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

FABIO WRONSKI

Alocação Dinâmica de Tarefas Periódicas em NoCs Malha com Redução do Consumo de Energia

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

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, março de 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Wronski, Fabio

Alocação Dinâmica de Tarefas Periódicas em NoCs Malha com Redução do Consumo de Energia / Fabio Wronski – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.

103 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Orientador: Flávio Rech Wagner.

1.Alocação e particionamento de tarefas. 2.Escalonamento de tarefas 3.Redes em-chip. I. Wagner, Flávio Rech. II. Título.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

VLSI	*Very Large Scale Integration*
SoC	*System on-chip*
PDA	*Personal Digital Assistant*
NoC	*Network on-chip*
MPSoC	*Multiprocessor System on-Chip*
E3S	*Embedded System Synthesis Benchmark Suite*
TGFF	*Task Graph for Free*
EEMBC	*Embedded Microprocessor Benchmark Consortium*
MIT	*Massachusetts Institute of Technology*
CMOS	*Complementary Metal Oxide Semiconductor*
DAG	*Direct Acyclic Graph*
VS	*Voltage Scaling*
PM	*Power Management*
WCET	*Worst Case Execution Time*
RM	*Rate-Monotonic*
DM	*Deadline Monotonic*
LLF	*Least Laxity First*
EDF	*Earliest Deadline First*
WCRT	*Worst-Case Response Time*
DVS	*Dynamic Voltage Scaling*
SPM	*Simple Power Management*
AVR	*Average Rate Heuristic*
LPFPS	*Low-Power Fixed Priority Scheduling*
LPPS	*Low-Power Priority Based Real-Time Scheduling*
LA	*Look-ahead*
LPSEH	*Low-Power Scheduling using Slack Estimation heuristic*
DAR	*Dynamic Average Rate*

LAN	*Local Area Network*
PHIT	*PHysical UnIT*
BNP	*Bounded Number of Processor*
UNP	*Unbounded Number of Processors*
BP	*Bin-packing*
NF	*Next Fit*
FF	*First Fit*
BF	*Best Fit*
WF	*Worst Fit*
MC	*Minimizing message-passing Contention*
MBS	*Multiple Buddy Strategy*
DEDF	*Deferred EDF*
EAT	*Earliest Available Time*
LST	*Latest Start Time*
HLFET	*Highest Levels First with Estimated Times*
LPST	*Latest Possible Start Time*
EST	*Earliest Start Time*
MCP	*Modified Critical Path*
ETF	*Earliest Task First*
DLS	*Dynamic Level Scheduling*
DL	*Dynamic Level*
EZ	*Edge Zeroing*
LC	*Linear Clustering*
DSC	*Dominant Sequence Clustering*
MD	*Mobility Directed*
LSSR	*Fixed-Order List scheduling with Shared Reclamation*
PDP	*Proporcional Distribuition and Paralelism*
LPDEDF	*Low-Power Distributed EDF*
WFD	*Worst-Fit Decreasing*
FFD	*First-Fit Decreasing*
FFD	*First-Fit Decreasing*
BFD	*Best-Fit Decreasing*
TGFF	*Task Graphs for Free*
CWM	*Communication Weight Model*
RTL	*Register Transfer Level*

TLM	*Transaction Level Model*
FPGA	*Field-programable Gate Array*
BOP	*Begin Of Packet*
EOP	*End Of Packet*
RTSJ	*Real-Time Specification for Java*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE EQUAÇÕES

RESUMO

O objetivo deste trabalho é propor técnicas de alocação dinâmica de tarefas periódicas em MPSoCs homogêneos, com processadores interligados por uma rede em-chip do tipo malha, visando redução do consumo de energia do sistema. O foco principal é a definição de uma heurística de alocação, não se considerando protocolos de escalonamento distribuído, uma vez que este ainda é um primeiro estudo para o desenvolvimento de um alocador dinâmico.

Na arquitetura alvo utilizada, cada nodo do sistema é dado como autônomo, possuindo seu próprio escalonador EDF. Além disso, são aplicadas técnicas de *voltage scaling* e *power managmenent* para redução do consumo de energia durante o escalonamento.

Durante a pesquisa do estado da arte, não foram encontradas técnicas de alocação dinâmica em NoCs com restrições temporais e minimização do consumo de energia. Por isso, esse trabalho se concentra em avaliar técnicas de alocação convencionais, como *bin-packing* e técnicas baseadas em teoria de grafos, no contexto de sistemas embarcados. Dessa forma, o modelo de estimativas do consumo de energia de alocações é baseado no escalonamento de grafos de tarefas, e foi utilizado para implementar a ferramenta Serpens com este propósito.

Os grafos de tarefas utilizados nos experimentos são tirados do *benchmark* E3S – *Embedded System Synthesis Benchmark Suite*, composto por um conjunto de grafos de tarefas gerados aleatoriamente com a ferramenta TGFF – *Task Graph for Free*, a partir de dados de aplicações comuns em sistemas embarcados obtidos no EEMBC – *Embedded Microprocessor Benchmark Consortium*.

Entre as heurísticas de *bin-packing, Best-Fit, First-Fit* e *Next-Fit* geram alocações com concentração de carga, enquanto a heurística *Worst-Fit* faz balanceamento de carga. O balanceamento de carga favorece a aplicação de *voltage scaling* enquanto a concentração favorece o *power management*.

Como o *bin-packing* não contempla comunicação e dependência entre tarefas em seu modelo, o mesmo foi reformulado para atender esta necessidade. Nos experimentos, a alocação inicial com *bin-packing* original apresentou perdas de *deadlines* de até 84 % para a heurística *Worst-Fit*, passando para perdas em torno de 16% na alocação final, praticamente com o mesmo consumo de energia, após a reformulação do modelo.

Palavras-Chave: Alocação e particionamento de tarefas, escalonamento de tarefas, redes em-chip.

Energy-Aware Dynamic Allocation of Periodic Tasks on Mesh NoCs

ABSTRACT

The goal of this work is to offer dynamic allocation techniques of periodic tasks in mesh networks-on-chip, aiming to reduce the system power consumption. The main focus is the definition of an allocation heuristic, which does not consider distributed scheduling protocols, since this is the beginning of a study for the development of a dynamic partitioning tool. In the target architecture, each system node is self-contained, that is, the nodes contain their own EDF scheduler. Besides, voltage-scaling and power management techniques are applied for reducing power consumption during the scheduling.

To the best of our knowledge, this is the first research effort considering both temporal constraints and power consumption minimization on the dynamic allocation of tasks in a mesh NoC. This way, our concentrates in the evaluation of dynamic allocation techniques, which are generally used in distributed systems, in the embedded systems context, as *bin-packing* and graph theory based techniques. Therefore, the estimation model for power consumption is based on task graph scheduling, and it was used for implementing the Serpens tool with this purpose.

The task graphs used in the experiments were obtained from the E3S benchmark (Embedded System Synthesis Benchmark Suite), which is composed by a set of task graphs randomly generated with the TGFF tool (Task Graph for Free), from common application data obtained from the EEMBC (Embedded Microprocessor Benchmark Consortium).

Among the bin-packing heuristics, Best-Fit, First-Fit, and Next-Fit generate allocations with load concentration, while the Worst-Fit heuristics works with load balancing. Load balancing favors the application of voltage scaling, while load concentration favors the utilization of power management.

Since the bin-packing model does not consider inter-task communication and dependency, it has been modified to fulfill this need. In the experiments, the initial allocation using the original bin-packing model presented deadline losses of up to 84% for the Worst-Fit heuristic, changing for losses around 16% in the final allocation, after modification of the model, maintaining almost the same power consumption.

Keywords: Task allocation and partitioning, task scheduling, network-on-chip.

1 INTRODUÇÃO

1.1 Motivação

Com o avanço das tecnologias de fabricação de circuitos integrados VLSI – *Very Large Scale Integration*, está cada vez mais comum integrar vários elementos de hardware em um único *chip* chamado SoC – *System on-chip*.

À medida que os SoCs se tornam mais complexos e incorporam mais funcionalidades passam a consumir mais energia. Esse aumento de consumo tem como principal efeito a redução de autonomia de dispositivos alimentados por baterias.

Dispositivos móveis não são os únicos a sofrer as conseqüências do aumento do consumo de potência dos processadores. Os computadores de mesa também padecem do mesmo problema, principalmente no que diz respeito à dissipação de calor.

As técnicas atualmente utilizadas para a redução do consumo de energia em nível de sistema são o VS – *Voltage Scaling* e PM – *Power Management* (BENINI; BOGLIOLO; MICHELI, 2000). O PM visa economia de energia através do desligamento de componentes do sistema que não estão em uso no momento. Já a técnica de VS busca reduzir a tensão de alimentação destes componentes, uma vez que esta tem impacto quadrático no consumo de energia e causa redução de freqüência apenas linear.

Há basicamente duas formas de se aplicar VS: distribuir o processamento de um processador entre *x* processadores com freqüência de operação de 1/x cada, ou balancear o desempenho do processador entre os períodos de ociosidade e de alta utilização, de forma a uniformizar a freqüência utilizada e assim executar sempre com uma tensão baixa.

Intuitivamente, escalonar para PM é tornar concentrados e longos os períodos ociosos do escalonamento. Desse modo, o processador fica desligado mais tempo e desperdiça menos com o *overhead* de desligar e religar. Ao contrário, com VS tenta-se diluir os períodos ociosos para atenuar os picos de utilização.

Além da restrição de consumo de energia, muitos sistemas embarcados apresentam restrições temporais, do tipo prazo de resposta, que transformam o momento em que a resposta do sistema é dada numa componente da própria solução. Esses sistemas são chamados de sistemas de tempo real.

Da mesma forma que em sistemas monoprocessados, VS e PM também podem ser aplicados em MPSoCs – *Multi-processor SoCs*, mesmo se for considerada a existência de restrições temporais, podendo-se ainda explorar completamente o compromisso entre

potência e desempenho estaticamente, quando as cargas de trabalho são previsíveis ou conhecidas a priori.

Infelizmente, na maioria das situações reais não há tal conhecimento a priori. A carga de trabalho, ao contrário, é geralmente imprevisível em muitos sistemas, desde simples telefones celulares e PDAs – *Personal Digital Assistant* até SoCs mais complexos. Neste contexto, pode haver desperdício de energia, uma vez que a alocação das tarefas e recursos é feita em função do pior caso, que raramente ocorre.

Além disso, os MPSoCs tendem a integrar um grande número de elementos, muitas vezes heterogêneos, em único um *chip*, requerendo estruturas de interconexão bastante elaboradas, com escalabilidade e alta nível de paralelismo, como as NoCs – *Networks on-Chip*.

Mesmo em SoCs homogêneos, diferentes alternativas de alocação geram diferentes caminhos de roteamento e volumes de informação. Com isso, duas alocações de tarefas viáveis podem resultar em consumos de energia completamente diferentes.

O processo de se escalonar tarefas em um sistema composto por múltiplos processadores tem basicamente duas abordagens: o escalonamento global e o particionamento.

No escalonamento global há apenas um escalonador para todos os processadores, o qual mapeia a execução de cada tarefa para um processador, possivelmente diferente a cada ocorrência da mesma. Essa abordagem é preferencialmente utilizada em sistemas com memória compartilhada, que oferecem preempção rápida e fácil entre processadores.

No esquema de particionamento, as tarefas são alocadas em processadores e todas as suas ocorrências subseqüentes devem ser executadas neste processador. Nessa abordagem, cada processador possui seu próprio escalonador local, que faz o escalonamento de acordo com sua política. Desta maneira, o processo de escalonamento fica dividido em duas fases distintas: alocação e escalonamento (local), permitindo o reaproveitamento de técnicas de escalonamento para um único processador.

A abordagem mais comum em sistemas dinâmicos é alocar as tarefas nos próprios processadores onde são criadas. A partir do momento que o sistema degrada em função da má distribuição de carga ou do alto custo de comunicação, as tarefas começam a ser redistribuídas.

Uma alternativa é distribuir as tarefas tão logo entrem no sistema, para reduzir a sua degradação e o respectivo custo para correção. Nesse caso, técnicas eficientes, e principalmente muito rápidas, se fazem necessárias, uma vez que soluções ótimas, nesse contexto, são reconhecidamente problemas do tipo NP-completo.

Além disso, um sistema aonde aplicações iniciam e finalizam aleatoriamente tende a fragmentar e perder desempenho, mesmo que um algoritmo de alocação ótimo seja utilizado. Nesse caso, vai sempre ser necessário o uso de técnicas de relocação, lançando-se mão de migração de tarefas já iniciadas. Por isso, o mais importante é que a heurística de alocação seja rápida e não ótima.

As principais heurísticas de alocação dinâmica de tarefas são geralmente baseadas nas abordagens de *bin-packing* e teoria de grafos, sendo que o *bin-packing* não considera dependências de tarefas e/ou comunicação, como na abordagem de grafos que geralmente usa um modelo de tarefas baseado em DAGs.

1.2 Objetivos

O objetivo deste trabalho é propor técnicas de alocação dinâmica de tarefas periódicas em MPSoCs homogêneos, com processadores interligados por uma NoC malha, visando redução do consumo de energia do sistema. A NoC malha foi escolhida devido à sua simplicidade e escalabilidade

O desenvolvimento de um particionador dinâmico envolve a definição de duas políticas: a heurística de alocação, que visa definir qual o melhor destino de uma tarefa; e o algoritmo de escalonamento distribuído, responsável por implementar essa heurística através da colaboração dos nodos do sistema.

Este trabalho é um primeiro estudo para o desenvolvimento de um particionador dinâmico voltado para a redução de consumo de energia do sistema. Dessa forma, ele se concentra no desenvolvimento de uma heurística de alocação, deixando para trabalhos futuros o estudo de algoritmos de escalonamento distribuído.

Consequentemente, sem escalonador distribuído implementado, torna-se necessário desenvolver modelos que forneçam o grau de abstração adequado para se alocar e escalonar tarefas nos processadores. Além disso, devem permitir também a estimação dos custos de cada alocação, como taxa de atendimento de *deadlines* das tarefas e consumo de energia do sistema.

1.3 Contribuições

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 no contexto de embarcados.

A principal técnica estudada é a de *bin-packing*, que visa reduzir o número de recursos utilizados através da alocação e possui heurísticas de baixa complexidade, podendo serem facilmente aplicadas em alocação *on-line*.

Contudo, o modelo de *bin-packing* não contempla comunicação e dependência entre tarefas, por isso, propõe-se combina-lo com técnicas baseadas em grafos, como *clustering* para contornar essa limitação. Por este motivo, o modelo de tarefas utilizado é baseado em grafos de tarefas, onde os nodos dos grafos representam as tarefas e, a arestas, as dependências de comunicação.

Os grafos de tarefas utilizados nos experimentos são tirados do *benchmark* E3S – *Embedded System Synthesis Benchmark Suite*, composto por um conjunto de grafos de tarefas gerados aleatoriamente com a ferramenta TGFF – *Task Graph for Free*, a partir de dados de aplicações comuns em sistemas embarcados obtidos no EEMBC – *Embedded Microprocessor Benchmark Consortium*.

A arquitetura alvo utilizada é uma NoC malha de tamanho 4x4, do mesmo tamanho utilizado por Hu; Marculescu (2004), o qual julga-se razoável para MPSoCs atuais.

Heurísticas de *bin-packing* foram avaliadas anteriormente para alocação de tarefas em sistemas multiprocessadores com barramento centralizado (AYDIN; YANG, 2003). Porém, os resultados obtidos não foram validados em redes do tipo malha, por isso, o primeiro experimento deste trabalho consiste em refazer esse estudo na arquitetura alvo utilizada.

Para se efetuarem as estimativas dos custos das alocações não foi encontrada uma ferramenta que apresentasse o modelo de abstração necessário. Dessa forma, um novo modelo foi proposto e implementado no simulador denominado Serpens.

Em relação aos algoritmos de alocação, foram implementadas três modificações em seqüência no modelo de *bin-packing* convencional. A alocação inicial utilizando *bin-packing* convencional apresentou desempenho extremamente precário, com perdas de *deadlines* de 84 % para a heurística *Worst-Fit*, passando para perdas em torno de 16% na alocação final, praticamente com o mesmo consumo de energia.

Apesar de o ganho de energia não ter sido substancial, a vantagem desta técnica vem da possibilidade de realmente se utilizar uma NoC de processadores com menor freqüência para substituir um único processador com freqüência proporcionalmente maior, permitindo dessa forma ganhos de VS, sem que isso seja feito através de alocação estática.

1.4 Estrutura do texto

Este trabalho está dividido da seguinte forma:

O Capítulo 2 apresenta os conceitos de escalonamento de tarefas em sistemas de tempo real embarcados e as técnicas atualmente utilizadas para minimização do consumo de energia, em nível de sistema, durante o escalonamento.

O Capítulo 3 é complementar ao Capítulo 2, no sentido de apresentar as técnicas de redução do consumo de energia através de técnicas de alocação. Também são apresentados os conceitos de redes em-*chip* e as principais técnicas de alocação utilizadas em sistemas distribuídos convencionais, principalmente do tipo malha.

A proposta desenvolvida neste trabalho é apresentada no Capítulo 4, onde são melhor detalhados os objetivos, justificativas e metodologia utilizados.

No Capítulo 5 são apresentados os modelos de tarefas, roteadores e processadores utilizados na implementação do simulador Serpens, bem como o modelo de energia utilizado. O Serpens foi desenvolvido especialmente para estimar o consumo de energia resultante de execução de uma alocação, onde as aplicações são representadas por grafos de tarefas.

No Capítulo 6 é feito um estudo da aplicação de *bin-packing* para a alocação de tarefas no contexto deste trabalho, visando redução do consumo de energia, contudo, mantendo um nível de eficiência aceitável que justifique a distribuição de tarefas entre processadores. Para tanto, são apresentados quatro experimentos, que implementam modificações sucessivas no modelo original de *bin-packing*, combinando-o com técnicas apresentadas no Capítulo 3. 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, no Capítulo 7 são apresentadas as conclusões obtidas neste estudo.

2 REDUÇÃO DO CONSUMO DE ENERGIA ATRAVÉS DO ESCALONAMENTO

Um sistema embarcado é qualquer dispositivo que inclui um computador programável, não desenvolvido com o propósito de ser um computador de uso geral (WOLF, 2001).

Computadores vêm sendo embarcados em aplicações desde o surgimento da computação. Um exemplo é o Whirlwind projetado no MIT por volta da década de 40, originalmente concebido como simulador de vôo. O Whirlwind foi também o primeiro computador com suporte a operações de tempo real (WOLF, 2001).

Desde os anos 70, a tecnologia VLSI – *Very Large Scale Integration* tornou possível acomodar uma CPU completa em uma única pastilha. Esse dispositivo passou a ser chamado de microprocessador.

Nos dias de hoje, há uma ampla variedade de microprocessadores disponíveis, desde processadores de uso geral, até aqueles especializados em executar certo tipo de processamento, como processamento de sinais.

À medida que os sistemas se tornam mais complexos e incorporam mais funcionalidades passam a consumir mais energia. Esse aumento de consumo tem como principal efeito a redução de autonomia de dispositivos alimentados por baterias; porém, dispositivos como computadores de mesa também sofrem as conseqüências do aumento do consumo de potência dos processadores, principalmente no que diz respeito à dissipação de calor.

2.1 Modelo de potência em circuitos CMOS

A potência de um processador é a quantidade de energia que ele consome por unidade de tempo. Atualmente, a maioria dos circuitos digitais são construídos com tecnologia CMOS – *Complementary Metal Oxide Semiconductor*, cujo consumo de potência apresenta três componentes (MUDGE, 2001):

$$Pot = Pot_{din} + Pot_{estat} + Pot_{curto} \tag{2.1}$$

onde Pot_{din} é a potência dinâmica, Pot_{estat} é a potência estática e Pot_{curto} é a potência de curto circuito.

A potência dinâmica mede o consumo de potência causado pela carga e descarga da capacitância das portas do circuito, dada pela seguinte equação:

$$Pot_{din} = \alpha C V_{dd}^2 F \tag{2.2}$$

onde α é a fração de portas lógicas trocando de valor neste instante, C é a capacitância de carga de uma porta, F é a freqüência de operação e V_{dd} é a tensão de alimentação do circuito.

A potência estática é aquela perdida com a corrente de fuga I_{leak} que ocorre estaticamente para manter o estado das portas:

$$Pot_{estat} = V_{dd} I_{leak} \tag{2.3}$$

A potência de curto circuito é o consumo decorrido da corrente de curto circuito I_{short}, que flutua entre V_{dd} e terra durante τ instantes, quando a saída de uma porta lógica muda de valor:

$$Pot_{curto} = \tau \alpha V_{dd} I_{short} F \tag{2.4}$$

A corrente de curto circuito é geralmente ignorada porque é relativamente pequena e acaba sendo absorvida pela potência dinâmica.

A freqüência F de um circuito é definida em função do desempenho necessário e limitada pelo caminho crítico do circuito. O V_{dd} respectivo a um dado valor de F é dado pela seguinte relação (KIM, N. S. et al., 2003):

$$F \propto \frac{(V - V_{th})^a}{V} \tag{2.5}$$

onde V é a tensão de alimentação do transistor, V_{th} é a tensão de *threshold* e o expoente a é uma constante experimentalmente derivada, que para a tecnologia 100 nm é de aproximadamente 1.3 (KIM, N. S. et al., 2003).

A partir da relação 2.5, obtém-se a equação 2.6 para o cálculo de V proporcional a F. Para isto, ambos são normalizados em função de seus máximos V_{max} e F_{max}, obtendo-se respectivamente $V_{norm} = V / V_{\max}$ e $F_{norm} = F / F_{\max}$.

$$V_{norm} = \beta_1 + \beta_2 F_{norm} \tag{2.6}$$

com $\beta_1 = V_{th} / V_{max}$ e $\beta_2 = 1 - \beta_1$. Para a tecnologia 100 nm β_1 é estimado em 0.3.

Tabela 2.1: Valores de k_{design} e N (Número de bits total).

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

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

$$Pot_{estat} = V_{dd} \cdot N \cdot k_{design} \cdot I_{leak} \quad (2.7)$$

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 2.1.

2.2 Técnicas de redução do consumo de energia em nível de sistema

As técnicas de redução de consumo de energia em nível de sistema surgiram baseadas na premissa de que os sistemas possuem cargas de trabalho não uniformes no decorrer do tempo; dessa forma, se o uso dos recursos do sistema for gerenciado corretamente no decorrer do tempo, ganhos significativos em economia de energia podem ser alcançados (CHANDRAKASAN; BRODERSEN, 1995).

São duas as principais técnicas em nível de sistema para minimização do consumo de energia: VS – *Voltage Scaling* (WEISER et al., 1995) que seleciona a tensão de operação de acordo com os requerimentos de desempenho da aplicação; e PM – P*ower Management* (BENINI; BOGLIOLO; MICHELI, 2000), que consiste em desligar processadores ou componentes do sistema ociosos. Ambas as técnicas consistem em regular a disponibilidade de recursos, como capacidade de processamento, para evitar desperdício de energia. Todavia, o escalonador do sistema pode não apenas aproveitar essas oportunidades no escalonamento, mas pode também escalonar as tarefas de forma a maximizar sua ocorrência.

Escalonar um conjunto de tarefas em um processador é determinar quando e qual tarefa executa e assim determinar a ordem de execução das mesmas. No caso de sistemas com múltiplos processadores, isto implica também em determinar um mapeamento de tarefas para processadores específicos (CHENG, 2002).

Intuitivamente, escalonar para PM é tornar concentrados e longos os períodos ociosos do escalonamento. Desse modo, o processador fica desligado mais tempo e desperdiça menos com o *overhead* de desligar e religar. Ao contrário, com VS tenta-se diluir os períodos ociosos para atenuar os picos de utilização, de forma que o processador possa trabalhar sempre com a freqüência mais baixa possível e com máxima uniformidade no decorrer do tempo.

Tipicamente, o PM implementa um procedimento de controle baseado em algumas observações ou suposições sobre a carga de trabalho, que é chamado de política. Um exemplo de uma política simples utilizada em *laptos* e *palmtops* é o *timeout*, que seleciona um componente para ser desligado após um período de inatividade.

A técnica de VS baseia-se no fato de que a potência dinâmica domina o consumo de energia de um circuito, na qual o V_{dd} tem um peso quadrático (Equação 2.2), sugerindo que o meio mais efetivo de economia de energia é reduzir o V_{dd}. Como F deve ser reduzida proporcionalmente, obtém-se redução linear no desempenho e redução quadrática na energia dinâmica consumida (WELCH, 1995).

O controle da freqüência de operação do processador pode ser feito via software, se este possuir instruções para tal. Muitas vezes, apenas um conjunto discreto de

freqüências é oferecido e, assim, uma única freqüência pode não ser ótima para a execução de uma tarefa (ISHIHARA; YASUURA, 1998). Nesse caso, são três as soluções geralmente adotadas: arredondar para a freqüência mais próxima (LORCH; SMITH, 2001); arredondar para a maior freqüência discreta (YUAN; NAHRSTEDT, 2003); ou converter para uma combinação das duas, executando a tarefa em duas freqüências diferentes (GRUIAN, 2001).

2.3 Escalonamento de Tempo Real

Muitos sistemas embarcados apresentam restrições temporais, do tipo prazo de resposta, além da restrição de consumo de energia. Esse tipo de restrição transforma o momento em que a resposta do sistema é dada numa componente da própria solução. Esses sistemas são chamados de sistemas de tempo real (LI, Q.; YAO, 2003).

Sistemas de tempo real são sistemas de computação que monitoram, respondem ou controlam um ambiente externo. Neste contexto, uma de suas características mais importantes é a previsibilidade de seu comportamento, visando garantir, principalmente em tempo de projeto, que os prazos serão atendidos durante a execução (SHAW, 2003).

Os *deadlines* das tarefas de um sistema de tempo real podem ser classificados em críticos (*hard real time*) e não críticos (*soft real time*). As restrições críticas devem obrigatoriamente ser atendidas, pois o sistema falha se uma delas atrasar. Em contraste, a perda de *deadlines* não críticos não é desejável, mas se ocorrer não compromete o sistema, apenas sua performance vai degradando à medida que mais tarefas não críticas finalizam atrasadas (CHENG, 2002).

Em relação a seu padrão de liberação, as tarefas de um sistema de tempo real podem ser classificadas em periódicas e aperiódicas, sendo que as tarefas periódicas possuem períodos de repetição e *deadlines* estabelecidos.

As tarefas aperiódicas podem ser também tarefas esporádicas quando apresentam um intervalo mínimo entre duas liberações consecutivas. Dessa forma, não se sabe a cada quanto tempo essas tarefas podem ser liberadas, pois geralmente dependem de eventos externos, contudo, um pior caso fica definido (BURNS, 1997).

2.3.1 Modelo de tarefas periódicas

Uma aplicação composta por um conjunto de tarefas pode ser representada por um grafo de tarefas acíclico (DAG – *Direct Acyclic Graph*) $G = (K, A)$, onde cada nodo $k_i \in K$ representa uma tarefa 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.

No modelo periódico, cada tarefa k_i apresenta um pior tempo de execução C (WCET – *Worst Case Execution Time*), um período de repetição T e um *deadline* D, todos conhecidos em tempo de projeto (BURNS, 1997).

Na Figura 2.1, um exemplo de escalonamento de tarefas periódicas é apresentado, onde cada tarefa consome exatamente seu WCET, que caracteriza um escalonamento canônico. Por simplificação, neste exemplo todas as tarefas possuem os mesmos períodos e *deadlines*.

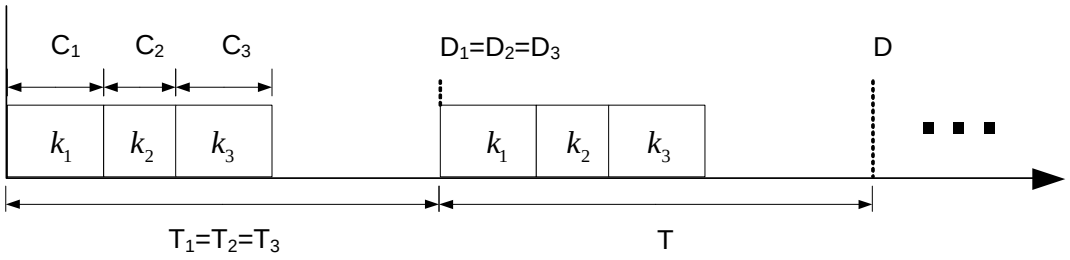

Figura 2.1: Escalonamento periódico.

2.3.2 Escalonamento

A maioria dos sistemas operacionais de tempo real utiliza escalonamento preemptivo baseado em prioridades. Esses sistemas atribuem a cada tarefa um nível de prioridade único. O escalonador é o responsável por garantir que entre as tarefas prontas para executar, aquela que está executando é a que tem a maior prioridade. Para isso, o escalonador pode preemptar uma tarefa de menor prioridade no meio de sua execução (BURNS, 1997).

Quando as prioridades das tarefas são determinadas em tempo de execução, diz-se que o escalonamento é dinâmico, ou baseado em prioridades variáveis. Caso contrário, se as prioridades são definidas durante a fase de projeto, o escalonamento é dito estático ou baseado em prioridades fixas.

A teoria de escalonamento por prioridades fixas iniciou com a publicação do algoritmo RM – *Rate-Monotonic*, uma heurística simples de atribuição de prioridades, que se dá em ordem inversa do período das tarefas. Assim, quanto maior o período da tarefa, menor é sua prioridade: $T_i < T_j \Rightarrow P_i > P_J$ (LIU; LAYLAND, 1973). Outra possibilidade é atribuir prioridades em função do *deadline* ao invés do período, caracterizando o algoritmo DM – *Deadline Monotonic*, ou seja: $D_i < D_j \Rightarrow P_i > P_j$. Se os *deadlines* são iguais aos períodos então DM e RM geram o mesmo escalonamento (AUDSLEY et al., 1993).

Os algoritmos de escalonamento de tempo real baseados em prioridades dinâmicas calculam as prioridades das tarefas *on-line*, sendo dessa forma mais adaptáveis à mudanças no sistema, como por exemplo variação de carga. Os algoritmos clássicos são LLF – *Least-Laxity-First* e EDF – *Earliest-Deadline-First* (LIU; LAYLAND, 1973).

O EDF atribui maior prioridade para a tarefa com o *deadline* absoluto mais próximo, ou seja: $DA_i < DA_j \Rightarrow P_i > P_J$. Uma variação do EDF é o LLF que atribui a prioridade em função da folga L da tarefa, dada pela diferença entre o tempo que a tarefa necessita e seu próximo *deadline*: $L = DA - C_{restante}$. Assim, a tarefa com menor folga tem a maior prioridade: $L_i < L_j \Rightarrow P_i > P_J$. Se $L < 0,$ então a tarefa não será concluída antes de seu *deadline*.

