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

EDUARDO WENZEL BRIÃO

Métodos de Exploração de Espaço de Projeto em Tempo de Execução em Sistemas Embarcados de Tempo Real Soft Baseados em Redes-Em-Chip

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

Prof. Dr. Flavio Rech Wagner
Orientador

Porto Alegre, março de 2008.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Brião, Eduardo Wenzel

Métodos de Exploração de Espaço de Projeto em Tempo de Execução em Sistemas Embarcados de Tempo Real Soft Baseados em Redes-Em-Chip / Eduardo Wenzel Brião – Porto Alegre: Programa de Pós-Graduação em Computação, 2008.

183 f.:il.

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

1.Projeto em Tempo de Execução. 2.Algoritmos de Alocação de Tarefas 3. Network-on-chip (NoC). I. Wagner, Flávio Rech Orientador da. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenadora do PPGC: Profa. Luciana Porcher Nedel
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

À minha esposa Eliana,
Que me ensinou a viver.

"*Pleasure in the job puts perfection in the work*."

Aristóteles (384-322 a.C)

AGRADECIMENTOS

Em primeiro lugar, agradeço especialmente a minha esposa Eliana F. Dourado Brião, por todo o amor, companheirismo, compreensão e paciência que me deram forças para vencer esse desafio. Além disso, minha esposa mostrou-me enxergar a vida de modo diferente, no qual pude rever meus conceitos e entender a vida de uma forma mais clara, transparente e livre de qualquer conflito. Finalmente, não posso deixar de fazer menção a todo o carinho que me destes. Nossa união é prova concreta de que amor é inabalável e este com certeza permitirá vivermos sempre em total harmonia junto com nossos futuros filhos e netos. Obrigado, Linda!

Agradeço também aos meus pais, Paulo Renato e Lea Beatriz, pelo incentivo ao estudo e por mostrarem que este é uma das maiores heranças que os pais podem oferecer aos seus filhos, pois este permite a abertura de muitas portas para a realização pessoal e profissional. Obrigado pai, pelas suas sábias palavras (seja em latim, seja em português), pois estas serão, com certeza, propagadas por outras gerações de nossa família. Mãe, obrigado por teu incentivo e orações que com certeza objetivaram meu bem-estar e crescimento profissional.

Agradeço a Deus pelas coisas boas da vida, por minha saúde, por minha vontade de lutar e vencer, pelos meus amigos, pelo ar que respiro e pelo meu maior tesouro: minha Eliana.

Agradeço àqueles que me ajudaram de modo direto e indireto para o desenvolvimento e a realização desta tese: em primeiro lugar, agradeço ao meu orientador Flavio Rech Wagner, por sua dedicação, objetividade e acima de tudo, ética no seu trabalho. Além de um excelente orientador, é um grande amigo. Quando parecia que minhas perspectivas estavam perdendo sentido, meu orientador firmava-as novamente no seu devido lugar, com demonstrações de incentivo e segurança. Ao professor Luigi Carro que me acolheu nesta instituição e acreditou no meu potencial, sempre contribuindo para o desenvolvimento deste trabalho. Sou grato aos professores Carlos Eduardo, Altamiro Susin e Ivan Saraiva pelas dicas importantes para a evolução e fechamento deste trabalho. Aos colegas Daniel Barcelos, Alexis Lazzarotto e Fabio Wronski que contribuíram de forma indispensável para o desenvolvimento deste trabalho. Aos demais colegas e ex-colegas pelo convívio diário no Instituto de Informática: Marcio Oliveira, Marco Wehrmeister, Marcio Oyamada, Gustavo Girão, Edgard Corrêa, Marcio Kreutz, Mateus Rutzig, Antônio Schneider, Elias Teodoro, Rodrigo Motta, Vitor Moscon, Ronaldo Ferreira (Bixo), Dalton Colombo, Eduardo Rhod, Júlio Mattos, Francisco Assis entre outros.

Finalmente, agradeço ao órgão de fomento CNPq, que através do Programa Nacional de Microeletrônica (PNM) concedeu-me a bolsa de doutorado que me manteve durante o curso.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ACO	*Ant Colony Optimization*
ACP	*Application Communication Pattern*
APCG	*Application Characterization Graph*
API	*Application Program Interface*
ASIP	*Application Specific Integrated Processors*
AVR	*Average Rate Heuristic*
BF	*Best-Fit*
BOP	*Begin of Packet*
BP	*Bin-packing*
CAN	*Controller Area Network*
CDCM	*Communication Dependence and Computation Model*
CP	*Constraint Programming*
CPU	*Central Processing Unit*
CTG	*Communication Task Graph*
CWG	*Communication Weighted Graph*
CWM	*Communication Weighted Model*
DAG	*Direct Acyclic Graphs*
DAR	*Dynamic Average Rate*
DPM	*Dynamic Power Management*
DSE	*Design Space Exploration*
DSP	*Digital Signal Processing*
DVS	*Dynamic Voltage Scaling*
EDF	*Earliest Deadline First*
EEP	Exploração de Espaço de Projeto
EOP	*End of Packet*
EZ	*Edge Zeroing*

FF	*First-Fit*
GALS	*Globally Asynchronous and Locally Synchronous*
HD	*Hard-disk*
HDTV	*High-definition Television*
HoL	*Head-of-Line*
ILP	*Integer Linear Programming*
IP	*Intellectual Property*
ITRS	*International Technology Roadmap for Semiconductors*
LC	*Linear Clustering*
LSE	Laboratório de Sistemas Embarcados
MPSoC	*Multi Processor SoC*
NoCs	*Networks-on-Chip*
NRE	*Non-Recurring Engineering*
NF	*Next-Fit*
PC	*Personal Computer*
PCO	*Particle Swarm Optimization*
PE	*Processing Elements*
PHIT	*PHysical unIT*
PPGC	Programa de Pós-Graduação em Computação
QoS	*Quality of Service*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
SASHIMI	*System As Software and Hardware In MIcrocontrollers*
SoC	*System-on-Chip*
SoCIN	*System-on-Chip Interconnection Network*
SPIN	*Scalable Programmable Integrated Network*
TGFF	*Task Graphs for Free*
TLM	*Transaction Level Model*
UFRGS	Universidade Federal do Rio Grande do Sul
VLIW	*Very Long Instruction Word*
VS	*Voltage Scaling*
WCET	*Worst Case Execution Time*
WF	*Worst-Fit*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

RESUMO

A complexidade no projeto de sistemas eletrônicos tem aumentado devido à evolução tecnológica e permite a concepção de sistemas inteiros em um único chip (SoCs – do inglês, *Systems-on-Chip*). Com o objetivo de reduzir a alta complexidade de projeto, custos de projeto e o tempo de lançamento do produto no mercado, os sistemas são desenvolvidos em módulos funcionais, pré-verificados e pré-projetados, denominados de núcleos de propriedade intelectual (IP – do inglês, *Intellectual Property*). Esses núcleos IP podem ser reutilizados de outros projetos ou adquiridos de terceiros. Entretanto, é necessário prover uma estrutura de comunicação para interligar esses núcleos e as estruturas atuais (barramentos) são inadequadas para atender as necessidades dos futuros SoCs (compartilhamento de banda, falta de escalabilidade). As redes-em-chip (NoCs{ XE "NoCs" } – do inglês, *Networks-on-Chip*) vêm sendo apresentadas como uma solução para atender essas restrições. No desenvolvimento de sistemas embarcados baseados em redes-em-chip, deve-se personalizar a rede para atendimento de restrições. Essa exploração de espaço de projeto (EEP), segundo uma infinidade de trabalhos, é realizada em tempo de projeto, supondo-se que é conhecido o perfil das aplicações que devem ser executadas pelo sistema. No entanto, cada vez mais sistemas embarcados aproximam-se de dispositivos genéricos de processamento (como *palmtops*), onde as tarefas a serem executadas não são inteiramente conhecidas a priori. Com a mudança dinâmica da carga de trabalho de um sistema embarcado, a busca pelo atendimento de requisitos pode então ser enfrentada por mecanismos adaptativos, que implementam dinamicamente a EEP. No âmbito deste trabalho, a EEP em tempo de execução provê mecanismos adaptativos que deverão realizar suas funções para atendimento de restrições de projeto. Consequentemente, EEP em tempo de execução pode permitir resultados ainda melhores, no que diz respeito a sistemas embarcados com restrições de projetos rígidas. É possível maximizar o tempo de duração da energia da bateria que alimenta um sistema embarcado ou, até mesmo, diminuir a taxa de perda de deadlines em um sistema de tempo real *soft*, realocando em tempo de execução tarefas de modo a gerar menor taxa de comunicação entre os processadores, desde que o sistema seja executado em um tempo suficiente para amortizar os custos de migração. Neste trabalho, foi utilizada a combinação de heurísticas de alocação da área dos Sistemas Computacionais Distribuídos como, por exemplo, algoritmos *bin-packing* e *linear clustering*. Resultados mostraram que a realocação de tarefas, utilizando uma combinação Worst-Fit e Linear Clustering, reduziu o consumo de energia e a taxa de perda de deadlines em 17% e 37%, respectivamente, utilizando o modelo de migração por cópia.

Palavras-Chave: sistemas embarcados, exploração de espaço de projeto, redes-em-chip, sistemas em chip, sistemas distribuídos.

Methods of Run-time Design Space Exploration in NoC-based Soft Real Time Embedded Systems

ABSTRACT

The complexity of electronic systems design has been increasing due to the technological evolution, which now allows the inclusion of a complete system on a single chip (SoC – System-on-Chip). In order to cope with the corresponding design complexity and reduce design costs and time-to-market, systems are built by assembling pre-designed and pre-verificated functional modules, called IP (Intellectual Property) cores. IP cores can be reused from previous designs or acquired from third-party vendors. However, an adequate communication architecture is required to interconnect these IP cores. Current communication architectures (busses) are unsuitable for the communication requirements of future SoCs (sharing of bandwidth, lack of scalability). Networks-on-Chip (NoC) arise as one of the solutions to fulfill these requirements. While developing NoC-based embedded systems, the NoC customization is mandatory to fulfill design constraints. This design space exploration (DSE), according to most approaches in the literature, is achieved at compile-time (off-line DSE), assuming the profiles of the tasks that will be executed in the embedded system are known a priori. However, nowadays, embedded systems are becoming more and more similar to generic processing devices (such as palmtops), where the tasks to be executed are not completely known a priori. Due to the dynamic modification of the workload of the embedded system, the fulfillment of requirements can be accomplished by using adaptive mechanisms that implement dynamically the DSE (run-time DSE or on-line DSE). In the scope of this work, DSE is on-line. In other words, when the system is running, adaptive mechanisms will be executed to fulfill the requirements of the system. Consequently, on-line DSE can achieve better results than off-line DSE alone, especially considering embedded systems with tight constraints. It is thus possible to maximize the lifetime of the battery that feeds an embedded system, or even to decrease the deadline miss ratio in a soft real-time system, for example by relocating tasks dynamically in order to generate less communication among the processors, provided that the system runs for enough execution time in order to amortize the migration overhead.In this work, a combination of allocation heuristics from the domain of Distributed Computing Systems is applied, for instance bin-packing and linear clustering algorithms. Results shows that applying task reallocation using the Worst-Fit and Linear Clustering combination reduces the energy consumption and deadline miss ratio by 17% and 37%, respectively, using the copy task migration model.

Keywords: embedded systems, design space exploration, network-on-chip, system-on-chip, distributed system.

1 INTRODUÇÃO

Sistemas Embarcados podem ser definidos como sistemas eletrônicos de processamento de informação embutidos em um produto de forma transparente para o usuário (MARWEDEL, 2003). O projeto de sistemas embarcados integrados e complexos (CARRO, 2003), que têm sido responsáveis pela crescente disseminação de tecnologias de informação e de comunicação nas atividades humanas, é um dos grandes desafios da atualidade para aquelas corporações que atuam no campo de microeletrônica e sistemas eletrônicos. Embora se possam identificar algumas características que são comuns a todos os sistemas embarcados, eles estão longe da homogeneidade que pode ser encontrada nos computadores de propósito geral. Atualmente, uma grande variedade de produtos são sistemas embarcados: desde dispositivos simples encontrados no cotidiano de nossas vidas, como telefones celulares, aparelhos de DVD, televisões digitais, câmeras digitais, mp3 *players*, iPods, Palmtops e outros; a dispositivos complexos, como controles de sistemas automotivos, de auxilio médico, de controle em aeronaves ou indústrias e, até mesmo, sistemas de controles aeroespaciais. Com o avanço da tecnologia, cada vez mais serão criados produtos que contenham sistemas embarcados, e à medida que aumenta a quantidade e variedade destes produtos, há acréscimo de complexidade destes sistemas devido à integração de mais componentes de *software* e de *hardware* ao sistema (GRAAF, 2003) e, consequentemente, acréscimo de uma arquitetura de comunicação que permita tal integração. Segundo Wolf (2001), sistemas embarcados apresentam, além dos requisitos funcionais, alguns requisitos não-funcionais, tais como: minimização do consumo de energia; redução do peso do equipamento e da área ocupada; diminuição do custo. Eles possuem, por outro lado, restrições que delimitam o seu escopo, tais como: consumo máximo de energia, número máximo de pinos de entrada e saída, etc. O projeto de um sistema embarcado é guiado por estes requisitos e restrições, que determinam a natureza dos elementos computacionais que o implementarão.

1.1 SoCs (*System-on-Chip*) e MPSoCs (*Multiprocessor* SoCs)

O advento da tecnologia submicrônica tem permitido um aumento da integração de transistores em uma mesma pastilha de silício e, conseqüentemente, possibilitará a sustentação da Lei de Moore (MOORE, 1975)(SCHALLER, 1997) por ainda muitos anos. Essa lei caracteriza este advento, que já na década de 70, quando os primeiros microprocessadores foram desenvolvidos predizia que a indústria de microeletrônica avançaria a tal ponto de poder dobrar o nível de integração dos componentes em um chip a cada 18 meses. Este avanço tecnológico tem possibilitado a conexão e integração de múltiplos componentes, como processadores, memórias e controladores, aumentando

a complexidade e funcionalidades dos sistemas embarcados, resultando em um sistema completo em uma mesma pastilha de silício (ITR, 2005). Esses sistemas são denominados SoCs (em inglês, *System-on-Chip*) que passam a ser chamados de MPSoC (*Multiprocessor SoC*) se vários desses elementos são processadores. SoC é o termo dado à generalização do MPSoC.

Uma indústria que desenvolve SoCs deve atender às pressões do mercado como, por exemplo, curto *time-to-market* e maior redução do tempo de vida do produto (KORDON, 2003). Geralmente o tempo de projeto é de alguns meses. Conforme a Figura 1.1 (BERGAMASCHI, 2001), o atraso de poucas semanas no lançamento de um produto pode comprometer seriamente os ganhos esperados de um novo produto no mercado. Além desta pressão mercadológica, existem outros problemas, como por exemplo os custos de engenharia não recorrentes (NRE – *Non-Recurring Engineering*). O projeto de um sistema embarcado de grande complexidade é bastante caro para uma empresa, envolvendo equipes multidisciplinares (hardware digital, hardware analógico, software, teste) e a utilização de ferramentas computacionais de elevado custo. São elevados especialmente os custos de fabricação de sistemas integrados numa pastilha.

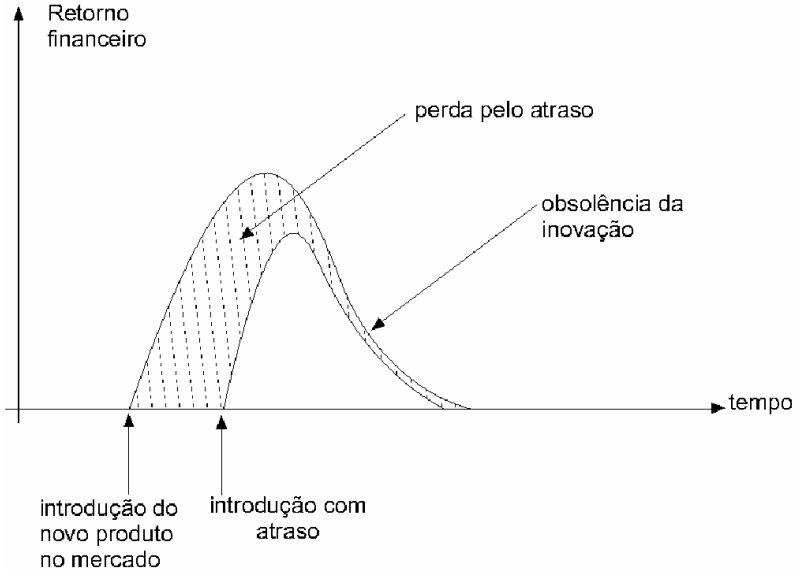


Figura 1.1: Retorno financeiro da introdução de produtos no mercado em relação ao tempo.

Em muitas aplicações, é adequada a integração do sistema em uma única pastilha (SoC). Em situações onde requisitos de área, potência e desempenho sejam críticos, o projeto do SoC na forma de um ASIC (circuito integrado para aplicação específica) pode ser mandatório, elevando bastante os custos de projeto e fabricação. Devido à heterogeneidade de um SoC, ferramentas para projeto no nível de sistema devem ser desenvolvidas com o objetivo de obter uma forma mais adequada para especificação, simulação e síntese de tais sistemas (CARRO, 2003). Além disso, é importante que os componentes integrados em SoC sejam reutilizáveis para a amortização dos custos do projeto entre vários sistemas. É importante que as metodologias de projeto adotadas sejam baseadas no reuso de componentes pré-projetados, pré-caracterizados e pré-verificados. Estes núcleos são denominados núcleos de propriedade intelectual, núcleos IP (do inglês, *Intellectual Property*) ou simplesmente *cores*. Um núcleo IP é um módulo de hardware, digital ou analógico, podendo ser descrito em diferentes níveis de abstração (MADISETTI, 1997). Estes núcleos são pré-projetados, pré-verificados e

prototipados em hardware pelo menos uma vez. Podem ser desenvolvidos pela empresa responsável pelo projeto do sistema ou adquiridos de terceiros. Nestas novas metodologias, o projetista pode concentrar-se no sistema completo sem ter que se preocupar com a funcionalidade interna ou com o desempenho de componentes individuais. Conforme as estimativas da indústria de semicondutores, o percentual de reuso em CIs será de 90% em 2012 (SAI, 2005).

Entretanto, um SoC não é apenas um chip composto por múltiplos núcleos, viabilizado pela crescente densidade de transistores das tecnologias de fabricação. Mais do que isso, SoCs são arquiteturas customizadas, que equilibram as restrições da tecnologia e as necessidades das aplicações, geralmente de um domínio específico (JERRAYA, 2005). Como conseqüência dessas restrições, os SoCs assumem um caráter altamente heterogêneo e acabam por requerer estruturas de comunicação bastante elaboradas.

Assim, os núcleos de um SoC são interconectados por uma estrutura de comunicação denominada arquitetura de comunicação. Tais arquiteturas de comunicação atualmente utilizadas (por exemplo, barramentos (ARM, 2003) (IBM, 2004)) podem se tornar o gargalo no projeto dos futuros SoCs (KUMAR, 2003), onde dezenas ou centenas de núcleos vão compor um determinado sistema computacional. As atuais estruturas de comunicação oferecem pouca escalabilidade e pouca ou até ausência de paralelismo. Consequentemente, com o aumento do número de núcleos de hardware conectados aos canais do barramento, a carga capacitiva desses canais é incrementada, resultando em um aumento no tempo e na energia necessárias à propagação dos sinais pelos fios do barramento. Segundo Benini et al. (2001), a interconexão por barramento é simples, sob o ponto de vista de implementação, apresentando, entretanto, diversas desvantagens: (i) apenas uma troca de dados pode ser realizada por vez em cada barramento, pois o meio físico é compartilhado por todos os núcleos de hardware, reduzindo o desempenho global do sistema; (ii) necessidade de mecanismos inteligentes de arbitragem do meio físico para evitar desperdício de largura de banda; (iii) a escalabilidade é limitada, ou seja, a quantidade de núcleos de hardware que podem ser conectados a cada barramento é muito baixa, tipicamente na ordem de dezenas; (iv) o uso de linhas globais em um circuito integrado com tecnologia submicrônica impõe sérias restrições ao desempenho do sistema devido às altas capacitâncias e resistências parasitas inerentes aos fios longos.

Nos futuros sistemas, os comprimentos dos fios de um barramento irão se manter proporcionais à pastilha de silício (*die size*) e não irão diminuir com o aumento da freqüência de relógio devida à redução dos transistores. Quanto à potência, o problema é que o barramento opera por difusão e cada sinal deve chegar a todos os núcleos, em todos os pontos de acesso, exigindo grande quantidade de energia.

No intuito de resolver os problemas inerentes aos futuros SoCs, diversos autores (LIANG, 2000) (GUERRIER, 2000) (DALLY, 2001) (BENINI, 2002) (KUMAR, 2002) (KARIM, 2001) (SGROI, 2001) (SAASTAMOINEN, 2002) (ZEFERINO, 2003a) (ZEFERINO, 2003b) (MORAES, 2003) propõem o uso de uma rede de interconexões chaveadas dentro do chip. Dally e Towles (2001) analisaram várias infra-estruturas de comunicação e concluíram que as redes-em-chip são adequadas para projetos modulares devido a sua alta escalabilidade e regularidade com baixo custo extra de área se comparado com soluções *ad hoc*.

As principais vantagens desse tipo de rede com relação às outras arquiteturas de comunicação usadas são: (*i*) aumento do reuso, tanto dos núcleos como da plataforma de comunicação; (*ii*) desenvolvimento de um sistema GALS (*Global Asynchronous Local Synchronous*), eliminando o problema de sincronização do relógio em todo o sistema e permitindo que os núcleos trabalhem com diferentes relógios; (*iii*) freqüência constante do relógio com o aumento de núcleos do sistema; (iv) mais paralelismo, (v) mais escalabilidade e maior eficiência em termos de consumo de energia. Essas vantagens são apresentadas em maior detalhe na próxima seção.

1.2 Redes-em-chip

Segundo Rijpkema et al. (2003), *Micronetworks* (TEWKSBURRY, 1992) ou *Networks-on-chip* (NoC) (HERMANI, 2000) estão emergindo como uma possível solução para os problemas associados às estruturas de interconexão, devido a características como:

- *Escalabilidade*: maior largura de banda nos canais, com isso podendo-se conectar mais núcleos de propriedade intelectual na rede-em-chip;

- *Paralelismo:* na rede-em-chip existem vários canais de comunicação que permitem o tráfego de mensagens de maneira paralela. Assim, existem menos colisões de mensagens no sistema, aumentando significativamente o desempenho global do sistema;

- *Eficiência em termos de consumo de energia*: fios curtos entre roteadores na rede-em-chip tendem a reduzir ou eliminar as capacitâncias parasitas e com isso a diminuir o consumo de energia;

- *Reusabilidade*: permite conexão de núcleos IP na rede-em-chip sem adaptações ou com poucas adaptações no protocolo de comunicação entre os núcleos IP e a rede-em-chip.

Além dessas características para possível solução de problemas associados a atuais estruturas de comunicação, as redes-em-chip ocupam significativa área da pastilha, levando o projetista a considerar esta característica entre outras no que diz respeito ao desenvolvimento de um SoC usando tal arquitetura de comunicação.

A arquitetura de comunicação baseada em redes-em-chip é bem mais recente que os barramentos e por este motivo o ferramental disponível para a síntese de sistemas através deste recurso ainda é incipiente. Este é um dos principais motivos pelas quais as redes-em-chip compõem uma parcela ainda muito pequena das plataformas existentes.

No desenvolvimento de sistemas baseados em redes-em-chip, estas devem atender os requisitos de projeto. Para atingir este objetivo, é desejável que as redes-em-chip sejam otimizadas para uma dada aplicação, altamente especializadas para o atendimento das restrições da mesma ou, então, que existam mecanismos de adaptação dinâmica para aplicações menos específicas/mais genéricas. As redes-em-chip permitem a exploração de diversas características para encontrar uma determinada configuração que atenda os

requisitos do projeto. A sua alta escalabilidade é um dos principais motivos para a maior quantidade de trabalhos relacionados à exploração de espaço de projeto (BOLOTIN, 2004) (HU, 2003)(MURALI, 2004a)(MURALI, 2004b) e (YE, 2002).

1.3 Exploração de espaço de projeto

Sistemas embarcados, em geral, demandam vários tipos de restrições, como desempenho, energia, potência e custo, entre outros. O projeto do sistema embarcado deve ser concretizado de acordo com as especificações de projeto e restrições. Para projetar um sistema embarcado que consiga respeitar todos os requisitos de projeto, os projetistas devem levar em conta tipos de processadores, tipos de blocos de hardware dedicados, carga de software em cada processador, tamanhos de memórias, tamanhos de caches, organização da memória, etc. Variando estas características, existe uma vasta exploração de espaço de projeto (EEP) para desenvolver um sistema embarcado que atenda os requisitos impostos para uma dada função-objetivo. Contudo, para fazer esta exploração automática, usar computação exaustiva é proibitivo pelo tempo de processamento. Para encontrar resultados sub-ótimos em um tempo viável, são utilizadas técnicas heurísticas para busca de melhores alternativas de projetos, diminuindo o tempo de exploração para amplos espaços de projetos (muitas variáveis para exploração espacial). A busca de alternativas na exploração de espaço de projeto é um problema NP-completo, onde a inserção de um novo componente no projeto a ser explorado aumenta fatorialmente o tempo de exploração.

Os métodos de exploração de espaço de projeto pesquisados recentemente na literatura são usualmente aplicados em tempo de projeto, supondo-se que é conhecido o perfil das aplicações que devem ser executadas pelo sistema embarcado. No entanto, cada vez mais sistemas embarcados aproximam-se de dispositivos genéricos de processamento (como palmtops), onde as tarefas a serem executadas não são inteiramente conhecidas a priori. Com a mudança dinâmica da carga de trabalho de um sistema embarcado, a busca pelo atendimento de requisitos (atendimento de *deadlines*, minimização no consumo de energia) pode então ser enfrentada por mecanismos adaptativos, que implementam dinamicamente a EEP. A adaptabilidade ou reconfiguração dinâmica no sistema embarcado pode ser realizada nas dimensões do hardware e do software. Em hardware, componentes que se reconfiguram ao longo da execução do sistema podem, por exemplo, economizar energia. Em software, a adaptabilidade pode ser feita através de um balanceamento de carga de tarefas entre os processadores, ou pela mudança nas prioridades de mensagens numa rede-em-chip, a fim de atender requisitos do sistema como consumo de energia e desempenho. Em um sistema de tempo real soft, a migração de tarefas pode diminuir o número de prazos de execução das tarefas (*deadlines*) perdidos, desde que o custo de migração seja amortizado. Tarefas que estavam sendo executadas num processador mais simples (com menos consumo de energia) podem precisar migrar para processadores mais rápidos, ou mesmo até passarem a ser executadas por blocos dedicados de hardware, que numa situação menos crítica podem permanecer desligados.

Os mecanismos adaptativos aqui envolvidos podem ser baseados em heurísticas utilizadas na área de Sistemas Computacionais Distribuídos, que se preocupa apenas em tornar eficiente o desempenho do sistema. A natureza de comunicação entre nodos na área de sistemas distribuídos é semelhante à de um sistema baseado em rede-em-chip. Entretanto, em sistemas baseados neste último, restrições de projeto tais como energia e potência tornam a exploração mais complexa, pois todas essas restrições geralmente são

conflitantes. O propósito das heurísticas distribuídas pode ser estendido para a área dos Sistemas Embarcados, considerando a minimização e atendimento de figuras como energia, potência e deadlines das tarefas em sistemas de tempo real. Essas heurísticas, ainda, podem contribuir para a melhor distribuição de carga, para balanceamento de tarefas em elementos processantes, evitando altos *hot-spots*, ou seja, grandes diferenças térmicas entre os elementos processantes. Essas grandes diferenças podem acarretar em problemas no nível submicrônico (HUNG, 2004).

Além disso, com o uso de tais técnicas de EEP em tempo de execução aplicadas em uma plataforma baseada em rede-em-chip, é possível que esta possa abranger vários mercados (telecom, automotivo, multimídia, etc), desde que esta plataforma consiga equiparar ou suplantar as plataformas existentes no mercado. Assim, várias aplicações ou domínios de aplicações podem ser executados nesta plataforma por diferentes fabricantes, sem a preocupação do projetista sobre a alocação de tarefas ao longo da execução da aplicação. O sistema se adapta de acordo com as restrições designadas e, dessa forma, obtém-se eficiência na execução, de acordo com as heurísticas executadas dinamicamente e com as figuras de mérito envolvidas tais como: desempenho, energia e potência.

A Figura 1.2 ilustra a motivação do emprego de mecanismos em tempo de execução para prover exploração de espaço de projeto em um sistema embarcado do tipo *handheld* (*Palm Top*, por exemplo). Inicialmente, o sistema está no modo inativo e permanece como tal durante 30 minutos. Subitamente, o usuário escolheu uma aplicação de entretenimento. E assim permaneceu durante mais 30 minutos. Finalmente, o usuário selecionou outra aplicação (por exemplo, um filme) e assim as duas aplicações rodam em paralelo durante mais 30 minutos. As curvas no gráfico apresentam o modo aleatório, ou seja, alocação das tarefas das aplicações sem nenhuma otimização ou técnica para exploração de espaço de projeto e a curva EEP dinâmica, que provê mecanismos de alocação de tarefas levando em conta redução de energia, minimização de perdas de deadlines, etc.

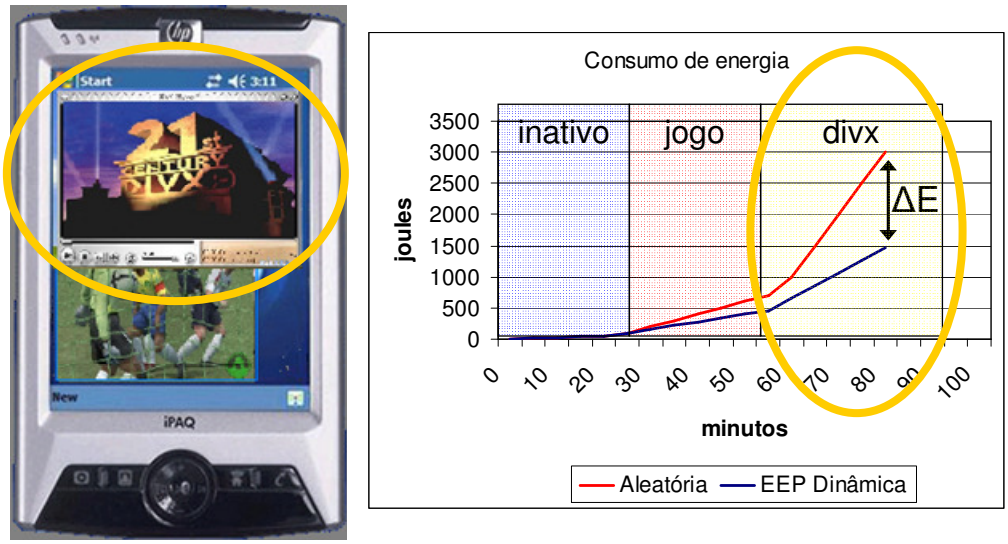


Figura 1.2: Motivação para o uso de mecanismos em tempo de execução para prover exploração de espaço de projeto.

A Figura 1.2 apresenta a diferença de energia entre os dois métodos ao longo dos 90 minutos (ΔE), e esta diferença é cada vez mais significativa em relação ao tempo. A motivação do uso de mecanismos para habilitação de EEP em tempo de execução não se

resume apenas à minimização de energia e diminuição de taxas de perdas de deadlines, mas sim pode ser estendido para outros propósitos tais como: aumento da largura de banda, diminuição de *hot-spots* (diferença térmica entre os núcleos IP), diminuição de potência, tolerância a falhas, etc. Os resultados apresentados na Figura 1.2 foram extraídos através de simulações de aplicações sintéticas. O comportamento das aplicações é baseado na premissa de que cada tarefa tem alta taxa de computação e comunicação. Desta forma, as aplicações mostradas na Figura 1.2 são meramente ilustrativas, mas com caráter informativo e motivacional.

Atualmente, na literatura, existem muito poucos trabalhos relacionados à exploração de espaço de projeto realizado em tempo de execução em arquiteturas de comunicação baseadas em redes-em-chip. Estes trabalhos focam em aumento de desempenho e largura de banda, porém existem poucos trabalhos que abordam minimização de perdas de deadlines e consumo de energia.

1.4 Objetivo do trabalho

O objetivo proposto neste trabalho é o provimento de metodologias para exploração de espaço de projeto e para atendimento de restrições em tempo de execução em sistemas embarcados baseados em redes-em-chip. Estes métodos para atendimento de restrições implicam heurísticas baseadas no âmbito dos Sistemas Computacionais Distribuídos para compartilhamento e balanceamento de carga, migração de tarefas, e métodos baseados na área de Sistemas Eletrônicos Embarcados de economia de energia como o escalonamento de tensão e desligamento de elementos processantes. Todos esses métodos de exploração serão aplicados à topologia de rede grelha 2d, para avaliação dos custos dos mecanismos de exploração. Deste modo, será possível obter resultados que permitam que o projetista possa escolher, em tempo de projeto, uma combinação adequada de métodos de atendimento de restrições *on-line* para uma determinada aplicação ou domínio de aplicações.

Na academia, não se tem conhecimento de trabalhos da área que utilizam a migração de tarefas para tentar minimizar o problema de *deadlines* perdidos e o consumo de energia simultaneamente em sistemas embarcados baseados em redes-em-chip, pois outros trabalhos atendem apenas requisitos de desempenho e energia.

Utilizando metodologias de EEP em tempo de execução, esse trabalho pretende aplicar heurísticas para atendimento de restrições energéticas e de tempo-real em um sistema embarcado baseado em redes-em-chip. Além disso, pretende-se ter resultados que permitam o desenvolvimento de sistemas embarcados baseados em redes-em-chip otimizados dinamicamente para dados requisitos de projeto e carga de trabalho.

Como parte do trabalho, foram determinados e utilizados estudos de caso que, em alguns deles, envolvam necessidades de alto desempenho que justifiquem o uso de redes-em-chip e que apresentem uma característica de comportamento dinamicamente modificável.

1.5 Organização do volume

O restante deste texto está organizado em mais seis capítulos.

O Capítulo 2 apresenta diversas arquiteturas de comunicação intrachip utilizadas no âmbito dos Sistemas Embarcados e redes-em-chip atuais pesquisadas na academia. Aqui é realizada a justificativa da escolha da rede-em-chip empregada neste trabalho.

O Capítulo 3 apresenta os principais trabalhos de exploração de espaço de projeto baseados em redes-em-chip, bem como um levantamento da bibliografia básica para migração de tarefas em sistemas distribuídos. No final do Capítulo 3, é realizada uma análise sobre a possibilidade do emprego de mecanismos dinâmicos de exploração no contexto dos Sistemas Embarcados.

O Capítulo 4 apresenta algumas heurísticas, utilizadas neste trabalho, para alocação de tarefas em sistemas multiprocessados. Essas heurísticas são utilizadas dentro do escopo da área dos Sistemas Computacionais Distribuídos, a qual concede conceitos para a área de Sistemas Embarcados.

No Capítulo 5 são apresentados os modelos de tarefas, roteadores, processadores e componentes para gerenciar e migrar tarefas utilizados na implementação do simulador Serpens.

O Capítulo 6 apresenta diversos experimentos considerando os conceitos apresentados nos Capítulos 3, 4 e 5. Vários experimentos foram realizados para justificar um nível de eficiência aceitável dos algoritmos apresentados no Capítulo 4. Cada experimento apresenta os resultados de consumo de energia e perdas de *deadlines*, obtidos com a simulação das respectivas alocações no simulador Serpens, descrito no Capítulo 5.

Finalmente, o Capítulo 7 encerra este documento apresentado considerações finais referentes ao trabalho, apresentando resumo das contribuições da tese, conclusões e considerações sobre trabalhos futuros.

2 ARQUITETURAS DE COMUNICAÇÃO INTRACHIP

Um sistema integrado não é constituído apenas por núcleos. Conforme Madisetti e Shen (1997), ele é formado por um conjunto de núcleos integrados através de uma rede de interconexão. A Figura 2.1 ilustra uma arquitetura genérica de um SoC. Os núcleos são integrados através de uma rede de interconexão comercial ou adaptada, com pelo menos um controlador (PPi) e funções de interface com o meio externo. Caso os núcleos sejam obtidos de diferentes fontes, a integração dos módulos e o teste do sistema podem ser difíceis, podendo até haver necessidade de que os mesmos sejam re-projetados, para adequá-los a um protocolo de interface comum.

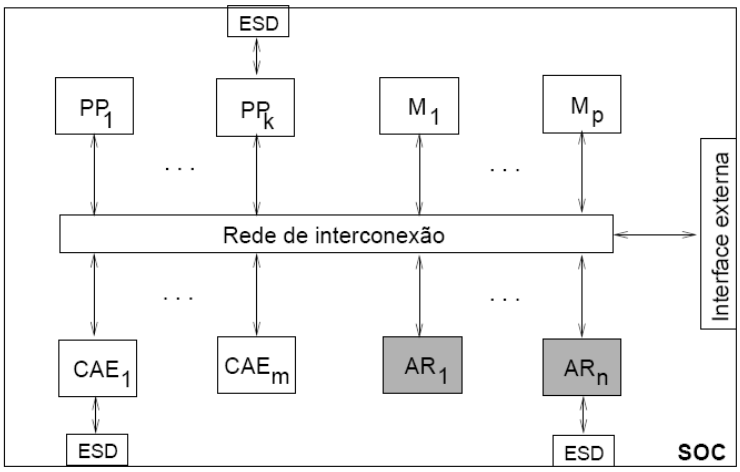


Figura 2.1: Arquitetura genérica de um SoC.

Tais núcleos de hardware são interconectados por uma arquitetura de comunicação. Esta arquitetura deve permitir a troca de mensagens entre núcleos a ela conectados. As arquiteturas de comunicação são divididas em dois grandes grupos: (*i*) arquiteturas de comunicação baseadas em conexões ponto a ponto e (*ii*) arquiteturas de comunicação baseadas em conexões multiponto (MARCON, 2005). Redes-em-chip são exemplos das primeiras e barramentos são exemplos das últimas. O objetivo desta seção é

apresentar algumas arquiteturas de comunicação intrachip e suas características, usadas na academia e na indústria.

2.1 Arquiteturas de Comunicação de Núcleos Baseadas em Conexões Multiponto

A comunicação baseada em conexões multiponto permite que mais de um núcleo compartilhe de um mesmo canal físico da arquitetura de comunicação para a escrita ou leitura de informações. Entretanto, o compartilhamento para escrita deve ser realizado em tempos distintos, ou seja, o compartilhamento da arquitetura de comunicação é temporal. Embora as conexões multiponto permitam apenas um escritor em um dado intervalo de tempo, pode existir mais de um leitor durante este intervalo. O compartilhamento temporal é um fator limitante do paralelismo da aplicação, sendo que este limite é dado pela serialização da comunicação na arquitetura de comunicação. Esta limitação pode reduzir o tempo global de execução para muitas aplicações.

Uma arquitetura de comunicação típica que utiliza comunicação multiponto é o barramento, ilustrado na Figura 2.2. Nestes, vários *tiles*[1] estão conectados através de chaves a um meio físico comum, o barramento. Em geral, para este trabalho, as chaves são os pontos de acesso e cada *tile* pode conter um núcleo.

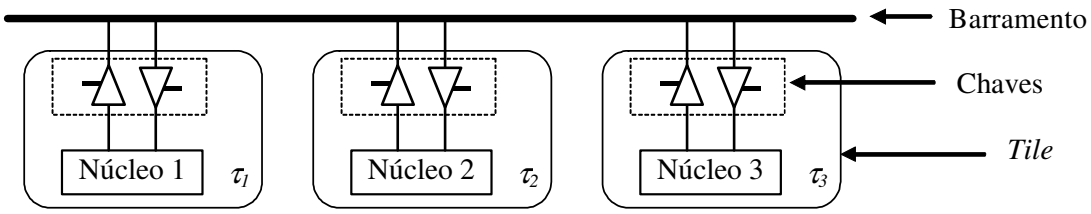


Figura 2.2: Módulos de comunicação interligados por barramento.

Outra arquitetura de comunicação multiponto é o barramento segmentado, em inglês *split bus*, que envolve a segmentação de um barramento monolítico em múltiplos segmentos com *portas de roteamento*[2] (p_R). Ao segmentar um barramento, cada segmento conectará um número menor de núcleos e o comprimento de cada segmento pode ser menor, e em conseqüência, o consumo de energia pode ser reduzido e a freqüência de operação pode ser aumentada. Além do mais o paralelismo é aumentado, uma vez que a comunicação dentro de cada segmento independe das comunicações nos outros segmentos.

---

[1] Área da arquitetura alvo onde são implementados os componentes de hardware/software (núcleos) e o ponto de acesso a uma arquitetura de comunicação da mesma arquitetura alvo.

[2] Recurso de comunicação utilizado para conectar dois ou mais segmentos de barramento.

A Figura 2.3 mostra um barramento monolítico um barramento segmentado com a mesma funcionalidade. Neste exemplo, a porta de roteamento que interliga os segmentos é implementada por uma porta bidirecional. Quando a porta é habilitada, a funcionalidade do barramento segmentado se assemelha à do barramento monolítico (HSIEH, 2002).

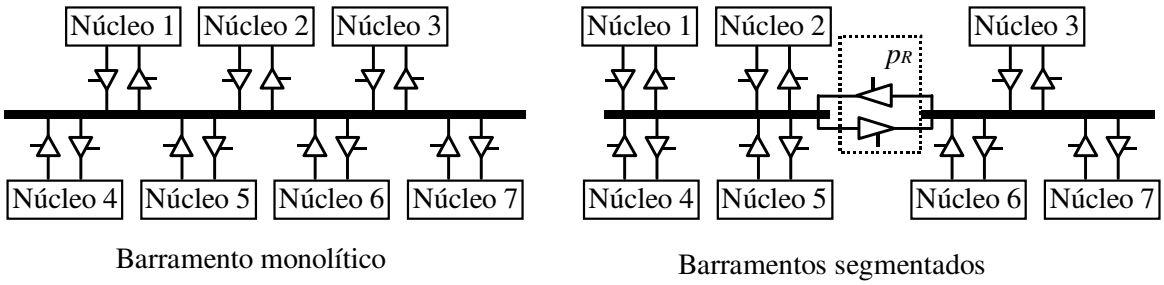


Figura 2.3: Equivalência de sistema com barramento monolítico com um sistema com barramentos segmentados.

Casos típicos de barramentos segmentados encontrados na literatura são: AMBA da ARM (2003), Avalon da Altera (2002) e CoreConnect da IBM (2004). Geralmente, estas arquiteturas de barramentos estão vinculadas à arquitetura de um processador, tal como o AMBA vinculado ao processador ARM, o CoreConnect vinculado ao processador PowerPC e o Avalon vinculado ao processador Nios.

2.2 Arquiteturas de Comunicação de Núcleos Baseada em Conexões ponto a ponto

Na comunicação baseada em conexões ponto a ponto, ilustrada na Figura 2.4, os núcleos são interligados diretamente. Esta é uma arquitetura de comunicação que oferece alto grau de paralelismo, pois entre cada *tile* a comunicação ocorre de forma independente, ou seja, a comunicação entre núcleos é realizada através de canais de comunicação exclusivos. Este grau de paralelismo confere a este tipo de arquitetura um formato de comunicação conhecido como espacial.

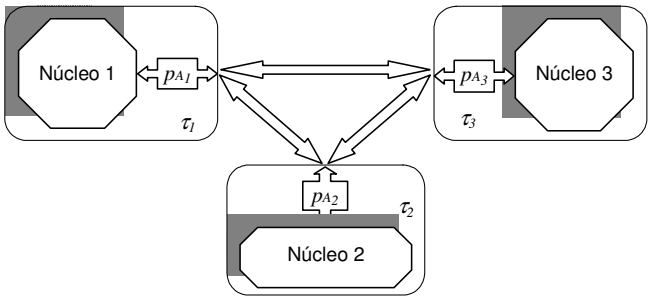


Figura 2.4: Arquitetura de comunicação baseada em conexão ponto a ponto.

A infra-estrutura de comunicação dedicada ponto a ponto é eficaz para a intercomunicação entre um pequeno número de núcleos. Todavia, o número de

conexões dedicadas aumenta proporcionalmente com o quadrado do número de núcleos, podendo gerar congestionamento de conexões para sistemas maiores (ZEFERINO, 2003a) (ZEFERINO, 2003b). Este fator e a sua irregularidade não conferem escalabilidade para esta infra-estrutura. A escalabilidade é um fator limitante à medida que cresce a complexidade dos projetos computacionais, sendo que se projeta a existência de SoCs com dezenas a centenas de núcleos (KUMAR, 2003)(ITRS, 2005).

2.2.1 Redes-em-Chip

Uma rede-em-chip é uma arquitetura de comunicação normalmente projetada de forma ponto a ponto. Neste trabalho, define-se rede-em-chip como um conjunto de roteadores (em inglês, *routers*), conectados entre si por canais de comunicação (em inglês, *links*), que são o meio físico pelo qual as informações trafegam. A forma como os roteadores estão conectados entre si, e como os núcleos estão conectados aos roteadores, define a topologi*a* da rede. Um roteador é um dispositivo que transfere informações disponíveis nos seus canais de entrada para os seus canais de saída, através de portas de comunicação (NI, 1993). O intervalo de tempo entre a entrada e a saída de uma informação do roteador é denominado de atraso de roteamento ou latência. Geralmente, a estrutura de um roteador consiste de uma estrutura de chaveamento entre os canais de entrada e saída, um módulo de controle de chaveamento, e elementos de armazenamento temporário dos dados nos canais de entrada e/ou de saída. A estrutura de um roteador genérico é ilustrada na Figura 2.5.

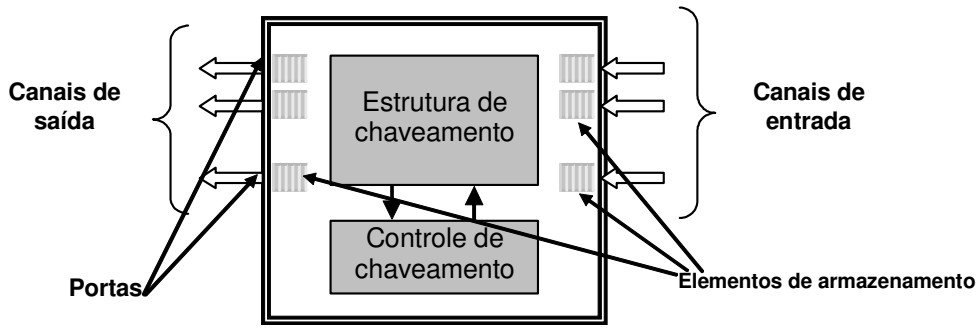


Figura 2.5: Estrutura básica de um roteador.

Uma arquitetura de comunicação deve proporcionar a transferência de informações de um núcleo origem para um núcleo destino. A transferência de informações entre os núcleos se dá através da troca de mensagens efetivada através do envio de pacotes (BENINI, 2002)(DALLY, 2004)(DUATO, 1997)(GUERRIER, 2000). Uma mensagem consiste de um conjunto de informações a ser transmitido do núcleo origem para o núcleo destino. Um pacote é a unidade de transmissão de informação empregada pelo meio de comunicação para transmitir mensagens. Pacotes podem representar frações de uma mensagem, uma mensagem inteira ou ainda múltiplas mensagens. Pacotes são normalmente constituídos de um cabeçalho (em inglês, *header*), corpo (em inglês, *payload*) e finalizador (em inglês, *trailer*). Em geral, o cabeçalho contém dados úteis para o roteamento, o corpo da mensagem contém dados úteis ao núcleo destino e o finalizador contém dados que garantem a coerência do pacote que está sendo enviado.

A topologia da rede consiste na organização da mesma sob a forma de um grafo, no qual os roteadores são os vértices deste e os canais de comunicação os arcos (NI, 1993). Citam-se como exemplos de topologias de rede: malha (em inglês, *mesh*), toro-dobrado (em inglês *folded torus*) e hipercubo, ilustradas na Figura 2.6.

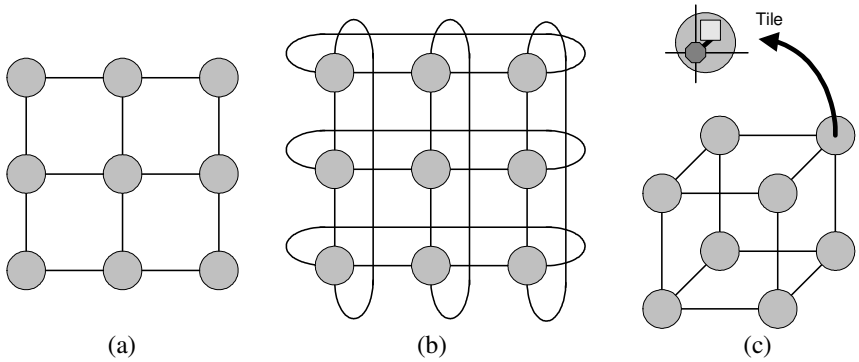


Figura 2.6: Exemplos de topologias estáticas: (a) malha, (b) toro-dobrado (c) hipercubo.

Redes diretas são aquelas cuja topologia implica roteadores com canal de comunicação com o núcleo local. Em contraposição, as redes indiretas são aquelas cuja topologia implica roteadores que não têm comunicação com um núcleo local (DALLY, 2004) (DUATO, 2003). Um melhor detalhamento do *tile* de uma rede com topologia malha está ilustrado na Figura 2.7. Tanto o roteador quanto seu núcleo local estão fisicamente posicionados em *tiles*, que são os nodos da rede. O roteador faz o papel do ponto de acesso. O que difere esta infra-estrutura de comunicação das infra-estruturas com conexão dedicada ponto a ponto é exatamente a capacidade do ponto de acesso de direcionar as informações não apenas para o núcleo local ao *tile*, mas também para outros pontos de acesso, sem que o núcleo local tome conhecimento da informação transmitida.

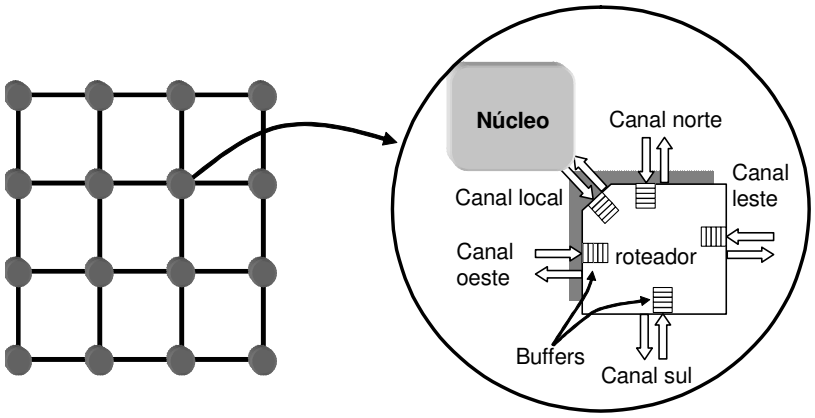


Figura 2.7: Exemplo de um roteador de uma rede com topologia malha.

O algoritmo de roteamento define o caminho a ser utilizado por um pacote a partir do seu núcleo de origem até o seu núcleo destino. Os algoritmos de roteamento podem ser classificados de acordo com seu objetivo.

A responsabilidade pela definição do roteamento pode ser distribuída, quando o próximo destino do roteamento de um pacote ou mensagem é definido a cada roteador visitado; fonte, quando um roteador decide sozinho a sua rota, geralmente o remetente, e finalmente, centralizada, onde caminhos são estabelecidos por um controlador central de rede. O método fonte tem a desvantagem de necessitar que o cabeçalho da mensagem contenha toda a rota do pacote.

Os algoritmos de roteamento podem ser classificados em determinísticos ou adaptativos. No roteamento determinístico existe apenas uma rota possível entre um par remetente/destinatário, enquanto que no roteamento adaptativo existem mais opções de rotas. Nesta abordagem, o caminho de um pacote é estabelecido dependendo das condições da rede, como tráfego e congestionamento de canais.

Um algoritmo de roteamento determinístico e livre de *deadlocks* bastante utilizado na literatura em redes malha é o XY (DUATO, 1997). Nele, o pacote é roteado em X até atingir a coluna destino e depois é roteado em Y, até o destino da mensagem, como demonstrado na Figura 2.8. A distância entre os nodos a e b, nesse esquema, é chamada de distância de Manhattan e é dada por:

Equação 2.1: $dist = | x_b - x_a | + | y_b - y_a |$

O roteamento adaptativo pode ser ainda classificado como parcialmente e totalmente adaptativo. Na classificação totalmente adaptativo é possível rotear um pacote através de qualquer caminho físico existente, enquanto que na classificação parcialmente adaptativo apenas um subconjuntos dos caminhos disponíveis é considerado.

O algoritmo de roteamento também pode ser mínimo ou não-mínimo. No primeiro caso, o pacote deve aproximar-se do destino após cada roteador visitado; enquanto que, no segundo, um pacote pode ser enviado para um caminho que o deixa mais longe do destino.

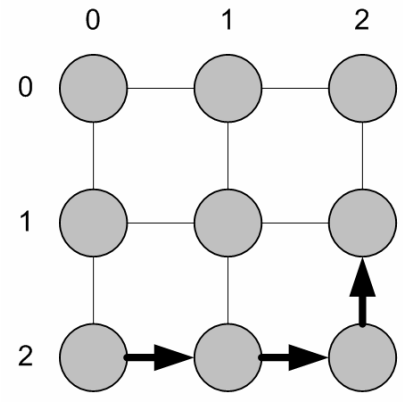


Figura 2.8: Caminho de um roteamento XY entre (0,2) e (2,1).

A escolha da forma pelo qual pacotes são transferidos da entrada de um roteador para um de seus canais de saída é o que determina o comportamento das chaves internas de cada roteador. Dois métodos de transferência de pacotes são utilizados: chaveamento de circuito e chaveamento de pacotes.

- No C*haveamento de circuitos* (em inglês, *circuit switching*), inicialmente é reservado um caminho do núcleo origem até o núcleo destino, para posteriormente serem enviados todos os pacotes da mensagem;

- No C*haveamento de pacotes* (em inglês, *packet switching*), cada pacote informa a cada roteador qual o destino que irá seguir na rede, ou seja, não existe um caminho pré-definido.

O emprego de chaveamento de pacotes implica o uso de modos de chaveamento. Entre os principais utilizados citam-se *store-and-forward*, *virtual cut-through* e *wormhole*, definidos a seguir:

1. No modo *store-and-forward* um pacote tem que ser completamente armazenado em um roteador antes de ser enviado para o próximo roteador;
2. No modo *virtual cut-through*, um roteador pode enviar um pacote a partir do momento que o próximo roteador garanta que possa receber todo o pacote;
3. No modo *wormhole* os pacotes são quebrados e transmitidos entre os roteadores em pequenas unidades, denominadas *flits*[3]. O modo de chaveamento *wormhole* é uma variação do virtual *cut-through* que permite utilizar menos armazenamento intermediário. Uma desvantagem associada a este modo é que apenas o flit de cabeçalho contém informações sobre o endereçamento destino. Logo, os demais flits que compõem o pacote devem seguir o mesmo caminho reservado para o cabeçalho. Se um cabeçalho não puder avançar na rede em função de um congestionamento, todos os flits restantes são bloqueados ao longo do caminho, até que este seja liberado.

O controle de fluxo trata da alocação de recursos da rede, como buffers e canais. Esses recursos são necessários para que uma mensagem possa percorrer a rede. O controle de fluxo realiza a regulação de tráfego nos canais. Se o buffer de entrada do receptor estiver cheio, o transmissor irá manter os dados no seu buffer até que o recurso ocupado no receptor esteja livre. Existem três alternativas usualmente utilizadas na implementação do controle de fluxo (ZEFERINO, 2003a): (*i*) *handshake*; (*ii*) controle baseado em canais virtuais e (*iii*) controle baseado em créditos.

1. O controle de fluxo do tipo *handshake* consiste em duas linhas, uma de validação e outra de reconhecimento (*acknowledge*). Pela linha de validação o nodo emissor avisa que possui mensagem para enviar. O nodo receptor irá confirmar se há espaço em seu buffer através da linha de reconhecimento, estabelecendo a transferência da mensagem.
2. O controle de fluxo baseado em canais virtuais é utilizado em redes de interconexão com chaveamento *wormhole* e foi concebido com o objetivo de resolver os problemas de deadlock nestas redes. Esse método consiste em dividir o buffer de entrada em filas independentes de profundidades menores que irão formar os canais-virtuais, procurando desta forma evitar o bloqueio de cabeça de linha (HoL – do inglês, *Head-of-Line blocking*). Este bloqueio ocorre quando, durante a transmissão de vários flits seguidos, o

[3] *Flit* é uma unidade lógica do modo de chaveamento *wormhole* que em redes-em-chip tem o mesmo tamanho de um *phit*. O *phit* é uma unidade física que indica o número de fios físicos que transmitem informações úteis entre dois roteadores.

buffer de entrada do receptor enche, provocando o bloqueio do canal físico por onde o pacote é transmitido. O uso de canais virtuais permite uma maior utilização do canal físico, que poderá usar um outro canal virtual em caso de bloqueio.

3. No controle de fluxo baseado em créditos, uma transmissão só ocorre se houver espaço no buffer do receptor. O protocolo opera baseado no número de créditos, que corresponde ao espaço no buffer que o receptor possui. Assim, o transmissor recebe informação relativa ao número de créditos do receptor e, se este possuir créditos, a mensagem é enviada e o número de créditos é decrementado. Se a mensagem é passada adiante, esvaziando uma posição do buffer, o número de créditos é incrementado novamente.

2.2.1.1 Modelo de energia

Ye, Benini *et al.* (2002) propuseram um modelo de estimativa de energia dinâmica de roteadores de redes-em-chip através da definição da métrica Energia de Bit (E_{bit}), como sendo a energia consumida quando um bit de dados é transferido através de um roteador. O E_{bit} é calculado da seguinte forma:

Equação 2.2: $E_{bit} = E_{S_{bit}} + E_{B_{bit}} + E_{L_{bit}}$

onde $E_{S_{bit}}$, $E_{B_{bit}}$ e $E_{L_{bit}}$ representam a energia consumida na matriz de chaveamento, nos *buffers* e no canal de comunicação, respectivamente. Com base na equação Equação 2.2, a energia consumida em mandar um bit de dados de um núcleo a outro, com d roteadores no caminho, pode ser calculada da seguinte forma:

Equação 2.3: $E_{bit}^{i,j} = d \times (E_{S_{bit}} + E_{B_{bit}}) + (d-1) \times E_{L_{bit}}$

Para efeito do cálculo da energia dinâmica total de um *phit*, somente são considerados os bits que mudam de valor de um ciclo para outro. O método mais adequado para obtenção desse valor é a simulação. Mas na maioria das vezes, uma taxa de chaveamento de bits arbitrária pode ser definida.

Um ferramenta de simulação de estruturas de interconexão de NoCs, tanto de performance quanto de energia, é o Orion (WANG, 2002). Ele foi inicialmente desenvolvido para estimar o consumo dinâmico e mais tarde aperfeiçoado para incluir também o consumo estático.

O modelo de energia implementado no Orion é composto por equações detalhadas e parametrizáveis, desenvolvidas com o objetivo de estimar com maior precisão a capacitância de chaveamento C e a atividade de chaveamento α dos componentes de um roteador através de simulação.

Com C e α calculados, a energia dinâmica é dada por:

Equação 2.4: $E_{din} = \frac{1}{2} \alpha C V_{dd}^2$

Os componentes descritos na biblioteca são: árbitro, *buffers* e matriz de chaveamento. Com base nestes, a energia consumida por um *phit* ao atravessar um roteador é dada por:

Equação 2.5: $E_{phit} = E_{wrt} + E_{arb} + E_{read} + E_{xb} + E_{link}$

onde E_{wrt}, E_{read}, E_{xb} e E_{link} representam respectivamente as energias consumidas em: escrever o *phit* no buffer de entrada, arbitrar o canal de saída, ler o *phit* do buffer, atravessar a matriz de chaveamento e no canal de saída.

Por sua vez, a energia estática de um componente do roteador é dada por:

Equação 2.6: $E_{estat} = I_{leak} \cdot V_{dd} \cdot T \cdot SCALE\_S \cdot \eta$

onde T é o período de ciclo, η é o número de ciclos e $SCALE\_S$ é uma constante de tecnologia.

Para o cálculo da corrente de *leakage*, um modelo foi desenvolvido, onde os parâmetros independentes de tecnologia, como comprimento de transistores, foram separados daqueles que permanecem constantes em relação a esta. No modelo resultante apresentado na Equação 2.7, $I'_{leak}(i,s)$ representa a corrente de *leakage* por unidade de largura do transistor sobre o comprimento do mesmo, onde i é o componente fundamental do circuito, s é o nível lógico atual da entrada e $type(i,s)$ indica o tipo de transistor dominante na corrente de *leakage* (PMOS ou NMOS).

A corrente de *leakage* total $I_{leak}(i,s)$ é calculada da seguinte forma:

Equação 2.7: $$I_{leak}(i,s) = \frac{W(type(i,s))}{L} \cdot I'_{leak}(i,s)$$

onde W é a largura e L o comprimento do transistor. Os valores de $I'_{leak}(i,s)$ para as combinações de s e i foram obtidos por simulação no HSPICE (CHEN, 2003) e são apresentados na Tabela 2.1.

