h	32
./cores/fjmulticiclo/rambus_arb.h	83
./cores/fjmulticiclo/reg.h	26
./cores/fjmulticiclo/reg32.h	26
./cores/fjmulticiclo/reg8.h	26
./cores/fjmulticiclo/rom.h	37
./cores/fjmulticiclo/rtc.h	32
./cores/fjmulticiclo/serial.h	51
./cores/fjmulticiclo/subtr.h	23
./cores/fjmulticiclo/timer_1.h	24
./cores/fjmulticiclo/timer_2.h	24
./cores/fjmulticiclo/timer_3.h	23
./cores/fjmulticiclo/timer_4.h	23
./cores/fjmulticiclo/unistd.h	0
./cores/PE.h	6
./JobScheduler.h	33
./Link.h	69
./Mesh.h	745
./Msg.h	230
./ni.h	74
./PacketArray.h	100
./RA.h	275
./Router.h	37
./Simulator.h	91
./Switch.h	88
Subtotal	6378

Arquivos de Código

Nome do Arquivo	Linhas
./Arbiter.cpp	69
./Blocked_queue.cpp	26
./Buffer.cpp	138
./common.cpp	27
./cores/fjmulticiclo/adder.cpp	27
./cores/fjmulticiclo/addsub.cpp	38
./cores/fjmulticiclo/alu32.cpp	134
./cores/fjmulticiclo/alu32_branch.cpp	40
./cores/fjmulticiclo/and_2p.cpp	29
./cores/fjmulticiclo/clock5div.cpp	59
./cores/fjmulticiclo/CSerialCom/SerialCom.cpp	156
./cores/fjmulticiclo/dbus_arb1.cpp	75
./cores/fjmulticiclo/dbus_arb10.cpp	27
./cores/fjmulticiclo/dbus_arb11.cpp	35
./cores/fjmulticiclo/dbus_arb12.cpp	35
./cores/fjmulticiclo/dbus_arb13.cpp	35
./cores/fjmulticiclo/dbus_arb19.cpp	37
./cores/fjmulticiclo/dbus_arb2.cpp	35
./cores/fjmulticiclo/dbus_arb28.cpp	37
./cores/fjmulticiclo/dbus_arb29.cpp	36
./cores/fjmulticiclo/dbus_arb3.cpp	38
./cores/fjmulticiclo/dbus_arb30.cpp	37
./cores/fjmulticiclo/dbus_arb31.cpp	36
./cores/fjmulticiclo/dbus_arb4.cpp	35
./cores/fjmulticiclo/dbus_arb5.cpp	35
./cores/fjmulticiclo/dbus_arb6.cpp	35
./cores/fjmulticiclo/dbus_arb7.cpp	35
./cores/fjmulticiclo/dbus_arb8.cpp	36
./cores/fjmulticiclo/dbus_arb9.cpp	36
./cores/fjmulticiclo/dffa.cpp	44
./cores/fjmulticiclo/dfs.cpp	49
./cores/fjmulticiclo/femtoJava.cpp	364
./cores/fjmulticiclo/fsm.cpp	232
./cores/fjmulticiclo/gen_0.cpp	27
./cores/fjmulticiclo/gen_1.cpp	27
./cores/fjmulticiclo/gen_cons.cpp	41
./cores/fjmulticiclo/gen_cons32.cpp	41
./cores/fjmulticiclo/gen_m2.cpp	27
./cores/fjmulticiclo/get_2_bits.cpp	27
./cores/fjmulticiclo/get_bits_16_9.cpp	27
./cores/fjmulticiclo/get_bits_8_1.cpp	27
./cores/fjmulticiclo/get_bit_0.cpp	27

./cores/fjmulticiclo/get_bit_26.cpp	27
./cores/fjmulticiclo/get_bit_27.cpp	27
./cores/fjmulticiclo/get_bit_28.cpp	27
./cores/fjmulticiclo/get_bit_29.cpp	27
./cores/fjmulticiclo/get_bit_32.cpp	44
./cores/fjmulticiclo/hwti.cpp	298
./cores/fjmulticiclo/hwtul.cpp	322
./cores/fjmulticiclo/iconsctrl.cpp	32
./cores/fjmulticiclo/imm_cat.cpp	27
./cores/fjmulticiclo/imm_log_h.cpp	43
./cores/fjmulticiclo/imm_signal_ext.cpp	39
./cores/fjmulticiclo/Interrupt_Control.cpp	143
./cores/fjmulticiclo/main.cpp	178
./cores/fjmulticiclo/maw_mux2_1.cpp	49
./cores/fjmulticiclo/mux2_1.cpp	35
./cores/fjmulticiclo/mux4_1.cpp	39
./cores/fjmulticiclo/mux8_1.cpp	47
./cores/fjmulticiclo/not1b.cpp	27
./cores/fjmulticiclo/or_2p.cpp	29
./cores/fjmulticiclo/pass.cpp	27
./cores/fjmulticiclo/ram32.cpp	63
./cores/fjmulticiclo/rambus_arb.cpp	77
./cores/fjmulticiclo/reg.cpp	48
./cores/fjmulticiclo/reg32.cpp	43
./cores/fjmulticiclo/reg8.cpp	42
./cores/fjmulticiclo/rom.cpp	61
./cores/fjmulticiclo/rtc.cpp	60
./cores/fjmulticiclo/serial.cpp	51
./cores/fjmulticiclo/subtr.cpp	27
./cores/fjmulticiclo/timer_1.cpp	31
./cores/fjmulticiclo/timer_2.cpp	28
./cores/fjmulticiclo/timer_3.cpp	28
./cores/fjmulticiclo/timer_4.cpp	63
./JobScheduler.cpp	51
./Link.cpp	7
./main.cpp	89
./Mesh.cpp	72
./Msg.cpp	11
./ni.cpp	212
./PacketArray.cpp	10
./RA.cpp	71
./Router.cpp	39
./Simulator.cpp	217
./Switch.cpp	141
Subtotal	**5407**
Total	**11785**

ANEXO B TEMPOS DE SIMULAÇÃO

Simulações realizadas em um computador com processador Pentium IV, 2.66GHz, 1GB de memória RAM, cache L1 de 16kB e cache L2 de 1MB com carga variável.

Benchmark	**Número de PEs**	**Tamanho da Rede**	**Tempo de Simulação**	**Ciclos**	**Ciclo/s**
Bertozzi com contexto de 16kB	2	4x4	90min.	40.000.000	~7.407
Bertozzi com contexto de 8kB	2	4x4	121min.	40.000.000	~5.509
Bertozzi com contexto de 0kB	2	4x4	76min.	40.000.000	~8.771