Os algoritmos EDF e LLF são reconhecidamente ótimos se as tarefas são escalonadas em apenas um processador, não compartilham recursos e podem sofrer preempção. Isso significa que se existe um escalonamento do conjunto de tarefas nessas condições, que não viola nenhum *deadline*, este pode ser obtido com um dos dois algoritmos (DERTOUZOS, 1974; MOK; DERTOUZOS, 1978).

Nos algoritmos dinâmicos, uma tarefa pode sofrer interferência de qualquer outra tarefa no sistema, ao contrário do que acontece com os algoritmos estáticos, nos quais uma tarefa somente sofre interferência de outras de maior prioridade. Como conseqüência, nos algoritmos dinâmicos qualquer tarefa pode perder o *deadline*, enquanto que, nos estáticos, as que apresentam maior propensão para isso são as de menor prioridade, que geralmente são também as menos críticas do sistema.

Estudos sobre várias abordagens na implementação de escalonadores baseados em prioridades foram feitos por Katcher; Arakawa; Strosnider (1993). No modelo proposto, um escalonador é composto por duas filas: uma fila de execução que armazena as tarefas prontas para executar; e uma fila de espera onde são armazenadas as tarefas que já executarem em seu período e estão aguardando liberação para o próximo. Adicionalmente, a tarefa que está em um determinado instante executando no processador é chamada de tarefa ativa.

Ao ser liberada, uma tarefa é removida da fila de espera e acrescentada à fila de execução. A primeira é ordenada por instante de liberação, enquanto a segunda é ordenada por prioridade.

Os instantes em que o escalonador é invocado são chamados de pontos de escalonamento. Os pontos de escalonamento geralmente ocorrem em decorrência do término da tarefa ativa, interrupções, ou de chamadas de serviços do sistema operacional. Uma interrupção pode ser causada por uma fonte externa qualquer ou pelo temporizador do escalonador, que é programado para disparar nos instantes das próximas liberações.

Se o ponto de escalonamento é resultado de uma interrupção do temporizador, então vai ocorrer uma liberação, que pode também forçar uma troca de contexto, caso uma das tarefas liberadas tenha prioridade maior que a tarefa ativa. Porém, se o ponto de escalonamento foi gerado pela finalização da tarefa ativa, então esta segue para a fila de espera, enquanto a tarefa da cabeça da fila de execução se torna a tarefa ativa.

Alguns rótulos podem ser acrescentados a este modelo de escalonador, como na Figura 2.2. O rótulo A define o instante em que ocorre uma liberação da tarefa k_j; o intervalo $[B_i, E_i]$ representa o período pelo qual a tarefa k_j se tornou a tarefa ativa e em seguida foi interrompida ou finalizada pela i-ésima vez; e o *deadline* absoluto DA marca o prazo de finalização de k_j.

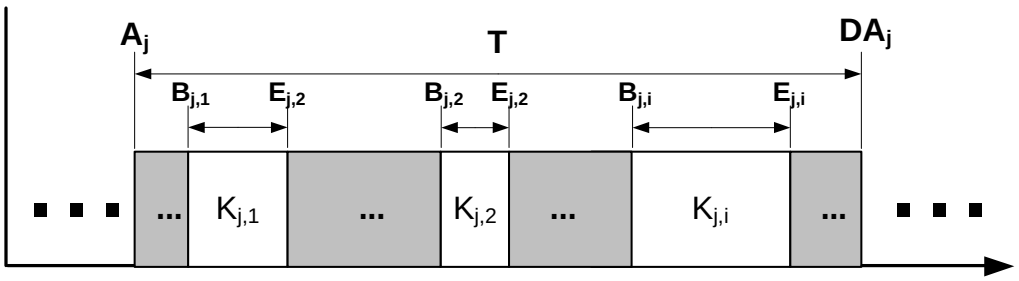

Figura 2.2: Atributos de uma tarefa no escalonamento.

2.3.3 Análise de escalonabilidade

Da mesma forma que o escalonamento, a análise de escalonabilidade também pode ser realizada tanto em tempo de projeto quanto em tempo de execução.

Analisar a escalonabilidade de um sistema é verificar se suas tarefas podem ser escalonadas respeitando-se as respectivas restrições temporais (CHENG, 2002). Para isso um teste de escalonabilidade ou controle de admissão é aplicado.

Segundo Cheng (2002), são duas as principais abordagens utilizadas para se analisar a escalonabilidade de sistemas de tempo real: análise baseada na utilização do processador e análise baseada no pior tempo de resposta (WCRT – *Worst-Case Response Time*).

Dado um conjunto de n tarefas periódicas e preemptíveis $K = \{k_1, ..., k_n\}$, cada tarefa k_i será ativada no sistema a uma taxa de $1/T_i$ e executará no máximo C_i unidades de tempo. Em outras palavras, cada tarefa utilizará no máximo $u_i = C_i/T_i$ do potencial do processador. Assim, o fator de utilização total do conjunto de tarefas é dado por:

$$u = \sum_{i=i}^{n} (\frac{C_i}{T_i}) \tag{2.8}$$

A Equação 2.8 é a forma geral da análise de escalonamento baseada na utilização do processador. Por tratar-se de uma função matemática simples, pode ser calculada em tempo de execução e ser assim usada como teste de escalonabilidade em uma abordagem *on-line*.

Liu; Layland (1973) mostraram que, para o EDF, o fator de utilização u é de 100%, ou seja, se $u \leq 1$ então as n tarefas são escalonáveis. Para o RM o limite de u é de aproximadamente 70%.

O teste de escalonabilidade baseado no WCRT é composto por dois estágios, em que no primeiro se calcula o WCRT de cada tarefa (R_i), para, em seguida, se compará-los aos respectivos *deadlines*. Se $R_i < D_i$ para todas as tarefas, então o conjunto de tarefas é escalonável. Esse tipo de análise exige que as prioridades sejam definidas em tempo de projeto.

2.4 Algoritmos de DVS para Tempo Real

Os primeiros trabalhos de DVS – *Dynamic Voltage Scaling* foram feitos para sistemas sem restrições temporais, como Weiser *et al.* (1995). 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.

Na presença de restrições temporais, o algoritmo de escalonamento de tensão deve baixar a freqüência do processador enquanto ainda cumpre os *deadlines*.

Para sistemas de tempo real com cargas de trabalho previsíveis, ou conhecidas a priori, o VS pode explorar completamente os compromissos entre potência e desempenho. Infelizmente, na maioria das situações reais não há tal conhecimento a priori. A carga de trabalho, ao contrário, é imprevisível em muitos sistemas, desde simples telefones celulares e PDAs – *Personal Digital Assistant*, até SoCs mais complexos.

Segundo Kim, W. *et al.* (2002), os algoritmos de DVS são classificados em: inter-tarefas e intra-tarefas.

2.4.1 Intra-tarefas

Os algoritmos intra-tarefas aplicam DVS durante a execução da tarefa. Cada uma delas recebe uma freqüência de execução em função do caminho crítico no seu grafo de execução (laços de repetição e desvios condicionais). Naturalmente, em tempo de execução, outros caminhos podem ser tomados que não o crítico, exigindo que a execução seja monitorada e a freqüência alterada de acordo com os caminhos tomados (KIM, W. et al., 2002).

Os dois métodos intra-tarefas mais comuns são os baseados em caminhos e os estocásticos. No método estocástico calcula-se a freqüência em função de uma distribuição de probabilidade. Geralmente, uma velocidade inicial mais baixa é utilizada, pois se a tarefa executar menos que o WCET não será necessário aumentar a freqüência mais tarde (KIM, W. et al., 2002).

2.4.2 Inter-tarefas

Os algoritmos inter-tarefas ajustam a freqüência de cada tarefa apenas uma vez, no início de sua execução. Caso a tarefa termine antes do WCET, o tempo restante é distribuído para as próximas. Dessa forma, o algoritmo é composto de duas fases: estimativa de folga e distribuição de folga (KIM, W.; KIM, J.; MIN, 2002).

A estimativa de folgas consiste em calcular quanto de tempo há disponível para as tarefas no escalonamento, enquanto que a distribuição de folgas trata de sua distribuição, tentando manter a tensão mais uniforme possível no decorrer do tempo.

As fontes de folga em um escalonamento são duas: estática e dinâmica. A folga estática é a diferença entre o WCET do escalonamento e o *deadline*, enquanto que a folga dinâmica é aquela surgida em tempo de execução pelo fato de as tarefas executarem menos que o WCET.

As três principais técnicas para a estimação de folgas dinâmicas, durante o escalonamento, são:

- Até a liberação da próxima tarefa: A folga é dada como sendo a diferença entre o tempo atual e a chegada prevista da próxima tarefa.
- Baseada em prioridades: Tarefas de menor prioridade podem receber folgas deixadas por tarefas de maior prioridade.
- Por utilização do processador: Calcula a folga em função da carga atual do processador, dada pela sua utilização. Nesse caso, tarefas que não estão prontas para rodar não entram no cálculo da utilização.

As técnicas mais comuns de distribuição de folgas estáticas são: distribuição uniforme em função do WCET, chamada de SPM – *Simple Power Management*; e *greedy*, no qual a folga total é alocada para a próxima tarefa, deslocando todas as demais em direção ao *deadline* (ZHU; MELHEM; CHILDERS, 2001).

Os principais algoritmos de DVS inter-tarefas são apresentados nas próximas seções.

2.4.2.1 AVR

O algoritmo AVR – *Average Rate Heuristic* proposto por Yao; Demers; Shencker (1995), baseia-se na premissa de que sistemas de tempo real possuem tarefas com WCET C_i 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 AVR pode ser aplicado estaticamente com o RM em função do intervalo crítico. A intensidade $g(I)$ de um intervalo qualquer $I=[z,z']$ (Equação 2.9) é dada pela razão entre a soma do WCET das tarefas que estão contidas em I e o seu comprimento.

$$g(I)=\frac{\sum C_j}{z'-z} \mid [A_j, DA_j]\subseteq[z,z'] \tag{2.9}$$

Se $I^*=[z,z']$ é um intervalo que maximiza $g(I)$ então I^* é um intervalo crítico e $g(I^*)$ é um valor normalizado entre [0,1] que representa a utilização máxima do processador neste intervalo. Por conseqüência $g(I)$ pode ser usado como F_{norm} na Equação 2.6.

O AVR pode ser usado para efetuar o mesmo processo *on-line*, dessa vez aplicando a Equação de Utilização do Processador (Equação 2.8).

No AVR a distribuição de folga é feita com algoritmo *greedy,* como demonstrado na Figura 2.3, ou seja, a folga final é distribuída proporcionalmente entre as demais tarefas, "esticando" o escalonamento original.

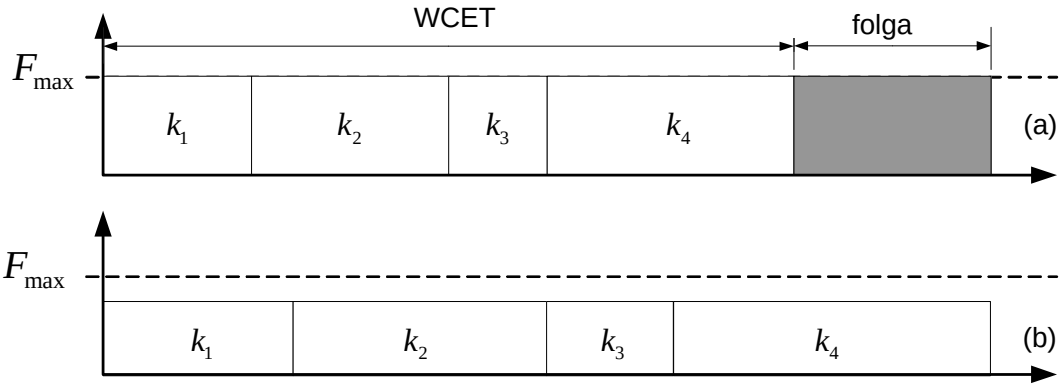

Figura 2.3: (a) Escalonamento antes (b) Escalonamento após AVR.

2.4.2.2 LPFPS

Existem vários trabalhos de análise de folgas em escalonamento de tempo real para acomodar nelas tarefas aperiódicas ou esporádicas, sem comprometer a escalonabilidade do sistema. Essas folgas podem também ser utilizadas para baixar a velocidadedo processador, como proposto pelo algoritmo LPFPS – *Low Power Fixed Priority Scheduling* (SHIN, Y.; CHOI, 1999).

O LPFPS é aplicado quando a fila de execução está vazia, podendo haver tarefa ativa ou não. Caso não haja, o processador é posto em modo inativo e programado para acordar na próxima liberação de tarefa. Se existir tarefa ativa, então o VS é aplicado.

Como se verifica na Figura 2.4 (b), o LPFPS aplica a estimativa de folga somente na última tarefa do escalonamento, quando a lista de execução ficou vazia. Então toda a folga existente é atribuída para a tarefa ativa (SPM). Mas se a tarefa ativa executar menos que o WCET então o processador entrará em modo inativo.

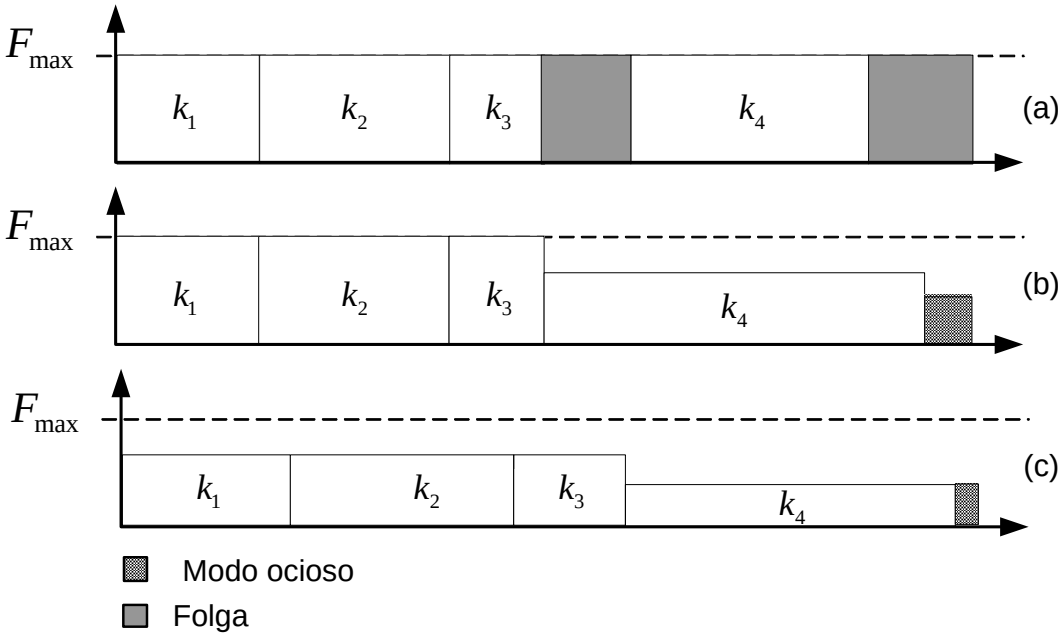

Figura 2.4: (a) Escalonamento original (b) LPFPS (c) LPPS.

A atribuição da folga para a última tarefa é feita calculando-se F_{norm} de forma a estender a execução da tarefa ativa até a liberação da próxima tarefa. A seguinte heurística foi proposta:

$$r_{heu} = \frac{C_i - C_i^e}{t_a - t_c} \tag{2.10}$$

onde C_i^e é a quantidade de tempo que a tarefa já executou, t_c é o tempo atual e t_a é o momento da liberação da próxima tarefa.

Apesar de aplicado em um escalonamento de prioridades fixas, o LPFPS pode ser aplicado dinamicamente e ser assim combinado com EDF.

2.4.2.3 LPPS

O algoritmo LPPS – *Low-Power Priority Based Real-Time Scheduling* (SHIN, Y.; CHOI; SAKURAI, 2000), apresentando na Figura 2.4 (c), é uma combinação dos algoritmos LPFPS e AVR.

No LPPS, primeiramente, a folga estática é distribuída uniformemente entre as tarefas com *greedy*, como feito no AVR (Figura 2.3). Em seqüência, a folga dinâmica toda é entregue para a última tarefa do escalonamento a executar (SPM) com a heurística r_{heu} do LPFPS (Equação 2.10).

2.4.2.4 LA

O LA – *Look-ahead* EDF tenta atingir maiores ganhos de energia tentando determinar a computação futura necessária, e assim poder adiar a execução de tarefas, absorvendo agora a folga futura prevista, ou seja, aplicando SPM (PILLAI; SHIN, 2001).

As técnicas anteriores executam numa freqüência maior prevendo o WCET e, quando as tarefas completam antes, diminuem a freqüência para compensar. Já o LA deixa para executar com maior freqüência no final do escalonamento, mas como as tarefas tendem a executar menos que o WCET, o uso de altas freqüências acaba não sendo necessário, permitindo que o sistema continue operando em baixa freqüência.

Basicamente, o algoritmo aloca os tempos de execução WCET para as tarefas no escalonamento em ordem inversa do EDF, ou seja, reserva as fatias de tempo a partir da última tarefa no escalonamento para a primeira, deixando toda a folga do escalonamento para ser absorvida pela tarefa atual.

2.4.2.5 LPSEH

A eficiência de um algoritmo de escalonamento de tensão está diretamente ligada a sua capacidade de perceber e utilizar as folgas de escalonamento. Tentando obter o máximo na estimação de folgas, o LPSEH – *Low-Power Scheduling using Slack Estimation heuristic* (KIM, W.; KIM, J.; MIN, 2002) é um algoritmo agressivo que calcula a folga S_H deixada pelas tarefas de maior prioridade e tenta estimar a folga S_L que será deixada pelas tarefas de menor prioridade.

O algoritmo define U^{rem} como sendo a quantidade de tempo disponível e W^{rem} o WCET restante de uma tarefa ainda não completada, utilizados para o cálculo de S_H da seguinte forma:

$$S_H = \sum_{k_i \in K_H} U_i^{rem} \tag{2.11}$$

onde K_H são as tarefas de maior prioridade já completadas.

Em um primeiro momento, o tempo disponível para a tarefa ativa k_a executar é dado por $S_H + U_a^{rem}$, sendo que a tarefa pode ser estendida com a seguinte freqüência normalizada:

$$F_{norm} = \frac{W_a^{rem}}{U_a^{rem} + S_H} \tag{2.12}$$

Com essa freqüência, a tarefa k_a é "esticada" até o tempo $t_i = t_{atual} + S_H + U_a^{rem}$, a não ser que uma tarefa de maior prioridade seja liberada nesse intervalo. Porém, estimar a interferência de tarefas de maior prioridade é muito custoso e, por isso, essa possibilidade não é considerada.

Caso não exista outra tarefa na fila de tarefas prontas, além da tarefa ativa k_a, esta pode ser estendida até a próxima liberação; ou seja, até t_i, como no LPFPS. Todavia, se existir pelo menos uma tarefa na fila de tarefas prontas, esta ou a tarefa com maior

prioridade entre elas será a próxima tarefa ativa k_b, e sua folga é estimada da seguinte forma:

$$S_L = U_b^{rem} - W_b^{rem} + \sum_{k_i \in T_L} U_i^{rem} \quad (2.13)$$

onde T_L é o conjunto de tarefas que possuem prioridade e *deadline* menores ou iguais aos de k_b, que serão completadas até t_i.

A estimativa de S_L acima só é válida quando não existe uma tarefa j com maior prioridade que a tarefa ativa para ser liberada durante $[t_i, t_i + S_L]$. Caso contrário, o tempo de execução da tarefa ativa está restrito ao momento da chegada dessa tarefa denominado A_j tal que: $S = A_j - t_i$.

Com a folga S calculada, a freqüência normalizada do processador é dada por:

$$F_{norm} = \frac{W^{rem}}{S} \quad (2.14)$$

Experimentos mostraram que o LPSEH reduz o consumo de energia em 40% comparado ao LPPS e é o mais próximo do ótimo entre os anteriores.

2.4.2.6 DAR

O algoritmo DAR – *Dynamic Average Rate* (ZHUO; CHAKRABARTI, 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 da Equação 2.8, para o cálculo da freqüência normalizada, o DAR usa a densidade S da tarefa dada por:

$$S_i = \frac{C_i - C_i^e}{DA_i - t_a} \quad (2.15)$$

onde DA_i é o *deadline* absoluto da tarefa, t é o instante atual e C_i^e é a fatia do WCET já executado. Na prática, a densidade S_i é a 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:

$$F_{norn} = \sum_{j \in Tarefas\,\mathrm{Pr}ontas} S_j \quad (2.16)$$

Segundo os autores, o DAR apresenta menor custo computacional que o LPSEH e melhor desempenho em termos de economia de energia.

3 REDUÇÃO DO CONSUMO DE ENERGIA ATRAVÉS DA ALOCAÇÃO

Com o avanço das tecnologias de fabricação de circuitos integrados VLSI, está cada vez mais comum integrar vários elementos de hardware em um único *chip* (SoC – *System on-chip*), que passa a ser chamado de MPSoC – Multiprocessor SoC se vários desses elementos são processadores.

Entretanto, um MPSoC não é apenas um *chip* composto por múltiplos processadores, viabilizado pela crescente densidade de transistores das tecnologias de fabricação. Mais do que isso, MPSoCs são arquiteturas customizadas, que equilibram as restrições da tecnologia e as necessidades das aplicações, geralmente de um domínio específico (JERRAYA; TENHUNEN; WOLF, 2005).

Como conseqüência dessas restrições, os MPSoCs assumem um caráter altamente heterogêneo e acabam por requerer estruturas de comunicação bastante elaboradas. Duas são as abordagens adotadas atualmente: barramento centralizado e uma abordagem de redes de comunicação, que gerou o conceito de NoC - *Network on-Chip* (BENINI; MICHELI, 2002).

O barramento centralizado é muito utilizada em arquiteturas de computadores clássicas, sendo natural seu uso em MPSoCs. Contudo, apresenta um defeito grave em relação às NoCs: falta de escalabilidade, que torna proibitiva a incorporação de um número massivo de núcleos.

Em termos de eficiência energética, os desafios das estruturas de comunicação em-*chip* são o comprimento das conexões e o número de núcleos conectados a elas. Quanto maior o comprimento dos fios, maior é a quantidade de energia necessária para efetuar a difusão do sinal em toda a sua amplitude. Isso é agravado pela capacitância parasita, que ocorre em decorrência do aumento do número de núcleos conectados ao barramento e que implica no aumento significativo no atraso da movimentação de dados.

As NoCs compartilham muitos conceitos com redes de interconexão de computadores paralelos e redes locais (LAN – *Local Area Network*), contudo existem grandes diferenças no que diz respeito ao número de componentes e, principalmente, às distâncias entre eles.

Essas peculiaridades favorecem o desenvolvimento de protocolos particulares às redes em-*chip*.

3.1 Redes em-*chip*

Uma NoC é constituída por roteadores e pelos enlaces que os conectam, de acordo com a topologia da rede e pode ser descrita pela suas estratégias de roteamento, controle de fluxo, chaveamento, arbitração e armazenamento de pacotes:

- Roteamento: É o mecanismo pelo qual se escolhe um canal de saída para um pacote que chegou em uma porta de entrada;
- Controle de fluxo: É o mecanismo que regula o tráfego dos pacotes chegando e saindo nos canais do roteador;
- Chaveamento: Define como uma mensagem passa através do roteador,
- Arbitração: Decide quando um chaveamento pode ser efetivado;
- Armazenamento: Determina como e onde as mensagens aguardam até deixarem o roteador.

3.1.1 Roteador

Um roteador pode ser definido como um dispositivo que conecta um número de canais de entrada a um número de canais de saída, com a função de transferir as informações entre ambos, de forma que os pacotes possam efetivamente trafegar pelos enlaces, cada um até seu destino (NI; MCKINLEY, 1993).

Geralmente, um roteador é composto por módulo de controle de chaveamento, roteamento interno e filas de armazenamento de entrada e/ou saída. A estrutura de um roteador genérico é ilustrada na Figura 3.1.

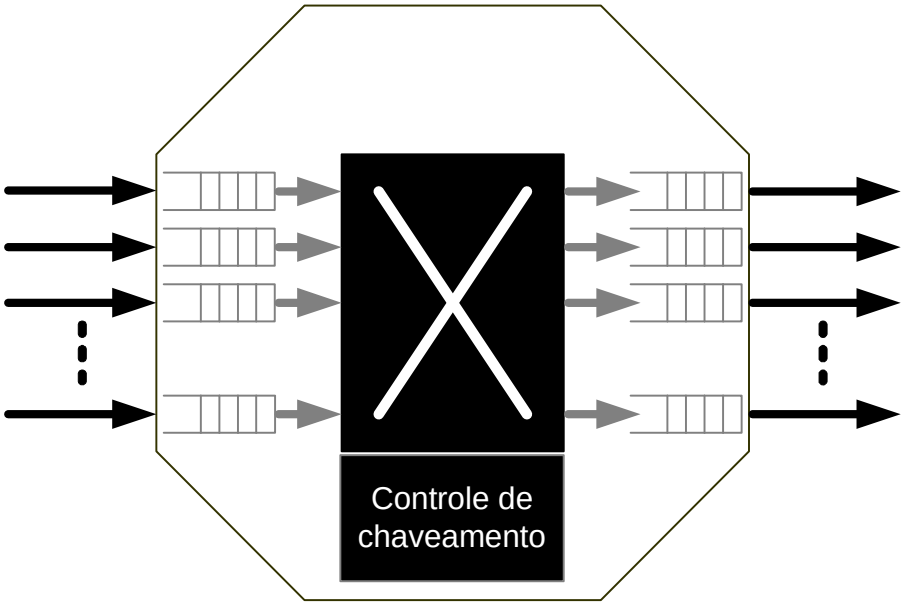

Figura 3.1: Modelo de um roteador genérico.

O roteador é o elemento principal da NoC, logo, seu impacto na área final do SoC deve ser minimizado. O componente que mais influencia em seu consumo, tanto de área quanto de energia, são as filas de armazenamento temporário. Ao mesmo tempo, elas impactam diretamente no desempenho da rede.

O roteador pode ser implementado de forma centralizada ou distribuída. Na abordagem distribuída, chaveamento, roteamento interno e arbitragem são implementados de forma independente para cada porta.

3.1.2 Enlace

Um enlace liga um roteador a um núcleo, ou a outro roteador, podendo ser constituído de um ou dois canais físicos de comunicação. Tipicamente, os enlaces utilizados em redes de interconexão são bidirecionais constituídos por dois canais unidirecionais opostos, de modo a permitir a transferência simultânea de informação nas duas direções do enlace.

3.1.3 Mensagens, pacotes e *phits*

Uma mensagem geralmente é dividida em vários pacotes. Esta unidade de informação contém detalhes sobre o roteamento e seqüenciamento dos dados, separadas em três partes: cabeçalho, carga útil e terminador.

O cabeçalho inclui informações de roteamento e de controle, utilizadas pelo roteador para propagar o pacote em direção ao destino da comunicação. Junto com o terminador, eles formam um envelope ao redor da carga útil. O terminador pode incluir informações usadas para a detecção de erros e para a sinalização de fim do pacote.

Para ser transmitido, um pacote é dividido em uma seqüência de unidades cuja largura é igual à largura física do canal, a qual é denominada *phit – PHysical UnIT*, que nada mais é senão a quantidade de bits de dados transmitidos simultaneamente, sendo igual a um nos enlaces seriais e a n nos enlaces paralelos de n bits. O *phit*, além da carga útil, também inclui bits extras para sinalização do seu fluxo.

3.1.4 Topologias

A topologia de uma rede de interconexão é caracterizada pela forma como os roteadores desta são interligados. Essa estrutura pode ser representada por um grafo $G = (R, L)$, onde cada $r_i \in R$ é um roteador e cada $l_{i,j} \in L$ representa um enlace de comunicação ligando os roteadores r_i e r_j.

Cada roteador possui ligações ponto-a-ponto para um determinado número de roteadores vizinhos. A maioria das topologias utilizadas em NoCs são do tipo redes diretas, onde cada núcleo está associado a um roteador, formando um par que pode ser referenciado simplesmente pelo termo nodo (ZEFERINO; SUSIN, 2003).

Tendo em vista que o custo de uma rede totalmente conectada é proibitivo, a grande maioria das implementações restringe-se ao uso de topologias ortogonais. Uma rede apresenta topologia ortogonal se, e somente se, seus nodos podem ser arranjados em um espaço n-dimensional e cada enlace entre nodos vizinhos produz um deslocamento em uma única dimensão.

As topologias de redes diretas ortogonais mais utilizadas são: malha, toróide e hipercubo (Figura 3.2).

Numa rede parcialmente conectada, uma mensagem trocada entre dois nodos não-vizinhos deve obrigatoriamente passar por pelo menos um nodo intermediário, sem a interferência do núcleo associado a ele. A fim de decidir para qual nodo vizinho uma

mensagem deve ser repassada para atingir seu destino, um algoritmo de roteamento é utilizado.

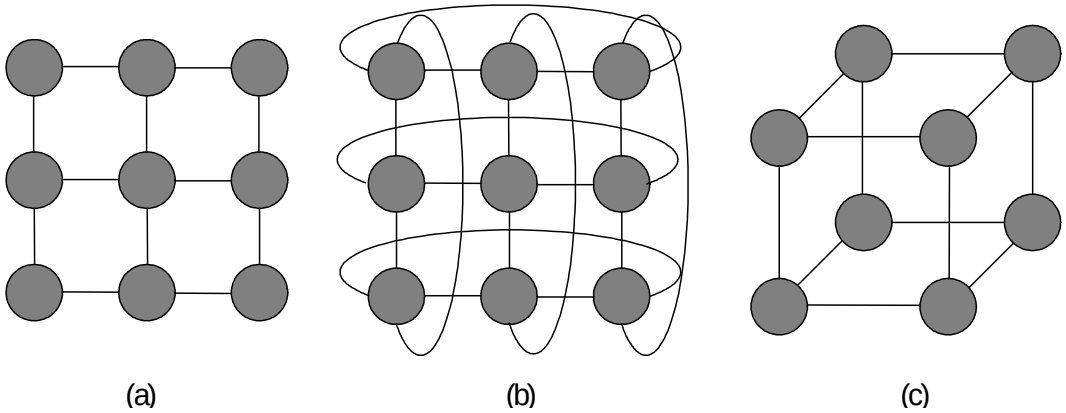

Figura 3.2: Topologias de redes diretas: (a) malha (b) toróide (c) hipercubo.