Tabela 2.1: $I'_{leak}(i,s)$ para várias tecnologias.

i	s	Tipo(i,s)	$I'_{leak}(i,s)$		
			Tecnologia (nm)		
			180	100	70
NMOS	0	N	7,9e-9	10,9e-9	67,6e-9
PMOS	1	P	4,0e-9	9,7e-9	80,4e-9
INV	0	N	7,9e-9	10,9e-9	67,6e-9
	1	P	4,0e-9	9,7e-9	80,4e-9
NAND2	00	N	0,3e-9	0,4e-9	9,6e-9
	01	N	7,9e-9	10,8e-9	46,0e-9
	10	N	4,7e-9	5,1e-9	44,0e-9
	11	P	8,1e-9	19,4e-9	159,5e-9
NOR2	00	N	15,9e-9	21,7e-9	133,8e-9
	01	P	3,6e-9	5,9e-9	45,3e-9
	10	P	4,3e-9	9,7e-9	77,5e-9

	11	P	0,9e-9	0,7e-9	5,9e-9

2.3 Principais Redes-em-Chip na Literatura

Os primeiros resultados experimentais com redes-em-chip surgiram em 2000 com as redes aSoC{ XE "aSoC" } (*adaptative SoC*) (LIANG, 2000) e SPIN{ XE "SPIN" } (*Scalable Programmable Integrated Network*) (GUERRIER, 2000). Deste então, outras redes foram apresentadas, entre as quais: CLICHÉ (KUMAR, 2002), Octagon (KARIM, 2001) (KARIM, 2002), Proteo (Saastamoinen, 2002) e a rede com topologia *Torus* 2D (DALLY, 2001). No Brasil, foram desenvolvidas duas redes-em-chip: a rede SoCIN (ZEFERINO, 2003a) (ZEFERINO, 2003b) e a rede Hermes (MORAES, 2003). Na Tabela 2.2 são listadas as principais características destas redes.

Tabela 2.2: Características das principais das redes-em-chip.

Redes	**Topologia**	**Roteamento**	**Chaveamento**	**Controle de Fluxo**	**Arbitragem**	**Memorização**
aSoC	direta	estático determinístico	variação do chaveamento por circuito	tipo FIFO	-	entrada e centralizada
SPIN	árvore gorda	adaptativo	*wormhole*	baseado em crédito	distribuída	entrada e centralizada
Torus 2D	torus dobrado	determinístico	*wormhole*	canais virtuais	distribuída	entrada
Cliché	grelha 2D	determinístico	*wormhole*	(#)	distribuída	entrada
Octagon	anel cordal	determinístico	circuito ou pacotes	(#)	centralizada	(#)
Proteo	anel bidirecional	determinístico	*wormhole*	*handshake*	distribuída	entrada e saída
HERMES	grelha 2D	parcialmente adaptativo	*wormhole*	*handshake*	centralizada dinâmica	entrada
SoCIN	direta	determinístico	*wormhole*	*handshake*	distribuída dinâmica	entrada

(#) informações não disponíveis nas referências consultadas.

A Tabela 2.2 mostra que a topologia de rede predominante na literatura é a grelha. A razão para esta escolha deriva de três vantagens: (*i*) facilidade de implementação usando tecnologias planares dos circuitos integrados atuais; (*ii*) estratégia de chaveamento simplificada (XY); (*iii*) rede facilmente escalável. Outra topologia utilizada é a *torus* bidirecional, que pode ser utilizada para diminuir o diâmetro da rede (FORSELL, 2002) (MARESCAUX, 2002). O *torus* 2D dobrado (DALLY, 2001) é uma opção para redução do aumento do comprimento dos fios quando comparado à *torus*. Um problema associado às topologias *torus* e grelha é a latência de rede. Duas NoCs utilizam topologias alternativas para obter redução de latência. A rede SPIN adota a topologia de árvore gorda, enquanto a rede Octagon sugere o uso da topologia anel cordal. Ambas as topologias conduzem a redes de menor diâmetro, com a conseqüente redução de latência.

Uma escolha comum à maioria das NoCs é a escolha do roteamento determinístico. Em geral, o algoritmo de roteamento determinístico utilizado é do tipo XY. Outra característica predominante é o uso de chaveamento por pacotes. Uma exceção é a rede aSoC, onde a definição das rotas das mensagens é fixada no momento da síntese do hardware. Quanto aos mecanismos de controle de fluxo, existe uma diversidade de métodos, com destaque para os do tipo *handshake* e os que utilizam canais virtuais. Na arbitragem existe um predomínio na utilização da forma distribuída.

Outro parâmetro importante é o tamanho do buffer, que implica na necessidade de ajuste entre esse tamanho e a contenção da rede, latência de entrega dos pacotes e sobrecarga de área do roteador. Buffers grandes conduzem a pouca contenção de rede, a alta latência de pacote e a roteadores com bastante área. Em contrapartida, buffers pequenos implicam em situações inversas.

2.3.1 A Rede SoCIN

A rede-em-chip denominada SoCIN foi desenvolvida no escopo de uma tese de doutorado do Programa de Pós-Graduação em Computação (PPGC) da Universidade Federal do Rio Grande do Sul (UFRGS) (ZEFERINO, 2003a) (ZEFERINO, 2003b).

A SoCIN possui topologia direta, podendo ser configurada como uma grelha 2D, um toróide 2D ou até mesmo como toróide dobrado. Suas principais características são: controle de fluxo do tipo *handshake*; roteamento tipo fonte e determinístico (XY); chaveamento por pacote do tipo *wormhole*; arbitragem distribuída do tipo *round-robin* e memorização de entrada usando buffers do tipo FIFO.

Em cada canal de entrada existe um buffer FIFO responsável pela memorização dos phits que chegam a cada porta de comunicação. Esses buffers possuem p posições com $n + 2$ bits em cada posição.

A rede SoCIN é composta pelo roteador RASoC (*Router Architecture for Systems-on-chip*), um *soft-core* descrito em VHDL com as seguintes características parametrizáveis: largura dos canais de comunicação (n), profundidade dos *buffers* (p) e largura de informação de roteamento no cabeçalho do pacote (m). Cada roteador possui cinco portas de comunicação bidirecional, com uma porta dedicada à comunicação com o núcleo local e quatro para a comunicação com os demais roteadores: N (*North*), S (*South*), W(*West*) e E (*East*).

2.4 Conclusão

De todas as redes-em-chip apresentadas neste capítulo, a rede SoCIN, dentro do escopo deste trabalho, foi escolhida como arquitetura de comunicação devido à sua disponibilidade, integração com o grupo do Laboratório de Sistemas Embarcados (LSE) da UFRGS, e por ser passível de configuração em diversos aspectos, como topologia, tamanho de flit e tamanho de buffer.

3 ESTADO DA ARTE

Este Capítulo apresenta trabalhos relacionados ao âmbito desta tese. Os trabalhos são relacionados às áreas de exploração de espaço de projeto e sistemas distribuídos. Primeiramente são apresentados trabalhos na área de EEP, em tempo de projeto e execução. Em seguida, são apresentados fundamentos básicos e uma revisão do estado da arte no âmbito dos Sistemas Distribuídos. Finalmente, este Capítulo é concluído através de uma análise sobre os pontos apresentados ao longo do mesmo e como se aplicam neste trabalho.

3.1 Métodos de exploração de espaço de projeto em sistemas MPSoCs

3.1.1 Exploração em tempo de projeto (EEP *off-line)*

Esta Seção apresenta trabalhos relacionados a métodos de exploração de espaço de projeto em tempo de projeto (*off-line*, estático ou *compile-time design space exploration*), ou seja, métodos executados na compilação do projeto. Estes métodos, segundo a literatura (POP, 2004), podem realizar configuração e seleção da topologia da infra-estrutura de comunicação, posicionamento de núcleos IP na infra-estrutura de comunicação, pré-alocação de tarefas em elementos de processamento, entre outros. Esta exploração é realizada para um domínio de aplicações, na qual já existe o conhecimento prévio de algumas funções ou comportamentos dentro deste domínio. Este tipo de EEP não está dentro do escopo deste trabalho, onde a EEP será efetuada em tempo de execução com alguma configuração mínima do sistema em tempo de projeto. É importante ressaltar alguns trabalhos relacionados com a EEP em tempo de projeto. Além de existir grande número desses trabalhos na área de sistemas embarcados baseados em redes-em-chip, estes são importantes para definir as configurações necessárias de inicialização do sistema.

No trabalho desenvolvido por Hu e Marculescu (2003), ambos propõem uma abordagem de mapeamento de núcleos IP heterogêneos de uma aplicação em redes-em-chip. A aplicação é modelada por um grafo denominado grafo de caracterização da aplicação, em inglês, *Application Characterization Graph* (APCG). Esta abordagem é baseada em modelos de comunicação com pesos, em inglês *Communication Weighted Model* (CWM), onde o peso associado ao canal de comunicação corresponde à quantidade de bits das mensagens transmitidas por este canal. Os mesmos autores mostraram que, com o uso de um algoritmo adequado de posicionamento automático de núcleos IP em sistemas embarcados que utilizem como arquitetura de comunicação as redes-em-chip, é possível reduzir 60% da energia consumida e garantir o desempenho global do sistema. Aqui, um algoritmo de mapeamento foi desenvolvido para posicionar

os núcleos IP em uma rede-em-chip regular genérica, de modo a minimizar a energia total de comunicação, garantindo a restrição de desempenho através da reserva de largura de banda.

Hu e Marculescu (2004) apresentam uma abordagem de escalonamento e particionamento para alocação de tarefas, em tempo de projeto, em elementos processantes (PE – *Processing Element*) heterogêneos conectados a uma rede-em-chip *mesh 2d*. Este modelo é implementado através de um grafo de tarefas de comunicação, em inglês *Communication Task Graph* (CTG), que é o formato interno do algoritmo de particionamento. A comunicação é representada pela quantidade total de bits enviada de uma tarefa para outra e a computação é representada pelo escalonamento das tarefas com a correspondente dependência de execução e o deadline para executar a tarefa. O CTG permite obter resultados mais precisos que os apresentados em (HU; MARCULESCU, 2003), pois o modelo subjacente leva em consideração, além da quantidade total de bits das tarefas, a dependência de execução das tarefas. Outra questão endereçada com este modelo é o tratamento de aplicações de tempo real, de onde deriva a necessidade de modelar o deadline de cada tarefa. O particionamento é feito estaticamente, com atribuição de prioridade às tarefas. A prioridade é o produto das variâncias do desempenho e da energia das tarefas nos processadores. Como não se trata de uma estimativa em pior caso, perdas de *deadlines* podem ocorrer. Tais perdas de *deadlines* podem ser minimizadas pelo algoritmo de escalonamento proposto. Durante o particionamento, são considerados eventos de escalonamento e de comunicação. A distribuição de folgas é por *budgeting*, que aloca mais folgas para as tarefas que foram mapeadas para PEs que têm maior impacto no consumo de energia e performance. O objetivo é minimizar o consumo de energia atendendo as restrições temporais.

Murali e De Micheli (2004a) implementam uma solução similar à apresentada por Hu e Marculescu em (2003) e apresentam um algoritmo para automatizar o posicionamento de núcleos de hardware que satisfaz as restrições de largura de banda de uma rede-em-chip de topologia *mesh 2d*. A aplicação também é representada por um modelo do tipo CWM, denominado de grafo de núcleos (em inglês, *core graph*). O algoritmo minimiza o atraso médio de comunicação global do sistema, pela divisão de tráfego de mensagens ao longo da rede-em-chip. Como extensão deste trabalho, Murali et al. (2004b) desenvolveram uma ferramenta chamada SUNMAP. Esta ferramenta constrói internamente uma biblioteca de topologias de redes-em-chip e usa uma função multi-objetivo a ser otimizada, que inclui média de atraso de comunicação, consumo de área e consumo de energia. O principal objetivo desta ferramenta é selecionar automaticamente a melhor topologia para uma dada aplicação e gerar um mapeamento dos núcleos de hardware na topologia, de forma a atender o conjunto de restrições da aplicação, usando o mesmo modelo apresentado em (MURALI; DE MICHELI 2004a). Uma vez que as bibliotecas de topologias estejam caracterizadas, a abordagem adotada permite aumentar a precisão das estimativas no processo de síntese de alto nível. Todavia, para estimativas precisas, as alturas e larguras dos *tiles* da NoC devem ser fixas, o que pode implicar maior consumo de área e energia.

Hung et al. (2004) apresentam uma metodologia que realiza o posicionamento de núcleos IPs em tempo de projeto e a reconfiguração de PEs. Esta é realizada com o intuito de diminuir o consumo de potência em todos os PEs e a comunicação entre eles. A comunicação entre os PEs afeta o desempenho global do sistema e o consumo de potência de cada roteador, dependendo do posicionamento dos núcleos IPs. Porém, este

trabalho difere das outras metodologias, pelo fato de utilizar a distribuição de temperatura como função-objetivo. O *framework* de posicionamento de núcleos IP apresentado neste trabalho foi desenvolvido através do uso da heurística de algoritmo genético. O algoritmo distribui e configura a lógica de todos os PEs para diminuir a temperatura levando em conta a distribuição de temperatura de maneira homogênea no sistema, e ao mesmo tempo o algoritmo tenta minimizar o custo de comunicação.

Bolotin et al. (2004) propõem uma rede-em-chip com qualidade de serviço (QoS NoC). Uma vez que as restrições da comunicação de uma aplicação-alvo tenham sido identificadas, a rede-em-chip então é personalizada para atender estas restrições. Primeiro, os módulos de hardware são posicionados para minimizar o tráfego de mensagens na rede. A seguir, roteadores e conexões desnecessárias são removidos, e a largura de banda alocada permanece com roteadores e conexões ativas de acordo com sua carga relativa. Deste modo, a utilização da conexão é balanceada. Este processo modifica uma arquitetura de rede genérica baseada em topologia malha bidimensional e roteamento XY. Este trabalho propõe, como principal mérito, um fluxo de síntese de projeto multi-objetivo com requisitos de qualidade de serviço, que permite tratar de aplicações de tempo real, para configuração da rede-em-chip. Entretanto, não é apresentado nenhum algoritmo de posicionamento dos núcleos IPs, somente aponta que este algoritmo deverá ter o objetivo de otimizar a rede-em-chip minimizando energia, trafego e área do sistema, garantindo QoS.

Kreutz et al. (2005) introduzem um modelo chamado ACP (*Application Communication Pattern*) que permite capturar não somente a capacidade de comunicação, mas também a ordem de comunicação. Este modelo é usado para avaliar o consumo de energia e a latência para comparar diferentes topologias de redes-em-chip. O artigo também explora o espaço de projeto para topologias de redes-em-chip. Através do ACP, faz-se uma pesquisa para modelos de energia analíticos para encontrar a melhor escolha de topologia, de acordo com as restrições de área, energia e redução da latência de mensagens da aplicação.

Lei et al. (2003) apresentam um algoritmo genético para mapeamento de um grafo de tarefas na arquitetura de rede-em-chip. O objetivo principal deste trabalho é realizar o mapeamento de núcleos IP, provendo minimização do tempo de execução global da aplicação na rede-em-chip regular. Neste trabalho também foi desenvolvido um modelo de atraso de comunicação para uma arquitetura específica para estimar o tempo de execução. É um modelo simples, porém razoável para a realização da estimativa de execução da aplicação. O algoritmo genético foi dividido em dois passos. O primeiro passo calcula o melhor particionamento de tarefas em núcleos e o segundo passo posiciona estes núcleos sobre uma rede-em-chip com topologia malha, para minimizar o tempo de execução global do sistema. O modelo de atraso de comunicação não leva em conta os parâmetros internos da rede-em-chip, como tamanho de buffer, algoritmos de roteamento, etc.

Varatkar e Marculescu (2003) apresentam uma metodologia para minimizar o consumo de energia global em uma rede-em-chip, pelas técnicas de: (i) alocação de tarefas nos PEs e escalonamento e (ii) seleção de tensão de cada PE, maximizando a folga de escalonamento entre as tarefas. O primeiro passo, voltado para redução de comunicação entre os processadores, executa a alocação de tarefas e o escalonamento das tarefas. No segundo, é ajustada a tensão de cada PE para maximizar a folga de escalonamento (com o objetivo de diminuir a freqüência, e conseqüentemente, baixar a potência). A técnica da seleção da tensão explora a distribuição não-uniforme das

tarefas para encontrar e organizar as folgas de escalonamento a ponto de ajustar a freqüência desejada no pior caso da execução das tarefas. Esta técnica considera o tempo e energia desperdiçados pelo chaveamento de seleção da tensão. Entretanto, esta abordagem considera apenas o custo de comunicação entre os PEs.

Szymanek e Kuchcinski (2003) apresentam uma técnica de alocação de tarefas em processadores heterogêneos conectados a um barramento, conjuntamente com escalonamento de tarefas. Esta técnica considera o tamanho da memória do código e dados de tarefas como principal restrição de projeto, respeitando *deadlines* das tarefas. As tarefas são modeladas como um grafo onde cada nodo tem restrições de memória de código e de dados. Resultados demonstraram que esta abordagem diminuiu a quantidade de comunicação no barramento além de alocar as tarefas de acordo com as restrições. Os autores afirmam que esta técnica pode ser aplicada em arquiteturas de comunicação de naturezas diferentes e em processadores heterogêneos. Contudo, esta técnica não considera contenções de pacotes no barramento.

Corrêa et al. (2004) discutem a adaptação de redes-em-chip para restrições de sistemas de tempo real, tratando-se especificamente de *deadlines* de tarefas. O artigo também apresenta uma abordagem de posicionamento em tempo de projeto dos núcleos de hardware na rede-em-chip baseada nas exigências das larguras de banda das mensagens dos canais de comunicação entre os núcleos e uma estratégia de alteração dinâmica de prioridade nas mensagens. Estas estratégias podem diminuir o número de *deadlines* perdidos de tarefas, em sistema de tempo real soft. O artigo discute também o impacto dessas estratégias no consumo de energia do sistema e mostra um espaço de projeto a ser explorado. De acordo com a combinação das estratégias citadas anteriormente, segundo os autores, é possível extrair um compromisso adequado entre as restrições do sistema de tempo real *soft* e consumo de energia.

Ruggiero et al. (2006) apresentam um ambiente para alocação de tarefas em uma plataforma MPSoC apresentada na Figura 3.1. O problema é dividido em dois grupos: alocação de tarefas (posicionamento espacial das tarefas neste ambiente para aumentar o desempenho do sistema) em processadores e escalonamento de tarefas (posicionamento temporal das tarefas). A alocação de tarefas é resolvida com o algoritmo Programação Linear Inteira (ILP, do inglês, *Integer Linear Programming*) e é assim definido: “dado um grafo de tarefas, o problema consiste em alocar *n* tarefas em *m* processadores, de forma que a quantidade total de memória alocada, para cada processador, não exceda o tamanho de cada memória local”. O escalonamento é resolvido através de Programação por Restrições (CP, do inglês, *Constraint Programming*). As soluções para ambos os problemas interagem convergindo a uma solução ótima. Os grafos de tarefas são modelados como um *pipeline*, abordagem utilizada em certas aplicações multimídias reais, onde as tarefas ocorrem seqüencialmente no tempo. A arquitetura alvo utiliza memória distribuída com troca de mensagens. Os processadores são homogêneos (utilizam ARM7) e são interligados por barramento. Cada processador tem uma memória local e pode acessar memórias remotas com um sistema operacional de tempo-real para prover APIs para troca de mensagens. Um estudo de caso, baseado em multimídia, demonstra uma maior eficiência computacional da abordagem que utiliza a interação de ILP com CP em relação às abordagens tradicionais (puramente ILP ou CP).

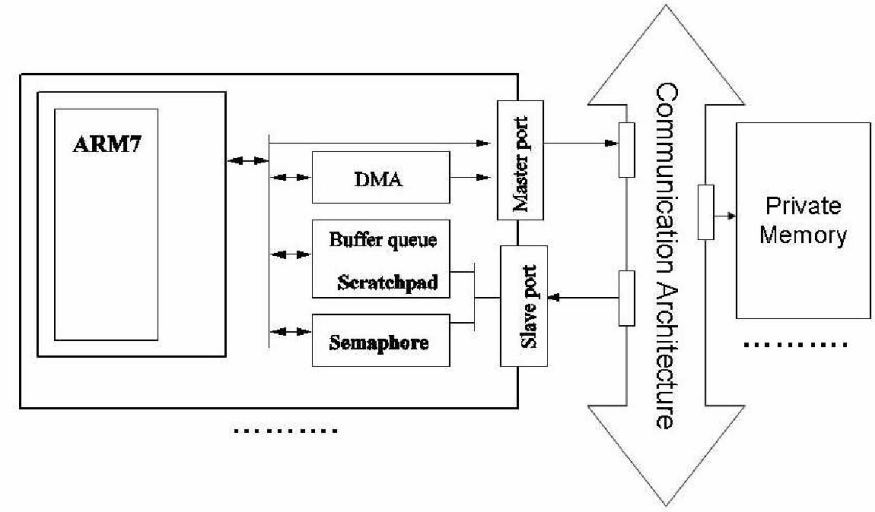


Figura 3.1: Plataforma MPSoC baseada em barramento.

Wronski et al. (2006) apresentam estratégias para minimização de energia em tempo de execução de uma arquitetura de comunicação baseada em redes-em-chip *mesh 2d* 3x3. Dois algoritmos de alocação de tarefas foram aplicados no sistema, antes da execução da aplicação: Worst-fit (distribuição de carga) e Best-fit (concentração de carga). Segundo os autores, estes são algoritmos simples que têm baixo custo no sistema, e consequentemente, podem ser empregados em tempo de execução. As aplicações são baseadas em grafo de tarefas TGFF (*Task Graphs for Free*) (DICK, 1998) e esse grafo tem todas as informações de cada tarefa conhecidas *a priori*. As aplicações são baseadas em tempo-real. Os algoritmos de alocação são executados antes da execução da aplicação, ou seja, em tempo de projeto. Os autores afirmam que somente balanceamento de carga não é suficiente para minimizar o consumo de energia em tempo de execução de maneira eficiente, quando é considerada a comunicação entre os processadores. É necessária a execução combinada entre balanceamento de carga e escalonamento de tensão.

3.1.2 Exploração em tempo de execução (EEP *on-line)*

Esta Seção apresenta trabalhos relacionados a métodos de exploração de espaço de projeto em tempo de execução (*on-line*, dinâmico ou *runtime design space exploration*), ou seja, métodos de atendimento a exigências de projeto na execução do sistema.

A técnica de DVS (*Dynamic Voltage Scaling*) é muito importante para sistemas com mecanismos de exploração de espaço de projeto em tempo de execução. Com DVS, é possível diminuir a freqüência do sistema, para economizar energia, atendendo restrições de desempenho. Os primeiros trabalhos de DVS foram feitos para sistemas sem restrições temporais, como Weiser et al. (1994). Eles foram os primeiros a propor uma técnica de escalonamento que varia a freqüência e a tensão dinamicamente, na qual são adotadas janelas de execução, durante as quais a velocidade do processador é mantida constante. Ao final de cada janela o desempenho é re-avaliado, podendo compensar uma demanda por desempenho na janela anterior não suprida.

Yao et al. (1995) apresentam o algoritmo AVR (*Average Rate Heuristic*). Este algoritmo baseia-se na premissa de que sistemas de tempo real possuem tarefas com WCET (*Worst-Case Execution Time* – pior tempo de execução) definido a priori. Neste contexto, a freqüência do processador pode ser reduzida estática ou dinamicamente de forma a prover o desempenho necessário para o pior caso, economizando assim a energia referente à folga estática. O algoritmo DAR (*Dynamic Average Rate*) (ZHUO, 2005) é a aplicação dinâmica da heurística AVR para cálculo da freqüência do processador. O cálculo é refeito a cada alteração na fila de tarefas prontas do escalonador, considerando os WCETs restantes das tarefas. Enquanto o AVR aplica a heurística de utilização do processador (razão entre o tempo de execução de uma tarefa e o período da mesma) para o cálculo da freqüência normalizada, o DAR usa a *densidade da tarefa* que corresponde à razão entre o quanto uma tarefa ainda precisa executar para completar seu pior caso e o tempo disponível até o *deadline*. A freqüência normalizada do processador é dada pela soma das densidades de todas as tarefas prontas para executar. Desta forma, o DAR é um algoritmo dinâmico com um custo computacional razoável e menor consumo de energia que o AVR.

Nollet et al. (2005) apresentam um método de gerenciamento de recursos em tempo de execução em redes-em-chip. A análise realizada é baseada em prototipação, contudo não são apresentados muitos dados a respeito do desempenho, seja temporal ou energético. A contribuição principal do artigo é a descrição de uma heurística para gerenciamento de recursos na rede-em-chip, a qual é aplicada em PEs reconfiguráveis de grão fino (ex: FPGA) e é baseada no mecanismo de migração de tarefas. A heurística é baseada em *backtracking* que é executada em alguns passos: (i) calcular a carga requisitada pelo recurso; (ii) calcular a variação de execução de cada tarefa (executada num PE de propósito geral, ou numa lógica reconfigurável, por exemplo; (iii) Calcular o peso de comunicação total da aplicação; (iv) Ordenar as tarefas de acordo com um critério de importância de mapeamento calculada pelo produto do peso de comunicação total e a variação do tempo de execução de uma tarefa em cada PE (por exemplo, a variação do tempo de execução entre uma tarefa executada em um processador e um hardware reconfigurável); (v) ordenar os PEs para as tarefas não mapeadas mais importantes; (vi) finalmente é realizado o mapeamento das tarefas, associando-as aos PEs. Os passos v e vi são repetidos até que todas as tarefas sejam mapeadas. Porém, computação exaustiva neste último passo não pode ser aplicada pelo aumento fatorial de tempo. Então, os autores propuseram a implantação do algoritmo clássico de *backtracking*. Neste sistema, o sistema operacional está centralizado num núcleo IP mestre que controla os demais núcleos IPs escravos. É usada uma rede-em-chip *mesh* 3x3 e o mapeamento dinâmico de tarefas é realizado para melhorar o desempenho global do sistema (tentar mapear tarefas com computação intensiva em lógica reconfigurável). Entretanto, o artigo não apresentou nenhum resultado sobre algum atendimento de restrições (no caso deste trabalho, ocorre um aumento no desempenho do sistema) de projeto em tempo de execução, nem mesmo comparações com algum outro trabalho encontrado na literatura. Um valor para o tempo perdido na migração de uma tarefa é apenas inferido analiticamente, não sendo demonstrada nenhuma confirmação da correção do mesmo.

Neste trabalho, 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 3.2 (NOLLET, 2005).

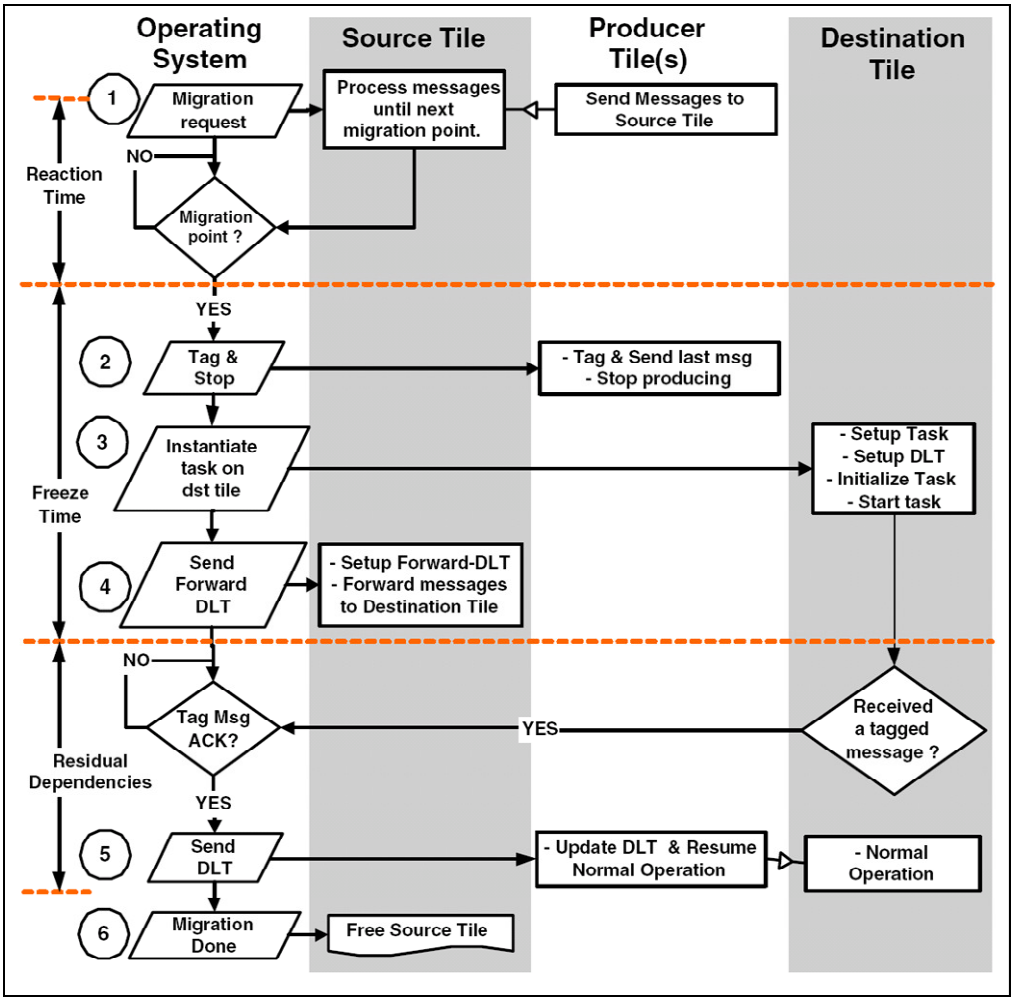


Figura 3.2: Mecanismo de Migração.

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.

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.

No trabalho apresentado por Aydin e Yang (2003), são usadas heurísticas de *bin-packing on-line* (*First-Fit*, *Best-Fit*, *Next-Fit* e *Worst-Fit*) e *off-line* (*Worst-Fit Decreasing, First-Fit Decreasing e Best-Fit Decreasing*) para a alocação de tarefas com minimização do consumo de energia, considerando tarefas independentes sem custo de comunicação. O trabalho propõe que o desempenho ótimo em termos de energia, para o escalonamento de um grupo de tarefas em um processador com política EDF, é constante e mantém a utilização total do processador próximo a 1. Desse modo, a alocação distribui as tarefas uniformemente entre os processadores (balanceamento de carga). As aplicações são baseadas em tempo-real, diferentemente das aplicações usadas em Bertozzi (2006) e Nollet (2005) e as heurísticas usadas aqui realizam alocação de tarefas em tempo de execução para atender todos os *deadlines*. Além disso, esse trabalho abstrai a infra-estrutura de comunicação.

Para alocação *off-line,* o WFD - *Worst-Fit Decreasing*, que ordena a lista de itens de forma decrescente antes de aplicar o WF, ganha em porcentagem de escalonamentos viáveis gerados e na porcentagem de energia ganha. Por outro lado, FFD - *First-Fit Decreasing* e BFD - *Best-Fit Decreasing* geram alocações concentradas e não uniformes e por isso têm péssimo desempenho de energia.

No escalonamento *on-line*, sem ordenar a lista a priori, não há um algoritmo vencedor em relação ao tempo de execução e à energia. FF, NF e BF têm boas performances em termos de tempo de execução e performances ruins em termos de energia. O WF apresenta desempenho razoável em baixas utilizações, degradando rapidamente com crescimento destas. Para resolver esse problema é sugerida a reserva de processadores, na qual metade dos processadores é reservada para tarefas com utilização abaixo da média. Dentro de cada partição aloca-se com WF. O WF com reserva apresenta a boa viabilidade do BF/FF aliado ao bom resultado de energia do WF.

Bertozzi et al. (2006) propõem uma infra-estrutura de software para gerenciamento de tarefas em sistemas MPSoCs baseados em barramento com sistemas operacionais distribuídos. 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). Os autores propõem também um método para balanceamento de carga baseado em migração de tarefas controlado pelo usuário e executado através de primitivas de troca de mensagens baseadas em memória compartilhada. Este balanceamento é realizado no início da execução do sistema, respeitando apenas restrições de desempenho. Um processador mestre é responsável por executar um algoritmo para balanceamento de carga, migrando tarefas entre os processadores escravos através da memória compartilhada. 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*, porém baixa transparência (Seção 3.2.1). 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 pelo barramento, é mais vantajoso migrar tarefas do que sobrecarregar um determinado processador. Bertozzi et al. (2006) mostram 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. A topologia do sistema é apresentada na Figura 3.3 (BERTOZZI, 2006).

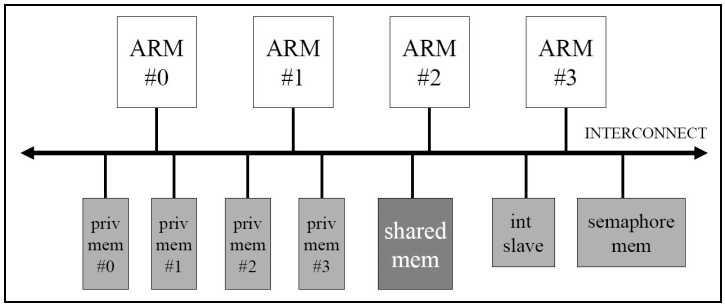


Figura 3.3: Topologia da Plataforma.

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. Porém, 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 realizada. Finalmente, Bertozzi et al. (2006) não fazem uma análise sob o aspecto energético da solução desenvolvida. A falta de transparência desta abordagem totalmente controlada pelo usuário, ao invés de ser uma desvantagem, pode ser uma característica desejável para os futuros sistemas embarcados que utilizam sistemas operacionais distribuídos que provêem mecanismos de migração de tarefas, segundo os autores.

Ozturk et al. (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 (tempo de execução), baseada em estatísticas coletadas estaticamente (tempo de compilação). O objetivo deste trabalho é reduzir o consumo de energia de um MPSoC durante a comunicação entre os processadores.

A Figura 3.4 ilustra as duas opções de migração. A migração de dado é representada na Figura 3.4a, 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 3.4(b), 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).

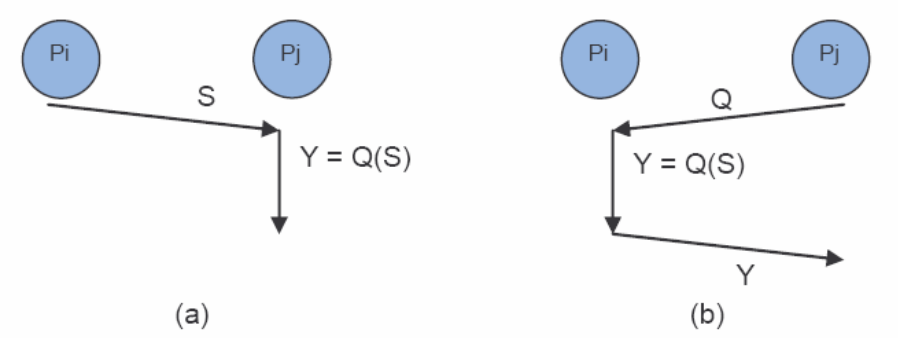


Figura 3.4: (a) Migração de dados; (b) migração de tarefa (código).

A arquitetura MPSoC utilizada no trabalho é composta de 8 processadores interligados por um barramento. Cada processador possui memória local de 32 KB de capacidade. A comunicação entre os processadores é realizada através de troca de mensagens. 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.

Ngouanga et al. (2006) exploram a alocação dinâmica de tarefas em uma arquitetura reconfigurável de grão grande. O trabalho consiste em uma plataforma chamada APACHES composta por elementos de processamento homogêneos conectados a uma rede-em-chip. O número de nodos é totalmente parametrizável. Os nodos de processamento (escravos) executam tarefas e são agrupados em clusters. Os nodos mestres ou monitores de redes gerenciam os clusters e desempenham a alocação de tarefas nos mesmos. Nesta plataforma, existe um nodo mestre para cada cluster.

A Figura 3.5 (NGOUANGA , 2006) apresenta o funcionamento da plataforma APACHES. Os nodos *M* e *I/O* são respectivamente o monitor da rede e o controlador de I/O. Os nodos restantes são passivos, ou seja, são nodos escravos. Apenas o nodo mestre (*M*) é responsável por desenvolver o mapeamento das tarefas. Na Figura 3.5 (a),

uma aplicação está em execução na plataforma. Quando uma nova aplicação estiver pronta para ser mapeada na rede, o sistema realiza seqüencialmente as seguintes operações:

1. Uma nova alocação é computada pelo nodo mestre;
2. Se a nova alocação implicar em realocar algumas tarefas de aplicações que já estiverem em execução, o nodo mestre move essas tarefas para suas novas localizações na rede-em-chip. Para isso é enviada uma requisição para cada elemento de processamento que executará uma tarefa realocada. (Figura 3.5 (b));
3. Os códigos das tarefas que foram realocadas e das tarefas da nova aplicação são enviados para os elementos de processamento aos quais elas foram designadas (Figura 3.5 (c) e Figura 3.5(d));
4. O sistema retoma a execução de tarefas interrompidas e inicia o processamento da nova aplicação mapeada na rede (Figura 3.5 (e)).

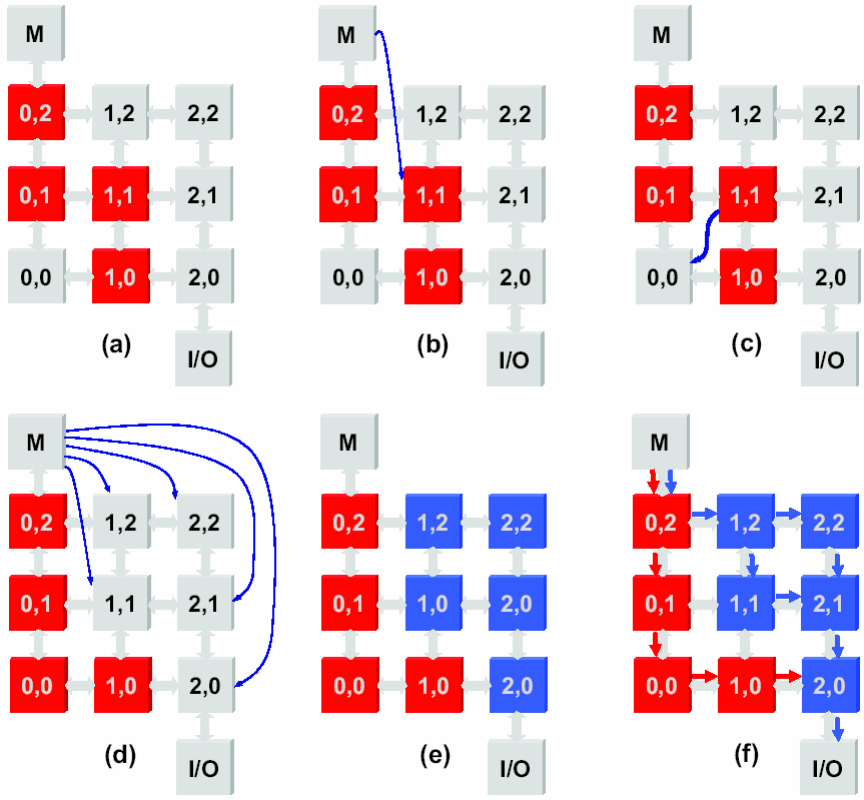


Figura 3.5: Alocação de tarefas e roteamento de dados na plataforma APACHES.

Cada elemento de processamento executa um microkernel multitarefa. São empregados dois algoritmos para alocação de tarefas: *Simulated Annealing* e *Force Directed*. O primeiro é baseado na simulação de um fenômeno físico, o qual muda aleatoriamente a posição das tarefas na rede, estima custos e aceita ou rejeita propostas de mudança de estados de acordo com um critério de temperatura. O segundo algoritmo

calcula o posicionamento das tarefas, considerando a força resultante produzida pela atração entre tarefas comunicantes. Essa força de atração é proporcional ao volume de comunicação entre tarefas interligadas. Os resultados mostram que a alocação dinâmica traz benefícios em termos de aumento de *throughput* do sistema. No entanto nenhum resultado em termos de consumo de energia e perda de deadlines foi demonstrado neste trabalho.

Acquaviva et al. (2007) apresentam um middleware que implementa a migração de tarefas em MPSoCs e a avaliação de sua eficiência no contexto de aplicações multimídias de tempo real *soft*. A Figura 3.6 apresenta a organização de hardware e software da plataforma. O middleware foi implementado em cima do sistema operacional uClinux rodando em uma plataforma de emulação de vários núcleos IP prototipados em um FPGA (CARTA, 2007). Esta plataforma foi caracterizada em termos de desempenho e energia e a eficiência do middleware foi avaliada através do uso de uma aplicação Radio FM.

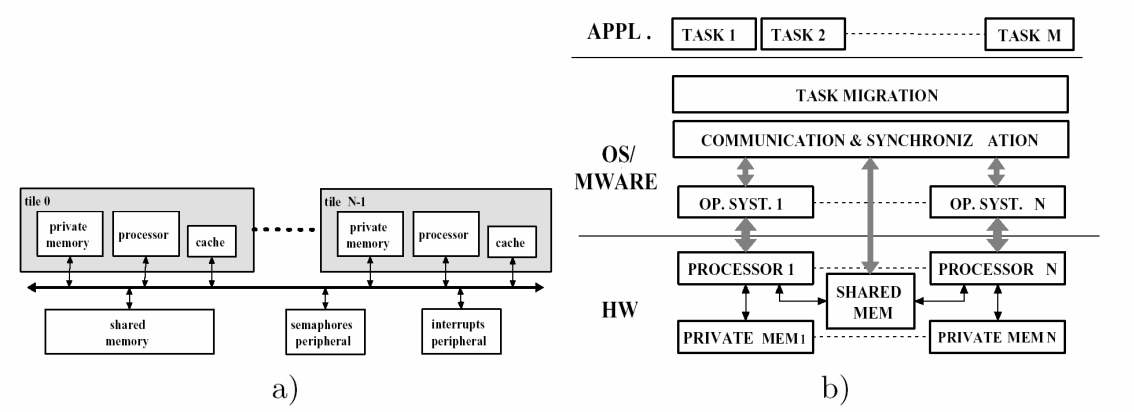


Figura 3.6: Organização de hardware e software da plataforma: (a) arquitetura de hardware baseada em comunicação por barramento; (b) Middleware e a camada de abstração que provê a migração de tarefas.

Os autores implementaram migração de tarefas em MPSoC homogêneo (mesmos processadores) baseado na arquitetura de comunicação de barramento com suporte a memória compartilhada, onde cada núcleo tem sua memória privada. Os experimentos deste trabalho demonstraram que a migração, no nível do middleware do SO, é viável e aumenta o desempenho do sistema, em termos de energia e em termos de QoS (perda de deadlines).

Carta et al. (2007) desenvolveram uma plataforma de emulação de um sistema operacional para MPSoCs que provê uma política de gerenciamento de temperatura do sistema implementado em um FPGA Xilinx Virtex II com 4 processadores (Figura 3.7). Segundo os autores, esta plataforma permite que projetistas desenvolvam várias políticas de gerenciamento de temperatura para teste e viabilidade. A política de gerenciamento de temperatura adotada neste trabalho provê minimização de *hot-spots*, ou seja, minimização da diferença da temperatura entre os processadores utilizando migração de tarefas. No sistema, existe um núcleo mestre que dispara as tarefas nos processadores. Quando uma tarefa é inicializada por um usuário, esta tarefa é replicada em todos os processadores através de uma chamada de sistema (*fork*). Um processador com esta tarefa é escolhido para execução pelo núcleo mestre. Nos demais

processadores, as tarefas replicadas pelo núcleo mestre são inseridas em uma fila local a cada processador, e permanecem suspensas. Com esta técnica, segundo os autores, mesmo com o desperdício de memória para armazenamento das réplicas, o tempo de migração é desprezível. O núcleo mestre gerencia a diferença de temperatura, e segundo a sua heurística, ele envia, através da comunicação de memória compartilhada, mensagens para os processadores, de forma a desabilitar ou habilitar as réplicas de uma determinada tarefa para execução. Através de um software rodando em cima do sistema operacional Linux em um PC, este executa um modelo de temperatura do sistema. Este software, a cada intervalo de tempo, recebe dados do sistema de emulação para atualização das informações de temperatura e repassa para o núcleo mestre. Este então toma as decisões de migração de acordo com a temperatura recebida do PC. Os autores mostraram, através dos experimentos, que a migração de tarefas é um eficiente mecanismo para gerenciamento e controle da temperatura dos processadores.

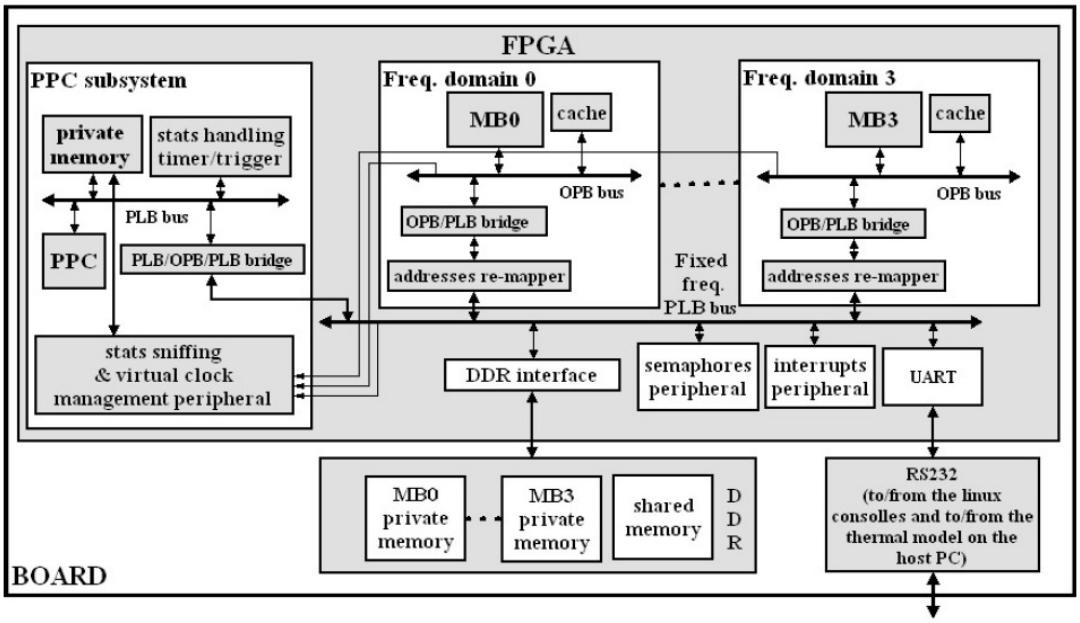


Figura 3.7: Plataforma MPSoC de emulação com gerenciamento de temperatura ativada por migração de tarefas.

Carvalho et al. (2007) avaliaram diversos algoritmos para alocação dinâmica de tarefas para otimizar a ocupação dos canais da rede-em-chip de um sistema embarcado heterogêneo prototipado em FPGA (Figura 3.8), diminuindo assim a congestionamento de pacotes na rede. A alocação das tarefas ocorre em tempo de execução, quando ocorre uma requisição de comunicação e de carga na rede. As tarefas são implementadas em hardware (lógica reconfigurável implementada nas áreas reconfiguráveis) ou software (processadores). As tarefas são definidas por um id, pelo tempo de execução, qual tipo de núcleo alvo (áreas reconfiguráveis ou processadores) e taxa de comunicação entre as tarefas. Segundo os autores, é possível aumentar o desempenho do sistema como um todo.

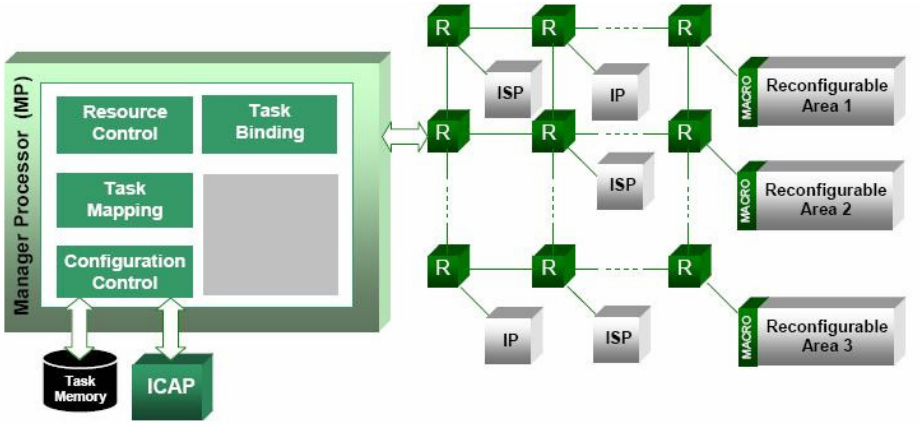


Figura 3.8: Arquitetura MPSoC prototipada em FPGA.

Os autores mencionam no artigo que para reduzir o tempo de execução e consumo de energia no sistema, deve-se evitar congestionamento nos canais. No entanto, não foi apresentado nenhum dado relacionado com o consumo de potência e energia, não foram consideradas perdas de deadlines (sem suporte a tempo-real), e não foram apresentados custos de alocação (dados do contexto da tarefa).

3.1.3 Conclusão

Na exploração do espaço de projeto de redes-em-chip, os parâmetros que afetam os requisitos dessas redes são vários, como por exemplo: o tipo de arquitetura de interconexão usada, a topologia da mesma, a largura de banda e tráfego de mensagens na arquitetura de interconexão, a distribuição de tarefas entre os processadores, o tamanho dos *buffers* dos roteadores, o posicionamento dos recursos na rede, dentre outros. Em função disso é necessária a utilização de uma metodologia para a exploração desse espaço de projeto.

Nas várias metodologias que buscam explorar o espaço de projeto em redes-em-chip, o objetivo é encontrar uma solução ótima, ou sub-ótima, para uma dada função. Para tanto, essa solução é calculada, geralmente por processamento oneroso e intensivo e esse processamento deve ser realizado em tempo de projeto. Porém, como foi dito anteriormente, quando a carga do sistema muda seu comportamento em tempo de execução, os objetivos que foram determinados em tempo de projeto não terão grande eficiência, pois esta mudança de carga não estava prevista no método de otimização. Como os sistemas embarcados estão se aproximando cada vez mais dos sistemas de computação de propósito geral, métodos de exploração de espaço de projeto em tempo de execução serão necessários para tornar a otimização mais eficiente, desde que os custos e impacto dos algoritmos de exploração sejam amortizados pela execução do sistema.

Dentre todos os trabalhos apresentados nas Seções 3.1.1 e 3.1.2, muitos artigos relativos à exploração *off-line* para minimização de energia, maximização de desempenho e minimização de *deadlines* perdidos foram desenvolvidos. Porém, ainda existe carência de trabalhos na academia sobre exploração *on-line* para aumento de desempenho e os resultados apresentados são bastante incipientes (BERTOZZI, 2006) (NOLLET, 2005). Em todos esses trabalhos apresentados, não há nenhum que envolva exploração *on-line* para sistemas embarcados de tempo-real baseados em redes-em-chip que combine minimização de energia, aumento de desempenho e minimização de *deadlines* perdidos.

A Tabela 3.1, baseada em tabela similar apresentada em (BRIÃO, 2005), resume os métodos apresentados por estes trabalhos, apresentando suas características e peculiaridades. No entanto, alguns trabalhos aqui foram inseridos no contexto da exploração de espaço de projeto em tempo de execução.

Tabela 3.1: Propostas de exploração do espaço de projeto de redes-em-chip.

Autor	**Metodologia**	**Tipo de EEP**	**Arquitetura de comunicação/topologia**	**Tipo de plataforma**	**Restrições**	**Objetivo**	**Suporte a Tempo Real**
Acquaviva 2007	migração e alocação de tarefas	dinâmica	barramento	homogênea	energia	minimizar energia e perda de deadlines	*soft deadlines*
Aydin e Yang 2003	alocação de tarefas	híbrida – WFD, FFD e BFD são estáticos e WF, FF, BF e NF são dinâmicos.	Nenhuma, pois os autores abstraíram a arquitetura de comunicação	homogênea	utilização do processador	minimizar energia	*hard deadlines*
Bertozzi 2006	migração e alocação de tarefas	dinâmica	barramento	heterogênea	desempenho	minimizar tempo de execução	não
Bolotin 2004	personalização de NoC	estática	irregular e personalizável	heterogênea	largura de banda ou desempenho	garantir QoS, minimizando energia, tráfego, e área	*hard deadlines*
Carta 2007	migração de tarefas	dinâmica	barramento	homogênea	temperatura	minimizar a diferença de temperatura entre os processadores	não
Carvalho 2007	alocação de tarefas	dinâmica	grelha regular	heterogênea	ocupação do canal	diminuir o congestionamento da rede para aumentar o desempenho do sistema em termos de tempo de execução.	não
Correa 2004	mapeamento de núcleos IP	estática	grelha regular	homogênea	largura de banda	garantir QoS, minimizando perda de deadlines	*soft deadlines*
Hu 2003	mapeamento de núcleos IP	estática	grelha regular	heterogênea	largura de banda e desempenho	minimização de energia total de comunicação	não

Autor	**Metodologia**	**Tipo de EEP**	**Arquitetura de comunicação/topologia**	**Tipo de plataforma**	**Restrições**	**Objetivo**	**Suporte a Tempo Real**
Hu 2004	alocação e escalonamento de tarefas	estática	grelha regular	heterogênea	desempenho	reduzir consumo de energia	*hard deadlines*
Hung 2004	mapeamento de núcleos IP	estática	grelha regular	heterogênea	comunicação	minimizar comunicação entre núcleos IP e balanceamento de temperatura.	não
Kreutz 2005	mapeamento de tarefas/ escolha de topologia	estática	grelha regular, árvore gorda, *torus*	homogênea – núcleos IP; heterogênea – roteadores	desempenho	minimizar latência e consumo de energia	não
Lei 2003	mapeamento de núcleos IP	estática	grelha regular	homogênea	desempenho	minimizar o tempo global de execução do sistema	*soft deadlines*
Murali 2004a, Murali 2004b	mapeamento de núcleos IP	estática	grelha regular	heterogênea	largura de banda	minimizar o atraso médio de comunicação dividindo o tráfego de mensagens entre os núcleos	não
Ngouanga 2006	alocação de tarefas	dinâmico	grelha regular	homogênea	desempenho	aumentar o throuput do sistema	não
Nollet 2005	mapeamento (migração de tarefas)	dinâmica	grelha regular	heterogênea	desempenho	minimizar o atraso de comunicação	não
Ozturk 2006	migração de tarefas (código ou dados) entre os PEs	dinâmica	barramento	homogênea	energia	reduzir consumo de energia durante a comunicação entre os processadores	não
Ruggiero 2006	Alocação e escalonamento de tarefas nos PÉS	estática	barramento	homogênea	memória	minimizar tempo de execução da aplicação	*soft deadlines*

Autor	**Metodologia**	**Tipo de EEP**	**Arquitetura de comunicação/topologia**	**Tipo de plataforma**	**Restrições**	**Objetivo**	**Suporte a Tempo Real**
Varatkar 2003	alocação, escalonamento de tensão e seleção de tensão	estática	grelha regular	homogênea	comunicação	minimizar freqüência, p/reduzir o consumo de potência e diminuição do volume de comunicação do sistema	*soft deadlines*
Wronski 2006	alocação de tarefas e escalonamento de tensão para minimização de energia	estática	grelha regular	homogênea	comunicação e utilização do processador	minimizar energia, minimizar comunicação	*soft deadlines*

3.2 Sistemas distribuídos

Esta Seção apresenta alguns trabalhos na literatura relacionados à área de Sistemas Distribuídos, que fundamentam este trabalho.

3.2.1 Migração de processos

Segundo Milojicic (2000), migração de processos é o ato de transferir um processo entre duas máquinas durante sua execução. A migração é uma técnica aplicada em sistemas distribuídos. Dentro deste contexto, diversas técnicas vêm sendo estudadas e aplicadas na migração de processos. Sistemas do tipo *cluster* e *grid* são exemplos de aplicações que necessitam de um gerenciamento de processos distribuídos ao longo dos nós de processamento presentes nas redes. Ao contrário, ainda não há solução clara e definitiva para o problema da migração de tarefas[4] em sistemas multiprocessados de uma única pastilha (MPSoCs).

A técnica de migração de processos, segundo Sinha (1997), é classificada de duas diferentes formas: preemptiva e não preemptiva. Migrações preemptivas ocorrem em tempo de execução, após o início da computação dos processos. Por alguma razão o sistema decide migrar o processo; é necessário, então, salvar o contexto do mesmo e transferi-lo para o processador de destino.

Migrações não preemptivas são aquelas que ocorrem antes do início da execução de um processo, operando da seguinte maneira: o nó mestre, responsável pelo sistema, precisa iniciar um novo processo; ele decide, por sua vez, 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.

A funcionalidade de migração de processos 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 utilização de migração de processos. A migração de processos reduz o tempo médio de resposta dos programas, aumenta a vazão computacional aparente do sistema, promove uma utilização mais efetiva dos recursos presentes, reduz o tráfego de comunicação pelo sistema de comunicação, aumenta a confiabilidade do sistema, entre outros. Entre as vantagens mencionadas, pelo menos duas não são intuitivas: *diminuição do tráfego do sistema de comunicação* e *aumento da confiabilidade do sistema*. Este último assunto está fora do escopo deste trabalho.

A diminuição do tráfego de comunicação da rede se dá pelo agrupamento de processos que se comunicam periodicamente ou com grande volume de dados em um mesmo processador ou em processadores em locais próximos. Para isso é necessário conhecer a natureza das aplicações, ou a priori ou através do monitoramento do tráfego da rede, e, no momento da alocação ou de uma realocação, colocá-las próximas entre si. Essa é uma função do gerenciador de carga.

---

[4] Processo, segundo Milojicic (2000), é uma abstração do sistema operacional que representa uma instância de um programa computacional em execução em sistemas computacionais distribuídos. Tarefa é uma abstração do sistema operacional de uma funcionalidade no âmbito de sistemas embarcados. Tarefas geralmente são menores que os processos, em custos tais como tamanho de código, memória e complexidade algorítmica.

Para permitir a implementação da migração de processos, 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, dispositivos, conexões). Alguns sistemas operacionais distribuídos, como MOSIX (BARAK, 1989) e Sprite (OUSTERHOUT, 1988), fornecem suporte nativo à migração de processos. A migração de tarefas quando suportada pelo sistema operacional se torna mais transparente[5], 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.

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.

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. 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 do nó destino a partir do ponto onde a execução foi suspensa, ou próximo a ele.

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

---

[5] Transparência, dentro do contexto da área de Sistemas Computacionais Distribuídos é a não preocupação do usuário (programador) com a migração de tarefas. Menor o nível de transparência, maior será exigência do sistema de que o programador utilize funções de migração providas pelas bibliotecas do sistema operacional distribuído.

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.

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 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.
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.

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*.

3.2.1.1 Transferência do contexto

Transferência do contexto de uma aplicação engloba a transferência dos valores dos registradores do processador (PC, *Stack Pointer*, uso geral, etc.), dos dados (variáveis), do código da aplicação e das informações a respeito dos recursos (arquivos, dispositivos) utilizados pelo processo. Fuggetta (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.

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.

Cópia (Total Freezing)

Esse método congela a aplicação enquanto copia para o nó destino todo o contexto da aplicação (Figura 3.9) (SINHA, 1997). 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, 2005) (BERTOZZI, 2006).

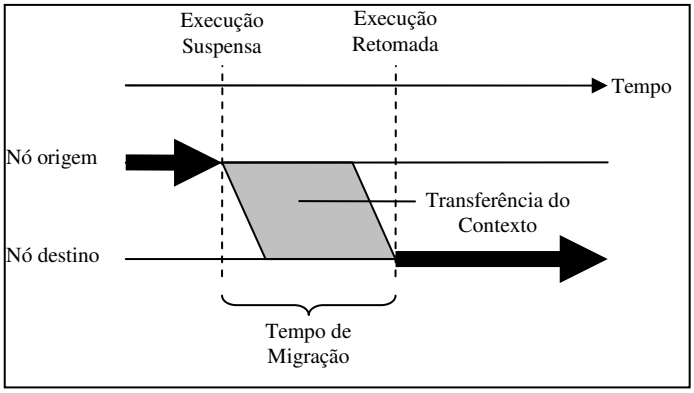


Figura 3.9: Transferência de contexto por cópia.

Pré-cópia

Nesse método o processo a ser migrado continua a ser executado no nó origem enquanto o contexto é migrado (Figura 3.10) (SINHA, 1997). 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).

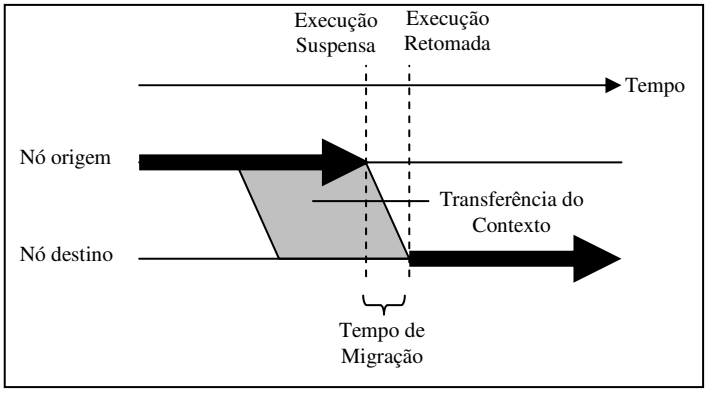


Figura 3.10: Transferência de contexto por pré-cópia.

Cópia sob Demanda (Copy-On-Reference)

Esse método baseia-se na prerrogativa de que apesar de 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 (Figura 3.11) (SINHA, 1997). Porém, quando o processo migrado necessitar de algum dado, este deverá ser buscado no nó origem causando uma grande latência no acesso ao mesmo.

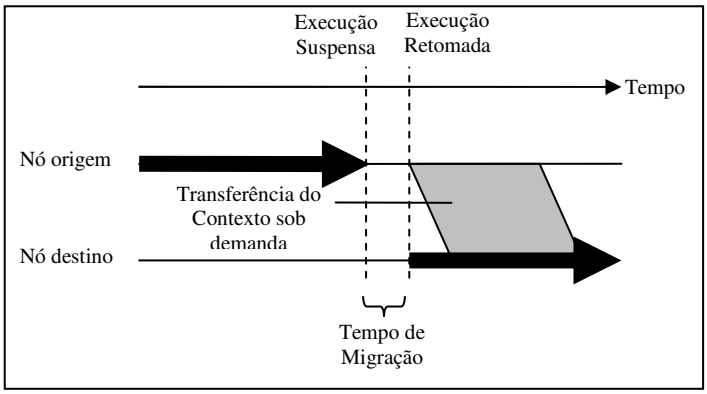


Figura 3.11: Transferência de contexto por cópia sob demanda.

Técnicas de Servidor

Técnicas de Servidor consistem em utilizar um nó seguro como servidor de dados, esperando 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.

Segundo Barcelos (2006), a Tabela 3.2 apresenta uma comparação entre diferentes tipos de transferência de contexto.

Tabela 3.2: 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

3.2.1.2 Métodos de Redirecionamento de Mensagens

Na migração de um processo de um nó para outro, é necessária a garantia de 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 do processo 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 do processo. Entretanto, não existe uma única forma de solucionar o problema das mensagens pendentes e, na proposta dos diversos sistemas operacionais distribuídos, diferentes técnicas de lidar com o problema foram propostas: (i) Reenvio de mensagens: o método consiste em forçar o reenvio das mensagens por parte do emissor; (ii) Nó origem: consiste em manter em uma tabela no nó origem a nova localização do processo após a migração; (iii) *Link Transversal*: 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 das categorias 2 e 3 o sistema armazena no nó origem um *link*, informando a nova localização do processo; (iv) *Link Update*: 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. Assim, mensagens da categoria 3 são enviadas diretamente ao nó destino. Mensagens das categorias 1 e 2 são armazenadas no nó origem até o reinício da execução da aplicação, quando são, então, encaminhadas ao nó destino.

3.2.2 Gerenciador de Carga

Em sistemas distribuídos, 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 distribuídos gerenciados por 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, manter o sistema executando dentro de restrições da aplicação, dá-se o nome de *gerenciamento de carga do sistema*.

Na implementação de um gerenciador de carga duas questões são relevantes: que informações são para 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 número de tarefas na fila de execução, carga média, utilização do processador, utilização da E/S, quantidade de tempo livre do processador, quantidade de memória livre no nó (SMITH, 1988).

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. 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.

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 rede informando sua situação e espera a resposta de outro nó informando que pode dar início à migração. Caso o nó sobrecarregado tenha uma informação fiel 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*.

Também, pode-se 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.

Dentro deste contexto, existem heurísticas e algoritmos para o gerenciamento de carga nos sistemas distribuídos. Tais algoritmos podem causar balanceamento ou concentração de carga, de modo a alcançar algum determinado objetivo.

Em redes do tipo malha é um pouco mais complexo o gerenciador de carga decidir qual tarefa será migrada, porque o custo de comunicação entre processadores e a distância entre os mesmos precisam ser levados em conta. Em sistemas do tipo barramento, ao contrário, a distância lógica entre processadores é sempre um.

Algoritmos de escolha ótimos exigem conhecimento detalhado do estado do sistema, para basicamente testar todas as possibilidades e escolher a melhor. Dessa forma, apresentam alto custo de processamento, acabando por concorrer com as aplicações em execução no sistema em caso de serem aplicados *on-line*.

Por conseqüência, em sistemas com alocação *on-line* geralmente utiliza-se algoritmos sub-ótimos ou heurísticas, que apresentam uma boa relação custo-benefício.

3.2.3 Análise

Nesta seção será discutida a aplicabilidade do mecanismo de migração de tarefas em sistemas embarcados multiprocessados (MPSoCs), em especial aqueles interconectados por meio de redes-em-chip (NoCs).

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. Dessa

maneira, a alteração das aplicações a fim de implementar o suporte à migração não é um problema maior.

Já a escalabilidade, com o ritmo de aumento do potencial 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 grelha seja razoavelmente 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, 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. Assim 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 (BARCELOS, 2006). 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 páginas 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 a dissipação de energia por parte dos *links* e roteadores.

No que tange o redirecionamento de mensagens, novamente no intuito de diminuir a complexidade e o consumo, uma estratégia viável 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. No entanto, esta perda de mensagens transmitidas poderia causar impacto na perda de *deadlines* em um sistema onde poderão existir processos de *hard real time* rodando em nós suscetíveis a sobrecarga. Uma estratégia a ser considerada e mensurada, embora que comprometa a transparência seria informar os processos comunicantes da migração da aplicação, fazendo que o envio de mensagens seja suspenso momentaneamente até segunda ordem. Assim mensagens não seriam perdidas e, principalmente, não seriam enviadas, aliviando a rede e o consumo.

Finalizando, 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 HEURÍSTICAS DE ALOCAÇÃO

A alocação consiste em distribuir elementos em recipientes, considerando uma função de distribuição. No âmbito desta tese, elementos representam tarefas e recipientes representam processadores. A função de distribuição é uma heurística ou um conjunto de heurísticas que define regras para onde as tarefas devem ser alocadas. Encontrar uma alocação ótima é reconhecidamente um problema NP-Difícil (LEUNG, 1992). Por isso várias técnicas foram desenvolvidas desde o inicio da computação para contornar esse problema.

Existem várias técnicas de alocação e otimização na literatura. São elas:

- *algoritmos de indexação*
- *bin-packing clássico*
- *bin-packing com restrição*
- *bin-packing bidimensional*
- *clustering*
- *simulated annealing*

4.1 Algoritmos de indexação

Leung, Arkin *et al.* (2002) desenvolveram um algoritmo que consiste em dividir a malha em pequenos blocos quadrados (página), que constituem a unidade básica de alocação. Uma requisição por k processadores é satisfeita através de alocação de páginas livres até atingir a quantia desejada. A ordem em que as páginas são visitadas depende da indexação utilizada. Os autores propõem usar curvas fractais para indexar os processadores (Figura 4.1). O algoritmo considera cada conjunto de linhas que compõe um intervalo contíguo de processadores livres como um recipiente parcialmente preenchido. Quando nenhum dos recipientes contém processadores livres suficientes para satisfazer uma requisição, seleciona-se aquele que ocupa o menor número de linhas. Existindo recipientes com espaço livre suficiente, a alocação é feita combinada com algoritmos *bin-packing*. Os experimentos indicaram que a escolha da curva é mais importante que o algoritmo usado para a seleção dos processadores ao longo desta curva, mas ambas as escolhas afetam a performance do sistema.

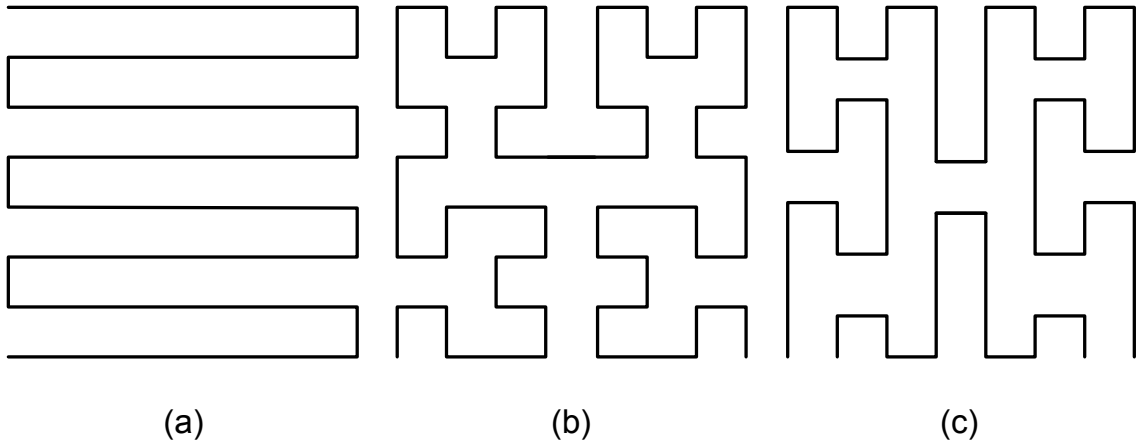


Figura 4.1: Curvas: (a) S (b) Hilbert (c) H-Index.

Dentro do contexto deste trabalho, os algoritmos de indexação estão fora de escopo, devido ao tamanho da malha. Segundo Leung, Arkin *et al.* (2002), os experimentos foram realizados com malhas de 20x20 processadores e tais algoritmos de indexação mostraram impacto no sistema de maneira significativa. Porém Wronski (2007) demonstrou em seus experimentos que em uma malha pequena (menor que 10x10) tais algoritmos têm um impacto desprezível.

As várias técnicas de alocação citadas não são mutuamente exclusivas, podendo ser combinadas de acordo com a necessidade. Nas próximas seções são discutidas técnicas de *bin-packing* e *clustering*.

4.2 Bin-packing clássico

O modelo de *packing* é formado por um conjunto de itens que devem ser alocados em um conjunto de recipientes. Na teoria clássica esse problema é chamado de *bin-packing* (JOHNSON, 1973) e admite a possibilidade de que tanto itens quanto recipientes sejam elementos multi-dimensionais, como nas redes malhas.

O BP – *bin-packing* clássico é definido como uma seqüência de itens $L = \{k_1, k_2, ..., k_n\}$, cada um com tamanho normalizado $s(k_i) \in (0,1]$, onde 1 representa 100%. Os itens devem ser alocados em um número mínimo de recipientes, isto é, L deve ser particionado em um conjunto mínimo de m subconjuntos $B = \{B_1, B_2, ..., B_m\}$ tal que $level(B_j) \leq 1$ e $1 \leq j \leq m$, onde $level(B_j)$ representa a ocupação do j-ésimo recipiente dada por $\sum_{k_i \in B_j} s(k_i)$.

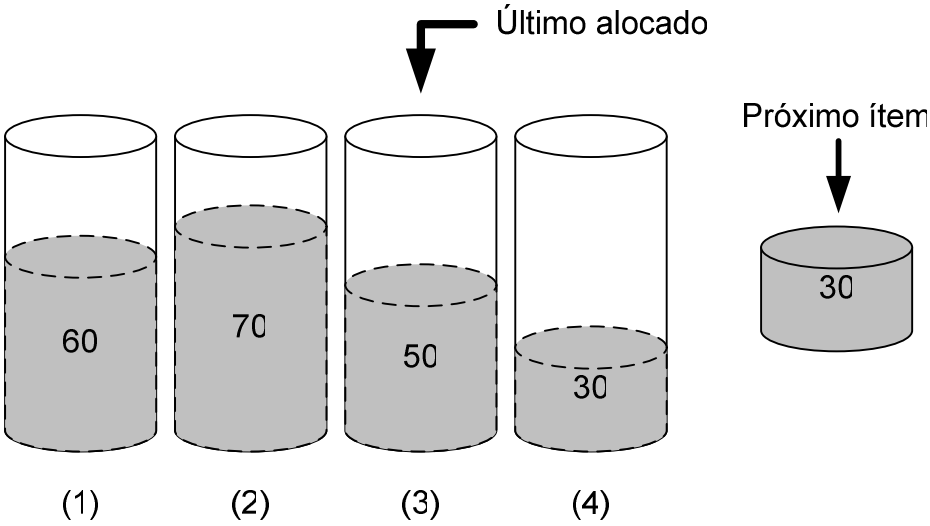


Figura 4.2: Modelo de *bin-packing*.

O modelo de *bin-packing* é representado na Figura 4.2, onde se tem quatro recipientes: B_1, B_2, B_3 e B_4, cada um com capacidade de 100 unidades (100%), sendo que: $level(B_1) = 60$, $level(B_2) = 70$, $level(B_3) = 50$ e $level(B_4) = 30$. O próximo item a ser alocado (k_i) tem tamanho $s(k_i) = 30$.

O BP é um problema NP-Completo e por isso várias heurísticas são propostas para a sua solução, são elas: NF – *Next Fit*, FF – *First Fit,* BF – *Best Fit* e WF – *Worst Fit*.

O NF é a heurística de *bin-packing* mais simples, na qual somente o último recipiente utilizado (B_j) fica ativo, este é denominado "aberto". Na Figura 4.2 o recipiente aberto é o B_3. Um item pode ser alocado em B_j (ou B_3 nesse caso) se passar no teste de aceitação da heurística, que é: $s(k_i) \leq 1 - level(B_j)$, ou seja, se o item é menor que o espaço disponível em B_j. Caso contrário, B_j é fechado e o próximo recipiente B_{j+1} é aberto. Dessa forma, o item é alocado sempre no próximo recipiente que couber, a partir do recipiente aberto. Aplicando esse raciocínio na Figura 4.2, o próximo item será alocado em B_3.

A diferença entre o NF e o FF é que, neste último, o teste de alocação é aplicado em ordem sempre a partir do primeiro recipiente e não do último utilizado. Dessa forma, o FF tende a concentrar os itens nos primeiros recipientes, criando assim menos fragmentação. Na Figura 4.2 o próximo item seria alocado, com o FF, em B_1.

No BF, a cada alocação todos os recipientes são avaliados e o que apresentar a menor sobra de espaço após a alocação será selecionado. Dessa forma, o recipiente B_2 seria o selecionado.

O WF aplica o teste de alocação inverso ao do BF, alocando o item no recipiente em que resta mais espaço após a alocação. Por isso, com o WF, o próximo item seria alocado no recipiente B_4.

Caso a alocação seja feita estática (*off-line*), a lista de itens pode ser ordenada a priori, gerando soluções mais eficientes. A melhor ordenação é a decrescente que primeiro acomoda os itens maiores e depois os menores nos espaços restantes.

A eficiência de uma heurística é medida pela razão de performance $R_A(L) \equiv {A(L)}/{OPT(L)}$, onde para uma lista L, $A(L)$ representa o número de conjuntos formados quando a heurística A é aplicada sobre a lista L e $OPT(L)$ é o número mínimo de conjuntos necessários para alocar essa mesma lista. Se considerarmos todas as listas L possíveis, R_A^∞ representa a razão de performance absoluta da heurística A.

O NF executa em tempo linear com $R_{NF}^\infty = 2$. Tanto BF quanto FF executam em tempo $O(n \log n)$ com $R_{FF}^\infty \approx 1.69103...$ se a estrutura de dados apropriada for utilizada. O WF mantém o mesmo R_{BF}^∞ do BF com a pior fragmentação, ou de outra forma, o melhor balanceamento de carga. Esses valores são para alocações *off-line* com ordenação decrescente da lista.

4.3 Bin-packing com restrição

O modelo de *packing* com restrição é formado por um conjunto de itens que devem ser alocados em um conjunto de recipientes, porém, é usada uma variável de restrição. Esta restrição, no âmbito de sistemas embarcados, é caracterizada pela ocupação de memória, potência, etc.

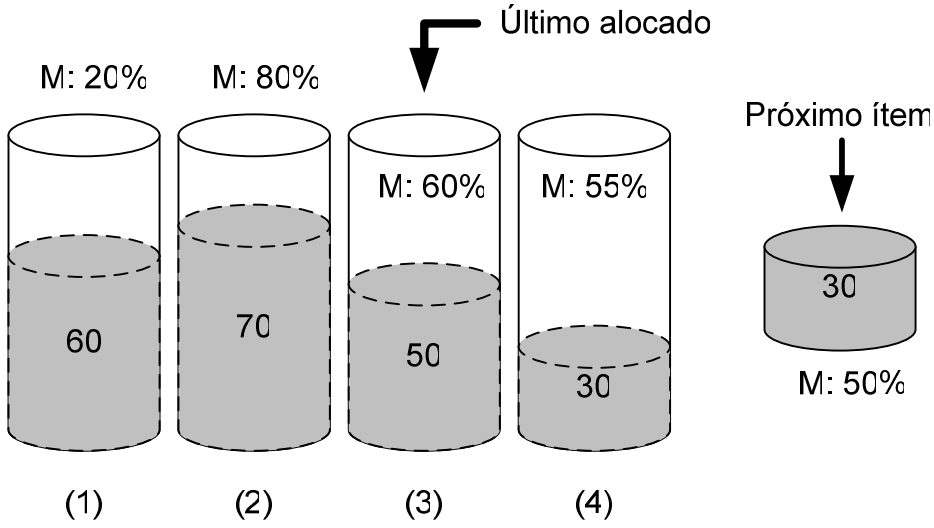


Figura 4.3: Modelo de *bin-packing* com restrição.

O modelo de *bin-packing* com restrição é representado na Figura 4.3, onde se tem quatro recipientes: B_1, B_2, B_3 e B_4, cada um com capacidade de 100 unidades (100%), sendo que: $level(B_1) = 60$, $level(B_2) = 70$, $level(B_3) = 50$ e $level(B_4) = 30$. Cada recipiente tem uma restrição denominada "M". Os recipientes têm as seguintes restrições: $constraint(B_1) = 20$, $constraint(B_2) = 80$, $constraint(B_3) = 60$ e $constraint(B_4) = 55$. Para qualquer algoritmo de *bin-packing*, neste caso, o item será alocado somente na posição B_1, pois a capacidade do recipiente nesta posição

($level(B_1)$) será de 90%, sem exceder os 100%, e a restrição ($constraint(B_1)$) será de 70% (a soma de 50% do item e 20% do recipiente B_1). Nos outros recipientes, a alocação do item excederá a restrição em mais de 100%. Analogamente, dentro do contexto dos sistemas embarcados, os processadores podem ser representados por recipientes, a utilização dos processadores pode ser representada pela capacidade dos recipientes e a ocupação da memória pode ser representada pelas restrições dos recipientes.

4.4 Bin-packing bidimensional

O *bin-packing* pode também ser multidimensional. No caso 2D, se considera o problema de alocar um conjunto de itens retangulares em recipientes também retangulares minimizando a quantidade de recipientes utilizados. Uma variação é o *strip-packing*, no qual se considera um único recipiente com largura fixa e um comprimento infinito. O objetivo é alocar todos os itens dentro do menor comprimento possível (LODI, 2002). A Figura 4.4 apresenta o modelo do *bin-packing* bidimensional.

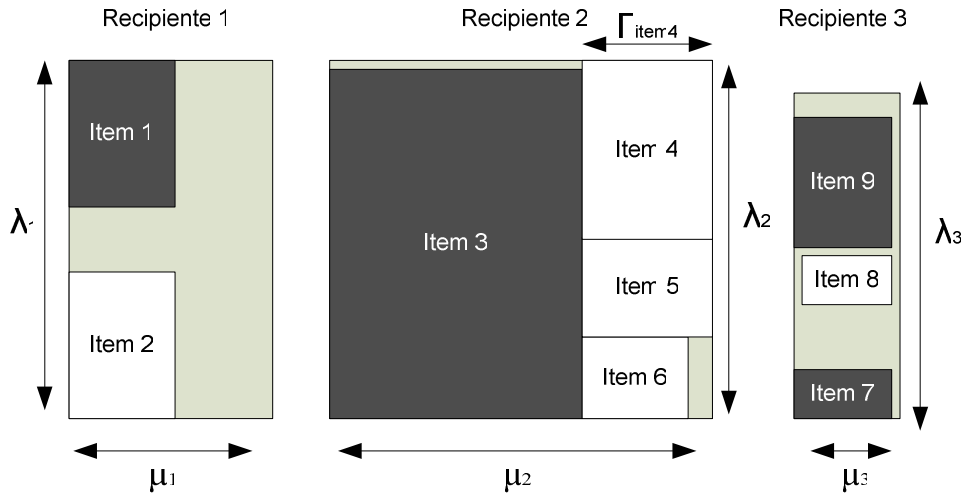


Figura 4.4: Modelo de *bin-packing* bidimensional.

Cada recipiente, no caso do 2D, tem duas capacidades que representam o comprimento e a largura. Cada item também tem sua própria largura e comprimento. Porém deve-se levar em conta que não é apenas a área que será determinante para alocar um item em um recipiente. Considera-se, por exemplo, o item 4 na Figura 4.4. Se for considerada apenas a área (produto da largura com o comprimento) do item 4, este pode ser alocado no recipiente 3, pois este tem maior área que o item 4. Porém a largura, representada por Γ_{item4}, do item 4 é maior que a largura do recipiente 3 representada por μ_3. Desta forma, não é possível alocar o item 4 no recipiente 3, pois deve-se levar encontra não apenas a área da ocupação, mas também as capacidades do recipiente.

A Figura 4.5 apresenta três itens com a mesma área para serem alocados ao recipiente. A área dos items é menor que a área do recipiente. A Figura 4.5-a apresenta um caso particular de rejeição da alocação. O item 1 tem menor área que o recipiente, e teoricamente este poderia ser alocado nele. Contudo, a largura do item 1 é maior que a largura do recipiente. Desta forma, houve violação da capacidade da largura do recipiente. Consequentemente o item 1 não pode ser alocado no recipiente. A Figura

4.5-b apresenta um outro caso particular de rejeição da alocação. O item 2 tem menor área que o recipiente, e teoricamente este poderia ser alocado nele também, da mesma forma como foi observado na Figura 4.5-a. Todavia, a altura do item 2 é maior que a altura do recipiente. Neste caso, houve violação da capacidade da altura do recipiente, impossibilitando a alocação do item 2 no mesmo. A Figura 4.5-a apresenta o único caso em que um item (item 3) de menor área que o recipiente pode ser alocado no mesmo. Em razão da não violação da capacidade da largura e altura do recipiente, o item 3 pode ser alocado sem nenhum impedimento.

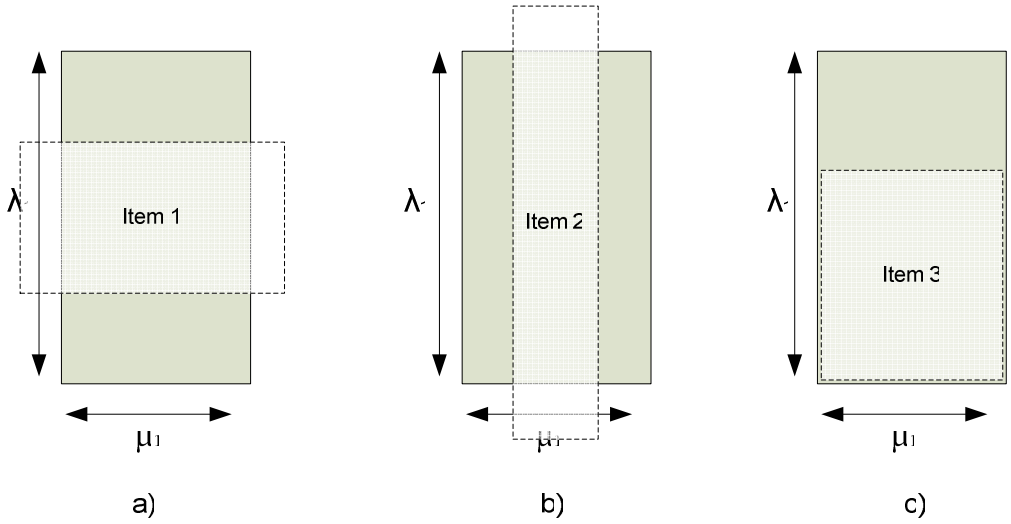


Figura 4.5: Itens de mesma área, porém com diferentes alturas e larguras.

Enquanto *bin-packing* multidimensional e *strip-packing* são problemas geométricos, no *vector-packing* as dimensões dos itens e recipientes são independentes. Nesse caso os elementos do vetor representam objetivos concorrentes e uma forma de medir a utilização considerando todas as dimensões do vetor deve ser provida (Richard, Michael *et al.*, 1984).

Como exemplo, Beck e Siewiorek (1996) podem ser citados. Eles modelaram alocação de tarefas em multiprocessadores como um problema de *vector-packing*, onde a estrutura de comunicação utilizada é uma rede CAN (*Controller Área Network*). No modelo proposto existem vários processadores com restrições de capacidade de processamento, tamanho de memória RAM, ROM, etc; totalizando 6 dimensões. Cada tarefa do sistema apresenta certa demanda por cada um destes elementos. Claramente cada recipiente é multidimensional, entretanto as dimensões são independentes. Existe ainda um recipiente unidimensional que representa o barramento de comunicação, uma vez que uma mensagem pode ser considerada como uma tarefa alocada no barramento. O objetivo é alocar todas as tarefas sem ultrapassar a capacidade dos recipientes. As heurísticas usadas para calcular a utilização ou capacidade dos recipientes são: tamanho do maior elemento do vetor e média dos elementos.

4.5 Clustering

No *clustering* adota-se um modelo de tarefas dependentes que é representado por um grafo de tarefas acíclico, composto por um conjunto de nodos L e outro de arestas A. Cada nodo $k_i \in L$ é uma tarefa e cada aresta $a_{i,j} \in A$ é uma dependência entre k_i e k_j.

Segundo Backer e Jain (1981), “na análise de *clusters*, um grupo de objetos é dividido em um número de subgrupos mais ou menos homogêneos com base em uma medida de similaridade, tal que a similaridade entre os objetos dentro de um subgrupo seja maior que a entre objetos pertencendo a grupos diferentes”.

Na literatura existem alguns algoritmos de *clustering*. Dois deles são detalhados nas próximas sub-seções.

4.5.1 Edge Zeroing (Zerando Arestas)

EZ - Edge Zeroing (SARKAR, 1989) é um método em dois passos para escalonar tarefas com comunicação em sistemas multiprocessados. A primeira fase é a aplicação de um clustering que cria subgrupos de tarefas, onde arestas em um mesmo cluster são consideradas zeradas. A segunda fase consiste na alocação dos clusters para os processadores. A idéia básica é unir clusters, zerando as arestas de maior peso entre clusters. Uma união é permitida se o comprimento do escalonamento não aumentar. Se o comprimento do escalonamento aumentar, significa que aumentou a serialização das tarefas e consequentemente reduziu o paralelismo. Dessa forma, depois de cada união o comprimento do escalonamento precisa ser recalculado. Para minimizar a comunicação, as arestas são ordenadas em ordem decrescente de peso e, dessa forma, aquelas que representam maior comunicação são avaliadas antes. Este algoritmo apenas considera pesos de arestas e não leva em conta as dependências entre as tarefas.

4.5.2 Linear Clustering (Agrupamento Linear)

Gerasoulis e Yang (1993) classificam um *cluster* em linear e não linear. Um *cluster* é chamado não linear se possui pelo menos duas tarefas independentes (uma tarefa conectada a dois nós folhas, por exemplo). A Figura 4.6-a mostra dois clusters não lineares com as tarefas de 0 a 5 e o outro com as tarefas 6 a 10. Caso contrário, o cluster é chamado de linear e todas as tarefas fazem parte da mesma cadeia de dependências. A Figura 4.6-b apresenta três clusters: o mais à esquerda agrupa tarefas 1, 4 e 7; o cluster central que agrupa as tarefas 0, 2, 5, 8 e 10; e finalmente o cluster mais à direita que agrupa as tarefas 3, 6 e 9. O *clustering* linear explora totalmente o paralelismo do grafo de tarefas, enquanto o não linear reduz o paralelismo serializando tarefas independentes no mesmo *cluster* para diminuir o custo de comunicação. Os autores também propõem impor a restrição de linearidade ao algoritmo EZ, para que o algoritmo rejeite zerar uma aresta se isso resultar em um *cluster* não linear. Com isso evita-se recalcular o comprimento do escalonamento a cada passo e a complexidade do algoritmo diminui. Como conseqüência dessa modificação, o algoritmo Linear Clustering (LC) remove o paralelismo inútil do escalonamento, aquele que não diminui o comprimento deste, podendo assim resultar em um número de *clusters* diferente do esperado. Linear clustering provê maior paralelismo (menos comprimento do escalonamento) e menor volume de comunicação entre os clusters em relação ao algoritmo EZ. A Figura 4.7 apresenta o algoritmo *Linear Clustering* apresentado em forma de pseudocódigo.

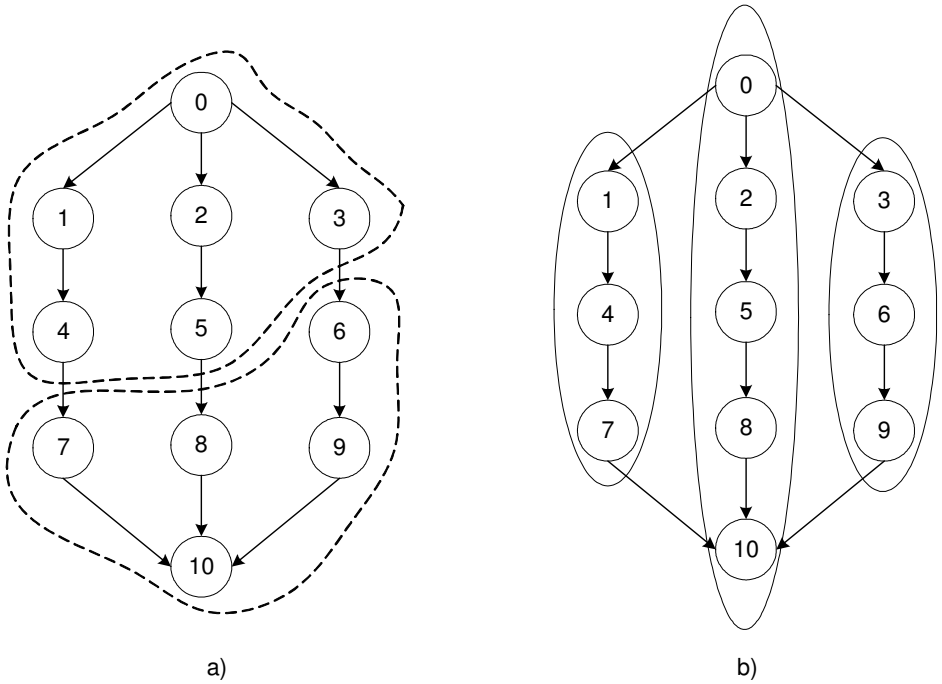


Figura 4.6: *Clusters:* (a) não linear (b) linear.

```
PROCEDIMENTO LINEAR_CLUSTERING()
INPUT: GRAFO(nonlinear);
OUTPUT:CLUSTER_LIST()(linear);
CLUSTER α,β,ω;
ARESTA φ;
NODO Ψ;
LISTA_DE_ARESTAS Γ;

PARA cada nodo Ψ do GRAFO FAÇA
        CLUSTER_LIST.add(Ψ); (Cria um cluster para cada nodo)
FIM_PARA

Cria lista de arestas Γ do GRAFO;
Ordene lista de arestas Γ (decrescente);

PARA cada aresta φ de Γ FAÇA
        α = getCluster(φ.nodo_inicial);
        β = getCluster(φ.nodo_final);
        ω = merge_clusters(α,β);
        SE (linear(ω) = TRUE) E (α ≠ β) ENTÃO
                CLUSTER_LIST.remove(α);
                CLUSTER_LIST.remove(β);
                CLUSTER_LIST.add(ω);
        FIM_SE
FIM PARA
```

Figura 4.7: Pseudocódigo do algoritmo *Linear Clustering*.

No algoritmo para resolução *do Linear Clustering*, CLUSTER_LIST é um conjunto de agrupamento de nodos unitários. O conjunto GRAFO contém todos os nodos e arestas do grafo a ser clusterizado. O algoritmo faz uma varredura na lista de arestas e

testa se a junção de dois clusters é linear. Caso sejam lineares e diferentes, eles são adicionados na lista CLUSTER_LIST.

Depois de varrer todas as arestas do grafo, os nodos então são agrupados e as arestas internas ao agrupamento (cluster) permanecem em um mesmo processador, anulando os efeitos de comunicação e contenção da arquitetura de comunicação. Desta forma aumenta-se o paralelismo da aplicação agrupando tarefas seqüenciais e com alto peso de comunicação nas arestas.

4.6 Simulated Annealing

Simulating Annealing (SA) (KIRKPATRICK, 1983) é um função meta-heurística que pertence à classe dos algoritmos de pesquisa local, que se pode aplicar aos problemas de otimização combinatória. SA é consiste numa técnica de busca local probabilística, e se fundamenta numa analogia com a termodinâmica.

Esta meta-heurística é uma metáfora de um processo térmico, dito *annealing* ou recozimento, utilizado em metalurgia para obtenção de estados de baixa energia num sólido. O processo consiste de duas etapas: na primeira a temperatura do sólido é aumentada para um valor máximo no qual ele se funde; na segunda o resfriamento deve ser realizado lentamente até que o material se solidifique, sendo acompanhado e controlado esse arrefecimento (esfriamento). Nesta segunda fase, executada lentamente, os átomos que compõem o material organizam-se numa estrutura uniforme com energia mínima. Isto provoca que os átomos desse material ganhem energia para se movimentarem livremente e, ao esfriar de forma controlada, dar-lhes uma melhor hipótese de se organizarem numa configuração com menor energia interna, para ter, como resultado prático, uma redução dos defeitos do material.

De forma análoga, SA tem como objetivo encontrar a melhor solução entre um número finito de soluções. Consiste na procura de um ótimo local cujo valor está próximo do valor do ótimo global, não sendo possível saber se a solução encontrada corresponde ao ótimo local ou ao global. Permite encontrar soluções próximas da ótima sem grande esforço computacional, sendo, portanto um processo viável para usar em situações em que o conhecimento é escasso ou que aparenta difícil aplicação algorítmica.

O algoritmo funciona da seguinte maneira: inicialmente deve ser gerada uma solução aleatória, o que equivale a um material cheio de imperfeições. Inicia-se a simulação com uma temperatura alta. A temperatura é diminuída ao longo da simulação, e determina que o algoritmo apresente comportamentos diferentes dependendo da temperatura em que se encontra o sistema. Em altas temperaturas, ocorrem movimentos bruscos na configuração: grandes saltos são dados. Em temperaturas baixas, entretanto, pequenas variações acontecem, de forma que o sistema evolui para um estado estacionário.

A implementação do algoritmo, como apresentado na Figura 4.8, consiste de dois laços aninhados: o laço externo controla a temperatura enquanto que o laço interno realiza três passos: perturbação, avaliação e decisão. A partir de uma solução corrente, uma perturbação gera uma nova solução, vizinha à anterior. A nova solução é então avaliada e tem-se um ΔC (custo final – custo inicial) indicando a qualidade da nova solução em relação à anterior. Se a solução proposta é melhor que a anterior, então esta é a nova solução corrente. Se for pior, aqui está o ponto importante do SA: a solução

proposta é aceita com probabilidade $e^{\frac{-\Delta C}{KbT}}$ onde ΔC é a variação do custo da solução, *Kb* é a constante de *Boltzman* e *T* é a temperatura do sistema. Assim, o sistema evolui de acordo com a distribuição de *Boltzman*.

```
PROCEDIMENTO SIMULATED_ANNEALING()
INPUT: temperaturaInicial, temperaturaFinal: REAL;
OUTPUT: solução: ESTADO

solução = soluçãoAleatória
temperatura = temperaturaInicial;

ENQUANTO (temperatura > temperaturaFinal) FAÇA
        REPITA
                novaSolução = Perturbação(solução);
                ΔC = Avaliação(solução, novaSolução);
                SE (Decisão(ΔC, temperatura) == ACEITA) ENTÃO
                        solução = novaSolução;
                FIM_SE
        ATÉ (sistema em equilíbrio nessa temperatura);
        temperatura = funçãoDeResfriamento(temperatura);
FIM_ENQUANTO
FIM_PROCEDIMENTO

FUNÇÃO Perturbação(solução:ESTADO): ESTADO
RETORNA estado válido, vizinho a solução;
FIM_FUNÇÃO

FUNÇÃO Avaliação(solução:ESTADO, novaSolução:ESTADO): REAL
RETORNA Custo(novaSolução) – Custo(solução);
FIM_FUNÇÃO

FUNÇÃO Decisão(ΔC:REAL, temperatura:REAL): BOOLEAN
Aceita_condição:BOOLEAN
        SE (ΔC < 0) OU (e^(-ΔC/KbT) < rand())
        ENTÃO aceita_condição = ACEITA;
        SENÃO aceita_condição = REJEITA;
RETORNA aceita_condição;
```

Figura 4.8: Pseudocódigo do algoritmo *Simulated Annealing*.

Apesar do algoritmo ser simples e facilmente aplicável a qualquer problema, muitas questões surgem quanto a sua implementação. SA apresenta muitos parâmetros, e a qualidade da solução e os tempos de execução do algoritmo estão intimamente relacionados à configuração apropriada desses parâmetros. Dentre outros, os pontos mais importantes do algoritmo são:

- *Função de Avaliação de Custo*: SA realiza refinamento da solução em busca da otimização de um conjunto de parâmetros. De fato, o problema de otimização deve ser bem definido, de forma que o cálculo da função de avaliação da solução esteja de acordo com a solução desejada. O desenvolvimento de uma boa função de avaliação requer conhecimento do problema e não é um trabalho trivial.

- *Função de Perturbação*: A função de perturbação deve permitir que qualquer solução seja visitada. Além disso, é importante que a função de perturbação não ocasione correlações no sistema. Deve ser utilizado um gerador de números aleatórios confiável.
- *Função de Resfriamento*: A função de resfriamento mais utilizada é tal que a temperatura é reduzida por uma taxa constante: $T = \alpha T$. Contudo, existem vários esquemas de funções de resfriamento utilizadas na literatura. A função adotada neste trabalho é uma função adaptativa, onde a temperatura é decrescida por um fator que varia conforme a taxa de aceitação na iteração em questão.
- *Critério de Parada do Algoritmo*: O critério de parada mais utilizado consiste primeiramente de uma avaliação da variação do custo da solução atual em relação à solução obtida na temperatura anterior. Uma vez alcançado um número máximo sem melhoras no custo da solução, o algoritmo pára. Assim, o algoritmo deve parar quando chegamos num estado onde não é possível ocorrer mudanças significativas no custo da solução.

4.7 Contribuições e inovações

Como foi mencionado na Seção 1.4, a principal contribuição deste trabalho é o provimento de metodologias para exploração de espaço de projeto e métodos para atendimento de restrições de projeto, ambos em tempo de execução, em sistemas embarcados baseados em redes-em-chip. Vale a pena ressaltar que estes métodos compreendem a utilização das heurísticas de alocação de tarefas, uso do DVS, PM e finalmente, migração de tarefas.

O objetivo secundário deste trabalho é apresentar a viabilidade da migração de tarefas em sistemas embarcados baseados em redes-em-chip dentro do contexto de consumo de energia e sistemas de tempo real. Dentro deste contexto, mais precisamente, não se tem conhecimento de trabalhos da área que utilizam a migração de tarefas para tentar minimizar o problema de *deadlines* perdidos e minimização de energia em sistemas embarcados baseados em redes-em-chip, pois outros trabalhos da academia atendem apenas requisitos de desempenho. Não se tem conhecimento também de trabalhos que apresentam compromissos entre deadlines perdidos, energia e tamanho de memória. As contribuições secundárias deste trabalho são:

- O uso de heurísticas de alocação e de *clustering* (agrupamento) inspiradas na área de Sistemas Computacionais Distribuídos combinado com os algoritmos de *bin-packing* em uma plataforma suportada por mecanismos de migração de tarefas;
- Exploração dessas heurísticas em termos de energia e deadlines perdidos;
- Implementação de um gerenciador/monitor da utilização de processadores e ocupação de memória para execução de heurísticas de alocação de tarefas;
- Implementação do protocolo de comunicação entre o gerenciador/monitor com os processadores para tornar o modelo mais realista;

- Modificação dos algoritmos *bin-packing* para suporte de restrição da ocupação de memória dos processadores;
- Modificação nos algoritmos *bin-packing* para tornar estes bi-dimensionais, ou seja, que considerem não apenas a utilização (ocupação) do processador, e sim considerem o espaço que as tarefas irão ocupar na memória.

Como inovação, este trabalho, diferentemente dos outros mencionados no Capítulo 3, apresenta heurísticas inspiradas na área dos Sistemas Computacionais Distribuídos que serão utilizadas para prover balanceamento ou concentração de carga. Esses algoritmos são combinados com métodos relativos à área de Sistemas Eletrônicos Embarcados como o desligamento de processadores e escalonamento de tensão em uma plataforma auto-adaptável. A grande maioria das metodologias, de atendimento a restrições em tempo de execução baseadas na área de Sistemas Distribuídos, somente consideram o desempenho como principal restrição. Este trabalho apresenta resultados focando consumo de energia e número de *deadlines* perdidos, os quais são importantes métricas para o projeto e desenvolvimento de sistemas embarcados de tempo-real.

Outra inovação é realizar comparações das heurísticas e indicar qual heurística de atendimento às restrições em tempo de execução será mais adequada para todas ou um conjunto de aplicações. Dependendo dos resultados, pode-se obter uma EEP em termos de heurísticas. Uma heurística pode consumir maior quantidade de energia, porém pode ser mais eficiente para o atendimento de restrições de projeto. Inversamente, pode-se encontrar uma heurística que pode ser menos eficiente para atendimento de restrições, no entanto pode consumir pouca quantidade de energia.

5 ARQUITETURA ALVO

Para demonstrar a viabilidade da migração de tarefas em sistemas embarcados baseados em redes-em-chip, o presente trabalho vem seguindo a abordagem *top-down,* ou seja, primeiramente implementa-se o modelo comportamental da plataforma (processadores, roteadores e mecanismo de migração de tarefas) e dos mecanismos de exploração *on-line* (heurísticas de distribuição de carga) e depois desenvolve-se componentes da plataforma em um nível mais baixo de abstração (por exemplo, arquitetura mais detalhada de processadores), para realimentar com dados mais realistas o modelo comportamental citado anteriormente. Para implementação desse modelo comportamental da plataforma, é necessário o desenvolvimento de elementos processantes em um nível de abstração mais alto. A entrada para a plataforma de simulação são modelos baseados em grafos executados pelo simulador chamado *Serpens*, que será melhor explicado na próxima Seção. Este nível de abstração permite explorar técnicas de escalonamento e alocação de tarefas em tempo de execução em um tempo viável de simulação. Dados de potência e energia são extraídos através de uma biblioteca de software. Porém, a precisão do simulador é discutível, pelo seu nível de abstração maior que o nível de transferência de registradores (RTL – *Register Transfer Level*). Para isso, é necessário um simulador no nível RTL, ou em um nível de abstração próximo a este, para realimentar, em dados estatísticos relacionados à potência, desempenho e energia, o simulador *Serpens*. A ausência deste força a utilização de informações de experimentos baseados em benchmarks e uso de modelos para caracterização de energia do sistema. No grupo local está em andamento o desenvolvimento de simulador da rede-em-chip com processadores implementados em baixo nível de abstração, todos em nível de ciclo de relógio (BARCELOS, 2008).

A seguir, são apresentados a plataforma de simulação utilizada para este trabalho, a organização dos elementos de processamento, o mecanismo de migração, o protocolo de comunicação do núcleo mestre, bem como o fluxo de projeto proposto.

5.1 Plataforma e ferramental

Esta Seção apresenta uma breve descrição do simulador Serpens e os detalhes de implementação do mecanismo de migração de tarefas desenvolvido para este trabalho.

5.1.1 Simulador Serpens

O simulador Serpens foi desenvolvido para simular o comportamento de sistemas embarcados distribuídos que executam e escalonam aplicações representadas por grafos baseados em TGFF (DICK, 1998), que são dinamicamente carregados e executados pelo simulador. Além disso, este simulador é baseado em redes-em-chip e desenvolvido em SystemC no nível de transação (TLM – *Transaction Level Model*). O simulador também executa algoritmos de *clustering* e *bin-packing* para alocação de tarefas e suporta escalonamento dinâmico de tarefas (EDF) em um mesmo processador. Mecanismos de DVS e DPM são aplicados na execução de tarefas no sistema para minimizar o consumo de energia. Os elementos processantes do simulador são baseados em processadores Java chamados FemtoJava (ITO, 2001), todos com memória local, e a arquitetura de comunicação é baseada em rede-em-chip SoCIN (ZEFERINO, 2003a) (ZEFERINO, 2003b), ambos desenvolvidos no nosso grupo de pesquisa. O simulador usa a biblioteca chamada Orion (WANG, 2002), usada também na avaliação de potência da rede-em-chip Xpipes (DALL'OSSO, 2003), para avaliar o consumo de potência e energia.

O modelo de simulação usa os roteadores RaSoC (ZEFERINO, 2003a) (ZEFERINO, 2003b), projetados para a síntese de sistemas embarcados baseados em redes-em-chip com baixo consumo de energia e área. A arquitetura da RaSoC utiliza chaveamento de pacotes baseados em *wormhole*, algoritmo XY para roteamento de pacotes e fluxo de controle baseado em *handshake*. A avaliação de consumo de energia do RaSoC foi implementada usando biblioteca Orion. Cada roteador tem 5 portas bidirecionais com buffer de entrada com o tamanho de 4 phits. O tamanho de cada phit é de 4 bytes.

A hierarquia de memória utilizada atualmente na plataforma é a distribuída não-compartilhada (memória privada local), ou seja, *todos* os processadores fazem referências somente a sua memória local (Figura 5.1) no qual o espaço de endereçamento não é compartilhado. A comunicação entre os processadores é baseada em troca de mensagens. Essa hierarquia foi escolhida *a priori* pelo fato de existir a coerência de dados nas memórias (já que não há compartilhamento) e por esse tipo de organização já estar disponível na plataforma. Atualmente, os experimentos estão sendo realizados utilizando a plataforma Serpens com esta organização.

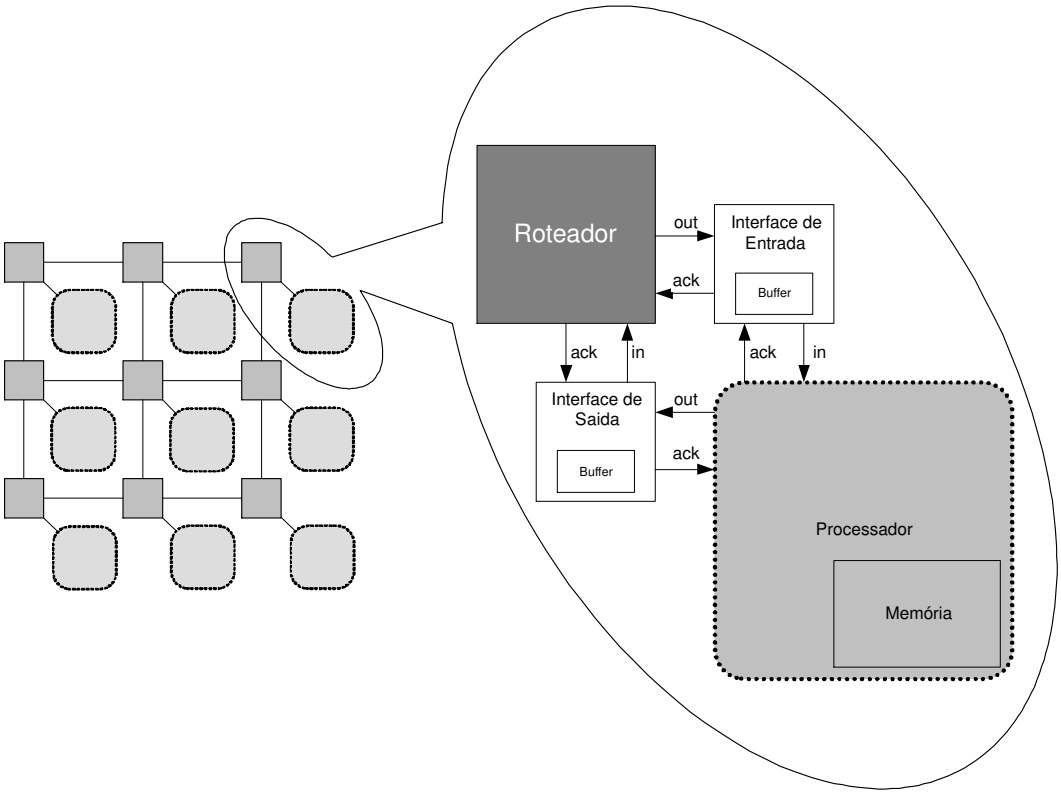


Figura 5.1: Plataforma Serpens com memória local distribuída não-compartilhada e interfaces de E/S de dados.

O escalonador de tarefas de cada elemento processante é baseado na implementação de um escalonamento em prioridade. A política de escalonamento é baseada no algoritmo EDF (o mais cedo deadline primeiro, do inglês, *earliest deadline first*). Para implementar as dependências de tarefas, um algoritmo list-scheduling foi combinado com o EDF.

Como DVS, foi utilizado o algoritmo DAR (*Dynamic Average Rate*) (ZHUO, 2005). Este algoritmo reduz dinamicamente a freqüência do processador, sem comprometer o deadline, aproveitando a diferença entre o WCET (pior caso de tempo de execução, ou do inglês, *worst-case execution time*), e o tempo real da execução (*scheduling slack*), ou seja, ele reduz o tempo desperdiçado com estimativas pessimistas. Os valores máximos e mínimos da freqüência de operação são considerados nesta abordagem. Se o processador está inativo, então a sua freqüência de operação se torna o menor valor de sua freqüência. O algoritmo suporta qualquer valor contínuo entre as freqüências máximas e mínimas de operação do processador desde que exista este recurso no mesmo.

5.1.2 Modelo dos roteadores

Os roteadores foram implementados baseados no modelo RTL original e assim são bastante precisos no que diz respeito a consumo de energia e comportamento temporal. O modelo original é o roteador de redes malha RaSoC (ZEFERINO, 2004), desenvolvido para a síntese de sistemas embarcados em FPGA – *Field-programable Gate Array*, com baixo consumo de energia e área.

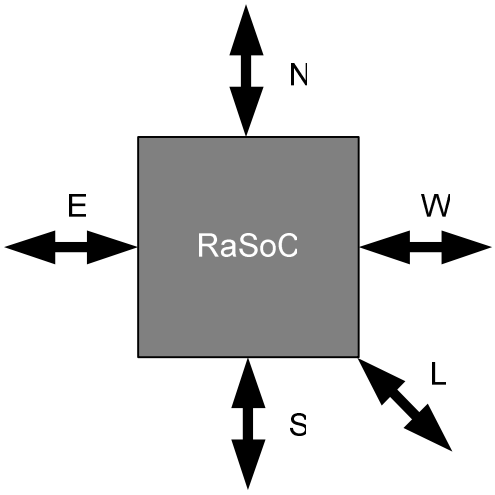


Figura 5.2: Visão externa do RaSoC.

O RaSoC utiliza chaveamento por pacotes baseados em *wormhole*, roteamento XY e controle de fluxo por *handshake*. Externamente, o RASoC é um roteador com 5 portas bidirecionais (Figura 5.2). São elas: norte (N), sul (S), leste (E), oeste (W) e local (L). Dependendo da posição do roteador, uma delas pode não ser implementada, reduzindo a área da NoC.

Cada porta do RaSoC inclui dois canais de comunicação unidirecionais opostos (Figura 5.3). Os canais são compostos por bits de dados, de marcação de pacotes e de controle de fluxo. Os bits de marcação de pacotes são os sinais de início e fim: BOP – *Begin Of Packet* e EOP – *End Of Packet*. O BOP é ativado apenas no cabeçalho do pacote e EOP somente no último *phit* deste. Tamanhos típicos para um *phit* são 8, 16 ou 32 bits. Os bits de controle de fluxo são: *val* e *ack*. Em *val* é informada a existência de um novo *phit* no canal e *ack* é o aviso de que o mesmo foi aceito pelo destino.

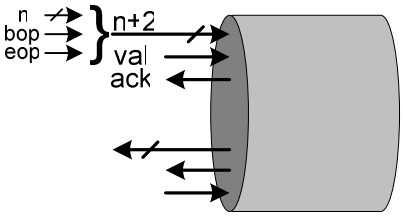


Figura 5.3: Canal de comunicação do RaSoC.

Uma vez que um modelo TLM foi utilizado na implementação, cada canal foi descrito em SystemC como sendo composto por dois canais *sc_channel* opostos, um para dados (*in* ou *out*) e outro para confirmação (*in_ack* ou *out_ack*), como demonstrado na Figura 5.1.

Internamente, o RASoC é implementado de forma distribuída. Cada porta de entrada possui um controlador de roteamento. Vários deles podem rotear para uma mesma porta de saída. Quem decide qual entrada terá acesso à saída é o árbitro da mesma. Dessa forma, quatro controladores de roteamento se ligam a um árbitro. Como é proibido que um roteamento seja feito utilizando as portas de entrada e saída do mesmo canal, o quinto controle de roteamento não é conectado ao árbitro do próprio canal. A política de arbitragem do RaSoC é *round-robin* e somente as portas de entradas possuem buffers.

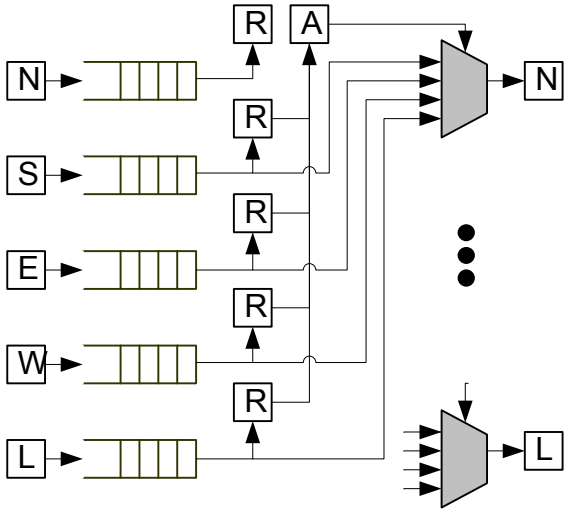


Figura 5.4: Estrutura interna do RaSoC.

Na Figura 5.4 é mostrada a estrutura interna do RaSoC. Para simplificar o desenho, somente as conexões com a porta de saída norte (N) são apresentadas.

A potência, tanto estática quanto dinâmica, é calculada com a biblioteca Orion, utilizando modelos de capacitância combinados aos chaveamentos causados nos componentes arquiteturais do roteador em cada transação. A utilização da biblioteca se dá através das funções de sua interface, como as funções *SIM_buf_power_data_write* e *SIM_buf_power_data_read,* utilizadas respectivamente para se escrever e ler pacotes nos buffers.

Para chamar a função *SIM_buf_power_data_write* é necessário informar o novo pacote que está sendo escrito, o último que foi escrito e o que está armazenado no buffer na posição onde o novo será escrito. Para a função de leitura, deve-se informar apenas o pacote que está sendo lido.

Fica claro que o Orion não gerencia um buffer, apenas possui os modelos para efetuar a contabilização do consumo, sendo necessário que outro simulador controle quando, em qual posição e qual pacote será escrito.

Infelizmente, o Orion não oferece suporte para o cálculo do consumo de energia estática do enlaces, por isso somente o cálculo do consumo dinâmico dos enlaces está implementado no Serpens.

5.1.3 Modelo das tarefas

Cada aplicação, a ser alocada e escalonada no Serpens, é representada por um grafo de tarefas, $G = (K, A)$, onde cada nodo $k_i \in K$ representa uma tarefa periódica e cada aresta $a_{i,j} \in A$ representa uma dependência ou fluxo de mensagens entre as tarefas k_i e k_j. O peso da aresta, denotado por $a_{i,j}^{W}$, é a quantidade de bytes a ser transferida entre as respectivas tarefas durante a efetivação da comunicação.

Cada tarefa é uma tupla {C,T,D,α,S}, onde C é o número de ciclos de execução em pior caso, T é o período de repetição, D é o *deadline*, α é o número de chaveamentos por ciclo da tarefa e S é o tamanho das tarefas em bytes.

É comum se utilizar grafos de tarefas gerados sinteticamente para se efetuar escalonamento. Nesse caso, as tarefas geralmente apresentam os parâmetros C, T e D. O parâmetro α foi acrescentado, para que, além do escalonamento das tarefas, seja possível calcular-se também o consumo de energia. Este parâmetro também pode ser calculado aleatoriamente da mesma forma que os demais, caso não existam dados obtidos a partir de tarefas reais. O parâmetro S também foi acrescentado para que as tarefas tenham um custo ao serem migradas. Ele é um parâmetro importante neste trabalho, pois identifica custos de migração para cada tarefa e ocupação de memória. Tarefas também podem ser classificadas em dois tipos: tarefas de execução e tarefas de E/S. As últimas recebem e enviam dados através das portas de E/S da rede-em-chip. As portas de E/S são canais dos roteadores que estão ligados ao meio externo do SoC. A Figura 5.5 apresenta a localização das portas de E/S no SoC.

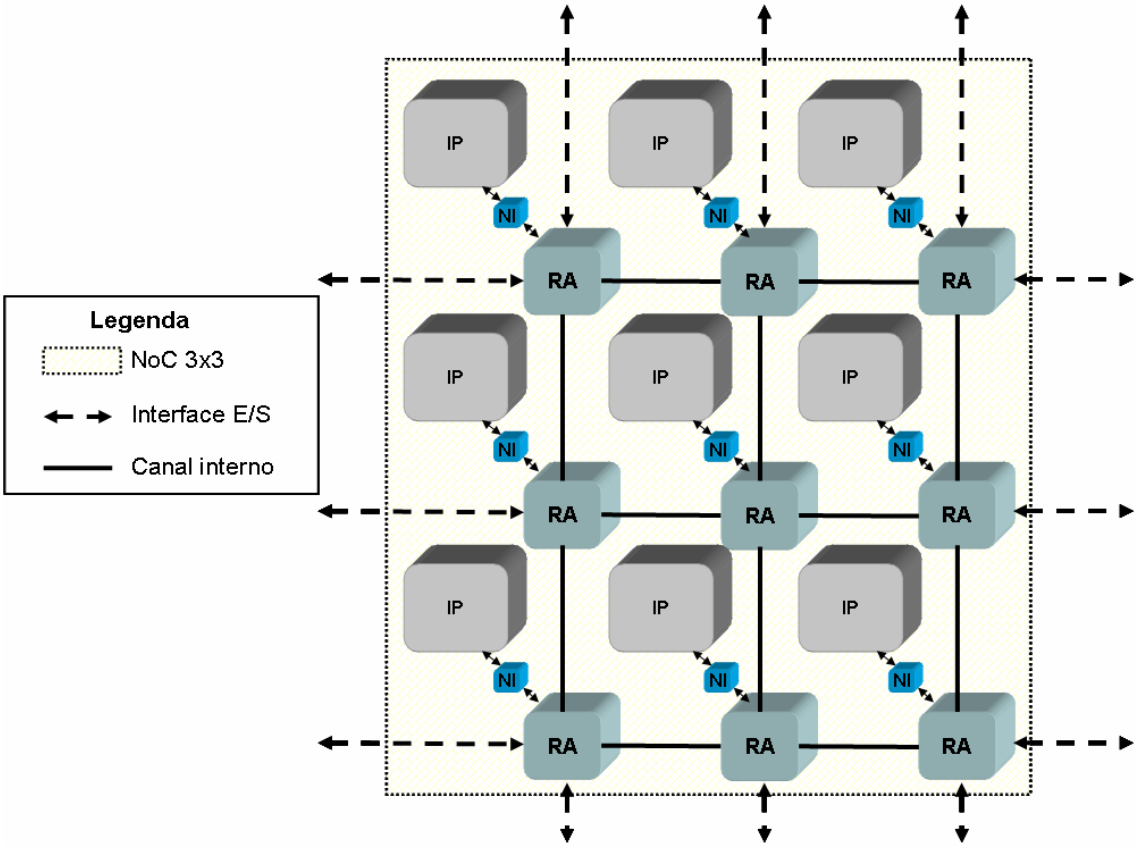


Figura 5.5: Interfaces de E/S da rede-em-chip SoCIN.

Cada tarefa é caracterizada com o tempo de execução WCET e chaveamento de circuitos por ciclo. Com estes dados, as tarefas são escalonadas e o consumo de energia é avaliado em tempo de execução. O tempo WCET raramente ocorre na execução das tarefas. Então foi criado um parâmetro chamado *slack* que dimensiona a diferença entre o pior e o melhor caso de execução. Por *default*, o *slack* está definido como 30%, o que significa que o melhor caso de execução representa 70% do WCET. O simulador escolhe aleatoriamente valores de *slack* que estão definidos de 0% a 30% para maior realismo, pois esta folga é dependente dos dados em sistemas reais. Como o Serpens utiliza grafos TGFF, os quais têm um nível de abstração que não permite a execução das tarefas dependentes de dados, ou seja, não permite a variação do tempo de execução de acordo com os dados, então este procedimento de escolha da folga se considera viável e

realista. Os custos do escalonador e do envio de mensagens também são considerados, incluindo o custo da execução de processadores inativos, o que significa que existem processadores que não estão executando nenhuma tarefa. Estes custos podem ser vistos na Tabela 5.1. Estes custos foram definidos através de observações de benchmarks executados em um simulador ciclo-a-ciclo denominado CACO-PS (BECK, 2003b). O custo de 100 ciclos é um custo fixo, que ocorre sempre que uma das tarefas do sistema é invocada (WRONSKI, 2007).

Tabela 5.1: Caracterização das tarefas do sistema.

Tarefa	Ciclos	α
Liberação	100	153017
Escalonador	100	199802
Enviar pacote	40 ciclos X *phit*	188617
Receber pacote	40 ciclos X *phit*	188617
Modo ocioso	-	29388

Adicionalmente, são definidos custos relativos à manipulação das filas do escalonador:

- Varrer a lista de tarefas prontas: 100 ciclos por tarefa;
- Inserir na lista de tarefas prontas: 100 ciclos por iteração de busca na lista, incluindo mais 50 ciclos para inserção efetiva. Esta lista é ordenada por *deadline* absoluto, pois o escalonamento é EDF;
- Remover cabeça da lista de tarefas prontas: 50 ciclos;
- Inserir na lista de tarefas bloqueadas: 50 ciclos;
- Remover da lista de tarefas bloqueadas: 50 ciclos;
- Inserir na lista de liberação: 100 ciclos por iteração de busca + 100 para inserção efetiva. Esta lista é ordenada por próxima liberação;
- Remover cabeça da lista de liberação: 50 ciclos.