3.1.5 Roteamento

O algoritmo de roteamento define o caminho a ser utilizado por um pacote a partir do remetente até o destinatário e está fortemente correlacionado à topologia adotada (NI; MCKINLEY, 1993).

A responsabilidade pela definição do roteamento pode ser distribuída, quando o próximo passo é definido a cada roteador visitado; ou centralizada, quando um roteador decide sozinho a sua rota, geralmente o remetente. Este último método tem a desvantagem de necessitar que o cabeçalho do pacote 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 para remetente/destinatário, enquanto que no roteamento adaptativo existem opções de caminhos. Nesta abordagem, o caminho de um pacote é estabelecido dependendo das condições da rede, como tráfego e congestionamento de canais.

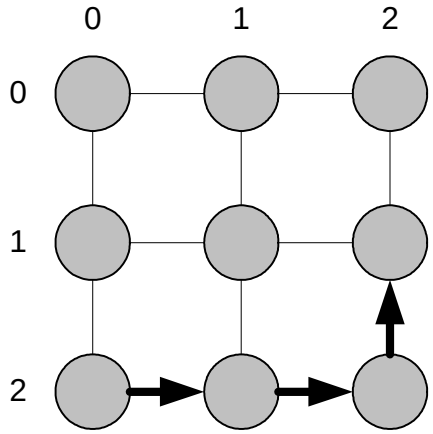

Figura 3.3: Caminho de um roteamento XY entre (0,2) e (2,1).

Um algoritmo de roteamento determinístico bastante utilizado em redes malha é o XY (DUATO; YALAMANCHILI; NI, 1997). Nele, o pacote é roteado em X até atingir a coluna destino e depois em Y até o destino, como demonstrado na Figura 3.3. A

distância entre dois nodos a e b nesse esquema é chamada de distância de *Manhattan* e é dada por:

$$dist = | x_b - x_a | + | y_b - y_a | \quad (3.1)$$

O roteamento adaptativo pode ser ainda classificado como parcialmente e totalmente adaptativo. No totalmente adaptativo é possível rotear um pacote através de qualquer caminho físico existente, enquanto que no 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 por um caminho que o deixa momentaneamente mais longe do destino.

3.1.6 Chaveamento

Um chaveamento consiste na conexão lógica entre uma porta de entrada e uma porta de saída do roteador, que ficam alocadas para a transferência de um ou mais pacotes, podendo ser implementado, basicamente, de duas formas: chaveamento de circuito e chaveamento de pacotes.

- Chaveamento de circuitos: Consiste em estabelecer um caminho fim-a-fim, denominado conexão, formando um circuito virtual. Somente após o fechamento do circuito, os pacotes começam a ser transferidos;
- Chaveamento de pacotes: Neste método, cada pacote é roteado individualmente de acordo com seu cabeçalho, sem a definição de uma conexão; logo, cada pacote pode seguir um caminho diferente.

A vantagem do chaveamento de circuitos é que o roteamento precisa ser efetuado apenas uma vez por mensagem em cada roteador; entretanto, pode levar à subutilização dos canais reservados e assim ao desperdício de largura de banda, a partir do momento que estes ficam inativos. O que não ocorre com o chaveamento de pacotes, que permite o compartilhamento de canais da rede por pacotes de mensagens diferentes; todavia, o custo de rotear cada pacote individualmente é acrescentado.

O emprego da técnica de chaveamento de pacotes implica no uso de uma política de armazenamento de pacotes nos roteadores. As mais utilizadas são: *store-and-forward*, *virtual cut-through* e o *wormhole (NI; MCKINLEY, 1993):*

- *Store-and-forward*: Cada pacote tem que ser completamente armazenado antes de ser enviado para o próximo roteador, causando latência na entrega. Além disso, é necessário que os roteadores tenham grande capacidade de armazenamento, para suportar em pior caso um pacote de tamanho máximo;
- *Virtual cut-through*: Cada roteador pode enviar um pacote, apenas, a partir do momento que o próximo garanta poder recebê-lo por completo. Assim, no pior caso pode ser necessário armazenar todo o pacote como na técnica anterior;
- *Wormhole*: É uma variação do *virtual cut-through* com menor utilização de buffers. Neste modo, os pacotes são quebrados e transmitidos entre os roteadores em unidades menores denominadas *phits.* Portanto, apenas o *phit* cabeçalho contém informações sobre o roteamento, obrigando os demais

phits que compõem o pacote a seguir o mesmo caminho reservado pelo cabeçalho. Se este não puder avançar na rede em função da contenção de recursos, todos os *phits* restantes são bloqueados ao longo do caminho.

3.1.7 Modelo de energia

Ye; Benini; Micheli (2002) propuseram um modelo de estimativa de energia dinâmica de roteadores de NoCs 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:

$$E_{bit} = E_{S_{bit}} + E_{B_{bit}} + E_{L_{bit}} \tag{3.2}$$

onde $E_{S_{bit}}$, $E_{B_{bit}}$ e $E_{L_{bit}}$ representam a energia consumida na matriz de roteamento, nos *buffers* e no enlace, respectivamente. Com base na equação 3.2, 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:

$$E_{bit}^{i,j} = d \times (E_{S_{bit}} + E_{B_{bit}}) + (d+1) \times E_{L_{bit}} \tag{3.3}$$

Para efeito do cálculo da energia 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:

$$E_{din} = \frac{1}{2} \alpha C V_{dd}^2 \tag{3.4}$$

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

$$E_{phit} = E_{wrt} + E_{arb} + E_{read} + E_{xb} + E_{link} \tag{3.5}$$

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 escrever o *phit* no enlace de saída.

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

$$E_{estat} = I_{leak} \cdot V_{dd} \cdot T \cdot SCALE\_S \cdot \eta \tag{3.6}$$

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 3.7 , $I'_{leak}(i,s)$ representa a corrente de *leakage* por unidade de largura do transistor sobre o comprimento, 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:

$$I_{leak}(i,s) = \frac{W(type(i,s))}{L} \bullet I'_{leak}(i,s) \tag{3.7}$$

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 e são apresentados na Tabela 3.1.

Tabela 3.1: $I'_{leak}(i,s)$ para várias tecnologias.

i	s	Type (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

3.2 Escalonamento de tarefas em sistemas distribuídos

Há duas abordagens principais para se escalonar tarefas em um sistema composto por múltiplos processadores: o escalonamento global e o particionamento (LAUZAC; MELHEM; MOSSÉ, 1998).

No escalonamento global há apenas um escalonador para todos os processadores, que mapeia a execução de cada tarefa para um processador, possivelmente diferente a cada ocorrência da mesma.

O escalonamento global é preferencialmente utilizado em sistemas com memória compartilhada, que oferecem preempção rápida e fácil entre processadores. Nestes sistemas não é necessário transferir código e dados de tarefas entre processadores, já que todos acessam a mesma memória, o que não ocorre em sistemas com memória distribuída, nos quais o *overhead* da migração de tarefas faz com que o escalonamento global deixe de ser atrativo, sendo preferida em seu lugar a técnica de particionamento (LAUZAC; MELHEM; MOSSÉ, 1998).

No esquema de particionamento, as tarefas são alocadas em processadores e todas as ocorrências destas devem ser executadas neste processador. Nessa abordagem, cada processador tem seu próprio escalonador local, que faz o escalonamento de acordo com sua política, como se fosse o único processador do sistema. Desta maneira, o processo de escalonamento fica dividido em duas fases distintas: alocação e escalonamento (local), permitindo o reaproveitamento das técnicas de escalonamento para um único processador, como RM e EDF em sistemas de tempo real.

Embora seja clara a distinção entre alocação e escalonamento, ambos não são completamente independentes, uma vez que o número máximo de tarefas que pode ser alocado em um processador é determinado pelo escalonador local. Em sistemas de tempo real, um teste de aceitação como o de Liu; Layland (1979) é geralmente usado para esse fim.

Segundo Casavant; Kuhl (1988), os algoritmos para escalonamento distribuído de tarefas podem ser classificados pelos seguintes critérios:

- Momento da escolha da máquina: *on-line* ou *off-line*;
- Tipo da distribuição de carga: Balanceamento de carga ou compartilhamento de carga;
- Qualidade da solução: Ótima ou sub-ótima;
- Permanência das tarefas após a alocação: Imutável ou migratória;
- Ordem de mapeamento: Tarefas para processadores ou processadores para tarefas;
- Responsabilidade pelo escalonamento: Centralizada ou distribuída;
- Tipo de participação: Voluntária ou obrigatória.

Um exemplo bastante didático de um algoritmo de escalonamento distribuído é o Leilão (*bidding*), descrito por Tanenbaum (1995), que modela o escalonamento como um sistema econômico formado por compradores e vendedores. Geralmente, o produto desse comércio é a capacidade de processamento dos processadores. Contudo, outros recursos, como memórias e dispositivos específicos, também podem ser negociados. Esse processo de negociação pode variar um pouco, em função de quem inicia o leilão:

- Iniciado pela remetente: Quem oferece a tarefa é um processador sobrecarregado tentando se livrar da carga extra;
- Iniciado pelo destino: Neste caso, um processador ocioso procura por tarefas para executar.

Outra questão a se definir é se os processadores têm participação voluntária, ou se todos são obrigados a participar sempre dos leilões.

Uma abordagem comum em sistemas distribuídos é alocar as tarefas nos próprios processadores onde elas surgirem. Somente a partir do momento que o elemento fica sobrecarregado ou desbalanceado é que tarefas começam a ser relocadas.

Este é um caso típico de compartilhamento de carga, que tem por objetivo evitar que elementos do sistema se sobrecarreguem enquanto outros têm capacidade ociosa. O balanceamento de carga é um caso específico de compartilhamento, que visa a distribuição homogênea da carga entre os elementos do sistema.

Ao se aplicar compartilhamento de carga, passa a ser necessário que as tarefas possam migrar do lugar onde foram alocadas inicialmente. A migração envolve o congelamento da tarefa, salvamento do estado, transferência para um novo elemento e reativação da tarefa. Muitas vezes o ganho com a manutenção do balanceamento não compensa esse esforço extra.

Quando um processador leiloa um grupo de tarefas e toma sozinho a decisão de onde as tarefas serão alocadas, tem-se um escalonamento centralizado. O escalonamento centralizado implica em coletar toda a informação em um ponto do sistema, permitindo uma tomada de decisão global, mas potencialmente criando um gargalo. Algoritmos onde a decisão é distribuída não apresentam esta desvantagem, sendo por isso mais robustos.

A escolha do lance vencedor se dá em função dos objetivos do próprio sistema, variando bastante de sistemas que buscam reduzir o consumo de energia para aqueles que buscam maximizar desempenho.

Em redes do tipo malha, há uma maior dificuldade em se escolher o lance vencedor, porque o custo de comunicação entre processadores e as distâncias entre estes 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.

3.3 Alocação de tarefas em redes malhas

A alocação consiste em uma função de mapeamento de elementos, nesse caso tarefas, para recipientes que representam os processadores. Encontrar uma alocação ótima é reconhecidamente um problema NP-difícil (LEUNG, J. Y. T.; WHITEHEAD, 1982). Por isso várias técnicas foram desenvolvidas para se contornar esse problema.

As principais abordagens do problema de alocação são: *packing* e teoria de grafos. No *packing* não são consideradas dependências de tarefas e/ou comunicação como na abordagem de grafos, a qual geralmente usa um modelo de DAGs.

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 (GAREY, 1979) e admite a possibilidade de que tanto itens quanto recipientes sejam elementos multidimensionais.

Em redes malha, tem-se claramente um caso de alocação bidimensional, onde tarefas podem ser alocadas em um conjunto de processadores contíguos com forma retangular, denominados sub-malhas. Este método de alocação é chamado de alocação de sub-malhas, podendo ou não utilizar técnicas de *bin-packing*.

Os algoritmos de alocação de tarefas baseados em teoria de grafos são classificados em dois grupos, aqueles em que o número de processadores é restrito a priori BNP – *Bounded Number of Processor* e os demais em que o número de processadores não é fixo a priori UNP – *Unbounded Number of Processors*. A maioria dos algoritmos BNP é baseada na técnica de *list scheduling*, enquanto os UNP geralmente são algoritmos de *clustering* (AHMAD; KWOK; WU, 1996)

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*, alocação de sub-malhas, *list scheduling* e *clustering*.

3.3.1 *Bin-packing*

O BP – *bin-packing* (GAREY, 1979) 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 $\sum_{k_i \in B_j} s(k_i)$.

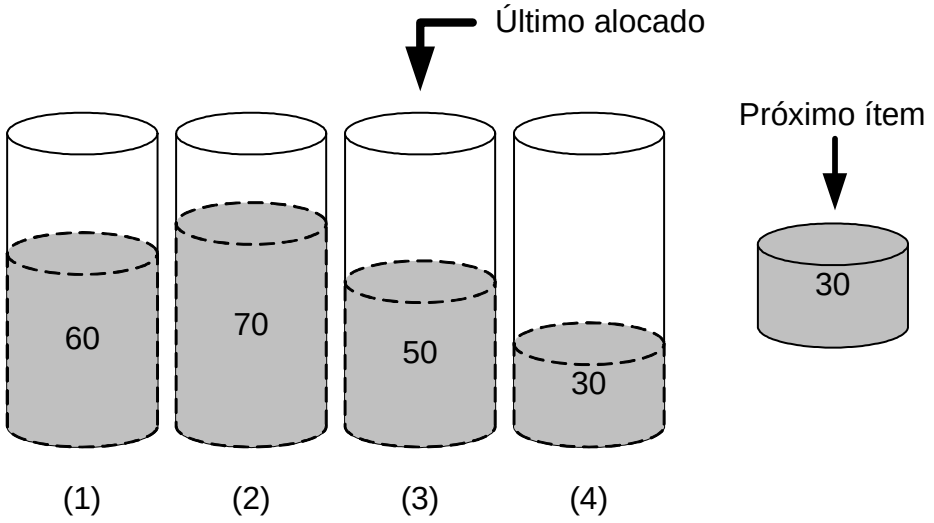

Figura 3.4: Modelo de *bin-packing.*

O modelo de *bin-packing* é representado na Figura 3.4, 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 (GAREY, 1979)*.

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 3.4 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 3.4, 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 3.4 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 $R_A(L) \equiv A(L) \big/ OPT(L)$, 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 (GAREY, 1979).

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.

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 utilizadas. 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; MARTELLO; MONACI, 2002).

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; MARCHETTI-SPACCAMELA, 1984).

Como exemplo, Beck, J.; 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 do tipo barramento centralizado. 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.

3.3.2 Alocação de sub-malhas

A alocação de sub-malhas consiste em particionar uma malha em m sub-malhas que podem ser alocadas para m tarefas ou trabalhos com estruturas retangulares. Este é um problema de alocação bidimensional e, portanto, *bin-packing* pode ser usado para a sua resolução (HWANG, I., 1997).

Nesse modelo, os trabalhos chegam ao sistema, requisitando uma sub-malha de tamanho específico. Se o trabalho não puder executar devido à falta de processadores livres, ou por haver outros trabalhos esperando, ele deve ser enviado para a fila do sistema. No momento em que o trabalho puder ser executado, o escalonador deve colocá-lo na cabeça da fila de espera, sendo então de responsabilidade do alocador providenciar uma partição de processadores para iniciar a execução.

A sub-malha gerada pelo alocador pode ser contígua ou não, dependendo da estratégia utilizada. Embora outras mensagens possam passar através dessa sub-malha, o trabalho a mantém exclusiva até o fim de sua execução, quando deixa o sistema e a libera para os demais trabalhos.

Alocação contígua é a técnica de alocação de sub-malhas utilizada na maioria das máquinas paralelas comerciais. Nessa técnica, as tarefas de uma mesma aplicação ficam fisicamente adjacentes e por este motivo, o sistema sofre de considerável fragmentação. A fragmentação interna ocorre quando mais processadores são alocadas às tarefas do que o necessário. Já a fragmentação externa existe quando há um numero suficiente de processadores disponíveis para satisfazer uma requisição, mas eles não estão contíguos ou não podem ser encontrados pelo algoritmo de alocação. Resultados experimentais mostram que pouco aperfeiçoamento em desempenho pode ser obtido através de refinamentos em algoritmos contíguos (KRUEGER; LAI; RADIYA, 1994).

Tecnologias de comunicação como roteamento *wormhole* (NI; MCKINLEY, 1993) permitem considerar o uso de alocação não contígua, já que o número de roteadores

intermediários no caminho da mensagem não é o fator dominante na determinação da latência de mensagem. Entretanto, deve-se observar que com alocação não contígua as mensagens acabam ocupando mais canais. Por este motivo, a probabilidade da ocorrência de contenção é maior, enquanto que na alocação contígua os trabalhos causam contenção apenas em suas próprias tarefas.

O algoritmo Buddy-2D (LI, K.; CHENG, 1990), baseado na técnica de gerenciamento de memória Buddy, é a estratégia mais conhecida de alocação contígua. Ela é aplicável somente em sub-malhas quadradas, em que todos os trabalhos requisitam sub-malhas também quadradas. Além disso, os lados da malha e sub-malhas devem ser potências de dois. O *overhead* de alocação é relativamente baixo: $O(\log n)$. Entretanto, a restrição na dimensão das sub-malhas faz com o sistema sofra de severa fragmentação externa e interna.

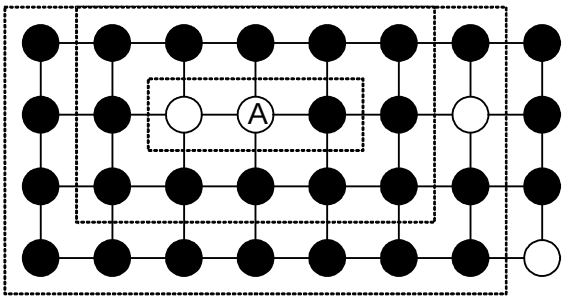

Figura 3.5: MC: Conchas em volta de A para uma requisição 3 x 1.

O algoritmo MC – *Minimizing message-passing Contention* (MACHE; LO; WINDISCH, 1997) é um algoritmo de alocação não contígua. Nele, cada processador livre avalia a qualidade de uma alocação do tamanho especificado, formando conchas de sub-malhas ao seu redor: a primeira com o tamanho requisitado; a segunda com cada lado incrementado em um; e assim sucessivamente, como na Figura 3.5, até incluir uma quantidade de processadores livres igual à requisitada. O peso do processador é dado pela concha que o contém: 0 para a sub-malha inicial; 1 para a primeira sub-malha; 2 para a segunda e assim por diante. A soma dos pesos dos processadores resulta no custo da alocação. A alocação que apresenta o menor custo é a escolhida.

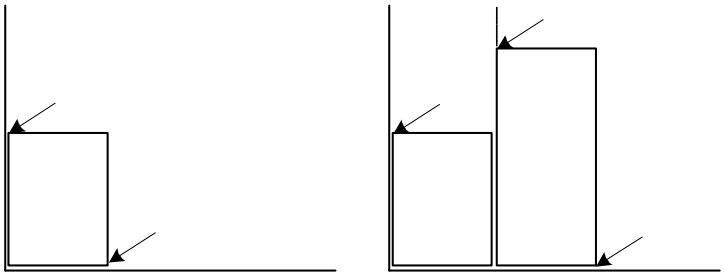

Figura 3.6: Alocação *Tight-packing*.

Um exemplo de alocação bidimensional em redes malha, sem usar *bin-packing* diretamente, é o *Tight-Packing* (HWANG, I., 1997). Ele mantém uma lista de pontos, cada um com o respectivo número de processadores livres nas direções horizontal e vertical. Uma sub-malha pode ser alocada, com um dos cantos conectado a um destes pontos, se houver espaço suficiente para ela e se o canto estiver livre. Um canto é dito livre se não houver uma sub-malha alocada acima e abaixo dele. Somente um canto sudoeste ou noroeste pode ser um canto livre. Quando uma sub-malha é alocada, o canto

livre utilizado é retirado da lista e outros dois relativos à nova sub-malha são acrescentados (Figura 3.6). O algoritmo seleciona, entre os cantos livres, aquele que suportar uma sub-malha do tamanho requerido e que ao mesmo tempo mantenha a proporção entre a seção alocada e o tamanho total da malha.

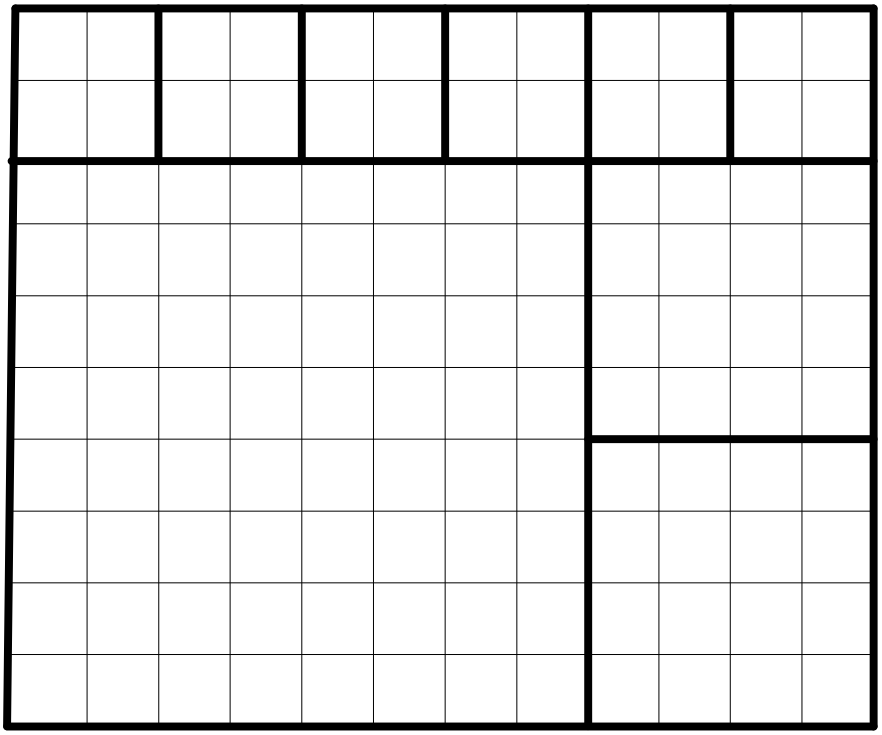

Figura 3.7: MBS.

Outro esquema de alocação não contígua é o MBS – *Multiple Buddy Strategy* (BURNS, 1997). O MBS é baseada na estratégia Buddy-2D, entretanto é modificado de forma a permitir alocação não contígua. Isso é feito quebrando blocos maiores em blocos menores de até no mínimo um processador (Figura 3.7), lembrando que somente blocos quadrados de lados potência de dois são permitidos no Buddy. Apesar de um modelo não contíguo ter sido adotado, a contigüidade dentro dos blocos individuais ainda é mantida.

Uma estratégia de alocação simples, que pode ser utilizada tanto para alocação contígua como para não contígua, é a paginação. A paginação consiste em dividir a malha em pequenos blocos quadrados, 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.

Um exemplo de paginação não contígua é visto em Lo et al. (1997). Em seu trabalho, foram consideradas várias ordenações de páginas, incluindo em linha e curva S. Na Figura 3.8 é apresentado o modelo de indexação em linha utilizado, onde cada página é composta por 4 processadores, que recebe o endereço do processador base. O processador base é o inferior mais à esquerda da página. Uma lista de endereços de páginas livres é mantida, de acordo com a indexação. As páginas livres são as que estão em branco. Lembrando que uma ordem em linha de todas as páginas irá de 0 a 15 em seqüência.

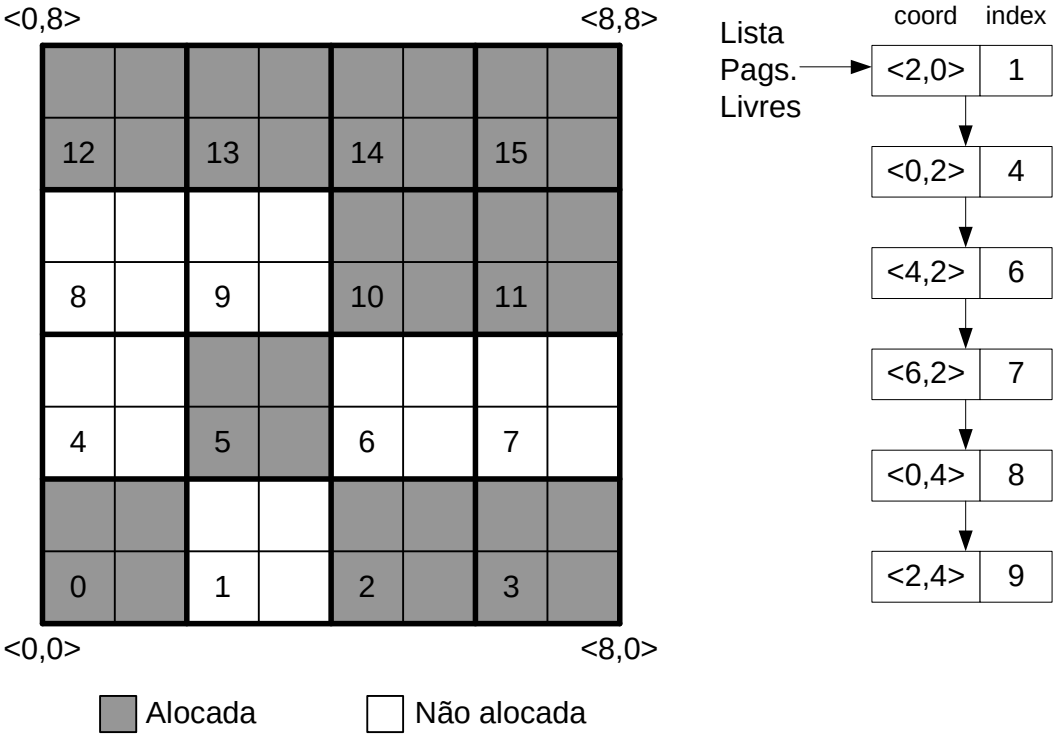

Figura 3.8: Alocação de sub-malhas por paginação.

Leung, J. Y. T.; Whitehead (2002) independentemente desenvolveram um algoritmo similar à paginação, propondo usar curvas fractais para indexar os processadores. 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 com FF ou BF. 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 ambos escolhas afetam a performance do sistema.

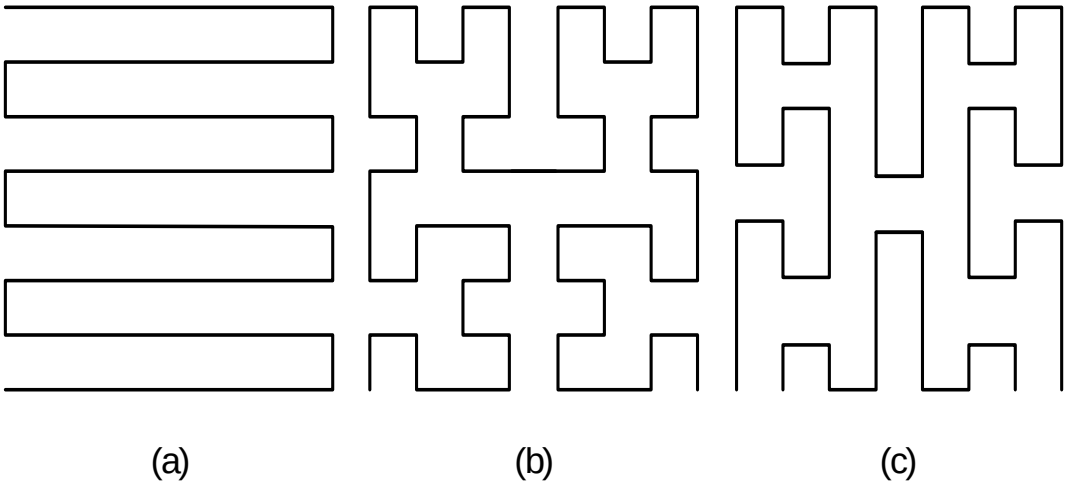

Figura 3.9: Curvas: (a) S (b) Hilbert (c) H-Index.

Uma curva fractal mapeia um intervalo unidimensional em uma área bidimensional, preenchendo os planos sem deixar buracos e ao mesmo tempo passando por todos os pontos (WAGON, 1991). Estas curvas são definidas recursivamente e reconhecidamente preservam várias medidas de localidade. Na Figura 3.9, são mostradas algumas curvas clássicas: curva S, Hilbert e H-Index. Bunde; Leung; Mache (2004) utilizaram as mesmas curvas fractais para ordenar os processadores de uma malha, ao invés de páginas.

Existem dois algoritmos de alocação de sub-malhas que consideram alocação para sistemas de tempo real, são eles: Buddy-RT (BABBAR; KRUEGER, 1994), baseado no Buddy-2D; e DEDF – *Deferred* EDF (MOHAPATRA, 1997). A abordagem adotada em ambos difere do modelo de escalonamento de tarefas periódicas. Na verdade, é um modelo de trabalhos aperiódicos, ou seja, um trabalho chega para executar, precisando ser alocado e escalonado antes do seu *deadline*. Depois de executar, o trabalho é desalojado e deixa o sistema.

O Buddy-RT associa uma etiqueta a cada processador, que define seu instante de disponibilidade mais próximo (EAT – *Earliest Available Time*). O algoritmo é composto por duas fases, sendo a segunda opcional. Na primeira fase, o alocador tenta alocar todos os trabalhos na ordem em que chegam, procurando por sub-malhas do tamanho adequado, cujo EAT seja menor que o instante de disponibilidade mais tardio (LST – *Latest Start Time)* do trabalho. Caso tal sub-malha não exista, o alocador falha nessa fase. A decisão de ir ou não para a próxima fase depende da quantidade de tempo disponível até o LST de todos os trabalhos na malha. Se o tempo disponível é maior que o WCET da segunda fase, então ela é iniciada, de outra forma, o trabalho é rejeitado. Na segunda fase, o alocador tenta relocar todos os trabalhos, inclusive os que estão executando.

O DEDF difere um pouco do Buddy-RT, na medida que tenta atrasar ao máximo o escalonamento de um trabalho, para que, mais tarde, possa alocar um conjunto deles em vez de apenas um. Isso tem por finalidade reduzir a fragmentação e as preempções de trabalhos se comparado com o escalonamento de um único trabalho por vez. Enquanto o Buddy-RT utiliza uma etiqueta de tempo com o EAT, o DEDF considera o intervalo de tempo que processador estará disponível, chamado de janela de tempo.