5.1.4 Modelo de mensagens

Como foi mencionado na Seção 5.1.2, os roteadores RaSoC realizam o chaveamento por pacotes baseados em *wormhole*. As mensagens que trafegam na rede-em-chip são quebradas em pacotes com carga útil máxima de 256 bytes. Neste trabalho, foram implementados três tipos de mensagens:

- *Mensagem de comunicação inter-tarefas*: são mensagens que são trafegadas pela rede-em-chip para comunicação entre as tarefas do sistema que estejam em diferentes processadores;
- *Mensagem de migração*: são mensagens utilizadas para transportar uma tarefa (incluindo dados e código). Quando todas as mensagens que

transportam uma tarefa são entregues para o processador destinatário, a tarefa será incluída em uma fila de tarefas prontas, ou seja, pronta para ser carregada pelo processador destinatário e pronta para ser executada, apenas esperando ser chamada pelo escalonador;

- *Mensagem de gerenciamento*: são mensagens que atualizam o núcleo mestre (Seção 5.1.8) sobre a ocupação (processamento e memória) do sistema. Quando um processador recebe uma tarefa via migração, esta tarefa é instanciada no mesmo e então sua ocupação é atualizada. Logo a seguir, uma mensagem de gerenciamento é enviada para o núcleo mestre para atualizá-lo sobre essa alteração da ocupação. Para este tipo mensagem, o tamanho máximo da carga útil de cada pacote é de 254 bytes.

A Figura 5.6 apresenta a estrutura de um pacote da mensagem que tem mesmo formato para as mensagens de comunicação inter-tarefas e mensagens de migração. O campo "MIG" tem dois bits, ou seja, nestes dois bits selecionam-se os três tipos das mensagens.

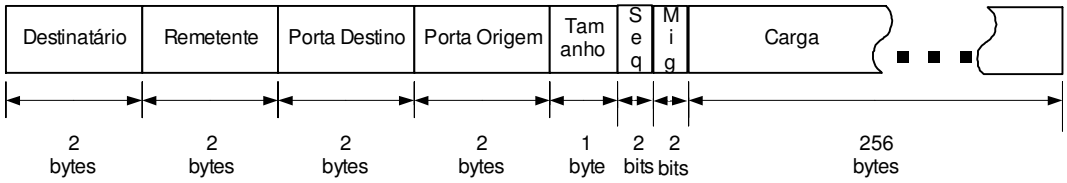


Figura 5.6: Estrutura de um pacote com formato que suporta as mensagens de comunicação inter-tarefas e mensagens de migração.

A Tabela 5.2, apresenta a relação de valores do campo "MIG" para a seleção do tipo da mensagem.

Tabela 5.2: Semântica dos valores do campo "MIG".

Valor de "MIG"	**Tipo da mensagem**
00	Msg. de comunicação inter-tarefas
01	Mensagem de migração
10	Mensagem de gerenciamento
11	Reservado

Na Figura 5.6, o campo "Seqüencial" (SEQ) indica o tipo de pacote:

- 0: Primeiro pacote de uma mensagem;
- 1: Um pacote do meio de uma mensagem;
- 2: Último pacote de uma mensagem;
- 3: Único pacote de uma mensagem.

Alternativamente, a Figura 5.7 apresenta a estrutura do pacote semelhante com a estrutura apresentada na Figura 5.6, com suporte a mensagens de gerenciamento. Percebe-se que neste tipo de mensagem, pacotes são providos de carga útil de 254 bytes ao invés de 256 bytes. Dois bytes foram reservados para armazenamento das informações de ocupação: utilização do processador (*Ocup. Proc.)* e utilização de memória (*Ocup. Mem.*). Estes dados são indispensáveis para que o núcleo mestre possa obter as informações de todos os processadores do sistema e conseqüentemente aplicar os algoritmos de alocação de uma maneira mais realista.

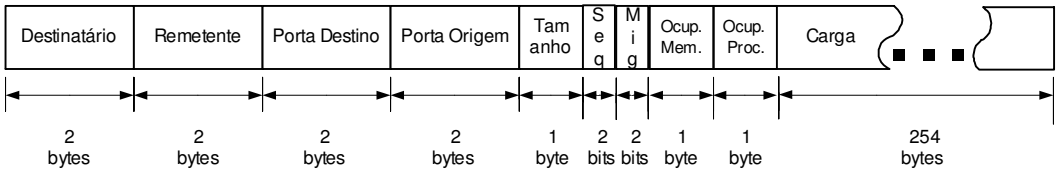


Figura 5.7: Estrutura de um pacote com formato que suporta as mensagens de gerenciamento.

5.1.5 Modelo dos processadores

A seguir, será apresentado o modelo dos processadores em alto nível usado no simulador Serpens proposto por Wronski (2007).

O sistema foi projetado para ser um MPSoC de processadores de mesma ISA, porém com diferentes organizações, onde cada processador possui sua própria memória. O modelo de processador utilizado é em alto nível, ao contrário do modelo de roteador. De fato, o modelo de processador utilizado é análogo a um escalonador de tarefas. Dessa forma, no decorrer da execução o processador assume vários estados em função dos eventos de escalonamento e da comunicação externa.

Essa simplificação torna possível a avaliação rápida do consumo de energia de alocações, sem requerer o desenvolvimento e a execução de aplicações reais (WRONSKI, 2007). Como desvantagem, o modelo se torna menos preciso, uma vez que assume natureza estatística.

O escalonador de tempo real implementado é baseado no modelo proposto por Katcher, Arakawa *et al.* (1993) (Figura 5.8) e utiliza a política de prioridades dinâmicas EDF. Para o escalonamento são aceitas tarefas que obrigatoriamente tenham *deadline* menor ou igual ao período de repetição.

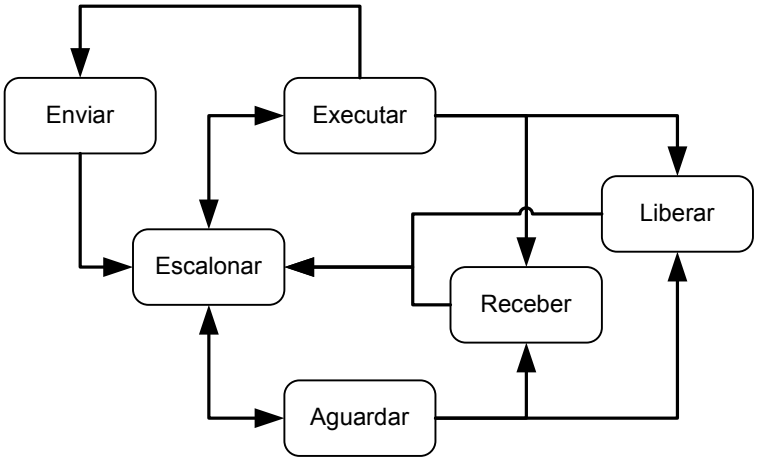


Figura 5.8: Estados do processador.

Quanto à rigidez dos prazos, adota-se uma política tolerante, que permite, assim, a perda de *deadlines*. Contudo, não se pode permitir a existência de mais de uma instância de uma mesma tarefa em execução. Por isso, tarefas que não finalizam durante o seu período não são liberadas novamente até que sejam concluídas.

O ajuste de tensão do processador é feito com o algoritmo DAR (ZHUO, 2005) que é ao mesmo tempo bastante simples e eficiente (WRONSKI, 2007).

Para simular as dependências entre as tarefas, um algoritmo de *list-scheduling* foi utilizado. Como conseqüência, duas listas de tarefas bloqueadas foram acrescentadas ao modelo de Katcher, Arakawa *et al.* (1993). Elas armazenam tarefas que estão aguardando pelo envio ou recebimento de mensagens (WRONSKI, 2007).

Os estados convencionais que o processador pode assumir são *Escalonar*, *Executar* e *Aguardar*. Além destes, foram acrescentados os estados *Enviar* pacotes, *Receber* pacotes e *Liberar* tarefas. Os estados *Enviar* e *Receber* pacote são alcançados através de chamadas do sistema e interrupções do controlador de rede. Por sua vez, o estado *Liberar* tarefas é alcançado em conseqüência de interrupção causada pelo temporizador do escalonador, cuja função é controlar os períodos de liberação das tarefas.

A comunicação entre tarefas implementada é do tipo bloqueante, ou seja, o remetente da mensagem fica aguardando na lista de tarefas bloqueadas, até que o destinatário efetivamente leia a mensagem, da mesma forma que o destinatário o faz quando quer ler uma mensagem ainda não recebida.

Para envio de uma mensagem entregue por uma tarefa através da rede-em-chip, a mesma vai sendo quebrada em pacotes à medida que vai sendo enviada. Esse processo é chamado de empacotamento. Cada pacote gerado é escrito no *buffer* de saída da interface de rede, onde cabe apenas um pacote de tamanho máximo. Em função do número de *phits* usados para construir um pacote, calcula-se o seu custo, pois cada *phit* utilizado consome uma quantidade de ciclos e gera um número de chaveamentos por ciclo na arquitetura, ambos previamente estabelecidos.

Após escrever o pacote na interface de rede, o escalonador volta à operação normal, até receber uma interrupção de pacote enviado, quando é o momento de um novo pacote ser escrito no *buffer*, até que todos tenham sido enviados.

O recebimento de mensagens se dá de forma análoga. Quando há um pacote para ser lido do *buffer* de entrada da interface de rede, uma interrupção é recebida. Cada mensagem completamente recebida é entregue para a respectiva tarefa.

Este modelo de processador, como foi mencionado no segundo parágrafo desta seção, permite avaliações de desempenho e energia em alto nível. Além disso, este modelo permite ser configurado em termos de dados estatísticos relativos à energia, número de chaveamentos, tensão, e número de ciclos. Neste trabalho, será necessária a implementação de processadores em nível RTL para extrair estes dados estatísticos, e então esses dados realimentarão o modelo de processador de alto nível aqui apresentado.

Mais informações sobre o simulador podem ser encontradas em (WRONSKI, 2006).

5.1.6 Modelo de energia

Como dito, cada nodo da NoC é composto por processador, memória local e roteador. O modelo de energia do roteador, tanto dinâmico quanto estático, é provido pelo Orion e já foi discutido. O mesmo modelo é usado para estimativa do consumo das interfaces de rede.

Butts e Sohi (2000) apresentam um modelo para estimação do consumo estático de circuitos CMOS, através da seguinte equação:

Equação 5.1: $Pot_{estat} = V_{dd} \cdot N \cdot k_{design} \cdot I_{leak}$

onde N é o número de portas do circuito, k_{design} é uma constante de projeto fornecida, que diferencia os vários tipos de projeto, e I_{leak} é a corrente de fuga de uma porta. Valores de k_{design} e N fornecidos no estudo são apresentados na Tabela 5.3.

Tabela 5.3: Valores de k_{design} e N.

Circuito	N	k_{design}
Flip-Flop D	22 / bit	1.4
Latch D	10 / bit	2.0
2-input mux	2 / bit / entrada	1.9
6T RAM Cell	6 / bit	1.2
CAM Cell	13 / bit	1.7
Static Logic	2 / porta / entrada	1.1

O consumo dinâmico da memória é assumido como incluído no fator α de cada tarefa. A potência estática, por sua vez, é estimada pela Equação 5.2, com $N = 6/bit$ e $k_{design} = 1.2$, de acordo com a Tabela 5.3, obtendo-se:

Equação 5.2: $P_{estat} = V_{dd} \cdot 6n \cdot 1.2 \cdot I_{leak}$

onde n é o número de bits da memória.

O consumo de energia dinâmica do processador é a soma do consumo de cada ciclo e depende da tarefa executando naquele ciclo. Dessa forma, a energia dinâmica da tarefa i no j-ésimo ciclo de execução, com capacitância de chaveamento C, é apresentada na Equação 5.3.

Equação 5.3: $E^i_{din}(j) = \frac{\alpha_i}{2} C V^2_{dd}$

Lembra-se que o V_{dd} é controlado pelo algoritmo de VS e pode variar de uma instância de execução da tarefa para outra. Dessa forma, a potência dinâmica de uma instância de execução de uma tarefa é obtida pela razão entre a energia dinâmica e o tempo de computação $T_C = \frac{\eta}{F}$ ou $T_C = \eta \cdot T_{ciclo}$, com η sendo o número de ciclos executados e T_{ciclo} o período de ciclo:

Equação 5.4: $Pot_{din} = \frac{\sum_{j=1}^{\eta} E^i_{din}(j)}{T_C}$

A potência estática é dada pela Equação 5.5, acrescida de um multiplicador representando o número total de portas do processador (N_g):

Equação 5.5: $Pot_{estat} = V_{dd} \cdot I_{leak} \cdot N_g$

Finalmente, a correspondente energia estática referente a uma instância de execução de uma tarefa é calculada da seguinte forma:

Equação 5.6: $E_{estat} = V_{dd} \cdot I_{leak} \cdot N_g \cdot T_C$

5.1.7 Organização do processador

Na plataforma desenvolvida de simulação, foi utilizado um modelo de processador: Femtojava *Pipeline*. O microcontrolador FemtoJava (ITO, 2001) é um microcontrolador baseado na arquitetura de pilha com capacidade de execução nativa de bytecodes Java. Suas principais características são: um reduzido conjunto de *bytecodes*, arquitetura Harvard (memórias separadas de dados e instruções), pequeno tamanho e facilidade de inclusão e remoção de instruções. A arquitetura do microcontrolador FemtoJava foi concebida mediante um detalhado estudo do conjunto de instruções Java e da arquitetura da Máquina Virtual Java (JVM – *Java Virtual Machine*). A Figura 5.9 apresenta a detalhes da microarquitetura do FemtoJava.

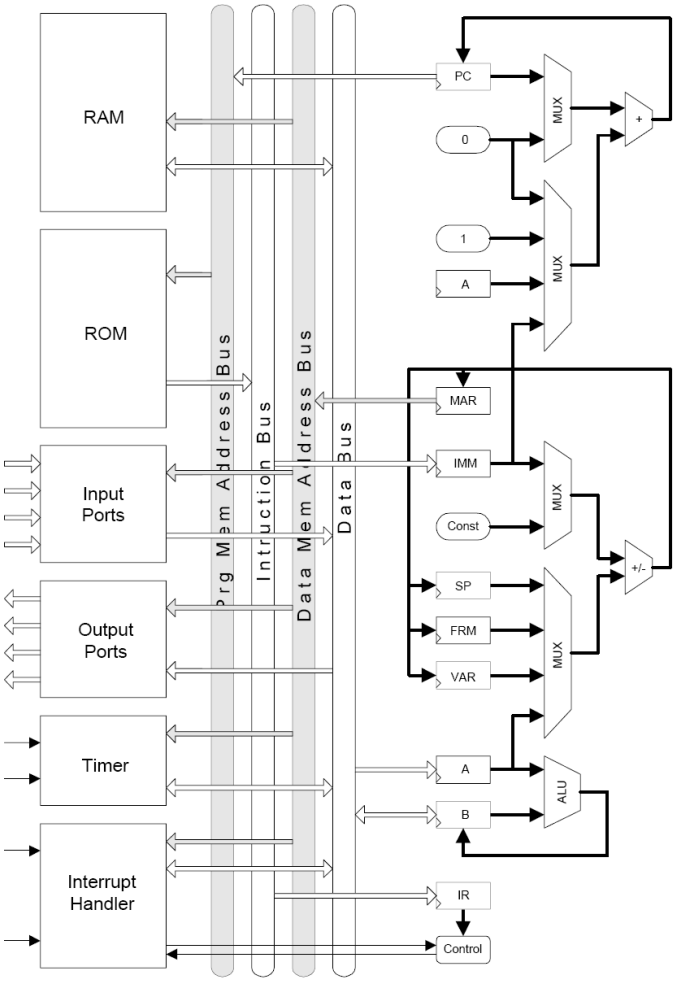


Figura 5.9: Microcontrolador Femtojava.

Além do microcontrolador ser composto por uma unidade de processamento baseada em arquitetura de pilha, possui memórias RAM e ROM integradas, portas de entrada e saída mapeadas em memória e um mecanismo de tratamento de interrupções com dois níveis de prioridade. A arquitetura da unidade de processamento implementa um subconjunto de 66 instruções da Máquina Virtual Java, e seu funcionamento é consistente com a especificação da MVJ.

Como extensão do trabalho de Ito (2001), foi desenvolvido o microcontrolador Femtojava Pipeline (BECK, 2003a). Este microcontrolador tem 5 estágios de pipeline: busca de instruções, decodificação de instruções, busca de operandos, execução e salvamento de registradores, como mostrado na Figura 5.10. A principal característica dessa arquitetura é a presença de registradores para armazenamento de operandos na pilha e variável local (usada para manter valores de variáveis locais de um método). Essa característica evita o acesso da memória principal, diminuindo o tempo de acesso e aumentando o desempenho do microcontrolador.

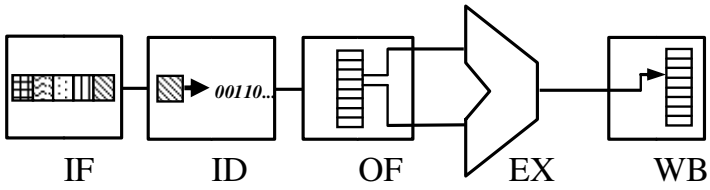


Figura 5.10: Arquitetura do Femtojava Pipelined.

Realizando uma comparação entre o *Femtojava Pipelined* e o *Femtojava Multiciclo*, o *Femtojava Pipelined* consome mais potência, pois sua implementação exige mais área. No entanto, este tipo de processador apresenta um desempenho de execução bem maior que o *Femtojava Multiciclo*. Em conseqüência, a versão *Pipelined* do processador consome menor energia que a versão Multiciclo. Foi escolhido o processador *Femtojava Pipelined* por ser o processador mais usado no grupo do LSE (grupo local).

5.1.8 Gerenciamento e monitoração

Na plataforma Serpens é utilizado um núcleo mestre que realiza a execução das heurísticas de alocação e balanceamento de carga no sistema (escalonamento global). O núcleo mestre é implementado através de um processador *Femtojava*, conectado a um dos roteadores da rede-em-chip, o qual executa um determinado algoritmo de alocação de tarefas. Esse núcleo armazena uma imagem da utilização de todos os núcleos do sistema. Através dessa informação, o núcleo mestre é responsável pela tomada de decisões em tempo de execução para migração de tarefas, toda vez que é carregada uma aplicação no sistema. Essa tomada de decisões depende da heurística escolhida *a priori* para distribuição e alocação das tarefas nos demais núcleos. O núcleo mestre é atualizado em tempo de execução cada vez que houver alteração do status (utilização do processador e tamanho da memória) de um elemento de processamento. Este, quando seu status for modificado, envia uma mensagem para o núcleo mestre para que seja realizada a atualização da imagem do sistema. A estrutura do pacote desta mensagem é ilustrada na Figura 5.7.

As alocações de novas tarefas também são realizadas pelo núcleo mestre. Quando as tarefas são carregadas pelo sistema, o núcleo mestre lê de sua memória local[6] todas as tarefas ali armazenadas, e através de um algoritmo de distribuição, elas são alocadas. A alocação da tarefa então é realizada através do mecanismo de migração, o qual transporta uma tarefa pela rede do núcleo mestre para o processador destino.

A Figura 5.11 e a Figura 5.12 apresentam dois pseudocódigos para o tratamento das mensagens de gerenciamento. A Figura 5.11 apresenta um método chamado *AddTaskToLoad*. Este método é responsável pela carga da tarefa no processador local. Esta tarefa é originada do núcleo mestre, e a migração envia esta tarefa para o núcleo local (destino). A tarefa então é inserida em uma fila para que o escalonador local possa tomar decisões sobre a mesma. Este método também executa outros dois métodos:

[6] Pressupõe-se que dados são armazenados nesta memória. Estes dados podem ser originados por um *download* de algum aplicativo por um usuário, ou então dados lidos de um HD. Os custos da cópia destes dados originados de alguma fonte para a memória local do processador mestre estão fora do escopo deste trabalho.

atualização dos dados de ocupação do núcleo local (*update_local_information*) e envio da mensagem de gerenciamento sobre informações da ocupação do núcleo local para o núcleo mestre (*send_update_local_data*).

```
void Sim_pe::AddTaskToLoad(task p_task){
         occupation_data: tOccupation;
         m_load_task_list.push_back(p_task);// insere task na lista para
                                            // escalonamento
         update_local_information(p_task,&occupation_data);
         // atualiza ocupações
         send_update_local_data(MASTER_X, MASTER_Y, occupation_data.mem,
occupation_data.proc);
         // envia dados de ocupação de mem e proc para o processador
mestre.
         load_application_event_notify(); // notifica evento de carga
}
```

Figura 5.11: Método para carga da tarefa no processador local. Este método também é responsável pelo envio das informações das ocupações do processador e memória para o núcleo mestre.

A Figura 5.12 apresenta um método chamado *receivePack* que lida com migração de tarefas (Seção 5.1.9), recepção de dados originados por tarefas e dados sobre o gerenciamento dos processadores em termos de ocupação de processamento e memória.

```
bool Sim_pe::receivePack()
{…
      p = in.read();
      fields = phitFactory.WritePhit(&p);// remoção dos campos
      if (fields->mig==1){
           ... // tratamento de migração
      if (fields->mig==2)}
            updateMasterOccupationsTable(fields->proc,
                      fields->OccupProc, fields->OccupMem);
            //atualiza dados do processador mestre;
      }else{
            data = receiveData(); // mais in.read aqui dentro
}
```

Figura 5.12: Método *receivePack* no contexto da recepção de mensagens para o gerenciamento da ocupação dos processadores.

A condição “if (fields->mig==2)” indica que se a mensagem for do tipo de gerenciamento dos processadores, então é executado um método chamado *updateMasterOccupationsTable*, o qual atualiza dados de ocupação de processamento e memória em uma estrutura de dados armazenada no núcleo mestre. Os campos do pacote são extraídos pelo método *phitFactory.WritePhit* e armazenados no registro *fields*.

A Figura 5.13 mostra um possível posicionamento do núcleo mestre em uma malha 3x3 a fim de ilustrar a idéia do posicionamento e da gerência da rede.

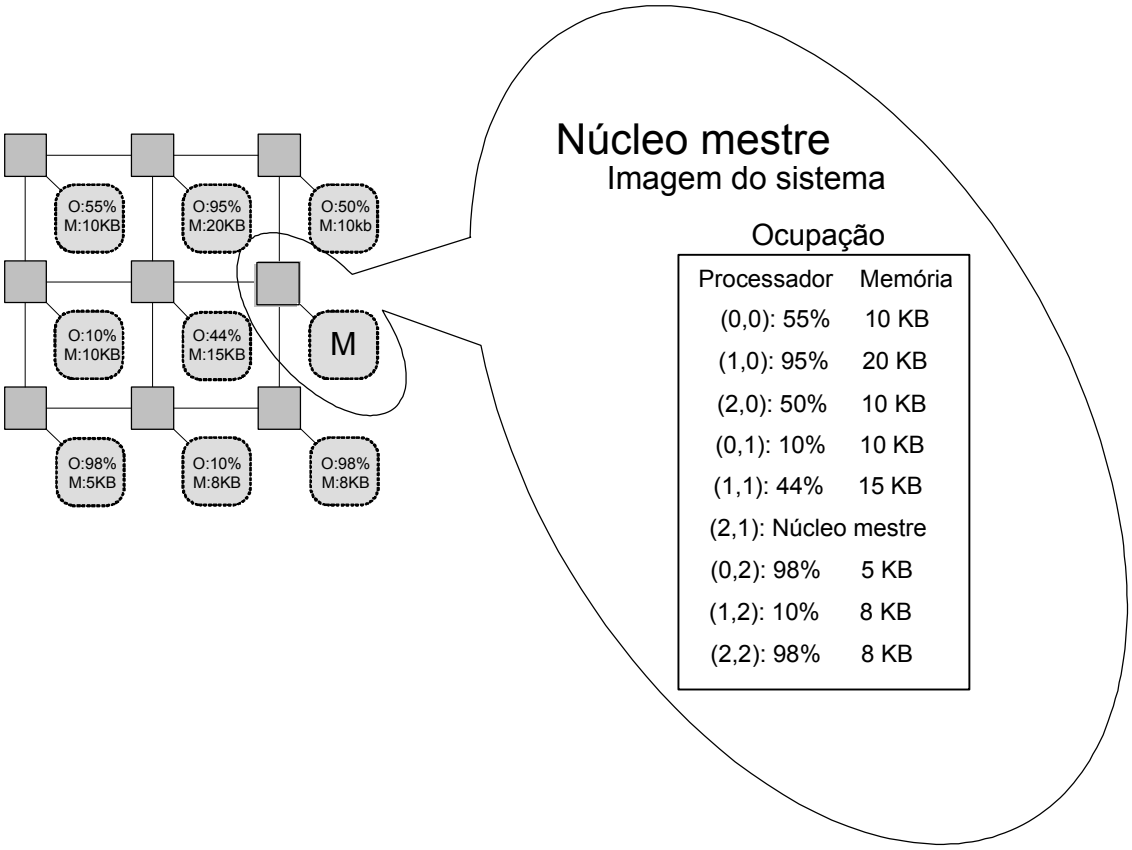


Figura 5.13: Posicionamento e a imagem do sistema em termos de utilização (para simplificação da figura) armazenada no núcleo mestre em uma rede-em-chip 3x3.

Atualmente, as decisões de migração bem como o escalonamento global são executadas por um arquivo de configuração do simulador Serpens. A Figura 5.14 mostra o conteúdo do arquivo de escalonamento de aplicações no Serpens.

```
1. TASKGRAPH telecom-tg0core.t START 0 ms STOP 200 ms CLUSTERING lr PACKING
2. wf UTILIZATION simple INITIAL_MIGRATION naive

3. TASKGRAPH telecom-tg1core.t START 0 ms STOP 200 ms CLUSTERING lr PACKING
4. wf UTILIZATION simple INITIAL_MIGRATION naïve

5.  TASKGRAPH  network-tg0core.t  START  200  ms  STOP  500  ms  CLUSTERING  lr
6.PACKING wf UTILIZATION simple INITIAL_MIGRATION naive
```

Figura 5.14: Conteúdo do arquivo de escalonamento global do Serpens.

Como visto na Figura 5.14, o comando *taskgraph* define qual é o nome do arquivo (aplicação ou um grupo de tarefas da aplicação) a ser carregado. Os comandos *start* e

stop definem o tempo de execução da aplicação. O comando *clustering* define a alocação de tarefas no sistema e é baseado em *linear clustering*. O comando *packing* define qual estratégia de alocação deve ser utilizada, se é concentração de carga (bf : *Best Fit*) ou se é balanceamento de carga (wf: *Worst Fit*). O comando *utilization* define se a utilização é calculada apenas pelo número de ciclos divididos pelo período de uma tarefa (*simple*) ou pela ocupação em relação ao processamento e comunicação. Finalmente o comando *initial_migration* define qual é o tipo de migração utilizado. O tipo de migração *naive* é o método de cópia, ou seja, a migração é realizada através da transferência de todo o contexto de uma tarefa de um núcleo para o outro. Enquanto a migração é realizada, as tarefas já alocadas executam em paralelo.

5.1.9 Mecanismo de migração

O modelo de migração de tarefas no sistema de memórias distribuídas não-compartilhadas utilizado neste trabalho é baseado no modelo de migração por cópia. Este modelo é muito simples com alto custo de migração, já que todo o contexto (código, dados, pilha, e conteúdo de registradores internos) é migrado.

O sistema de simulação Serpens tem suporte para processadores de mesma ISA (*Instruction Set Architecture*) (embora eles possam ter diferentes organizações). Tradução binária entre os processadores não é exigida pelo fato da homogeneidade dos processadores (mesma ISA). No entanto, a plataforma utilizada neste trabalho é totalmente homogênea, ou seja, todos os processadores utilizados têm mesma organização.

Tarefas podem ser migradas de dois tipos de núcleos: núcleo de referência e núcleo mestre. A rede-em-chip sem a presença de núcleos mestres obrigatoriamente necessita de pelo menos um núcleo de referência. O núcleo de referência é um núcleo escravo qualquer, que aloca tarefas e este pode executar tarefas da aplicação. No entanto, os custos dos algoritmos de alocação (*bin-packing*) não são considerados. Por outro lado, o núcleo mestre é um núcleo dedicado a apenas executar o algoritmo de alocação. Todavia, os custos de gerenciamento são levados em conta. Em cada processador, existem mecanismos de comunicação inter-tarefas baseados em mensagens (uso das primitivas *send* e *receive*). Como mencionado na Seção 5.1.4, tais primitivas suportam três tipos de mensagens: mensagens de comunicação inter-tarefas; mensagens de migração de tarefas e mensagens de gerenciamento. Quando um evento de migração ocorre originado do processador mestre, a primitiva *send* escreve no cabeçalho de uma mensagem um identificador de serviço (MIG) (Figura 5.6) o valor “01” (o valor “00” indica uma mensagem de comunicação inter-tarefas e o valor “10” indica uma mensagem de gerenciamento conforme Tabela 5.2), determinando que a mensagem é o tipo de mensagem de migração. A primitiva *receive* é executada através do método *polling* (varredura), tentando ler algum pacote enviado de algum outro processador do canal de entrada da rede. Se existir um pacote no canal de entrada do processador destino, a primitiva lê o cabeçalho. Se o identificador de serviço está ajustado como migração, outro campo é lido: o identificador da tarefa (remetente), como seu próprio nome se refere, identifica a tarefa que será instanciada no processador destino. Atualmente, o código e os dados da tarefa são então lidos do processador que enviou esta tarefa, através de mensagens que chegam ao processador destino. Depois deste passo, a tarefa então é instanciada completamente no processador destino e pronto para ser executada, sendo esta inserida em uma fila de execução, continuando do ponto em que esta parou de executar.

A Figura 5.15 apresenta a migração de tarefas em duas etapas: na primeira etapa, (a), todas as tarefas estão armazenadas em uma fila de tarefas a serem migradas do núcleo origem. Nesta mesma figura, a tarefa T1 é inserida na fila de tarefas a serem migradas. Quando o núcleo de origem (P0) decide migrar uma tarefa, (T1, por exemplo), é realizado um processo de encapsulamento da tarefa em diversos pacotes, dependendo do tamanho do código e dos dados da tarefa a serem migrados. Todo o contexto é transferido através dos canais de comunicação da rede-em-chip, como mostrado na Figura 5.15b. Quando o último pacote é recebido pelo núcleo destino (P3), então a tarefa migrada é inserida na fila de execução do P3, pronta para ser executada pelo escalonador local.

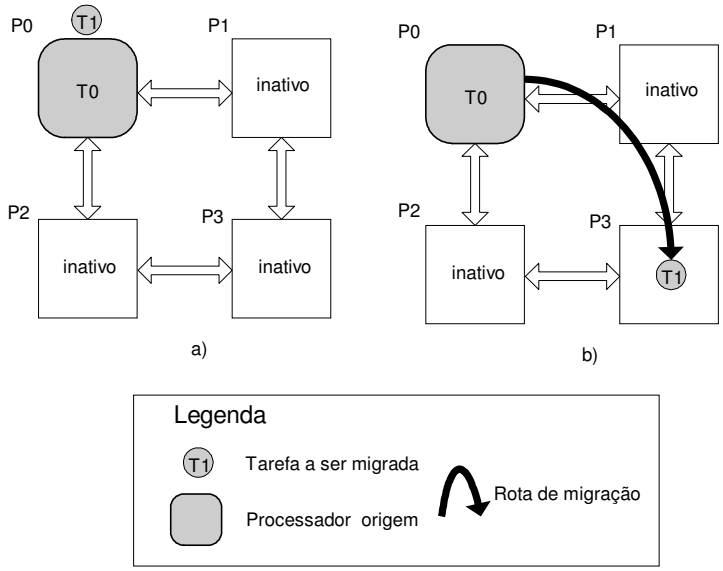


Figura 5.15: A migração de tarefas em dois passos: a) antes da carga da tarefa T1 do núcleo de origem; b) depois da carga da tarefa do núcleo de origem.

Como foi dito anteriormente, na Seção 5.1.1, cada processador tem seu próprio escalonador baseado em EDF. A Figura 5.16 apresenta o escalonamento de tarefas para quatro processadores mostrados na Figura 5.15. O primeiro processador (núcleo origem) P0 está executando a tarefa T0. Uma interrupção acontece subitamente pelo processador mestre e então o núcleo de origem decide migrar uma tarefa (T1). A tarefa T1 é enviada para o processador P3. O rótulo "MS" no escalonamento indica que o P0 (núcleo origem) está enviando T1 para P3. O processador P3 está inativo (*idle*). Ele ficará neste estado até no momento em que receber todos os pacotes enviados por P0. Esta recepção está rotulada por "MR". Quando o último pacote for recebido por P3, o escalonador rotulado como "S" em P3 realiza o monitoramento se existe uma nova tarefa. Como a tarefa T1 é uma nova tarefa a ser executada, ela é inserida na fila de liberação, pois é a primeira vez que ela será executada neste processador P3. Então, o rótulo "R" na figura sinaliza que o escalonador libera a tarefa para execução, e a tarefa T1 é inserida na fila de execução de P3. A tarefa migrada T1 então começa a sua execução em P3.

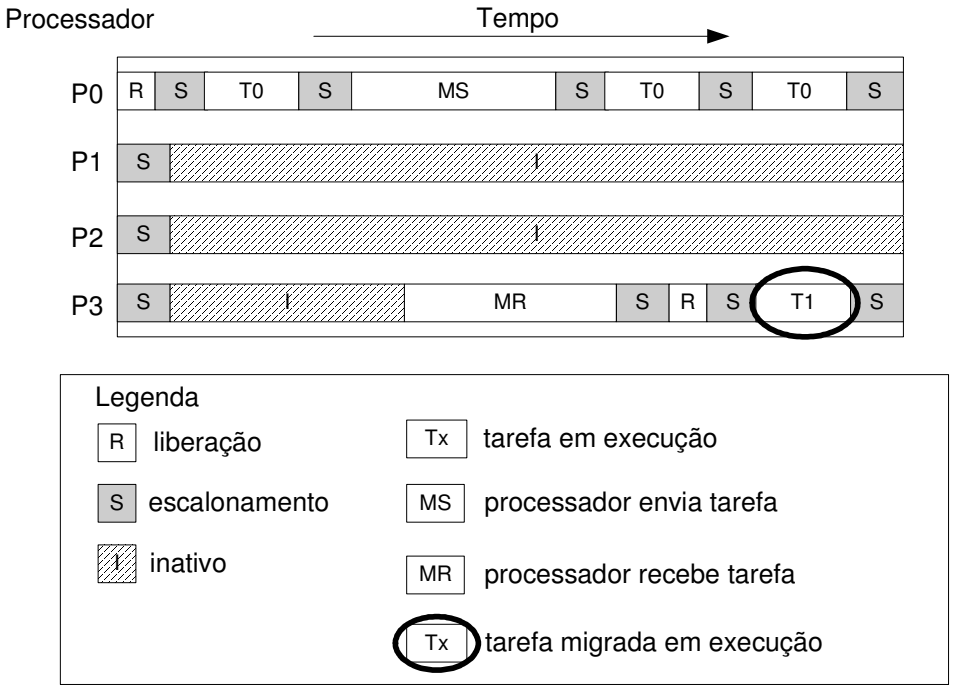


Figura 5.16: Escalonamento de tarefas nos processadores apresentados na Figura 5.15.

A Figura 5.17 apresenta o método de migração de tarefas. Este método sempre verificará se existe uma tarefa na fila de migração originada pela leitura desta tarefa da memória do núcleo mestre. Esta fila de migração armazena tarefas a serem migradas. Se a tarefa que estiver na fila tiver seu campo de destinatário apontado para o processador local, então significa que a tarefa encontrou seu destino e desta forma ela será alocada no processador através do método *AddTaskToLoad*. Como foi mencionado na Seção 5.1.8, este método é responsável também pelo envio de mensagens de gerenciamento para o núcleo mestre. Se o destinatário for outro processador, então a tarefa é encapsulada na estrutura chamada *taskMigrationInfoList* e é enviada pela rede através da API *sendMigrationData*. Esta API fragmenta os dados e o código da tarefa em pacotes e então estes são enviados pela rede utilizando a primitiva *send* que envia pacotes pela rede para um destinatário.

```
void Sim_pe::migrate(){
        while (!m_tasks_migration_list.empty()){
           tMigration
              taskMigrationInfoList = m_tasks_migration_list.front();
           m_tasks_migration_list.pop_front();  // remove first element
           if (taskMigrationInfoList.target == local_position){
                           AddTaskToLoad(taskMigrationInfoList.task);
           }else
                           sendMigrationData(taskMigrationInfoList);
        }
}
```

Figura 5.17: Método implementado no simulador Serpens para migração de uma determinada tarefa.

Por outro lado, a Figura 5.18 apresenta o método *receivePack*, já mencionado na Seção 5.1.8, porém no contexto de gerenciamento e monitoramento dos processadores. A Figura 5.18 apresenta o método *receivePack* no contexto de migração de tarefas. Este método é responsável por receber dados relativos à migração de tarefas, dados de comunicação entre as tarefas e recepção de dados relativos ao gerenciamento dos processadores em termos de ocupação de processadores e memória, como foi mencionado na Seção 5.1.8. Este método sempre faz a análise se o pacote que chegou é do tipo de migração ou não. Se for, então ele extrai os dados do cabeçalho do pacote e monta a tarefa. Esta então é instalada no núcleo local para sua execução. Caso contrário, dados de comunicação são recebidos normalmente e então são contabilizados pela biblioteca Orion ou dados de gerenciamento são recebidos para atualização do núcleo mestre.

```
bool Sim_pe::receivePack()
{…
      p = in.read();
      fields = phitFactory.WritePhit(&p);
      if (fields->mig==1){
           task=receive_Migration_data();// in.read() aqui dentro
           installTaskOntoCore(task);
      }
      if (fields->mig==2){
          ...// Tratamento das mensagens de gerenciamento
      }else{
            data = receiveData(); // mais in.read aqui dentro
}
```

Figura 5.18: Método *receivePack* no contexto de migração de tarefas.

5.1.10 Sistema operacional

Neste trabalho, o sistema operacional utilizado no núcleo mestre provê serviços de alocação *on-line* e gerenciamento da imagem de ocupação dos demais núcleos. O núcleo mestre é dedicado para execução de serviços de alocação de tarefas e das tomadas de decisões de migração. A implementação desses serviços foi realizada em cima do simulador Serpens através da implementação de API´s e uma descrição RTL do processador para extração do número de ciclos de execução e ocupação em memória de algoritmos para alocação de tarefas.

Por outro lado, nos demais núcleos, os serviços providos pelo sistema operacional são baseados em um escalonador EDF e a execução de uma API de comunicação para troca de mensagens e migração, API de criação e remoção da tarefa. Os números de ciclos da execução do escalonador poderiam ser estimados baseados em dados de simulação da descrição RTL do processador executando pequenos trechos de código e uma implementação da API RTSJ – *Real-Time Specification for Java* com escalonamento EDF de Wehrmeister, Becker *et al.* (2004). Contudo, por falta de tempo hábil, esses valores acabaram sendo definidos baseado nas observações de benchmarks realizadas no trabalho apresentado por Wronski (2007).

5.1.11 Implementação e obtenção dos custos das heurísticas

Dentro do escopo deste trabalho, foi implementado o algoritmo *bin-packing* BF para obtenção dos seus custos através de sua execução no processador *Femtojava*. Como foi

mencionado na Seção 5.1.8, o processador Femtojava implementa o núcleo mestre, pois este é responsável pela execução das heurísticas de alocação. Segundo Johnson et al. (1973), as complexidades dos algoritmos WF e BF são as mesmas ($O(n \log n)$). Por essa razão, somente o algoritmo BF foi implementado, pois os custos dos dois algoritmos são muito semelhantes.

A Figura 5.19 apresenta o algoritmo BF implementado em linguagem JAVA compatível com a ferramenta de síntese SASHIMI (ITO, 2001). A arquitetura alvo desta ferramenta é o processador *Femtojava*. O processador foi configurado à freqüência de 600 MHz, com tecnologia de 180 nm. Cada passo do algoritmo[7] custou 1119 ciclos de relógio e 10,1913 nanoJoules.

```
public static void initSystem(){
    Init();
    for(j = 0; j < m_TaskListCount; j++){
        Step(j);
    }
}// initSystem
public static void Step(int j)
{
      period = m_TimesTaskList[j].getPeriod().convertToFJTimeoutValue();
      wcet = m_TimesTaskList[j].getCost().convertToFJTimeoutValue();
      utilization = (100 * wcet) / period;
      highest_load = -1;  is_feasible = -1;
      for(i = 0; i < NUMBER_OF_PES; i++) {
        if((m_ProcessorsLoad[i] > highest_load) && ((m_ProcessorsLoad[i]
+ utilization) < 100)) {
             highest_load = m_ProcessorsLoad[i];
             is_feasible = i; break;
         }
       }// for
      if(is_feasible != -1){
      m_ProcessorsLoad[is_feasible]  =  m_ProcessorsLoad[is_feasible]  +
utilization;
      m_ProcessorsTaskList[is_feasible][m_TaskCount[is_feasible]++] = j;
      } else {erro = 1;}
}// Step
```

Figura 5.19: Implementação do algoritmo *bin-packing* BF em linguagem Java compatível com a ferramenta de síntese SASHIMI.

[7] O passo do algoritmo é relativo ao número de tarefas. Os processadores são "criados" em tempo de execução quando a capacidade do processador atual for sobrecarregada. Desta forma o número de processadores não influi no passo do algoritmo.

A Figura 5.20 apresenta os custos em termos de tempo de execução e consumo de energia, de acordo com o número de tarefas a serem alocadas. Este algoritmo, mesmo para 100 tarefas, é executado em 373 us. Este é um tempo pequeno e viável para o algoritmo ser utilizado em tempo de execução.

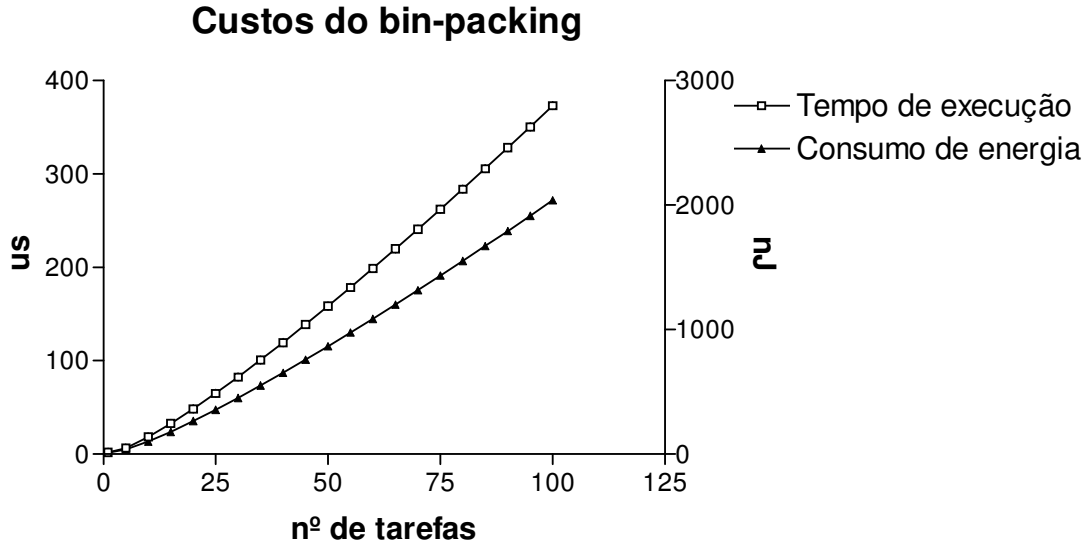


Figura 5.20: Custos do algoritmos *bin-packing* executados na arquitetura *Femtojava Pipelined*.

Para uma simulação mais realista, foi criado um conjunto de tarefas que são executadas no núcleo mestre, as quais têm o mesmo comportamento que o algoritmo BF, dependendo do número de tarefas de cada aplicação. Dependendo da aplicação, essas tarefas são carregadas sob demanda no núcleo mestre, para simular o tempo de execução da heurística de acordo com o número de tarefas da aplicação a ser alocada na rede-em-chip. A Tabela 5.4 apresenta os custos em termos de tempo de execução e consumo de energia para cada tipo de tarefa. Maiores detalhes das aplicações utilizadas no âmbito deste trabalho são apresentados na Tabela 6.1.

Tabela 5.4: Custo da tarefa (heurística) executada no núcleo mestre de acordo com o número de tarefas da aplicação a ser alocada.

Heurística	**Nº de tarefas a serem alocadas**	**Caracterização do tempo de execução (us)[8]**	**Caracterização do consumo de energia (nj)**	**Aplicação**
BP1	11	21,36	116,74	*App-synth*
BP2	30	82,64	451,61	*Telecom* e *Telecom 2*
BP3	64	215,58	1178,03	*Synthetic*

[8] Tempo calculado na freqüência de 600 MHz.

Exemplificando, quando a aplicação *App-synth* for alocada no sistema, o núcleo mestre deverá executar a tarefa *BP1* para simular o algoritmo *bin-packing* em execução para alocação de 11 tarefas. Caso contrário, se a aplicação *Telecom* for alocada no sistema, o núcleo mestre deverá executar a heurística representada por *BP2* para simular o algoritmo *bin-packing* em execução para alocação de 30 tarefas.

Finalizando esta Seção, o algoritmo *linear clusterization* não foi implementado e seus custos não foram avaliados. No entanto, de acordo com as características do pseudocódigo apresentado na Figura 4.7, pode-se inferir que o custo de cada passo do algoritmo é semelhante ao do algoritmo BP. De acordo com o trabalho publicado em (GERASOULIS, 1993), a complexidade do algoritmo *linear clusterization* é semelhante à complexidade do algoritmo BP, ou seja, $O(n \log n)$ salvo pela semântica do valor de n. No algoritmo BP, a quantidade de passos n é diretamente relacionada ao número de tarefas. Por outro lado, a quantidade de passos n, no algoritmo *linear clusterization*, é diretamente relacionada com o número de arestas. Sabendo que o número de arestas tem em média a mesma ordem de grandeza do número de tarefas, pode-se deduzir então que o emprego do algoritmo *linear clusterization* em tempo de execução pode ser viável. Contudo, para ratificação dessa inferência, maiores investigações são necessárias para conclusões mais plausíveis.

5.2 Fluxo de projeto

Esta Seção apresenta o fluxo de projeto utilizado para este trabalho.

A metodologia utilizada consiste em simular as aplicações com heurísticas de distribuição de tarefas (*bin-packing*) conjuntamente com o monitor do sistema (núcleo mestre) ou com a ausência deste, para posterior comparação entre as heurísticas, tamanho da rede-em-chip, variação do tamanho do contexto, variação do tamanho de memória, em função do consumo de energia, taxa de deadlines perdidos e desempenho. Diversos experimentos são apresentados no Capítulo 6.

As heurísticas mencionadas no parágrafo anterior são selecionadas em tempo de projeto e realizam alocações das aplicações em tempo de execução. Entretanto, para a execução de um conjunto de aplicações multitarefa como *benchmarks*, seria necessário que essas aplicações fossem compatíveis com a API (*Application Program Interface*) do sistema operacional de tempo real distribuído e embarcado da arquitetura utilizada. Uma vez que não foram encontradas técnicas de alocação dinâmica em NoCs com restrições temporais e minimização do consumo de energia, esse trabalho se concentra em avaliar técnicas de alocação dinâmica convencionais, como *bin-packing* e *clustering*, considerando custos de migração e gerenciamento dos processadores.

Infelizmente, não se tem até o momento a arquitetura proposta implementada, muito menos um sistema operacional com essas características. Porém, está sendo desenvolvida no grupo de pesquisa local (BARCELOS, 2008) uma plataforma embarcada distribuída baseada em redes-em-chip, constituída por componentes em nível RTL para avaliação e comparação com outras plataformas de nível de abstração maior. Neste caso, a plataforma embarcada em desenvolvimento realimentaria dados de simulação para o simulador Serpens, aumentando assim sua precisão e aproximando o comportamento do simulador Serpens para a plataforma mais realista. No entanto, por falta de tempo, esta atividade está fora do escopo deste trabalho.

Para contornar estas dificuldades, um modelo de estimativa do consumo de energia, baseado em grafos de tarefas, foi utilizado. Neste modelo, cada grafo de tarefas representa uma aplicação independente a ser executada no sistema. Obviamente, este modelo é mais rápido para simulá-lo, porém sua precisão é discutível, comparado com um sistema real.

Além dos grafos de tarefas, um grafo que representa a arquitetura de comunicação e seus parâmetros de configuração também é necessário. Neste trabalho, o escalonador global é fixado estaticamente através de um arquivo de configuração, no qual ajusta quais grafos de tarefas serão alocados, tempo de execução de cada um, e algoritmo de alocação utilizado.

A Figura 5.21 apresenta o diagrama de fluxo de dados que ilustra a metodologia adotada, com base nessa abordagem.

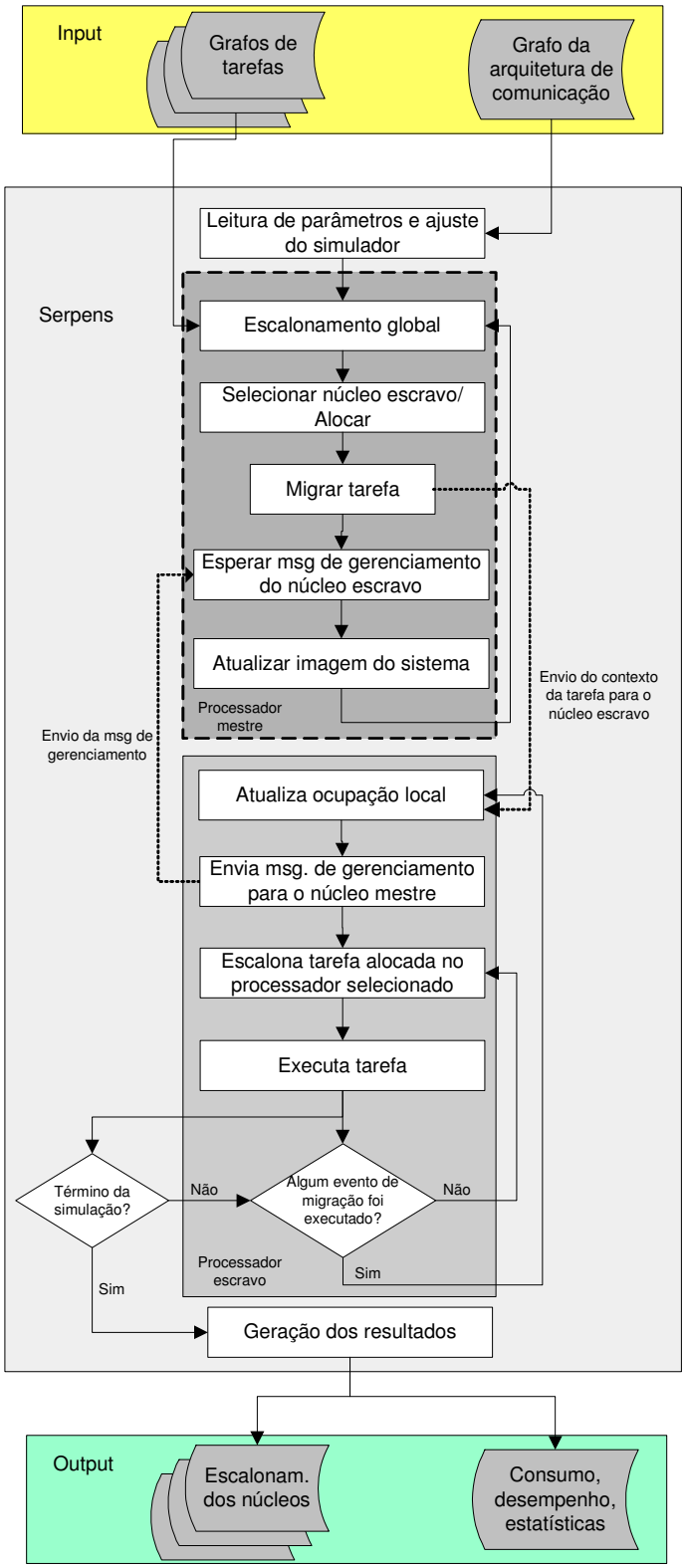


Figura 5.21: Fluxo de projeto para execução e experimentação das aplicações no simulador Serpens.

Inicialmente o grafo de arquitetura de comunicação é carregado pelo simulador para configuração e ajuste da rede-em-chip. As configurações do tamanho da rede-em-chip, freqüência dos processadores, custos de escalonamento, custos de envio e recepção de pacotes, tensão e posicionamento do processador mestre são definidos através do grafo

de arquitetura de comunicação. Logo após, o arquivo de escalonamento global é lido, e então o simulador carrega o arquivo sob demanda, de acordo com a entrada para alocação e execução de cada grafo de tarefa. Para cada tarefa lida, esta é alocada em um núcleo escravo selecionado pela heurística de alocação com base nas informações da imagem do sistema armazenadas no núcleo mestre. A seguir, a tarefa então é empacotada e enviada para o núcleo escravo selecionado pelo núcleo mestre. A tarefa então é desempacotada e instanciada no núcleo escravo e inserida em uma fila de tarefas migradas. O núcleo escravo atualiza as informações de ocupação de memória e processamento logo após a instanciação da tarefa alocada. Então o núcleo escravo envia uma mensagem de gerenciamento para o núcleo mestre, com informações para atualização da imagem do sistema. O núcleo mestre espera por essa mensagem e então repete o processo de alocação das tarefas da aplicação. Cada processador tem seu escalonamento local e as tarefas são executadas de maneira preemptiva. O escalonador irá interromper sua execução apenas quando algum evento de migração ocorrer. Caso isto aconteça, o processador escravo receberá a tarefa através das interfaces do roteador e então deverá atualizar a ocupação local do sistema, com a chegada da nova tarefa. Na abordagem deste trabalho, as tarefas são alocadas sempre quando há sempre uma nova carga a ser carregada no sistema. As tarefas são carregadas e alocadas em tempo de execução. Para tornar esse sistema mais realista, como foi mencionado na Seção 5.1.8, um núcleo mestre de gerenciamento da carga das tarefas foi desenvolvido no sistema.

5.3 Crítica do processo

Apesar desta metodologia apresentar resultados que comprovem a viabilidade da migração de tarefas em sistemas embarcados distribuídos e alocação de tarefas dinamicamente, ainda não existe uma plataforma para comparação. Uma plataforma projetada em RTL está sendo desenvolvida para realimentação e calibração do Serpens (BARCELOS, 2008). A simulação de experimentos fica limitada em um escopo mais simplificado que um sistema real, pelo fato do tempo de simulação poder ser proibitivo.

Outra limitação deste trabalho é que a plataforma Serpens tem sua heterogeneidade limitada. A heterogeneidade dos processadores é limitada apenas às estatísticas relativas aos tempos e consumo de energia escalonamento, liberação e comunicação da tarefa, bem como a configuração de tensão e freqüência. Todos os processadores são homogêneos em termos de arquitetura e organização.

Finalmente, a execução das heurísticas de alocação de tarefas é realizada pelo simulador Serpens como componente do mesmo e não como parte do software executado pelo núcleo mestre. Como o modelo dos processadores utilizados não permite execução de algoritmos em nível de instruções ou ciclo a ciclo, reduzindo a precisão dos resultados, definiu-se então um tipo de modelo de tarefa executada no núcleo mestre que contem aproximadamente o comportamento da execução das heurísticas aqui envolvidas. Desta forma, foi possível obter resultados mais realistas, embora este ainda não seja uma solução ideal.

6 EXPERIMENTOS REALIZADOS E RESULTADOS OBTIDOS

Este Capítulo apresenta detalhes dos modelos dos estudos de caso utilizados nos experimentos, detalhes da metodologia de experimentação e explicação detalhada dos resultados obtidos. A experimentação baseia-se na alteração de diversos parâmetros tais como: variação do tamanho do contexto das tarefas de cada estudo de caso, execução comparativa de vários algoritmos de alocação, variação do tamanho da memória de cada processador do sistema, variação da freqüência dos processadores, variação de carga em tempo de execução, com dois tipos de tamanhos de contexto, e finalmente, resultados relacionados ao mecanismo de gerenciamento e alocação, considerando núcleos mestres.

Finalmente, este Capítulo é concluído através de uma análise sobre os resultados obtidos ao longo deste trabalho.

6.1 Estudos de caso e benchmarks

Neste trabalho, foram desenvolvidos quatro estudos de caso. Dois estudos de caso (*telecom* e *telecom2*) são baseados em uma aplicação do domínio das telecomunicações obtida do *benchmark* E3S - *Embedded System Synthesis Benchmark Suite* (DICK, 2004) e os outros dois estudos de caso (*app-synth* e *synthetic*) são baseados em grafos sintéticos de tarefas. Existem quatro cenários de aplicação: a aplicação baseada em telecomunicações *telecom* tem um pequeno número de tarefas, porém uma média quantidade de comunicação entre elas; a aplicação *telecom2* tem a mesma característica da aplicação *telecom*, porém com alta quantidade de comunicação entre as tarefas. De forma contrária, a aplicação sintética *synthetic* tem grande quantidade de tarefas e pequena quantidade de comunicação entre as tarefas. E finalmente a aplicação sintética *app-synth* tem pequeno número de tarefas e grande quantidade de comunicação. A Tabela 6.1 apresenta estatísticas relacionadas com os grafos de tarefas das quatro aplicações em questão.

Tabela 6.1: Estatísticas dos grafos de tarefas utilizados nos experimentos.

Grafo	WCET[9]	Período do grafo	#Tarefas	#Arestas	#Tam. contexto. médio p/ tarefa	#Comunicação em bytes
Total App-synth	**38 ms**	**108 ms**	**11**	**16**	**1,1 KB**	**47 MBytes**
Telecom_tg0	0,5 ms	3 ms	4	4	1,3 KB	20000
Telecom_tg1	1 ms	5 ms	6	6	1,3 KB	26000
Telecom_tg2	1 ms	5 ms	6	6	1,3 KB	26000
Telecom_tg3	0,3 ms	3 ms	3	2	1,3 KB	6000
Telecom_tg4	0,5 ms	3 ms	3	2	1,3 KB	6000
Telecom_tg5	0,3 ms	2 ms	2	1	1,3 KB	1000
Telecom_tg6	0,4 ms	0,5 ms	2	1	1,3 KB	1000
Telecom_tg7	0,4 ms	0,5 ms	2	1	1,3 KB	1000
Telecom_tg8	0,1ms	0,3 ms	2	1	1,3 KB	1000
Total Telecom	**Não se aplica**	**Não se aplica**	**30**	**24**	**1,3 KB**	**85 KBytes**
Total Telecom2	**Não se aplica**	**Não se aplica**	**30**	**24**	**1,3 KB**	**850 KBytes**
Synthetic_tg0	4 ms	40 ms	16	13	1,1 KB	13
Synthetic_tg1	4 ms	40 ms	16	13	1,1 KB	13
Synthetic_tg2	4 ms	40 ms	16	13	1,1 KB	13
Synthetic_tg3	4 ms	40 ms	16	13	1,1 KB	13
Total Sintético	**Não se aplica**	**Não se aplica**	**64**	**52**	**1,1 KB**	**52 Bytes**

As aplicações foram baseadas em grafos TGFF e as arestas (comunicação) dos grafos indicam dependência entre os nodos (tarefas). A Figura 6.1 apresenta o modelo da aplicação Telecom baseadas em um conjunto de grafos de tarefas. A aplicação Telecom 2 é baseada na aplicação Telecom, salvo pelo alto volume de comunicação entre as tarefas.

[9] Tempos de WCET e período do grafo obtidos a uma freqüência de 133 MHz.

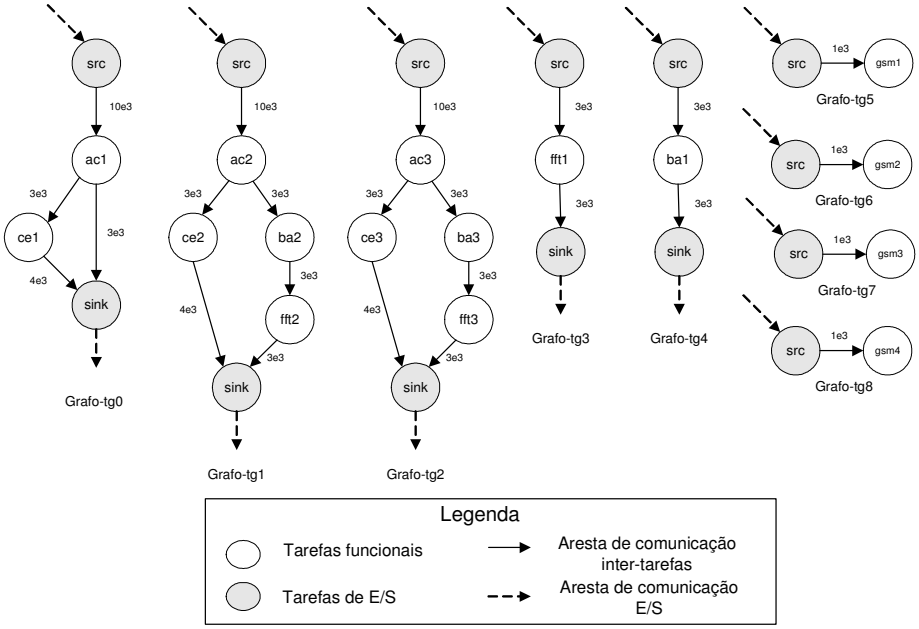


Figura 6.1: Aplicação E3S de Telecom e Telecom 2 baseada em aplicações no domínio da área da telecomunicação (volume de comunicação em bytes).

A aplicação Telecom (Figura 6.1) é modelada por 30 tarefas com 24 arestas ao todo. A aplicação é dividida em 9 grafos distintos com 5 classes de tarefas:

- Autocorrelação (ac*n*): classe de tarefa responsável por executar funções matemáticas para processamento de sinais (seno, decodificação de voz).
- Alocação de bits (ba*n*): modula e transmite dados para linhas ADSL.
- *Enconder* de Convolução (ce*n*): suporta um tipo de código de correção de erros, e é baseado em um algoritmo frequentemente usado na literatura para melhorar o desempenho de sistemas embarcados que utilizem sinais de rádio como meio físico.
- Transformada Rápida de Fourier (FFT) (fft*n*): resolve equações diferenciais parciais a algoritmos para multiplicação de grandes inteiros.
- Decodificação de Viterbi (gsm*n*): Decodificador usado amplamente em sistemas de tempo real e sistemas para transmissão de áudio e vídeo.