3.3.3 *List scheduling*

List scheduling (GRAHAM, 1966) é uma técnica que gera um escalonamento de tarefas, a partir de um grafo, respeitando suas dependências. As tarefas ainda não escalonadas, que já tiverem suas dependências resolvidas, são chamadas de tarefas prontas e são mantidas em uma lista ordenada por prioridade. A cada passo do algoritmo, a lista é atualizada e a tarefa com maior prioridade é removida e escalonada.

Os algoritmos de *list scheduling* levam exatamente n passos para escalonar um grafo de tarefas, onde n representa o número de nodos, e geralmente diferem apenas na forma como atribuem prioridades e como o processador mais adequado é escolhido. Exemplos de critérios para a atribuição de prioridade são: nível, tempo de processamento, caminho crítico e critérios temporais.

O caminho crítico é um conjunto de nodos e/ou arestas que formam um caminho entre dois nodos, no qual a soma dos pesos é a máxima possível no grafo em questão. Quando estes nodos são um par raiz/folha do grafo, o caminho crítico potencialmente determina o comprimento do escalonamento, ou pelo menos o comprimento mínimo.

O nível de um nodo no grafo é definido como a soma dos pesos do caminho crítico, entre ele e uma das folhas do grafo. Essa métrica é utilizada no algoritmo HLFET – *Highest Levels First with Estimated Times* (ADAM; CHANDY; DICKSON, 1974). Todavia, no algoritmo original, os pesos das arestas são desconsiderados no cálculo do caminho crítico.

Os critérios de atribuição de prioridades, baseadas em atributos dos nodos ou grafos, são critérios estáticos, que podem ser calculados apenas uma vez, antes do escalonamento. Critérios temporais como o mais tarde possível (LPST – *Latest Possible Start Time*) e o mais cedo possível (EST – *Earliest Start Time*) dependem do escalonamento parcial e por isso são ditos dinâmicos. Ambos são calculados avaliando-se o instante de início de um nodo, exaustivamente em todos os processadores, para se encontrar o mínimo ou o máximo. Enquanto o EST é calculado em função do escalonamento normal, no LPST o grafo de tarefas é escalonado ao contrário, a partir das folhas em direção à(s) raiz(es). Dessa forma, cada nodo tem seu início atrasado ao máximo.

Um algoritmo baseado no LPST é o MCP – *Modified Critical Path* (WU; GAJSKI, 1990). Ele calcula o LPST dos nodos a priori, para dinamicamente usá-lo como prioridade. Quanto menor o LPST, maior será a prioridade do nodo. O algoritmo é chamado de Caminho Crítico Modificado porque o LPST é restringido pelo caminho crítico. Dessa forma, se uma tarefa tem o menor LPST, ela é a tarefa crítica nesse passo do escalonamento. O processador selecionado para escalonar a tarefa com maior prioridade é aquele que consegue fazê-lo antes. A complexidade do MCP é $O(v^2 1\log v)$, onde v é o número de nodos do grafo.

Por sua vez, o ETF – *Earliest Task First* (HWANG, J.-J., 1989) é baseado no EST. Dessa forma, ele seleciona um nodo em função desse critério de prioridade dinâmica. Porém, havendo empate, a prioridade estática é usada para o desempate. Ela pode ser calculada tanto por níveis quanto por LPST. A complexidade do ETF é $O(pv^2)$, onde p é o numero de processadores.

O DLS – *Dynamic Level Scheduling* (SIH; LEE, 1993) usa como prioridade o nível dinâmico DL – *Dynamic Level* que é calculado da seguinte forma:

$$DL(n_i, P) = SL(n_i) - ST(n_i, P) \tag{3.8}$$

onde n_i é um nodo e P um processador. O primeiro componente é o nível estático $SL(n_i)$ e o segundo é o momento em que a tarefa n_i será escalonada se for alocada em P. A cada passo do escalonamento, o DLS computa o DL para cada nodo pronto em cada processador. Então, o par processador-nodo que apresenta o maior DL é selecionado para alocação e execução. A complexidade do DLS é $O(v^3 pf(p))$ onde p é o número de processadores e $f(p)$ é a complexidade do algoritmo de cálculo do ST.

3.3.4 *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 K e outro de arestas A. Cada nodo $k_i \in K$ é uma tarefa e cada aresta $a_{i,j} \in A$ é uma dependência entre k_i e k_j. Ambos, k_i e $a_{i,j}$, costumam ser anotados com seus respectivos pesos.

Segundo Backer; 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”.

Não há uma definição unânime, mas algoritmos de *clustering* são geralmente classificados em *clustering* hierárquico e partitivo. Enquanto o hierárquico agrupa os *clusters* numa seqüência de particionamentos sucessivos, o partitivo divide o *cluster* diretamente no número de *clusters* requerido (XU; II, 2005).

O *clustering* hierárquico pode ser classificado em divisivo ou aglomerativo. O aglomerativo gera os *clusters* de cada nível por um processo de junção entre os *clusters* existentes no nível anterior, começando a partir do conjunto inicial onde cada *cluster* possui apenas um objeto ou nodo. O divisivo procede de forma oposta. No inicio há apenas um *cluster* que é dividido sucessivamente até o número de *clusters* requeridos ter sido gerado.

O *clustering* hierárquico acaba sendo sensível a “ruídos”, uma vez que, após um objeto ser alocado em um *cluster*, não será considerado novamente. A complexidade computacional do *clustering* hierárquico é de pelo menos $O(n^2)$.

Contrastando com *clustering* hierárquico, que faz sucessivos níveis de *clusters* por fusões ou divisões consecutivas, o *clustering* partitivo cria já no primeiro passo o número de k partições esperado. Esse particionamento poderia ser criado com um método de força bruta, que na prática é inviável. Ao invés disso, heurísticas iterativas são utilizadas. Dessa forma, as k partições são inicializadas aleatoriamente ou baseadas em algum critério como balanceamento de carga, ou minimização da comunicação entre partições. A partir daí, elementos são selecionados e migrados de partição, até que não haja mais possibilidade de mudança ou a qualidade do particionamento esteja satisfatória.

3.3.4.1 EZ

O algoritmo EZ – *Edge Zeroing* (SARKAR, 1989) é um método em dois passos para escalonar tarefas com comunicação em sistemas multiprocessadores. A primeira fase é a aplicação de um *clustering* partitivo aditivo, onde arestas em um mesmo *cluster* são consideradas zeradas. A segundo fase consiste na alocação dos *clusters* para os processadores. A complexidade do EZ é de $O(e\,(e + v))$, onde “e” é o número de arestas e “v” o número de nodos do grafo.

A idéia básica é unir *clusters*, zerando as arestas entre *clusters*. Uma união é permitida se o comprimento do escalonamento não aumentar. 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.

3.3.4.2 LC

Em função das dependências entres tarefas, Gerasoulis; Yang (1993) classificam um *cluster* em linear e não linear. Um *cluster* é chamado não linear se possui pelo menos duas tarefas independentes (Figura 3.10 - a), caso contrário é chamado de linear (Figura 3.10 - b) e todas as tarefas fazem parte da mesma cadeia de dependências.

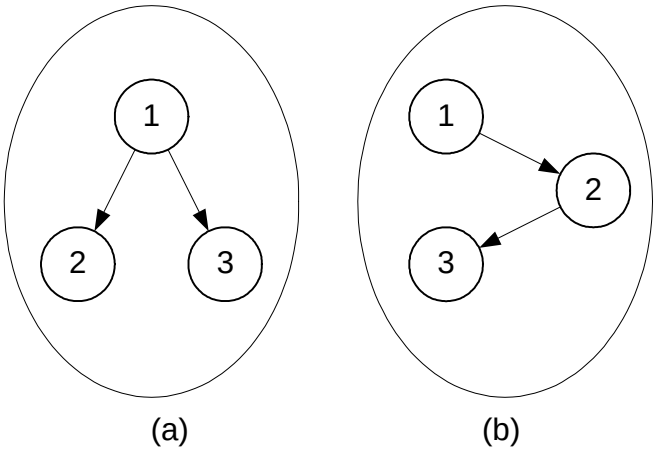

Figura 3.10: *Clusters:* (a) não linear (b) linear.

Essa classificação pode ser aplicada em algoritmos de *clustering*, de forma que somente um dos tipos de *cluster* possa ser gerado. O efeito de ambas as abordagens aparece ao se escalonar as tarefas. 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.

Numa arquitetura paralela, reduzir o paralelismo não faz sentido e comunicação é um preço a se pagar por paralelismo. Por isso, Gerasoulis; Yang (1993) propuseram 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. Este é o algoritmo LC – *Linear Clustering*.

Como conseqüência dessa modificação, o LC remove o paralelismo inútil do escalonamento, aquele que não diminui seu comprimento, podendo assim resultar em um número *clusters* diferente do esperado.

3.3.4.3 DSC

O DSC – *Dominant Sequence Clustering* (YANG; GERASOULIS, 1994) é outro algoritmo baseado no EZ. Sua idéia principal é diminuir o comprimento do escalonamento a cada passo, zerando arestas da seqüência dominante do grafo. A seqüência dominante nada mais é que o caminho crítico do grafo escalonado. Os nodos que fazem parte do DS são chamados de Nodos DS, enquanto os demais são chamados de Nodos SubDS.

O Nodo DS é aquele que possui a maior prioridade em cada passo do escalonamento, que é dada por: $PRIO(n)=tlevel(n)+blevel(n)$, onde $tlevel(n)$ é a soma dos pesos dos nodos e arestas que compõem o caminho crítico entre uma raiz do grafo e n, sem incluir o peso de n. De forma simétrica, o $blevel(n)$ é a soma dos pesos do caminho crítico a partir de n até uma folha.

Durante o escalonamento, duas listas de nodos ordenadas por prioridades são mantidas. Uma com os nodos que já foram examinados, e outra com os nodos restantes ainda não examinados.

Inicialmente todos os nodos são marcados como não examinados e, como em um algoritmo de *list scheduling*, somente um nodo que já teve seus predecessores visitados

nos passos anteriores, pode ser visitado no passo atual. Isso faz com que os nodos sejam visitados em ordem topológica.

Enquanto houver nodos não examinados, o algoritmo seleciona aquele que tiver a maior prioridade e tenta zerar uma aresta. A aresta selecionada é aquela que tem o maior peso e liga o nodo a um de seus predecessores. Ela somente pode ser zerada se o tlevel do nodo não aumentar. Caso contrário, o nodo se mantém como um *cluster*.

Após processar um nodo, as prioridades devem ser atualizadas. O blevel é calculado antes do escalonamento e não precisa ser re-calculado. O tlevel, ao contrário, vai se alterando de acordo com os *clusters* que vão sendo formados e por isso deve ser calculado incrementalmente. Entretanto, somente os nodos filhos do nodo escalonado precisam ser atualizados, uma vez que os nodos são visitados em ordem topológica.

Outra conseqüência dessa ordenação é que não apenas nodos que fazem parte da seqüência dominante são visitados. Na verdade existe uma intercalação entre eles e os nodos SubDS.

A complexidade do DSC é $O((e + v) \log v)$.

3.3.4.4 MD

No algoritmo MD – *Mobility Directed* (WU; GAJSKI, 1990) a prioridade de um nodo é baseada na sua mobilidade relativa, definida pela razão entre sua mobilidade e seu peso, onde a mobilidade é a diferença entre o EST e o LST, calculados como nos algoritmos de *list scheduling*.

Basicamente, um nodo com mobilidade zero é um nodo do caminho crítico. A cada passo, o MD escalona o nodo com a menor mobilidade para o primeiro processador que possuir uma janela de tempo larga o suficiente para acomodá-lo. Nenhum esforço adicional é feito no sentido de minimizar o EST do nodo.

Quando um nodo é escalonado, todas as mobilidades relativas devem ser atualizadas. A complexidade do MD é $O(v^3)$.

3.4 VS para MPSoCs com restrições temporais

Da mesma forma que em sistemas unitários, VS e PM também podem ser aplicados em MPSoCs, mesmo se ainda for considerada a existência de restrições temporais. Desse modo, o processo de escalonamento de tarefas pode apresentar até três fases: alocação, escalonamento e VS. Cada uma delas pode ser aplicada *on-line* ou *off-line*. Todavia, como elas apresentam dependência entre si, a aplicação de VS estaticamente exige que as fases anteriores também o sejam, da mesma forma que escalonamento estático exige alocação estática anterior.

Em Okuma; Yasuura; Ishihara (2001) é apresentada uma das primeiras abordagens de VS em multiprocessadores para tempo real, nesse caso tratando-se de uma abordagem totalmente estática, onde o problema é definido e otimizado por programação linear inteira. Obviamente, a abordagem estática baseia-se na premissa que a carga de trabalho dinâmica do sistema é conhecida a priori. São apresentadas também duas simplificações, uma para o caso onde apenas os tempos de chegadas de tarefas são conhecidos e outra para o caso totalmente dinâmico com uso de predição.

Como discutido nas seções anteriores, a existência de escalonamento e alocação em fases distintas está relacionada à abordagem de escalonamento distribuído adotada. Por sua vez, essa escolha é fortemente determinada pela estrutura de interconexão utilizada.

No caso de Okuma; Yasuura; Ishihara (2001), o sistema possui interconexão barramento e memória compartilhada, implicando na não existência de uma fase de alocação, uma vez que escalonamento global é o mais indicado nesse caso. O custo de comunicação entre tarefas também é desprezado, visto que permanece constante para qualquer distribuição de tarefas sobre os processadores.

Os algoritmos dinâmicos de VS em multiprocessadores atuam da mesma forma que os para processadores unitários, ou seja, são compostos por duas fases: detecção e distribuição de folgas. Esses algoritmos focam na distribuição de folgas e por isso costumam utilizar alocação e até mesmo escalonamento, feitos estaticamente.

O algoritmo *Shared* (ZHU; MELHEM; CHILDERS, 2001) foi proposto para tentar contornar a limitação da distribuição de folga *greedy* em cumprir os *deadlines* em sistemas multiprocessadores com escalonamento global, pois a perda de *deadlines* acaba ocorrendo mesmo que a execução canônica termine antes do *deadline*. A primeira suposição do algoritmo é que as tarefas com maior tempo de execução deixem as maiores folgas. Por este motivo, a lista de tarefas prontas para executar do escalonador global é ordenada pelo critério de maior tarefa primeiro, ou seja, em ordem decrescente de tempo de execução. Essa suposição é valida para tarefas independentes. O *shared* distribui a folga atual entre a próxima tarefa no processador atual e a tarefa ativa no processador que tem a tarefa mais próxima do término. Para modelar o efeito de tarefas dependentes é usado um algoritmo de *list scheduling*, nesse caso o *shared* também perde *deadlines* devido à interferência do *list scheduling* na ordem de execução das tarefas. A solução adotada é dada pelo algoritmo LSSR – *Fixed-Order List scheduling with Shared Reclamation*, que consiste em ordenar a lista de tarefas prontas do escalonador na mesma ordem do escalonamento canônico.

Zhang; Hu; Chen (2002) apresentam um algoritmo de escalonamento de tensão para tarefas dependentes em sistemas distribuídos heterogêneos, com estimativa de folga estática baseada na análise do caminho crítico e distribuição SPM. A folga dinâmica, por sua vez, é distribuída por *greedy*. O particionamento é gerado por um *framework* e o algoritmo de tensão atua separadamente em cada processador desconsiderando a comunicação inter-processador do mesmo modo que Okuma *et al.* (OKUMA; YASUURA; ISHIHARA, 2001).

Mishra *et al.* (2003) propõem uma técnica dinâmica de distribuição de folgas, combinada às técnicas convencionais *greedy* e SPM, chamada de *gap filing*. A arquitetura do sistema é distribuída e por isso o custo de comunicação é considerado no modelo. Para alocação e escalonamento são utilizados os algoritmos LC (SELVAKUMAR; MURTHY, 1994) e *list scheduling* respectivamente. Verificou-se que tanto o *greedy* quanto o SPM não são ótimos em sistemas com múltiplos processadores devido à variação do paralelismo da aplicação. Dessa forma, maiores ganhos são obtidos dando-se mais folgas às seções do escalonamento que possuem maior paralelismo. A folga dinâmica também é distribuída com *greedy*, que pode levar o processador a um estado de ociosidade se a última tarefa do escalonamento executar menos que WCET. Esse tempo ocioso nada mais é do que uma folga que pode ser aproveitada. O *gap filing* sugere adiantar nessa folga a execução de uma tarefa que está aguardando para ser liberada, permitindo assim execução fora de ordem.

O algoritmo estático PDP – *Proportional Distribution and Parallelism* (HUA; QU, 2005) foi proposto com o objetivo de eliminar a folga local. Enquanto a folga global é a diferença entre o caminho crítico do escalonamento canônico e o *deadline*, a folga local é aquela que surge nos processadores devido às dependências das tarefas. O PDP tem o intuito de eliminar esta folga e ainda obter maiores ganhos com exploração do paralelismo em uma alocação feita com o algoritmo DLS (SIH; LEE, 1993). O PDP consiste na aplicação iterativa de distribuição de folgas SPM sobre as folgas local e global, seguida de relocação de tarefas para exploração do paralelismo.

O LPDEDF – *Low-Power Distributed EDF* proposto por Moncusi; Arenas; Labarta (2003) aplica VS em sistemas de tempo real rigorosos e distribuídos a partir de uma alocação arbitrária. A abordagem mais comum nesse contexto é distribuir os *deadlines* das folhas do grafo entre as tarefas nos processadores de acordo com o WCRT, inclusive das comunicações. Dessa forma, cada processador pode ser considerado como independente. Entretanto, os WCRTs das tarefas podem ser extremamente pessimistas, por isso o LPDEDF não usa essa abordagem. Como o sistema possui *deadlines* rigorosos, uma abordagem conservadora continua sendo adotada, na qual uma freqüência mínima é estabelecida estaticamente, de forma a garantir o cumprimento dos *deadlines* mesmo que menos energia seja economizada. O algoritmo de distribuição de folgas utilizado é o LPPS, ou seja, toda folga é atribuída à tarefa ativa, somente se a fila de tarefas prontas para executar estiver vazia. Assim, para atribuir a folga, é necessário saber o momento em que a próxima tarefa será liberada, que no caso de uma tarefa independente é o instante de sua liberação para o próximo período. Para o caso de tarefas com dependência que estão bloqueadas esperando comunicação, isso se torna mais complexo. O algoritmo propõe atribuir a cada tarefa uma estimativa de liberação. A cada tarefa liberada, as estimativas de liberação das tarefas que fazem parte da sua cadeia de dependências são atualizadas. Na prática o LPDEF aplica uma distribuição dinâmica do *deadline* das nodos folhas à medida que aplica VS.

3.5 Redução do consumo de energia através de alocação

Quando o SoC é formado por elementos heterogêneos, a forma como as tarefas são alocadas aos processadores tem grande impacto no consumo de energia geral do sistema, pois alocar a mesma tarefa em diferentes elementos traz diferentes custos de energia e desempenho. Mesmo em SoCs homogêneos, alocações geram diferentes caminhos de roteamento e também diferentes volumes de informação e assim duas alocações viáveis de tarefas podem resultar em consumos de energia completamente diferentes.

Um experimento de exploração de heterogeneidade de processadores para economia de energia foi feito por Kumar *et al.* (KUMAR et al., 2003), utilizando um SoC de processadores Alpha com memória externa compartilhada. Durante a execução da aplicação, o escalonador escolhe um processador para executar a tarefa ativa de acordo com a função objetivo, geralmente mantendo certo desempenho com máxima eficiência energética. Apenas o processador selecionado fica ativo durante um período de teste. Após esse período, a tarefa pode ser migrada, se o processador atual não for adequado. A ordem de visita dos processadores é definida pelas políticas: vizinhança local, vizinhança global, randômica, ou em ordem pré-definida.

Os trabalhos de VS demonstrados na seção anterior se preocupam exclusivamente com a detecção e distribuição de folgas de tarefas em sistemas multiprocessadores com

alocação dada ou escalonamento global simples. Na medida que a quantidade de folgas que podem ser distribuídas varia em função da alocação, Zhang; Hu; Chen (2002) propõem que tanto a alocação quanto o escalonamento devem ser feitos, focados em proporcionar a ocorrência máxima de folgas no escalonamento, maximizando assim o ganho de energia. Eles apresentam um esquema de alocação no qual as tarefas são ordenadas por ordem crescente de prioridade e depois alocadas aos processadores. A prioridade de uma tarefa é dada por $P_i = lft_i + U_i$, onde lft_i é o LST da tarefa e U_i a sua utilização. A alocação obedece as seguintes regras:

- Seleciona o processador que ficou disponível ao mesmo tempo que a tarefa;
- Seleciona o processador que ficou disponível por último antes da tarefa;
- Seleciona o processador que ficou livre primeiro.

Em Aydin; Yang (2003) são usadas heurísticas de *bin-packing on-line* e *off-line* para o particionamento de tarefas com minimização do consumo de energia, considerando tarefas independentes sem custo de comunicação. O trabalho propõe que a velocidade ótima, para o escalonamento de um grupo de tarefas em um processador com política EDF, em termos de energia, é constante e mantém a utilização total do processador próximo a 1. Desse modo, o particionamento ótimo distribui as tarefas uniformemente entre todos os processadores (balanceamento de carga).

Para alocação *off-line*, o WFD – *Worst-Fit Decreasing*, que ordena a lista de tarefas 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 viabilidade e energia. FF, NF e BF têm bons números de escalonamentos viáveis gerados e performances ruins em termos de consumo energia. O WF apresenta performance razoável em baixas utilizações, degradando rapidamente com crescimento destas. Como solução 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 os bons números de viabilidade do BF/FF aliado ao bom resultado de energia do WF.

Alenawy; Aydin (2005) usam os mesmos algoritmos de *bin-packing* para particionamento e escalonamento estático de tarefas com RM, já que muitos sistemas de tempo real ainda são escalonados estaticamente. Também são estudados os efeitos de vários testes de controle de admissão no escalonamento de tensão.

Hu; Marculescu (2004) foram os primeiros a propor um modelo de escalonamento de tensão para qual a comunicação é através de uma NoC e os processadores são heterogêneos. O particionamento é feito estaticamente, com atribuição de prioridades 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. Isso pode ser evitado com a aplicação do algoritmo de reparação do 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 processadores que têm maior impacto no consumo de energia e performance.

4 PROPOSTA

4.1 Considerações Iniciais

Técnicas comumente utilizadas para redução de consumo de energia em nível de sistema, como PM e VS, são geralmente aplicadas durante o escalonamento após a alocação estar efetuada. Todavia, a forma como as tarefas são alocadas nos processadores em um MPSoC tem grande impacto no consumo do sistema, sendo ele heterogêneo ou não, pois o processo de alocação acaba por limitar o espaço de trabalho das técnicas aplicadas posteriormente durante o escalonamento. Dessa forma, já durante a alocação existe a preocupação com redução do consumo (ZHANG; HU; CHEN, 2002).

Na literatura existem três idéias principais, no que diz respeito à redução de consumo durante a alocação de tarefas em MPSoCs: minimização da comunicação, balanceamento de carga para aplicação de VS e concentração de carga para a aplicação de PM. Elas podem ser combinadas em técnicas mais completas, apesar de conterem alguns pontos mutuamente exclusivos, gerando funções multi-objetivo.

PM e VS se baseiam na premissa de que economia vem do uso meticuloso dos recursos do sistema, evitando-se assim desperdício. Porém, principalmente em sistemas de tempo real, um desempenho mínimo deve ser mantido, de forma a não comprometer o comportamento temporal do sistema.

Numa situação ideal de VS, dois processadores com a metade da freqüência devem fazer o trabalho de um. Conseqüentemente, uma NoC de processadores com menor capacidade poderia ser utilizada para substituir um processador com capacidade equivalente e freqüência muito mais alta. Contudo, mesmo explorando o máximo de paralelismo, isso não se confirma na prática. Acaba ocorrendo desperdício de energia, porque processadores ficam ociosos em função das dependências das tarefas e mais tarde precisam se recuperar, executando com freqüência mais alta.

A minimização da comunicação visa economizar a energia dinâmica gasta pelos elementos da rede de intercomunicação; entretanto, não deve ser esquecido que a comunicação entre processadores é um preço a se pagar por paralelismo. O paralelismo, por sua vez, é essencial para que a eficiência do sistema justifique o custo de se distribuir carga entre processadores.

Em se tratando de sistemas com barramento centralizado, nos quais a comunicação entre processadores não é um problema e geralmente nem é considerada, existem vários trabalhos de alocação e/ou escalonamento de tarefas, preocupados com a minimização do consumo de energia. Entretanto o barramento centralizado não é uma solução

escalável e apresenta seus problemas de eficiência energética. Em seu lugar adota-se estruturas de comunicação, como NoCs do tipo malha, que superam essas deficiências. Entretanto, o uso de sistemas descentralizados traz como conseqüência o aumento da complexidade do processo de alocação, uma vez que o custo de comunicação e as contenções passam a ser relevantes.

Como apresentado no capítulo anterior, o primeiro e único trabalho de alocação com minimização do consumo de energia com restrições temporais em NoCs é Hu; Marculescu (2004). Contudo, se trata de uma abordagem estática. Tal falta de soluções nesse contexto, principalmente em ambientes dinâmicos, é facilmente explicada pela complexidade do problema. Sob essas circunstâncias, a alocação é um problema NP-Completo, para o qual existem reduções polinomiais, mas que ainda são lentas demais para serem aplicadas *on-line*.

4.2 Objetivo

O objetivo deste trabalho é propor técnicas de alocação dinâmica de tarefas periódicas em MPSoCs heterogêneos, com processadores interligados por uma NoC malha, visando redução do consumo de energia do sistema, sendo este um primeiro estudo para o desenvolvimento de um particionador dinâmico com estas características, que envolveria a definição de duas políticas: a heurística de alocação, que visa definir qual o melhor destino de uma tarefa; e o algoritmo de escalonamento distribuído, responsável por implementar essa heurística através da colaboração dos nodos do sistema. Esse trabalho se concentra na primeira, ou seja, na busca de uma heurística de alocação baseada em *bin-packing*.

Bin-packing é um ponto de partida interessante, pois visa máxima utilização dos recursos do sistema, tentando evitar desperdício. Esse critério combina perfeitamente com a idéia de utilização judiciosa dos recursos, das técnicas de redução do consumo do sistema em nível de sistema, adotadas em sistemas embarcados, mesmo que, máxima utilização não seja o mais importante em sistemas de tempo real..

Outra característica importante de *bin-packing* é o fato de suas heurísticas serem de baixa complexidade, podendo ser facilmente aplicadas para alocação *on-line*.

A arquitetura alvo, para a qual se está desenvolvendo o alocador, é uma NoC malha, escolhida devido a sua simplicidade e escalabilidade. O tamanho da rede é fixado em 4x4, o qual julga-se razoável para MPSoCs atuais.

Uma vez que um modelo de particionamento é adotado, cada nodo do sistema é dado como autônomo, tendo seu próprio escalonador. Cada escalonador local utiliza política EDF e aplica VS para minimização do consumo de energia, podendo também desligar totalmente processadores sem carga alocada (PM). O EDF foi escolhido em virtude do fato de que é a política de escalonamento dinâmica geralmente utilizada com algoritmos de VS, como na maioria dos algoritmos dinâmicos listados na Seção 2.4.2. Neste trabalho busca-se redução do consumo de energia nos processadores e NoC, não sendo considerado outros componentes do sistemas, como elementos de E/S.

Para a comparação das heurísticas é necessário definir e implementar um modelo que permita medir o consumo de energia e a eficácia das alocações, principalmente no que diz respeito ao atendimento de *deadlines*.

Heurísticas de *bin-packing* foram avaliadas anteriormente para alocação de tarefas em sistemas multiprocessadores com barramento centralizado. Porém, os resultados obtidos não foram validados em redes do tipo malha.

4.3 Justificativa

Em sistemas com cargas de trabalho previsíveis, ou conhecidas a priori, as técnicas de gerenciamento de energia podem estaticamente explorar completamente os ganhos entre potência e desempenho. Entretanto, na maioria das situações reais não há tal conhecimento a priori, de forma que a carga de trabalho é imprevisível em muitos sistemas, desde simples telefones celulares e PDAs, até SoCs mais complexos. Nessa situação, é desejável que o sistema possa adaptar-se dinamicamente à carga de trabalho, economizando recursos que provavelmente foram disponibilizados em função do pior caso.

A abordagem mais comum em sistemas dinâmicos é alocar as tarefas nos próprios processadores onde são criadas. A partir do momento que o sistema passa a degradar em função da má distribuição de carga ou do alto custo de comunicação, as tarefas começam a ser redistribuídas.

Uma alternativa é distribuir as tarefas tão logo entrem no sistema, para reduzir a sua degradação e respectivo custo para correção. Nesse caso técnicas eficientes, e principalmente muito rápidas, se fazem necessárias, uma vez que soluções ótimas, nesse contexto, são reconhecidamente um problema NP-completo.

Um sistema onde aplicações iniciam e finalizam aleatoriamente tende a fragmentar e se tornar ineficiente, mesmo que um algoritmo de alocação ótimo seja utilizado. Nesse caso, vai sempre ser necessário o uso de técnicas de correção, fazendo-se uso de migração de tarefas já iniciadas. Por isso, o mais importante é que a heurística de alocação seja rápida e não ótima.

4.4 Metodologia

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*, neste novo contexto.

A metodologia utilizada consiste em alocar aplicações com as heurísticas e compará-las em função do consumo de energia e estatística de atendimento de *deadlines*, obtidos por simulação. Para tanto, seria necessário um conjunto de aplicações multitarefas para serem utilizadas como *benchmarks*. Em segundo lugar, necessitar-se-ia de um sistema operacional de tempo real distribuído e embarcado, do qual o particionador seria um componente.

Infelizmente, não se tem até o momento a arquitetura proposta implementada, muito menos um sistema operacional com essas características, sendo que implementar todos esses elementos não faz parte do escopo deste trabalho. Além disto, o foco deste trabalho é a heurística de alocação e não a política de escalonamento distribuído.

Em face das dificuldades citadas, 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.

Além dos grafos de tarefas, outro grafo que representa a arquitetura de comunicação e seus parâmetros de configuração também é necessário. O modelo completo é apresentado no Capítulo 5.

A Figura 4.1 traz um diagrama de fluxo de dados que demonstra como fica a metodologia adotada, com base nessa abordagem.

Inicialmente, o conjunto de grafos de tarefas passa por um processo de alocação. Após todos eles terem sidos alocados, a alocação resultante é simulada. Ambos os processos requerem que sejam fornecidos o Grafo de Caracterização da Arquitetura de Comunicação e parâmetros adicionais, estes últimos específicos a cada processo.