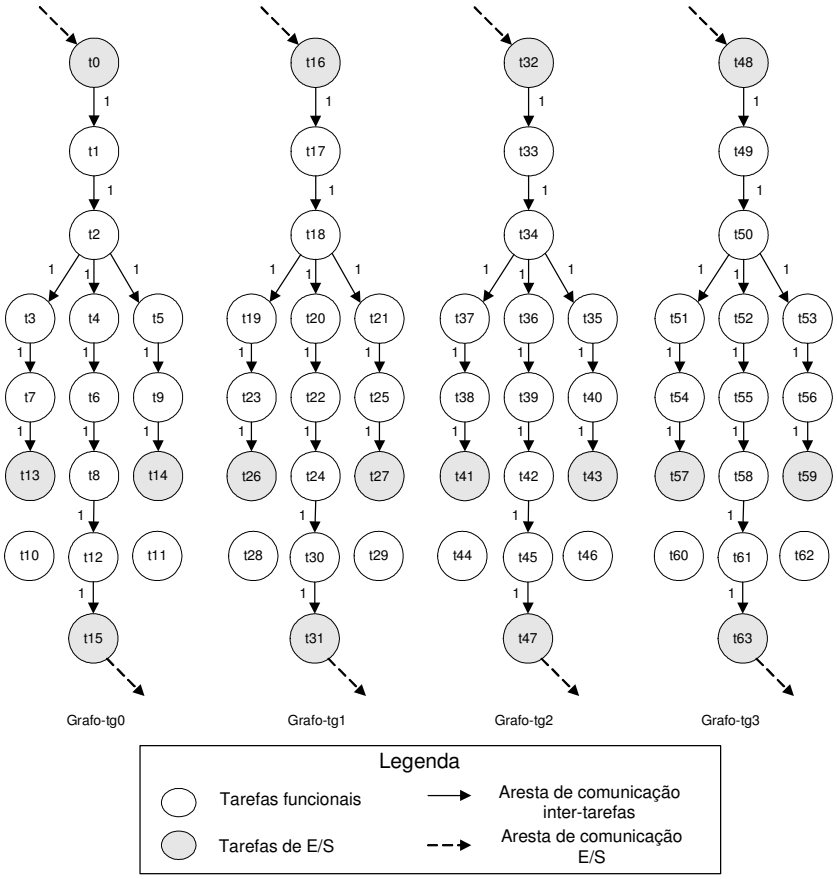


Figura 6.2: Aplicação sintética *Synthetic*. Formada por 64 tarefas ao todo, divididas em 16 tarefas em 4 grafos.

A Figura 6.2 apresenta a aplicação *Synthetic* formada por 4 grafos com pouca comunicação. A característica principal desta aplicação é obter uma maior ocupação dos processadores comparado com as outras aplicações. A comunicação interna é quase desprezível em termos de energia e tempos de comunicação.

A Figura 6.3 apresenta a aplicação *App-synth*, formada por 1 grafo e 12 tarefas com razoável quantidade de comunicação interna. A principal característica desta aplicação é apresentar pequeno número de tarefas com uma comunicação média interna comparada com os outros estudos de caso.

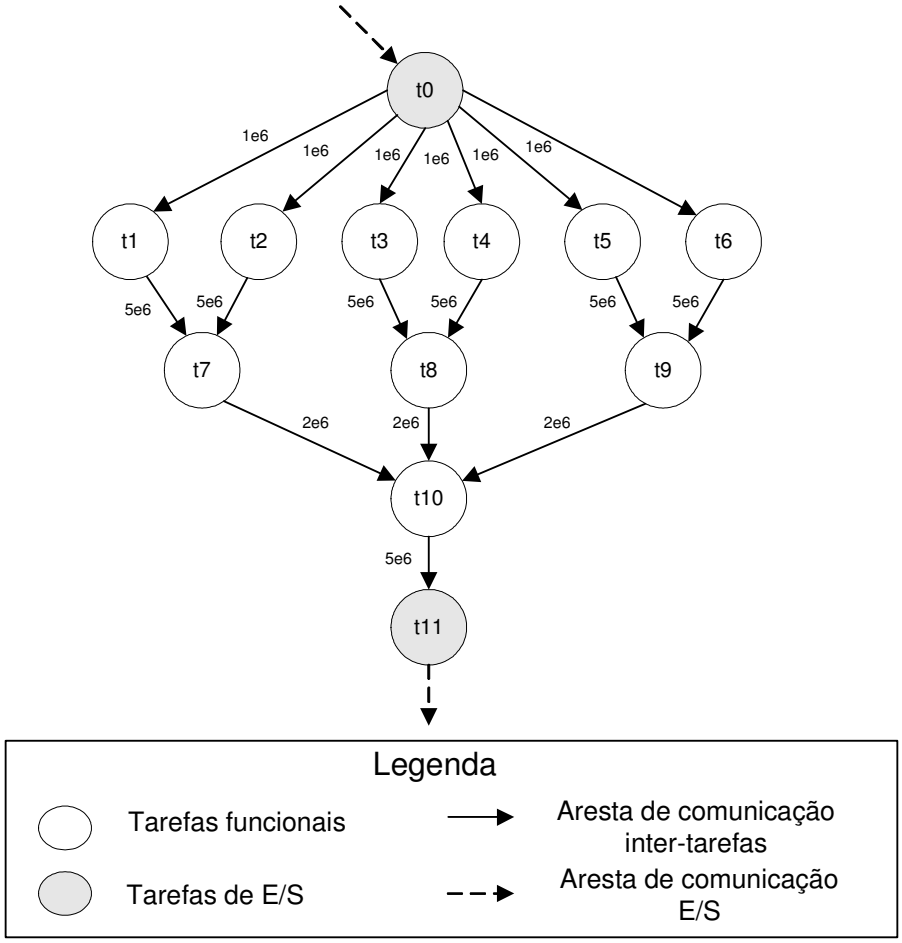


Figura 6.3: Aplicação sintética *App-synth*. Formada por 16 arestas e 12 tarefas ao todo.

6.2 Variação do tamanho do contexto das tarefas

6.2.1 Estratégias de simulação

Neste experimento, os estudos de caso *Synthetic* e *Telecom* foram executados em um tempo de 200 milissegundos. A freqüência dos processadores foi variada de 133 MHz a 266 MHz usando DVS sem PM (não houve desligamento dos processadores que estiveram inativos). A rede-em-chip de topologia tipo grelha utilizada tem o tamanho de 4x4 núcleos rodando a 266 MHz com processadores com uma variação de 1.1 a 1.5 Volts. Para todos os dois estudos de caso mencionados, a tecnologia selecionada para os experimentos é de 100 nanômetros de largura do canal do transistor.

Para este experimento, foi utilizado apenas o algoritmo *bin-packing* de Worst-Fit (WF), baseado na função utilização do processador, combinado com o algoritmo de *linear clustering* para minimização do custo de comunicação entre as tarefas. Este algoritmo, para uma mesma freqüência de operação, obteve o menor número de perdas de deadlines conforme observado nos experimentos da Seção 6.3. Desta forma, utilizou-se este algoritmo nestes experimentos.

Em ambos os estudos de caso, primeiro foram realizados a simulação da aplicação com tarefas alocadas de modo aleatório[10]. Depois da execução desta simulação, realizou-se uma nova simulação, com a execução do algoritmo WF combinado com *linear clustering* para distribuição de carga, usando a mesma alocação original das tarefas. Este trabalho visa comparar um sistema alocado de modo aleatório e um sistema alocado com worst-fit, porém com os custos de migração. O algoritmo WF é responsável por distribuir e alocar, de maneira dinâmica, as tarefas nos núcleos pré-definidos por ele. Esta distribuição é feita através do mecanismo de migração de tarefas. Ambas as simulações (modo aleatório e com WF combinado) foram simulados durante 200 milissegundos cada.

6.2.2 Experimentos

Nos experimentos apresentados abaixo, as tarefas são consideradas periódicas. Elas são escalonadas preemptivamente e, depois de serem concluídas, é contabilizado o número de tarefas finalizadas, mesmo se a execução da tarefa terminar depois do seu período de execução. Quando este evento ocorrer, ocorrerá então uma perda de deadline. Para não haver contabilização de tarefas que não foram concluídas ainda, é calculado então o consumo de energia média por finalização de tarefa. Este cálculo já inclui os custos da migração de tarefas.

A Figura 6.4 apresenta o consumo de energia média por finalização de tarefas como uma função do tamanho do contexto das tarefas (dados e código), variando de 10 bytes a 10 Megabytes para as duas aplicações (*telecom* e *synthetic*) mostradas na Tabela 6.1. Os tamanhos de contexto são números meramente estipulados sem nenhuma relação com as estatísticas apresentadas na Tabela 6.1. A aplicação sintética exige mais energia porque esta aplicação tem um grande número de tarefas para serem executadas. As linhas horizontais pontilhadas ou hachuradas no gráfico representam o consumo de energia sem a migração de tarefas. Nota-se que, para tarefas com tamanho de contexto acima dos 100 Kbytes, a migração de tarefa tem influência relevante na energia consumida pelo sistema. Para tamanho de contextos entre 1M a 10M, houve saturação na rede, ou seja, muito volume de comunicação gerado pela migração de tarefas. Diante do exposto, houve um pico energético 5 vezes maior que o consumo de energia obtido pelos demais tamanhos de contexto.

[10] Nos experimentos realizados neste trabalho, o método aleatório se refere às tarefas alocadas aleatoriamente, sem nenhum critério de alocação.

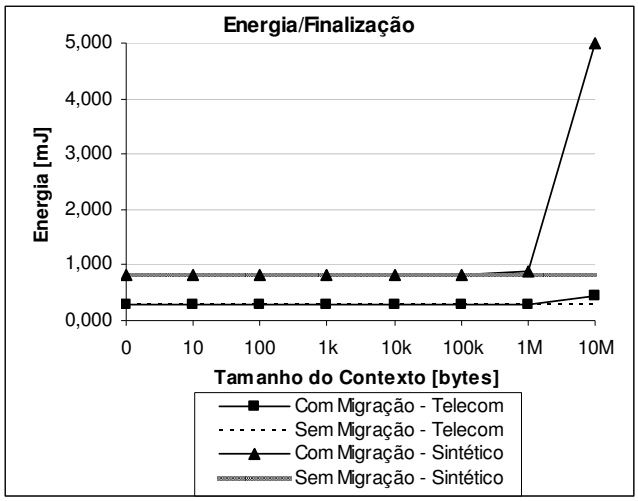


Figura 6.4: Efeito do tamanho do contexto na migração de tarefas em relação ao consumo de energia por finalização de tarefas.

A Figura 6.5 mostra o impacto da migração de tarefas no desempenho geral do sistema e atendimento dos prazos de execução (*deadlines*) das tarefas nas aplicações *Synthetic* e *Telecom*. As barras de cor preta representam a relação entre o número de tarefas que finalizaram quando a migração de tarefas foi aplicada e o número de tarefas que finalizaram sem a migração. Essa percentagem é interessante para visualização dos custos reais da migração em termos de finalização de tarefas. Como pode ser visto nas figuras, a migração de tarefas degrada o sistema em termos de desempenho quando se tem tamanho e contextos maiores do que 100 KB. Essa degradação de desempenho foi ocasionada na simulação num tempo de 200 ms a partir do final da execução da simulação no modo aleatório. Muitas tarefas, de contexto razoavelmente grande, requerem mais tempo de simulação para completar sua execução. Se o tempo de simulação fosse maior, é possível que essa degradação fosse atenuada. A aplicação sintética, como mostrado na Figura 6.5, à esquerda, apresenta grande degradação do sistema causada pela migração, por causa da alta taxa de utilização dos processadores. A Figura 6.5 à direita apresenta degradação do sistema causada pela migração na aplicação Telecom, devido à alta taxa de comunicação entre as tarefas da aplicação.

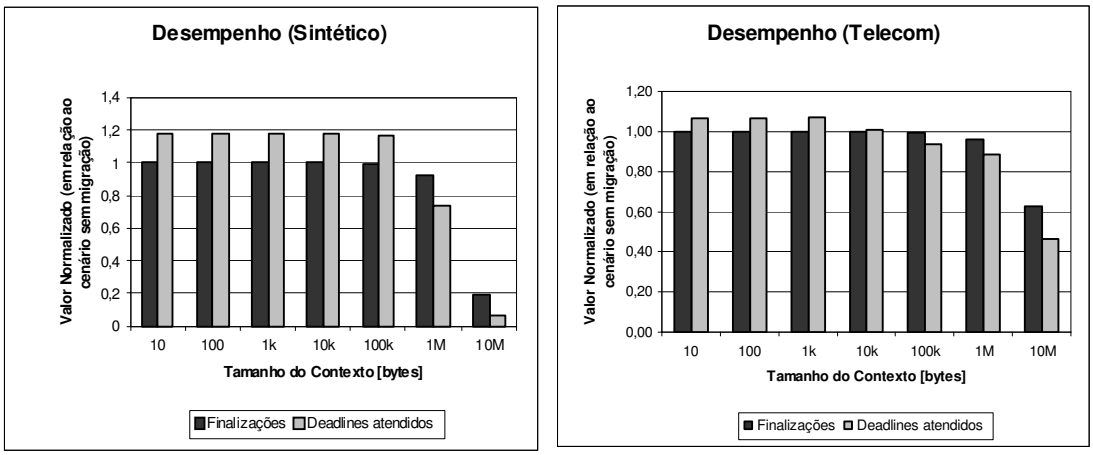


Figura 6.5: Impacto do tamanho do contexto no desempenho do sistema para aplicações Sintético (*Synthetic) e Telecom* com mecanismo de migração.

As barras de cor cinza representam o número relativo de tarefas que executaram suas ações dentro dos seus *deadlines* com migração como percentagem do número de tarefas que executaram suas ações dentro de seus deadlines sem o mecanismo de migração. Essa percentagem é interessante para visualização dos custos reais da migração em termos do número de tarefas que conseguiram finalizar dentro de seus *deadlines*. Como se pode visualizar, para tamanho de contextos menores ou iguais a 100 KB, a migração de tarefas melhora o atendimento dos deadlines, porque uma melhor distribuição de tarefas diminui a utilização dos processadores. O algoritmo de *linear clusterization* combinado com o WF ajuda e muito neste quesito. Vários clusters agrupam tarefas com fortes dependências em termos de comunicação em um mesmo processador, diminuindo o custo de comunicação inter-processadores, aumentando assim o paralelismo e diminuindo a utilização dos processadores ao balancear a carga de maneira homogênea no sistema. Para tamanho de contextos maiores que 10 KB, para a aplicação telecom, ou maiores que 100 KB para a aplicação sintética, esse panorama muda. O número de deadlines perdidos aumenta por causa do mecanismo da migração de tarefas. Isto ocorre devido ao aumento da contenção da rede, criado por este mecanismo.

A Figura 6.6 apresenta o impacto da migração no número de deadlines perdidos, medidos pela percentagem em relação ao número de tarefas finalizadas. Na aplicação de telecom, a migração de tarefas reduz o número de deadlines perdidos comparado com o sistema sem migração de tarefas para tamanho de contextos de até 10 KB. Porém, para tamanho de contexto de 10 MB, o número de deadlines perdidos aumenta de maneira brusca. Para a aplicação sintética, a migração de tarefas quase que elimina todos os deadlines perdidos para tamanho de contexto de até 100 KB. Essa eliminação quase completa dos deadlines perdidos se deve ao fato da aplicação sintética ter pouca comunicação entre tarefas. Quando se diminui o número de tarefas, menos dependências existem entre elas e mais paralelo o sistema tende a se estabelecer, diminuindo drasticamente os deadlines perdidos.

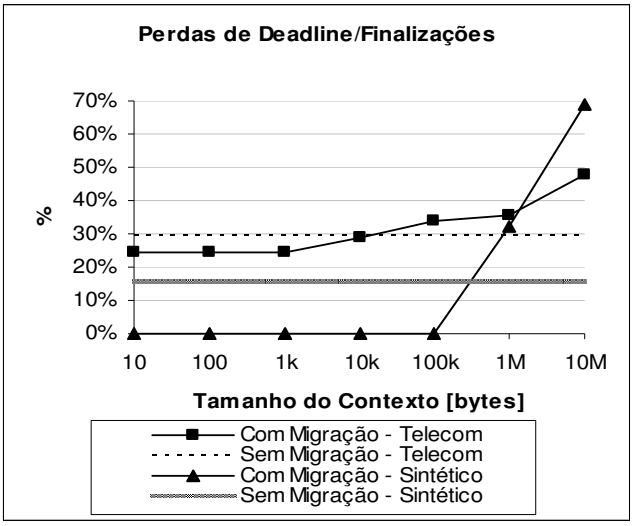


Figura 6.6: Efeito do tamanho do contexto na migração de tarefas, no que diz respeito ao número de deadlines perdidos por finalização.

A Figura 6.7 apresenta a energia consumida na comunicação da rede das aplicações de telecom e sintética, respectivamente. No cálculo de energia, estão incluídas a energia consumida nos roteadores, nos canais, e a energia dissipada pelo processador, no envio e recepção de dados. Se duas tarefas têm grande comunicação entre elas, a heurística de balanceamento de carga (*linear clustering*) tende a posicionar essas uma perto da outra ou até mesmo alocar as duas tarefas no mesmo processador, eliminando a comunicação pelo canal. Isto permite uma redução na energia de comunicação para tamanho de contexto de até 100 KB para a aplicação de telecom e 10 KB para aplicação sintética. Para grandes tamanhos de contextos, a comunicação é dominada pela transferência de contexto de tarefas. O custo de comunicação relativo imposto pela migração é menor para sistemas que têm alta utilização da rede, no caso a aplicação de telecom.

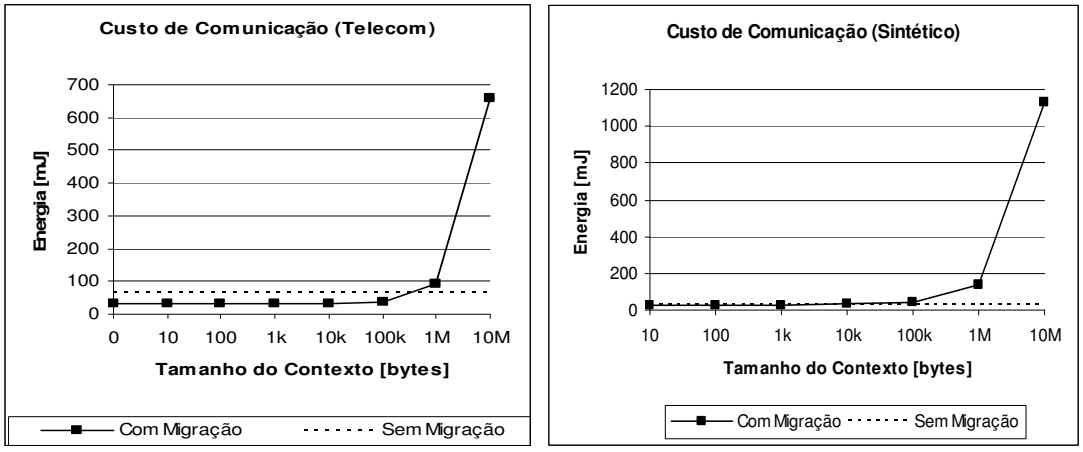


Figura 6.7: Custos de energia de comunicação da rede para as aplicações Telecom e Sintético.

A Figura 6.8 apresenta o custo de energia para o mecanismo de migração de tarefas para a aplicação sintética. O gráfico do mesmo custo energético para a aplicação de telecom não foi apresentado aqui, pois a curva é muito similar. Como foi apresentado, quanto maior o tamanho do contexto, maior o custo energético para transferência de contexto da tarefa.

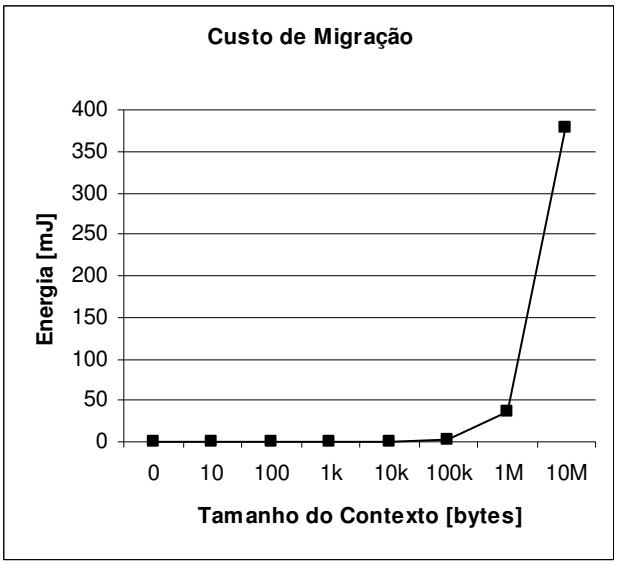


Figura 6.8: Custo energético da migração de tarefas em vários tamanhos de contexto de tarefas.

A Figura 6.9 mostra a energia total consumida pelo sistema para a execução da aplicação sintética durante 200 milissegundos. Para contextos maiores do que 100 KB, o mecanismo de migração aumenta o consumo total de energia de maneira abrupta. A curva da aplicação telecom tem o mesmo comportamento.

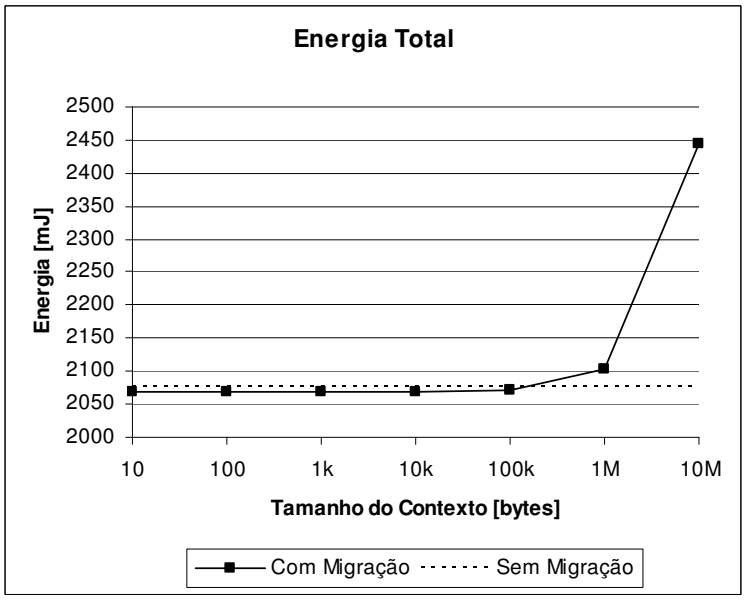


Figura 6.9: Energia total do sistema com e sem o mecanismo de migração de tarefas.

6.2.3 Conclusões

Considerando todos os custos de migração e até mesmo considerando situações de piores casos (tempo de execução razoavelmente pequeno), a migração de tarefas pode ajudar no atendimento de deadlines para sistemas de tempo real *soft*. Porém, estes experimentos não apresentam melhorias quanto ao consumo de energia. Como os processadores foram sobrecarregados e não houve folga no escalonamento das tarefas, o algortimo de DVS não conseguiu obter uma redução energética. Por outro lado, a economia de consumo de energia pode ser obtida pelo desligamento de núcleos inativos (processadores sem tarefas), como será avaliado na próxima Seção. A combinação das técnicas de *bin-packing* e *linear clustering* é interessante para diminuição de deadlines perdidos em tempo de execução, pois assim, o sistema tem menor comunicação entre as tarefas e consegue maior atendimento a deadlines, simultaneamente aumentando o seu paralelismo.

Na Seção anterior, os experimentos mostram que a migração é benéfica, em termos de perda de deadlines, e sem consumir maior custo energético para tarefas com tamanho de contexto menor que 100 Kbytes. Porém, esta informação é totalmente dependente dos parâmetros concretos para estes experimentos tais como política de migração, protocolos de comunicação da rede, política de escalonamento, tamanho da rede, etc.

6.3 Análise das heurísticas de alocação

6.3.1 Estratégias de Simulação

Neste experimento, foram utilizados dois algoritmos *bin-packing*: Worst-Fit e Best-Fit. O primeiro algoritmo tende a balancear carga no sistema e o segundo tende a concentrar cargas em alguns processadores, baseados na utilização do processador. Estes algoritmos foram combinados com o algoritmo de *linear clusterization* (LC). Este algoritmo agrupa tarefas que se comunicam com alta taxa de dados. Desta forma, esta técnica pode minimizar o custo de comunicação entre as tarefas, pelo fato de alocar um grupo de tarefas que se comunicam muito em um mesmo processador. A inclusão deste algoritmo de *linear clusterization* aumenta o número de opções de um algoritmo apropriado para minimizar as figuras tais como consumo de energia e número de deadlines perdidos.

Nesta simulação, foram utilizados três estudos de caso: sintético (*synthetic*), telecom e telecom 2. Em todos estes, primeiramente simularam-se as aplicações com tarefas alocadas de modo aleatório. Os resultados no modo aleatório de alocação correspondem a uma média dos valores de perdas de deadlines e consumo de energia de 10 diferentes alocações aleatórias de tarefas. Em outra etapa, aplicaram-se quatro algoritmos para decidir onde as tarefas serão migradas: Best-Fit, Best-Fit combinado com *linear clustering,* Worst-Fit e finalmente Worst-Fit combinado com *linear clustering*. O tempo de simulação utilizado para todos os experimentos foi de 200 ms. A freqüência dos processadores varia de 100 MHz a 600 MHz, e a tensão dos processadores varia entre 1.3 V a 2.0 V usando DVS. Os processadores que não tiveram tarefas alocadas são desligados, para economia de energia. A rede-em-chip funciona a 266 MHz de freqüência e o tamanho da rede varia de 4x4 a 7x7 processadores. O tamanho de memória para cada processador é de 64KB. Finalmente, os tamanho dos contextos das tarefas são definidos através da Tabela 6.1.

6.3.2 Experimentos

A Figura 6.10, Figura 6.11 e Figura 6.12 apresentam respectivamente a taxa de perda de deadlines[11] das aplicações *synthetic*, *telecom* e *telecom 2* para cada configuração do tamanho da rede-em-chip (4x4 a 7x7 processadores). Essas figuras mostram duas situações distintas: um cenário antes da migração de tarefas (alocação aleatória) e um cenário depois da migração de tarefas (todos os demais algoritmos).

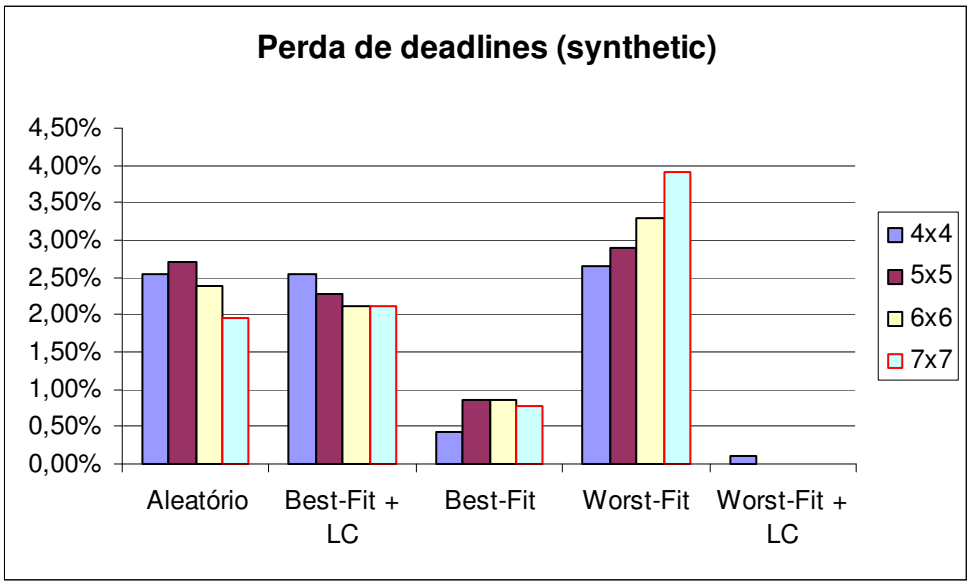


Figura 6.10: Taxa de perda de deadlines para a aplicação *Synthetic*.

Como mostrado na Figura 6.10, não houve uma diminuição significativa da taxa de perda de deadlines na aplicação do algoritmo Best-Fit + LC (combinado com *linear clusterization*) quando comparado com a alocação aleatória. Esta estratégia executa a migração e a alocação de agrupamento de tarefas (*clusters*) nos processadores ao invés de migrar e alocar apenas tarefas simples, como é o caso do Best-Fit puro. Mesmo agrupando tarefas com alta taxa de comunicação em um mesmo processador, o algoritmo Best-Fit + LC não conseguiu diminuir o número de perda de deadlines, pois a alocação dos clusters, em alguns casos, ultrapassou a utilização do processador de 100%, causando perdas de deadlines das tarefas. Na média, o pico máximo de utilização do processador foi de 115%, o que foi suficiente para que muitas tarefas não conseguissem atingir seus prazos de execução.

Por outro lado, o algoritmo Best-Fit puro não cria clusters. Este algoritmo aloca apenas tarefas simples no processador, sem considerar a taxa de comunicação entre as tarefas. Na aplicação do algoritmo Best-Fit, o número de perda de deadlines diminui comparado com a alocação aleatória e Best-Fit + LC, pois as tarefas não excederam a capacidade ocupação de processamento máximo dos processadores.

[11] A taxa de 100% significa que todas as tarefas não conseguiram alcançar seus prazos de execução.

O algoritmo Worst-Fit, diferentemente do algoritmo Best-Fit, distribui tarefas entre os processadores de maneira homogênea. Este algoritmo tende a balancear o sistema em termos de carga de processamento. De acordo com a carga das tarefas, se houver folga nos processadores, é possível aplicar DVS para diminuir o consumo de energia dinâmica. Entretanto, o algoritmo Worst-Fit, com o "espalhamento" de tarefas no sistema, aumenta também a dependência de comunicação entre as tarefas (aumento da taxa de comunicação entre os processadores) e, desta forma, aumenta o número de perda de deadlines. Quando se aumenta o tamanho da rede-em-chip, o número de perda de deadlines aumenta proporcionalmente, pois aumenta o número de *hops* para o tráfego de mensagens. Embora foi obtido um pico de 4% de perdas de deadline, esta taxa é bastante razoável e aplicável em sistemas de tempo real *soft*.

Finalmente, a combinação do algoritmo Worst-Fit + LC reduziu a perda de deadlines para 0% (aplicável até mesmo em sistemas de tempo real *hard*) para a maioria das configurações de tamanho da redes-em-chip. Os clusters puderam ser distribuídos ao longo da rede-em-chip, sem exceder a ocupação de processamento dos processadores. Desta forma o WF+LC possibilitou reduzir a comunicação entre os processadores e consequentemente reduziu o congestionamento e conseguiu executar os clusters sem ou com muito pouca perda de deadlines das tarefas.

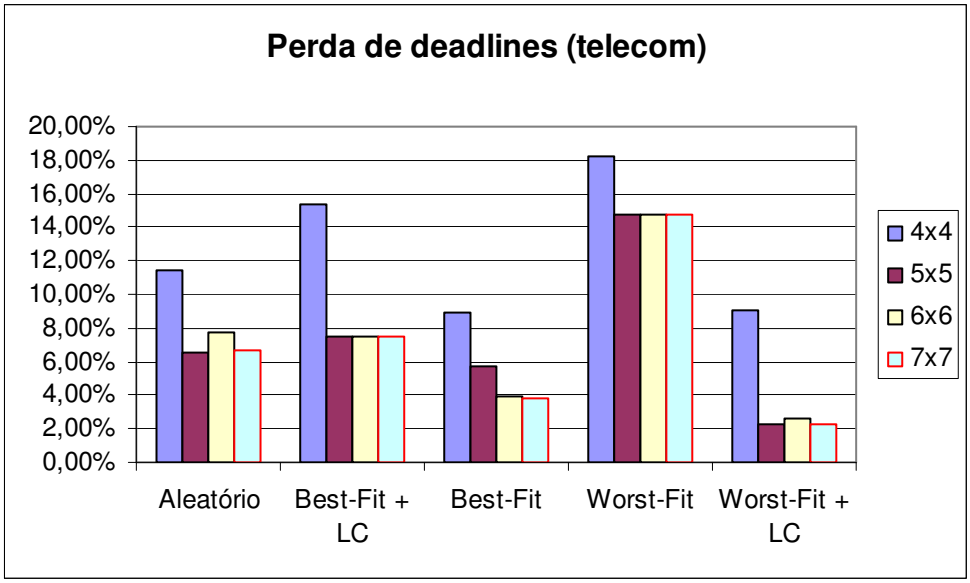


Figura 6.11: Taxa de perda de deadlines para a aplicação *Telecom*.

A Figura 6.11 apresenta a taxa da perda de deadline das tarefas para a aplicação *telecom*, para diferentes configurações de tamanho da rede-em-chip e algoritmos de alocação. Observa-se um comportamento similar à Figura 6.10. Entretanto, a taxa de perdas de deadlines é mais alta, pois a comunicação entre as tarefas da aplicação *telecom* é muito mais alto que a aplicação *synthetic*. Neste caso, o algoritmo WF + LC não conseguiu reduzir a taxa de perda de deadlines para 0%. O número de perda de deadlines do algoritmo WF + LC ficou equiparado com o algoritmo BF na rede-em-chip de tamanho 4x4, pois este tamanho é pequeno demais para esta aplicação, e os algoritmos não conseguem tirar grande proveito na diminuição da taxa de perdas de deadlines.

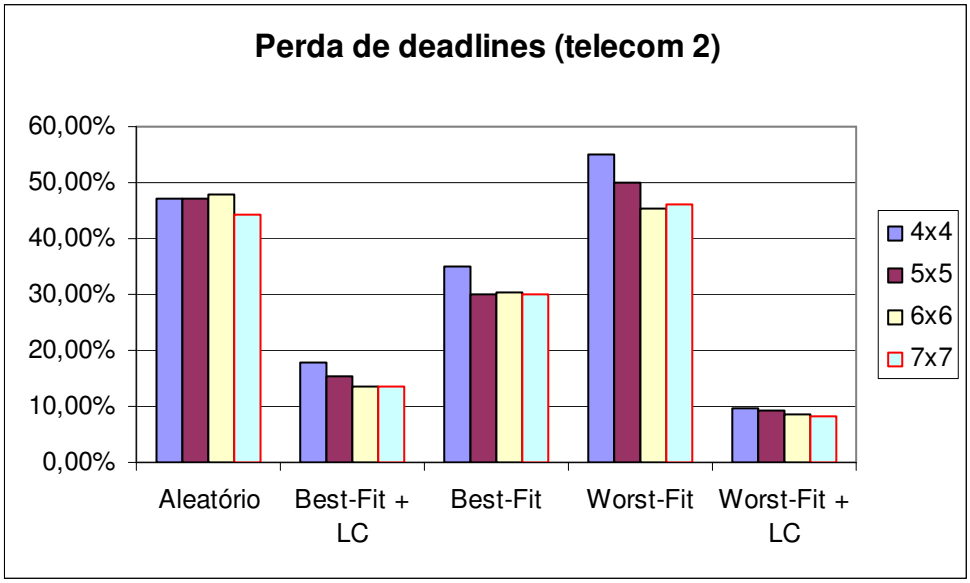


Figura 6.12: Taxa de perda de deadlines para a aplicação *Telecom 2*.

A Figura 6.12, apresenta a taxa da perda de deadline das tarefas para a aplicação *telecom 2*, para diferentes configurações de tamanho da rede-em-chip e algoritmos de alocação. O comportamento dos algoritmos em termos do número perdas de deadlines é muito similar à Figura 6.11. Todavia, devido à alta taxa de comunicação da aplicação *telecom 2*, o algoritmo BF + LC obteve uma menor taxa de perda de deadlines que o algoritmo BF. Neste caso, o LC foi decisivo para a diminuição desta taxa, pois o agrupamento de tarefas que têm alta taxa de comunicação diminui o congestionamento da rede e aumentou o número de tarefas que conseguiram executar dentro dos seus prazos de execução. Quanto à execução do algoritmo WF + LC, o tamanho da rede-em-chip não causou relevante impacto no número de perdas de deadline. O número de *clusters* obtidos neste experimento foi menor que 16 e, desta forma, o algoritmo WF + LC executado em redes com tamanho maior que 4x4 não consegue obter menores números de perda de deadlines.

Estes experimentos consideram aplicações de tempo-real *soft*, onde pequenas taxas de perda de deadlines são toleradas. Obviamente que os números de perda de deadlines obtidos através da Figura 6.10, Figura 6.11 e Figura 6.12 poderiam ser reduzidos através do aumento da largura de banda da rede-em-chip e aumentando a freqüência dos processadores e da rede-em-chip. No entanto, um aumento de consumo significativo de energia no sistema seria inevitável.

A Figura 6.13, Figura 6.14 e a Figura 6.15 apresentam o consumo de energia resultante de todos os algoritmos de alocação aplicados nos três estudos de caso: *synthetic, telecom e telecom 2*. A Figura 6.13 apresenta o consumo de energia resultante de todos os algoritmos de alocação aplicados no estudo de caso *Synthetic*. Percebe-se que os algoritmos BF e BF + LC, como esperado, reduziram o consumo de energia total do sistema, pois os clusters e tarefas são concentrados em um número reduzido de processadores da rede-em-chip, deixando vários processadores sem tarefas. Mesmo os processadores funcionando sem tarefas e com baixa freqüência, muita energia é consumida principalmente pela energia estática e devido às memórias locais. Estes

processadores e suas memórias locais então são desligados e desta forma, é possível reduzir o consumo de energia. Por outro lado, o algoritmo WF tende a espalhar as tarefas na rede-em-chip. Na média, o consumo de energia é maior com a aplicação do algoritmo WF que com os outros algoritmos de alocação. O algoritmo WF + LC apresenta um consumo de energia constante a partir das configurações do tamanho 5x5 a 7x7 da rede-em-chip. Isto se deve ao número de clusters menor que 25, e desta forma, mesmo aumentando o tamanho da rede-em-chip, o número de processadores ativos permanece constante e consequentemente o consumo de energia também permanece constante.

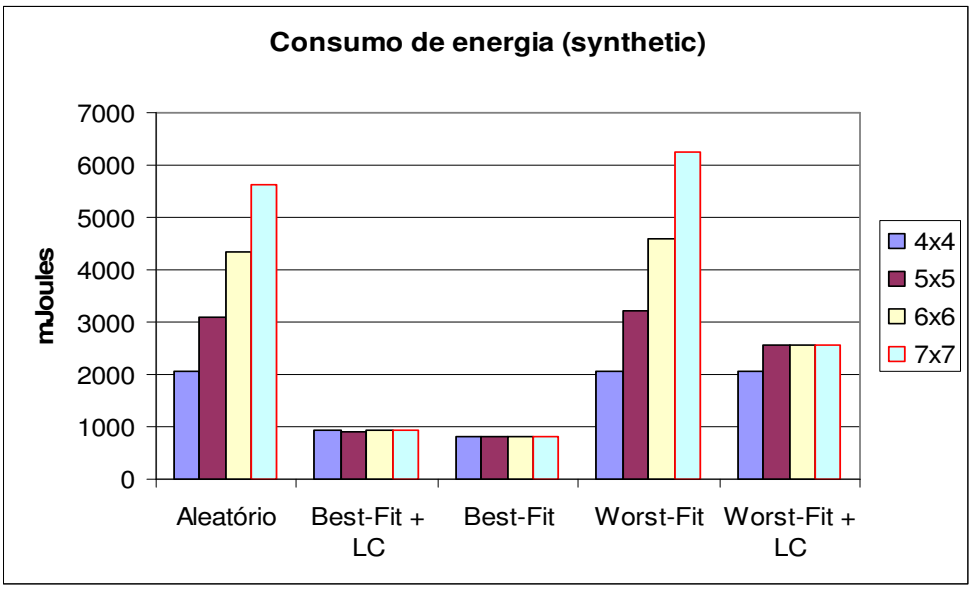


Figura 6.13: Consumo de energia para a aplicação *Synthetic*.

A Figura 6.14 e a Figura 6.15 apresentam o consumo de energia para aplicações *telecom* e *telecom 2* respectivamente. O comportamento dos gráficos, em termos de consumo de energia, é similar aos resultados apresentados na Figura 6.13. Os algoritmos BF e BF + LC apresentam menor consumo de energia, enquanto WF + LC apresenta menor consumo de energia que o algoritmo WF, pois LC agrupa tarefas com alta taxa de comunicação, concentrando algumas tarefas em clusters, e tornando alguns processadores inativos, aptos a serem desligados, assim, economizando energia.

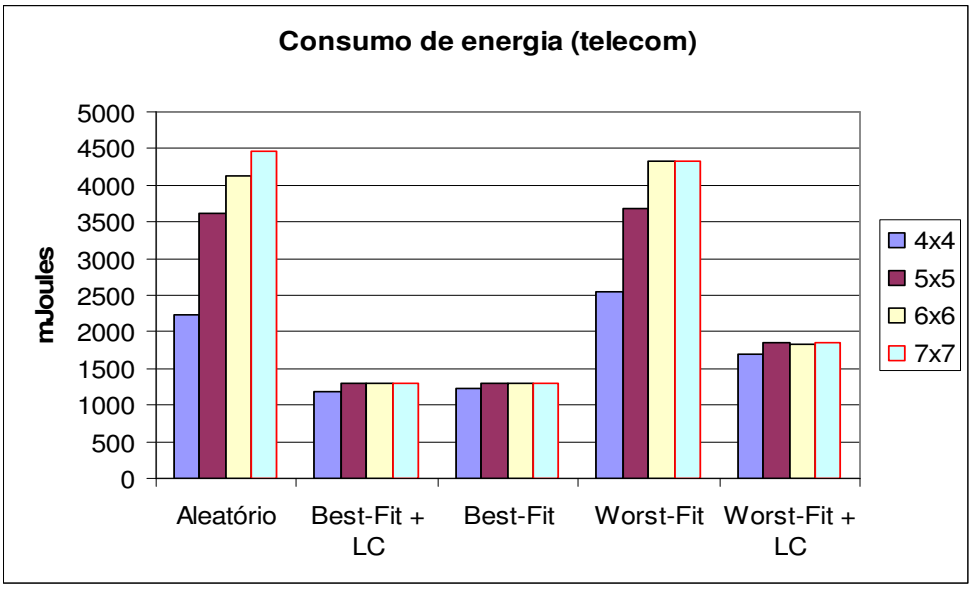


Figura 6.14: Consumo de energia para a aplicação *Telecom*.

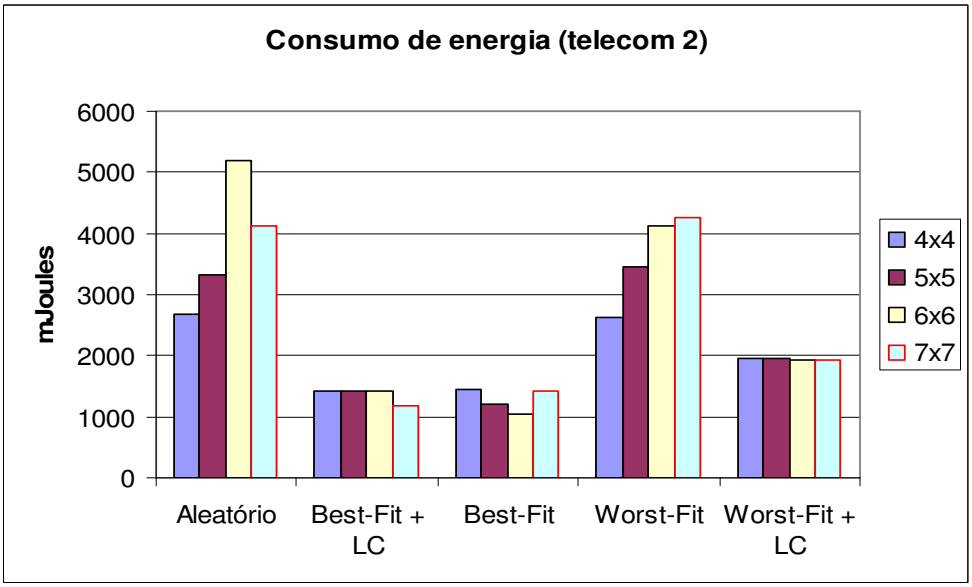


Figura 6.15: Consumo de energia para a aplicação *Telecom 2*.

No processo de migração de tarefas, foi medido o custo de migração em termos de energia e desempenho. A Figura 6.16 e a Figura 6.17 apresentam estes custos para todos os estudos de caso utilizados. Os resultados apresentados na Figura 6.16 e na Figura 6.17 são valores da média dos custos de migração de todas as heurísticas de alocação. Para todas as aplicações, em termos de tempo e energia, os resultados foram muito similares. O tamanho da rede-em-chip não afeta os resultados de maneira significativa, pois os benchmarks não sobrecarregaram as redes-em-chip com tamanhos acima de 4x4.

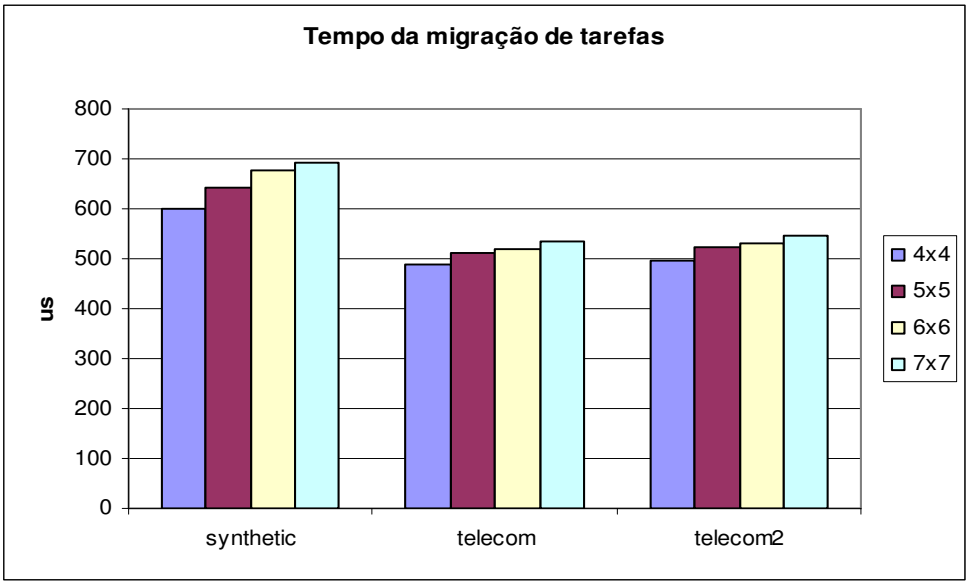


Figura 6.16: Tempo total de migração de tarefas avaliado para as três aplicações.

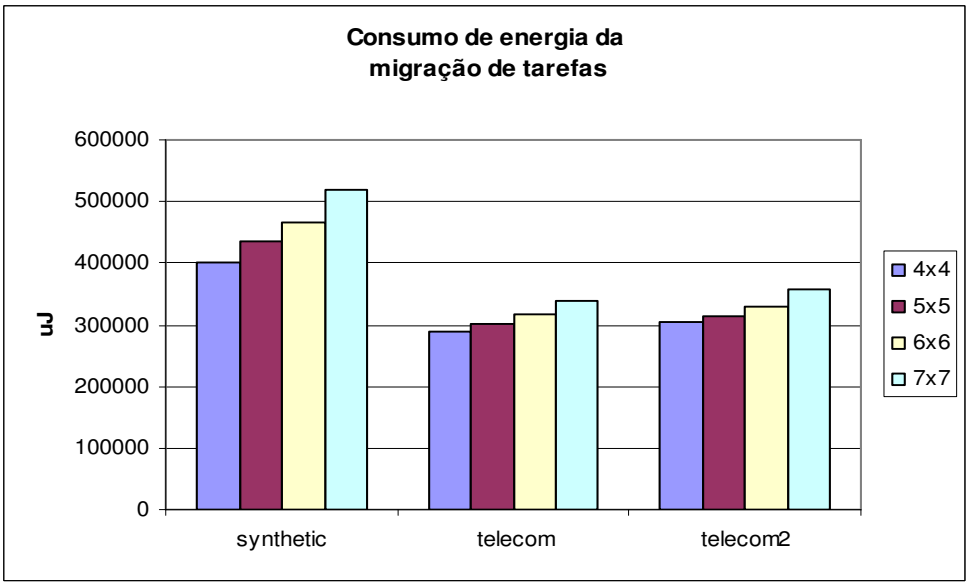


Figura 6.17: Consumo de energia total avaliado da migração de tarefas para as três aplicações.

Os resultados apresentados na Figura 6.16 e na Figura 6.17 também mostram que as migrações levaram menos que 1% do tempo de execução total do sistema, e são responsáveis por pelo menos 2% do consumo de energia, na média. Isto significa que a migração de tarefas pode ser facilmente amortizada pela execução do sistema, desde que o consumo de energia e a perda de deadlines sejam reduzidos de maneira significativa. A aplicação *Synthetic*, segundo a Figura 6.16 e a Figura 6.17, apresentou maior custo de migração (perdas de deadlines e consumo de energia) devido ao número

de tarefas e tamanho de tarefas maiores que as tarefas das aplicações *Telecom* e *Telecom 2*.

6.3.3 Conclusões

Os resultados apresentados demonstraram que existe um compromisso entre perdas de deadlines e consumo de energia quando se aplicam algoritmos *bin-packing* e *linear clusterization* para alocação de tarefas em sistemas embarcados baseados em redes-em-chip. Existem dois algoritmos ilustram este panorama: Best-Fit e Worst-Fit combinados com *linear clusterization*. O algoritmo BF + LC resulta no menor consumo de energia do sistema, porém com razoável perda de deadlines. De forma contrária, o algoritmo WF + LC resulta na menor perda de deadlines, no entanto, resulta um razoável consumo de energia. Consequentemente, em um sistema de tempo-real *soft*, o algoritmo WF + LC seria mais adequado pela redução da perda dos deadlines. Em um sistema que exige o menor consumo de energia, o algoritmo BF + LC é o mais adequado.

Finalmente, considerando o tamanho da rede-em-chip, quanto maior a rede-em-chip, maior será o número de perda de deadlines, desde que cada processador seja ocupado por uma tarefa, no mínimo, e desde que a ocupação das tarefas não ultrapasse os 100% de ocupação do processador. Uma distribuição WF poderia ilustrar esse caso. Por outro lado, a combinação WF + LC apresenta redução da comunicação entre as tarefas, porém aumento da ocupação nos processadores. Isto explica casos em que o tamanho 4x4 tem maior perda de deadlines que as outras configurações de tamanho da rede-em-chip. A rede-em-chip 4x4 tem processadores acima dos 100% de ocupação. Os algoritmos BF e BF + LC tem razoável semelhança destas características.

6.4 Experimentos com restrição de tarefas de E/S

6.4.1 Estratégias de Simulação

Neste experimento, foram utilizadas duas classes de algoritmos: bin-packing convencional (algoritmos utilizados nos experimentos da Seção anterior) e bin-packing com restrição de E/S (*I/O bin-packing*). Tarefas que recebem ou enviam dados para as interfaces de E/S da rede-em-chip são chamadas tarefas de E/S. Os algoritmos I/O *bin-packing* forçam a alocação dessas tarefas na borda da rede-em-chip, possibilitando a maior proximidade da tarefa de E/S juntamente com a interface de E/S da rede-em-chip. Dados de entrada e saída são gerados por módulos de geração de tráfego fora da rede-em-chip. Esses dados de energia e desempenho dos módulos não são contabilizados no simulador, pois esses apenas geram tráfego na rede. A Figura 6.18 apresenta a disposição dos módulos geradores de tráfego na rede-em-chip. Os módulos geradores de tráfego sempre se conectam com uma tarefa de E/S. Desta forma, uma aplicação tem o número de tarefas de E/S limitada de acordo com o número de portas de E/S (Ex. uma rede-em-chip de tamanho 2x2 tem 8 interfaces de E/S, uma rede-em-chip 3x3 tem 12, e assim por diante).

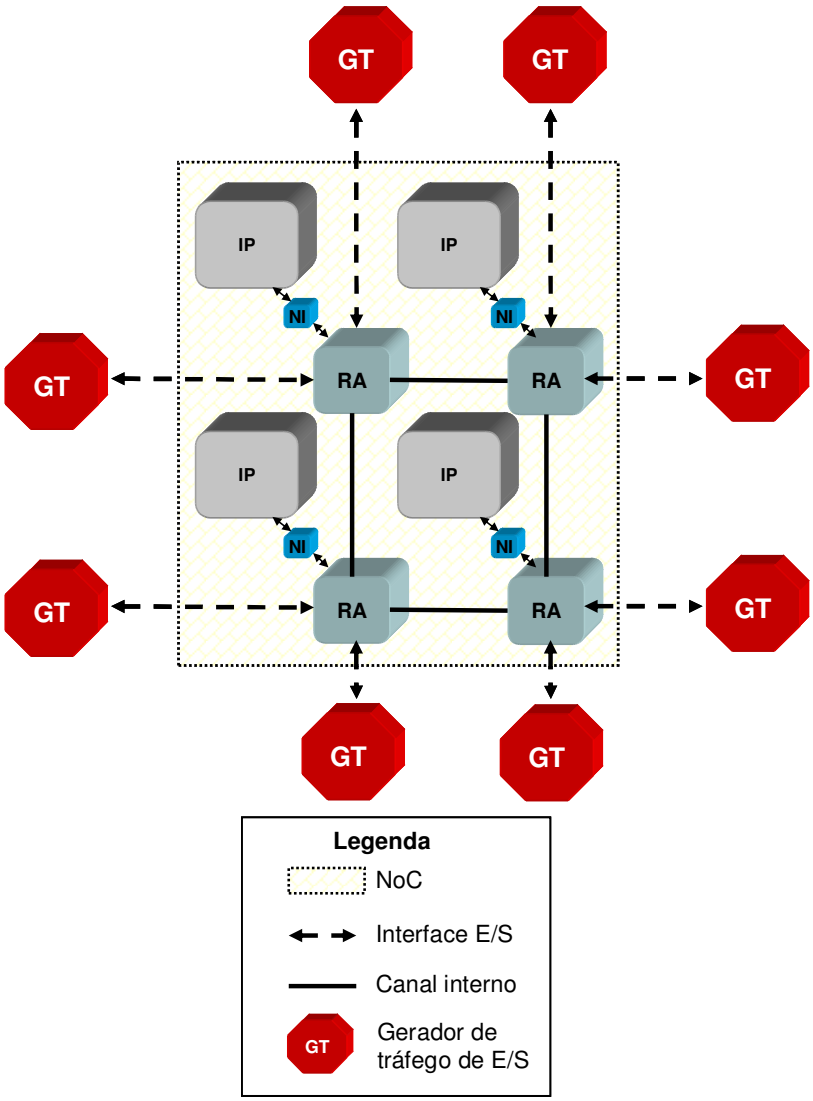


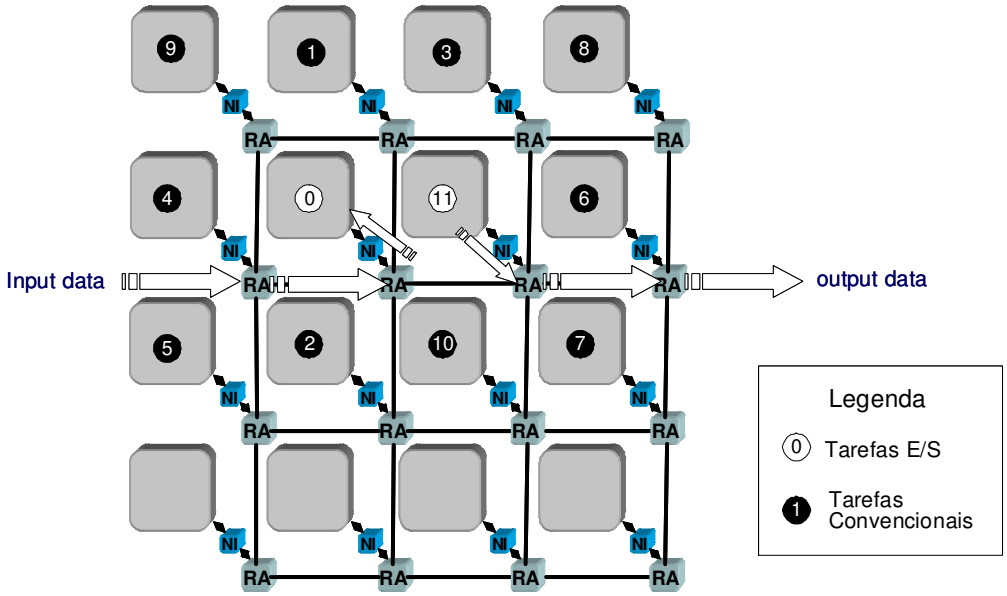
Figura 6.18: Disposição dos módulos geradores de tráfego de E/S na rede-em-chip.

Nesta simulação, foram utilizados três estudos de caso: sintético (*synthetic*), *telecom* e *telecom 2*. Em todos estes, primeiramente simularam-se as aplicações utilizando os algoritmos *bin-packing* convencionais. Os algoritmos são BF, WF e estes combinados com LC (WF + LC e BF + LC). Em outra etapa, simularam-se as aplicações utilizando os algoritmos da classe I/O *bin-packing*. Aplicaram-se quatro algoritmos para decidir onde as tarefas serão migradas: IOPureBF (BF com restrição de E/S), IOPureWF (WF com restrição de E/S), IOBestFit + LC e IOWorstFit + LC. O tempo de simulação utilizado para todos os experimentos foi de 200 ms. A freqüência dos processadores varia de 100 MHz a 600 MHz, e a tensão dos processadores varia entre 1.3 V a 2.0 V usando DVS. Os processadores que não tiveram tarefas alocadas são desligados, para economia de energia. A rede-em-chip funciona a 266 MHz de freqüência e o tamanho da rede foi ajustado para 4x4 processadores. O tamanho de memória para cada processador é de 64KB.

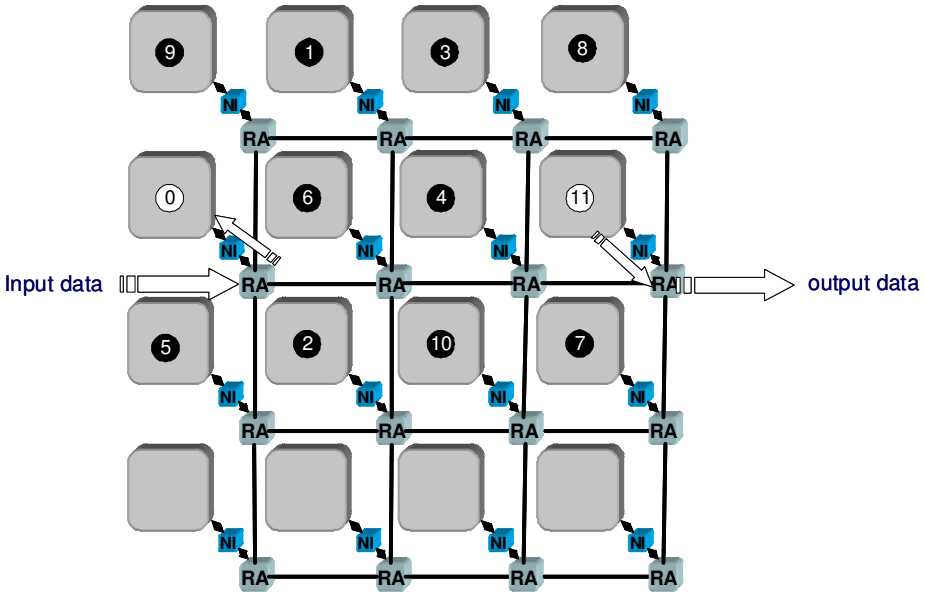
Foi definido, nestes experimentos, que a quantidade de comunicação externa (quantidade de dados de E/S) pode ser 50% maior que a comunicação interna da aplicação usada, ou 50% menor que a comunicação interna da mesma, para diferentes

tipos de cenários (experimentos com quantidade de comunicação externa maior que a quantidade de comunicação interna ou quantidade de comunicação externa menor que a quantidade de comunicação interna). O período de envio de dados de E/S é de 1 ms.

A Figura 6.19 exemplifica as alocações dos algoritmos *bin-packing* convencional e *I/Obin-packing*.



a) Alocação bin-packing convencional



b) Alocação I/O bin-packing

Figura 6.19: a)- Alocação de tarefas utilizando um algoritmo bin-packing convencional (WF neste caso); b)- Alocação de tarefas utilizando um algoritmo bin-packing com restrições de E/S.

Na Figura 6.19-a, com a alocação utilizando um algoritmo bin-packing convencional, (ex: WF) as tarefas de E/S são tratadas sem nenhuma distinção, e então essas tarefas podem ser alocadas a *hops* de distância das interfaces de E/S. Por outro lado, a Figura 6.19-b apresenta a alocação de tarefas resultante da execução de um algoritmo I/O bin-packing (WF com restrição de E/S). Nota-se que as tarefas de E/S são alocadas nos processadores que estão conectados aos roteadores que fazem comunicação com interfaces de E/S. As demais tarefas são alocadas da mesma maneira que no algoritmo *bin-packing* convencional. Estas duas classes de algoritmos geram diferentes alocações, e consequentemente diferentes resultados, como pode ser visto na próxima Seção.

6.4.2 Experimentos

Como foi mencionado na Seção anterior, foram realizadas duas classes de experimentos. Além disso, foram realizados dois tipos de experimentos na classe de algoritmos *I/Obin-packing*: experimentos utilizando alta taxa de dados na interface de E/S da rede-em-chip, maior que a comunicação interna da rede-em-chip, e experimentos com taxas de dados de E/S menor que a comunicação interna de rede-em-chip. O gráfico da Figura 6.20 apresenta os dados das taxas de redução de energia e perda de deadlines em cada um dos algoritmos da classe *I/Obin-packing* em relação a cada um dos algoritmos correspondentes da classe *bin-packing* convencional. Este gráfico apresenta o cenário em que a comunicação interna da rede-em-chip é maior que a comunicação externa.