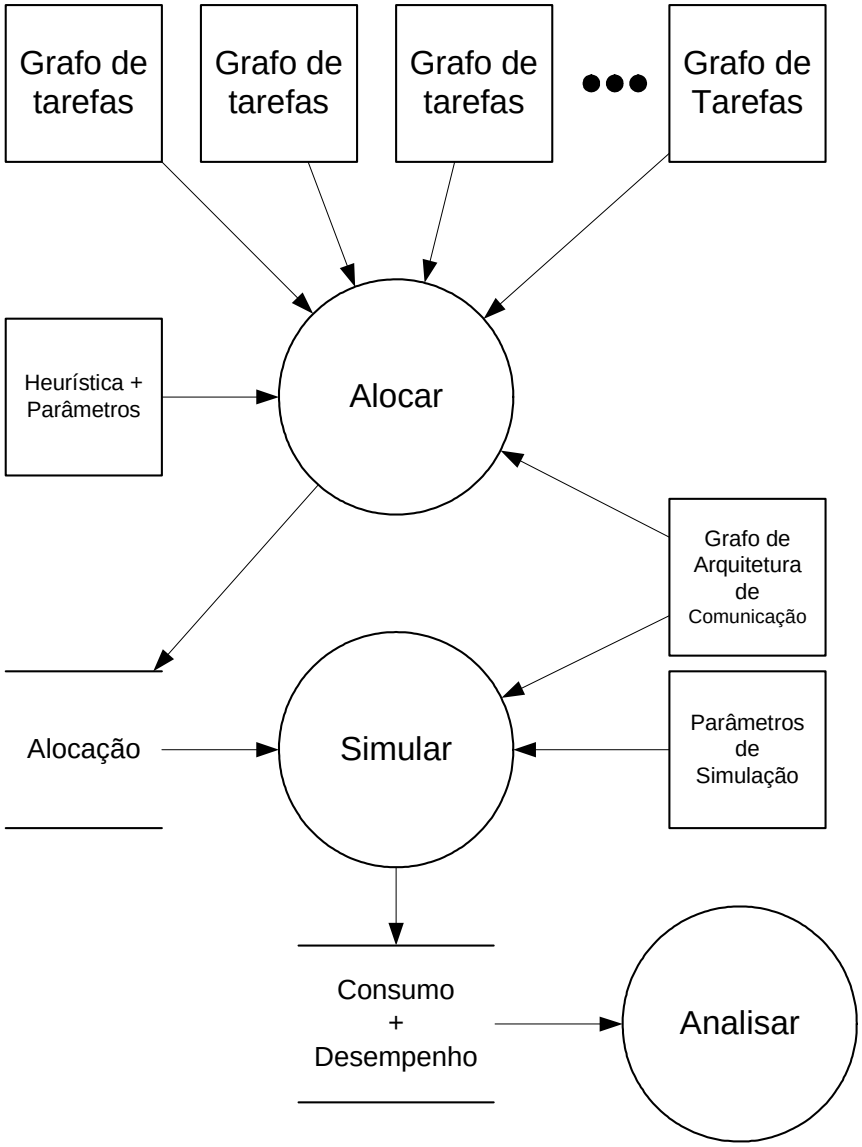

Figura 4.1: Metodologia

Verifica-se que os dois processos, alocação e simulação, são feitos separadamente. A alocação é gerada *off-line*, sendo depois simulada para a obtenção dos resultados. Dessa forma, o impacto de um possível algoritmo de particionamento dinâmico que implemente a heurística em teste não é considerado nos custos da alocação.

5 ESTIMATIVA DO CONSUMO DE ENERGIA

Algoritmos de alocação são geralmente avaliados em termos de desempenho; entretanto, no contexto de MPSoCs, o consumo de energia se torna um fator predominante.

Aydin; Yang (2003) utilizaram o quadrado da utilização do processador como heurística para comparar o consumo de energia de diferentes alocações, em processadores com VS, baseados nas premissas de que V_{norm} é proporcional à utilização do processador e V_{dd} é variável relevante na equação da potência dinâmica (Equação 2.2), devido a seu fator quadrático.

Hu; Marculescu (2004) utilizaram grafos de tarefas compostos por tarefas caracterizadas com número de ciclos e consumo de potência, gerados aleatoriamente na ferramenta TGFF – *Task Graphs for Free* (DICK; RHODES; WOLF, 1998). As arestas também estão caracterizadas com os respectivos números de bytes a serem transferidos. A potência na estrutura de interconexão é calculada através da métrica Energia de Bit (Seção 3.1.7). Com estes dados, o consumo de energia pode ser aferido a partir do escalonamento gerado.

Em termos de comunicação, as alocações também podem ser comparadas em função da quantidade de bytes transferidos entre processadores. A quantidade de bytes geralmente é o produto entre distância entre fonte e destino da transmissão e o número de bytes transferidos. No caso de uma rede malha com roteamento XY, a distância de *Manhattan* (Equação 3.1) é utilizada. Num escalonamento periódico pode ser acrescentado um terceiro multiplicador, que é a freqüência de repetição da comunicação. Aplicando-se esta heurística, a taxa de comunicação por segundo entre duas tarefas, k_i e k_j, é dada pela seguinte equação:

$$comunic = \sum a_{i,j}^{W} \cdot dist(i, j) \cdot \frac{1}{T_i} \tag{5.1}$$

onde T_i é o período de repetição da tarefa k_i, $dist(i, j)$ é a distância de *Manhattan* entre os processadores onde as tarefas estão alocadas e $a_{i,j}^{W}$ é a quantidade de bytes da transmissão.

O *framework* CAFES desenvolvido por Marcon; Palma; Calazans; Moraes; Susin; Reis (2005) avalia o consumo de energia de comunicação combinando modelos de comunicação e extensões da métrica Energia de Bit para várias topologias. Um dos modelos suportados é o CWM – *Communication Weight Model*, no qual as aplicações

são modeladas como grafos de tarefas, com os arcos anotados com as quantidades de bits trocadas entre as tarefas.

Ainda supondo que a energia dinâmica domina o consumo de energia em circuitos CMOS, vários simuladores foram desenvolvidos com o objetivo de estimar o consumo de energia em nível de arquitetura, causado pela carga e descarga da capacitância nas saídas das portas lógicas do circuito.

Na literatura, duas técnicas são freqüentemente utilizadas com este fim: cálculo do consumo pela média de portas que chaveiam no processador por cada tipo de instrução (TIWARI; MALIK; WOLFE, 1994) e cálculo através do número de portas chaveando nos componentes da arquitetura em cada ciclo (BECK, A. C. S.; MATTOS; WAGNER; CARRO, 2003).

Na primeira técnica, um simulador em nível de instruções é utilizado, enquanto que na segunda, os componentes arquiteturais do processador precisam ser simulados com precisão de ciclo.

O simulador ciclo-a-ciclo é muito mais preciso, em termos de energia e comportamento, que o em nível de instruções; contudo, é também muito mais lento. Apesar disso, vários simuladores ciclo-a-ciclo têm sido desenvolvidos para a execução de estimativas, evidenciando que a precisão justifica o custo adicional.

Outro ambiente de simulação, este para avaliação de algoritmos de VS, que mede o consumo de energia através do número de chaveamentos em cada ciclo, é o SimDVS (SHIN, D. et al., 2002). O ambiente possui alguns processadores do estado da arte previamente modelados, oferecendo também a possibilidade de que outros possam ser desenvolvidos e acrescentados pelo usuário. Porém o SimDVS não suporta a simulação de sistemas com múltiplos processadores.

No SimDVS, as aplicações são representadas por grafos de tarefas, uma vez que a codificação de *benchmarks* seria pouco eficiente, dada a quantidade de aplicações multitarefas que teriam que ser desenvolvidas e o tempo de simulação necessário. Num ambiente multiprocessador, isso tornaria impossível a exploração rápida de diferentes algoritmos de alocação.

Outro simulador que mede o consumo de energia por número de chaveamentos por ciclo, só que voltado para NoCs, é o Xpipes (BERTOZZI; BENINI, 2004). O Xpipes é uma biblioteca de componentes de NoCs customizáveis, desenvolvida em SystemC. O cálculo do consumo de energia é efetuado com a biblioteca Orion (WANG, 2002), que, como visto na Seção 3.1.7, possui modelos de capacitância de baixo nível para cada componente do roteador. Esses modelos permitem estimativas mais precisas do consumo de energia, tanto dinâmico quanto estático.

5.1 Serpens

Segundo as necessidades deste trabalho, definiram-se três requisitos que um simulador deve possuir para permitir a avaliação do consumo de energia de alocações.

Primeiramente, tanto o escalonamento das tarefas nos processadores quanto a comunicação na NoC precisam ser simulados juntos, pelo fato de que ambos causam interferência mútua. Essa exigência desqualifica o SimDVS, que além disso, simula apenas sistemas monoprocessadores, e o CAFES, que enfatiza apenas a comunicação.

Em segundo lugar, o simulador deve ser capaz de fazer estimativa de consumo a partir de grafos de tarefas. Consequentemente o XPipes também não pode ser utilizado.

Em terceiro lugar, o simulador deve permitir a implementação de técnicas de VS e PM nos processadores para economia do consumo de energia e utilizar como estrutura de comunicação uma NoC malha.

Uma vez que um simulador que contemple todos esses requisitos não foi encontrado, um novo foi implementado, chamado Serpens. Funcionalmente, o modelo do Serpens é semelhante ao utilizado por Hu; Marculescu (2004). Contudo, para tornar o simulador mais realista, custos relativos às tarefas do núcleo do sistema operacional foram acrescentados no modelo de processador.

Ao contrário do modelo de Hu; Marculescu (2004), no Serpens não se define o consumo de potência de cada tarefa, já que ele varia no decorrer do tempo em função do VS. Dessa forma adotou-se em seu lugar o parâmetro α médio (Equação 2.2), que se mantém constante numa arquitetura homogênea.

Para o cálculo do consumo da comunicação, o Serpens utiliza a biblioteca Orion, da mesma forma que o XPipes. A grande diferença entre os dois simuladores é o modelo de processadores utilizado. O Serpens utiliza uma abordagem de escalonamento de grafos de tarefas similar ao SimDVS e Hu; Marculescu (2004), em contraste com o Xpipes, que usa modelos de processadores em nível RTL – *Register Transfer Level*. Esta abordagem torna inviável, no Xpipes, a estimativa do consumo de energia em alto nível a partir de grafos de tarefas.

A linguagem de programação escolhida para a implementação do simulador é SystemC TLM - *Transaction Level Model*, que oferece a abstração requerida e os mecanismos de sincronização de modelos de hardware necessários.

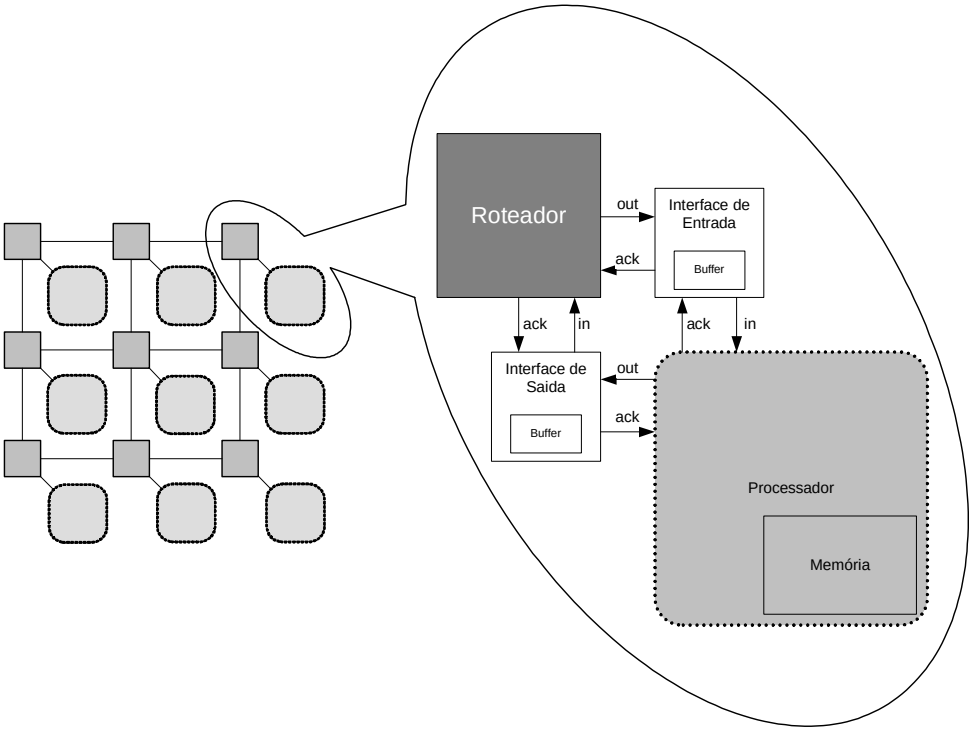

Figura 5.1: SoC implementado no simulador Serpens.

Na Figura 5.1 é apresentada a estrutura do SoC implementado no simulador Serpens. Basicamente, trata-se de uma Noc 4x4, onde cada nodo é composto por um processador, um roteador e interfaces de rede interligando ambos.

As mensagens trocadas por tarefas em processadores diferentes são quebradas em pacotes, de acordo com a estrutura mostrada na Figura 5.2, com carga útil máxima de 256 bytes.

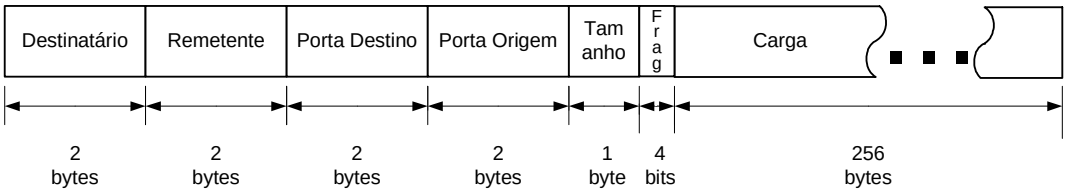

Figura 5.2: Estrutura de um pacote.

Na Figura 5.2, o campo “Frag” 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.

5.1.1 Modelo de Roteador

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; KREUTZ; SUSIN, 2004), desenvolvido para a síntese de sistemas embarcados em FPGA – *Field-programable Gate Array*, com baixo consumo de energia e área.

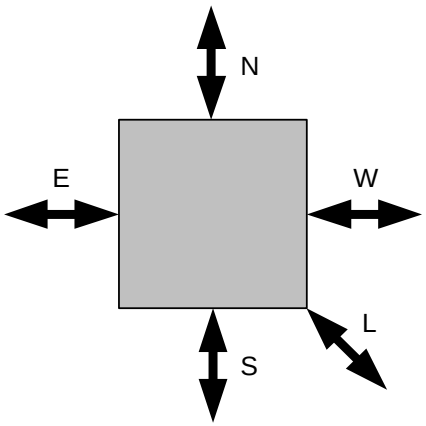

Figura 5.3: Visão externa do RaSoC.

O RaSoC utiliza chaveamento *wormhole*, roteamento XY e controle de fluxo por *handshake*. Externamente, o RASoC é um roteador com 5 portas bidirecionais (Figura 5.3). 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 5.4). Os canais são composto 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.

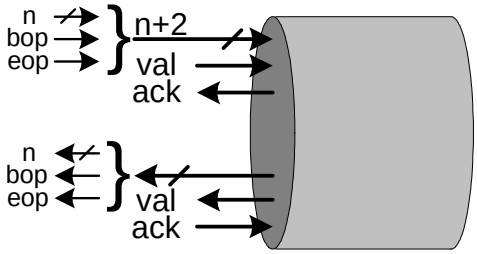

Figura 5.4: Canal de comunicação do RaSoC.

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.

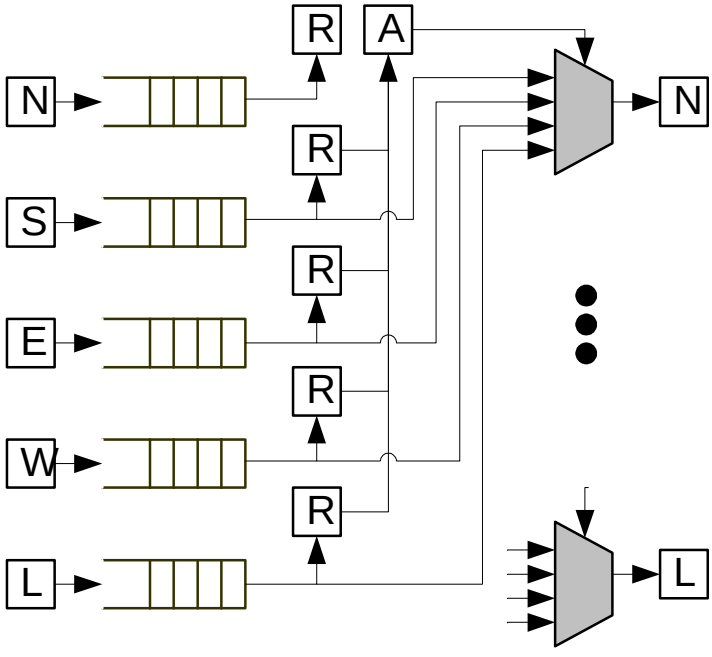

Figura 5.5: Estrutura interna do RaSoC.

Na Figura 5.5 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.

5.1.2 Modelo de 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}, onde C é o número de ciclos de execução em pior caso, T é o período de repetição, D é o *deadline* e α é o número de chaveamentos por ciclo da tarefa.

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.

O valor de α pode ser estimado por ferramentas que simulem os chaveamentos nas portas dos componentes de uma arquitetura em cada ciclo como em Beck, A. C.; Mattos; Wagner; Carro (2003) ou deduzido a partir da equação 2.2, se capacitância de porta, tensão de alimentação, freqüência de operação e consumo de potência das tarefas são conhecidos.

É sabido que o pior tempo de execução raramente ocorre quando as tarefas são executadas, gerando o que se convencionou chamar de folga dinâmica no escalonamento. Para contemplar esse fato no modelo, um parâmetro caracterizando a folga em porcentagem é usado. Por definição, este valor é de 30%, o que significa que o melhor caso de execução é de 70 % do WCET. O tempo de execução da tarefa ativa é calculado aleatoriamente entre estes limites.

5.1.3 Modelo de Processador

O sistema foi projetado para ser um MPSoC de processadores homogêneos, 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. 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; Strosnider (1993), apresentado na Seção 2.3.2, 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, ou seja: $k_i^D \leq k_i^T \ \forall \ k_i \in K$.

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; CHAKRABARTI, 2005) apresentado na Seção 2.4.2.6, que é ao mesmo tempo simples e eficiente.

Para simular as dependências entre as tarefas, um algoritmo de *list-scheduling* foi utilizado, por isso, duas novas listas foram acrescentadas ao modelo de Katcher; Arakawa; Strosnider (1993). Elas têm a função de armazenar as tarefas que estão bloqueadas, aguardando pelo envio e recebimento de mensagens. Somente depois da satisfação de todas as dependências é que cada tarefa pode se juntar as demais tarefas prontas para executar, na lista de tarefas prontas do escalonador.

Os estados convencionais que o processador pode assumir são “Escalonando”, “Executando” e “Aguardando”. Além destes, foram acrescentados os estados “Enviando Pacote”, “Recebendo Pacote” e “Liberando Tarefas”. Os estados Enviando e Recebendo Pacote são alcançados através de chamadas do sistema e interrupções do controlador de rede. Por sua vez, o estado Liberando Tarefas é alcançado em conseqüência de interrupção causada pelo temporizador do escalonador, cuja função é controlar os períodos de liberação das tarefas.

Durante a operação, o escalonador basicamente gerencia suas listas de tarefas. Para se avaliar o consumo de energia e tempo do escalonador, definiu-se quantidades de tempo e consumo de energia para as operações básicas nessas filas, que são basicamente inserções, remoções e varreduras. Esses custos variam em função de a lista ser ordenada ou não. Também foram definidos custos constantes para cada operação. Dessa forma, cada estado consome energia e tempo da mesma forma que uma tarefa de uma aplicação qualquer. Isso torna o modelo significativamente mais realista.

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 NoC, a mesma vai sendo dividida 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 no buffer da 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.

A principal função da interface de rede é possibilitar a troca de pacotes entre o processador e a NoC, que são assíncronos devido ao VS do processador. Dessa forma, a escrita na interface de saída se dá na freqüência do processador e a leitura pelo roteador se dá na freqüência da NoC, e vice-versa.

5.1.4 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.

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 2.7, com $N = 6/bit$ e $k_{design} = 1.2$, de acordo com a Tabela 2.1, obtendo-se:

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

onde n é o número de bits da memória.

O consumo de energia dinâmica do processador é a soma do consumo de cada ciclo, de acordo com a Equação 2.1, 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:

$$E_{din}^{i}(j) = \frac{\alpha_i}{2} C V_{dd}^2 \tag{5.3}$$

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 pelo razão da 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:

$$Pot_{din} = \frac{\sum_{j=1}^{\eta} E_{din}^{i}(j)}{T_C} \tag{5.4}$$

A potência estática é dada pela Equação 2.1, acrescida de um multiplicador representando o número total de portas do processador (N_g):

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

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

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

5.2 Exemplo

A Figura 5.6 apresenta um pequeno grafo de tarefas, composto por 5 tarefas, alocado em três processadores que ocupam as posições (0,0), (1,0) e (1,1) da rede malha.

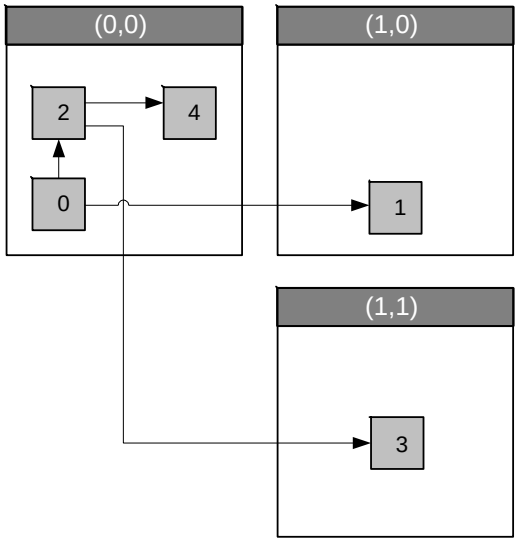

Figura 5.6: Alocação exemplo.

A caracterização de cada tarefa, segundo o modelo de tarefas apresentado na Seção 5.1.2, é mostrada na Tabela 5.1.

Tabela 5.1: Caracterização das tarefas da aplicação.

Tarefa	C (us)	T (us)	D (us)	α
0	1	1000	997	209018
1	1	1000	1000	150930
2	2	1000	999	132631
3	2	1000	1000	177681
4	2	1000	1000	112461

Como dito, para a geração do escalonamento e cálculo do consumo de energia, as tarefas do escalonador também precisam ter seus custos caracterizados. Estes custos podem ser vistos na Tabela 5.2 e foram definidos arbitrariamente. O custo de 100 ciclos é um custo fixo, que ocorre sempre que uma das tarefas do sistema é invocada.

Tabela 5.2: Caracterização das tarefas do sistema.

Id	Tarefa	Ciclos	α
R	Liberação	100	153017
S	Escalonador	100	199802
SP	Enviar pacote	40 ciclos X *phit*	188617
RP	Receber pacote	40 ciclos X *phit*	188617
I	Modo ocioso	-	29388

Adicionalmente, são definidos custos relativos à manipulação das filas do escalonador. Na versão em uso do Serpens estes custos estão fixados, porém sugere-se que em trabalhos futuros esses custos possam ser parametrizáveis:

- 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;
- 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.

Na Figura 5.7 são mostrados os eventos de escalonamento gerados pelo Serpens, referentes à execução da alocação (Figura 5.6) por 100 us, com os custos apresentados.

Em resumo, todas as tarefas são liberadas em todos os processadores. Contudo, todas elas vão para a lista de tarefas bloqueadas, exceto k_o, a primeira a executar no processador (0,0). Consequentemente, os processadores (1,0) e (1,1) permanecem em modo ocioso. Ao finalizar, k_o envia as mensagens de acordo com as dependências do grafo, liberando k_2 em (0,0). Porém, k_1 e k_3 estão em outros processadores e, portanto, mensagens precisam ser enviadas através da NoC, enquanto k_o aguarda bloqueada. A partir daí, começa a execução de k_2, intercalada com o envio dos pacotes da mensagem 1, com destino à tarefa k_1 em (1,0); e da mensagem 2, com destino à k_3 em (1,1). Esses pacotes vão sendo recebidos com alguma latência, primeiro por (1,0) e depois por (1,1). Após receber todos os pacotes, (1,0) desbloqueia k_1 para executar. Ao efetivamente ler

a mensagem 1, k_1 envia um pacote de confirmação de volta para k_o em (0,0), porém k_o só é desbloqueada quando k_3 faz o mesmo em (1,1). Quando k_2 finaliza em (0,0), k_4 é desbloqueada e executada. Após cada processador completar a execução de todas as suas tarefas, eles entram em modo ocioso, aguardando as próximas liberações de tarefas.

Tempo (us)	Processadores		
	(0,0)	(1,0)	(1,1)
0,000	S	S	S
0,376		R	R
1,504		S	S
1,879	R	I	I
5,826	S		
7,142	0		
10,561	SP (1/1)		
21,355	S		
21,961		RP (1/1)	
22,671	2		
28,030	SP (1/2)		
32,488		I	
38,824	S		
39,426		RP (1/2)	
40,140	4		
43,694	SP (1/3)		
47,553	S		
47,929	SP (2/1)		
49,953		I	
50,019		RP (1/3)	
54,344		S	
55,659		1	
56,033		SP (1/ack)	
...			

...			
58,043	SP (2/1)	S	I
58,721	RP (1/ack)		
59,345			RP (2/1)
59,358		1	
60,864	SP (2/2)		
61,229		S	
61,606		I	
69,872			I
71,656	S		
72,032	I		
72,195	SP (2/3)		
72,277			RP (2/2)
76,824	I		
82,804			I
82,907			RP (2/3)
87,984			S
89,299			3
89,673			SP (2/ack)
91,683			S
91,876	RP (2/ack)		
92,998			3
94,020	S		
94,396	I		
96,854			S
97,231			I

Figura 5.7: Eventos de escalonamento gerados pelo Serpens.

Para o cálculo do consumo de energia do escalonamento gerado, utiliza-se a tecnologia 100 nm, com as constantes vistas na Tabela 5.3, obtidas da ferramenta Orion.

Assume-se que cada processador executa com freqüência variando entre 133 e 266 MHz, com tensão de 2 V na freqüência máxima e área de 123608 células lógicas.

Tabela 5.3: Parâmetros da Tecnologia 100 nm.

Parâmetro	Valor
Capacitância de porta C	1,95E-15 F
Leakage de uma célula lógica	7,63E-08 A
Leakage de um Transistor NMOS	2,27E-08 A

Tabela 5.4: Consumo de energia do processador (0,0).

Tarefa	Ciclos η	$\alpha \cdot \eta$	F (MHz)	F_{norm} (MHz)	V_{norm}	V_{dd} (V)	E_{estat} (µJ)	E_{din} (µJ)	E_{total} (µJ)
R	1050	160667850	266	1,0	1,00	2,0	0,074	0,626	0,700
S	350	69930700	266	1,0	1,00	2,0	0,024	0,272	0,296
0	255	53299590	132	0,5	0,65	1,3	0,023	0,087	0,110
SP (1/1)	2872	541708024	266	1,0	1,00	2,0	0,203	2,112	2,315
S	350	69930700	266	1,0	1,00	2,0	0,024	0,272	0,296
2	513	68039703	132	0,5	0,65	1,3	0,047	0,112	0,159
SP (1/2)	2872	541708024	266	1,0	1,00	2,0	0,203	2,112	2,315
S	350	69930700	266	1,0	1,00	2,0	0,024	0,272	0,296
4	473	53194053	132	0,5	0,65	1,3	0,043	0,087	0,130
SP (1/3)	1027	193709659	266	1,0	1,00	2,0	0,072	0,755	0,827
S	100	19980200	266	1,0	1,00	2,0	0,007	0,077	0,084
SP (2/1)	2871	541519407	266	1,0	1,00	2,0	0,203	2,112	2,315
RP (1/ack)	570	107511690	266	1,0	1,00	2,0	0,04	0,419	0,459
SP (2/2)	2871	541519407	266	1,0	1,00	2,0	0,203	2,112	2,315
S	100	19980200	266	1,0	1,00	2,0	0,007	0,077	0,084
I	22	646536	132	0,5	0,65	1,3	0,002	0,001	0,003
SP (2/3)	1232	232376144	266	1,0	1,00	2,0	0,087	0,906	0,993
I	2002	58834776	132	0,5	0,65	1,3	0,096	0,183	0,279
RP (2/ack)	571	107700307	266	1,0	1,00	2,0	0,04	0,42	0,460
S	100	19980200	266	1,0	1,00	2,0	0,007	0,077	0,084
I	120443	3539578884	132	0,5	0,65	1,3	11,101	5,832	16,933
Total							12,53	18,923	31,453

A Tabela 5.4 apresenta o consumo de energia do processador (0,0), calculado segundo o modelo apresentado na seção anterior, gerando as seguintes equações:

$$F_{norm} = \frac{F}{266} \tag{5.7}$$

$$V_{norm} = 0,3 + 0,7F_{norm} \tag{5.8}$$

$$V_{dd} = V_{norm} \cdot 2 \tag{5.9}$$

$$E_{estat} = V_{dd} \cdot 7,63\text{E-}08 \cdot 123608 \cdot \frac{\eta}{F} \tag{5.10}$$

$$E_{din} = \frac{\alpha \cdot \eta}{2} \cdot 1,95\text{E-}15 \cdot V_{dd}^2 \tag{5.11}$$

$$E_{total} = E_{dyn} + E_{estat} \tag{5.12}$$

Como os processadores estão pouco carregados, eles executam as tarefas na freqüência mínima de 133 MHz. Já as tarefas do sistema operacional, por definição, são sempre executadas com freqüência máxima.

Cada processador inclui uma memória de 64 k posições de 32 bits. A potência estática de cada memória é de 617 mW, dados: $V_{dd} = 1,8$ e $n = 32 \cdot 65535$, calculada da seguinte forma, fazendo-se uso da Equação 5.2:

$$P_{estat} = 1,8 \cdot 6 \cdot 32 \cdot 65535 \cdot 1,2 \cdot 2,27\text{E-}08 = 617 \text{ mW} \tag{5.13}$$

O respectivo consumo de energia estático é, então:

$$E_{estat} = 617\,mW \cdot 100\,us = 61\,\mu\text{J} \tag{5.14}$$

Com os parâmetros adotados neste exemplo, o processador apresenta um consumo de energia total de 31 µJ, enquanto que a memória consumiu estaticamente um total de 61 µJ, lembrando que o consumo dinâmico da memória está incluído no fator α. Dessa forma, verifica-se que o consumo de energia estática da memória é o dobro do consumo de energia total do processador, incluindo o consumo dinâmico das memórias. Esse resultado demonstra o impacto significativo da configuração de memória utilizada no consumo total do sistema.

6 ALOCANDO GRAFOS DE TAREFAS COM *BIN-PACKING* EM NOCS MALHA

As NoCs têm muito em comum com redes de computadores. Por conseqüência, é natural que várias técnicas aplicadas em sistemas paralelos e distribuídos sejam adotadas em NoCs, como *bin-packing*, *clustering* e alocação de sub-malhas.

Em se tratando de alocação de sub-malhas, principalmente a alocação contígua parece ser bastante interessante em grandes malhas. Entretanto, isso não se confirma nas menores, pois a fragmentação acaba por representar uma porcentagem considerável do sistema, contrapondo a idéia de uso eficiente dos recursos e limitando a sua eficiência.

A paginação é uma técnica de alocação não contígua que contorna o problema da fragmentação externa. Todavia, ela apresenta fragmentação interna, mesmo quando as páginas são compostas por apenas um processador cada, pois uma aplicação inteira pode ocupar menos que um processador.

Uma das técnicas de alocação convencional mais conhecida é a de *bin-packing*. Ela pode ser utilizada em sistemas com restrições temporais, visto que estes sistemas possuem as noções de utilização e capacidade em comum.

Heurísticas de *bin-packing* foram avaliadas anteriormente para alocação de tarefas em sistemas multiprocessadores com barramento centralizado (AYDIN; YANG, 2003). Porém, os resultados obtidos não foram validados em redes do tipo malha. Apesar disso, elas representam um ponto de partida interessante, principalmente pela idéia de reduzir o número de recursos utilizados no sistema através da alocação.

Outra vantagem do *bin-packing* é a possibilidade do modelo ser estendido para *bin-packing* multidimensional e/ou *vector packing*, passando a considerar também outros recursos necessários por tarefas, como memória e até mesmo energia.

Entretanto, a característica mais interessante de *bin-packing* é o fato de suas heurísticas serem de baixa complexidade, podendo ser facilmente aplicadas para alocação *on-line*.

O modelo de *bin-packing* não contempla comunicação entre tarefas, como o modelo de *clustering*. O *clustering*, por sua vez, não busca reduzir o número de recursos utilizados no sistema. Isso sugere uma combinação de ambas as técnicas para a obtenção de um melhor resultado.

Os alocadores geralmente utilizam um modelo semelhante ao do *bin-packing* e por isso não conhecem as estruturas das aplicações. A partir do momento que técnicas de escalonamento dinâmicas baseadas em teoria de grafos são adotadas, o alocador passa a necessitar conhecer a estrutura da aplicação. Dessa forma, o programador passa a ter

que usar APIs específicas do sistema operacional para informar ao alocador essa estrutura.

Neste capitulo é apresentada a combinação de técnicas de *clustering* e alocação de sub-malhas com *bin-packing*, na tentativa de utilizar *bin-packing* como técnica de alocação dinâmica, visando redução do consumo de energia. O primeiro passo, apresentado na próxima seção, consiste se refazer o experimento de Aydin; Yang (2003) na arquitetura alvo proposta neste trabalho.

Optou-se por utilizar uma abordagem *Best Effort*, de forma que, se não for possível alocar uma tarefa ou *cluster* com uma das heurísticas de *bin-packing* por falta de espaço, o respectivo item será então alocado no processador menos carregado, não existindo assim, garantia de atendimento de *deadlines*.

6.1 Arquitetura Alvo

A arquitetura, implementada pelo Serpens, sobre a qual os experimentos serão realizados é uma NoC 4 x 4 do tipo malha, composta por roteadores RaSoC executando a 133 MHz, interligando 16 processadores FemtoJava (ITO; CARRO; JACOBI, 2001) *pipeline* 32 bits com freqüência variando entre 133 e 266 MHz, e 2 V de tensão na freqüência máxima.

Dados da síntese na ferramenta Leonardo Spectrum mostram que o processador ocupa uma área de 123608 células lógicas, que se considera equivalente a 123608 portas NAND.

De acordo com o modelo apresentado no Capítulo 5, o processador é na verdade um escalonador configurado com dados de arquitetura. A Tabela 5.2 apresenta os custos das tarefas de sistema utilizados nesta configuração. Eles foram estimados a grosso modo, sendo que como trabalho futuro, sugere-se o uso dos números de ciclos e também consumo de energia da API RTSJ – *Real-Time Specification for Java* implementada por Wehrmeister; Becker; Pereira (2004) para o processador FemtoJava. Estes custos podem ser obtidos simulando-se a API e a descrição RTL do processador FemtoJava na ferramenta CACO-PS (CACO, A. C. S.; MATTOS; WAGNER; CARRO, 2003).

O número de chaveamentos por ciclo foi calculado aleatoriamente, com distribuição normal, tanto nas tarefas de sistema (Tabela 5.2) como nas tarefas das aplicações, restrito ao intervalo entre 106696 e 256597 chaveamentos por ciclos, observado em alguns *benchmarks* simulados no CACO-PS, entre eles: seno cordic, *Bubble Sort*, *Select Sort*, *Quick Sort*, busca binária, busca seqüencial, IMDCT, somas de ponto flutuante e decodificador de mp3. A exceção é o modo ocioso, que apresenta um número de chaveamentos por ciclo suficiente para gerar um consumo de potência de 10% da média.

O sistema possui memória distribuída, de forma que cada processador tem sua própria memória local de 64 K posições de 32 bits, com consumo de 617 mW (Equação 5.13).

O roteador utilizado contém cinco portas de entrada, cada uma com seu respectivo buffer. Para estes experimentos, os buffers foram dimensionados para comportarem até 16 *phits*, sendo que cada *phit* ocupa 4 bytes.

O comprimento médio dos enlaces que interligam processadores e roteadores, formando a rede malha, foi definido arbitrariamente em 3 mm.

6.2 Benchmarks

Para avaliação dos métodos utilizados, a alocação e escalonamento são efetuados sobre aplicações obtidas do *benchmark* E3S - *Embedded System Synthesis Benchmark Suíte (DICK; JHA, 2000)*.

O E3S é um conjunto de grafos de tarefas gerados aleatoriamente com o TGFF, com base em dados de tarefas comuns em sistemas embarcados obtidos no EEMBC - *Embedded Microprocessor Benchmark consortium*, como idct, fft e jpeg.

O E3S contém cinco conjuntos de aplicações: automotivo, consumidor, rede, escritório e telecomunicação. Cada grafo tem custos de execução em 17 processadores diferentes, variando de microcontroladores de 40 MHz a *desktops* de 600 MHz.

Para os experimentos neste trabalho, foram selecionados os tempos de execução do processador PPC405 a 266 MHz. Também se assumiu que esses tempos de computação se mantêm na arquitetura alvo utilizada.

Tabela 6.1: Grafos de Tarefas E3S utilizados.

Grafos	# Tarefas	# Arestas	**Utilização total**
Auto-indust-tg0	6	4	**2,7**
Auto-indust-tg1	4	3	**19,1**
Auto-indust-tg2	9	9	**205,5**
Auto-indust-tg3	5	4	**7,33**
Consumer-tg0	7	8	**38,87**
Consumer-tg1	5	4	**96,86**
Networking-tg0	1	0	**62,07**
Networking-tg1	4	3	**173,70**
Networking-tg2	4	3	**176,22**
Networking-tg3	4	3	**143,8**
Office–tg0	5	5	**19,4**
Telecom-tg0	4	4	**14,02**
Telecom-tg1	6	6	**69,09**
Telecom-tg2	4	3	**69,9**
Telecom-tg3	3	2	**8,7**
Telecom-tg4	3	2	**31**
Telecom-tg5	2	1	**95,80**
Total	**76**	**64**	**1234,06**

A Tabela 6.1 apresenta os grafos de tarefas utilizados, discriminando o número de tarefas, arestas e utilização do processador por grafo. Os grafos que apresentam

utilização maior que 100% não são escalonáveis na arquitetura citada. Também não há garantia que os demais o sejam, uma vez que o custo de comunicação não está incluído e vai depender muito da alocação gerada.

Os custos de potência, fornecidos nos grafos do E3S, correspondem na verdade à potência nominal do processador, tratando-se do mesmo valor para todas as tarefas, e de qualquer maneira não são válidos na arquitetura alvo utilizada. Dessa forma, os custos de potência, representados por chaveamentos por ciclos α, foram calculados aleatoriamente em um determinado intervalo observado em alguns *benchmarks* do processador FemtoJava.

6.3 Aplicando *bin-packing* convencional

Inicialmente, se está repetindo o experimento de alocação com heurísticas de *bin-packing* convencional semelhante ao apresentado em Aydin; Yang (2003) na arquitetura alvo definida.

Entretanto, deve ser lembrado que Aydin; Yang (2003) utilizaram uma arquitetura baseada em barramento centralizado e avaliaram o consumo de energia com a heurística quadrado da utilização do processador. Essa heurística é utilizada para comparar apenas o consumo dinâmico e, além disso, não se considera comunicação entre processadores.

Ao comparar as diferentes alocações, Aydin; Yang (2003) verificaram que a freqüência ótima, para a minimização do consumo de energia dinâmico, é constante e mantém a utilização total do processador próximo a 100%. Desse modo, o particionamento ótimo distribui as tarefas uniformemente entre todos os processadores (balanceamento de carga). Consequentemente, a heurística de *bin-packing* mais eficiente é o WF.

6.3.1 Alocação

Ao alocar-se os grafos de tarefas sob a arquitetura alvo, com heurísticas de *bin-packing*, obteve-se as distribuições de utilização por processador apresentadas na Tabela 6.2.

Tabela 6.2: Utilização por processador em cada alocação.

X \ Y	BF				NF				FF				WF			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	99	99	99	99	99	88	88	94	100	100	91	89	97	48	50	99
1	100	99	88	0	100	97	87	0	100	99	85	0	113	102	78	54
2	99	100	99	0	99	95	96	0	100	100	87	0	49	93	50	97
3	100	65	83	0	97	90	98	0	100	100	83	0	51	90	101	62

Como esperado, as heurísticas BF, NF e FF causam concentração de carga, deixando os últimos três processadores livres, enquanto o WF faz balanceamento de carga, utilizando todos os processadores. Os processadores com utilização de 0% são considerados desligados.

Na Tabela 6.3 é apresentado o desvio padrão das utilizações e a quantidade de comunicação inter-processadores. Por ser o mais balanceado, o WF é também o que

apresenta o menor desvio padrão. A comunicação é calculada com a Equação 5.1 e representa a taxa de transmissão em Gigabytes por segundo (GB/s) da alocação gerada.

Tabela 6.3: Utilização e comunicação por alocação.

Algoritmo	Utilização (Desv. Pad. %)	Comunicação (GB/s)
BF	38,1	63,3
NF	37,2	24,8
FF	37,5	64,4
WF	23,4	56,3

Os custos de comunicação obtidos nesse experimento são extremamente altos, da ordem de GB/s. Isso se deve ao fato de que nenhum cuidado especial é dado à comunicação no *bin-packing* convencional, sendo que as tarefas são alocadas como se fossem independentes. Estes custos são estimativas estáticas. Dessa forma, provavelmente não haverá capacidade de processamento e largura de banda suficientes para sua realização.

6.3.2 Consumo de Energia

As alocações geradas foram simuladas durante 100 ms, obtendo-se os consumos dinâmicos apresentados na Tabela 6.5 e estáticos na Tabela 6.4.

Em virtude do maior número de processadores ligados, na verdade todos, o WF apresenta o maior consumo estático, enquanto nas demais alocações foi possível desligar três processadores com as respectivas memórias.

Tabela 6.4: Energia estática (mJ).

Algoritmo	Processador		Memória		NoC		Total
	(mJ)	%	(mJ)	%	(mJ)	%	
BF	19,259	2,3	802,043	95,9	15,084	1,8	836,386
NF	20,221	2,4	802,043	95,8	15,084	1,8	837,348
FF	18,989	2,3	802,043	95,9	15,084	1,8	836,116
WF	23,875	2,3	987,129	96,2	15,084	1,5	1026,088

Notadamente, o custo estático da NoC se aproxima do custo estático dos processadores, muito mais complexos, porque roteadores e interfaces de rede são basicamente memórias. De fato, memória é o elemento estático que apresenta o maior consumo. A memória local representou em média 96% do custo estático do sistema em cada alocação.

Na Figura 6.1 é apresentado o gráfico comparativo do consumo dinâmico, discriminado na Tabela 6.5. A coluna Escalonamento representa o consumo relativo ao Escalonador e ao tratamento de interrupção do temporizador para Liberação de Tarefas. Por sua vez, a coluna comunicação representa o montante de energia despendido no tratamento de comunicação inter-processadores.

Tabela 6.5: Energia dinâmica (mJ)

Algorit.	Processadores + Memória					NoC	Enlaces	Total
	Escalon.	Comunic.	Tarefas	Ocioso	Total			
BF	3,938	84,581	13,826	4,305	106,650	0,236	3,555	110,441
NF	3,847	97,799	33,428	2,493	137,567	0,267	4,020	141,854
FF	4,504	78,618	10,481	4,721	98,324	0,199	2,950	101,473
WF	3,093	117,989	9,133	5,313	135,528	0,310	4,628	140,466

Verifica-se que o consumo dinâmico, referente à NoC e enlaces, é pequeno mesmo quando uma grande quantidade de bytes é transferida. O mesmo ocorre com o custo de escalonamento. Juntos eles representam apenas 6% do consumo dinâmico total. O maior consumo de energia dinâmica foi registrado pela manipulação de mensagens nos processadores, em média 76%, seguida pela execução de tarefas das aplicações, variando de 6 a 23%.

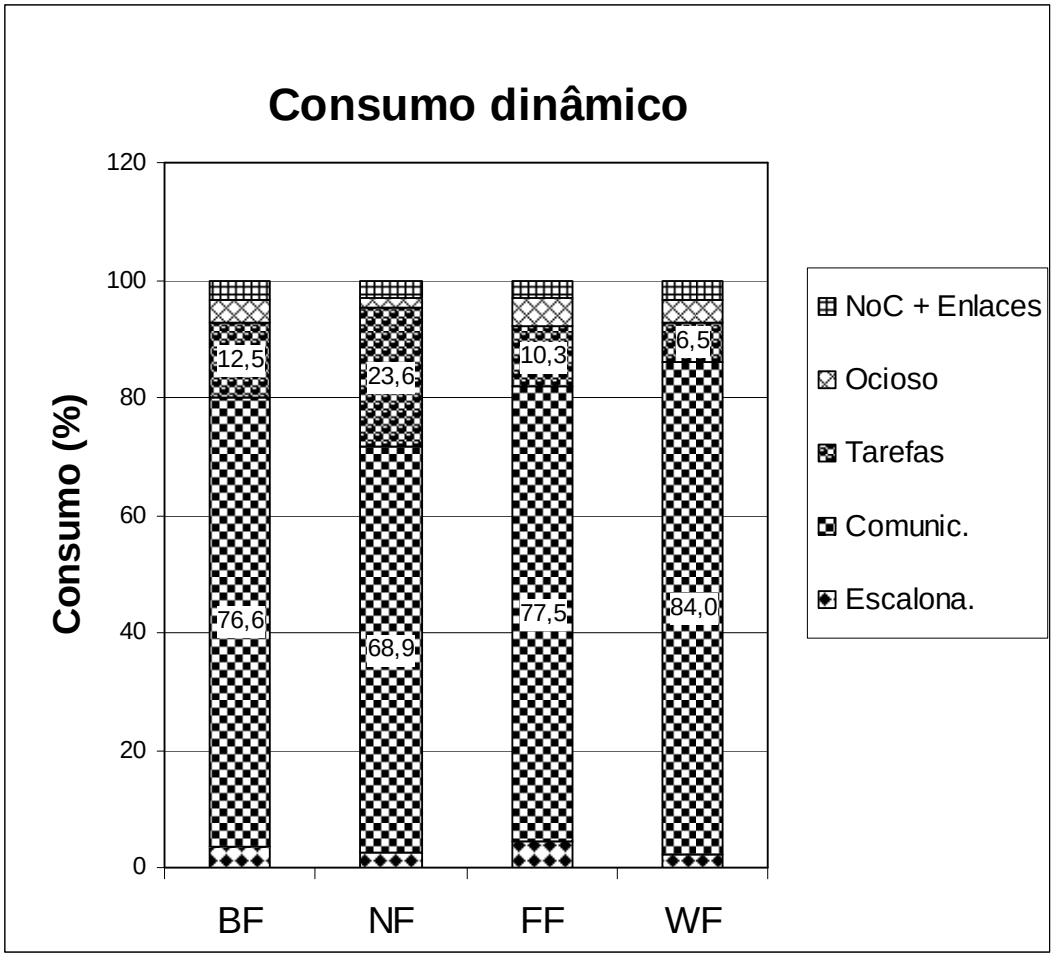

Figura 6.1: Distribuição do consumo dinâmico.

Ao contrário do que era esperado, o consumo de energia do WF não foi o menor de todos. Isso porque, em todas as alocações, a maior parte da energia foi consumida por comunicação.

A comparação entre os consumos estático e dinâmico é feita na Tabela 6.6. Em virtude do grande custo das memórias, o consumo estático foi responsável por pelo menos 85% do consumo. Dessa forma, o WF apresenta o maior consumo de energia, em média 22% maior que os demais.

Tabela 6.6: Consumo estático X dinâmico.

Algorit.	Estático		Dinâmico		Total (mJ)
	(mJ)	%	(mJ)	%	
BF	836,386	88,3	110,441	11,7	946,827
NF	837,348	85,5	141,854	14,5	979,201
FF	836,116	89,2	101,473	10,8	937,590
WF	1026,088	88,0	140,466	12,0	1166,555

6.3.3 Eficiência no atendimento de *deadlines*

Na Tabela 6.7 e na Figura 6.2 está demonstrada a distribuição média de processamento entre as tarefas nos processadores, em cada alocação.

Tabela 6.7: Distribuição de processamento nos processadores (%).

Algorit.	Escalona.	Comunic.	Tarefas	Ocioso	# Process.
BF	1,8	33,3	13,5	51,5	13
NF	1,7	38,5	30,0	29,8	13
FF	2,1	30,9	10,6	56,5	13
WF	1,1	37,7	9,6	51,6	16

Fica claro porque a maior porção do consumo de energia dinâmico é utilizada para enviar e receber dados, pois o sistema passou quase 40% do tempo nesta tarefa, o que não é surpresa dada a taxa de dados a serem transmitidos nas alocações geradas.

Somente em uma pequena parte do tempo o processador ficou dedicado à execução das tarefas das aplicações, em torno de 10%. Para piorar a situação, esse tempo foi menor que o tempo que o sistema passou ocioso, mesmo tendo carga para executar. Isso se deve às dependências entre tarefas alocadas em diferentes processadores, de forma que um processador fica ocioso porque todas as suas tarefas dependem da execução de alguma tarefa em outro processador.

A exceção foi a alocação NF que passou praticamente um terço do tempo em cada uma das tarefas: comunicando, ocioso e executando. Essa variação se dá em conseqüência de atualmente a comunicação não ser uma variável sob controle da alocação, de forma que uma mudança qualquer na alocação pode gerar uma perturbação, trazendo custos de comunicação totalmente diferentes. Também não se deve tirar o mérito do NF, que apresenta a maior localidade entre as heurísticas de *bin-packing*.

No WF a distribuição das tarefas entre os processadores foi maior, por conseqüência surgiram mais dependências, de tal forma que o sistema passou menos tempo dedicado à execução de tarefas que nas demais alocações. Isso não era o esperado, uma vez que o

WF é o que apresenta maior folga em cada processador, que deveria compensar a demanda por processamento maior causada pela comunicação. Na prática, essa alta capacidade de processamento se traduziu apenas em uma quantidade de ociosidade maior que nas demais alocações.

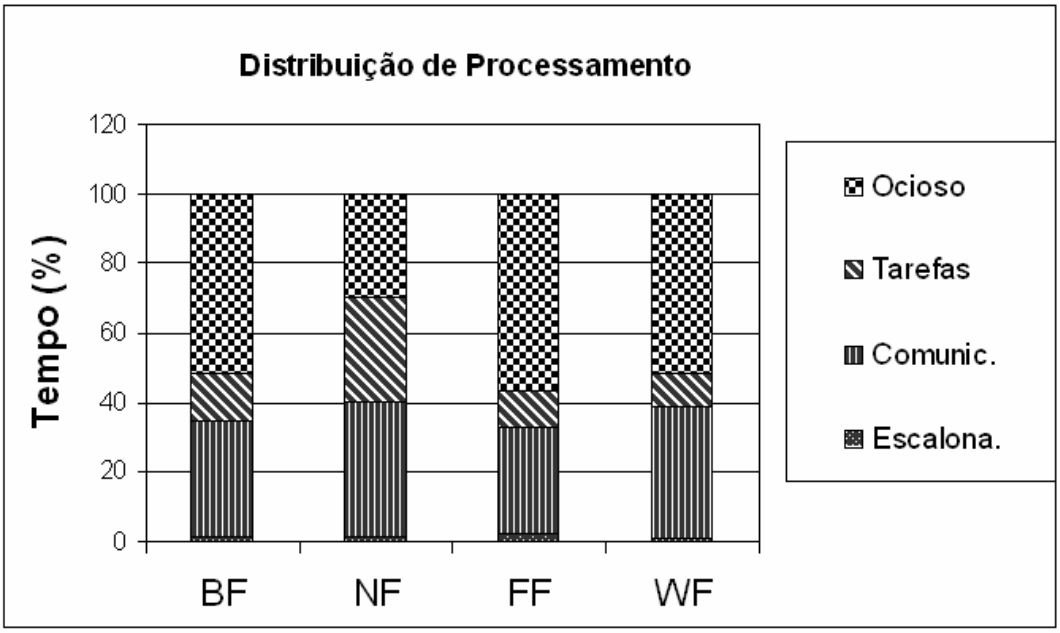

Figura 6.2: Distribuição do tempo de processamento.

Na Tabela 6.8, o impacto das dependências e da comunicação pode ser verificado. As tarefas deveriam ter finalizado em torno de 6984 vezes no período, contudo finalizam somente entre 15 e 25% das vezes. Praticamente, em 70 a 85% das liberações houve perda de *deadline*. Lembrando que quando uma tarefa não finaliza dentro de seu período, uma nova liberação é contada, mas a tarefa não é interrompida, como também ocorre com o *deadline*.

Tabela 6.8: *Deadlines* e finalizações.

Algoritmo	Liberações	Finalizações		*Deadlines* Perdidos	
		Total	%	Total	%
BF	6984	1633	23,4	5275	75,5
NF	6984	2065	29,6	4843	69,3
FF	6983	1885	27,0	5022	71,9
WF	6984	1025	14,7	5883	84,2

6.3.4 Conclusão

Para o conjunto de grafos de tarefas utilizados, verificou-se que o custo de comunicação da NoC, em termos de energia, é irrelevante se comparado ao custo total do sistema. Porém, a comunicação apresenta grande impacto no desempenho e consequentemente no consumo de energia dos processadores.

Os dois fatores identificados como causadores deste alto custo de comunicação são: a quantidade de comunicação entre processadores e, principalmente, as dependências entre tarefas alocadas em processadores diferentes.

O alto custo de comunicação obtido interferiu de tal forma que o WF não se confirmou como o mais eficiente em consumo dinâmico. Ao invés disso, foi o pior no que diz respeito a consumo total do sistema, que inclui o consumo estático.

Porém, o grande vilão identificado são as dependências entre tarefas alocadas em processadores diferentes, que, devido à má distribuição, impediram o fluxo normal de processamento do sistema. A contenção causada foi tamanha, que o sistema passou mais tempo ocioso do que dedicado à execução de tarefas.

Uma abordagem que minimize a comunicação somente poderia facilmente levar a uma pior distribuição das dependências. Dessa forma, o provável ganho obtido com a redução da comunicação se converteria em mais ociosidade com a mesma ineficácia.

Verificou-se que uma alocação utilizando apenas *bin-packing* numa rede malha apresenta desempenho extremamente precário. Dessa forma, seu uso se torna inviável, a não ser que se consiga contornar esse problema.

6.4 Aplicando *Clustering*

Como visto, a grande contenção no processamento de tarefas, encontrada no primeiro experimento, foi causada principalmente pela suas dependências.

Na medida em que *bin-packing* não possui mecanismos para lidar com dependências, resolveu-se combiná-lo a técnicas de *clustering*, especializadas em tratar esse tipo de problema. Para tanto, necessita-se uma técnica cujo objetivo não seja apenas minimizar a comunicação. Por esse motivo, a técnica de Linear *Clustering* (Seção 3.3.4.2) foi adotada.

Nesta seção, o experimento anterior foi refeito. Contudo, antes dos grafos serem alocados, cada um deles passou por um processo de *clustering* com o algoritmo LC. Como resultado, obteve-se um conjunto de *clusters* com paralelismo entre si. Além disso, as arestas que foram zeradas para a formação dos *clusters* eram as que tinham maior peso.

Para a alocação, cada um dos *clusters* é considerado um item atômico, ou seja, é alocado inteiramente em um processador. Em virtude do paralelismo que eles apresentam, não faz sentido alocar mais de um *cluster* de um mesmo grafo em um mesmo processador.

Consequentemente, uma restrição adicional foi imposta ao *bin-packing*: um *cluster* não pode ser alocado em um processador, se nele já houver outro *cluster* do mesmo grafo previamente alocado. Uma exceção é aberta para caso em que todos os processadores já tenham sido utilizados, não havendo outros processadores disponíveis.

6.4.1 Alocação

Após a alocação dos grafos de tarefas, combinando *bin-packing* e *LC*, obteve-se a distribuição de carga apresentada na Tabela 6.9.

Em função do baixo paralelismo de alguns grafos e do tamanho de algumas tarefas, ao se combiná-las para a formação dos *clusters* lineares a soma de suas utilizações

resultou maior que 100%, sobrecarregando os processadores que receberam a alocação desses *clusters*. Estes *clusters* poderiam ser divididos em dois ou mais e alocados em processadores diferentes. Contudo, como visto na Seção 3.3.4.2, se um *cluster* não apresenta paralelismo interno, dividi-lo em mais processadores vai apenas aumentar o comprimento do escalonamento, uma vez que algum custo de comunicação será acrescentado.

Tabela 6.9: Utilização dos processadores ao combinar BP e LC.

X \ Y	BF				NF				FF				WF			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	99	84	99	0	97	84	96	0	95	84	97	0	98	62	14	14
1	99	65	174	0	100	70	174	0	100	65	174	0	175	19	31	46
2	205	144	0	0	205	144	0	0	205	144	0	0	177	96	56	27
3	86	176	0	0	87	176	0	0	94	176	0	0	205	56	14	144

Estes *clusters* de tamanhos exagerados provavelmente formam caminhos críticos nos grafos. O fato de o caminho crítico possuir utilização maior que 100% implica na impossibilidade do mesmo ser escalonável na arquitetura utilizada e nada pode ser feito para evitar a conseqüente perda de *deadlines*.

O acúmulo de carga em alguns processadores, resultante deste processo, possibilitou que menos processadores fossem utilizados. Enquanto no primeiro experimento BF, NF e FF deixaram três processadores livres cada, com *LC* esse número subiu para seis.

Na alocação WF, o impacto foi um pouco diferente. Ao se trocar tarefas por *clusters*, aumentou-se a granularidade dos itens. Consequentemente há uma maior variação na carga dos processadores, prejudicando o balanceamento, como demonstrado pelo desvio padrão na Tabela 6.10, que apresenta ainda o custo de comunicação obtido, que, comparado ao do primeiro experimento, apresenta redução drástica.

Tabela 6.10: Utilização e comunicação ao combinar BP e LC.

Algoritmo	Utilização (Desv. Pad %)	Comunicação	
		Sem LC (GB/S)	Com LC (GB/S)
BF	69,6	63,3	1,2
NF	69,4	24,8	0,1
FF	69,5	64,4	0,1
WF	62,8	56,3	0,1

6.4.2 Consumo de Energia

À primeira vista, o consumo de energia dinâmico (Tabela 6.11) não apresenta grandes diferenças em relação ao primeiro experimento. De fato, o consumo dinâmico é ligeiramente maior.

Na Figura 6.3 é mostrada a distribuição do consumo de energia por alocação no experimento atual. A grande diferença entre os dois experimentos, com e sem LC, é a inversão dos papéis da comunicação e da computação no consumo de energia. Neste experimento, em torno de 70% da energia foi consumida na execução de tarefas e

apenas entre 15 e 30% em comunicação. Esse resultado é exatamente o oposto do primeiro experimento.