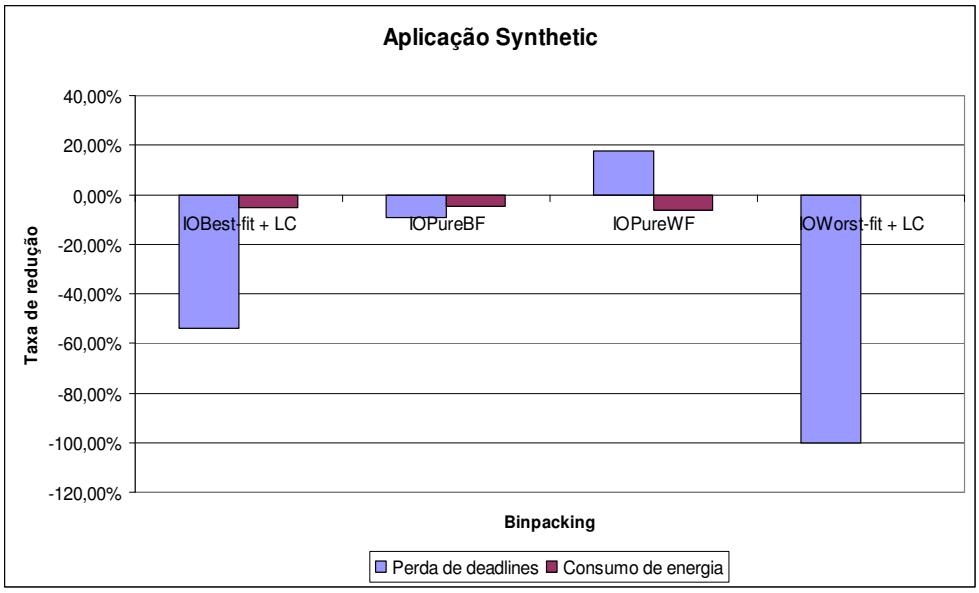


Figura 6.20: Taxa de redução de deadlines perdidos e consumo de energia para aplicação *Synthetic* onde a quantidade de comunicação externa é menor que a quantidade de comunicação interna.

Nota-se que no gráfico da Figura 6.20 são apresentados vários valores negativos. Apenas o algoritmo IOPureWF conseguiu reduzir o número de deadlines perdidos em

relação ao algoritmo WF convencional (valor de taxa de redução positiva). Este resultado se deve ao fato de que o algoritmo IOPureWF alocou algumas tarefas de E/S e tarefas normais mais próximas uma das outras. Estas tarefas de E/S contêm alta taxa de comunicação com tarefas normais que estejam nas proximidades das interfaces de E/S. Desta forma, por essa aproximação destas tarefas, houve redução de perdas de deadlines. Porém, houve uma pequena perda em relação ao consumo de energia, pois, neste caso, houve algum núcleo IP que recebeu mais alguma tarefa, e, desta forma, o DVS deste determinado núcleo IP não conseguiu baixar a freqüência do processador, para poder executar as tarefas sem perda de deadlines. Por outro lado, o algoritmo IOWorst-Fit + LC dobrou a taxa de perda de deadlines, pois mesmo para tarefas de E/S que se comunicavam dentro de clusters alocados em processadores, as tarefas de E/S são alocadas nos processadores que estão nas bordas da rede-em-chip. Desta forma, pelo afastamento das tarefas de E/S dos clusters, foi gerada maior comunicação interna nos clusters, aumentando dramaticamente o número de perda de deadlines. Pela mesma razão, mas com menor impacto que no algoritmo IOWorst-Fit + LC, no algoritmo IOBest-Fit + LC as tarefas de E/S se desvincularam dos clusters e então geraram maior comunicação interna. Finalmente, no algoritmo IOPureBF, houve desprezível redução de perda de deadlines, pois mesmo as tarefas de E/S permanecendo perto das interfaces de E/S na rede-em-chip, essas tarefas com o BF convencional estavam se comunicando com tarefas que se posicionam de maneira mais afastada da interface de E/S. Por essa razão já havia sido gerada uma comunicação interna dentro da rede-em-chip.

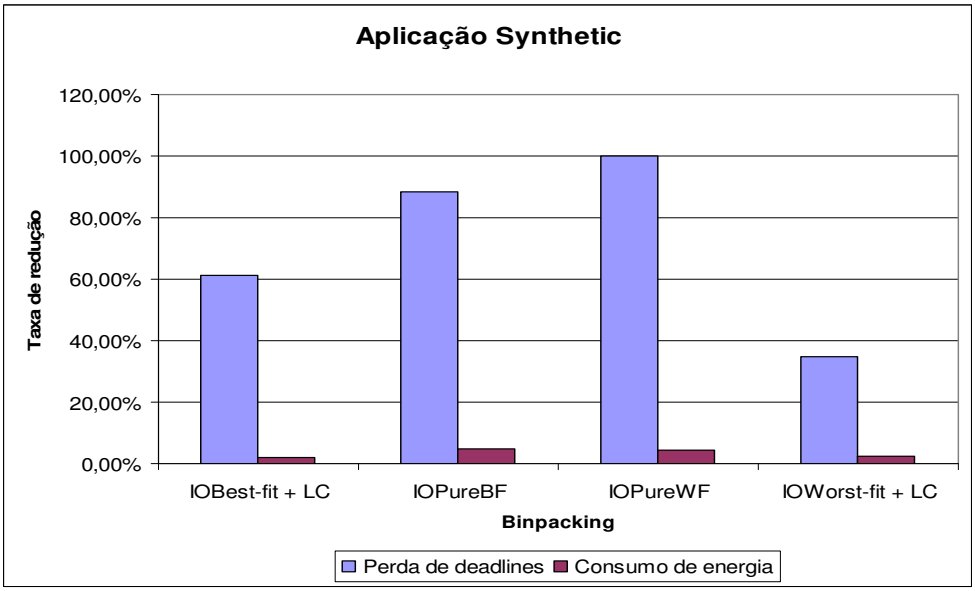


Figura 6.21: Taxa de redução de deadlines perdidos e consumo de energia para aplicação *Synthetic* onde a quantidade de comunicação externa é maior que a quantidade de comunicação interna.

O gráfico da Figura 6.21 apresenta resultados obtidos pela simulação dos algoritmos da classe *bin-packing* convencional e *IObin-packing* para a aplicação *Synthetic*, onde a comunicação externa é maior que a comunicação interna. Nota-se que todos os algoritmos, houve redução das taxas de perdas deadlines e energia, pois a comunicação externa é bem maior que a comunicação interna. Tarefas foram alocadas na

proximidade das interfaces de E/S, de forma a diminuir o número de hops dos dados de E/S e consequentemente diminuir o tráfego interno da rede-em-chip. Desta forma o número de perda de deadlines caiu consideravelmente. Neste caso, todos os algoritmos da classe *IObin-packing* apresentaram vantagens em temos de redução de taxa de perdas de deadlines e consumo de energia comparado com os algoritmos da classe *bin-packing* convencional. O algoritmo IOWorst-Fit apresentou menor taxa de redução, pois, com o distanciamento das tarefas de E/S dos clusters, foi gerado considerável quantidade de comunicação interna, e consequentemente, aumento do número de perdas de deadlines.

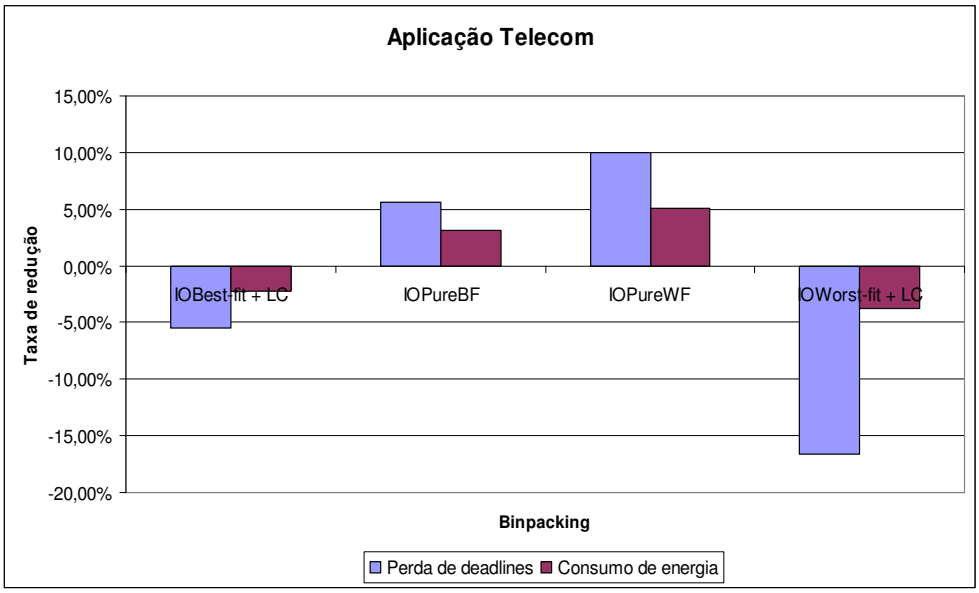


Figura 6.22: Taxa de redução de deadlines perdidos e consumo de energia para aplicação *Telecom* onde a quantidade de comunicação externa menor que a quantidade de comunicação interna.

.

O gráfico da Figura 6.22 apresenta resultados obtidos pela simulação dos algoritmos da classe bin-packing convencional e IObin-packing para a aplicação Telecom, onde a comunicação externa é menor que a comunicação interna. Como é de se esperar, os algoritmos que são combinados com *Linear Clusterization* têm taxa de redução negativa, pois tarefas de E/S que estão clusterizadas na versão bin-packing convencional são alocadas nas bordas da rede-em-chip, gerando maior comunicação interna dentro da rede-em-chip. Os algoritmos IOPureBF e IOPureWF obtiveram resultados positivos da taxa de redução da perda de deadlines e energia, pelo fato das tarefas de E/S, ao serem migradas para perto das interfaces de E/S, não gerarem alta comunicação interna com as demais tarefas do sistema.

A Figura 6.23 apresenta dados relativos à aplicação Telecom, onde a quantidade de comunicação externa é maior que a quantidade de comunicação interna.

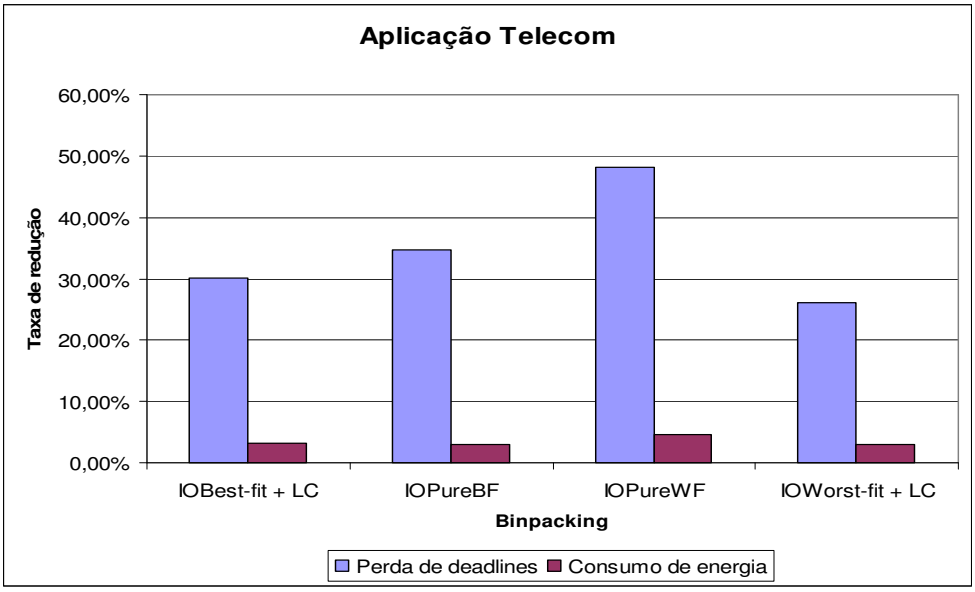


Figura 6.23: Taxa de redução de deadlines perdidos e consumo de energia para aplicação *Telecom* onde a quantidade de comunicação externa maior que a quantidade de comunicação interna.

Neste cenário, todos os algoritmos da classe IObin-packing obtiveram taxas de redução de perda de deadlines entre 37 a 49%. Neste caso, o algoritmo IOpureWF obteve maior taxa de redução, como era de se esperar. Houve taxas de redução menores nos algoritmos da classe IObin-packing combinados com LC, pois ao alocar tarefas de E/S na borda da rede-em-chip, gerou-se comunicação interna e desta forma, houve mais perdas de deadlines.

Finalmente, a Figura 6.24 e a Figura 6.25 apresentam resultados obtidos dos algoritmos bin-packing e IObin-packing da aplicação Telecom2 nos dois cenários de simulação. A Figura 6.24 apresenta resultados de perda de deadlines e energia, onde a comunicação interna é maior que a comunicação externa. Como as tarefas da aplicação Telecom 2 se comunicam com alta quantidade de comunicação, as tarefas de E/S geraram comunicação interna suficiente para aumentar o número de perda de deadlines e consumo de energia. Por outro lado, a Figura 6.25 apresenta uma taxa de redução positiva das figuras perda de deadlines e consumo de energia em todos os algoritmos IObin-packing no cenário onde a quantidade de comunicação externa é maior que a quantidade de comunicação interna. Porém, na média, a taxa de redução da aplicação Telecom 2 é menor que as demais aplicações pela alta taxa de comunicação entre as tarefas. Mesmo alocando tarefas de E/S nas proximidades das interfaces de E/S, a taxa de redução não foi maior que nas outras aplicações pela alta quantidade de comunicação interna.

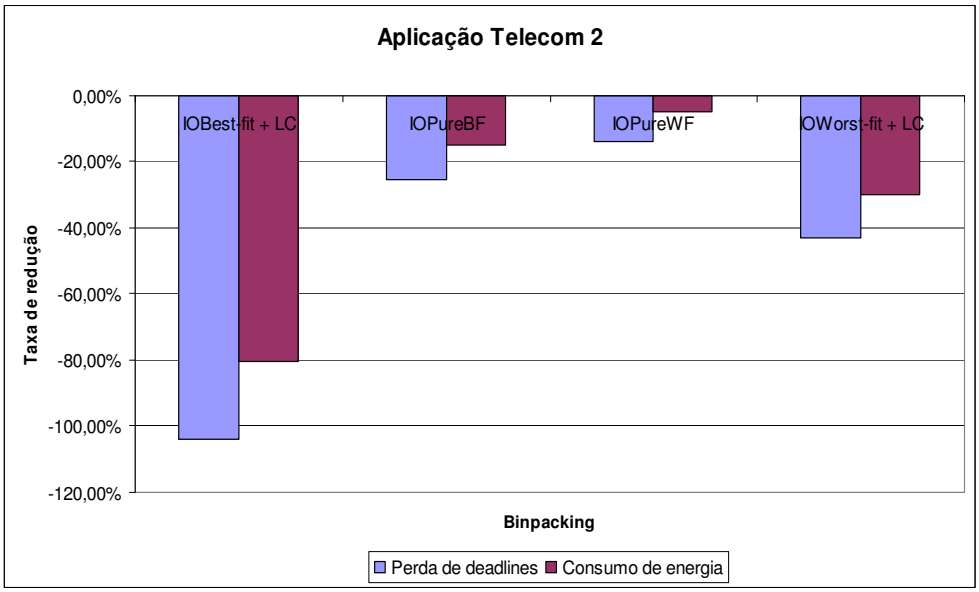


Figura 6.24: Taxa de redução de deadlines perdidos e consumo de energia para aplicação *Telecom2* onde a quantidade de comunicação externa é menor que a quantidade de comunicação interna.

.

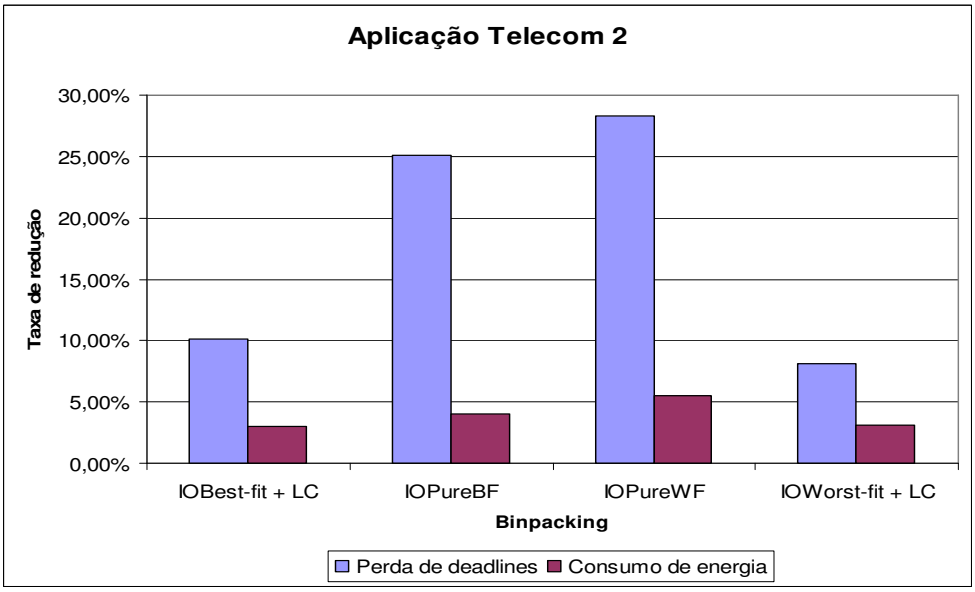


Figura 6.25: Taxa de redução de deadlines perdidos e consumo de energia para aplicação *Telecom2* onde a quantidade de comunicação externa é maior que a quantidade de comunicação interna.

6.4.3 Conclusões

Os resultados apresentados na seção anterior corroboraram o uso de algoritmos bin-packing com restrições de E/S para redução de perda de *deadlines* em aplicações onde a comunicação de E/S é maior que a comunicação interna. De forma contrária, quando a comunicação de E/S é menor que a comunicação interna, os algoritmos bin-packing com restrição de E/S podem trazer alguma taxa de redução negativa. Portanto, os algoritmos de E/S devem ser utilizados se e somente se há o conhecimento *a priori* do comportamento dos dados de E/S da aplicação. Exemplificando, em uma aplicação de HDTV, podem se empregar algoritmos com restrição de E/S para obter menores taxas de perdas de deadlines. No entanto, não se pode dizer a mesma afirmação em aplicações onde a E/S é destinada a periféricos com baixa taxa de comunicação.

Quanto ao consumo de energia, as taxas de redução são significativas no emprego dos algoritmos com restrição de E/S somente quando há alta taxa de comunicação entre as tarefas. Por outro lado, os resultados obtidos no que diz respeito ao consumo de energia indicam pequena redução.

6.5 Variação do tamanho da memória

6.5.1 Estratégias de Simulação

Neste experimento, foram realizadas sete simulações com diferentes configurações do tamanho da memória para cada uma das três aplicações (synthetic, telecom e telecom 2) utilizando o algoritmo Best-Fit para migração de tarefas. O algoritmo Best-Fit foi modificado para suporte de restrição de quantidade de memória. Assim, quanto maior quantidade de memória, mais tarefas estarão alocadas em um mesmo processador (caso não ultrapasse a utilização do processador). Caso contrário, as tarefas tendem a ser espalhadas na rede-em-chip, pois a quantidade de memória pode ser insuficiente para alocação de várias tarefas em um processador.

Os experimentos foram realizados em um tempo de simulação de 200 ms. A freqüência dos processadores varia de 100 MHz a 600 MHz, e a tensão dos processadores varia entre 1.3 V a 2.0 V usando DVS. Para cada simulação, foram obtidos os valores de energia com e sem aplicação de gerenciamento de energia. Os processadores que não tiveram tarefas alocadas são desligados para economia de energia. A rede-em-chip funciona a 266 MHz de freqüência e o tamanho da rede é 4x4 processadores. O tamanho de memória para cada processador variou de 2 a 64KB. Todos os processadores têm o mesmo tamanho de memória.

6.5.2 Experimentos

A Figura 6.26, Figura 6.28 e Figura 6.29 apresentam dados relativos à perda de deadlines das tarefas e consumo de energia em relação à variação do tamanho da memória. A Figura 6.26 apresenta resultados da execução da aplicação Synthetic. Percebe-se que o número de perdas de deadlines permanece constante quando a memória diminui tamanho de 64 KB para 32 KB. A diminuição da memória não afetou o comportamento do algoritmo Best-Fit, que conseguiu alocar todas as tarefas nos processadores, exatamente como na configuração do tamanho de memória de 64 KB. Porém quando o tamanho da memória passa a ser de 10 Kb, o número de perda de deadlines começa a aumentar. Isto ocorreu quando a ocupação de memória foi ultrapassada, e o algoritmo Best-Fit, de maneira forçada, alocou uma tarefa, que seria

alocada a um processador com pouca memória disponível, em um outro processador com menos carga. Consequentemente esta alocação gerou maior comunicação na rede e gerou maior perda de deadlines. Nota-se que, na configuração de memória de 2 KB, o número de perda de deadlines é praticamente o quíntuplo do número de perdas de deadline da configuração de memória de tamanho de 64 KB e 32 KB. Percebe-se que o gerenciamento de energia conseguiu minimizar o consumo de energia do sistema, porém isto foi pouco eficaz para tamanhos de memória de 2 a 4 KB.

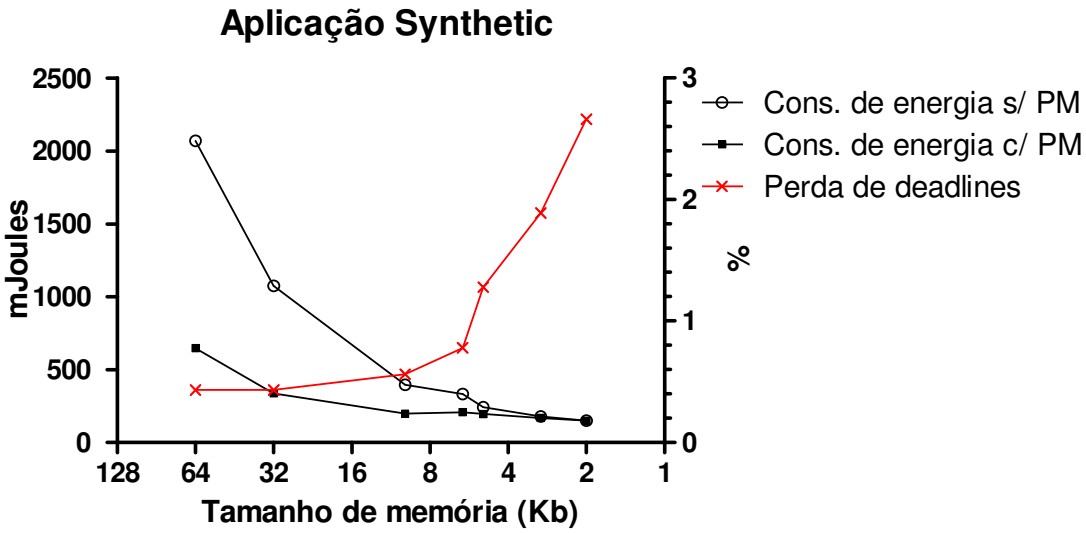


Figura 6.26: Gráfico tamanho de memória versus perda de deadlines e consumo de energia da aplicação *Synthetic*.

Quanto menor o tamanho de memória, o algoritmo Best-Fit se comporta cada vez de forma mais semelhante ao Worst-Fit, pois tende a distribuir as tarefas de maneira homogênea[12] na rede-em-chip. Grande quantidade de memória acarreta em processadores inativos e processadores com muita carga (perto dos 100%). A redução do tamanho de memória resulta em uma redução da carga dos processadores. A Figura 6.27 apresenta quatro configurações de tamanho de memória e a carga dos processadores depois da execução do algoritmo Best-Fit: O Cenário "A" corresponde aos tamanhos de memória de 32Kb a 64 Kb. Nestes casos, o algoritmo Best-Fit não alcançou nenhuma restrição de memória, e sim apenas a restrição da utilização do processador. O Cenário "B" corresponde à memória de tamanho de 10 KB. Neste caso, algumas tarefas foram obrigatoriamente alocadas em outro processador, pois não houve espaço na memória para armazenar as tarefas como foram alocadas no cenário A. No cenário "C", o tamanho de memória é de 5KB e quase todos os processadores foram ocupados. Percebe-se que a utilização de cada processador é reduzida. Finalmente no cenário "D", o tamanho de memória é de 2 KB e, neste caso, todos os processadores são alocados com alguma tarefa, de maneira balanceada. O comportamento do cenário "D" é muito similar ao comportamento do algoritmo Worst-Fit.

[12] Esta premissa é valida para tarefas com pouca variação de utilização dos processadores da rede-em-chip.

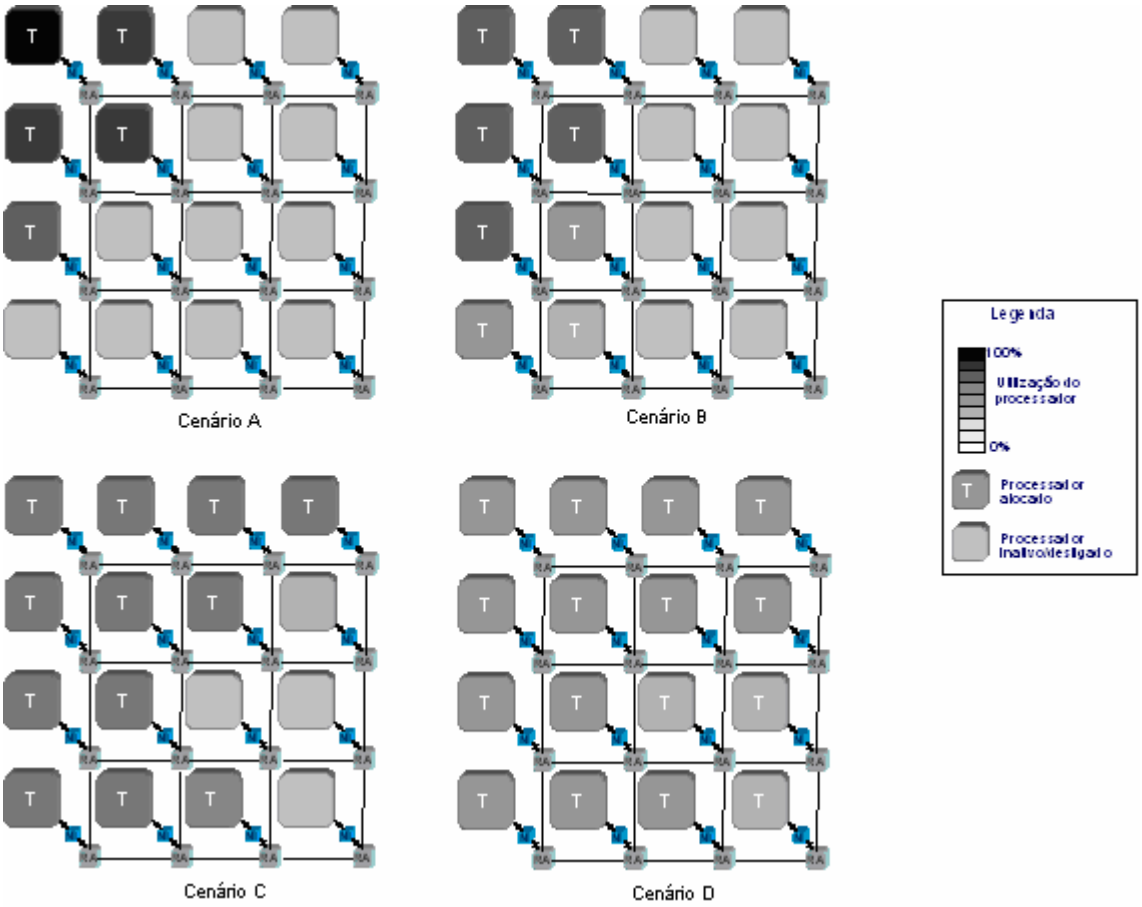


Figura 6.27: Quatro configurações de tamanho da memória para a aplicação *Synthetic* e a distribuição de tarefas realizadas pelo algoritmo Best-Fit com restrição de memória nos quatros cenários.

A Figura 6.28 apresenta resultados da execução da aplicação Telecom. Percebe-se que o número de perdas de deadlines permanece constante quando a memória tem tamanho de 64 KB a 10 KB. A razão para este comportamento é semelhante ao comportamento do experimento da Figura 6.26. Porém, quando o tamanho da memória passa a ser de 6 Kb, o número de perdas de deadlines começa a aumentar. Isto ocorre da mesma forma como foi explicado no parágrafo referente ao experimento da Figura 6.26. Nota-se que, na configuração de memória de 2 KB, o número de perdas de deadlines é praticamente o triplo do número de perdas de deadline da configuração de memória de tamanho de 64 KB, 32 KB e 10 KB.

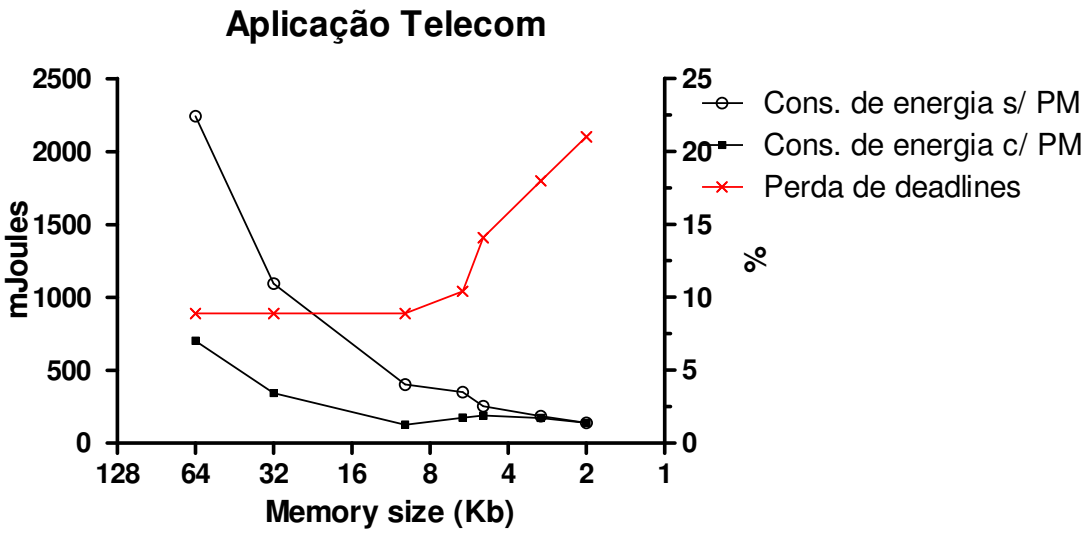


Figura 6.28: Gráfico tamanho de memória versus perda de deadlines e consumo de energia da aplicação Telecom.

Finalmente, a Figura 6.29 apresenta a resultados da execução da aplicação Telecom 2. Nota-se que o comportamento dos gráficos da Figura 6.28 e da Figura 6.29 é similar, porém com uma razoável diferença entre os custos do consumo de energia e diferença significativa entre os custos da perda de deadlines. As aplicações Telecom e Telecom 2 têm o mesmo comportamento, porém a diferença entre ambas está na quantidade de comunicação entre as tarefas. Esta quantidade tem um impacto proporcional no número de perdas de deadlines. A Figura 6.29 apresenta uma variação de perdas de deadlines de 18% a 58 %.

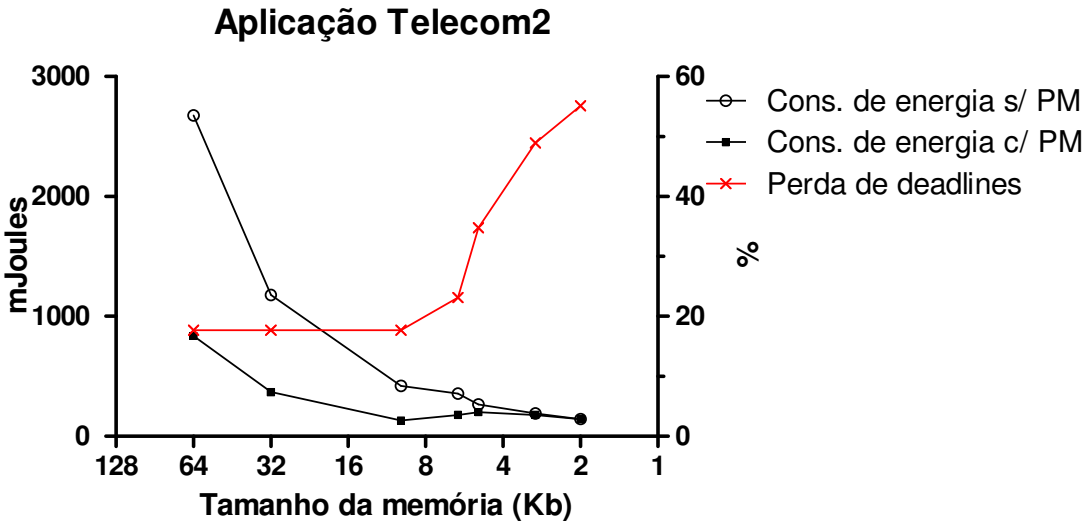


Figura 6.29: Gráfico tamanho de memória versus perda de deadlines e consumo de energia da aplicação Telecom 2.

6.5.3 Conclusões

Os resultados apresentados demonstraram que existe um compromisso entre perdas de deadlines e tamanho de memória (consequentemente consumo de energia), quando se aplica o algoritmo Best-Fit com restrição de memória. Este compromisso é melhor visualizado com a alteração do tamanho de memória. O número de perdas de deadlines permanece constante quando o tamanho da memória não for ultrapassado na alocação de tarefas. Quando o tamanho de memória for insuficiente, o algoritmo Best-Fit alocará as demais tarefas que seriam alocadas no processador em um outro processador com maior memória disponível. Desta forma, gera-se maior comunicação na rede-em-chip e consequentemente aumenta-se o número de perdas de deadlines.

Percebe-se também que o gerenciamento de energia (PM) é muito eficiente em tamanho de memórias de 64 KB a 10 KB. Ganhos energéticos neste intervalo são bastante significativos. Entretanto, em tamanhos de memória de 2 a 4 KB, o gerenciamento de energia demonstra-se pouco eficiente.

Finalmente, quando o algoritmo Best-Fit alcança alguma restrição de memória e quanto o tamanho de memória é reduzido, este algoritmo se assemelha cada vez mais ao algoritmo Worst-Fit, pois quanto menor o tamanho de memória de cada processador, mais tendencioso a distribuir tarefas o algoritmo Best-Fit será. Quanto menor o tamanho de memória, mais homogênea será a distribuição das tarefas e da carga delas.

6.6 Bin-packing 2D

6.6.1 Estratégias de Simulação

A estratégia empregada nestes experimentos é similar à estratégia apresentada na Seção 6.5.1. Nesta estratégia, foram realizadas sete simulações com diferentes configurações do tamanho da memória para cada uma das três aplicações (synthetic, telecom e telecom 2) utilizando o algoritmo Best-Fit da categoria *bin-packing 2D* para migração de tarefas. Ao invés de utilizar apenas a ocupação do processador como parâmetro para alocação, o algoritmo bin-packing 2D utiliza a área retangular formada pelo produto da ocupação do processador e ocupação de memória como parâmetro de alocação. Cada tarefa é formada por retângulos menores formada pelo produto desses dois parâmetros individuais para cada tarefa. Baseando-se dos conceitos sobre *bin-packing* 2D apresentados na Seção 4.4, cada uma das capacidades dos recipientes representando processadores é definida pela ocupação do processador e ocupação de memória.

Os experimentos foram realizados em um tempo de simulação de 500 ms. A freqüência dos processadores varia de 100 MHz a 600 MHz, e a tensão dos processadores varia entre 1.3 V a 2.0 V usando DVS. Para cada simulação, foram obtidos os valores de energia com e sem aplicação de gerenciamento de energia. Os processadores que não tiveram tarefas alocadas são desligados para economia de energia. A rede-em-chip funciona a 266 MHz de freqüência e o tamanho da rede é 4x4 processadores. O tamanho de memória para cada processador variou de 2 a 64KB. Todos os processadores têm o mesmo tamanho de memória.

6.6.2 Experimentos

A Tabela 6.2 apresenta o consumo de energia e a taxa de perda de deadlines para os algoritmos *bin-packing* convencional e *bin-packing* 2D para cada uma das duas aplicações apresentadas nas colunas em negrito (*Synthetic* e *Telecom*). Nota-se que para cada uma das configurações de tamanho de memória, os resultados obtidos através do algoritmo *bin-packing* convencional e o algoritmo *bin-packing* 2D, em termos de consumo de energia e perda de deadlines, foram equivalentes.

Tabela 6.2: Tabela comparativa entre os algoritmos BP convencional e BP 2D em termos de consumo de energia e perda de deadlines.

	Synthetic				**Telecom**			
	Energia (mJ)		Perda de deadlines		Energia (mJ)		Perda de deadlines	
Tamanho da memória	BP conv.	BP 2D	BP conv.	BP 2D	BP conv.	BP 2D	BP conv.	BP 2D
64 Kb	646,5	646,5	0,43%	0,43%	700,9	700,9	17,69%	17,69%
32 Kb	335,7	335,7	0,43%	0,43%	341,8	341,8	17,69%	17,69%
10 Kb	197,4	197,4	0,56%	0,56%	125,0	125,0	17,69%	17,69%
6 Kb	208,3	208,3	0,78%	0,78%	173,9	173,9	23,12%	23,12%
5 Kb	195,5	195,5	1,28%	1,28%	189,1	189,1	34,78%	34,78%
3 Kb	167,7	167,7	1,89%	1,89%	172,6	172,6	48,91%	48,91%
2 Kb	148,8	148,8	2,66%	2,66%	138,2	138,2	55,14%	55,14%

No caso do algoritmo *bin-packing* 2D, este utiliza como função de alocação o produto entre a ocupação do processador e a ocupação da memória. Este produto é visto como uma área entre essas figuras. Se a área da tarefa, ou seja, o produto de suas ocupações de memória e processador, for menor que a área do processador, então duas restrições devem ser validadas: (i) se a tarefa tem ocupação do processador menor ou igual à ocupação restante do processador a ser alocado; (ii) se a tarefa tem ocupação de memória menor ou igual à ocupação de memória restante do processador a ser alocado. Caso qualquer uma dessas verificações não for válida, então a tarefa deverá ser alocada em outro processador.

6.6.3 Conclusões

Através dos resultados apresentados na Tabela 6.2, foi visto que os resultados de consumo de energia e perda de deadlines para cada um dos tipos dos algoritmos *bin-packing* (convencional e 2D) são rigorosamente iguais. Estes resultados demonstram que a abordagem 2D não tem qualquer eficiência em comparação ao algoritmo *bin-packing* convencional com restrição de memória. O algoritmo *bin-packing* 2D é similar, em termos de complexidade, ao algoritmo *bin-packing* convencional, salvo a maneira como ele ordena os valores para alocação, que, no caso da implementação 2D, ordena os valores do produto da ocupação de memória com a ocupação do processador.

6.7 Uso de heurística de otimização em tempo de projeto

6.7.1 Estratégias de Simulação

O motivo deste experimento é apresentar a comparação dos algoritmos *bin-packing* com um algoritmo *offline*, o qual apresenta um resultado sub-ótimo. Para a realização deste experimento, o algoritmo apresentado na Figura 4.8 foi modificado para possibilitar a integração com o Serpens. Esta estratégia de simulação é baseada nos trabalhos publicados em (OLIVEIRA, 2007) e (OLIVEIRA, 2006). A Figura 6.30 apresenta essas modificações.

```
PROCEDIMENTO SIMULATED_ANNEALING_MODIFICADO()
INPUT: temperaturaInicial, temperaturaFinal: REAL;
OUTPUT: solução: ESTADO
VAR dados_serpens = REGISTRO
        Energia: REAL;
        Deadlines: REAL; //taxa em relação ao número de tarefas
        Solução: MATRIZ[NxN]; // solução de posicionamento
        FIM_REGISTRO;

dados: dados_serpens;

Executa_Serpens(NULL);// Executa o simulador.  Alocação aleatória
dados = Ler_Dados_Arquivo_Serpens();
solução = dados.solução;
temperatura = temperaturaInicial;

ENQUANTO (temperatura > temperaturaFinal) FAÇA
        REPITA
                novaSolução = Perturbação(solução);
                ΔC = Avaliação(solução, novaSolução);
                SE (Decisão(ΔC, temperatura) == ACEITA) ENTÃO
                        solução = novaSolução;
                FIM_SE
        ATÉ (sistema em equilíbrio nessa temperatura);
        temperatura = funçãoDeResfriamento(temperatura);
FIM_ENQUANTO
FIM_PROCEDIMENTO

FUNÇÃO Perturbação(solução:ESTADO): ESTADO
        SE prob > 70% ENTÃO
                Escolher aleatoriamente duas tarefas na solução;
                Trocar as posições entre elas na solução;
        SENÃO SE prob <= 70% ENTÃO
                Escolher aleatoriamente uma tarefa na solução;
                Mover a tarefa para outro processador na solução;
        FIM_SE
        Executa_Serpens(solução); //Serpens executará as
modificações realizadas
        dados = Ler_Dados_Arquivo_Serpens();
        (dados atualizados depois da execução da perturbação)
RETORNA solução
FIM_FUNÇÃO
```

Figura 6.30: Pseudocódigo do algoritmo *Simulated Annealing* modificado para integração com o simulador *Serpens*.

As modificações apresentadas na Figura 6.30 mostram a integração do *Simulated Annealing* com o simulador *Serpens*. O procedimento ExecutaSerpens(NULL) executará o simulador e, quando seu parâmetro é vazio (NULL), a alocação das tarefas é feita de maneira totalmente aleatória. Depois que o simulador é executado, este gerará arquivos com várias resultados, entre eles o número de perdas de deadlines e o consumo de energia. Estes arquivos então são lidos pela função Ler_Dados_Arquivo_Serpens() e os dados são armazenados no registro *dados*.

Outra modificação que foi realizada para permitir a integração do Serpens foi a alteração da função Perturbação. Esta função é responsável por gerar soluções vizinhas à solução atual. O simulador gera uma determinada alocação. Então, a perturbação pode escolher duas tarefas e trocá-las de processador ou escolher uma tarefa e movê-la para outro processador. A função Perturbação adota 70% de probabilidade para realização de trocas de tarefas e 30% de probabilidade de movimentos de tarefas. Segundo Hentschke (2003), a probabilidade com a distribuição de 70% de trocas e 30% de movimentos resultou em uma convergência mais rápida do SA. Depois, estas modificações no posicionamento das tarefas são realimentadas no simulador, e este gerará os dados (energia e perdas de deadlines) para o cálculo do custo da solução perturbada. A cada iteração da função Perturbação, quando a solução gerada é melhor que a solução atual, então esta passa a ser a solução gerada. A perturbação será executada diversas vezes até o algoritmo *Simulated Annealing* termine a sua execução. O término de execução pode ser de acordo com o número de iterações definido pelos parâmetros do SA ou quando o algoritmo converge em algum resultado antes mesmo do numero de iterações.

Neste experimento, além da simulação dos algoritmos *bin-packing* convencionais em 2 s, foram utilizados três métodos para execução do algoritmo *Simulated Annealing*:

- *SA_Energy*: a função custo baseia-se apenas nos dados de consumo de energia resultantes da simulação do Serpens. Função custo é dada por:

```
FUNÇÃO Custo(solução:ESTADO): REAL
RETORNA solução.Energia;
FIM_FUNÇÃO
```

- *SA_Deadline*: a função custo baseia-se apenas nos dados de perdas de deadlines resultantes da simulação do Serpens. Função custo é dada por:

```
FUNÇÃO Custo(solução:ESTADO): REAL
RETORNA solução.Deadlines;
FIM_FUNÇÃO
```

- *SA_Both*: a função custo baseia-se nos dados de perdas de deadlines e consumo de energia simultaneamente, resultantes da simulação do Serpens. As constantes, α, β e Λ normalizam os dados consumo de energia e perdas de deadlines, pois estes têm grandezas diferentes. Função custo é dada por:

```
FUNÇÃO Custo(solução:ESTADO): REAL
RETORNA (α*solução.Deadlines + β*solução.Energia)/Λ;
FIM_FUNÇÃO
```

Os três métodos são baseados nos trabalhos publicados em (OLIVEIRA, 2007) e (OLIVEIRA, 2006). Como parâmetros para a simulação, para cada uma das três

simulações, 1000 perturbações[13] foram realizadas, o que resulta em 1000 execuções diferentes para cada tipo de simulação (*SA_Energy*, *SA_Deadlines* e *SA_Both*).

Os experimentos foram realizados em um tempo de simulação de 2 s. Foram utilizados os algoritmos *bin-packing* com restrição de memória para alocação e migração das tarefas. A freqüência dos processadores varia de 100 MHz a 600 MHz, e a tensão dos processadores varia entre 1.3 V a 2.0 V usando DVS. A rede-em-chip funciona a 266 MHz de freqüência e o tamanho da rede é 4x4 processadores. O tamanho de memória para cada processador é de 64 KB.

6.7.2 Experimentos

Em cada uma das três aplicações (*Synthetic*, *Telecom* e *Telecom 2*), foram realizadas quatro simulações relativas aos algoritmos *bin-packing* (BF, BF + LC, WF e WF + LC) com restrição de memória, uma simulação de modo aleatória[14] e três simulações usando a meta-heurística Simulated Annealing, cada uma priorizando as seguintes figuras: perda de deadlines (*SA_Deadline*), consumo de energia (*SA_Energy*) e ambos (*SA_Both*). A Figura 6.31 apresenta os dados de consumo de energia e perdas de deadlines de todos os experimentos relacionados aos algoritmos *bin-packing* e *Simulated Annealing* para aplicação *Synthetic*.

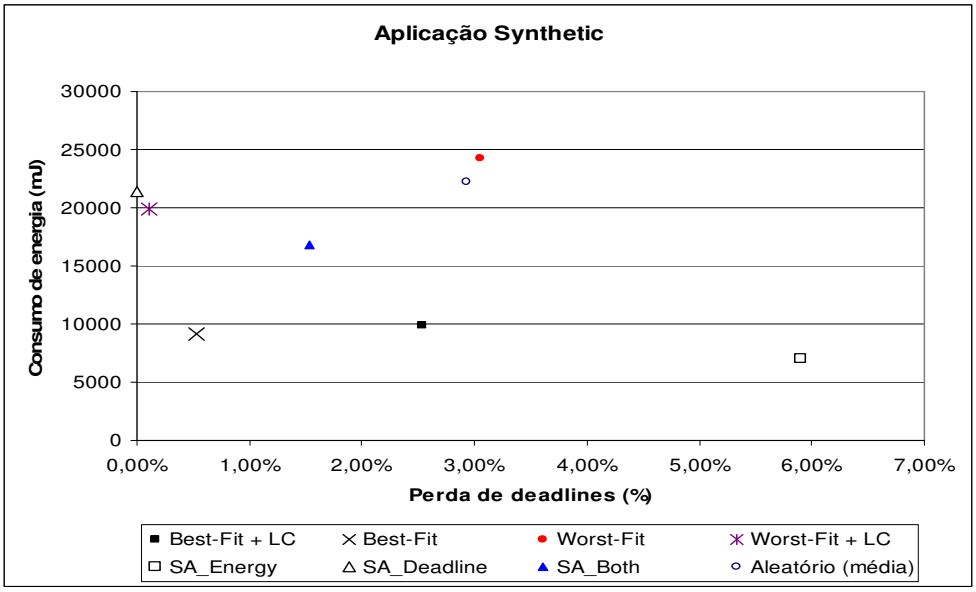


Figura 6.31: Gráfico Perda de deadlines versus consumo de energia dos algoritmos bin-packing convencional e *Simulated Annealing* para a aplicação *Synthetic*..

[13] Das 1000 perturbações, em média, cerca de 700 perturbações foram realizadas através da troca de posições entre duas tarefas, e cerca de 300 perturbações foram realizadas através da alteração da posição de apenas 1 tarefa.

[14] O método aleatório de alocação corresponde a 50 simulações aleatórias e então foi extraída a média do número de perda de deadlines e do consumo de energia.

A Figura 6.31 mostra que, como era de se esperar, os algoritmos *SA_Energy* e *SA_Deadline* conseguiram obter resultados melhores que os algoritmos bin-packing, pois eles foram executados de maneira *offline*. O algoritmo *SA_Deadline*, neste caso, conseguiu anular qualquer efeito de perda de deadlines. O algoritmo *SA_Energy* conseguiu minimizar o consumo de energia em 23% em relação ao algoritmo Best-Fit, onde este último resultou em menor consumo de energia dos algoritmos bin-packing. O algoritmo *SA_Both* resultou em menor consumo de energia que os algoritmos WF+LC, WF, do tipo modo aleatória e *SA_Deadline*. *SA_Both* foi desenvolvido com o intuito de reduzir ambas as figuras: número de deadlines e energia. Porém, este algoritmo não conseguiu alcançar este objetivo.

A Figura 6.32 apresenta os dados de consumo de energia e perdas de deadlines de todos os experimentos relacionados aos algoritmos *bin-packing* e *Simulated Annealing* para aplicação *Telecom*.

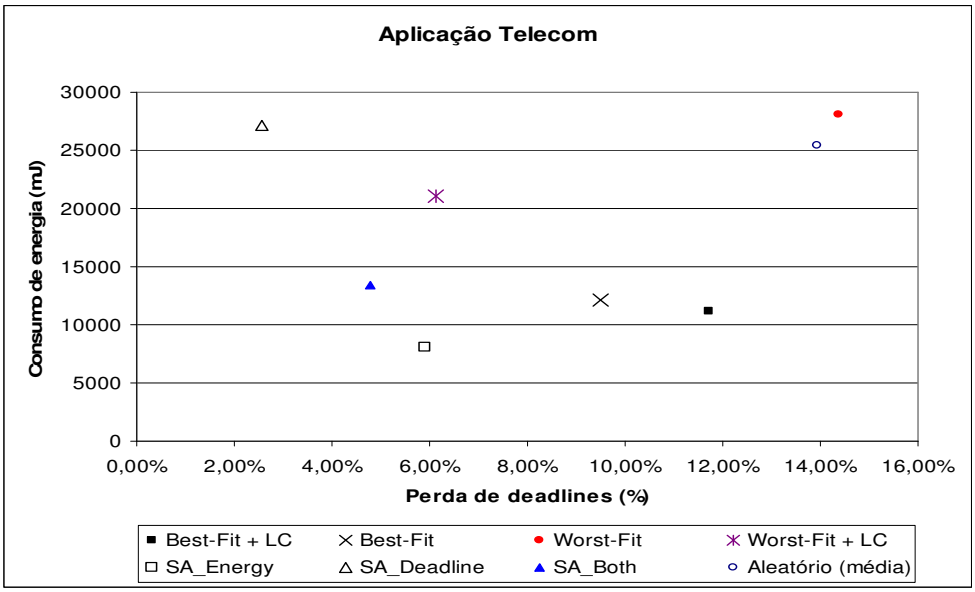


Figura 6.32: Gráfico Perda de deadlines versus consumo de energia dos algoritmos bin-packing convencional e *Simulated Annealing* para a aplicação *Telecom.*

Os resultados da aplicação *Telecom* apresentam um ponto bastante interessante que é o *SA_Energy*. Este algoritmo conseguiu gerar uma alocação de tarefas que obteve simultaneamente menor número de perda de deadlines e menor consumo de energia, em relação aos algoritmos *bin-packing*. Em termos de perda de deadlines, ele é ligeiramente melhor (0,05%) que o algoritmo WF+LC. O algoritmo *SA_Energy* conseguiu minimizar o consumo de energia em 27,4% em relação ao BF+LC. Por outro lado, o algoritmo *SA_Deadline* resultou em uma alocação que reduziu o número de perda de deadlines em 58% em relação ao algoritmo WF+LC. Porém, este ponto apresentou um alto consumo de energia comparado com a maioria dos algoritmos bin-packing (exceto o algoritmo WF que apresentou consumo de energia maior que o algoritmo *SA_Deadline*). Da mesma forma, o algoritmo *SA_Both* não conseguiu minimizar os valores das duas métricas.

Finalmente, a Figura 6.33 apresenta os dados de consumo de energia e perdas de deadlines de todos os experimentos relacionados aos algoritmos *bin-packing* e *Simulated Annealing* para a aplicação *Telecom 2*. Os resultados foram previsíveis,

tomando como base os resultados anteriores, apresentados na Figura 6.31 e na Figura 6.32. O algoritmo *SA_Energy* conseguiu minimizar o consumo de energia em 25,2% em relação ao BF+LC que foi o algoritmo bin-packing que resultou no menor consumo de energia. Por outro lado, o algoritmo *SA_Deadline* resultou em uma alocação que reduziu o número de perda de deadlines em 68% em relação ao algoritmo WF+LC.

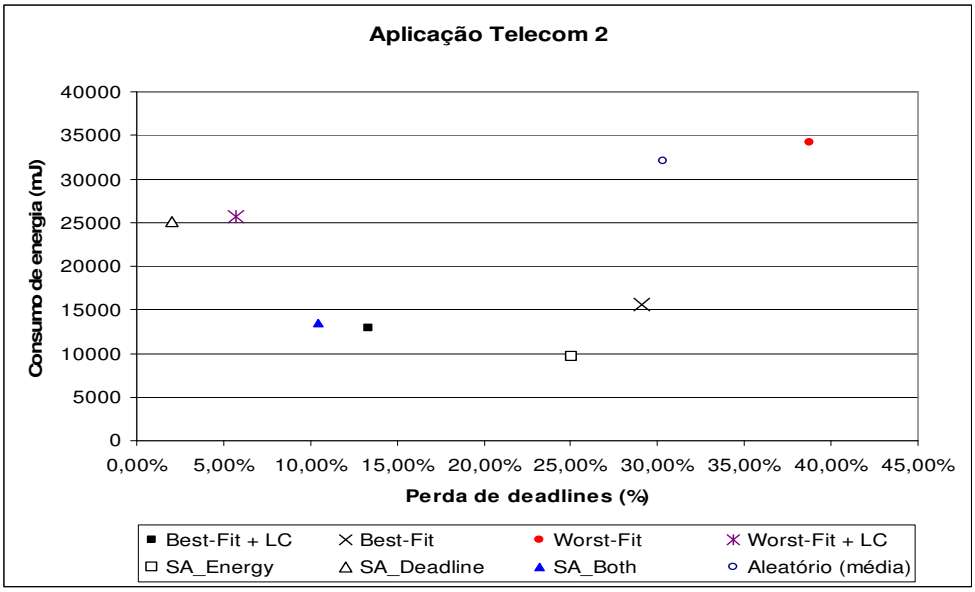


Figura 6.33: Gráfico Perda de deadlines versus consumo de energia dos algoritmos *bin-packing* convencional e *Simulated Annealing* para a aplicação *Telecom 2.*

6.7.3 Conclusões

Os resultados apresentados demonstraram que o algoritmo *Simulated Annealing*, por ter um custo computacional muito alto e por ser aplicado de forma *offline*, todas as instâncias obtiveram resultados melhores no que diz respeito ao consumo de energia e perda de deadlines. Percebe-se que o algoritmo *SA_Energy*, em média, reduz 23% a 27% o consumo de energia do sistema em relação aos algoritmos bin-packing. Por outro lado, o algoritmo *SA_Deadline* apresenta resultados que variam de 58% a 100% de redução de perda de deadlines. Porém, ainda assim, a justificativa do uso dos algoritmos *bin-packing* é que eles podem ser usados no sistema de maneira *on-line,* já que seus custos computacionais são baixos. Outra justificativa é a possibilidade de aumento da freqüência que poderia diminuir ainda mais a distância entre os algoritmos *bin-packing* com o SA. No entanto, aumentar a freqüência significa aumentar o consumo de energia, figura crítica para os sistemas embarcados atuais.

6.8 Variação de freqüência dos processadores

6.8.1 Estratégias de Simulação

Neste experimento, foram realizadas várias simulações, com diferentes configurações das freqüências dos processadores para a aplicação *Telecom 2.* Nestes experimentos, foram utilizados os algoritmos *bin-packing* combinados com LC. Para cada experimento, variou-se a freqüência de todos os processadores de modo a diminuir

o número de perda de deadline de cada algoritmo. Deste experimento, também foram extraídos resultados relativos ao consumo de energia.

A freqüência dos processadores varia de 100 MHz a 2200 MHz, e a tensão dos processadores varia entre 1.3 V a 2.5 V usando DVS. Os processadores que não tiveram tarefas alocadas são desligados para economia de energia. A rede-em-chip funciona de 100 a 2200 MHz de freqüência e o tamanho da rede foi ajustada para 4x4 processadores. O tamanho de memória para cada processador é de 64KB. O tempo de simulação de cada um dos experimentos é de 2 s.

6.8.2 Experimentos

A Figura 6.34 apresenta a taxa de perda de deadlines em relação à freqüência dos processadores para cada um dos algoritmos *bin-packing* e combinados com LC. Com o aumento da freqüência, o desempenho dos processadores aumenta e, desta forma, conseguem executar de forma crescente, as tarefas dentro de seus prazos de execução. Percebe-se que, para os algoritmos combinados com LC (BF e WF), estes conseguiram zerar o número de perda de deadlines com uma freqüência menor que as outras abordagens. Isto obviamente se deve ao fato do algoritmo LC, como mencionado anteriormente, diminuir o número de perda de deadlines. Nota-se também que os algoritmos que obtiveram maiores taxas de perdas de deadlines (modo aleatório e WF) necessitaram de aumento da freqüência dos processadores de mais de 100% da freqüência empregada para zerar os números de deadlines dos algoritmos combinados com LC.

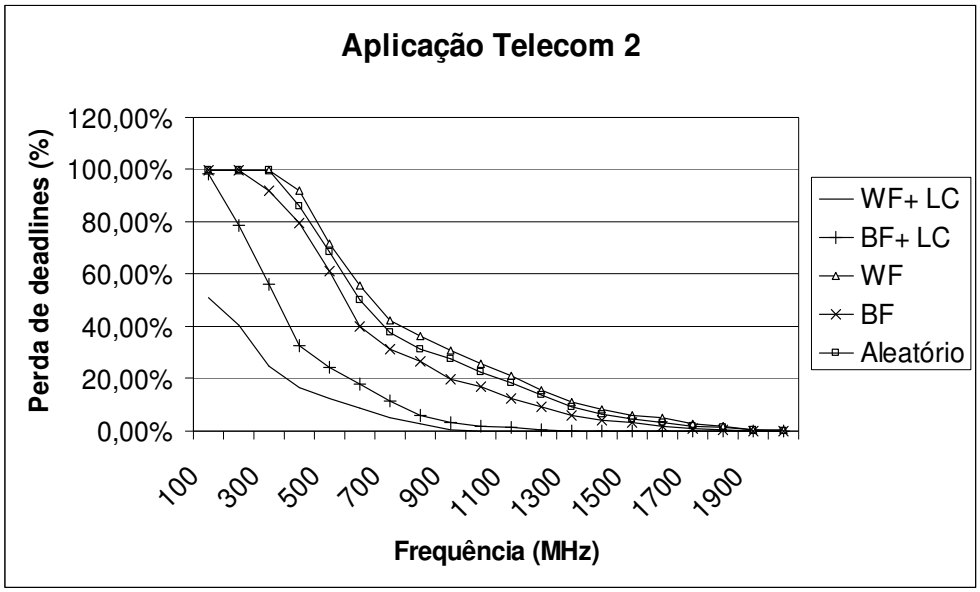


Figura 6.34: Gráfico Perda de deadlines versus freqüência de operação dos processadores para cada um dos algoritmos *bin-packing* para a aplicação *Telecom 2.*

A Figura 6.35 apresenta o consumo de energia em relação à freqüência dos processadores para cada um dos algoritmos utilizados neste experimento. Com o aumento da freqüência, o consumo de energia aumenta também, de acordo com os

modelos de potência e energia utilizados que podem ser encontrados na Seção 5.1.6. Percebe-se que os algoritmos do tipo modo aleatório e WF são os que mais consomem energia no sistema, pois as tarefas tendem a estarem espalhadas. Consequentemente, existirão menos processadores inativos, e com isso há um aumento significativo de consumo de energia.

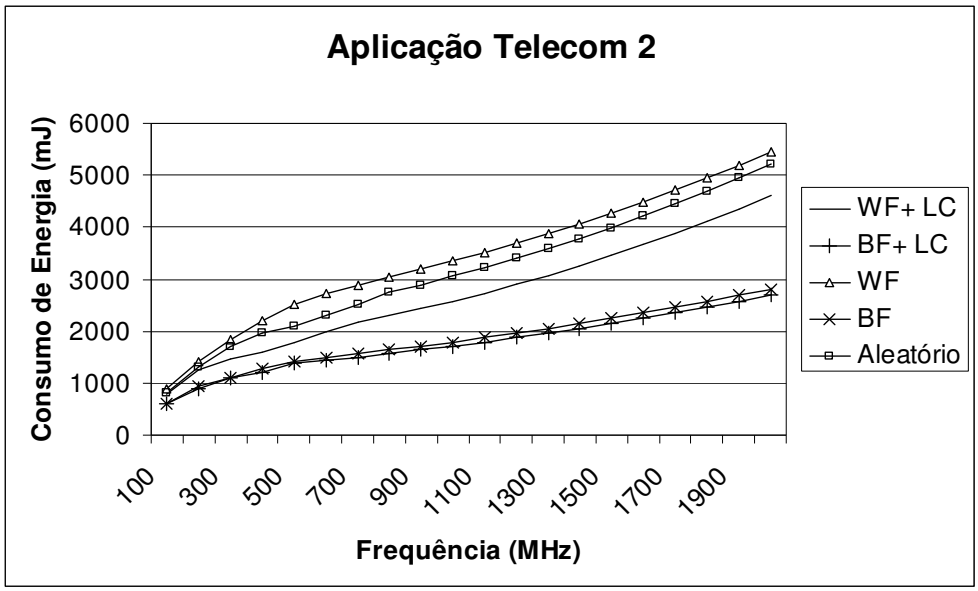


Figura 6.35: Gráfico Consumo de energia versus freqüência de operação dos processadores para cada um dos algoritmos bin-packing para a aplicação *Telecom 2*.