Tabela 6.11: Energia dinâmica na alocação BP + LC (mJ)

Algorit.	Processadores + Memória					NoC.	Enlace	Total Com LC	Total Sem LC
	Escalona.	Comunic.	Tarefas	Ocioso	Total				
BF	5,956	31,815	87,901	1,084	126,756	0,080	1,183	128,019	110,441
NF	5,799	19,651	89,209	1,609	116,268	0,046	0,671	116,985	141,854
FF	5,497	16,791	87,843	1,558	111,689	0,036	0,518	112,244	101,473
WF	4,379	43,641	94,540	3,386	145,945	0,105	1,536	147,587	140,466

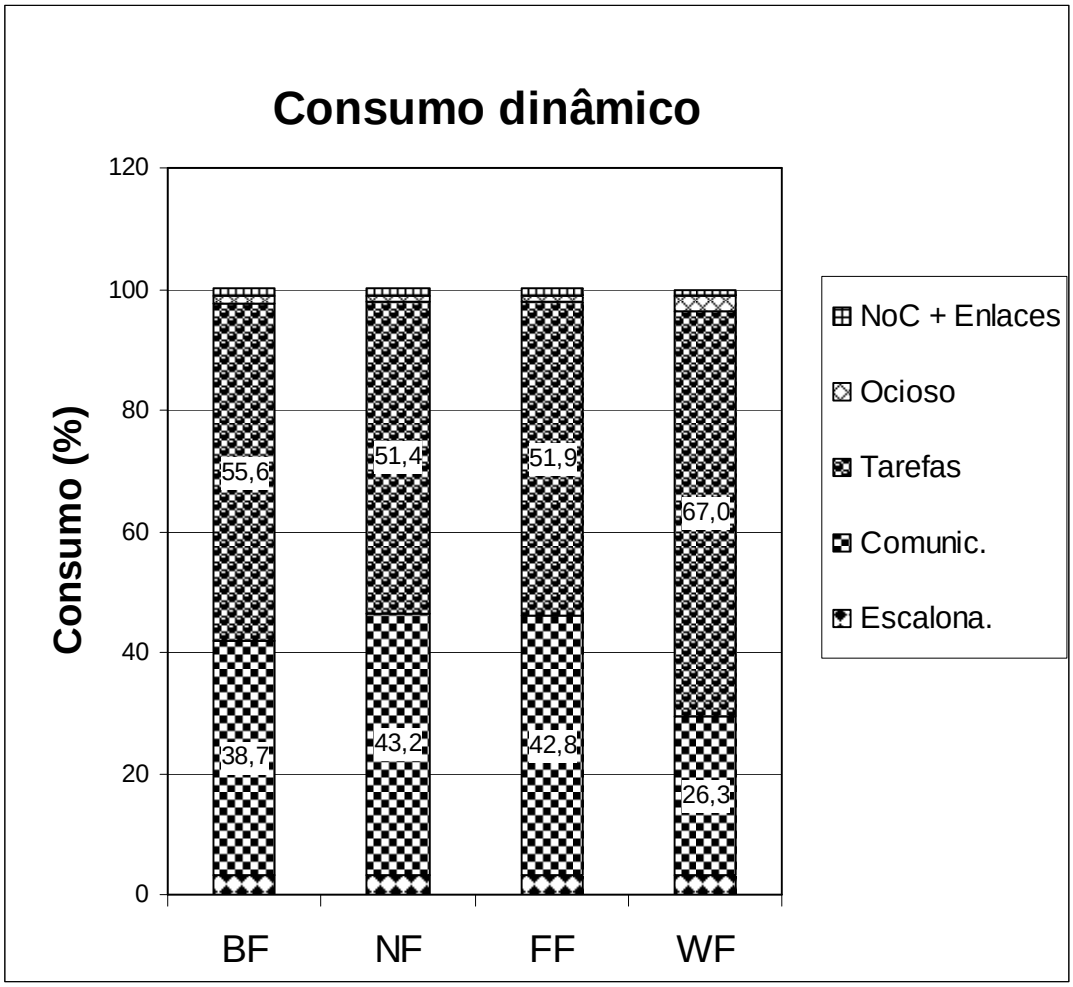

Figura 6.3: Distribuição do consumo dinâmico na alocação *bin-packing* + LC.

A redução da quantidade da comunicação alcançada com o LC teve também como conseqüência a redução do consumo dinâmico da NoC e enlaces em torno de 4 vezes. Contudo, essa redução não tem grande impacto no total, devido a sua pequena proporção.

Novamente, o consumo dinâmico total do WF não foi menor que o das demais alocações, mesmo com a excessiva comunicação do primeiro experimento tendo sido corrigida.

Em relação ao consumo estático (Tabela 6.12), as heurísticas BF, NF e FF apresentaram redução em torno de 20%, em virtude do fato de terem desligado mais processadores e memórias. O mesmo não ocorreu com WF que continuou utilizando todos os processadores.

Tabela 6.12: Energia estática na alocação *BP* + LC (mJ)

Algoritmo	Processador		Memória		Roteador		Total com LC	Total sem LC	Redução (%)
	(mJ)	%	(mJ)	%	(mJ)	%			
BF	16,391	2,5	616,956	95,1	15,084	2,3	648,431	836,386	22,5
NF	16,081	2,5	616,956	95,2	15,084	2,3	648,122	837,348	22,6
FF	15,843	2,4	616,956	95,2	15,084	2,3	647,883	836,116	22,5
WF	24,313	2,4	987,129	96,2	15,084	1,5	1026,527	1026,088	0,0

Em função do decréscimo do consumo estático na ordem de 20% nas alocações BF, NF e FF, e pequenas variações no consumo dinâmico, o consumo total do sistema reduziu também em torno de 20%, exceto no WF que permaneceu quase constante (Tabela 6.13).

Tabela 6.13: Consumo estático X dinâmico na alocação BP + LC.

Algorit.	Estático		Dinâmico		Total com LC	Total sem LC	Redução (%)
	(mJ)	%	(mJ)	%			
BF	648,431	83,5	128,019	16,5	776,450	946,827	18,0
NF	648,122	84,7	116,985	15,3	765,106	979,201	21,9
FF	647,883	85,2	112,244	14,8	760,127	937,590	18,9
WF	1026,527	87,4	147,587	12,6	1174,114	1166,550	-0,6

6.4.3 Eficiência no atendimento de *deadlines*

Na Tabela 6.14 e na Figura 6.4 é apresentada a distribuição dos tempos de processamento. Coerentemente com o consumo de energia, grande parte do tempo passou a ser dedicado à execução de tarefas e uma quantia menor à comunicação, de forma oposta ao experimento sem LC. Nota-se também que o WF apresentou maior quantidade de tempo ocioso. Esse tempo ocioso remanescente é devido ao fato do *clustering* ter deixado vários processadores com pouca carga (menor que 50%) e o algoritmo de VS não pode baixar a freqüência para menos do que 50%.

As perdas de *deadlines* e finalizações de tarefas também passaram para níveis mais aceitáveis em um sistema de tempo real (Tabela 6.15). As perdas de *deadlines* foram reduzidas pela metade e as finalizações no mínimo dobraram. O WF se mostrou o mais eficiente, com o maior número de finalizações, quase 80%, e perdas de *deadline* de 20%.

Tabela 6.14: Distribuição de processamento na alocação BP + LC (%).

Algorit.	Escalona.	Comunic.	Tarefas	Ocioso	# Process.
BF	3,4	16,3	63,5	16,9	10
NF	3,3	10,1	61,6	25,0	10
FF	3,1	8,6	64,0	24,3	10
WF	1,5	14,0	51,6	32,9	16

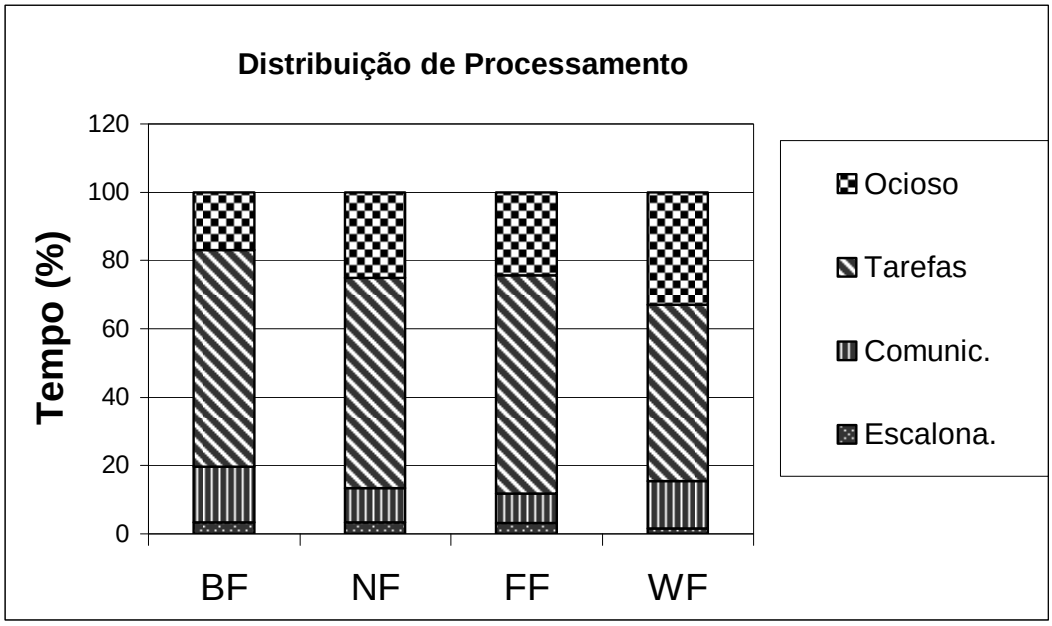

Figura 6.4: Distribuição de processamento na alocação BP + LC.

Tabela 6.15: *Deadlines* perdidos e finalizações na alocação *BP* + LC.

Algoritmo	Liberações	Finalizações		*Deadlines* Perdidos	
		Total	%	Total	%
BF	6984	4647	66,5	2261	32,4
NF	6982	4342	62,2	2564	36,7
FF	6984	4634	66,4	2274	32,6
WF	6984	5547	79,4	1361	19,5

6.4.4 Conclusão

Combinando *clustering* e *bin-packing* não houve melhoria no consumo dinâmico do sistema, mas sim na sua eficiência no atendimento de *deadlines*. O consumo estático reduziu em torno de 20%, em função do menor número de processadores utilizados. Menos processadores foram utilizados porque *clusters* sem paralelismo interno, que

antes ocupavam mais de um processador, agora foram alocados inteiramente em um processador. Ou seja, nesta alocação trocou-se paralelismo inútil por economia de recursos.

Fica evidente a vantagem do uso de *bin-packing*, que tenta reduzir o número de recursos utilizados, num sistema onde o custo estático é significativo. Como conseqüência da combinação das duas técnicas, houve um ganho da ordem de 20% no consumo de energia, com significativa melhoria na eficiência.

Apesar da grande redução, perdas de *deadlines* ainda se mantiveram, principalmente pelo fato de existirem grafos de tarefas não escalonáveis. Isto poderia ser corrigido com aumento da freqüência dos processadores. A aplicação de LC permite que o aumento de freqüência exigido seja bem menor do que no caso de *bin-packing* aplicado isoladamente.

6.5 Aplicando curvas fractais

Um dos problemas encontrados no primeiro experimento é a quantidade de comunicação entre processadores. Leung, J. Y. T.; Whitehead (2002) propuseram modificar a ordem dos processadores durante a alocação em redes malhas, de forma a diminuir a distancia média entre os processadores. O objetivo é tirar proveito da localidade durante a alocação para reduzir também o custo de comunicação. As ordenações utilizadas foram curvas S, Hilbert e H-Index (Figura 3.9), que reconhecidamente preservam o fator localidade, principalmente a Hilbert que é uma curva recursivamente definida.

Os experimentos de Leung, J. Y. T.; Whitehead (2002) indicaram que a escolha da curva é mais importante que o algoritmo usado para a seleção dos processadores ao longo desta, mas ambas escolhas afetam a performance do sistema. Deve-se salientar que foram utilizadas malhas grandes, com em torno de 20 X 20 processadores.

Nos experimentos anteriores, a ordenação de processadores utilizada foi a Em Linha, ou seja, uma linha por vez, começando sempre da esquerda para a direita. Na Tabela 6.16 estão comparados os custos de comunicação em GB/S, obtidos de alocações *bin-packing* combinadas com LC, utilizando curvas Em Linha, S e Hilbert.

Tabela 6.16: BP + LC com Curvas Fractais (Comunicação em GB/S).

Algoritmo	Em Linha	Curva s	Curva Hilbert
BF	1,2	1,4	1,6
NF	0,1	0,3	1,8
FF	0,1	0,3	1,8
WF	0,02	0,04	0,2

A curva S apresentou comunicação um pouco maior que Em Linha, enquanto Hilbert teve um custo de comunicação bem maior que ambas, nas alocações NF, WF e FF.

Apesar de uma diferença da ordem de várias vezes, essa alteração na comunicação não trouxe modificação no consumo de energia (Tabela 6.17) e nem no desempenho (Tabela 6.18) em comparação com o experimento BP + LC em Linha, apresentado na Seção 6.4.

Esse resultado, contrariando Leung, J. Y. T.; Whitehead (2002), provavelmente se deu devido à pouca comunicação resultante após a aplicação do LC; baixa localidade inerente aos algoritmos de *bin-packing*; e principalmente, o tamanho da NoC utilizado, que é pequeno em relação aos experimentos de Leung, J. Y. T.; Whitehead (2002), apresentando pouca alteração na distância média entre os processadores de uma curva para outra.

Tabela 6.17: Consumo de Energia por Curva. (BP+LC).

Alg.	Em Linha			S			Hilbert		
	Din.	Estat	Total	Din.	Estat.	Total	Din.	Estat.	Total
BF	128,0	648,4	776,45	128,3	648,4	776,81	128,0	648,4	776,43
NF	116,9	648,1	765,10	120,5	648,2	768,78	116,5	648,0	764,60
FF	112,2	647,8	760,12	113,0	647,9	760,93	113,7	647,9	761,69
WF	147,5	1026,5	1174,11	149,2	1026,6	1175,79	142,2	1026,3	1168,62

Tabela 6.18: Atendimento de *deadlines* por Curva (BP+LC).

Alg	Em Linha (%)		S (%)		Hilbert (%)	
	Deadlines Perdidos	Finalizações	*Deadlines* Perdidos	Finalizações	*Deadlines* Perdidos	Finalizações
BF	32,4	66,5	31,8	67,1	33,9	65,0
NF	36,7	62,2	34,8	64,2	34,7	64,2
FF	32,6	66,4	32,1	66,8	32,6	66,3
WF	19,5	79,4	19,0	79,9	18,8	80,1

Uma vez que não houve diferenças significativas não é possível definir um vencedor claro. Apesar de a alocação Em Linha ter sempre sido melhor no custo de comunicação estático gerado, consumo e perdas de *deadlines* permaneceram os mesmos. Dessa forma, este experimento não é suficiente para justificar uma escolha. À medida que NoCs maiores e grafos de tarefas com mais comunicação sejam utilizados, os experimentos de Leung, J. Y. T.; Whitehead (2002) devem se confirmar. Por isso, a curva Hilbert foi selecionada e será utilizada nos experimentos a seguir.

6.6 Incluindo custo de comunicação no controle de admissão

Aplicando-se LC houve redução significativa na comunicação entre processadores. A comunicação entre processadores causa interferência no escalonamento, uma vez que tarefas adicionais surgem para tratar esse evento, como mostrado na Figura 5.7.

Essas tarefas adicionais consomem capacidade do processador que não foi considerada no momento da alocação, uma vez que o controle de admissão considera apenas o custo de computação das tarefas. Isso permite que as alocações BF, NF e FF exagerem na concentração de carga, não deixando espaço para a comunicação no escalonamento.

Uma solução possível é reservar uma fatia da capacidade do processador para a comunicação, que poderia ser em torno de 20%, limitando em 80% a utilização disponível para execução de tarefas. Tal esquema apresenta provável desperdício, pois

nem todos os processadores necessitarão exatamente de 20% para comunicação, além do fato de que alguns deles poderão de necessitar mais.

Para resolver esse problema propõe-se considerar o tempo de computação referente à comunicação entre os processadores na alocação, modificando-se o controle de admissão usado até agora no *bin-packing*.

6.6.1 Formulação do Problema

Cada aplicação 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}, onde k_i^C é o número de ciclos de execução em pior caso, k_i^T é o período de repetição, k_i^D é o *deadline* e k_i^α é o número de chaveamentos por ciclo da tarefa.

O conjunto $C = \{c_1, .., c_n\}$ é aquele formado pelos *clusters* de todos os grafos de tarefas $g \in G$, de forma que o *cluster* $c_i = \{k_i, ..., k_m\}$ é um conjunto de tarefas não vazio e $C(g)$ é o conjunto de *clusters* que pertencem ao grafo g.

Neste modelo, cada *cluster* $c_i \in C$ corresponde a um item alocável em um conjunto de processadores $B = \{b_1, b_2, ..., b_k\}$.

A utilização da computação de um *cluster* $s_k(c_i)$ é calculada pela soma das utilizações das tarefas que o compõem:

$$s_k(c_i) = \sum_{\forall k_j \in c_i} \frac{k_j^C}{k_j^T} \in (0,1] \tag{6.1}$$

Para uma aresta $a_{i,j}$ qualquer, seu custo de execução é dado pelo produto do número de pacotes necessários para enviar a mensagem e o tempo gasto para a construção de um pacote. Obtém-se o numero de pacotes da mensagem pela razão entre o peso da respectiva aresta $a^W{}_{i,j}$ e o tamanho máximo de um pacote (Equação 6.2).

$$a_{i,j}^C = \left\lceil \frac{a^W{}_{i,j}}{\max(pack)} \right\rceil \cdot t(pack) \tag{6.2}$$

Consequentemente, a utilização de comunicação de uma aresta ($a_{i,j}^U$) é dada pela razão entre o seu custo de execução $a_{i,j}^C$ e o seu período de repetição:

$$a_{i,j}^U = \frac{a_{i,j}^C}{k_i^T} \in (0,1] \tag{6.3}$$

O custo de comunicação externa de um *cluster* c_i alocado em b_j ($c_i \subset b_j$) é definido como a somatória da utilização de comunicação de todas as arestas inter-*clusters* que o compõem:

$$s_a(c_i) = \sum_{\forall a_{j,l} | k_j \notin c_i \vee k_l \notin c_i} a^U_{i,j} \in (0,1] \quad (6.4)$$

Finalmente, a utilização total de um *cluster* $s(c_i)$ é obitdo pela soma das suas utilizações de comunicação e computação:

$$s(c_i) = s_a(c_i) + s_k(c_i) \quad (6.5)$$

Para um dado processador b_j, $level(b_j)$ representa a sua ocupação atual, calculada pelo somatório da utilização de todos os *clusters* nele alocados:

$$level(b_j) = \sum_{c_i \subset b_j} s(c_i) \quad (6.6)$$

O objetivo é alocar cada item $c_i \in C$ no conjunto de processadores $B = \{b_1, b_2, ..., b_m\}$, de forma a minimizar m, com a restrição adicional de que dois *clusters* que pertencem ao mesmo conjunto $C(g)$, ou seja, ao mesmo grafo de tarefas, não podem ser alocados no mesmo recipiente b_j.

6.6.2 Solução Proposta

A novidade neste modelo é apenas a forma como é calculada a utilização da tarefa no controle de admissão (Equação 6.5) e consequentemente a ocupação do processador (Equação 6.6). Dessa forma, continua-se utilizando heurísticas de *bin-packing* para a alocação.

O grande impacto da alteração é o aumento da complexidade do algoritmo. Contudo, grande parte do processamento pode ser feito *off-line*. Pois, o *clustering* e os cálculos de utilização de computação, tanto dos *clusters* quanto das arestas inter-*clusters*, podem ser feitos em tempo de projeto e fornecidos em uma tabela para o escalonador, já que estes não variam dinamicamente.

O elemento que varia dinamicamente é a utilização de comunicação, que depende de onde *clusters* adjacentes estão alocados. A seguir é apresentado o algoritmo para o cálculo da utilização de um *cluster* c_i em cada recipiente $b_i \in B$:

```
Algoritmo Utilização ( c_i )
   Entrada: cluster c_i ∈ C ;
   Saída: matriz de utilização u[b_1..b_m] ;
Inicio
   Para cada b_i ∈ B
      u[b_i] = s_n(c_i) ;
      Para cada c_j adjacente a c_i
         Se c_j ⊄ b_i
            u[b_i] += a^U_{i,j}
Fim
```

No algoritmo, c_j é cada *cluster* adjacente a c_i. Se c_j ainda não foi alocado, ele é considerado como alocado em um processador diferente de b_i, ou seja, o custo de aresta que interliga os dois *clusters* é computado.

No modelo proposto, a comunicação é considerada como duas tarefas: enviar e receber, uma em cada processador. Contudo, seu tempo de computação é estimado em função do total de bytes e da taxa de transferência. Assim o modelo não considera possíveis contenções na NoC.

6.6.3 Alocação

As alocações obtidas utilizando o modelo proposto na Seção 6.6.1, com LC e curva Hilbert, são apresentadas na Tabela 6.19, na qual apenas as utilizações de computação estão sendo mostradas, apesar de o modelo ter considerado também a utilização da comunicação para alocação.

Tabela 6.19: Utilização na alocação com *bin-packing* modificado + LC + Hilbert.

X \ Y	BF				NF				FF				WF			
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	99	86	84	14	94	86	84	38	100	87	84	33	98	205	62	19
1	97	205	176	144	97	205	176	144	97	205	176	144	175	177	56	96
2	0	51	0,1	96,8	0	31	0,1	96	0	32	0,1	97	27	46	14	31
3	0	2	2	174	0	2	2	174	0	2	2	174	144	14	14	56

Como esperado, a grande diferença em relação à alocação não modificada (ver Tabela 6.9) é que mais processadores foram necessários, já que alguns tiveram parte de sua capacidade ocupada pela comunicação. Na verdade, apenas 2 processadores ficaram livres.

A Tabela 6.20 apresenta o custo de comunicação entre processadores em GB/s obtido com as alocações. A comunicação reduziu para 0,1 GB/s, o menor valor obtido nos experimentos. Chama a atenção que, ao contrário dos experimentos anteriores, o custo de comunicação ficou bastante homogêneo nas quatro alocações.

Tabela 6.20: Comunicação na alocação com BP modificado + LC + Hilbert.

Algoritmo	Utilização (Desv. Pad %)	Comunicação (GB/s)	
		Modificado	Normal
BF	68,5	0,14	1,6
NF	67,9	0,14	0,8
FF	68,2	0,14	0,8
WF	62,8	0,25	0,2

6.6.4 Consumo

O consumo de energia dinâmica obtido (Tabela 6.21) apresenta um acréscimo significativo de até 60%, caracterizado por um maior equilíbrio entre os consumos das alocações BF, NF e FF, deixando para o WF o menor consumo dinâmico. O acréscimo se deu principalmente na quantidade de energia consumida pela execução de comunicação entre processadores.

Devido ao maior número de processadores utilizados, a energia estática apresentou um aumento médio de 39%, exceto para o WF que já utilizava todos os processadores (Tabela 6.22).

Tabela 6.21: Energia dinâmica na alocação com *BP* modificado + LC + Hilbert (mJ).

Algorit.	Processadores + Memória					NoC.	Enlaces	Total Modif.	Total Normal
	Escalona.	Comunic.	Tarefas	Ocioso	Total				
BF	5,591	66,273	95,180	2,171	169,2	0,125	1,744	171,0	128,0
NF	5,683	77,229	91,830	1,707	176,4	0,147	2,049	178,6	116,5
FF	5,802	77,941	94,546	1,755	180,0	0,148	2,068	182,2	113,7
WF	4,429	37,308	94,977	3,589	140,3	0,090	1,313	141,7	142,2

Tabela 6.22: Energia estática na alocação com *BP* modificado + LC + Hilbert (mJ).

Algorit.	Processador		Memória		Roteador		Total Modif.	Total Normal	Aumento (%)
	(mJ)	%	(mJ)	%	(mJ)	%			
BF	22,742	2,5	863,738	95,8	15,084	1,7	901,565	648,414	39,0
NF	22,922	2,5	863,738	95,8	15,084	1,7	901,745	648,083	39,1
FF	23,097	2,6	863,738	95,8	15,084	1,7	901,920	647,918	39,2
WF	24,125	2,4	987,129	96,2	15,084	1,5	1026,338	1026,355	0,0

Na comparação (Tabela 6.23) houve um acréscimo no consumo do sistema de em média 39% para as alocações BF, NF e FF, sendo que para WF o consumo permaneceu constante. O WF não varia porque já utilizava balanceamento de carga máximo, enquanto as outras heurísticas faziam concentração de carga exagerada. O acréscimo do custo de computação da comunicação propiciou um ponto de equilíbrio entre estes extremos.

Tabela 6.23: Energia estática X dinâmica na alocação com *BP* modificado + LC + Hilbert (mJ).

Algorit.	Estático		Dinâmico		Total Modif.	Total Normal	Aumento (%)
	(mJ)	%	(mJ)	%			
BF	901,565	84,1	171,084	15,9	1072,649	776,437	38,1
NF	901,745	83,5	178,644	16,5	1080,389	764,601	41,3
FF	901,920	83,2	182,261	16,8	1084,181	761,693	42,3
WF	1026,338	87,9	141,706	12,1	1168,045	1168,628	0,1

6.6.5 Eficiência no atendimento de *deadlines*

Na Tabela 6.24 e Figura 6.5, é mostrada a distribuição de processamento obtida na simulação das alocações. Houve aumento em praticamente todos os itens, de todas as alocações, exceto WF. Nota-se principalmente um grande aumento de comunicação, que, junto ao tempo ocioso, tomou espaço do tempo dedicado à execução de tarefas, em comparação com a alocação com *bin-packing* normal. Com isso, o acréscimo de processamento, obtido com a inclusão de mais processadores na alocação, foi em grande parte utilizado para execução de comunicação.

Apesar de o sistema ter passado em média menos tempo executando tarefas, as perdas de *deadlines* reduziram entre 10 e 25% conforme mostrado na Tabela 6.25,

enquanto as finalizações aumentaram em torno de 5% nas alocações com concentração de carga. Não houve variação significativa no desempenho do WF.

Tabela 6.24: Distribuição de processamento na alocação com *BP* modificado + LC + Hilbert (%).

Algorit.	Escalona.		Comunic.		Tarefas		Ocioso		# Process.	
	Modif.	Normal	Modif	Normal	Modif.	Normal	Modif.	Normal	Modif.	Normal
BF	2,2	3,3	24,2	16,1	49,4	64,5	24,1	16,1	14	10
NF	2,3	3,4	28,3	10,0	50,5	61,5	19,0	25,1	14	10
FF	2,3	3,1	28,5	9,0	49,7	65,0	19,5	22,9	14	10
WF	1,5	1,5	11,9	11,9	51,6	51,8	34,9	34,8	16	16

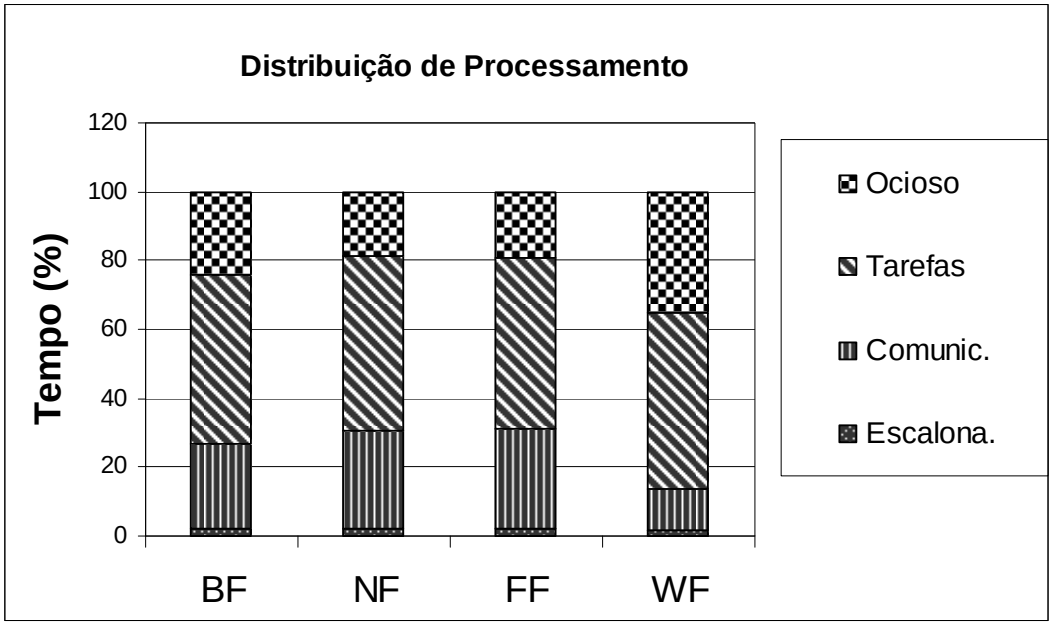

Figura 6.5: Distribuição de processamento na alocação BP modificado + LC + Hilbert.

Tabela 6.25: *Deadlines* e Finalizações na alocação *BP* modificado + LC + Hilbert.

Algorit.	Modificado					Normal	
	Liberações	Finalizações		*Deadlines* Perdidos		Finalizações	*Deadlines* Perdidos
		Total	%	Total	%	%	%
BF	6984	4808	68,8	2070	27,8	65,0	33,9
NF	6984	4894	70,1	1793	28,7	64,2	34,7
FF	6984	4889	70,0	1945	28,4	66,3	32,6
WF	6984	5597	80,1	1867	16,6	80,1	18,8

6.6.6 Conclusão

Neste experimento buscou-se corrigir a concentração de carga excessiva das heurísticas BF, NF e FF, deixando-se espaço no escalonamento para os eventos de

comunicação, representados pelo envio e recebimento de pacotes. Em conseqüência, mais processadores foram utilizados, obtendo-se um maior consumo estático, de forma que o consumo total se aproximou do consumo do WF.

Esperava-se que os níveis de atendimento de *deadlines* das alocações com concentração de carga também se aproximassem dos níveis do WF. Contudo, mesmo com os ganhos obtidos isso não ocorreu, sendo que grande parte da capacidade de processamento adicionada foi direcionada para a execução de comunicação.

Porém, reservar espaço para a comunicação no escalonamento é um passo necessário na busca por escalonabilidade e atendimento de *deadlines*, mesmo em uma alocação WF, na qual não se verificou efeito neste experimento. Na prática, a técnica aplicada não foi suficiente para superar as dependências das tarefas e o volume de comunicação necessário. Portanto, deve-se prosseguir com a busca por técnicas que minimizem dependências e se preocupem também com alocação de banda na NoC.

6.7 Análise final

Na Tabela 6.26 é apresentado o resumo das melhorias alcançadas. São comparadas a alocação inicial (Seção 6.3) e a última realizada (Seção 6.6), onde a alocação inicial utiliza curva em Linha com *bin-packing* convencional e a final curva Hilbert, LC e *bin-packing* modificado.

Tabela 6.26: Resumo dos Resultados.

Algoritmo	Comunicação (GB/s)		Perda *Deadlines* (%)		Consumo (mJ)	
	Inicial	Final	Inicial	Final	Inicial	Final
BF	63,3	0,14	75,5	27,8	946,827	1072,649
NF	24,8	0,14	69,3	28,7	979,201	1080,389
FF	64,4	0,14	71,9	28,4	937,590	1084,181
WF	56,3	0,25	84,2	16,6	1166,555	1168,045

Houve uma melhoria considerável na eficácia do escalonamento com a aplicação das técnicas sugeridas no estudo, demonstrando que *bin-packing* pode ser utilizado para alocação dinâmica com eficiência aceitável se devidamente modificado.

A alocação inicial apresentou desempenho extremamente precário, com perdas de *deadlines* de até 84% para o WF, passando para perdas em torno de 25% na alocação final.

Inicialmente, as alocações BF, FF e NF apresentavam consumo de energia em torno de 15% menor que o WF, pois enquanto este aplicava distribuição de carga, as demais faziam concentração de carga e economizavam energia estática. Contudo, essa concentração era excessiva. Quando, essa distorção foi corrigida no último experimento, os consumos das demais alocações ficaram apenas 8% menores que o consumo do WF.

Aparentemente, a distribuição de carga, para aplicação de VS, mostrou-se mais eficiente que a concentração para aplicação de PM, na configuração utilizada, pois, apesar do maior consumo, apresentou também melhores níveis de atendimento de *deadlines*. Mas essa relação custo benefício depende da proporção entre custo estático e dinâmico da arquitetura em questão, na qual o tamanho de memória utilizado é decisivo.

Na Figura 6.6 é mostrada uma simulação do consumo total de energia das alocações BF e NF, variando-se o tamanho da memória entre 16 e 256 K posições de 32 bits. Inicialmente, BF e WF estão quase empatados, contudo o BF tende a abrir vantagem à medida que o tamanho de memória aumenta. Quando um tamanho de 256 K é utilizado, a diferença entre as duas alocações é de 465 mJ, maior que o custo inicial total com 16 K.

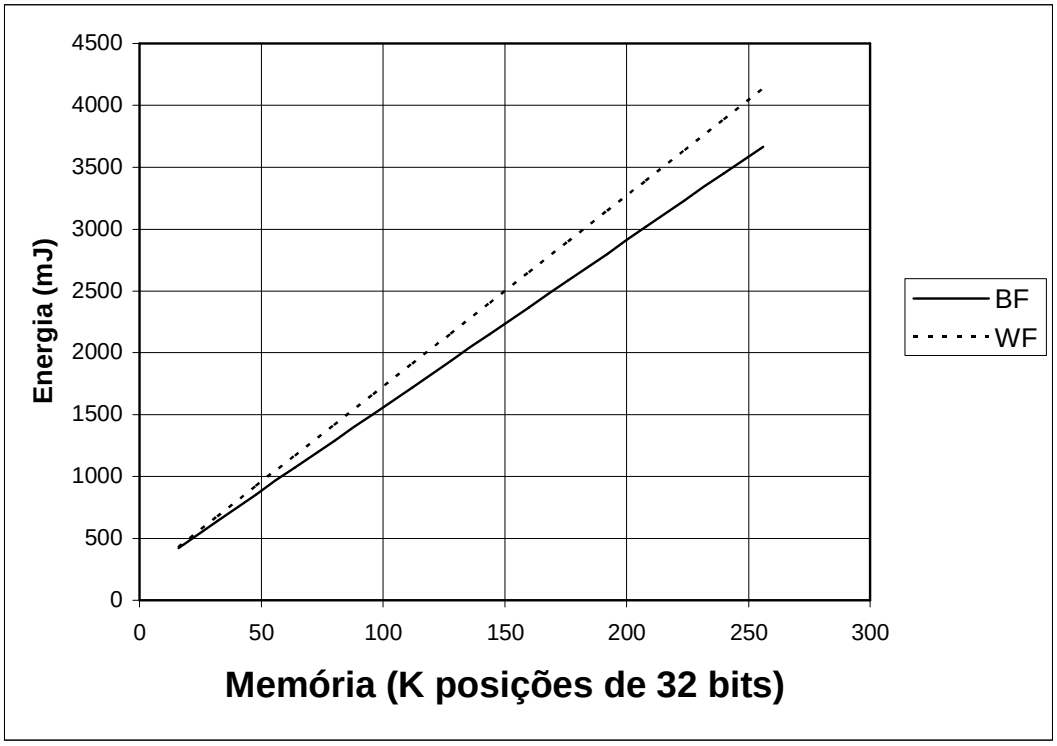

Figura 6.6: Consumo de energia versus tamanho de memória.

Também se está supondo um número de processadores utilizados segundo as alocações efetuadas neste capítulo. Essa relação de consumo entre alocações pode variar bastante em função da carga total do sistema. O BF é mais indicado para sistema com alto custo estático, ou baixa utilização, enquanto o WF se apresenta melhor na situação oposta. Em um sistema dinâmico, uma abordagem combinando as duas técnicas poderia ser utilizada, de acordo com o sistema e seu estado atual.

Outra questão que não foi considerada é a possibilidade do uso de técnicas de redução do consumo de energia das memórias, desligando-se células que não estão em uso, economizando seu custo estático. Considerando-se um modelo ideal, a quantidade de memória ligada seria a mesma, nas diferentes alocações experimentadas, com concentração de carga ou não. Dessa forma, o custo de memória seria uma constante e o WF seria o melhor algoritmo de todos, por possuir também o melhor atendimento de *deadlines* e menor consumo dinâmico. Contudo, essa situação ideal não existe e sugere-se esse estudo como um trabalho futuro.

7 CONCLUSÃO

Este trabalho teve como objetivo propor técnicas de alocação dinâmica de tarefas periódicas sobre os processadores de uma NoC malha, visando redução do consumo de energia em nível de sistema, ou seja, durante o escalonamento das tarefas, e simultaneamente atender restrições de tempo real.

Os resultados e constatações deste trabalho são um primeiro estudo para o desenvolvimento de um particionador dinâmico no contexto citado.

Adotou-se um modelo distribuído onde cada nodo do sistema é autônomo, tendo seu próprio escalonador local. As políticas e técnicas de escalonamento utilizadas foram EDF para escalonamento, DAR para VS e uma política simples de PM, onde um processador sem carga de trabalho foi considerado desligado, incluindo sua memória local.

O escalonamento distribuído de tarefas foi um tema bastante estudado nas últimas décadas. Entretanto, a abordagem usual é a de maximização de desempenho, em sistemas de alta performance, enquanto que em sistemas embarcados o foco se transfere para a questão do consumo de energia, apesar de o desempenho ainda ser relevante. Muitas destes trabalhos consideram estruturas de comunicação do tipo barramento centralizado e não podem ser diretamente aplicados em redes malha com os mesmos resultados.

No decorrer deste trabalho, foram consultadas fontes referentes a escalonamento distribuído, principalmente em redes malha, sendo que várias destas técnicas são citadas. Contudo, não se deu ênfase a protocolos de escalonamento distribuído, pois o objeto do estudo é a definição de uma heurística de alocação.

Uma vez que um sistema dinâmico tende a fragmentar e perder desempenho, mesmo que uma heurística ótima seja utilizada, preferiu-se heurísticas de alocação eficientes e rápidas.

Na medida em que existem poucas técnicas de alocação dinâmica em NoCs com restrições temporais e minimização do consumo de energia, optou-se por aplicar técnicas de alocação convencionais, como *bin-packing* e *clustering*, no contexto de sistemas embarcados.

Uma das primeiras questões surgidas no inicio do trabalho foi o método pelo qual seria feita a avaliação do consumo de energia de alocações. Acabou-se por adotar um modelo de grafos de tarefas, dadas as dificuldades de se utilizar sistemas reais para experimentação. Neste modelo, utilizado para implementar a ferramenta Serpens, cada grafo de tarefas representa uma aplicação independente a ser executada no sistema.

De posse do simulador e técnicas de alocação convencionais, a abordagem utilizada foi a de gerar estaticamente as alocações para depois serem simuladas, apesar de se estar buscando um alocador dinâmico. Contudo, esse artifício foi válido, já que num primeiro momento interessava apenas conhecer o comportamento das heurísticas e utilizou-se heurísticas de baixa complexidade, que podem ser facilmente aplicadas *on-line*. Porém, futuramente, será necessário implementar o alocador como parte do simulador, de forma que o custo de desempenho e energia do mesmo faça parte dos custos da alocação.

Ao aplicar-se *bin-packing* convencional obteve-se desempenho extremamente precário, com perdas de *deadlines* de até 84% para heurística WF. Foram identificados o custo de comunicação, e principalmente as dependências entre as tarefas, como os responsáveis por este resultado. Ao aplicar-se a técnica de LC que visa maximizar o paralelismo e reduzir as dependências, obteve-se perdas de *deadlines* muito inferiores. Fica claro que tentar reduzir a comunicação somente, sem a preocupação com dependências e paralelismo, não é suficiente. Ao final, a heurística WF combinada com LC obteve perdas de *deadlines* de apenas 16%, demonstrando que é possível utilizar *bin-packing* para alocação on-line, desde que combinada com técnicas que tratem as dependências entre tarefas.

As heurísticas BF, FF e NF aplicam concentração de carga, enquanto WF aplica balanceamento de carga. Dessa forma, WF tira proveito da técnica de VS para economia de energia, enquanto as demais heurísticas parecem mais úteis para aplicação de PM. Dessa forma, inicialmente, as alocações BF, FF e NF apresentavam consumo de energia em torno de 15% menor que o WF, já que o custo estático do sistema era predominante, tendo a memória o custo mais significativo neste critério.

Contudo, ao se aplicar o conceito de utilização de tarefa, comum em modelos de tarefas periódicas, combinado ao *bin-packing*, gera-se alocações que não reservam espaço para comunicação entre processadores. Para o WF isso não é um problema tão grande, porque este tende a deixar folgas nos processadores. Para as outras heurísticas, ao contrário, acaba ocorrendo uma concentração excessiva de carga.

Tentou-se corrigir esta distorção modificando-se o controle de admissão utilizado, acrescentando-se neste o custo de computação das arestas inter-*clusters*. Dessa forma, esperava-se que as heurísticas de concentração de carga apresentassem eficiência semelhante ao WF, porém ainda economizando energia. Quando essa distorção foi efetivamente corrigida no último experimento, os consumos de energia de BF, NF e FF ficaram em torno de apenas 8% menores que o consumo do WF, com perdas de *deadlines* de 25% contra 16% do WF.

Porém, reservar espaço para a comunicação no escalonamento é um passo necessário na busca por escalonabilidade e atendimento de *deadlines*, mesmo em uma alocação WF, na qual não se verificou efeito.

Conclui-se que a técnica sozinha não foi suficiente para superar as dependências das tarefas e o volume de comunicação necessário. Portanto, deve-se prosseguir com a busca por técnicas que minimizem dependências e se preocupem também com alocação de banda na NoC.

Aparentemente, a distribuição de carga, para aplicação de VS, mostrou-se mais eficiente que a concentração para aplicação de PM, na configuração utilizada, pois, apesar do maior consumo, trouxe também melhores níveis de atendimento de *deadlines*. Entretanto, essa relação custo benefício é bastante influenciada pela proporção entre os

custos estático e dinâmico da arquitetura em questão, na qual o tamanho de memória utilizado é decisivo.

Outro fator de impacto na comparação entre as heurísticas é a carga de operação do sistema. O BF é mais indicado para sistema com alto custo estático, ou baixa utilização, enquanto o WF se apresenta melhor na situação oposta. Em um sistema dinâmico, uma abordagem combinando as duas técnicas poderia ser utilizada.

Em última instância, o WF é uma heurística que resulta numa alocação com carga balanceada. Portanto, algoritmos de balanceamento de carga dinâmicos, utilizados em sistemas distribuídos, poderiam ser usados em seu lugar, com o mesmo resultado.

Neste trabalho não se considerou a possibilidade do uso de técnicas de redução do consumo de energia das memórias, desligando-se células que não estão em uso. Desse modo, se fosse possível manter apenas a quantidade de memória necessária ligada, o custo de memória seria constante independendo da alocação e novamente o critério do consumo dinâmico seria decisivo, no qual o WF é mais eficiente.

Contudo, essa situação ideal não existe e sugere-se esse estudo como um trabalho futuro, incluindo-se no Serpens a quantidade de memória utilizada por cada tarefa. Com essa modificação, seria possível também a aplicação de migração de tarefas e a avaliação de seus custos, mesmo que esta se restringisse apenas à carga inicial das tarefas, que também não foi considerada neste estudo e é de suma importância.

A partir do momento que as tarefas tenham custo de memória e não apenas de processamento, tem-se um fator limitador da quantidade de tarefas que podem ser alocadas em um processador. Sugere-se, para a solução deste problema, como continuação deste trabalho, o uso de *bin-packing* multidimensional, já que existem extensões, como *vector packing*, que permitem a existência de dimensões independentes ou não, possibilitando a modelagem de sistemas multi-objetivos, como neste caso. Além disso, outros recursos adicionais, como energia, poderiam também ser acrescentados.

Nos modelos utilizados, considerou-se apenas a existência de processadores homogêneos. Outra extensão importante é a de permitir o uso de organizações diferentes de uma mesma arquitetura, ou até mesmo de arquiteturas diferentes, de forma a permitir a exploração do compromisso entre energia e desempenho dos elementos do SoC. O impacto direto dessa extensão é que o custo de computação k_i^C de cada tarefa passaria a ser um vetor ao invés de um escalar.

O Serpens foi desenvolvido com o intuito de estimar o consumo de energia de alocações, mas pode ser utilizado como ferramenta de avaliação de políticas de NoC, como roteamento, arbitração e armazenamento temporário de pacotes, com bastante rapidez, por ser um modelo implementado em nível TLM. Entre as melhorias necessárias em seu modelo estão a implementação do consumo estático dos enlaces e da arbitragem por prioridades nos roteadores.

O modelo de processador pode ser estendido com um escalonador em nível de sistema, de forma a oferecer o serviço de escalonamento para o desenvolvimento de aplicações compostas por software e hardware. Dessa forma, seria fornecida uma camada de abstração que separaria logicamente o modelo de software da API do SystemC, permitindo sua compilação sem grandes modificações na arquitetura alvo, desde que a API seja a mesma nos dois sistemas e a linguagem seja C++.

O acréscimo de uma API de comunicação, nos mesmos moldes, permitiria também interfacear a aplicação, rodando sobre o modelo de escalonador, com outros modelos de software ou hardware descritos em SystemC, comunicando-se através de canais TLM.

REFERÊNCIAS

ADAM, T. L.; CHANDY, K.; DICKSON, J. A Comparison of List Scheduling for Parallel Processing Systems. **Journal of Communications of the ACM**, New York, NY, USA, v.17, n.12, p.685-690, Dec.1974.

AHMAD, I.; KWOK, Y.-K.; WU, M.-Y. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In: INTERNATIONAL SYMPOSIUM ON PARALLEL ARCHITECTURES, ALGORITHMS, AND NETWORKS, I-SPAN, 2., 1996, Beijing, China. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1996. p. 207-213.

ALENAWY, T. A.; AYDIN, H. Energy-Aware Task Allocation for Rate Monotonic Scheduling. In: IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM, RTAS, 11., 2005, Toronto, Canada. **Proceedings...** San Francisco, CA: IEEE Computer Society, 2005. p. 213-223.

AUDSLEY, N. C. et al. Applying new scheduling theory to static priority pre-emptive scheduling. **Software Engineering Journal**, Piscataway, NJ, USA v.8, n.5, p.284-292, 1993.

AYDIN, H.; YANG, Q. Energy-Aware Partitioning for Multiprocessor Real-Time Systems. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 17., 2003, Nice, France. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p. 113-123.

BABBAR, D.; KRUEGER, P. On-Line Hard Real-time Scheduling of Parallel Tasks on Partitionable Multiprocessors. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ICPP, 23., 1994, North Carolina, USA. **Proceedings...** North Carolina, USA: CRC Press, 1994. p. 29-38.

BACKER, E.; JAIN, A. A clustering performance measure based on fuzzy set decomposition. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Piscataway, NJ, USA v.PAMI-3, n.1, p.66–75, Jan.1981.

BECK, A. C. S.; MATTOS J. C. B.; WAGNER F. R.; CARRO L. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, São Paulo, SP, Brasil. **Proceedings...** Los Alamitos, California: IEEE Computer Society, 2003. p. 349- 354.

BECK, J.; SIEWIOREK, D. Modeling Multicomputer Task Allocation as a Vector Packing Problem. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, ISSS, 9., 1996, La Jolla, California. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1996. p. 115-121.

BENINI, L.; BOGLIOLO, A.; MICHELI, G. D. A Survey of Design Techniques for System-level Dynamic Power Management. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, San Diego, CA, USA, v.8, n.3, p.299-316, June 2000.

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

BERTOZZI, D.; BENINI, L. Xpipes: A Network-on-chip Architecture for Gigascale Systems-on-Chip. **IEEE Circuits and Systems Magazine**, Piscataway, NJ, USA, v.4, n.2, p.18-31, 2004.

BUNDE, D. P.; LEUNG, V. J.; MACHE, J. Communication Patterns and Allocation Strategies. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 18., 2004, Santa Fe, New Mexico. **Proceedings...** Washington, DC, USA: EEE Computer Society, 2004. p. 284-292.

BURNS, A. **Real-time systems and programming languages.** 2nd ed. Harlow: Addison-Wesley, 1997.

BUTTS, J. A.; SOHI, G. S. A Static Power Model for Architects. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 33., 2000, Monterey, California. **Proceedings...** New York, NY, USA: ACM Press, 2000. p. 191-201.

CASAVANT, T. L.; KUHL, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. **IEEE Transactions on Software Engineering**, Piscataway, NJ, USA, v.14, n.2, p.141-154, Feb.1988.

CHANDRAKASAN, A.; BRODERSEN, R. **Low Power Digital CMOS Design.** Norwell, MA, USA: Kluwer, 1995.

CHENG, A. M. K. **Real-Time Systems:** Scheduling, Analysis, and Verification. Hoboken: Wiley-Interscience, 2002.

DERTOUZOS, M. L. Control Robotics: The Procedural Control of Physical Processes. In: IFIP CONGRESS, 1974, Stockholm. **Information Processings 74:** proceedings. Amsterdan: North-Holland, 1974. p. 807-813.

DICK, R. P.; JHA, N. K. COWLS: hardware-software cosynthesis of wireless low-power distributed embedded client-server systems. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, VLSID, 13., 2000, Calcutta, India. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2000. p. 114-124.

DICK, R. P.; RHODES, D. L.; WOLF, W. TGFF: Task Graphs for Free. In: INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN, CODES, 6., 1998, Seattle, WA, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1998. p. 97-101.

DUATO, J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks:** An Engineering Approach. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.

GAREY, M. R. **Computers and intractability:** a guide to the theory of np-completeness. San Franciso: W. H. Freeman, 1979.

GERASOULIS, A.; YANG, T. On the Granularity and Clustering of Directed Acyclic Task Graphs. **IEEE Transactions on Parallel and Distributed Systems,** Piscataway, NJ, USA, v.4, n.6, p.686-701, June 1993.

GRAHAM, R. L. Bounds for Certain Multiprocessing Anomalies. **Bell System Technical Journal**, New York, v.45, p.1563-1581, Nov.1966.

GRUIAN, F. Hard real-time scheduling for low-energy using stochastic data and DVS processors. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 2001, Huntington Beach, California, USA. **Proceedings...** New York, NY, USA: ACM Press, 2001. p. 46-51.

HU, J.; MARCULESCU, R. Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXPOSITION, DATE, 2004, Paris, France. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2004. p. 234-239.

HUA, S.; QU, G. Power Minimization Techniques on Distributed Real-Time Systems by Global and Local Slack Management. In: ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 13., 2005, China. **Proceedings...** New York, NY, USA: ACM Press, 2005. p. 830-835.

HWANG, I. An Efficient Processor Allocation Algorithm Using Two-Dimensional Packing. **JPDC: Journal of Parallel and Distributed Computing**, Orlando, FL, USA, v.42, n.1, p.75-81, Oct.1997.

HWANG, J.-J. et al. Scheduling precedence graphs in systems with interprocessor communication times. **SIAM Journal on Computing**, Philadelphia, PA, USA, v.18, n.2, p.244-257, April 1989.

ISHIHARA, T.; YASUURA, H. Voltage scheduling problem for dynamically variable voltage processors. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISLPED, 1998, Monterey, California, USA. **Proceedings...** New York, NY, USA: ACM Press, 1998. p. 197-202.

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

JERRAYA, A.; TENHUNEN, H.; WOLF, W. Multiprocessor Systems on-Chips. In: IEEE DESIGN AUTOMATION CONFERENCE, DAC, 41., 2005, San Diego, CA, USA. **Proceedings...** New York, NY, USA: IEEE Computer Society, 2005. p. 681-684.

KATCHER, D. I.; ARAKAWA, H.; STROSNIDER, J. K. Engineering and Analysis of Fixed Priority Schedulers. **IEEE Transactions on Software Engineering,** Piscataway, NJ, USA, v.19, n.9, p.920-934, Sept.1993.

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

KIM, W.; KIM, J.; MIN, S. A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE & EXHIBITION, DATE, 2002, Paris, France. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2002. p. 788-794.

KIM, W. et al. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-Time Systems. In: REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM, RTAS, 18., 2002, San Jose, CA, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2002. p. 219-228

KRUEGER, P.; LAI, T. H.; RADIYA, V. A. Job Scheduling is More Important than Processor Allocation for Hypercube Computers. **IEEE Transactions on Parallel and Distributed Systems**, Piscataway, NJ, USA, v.5, n.5, p.488 - 497, May 1994.

KUMAR, R. et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 36., 2003, San Diego, CA. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p. 81-92.

LAUZAC, S.; MELHEM, R.; MOSSÉ, D. Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor. In: EUROMICRO WORKSHOP REAL-TIME SYSTEMS, ECRTS, 10., 1998, Berlin, Germany. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1998. p. 188-195.

LEUNG, J. Y. T.; WHITEHEAD, J. On the Complexity of Fixed-Priority Scheduling of Periodic Real-time Tasks. **Performance Evaluation**, [S.l.], v.4, n.2, p.237-250, Dec.1982.

LEUNG, V. J. et al. Processor Allocation on Cplant: Achieving General Processor Locality Using One-Dimensional Allocation Strategies. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 4., 2002, Chicago, Illinois. **Proceedings...** Las Alamos, CA, USA: IEEE Computer Society, 2002. p. 296–304.

LI, K.; CHENG, K. H. A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. In: ACM ANNUAL COMPUTER SCIENCE CONFERENCE, 1990, Washington, D.C., USA. **Proceedings...** New York, NY, USA: ACM Press, 1990. p. 22-27.

LI, Q.; YAO, C. **Real-time concepts for embedded systems.** San Francisco, CA: CMP Books, 2003.

LIU, C. L.; LAYLAND, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. **Journal of the ACM**, New York, NY, USA, v.20, n.1, p.46-61, 1973.

LO, V. et al. Noncontiguous processor allocation algorithms for mesh-connected multicomputers. **IEEE Transactions on Parallel and Distributed Systems**, Piscataway, NJ, USA, v.8, n.7, p.712-726, July 1997.

LODI, A.; MARTELLO, S.; MONACI, M. Two-dimensional packing problems: A survey. **European Journal of Operational Research**, Amsterdam, v.141, n.2, p.241-252, 2002.

LORCH, J. R.; SMITH, A. J. Improving dynamic voltage scaling algorithms with PACE. **ACM SIGMETRICS Performance Evaluation Review**, New York, NY, USA, v.29, n.1, p.50-61, June 2001.

MACHE, J.; LO, V.; WINDISCH, K. Minimizing message-passing contention in fragmentation-free processor allocation. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING SYSTEMS, PDCS, 10., 1997, New Orleans, Louisiana, USA. **Proceedings...** [S.l.]: The International Society for Computer and Their Applications, 1997. p. 120-124.

MARCON, C. et al. Modeling the Traffic Effect for the Application Cores Mapping Problem onto NoCs. In: IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION, VLSI-SoC, 13., 2005, Perth, Australia. **Proceedings...** New York: Springer, p.101-107.

MISHRA, R. et al. Energy Aware Scheduling for Distributed Real-Time Systems. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, IPDPS, 17., 2003, **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p. 21-24.

MOHAPATRA, P. Dynamic real-time task scheduling on hypercubes. **Journal of Parallel and Distributed Computing**, Orlando, FL, USA, v.41, n.1, p.91-100, Oct.1997.

MOK, A. K.; DERTOUZOS, M. L. Multiprocessor scheduling in a hard real-time environment. In: IEEE TEXAS CONFERENCE ON COMPUTING SYSTEMS, 7., 1978, Houston, Texas. **Proceedings...** New York: IEEE Press, 1978. p. 5-1-5-12.

MONCUSI, M. A.; ARENAS, A.; LABARTA, J. Energy Aware EDF Scheduling in Distributed Hard Real Time Systems. In: IEEE INTERNATIONAL REAL-TIME SYSTEMS SYMPOSIUM - WORK-IN-PROGRESS, RTSS, 24., 2003, Cancun, Mexico **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p. 103-106.

MUDGE, T. Power: A First-Class Architectural Design Constraint. **Computer**, Los Alamitos, CA, USA, v.34, n.4, p.52-58, April 2001.

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

OKUMA, T.; YASUURA, H.; ISHIHARA, T. Software Energy Reduction Techniques for Variable-Voltage Processors. **IEEE Design & Test**, Los Alamitos, CA, USA, v.18, n.2, p.31-41, Mar.2001.

PILLAI, P.; SHIN, K. Real-Time dynamic voltage scaling for low-power embedded operating systems. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, SOSP, 18., 2001, Banff, Alberta, Canada. **Proceedings...** New York, NY, USA: ACM Press, 2001. p. 89-102.

RICHARD, M. K.; MICHAEL, L.; MARCHETTI-SPACCAMELA, A. A probabilistic analysis of multidimensional bin packing problems. In: ACM SYMPOSIUM ON THEORY OF COMPUTING, 16., 1984, Washington, DC. **Proceedings...** New York, NY, USA: ACM Press, 1984. p. 289 - 298

SARKAR, V. **Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.** Stanford, CA, USA: Stanford University, 1989.

SELVAKUMAR, C.; MURTHY, C. S. R. Scheduling Precedence Constrained Task Graphs with Non-Negligible Intertask Communication onto Multiprocessors. **IEEE Transactions on Parallel and Distributed Systems**, Piscataway, NJ, USA, v.5, n.3, p.328-336, Mar.1994.

SHAW, A. C. **Sistemas e Software de Tempo Real.** Porto Alegre: Bookman, 2003.

SHIN, D. et al. SimDVS: An Integrated Simulation Environment for Performance Evaluation of Dynamic Voltage Scaling Algorithms. In: WORKSHOP ON POWER-AWARE COMPUTER SYSTEMS, PACS, 2002, Cambridge, MA, USA. **Proceedings...** [S. l.]: Springer, 2002. p. 141-156.

SHIN, Y.; CHOI, K. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In: ANNUAL CONFERENCE ON DESIGN AUTOMATION, DAC, 36., 1999, New Orleans, Louisiana, USA. **Proceedings...** New York, NY, USA: ACM Press, 1999. p. 134-139.

SHIN, Y.; CHOI, K.; SAKURAI, T. Power optimization of real-time embedded systems on variable speed processors. In: IEEE/ACM INTERNATIONAL CONFERENCE ON

COMPUTER-AIDED DESIGN, ICCAD, 2000, San Jose, California. **Proceedings...** Piscataway, NJ, USA: IEEE Press, 2000. p. 365-368.

SIH, G. C.; LEE, E. A. A compile-time scheduling heuristic for interconnection-constrainedheterogeneous processor architectures. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.4, n.2, p.175-187, Feb.1993.

TANENBAUM, A. S. **Distributed operating systems.** 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1995.

TIWARI, V.; MALIK, S.; WOLFE, A. Power analysis of embedded software: a first step towards software power minimization. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, 1994, San Jose, CA, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. p. 384 - 390

WAGON, S. **Mathematica in Action.** New York: Freeman, 1991.

WANG, H. Orion: A Power-performance Simulator for Interconnection Networks. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 35., 2002, Istanbul, Turkey. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society Press, 2002. p. 294-305.

WEHRMEISTER, M. A.; BECKER, L. B.; PEREIRA, C. E. Optimizing Real-Time Embedded Systems Development Using a RTSJ-Based API. In: WORKSHOP ON JAVA TECHNOLOGIES FOR REAL-TIME AND EMBEDDED SYSTEMS, JTRES, 2., 2004, Larnaca, Cyprus. **Proceedings...** Berlin: Springer, 2004. p. 292-302.

WEISER, M. et al. Scheduling for Reduced CPU Energy. In: ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, FOCS, 36., 1995, Milwaukee, Wisconsin. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 1995. p. 13-23.

WELCH, G. F. A Survey of Power Management Techniques in Mobile Computing Operating Systems. **ACM SIGOPS Operating Systems Review**, New York, NY, USA, v.29, n.4, p.47-56, Oct.1995.

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

WU, M. Y.; GAJSKI, D. D. Hypertool: a programming aid for message-passing systems. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.1, n.3, p.330, May 1990.

XU, R.; II, W. D. Survey of clustering algorithms. **IEEE Transactions on Neural Networks**, Los Alamitos, CA, USA, v.16, n.3, p.645-678, May 2005.

YANG, T.; GERASOULIS, A. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. **IEEE Transactions on Parallel and Distributed Systems,** Los Alamitos, CA, USA, v.5, n.9, p.951-967, Sept.1994.

YAO, F.; DEMERS, A.; SHENKER, S. A Scheduling Model for Reduced CPU Energy. In: SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE, FOCS, 36., 1995, Milwaukee, Wisconsin, USA. **Proceedings...** Washington, DC, USA: IEEE Computer Society Press, 1995. p. 374-382

YE, T. T.; BENINI, L.; MICHELI, G. D. Analysis of Power Consumption on Switch Fabrics in Network Routers. In: DESIGN AUTOMATION CONFERENCE, DAC, 39., 2002, New Orleans, Louisiana, USA. **Proceedings...** New York, NY, USA: ACM Press, 2002. p. 524-529.

YUAN, W.; NAHRSTEDT, K. Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, SOSP, 19., 2003, Bolton Landing, NY, USA. **Proceedings...** New York, NY, USA: ACM Press, 2003. p. 149-163.

ZEFERINO, C.; SUSIN, A. C. SoCIN: a parametric and scalable network-on-chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, SBCCI, 16., 2003, Sao Paulo, BR. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2003. p. 169–174.

ZEFERINO, C. A.; KREUTZ, M. A.; SUSIN, A. A. RASoC: A Router Soft-Core for Networks-on-Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, DATE, 2004, Paris, France. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2004. p. 198-205.

ZHANG, Y.; HU, X. S.; CHEN, D. Z. Task scheduling and voltage selection for energy minimization. In: DESIGN AUTOMATION CONFERENCE, DAC, 39., 2002, New Orleans, Louisiana, USA. **Proceedings...** New York, NY, USA: ACM Press, 2002. p. 183-188.

ZHU, D.; MELHEM, R.; CHILDERS, B. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems. In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, RTSS, 22., 2001, London, UK. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2001. p. 84-94.

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