Por outro lado (Figura 6.35), os algoritmos que são combinados com LC apresentam menor consumo de energia que os outros algoritmos. Os algoritmos combinados com LC agrupam tarefas em clusters e, assim sendo, poucos processadores são utilizados. Diante do exposto, o número de processadores inativos aumenta e estes são desligados automaticamente pelo sistema. É notável que, quanto maior a freqüência, maior a vantagem dos algoritmos bin-packing combinados com LC sobre os algoritmos bin-packing convencionais.

A Tabela 6.3 apresenta os valores das freqüências mínimas para não haver perda de deadlines para todos os algoritmos empregados na Figura 6.34 e na Figura 6.35. Também são apresentados valores de consumo de energia para cada um dos algoritmos aplicados. Percebe-se que o algoritmo WF + LC, para eliminar o número de perdas de deadlines, deve ser executado à menor freqüência da Tabela 6.3, ou seja, a 900 MHz. Seu consumo de energia é o segundo menor da tabela. No entanto, o algoritmo BF + LC obteve menor consumo de energia, com uma freqüência de 1200 MHz. Mesmo com o aumento da freqüência de 33% em relação à freqüência de 900 MHz, não se gerou maior consumo de energia, pois o algoritmo BF+LC neste caso tornou alguns processadores inativos. Estes por sua vez são desligados para economizar energia.

Tabela 6.3: Freqüência mínima para não haver perda de deadlines para cada um dos algoritmos e seus respectivos consumos de energia.

Algoritmo	**Freqüência mínima para taxa de deadlines a 0,0% (MHz)**	**Consumo de energia (mJ)**
BF + LC	1200	1876,23
WF + LC	900	2432,87
BF	1900	2570,81
Aleatório	1900	4944,16
WF	2200	5861,34

6.8.3 Conclusões

Os resultados apresentados demonstram que, para cada algoritmo, existe uma freqüência diferente que resulta na eliminação de perda de deadlines. Em particular, no caso da aplicação *Telecom 2*, o algoritmo BF + LC, rodando a 1200 MHz, obteve menor consumo de energia e mesma eficiência que o algoritmo WF + LC rodando a 900 MHz. Neste caso, obviamente o algoritmo BF + LC é selecionado como o melhor algoritmo para distribuição de tarefas no sistema. Isto se deve ao fato do algoritmo BF + LC tornar processadores inativos, sem nenhuma tarefa, e estes por sua vez são desligados. Desta forma existe economia de energia no sistema.

Todavia, é provável que, se houver um número alto de clusters formados pelo algoritmo LC, no qual todos os processadores fiquem ativos, o algoritmo BF + LC pode não ser o melhor, pois o consumo de energia poderá ser equivalente ao consumo de energia proporcionado pelo algoritmo WF + LC.

6.9 Variação de carga em tempo de execução

6.9.1 Estratégias de Simulação

Neste experimento, foram realizadas várias simulações com diferentes tempos de execução do sistema para verificação a taxa de crescimento de perda de deadlines ao longo do tempo. Além disso, a carga do sistema variou ao longo execução da simulação, desalocando grafos da aplicação com e sem realocação.

Nestes experimentos, foram utilizadas aplicações: *telecom 2* e *app-synth*. Estas aplicações foram escolhidas por terem grande quantidade de comunicação entre as tarefas.

Os experimentos foram realizados em diversos tempos de simulação, variando de 10 ms a 5 minutos (300.000 ms). A freqüência dos processadores varia de 100 MHz a 600 MHz, e a tensão dos processadores varia entre 1.3 V a 2.0 V usando DVS. Para cada simulação, foram obtidos os valores de energia com o emprego de gerenciamento de energia. Os processadores que não tiveram tarefas alocadas são desligados para economia de energia. A rede-em-chip funciona a 266 MHz de freqüência e o tamanho da rede é 4x4 processadores. O tamanho de memória para cada processador é de 64KB.

6.9.2 Experimentos

A seguir, os experimentos estão divididos em duas subcategorias: tamanho do contexto convencional e modificado. A primeira categoria (tamanho de contexto convencional) utiliza os tamanhos dos contextos apresentados na Tabela 6.1. A segunda categoria (tamanho de contexto modificado) utiliza tamanhos de contexto 10 vezes maiores do que os apresentados na Tabela 6.1. Esta variação de contexto é interessante para apresentar dados dos custos de migração na variação de carga em tempo de execução. Na primeira categoria usaram-se dois estudos de caso: Telecom 2 e App-synth. No entanto, na segunda categoria, usou-se apenas a aplicação Telecom 2, para apresentação do impacto da variação do tamanho do contexto no sistema.

6.9.2.1 Tamanho do contexto convencional

A Figura 6.36 apresenta o custo da migração em termos de perda de deadlines para cada um dos algoritmos *bin-packing* para aplicação *Telecom 2* na inicialização do sistema. Na inicialização, tarefas são lidas da memória que está no processador mestre, o qual tem a função de monitorar a carga da rede-em-chip. Percebe-se que os algoritmos, em termos da ordem de eficiência, estão coerentes com os resultados apresentados na Seção 6.3. Como foi mostrado, WF + LC apresenta menor taxa de perda de deadlines que os outros algoritmos, em uma mesma freqüência de operação. Nota-se que, do tempo de 1 a 20 ms, existe um decréscimo da taxa de perda de deadlines. Quando é realizada a migração de tarefas, como por exemplo, alocação as tarefas para inicialização, várias tarefas perdem seus prazos de execução, e o número de perda de deadlines aumenta significativamente. Picos de taxas de perda de deadlines ocorrem em conseqüência da migração de tarefas. Ao decorrer do tempo, logo após a fase de migração, as tarefas conseguem recuperar seu estado de equilíbrio de acordo com a heurística como elas foram dispostas no sistema. Obviamente este pico inicial da taxa de perda de deadlines é maior quando se tem tarefas de contextos maiores. Estes resultados, quanto ao tamanho do contexto, podem ser visualizados na Seção 6.2.

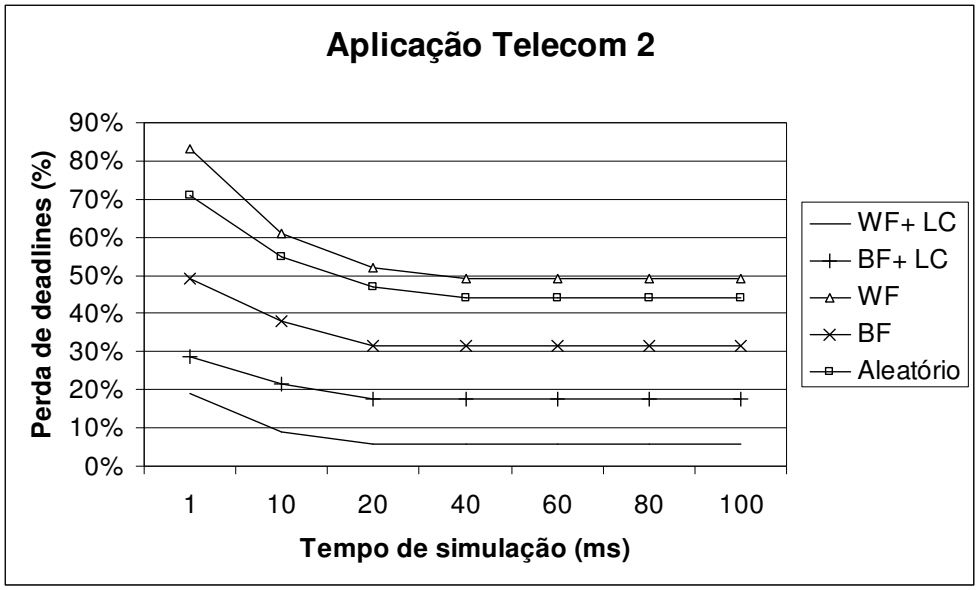


Figura 6.36: Custo da migração na inicialização do sistema em termos de perda de deadlines para cada um dos algoritmos bin-packing para a aplicação *Telecom 2*.

Para demonstração da viabilidade da migração de tarefas em um sistema embarcado baseado em redes-em-chip, foram realizados dois experimentos que variam a carga de ocupação dos processadores em tempo de execução. O primeiro experimento é baseado na aplicação Telecom 2, porém foram utilizados apenas 4 dos 8 grafos disponíveis da aplicação. O segundo experimento é baseado na aplicação *app-synth*, com quatro instâncias (grafos) da mesma. A Figura 6.37 apresenta um gráfico resultante, em termos de perda de deadlines, de várias simulações de tempos distintos na aplicação Telecom 2. Primeiramente, três conjuntos de simulações foram realizados. O primeiro deles é um conjunto de simulações da aplicação Telecom 2 com 4 grafos de tarefas. Neste primeiro conjunto, foram realizadas 15 simulações com diferentes tempos de simulação. Este conjunto de simulações foram realizados no tempo 0 ms ao tempo de 1000 ms. Neste tempo, inicialmente os 4 grafos de tarefas são alocados na rede-em-chip utilizando o algoritmo WF + LC para minimização da taxa de perda de deadlines. Percebe-se que existe um overhead inicial do período de 0 a 40 ms de simulação causado pela alocação inicial das tarefas. Todas as tarefas são originadas e migradas em um único processador, simulando a existência de um processador para distribuição das tarefas. O segundo conjunto de simulações é realizado no tempo de 1000 ms a 2000ms. Foram realizadas 6 simulações com tempos de execução dentro deste intervalo. No simulador, foram capturadas as posições originadas pelo WF + LC da primeira alocação realizada na primeira simulação do primeiro conjunto de simulações. Antes da execução do segundo conjunto de simulações, foram posicionadas a priori todas as tarefas nas posições originadas pela heurística. Desta forma, apenas executou-se as 6 simulações em um tempo de até 1000 ms. Os números que representam à perda de deadlines e o tempo de simulação do segundo conjunto de simulações obviamente são agregados aos valores da última simulação do primeiro conjunto de simulações para mostrar o cenário mais realista. Diante do exposto, foi possível emular o comportamento dinâmico das tarefas, como por exemplo, migração de um processador para o outro ou desalocação de tarefas.

Desta forma, a partir da execução do tempo maior de 1000 ms da aplicação Telecom 2 com 4 tarefas, houve a desalocação de um grafo da aplicação. A aplicação Telecom 2 é executada apenas com 3 dos seus grafos. A linha pontilhada na Figura 6.37 apresenta uma diminuição na taxa de perda de deadlines, justamente pela desalocação de um grafo da aplicação.

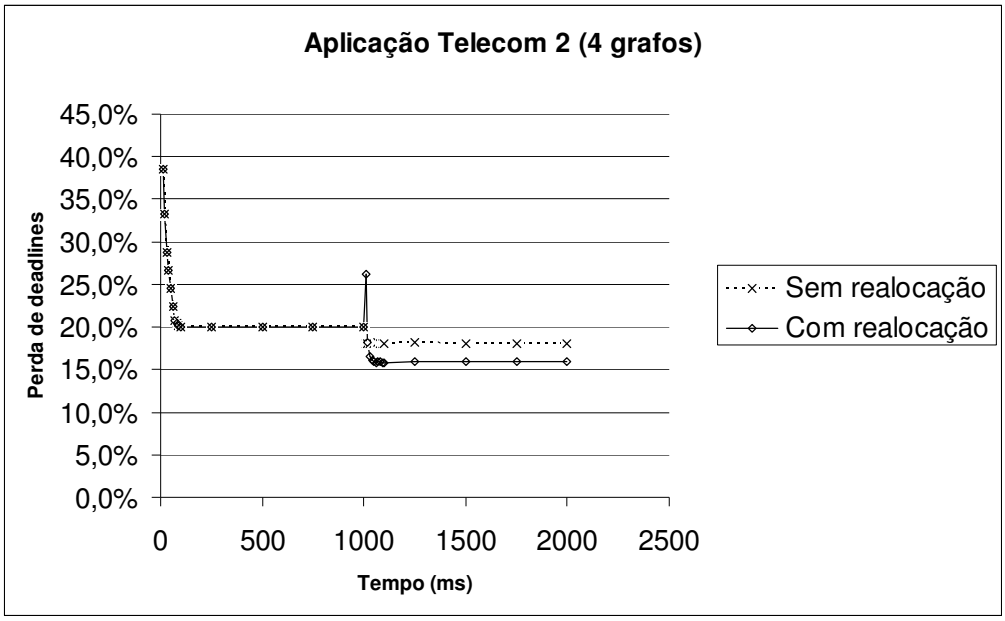


Figura 6.37: Gráfico Perda de deadlines versus Tempo de simulação que apresenta dois cenários para a aplicação *Telecom 2* onde um grafo da aplicação é desalocado no sistema: (i) sem realocação das tarefas remanescentes; (ii) com realocação das tarefas remanescentes.

No terceiro e último experimento, foi realizado um conjunto de simulações a partir do tempo de 1000 ms. As tarefas são alocadas inicialmente nas posições originais resultantes da simulação entre 0 e 1000 ms, de modo similar ao segundo experimento. No entanto, nos 3 grafos restantes, empregou-se o algoritmo WF + LC novamente de forma a simular uma realocação das tarefas que permaneceram depois da desalocação de um grafo da aplicação.

Na Figura 6.37, é possível observar os custos de migração ocorrido nos tempo de 0 a 40 ms e 1000 ms a 1040 ms. A primeira migração (inicialização do sistema) resultou em um pico de 38,5% da taxa de perda de deadlines. No entanto, a segunda migração resultou em um pico de 26,2% da taxa de perda de deadlines, pois existem menos tarefas para serem migradas, e algumas tarefas poderão permanecer nos processadores onde estavam alocadas antes da realocação. No tempo de 50 ms depois da segunda migração (simulação da realocação das três aplicações restantes) de tarefas, a taxa de perda de deadlines passou de 18,1% para 15,9% comparada ao segundo experimento que consiste em simular as aplicações restantes sem realocação. A realocação possibilitou a aproximação de clusters que têm razoável quantidade de comunicação. Quando um grafo foi desalocado, alguns processadores ficaram desocupados, e estes possivelmente estavam entre dois clusters que se comunicavam. A realocação então reposicionou os clusters para se estabelecerem em processadores próximos, reduzindo o número de *hops* de distância, e consequentemente diminuindo o número de perda de deadlines. A ilustração dos três cenários é apresentada no Apêndice B.

A Figura 6.38 apresenta um gráfico consumo de energia versus tempo de execução para os mesmos cenários apresentados na Figura 6.37. Percebe-se que existem duas etapas em que os custos de migração aparecem em destaque: (i) na primeira migração que consiste na inicialização do sistema; (ii) na segunda migração, que consiste na

realocação das tarefas restantes. Nota-se também que, em 2000 ms de tempo de simulação do estudo de caso Telecom 2, já se observa uma pequena redução do consumo de energia da abordagem com realocação comparado à abordagem sem realocação.

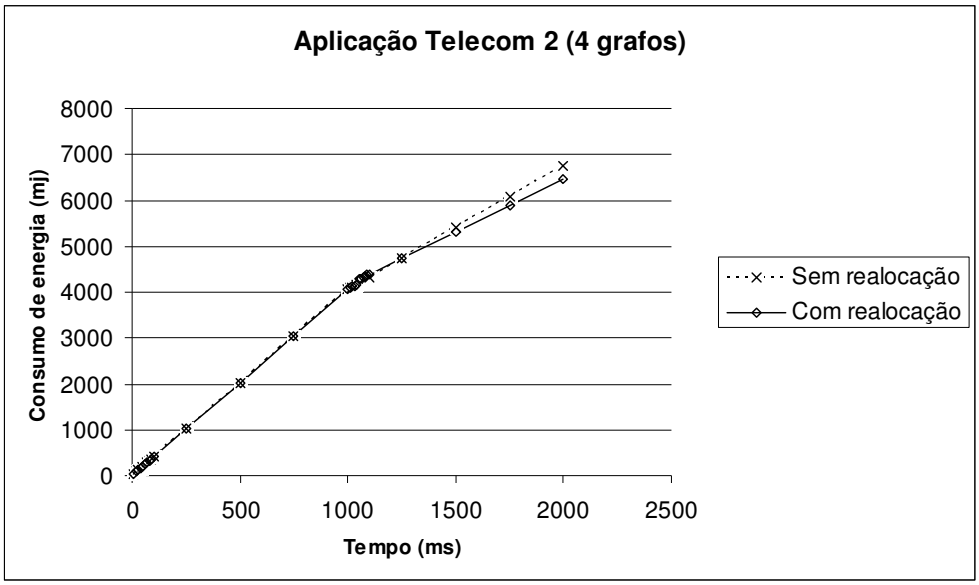


Figura 6.38: Gráfico Consumo de energia versus Tempo de simulação nos dois cenários apresentados na Figura 6.37.

As tarefas com alta taxa de comunicação são alocadas nos mesmos processadores, a comunicação da rede é reduzida e consequentemente o consumo de energia também é reduzido. Depois da desalocação, tarefas ficam espalhadas na rede. Com o emprego do WF + LC, estas tarefas podem ser alocadas em um mesmo processador, resultando em processadores inativos, aptos para serem desligados com a aplicação do gerenciamento de energia.

A Figura 6.39 apresenta uma ampliação do custo de migração da Figura 6.38 entre os intervalos de tempo 950 ms e 1500 ms. O gráfico apresenta o momento antes das curvas se cruzarem (custo de migração) e o momento depois do cruzamento destas (ganho obtido pela migração). A Figura 6.39 também apresenta a curva de amortização, ou seja, para amortizar o custo da migração, deve-se executar o sistema por pelo menos 242 ms depois da realocação. A partir deste tempo de execução, a migração de tarefas apresenta redução do consumo de energia em relação à desalocação do grafo sem realocação. Além disso, este tempo garante que a migração de tarefas não degrade o sistema.

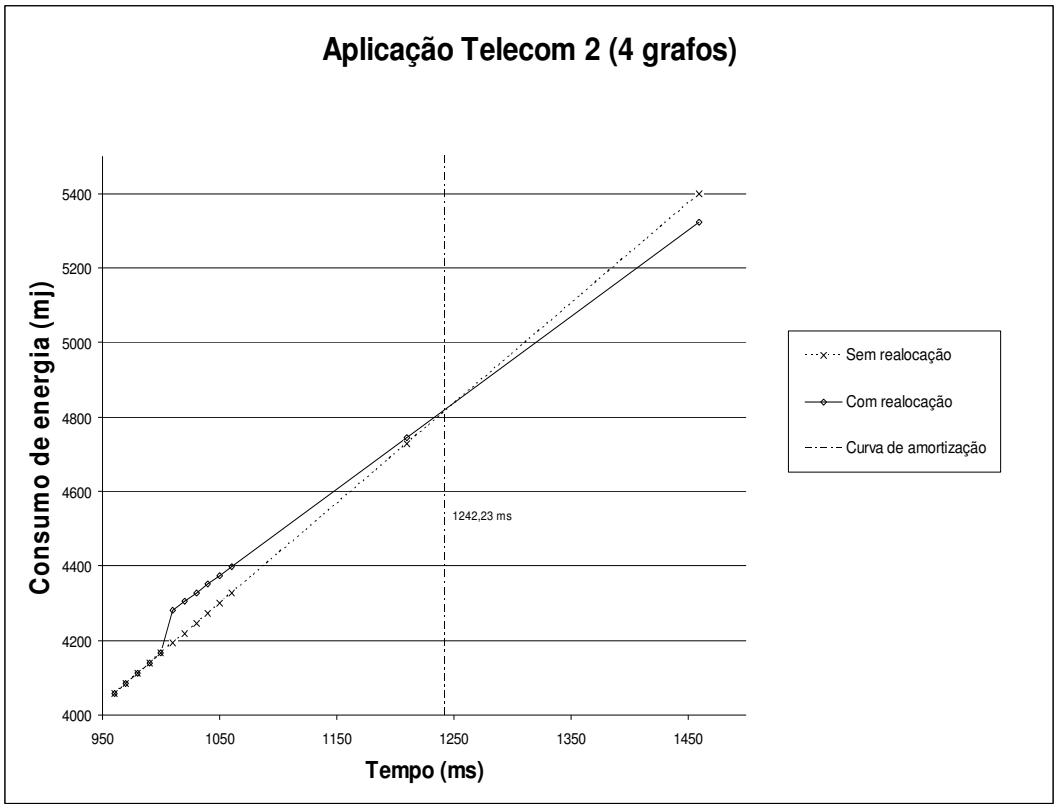


Figura 6.39: Ampliação da Figura 6.38 do custo de migração e curva de amortização entre os tempos de execução 950 ms a 1500 ms.

A Figura 6.40 apresenta uma estimativa do tempo de execução de 5 minutos dos mesmos cenários apresentados na Figura 6.38. Este tempo de execução considerou-se viável em um sistema embarcado atual (ex. celular, *palms*). Nota-se que, quanto maior o tempo de execução, maior será a economia do consumo de energia, comparando a abordagem com realocação com a abordagem sem realocação. Em 5 minutos de execução, a realocação permitiu a redução do consumo de energia de 800 joules para 690 joules, ou seja, uma redução de 14%. Isto significa que para a execução do sistema com a abordagem com realocação alcançar, por exemplo, o mesmo consumo de energia da abordagem sem realocação, de 800 joules, a abordagem com realocação terá que executar em um tempo de 347 s ou 5 minutos e 50 segundos. Isto significa que a realocação conseguiu um ganho de 50 segundos na execução do sistema. Este resultado é bastante significativo, pois baterias de celulares permanecem dias em operação. A abordagem com realocação possibilita o prolongamento do uso da bateria sem eventual recarga.

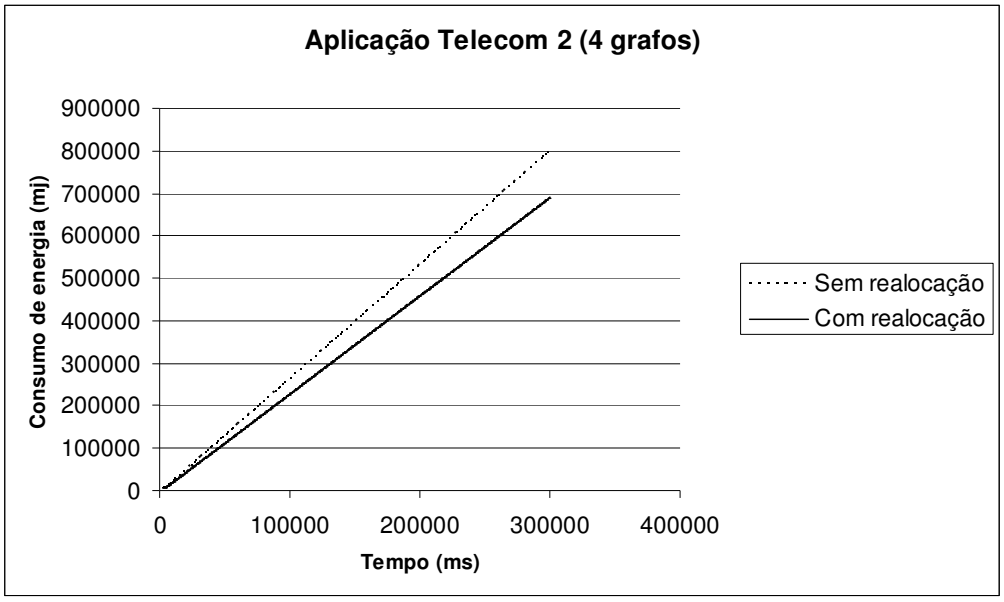


Figura 6.40: Gráfico Consumo de energia versus Tempo de simulação nos dois cenários de aplicação apresentados na Figura 6.37 em uma estimativa para o tempo de simulação de 300.000 ms (5 minutos).

Outro experimento realizado é similar ao experimento anterior, este apresentado na Figura 6.37. No entanto, foi utilizada como estudo de caso a aplicação *App-synth*, a qual tem como característica alta taxa de comunicação entre as tarefas (cerca de 50 vezes mais volume de comunicação que a aplicação *Telecom 2*). Este experimento, na verdade é um conjunto de cinco experimentos. Cada experimento é descrito a seguir:

1. *Experimento 1*: Similar ao experimento realizado com a aplicação Telecom 2, alocou-se 4 grafos que correspondem 4 instâncias da aplicação *App-synth*. Simularam-se esses 4 grafos por 2 segundos e obtiveram-se os resultados em termos de consumo de energia e perda de deadlines.
2. *Experimento 2*: Desalocou-se uma instância da aplicação *App-synth* e simulou-se 3 grafos restantes por 2 segundos. Preservou-se a posição original das tarefas para a simulação.
3. *Experimento 3*: Desalocou-se uma segunda instância da aplicação *App-synth* e simulou-se 2 grafos restantes por mais 2 segundos. Preservou-se a posição original das tarefas para a simulação.
4. *Experimento 4*: É similar ao experimento 2, porém foi empregado inicialmente o algoritmo WF +LC para simular a realocação e minimizar o número de perda de deadline e consumo de energia em relação ao experimento 2.
5. *Experimento 5*: Preserva-se o posicionamento do experimento 4 e emprega-se novamente a heurística WF+ LC.

Como pode ser observado no cenário apresentado na Figura 6.41, novamente a realocação nos dois momentos (experimento 4 e experimento 5) demonstrou ser eficiente em termos de redução de perda de deadlines. No primeiro momento (tempo de simulação apresentado na Figura 6.41 de 2000 ms a 4000 ms), a migração proveu a

redução de perda de deadlines de 38% para 24%, ou seja, houve uma redução de 37%. Na segunda realocação (tempo de 4000 ms a 6000 ms), a migração proveu novamente a redução de perda de deadlines de 26% para 20%, ou seja, uma redução de 23%. Nota-se que a taxa de redução de perda de deadlines na primeira migração é maior que a segunda, pois as tarefas remanescentes da primeira realocação têm maior volume de comunicação que as tarefas remanescentes da segunda realocação. O algoritmo WF + LC consegue ser mais eficiente na medida em que o volume de comunicação entre as tarefas aumenta. A realocação, além de aproximar clusters que têm razoável volume de comunicação, diminuindo o número de *hops* de distância, permite que clusters que estavam em processadores sobrecarregados (acima de 100% da utilização do processador é inevitável a perda de deadlines) seriam realocados em processadores com cargas de processamento mais leves, ocasionando menor número de perda de deadlines.

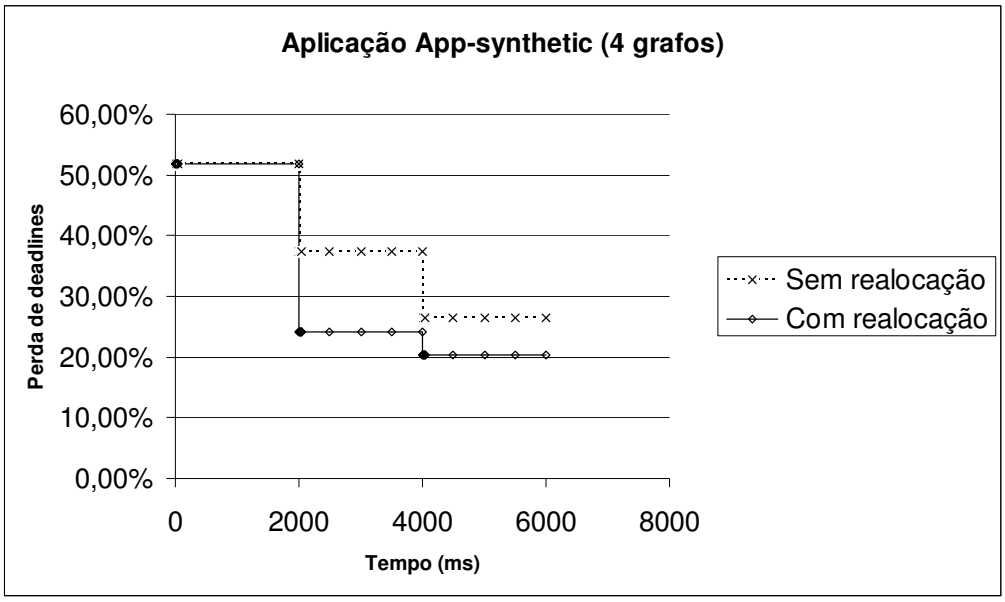


Figura 6.41: Gráfico Perda de deadlines versus Tempo de simulação para o cenário da *App-synth*.

Na Figura 6.41, não foram inseridas as curvas dos overheads das migrações, pelo grande número de experimentos isolados. No entanto, obtiveram-se os valores dos picos das taxas de perdas de deadlines para a alocação de inicialização (tempo 0 ms), primeira realocação (tempo 2000 ms) e segunda realocação (4000 ms) equivalentes a 100%, 73% e 45% respectivamente.

A Figura 6.42 apresenta um gráfico do consumo de energia versus tempo de execução para os mesmos cenários apresentados na Figura 6.41. A curva pontilhada representa o consumo de energia para abordagem sem realocação das tarefas restantes. A curva em linha contínua apresenta o consumo de energia em relação ao tempo para abordagem com realocação de tarefas. Percebe-se que existem duas suaves inclinações na curva (uma aos 2000 ms e outra aos 4000 ms), causadas pela melhor realocação das tarefas, que conseguiu amortizar o custo das realocações, e desta forma obteve menores consumos de energia, em relação à abordagem sem realocação. Em 6 segundos de execução do sistema, percebe-se que, na Figura 6.42, a abordagem com realocação obteve uma pequena, porém significativa redução do consumo de energia em comparação com a abordagem sem realocação.

Para amortizar o custo da migração no sistema executando as aplicações *App-synth*, deve-se executar o sistema por pelo menos 375 ms depois da primeira realocação e 334 ms após a segunda migração. Estes tempos garantem que a migração de tarefas não degrade o sistema.

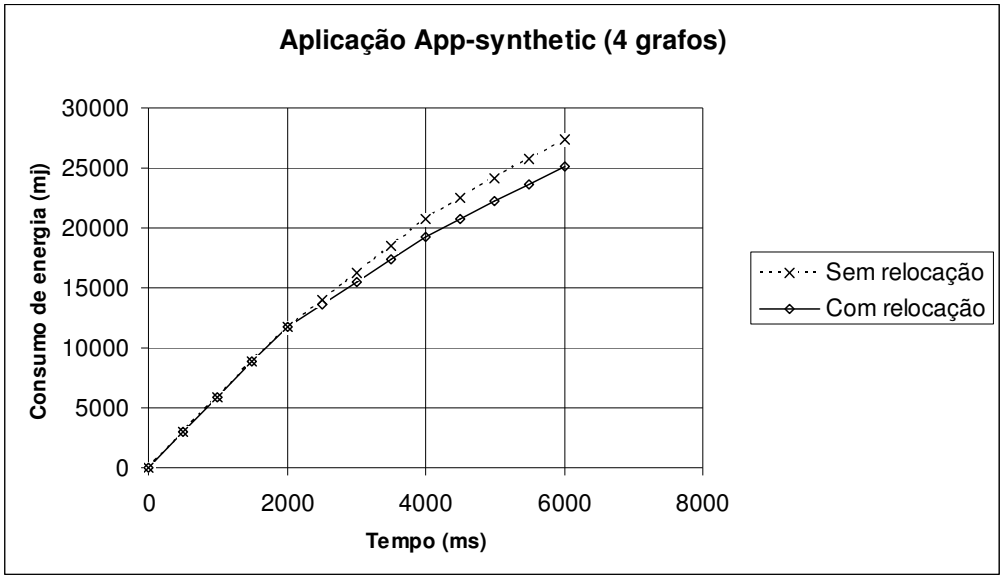


Figura 6.42: Consumo de energia versus Tempo de simulação para o cenário da *App-synth*.

A Figura 6.43 apresenta uma estimativa do tempo de execução de 5 minutos dos mesmos cenários apresentados na Figura 6.42. Em 5 minutos de execução, a realocação permitiu a redução do consumo de energia de 1005 joules para 838 joules, ou seja, uma redução de 17%. Isto significa que, para a execução do sistema com a abordagem com realocação alcançar, por exemplo, o mesmo consumo de energia da abordagem sem realocação, de 838 joules, a abordagem com realocação terá que executar em um tempo de 365 s ou 6 minutos e 5 segundos. Isto significa que a realocação conseguiu um ganho de 1 minuto e 5 segundos na execução do sistema.

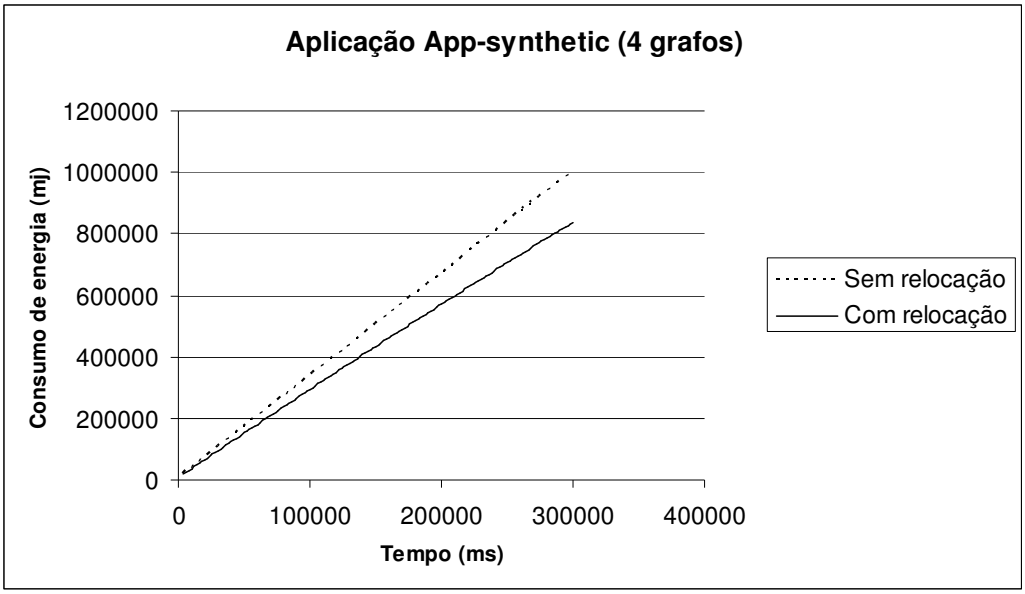


Figura 6.43: Gráfico Consumo de energia versus Tempo de simulação para o cenário apresentado na Figura 6.41 em uma estimativa para o tempo de simulação de 300.000 ms (5 minutos).

6.9.2.2 Tamanho do contexto das tarefas modificado

A Figura 6.44 apresenta um gráfico resultante em termos de perda de deadlines de várias simulações de tempos distintos na aplicação *Telecom 2* do mesmo cenário apresentado na Figura 6.37. No entanto, o tamanho do contexto das tarefas deste experimento é 10 vezes maior que na aplicação *Telecom 2* convencional.

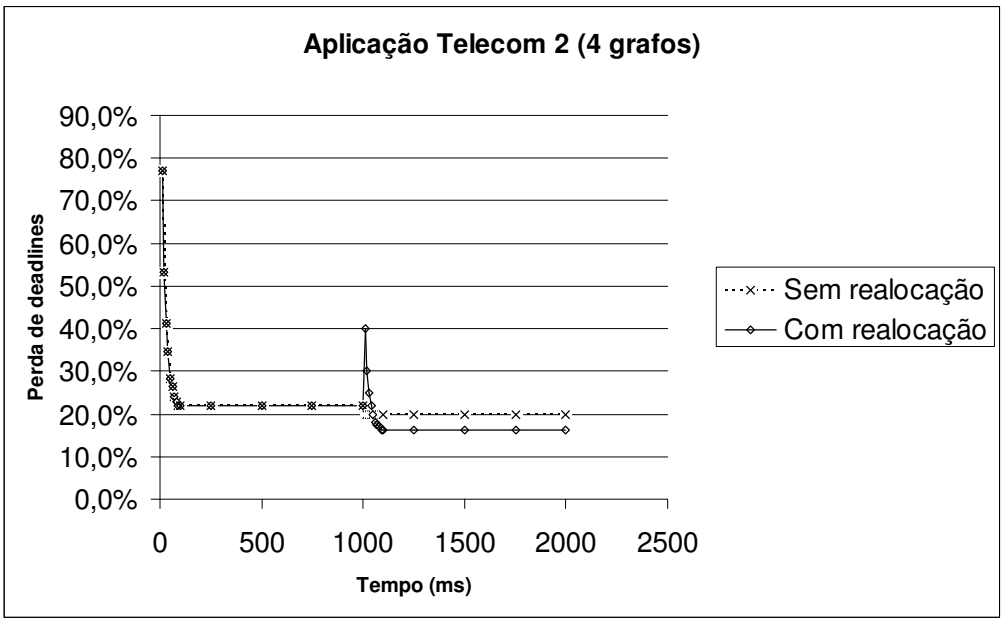


Figura 6.44: Gráfico Perda de deadlines versus Tempo de simulação da aplicação *Telecom 2* com tamanho do contexto modificado.

Como era de se esperar, a Figura 6.44 apresenta custos de migração em termos de perda de deadlines maiores que os custos mostrados na Figura 6.37. Isto obviamente deve-se à modificação do tamanho do contexto das tarefas. Quanto maior o contexto das tarefas, maior volume de comunicação interna na rede-em-chip é gerado, e consequentemente maiores serão as perdas de deadlines. Os picos da taxa de perda de deadlines da migração de tarefas no momento da inicialização do sistema e da realocação das tarefas são 76,9% e 40% respectivamente. Logo após a segunda migração de tarefas, a taxa de perda de deadlines passou de 20,1% para 16,4% comparada ao segundo experimento que consiste em simular as aplicações restantes sem realocação.

A Figura 6.45 apresenta o custo de migração entre os intervalos de tempo 950 ms e 1700 ms, método similar utilizado para obtenção dos resultados apresentados na Figura 6.39. O tamanho do contexto das tarefas é maior que na aplicação Telecom 2 convencional, e consequentemente, o custo de migração é maior. Assim, o tempo de amortização dos custos de migração é maior que o tempo de amortização apresentado na Figura 6.39. Para a consumação da amortização, deve-se executar o sistema por pelo menos 412 ms depois da realocação (contra 242 ms no caso da Figura 6.39 com contextos menores). A partir deste tempo de execução, a migração de tarefas apresenta redução do consumo de energia em relação à desalocação do grafo sem realocação. Além disso, este tempo garante que a migração de tarefas não degrade o sistema.

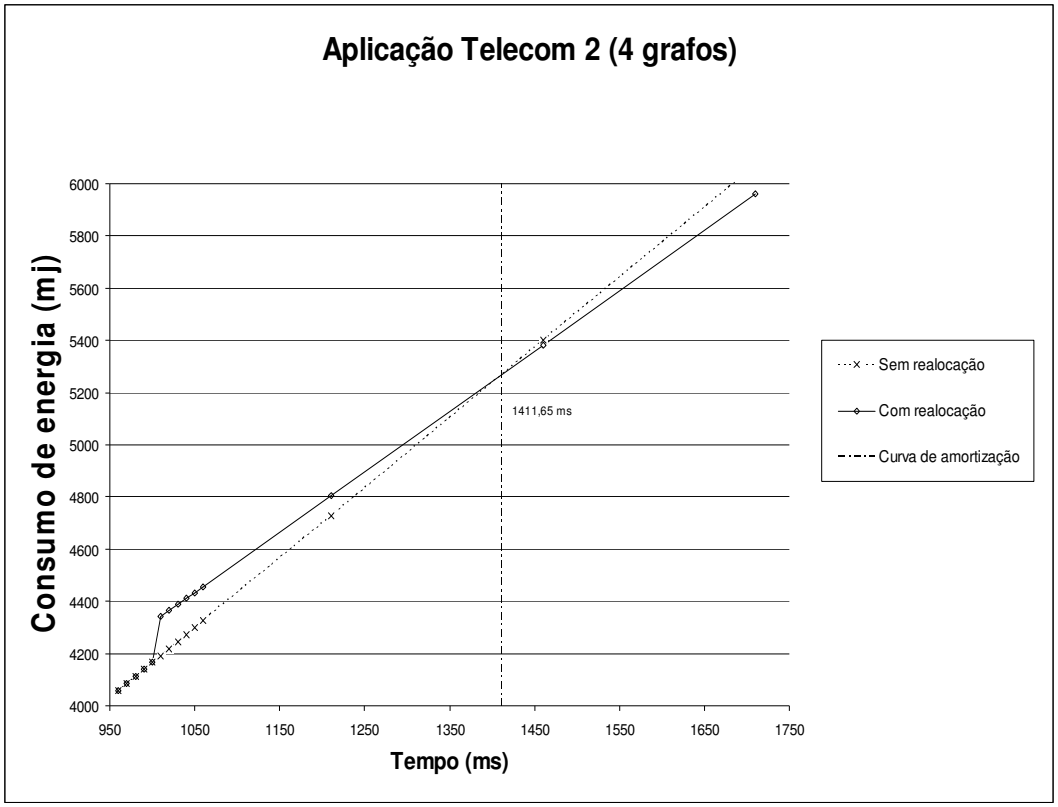


Figura 6.45: Ampliação do custo de migração e curva de amortização entre os tempos de execução 950 ms a 1700 ms da aplicação *Telecom 2* com tamanho do contexto modificado.

Para finalizar, foi realizada uma estimativa do tempo de execução de 5 minutos da aplicação *Telecom 2* com tamanho de contexto modificado similar à Figura 6.40. Os resultados são muito similares à Figura 6.40 e, por esta razão, não houve necessidade de apresentar esses resultados aqui nesta Seção.

6.9.3 Conclusões

Os resultados apresentados nesta Seção demonstraram que a realocação das tarefas restantes, após uma desalocação de um grafo ou subconjunto da aplicação, reduz de maneira significativa tanto o consumo de energia quanto a taxa de perda de deadlines. Esta redução é atingida somente quando o sistema é executado em um tempo suficiente para amortizar o custo de migração. Se a freqüência de migração for maior que o tempo de amortização de cada realocação, o sistema poderá se degradar, aumentando excessivamente a taxa de perdas de deadline e consumo de energia. A taxa de redução, tanto no consumo de energia como no número de perda de deadlines, é maior em aplicações que têm maior volume de comunicação, pois o algoritmo WF + LC contempla a comunicação entre as tarefas.

Outra observação é a respeito do ganho considerando o consumo de energia. Em poucos segundos de execução, a taxa de redução do consumo de energia não é significativa na comparação da abordagem com realocação com a abordagem sem realocação. No entanto, como foi apresentado nos resultados na Figura 6.40 e na Figura 6.43, em um tempo de 5 minutos a taxa de redução do consumo de energia da abordagem com realocação se torna significativa. Como foi mencionado anteriormente, quanto maior o tempo de execução, maior será a economia do consumo de energia, na comparação da abordagem com realocação com a abordagem sem realocação. Estes resultados são significativos, visto que baterias de celulares, por exemplo, permanecem dias em operação. A abordagem com realocação possibilita o prolongamento do uso da bateria sem eventual recarga.

Finalizando, indiscutivelmente o tamanho do contexto das tarefas tem um impacto decisivo nos custos de migração e consequentemente no tempo de amortização. No que diz respeito à taxa de perda de deadlines, quanto maior o tamanho do contexto, maiores serão os picos de perdas de deadlines no sistema. Isto significa que a técnica de migração de tarefas pode ser aplicada em um sistema de tempo-real hard, desde que existam obrigatoriamente mecanismos que garantam a taxa de perda de deadlines nula. Por outro lado, no que diz respeito ao consumo de energia, quanto maior o contexto, maior será o custo de migração e maior será o tempo de amortização.

6.10 Avaliação do mecanismo de gerenciamento e alocação

6.10.1 Estratégias de Simulação

Para avaliação do mecanismo de gerenciamento e alocação, o qual é executado em um ou mais núcleos mestres, foram realizados diversos experimentos para estimar o impacto desses núcleos no sistema. Nestes experimentos, não se priorizou adicionar núcleos no sistema, e sim utilizar aqueles que já constituíam a *mesh*. Por conseguinte, exemplificando, uma rede-em-chip de tamanho 4x4, com um único núcleo mestre, e outra rede-em-chip do mesmo tamanho, porém com quatro núcleos mestres, terão exatamente o mesmo número de PEs. A adição de um núcleo mestre implica na substituição de um núcleo escravo por um mestre. O sistema sem nenhum núcleo mestre

permite a alocação de tarefas em todos os núcleos da rede-em-chip. Neste caso, um núcleo de referência deve ser adotado, pois as tarefas são originadas deste núcleo, podem ser alocadas neste com custo zero de comunicação ou então ser migradas para os núcleos remanescentes. Esta última configuração foi utilizada em todos os experimentos anteriores apresentados neste Capítulo. Os núcleos mestres foram configurados sem a utilização do mecanismo de DVS a uma freqüência de 600 MHz para execução rápida do algoritmo *bin-packing*. O núcleo mestre é dedicado apenas para execução das heurísticas de alocação. As aplicações foram distribuídas de maneira homogênea em cada um dos mestres (no caso da rede-em-chip estar configurada para utilização de mais de um núcleo mestre). Foram realizados 12 experimentos para cada aplicação, ou seja, quatro configurações de núcleos mestres para três tamanhos distintos de redes-em-chip (4x4, 5x5 e 6x6). Foi utilizado o algoritmo WF + LC, pois este obteve menor perda de deadlines no sistema configurado com uma mesma freqüência de operação. Foram utilizados dois estudos de caso: uma instância da aplicação *Telecom* e quatro instâncias da aplicação *App-synth*.

Os núcleos mestres foram posicionados conforme a Figura 6.46 para a rede-em-chip de tamanho 6x6.

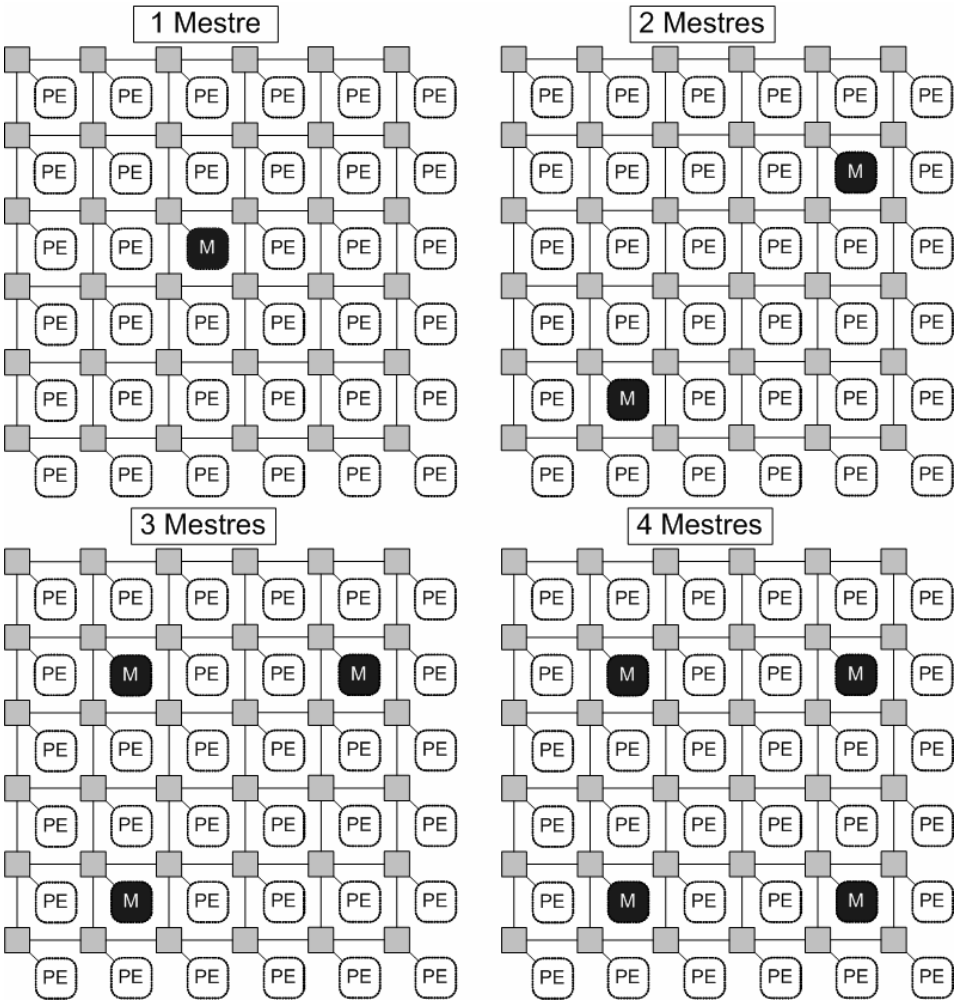


Figura 6.46: Posicionamento dos núcleos mestres para a configuração do tamanho de 6x6 da rede-em-chip.

O posicionamento dos núcleos mestres para as outras configurações de tamanho foi realizado obedecendo ao mesmo padrão ilustrado na Figura 6.46. Escolheu-se este padrão de posicionamento porque este permitiu uma distribuição homogênea dos núcleos mestres, minimizando a latência média e o número de *hops* para migrar uma determinada tarefa para um núcleo.

Quanto à configuração dos processadores, a freqüência de operação varia de 100 MHz a 600 MHz, e a tensão dos processadores varia entre 1.3 V a 2.0 V usando DVS. Os processadores que não tiveram tarefas alocadas são desligados, para economia de energia. O tamanho de memória para cada processador é de 64KB. O tempo de simulação de cada um dos experimentos é de 200 ms.

6.10.2 Experimentos

6.10.2.1 Aplicação Telecom

A Figura 6.47 apresenta o consumo de energia total nas diferentes configurações mencionadas na Seção anterior. O algoritmo WF + LC gerou 17 grupos (*clusters*) das 30 tarefas da aplicação *Telecom*. Desta forma, o desligamento de núcleos só ocorre quando os tamanhos da rede forem 5x5 (25 núcleos) e 6x6 (36 núcleos). O consumo de energia aumenta com a inclusão de um novo núcleo mestre no sistema.

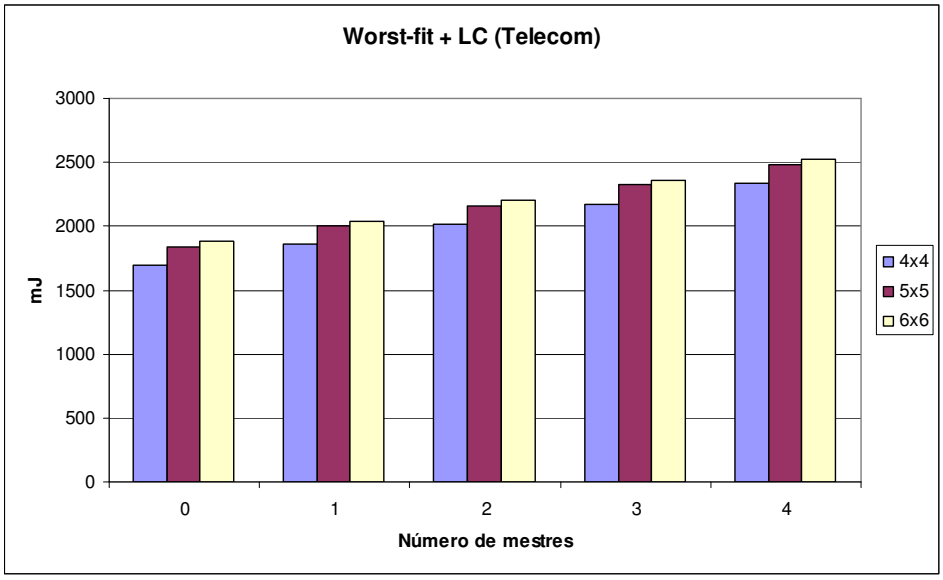


Figura 6.47: Consumo de energia total do sistema (*Telecom*).

A Figura 6.48 apresenta a taxa de acréscimo do custo energético de inclusão dos núcleos mestres no sistema. A rede-em-chip de tamanho 4x4 apresenta maiores taxas, pois o impacto energético da inclusão dos núcleos é maior que nos demais tamanhos da rede. Quanto maior o número de mestres adicionados no sistema, maior será o impacto energético. O impacto energético poderia ser menor nas redes de tamanho 5x5 e 6x6 caso os núcleos com utilização de 0% não fossem desligados.

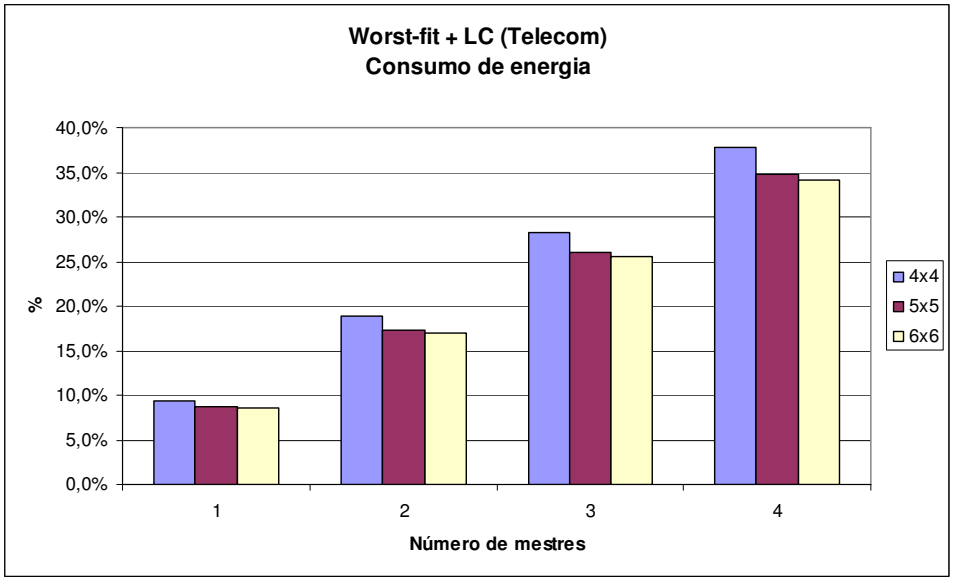


Figura 6.48: Taxa do acréscimo do custo energético na inclusão dos núcleos mestres no sistema (*Telecom*).

Finalmente, a Figura 6.49 apresenta as taxas de perda de deadlines para cada uma das configurações. A rede-em-chip de tamanho 4x4 apresentou um crescente aumento da taxa de perdas de deadlines, de acordo com o número de núcleos mestres. Apesar de haver mais núcleos mestres que possam distribuir/migrar as tarefas com um tempo menor que o sistema configurado apenas com um núcleo mestre, a taxa de deadlines aumentou pelo fato dos processadores que foram substituídos pelos núcleos mestres não poderem mais alocar as tarefas da aplicação. Desta forma, algumas tarefas excederam a capacidade de alguns processadores, com a inclusão de cada novo núcleo mestre.

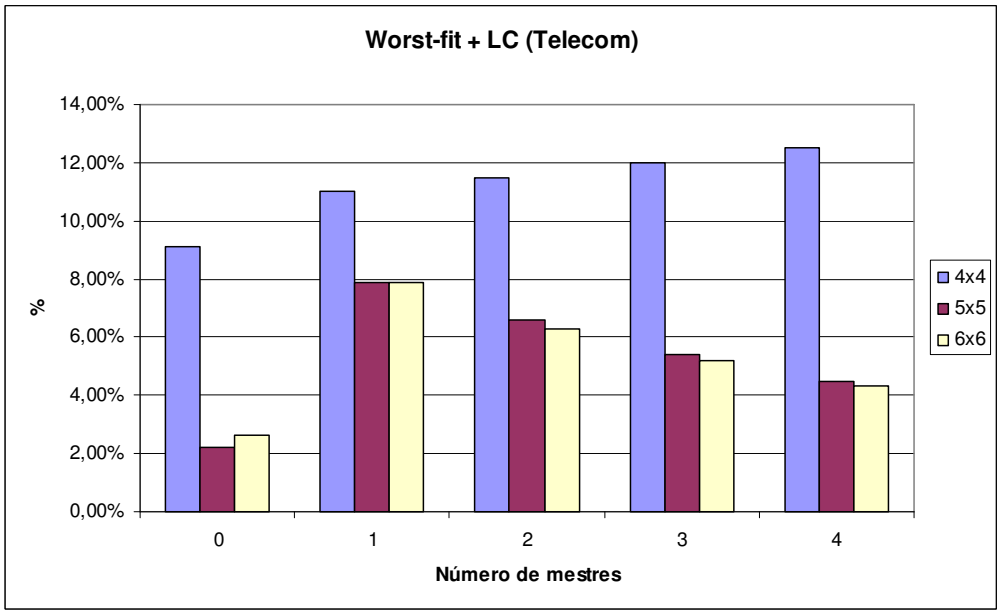


Figura 6.49: Taxa de perda de deadlines (*Telecom*).

Por outro lado, para tamanhos de rede-em-chip 5x5 e 6x6, quando se aumenta o número de núcleos mestres, ocorre à diminuição das taxas de perda de deadlines devido à redução do tempo de migração. Apesar do aumento das substituições de núcleos escravos por núcleos mestres, maior número de núcleos mestres proporciona maior número de migrações em paralelo, e desta forma o tempo de migração total é reduzido, salvo quando o número de núcleos mestres for excessivamente alto, em relação ao tamanho da *mesh*.

6.10.2.2 Aplicação App-synth

A Figura 6.50 apresenta o consumo de energia total nas diferentes configurações da rede-em-chip em termos de tamanho da rede e número de núcleos mestres. O algoritmo WF + LC gerou 36 grupos (*clusters*) das 44 tarefas da aplicação *App-synth*. Diante do exposto, não há desligamento de nenhum núcleo em todas as configurações de tamanho apresentadas nesta Seção, pois todos os núcleos terão pelo menos uma tarefa em execução. Inequivocamente, o consumo de energia aumenta sensivelmente com a inclusão de um novo núcleo mestre no sistema, pois neste último não há o mecanismo de DVS. Desta forma, seu consumo de energia é maior que o consumo de um núcleo escravo em torno de 30%.

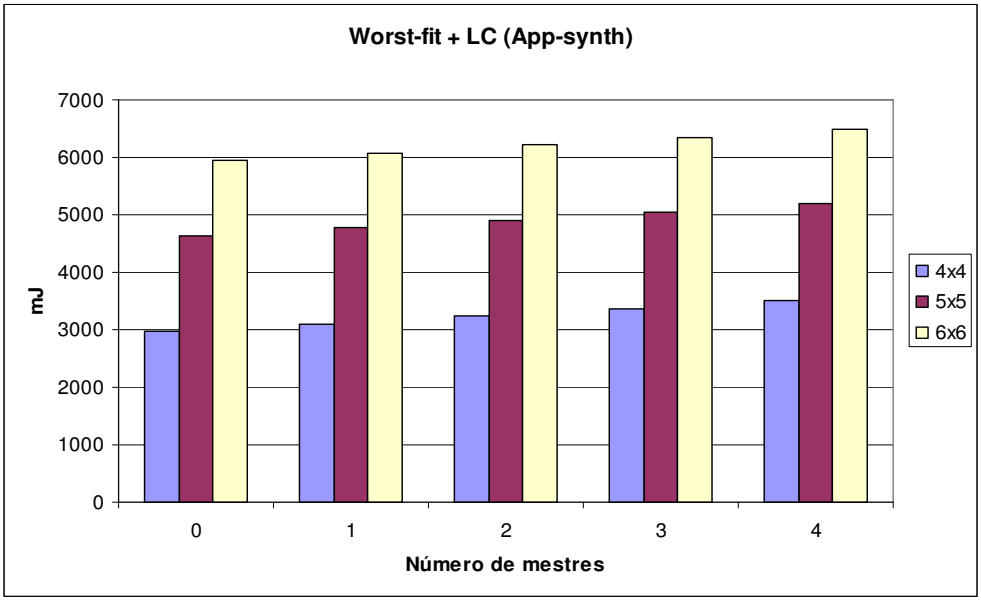


Figura 6.50: Consumo de energia total do sistema (*App-synth*).

A Figura 6.51 apresenta a taxa de acréscimo do custo energético da inclusão dos núcleos mestres no sistema em relação ao sistema sem núcleos mestres. Como era de se esperar, a configuração de tamanho 4x4 apresenta a maior taxa de acréscimo do custo energético. Quanto maior o tamanho da rede-em-chip, menor o impacto energético dos núcleos mestres.

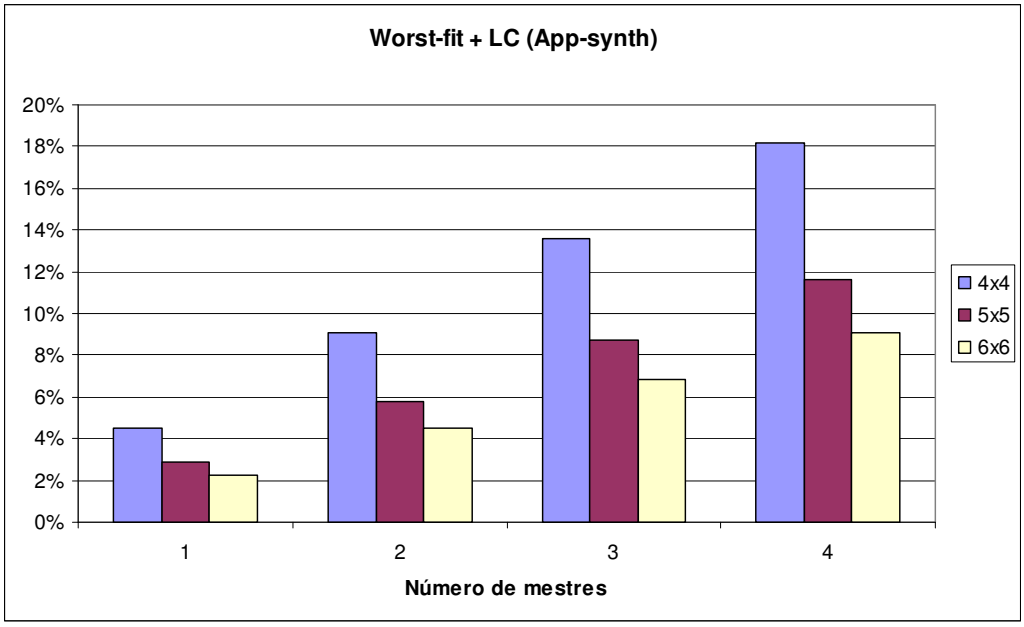


Figura 6.51: Taxa do acréscimo do custo energético na inclusão dos núcleos mestres no sistema (*App-synth*).

Finalmente, a Figura 6.52 apresenta a taxa de perda de deadlines para cada uma das configurações de tamanho da rede-em-chip e número de núcleos mestres. A taxa de perda de deadlines aumenta gradativamente, na configuração de tamanho 4x4. A cada inserção dos núcleos mestres, o tempo de migração total é reduzido, porém, pela rede 4x4 já estar saturada (todos os processadores com utilização perto dos 100%), qualquer adição de um novo núcleo mestre aumentará a taxa de perda de deadlines. Por outro lado, a configuração 5x5 permitiu uma redução de perda de deadlines com a adição de dois núcleos mestres. Mesmo com a adição de dois núcleos mestres, a utilização das tarefas não extrapolou o limite de 100% nos processadores. Esta redução se deve pela diminuição do tempo de migração. Por outro lado, para três e quatro núcleos mestres no sistema, a taxa de perdas de deadlines aumenta, pelo aumento da utilização dos processadores. Neste caso, alguns processadores excederam os 100% e consequentemente aumentou o número de perda de deadlines.

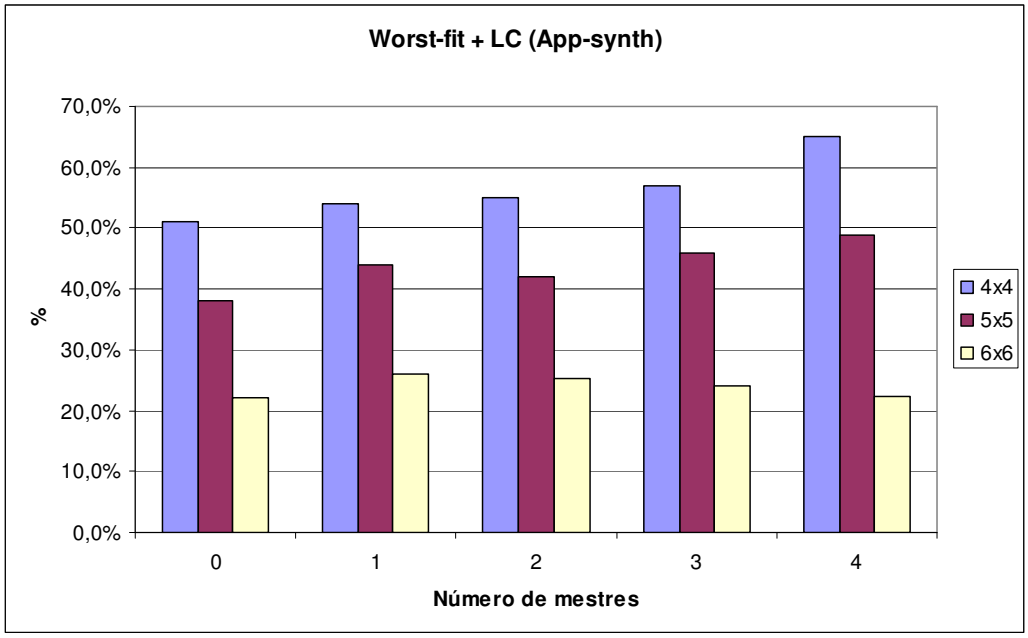


Figura 6.52: Taxa de perda de deadlines (*App-synth*).

Na configuração do tamanho de rede-em-chip 6x6, o aumento do número de núcleos mestres proporcionou redução da taxa de perda de deadlines. O tempo de migração foi reduzido com a adição de núcleos mestres. Mesmo com a substituição de núcleos escravos por núcleos mestres, a utilização dos processadores não ultrapassou os 100%, ocasionando redução da taxa de perda de deadlines.

6.10.2.3 Custo das mensagens de gerenciamento

A Figura 6.53 apresenta os custos de comunicação do mecanismo de gerenciamento em relação ao número de tarefas e ao número de *hops*.

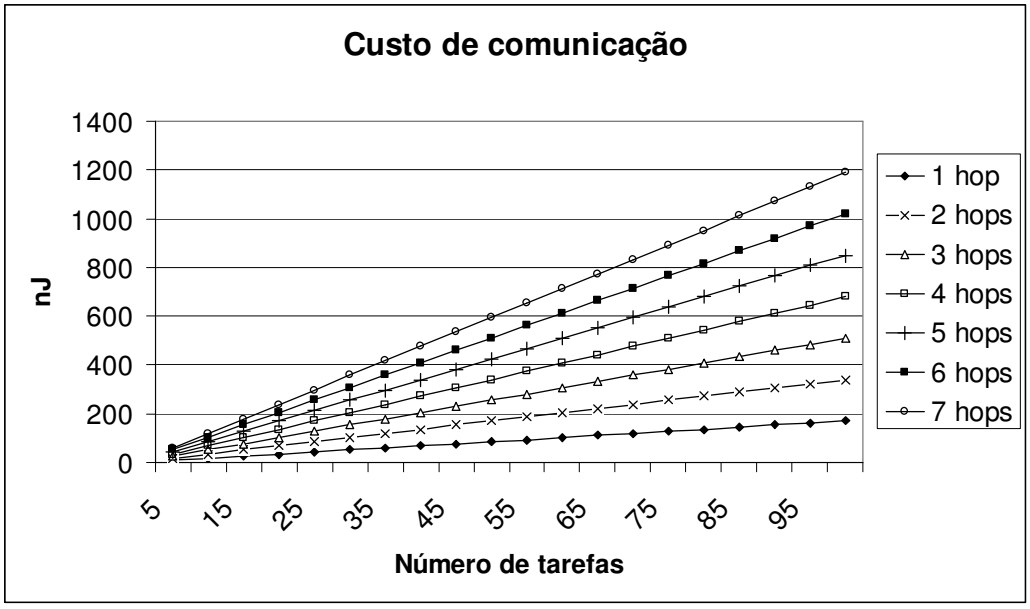


Figura 6.53: Custo energético de comunicação das mensagens de gerenciamento.

Logicamente, quanto maior o número de tarefas, maior o número de mensagens, e consequentemente maior o consumo de energia. Quanto maior o número de *hops*, maior o consumo de energia agregado ao número de mensagens relacionadas à quantidade de tarefas. Cada tarefa gera mensagens com o custo de 1,7 nJ em 1 *hop* de distância do núcleo escravo ao núcleo mestre.

6.10.3 Conclusões

Os resultados apresentados na Seção anterior demonstram que existem dois cenários que devem ser avaliados no que diz respeito à adição de núcleos mestres: o primeiro cenário é a alocação de tarefas que não ocupam todos os processadores; e o segundo cenário é a alocação de tarefas que ocupam todos os processadores na rede-em-chip. No primeiro cenário, o impacto energético dos núcleos mestres é maior pelo fato dos processadores inativos serem desligados. Desta forma, o consumo total de energia é reduzido e o impacto dos núcleos mestres tende a ser maior que na abordagem sem desligamento de núcleos inativos. No que diz respeito à perda de deadlines, esta tende a diminuir com a inserção de um novo núcleo mestre, desde que a rede-em-chip não esteja com todos os processadores utilizados. Desta forma, a inserção de um novo núcleo reduzirá o tempo de migração e não comprometerá a utilização dos processadores escravos. Por outro lado, considerando o segundo cenário, o impacto energético dos núcleos mestres é bem menor em comparação com o primeiro cenário, pelo fato de todos os processadores estarem ocupados e em conseqüência há aumento no consumo de energia total do sistema. Considerando a perda de deadlines, esta tende a diminuir com a inclusão de um novo núcleo mestre, desde que a utilização dos processadores não ultrapasse os 100%.

Finalizando, o custo de comunicação das mensagens de gerenciamento, em comparação com o consumo de energia total do sistema, tem um impacto energético muito pequeno e negligenciável. Estas mensagens são trafegadas apenas nos momentos de alocação e desalocação das tarefas, para informar aos núcleos mestres para atualizarem suas imagens do sistema.

6.11 Análise dos resultados

Os primeiros resultados deste trabalho apresentados na Seção 6.2, publicados em (BRIÃO,2007), mostraram resultados relativos apenas ao tamanho do contexto das tarefas. Estes foram simulados em um sistema sem o desligamento de núcleos inativos (*Power Management)* em um período de tempo de apenas 200 ms de simulação. Estes resultados apresentaram-se insatisfatórios em relação ao consumo de energia em comparação com a abordagem sem o uso de migração de tarefas. A migração de tarefas proveu minimização de perda de deadlines. No entanto, o tempo 200 ms é um tempo pequeno para tirar maiores conclusões sobre a migração, e desta forma, estes experimentos concluem que a migração é somente utilizada para aplicações com pouca comunicação e com tamanhos pequenos de contexto. Neste trabalho, estes experimentos serviram como base para uma análise mais aprofundada e possibilitaram o início de uma série de novos experimentos que pudessem demonstrar a viabilidade da migração de tarefas no âmbito da Exploração de Espaço de Projeto em tempo de execução.

Os resultados apresentados na Seção 6.3, publicados em (BRIÃO,2008), apresentam resultados que permitem avaliar de maneira mais aprofundada os algoritmos *bin-*

packing, em uma mesma freqüência de operação dos processadores, combinados com LC. Foram realizados diversos experimentos referentes ao tamanho da rede-em-chip e a aplicação desses algoritmos. Nestes experimentos, considerou-se o desligamento de processadores inativos conjuntamente com a sua memória local. No contexto da experimentação, descobriu-se um compromisso entre perdas de deadlines e consumo de energia quando se aplicam algoritmos *bin-packing* e *linear clusterization* para alocação de tarefas em sistemas embarcados baseados em redes-em-chip. Existem dois algoritmos que ilustram este panorama: Best-Fit e Worst-Fit combinados com *linear clusterization*. O algoritmo BF + LC resulta no menor consumo de energia do sistema, porém com razoável perda de deadlines. De forma contrária, o algoritmo WF + LC resulta na menor perda de deadlines, no entanto, resulta um razoável consumo de energia. Outra conclusão, é que quanto maior o tamanho da rede-em-chip, para algoritmos WF + LC, maior será o impacto da migração, já que os custos de migração são proporcionais ao número de *hops*.

Os experimentos até aqui mencionados não consideram dados providos do meio externo da rede-em-chip. Esses dados são tratados pelas tarefas específicas de E/S e a Seção 6.4 apresenta resultados relacionados aos algoritmos *bin-packing* modificados para atender esses tipos de tarefas. Os resultados apresentaram-se evidentes, ou seja, quando a comunicação externa é maior que a quantidade de comunicação interna, vale a pena alocar tarefas de E/S nas bordas da rede-em-chip, mais próximas possíveis das interfaces de E/S. Caso contrário, as tarefas de E/S devem ser tratadas da mesma forma que as tarefas convencionais, pois essas, ao serem alocadas perto das interfaces de E/S da rede-em-chip, podem gerar comunicação interna, e desta forma, pode aumentar o número de perda de deadlines e o consumo de energia.

Estes três experimentos mencionados nas Seções 6.2, 6.3 e 6.4 foram desenvolvidos a partir de um modelo de algoritmo *bin-packing* que considerava apenas a ocupação dos processadores, sem levar em conta restrição de memória. A Seção 6.5 apresenta resultados de alocações com migração de tarefas considerando diferentes configurações de tamanhos de memória. Esses resultados permitiram observar que quanto menor a quantidade de memória, menor será obviamente o consumo de energia do sistema, e, além do mais, a taxa de perda de deadlines aumenta drasticamente devido ao maior espalhamento das tarefas e consequentemente ao aumento da comunicação das tarefas. Quanto menor o tamanho das memórias locais dos processadores, o algoritmo BF tende cada vez mais a se aproximar do comportamento do algoritmo WF. Também foi mostrado que a técnica de redução de energia PM é interessante aplicar em tamanhos de memória acima de 4 Kb.

A Seção 6.6 apresenta uma implementação diferente do algoritmo *bin-packing* com restrição, que é o algoritmo *bin-packing* 2D. Este algoritmo, como foi mencionado na Seção 6.6, usa como função a área obtida pelo produto entre tamanho de memória e ocupação dos processadores. No entanto, os resultados apresentados na Seção 6.6 foram equivalentes aos resultados referentes aos algoritmos *bin-packing* com restrição.

Para o uso dos algoritmos *bin-packing* combinados com *linear clusterization*, deve-se obter informações de comparações desses algoritmos com um algoritmo ótimo[15],

[15] O termo "ótimo" aqui empregado tem o significado de solução não-ideal, ou sub-ótimo, pois se sabe que na área de complexidade de algoritmos, algoritmos ótimos, diferentemente do algoritmo *Simulated Annealing*, são classificados como intratáveis, pois nenhum computador resolve-o num tempo satisfatório.

executado em tempo de projeto que gere uma alocação ótima de tarefas. A Seção 6.7 apresenta esses resultados em comparação com o algoritmo *Simulated Annealing* (SA). Os algoritmos *bin-packing*, especificamente os algoritmos BF + LC e WF + LC, obtiveram resultados que evidenciam um aumento de até 23% de consumo de energia e 58% de taxa de perda de deadline comparado com o algoritmo SA. Ainda assim, a justificativa do uso dos algoritmos *bin-packing* é que eles podem ser usados no sistema de maneira *on-line,* já que seus custos computacionais são muito menores que o algoritmo SA. No entanto, o algoritmo SA é mais preciso que os algoritmos *bin-packing*, já que a sua execução é realizada em tempo de projeto. O SA precisou de 1000 iterações de alocação do simulador .

A Seção 6.8 apresentou experimentos quanto à variação da freqüência dos processadores para cada um dos algoritmos. Os resultados mostraram que, para cada algoritmo, existe uma freqüência diferente para anulação da taxa de perda de deadlines. Os resultados corroboraram que o algoritmo WF + LC apresentou menor freqüência para a anulação da taxa de perda de deadlines, comparado com os outros algoritmos. No entanto, o algoritmo BF + LC apresentou menor consumo de energia total, mesmo aumentando a freqüência dos processadores. Estes experimentos demonstraram que o algoritmo BF + LC é selecionado como o melhor algoritmo para distribuição de tarefas no sistema, desde que exista a possibilidade de aumentar a freqüência dos processadores. Contudo é provável que, se houver um número alto de clusters formados pelo algoritmo LC, no qual todos os processadores fiquem ativos, o algoritmo BF + LC poderá ter consumo de energia similar ao consumo de energia proporcionado pelo algoritmo WF + LC.

Todos estes experimentos mencionados foram realizados considerando apenas a migração de tarefas de inicialização[16], ocorrida no início da simulação. No entanto a carga não é alterada em tempo de execução. A Seção 6.9 apresenta alguns experimentos que envolvem variação da carga de execução. Também foi alterado o tamanho do contexto para evidenciar os custos de migração e o impacto que estes ocasionam na rede-em-chip. Estes resultados demonstraram que a realocação das tarefas restantes a uma desalocação de um grafo ou subconjunto da aplicação reduz de maneira significativa tanto o consumo de energia quanto a taxa de perda de deadlines. Esta redução é atingida somente quando o sistema é executado em um tempo suficiente para amortizar cada custo de migração. Outra observação é quanto ao consumo de energia, pois a realocação das tarefas proporciona menor consumo de energia que a abordagem sem realocação, desde que as tarefas restantes sejam executadas em um tempo suficiente para amortizar o custo de migração. Este custo, como foi apresentado também na Seção 6.2, varia de acordo com o tamanho do contexto.

Os experimentos das Seções anteriores à Seção 6.10 não consideram o custo do núcleo mestre, o qual gerencia as ocupações dos demais processadores do sistema. A Seção 6.10 apresenta alguns experimentos que envolvem os custos do núcleo mestre e os custos adicionais de gerenciamento. Como foi mencionado na Seção 6.10, quanto maior o tamanho de rede-em-chip, menor será o impacto do núcleo mestre no sistema, desde que todos os processadores escravos estejam ativos. O tamanho da rede-em-chip conjuntamente com o número de núcleos mestres influenciaram nos custos em termos

[16] Para a realização da migração de inicialização, as tarefas são carregadas pelo sistema e estas são migradas de um mesmo núcleo (emulação do núcleo mestre) para todos os outros processadores inclusive o próprio núcleo mestre.

de perda de deadlines e consumo de energia. É provável que exista um número de processadores mestres máximo para cada configuração de tamanho de rede-em-chip, dependendo das características da aplicação. Finalmente, o custo energético das mensagens de gerenciamento é irrelevante, comparado com o custo energético total do sistema.

7 CONSIDERAÇÕES FINAIS

7.1 Resumo das contribuições

Este trabalho apresentou como principal contribuição o provimento de metodologias para exploração de espaço de projeto e métodos para atendimento de restrições de projeto, ambos em tempo de execução, em sistemas embarcados baseados em redes-em-chip. Estes métodos apresentados compreendem a utilização das heurísticas de alocação de tarefas baseadas nos algoritmos *bin-packing* e *linear clustering* e na combinação destes, uso do DVS e PM e finalmente, migração de tarefas. Todos esses métodos de exploração são aplicados à topologia de rede grelha 2d, para avaliação dos custos dos mecanismos de exploração. Desta forma, foi possível obter resultados que permitem que o projetista possa escolher, em tempo de projeto, uma combinação adequada de métodos de atendimento de restrições *on-line* para uma determinada aplicação ou domínio de aplicações. Como contribuições secundárias deste trabalho, são mencionadas:

- Em relação ao simulador Serpens original, foi realizado o desenvolvimento de um modelo de migração de tarefas (cópia), bem como modificação das primitivas *send* e *receive*, modificação das estruturas de dados das tarefas para prover o tamanho do contexto, e finalmente, modificação nos campos dos pacotes para dar suporte à migração;
- Implementação das heurísticas *bin-packing* para suporte de tarefas de E/S (tarefas de E/S são alocadas o mais próximo possível das interfaces de E/S da rede-em-chip);
- Implementação de geradores de tráfego para simulação de dados de E/S;
- Alteração dos algoritmos *bin-packing* para suporte da restrição do tamanho da memória;
- Implementação do algoritmo *bin-packing* 2D, considerando a área correspondente ao produto entre a ocupação do processador e a ocupação da memória;
- Modificação do algoritmo *Simulated Annealing* (SA) para suporte à execução do simulador *Serpens*, visto que os resultados gerados pelo simulador são dados que o SA avaliou para a execução da otimização. Foram apresentadas três instâncias do algoritmo SA com diferentes focos: minimização do consumo de energia, minimização da taxa de deadlines perdidos, e minimização de ambos;

- Implementação e avaliação dos algoritmos *bin-packing* sobre a plataforma *Femtojava*, em termos de consumo de energia e desempenho, variando o número de tarefas;
- Desenvolvimento de um núcleo mestre para gerenciamento das ocupações de processamento e memória de todos os processadores do sistema;
- Implementação de um protocolo para gerenciamento dos processadores da rede-em-chip e atualização dos dados das ocupações no núcleo mestre, a cada carga de aplicação;
- Desenvolvimento dos estudos de caso sintéticos (*Synthetic* e *App*-synth) com diferentes características de processamento e comunicação para avaliação dos métodos para exploração de espaço de projeto.

Finalmente, este trabalho contribui de forma estratégica para inserção do Laboratório de Sistemas Embarcados (LSE) da Universidade Federal do Rio Grande do Sul (UFRGS) às pesquisas relacionadas a métodos de exploração de espaço de projeto em tempo de execução em sistemas embarcados baseados em redes-em-chip no contexto acadêmico nacional e internacional. Os estudos e as pesquisas desenvolvidas no decorrer desta tese e a difusão dos conhecimentos adquiridos fomentaram o estabelecimento de uma linha de pesquisa sobre estes métodos no grupo local. Desde a consolidação do tema desta tese, em 2005, alguns alunos de graduação e pós-graduação se associaram à investigação de diferentes questões relacionadas a esses métodos de exploração mencionados e à aplicação de redes-em-chip em sistemas integrados, desenvolvendo modelos de plataformas cada vez mais precisos.

7.2 Conclusões

Neste trabalho, pôde-se compreender o cenário atual da pesquisa e desenvolvimento de métodos de EEP em tempo de execução no contexto dos sistemas embarcados. Segundo a literatura, os métodos de exploração de espaço de projeto são usualmente aplicados em tempo de projeto, supondo-se que é conhecido o perfil das aplicações que devem ser executadas pelo sistema embarcado. Entretanto, há um campo significativo de pesquisa a ser explorado na área de exploração de espaço de projeto dinâmico em redes-em-chip, e especialmente na sua utilização em Sistemas Embarcados, área na qual pouco trabalho foi desenvolvido e onde há interesses e projetos já correntes dentro do grupo local. Cada vez mais sistemas embarcados aproximam-se de dispositivos genéricos de processamento (como *palmtops*), onde as tarefas a serem executadas não são inteiramente conhecidas a priori. Com a mudança dinâmica da carga de trabalho de um sistema embarcado, a busca pelo atendimento de requisitos (atendimento de *deadlines*, minimização no consumo de energia) pode então ser enfrentada por mecanismos adaptativos, que implementam dinamicamente a exploração do espaço de projeto. Atualmente há poucos e incipientes trabalhos desenvolvidos que realizam EEP em tempo de execução em redes-em-chip (NOLLET, 2005)(OZTURK, 2006)(CARVALHO, 2007)(CARTA, 2007). EEP em tempo de execução pode permitir resultados ainda melhores, no que diz respeito a sistemas embarcados com restrições de projetos rígidas. Diversos experimentos foram realizados neste trabalho e corroboraram este hipótese. É possível, através do uso de EEP em tempo de execução, maximizar o tempo de duração da energia da bateria que alimenta um sistema embarcado, ou até

mesmo, aumentar a qualidade do sistema, considerando a taxa de perda de deadlines, desde que o custo gerado pela execução do mecanismo de EEP em tempo de execução seja amortizado ao longo da execução do sistema.

Dentro deste contexto, ficou evidenciado que, particularmente, os métodos de alocação e migração de tarefas podem trazer benefícios aos sistemas embarcados. Estes benefícios, dentro do escopo deste trabalho, são referentes ao consumo de energia e à taxa de deadlines perdidos. Além disso, foi observado que os métodos de alocação e migração de tarefas devem ser combinados com outros mecanismos de EEP em tempo de execução, tais como DVS e DPM, para obtenção de resultados ainda melhores que os métodos mencionados em termos de consumo de energia. Os algoritmos BF e BF + LC obtiveram resultados interessantes quanto à economia do consumo de energia, dependendo do volume de comunicação interna do sistema. Por outro lado, na grande maioria dos casos, o algoritmo WF + LC conseguiu obter menor taxa de perda de deadlines do que os outros algoritmos *bin-packing* combinados com *linear clusterization* em uma dada freqüência de operação. No entanto, os resultados obtidos da execução da combinação dos algoritmos WF + LC ainda são insatisfatórios, visto que a taxa de perda de deadlines é alta, mesmo para um sistema de tempo real *soft*.

Para diferentes valores de freqüências, a combinação dos algoritmos BF + LC apresentou menor taxa de perda de deadlines e consumo de energia quando comparada aos outros algoritmos, desde que a freqüência original dos processadores seja aumentada de forma que o algoritmo BF + LC consiga obter um valor de taxa de perda de deadlines menor que algoritmo WF + LC. Mesmo com o aumento da freqüência, foi observado que o consumo de energia do algoritmo BF + LC ainda assim é menor que o WF + LC, executado na sua freqüência original.

Em relação ao comportamento de dados de E/S na rede-em-chip, ficou corroborado que os algoritmos de E/S devem ser utilizados se e somente se há o conhecimento *a priori* do comportamento dos dados de E/S da aplicação. Exemplificando, em uma aplicação de HDTV, podem se empregar algoritmos com restrição de E/S para obter menores taxas de perdas de deadlines. No entanto, não se pode fazer a mesma afirmação em aplicações onde a E/S é destinada a periféricos com baixa taxa de comunicação.

Como foi analisado na Seção 6.9.3, o tamanho do contexto das tarefas tem um impacto decisivo nos custos de migração e consequentemente no tempo de amortização. No que diz respeito à taxa de perda de deadlines, quanto maior o tamanho do contexto, maiores serão os picos de perdas de deadlines no sistema. Isto significa que a técnica de migração de tarefas pode ser aplicada em um sistema de tempo-real hard, desde que haja obrigatoriamente mecanismos que garantam a taxa de perda de deadlines nula. Alternativamente, no que diz respeito ao consumo de energia, quanto maior o contexto, maior será o custo de migração e maior será o tempo de amortização.

Os tamanhos de contexto utilizados neste trabalho variaram aproximadamente entre 1Kb a 10 Kb de contexto. Para tamanhos de contexto de até 1 MB, acredita-se que estes custos sejam amortizados em um tempo ainda viável dentro do contexto dos sistemas embarcados. Esta crença é baseada no tempo de funcionamento dos sistemas embarcados atuais que podem durar até semanas sem recarga da bateria. Alternativamente, para tarefas de contextos de tamanho razoavelmente grande, sugere-se criar réplicas em dois ou mais processadores, justamente para reduzir de modo substancial o custo de migração. No entanto, maiores estudos devem ser realizados dentro deste escopo para maiores conclusões.

No que diz respeito ao volume de comunicação entre as tarefas de uma aplicação, a realocação de tarefas consegue obter maiores ganhos nas aplicações com maior volume de comunicação inter-tarefas que a abordagem sem realocação. Este é um interessante resultado, pois se sabe que domínios de aplicações baseados em multimídia requerem alto volume de comunicação para processamento de dados. No entanto, o tamanho do contexto das tarefas que constituem uma aplicação multimídia pode prejudicar o ganho obtido pela realocação. São necessárias maiores investigações para conclusões mais concretas.

Em relação aos custos de gerenciamento da alocação das tarefas no sistema, foi visto que, inequivocamente, quanto maior o tamanho da rede-em-chip, menor será o impacto de um núcleo mestre no sistema, em termos de consumo de energia e número de perda de deadlines, em relação à rede-em-chip sem esses custos. Isto ocorrerá desde que todos os processadores escravos estejam ativos. O tamanho da rede-em-chip conjuntamente com o número de núcleos mestres influenciaram nos custos em termos de perda de deadlines e consumo de energia. Maiores investigações devem ser feitas para encontrar o número máximo de núcleos mestres para cada tamanho de rede-em-chip e qual seria o melhor posicionamento dos mesmos na rede-em-chip.

7.3 Trabalhos futuros e oportunidades de pesquisa

O prosseguimento deste trabalho ocorre naturalmente com a exploração de um número maior de comportamentos de componentes arquiteturais. Neste trabalho, foi utilizado apenas um modelo de processador, baseado na arquitetura *Femtojava pipelined*, tornando a plataforma de simulação homogênea. No entanto, acredita-se que os resultados obtidos podem ser ainda melhores em uma plataforma heterogênea com outros tipos de elementos de processamento (processadores *superescalar*, multi-ciclo e VLIW, DSP, entre outros). Este crença deve-se ao fato de que tarefas com pequenos prazos de execução possam ser alocadas de forma vantajosa em processadores rápidos e próximos ao núcleo mestre. As demais tarefas são alocadas então nos processadores com menor freqüência de operação. Desta forma há maior consumo de energia em apenas alguns processadores para tarefas com computação intensiva ou deadlines curtos. Além disso, é necessária a realização de experimentos em níveis de abstração mais baixos, buscando-se maior precisão para os resultados, bem como a síntese da plataforma.

Por outro lado, no contexto de exploração, como foi mencionado na Seção anterior, os resultados obtidos da execução da combinação dos algoritmos WF + LC ainda são insatisfatórios, visto que a taxa de perda de deadlines é alta, mesmo para um sistema de tempo real *soft*, dada uma mesma freqüência de operação. Uma possível solução para este problema seria aumentar a largura de banda dos canais e/ou acrescentar canais virtuais à rede-em-chip. Outra solução poderia ser a implementação de um hardware dedicado ao processador para empacotar e desempacotar pacotes de modo eficiente, aumentado o desempenho da execução do núcleo IP.

Ainda dentro do mesmo contexto, algoritmos distribuídos de otimização tais como ACO (*Ant Colony Optimization*) ou PCO (*Particle Swarm Optimization*) poderiam ser utilizados para prover minimização das perdas de deadlines e consumo de energia. Um trabalho interessante dentro do contexto de exploração de espaço de projeto seria a comparação de um sistema com um ou mais nós mestres com um sistema totalmente distribuído, ou seja, cada processador executaria uma parte do algoritmo distribuído. Os

resultados obtidos nesta comparação poderiam guiar o projetista na implementação de métodos de EEP adequados de acordo com o dinamismo da carga do sistema e de acordo com o número de diferentes comportamentos das aplicações carregadas. Alternativamente, os fundamentos da Teoria dos Jogos poderiam ser incluídos para proverem melhor suporte para otimizações em tempo de execução.

Outra linha interessante seria a implementação de diferentes métodos de monitoração e gerenciamento em um sistema distribuído embarcado baseado nos conceitos dos sistemas computacionais distribuídos. Esses métodos poderiam ser baseados nas técnicas conhecidas como *sender-initiated policy* e *receiver-initiated policy*, ou seja, qual política adotar quando um nó está sobrecarregado. É possível que haja um compromisso entre a utilização dessas políticas nos sistemas embarcados distribuídos em diferentes pontos tais como perda de deadlines, consumo de energia, desempenho, etc.

O trabalho realizado nesta tese é baseado em uma exploração de algoritmos configurados em tempo de projeto que permitem uma exploração em tempo de execução. Como trabalho futuro, outros tipos de memórias, tamanhos de caches, topologia da rede, entre outros fatores, poderiam ser desenvolvidos e adicionados ao simulador, os quais implicam em ser configurados em tempo de compilação. A alteração desses fatores modifica drasticamente os requisitos e as restrições de projeto.

Ressaltando o que foi mencionado no parágrafo anterior, uma EEP muito importante para o desenvolvimento de um sistema embarcado é a exploração de organização de memórias, esta EEP sendo realizada *off-line*, pois memórias têm uma grande contribuição no consumo de energia estática de um sistema embarcado. Alterando a organização da memória, é possível alterar as características do acesso a dados, transferência de contexto das tarefas e localidade espacial e temporal. Tais características têm impacto decisivo quanto ao consumo de energia estática e dinâmica, perda de deadlines e desempenho do sistema, que são de fundamental importância para a concepção e o projeto de sistemas integrados em uma única pastilha de silício. Dentro deste contexto, como continuação deste trabalho, pretende-se explorar melhor metodologias para desenvolvimento de mecanismos de migração. É necessário calibrar e inserir mecanismos no simulador com custos mais detalhados e realizar mais experimentos para generalizar a idéia da migração de tarefas dentro do contexto de sistemas embarcados distribuídos. Tais experimentos estarão focados em diferentes políticas de migração (heurísticas), mecanismos de migração em diferentes organizações de memórias e maior gama de aplicações e diferentes tipos de elementos processantes. Desta forma será possível tirar conclusões sobre estas metodologias de exploração de espaço de projeto em tempo de execução em cada uma das plataformas distribuídas não-compartilhadas e compartilhadas, e de como utilizá-las dentro do contexto de sistemas embarcados baseados em redes-em-chip.

Uma oportunidade de pesquisa seria a implementação de diferentes algoritmos que focassem não somente os pontos mencionados neste trabalho como consumo de energia e perda de deadlines, mas também os aspectos de potência, tráfego, e qualidade de serviço. Essas heurísticas, ainda, poderiam contribuir para a melhor distribuição de carga, para aumentar a qualidade do balanceamento de tarefas em elementos de processamento, evitando altos *hot-spots*, ou seja, grandes diferenças térmicas entre os elementos processantes. Essas grandes diferenças, como foram mencionadas no Capítulo 1, podem acarretar problemas no nível submicrônico (HUNG, 2004).

Outra oportunidade de pesquisa seria a implementação de diferentes mecanismos de DVS e PM aplicados aos processadores. Neste trabalho esses mecanismos não foram avaliados em termos de custos, ou seja, não foram contabilizados os custos de execução/consumo de energia/perda de deadlines. No trabalho desta tese, não foram avaliados os custos de desligamento dos processadores, os quais poderiam impactar nos resultados obtidos. Várias implementações de DVS poderiam acrescentar uma maior EEP obtendo um compromisso de desempenho e consumo de energia.

Para aumentar a adaptabilidade da plataforma em questão, poderiam ser inseridos núcleos reconfiguráveis, os quais poderiam obter maiores graus de liberdade e maiores compromissos para um domínio de aplicações, considerando os custos para provimento da adaptabilidade. Desta forma, para uma dada aplicação, a plataforma se adaptaria a esta em vários contextos: reconfigurabilidade, alocação de tarefas, aumento ou diminuição da freqüência de cada processador (DVS) e desligamento de núcleos inativos (PM).

O desenvolvimento de aplicações reais como estudo de casos seria muito interessante para o grupo. Ao invés de utilizar estatísticas de alto nível de abstração, formadas por tarefas representadas por nodos de grafos, a execução de perfis de aplicações reais poderia trazer resultados mais acurados, os quais poderiam consolidar de modo inequívoco as conclusões deste trabalho.

REFERÊNCIAS

ACQUAVIVA, A. et al. Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications. **EURASIP Journal on Embedded Systems** [S.l.] , 2007

ALTERA. **Avalon Bus Specification:** Reference Manual. Document Version 1.2. 2002. Disponível em: <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf>. Acesso em: ago. 2006.

ARM CORPORATION. **AMBA 2.0 specification**. Disponível em: <http://www.arm.com/products/solutions/AMBA_Spec.html>. Acesso em: ago. 2006.

AYDIN, H.; YANG, Q. Energy-Aware Partitioning for Multiprocessor Real-Time Systems, In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2003, Nice, France. **Proceedings…** Los Alamitos, CA:IEEE Computer Society, 2003.

BACKER, E.; JAIN., E. A clustering performance measure based on fuzzy set decomposition. **IEEE Trans. Pattern Anal**. **Mach. Intell**., Los Alamitos, v.PAMI-3, n.1, p.66–75, 1981.

BARCELOS, D. **Um Modelo de Migração de Tarefas para MPSoCs baseados em Redes-em-chip.** 2008. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre

BARCELOS, D. **Aplicabilidade das Técnicas Clássicas de Migração de Processos de Sistemas Distribuídos em MPSoCs**. 2006. 38 f. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

BARAK, A.; WHEELER, R. MOSIX: An Integrated Multiprocessor UNIX. In: WINTER USENIX CONFERENCE, 1989, San Diego, CA. **Proceedings…**[S.l.:s.n.],1989. p.101-112.

BECK, A.; CARRO, L. Low Power Java Processor for Embedded Applications. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION VLSI-SoC, 12., 2003, Darmstadt, Germany. **Proceedings…**, [S.l.:s.n.], 2003.

BECK, A; MATTOS, J.; WAGNER, F.; CARRO, L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator, In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN ,SBCCI, 2003, São Paulo, Brasil. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2003.

BECK, J.;SIEWIOREK, D. Modeling Multicomputer Task Allocation as a Vector Packing Problem. In: INTERNATIONAL SYMPOSIUM ON SYSTEMS SYNTHESIS, 1996, La Jolla, CA, **Proceedings…** Washington: IEEE Computer Society, 1996. p. 115.

BENINI, L.; DE MICHELI, G. Powering Networks on Chips. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 2001, Montreal, Canada. **Proceedings...** Washington: IEEE Computer Society , 2001. p.33-38.

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

BERGAMASCHI, R. A. et al. Automating the design of SoCs using cores. **IEEE Design & Test of Computers**, Los Alamitos, v.18, n.5, p. 32-45, Sept. 2001..

BERTOZZI, S. et al. Supporting Task Migration in MPSoCs: A Feasibility Study. In: DESIGN AUTOMATION AND TEST ON EUROPE, DATE, 2006, Munich, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 2006.

BOLOTIN, E. et al. QNoC: QoS architecture and design process for network on chip. **Journal of Systems Architecture: the EUROMICRO Journal**, New York, v.50, n.2-3, p. 105–128, 2004.

BRIÃO, E. W.; BARCELOS, D.; WAGNER, F. R. Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications. In: DESIGN AUTOMATION AND TEST IN EUROPE, DATE , 2008, Munich, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society, 2008

BRIÃO, E. W.; BARCELOS, D.; WRONSKI, F.; WAGNER, F. R. Impact of Task Migration in NoC-based MPSoCs for Soft Real-time Applications. In: INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 2007, Atlanta, USA **Proceedings…** [S.l.:s.n.], 2007.

BRIÃO, E. W. **Levantamento e comparação de métodos e técnicas para exploração de espaço de projeto em sistemas baseados em redes-em-chip**. 2005. 51 f. Trabalho Individual (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CARRO, L.; WAGNER, F. R. Sistemas Computacionais Embarcados. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, JAI, 22., 2003, Campinas. **Livro Texto**. Campinas:SBC, 2003. p. 45-94.

CARTA, S. et al. Multi-Processor Operating System Emulation Framework with Thermal Feedback for Systems-on-Chip. In: GREAT LAKE SYMPOSIUM ON VERY LARGE SYSTEM INTEGRATION, GLS-VLSI, 2007, Stresa – Lago Maggiore, Italy **Proceedings…** [S.l.:s.n.], 2007.

CARVALHO, E.; CALAZANS, N.; MORAES, F. Congestion-Aware Task Mapping in NoC-based MPSoCs with Dynamic Workload. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, ISVLSI, 2007, Porto Alegre, Rio Grande do Sul, Brazil. **Proceedings…**[S.l.:s.n.], 2007. p.459-460.

CHEN, X.; PEH, L. Leakage Power Modeling and Optimization in Interconnection Networks. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 2003, Seoul, Korea. **Proceedings…**[S.l.:s.n.] , 2003. p. 90-95.

CORRÊA, E. F.; BASSO, E. W.; WILKE, G. R.; WAGNER, F. R.; CARRO, L. The implications of real-time behavior in networks-on-chip architectures. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2004, Toulouse, France. **Proceedings…** Boston: Kluwer Academic Publishers, 2004. p. 307–316.

DALL'OSSO, M. et al. Xpipes: A Latency Insensitive Parameterized Network-On-Chip Architecture for Multiprocessor Socs In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 2003. San Jose, CA, USA, **Proceedings…** Los Alamitos,CA: IEEE Computer Society**,** 2003. p.536 - 539

DALLY, W. J.; TOWLES, B. **Principles and practices of interconnection networks**. San Francisco: Morgan Kaufmann, 2004. 550 p.

DALLY, W.; TOWLES, B. Route packets not wires: on-chip interconnection networks. In: DESIGN AUTOMATION CONFERENCE, DAC, 2001, Las Vegas, NV, USA. **Proceedings...** New York: ACM Press, 2001. p. 684–689.

DICK, R. P.; RHODES, D. L.; WOLF, W. TGFF: task graphs for free. In: INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN, CODES/CASHE, 1998, Seattle, USA. **Proceedings…** Washington: IEEE Computer Society, 1998. p. 97–101.

DUATO, J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks an Engineering Approach**. San Francisco: Elsevier Science: Morgan Kaufmann Publishers, 2002. 600p.

FORSELL, M. A scalable high-performance computing solution for networks on chips. **IEEE Micro**, [S.l.], v.22, n.5, p. 46–55, Sept. 2002.

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

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

GAREY, M. R.; JOHNSON, D.S. **Computers and intractability:** a guide to the theory of NP-completeness. San Francisco: W. H. Freeman, 1979. 340 p.

GERASOULIS, A. YANG, T. On the Granularity and Clustering of Directed Acyclic Task Graphs. **IEEE Trans. Parallel Distrib. Syst**., Los Alamitos, v.4, n.6, June 1993.

GRAAF, B.; LORMANS, M.; TOETENEL, H. Embedded software engineering: the state of the practice. **IEEE Software**, [S.l.], v. 20, n. 6, p. 61 – 69, Nov. – Dec. 2003.

GUERRIER, P.; GREINER, A. A generic architecture for on-chip packet-switched interconnections. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2000. **Proceedings...** [S.l.: s.n.], 2000. p. 250–256.

HEMANI, A. et al. Network on chip: An architecture for billion transistor era. In: IEEE NORCHIP CONFERENCE, 2000. **Proceeding…** [S.l.:s.n.], 2000.

HENTSCHKE, R. **Algoritmos de Posicionamento para Circuitos VLSI.** 2003. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre

HSIEH, C.; PEDRAM, M. Architectural Energy Optimization by Bus Splitting. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v. 21, n. 4, p.408-418, Apr. 2002.

HU, J.; MARCULESCU, R. Energy-aware mapping for tile-based NoC architectures under performance constraints. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2003, Kytakyushu. **Proceedings…** New York:ACM Press, 2003. p. 233–239.

HU, J.; MARCULESCU, R. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In: DESIGN AUTOMATION AND TEST ON EUROPE, DATE, 2004, Paris, France. **Proceedings...** Los Alamitos: IEEE Computer Society, 2004. p. 234-239.

HUNG, W. et al. Thermal-aware IP virtualization and placement for network-on-chip architecture. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, ICCD, 2004, San Jose, CA. **Proceedings...** Washington: IEEE Computer Society, 2004. p. 430–437.

IBM. **The CoreConnect™ bus architecture**. Disponível em: <http://www-03.ibm.com/chips/products/coreconnect/>. Acesso em: ago. 2006.

ITO, S.; CARRO L.; JACOBI, R. P. Making Java Work for Microcontroller Applications. **IEEE Design & Test of Computers**, CA, v.18, n.5**.** p. 100-110, Sept./Oct. 2001.

ITRS: International Technology Roadmap for Semiconductors. Disponível em: <http://public.itrs.net/>. Acesso em: ago. 2006.

JANTSCH, A. TENHUNEN, H. **Networks on chip**. Boston: Kluwer Academic Publishers, 2003. 303 p.

JERRAYA, A.; TENHUNEN H.; WOLF, W. Multiprocessor System-on-Chips. **Computer**, Los Alamitos, v. 38, n. 7, p. 36-40, Feb. 2005.

JOHNSON, D. S. **Near-optimal Bin-packing Algorithms**. Cambridge, MA:MIT, 1973.

KARIM, F.; NGUYEN, A.; DEY, S. On-chip communication architecture for OC-768 network processors. In: DESIGN AUTOMATION CONFERENCE, DAC, 2001, Las Vegas. **Proceedings...** New York: ACM Press, 2001. p. 678-683.

KARIM, F.; NGUYEN, A.; DEY, S. An interconnect architecture for networking systems on Chips. **IEEE Micro**, [S.l.], v. 22, n. 5, p. 36–45, Sept. 2002.

KATCHER, D. et al. Engineering and Analysis of Fixed Priority Schedulers. **Software Engineering**, [S.l.], v. 19, n. 9, p. 920-934, 1993.

KEUTZER, K. et al. A system-level design: orthogonalization of concerns and platform-based design. **IEEE Transactions on Computer-Aided Design of Integrated Circuits**, [S.l.], v.19, n.12, p. 1523–1543, Dec. 2000.

KIM, D.; KIM, M.; SOBELMAN, G. DCOS: Cache Embedded Switch Architecture For Distributed Shared Memory Multiprocessor Socs In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2006, Island of Kos, Greece. **Proceedings…** [S.l.:s.n.], 2006. p. 979-982.

KIRKPATRICK, S.; GELATT JUNIOR, C. D.; VECCHI, M. P. Optimization by Simulated Annealing. **Science**, [S.l.], v.220, n.4598. p. 671-680, 1983.

KORDON, F.; HENKEL, J. An Overview of Rapid System Prototyping Today. **Design Automation for Embedded Systems**, Boston, v.8, n.4, p. 275-282, 2003.

KREUTZ, M.; MARCON, C.; CALAZANS, N.;SUSIN, A. A. Energy and latency evaluation of NoC topologies. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2005, Kobe. **Proceedings...** Los Alamitos: IEEE Computer Society, 2005. p. 5866–5869.

KUMAR, S. et al. A Network On Chip Architecture And Design Methodology. In: SYMPOSIUM ON VERY LARGE INTEGRATION SCALE, VLSI, 2002. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002. p.105-112.

KUMAR, S. On Packet Switched Network on Chip Communication. In: JANTSCH, A. ;TENHUNEN, H. (Ed.). **Networks on chip**. Boston: Kluwer Academic Publishers, 2003. p. 85-106.

LEI, T.; KUMAR, S.; A Two-Step Genetic Algorithm For Mapping Task Graphs to a Network-on-Chip Architecture. In: EUROMICRO SYMPOSIUM ON DIGITAL SYSTEM, 2003, Belek-Antalya, Turkey. **Proceedings...** Los Alamitos:IEEE Computer Society Press, 2003. p.180 – 187.

LEUNG, V. et al. Processor Allocation on Cplant: Achieving General Processor Locality Using One-Dimensional Allocation Strategies. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, Chicago, Illinois. **Proceedings...** Los Alamitos:IEEE Computer Society Press, 2002. p. 296–304.

LIANG, J.; SWAMINATHAN, S.; TESSIER, R. aSoC: a scalable, single-chip communication architecture. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT, 2000, Philadelphia. **Proceedings...** Washington: IEEE Computer Society, 2000. p. 37–46.

LITZKOW, M. et al. Condor – A Hunter of Idle Workstations. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 1988, San Jose, CA. **Proceedings…**[S.l.:s.n.], 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, USA. **Proceedings…** New York: ACM Press, 2004.

LYONNARD, D. et al. Automatic generation of application specific architectures for heterogeneous multiprocessor system-on-chip. In: DESIGN AUTOMATION CONFERENCE, DAC, 2001, New Orleans. **Proceedings…** New York: ACM Press, 2001. p. 518–523.

MADISETTI, V. K.; SHEN L. Interface Design for Core-Based Systems. **IEEE Design & Test of Computers**, Los Alamitos, v. 14, n.4, p. 42-51, Oct./Dec. 1997.

MARCON, C. M. **Modelos para o mapeamento de aplicações em infra-estruturas de comunicação intrachip**. 2005. 192 f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

MARESCAUX, T. et al. Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs. In: FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS FPL, 2002, Montpellier, France. **Proceedings...** [S.l.: s.n.], 2002. p. 795–805.

MARWEDEL, P. **Embedded System Design**. Dordrecht: Kluwer Academic Publishers, 2003.

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

MOORE, G. E. Progress in digital integrated electronics. In: IEEE INTERNATIONAL ELECTRON DEVICES MEETING, 1975. **Proceedings…** Los Alamitos: IEEE Computer Society, 1975. p. 11.

MORAES, F. G. et al. A low area overhead packet-switched networks on chip: architecture and prototyping. In: INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 2003, Darmstadt, Germany. **Proceedings...** [S.1: s.n.], 2003. p. 318–323.

MURALI, S.; DE MICHELI, G. Bandwidth-constrained mapping of cores onto NoC architectures. In: DESIGN AUTOMATION AND TEST IN EUROPE, DATE, 2004, Paris, France. **Proceedings…** Los Alamitos: IEEE Computer Society, 2004. p. 896–901.

MURALI, S.;DE MICHELLI, G. SUNMAP: a tool for automatic topology selection and generation for NoCs. In: DESIGN AUTOMATION CONFERENCE, DAC, 2004, San Diego, CA. **Proceedings...** New York: ACM Press, 2004. p. 914–919.

NGOUANGA, A.; SASSATELLI, G.; TORRES, L.; GIL, T.; SOARES, A.; SUSIN, A. A contextual resources use: A proof of concept through the APACHES platform. In: IEEE WORKSHOP ON DESIGN AND DIAGNOSTICS OF ELECTRONIC CIRCUITS AND SYSTEMS, DDECS, 2006, Czech Republic, **Proceedings...** Los Alamitos: IEEE Computer Society, 2006. p 42-47.

NI, L. M.; MCKINLEY, P. K. A Survey of Wormhole Routing Techniques in Direct Networks. **Computer**, Los Alamitos, v. 26, n. 2, p. 62-76, Feb. 1993.

NOLLET, V. et al. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2005, Munich, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society, 2005. p.234–239.

OGRAS, U.; MARCULESCU, R. Energy and performance-driven NoC communication architecture synthesis using a decomposition approach. In: DESIGN, AUTOMATION AND TEST IN EUROPE , DATE, 2005, Munich, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 2005. p.352–357.

OLIVEIRA, M. F. S.; BRIÃO, E. W.; NASCIMENTO, F. A.; WAGNER, F. R. Model Driven Engineering for MPSoC Design Space Exploration. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 2007, Rio de Janeiro, Brasil. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2007.

OLIVEIRA, M. F. S.; BRIÃO, E. W.; NASCIMENTO, F. R.; BRISOLARA, L. B.; CARRO, L.; WAGNER, F. R. Multi-objective Design Space Exploration based on UML. In: UML FOR SoC DESIGN, UML-SOC, 2006, San Francisco. **Proceedings…** [S.l.:s.n.], 2006.

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

OZTURK, O. et al. Selective Code/Data Migration for Reducing Communication Energy in Embedded MpSoC Architectures. In: GREAT LAKE SYMPOSIUM ON VERY LARGE SYSTEM INTEGRATION, GLSVLSI, 2006, Philadelphia, PA. **Proceedings…** [S.l.:s.n.], 2006. p. 386-391.

PÉTROT, F.; GREINER A.; GOMEZ, P., On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures, In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN, 2006, Cavtat, Croatia. **Proceedings…** [S.l.:s.n.], 2006. p. 53-60.

POP, R.; KUMAR, S. **A survey of techniques for mapping and scheduling applications to network on chip systems**. Jönköping, Sweden: Jönköping University, 2004. Technical Report.

RAGHUNATHAN, V.; SRIVASTAVA, M. B.; GUPTA, R. K. A survey of techniques for energy efficient on-chip communication. In: DESIGN AUTOMATION CONFERENCE, DAC, 2003, New Orleans. **Proceedings…** New York: ACM Press, 2003. p. 900–905.

RIJPKEMA, E. et al. Trade Offs in the design of a router with both guaranteed and best-effort services for networks on chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2003, Munich, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society, 2003. p. 350-355.

RUGGIERO, M. et al. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2006, Munich, Germany. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2006. p. 3-8.

SAASTAMOINEN, I.; SIGUENZA-TORTOSA, D.; NURMI, J. Interconnect IP node for future system-on-chip designs. In: WORKSHOP ON ELECTRONIC DESIGN TEST AND APPLICATION, DELTA, 2002. **Proceedings...** Washington, DC: IEEE Computer Society, 2002. p. 116-120.

SARKAR, V. **Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors**. Cambridge, MA: MIT Press, 1989.

SCHALLER, R. Moore´s Law: Past, Present and Future. **IEEE Spectrum**, [S.l.], v. 34, n. 6, p. 52-59, June 1997.

SGROI, M. et al. Addressing the system-on-chip interconnect woes through communication-based design. In: DESIGN AUTOMATION CONFERENCE, DAC, 2001, Las Vegas, USA. **Proceedings...** New York: ACM Press, 2001. p. 667–672.

SHIN, D.; KIM, J. Power-aware communication optimization for networks-on-chips with voltage scalable links. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, CODES, 2004, Stockholm, Sweden. **Proceedings...** [S.l.: s.n.], 2004. p.170–175.

SIA - Semiconductor Industry Association. **SIA Annual Report 2005**. Disponível em: <http://www.sia-online.org/downloads/SIA_AR_2005.pdf> Acesso em: ago. 2006.

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

SZYMANEK, R.; KRZYSZTOF, K. Partial task assignment of task graphs under heterogeneous resource constraints. In: DESIGN AUTOMATION CONFERENCE, DAC, 2003, Anaheim, CA. **Proceedings…** New York: ACM Press, 2003. p. 244-249.

TEWKSBURRY, S. K.; UPPULURI M.; HORNAK, L. A. Interconnections/micro-network for integrated circuits. In: GLOBAL TELECOMMUNICATIONS CONFERENCE, GLOBECOM, 1992. **Proceedings...** [S.l.: s.n.], 1992. p.180–186.

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

TRUYEN, E. et al. **Portable Support for Transparent Thread Migration in Java**. Berlin: Springer-Verlag, 2000. p. 29-43. (Lecture Notes in Computer Science, v. 1882).

uClinux: Embedded Linux Microcontroller Project. Disponível em: <http://www.uclinux.org>. Acesso em: out. 2007.

VARATKAR, G.; MARCULESCU, R. Communication-aware task scheduling and voltage selection for total systems energy minimization. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 2003, San Jose, CA. **Proceedings...** Los Alamitos,CA: IEEE Computer Society, 2003. p. 510–517.

WANG, H. Orion: A Power-performance Simulator for Interconnection Networks. ACM MICRO, 2002, Istambul, Turkey. **Proceedings…** [S.l.:s.n.], 2002. p. 294-305.

WEISER, M. et al. Scheduling for Reduced CPU Energy. In: SYMPOSIUM ON OPERATION SYSTEMS DESIGN AND IMPLEMENTATION, 1994, Monterey, CA. **Proceedings…** [S.l.:s.n], 1994. p.13-23.

WOLF, W. **Computers as components**: principles of embedded computing system design. San Francisco, CA: Morgan Kaufmann, 2001. 662 p.

WRONSKI, F. **Alocação Dinâmicas de Tarefas Periódicas em MPSoCs com Redução do Consumo de Energia.** 2007. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre

WRONSKI, F.; BRIÃO, E. W.; WAGNER, F. R. Evaluating Energy-aware Task Allocation Strategies for MPSOCS. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2006, Braga, Portugal. **Proceedings…** New York: Springer, 2006. p.215-224.

YAO, F. et al. A Scheduling Model for Reduced CPU Energy. ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, FOCS, 1995 . **Proceedings…** [S.l.:s.n.], 1995. p. 374- 382.

YE, T.; DE MICHELI, G.; BENINI, L. Analysis of Power Consumption on Switch Fabrics in Network Routers. In: DESIGN AUTOMATION CONFERENCE, DAC, 2002, New Orleans. **Proceedings…** New York: ACM Press, 2003. p. 524-529.

ZEFERINO, C. A. **Redes-em-chip:** arquiteturas e modelos para avaliação de área e desempenho**.** 2003. 242 f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

ZEFERINO, C. A.; SUSIN, A. A. SoCIN: a parametric and scalable network-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 2003, São Paulo, Brasil. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2003. p. 169–174.

ZEFERINO, C. A.; KREUTZ, M. E.; SUSIN, A. A. RASoC: A Router Soft-Core for Networks-on-Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2004. Paris, France. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2004. p. 198-205.

ZHOU, S. et al. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. **Software – Practive and Experience**, [S.l.], Dec. 1994.

ZHUO, J.; CHAKRABARTI, C. An Efficient Dynamic Task Scheduling Algorithm for Battery Powered DVS Systems. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2005, Shangai, China. **Proceedings…** New York: ACM Press, 2005. p. 846-849.

APÊNDICE A PUBLICAÇÕES GERADAS AO LONGO DA TESE

Trabalhos publicados relacionados diretamente com o trabalho desta tese:

BRIÃO, E. W.; BARCELOS, D.; WAGNER, F. R. Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications. In: DESIGN AUTOMATION AND TEST IN EUROPE, DATE, 2008, Munich, Germany. **Proceedings…** Los Alamitos: IEEE Computer Society, 2008

BRIÃO, E. W.; BARCELOS, D.; WRONSKI, F.; WAGNER, F. R., Impact of Task Migration in NoC-based MPSoCs for Soft Real-time Applications. In: INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 2007, Atlanta, USA.**Proceedings…**, [S.l.:s.n.], 2007.

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 ,SBCCI, 2007, Rio de Janeiro, Brasil. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2007.

WRONSKI, F.; BRIÃO, E. W.; WAGNER, F. R. Evaluating Energy-aware Task Allocation Strategies for MPSOCS. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2006, Braga, Portugal. **Proceedings…** New York: Springer, 2006. p.215-224.

Trabalhos publicados relacionados indiretamente com o trabalho da tese:

Os trabalhos publicados abaixo proporcionaram a implementação da ferramenta de exploração de espaço de projeto usando o algoritmo *Simulated Annealing*.

OLIVEIRA, M. F. S.; BRIÃO, E. W.; NASCIMENTO, F. A.; WAGNER; F. R. Model Driven Engineering for MPSoC Design Space Exploration. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 2007, Rio de Janeiro, Brasil. **Proceedings…** Los Alamitos, CA: IEEE Computer Society, 2007.

OLIVEIRA, M. F. S.; BRIÃO, E. W.; NASCIMENTO, F. R.; BRISOLARA, L. B.; CARRO, L.; WAGNER, F. R. Multi-objective Design Space Exploration based on UML. In UML FOR SoC DESIGN, UML-SOC, 2006, San Francisco. **Proceedings…** [Sl.:s.n], 2006.

APÊNDICE B VARIAÇÃO DA CARGA EM TEMPO DE EXECUÇÃO

A Fig. 1 apresenta três cenários de execução da aplicação *Telecom*. No Cenário 1, as tarefas da aplicação *Telecom 2* são alocadas utilizando o algoritmo WF + LC. Observa-se que existem dois clusters que tem uma razoável quantidade de comunicação. Um cluster está alocado a dois *hops* de distância de seu par, e o outro está a quatro *hops* de distância. O Cenário 2 ilustra o momento depois da desalocação de um dos quatro grafos da aplicação, tornando o processador desocupado, posicionado entre os clusters que se comunicam entre si. Finalmente o Cenário 3 apresenta a realocação dos clusters, utilizando o algoritmo WF + LC. Este algoritmo de realocação, neste estudo de caso, proporciona uma maior proximidade entre os clusters que se comunicam, reduzindo o número total de 6 *hops* de distância dos quatro clusters para 2 *hops*. Desta forma, como foi mostrado na Seção 6.9.2, a realocação pode reduzir o número de perda de deadlines e consumo de energia.

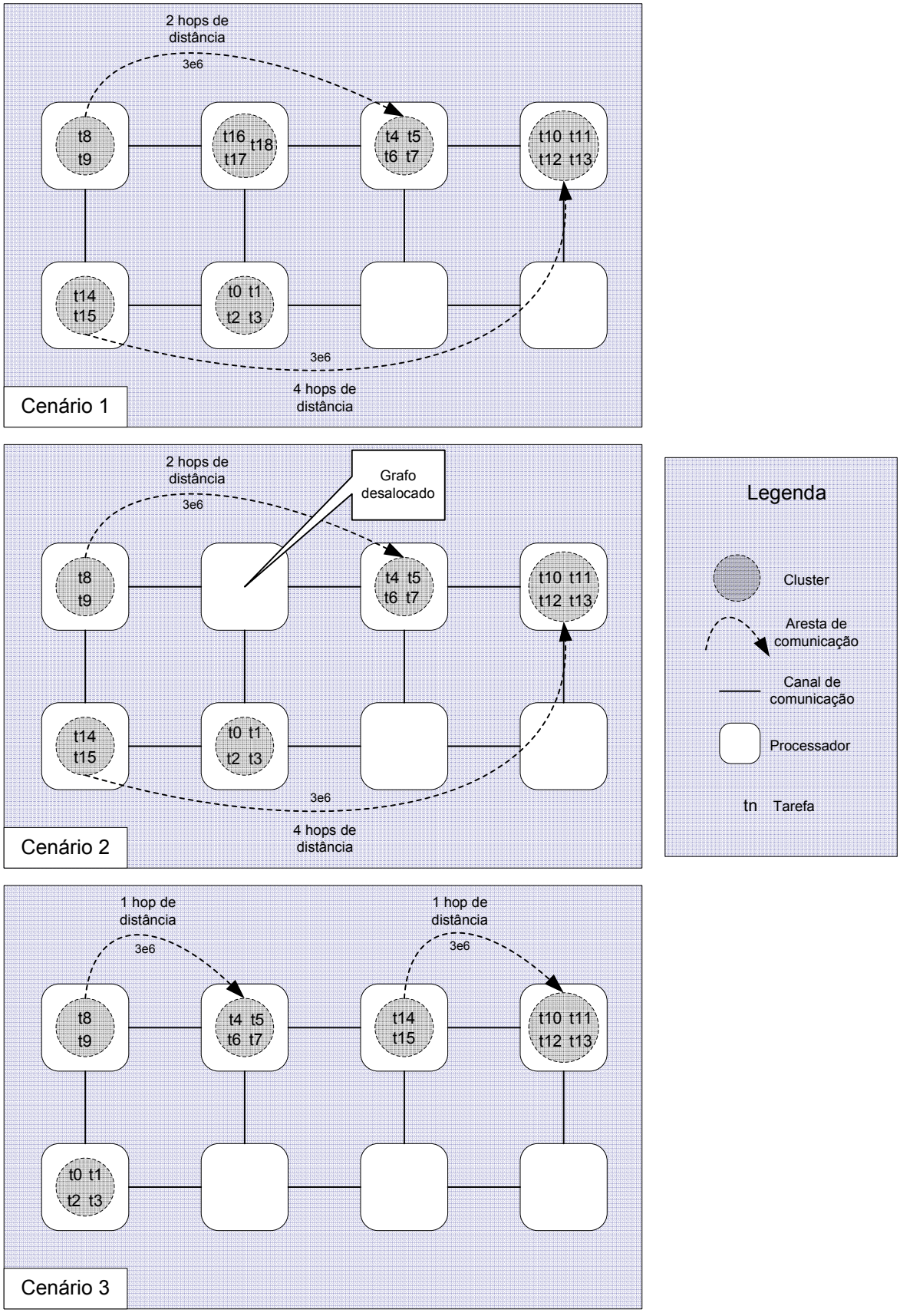


Fig. 1: Três cenários de execução da aplicação *Telecom 2* com 4 grafos.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo