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

MÁRCIO EDUARDO KREUTZ

Método para a Otimização de Plataformas Arquiteturais para Sistemas Multiprocessados Heterogêneos

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

Prof. Altamiro Amadeu Susin
Orientador

Porto Alegre, junho de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Kreutz, Márcio Eduardo

Método para a Otimização de Plataformas Arquiteturais para Sistemas Multiprocessados Heterogêneos/Márcio Eduardo Kreutz – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

177 p.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação , Porto Alegre, BR-RS, 2005. Orientador: Altamiro Amadeu Susin.

1. Otimização arquitetural 2. Projeto baseado em Plataformas 3. Networks-on-Chip (NoCs). I. Susin, Altamiro Amadeu. II. Título.

AGRADECIMENTOS

Gostaria primeiramente agradecer à instituição federal na qual tive a oportunidade de realizar o curso, provendo todo o suporte necessário. Mesmo que dificuldades sejam inerentes ao processo, é de grande satisfação saber que ainda existe a preocupação no país com a formação de recursos humanos. Nesse sentido sinto-me muito gratificado pela constante confiança e ajuda que recebi de diversos professores, destacando-se os professores Flávio Wagner e Luigi Carro. Ao meu orientador, Prof. Altamiro Amadeu Susin, a sensação de ter a sorte de ter conhecido, antes de tudo, um grande ser humano. Esse trabalho é sem dúvida, uma tentativa de realizar parte de seus sonhos. Ao Prof. Manfred Glenser por ter me acolhido na Alemanha com muito apreço e amizade. Agradeço também aos colegas de curso, tanto pela grande amizade, quanto pelo apoio técnico. Dentre tantos, gostaria de mencionar em especial Cesar Marcon, Cesar Zeferino e Leandro Indrusiak, o qual, além de tudo, apoiou-me sempre que necessário em minha estadia no exterior. Finalmente, agradecimentos muito especiais a Solange Thomaz, que além do suporte emocional, acabou dando a forma necessária a esse texto. São pessoas tão importantes que me permitem acreditar que uma tese deva ser um exercício de criatividade e inteligência e não o cumprimento da desconfiança.

SUMÁRIO

LISTA DE ABREVIATURAS **9**
LISTA DE FIGURAS **11**
LISTA DE TABELAS **13**
RESUMO **15**
ABSTRACT **17**
1 INTRODUÇÃO **19**
1.1 Caracterização dos Sistemas Eletrônicos Considerados para o Método **21**
1.2 Questões sobre o Projeto **22**
1.3 Objetivos e Contribuições Esperadas **23**
2 O PROCESSO DA CONCEPÇÃO DE SISTEMAS ELETRÔNICOS COMPLEXOS **27**
2.1 O Método dos Refinamentos Sucessivos **27**
2.1.1 Níveis de Descrição 29
2.2 Fluxo de Projeto para Sistemas Complexos **34**
2.3 Abordagens na Concepção de Sistemas Eletrônicos **36**
2.3.1 Conceitos sobre Projeto baseado em Plataformas 38
2.4 O Processo de Síntese e Ferramentas de Apoio ao Projeto **41**
2.4.1 Particionamento HW/SW, Co-projeto e Especificação 43
2.4.2 Ferramentas para Análise e Otimização de Arquiteturas de Processamento 45
2.4.3 Ferramentas para Análise e Otimização de Arquiteturas de Comunicação 48
2.4.4 Sistemas Operacionais Dedicados 51
2.4.5 Ferramentas para Análise e Otimização de Sistemas Completos 51
2.5 Conclusões **54**
3 OPERAÇÕES PARA OTIMIZAÇÃO DE PLATAFORMAS ARQUITETURAIS **57**
3.1 Definição de Aplicações Alvo e Plataformas Arquiteturais para Processamento e Comunicação **57**
3.2 Otimização arquitetural de Plataformas Arquiteturais **60**
3.3 Conclusões **64**
4 ESPECIFICAÇÃO DE UM MÉTODO PARA OTIMIZAR PLATAFORMAS ARQUITETURAIS **65**
4.1 O Problema encontrado para a Otimização de Plataformas **65**
4.2 Objetivos **66**
4.3 Fluxo de Projeto proposto para o Método **67**
4.4 Ferramentas de Apoio ao Projeto Consideradas para o Método **72**
4.5 Conclusões **73**
5 MODELO DE PROGRAMAÇÃO PARA ESPECIFICAÇÃO DE APLICAÇÕES DEDICADAS E PLATAFORMAS ARQUITETURAIS HETEROGÊNEAS **75**
5.1 Forma Canônica do Modelo de Programação **77**

5.2 Modelagem de Modelos de Computação 80
5.3 Níveis de Abstração Considerados no Modelo de Programação 82
5.4 Modelos para Componentes Arquiteturais de Processamento 83
5.5 Modelos para Componentes Arquiteturais de Comunicação 85
5.6 Modelos para Sistemas Completos 87
5.7 Síntese e Composição de Componentes Arquiteturais 88
5.8 Exemplos de Plataformas Arquiteturais 92
5.9 Conclusões 95
6 FERRAMENTAS DE APOIO AO PROJETO PARA OTIMIZAÇÃO DE PLATAFORMAS ARQUITETURAIS 97
6.1 Objetivos e o Contexto das Ferramentas para o Fluxo de Projeto do Método . 97
6.2 Especificação de Componentes para Plataformas Arquiteturais 97
6.2.1 Uso do Conceito de Frameworks Orientado a Objetos para a Especificação de Plataformas 98
6.3 Mapeamento entre Níveis Hierárquicos 99
6.4 Abordagem de Simulação para Componentes Arquiteturais 99
6.5 Análise, Reconfiguração e Otimização de Componentes Arquiteturais 100
6.5.1 Interface de Controle e Introspecção Computacional 101
6.5.2 O Padrão de Projeto “Proxy” 102
6.5.3 Otimizações em Componentes para Processamento 103
6.5.4 Otimizações em Componentes para Comunicação 104
6.5.5 Otimizações para Plataformas Completas 104
6.6 Conclusões 105
7 PLATAFORMAS ARQUITETURAIS PARA COMUNICAÇÃO 107
7.1 Componentes Arquiteturais Considerados 107
7.2 Frameworks para Arquiteturas de Comunicação 108
7.3 Ajuste das Funções Custo 111
7.4 Aplicações Testadas 113
7.5 Otimizações Consideradas para Arquiteturas de Comunicação 113
7.5.1 Posicionamento dos Processadores às Portas Locais de Roteadores de Redes Intrachip 115
7.5.2 Avaliação de Topologias 116
7.5.3 Políticas de Escalonamento 117
7.5.4 Redes Heterogêneas 118
7.6 Experimentos com Arquiteturas de Comunicação 121
7.7 Conclusões 128
8 PLATAFORMAS ARQUITETURAIS PARA SISTEMAS DEDICADOS COMPLETOS 129
8.1 Componentes Arquiteturais Considerados 131
8.2 Frameworks para Sistemas Multiprocessados Dedicados 132
8.3 Ajuste das Funções Custo 133
8.4 Aplicações Testadas 135
8.5 Otimizações Consideradas para Plataformas Arquiteturais para Sistemas Multiprocessados Completos 142
8.5.1 Particionamento de Tarefas 144
8.5.2 Conjunto de Processadores Heterogêneo 144
8.5.3 Sistema Completo 147
8.5.4 Experimentos com Sistemas Completos 148
8.6 Conclusões 157
9 CONSIDERAÇÕES FINAIS 159

9.1 Evolução do trabalho 160
REFERÊNCIAS 161

LISTA DE ABREVIATURAS

ASIC	Application Specific Integrated Circuit
ASIP	Application-Specific Instruction-Set Processor
CDFG	Control-Data Flow Graph; Grafo de Fluxo de Dados e Controle
CISC	Complex Instruction-Set Computer
DAMQ	Dynamically Allocated, Multiple Queue
DSP	Digital Signal Processing; Processamento Digital de Sinais
FIFO	First In-First Out
FPGA	Field Programmable Gate-Array
GA	Genetic Algorithms; Algoritmos Genéticos
GPP	General Purpose Processor; Processador de Propósito Geral
IP	Intelectual Property; Propriedade Intelectual
MPSoCs	Multiprocessors Systems-on-Chip
MoC	Model of Computation
NoC	Network-on-Chip; Redes Intrachip
PC	Parte de Controle
PbD	Platform-based Design; Projeto baseado em Plataformas
QoS	Quality-of-Service; Qualidade-de-Serviço
RA	Roteador Arquitetural
RASoC	Router Architecture for Systems-on-Chip
RISC	Reduced Instruction-Set Computer
SA	Simulated Annealing
SAMQ	Statically Allocated, Multiple Queue
Sashimi	System as Software and Hardware in Microcontrollers
SoC	System-on-Chip
SoCIN	System-on-Chip Interconnection Network
SoCINhet	System-on-Chip Interconnection Network Heterogeneous
TS	Tabu Search; Algortimo de Busca Tabu
VCI	Virtual Component Interconnect
VHDL	Very high speed integrated circuits Hardware Description Language
VLIW	Very Large Instruction Word Computer

LISTA DE FIGURAS

Figura 2.1: Níveis de Abstração na Concepção de Sistemas Eletrônicos 31
Figura 2.2: Níveis de Abstração para Componentes de Software 33
Figura 2.3: Domínios de especificação para aplicações dedicadas 34
Figura 2.4: Fluxo de Projeto para Sistemas Eletrônicos Complexos 35
Figura 2.5: Arquitetura de Rede Chaveado do tipo Árvore Gorda - SPIN 52
Figura 2.6: Arquitetura do barramento CoreConnect 52
Figura 3.1: Relação hierárquica entre blocos básicos, tarefas e processadores. 58
Figura 3.2: Pseudo-Código para o Algoritmo “Busca Tabu” 63
Figura 4.1: Fluxo de Projeto para a Concepção de Sistemas Completos 68
Figura 4.3: Representação UML do Fluxo de Projeto do Método 71
Figura 5.1: Forma Canônica para Diagrama de Classes UML 78
Figura 5.2: Forma Canônica para Diagrama de Colaborações 79
Figura 5.3: Forma Canônica para Diagrama de Seqüências 79
Figura 5.4: Projeto baseado em *Interfaces* 81
Figura 5.5: Diferentes versões funcionais definidas através de “projeto baseado em interfaces” 82
Figura 5.6: Diagramas de Classes (a) e Seqüências (b) para Componentes de Processamento no Nível de *Sistema* 84
Figura 5.8: Diagramas de Classes (a) e Seqüências (b) para Componentes de Processamento no Nível de *Transações* 85
Figura 5.9: Diagramas de Classes (a) e Seqüências (b) para Componentes de Comunicação no Nível de *Sistema* 86
Figura 5.10: Diagramas de Classes (a) e Seqüências (b) para Componentes de Comunicação no Nível de *Mensagens* 86
Figura 5.11: Diagramas de Classes (a) e Seqüências (b) para Componentes de Comunicação no Nível de *Transações* 87
Figura 5.12: Conexão de Processadores à Arquiteturas de Comunicação do tipo Network-on-Chip 88
Figura 5.13: Diagrama de classes UML para o Padrão de Projeto para Síntese e Composição 89
Figura 5.16: Modelo UML de Processadores de Propósito Geral com *Pipeline* 93
Figura 5.17: Ciclo de Busca em Processadores Pipeline 94
Figura 5.18: Ciclo de Execução em Processadores Pipeline 94
Figura 5.19: Modelo UML Genérico de Roteadores para Redes-em-Chip 95
Figura 6.1: Interface de Controle 102
Figura 6.2: Padrão de projeto *Proxy* 102
Figura 6.3: Relações entre as Ferramentas de Apoio ao Projeto 103
Figura 7.1: Relação hierárquica encontrada em descrições de Redes Chaveadas 110

Figura 7.2: Diagrama de Seqüências para MoC Arquitetural para Comunicação no nível de Mensagens 111
Figura 7.3: Diagrama de Seqüências para MoC Arquitetural para Comunicação no nível de Transações 111
Figura 7.4: Visão parcial das topologias mesh torus para NoCs 112
Figura 7.5: O Problema do Posicionamento dos Processadores em NoCs 115
Figura 7.6: Redes Diretas: (a) Grelha 2-D; (b) Torus 2-D; (c) Hipercubo 3-D 117
Figura 7.7: Redes Indiretas: (a) Crossbar 4x4; (b) Multiestágio 8x8 bidirecional 117
Figura 7.8: Arquiteturas dos Roteadores Rasoc, Tonga e Mago 120
Figura 7.9: Desempenho e Latência para diferentes Topologias de Redes Chaveadas ... 122
Figura 7.10: Desempenho e Latência para Redes com “Canais Virtuais” 122
Figura 7.11: Otimização da Posição dos Processadores na Rede 123
Figura 7.12: Evolução da busca realizada pelo algoritmo Busca Tabu 124
Figura 7.13: Otimização da Política de Escalonamento para uma Aplicação Síncrona .. 124
Figura 7.14: Otimização da Política de Escalonamento para uma aplicação balanceada 125
Figura 7.15: Exploração do espaço de projeto para as estratégias EP/LM e LP/EM 126
Figura 7.16: (a) Comparação do consumo de energia e latência para três redes homogêneas. (b) Compromisso Energia/Latência para cinco aplicações executando sobre redes homogêneas 127
Figura 7.17: Exploração do espaço de projeto para redes heterogêneas 128
Figura 8.1: Modelos de ULAs para diferentes tipos de processadores 132
Figura 8.2: Configurações em Processadores VLIW 133
Figura 8.4: Diagramas de Classes e de Seqüências UML para a Aplicação de Roteamento IPv4, versão distribuída 138
Figura 8.5: Pseudocódigo para a Aplicação para Análise de Ativos Farmacológicos, abordagem distribuída. 140
Figura 8.7: Diagramas UML de Classes e de Seqüências para a Aplicação do Jogo de Futebol 141
Figura 8.8: Pseudo-código para a função OPTIMUM() do algoritmo Busca Tabu para as operações de Particionamento e Mapeamento 146
Figura 8.9: Resultados para arquiteturas de comunicação, aplicação roteamento IPv4 a 10Gb/s 150
Figura 8.10: Resultados para arquiteturas de comunicação, aplicação para análise de ativos farmacológicos complexos 150
Figura 8.11: Resultados para arquiteturas de comunicação, aplicação Jogo de Futebol . 151
Figura 8.13: Resultados para arquiteturas de processamento, aplicação de roteamento IPv4 a 10Gb/s 152
Figura 8.15: Resultados para arquiteturas de processamento, aplicação Jogo de Futebol 153
Figura 8.16: Resultados para arquiteturas de processamento, abordagem VLIW para RISC, aplicação de roteamento IPv4 a 10Gb/s 154
Figura 8.18: Resultados para arquiteturas de processamento, abordagem VLIW para RISC, aplicação do Jogo de Futebol 155
Figura 8.20: Compromisso $Processamento_x Comunicação$ para a aplicação de roteamento IPv4 a 10Gb/s, processadores ASIP 156
Figura 8.21: Compromisso $Processamento_x Comunicação$ para a aplicação de roteamento IPv4 a 10Gb/s, processadores VLIW 156
Figura 8.22: Compromisso $Processamento_x Comunicação$ para a aplicação de roteamento IPv4 a 10Gb/s, processadores RISC e arquitetura de barramento 157

LISTA DE TABELAS

Tabela 7.1: Componentes Arquiteturais implementados para diferentes Topologias do tipo Rede Chaveada 109
Tabela 7.2: Área, consumo de energia e tempo de execução para os roteadore Rasoc, Tonga e Mago 121

RESUMO

A concepção dos sistemas eletrônicos previstos para o futuro próximo implica em uma atividade multidisciplinar, a qual demanda, para o projeto de sistemas eletrônicos, o uso de métodos e técnicas provenientes de diversos domínios do conhecimento humano. Esses domínios podem variar desde a especificação de aplicações, até a realização física de circuitos integrados.

A constante evolução dos processos de fabricação de circuitos integrados permite a criação de circuitos bastante complexos, seja em relação ao número de componentes eletrônicos e de suas inter-relações, seja em relação à heterogeneidade funcional presente nas aplicações alvo, previstas para estes sistemas. Para os próximos anos está prevista a possibilidade da inclusão de mais de um bilhão de transistores em uma única pastilha de silício, inaugurando a era da “gigaescala”.

Devido a essa situação, a comunidade científica vem demonstrando preocupação em relação às novas técnicas que se fazem necessárias para a concepção dos “gigacircuitos”. Essas técnicas envolvem o uso de diferentes níveis de abstração na concepção e análise das funcionalidades da aplicação alvo, além de abordagens para explorar o grande espaço de busca, inerente à disponibilidade de um grande número de componentes para a implementação da arquitetura alvo, a qual deve ser otimizada para as restrições de projeto.

As idéias apresentadas nesse trabalho vão de encontro à necessidade por novas técnicas para a concepção de circuitos eletrônicos complexos. Este trabalho procura contribuir para que esta classe de circuitos possa tornar-se realidade em um futuro próximo, avalizando a disponibilidade de informação, de entretenimento e de serviços para a sociedade.

Para tanto, um novo método é proposto, onde um fluxo de projeto considera as ferramentas necessárias para a exploração do espaço de busca em componentes de processamento e de comunicação, visando à sua otimização. As ferramentas seguem os princípios do projeto baseado em plataformas, onde componentes podem ser reutilizadas para aplicações da mesma classe, em diferentes níveis de abstração. Além disso, os princípios da especificação baseada em *interface* são considerados, visando explicitar a especificação de funcionalidades heterogêneas para componentes arquiteturais, bem como permitir a avaliação dinâmica do comportamento destes.

Palavras Chave: Redes-em-Chip; Exploração de espaço de projeto; Projeto baseado em plataformas; Modelo de programação para sistemas-em-chip.

Optimization Method for Architectural Platforms Targeting Heterogeneous Multiprocessor Systems

ABSTRACT

The modern electronic systems concept relies in a multidisciplinary activity, which involves methods and techniques from different human knowledge domains, varying from specification to physical realization of integrated circuits.

The constant evolution of the technology process implies very complex circuits. The complexity is expressed by a huge number of electronic components as well as by heterogeneous functionality. Higher integration levels, allowing the development of circuits comprising more than one billion transistors in a single die, are expected to the near future, inaugurating the “gigascale era” in integrated circuits realization.

Facing this situation, worldwide researches are concerned with the development of new techniques and methods devoted to develop the “gigacircuits”. These techniques imply the observation of different abstraction levels for the design, in order to enable system engineers to deal with the complexity in an affordable time. Moreover, when a big number of components are available to implement optimized architectures, a huge design space to be explored arises. To comply with this scenario, new approaches to find optimized architectural configurations are needed.

The ideas presented in this thesis are compliant with new techniques for modeling and optimization of complex electronic circuits. The tools developed within the scope of this work are expected to contribute to bring these circuits into reality in the near future, setting up the availability of information, entertainment and services to the society.

To deal with this situation, a new method is suggested, proposing a design flow focusing on design automation tools dedicated to turn into the reality the design space exploration for processing and communication components, optimizing them to dedicated applications constraints. The tools comply with the platform-based design approach, where components can be reused on similar behaviors applications, in different abstraction levels. Moreover, the interface-based design principles for architectural components modeling are employed, enabling heterogeneous functionalities modeling and evaluation.

Keywords: Networks-on-Chip; Design space exploration; Platform-based design; Programming model for Sytems-on-Chip.

1 INTRODUÇÃO

A constante evolução da microeletrônica tem trazido cada vez mais desafios para a engenharia de especificação de sistemas. Desde os primeiros microprocessadores, contendo pouco mais de 4 mil transistores, como é o caso do Intel 4004, até os recentemente lançados, com mais de 30.000.000, como por exemplo, Intel Pentium 4, verifica-se um enorme crescimento da indústria de microeletrônica.

Esse avanço vem sendo caracterizado pela Lei de Moore [MOO 75] que já na década de 70, quando os primeiros microprocessadores foram desenvolvidos predizia que a indústria de microeletrônica avançaria a tal ponto de poder dobrar o nível de integração dos componentes em um chip a cada 18 meses.

Os crescentes avanços atingidos pela microeletrônica vêm mantendo a Lei de Moore verdadeira por mais de 3 décadas, estando prevista inclusive a sua continuação para, pelo menos, mais uma década como está previsto no *International Technology Roadmap for Semiconductors* – ITRS [ITR 99]. Nesse relatório, pode-se verificar que para daqui a menos de 10 anos, preve-se pastilhas de silício a um nível tal de integração, contendo a incrível marca de 1 bilhão de componentes eletrônicos (transistores). Isso levando-se em consideração apenas as possibilidades previstas para a atual tecnologia de fabricação.

Certamente que tamanho nível de integração traz muitos desafios para a comunidade envolvida com o projeto e o desenvolvimento desse tipo de sistemas. Com tamanha integração, aplicações atualmente inimagináveis poderão estar disponíveis em poucos anos, como por exemplo, aplicações para telecomunicações, entretenimento, animações de alta definição, reconhecimento de voz, jogos distribuídos, etc.

No entanto, apenas o avanço da tecnologia de microeletrônica e o conseqüente aumento nos níveis de integração não são suficientes para garantir a efetiva implementação para essas aplicações. É necessário que novos métodos e ferramentas de apoio ao projeto tornem-se disponíveis a fim de garantir a projetistas de sistemas eletrônicos o uso eficiente uso da tecnologia disponível. A necessidade por novos métodos e ferramentas torna-se ainda mais urgente se considerado o curto ciclo de vida previsto para as aplicações, o que exige rápida evolução e atualização funcional.

A cada novo avanço da tecnologia, novos produtos podem vir a tornarem-se disponíveis e a uma velocidade cada vez maior. Essa situação, para muitos casos, encurta o ciclo de vida dos produtos, o que diminui o tempo disponível para o projeto, o desenvolvimento e a implementação de novas funcionalidades. Um grande desafio que se faz presente é a habilidade de uma equipe de desenvolvimento de novas aplicações conseguir projetar um sistema e colocá-lo no mercado antes que este se torne obsoleto.

No ITRS, está prevista a existência de uma lacuna entre a disponibilidade da tecnologia - em termos de níveis de integração - e a capacidade humana em utilizar

esses recursos. Verifica-se então, que a produtividade no desenvolvimento de novas aplicações não acompanha a velocidade na qual avança a tecnologia de microeletrônica. É previsto que a produtividade humana crescerá a uma taxa de 21% ao ano, enquanto que os níveis de integração avançarão a uma taxa de 58% ao ano. Por exemplo, para circuitos integrados contendo aproximadamente 1 bilhão de transistores, engenheiros de sistemas eletrônicos posssuirão capacidade de produção estimada em cerca de 10 milhões de transistores por mês.

Com isso, problemas de gerenciamento de grandes equipes se tornam evidentes. Já para os próximos anos está prevista a fabricação de circuitos integrados contendo 130 milhões de transistores, sendo necessários para o seu desenvolvimento (em tempo hábil) cerca de 800 engenheiros. O gerenciamento de equipes muito grandes constitui-se de uma tarefa muito complexa. Isso pode ser justificado pelo fato de que, equipes grandes normalmente são formadas por pessoas com diferentes culturas de desenvolvimento, como no uso de diferentes tipos de ferramentas, por exemplo. Isso complica a integração e o teste das funcionalidades do sistema, uma vez que vários componentes necessitam ser integrados e testados para compor um sistema único. Além disso, componentes podem estar especificados em diferentes níveis de abstração, com granularidades diferentes e ainda, através de diferentes linguagens de desenvolvimento.

Outra questão que surge dentro deste contexto recai sobre o mapeamento das funcionalidades de uma aplicação para as possíveis arquiteturas alvo. Devido à natureza dos sistemas passíveis de serem implementados através das novas tecnologias, estes poderão ser compostos por funções com diferentes características, o que demanda arquiteturas alvo também com características diferenciadas. Por exemplo, é normal que se pense que esses sistemas terão como arquiteturas alvo, sistemas compostos em parte por componentes arquiteturais de processamento e, em parte, por componentes arquiteturais de comunicação. Nesse caso, quanto maior for o número de engenheiros especificando o sistema, maior será a dificuldade para a tarefa de mapeamento das funções especificadas para a arquitetura alvo, devido ao grande número de combinações possíveis para a distribuição das tarefas entre processadores. Ainda, este problema encontra-se fortemente relacionado com a granularidade das funções: se uma aplicação for especificada por vários engenheiros, é natural imaginar-se que caberá a cada um o desenvolvimento de uma pequena parte do sistema. Assim sendo, a posterior integração das funcionalidades da aplicação para a implementação na arquitetura alvo deverá ser realizada através de granularidade comum às especficações.

Por todos os problemas citados acima, conclui-se que para a implementação de sistemas através das novas tecnologias previstas para o futuro próximo surge a necessidade de novos métodos e ferramentas de apoio ao projeto. Essas ferramentas terão que ser capazes de aplicar os conceitos de complexidade, abstração e hierarquia [SUS 2001] na implementação de sistemas eletrônicos. Isto se justifica na natureza complexa dos sistemas a serem implementados.

O avanço da tecnologia de automação de projeto aponta para soluções que visam tornar possível a concepção e o gerenciamento dos futuros sistemas eletrônicos. Dentre as soluções aceitas, pode-se destacar o reuso e a capacidade de produção conjunta entre engenheiros de software com engenheiros de hardware. A primeira alternativa propõe que as funcionalidades (tanto as de software com as de hardware) de uma classe de aplicações possam ser reutilizadas entre várias aplicações. Essa é uma maneira de lidar com a complexidade referente ao grande número de funções presentes nas aplicações, além de permitir a especificação concorrente de funcionalidade em difernetes níveis de abstração. O reuso de arquiteturas de SW permite que novas aplicações possam ser

implementadas através da derivação funcional de outras, ou como a composição de funções previamente implementadas. Essa propriedade conduz a tempos de projeto menores, uma vez que o reuso pode delimitar melhor o espaço de busca, o que vem a ser muito útil para projetos com prazo de entrega muito apertados.

1.1 Caracterização dos Sistemas Eletrônicos Considerados para o Método

Dentro da área de projeto de sistemas eletrônicos, diferentes denominações são utilizadas na tentativa de se definir corretamente os conceitos relativos à sua concepção. Esta seção procura contextualizar algumas denominações utilizadas para caracterizar os sistemas eletrônicos complexos, previstos para o futuro próximo.

A complexidade é inererente a natureza dos futuros sistemas eletrônicos (alvos do método aqui proposto), manifestando-se, por exemplo, no número de componentes eletrônicos (bilhões de transistores), no número de funções, em comportamentos heterogêneos, etc. Igualmente, como é esperada a realização destes sistemas uma única pastilha de silício - devido a alta integração permitida pelas modernas tecnologias de projeto - são comumentemente encontradas na literatura as expressões “Sistemas-em-Chip”, “Sistemas Integrados” ou normalmente, pelo equivalente em inglês, *System-on-Chip* (*SoC*) para os denominar.

No restante do texto, os sistemas eletrônicos alvos das ferramentas propostas no fluxo de projeto do método serão chamados arbitrariamente de sistemas integrados complexos, sistemas dedicados ou simplesmente, *SoCs*. Na literatura, estes sistemas também são normalmente chamados de Sistemas Multiprocessados ou “SoCs Multiprocessados” (*Multi-Processor SoCs*, *MPSoCs*), devido a sua natureza multi-comportamental, ou ainda, Sistemas Embarcados (*Embedded Systems*) por dedicarem-se à realização de aplicações dedicadas e normalmente, embutidas em sistemas móveis. Esta classe de sistemas é dedicada à realização de aplicações específicas, sendo o objetivo desse trabalho, definir uma infraestrutura de tecnologia de projeto, capaz de permitir a concepção, a atualização e a otimização de arquiteturas dedicadas.

As arquiteturas alvo dos SoCs aqui tratados caracterizam-se por implementarem de maneira distribuída as tarefas da aplicação, constituindo um sistema distribuído. As tarefas são implementadas em componentes arquiteturais de processamento, enquanto que as comunicações entre estas, são implementadas em componentes arquiteturais de comunicação.

O conceito de “componentes” implica na determinação das granularidades nas quais as funções que representam são especificadas. A granularidade por sua vez, é função do nível de abstração utilizado na descrição.

A *ortogonalidade* funcional estabelece que componentes podem ser classificados como *software* (SW) e *hardware* (HW); ou processamento e comunicação.

As funcionalidades de uma aplicação podem ser implementadas como tarefas ou componentes de *software*. Componentes de software são aqueles cujas funções são especificadas por operadores *virtuais*. Operadores virtuais representam uma abstração de operações fisicamente realizáveis em alguma arquitetura de hardware. Consequentemente os componentes de SW necessitam serem *mapeados* para componentes arquiteturais, os quais representam operações diretamente realizáveis em hardware. Componentes arquiteturais de processamento são aqueles que especificam funções para *transformação* de dados (através de operações lógico/aritméticas),

enquanto que componentes arquiteturais de comunicação implementam funções de *transferência* de dados.

Componentes arquiteturais de processamento podem ser exemplificados como a parte operativa de processadores e componentes arquiteturais de comunicação como roteadores de redes chaveadas.

O mapeamento pode ser direto, na proporção 1 para 1, como por exemplo, a operação de soma – em software – traduzida para a arquitetura de somadores – em hardware; ou indireto, quando operadores em software são mapeados para mais de um operador em hardware. Por exemplo, a operação de comparação pode ser mapeada para circuitos para subtração e comparação do resultado. Nesse caso, a abstração da operação em software é implementada como um *agregado* das funcionalidades dos operadores arquiteturais de subtração e comparação.

Sintaticamente, componentes de software são representados por símbolos de linguagens de programação e componentes de hardware, por símbolos de linguagens para descrição de hardware, como VHDL.

Componentes arquiteturais podem ser implementados no espaço (como circuitos combinacionais), no tempo (como circuitos seqüenciais) ou uma combinação de ambos [SUS 2001].

Assim, em relação ao processamento funcional, um componente arquitetural pode ser materializado como o hardware da arquitetura de um processador ou de um circuito dedicado. Em relação à comunicação, um componente arquitetural pode ser, por exemplo, o hardware que implementa a política de escalonamento para barramentos.

Os componentes arquiteturais também são chamados de núcleos de processamento/comunicação ou o equivalente em inglês, *cores*.

1.2 Questões sobre o Projeto

À medida que os sistemas tornam-se mais complexos, projetistas de sistemas necessitam trabalhar com um grande número de funcionalidades diferentes, o que demanda um conhecimento comportamental acerca de uma grande quantidade de componentes de software. Por isso, níveis de abstração mais altos devem ser considerados para as ferramentas de análise e síntese, viabilizando o trabalho com quantidades maiores de componentes, facilitando o gerenciamento do projeto devido à maior granularidade das operações.

Outra questão importante refere-se ao reuso das funcionalidades de aplicações. Com os tempos dos projetos cada vez mais enxutos e com o aumento da complexidade dos sistemas, uma equipe de projeto dificilmente teria condições para desenvolver todos os componentes necessários à implementação de todas as funcionalidades do sistema alvo. Para tanto, a equipe terá que *reutilizar* componentes previamente implementados, implicando em reuso de propriedade intelectual (*Intellectual Property*, *IP*). No entanto, muitas vezes os componentes necessitam serem reconfigurados a fim de atender às funcionalidades específicas de uma aplicação. A *Engenharia de Software* promove o reuso e a reconfiguração de componentes de software através do paradigma da *orientação a objetos* (OO) com o uso de técnicas como *encapsulamento*, *herança*, *padrões de projeto* [PAT 2001], etc. Tradicionalmente na *Engenharia de Hardware*, os componentes arquiteturais vem sendo reutilizados em diferentes sistemas e mais recentemente, ferramentas (citadas no capítulo 2) permitem a reconfiguração de componentes arquiteturais para adaptá-los a sistemas com diferentes características.

Como discutido no capítulo 3, o método aqui proposto considera o uso de técnicas e conceitos da orientação a objetos para a elaboração de um modelo de programação, o qual permite a especificação e o reuso de componentes arquiteturais, segundo os princípios do projeto baseado em plataformas [VIN 2002].

Quanto ao reuso de componentes arquiteturais, estes podem ser de dois tipos: *hardcores* e *softcores* [RAJ 2001]. Os hardcores caracterizam-se por possuírem a sua implementação totalmente definida em termos de síntese física (roteamento e posicionamento) não permitido reconfiguração; a sua reutilização implica em as suas funcionalidades estarem em conformidade com as esperadas para a aplicação alvo. Por outro lado, *softcores* podem ser implementados em diferentes tecnologias, sendo mais flexíveis em relação às restrições de projeto e muitas vezes, reconfiguráveis. Questões legais quanto à comercialização de *Propriedade Intelectual* estão fora do escopo desse texto.

Durante o processo de exploração do espaço de projeto, deve ter-se o cuidado para manter-se a continuidade do modelo, o que significa manter a coerência funcional da especificação das funções da aplicação entre os diferentes níveis de abstração. Assim, as ferramentas de apoio ao projeto devem permitir ao projetista verificar como uma alteração comportamental em um determinado nível de abstração é refletida nos demais. À medida que níveis de abstração mais baixos são levados em consideração durante a exploração do espaço de projeto e a conseqüente avaliação de componentes arquiteturais, mais detalhes acerca da arquitetura alvo são considerados. Nesse ponto, deve-se procurar auxiliar o projetista de sistemas a encontrar as arquiteturas alvo que melhor conseguirem implementar a aplicação em relação às restrições de projeto, dentro do nível de detalhamento necessário. Restrições de projeto típicas são área, potência, desempenho.

1.3 Objetivos e Contribuições Esperadas

Tendo em vista o esforço, as idéias e as tendências observadas pela comunidade científica da área de projeto e desenvolvimento de sistemas eletrônicos, podem-se elencar algumas idéias que servem de motivação para o avanço dessa importante área de pesquisa, que visa, dentre outras coisas, consolidar e democratizar a sociedade da informação. Essas idéias são ilustradas nessa seção, e incorporam os objetivos previstos para o método aqui apresentado.

Outra questão que se faz importante neste momento, diz respeito aos trabalhos já realizados pelo grupo de pesquisa local, uma vez que os objetivos deste trabalho alicerçam-se na consolidação e no aperfeiçoamento das ferramentas já desenvolvidas, no sentido do avanço das tecnologias de projeto.

Nesse sentido, podem ser considerados como objetivos do trabalho:

- Estender os métodos atuais de projeto de sistemas, para um método aplicável ao projeto e a implementação de sistemas complexos;
- Propor um fluxo de projeto que vise à exploração de comportamentos em componentes arquiteturais, com fins à otimização frente a restrições de projeto; e
- Sugerir e desenvolver as ferramentas de apoio ao projeto, necessárias à efetivação do fluxo de projeto proposto pelo método.

De maneira geral, pode-se considerar o processo de otimização como sendo composto por um conjunto de tarefas, as quais visam atender a Qualidade-de-Serviço, (*Quality-of-Service, QoS*) requerida pela aplicação. A Qualidade-de-Serviço determina

quais as restrições de projeto que devem ser atendidas para aplicações alvo, bem como a correta execução de suas operações.

Dentro de sua natureza, o método deverá compreender as propriedades que pretendem facilitar o complexo problema da concepção de sistemas heterogêneos, como por exemplo, o reuso de componentes arquiteturais. Neste texto, sistemas ou aplicações heterogêneas são entendidas como aquelas que se são especificadas por mais de um modelo de computação.

Modelos de computação (*Models of Computation – MoCs*) [LEE 98] são formalismos matemáticos que permitem a modelagem de comportamentos heterogêneos e portanto, são adequados para a modelagem de aplicações heterogênea. Exemplos de MoCs são *rendezvous* [LEE 2001] para comunicação síncrona e *dataflow* [LEE 95] para comunicação assíncrona.

O que se pretende então é criar as condições para que sistemas possam ser completamente especificados em diferentes níveis de abstração, através de componentes de software e de hardware, e que estes possam ser otimizados às restrições de projeto de aplicações dedicadas.

A principal contribuição desse trabalho para o estado-da-arte em projeto de sistemas eletrônicos dedicados, refere-se à proposta de um *método para a otimização arquitetural* de plataformas de processamento e comunicação. O método deve compreender:

1. A especificação dos componentes arquiteturais de plataformas para processamento e comunicação em diferentes níveis de abstração;

2. A definição da semântica das ferramentas de apoio ao projeto, necessárias à otimização arquitetural de plataformas de processamento e comunicação;

3. Um fluxo de projeto que demonstre as relações entre as ferramentas, bem como estes competem na busca por soluções otimizadas; e

4. As regras necessárias para que componentes arquiteturais possam ser configurados automaticamente por ferramentas de apoio ao projeto.

A otimização arquitetural permite a avaliação do compromisso processamento/comunicação em relação às restrições de projeto de aplicações dedicadas, através da exploração de diversas soluções arquiteturais em componentes de processamento e comunicação. O compromisso processamento/comunicação, o qual visa à implementação de aplicações dedicadas em plataformas arquiteturais otimizadas, é obtido através da distribuição de tarefas da aplicação em um conjunto de processadores. Cada processador por sua vez, pode ser configurado para executar uma ou mais tarefas de maneira otimizada. Como conseqüência do uso de um conjunto de processadores, arquiteturas de comunicação também necessitam ser avaliadas.

Além disso, sistemas completos podem ser construídos, os quais provêm suporte arquitetural para a realização em hardware das funcionalidades de aplicações dedicadas complexas. Entende-se por aplicações dedicadas complexas, as aplicações que requerem alto poder computacional e ao mesmo tempo, possuem sérias restrições de consumo de energia. As principais características para este tipo de aplicação podem ser:

- São compostas por várias tarefas;
- As tarefas podem executar concorrentemente;

- A comunicação entre as tarefas ocorre sob diferentes protocolos de comunicação; e

- As tarefas podem apresentar comportamento heterogêneo, o que demanda suporte arquitetural específico para a sua realização eficiente na arquitetura alvo.

A complexidade se manifesta pela grande quantidade de tarefas, bem como pelo comportamento heterogêneo.

Como exemplos de aplicações alvos deste trabalho podem-se citar, aplicações multimídia e de entretenimento, como jogos, TVs de alta definição, navegação em automóveis, set-top boxes, redes de alta velocidade; aplicações de segurança e saúde, como monitoramento por satélite, controle e análise das atividades do corpo humano, etc.

Espera-se a realização dessas aplicações alvo em um futuro próximo, provavelmente em uma única pastilha de silício, compondo um Sistema-em-Chip (*System-on-Chip, SoC*).

Para a realização de aplicações dedicadas complexas, esperam-se plataformas arquiteturais alvo compostas por até *n* elementos de processamento, sendo *n*, o número de tarefas da aplicação. Conseqüentemente, a arquitetura alvo será um sistema multiprocessado heterogêneo (*MPSoC* → *Multiprocessor System-on-Chip*), uma vez que os elementos de processamento devem prover suporte arquitetural para a efetivação – dentro das restrições de projeto – das funcionalidades heterogêneas de aplicações alvo distribuídas. Devido ao (possível) grande número de elementos de processamento, bem como da variação arquitetural de cada um – em termos de componentes internos – esta classe de arquitetura alvo também é considerada dentro do contexto desse trabalho, como sistema complexo.

Uma vez que sistemas multiprocessados são esperados, arquiteturas de comunicação devem ser consideradas para a implementação do comportamento de comunicação da aplicação. Em sistemas compostos por um grande número de elementos de processamento, redes chaveadas devem ser consideradas.

Tendo por base as ferramentas de apoio ao projeto propostas no método, as aplicações e arquiteturas alvo, podem também ser consideraras como contribuições do trabalho, as análises e otimizações de:

- Topologias e componentes internos de roteadores de redes chaveadas;

- Roteadores heterogêneos para redes chaveadas;

- Arquiteturas e componentes internos de processadores; e

- Unidades e instruções dedicadas em processadores.

Além disso, como suporte à realização das análises e otimizações acima elencadas, também são contribuições do trabalho, a implementação da síntese/particionamento das tarefas de aplicações alvo em elementos de processamento e o mapeamento de processadores em terminais locais de redes chaveadas.

O texto segue como uma contextualização acerca de como modernos sistemas eletrônicos podem ser concebidos e caracterizados, bem como dos recursos necessários à que estes possam ser adequados às necessidades para as quais são projetados. Em seguida, é analisado um possível fluxo de projeto adotado na concepção de sistemas eletrônicos e das ferramentas de apoio ao projeto que o implementam.

Essa tarefa é realizada com a finalidade de se estabelecer os esforços atuais na implementação de SoCs, bem como encontrar a base que leve a um avanço no estado-da-arte referente ao projeto de sistemas eletrônicos complexos. O citado avanço no estado-da-arte será então efetivado através das contribuições citadas neste capitulo. Como originalidade principal desse trabalho pode-se destacar a abordagem – em termos de ferramentas de apoio ao projeto e modelagem de componentes arquiteturais – proporcionada pelo método para exploração/otimização concorrente e ortogonal de componentes de comunicação e processamento. A otimização pode ser executada visando a adequação de cada uma dessas classes comportamentais para as restrições de projeto de aplicações dedicadas. Nessa direção, aplicações e arquiteturas podem ser especificadas separadamente, onde funcionalidades de processamento e comunicação são exploradas conjuntamente na busca por sistemas otimizados. Como resultado do processo de otimização proposto pelo método, a realização de tarefas em processadores é considerada em função da arquitetura de comunicação empregada e vice-versa.

Tendo por base o fluxo de projeto analisado, um fluxo de projeto específico é derivado, o qual visa principalmente à otimização arquitetural, objetivo principal desse trabalho.

2 O PROCESSO DA CONCEPÇÃO DE SISTEMAS ELETRÔNICOS COMPLEXOS

Neste capítulo são discutidos artifícios que podem ser utilizados na concepção de sistemas e que são necessários para que o ser humano possa melhor compreendê-los e otimizá-los às suas necessidades. Uma vez que encontradas as soluções para a compreensão e manipulação dos sistemas, torna-se necessário que métodos definam um fluxo de projeto, no qual ferramentas possam ser utilizadas a fim de auxiliar no projeto desses sistemas. Nessa direção, este capítulo discute também a adoção de um fluxo de projeto bem como as ferramentas necessárias à sua implementação. O capítulo termina com uma revisão das ferramentas de apoio ao projeto, atualmente encontradas na literatura.

Mesmo que o ser humano possua teoricamente uma capacidade infinita para o armazenamento e manipulação de informações, este também encontra restrições quanto ao tratamento simultâneo de um número excessivo destas. Mesmo que o tratamento de um grande número de informações não tenha que ser feito simultaneamente, o ser humano encontra dificuldades em lidar com as intricadas relações que podem ocorrer entre um grande número de informações, o que pode demandar um grande tempo para compreendê-las, além de estar sujeito a erros. Estas situações podem ser definidas como complexidade, no sentido da dificuldade de se tratá-las em tempo hábil ou útil. Dessa forma, torna-se necessário que se criem mecanismos ou artifícios que permitam ao ser humano tratar da complexidade.

Como citado na introdução, projeto dos atuais e futuros sistemas eletrônicos constitui-se de uma tarefa complexa para o ser humano, uma vez que estes sistemas possuem diversos componentes relacionadas e com comportamentos distintos. Assim, torna-se necessário que novos conceitos sejam criados a fim de que permitir à capacidade humana a compreensão necessária à concepção dos sistemas previstos.

Um esforço nesse sentido pode ser encontrado em [SUS 2001] onde é definido o método dos refinamentos sucessivos bem como os conceitos de abstração, hierarquia e complexidade, necessários à sua implementação. Nesse sentido cabe aqui citá-lo, uma vez que este método será a base para os paradigmas, fluxo de projeto e ferramentas que serão aqui discutidas para a concepção de sistemas integrados.

2.1 O Método dos Refinamentos Sucessivos

O método dos refinamentos sucessivos consiste em apresentar o objeto complexo em diversos níveis de descrição, partindo de uma descrição mais abstrata até chegar ao nível de implementação. Cada descrição considera apenas uma parte dos predicados dos elementos manipulados: os que são significativos no nível em questão. Uma descrição de alto nível abstrai detalhes de implementação. A dinâmica do método consiste em,

uma vez definidos os diversos níveis, estabelecer mecanismos para passar de uma descrição mais abstrata para uma mais detalhada.

- **Complexidade:** O termo "complexidade" é de uso corrente na informática, referindo-se à quantidade de recurso necessário para a realização de uma tarefa ou algoritmo. Os recursos constituem-se basicamente pelo número de passos do algoritmo e pela quantidade de memória. Para os sistemas digitais em geral e para os circuitos integrados em particular, esses conceitos são muito eficazes para o discernimento da viabilidade de alternativas ou mesmo para a escolha da mais interessante.

Para o domínio dos circuitos integrados convém separar os dois aspectos da complexidade distinguindo as duas componentes: o conjunto do número de elementos e o conjunto de suas relações. Um circuito será complexo quando um ou dois conjuntos forem (relativamente) grandes. Sob o aspecto prático é ainda mais importante poder classificar os circuitos quanto á sua complexidade do que obter o seu valor absoluto. Esse parâmetro permitirá, por exemplo, decidir qual dentre duas opções de implementação é mais complexa.

A complexidade do relacionamento entre os elementos de um sistema digital aparece materializada nos caminhos de interconexão e nas funções de transformações das informações veiculadas no interior do sistema. Uma forma de normalizar a medida da complexidade é a de referir-se a uma máquina padronizada. Se fosse adotada como referência um autômato finito, entrariam no cômputo da complexidade: o número de estados, a função de cálculo do próximo estado, o alfabeto de entrada e de saída.

A conseqüência do exposto acima é de fácil aplicação. Uma tarefa antes ou impossível ou tratada de forma intuitiva pode ser agora quantificada. A decisão entre duas opções de implementação de um sistema digital pode agora ser tomada automaticamente (pelo menos quanto a esse critério). A comparação quanto à complexidade de dois microprocessadores, por exemplo, pode ser feita automaticamente. Não é necessária esta metodologia para decidir se uma memória de 10 milhões de bits é mais ou menos complexa do que outra de 5 milhões. Fica, entretanto, claro e identificado porque duas máquinas seqüenciais com 20 *flip-flops* para memorizar o estado atual (um contador e o controle de um microprocessador) poderiam ser completamente diferentes quanto à complexidade.

O conceito de complexidade como exposto acima, subentende dois outros conceitos complementares: abstração e hierarquia. Na realidade estes três conceitos não são independentes e a análise poderia ter principiado por qualquer um deles. De fato, só podemos pensar em abstrair quando o elemento é complexo e, por outro lado, a abstração cria uma árvore de conceitos organizados hierarquicamente.

- **Hierarquia**: é a ordem de precedência entre os elementos de um conjunto, segundo alguns critérios. Desde o instante em que uma função é concebida, o conceito de hierarquia está presente, uma vez que uma função propõe relações entre elementos. Os critérios de hierarquização devem ser explicitados para evitar ambigüidade. Sendo estendida a todas as formas de expressão do circuito integrado, a hierarquia visa permitir identificar a posição de um elemento em qualquer descrição, a partir do conhecimento de sua posição em uma delas. Através da hieraquia pode-se indentificar um registrador pertence à execução de uma instrução de um microprocessador, que por sua vez tem uma semântiva associada a um conjunto de instruções e assim por diante.

- **Abstração**: é o ato de desprezar um subconjunto de predicados de um objeto. A utilidade da abstração vem de sua capacidade de reduzir a quantidade de informação

relativa ao objeto, tornando-o mais facilmente tratável. Na concepção de circuitos integrados, por exemplo, pode ser não significativo o seu peso, sua cor, sua espessura, sua rigidez mecânica, etc. Da mesma forma, ainda na área da concepção de circuitos integrados, a descrição a nível de algoritmo pode desprezar a capacidade de corrente de um determinado transistor. Em cada situação tem-se o objetivo de somente tratar os predicados do objetivo que são relevantes para o fim almejado. Isso é equivalente a criar um modelo do objeto. Como todo modelo é imperfeito, uma tarefa de validação é necessária para determinar a confiabilidade dos resultados obtidos com as informações parciais manipuladas. É necessário que em cada nível de abstração, seja estabelecido o que a alteração em algum ponto do comportamento causa nos outros níveis de abstração sob os quais uma determinada aplicação é considerada. Isto vale tanto para a análise manual da aplicação, quanto automática.

O caminho inverso da abstração é a síntese, ou implementação. Essas ações agregam detalhes a uma descrição, aumentando a quantidade de informação associada. A informação agregada pelo implementador pode vir dele mesmo (é a perícia do artífice a partir de sua criatividade, arte ou treinamento) ou do contexto (a partir de um conjunto de regras ou restrições).

2.1.1 Níveis de Descrição

A técnica dos refinamentos sucessivos será materializada no caso dos sistemas digitais, por uma estrutura que contém uma cadeia de linguagens de descrição, as quais representam diferentes níveis de abstração.

Um sistema digital é uma rede de componentes que pode ser descrita de várias formas: como um algoritmo, como uma rede de circuitos integrados, como uma rede de transistores, como um conjunto de máscaras, etc. Cada uma destas descrições convenientemente manipuladas pode gerar o sistema digital físico que deverá ser equivalente a todas as descrições. É evidente que quanto mais abstrata uma descrição maior a quantidade de informação a ser agregada.

Os níveis em que pode ser descrito um sistema digital são criados artificialmente, mesmo que, para fins práticos, tenha-se o interesse de fazê-los corresponder a fases do projeto pelas quais passaria naturalmente um projetista em função de sua cultura ou treinamento. Os níveis não são pois estanques nem fixos, mas variam com os métodos de trabalho, com a tecnologia, com as ferramentas disponíveis, etc.

Tendo em mente que a síntese automática faria a passagem de um nível para outro até se chegar ao objetivo físico, é de todo interesse definir os níveis de forma e adaptarem-se a esta tarefa. A passagem de nível é natural se em cada nível forem utilizados conceitos pré-definidos. Por isso se diz que um nível é definido por uma linguagem. A partir do momento em que se define uma linguagem de descrição, definiu-se um nível de descrição.

Um circuito está num determinado nível quando está descrito através de um conjunto de primitivas pertencentes à linguagem que define o dito nível. Um nível pode também ser definido como um conjunto coerente de primitivas que possibilitam a descrição de um objeto (em particular, de um circuito integrado).

Uma máquina que compreende as primitivas de um determinado nível é chamada de interpretador. Assim, dada uma descrição, se houver um interpretador para a linguagem utilizada para descrever, ele pode ser utilizado para executar o que a descrição representa, interpretando o circuito. O conjunto de primitivas define uma

máquina abstrata. O próprio interpretador pode ser descrito utilizando as primitivas de uma outra linguagem, requerendo assim mais um nível de interpretação.

A cadeia de interpretação deve terminar em determinado momento para deixar de ser máquina virtual e passar a ser uma máquina real. Isto pode acontecer em uma máquina hospedeira (quando se estaria simulando o circuito) ou uma máquina específica como um circuito integrado. Cada nível deve ser formalmente definido - um nome não basta pois cada qual o entende à sua maneira. Define-se formalmente um nível definindo-se uma linguagem e a semântica das suas operações. Consequentemente, os níveis existem posteriormente às linguagens que lhes dão origem.

Uma linguagem pode ser sintaticamente definida através de diversos formalismos, tais como gramáticas (combinação de símbolos, normalmente textuais), diagramas de grafos, autômatos, equações matemáticas, bibliotecas de classes e seus relacionamentos (ou API's, *Application Programming Interfaces*). O interprador de cada linguagem atribui à sintaxe, a semântica correspondente.

A figura 2.1 exemplifica alguns níveis que são comumente utilizados atualmente pelas ferramentas de apoio ao projeto.

A especificação é a que corresponde ao maior nível de abstração, sendo implementada através de linguagens de programação tais como C++ ou Java. As linguagens ditas intermediárias são utilizadas para que as ferramentas de síntese possam mapear as construções das linguagens de programação para os componentes arquiteturais. Dessa forma podem associar o fluxo de dados que existe em cada operação entre os operandos fonte, operador e operandos destino, a componentes arquiteturais, uma vez que operandos fonte e destino e os operadores são componentes arquiteturais. O nível de transferência entre registradores especifica os componentes arquiteturais em si, com informações temporais, ou seja, a nível de relógio. Como exemplo de linguagens tipicamente utilizadas para a descrição de componentes arquiteturais, pode-se citar VHDL ou Verilog. O nível lógico especifica uma rede de portas lógicas a serem otimizadas e finalmente o nível físico especifica as relações entre os transistores.

Dentro do contexto do método e ferramentas propostas nesse trabalho, serão considerados níveis de descrição que vão desde o nível de especificação até o nível de transferência de sinais. O foco do trabalho está centrado na otimização de componentes arquiteturais especificados em um desses níveis, devido a grande quantidade de componentes arquiteturais presentes nos sistemas atuais. Isto demanda diferentes níveis de abstração para que estes componentes possam ser analisados, devido ao grande espaço de projeto a ser explorado.

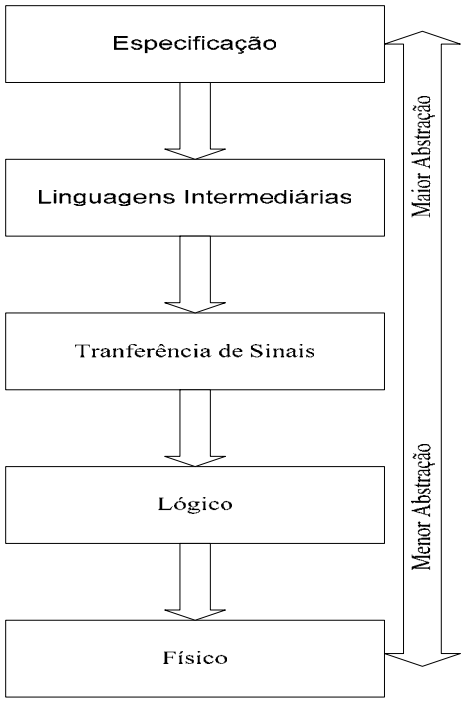


Figura 2.1: Níveis de Abstração na Concepção de Sistemas Eletrônicos

Os componentes arquiteturais podem ser classificados como:

- Processamento;
- Comunicação; e
- Armazenamento.

Estas três classes de componentes possuem comportamentos *ortogonais*. Por ortogonais, entende-se que esses componentes implementam um comportamento completo, ou seja, sua semântica pode ser especificada por um modelo de computação.

O fato de as três classes de componentes arquiteturais possuírem comportamentos ortogonais viabiliza a criação de ferramentas dedicas à análise separada – e concorrente - de cada uma dessas classes.

No entanto, deve-se atentar que modificações arquiteturais realizadas sobre uma das classes de componentes, afeta as demais. Como exemplo, pode-se verificar que, ao invés de se utilizar um processador mais rápido para a execução de uma tarefa, podem ser utilizados dois processadores mais lentos, desde que estejam conectados por uma arquitetura de comunicação suficientemente rápida. Neste caso, os operandos estarão localizados em uma organização de memória distribuída, entre os processadores.

A análise separada dessas três classes de componentes arquiteturais é denominada na literatura de separação de conceitos [KEU 2000].

A semântica das três classes de componentes arquiteturais corresponde aos três tipos de operações necessárias à execução de algoritmos: o *processamento*, que implementa as operações computacionais ou de transformação de dados; a *comunicação*, responsável pela transferência de dados entre os componentes de processamento e o *armazenamento*, implementado por memórias e registradores para as variáveis. Dessa forma, os componentes arquiteturais implementam o fluxo de dados

presente nas operações dos algoritmos: *operandos fonte*, *operadores* e *operando destino*.

Outra maneira interessante de se observar uma aplicação, ocorre sob o ponto de vista dos componentes de software. Devido ao grande número de funcionalidades encontradas em aplicações complexas, estas são descritas através de diferentes níveis de abstração. Devido à "distância" entre a especificação e a arquitetura alvo, é necessário que ferramentas de apoio ao projeto possam investigar o comportamento das aplicações em diferentes níveis de abstração.

O processo de mapear uma descrição de componentes de software em outra é chamado de *síntese de software*, em contrapartida à síntese de hardware.

A figura 2.2 mostra um exemplo de alguns níveis de abstração que podem ser utilizados quando da análise de uma aplicação através de componentes de software.

O nível mais alto de abstração corresponde à especificação através de uma linguagem de programação, como citado anteriormente.

O segundo nível corresponde a descrições cuja semântica corresponda ao comportamento dos componentes de comunicação e de processamento. Normalmente, descrições nesse nível de abstração são implementadas através de *metalinguagens* [PAS 2002]. Metalinguagens são linguagens não auto contidas, mas implementadas através de outras linguagens, normalmente através de biblioteca de classes – ou APIs. Nesse nível de abstração, APIs são utilizadas para a implementação de comportamentos específicos, como por exemplo, sincronismo entre componentes de comunicação. O uso de biblioteca de classes é interessante para um projetista de sistemas poder explicitar para sua aplicação determinados modelos de computação, que do contrário, devem ser descobertos por ferramentas dedicadas, como *parsers* para gramáticas de linguagens de programação, ou interpretadores para grafos.

O fato de o projetista de uma aplicação ter a consciência dos modelos de computação utilizados na sua concepção, torna o sistema mais fácil de ser verificado automaticamente, pois ferramentas podem ser criadas para os interpretarem. Além disso, o projetista é consciente das implicações decorrentes de uma atualização funcional da aplicação, como a adição de novos componentes ou de novos relacionamentos entre estes.

Outro ponto positivo para o uso de biblioteca de classes é a facilidade com que esta pode ser expandida para a implementação de novos comportamentos. Isto é especialmente interessante no contexto de sistemas dedicados, devido à heterogeneidade funcional. Pode-se citar como um exemplo de ambiente que implementa esse conceito, o Ocapi-XL [VAN 2001].

O terceiro nível corresponde ao nível das primitivas de Sistemas Operacionais. Estas são interessantes no sentido de que abstraem o comportamento de muitas operações que muitas vezes são realizáveis diretamente no hardware ou que são utilizadas por muitas aplicações diferentes. Como exemplos, podem-se citar o acesso a um arquivo em disco, a comunicação entre processos, bem como o seu escalonamento. Observa-se, no entanto, que o próprio Sistema Operacional, constitui-se de outra aplicação, portanto, passível de ser avaliada em conjunto com a aplicação alvo. Um SO pode determinar o grau de otimização de uma aplicação alvo em relação a certas restrições de projeto. Por exemplo, a política de escalonamento empregada pelo SO pode determinar a capacidade de um processador executar ou não, certo número de processos.

No quarto nível são implementadas funcionalidades que interagem diretamente com o hardware, sendo chamados também de software dependente do hardware (ou *Hardware-dependent Software, HdS*). Como exemplo, pode-se citar os *drivers* para dispositivos específicos, os quais se comunicam com componentes arquiteturais no nível RT. Uma linguagem cuja semântica corresponda componentes de software do tipo HdS é portanto única para o hardware a qual foi projetada, e como conseqüência, mais difícil de ser reconfigurada. Finalmente, o último nível corresponde ao nível de transferência entre registradores.

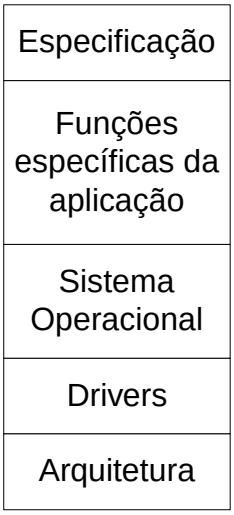


Figura 2.2: Níveis de Abstração para Componentes de Software

As relações entre os componentes de software e de hardware realizadas em diferentes níveis de abstração podem ser interpretadas através de *domínios de especificação*, como mostrado na figura 2.3. Domínios representam o comportamento de aplicações, considerados sob diferentes níveis de abstração. Na figura 2.3 os domínios estão representados como “frameworks”, uma vez que estes representam a especificação de todo o comportamento de uma aplicação. Isto também permite a criação de ferramentas específicas para cada domínio.

Em uma visão *top-down*, a relação entre ferramentas de diferentes domínios, ocorre através da *tradução* de comportamentos especificados em um nível de abstração mais alto, em outro nível mais baixo, o que pode ser realizado através do método de refinamentos sucessivos. A seguir, os componentes de cada domínio são classificados segundo os níveis de abstração considerados para o método, como comentado na seção 3.5.

No domínio de *aplicação*, os componentes das aplicações podem ser especificados no nível de sistema e de mensagens. Nesse domínio os componentes de software correspondem à funcionalidade da aplicação, enquanto que os componentes de hardware, ao conjunto de instruções de processadores ou a definição da topologia e operações funcionais das arquiteturas de comunicação.

O domínio do *sistema operacional* promove a separação das especificações puramente funcionais com as especificações funcionais e temporais. No domínio de aplicação o comportamento é identificado através da ordem dos relacionamentos entre os componentes. Já no domínio arquitetural, o comportamento é definido pela ordem e pelo tempo decorrido nas relações.

Componentes de software nesse domínio correspondem à implementação das funções de sistemas operacionais, especificados no nível de mensagens; enquanto que os componentes arquiteturais correspondem às partes operativas e de controle de processadores e aos componentes internos de roteadores em redes chaveadas. Os componentes arquiteturais são especificados nos níveis de mensagens ou transações. As funções dos sistemas operacionais que consideram a passagem do tempo executam sobre os componentes arquiteturais especificados no nível de transações; do contrário, executam sobre as especificações no nível de mensagens.

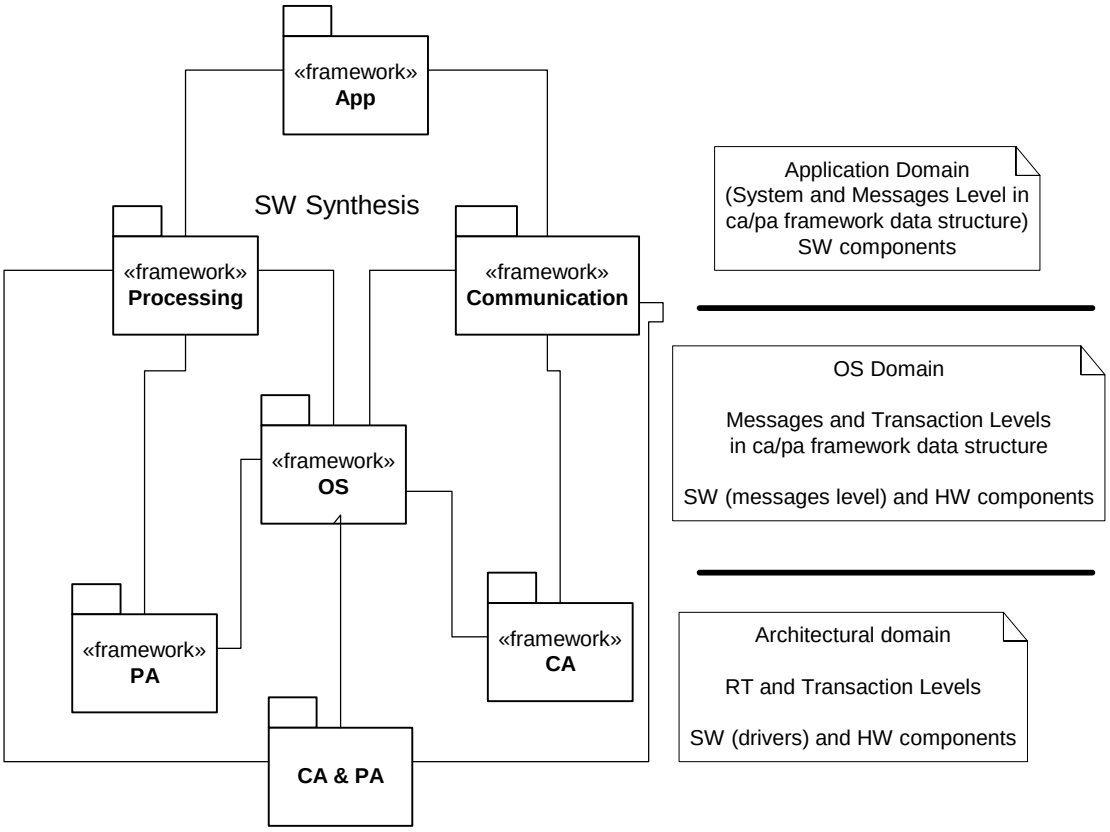


Figura 2.3: Domínios de especificação para aplicações dedicadas

Finalmente, no domínio *arquitetural* são especificadas as plataformas arquiteturais de comunicação (*Communication Architecture – CA*), de processamento (*Processing Architecture – PA*) ou ambas (*CA&PA*), nos níves de abstração de transações e transferência entre registradores. Componentes de software nesse nível, implementam funcionalidades as quais requerem o conhecimento do comportamento dos componentes arquiteturais desse domínio; software dependente de hardware.

2.2 Fluxo de Projeto para Sistemas Complexos

Devido a sua alta complexidade como já foi dito, para que sejam passíveis de compreensão por parte de seus projetistas, os futuros sistemas eletrônicos necessitam serem avaliados sob diferentes níveis de abstração. Nesse sentido, agumas idéias foram propostas no capítulo anterior, onde foram definidos alguns níveis de abstração, tanto em relação às linguagens utilizadas pelas ferramentas de apoio ao projeto, quanto pelo ponto de vista dos componentes de software.

Como citado na seção 2.1.1, o método a ser proposto nesse trabalho refere-se à otimização de aplicações específicas para componentes arquiteturais. Isto se deve ao

fato de que, como pode ser constatato na mesma seção, o processo de refinamentos sucessivos determina que cada linguagem especifique o sistema sob o seu nível de abstração e portanto, permite que os sistemas sejam analisados sob esse nível.

O nível de transferência entre registradores é do nível de abstração mais baixo a ser considerado para o escopo desse trabalho. Isso se deve à complexidade dos sistemas em questão, uma vez que níveis de abstração inferiores levariam a uma complexidade em relação ao espaço de projeto a ser considerado. Além disso, a partir desse nível de abstração ferramentas de apoio ao projeto referentes às sínteses lógica e física podem ser desenvolvidas independentemente das aqui tratadas, pois, para encontrarem um circuito integrado otimizado, basta que recebam como entrada, a descrição de uma arquitetura otimizada.

A figura 2.4 mostra um fluxo de projeto onde são especificadas as linguagens, ferramentas e as suas relações no processo de otimizar uma arquitetura alvo para uma aplicação específica.

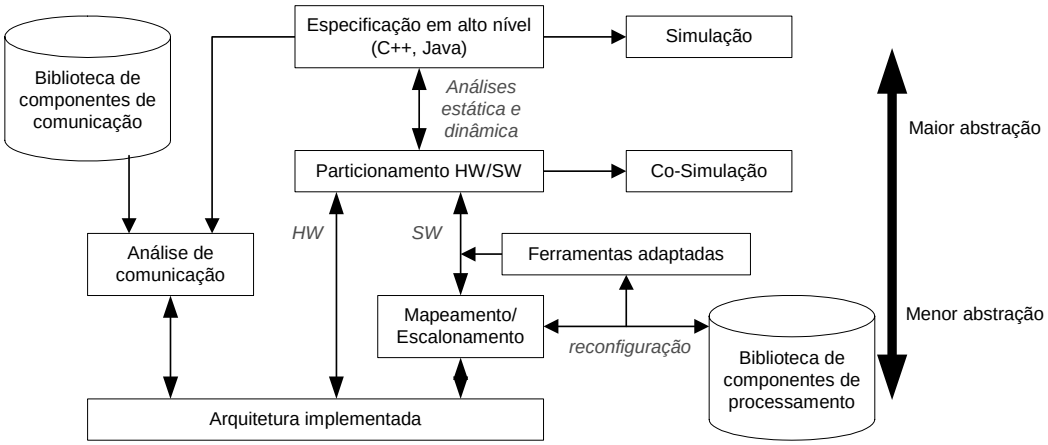


Figura 2.4: Fluxo de Projeto para Sistemas Eletrônicos Complexos

As setas bidirecionais da figura 2.4 indicam que o projeto de um sistema pode iniciar-se a partir do maior ou do menor nível de abstração. Isso é refletido nas abordagens atualmente existentes para a concepção de sistemas eletrônicos. A explicitação das ferramentas do fluxo de projeto e dos seus relacionamentos é realizada sob o ponto de vista do maior para o menor nível de abstração.

Primeiramente, a aplicação é especificada em alto nível através de linguagens como C++ ou Java. Nesse ponto a aplicação completa pode ser simulada funcionalmente - onde o tempo não é tão importante quanto a ordem de execução das tarefas da aplicação. Em seguida, são realizadas, sobre a especificação, análises estáticas e dinâmicas a fim de se descobrirem as características de cada função especificada. Características essas, que podem incluir o modelo de computação, tempo de computação, trocas de mensagens entre as funções, etc.

Em seguida, com base nas informações obtidas das análises, é realizado o particionamento HW/SW, para que as funções com maiores exigências, em relação às restrições de projeto, (como desempenho, por exemplo) possam ser implementadas por funções de HW dedicadas, mapeadas em FPGA ou circuito integrado dedicado (ASIC – *Application Specific Integrated Circuit*). Em termos de verificação pode-se efetuar uma co-simulação funcional e temporal do sistema, como realizado em [GHA 2002]. Em paralelo, também com base nos resultados das análises, algoritmos para análise dos padrões de comunicação da aplicação tentam encontrar a arquitetura de comunicação

que melhor corresponda aos requisitos do sistema. Modelos de diversas arquiteturas de comunicação devem estar presentes em uma biblioteca, para serem analisados.

As funções a serem implementadas como SW servem de entrada para algoritmos de mapeamento e escalonamento, que as implementam em componentes do tipo GPP (*General Purpose Processor*) ou ASIP (*Application Specific Instruction-set Processor*). Quando processadores ASIP estiverem presentes na biblioteca de componentes de processamento, estes podem ter os seus componentes arquiteturais otimizados, através de reconfiguração. Nesse caso, esses processadores são salvos em uma biblioteca para serem reutilizados em futuras alterações funcionais da aplicação ou até, em outras aplicações. Outra situação interessante ocorre quando mais de uma função – ou tarefa - da aplicação é mapeada para o mesmo processador. Neste caso, deve-se adotar uma política de escalonamento. Essa política é determinada por ferramentas dedicas à geração de Sistemas Operacionais dedicados. Por serem dedicados, esses sistemas podem também associar às suas funcionalidades, a capacidade de lidar com tarefas específicas de sistemas distribuídos, como por exemplo, gerenciamento de recursos na rede, trocas de mensagens e escalonamento distribuído de processos.

Finalmente, a arquitetura final é instanciada como componentes arquiteturais de processamento e comunicação otimizados, formando um sistema multiprocessado e distribuído. O instanciamento pode se dar pela geração de uma descrição ao nível de transferência de registradores (VHDL, Verilog, SystemC [GRO 2002]), para posterior síntese lógica e física e de código em linguagem montadora, para as funções em SW. Ainda, as ferramentas de desenvolvimento devem ser adaptadas para manter a compatibilidade de software com a arquitetura do ASIP reconfigurada. Isso é o equivalente a dizer que a linguagem que descreve o conjunto de instruções do processador do tipo ASIP deve refletir o “novo” comportamento gerado nos componentes arquiteturais desse processador. Como a nova configuração deve poder ser analisada em todos os níveis de abstração considerados, as linguagens acima da que descreve o conjunto de instruções otimizado, também devem ser notificadas. Por exemplo, se uma nova instrução for implementada, o compilador do processador deverá incorporá-la à sua sintaxe.

É interessante notar que o projetista de sistemas poderá analisar a aplicação a ser mapeada para a arquitetura alvo sob diversos níveis de abstração, tanto para as funcionalidades de processamento, quanto de comunicação. Ao nível de sistema, as operações de comunicação poderão ser observadas durante a simulação funcional através de primitivas de alto nível, como *send()* e *receive()*, implementadas nas linguagens utilizadas para especificação. Ainda no nível de sistema, as operações de processamento podem ser observadas através das funcionalidades descritas para a aplicação. Durante o particionamento e a co-simulação, as operações de comunicação podem ser observadas através de descrições das arquiteturas de comunicação contendo, por exemplo, os protocolos e políticas de roteamento e chaveamento implementadas. As operações de processamento nesse nível de abstração podem ser descritas através de grafos de fluxo de dados e controle executando sobre os componentes arquiteturais do tipo processadores. Finalmente, quando a aplicação é implementada através do instanciamento da arquitetura alvo, tantos os processadores, quanto as arquiteturas de comunicação, são descritas e analisadas ao nível de transferência de sinais.

2.3 Abordagens na Concepção de Sistemas Eletrônicos

Segundo o fluxo de projeto apresentado na figura 2.4, três principais abordagens para a concepção de sistemas foram criadas:

- *Top-Down* ou Síntese a nível de Sistema;
- *Bottom-up* ou Projeto baseado em Componentes; e
- Projeto baseado em *Plataformas*.

Nos dias atuais não existe um consenso acerca da definição formal dessas abordagens, no entanto verifica-se uma tendência no sentido de que sejam cada vez mais adotados pelas ferramentas de apoio ao projeto, seguindo essa taxonomia.

Na abordagem “de cima para baixo” (mais comumente chamada pelo seu equivalente em inglês, *top-down*) o fluxo de projeto da figura 2.4 deve ser analisado a partir do maior nível de abstração. Nessa abordagem, não há uma arquitetura pré-definida, sendo que as ferramentas têm total liberdade para a escolha dos componentes arquiteturais que irão formar a arquitetura alvo. Não existem componentes arquiteturais dedicados e nem tampouco uma arquitetura pré-definidos. Podem existir, no entanto, bibliotecas de componentes de processamento e comunicação sobre as quais análises são realizadas a fim de otimizá-los para as restrições de projeto da aplicação. Para que as análises possam ser realizadas, aplica-se o método dos refinamentos sucessivos, desde a descrição em alto nível, até o nível dos componentes arquiteturais presentes nas bibliotecas. Caso exista a necessidade do uso de componentes dedicados, estes devem ser criados em tempo de projeto por ferramentas de particionamento HW/SW e de síntese de alto nível.

A vantagem dessa abordagem reside no fato de que há uma boa chance de se encontrar uma arquitetura altamente otimizada, devido ao amplo de variáveis passíveis de serem otimizadas nos componentes arquiteturais presentes nas bibliotecas de processamento e comunicação. A desvantagem reside na conseqüência direta dessa liberdade: um grande número de variáveis induz a um enorme espaço de projeto a ser otimizado. Outra vantagem pode ser vista na grande capacidade de reconfiguração permitida.

Na abordagem “de baixo para cima” (mais comumente chamada pelo seu equivalente em inglês, *bottom-up*) primeiramente são especificados apenas os componentes arquiteturais, inclusive componentes dedicados para em seguida serem disponibilizados em bibliotecas. Devido à pré-definição de componentes arquiteturais, essa abordagem também é conhecida como Projeto baseado em Componentes (*Component-based Design, CbD*). A pré-especificação de uma arquitetura alvo não é realizada. Para essa abordagem o fluxo de projeto proposto na figura 2.4 deve ser analisado no sentido da menor para a maior abstração. O objetivo maior nessa abordagem consiste no uso de linguagens para a criação de componentes de software em níveis de abstração maiores do que o arquitetural. A especificação da aplicação é realizada através de pelo menos uma dessas linguagens e com isso, as ferramentas de apoio ao projeto podem analisar os componentes no nível de detalhamento mais apropriado. Essas análises geralmente são mais rápidas do que na abordagem top-down, uma vez que existe o caminho entre o nível de abstração utilizado na especificação e os componentes arquiteturais, sendo estes conhecidos. Por este mesmo motivo, análises de particionamento e síntese comportamental são desnecessárias.

A técnica dos refinamentos sucessivos somente necessita ser aplicada a partir do nível de abstração sobre o qual a análise é realizada.

A vantagem dessa abordagem em relação à abordagem *top-down* consiste na rapidez da análise, porém a arquitetura alvo deve ser totalmente especificada em tempo de projeto, o que pode exigir um grande esforço computacional para a sua escolha.

Atualmente, vem surgindo com cada vez mais força o conceito de *Projeto baseado em Plataformas*. Esta abordagem para o projeto de sistemas propõe-se a ser um compromisso entre as abordagens *top-down* e *bottom-up*, no sentido de permitir que o projeto possa ser conduzido através de especificações em alto nível, mas com uma (ou mais) arquiteturas alvo, chamadas de *plataformas*, pré-determinadas. Dessa forma, o fluxo de projeto considerado para o projeto de sistemas complexos (figura 2.4) pode ser analisado nos dois sentidos em relação aos níveis de abstração. Consequentemente, a técnica dos refinamentos sucessivos será utilizada a partir do mais alto nível, mas levando em consideração a pré-definição de uma ou mais plataformas.

A definição da(s) plataforma(s) advém de análise realizada sobre uma determinada *classe* de aplicações. Portanto, essa abordagem objetiva reutilizar a(s) plataforma(s) para uma classe de aplicações com características comportamentais semelhantes. Com isso, espera-se diminuir o tempo de projeto, uma vez que não há a necessidade de se especificar completamente a arquitetura alvo, apenas reconfigurá-la para a aplicação em questão, o que vem a promover o reuso de componentes arquiteturais. Nessa abordagem, o tempo de projeto tende a ser ainda menor do que na abordagem do projeto baseado em componentes, pois estando a arquitetura alvo pré-definida, muitas variáveis de projeto, como por exemplo, a topologia de arquiteturas de comunicação, não necessitam mais serem definidas.

É interessante notar que nas três abordagens, existe a possibilidade de promover um método para co-projeto, desde as análises realizadas sobre os componentes de software possam contar com uma estimativa (por exemplo, de desempenho) acerca de seus equivalentes arquiteturais.

Por tratar-se da abordagem adotada nesse trabalho e para justificar o seu uso, a seção a seguir define o que atualmente entende-se por essa nova abordagem no projeto de sistemas.

2.3.1 Conceitos sobre Projeto baseado em Plataformas

Como citado na seção anterior, o Projeto baseado em Plataformas (*Platform-based Design, PbD*) surge nos presentes dias como uma proposta para o reuso de componentes no projeto de sistemas complexos. Este reuso auxilia no projeto de sistemas de duas maneiras: reduzindo a necessidade de se projetar todos os componentes desde o início, e como conseqüência, pela redução do número de variáveis a serem levadas em consideração quando da otimização de cada componente arquitetural (redução do espaço de projeto).

No entanto, como o conceito exato do que vem a ser o Projeto baseado em Plataformas pode ser definido ? A principal idéia por traz do conceito de PbD reside no fato de que várias aplicações podem ser classificadas como pertencendo uma mesma classe. Isso quer dizer que estas aplicações *compartilham* algumas funcionalidades e comportamentos. Como conseqüência, essas aplicações podem ser sintetizadas para os mesmos componentes arquiteturais, mesmo que em certos casos, estes necessitem serem parcialmente reconfigurados.

Para a implementação de uma aplicação, os componentes da plataforma alvo podem ser utilizados exatamente como foram fornecidos ou podem necessitar de reconfiguração. A reconfiguração se justifica para adaptar um componente às restrições de projeto, tais como área, desempenho e potência.

Nos dias atuais não há ainda um consenso na comunidade científica acerca do que exatamente vem a ser o Projeto baseado em Plataformas. Em [ALT 2002], o PbD é

definido como um conjunto de características comuns, integradas e gerenciáveis sobre as quais um conjunto de produtos pode ser construído. No contexto de SoCs, isso equivale a uma biblioteca de componentes virtuais e um *framework* arquitetural, consistindo de um conjunto de integrado e pré-qualificado de componentes virtuais de hardware e software, modelos, ferramentas de projeto, bibliotecas e metodologias para suportar o rápido desenvolvimento de sistemas através de exploração arquitetural. Já em [WIL 2002] são definidos *tipos* de plataformas. De acordo com essa definição existem 3 tipos de plataformas: focadas em aplicações; focadas em processadores; ou focadas em comunicação.

Em [VIN 2002] uma plataforma é definida como uma coleção de camadas, onde cada uma representa um nível de abstração. Em cada camada, as funcionalidades das camadas inferiores são abstraídas. Dessa forma, é possível definir cada camada através de sua linguagem, bem como a maneira como as camadas se relacionam. Por exemplo, uma camada poderia ser os componentes arquiteturais, uma segunda, os serviços oferecidos por um Sistema Operacional dedicado, cujos *drivers* executam funções sobre esses componentes e uma terceira, a própria aplicação. Essa definição é interessante no sentido de que conceitualmente, idealiza uma aplicação através dos níveis de descrição citados na seção 2.1.1 em relação aos componentes de software. Como visto nessa seção, conceber uma aplicação sob o ponto de vista de diferentes níveis de abstração, é fundamental para a compreensão de sistemas complexos.

Dos parágrafos acima, pode-se constatar que, mesmo o conceito por trás do PbD ainda não estando completamente formalizado, alguns de seus principais atributos podem ser claramente identificados: componentes de comunicação e processamento e a necessidade de diferentes níveis de abstração para a especificação e análise da aplicação. Dessa forma, o projeto inicia-se com a descrição em software e com uma descrição dos componentes arquiteturais da plataforma. As análises sobre a aplicação são realizadas sobre os níveis de abstração representados pelos componentes de software, onde em cada nível é determinada a influência que este exerce sobre os níveis de abstração inferiores. Esta influência se manifesta sobre a alteração comportamental decorrente da alteração no estado das variáveis dos componentes em níveis de abstração inferiores.

É interessante observar que os comportamentos da aplicação relacionados à comunicação e ao processamento possam ser analisados separadamente, uma vez que implementam respectivamente as operações de transferência e transformação de dados. Como esses comportamentos são ortogonais, é possível que ferramentas dedicadas à análise de cada uma dessas classes de comportamento possam operar concorrentemente. Além disso, a comunicação pode ser vista como uma interface que abstrai as operações dos componentes de processamento, o que facilita a integração de propriedade intelectual.

Outra questão importante relacionada com os futuros sistemas diz respeito ao tempo em que um produto novo deve chegar ao mercado. Este tempo vem sendo constantemente reduzido, uma vez que o mercado exige novos produtos, o que faz com o tempo de vida dos produtos também seja reduzido. É esperado que a *vida* estimada para novos produtos, pode em alguns casos não ser superior a 6 ou 8 meses. Dada essa situação, espera-se que devido a sua flexibilidade, cada vez mais as funcionalidades de um aplicação serão especificadas como software.

No entanto, a semântica do comportamento para o software em SoCs tende a ser heterogênea, o que faz com que o ambiente de especificação deva por exemplo, detectar

se a comunicação entre componentes é realizada através um modelo de computação baseado em fluxo de dados ou se é baseado em fluxo de controle. Uma possível solução para este problema pode ser encontrada na definição de um *modelo de programação* para toda a plataforma, onde todos os modelos de computação possam ser expressos – um modelo de programação heterogêneo. Em [KEU 2002] é constatada a necessidade de tal modelo.

Algumas decisões focando a arquitetura alvo poderiam ser determinadas ao nível de sistema e isto deve também ser detectado pelas semânticas disponíveis para a especificação. Outra questão importante nesse ponto, diz respeito às restrições da aplicação. Pode ser importante para algumas aplicações que seja permitido, já nas primeiras fases de especificação, a determinação de algumas restrições. Essas restrições então, também devem ser capturadas no nível de sistema e propagadas em direção à níveis inferiores de abstração para que estas possam ser implementadas nos componentes arquiteturais da plataforma alvo.

Esforços na definição de um sistema através da abordagem PbD têm sido realizados pelo uso da linguagem UML (*Universal Modelling Language*) [LAR 2004]. A abordagem adotada pela UML parece ser bem interessante na definição de uma aplicação a ser mapeada para uma plataforma. Isso se justifica pelo fato de que a UML não se constitui de uma linguagem única com sintaxes e semânticas pré-definidas, podendo ser definida como uma coleção de notações, sintaxes e semânticas definidas para permitir a criação de linguagens para aplicações com comportamentos específicos. Isto é especialmente interessante para a especificação de aplicações com funcionalidades que apresentam comportamentos heterogêneos, como é o caso das aplicações alvo para os SoCs.

Dessa forma, a UML pode ser utilizada não apenas para a especificação das funções em software, mas também para especificar os componentes arquiteturais (de processamento e comunicação) da plataforma. Como diferentes semânticas podem definir diferentes níveis de abstração, toda a plataforma pode ser especificada, o que vem de encontro com a definição de plataforma encontrada em [VIN 2002].

Todos os aspectos relacionados ao projeto de sistemas complexos também estão sendo discutidos pelo grupo de trabalho da *Virtual Socket Interface Alliance*, conhecido como *platform-based design development working group* (PBD DWG) [GOE 02b]. Esse grupo de trabalho tem a intenção de formalizar e padronizar a definição da abordagem de projeto baseada em plataformas.

Como se pode perceber, o Projeto baseado em Plataformas constitui-se de uma abordagem interessante para o projeto, pois promove uma política eficiente para o reuso de componentes arquiteturais, além de criar as bases para a implementação de um método voltado ao co-projeto. Nesse contexto, uma política eficiente refere-se principalmente ao tempo de projeto, uma vez que o PbD proporciona o reuso de uma arquitetura completa, minizando portanto, o número de variáveis de projeto a serem levadas em consideração. Além disso, diversos esforços vêm sendo realizados pela comunidade científica e pela indústria, no sentido de padronizar essa abordagem, inclusive com a adoção da linguagem UML para a definição da plataforma.

No entanto, para a efetiva implementação do PbD novos métodos e ferramentas são necessárias. Um método que se propõe a implementar esse conceito de maneira nativa deve definir quais os diferentes níveis de abstração envolvidos, como cada um define semanticamente os comportamentos do software da aplicação e como estes interagem. As ferramentas, por sua vez, devem implementar essa semântica para que

possam analisar a aplicação e reconfigurar/instanciar os componentes arquiteturais da plataforma.

Como pode ser constatado através da figura 2.4 e das análises acima realizadas, a efetiva implementação de qualquer uma das três abordagens consideradas para o projeto de sistemas complexos exige que se adote um processo de síntese e abstração. De fato, o que diferencia essas abordagens é a maneira como esses processos são utilizados em cada uma. A implementação desses dois processos somente poderá ser eficientemente realizada se ferramentas apropriadas forem utilizadas.

A seção a seguir analisa os processos de síntese e abstração na concepção de sistemas e as ferramentas de apoio ao projeto associadas. Em seguida, são verificadas algumas ferramentas de apoio ao projeto encontradas na literatura e que são utilizadas na síntese de sistemas complexos, segundo as três abordagens citadas.

2.4 O Processo de Síntese e Ferramentas de Apoio ao Projeto

A comunidade científica refere-se às ferramentas de apoio ao projeto de sistemas eletrônicos como ferramentas de CAD (*Computer-Aided Design*), EDA (*Eletronic Design Automation*), HLDA (*High Level Design Automation*), etc. este texto refere-se normalmente à essa classe de ferramentas como ferramentas de apoio ao projeto. A seguir é apresentada uma revisão bibliográfica a fim de comparar o esforço que vem sendo realizado pela evolução das tecnologias para o projeto de sistemas complexos e para contextualizar o uso dessas ferramentas nas três abordagens de projeto discutidas na seção anterior.

A Engenharia de Software preocupa-se em tornar o processo da criação de algoritmos cada vez mais natural para o ser humano, a fim de facilitar a criação e manutenção de software. Isto pode ser verificado na evolução dos compiladores, desde a programação binária, passando pela programação em linguagem montadora, linguagens de programação de alto nível, até as ferramentas para especificação de sistemas da atualidade, através da utilização de componentes, padrões de projeto, API’s (*Application Programming Interfaces*), etc.

Por outro lado, a Engenharia de Hardware preocupa-se na criação de processadores e circuitos integrados cada vez mais otimizados, capazes de realizar um grande número de operações por segundo, com um mínimo de consumo e área.

Como já verificado, devido a sua grande complexidade, as aplicações atuais são especificadas em sua maior parte através de componentes de software em altos níveis de abstração. Para tanto, as aplicações utilizam para a sua especificação ferramentas da engenharia de software, como por exemplo, linguagens de programação, diagramas de classes, diagramas de interação entre objetos, etc. Sobre a especificação é aplicado então um processo de síntese que consiste em mapear (ou traduzir) a especificação em alto nível de abstração para níveis inferiores até se chegar na implementação final. No contexto desse trabalho, a implementação final corresponde à configuração de componentes arquiteturais, ou seja, síntese de software.

A funcionalidade de uma aplicação pode ser sintetizada para processadores ou para circuitos dedicados. A comunicação, por sua vez, pode ocorrer internamente a um processador, através de trocas de mensagens entre processos ou externamente, no caso da aplicação ser implementada na arquitetura alvo em mais de um processador ou circuito dedicado. Para a comunicação externa, normalmente são utilizadas arquiteturas dedicadas de comunicação como barramentos ou redes chaveadas.

Os processadores podem pertencer à classe dos processadores de propósito geral ou de propósito específico. Os processadores de propósito geral (*General Purpose Processor, GPP*) possuem o potencial para executar qualquer algoritmo através de programação, via conjunto de instruções. Já processadores de propósito específico são desenvolvidos para a execução de tarefas com comportamento específico. Isso ocorre porque algumas tarefas possuem certas características comportamentais que executam mais eficientemente (em relação às restrições de projeto), se executadas em circuitos otimizados para as restrições. Visto de outra forma, esses circuitos são projetados visando otimizar a execução das aplicações de acordo com as suas características, como por exemplo, as definidas em [CAR 96]. Isso ocorre porque processadores de propósito geral, por serem genéricos, não possuem circuitos dedicados à execução de comportamentos específicos.

A implementação de comportamentos específicos pode ser programável ou não. No caso de ser programável, esses comportamentos são implementados em processadores com o conjunto de instruções programáveis (*Application Specific Instruction-Set Processor, ASIP*). A implementação de comportamentos específicos em ASIPs ocorre através da inclusão/remoção de componentes arquiteturais na parte operativa e consequente conexão com os demais componentes e a criação de novas instruções para o conjunto de instruções, que utilizam os novos componentes arquiteturais.

Para soluções não programáveis, comportamentos específicos são implementados em circuitos dedicados (*Application Specific Integrated Circuit, ASIC*).

Em termos de nomenclatura, diz-se que um componente de software é sintetizado como *software* quando este é compilado para o conjunto de instruções de GPP e sintetizado como hardware quando este é implementado através de componentes arquiteturais cuja microarquitetura é dedicada à sua execução, ou seja, possui as partes de controle e operativa, otimizadas para a sua execução. A microarquitetura pode ser programável ou não.

Pelo exposto acima, verifica-se a necessidade de ferramentas para a realização do processo de particionamento HW/SW. Estas ferramentas devem decidir quais componentes de software especificados em alto nível devem ser sintetizados como hardware e quais, como software. Em seguida são então utilizadas ferramentas de síntese para o software, para o hardware e para a comunicação entre eles. Se aos componentes de software forem adicionadas estimativas acerca do seu comportamento equivalente em componentes arquiteturais, pode ser implementado um processo de co-projeto. O co-projeto estabelece que componentes de software e de hardware possam ser analisados e sintetizados em paralelo. É o caso onde ocorre em paralelo a síntese de hardware e de software.

Para sintetizar especificações como softwares, são utilizadas ferramentas do tipo compiladores que mapeiam uma descrição para a linguagem do conjunto de instruções. Para sintetizar especificações como hardware não programável, ferramentas de síntese comportamental devem ser utilizadas, as quais mapeiam descrições de componentes de software em componentes arquiteturais. Se for utilizada a abordagem de projeto baseado em componentes, estes devem já estar prontos, bastando serem instanciados. Para a implementação para hardware programável exige ferramentas para a reconfiguração arquitetural de ASIPs. Na abordagem *top-down* para a concepção de sistemas, processadores novos podem ser criados. Nesse caso, além do processador em si, surge a

necessidade de se criar também às ferramentas para sintetizar componentes de software para o conjunto de instruções desse processador.

Finalmente, são também necessárias ferramentas para a síntese de componentes de software de comunicação para arquiteturas de comunicação. Nas abordagens do projeto baseado em plataformas e projeto baseado em componentes algumas variáveis das arquiteturas de comunicação já estão definidas, como por exemplo, a topologia ou a política de roteamento de mensagens. Já para a abordagem *top-down* pode acontecer de todas as variáveis serem definidas em tempo de projeto.

Em seguida, são analisadas algumas ferramentas utilizadas para a implementação dos processos de síntese e abstração na concepção de sistemas, segundo as três abordagens consideradas.

2.4.1 Particionamento HW/SW, Co-projeto e Especificação

Na literatura são encontradas diversas ferramentas ou ambientes voltados para o co-projeto [TAM 97] [KUM 96] [THI 99] de sistemas, visando otimizar a execução das funções de um algoritmo através da possível alocação de algumas de suas funções para a execução em arquiteturas dedicadas.

Um exemplo de um ambiente voltado para o co-projeto pode ser encontrado em [TAM 97], onde a descrição funcional do sistema é realizada em C/C++. Nessa descrição são analisadas as informações sobre transferência de dados e execução do algoritmo para então realizar o particionamento hardware/software. O particionamento HW/SW é realizado automaticamente ou como a ajuda do usuário que pode optar pela execução de certas operações em software ou hardware. Como saída, o sistema gera código C++ para o software e as funções em hardware são descritas em VHDL.

Outra possível maneira de se realizar o particionamento HW/SW ocorre sob o ponto de vista do particionamento de tarefas: nessa abordagem, o sistema alvo é distribuído, de modo que a alocação de tarefas entre um conjunto de processadores, bem como a sua realização como HW ou SW, são consideradas simultaneamente. Consequentemente, estudos nesse sentido são realizados em áreas de pesquisa como a implementação e otimização de sistemas multiprocessados [ELR 92] e co-projeto HW/SW.

Iqbal et al [IQB 94] considera o problema de particionar múltiplas tarefas em sistemas computacionais heterogêneos através do uso de algoritmos de aproximações polinomiais. Esse algoritmo pode ser aplicado para até quatro processadores.

Ravikumar [RAV 95] aborda o uso de algoritmos genéticos para particionar tarefas sistemas multiprocessados. A idéia principal consiste em distribuir tarefas modeladas como grafos em diferentes processadores. Nesse trabalho são realizadas comparações entre algoritmos genéticos e *Simulated Annealing* (SA) [KIR 83] são realizadas. Em nenhum experimento realizado, o algoritmo SA conseguiu obter resultados melhores do que os genéticos, em termos de qualidade do resultado e tempo de execução.

Kumar [LEI 2003] utiliza algoritmos genéticos em duas etapas para particionar grafos de tarefas em redes intrachip. O objetivo é minimizar o tempo total de execução do comportamento de comunicação da aplicação. As tarefas são sempre mapeadas em processadores conectados às portas locais de roteadores, aos quais é anexado um modelo de tempo de execução, utilizado para estimar o tempo de tráfego das mensagens.

Wild em [WIL 2003] compara abordagens construtivas e recursivas para particionar tarefas em processadores e aceleradores em HW, conectados por barramentos. Ambas abordgens foram implementadas utilizando o algoritmo de Busca Tabu (Tabu Search, TS) [GLO 89, GLO 90, GLO 93] e SA. Os experimentos foram conduzidos em aplicações sintéticas, modeladas pela ferramenta para geração automática de grafos de tarefas TGFF (Task Graphs for Free) [DIC 98] em uma plataforma alvo composta por dois diferentes processadores e cinco aceleradores em HW. Os resultados foram obtidos através da simulação de 60 grafos de tarefas de diferentes tamanhos. Os autores concluem que, ao serem comparadas as duas heurísticas, o algoritmo TS oferece desempenho bastante superior do que o SA, para todos os experimentos realizados. Os autores recomendam, portanto, o uso do TS principalmente quando grandes aplicações forem testadas.

Outra comparação entre heurísticas para o particionamento de tarefas entre processadores é realizada em [AXE 97] por Axelsson, onde três heurísticas, TS, SA e algoritmos genéticos (GA), são utilizadas para particionar um conjunto de tarefas concorrentes para processadores e ASICs. Durante a busca por uma arquitetura otimizada, várias operações (chamadas no contexto desse trabalho de “transformações”) são realizadas, tais como a união de tarefas para execução em um único processador ou a distribuição de tarefas entre vários processadores. Os resultados mostraram que todos os algoritmos foram capazes de encontrar soluções otimizadas, mesmo quando as restrições são bastante inflexíveis. Ainda, os algoritmos TS e SA puderam encontrar soluções bem semelhantes em qualidade do resultado. No entanto, o algorimo TS pôde encontrar soluções otimizadas em muito menos tempo do que o algoritmo SA. O autor conclui que “se o número máximo de iterações fosse mais limitado, melhor ainda seriam as soluções encontradas pelo algoritmo TS, em comparação com SA”. Esses resultados nos encoragam a adotar a heurística de Busca Tabu para encontrar arquiteturas de sistemas multiprocessados otimizados.

Uma abordagem que vem sendo adotada para o co-projeto e co-simulação HW/SW de sistemas, diz respeito à utilização de uma linguagem única para a descrição dos componentes de software e dos arquiteturais. Esta idéia tem por objetivo aproximar os domínios de hardware e software para o projeto de sistemas. Seguindo-se os avanços para a descrição de algoritmos da engenharia de software, percebe-se que o paradigma de Orientação a Objetos aparece como uma solução interessante para a descrição de sistemas, devido a algumas características desejáveis em projeto, como por exemplo, confiabilidade, encapsulamento e reutilização de componentes. Como o encapsulamento e a reutilização de componentes são historicamente realizados na criação de componentes arquiteturais, torna-se bastante natural a utilização do paradigma de orientação a objetos também para a descrição de componentes arquiteturais.

Em [ARN 99], [MIC 99], [GHO 99] e [WAK 99] discute-se a utilização da linguagem C/C++ para a descrição de um sistema completo, com o objetivo de reutilizar a grande quantidade de código existente, além de facilitar a compreensão do sistema através de uma descrição única. Arnout em [ARN 99] justifica que o software existente escrito em C, pode ser mais bem compreendido do que as arquiteturas descritas através de linguagens para descrição de hardware, pelo fato de esta ser uma descrição no nível de transferência entre registradores. Também estes artigos mostram soluções para que projetistas de hardware possam utilizar a linguagem C nas suas descrições através da adição da semântica necessária para uma descrição RTL sintetizável, por exemplo.

Atualmente, surge com cada vez mais força a idéia de se especificar os sistemas através do uso da linguagem C++. Isso se deve ao fato da popularidade da linguagem e

da ampla base de profissionais capacitados na descrição de sistemas através dessa linguagem. Talvez o exemplo de maior sucesso nessa direção possa ser encontrado na linguagem SystemC [GRO 2002]. SystemC compreende a classe de ferramentas onde biblioteca de classes são utilizadas para a descrição do comportamento dos componentes arquiteturais e de software. Outras ferramentas que seguem essa tendência são SpecC [SPE 2004] e SoC++ [VAN 2001].

Em [KUN 99] o modelo de componentes *JavaBeans* é adaptado para permitir uma descrição estrutural e comportamental de um sistema através da utilização da linguagem Java. Dessa forma pode-se criar ou modificar arquiteturas através da conexão entre componentes JavaBeans. Além disso, esses componentes podem ser vistos em diversos níveis de abstração através de apropriada interpretação da funcionalidade dos objetos. Os componentes arquiteturais são chamados *HardwareBeans* que sendo utilizados na simulação e convertidos para equivalentes em VHDL.

Em relação ao particionamento HW/SW, co-projeto e especificação de componentes arquiteturais, o método aqui proposto procura contribuir para o estado-da-arte: 1) estabelecendo o uso de uma meta-linguagem única (e padrão) na especificação em alto nível dos comportamentos para componentes arquiteturais; 2) propondo a descrição separada dos comportamentos de processamento e comunicação, a fim de avaliar o impacto de cada um sobre as restrições de projeto da aplicacao alvo; 3) considerando o particionamento HW/SW em conjuntos de processadores heterogêneos e 4) avaliando a influência de arquiteturas de comunicação sobre a execução concorrente de tarefas.

2.4.2 Ferramentas para Análise e Otimização de Arquiteturas de Processamento

Esta classe de ferramentas avalia o grau de otimização de componentes de processamento em relação às restrições de projeto para aplicações alvo. O processo de avaliação é efetivado com o mapeamento e posterior execução das tarefas da aplicação em componentes arquiteturais para processamento, os quais podem apresentar conjunto de instruções genérico ou de propósito específico.

Uma abordagem para análise de processadores é encontrada em [CAR 99], onde se verifica o comportamento da execução de um algoritmo - representado através de um grafo de fluxo de controle e dados - sobre vários processadores. Dessa forma é idealizada a síntese baseada em seleção de processadores, onde é verificada a adequação dos processadores na execução da aplicação, com base nas funções para QoS definidas em [CAR 96].

Em [ALB 96] é definida uma ferramenta para a geração automática de microcontroladores dedicados, baseado no microprocessador Risco [JUN 93]. O sistema otimiza a arquitetura do processador, gerando para a parte operativa, unidades funcionais dedicadas às tarefas da aplicação não otimizadas à execução nas unidades especificadas para o processador. A otimização é realizada com base na execução de instruções de controle, de computação ou de acesso à memória [CAR 96].

Esforço semelhante é realizado em [KRE 97] onde, a partir de uma especificação em C, é gerado um microcontrolador MCS-8051 [INT 85] otimizado com um conjunto de instruções composto somente pelas instruções mais utilizadas pela aplicação. As outras instruções são emuladas e o compilador C adaptado para gerar somente o conjunto de instruções implementado. Nesse caso, o objetivo maior é a redução da área relativa à decodificação de instruções da Parte de Controle, pela redução do número de

instruções. Ainda, em [ITO 2000] uma abordagem semelhante é adotada para a geração de microcontroladores nativos para o *bytecode* Java, a partir de especificações Java. Nessa abordagem foi definida uma arquitetura dedicada para a Máquina Virtual Java onde, dependendo das características da aplicação, são implementadas somente aquelas instruções que a melhor caracterizam.

Uma outra abordagem, encontrada em [CAR 2000], visa à implementação de sistemas complexos multiprocessados. Nessa abordagem a implementação do sistema é realizada através de algoritmos para a seleção de processadores de propósito geral, GPPs. A síntese do sistema é realizada através da comparação dos MoCs definidos pelas tarefas da aplicação com o conjunto de instruções de processadores comerciais. O processador mais otimizado é selecionado para executar a aplicação.

Um dos objetivos dessa abordagem reside no fato de que em muitas empresas existe uma cultura no uso de determinados processadores, fundamentada na experiência dos projetistas de aplicações no uso das ferramentas de apoio ao projeto, tais como compiladores, *debuggers*, etc; e na própria arquitetura do processador. Assim, acredita-se que é interessante para as empresas tentar adequar o máximo possível, o uso dos processadores presentes na cultura de desenvolvimento local para a implementação das aplicações.

Entretanto, mesmo que haja uma grande biblioteca de processadores para serem selecionados de acordo com o método apresentado em [CAR 2000], pode acontecer de um processador somente poder ser selecionado para a execução da aplicação, se o seu conjunto de instruções for reconfigurado. Neste caso necessita-se de componentes arquiteturais do tipo ASIP.

As abordagens apresentadas em [ALB 96] e [KRE 97] tratam dessa questão uma vez que permitem a reconfiguração do conjunto de instruções para os processadores Risco [JUN 93] e MCS-8051 [INT 85] respectivamente. Ainda, em [ARC 2005] e [TEN 2005] uma abordagem semelhante é adotada, onde microprocessadores ASIP podem ter os seus componentes microarquiteturais configurados antes de serem utilizados. Duas principais diferenças podem ser ressaltadas entre essas abordagens: em [CAR 96] e [KRE 97] a reconfiguração do conjunto de instruções é automática, realizada com base no comportamento da aplicação, enquanto que em [ARC 2005] e [TEN 2005] a reconfiguração é manual, sendo delegada ao projetista de sistemas, a tarefa de analisar a execução da aplicação nos processadores. Por outro lado, essas abordagens permitem a reconfiguração de todos os componentes arquiteturais dos processadores, como por exemplo, a ULA, o banco de registradores, etc.

Uma característica interessante a ser observada nas abordagens acima é a opção pelo reuso dos componentes arquiteturais para a composição do conjunto de instruções. Em sistemas complexos, compostos por um grande número de componentes, o reuso de componentes torna-se um atrativo para a diminuição do tempo de projeto.

Ainda, sobre a geração e avaliação de ASIPs, Weber em [WEB 2004] propõe um framework para construir e simular arquiteturas de processadores dedicados. Nessa abordagem, os componentes de processadores são descritos através de uma linguagem estrutural, a qual requer apenas que sejam especificados os componentes pertencentes à parte operativa do processador. A parte de controle é automaticamente gerada pela ferramenta proposta. Finalmente, simuladores dedicados aos processadores descritos são gerados. O objetivo principal da ferramenta é a avaliação de diferentes processadores através de simulação.

Cong em [CON 2004] propõe a geração de instruções específicas para aumentar o desempenho de processadores ASIP. As instruções dedicadas implementadas para o processador são extraídas de um grafo de fluxo de dados através de uma função custo que considera o desempenho e a área ocupada pelos componentes arquiteturais necessários à implementação de cada instrução. Cada instrução dedicada refere-se à composição de instruções presentes no grafo de fluxo de dados.

Linguagens para a descrição de arquiteturas (ADL, Architecture Description Languages) são comumentemente utilizadas na especificação de arquiteturas de processamento. Como exemplo, pode-se citar as abordagens adotadas em LISA [HOF 2002], EXPRESSION [HAL 99] e nML [FAU 95]. Essas linguagens possuem primitivas pelas quais conjuntos de instruções podem ser especificados. Nesses trabalhos, a geração dos sinais de controle (micro-instruções) são gerados automaticamente.

Atasu em [ATA 2003] propõe o aumento de desempenho para aplicações *dataflow*, através da reconfiguração dos componentes de um processador. Essa abordagem é embasada em um algoritmo voltado à extração do paralelismo da aplicação, expresso como um grafo de fluxo de dados.

Glökler et al. em [GLO 2003], discute diversas possibilidades de reconfiguração arquitetural que podem ser realizadas sobre ASIPs a fim de se diminuir o consumo de energia. Nessa direção, otimizações são consideradas a nível do conjunto de instruções, a nível arquitetural (unidades funcionais) e a nível de software. As otimizações são realizadas através de análise dinâmica de aplicações em linguagem *assembly*. A arquitetura do processador analisado - ICORE – é descrita através da linguagem LISA [HOF 2002], para descrição de arquiteturas. No nível arquitetural são realizadas análises quanto ao compromisso flexibilidade$_x$consumo de energia. Os autores concluem que, em blocos onde o consumo de energia deve ser preservado a um nível mínimo, deve-se pagar o preço da perda da programabilidade, pela realização em HW. No nível do conjunto de instruções, otimizações são realizadas no sentido de se implementar em HW, seqüências de instruções mais utilizadas pela aplicação, além da compactação da palavra de instrução, visando diminuir o consumo da memória de programa. Ao nível de SW, são realizadas as operações de controle para o mode *stand-by* do processador e pelo controle (manual) do número de acessos à memória de dados, através do máximo uso dos registradores internos do processador. Resultados obtidos através da realização de todas as otimizações propostas, remetem a uma redução de 92% do consumo de energia, se comparada com a versão do processador sem nenhuma otimização.

Outra classe interessante de processadores a ser considerada para implementação aplicações paralelas, resume-se nos processadores para aplicações de protocolos de comunicação (*network processors*). Essa classe de processadores pode ser concebida para atender à certas particularidades específicas à comportamentos encontrados em aplicações distribuídas. Por exemplo, pode ser definida a implementação de certas camadas do protocolo TCP/IP como uma unidade funcional de um processador, a fim de aumentar o desempenho da comunicação.

Em [XIE 2003] são definidas otimizações no conjunto de instruções de processadores, com a finalidade de otimizar o desempenho na execução do protocolo TCP/IP. Dentre as otimizações realizadas, pode-se citar o aumento do desempenho na busca de instruções na memória *cache*. As avaliações são realizadas através de simulação, utilizando o simulador *SimpleScalar* [BUR 97].

Em [ERB 2003] é definido um ambiente para modelagem e simulação de aplicações em dois níveis de abstração: aplicação (especificação do algoritmo) e arquitetural. Nesse ambiente é proposto o mapeamento da aplicação para uma arquitetura multiprocessada, levando em consideração restrições de projeto como desempenho, por exemplo.

Ainda em relação às aplicações distribuídas, Quinn [QUI 2004] sugere a integração de um processador reconfigurável a uma plataforma para exploração de processadores a nível de sistema. A partir de uma especificação da aplicação em alto nível, é possível explorar-se diversas configurações para processadores, com a finalidade de aumentar o desempenho.

Considerando-se os trabalhos acima citados e comparando-os com os objetivos desse trabalho (seção 1.4), pode-se considerar que aqui se procura avançar o estado-da-arte no sentido de se avaliar o particionamento de tarefas da aplicação entre processadores. Além disso, quanto à avaliação de processadores ASIPs, na abordagem aqui apresentada, se procura considerar a inclusão de unidades e instruções específicas com base não apenas, na demanda computacional de tarefas, mas também no comportamento dessas.

2.4.3 Ferramentas para Análise e Otimização de Arquiteturas de Comunicação

Em relação às arquiteturas de comunicação verifica-se a necessidade de métodos e ferramentas para avaliar a qualidade-de-serviço dessa classe de arquiteturas em relação ao comportamento de comunicação de aplicações dedicadas.

É importante ressaltar que, sendo a arquitetura alvo um sistema distribuído, a comunicação pode representar um grande impacto nas restrições de projeto da aplicação, como desempenho e potência, por exemplo. O correto dimensionamento dessas arquiteturas torna-se então, de evidente importância para a garantia de QoS requerida pela aplicação. Além disso, as arquiteturas de comunicação devem ser ser implementadas por componentes arquiteturas capazes de prover o suporte necessário à realização de comportamentos de comunicação heterogêneos. Em relação à comunicação, a heterogeneidade pode ser expressa em relação aos diferentes protocolos utilizados para as comunicações entre as tarefas de aplicações alvo. A implementação destes é realizada através da configuração de diferentes políticas para roteamento e para arbitragem e o controle de fluxo.

Pesquisas recentes [BEN 2002] vêm apontando para a importância do uso de arquiteturas de comunicação do tipo redes chaveadas. Essas arquiteturas, inicialmente utilizadas no contexto de sistemas distribuídos, têm sido cada vez mais consideradas para a implementação de sistemas integrados, principalmente devido ao impacto que podem causar nas restrições de projeto de uma aplicação. Em sistemas dedicados, as redes chaveadas são denominadas “Redes-em-Chip” ou mais comumente, pelo seu equivalente em inglês, *Networks-on-Chip* - *NoCs*, uma vez que são projetadas para implementar a arquitetura de comunicação de SoCs.

Em [BEN 2001] é argumentada a importância de se considerar a influência que estas redes exercem na otimização de SoCs. Nesse trabalho é sugerido que a análise do nível de otimização das arquiteturas de SoCs para as restrições de projeto de aplicações dedicadas, seja iniciada e impulsionada pelas redes chaveadas, o chamado projeto baseado em comunicação.

A arquitetura das redes chaveadas consiste de uma *topologia* que conecta um determinado número de *roteadores*. Roteadores são componentes arquiteturais onde são definidos os comportamentos da comunicação. Cada roteador possui *n* canais de entrada e *m* canais de saída e (pelo menos) uma porta local, através dos quais as mensagens são encaminhadas. Mensagens que chegam ao roteador destino são enviadas a porta local e caso contrário, para umas das *m* portas de saída. O número de portas de cada roteador é determinado em função da topologia da rede. Comportamentos de comunicação são determinados pelas operações de:

- *Arbitragem*, a qual define *quando* mensagens podem ser enviadas adiante, através da seleção da porta de saída;

- *Roteamento*, que determina o caminho das mensagens desde a origem até o destino;

- *Controle de fluxo*, que determina quando um canal de comunicação e o buffer destino estão livres para o envio de um pacote; e

- *Memorização*, que define a arquitetura da memória interna de roteadores, onde pacotes de mensagens são temporariamente armazenados, até serem reenviadas para o próximo roteador.

Mais detalhes acerca do funcionamento e de características de arquitetura de roteadores para redes chaveadas, podem ser encontrados em [ZEF 99].

Estudos realizados recentemente [ZEF 2002] apontam para a necessidade do uso de NoCs em relação ao desempenho, em comparação com arquiteturas do tipo barramento. Neste trabalho, é demonstrado que, dependendo da largura de banda requerida pela aplicação, somente as redes chaveadas a conseguem atingir. Essa situação ocorre principalmente quando o sistema possuir um grande número de cores, pois essa é a situação onde o paralelismo oferecido pelas NoCs pode melhor se manifestar. As figuras 2.5 e 2.6 mostram respectivamente as arquiteturas do tipo grelha e árvore gorda, onde pode ser observado o paralelismo que a topologia de cada uma oferece. Um número grande de processadores (*cores*) exigiria de uma arquitetura do tipo barramento que fosse implementada através de fios muito longos, o que induz a um grande consumo de potência.

Outra questão pertinente em relação ao uso de arquiteturas de redes chaveadas recai sobre o problema da distribuição de relógio, que ocorre em sistemas formados por muitos componentes. Para a solução desse problema, propõe-se o uso de redes de comunicação, onde as comunicações locais – realizadas em um processador conectado à rede – são síncronas e as não-locais, assíncronas. Esse princípio é conhecido como GALS (*Global Asynchronous, Local Synchronous*) [SMI 2004].

Reconhecendo a importância das arquiteturas de comunicação para os sistemas integrados, ferramentas foram desenvolvidas para a realização da análise de desempenho e síntese dessa classe de arquiteturas. Algumas outras ferramentas avaliam o desempenho da arquitetura de barramento em relação às características de comunicação da aplicação, em alto [DAV 97] e baixo [KNU 99] nível de abstração. Ainda, em [KRE 2001] é realizada uma avaliação do desempenho de várias arquiteturas de comunicação em relação às variáveis que caracterizam o comportamento da aplicação. Esta ferramenta obtém o comportamento de comunicação da aplicação através de simulação, e o compara com as características que definem as arquiteturas de comunicação analisadas. A comparação é realizada em alto nível de abstração, sendo selecionada aquela arquitetura que melhor desempenho possuir em relação ao

comportamento da comunicação. A ferramenta foi especificada de modo a ser facilmente escalável para a realização da avaliação de arquiteturas de comunicação em níveis de abstração inferiores.

Em relação à ferramentas para de apoio ao projeto voltadas à otimização de arquiteturas de comunicação do tipo rede chaveadas, vários trabalhos podem ser destacados.

Em [CES 2002] um sistema pode ser modelado sob diferentes níveis de abstração e mapeamento de um padrão de comunicação, do nível de sistema para o nível RT, é proposto. Os protocolos são definidos pelo projetista sendo responsabilidade das ferramentas a geração de interfaces de redes e *wrappers*.

Em [ORT 98, PIN 2002] são apresentadas técnicas para a síntese de comunicação. Em [MEY 2003] é proposta a simulação, análise e síntese de políticas de escalonamento para barramentos, baseada em uma árvore que especifica níveis de arbitragem. Em [ABD 2003] é apresentada uma metodologia e algoritmos para o refinamento automático da comunicação de uma aplicação. Uma arquitetura de comunicação baseada em barramentos é gerada a partir de uma descrição de alto nível, a qual especifica uma coleção de processos comunicantes.

Em [ROW 2003] é proposta uma metodologia centrada no projeto baseado em interfaces para refinar a comunicação entre processos em uma arquitetura de comunicação. Em [GAU 01b, GHA 2002] são propostos algoritmos para a síntese de arquiteturas de barramentos, com suporte da geração de *wrappers*.

Mihal em [MIH 2003], apresenta uma metodologia para o mapeamento de aplicações concorrentes em arquiteturas de redes chaveadas. A aplicação e as arquiteturas de redes são modeladas e caracterizadas através de grafos. As ferramentas definidas para essa metodologia possuem como finalidade principal, a redução da energia total da aplicação, mantendo as restrições de desempenho.

Hu and Marculescu [MAR 2003] mostram que algoritmos para o mapeamento de processadores em terminais locais de NoCs conseguem reduzir em até 60% o consumo de energia se comparado com soluções *ad hoc*. Murali and De Micheli implementam uma solução similar em [MUR 2004]. O foco principal desses trabalhos reside no desenvolvimento de algoritmos para posicionar processadores em terminais locais de redes, com restrições de largura de banda, objetivando reduzir o consumo de energia.

Murali and De Micheli [MIC 2004] extendem o trabalho proposto em [MUR 2004], pela introdução de uma ferramenta, chamada *Sunmap*, a qual é construída sobre uma biblioteca predefinida de topologias de rede. A ferramenta utiliza uma função custo multi-objetivo, a qual considera atrasos de comunicação, área e consumo de energia. O objetivo principal dessa ferramenta refere-se à seleção automática de uma determinada topologia em conjunto como posicionamento dos processadores.

Hu and Marculescu [MAR 2004] propõem um modelo para capturer o escalonamento de comunicação e processamento, o qual é baseado em uma representação de um grafo de comunicação (communication task graph, CTG). Esse grafo permite a obtencao de resultados mais apurados dos encontrados em [MAR 03 e MUR 2004], uma vez que considera a dinâmica das comunicações.

Considerando-se os trabalhos acima citados e comparando-os com os objetivos desse trabalho (seção 1.4), pode-se considerar que aqui se procura avançar o estado-da-

arte na análise de arquiteturas de comunicação, considerando-se diferentes topologias, componentes internos de roteadores de NoCs, além de redes heterogêneas.

2.4.4 Sistemas Operacionais Dedicados

Por tratarem-se de sistemas dedicados, os sistemas aqui observados possuem funcionalidades específicas que são implementadas em um arquitetura alvo dedicada. De acordo com as restrições de projeto impostas, as funcionalidades da aplicação serão distribuídas para os componentes arquiteturais que melhor os otimizarem. Como conseqüência, a aplicação é implementada em uma arquitetura alvo distribuída.

As funções de comunicação dos processos da aplicação necessitarão de serviços de Sistemas Operacionais para serem efetivadas. Isso pode acontecer tanto internamente (trocas de mensagens entre processos) quanto externamente (trocas de mensagens entre componentes arquiteturais).

Além disso, quando mais de um processo for mapeado para o mesmo processador, existem a necessidade de que estes sejam escalonados de modo a que tenham o seu desempenho otimizado, tarefa típica de sistemas operacionais.

Como a aplicação é dedicada e muitas vezes, possui sérias restrições de memória e desempenho, torna-se necessário que um sistema operacional dedicado seja implementado para a aplicação. Por dedicado entende-se por exemplo, que as primitivas de comunicação implementadas no sistema operacional corresponderão somente àquelas utilizadas pelas funções da aplicação.

No caso do projeto baseado em plataformas, quando uma plataforma for reconfigurada, muitas vezes será necessário que se faça o mesmo com o sistema operacional. Por exemplo, se alguma variável da arquitetura de rede for reconfigurada, as funções que a utilizam necessitarão de outras primitivas do sistema operacional para enviar as mensagens. Outro exemplo ocorre quando for acrescentado um processo em um processador. Isso pode demandar uma nova política de escalonamento para aquele processador.

Uma vez que se está tratando de sistemas distribuídos, o sistema operacional também terá que possuir a capacidade para gerenciar recursos na rede.

Portanto, constata-se a necessidade de ferramentas para a geração automática de sistemas operacionais dedicados, devido ao grande número de variáveis no espaço de projeto envolvendo a otimização de sistemas operacionais.

Seguindo essa tendência há ferramentas dedicadas à geração de sistemas operacionais dedicados para sistemas complexos. Em [GAU 2001] é gerado um sistema operacional específico a partir de um núcleo básico, contendo apenas o escalonador de tarefas, um serviço de interrupções e um sistema de entrada e saída. A partir das características da aplicação é configurada a política de escalonamento, o sistema de interrupções e o protocolo de comunicações adotado para a aplicação. O código da aplicação é reconfigurado para realizar as chamadas às funções do Sistema Operacional que implementam a biblioteca de comunicação em questão.

2.4.5 Ferramentas para Análise e Otimização de Sistemas Completos

Seguindo a abordagem do projeto baseado em componentes, arquiteturas de comunicação específicas para sistemas multiprocessados estão sendo desenvolvidas, como por exemplo, a arquitetura do tipo árvore gorda chamada SPIN [GUE 2000], definida no Laboratoire d'Informatique Paris 6 - (LIP6) e a arquitetura do tipo

barramento chamada *CoreConnect* [IBM 2002], definida pela IBM As figuras 2.5 e 2.6 ilustram respectivamente essas duas arquiteturas de comunicação.

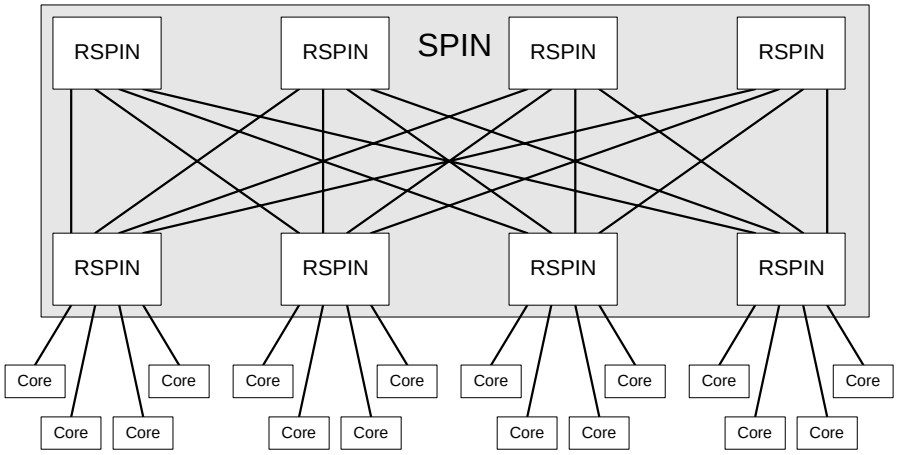


Figura 2.5: Arquitetura de Rede Chaveado do tipo Árvore Gorda - SPIN

Em [ALT 2003] e [XIL 2002] um sistema completo contendo componentes de processamento e comunicação do tipo barramento pode ser especificado e implementado através da arquitetura de FPGAs. É interessante notar que, mesmo os componentes arquiteturais sintetizados como hardware não programável, nesse caso passam a serem também programáveis devido a sua implementação em FPGA. Além dos componentes arquiteturais, essas soluções compreendem ferramentas de síntese e análise, como compiladores, simuladores e depuradores.

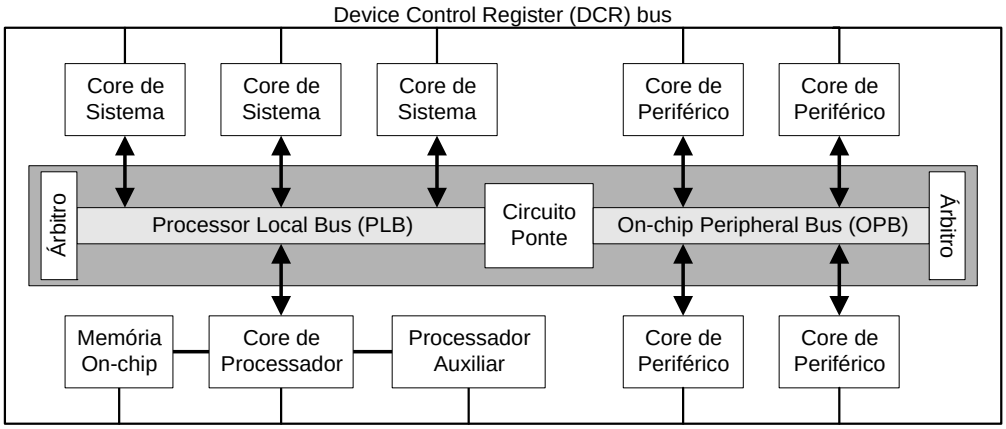


Figura 2.6: Arquitetura do barramento CoreConnect

Em [CES 2002] é definido um modelo onde são especificados tanto componentes arquiteturais, quanto de software. A ferramenta então implementa a comunicação entre os componentes de processamento e comunicação, através da geração de *wrappers* para o software e para o hardware. *Wrappers* de software são implementados através de chamadas a primitivas de comunicação de um sistema operacional dedicado e *wrappers* de hardware são implementados através da configuração de protocolos de comunicação descritos em VHDL.

CODEF [AUG 2002] é uma ferramenta que mapeia um grafo de fluxo de dados para componentes arquiteturais, presentes em uma biblioteca. Para tanto, existe uma ferramenta que realiza análise estática para determinar a viabilidade temporal da execução do grafo de entrada sobre os componentes arquiteturais de processamento

disponíveis. Sobre os componentes arquiteturais de comunicação, é realizada uma análise para determinar se estes atendem às restrições temporais da aplicação, através de comunicação síncrona e dirigida por interrupções.

Outras ferramentas voltadas ao projeto baseado em componentes são Cadence VCC (*Virtual Component Codesign*) [CAD 2002] e Sonics Silicon Backplane µNetwork [WIN 2001].

Em relação à síntese ao nível de sistemas, verifica-se que em [BRU 2000] é proposto um processo para o refinamento da comunicação HW/SW baseado em uma rede processos Kahn [KAH 74]. SpecC [SPE 2004] realiza a síntese de alto nível a partir de um modelo funcional não temporal, especificado em na linguagem C++, extendida para atender à semânticas específicas de sistemas dedicados, como concorrência, por exemplo. O processo de síntese é guiado através de componentes arquiteturais especificados também em SpecC. Os componentes que são sintetizados como software são compilados para o processador em questão e os componentes que são sintetizados como hardware utilizam uma ferramenta de síntese comportamental. A ferramenta CoWare N2C [COW 2002] também possui recursos para a síntese de alto nível, onde explorações arquiteturais são realizadas a partir de componentes de software descritos em SystemC.

O projeto baseado em plataformas encontra hoje em dia grande aceitação por parte da indústria. Várias empresas possuem produtos para a definição de plataformas. A definição dos componentes arquiteturais é realizada de forma manual. Como exemplos, podem-se citar as ferramentas Co-Ware N2C [COW 2002], Mentor Platform Express [MEN 2003] e Altera SOPC Builder [ALT 2003]

Ainda, em [BAG 2001] encontra-se a definição de um modelo arquitetural para a implementação de SoCs. Nessa abordagem é definida uma arquitetura genérica para sistemas, composta por componentes arquiteturais de processamento e de comunicação. Com base nessa arquitetura, os autores defendem que podem ser implementadas ferramentas para a geração automática de sistemas, com características como escalabilidade.

O ambiente Ptolemy [LEE 2002] define diversos modelos de computação para a especificação da aplicação, como por exemplo, Máquinas de Estados Finitos, Fluxo de Dados Síncrono, Tempo Discreto e Síncrono/Reativo. Para a especificação de sistemas foram implementadas diversas classes em UML e em Java, que correspondem à execução dos modelos de computação. Dessa forma, é possível descrever qualquer sistema dedicado, onde os diversos níveis de abstração são implementados através de composição hierárquica. Para tanto, é disponibilizada uma linguagem gráfica baseada em Java. Através desse método de composição hierárquica também é implementado um mecanismo que permite a interação entre diferentes modelos de computação.

O projeto MESCAL (Modern Embedded Systems: Compilers, Architectures, and Languages) [MES 2003] define um modelo de programação e um ambiente de desenvolvimento para a especificação de sistemas dedicados. Nesse ambiente, componentes arquiteturais são especificados utilizando-se o modelo de computação fluxo de dados *booleano* utilizando-se a linguagem definida no ambiente Ptolemy. São permitidas diversas granularidades (desde ULA's até processadores VLIW) na descrição dos componentes arquiteturais, descritos de forma hierárquica. Uma vez descritos os componentes arquiteturais, estes são exportados para um compilador e um simulador, que assim, são automaticamente configurados para a arquitetura. O

compilador e o simulador reconhecem os componentes arquiteturais através da semântica do modelo de computação fluxo de dados *booleano*.

Metropolis [GOE 02a] é um ambiente de desenvolvimento para sistemas dedicados que permite a especificação da semântica de componentes de comunicação e processamento através de blocos básicos. A descrição dos componentes é realizada através de um meta-modelo que especifica a sua semântica, através da definição formal de diversos modelos de computação. O meta-modelo é então traduzido por um compilador para uma linguagem intermediária que serve como entrada para ferramentas de síntese, análise e verificação.

Andréas em [WIE 2004] propõe uma metodologia para projetar e otimizar arquiteturas de processadores e de arquiteturas de comunicação do tipo barramento concorrentemente. Processadores são descritos através da linguagem LISA [HOF 2002] enquanto que barramentos são descritas em *SystemC* [GRO 2002]. A avaliação das arquiteturas é realizada por simulação.

Finalmente, Pierre em [PIE 2002] sugere o ambiente "*StepNP*" para a avaliação de arquiteturas de processadores RISC e de redes intrachip, focando em aplicações de rede. Aplicações são especificadas como tarefas, as quais são mapeadas estaticamente para processadores. As arquiteturas dos processadores e das redes são descritas na linguagem C++. O ambiente possui ferramentas para simulação e visualização gráfica do desempenho das arquiteturas analisadas. O projetista pode configurar o número de processadores utilizados para a aplicação alvo; o número de tarefas em cada processador e a latência para a rede intrachip e para as memórias.

2.5 Conclusões

Este capítulo tratou dos conceitos relativos ao projeto de sistemas eletrônicos complexos, bem como dos processos e ferramentas de apoio ao projeto, necessárias para a sua efetiva realização. Foram analisadas três abordagens para a concepção de sistemas, as quais se constituem atualmente, nos principais caminhos trilhados na tentativa de ajudar projetistas de sistemas na realização da complexa tarefa da exploração do espaço de projeto de sistemas dedicados complexos. Foram também discutidas algumas ferramentas dedicadas à implementação de sistemas, seguindo as abordagens citadas.

O próximo capítulo propõe um novo método para a concepção de sistemas complexos, o qual visa otimizar os componentes arquiteturais de plataformas de comunicação e de processamento. Para tanto, é definido um fluxo de projeto, onde são observadas as ferramentas necessárias para a análise concorrente de arquiteturas de comunicação e de processamento nos diferentes níveis de abstração considerados para o método.

Objetiva-se com a criação desse método e de suas ferramentas, avançar o estado-da-arte na especificação e otimização de plataformas arquiteturais de sistemas complexos. Para tanto, será proposto um novo fluxo de projeto que contempla a exploração concorrente e conjunta de arquiteturas de processamento e comunicação, além das ferramentas necessárias à sua realização. Como pode ser visto nesse capítulo, existem diversas ferramentas devotadas à especificação e síntese de aplicações em arquiteturas de comunicação e processamento. As ferramentas aqui propostas diferenciam-se principalmente pela capacidade de explorar em conjunto o comportamento de componentes arquiteturais de processamento e comunicação, a fim de encontrar uma plataforma otimizada à execução de cada um desses comportamentos,

frente a restrições de projeto. Para tanto, o impacto desses comportamentos nas restrições de projeto é analisado frente a diversas opções arquiteturais disponíveis para implementá-los.

Está-se, portanto, procurando automatizar a exploração do espaço de projeto, uma vez que a execução manual dessa tarefa reduz o espectro de análise possível, frente à grande variedade de componentes disponíveis, a serem avaliados.

3 OPERAÇÕES PARA OTIMIZAÇÃO DE PLATAFORMAS ARQUITETURAIS

Para que plataformas arquiteturais possam ser otimizadas, é necessário que sejam definidas especificadas as aplicações alvo, bem como os componentes arquiteturais de cada plataforma a ser analisada. Uma vez que aplicações e plataformas possam ser especificadas, devem ser determinadas as operações necessárias à otimização de cada componente arquitetural para as plataformas de processamento e comunicação.

Esse capítulo tem por objetivo determinar como aplicações alvo e componentes arquiteturais podem ser especificados, a fim de que estes sejam sujeitos à otimização. Com esse fim, uma linguagem de grafos é utilizada, uma vez que a sua sintaxe utiliza notações padrão, não incorrendo em sintaxes específicas.

Além da definição de aplicações alvo e de componentes arquiteturais, neste capítulo também são determinadas às operações necessárias ao processo de otimização arquitetural, bem como a sua complexidade.

3.1 Definição de Aplicações Alvo e Plataformas Arquiteturais para Processamento e Comunicação

Aplicações especificadas no nível de sistema são representadas por um conjunto de tarefas. Tarefa possui propriedades que especificam a natureza dos seus comportamentos. Uma vez que são esperadas aplicações distribuídas para a implementação das funcionalidades de aplicações dedicadas, supõe-se que tarefas executam concorrentemente, condicionadas por dependências de dados ou controle. Além disso, a comunicação entre tarefas é realizada através de protocolos de comunicação, os quais seguem a semântica definida por um MoC. Aplicações podem ser especificadas através de um grafo, onde os nodos representam as tarefas os arcos a comunicação entre estas.

A granularidade de tarefas é relacionada com o nível de abstração adotado. No nível de mensagens, tarefas são representadas por um conjunto de blocos básicos, como definido abaixo. Já no nível de transações, cada bloco básico é representado por operações para transformação de dados (lógico/aritméticas) e de controle. Operações de controle podem incluir operações dedicadas para comunicação.

Definição 3.1, *Bloco Básico (BB)*: um *BB* é um vetor *bb* = {*mem*, *la*, *ctrl*, *com*, *dst*}, onde: *mem*: número de instruções para acesso à memória; *la*: número de instruções lógico/aritméticas; *ctrl*: instrução de controle; *com*: número de bytes para comunicações ou número de repetições para instruções de laço e *dst*: tarefa destino (utilizado em comunicações). Instruções de controle podem ser:

ctrl: LOOP, laço; SYNC_N, comunicação síncrona em uma rede; ASYNC_N, comunicação assíncrona em uma rede; SYNC_M, ASYNC_M, comunicações síncrona

e assíncrona locais, utilizando memória compartilhada, local ao processador onde as tarefas executam.

Blocos básicos podem ser automaticamente gerados por ferramentas de instrumentação de código, como por exemplo, a ferramenta BIT, *Bytecode Instrumentation Tool* [ZOR 97], para aplicações fonte Java. Código pode ser automaticamente gerado a partir de blocos básicos, bastando a uma ferramenta gerar o número de instruções de cada tipo. Nesse nível de abstração, não havendo informações específicas de cada instrução, apenas aplicações sintéticas podem ser geradas. No entanto, na avaliação de processadores pode-se assumir que as instruções lógico/aritméticas de um conjunto de instruções específico possuem a mesma temporização.

Definição 3.2, *Tarefa*: uma tarefa pode ser caracterizada por um conjunto de *n* blocos básicos: T = {bb_0, bb_1, ..., bb_n}. Cada tarefa possui associada a identificação do processador ao qual está alocada para execução: pT[x] = y; tarefa *y*, alocada ao processador *x*; pT ∈ PROC, da definição 3.6.

A figura 3.1 mostra a relação hierárquica entre tarefas (T_x), blocos básicos (BB_x) e processadores (P_x), salientando os diferentes tipos de comunicação que podem ocorrer.

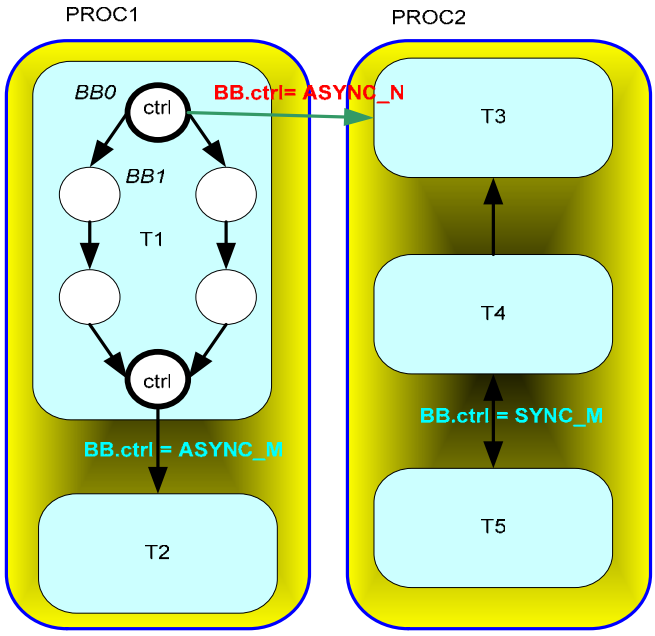


Figura 3.1: Relação hierárquica entre blocos básicos, tarefas e processadores.

Pode-se verificar na figura 3.1 que os BBs sempre devem terminar por uma instrução de controle, pois esta determina se instruções devem ser re-executadas – para o caso de laços – ou quais as próximas instruções a serem executadas – chamadas de funções locais ou remotas. Chamadas de funções locais utilizam memória compartilhada enquanto que as chamada remotas são mensagens enviadas pela rede. A informação acerca do tamanho das mensagens é determinada pelo campo "*com*" de blocos básicos, a qual é utilizada para a geração das mensagens, quando tarefas são mapeadas para processadores (definição 3.10).

Definição 3.3, *Aplicação*: APP = (*V*,*A*) é um grafo direto, onde cada vértice $v_i \in V$ representa uma tarefa da aplicação (como especificado na definição 3.2) e cada arco $a_{ij} \in A$ representa um caminho pelo qual as tarefas v_i e v_j trocam mensagens.

Aplicações são representadas no modelo de programação sugerido neste trabalho, através de diagramas de classes e de seqüências UML, como pode ser verificado no capítulo 5, seções 5.1 e 5.2.

Uma vez definidas as aplicações alvo, devem ser definidas as arquiteturas para as quais as suas tarefas serão sintetizadas. Tarefas são sintetizadas em componentes de processamento, enquanto que as comunicações entre estas, para componentes de comunicação.

Arquiteturas de comunicação e de processamento podem igualmente serem expressas através de grafos.

Definição 3.4, *Arquitetura de NoCs*: NOC = (R, L) é um grafo direto onde cada vértice $r_i \in$ R representa a arquitetura de um roteador em uma NoC e cada arco $l_{ij} \in$ L representa um canal de comunicação entre os roteadores r_i e r_j. "*l*" é função da topologia da rede. Cada roteador r_i possui associado como propriedades o comportamento dos componentes arquiteturais de roteadores para NoCs:

$r_{i.rout}$: representa a política de roteamento;

$r_{i.sch}$: representa a política de escalonamento ou arbitragem;

$r_{i.buf}$: representa a organização da memória interna (*buffer*);

$r_{i.cf}$: controle de fluxo.

Esses componentes podem ser agregados em uma estrutura maior, que corresponde à arquitetura de roteadores para NoCs.

Definição 3.5, *Arquitetura de Roteadores para NoCs*, (*Router Architecture*, *RA*), é um conjunto de componentes arquiteturais, necessários ao envio de pacotes em redes chaveadas: RA = { $r_{.rout}$ $r_{.sch}$ $r_{.buf}$ $r_{.cf}$}.

A cada um dos componentes estão associados os parâmetros $\{\varepsilon_{rout}, \varepsilon_{sch}, \varepsilon_{buf}, \varepsilon_{cf}\}$; $\{\tau_{rout}, \tau_{sch}, \tau_{buf}, \tau_{cf}\}$ e $\{\alpha_{rout}, \alpha_{sch}, \alpha_{buf}, \alpha_{cf}\}$ os quais retornam respectivamente a energia consumida, o tempo necessário para executar as comunicações da aplicação, além da área de cada componente.

A maneira como roteadores são conectados, por canais de comunicação, forma a topologia da rede.

Definição 3.6, *Arquitetura de Processamento*: PROC = {*type*, *cache*, *bp*, $un_{[n\_units]}$, *pipe*, *ifetch*} é um vetor onde: *type*: tipo do processador (CISC, RISC, VLIW, Superescalar, etc.); *cache*: organização da memória *cache*; *bp*: política para previsão de desvios; $un_{[n\_units]}$: *"n_units"*, número de unidades funcionais; *pipe*; organização do *pipeline* e *ifetch*: tamanho da palavra de busca de instruções. A semântica do vetor PROC representa as principais características de processadores, sendo implementadas algoritmicamente nas classes PC e PO, no nível de abstração correspondente. Cada unidade funcional pode executar microinstruções do conjunto de instruções do processador ou instruções dedicadas. No primeiro caso, diz se que a execução das instruções se dá por software ($un_{[n\_units] = SW}$) e no segundo, por hardware ($un_{[n\_units] = HW}$).

Às unidades funcionais dos processadores estão associados os parâmetros ε, τ e α os quais retornam respectivamente a energia consumida, o tempo necessário para executar as cada tarefa da aplicação, bem como a área de cada unidade; $\{\varepsilon_{cache}, \varepsilon_{bp}, \varepsilon_{un[i]}, \varepsilon_{ifetch}\}$; $\{\tau_{cache}, \tau_{bp}, \tau_{un[i]}, \tau_{ifetch}\}$ e $\{\alpha_{cache}, \alpha_{bp}, \alpha_{un[i]}, \alpha_{ifetch}\}$. Essas informações são utilizadas para a avaliação da arquitetura, frente às restrições de projeto.

Definição 3.7, *Sistema Multiprocessado*: MPSoC = (P,N) é um grafo onde cada nodo $p_i \in$ P (P $\in$ PROC) representa a arquitetura de um processador e cada arco $n_{ij} \in$ N representa a comunicação entre os processadores p_i e p_j.

No capítulo 5 é apresentado um modelo de programação que permite a especificação e modelagem de Arquitetura de NoCs e RAs (seção 5.5), de Arquiteturas de Processamento (seção 5.4) e de Sistemas Multiprocessados (seção 5.6). As modelagens são realizadas através de diagramas de classes e de seqüências UML, utilizando o paradigma de projeto baseado em *interfaces*.

3.2 Otimização arquitetural de Plataformas Arquiteturais

Uma vez que os principais componentes arquiteturais em um SoC estejam definidos e especificados segundo o modelo de programação, bem como a aplicação, é possível definir os problemas relativos à otimização em conjunto, ou separadamente, de cada componente arquitetural.

Primeiramente, é necessário que se definam as operações necessárias à otimização, conseqüência da execução das tarefas da aplicação nos componentes das plataformas de processamento e de comunicação a serem avaliadas. O processo de otimização é decorrente da avaliação da adequação de cada componente arquitetural às restrições de projeto, ao executar as tarefas da aplicação. Para tanto, é necessário que se definam as operações para síntese das tarefas da aplicação em processadores e destes em alguma arquitetura de comunicação, uma vez que um sistema multiprocessado é esperado.

Para o caso de serem utilizadas arquiteturas de comunicação do tipo rede intrachip (NoC), e como já citado no parágrafo acima, a síntese das tarefas para as plataformas de processamento e comunicação se realiza na execução de duas tarefas:

1. particionamento das tarefas da aplicação para um conjunto de processadores; e
2. mapeamento ou posicionamento dos processadores em terminais locais de uma topologia de NoC. Esta etapa, obviamente se torna desnecessária em arquiteturas do tipo barramento.

Definição 3.8, *Particionamento*: dada uma APP, para cada tarefa $v_i \in V$, deve existir um processador correspondente em um sistema multiprocessado, $p_i \in$ P, isto é, existe uma função de particionamento *part*: APP → MPSOC tal que $\forall\ v_i \in V\ \exists\ p_i \in$ P, i = 1,..., np; np = número de processadores. Durante o particionamento cada bloco básico de todas as tarefas é compilado para a execução no conjunto de instruções do processador alvo. Além disso, os cabeçalhos das mensagens a serem enviadas pela rede são gerados, de acordo com a política de roteamento utilizada pela rede alvo.

Definição 3.9, *Caminho*: um caminho $path_{ij} = (r_i, l_{iy}, \ldots, r_w, l_{wj}, r_j)$ é uma seqüência alternada de vértices e arcos de NOC, utilizados para enviar pacotes de mensagens do roteador r_i para o roteador r_j. O caminho é formado de acordo com a estratégia de roteamento implementada para o roteador, como definido em $r_{i.rout}$.

Definição 3.10, *Mapeamento*: dado um sistema multiprocessado MPSOC, para

cada $p_i \in$ P, deve existir um roteador correspondente em uma NoC, $r_i \in$ NOC, isto é, existe uma função de mapeamento *map*: MPSOC → NOC tal que $\forall\ p_i \in$ P $\exists\ path_{ij} \in$ NOC Λ i ≠ j.

Uma vez definidas as operações de particionamento e mapeamento, devem ser definidas as funções custo para cada plataforma. As funções custo servem para avaliar se uma determinada plataforma arquitetural consegue atender as restrições de projeto impostas pela aplicação alvo.

Definição 3.11, Caracterizando-se as restrições para componentes de processamento através do vetor $\kappa\rho$ = { ε_ρ, τ_ρ α_ρ }, onde ε_ρ, τ_ρ e α_ρ correspondem respectivamente às restrições para consumo de energia, área e desempenho para processadores, as *funções custo* podem ser expressas como:

Função custo para *energia*:

$$\kappa\varepsilon\rho = \{\mathbf{min} \sum_{p_i} \varepsilon_{cache[i]} + \varepsilon_{bp[i]} + \varepsilon_{un[i]} + \varepsilon_{ifetch[i]} \}, \text{ tal que } \kappa\varepsilon\rho \leq \varepsilon_\rho \quad (3.1)$$

Função custo para *desempenho*:

$$\kappa\tau\rho = \{\mathbf{min} \sum_{p_i} \tau_{cache[i]} + \tau_{bp[i]} + \tau_{un[i]} + \tau i_{fetch[i]} \}, \text{ tal que } \kappa\tau\rho \leq \tau_\rho \quad (3.2)$$

Função custo para *área*:

$$\kappa\alpha\rho = \{\mathbf{min} \sum_{p_i} \alpha_{cache[i]} + \alpha_{bp[i]} + \alpha_{un[i]} + \alpha_{ifetch[i]} \}, \text{ tal que } \kappa\alpha\rho \leq \alpha_\rho \quad (3.3)$$

Definição 3.12, Caracterizando-se as restrições para componentes de comunicação através do vetor $\kappa\chi$ = { ε_χ, τ_χ α_χ }, onde ε_χ, τ_χ e α_χ correspondem respectivamente às restrições de consumo de energia, área e desempenho para NoCs, as *funções custo* podem ser expressas como:

Função custo para *energia*:

$$\kappa\varepsilon\chi = \{\mathbf{min} \sum_{i} \varepsilon_{rout[i]} + \varepsilon_{sch[i]} + \varepsilon_{buf[i]} + \varepsilon_{cf[i]} \}, \text{ tal que } \kappa\varepsilon\chi \leq \varepsilon_\chi \quad (3.4)$$

Função custo para *desempenho*:

$$\kappa\tau\chi = \{\mathbf{min} \sum_{i} \tau_{rout[i]} + \tau_{sch[i]} + \tau_{buf[i]} + \tau_{cf[i]} \}, \text{ tal que } \kappa\tau\chi \leq \tau\chi \quad (3.5)$$

Função custo para *área*:

$$\kappa\alpha\chi = \{\mathbf{min} \sum_{i} \alpha_{rout[i]} + \alpha_{sch[i]} + \alpha_{buf[i]} + \alpha_{cf[i]} \}, \text{ tal que } \kappa\alpha\chi \leq \alpha_\chi \quad (3.6)$$

O processo de otimização de plataformas consiste então, na configuração de cada componente arquitetural da plataforma alvo, para que estes atendam as restrições da função custo. No entanto, o grau de sucesso de cada componente quanto à otimização depende diretamente das operações de *síntese*, uma vez que, tanto o particionamento, quanto o mapeamento exercem influência direta no desempenho, área e consumo de energia. Por exemplo, testando-se diferentes particionamentos, pode-se chegar a um número maior ou menor de processadores, o que implica em diferentes soluções em área e consumo de energia.

Conclui-se então que, o processo de otimização de componentes arquiteturais para plataformas que visam a implementação de SoCs, constitui-se de uma tarefa extremamente complexa, a qual envolve a execução bem sucedida de várias tarefas como, particionamento, mapeamento e configuração dos componentes arquiteturais de cada plataforma. Cada uma destas tarefas individualmente, possui solução de natureza complexa.

Observando-se as definições 3.8 e 3.10, pode-se verificar que as operações de particionamento e mapeamento admitem um grande número de soluções. Ambas operações pertencem a classe dos problemas NP-Completos.

O mapeamento de um conjunto em outro ($S_0 \rightarrow map(S_1)$) possui complexidade $O(n!)$ - sendo n o tamanho dos conjuntos - uma vez que cada elemento de S_0 pode ser mapeado para qualquer elemento de S_1; o número de soluções cresce fatorialmente com o tamanho dos conjuntos de componentes utilizados pela operação de mapeamento. Se os conjuntos possuírem tamanhos (w) diferentes, a complexidade é expressa por: N!, sendo $N = max \{ w_{(s0)}, w_{(s1)} \}$.

Para o caso de mapeamento de processadores em NoCs com topologia regular a complexidade é determinada pelo número de processadores, os quais são mapeados para terminais locais de NoCs. É assumido que cada roteador da rede possui apenas uma porta local. Por exemplo, para uma rede com dimensão 4x4, 16! combinações possíveis para mapeamento existem, uma vez que cada processador pode ser mapeado para qualquer uma das m linhas e n colunas ($m_x n!$) que compõem a topologia.

Operações com espaço de soluções desse porte podem ser caracterizadas como uma instância do problema *quadratic assignment* que é provado pertencer à classe dos problemas NP - completos [GAR 79]. Isso porque, se cada solução for testada para avaliar a função custo (através de um algoritmo de busca exaustiva), o tempo necessário para se encontrar a solução ótima torna-se proibitivo, podendo a chegar a vários anos. Supondo uma rede 4x4, e que um algoritmo de busca exaustiva necessite 1 *ms* para avaliar cada mapeamento (o que envolve simular o comportamento total de comunicação da aplicação, para obter o desempenho, por exemplo), seriam necessários aproximadamente 20.922.789.888 segundos para a execução total do algoritmo, ou 672 anos. Este tempo considera o uso de simulação para a avaliação das arquiteturas. A abordagem adotada neste trabalho utiliza simulação para a captura de comportamentos dinâmicos, como por exemplo, contenções em redes intrachip e em *pipelines*.

A operação de particionamento também possui natureza NP - Completo como demonstrado em [NAV 99]. O número de soluções para a operação de particionamento pode ser calculado pelo número de Bell: $Bn = \sum_{k=1}^{n} \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \frac{k!}{i!(k-i)!} i^n$, sendo n igual ao número de objetos (número de tarefas, para a aplicação do problema como aqui definido) a serem particionados.

Assim, a complexidade do particionamento é da ordem de $O(B_n)$. No entanto, essa complexidade é manifestada para o caso de tarefas poderem ser particionadas para apenas uma arquitetura de processador. Caso mais de um processador possa ser utilizado para o particionamento de tarefas (o que é plausível de se considerar em projetos baseados em plataformas) a complexidade aumenta ainda mais. Isso porque o número de possíveis partições deve ser considerado para cada processador disponível, gerando uma complexidade da ordem de $O(\rho! \cdot B_n)$, sendo ρ o número de diferentes processadores.

Finalmente, se as operações de particionamento e mapeamento forem consideradas em conjunto, para cada possível solução de particionamento, devem ser tratadas todas as soluções de mapeamento, o que gera uma complexidade da ordem de: $O((\rho! \cdot B_n) \cdot N!)$, sendo ρ o número de diferentes processadores e N, o número de processadores e roteadores de NoCs.

As operações de particionamento e mapeamento podem ser consideradas em conjunto para o caso em que se deseje particionar as tarefas de uma aplicação em processadores, e ao mesmo tempo, verificar a influencia da arquitetura de comunicação para as restrições do projeto.

A busca por soluções otimizadas para problemas NP - Completos requer o emprego de uma heurística, a qual procura reduzir o espaço de busca, considerando o comportamento específico e a natureza dos componentes a serem otimizados.

Por momento, para este trabalho foi adotada a heurística conhecida como "*Busca Tabu*", uma vez que essa abordagem vem sendo empregada com sucesso em vários problemas de otimização semelhantes, como constatado em [GLO 90].

A figura 3.2 ilustra o pseudo-código para o algoritmo Busca Tabu.

A estratégia geral do algoritmo Busca Tabu consiste na exploração de todos os possíveis movimentos em um conjunto de *n* "recursos", *S* = (1,2,…,*n*). Cada movimento corresponde à permutação entre dois recursos no vetor *S*, o que leva a uma nova solução (*y*) para o problema, resultado da aplicação de uma função custo (*k*) aos recursos. A cada nova solução é dado o nome de "vizinhança". A função OPTIMUM retorna a melhor solução (s(y)) encontrada entre *x* vizinhanças. A cada *x* vizinhanças uma nova iteração é gerada. O número de iterações (*nt*) executadas pelo algoritmo é determinado pelo usuário.

Os movimentos de cada iteração que levam às melhores soluções são colocados em uma lista "tabu" (T), o que indica que estes ficam proibidos de serem re-utilizados na criação de uma nova vizinhança; essa estratégia é adotada para que sejam evitados "mínimos locais".

```
Tabu_Search(resources S)  {
   select an initial solution:
       y ∈ Y e y* = y; k= 0; T = ∅
2  if (S(y)–T) == ∅
       go to 4;
   else
       t = t + 1;
   select best st = OPTIMUM(s(y): s ∈ S(y)-T);
   y = st;
   if k(y) < k(y*)  // y* → best solution
      y* = y;
   if t > nt
4     stop;
   else
      update T;
   go to 2;
}
```

Figura 3.2: Pseudo-Código para o Algoritmo "Busca Tabu"

O algoritmo Busca Tabu estabelece que a regra adotada para a escolha dos recursos a serem movimentados pode ser estabelecida por quem o utiliza, o que torna possível a customização do comportamento da heurística de acordo com as particularidades do problema que se queira resolver.

Dessa forma é possível à utilização do algoritmo busca tabu para a exploração do espaço de soluções para as operações de particionamento e mapeamento. Para tanto, basta adaptar a semântica do conjunto de "recursos" *S* ao problema que se queira resolver. Para o caso do particionamento, o conjunto *S* representa as tarefas da aplicação; ($S_{task[i]} = task_{[i]}$, $i = 1,..., n$; n: número de tarefas da aplicação; $task \in T$). Para o caso do mapeamento, *S* representa o conjunto de processadores disponíveis na plataforma alvo; ($S_{pl[i]} = p_{[i]}$, $i = 1,..., np$; $p \in PROC$).

Além de encontrar soluções otimizadas para as o particionamento e o mapeamento, o algoritmo de busca tabu também pode ser utilizado para a exploração do comportamento interno de cada componente arquitetural das plataformas de processamento e comunicação. Nesse caso, duas estratégias podem ser adotadas: otimização independente dos componentes arquiteturais ou em conjunto com as operações de particionamento e/ou mapeamento.

Para a execução da primeira estratégia, basta tornar o conjunto de recursos *S* igual à semântica do componente a ser observado. Por exemplo, se forem observadas políticas de escalonamento em árbitros para NoCs, *S* deve ser igual a política de escalonamento para cada árbitro da rede: $S_{sch[i]} = r_{i.sch}$; i = 1, ...,*nra*; nra: número de roteadores da rede.

A execução da segunda estratégia requer a definição de mais de um recurso simultaneamente. Para tanto, o algoritmo busca tabu deve ser adaptado para lidar com mais de um recurso. Por exemplo, o mapeamento pode ser abordado, ao mesmo tempo em que políticas de roteamento são consideradas para os diferentes roteadores em NoCs; ou diferentes políticas de previsão de desvios podem ser abordadas ao mesmo tempo que tarefas são particionadas para processadores. Salienta-se, no entanto, que todos esses casos trazem um aumento para a complexidade do algoritmo de busca, através do aumentando do espaço de soluções.

3.3 Conclusões

Neste capítulo foram analisadas possíveis maneiras de se descreverem aplicações alvo e componentes de plataformas arquiteturais. Esses *modelos de descrição* serão utilizados com base para a definição de um modelo de programação (objeto do capítulo 5), a ser utilizado para a descrição de componentes, de modo a que estes possam ser automaticamente avaliados por ferramentas. Além disso, também foram tratadas nesse capítulo as operações envolvidas no processo de otimização arquitetural e a complexidade associada.

A correta e precisa definição dessa classe de operações é de fundamental importância para que, a partir delas, se possam derivar métodos e ferramentas de apoio ao projeto necessárias à sua correta execução, o que deve levar a um sistema otimizado.

No próximo capítulo um método para exploração e otimização arquitetural é proposto, o qual procura reunir em um fluxo de projeto as ferramentas necessárias à correta execução das operações aqui discutidas.

No entanto, para que aplicações, componentes e operações para a otimização arquitetural possam ser especificadas e executadas por ferramentas de apoio ao projeto, é necessária também a definição de um modelo de programação que reflita de maneira clara e coerente os comportamentos que se deseja otimizados. Um modelo de programação que caminha nessa direção é detalhado no capítulo 5.

4 ESPECIFICAÇÃO DE UM MÉTODO PARA OTIMIZAR PLATAFORMAS ARQUITETURAIS

No capítulo 2 foram discutidas ferramentas e abordagens para a concepção de sistemas eletrônicos, as quais vem sendo consideradas por diversas equipes de pesquisa e por empresas. O presente capítulo visa explicitar as idéias desenvolvidas para o método aqui proposto, no sentido do desenvolvimento de uma abordagem que aponte ferramentas necessárias à concepção e otimização de plataformas arquiteturais. Considerando a concepção de sistemas dedicados completos como um processo de avaliação de componentes arquiteturais disponíveis em plataformas, o método procura colaborar para o fortalecimento dos conceitos e ferramentas que levem a uma solução efetiva na busca por arquiteturas otimizadas.

4.1 O Problema encontrado para a Otimização de Plataformas

O processo para a concepção dos modernos sistemas eletrônicos constitui-se complexo devido a fatores como o grande número de componentes arquiteturais disponíveis em plataformas, e o decorrente grande número de variáveis a serem otimizadas nos componentes. Além disso, é esperado que as aplicações alvo sejam compostas por várias tarefas, o que induz à modelagem e/ou execução concorrente destas em componentes de processamento. As tarefas podem ainda, apresentar comportamentos heterogêneos, o que demanda o emprego de mais de um modelo de computação para a especificação e modelagem das aplicações alvo.

A otimização de plataformas arquiteturais é realizada através de operações como particionamento e mapeamento, as quais (como discutido no capítulo anterior) possuem natureza complexa.

A complexidade do projeto de sistemas dedicados completos dificulta a que uma equipe de engenheiros e desenvolvedores consiga especificar e avaliar sistemas em tempo hábil, sem a ajuda de ferramentas para automatizar e/ou auxiliar em tarefas pertinentes ao projeto. Além do emprego de ferramentas, é necessário também que sejam reutilizadas “propriedades intelectuais” de diversas equipes ou empresas, para que um espectro representativo de componentes possam ser considerados e avaliados para o projeto, sem que sejam desenvolvidos por apenas uma equipe.

O uso de ferramentas por si só, pode não conduzir à solução por uma arquitetura otimizada, se estas não estiverem contextualizadas em um método que defina um fluxo de projeto para estas. O fluxo de projeto serve para determinar a semântica para as ferramentas, organizá-las e relacioná-las a fim de que conduzam a uma solução otimizada.

As operações diagnosticadas no fluxo de projeto devem idealmente levar à criação de um caminho efetivo, cuja direção aponta para a real possibilidade da criação de sistemas que correspondam às aspirações da sociedade.

Assim sendo, para ajudar o projetista de sistemas no desenvolvimento desse processo, diversos métodos, linguagens e ferramentas foram e estão sendo propostos. No entanto, pelo fato da área de projeto de sistemas dedicados constituir-se de uma área de pesquisa multidisciplinar e em certos aspectos recente, muito ainda deve e pode ser feito para chegar a soluções completas e confiáveis.

O método proposto nesse trabalho encaminha-se nessa direção, no sentido de que sua concepção visa somar-se aos esforços que vem sendo realizados na busca de melhores soluções para a implementação eficiente de sistemas eletrônicos completos. Acredita-se que este é o caminho natural para a evolução da tecnologia, uma vez que todo o processo de criação acaba por manifestar-se como um processo de realimentação positiva [SUS 2001].

4.2 Objetivos

Como discutido no parágrafo anterior, o processo de concepção de sistemas completos, comumentemente chamados “Sistemas-em-Chip” (*Systems-on-Chip*, *SoCs*), constitui-se por uma tarefa de projeto complexa, dado o grande espaço de projeto a ser investigado. Como conseqüência, são necessários métodos e ferramentas para auxiliar projetistas de sistemas a lidar com o grande número de componentes a serem avaliados para a implementação da arquitetura final para a aplicação alvo.

O caminho natural pelo qual o ser humano compreende e manipula a complexidade é criado pelo uso de um processo de abstração. Abstrair refere-se à capacidade humana de considerar somente as informações necessárias para a compreensão de conceitos.

A propriedade da abstração pode ser explorada por projetistas para o projeto de sistemas, para que estes considerem somente as variáveis arquiteturais “pertinentes” em cada estágio de desenvolvimento.

A abstração é válida (e necessária) mesmo quando a determinação do que “realmente” é pertinente em dado momento, seja difícil de ser determinado, uma vez que devido à abstração, o ser humano consegue compreender mais facilmente as propriedades do sistema em desenvolvimento.

Outra maneira para conseguir trabalhar a complexidade dá-se pela delegação de tarefas. O fato de livrar-se de determinadas tarefas, dá ao ser humano a liberdade para concentrar-se em pontos mais específicos. O uso de ferramentas para automatização de tarefas vem de encontro a esse objetivo.

No capítulo 2 foi constatada a necessidade do uso de ferramentas para a concepção de sistemas eletrônicos e de estas trabalharem em diferentes níveis de abstração. Atualmente existem diversas ferramentas, tanto para especificação e síntese para componentes de processamento, quanto para componentes de comunicação.

No entanto, dentro das abordagens utilizadas para a concepção dos sistemas dedicados, constata-se que existem ainda diversas tarefas que merecem maior consideração para que se chegar a uma maior consolidação dessas abordagens. Este trabalho toma a direção no sentido do projeto baseado em plataformas, pelos motivos citados acima, e também, por esta abordagem possuir atualmente, o aval da comunidade científica e da indústria. O projeto baseado em plataformas está consolidado, caminhando para tornar-se padrão em projeto de sistemas complexos.

Dentro do contexto do projeto baseado em plataformas, acredita-se necessária a definição de métodos para contemplar os seus conceitos, determinando um fluxo de

projeto que compreenda todas as ferramentas de apoio ao projeto necessárias, suas relações e os níveis de abstração a serem considerados.

Para a efetivação do projeto baseado em plataformas, verifica-se que atualmente existem poucas ferramentas para avaliar componentes arquiteturais, a fim de verificar o grau de otimização que estes podem possuir em relação às restrições de projeto de aplicações. Portanto, tanto a escolha, quanto a configuração dos componentes arquiteturais das plataformas, são determinados de maneira *ad-hoc*, onde prevalece a experiência do projetista. Portanto, um dos principais objetivos dessa abordagem será a exploração de componentes de plataformas arquiteturais. Esta tarefa terá a finalidade de verificar a adequação dos componentes na composição da plataforma, para que esta corresponda às restrições de projeto de aplicações.

Outra questão importante que surge nesse contexto é a necessidade de se considerar tanto os componentes de processamento, quanto de comunicação. Aplicações complexas tendem a implementar diversas tarefas, sintetizadas para vários componentes de processamento. Nesse contexto, a comunicação exerce um papel importante, visto que esta influencia diretamente restrições de projeto, como desempenho e área, por exemplo.

Nos dias atuais, poucas ferramentas comerciais são voltadas para o uso de NoCs quando da definição de plataformas. Normalmente ferramentas criadas para a definição de plataformas, configuram arquiteturas de comunicação do tipo barramento. Isso se deve em parte, pelo fato de que as aplicações atuais não requerem uma largura de banda maior do que os barramentos atuais conseguem oferecer. Como pode ser verificado no capítulo 2, ferramentas acadêmicas para a avaliação de NoCs existem. No entanto, as ferramentas aqui propostas diferenciam-se por considerarem diversos aspectos dessas redes, bem como o seu impacto frente a arquiteturas de processamento.

Existem soluções em termos de arquiteturas de barramentos muito eficientes [ARM 2004], [IBM 2002] implementando inclusive, soluções hierárquicas. Nesse tipo de solução, o barramento principal e rápido é utilizado para conectar os dispositivos que requerem maiores larguras de banda, enquanto que um ou dois barramentos mais lentos são conectados a este através de pontes. Aos barramentos secundários são conectados os periféricos que não requerem muito desempenho.

No entanto, este trabalho visa o uso de NoCs por acreditar que as futuras aplicações serão amplamente beneficiadas com o uso dessa classe de arquitetura de comunicação, como demonstrado em [ZEF 2002].

Assim sendo, pode-se definir que o maior objetivo desse trabalho consiste na definição de um método baseado no conceito do projeto baseado em plataformas, o qual se dedica à exploração do espaço de busca em projetos de sistemas complexos. No método, são concebidas as ferramentas de apoio ao projeto que se julgarem necessárias à otimização de componentes arquiteturais em plataformas.

O fluxo de projeto proposto pelo método sedimentará as regras para a definição de novas ferramentas, concebidas e evoluídas em num processo de realimentação positiva.

4.3 Fluxo de Projeto proposto para o Método

A figura 4.1 mostra um fluxo de projeto genérico, cujas ferramentas são comumentemente utilizadas para a síntese de sistemas completos.

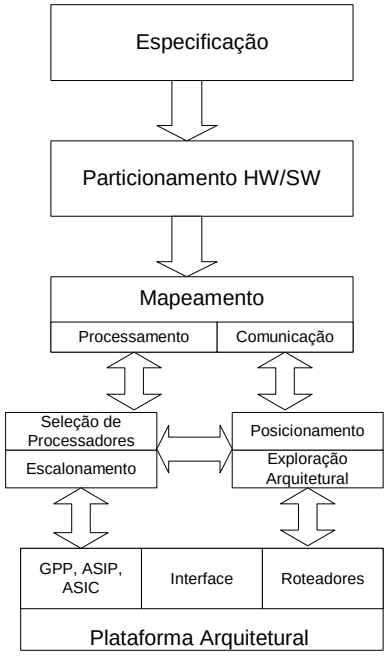


Figura 4.1: Fluxo de Projeto para a Concepção de Sistemas Completos

Primeiramente, uma aplicação é modelada em alto nível, por uma linguagem de programação, para em seguida ter as suas tarefas particionadas entre SW e HW. O particionamento entre tarefas programáveis e não programáveis pode ser realizado manualmente ou com a ajuda de ferramentas para particionamento automático, como por exemplo, *Cosyma* [THI 99].

Uma vez particionadas, as tarefas são *mapeadas* para componentes de processamento ou comunicação. Tarefas implementadas como HW são mapeadas em componentes arquiteturais com conjunto de instruções dedicados e implementadas como ASICs. O mapeamento considera a ortogonalidade presente entre os comportamentos de processamento e de comunicação.

As setas bidirecionais mostradas a partir do mapeamento indicam que este processo pode ocorrer em diferentes níveis de abstração. Isto se deve a possibilidade, prevista para o projeto baseado em plataformas, de que a aplicação possa ser analisada sob diferentes níveis de abstração.

As tarefas implementadas como SW podem ser sintetizadas para processadores com conjunto de instruções de propósito geral - GPPs - ou específico - ASIPs.

De acordo com o método, para a realização das otimizações das plataformas arquiteturais, devem ser concebidos:

- Os níveis de abstração sobre os quais as plataformas terão os seus componentes especificados;
- Um modelo de programação para a modelagem do comportamento heterogêneo dos componentes arquiteturais, nos diferentes níveis de abstração;
- Uma ferramenta para a síntese e composição do comportamento da aplicação nos níveis hierárquicos considerados para os componentes em plataformas
- A delimitação do espaço de projeto – variáveis dos componentes a serem analisados – e estratégias para explorá-las; e

- Ferramentas para analisar comportamentos de componentes em plataformas e para configurá-los com fins à otimização..

A figura 4.2 mostra o fluxo de projeto do método proposto, onde são destacadas as ferramentas necessárias à realização dos conceitos estabelecidos para o mesmo.

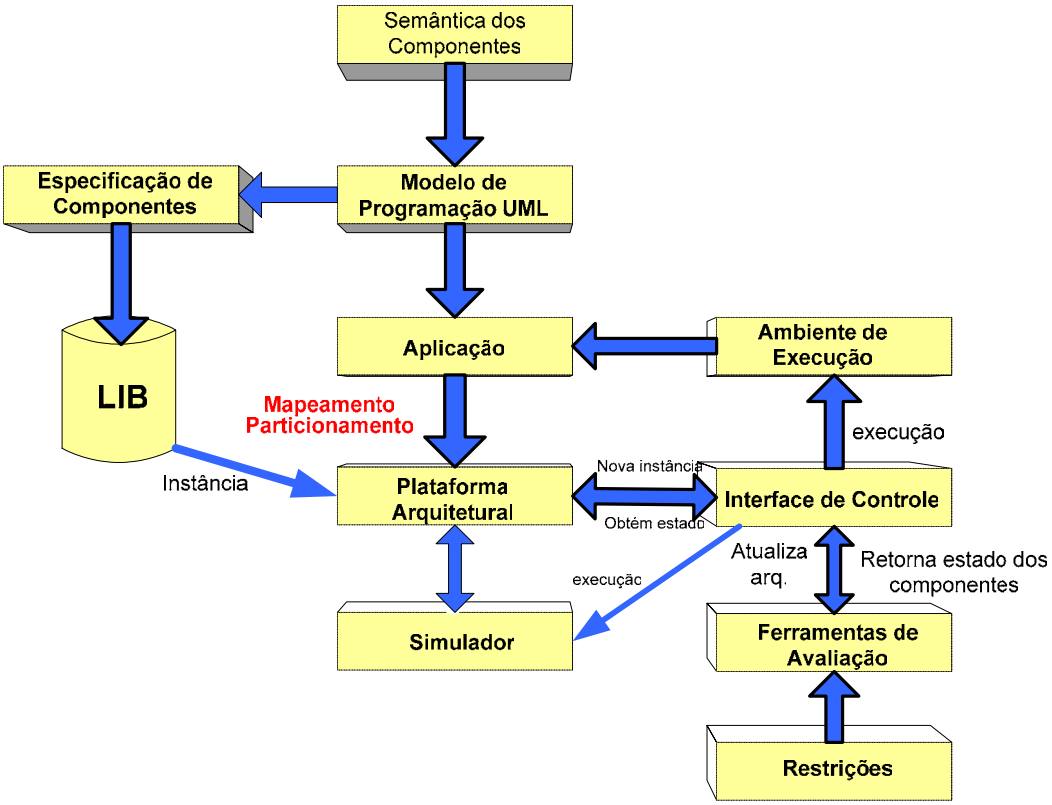


Figura 4.2: Fluxo de Projeto do Método para Otimização de Plataformas Arquiteturais

Pode-se verificar na figura 4.2 que, primeiramente, descrições de componentes arquiteturais devem ser disponibilizadas para a composição de plataformas – avaliadas pelas ferramentas de otimização. Plataformas são descritas UML, de acordo com as regras e os níveis de abstração estabelecidos por um modelo de computação, como descrito no capitulo 4.

Componentes arquiteturais de plataformas são armazenados em bibliotecas de componentes, a qual os disponibiliza para que sejam utilizados na criação SoCs otimizados. Aplicações podem ser definidas concorrentemente com as descrições das plataformas.

As granularidades das funções de aplicações são definidas pelo programador da aplicação, o que define as comunicações entre as tarefas. Assim, programadores têm consciência da concorrência entre as tarefas das aplicações. Aplicações são especificadas com base em um modelo de programação, o qual especifica as semântica de componentes arquiteturais para comunicação e processamento, utilizando a linguagem UML.

Seguindo-se o fluxo de projeto, pode-se verificar que descrições de aplicações são mapeadas e particionadas para plataformas, uma vez que o objetivo maior do método proposto é a otimização dos componentes arquiteturas de sistemas completos.

O processo de particionamento recebe como entrada um conjunto de tarefas e sintetiza-as para os componentes arquiteturais de processamento, enquanto que o processo de mapeamento posiciona os componentes arquiteturais de processamento em arquiteturas de comunicação do tipo NoCs, da plataforma a ser avaliada. Se arquiteturas de comunicação do tipo barramento forem utilizadas, o processo de mapeamento não necessita ser executado.

Para a avaliação de plataformas é utilizada uma abordagem de simulação. Para tanto, deve ser especificado um simulador específico para plataformas de comunicação e de processamento.

Plataformas são avaliadas com base nas restrições de projeto estabelecidas para a aplicação alvo. Avaliações são realizadas por ferramentas dedicadas as quais interagem com as descrições arquiteturais por meio de uma interface de controle (parte integrante do modelo de programação). Essa interface permite o acesso aos componentes, tanto para ativar as suas funcionalidades, quanto para reconfigurá-las.

A figura 4.3 mostra um diagrama UML onde aparecem relacionadas às ferramentas necessárias à implementação do fluxo de projeto para exploração do espaço de projeto, acima descrito.

A figura 4.3 revela algumas considerações importantes, adotadas para a efetivação do fluxo de projeto do método. Primeiramente, destaca-se o uso do *metanível* – o que implica em reflexão computacional – para as ferramentas de apoio ao projeto (representadas pela classe *evaluation*) acessarem as plataformas. Isso pode ser verificado na *metaclasse* “*control”*.

A reflexão computacional é empregada com dois objetivos: isolar o projeto das funcionalidades das ferramentas das funcionalidades dos componentes arquiteturais das plataformas; e permitir que os componentes das plataformas sejam avaliados durante o projeto e durante o tempo de execução.

O primeiro objetivo visa permitir que ferramentas sejam projetadas para a avaliação de qualquer plataforma e que componentes possam ser avaliados por qualquer ferramenta compatível com o seu comportamento. Para a efetivação do primeiro objetivo, é necessária a definição de uma interface, que exponha às ferramentas de apoio ao projeto, quais as funções e variáveis que podem ser otimizadas/reconfiguradas nos componentes.

É nesse momento que se faz presente o conceito da interface de controle para os processos de otimização do projeto: a interface de controle (representada pela classe “control”) permite que as ferramentas, executando no metanível, possam acessar as funcionalidades e o estado atual dos componentes, a fim de avaliá-los e otimizá-los. Isso é realizado através do conceito de projeto baseado em interfaces, onde as funcionalidades e variáveis que podem ser acessadas e/ou alteradas por outras classes, são expostas em classe denominada interface. Nesta classe não são implementadas efetivamente as funcionalidades, sendo estas apenas exteriorizadas pelas assinaturas dos métodos da classe.

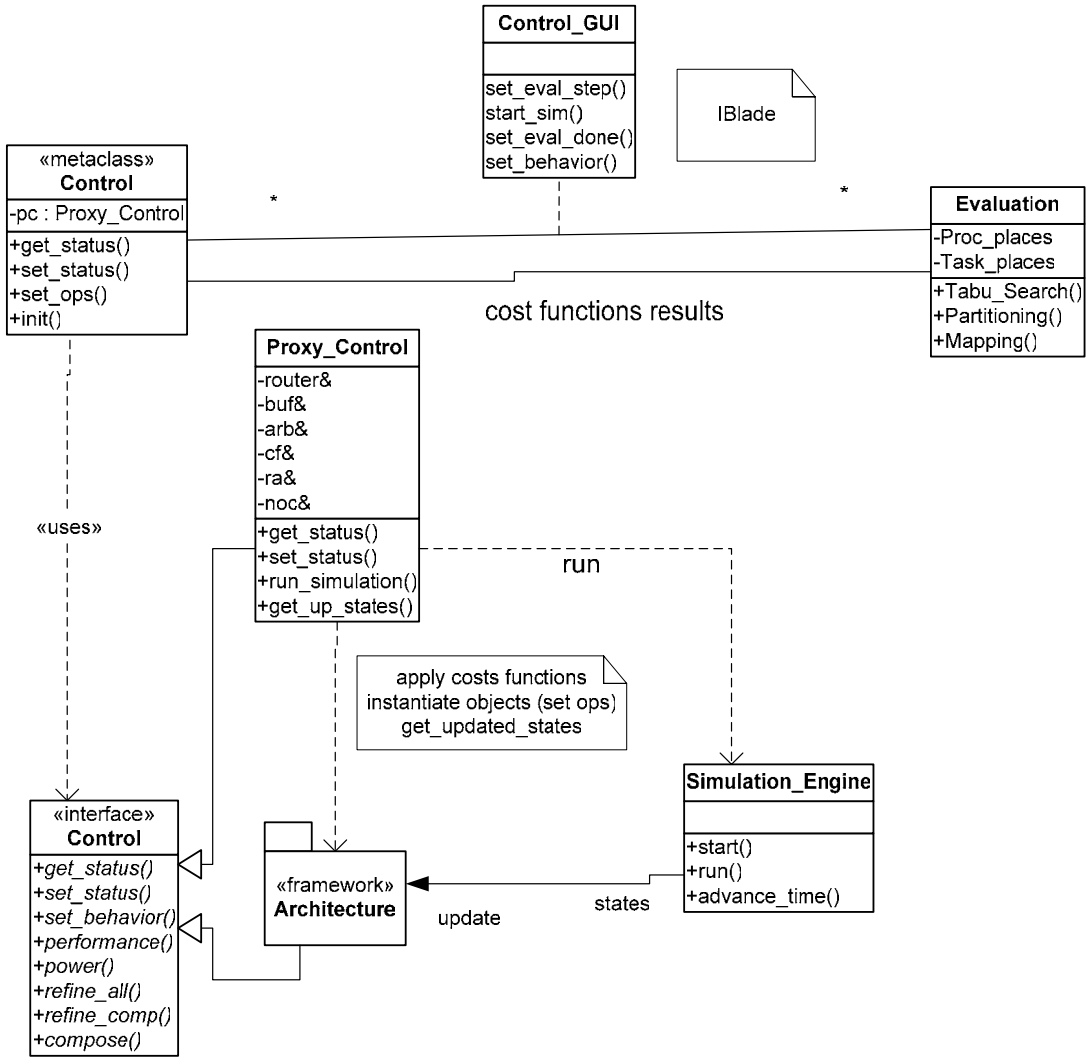


Figura 4.3: Representação UML do Fluxo de Projeto do Método

Em relação ao segundo objetivo, destaca-se que certas funcionalidades somente podem ser avaliadas durante o tempo de execução, observando o estado atual do componente. Por exemplo, em NoCs, políticas de escalonamento somente podem ser avaliadas observando-se o estado atual das contenções da rede. Por este motivo, foi adotada a abordagem de simulação para o método, representada pela classe "*simulation engine*" na figura 4.3. Os simuladores implementados são compatíveis com o modelo de programação utilizado na modelagem dos componentes, escalonando cada componente segundo o seu modelo de computação.

A implementação da reflexão computacional é realizada através do padrão de projeto "Proxy" (classe *proxy control*), o qual permite às ferramentas acessarem os componentes das plataformas.

Finalmente, pode ser verificado que as plataformas são implementadas como *frameworks* orientado a objeto. Frameworks são estruturas de dados extensíveis, que apontam uma ou mais soluções para uma mesma classe de aplicações e permitem que novas funcionalidades sejam agregadas às soluções existentes. Para o caso das plataformas arquiteturais, as soluções iniciais correspondem à arquitetura de uma

plataforma, a qual admite reconfigurações para as funcionalidades de seus componentes, bem como para as relações entre esses.

4.4 Ferramentas de Apoio ao Projeto Consideradas para o Método

As ferramentas de apoio ao projeto, como definidas no fluxo de projeto do método, cooperam para que sistemas completos possam ser modelados e analisados, com fins à otimização, sob diferentes níveis de abstração. Para tanto, são consideradas ferramentas para a modelagem, síntese/composição funcional e simulação.

A modelagem segue os princípios do modelo de programação baseado em UML, como definido no capítulo 5. O modelo de programação estabelece os níveis de abstração e as funcionalidades passíveis de serem modeladas em cada nível, para cada componente arquitetural, bem como as relações entre estes.

Nesse ponto, novamente o conceito de interfaces é utilizado, com dois objetivos: permitir a *inter-relação* entre os componentes e a especificação de diferentes *tipos* de componentes. No capítulo 5 esse conceito é novamente retomado, onde é explicada a opção por interfaces para a implementação desses objetivos. Para diferenciar da interface de controle (citada nos parágrafos anteriores), essa é chamada de interface operacional.

Para a modelagem de plataformas segundo o modelo de programação, qualquer ferramenta compatível com UML pode ser utilizada, uma vez que o uso de estereótipos é opcional, servindo apenas para tornar o modelo mais compreensível. Isso quer dizer que códigos especiais não são gerados a partir de estereótipos. Por exemplo, o protocolo de comunicação utilizado na comunicação entre componentes é implementado na interface operacional entre esses. No entanto, o nome do protocolo pode também ser explicitado nas classes que definem esses componentes como um estereótipo de comunicação.

Uma vez modelados os componentes para plataformas, estes podem analisados e se necessário, reconfigurados, a fim de que atendam as restrições de projeto da aplicação alvo. Para tanto, são utilizadas ferramentas para analisar o comportamento de cada componente da plataforma considerada. Caso algum componente não atenda a alguma restrição de projeto esse pode ter a sua arquitetura reconfigurada. Análises são realizadas considerando-se o comportamento de cada componente durante a execução da aplicação.

A execução das ferramentas no metanível, habilita-as a analisar e avaliar o estado atual dos componentes, durante a execução da aplicação. Nesse caso, as ferramentas interagem com os componentes através da interface de controle, como citado na seção anterior. Para executar a aplicação é utilizada uma abordagem de simulação.

Para tanto, um simulador dedicado deve ser definido e implementado, devendo interagir com as descrições dos componentes. A ordem de execução dos componentes é definida pelo modelo de computação no qual estes foram modelados. No entanto, como comportamentos heterogêneos podem existir, mais de um MoC pode estar presente no mesmo modelo. Nesse caso, o simulador deve considerar um passo de simulação com base em um MoC e normalizar os demais passos de simulação referentes aos outros MoCs por este. Por exemplo, se um modelo baseado em relógio é utilizado em conjunto com um modelo que não considera ordens temporais, então o simulador pode escalonar todos os componentes de acordo com a freqüência do relógio, tornando o passo de simulação igual a essa freqüência. Os componentes regidos pelo MoC atemporal,

mesmo escalonados, somente executam, caso exista algum token em uma de suas entradas, como definido em suas interfaces.

Finalmente, como as análises podem ser realizadas em diferentes níveis de abstração, deve se provida uma ferramenta para sintetizar ou compor descrições em diferentes níveis hierárquicos. Essa ferramenta – de mapeamento – indica ao simulador o modelo de descrição dos componentes a ser executada.

4.5 Conclusões

Nesse capítulo foram apresentadas as idéias principais referentes à concepção e implementação de um método para a otimização de componentes arquiteturais. O método reside nos conceitos do projeto baseado em plataformas, define um modelo de programação baseado na linguagem UML e estabelece as ferramentas de apoio ao projeto as quais se acredita serem necessárias para a análise e otimização dos componentes de plataformas arquiteturais. Plataformas são implementadas segundo o conceito de frameworks orientado a objetos, o que habilita que estas sejam reconfiguradas e expandidas para atender a requisitos de várias aplicações.

A abordagem do método considera um sistema completo, com plataformas de processamento e comunicação.

Espera-se que as idéias presentes no método possam efetiva e sistematicamente colaborar para que sistemas completos possam ser otimizados e concebidos para a execução de aplicações dedicadas heterogêneas.

No próximo capítulo é definido o modelo de programação pelo qual plataformas podem ser especificadas.

5 MODELO DE PROGRAMAÇÃO PARA ESPECIFICAÇÃO DE APLICAÇÕES DEDICADAS E PLATAFORMAS ARQUITETURAIS HETEROGÊNEAS

O processo de concepção e otimização de sistemas complexos requer ferramentas capazes de analisar, simular e configurar componentes em plataformas arquiteturais. No entanto, para que estas tarefas possam ser realizadas com sucesso é necessário que plataformas arquiteturais sejam especificadas ou modeladas de forma a que ferramentas possam automaticamente reconhecer a semântica de seus componentes e de suas inter-relações. Isso é imperativo em sistemas complexos, devido ao grande número de componentes a serem analisados. Somente com o auxílio de ferramentas de apoio ao projeto é possível identificar e otimizar as funcionalidades esperadas para aplicações alvo para sistemas dedicados. A semântica expressa o significado do comportamento dos componentes, determinando as operações para transformação e transferência (comunicação) de dados presentes em cada componente. O conhecimento da semântica proporciona que granularidades sejam determinadas, bem como a sua realização física, como software ou hardware.

Formalmente, a semântica pode ser expressa através de Modelos de Computação (*Models of Computation, MoCs*) [LEE 2002], os quais determinam os *tipos* de componentes e das *comunicações* entre estes. Segundo Edward Lee [LEE 2002], MoCs "podem ser pensados com as leis da física que governam a interação entre componentes; é o modelo de programação ou *framework*, sobre o qual aplicações são geradas através da composição de componentes". Dessa forma, aplicações podem ser expressas através de conjuntos de componentes e de suas relações, sendo a heterogeneidade comportamental expressa através dos diferentes tipos de operações (de transformação e comunicação) implementadas para os componentes. O modelo de programação proposto nesse capítulo especifica como a sintaxe da linguagem UML pode ser utilizada para a especificação de MoCs, a fim de que estes sejam utilizados na especificação comportamental de aplicações dedicadas e de plataformas arquiteturais heterogêneas. Nesse contexto, a heterogeneidade funcional pode ser entendida como comportamentos implementados por mais de um MoC. Isso implica em que, em um mesmo modelo, diferentes tipos de objetos podem coexistir e comunicarem-se sob diferentes protocolos de comunicação. No modelo de programação sugerido, MoCs são especificados através de *interfaces*, como comentado no capítulo anterior e explicado na seção 5.2 a seguir.

Modelos de computação consideram que componentes são expressos sob uma dada granularidade a qual delimita as suas funcionalidades e as suas relações com outros componentes da aplicação. Componentes podem ser *estados* e as relações entre estes, *transições* entre estados; por outro lado, componentes podem ser *processos* e as relações, *trocas* de mensagens. Modelos de computação especificam, dentre outras

características do comportamento, a concorrência, a passagem do tempo e o sincronismo para as comunicações entre componentes.

Em sistemas complexos, espera-se encontrar mais de uma semântica em uma mesma especificação, o que determina comportamentos heterogêneos; em um mesmo modelo, pode haver diferentes tipos de componentes, estes podem considerar a passagem do tempo de forma diferenciada, podendo comunicar-se por diferentes protocolos.

Uma vez que aplicações com comportamentos heterogêneos são esperadas para sistemas complexos, deve-se esperar um suporte arquitetural para as plataformas nas quais estas são sintetizadas, o que implica na realização de plataformas arquiteturais heterogêneas. Estas são compostas por componentes de diferentes tipos de operações para transformação de dados e de para comunicação. Para tanto, é necessário que se definam as funções específicas implementadas por cada componente, a sua granularidade, a semânticas das relações que mantêm com os demais componentes da plataforma, a concorrência e a passagem do tempo. Além disso, a implementação do comportamento de cada tarefa da aplicação poderá ser realizada por componentes de SW ou de HW.

A base para a efetivação de uma modelagem, tanto para aplicações, quanto para plataformas arquiteturais heterogêneas, deve ser centrada em um modelo de programação que defina as regras para a modelagem dos componentes e de suas relações. Como citado acima, para o caso de sistemas complexos, o modelo de programação deve prever ainda, a possibilidade de que sejam especificados comportamentos heterogêneos. Por exemplo, podem-se esperar sistemas para processamento digital de sinais, onde comunicações são geradas a intervalos constantes e enviadas sequencialmente de um componente para o outro, caracterizando um MoC do tipo *dataflow* ou *Process Network* [LEE 2002]. No entanto, junto com essas comunicações, sinais de controle podem estar sendo enviados entre os componentes, utilizando diferentes protocolos de comunicação. Um caso típico pode ser encontrado em aparelhos para telefonia celular, onde o processamento de sinal deve ser operado sobre sinais de radiofreqüência, ao mesmo tempo comportando diferentes funcionalidades, como controles para o teclado e o *display*, por exemplo.

Uma das características primordiais do comportamento de sistemas complexos manifesta-se sob a forma de concorrência; vários componentes que compõem o sistema podem executar concorrentemente, o que leva a uma especificação onde o paralelismo é explícito. Além disso, a passagem do tempo deve ser levada em consideração, quando MoCs que consideram o tempo, estão presentes. Portanto, para o desenvolvimento de software para sistemas dedicados é interessante que o projetista de sistemas tenha a consciência do comportamento dos componentes modelados. Isso pode ser conseguido se o modelo de programação permitir a modelagem explícita dos MoCs utilizados para a especificação dos componentes da aplicação. Isso é conseguido explicitando-se no modelo as características que definem a natureza de cada MoC empregado na especificação das funcionalidades da aplicação.

Como exemplo de modelo de programação para SoCs, pode-se citar a abordagem considerada para o ambiente Ptolemy [LEE 2003]. Nesse ambiente, é adotado o paradigma da Orientação a Objetos para a especificação dos componentes na linguagem Java [SUN 2005]. A especificação dos objetos é realizada através de uma linguagem gráfica – Vergil - onde componentes (chamados de “atores”) são instanciados e conectados. Cada grupo de componentes é simulado segundo um modelo de

computação, implementado por um “diretor”. O diretor escalona os componentes de acordo com a semântica determinada pelo MoC que controla. Múltiplos MoCs são permitidos em um único modelo, implementados hierarquicamente: grupos de componentes simulados por um único diretor são agrupados - formando um único objeto - o qual expõe uma interface (da composição) que pode relacionar-se com outros MoCs.

Outra alternativa para a implementação de sistemas heterogêneos refere-se ao projeto Metropolis [BAL 2003]. Nesse ambiente, um meta-modelo (baseado na linguagem Java) é definido sobre o qual ferramentas para simulação, síntese e verificação são derivadas. Aplicações são descritas através do conceito de “redes”: cada rede refere-se à composição de componentes e de suas inter-relações. A semântica de execução (especificada no meta-modelo) determina o comportamento da rede. O meta-modelo não especifica nenhum MoC explicitamente, uma vez que são disponibilizados blocos necessários à especificação de comportamentos de processamento e comunicação. Com o uso desses blocos, diversos MoCs podem ser definidos.

O emprego da orientação a objetos torna-se especialmente interessante, uma vez que provê a modularização de componentes, ou seja, a sua natureza é concorrente. Através do uso da orientação a objetos, os componentes e suas relações são explicitamente modelados, o que sugere um sistema distribuído, como o esperado para sistemas complexos. Além disso, é possível especificar-se a concorrência através de uma linguagem de alto nível qualquer, desde que seja compatível com a orientação a objetos.

Outra alternativa para a especificação concorrente seria com o emprego de uma linguagem específica, cuja sintaxe explicitasse a granularidade dos componentes e suas relações.

Assim como a abordagem adotada para os ambientes Ptolemy e Metropolis, o modelo de programação aqui sugerido, também é implementado segundo os princípios da orientação a objetos, pois se acredita que o uso de uma linguagem de propósito geral tende a ser melhor aceito pela comunidade de projetistas de sistemas, uma vez que não induz projetistas ao aprendizado de uma nova linguagem e sintaxe associada. Além disso, o uso da orientação a objetos e de linguagens compatíveis como C++ ou Java, por exemplo, encontra-se nos dias atuais disseminados o suficiente para que sejam amplamente aceitas.

O modelo de programação aqui sugerido possui como características próprias, o fato de diferentes modelos de computação poderem ser especificados no mesmo nível hierárquico, através da interface de controle e da maneira como é implementado o controle do simulador (ver seção 6.4).

5.1 Forma Canônica do Modelo de Programação

Para a modelagem dos componentes optou-se pela linguagem UML (*Unified Modeling Language*) [UML 2005] uma vez que esta se constitui padrão nos dias atuais, sendo amplamente aceita pelas comunidades cientificas. Além disso, existem diversos grupos de trabalho [OMG 2005] dedicados à sua avaliação, atualização e aperfeiçoamento, inclusive com propostas ao uso para sistemas dedicados. Como exemplo, pode-se citar o perfil UML para aplicações tempo-real, o qual define estereótipos específicos para atender às semânticas temporais na execução de aplicações. Outra abordagem interessante chamada de “embedded UML” [MAR 2001] propõe um perfil especifico para sistemas embarcados. Nesse trabalho são propostas

semânticas a serem adicionadas à linguagem UML relativas às operações para o co-projeto HW/SW, bem como para a especificação de concorrência e níveis de abstração para aplicações dedicadas.

Existem também ferramentas comerciais dedicadas à especificação e avaliação de sistemas dedicados, utilizando a linguagem UML, como por exemplo, a ferramenta Mentor, “Nucleus Modeling” [MEN 2005], ou Artisan, “Real-Time Studio” [ART 2005].

O modelo de programação aqui proposto está centrado em três diagramas UML:

• *Classes*, para a modelagem estática da concorrência e comunicação entre componentes;

• *Colaboração*, modela estaticamente os relacionamentos e a concorrência entre as instâncias das classes; e

• *Seqüências*, para a modelagem do tempo e da ordem das comunicações entre os componentes.

A esses diagramas são associados *estereótipos*, que servem para identificar no modelo a semântica da comunicação (diagramas de classes e de colaboração) e o tempo de cada comunicação (diagramas de seqüências).

Nas figuras 5.1 a 5.3 é mostrada a forma canônica do modelo de programação baseado em UML, aqui proposto. Nessas figuras podem ser observados os elementos de modelagem empregados em cada diagrama para a modelagem do comportamento de aplicações heterogêneas.

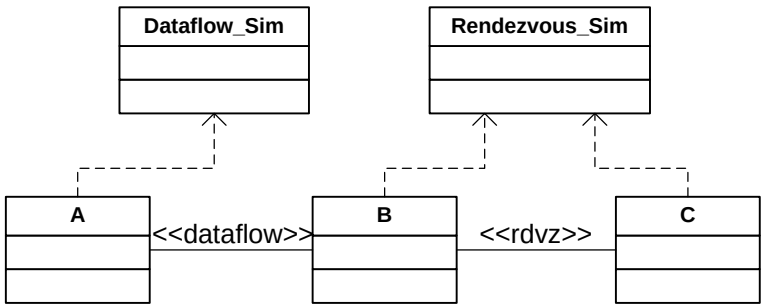


Figura 5.1: Forma Canônica para Diagrama de Classes UML

Na figura 5.1, três classes “A”, “B” e “C” comunicam-se segundo dois MoCs: A para B através de um protocolo de comunicação *dataflow*, e objetos B para C comunicam-se através do protocolo *rendezvous*. Em uma comunicação *dataflow* [LEE 2002], conexões representam um fluxo de dados entre duas classes, sendo uma o produtor e outra o consumidor; à medida que dados são disponibilizados na entrada do consumidor este provê um esquema de memória que é utilizada para o armazenamento dos dados. Questões como limites da memória e deadlock são tratados em um caso especial do dataflow, chamado *Synchronous Dataflow* (SDF) [LEE 2002].

Já no caso da comunicação Rendezvous [LEE 2002], a comunicação entre as classes é síncrona; se, no momento em que o componente produtor deseja enviar uma mensagem, o consumidor não estiver pronto, o produtor fica bloqueado até que o consumidor esteja pronto. Uma maneira de se implementar esse tipo de comunicação é através da implementação de um relógio único para ambos. De uma maneira geral pode-se dizer que a comunicação ocorre em um único passo.

Observa-se também na figura 5.1 a implementação dos protocolos expostos no modelo: isso é realizado através da implementação de classes especificas para a simulação de MoCs. Essas classes são subordinadas à estrutura de dados do simulador dedicado à simulação de modelos, construídos segundo o modelo de programação aqui proposto. A sugestão de tal simulador dedicado é exposta no capitulo 6. As classes de simulação para MoCs são utilizadas para implementar a semântica para as variáveis que definem o MoC a ser simulado. Essas variáveis são definidas abaixo.

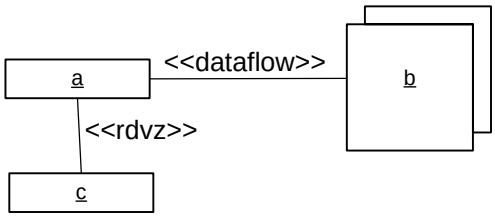


Figura 5.2: Forma Canônica para Diagrama de Colaborações

Na figura 5.2 percebe-se a concorrência entre os objetos (*b*) manifestada na multiplicidade desse grupo de objetos: a entidade “b” corresponde a um grupo de objetos, enquanto que “a” e “c” correspondem à apenas um objeto. Durante a execução/simulação da aplicação, objetos “a” deverão possuir um identificador que corresponda a um objeto do *conjunto* de objetos “b”. Além disso, pode se verificar o estereótipo “dataflow” utilizado para tornar claro o modelo empregado para as comunicações entre esses objetos.

Outra questão a ser observada em tais diagramas refere-se à ordem de execução das tarefas da aplicações. Em diagramas de colaboração entre objetos é priorizada a observação da relação entre estes e não a ordem nas quais as funcionalidades são executadas.

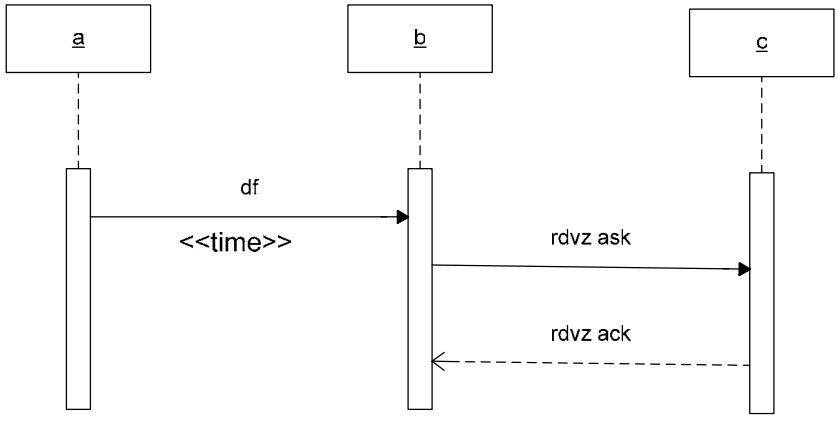


Figura 5.3: Forma Canônica para Diagrama de Seqüências

Por outro lado, na figura 5.3 observa-se a ênfase na ordem temporal de execução das comunicações e das funcionalidades dos objetos. A execução de cada método é ativada através de uma chamada de funções entre objetos. Em cada comunicação pode-se especificar o tempo de execução previsto para o método chamado através do estereótipo “time”. Através da ordem de execução temporal pode-se perceber por exemplo, a resposta de uma solicitação “rendezvous” por exemplo.

5.2 Modelagem de Modelos de Computação

O modelo aqui proposto é centrado no principio de que os objetos são ativos, ou seja, possuem *thread* própria de execução. Com isso, durante a execução da aplicação, todos os componentes executam concorrentemente, podendo gerar mensagens por conta própria. A execução dos métodos dos componentes ocorre através de chamadas internas ou de chamadas remotas. Chamadas internas ocorrem quando um método de um objeto chama outro método do mesmo objeto. Por outro lado, chamadas remotas ocorrem entre objetos, segundo um protocolo de comunicação cuja semântica é determinada pelo MoC do componente.

Os MoCs são definidos através das seguintes variáveis:

- Concorrência: todos componentes executam concorrentemente;
- Sincronismo: pela comunicação entre os componentes, determinada pelos protocolos de comunicação;
- Ordem das operações: ordem na qual as funcionalidades de cada componente executam, o que é determinado pela comunicação; e
- Passagem do tempo: tempo do passo de simulação.

Para que mais de um MoC possa ser especificado em um mesmo modelo, a especificação dos componentes deve permitir variações semânticas para essas variáveis. Isso implica em uma linguagem que possa expressar essas variáveis, além de um ambiente de execução/simulação que reconheça a semântica de cada uma. Como já discutido, a especificação aqui proposta é baseada na OO através da linguagem UML. Por tratar-se uma linguagem nativa para OO e por ser gráfica, a UML através de seus diagramas, pode ser utilizada para a expressão de MoCs.

Como visto nos parágrafos acima, a forma canônica do modelo de programação prevê primordialmente, três tipos de diagramas, os quais expressam MoCs. A concorrência é expressa em todos eles, como classes ou objetos. O sincronismo é expresso como estereótipo nas relações entre classes e de objetos nos respectivos diagramas e nas chamadas de métodos no diagrama de seqüências. A ordem das operações e a passagem do tempo são explicitadas no diagrama de seqüências pela ordem temporal de execução dos objetos. Deve-se ressaltar no entanto, que se diferentes MoCs utilizam diferentes unidades mínimas de tempo (*dataflow* e tempo contínuo, por exemplo) mais de um passo de simulação deve ser empregado pelo simulador da aplicação; passos com unidades de tempo distintas. Em MoCs onde não há a representação do tempo, os passos de simulação servem para executar as funcionalidades dos componentes na ordem correta.

Para que os componentes possam saber os relacionamentos que podem manter com os demais, a modelagem é realizada segundo os princípios da modelagem baseada em *interfaces* [ROW 2003]. Modelagens que seguem os princípios de interfaces apresentam vantagens como:

- Clara identificação de cada componente – a granularidade em cada nível de abstração;
- As relações entre os componentes são modeladas de maneira que seja possível determinar todas as dependências de dados, a concorrência e o sincronismo entre os componentes;

• As operações internas de cada componente podem ser reconfiguradas sem que isso afete as relações com os demais componentes;

• Os componentes podem ser desenvolvidos concorrentemente; e

• Ferramentas são capazes de automaticamente detectar o que acontece quando um componente é retirado ou substituído, mantendo o sistema em um estado seguro e semanticamente coerente.

Através desse paradigma de modelagem, cada componente expõe uma interface nas qual são explicitados os métodos que deve implementar como funcionalidade interna, além dos métodos de que dispõe para comunicação. Os demais componentes da aplicação necessitam apenas saber quais são esses métodos e a semântica de comunicação que implementam. Caso a semântica não seja compatível, tradutores devem ser inseridos.

A especificação de um componente é realizada através de herança com uma interface previamente definida, como ilustra a figura 5.4.

Os métodos são especificados em uma interface como *virtuais*, o que implica em que não são efetivamente implementados, apenas definidos para serem implementados na classe filha da *interface*. Pelo mesmo motivo as classes que implementam *interfaces* são também chamadas de “abstratas”.

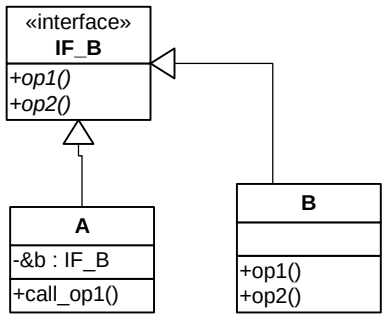


Figura 5.4: Projeto baseado em *Interfaces*

No exemplo mostrado na figura 5.4 o método *op1()* é utilizado para a realização de uma operação de comunicação entre os componentes *A* e *B*. Através da *interface*, cada vez que um componente de tipo *A* for implementado, é possível saber que este componente possui uma relação de tipo “op1” com componentes do tipo *B*. Para tanto, uma *referência* a um componente do tipo *B* é instanciada (&b) com o tipo da interface de *B* - IF_B – e a operação “op1()” executada a partir dessa instância, na função “call_op1()” – *b->op1()*. A operação “op2()” corresponde a uma funcionalidade interna de componentes de tipo *B*.

Em relação às funções específicas de cada componente, o projeto baseado em interfaces possibilita que o comportamento interno de cada função possa ser especificado de diferentes maneiras, sem que isso altere a sua relação com os demais componentes. A figura 5.5 ilustra essa situação. Através da interface, diferentes versões de op1() podem ser implementadas, sem que isso altere a maneira como outros objetos realizam operações com essa função.

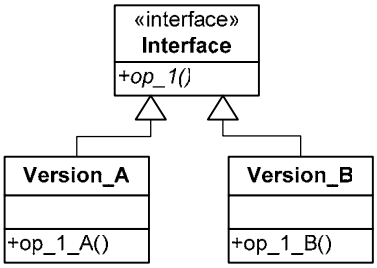


Figura 5.5: Diferentes versões funcionais definidas através de “projeto baseado em interfaces”

Os princípios de modelagem aqui propostos podem ser utilizados para a programação de qualquer aplicação heterogênea. No entanto, aqui objetiva-se utilizá-los para a especificação de plataformas arquiteturais heterogênea, o que incorre na especificação de componentes do tipo arquiteturais. São modelados os componentes tipicamente encontrados em plataformas de processamento e de comunicação, além de suas relações. Assim, diversas plataformas podem ser especificadas a fim de serem avaliadas e otimizadas por ferramentas de apoio ao projeto.

A especificação de plataformas arquiteturais envolve a determinação da granularidade utilizada na especificação de cada componente. Devido à complexidade relativa ao grande numero de componentes esperados para sistemas complexos, torna-se necessário a utilização do artifício da abstração para que o projetista de sistemas e as ferramentas de apoio ao projeto possam tratar com um grande numero de componentes em tempo hábil, reduzindo o risco de erros, como já discutido na capitulo 2.

Constatada essa necessidade, as descrições das plataformas devem ocorrer em mais de um nível de abstração. Para o presente modelo de programação foram definidos quatro níveis de abstração, sendo criados modelos nesses níveis para componentes arquiteturais de processamento e comunicação.

5.3 Níveis de Abstração Considerados no Modelo de Programação

Devido ao enorme espaço de projeto determinado pela complexidade das plataformas, a exploração arquitetural a nível RT revela-se excessivamente cheia de detalhes e demasiadamente longa para ser realizada. Ao longo das últimas décadas a evolução da microeletrônica, através de processos de fabricação cada vez mais sofisticados, levou à necessidade de se trabalhar em níveis de abstrações cada vez mais elevados, quando do projeto de sistemas eletrônicos.

Primeiramente, transistores tiveram que ser abstraídos por portas lógicas, que por sua vez, anos mais tarde, foram abstraídas por células. Já na década de 90 sentiu-se a necessidade de se trabalhar no nível RT o que demandou a proliferação de linguagens para descrição de HW, como VHDL ou Verilog. Nesse nível, os componentes de arquiteturais são caracterizados por blocos funcionais, como registradores ou operadores lógico/aritméticos.

Apesar de nos dias atuais essas linguagens ainda exercerem uma grande importância no projeto, com o advento do projeto baseado em plataformas, existe a necessidade de se elevar mais uma vez o nível de abstração para conjuntos de operadores arquiteturais que compõem uma solução – programável ou não – para a execução de uma determinada lógica composta por um conjunto de operações. Exemplos para esse tipo de componentes são processadores e roteadores de rede.

No entanto, a exigência por níveis de abstração mais altos não invalida as análises realizadas nos níveis inferiores, pois esses possibilitam uma maior precisão, devido à riqueza de detalhes – maior variedade de componentes a serem analisados. Como conseqüência, o projeto de circuitos eletrônicos complexos, envolve alguma abordagem *bottom-up* ou *top-down*, onde detalhes são acrescidos à medida que forem necessários, o que resulta num compromisso entre tempo de projeto e precisão na análise.

Tendo por base essa situação, para o método aqui proposto, são considerados quatro níveis de abstração para os componentes de SW arquiteturais:

1. *sistema*, onde a aplicação é especificada por alguma linguagem de alto nível, através de componentes de SW;

2. *mensagens*, primeiro dos níveis para componentes arquiteturais. Nesse nível a granularidade corresponde a unidades funcionais completas, como conjuntos de instruções em componentes de processamento e topologias de redes chaveadas. As instruções são especificadas através do caminho de dados e de todos os componentes da parte operativa necessários à sua execução. As topologias por sua vez, por arquiteturas de roteadores. Nesse nível, componentes de SW podem especificar funções relativas a Sistemas Operacionais;

3. *transações*, onde são especificados algoritmicamente os componentes arquiteturais elementares de componentes de processamento e de comunicação, como ULAs, roteadores de redes chaveadas, bancos de registradores, etc. Os componentes de SW nesse nível podem interagir diretamente com o comportamento dos componentes arquiteturais, dos quais dependem para ativar as suas funções. Isto permite a especificação de SW dependente de HW, como *drivers*; e

4. *transferência entre registradores*, os mesmos componentes do nível de transações, com precisão à nível de relógio.

É interessante notar que diferentes níveis de abstração estabelecem um nível hierárquico entre os componentes. A implementação da hierarquia de componentes é modelada através das relações entre os componentes em cada nível de abstração: quanto mais baixo navega-se pela hierarquia, maior é o número de componentes observados, decorrente da menor granularidade.

A modelagem da hierarquia é definida pelo modelo de programação da seguinte forma: quando componentes são agregados compondo um novo componente, um novo nível na hierarquia é estabelecido. O componente agregador pode implementar a semântica correspondente à execução das funções e relações dos componentes agregados através de código em linguagem de alto nível, em um método.

5.4 Modelos para Componentes Arquiteturais de Processamento

As figuras 5.6 a 5.8 mostram os modelos para componentes de processamento, para os níveis de abstração considerados.

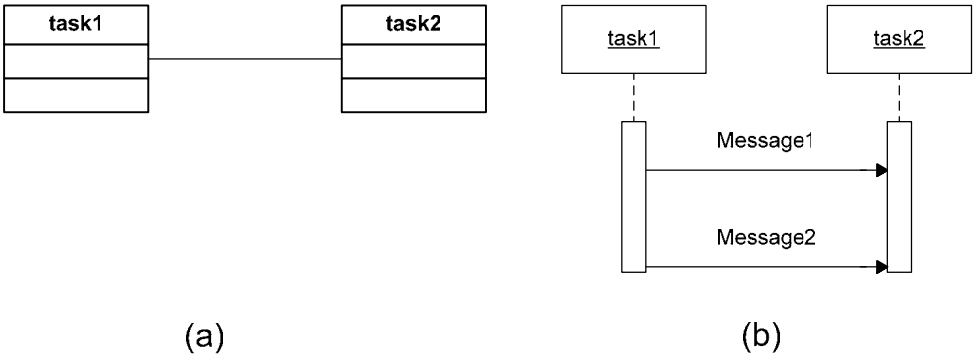


Figura 5.6: Diagramas de Classes (a) e Seqüências (b) para Componentes de Processamento no Nível de *Sistema*

No nível de sistema, a aplicação é vista como um conjunto de tarefas concorrentes que se comunicam por trocas de mensagens. As tarefas podem ser todas executadas em um único processador ou serem distribuídas entre até *n* processadores, sendo *n* o número de processadores disponíveis para a plataforma considerada. Tarefas mapeadas para o mesmo processador comunicam-se através de uma memória local ou do banco de registradores do processador, enquanto que tarefas mapeadas em processadores diferentes comunicam-se através de uma arquitetura de comunicação do tipo NoCs ou barramentos. Nos dois casos, a modelagem de comunicação é realizada através da interface dos componentes, como mencionado na seção 5.2.

No nível de sistema é possível observar-se e concorrência entre as tarefas, bem como o sincronismo das comunicações entre os componentes.

No nível de mensagens, observam-se as partes, operativa e de controle de componentes de processamento. A definição das duas principais entidades encontradas em processadores é realizada através de suas respectivas interfaces, o que permite a definição de variações funcionais, mantendo-se o mesmo protocolo de comunicação entre elas. Dessa forma, a mudança de alguma funcionalidade na PO - por exemplo, uma operação na ULA - não afeta a maneira como os relacionamentos ocorrem entre esta e a PC.

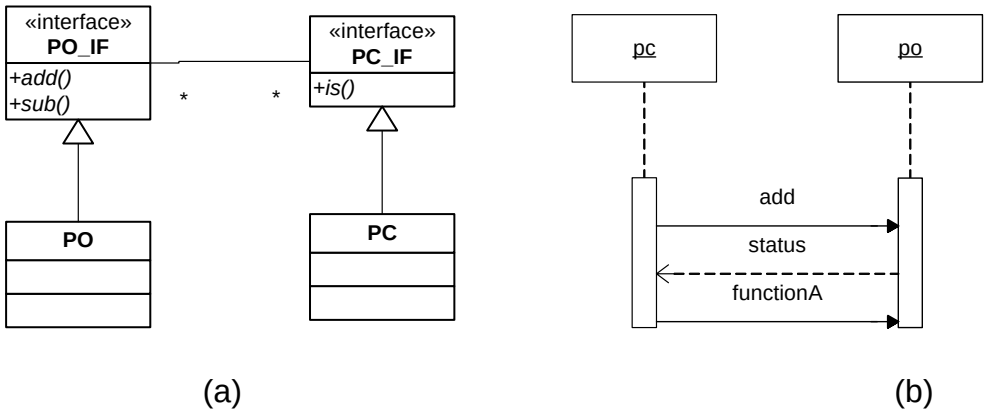


Figura 5.7: Diagramas de Classes (a) e Seqüências (b) para Componentes de Processamento no Nível de *Mensagens*

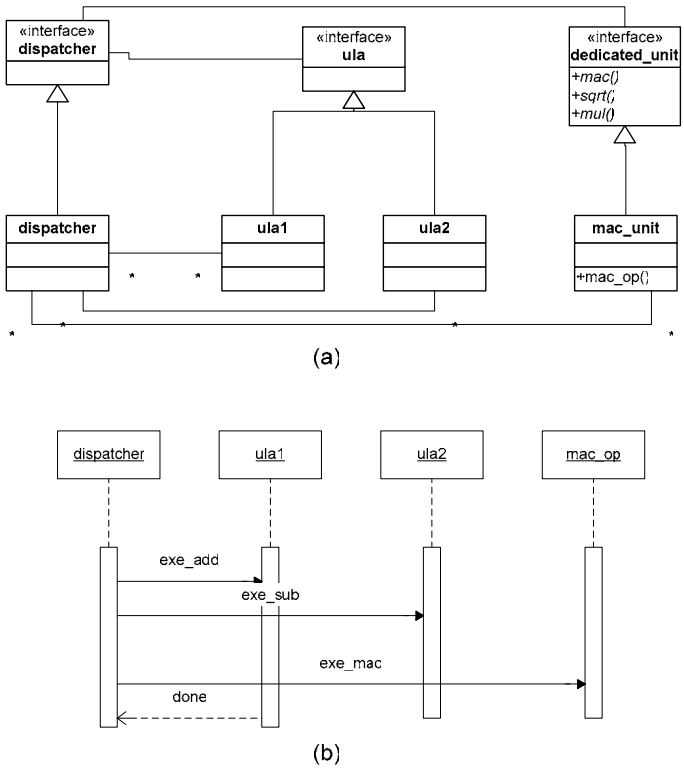


Figura 5.8: Diagramas de Classes (a) e Seqüências (b) para Componentes de Processamento no Nível de *Transações*

A granularidade corresponde a um conjunto de instruções, sendo as operações internas de cada instrução implementadas algoritmicamente na PO. Uma função completa também pode ser executada, como uma instrução dedicada em uma unidade especifica. Essa situação ocorre quando processadores reconfiguráveis (ASIPs) são considerados durante a avaliação de arquiteturas de processamento. Por exemplo, a um processador pode ser adicionada uma instrução específica para processamento de sinais a sua PO e ao seu conjunto de instruções. Nesse nível de abstração observam-se as relações que existem entre a PC e a PO.

No nível de transações a granularidade dos componentes corresponde às unidades funcionais da PO, como ULA, banco de registradores e aos componentes da parte de controle, como contador de programa e registrador de instruções. No diagrama de classes observa-se o relacionamento entre os componentes arquiteturais internos da PO e PC, enquanto que no diagrama de seqüências, a ordem de execução dos componentes envolvidos na execução de cada instrução do conjunto de instruções do processador.

Na interface da ULA se pode observar também as funções previstas para interfaces na modelagem de aplicações segundo o modelo de programação proposto: a especificação de diferentes tipos de ULAs – com operações específicas, por exemplo – e a comunicação com os demais componentes da PO.

5.5 Modelos para Componentes Arquiteturais de Comunicação

As figuras 5.9 a 5.11 mostram os modelos para componentes de comunicação, para os níveis de abstração considerados.

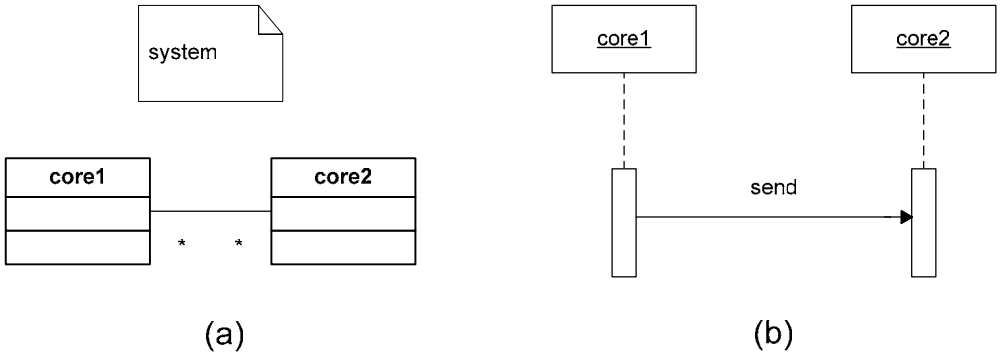


Figura 5.9: Diagramas de Classes (a) e Seqüências (b) para Componentes de Comunicação no Nível de *Sistema*

No nível de sistema a comunicação é vista como elementos de processamento (cores) comunicantes. A comunicação ocorre através de uma biblioteca de comunicação. Dessa forma, através da modelagem da comunicação, a concorrência é explicitada. A biblioteca de comunicação provê os protocolos necessários para a especificação de diferentes MoCs. Durante a simulação, o simulador também utiliza as primitivas de comunicação presentes na biblioteca para simular protocolos de comunicação específicos para cada MoC. Na figura 5.1 é mostrado um exemplo onde simuladores são utilizados para trocas de mensagens segundo os Modelos de Computação *dataflow* e *rendezvous*. No diagrama de seqüências observa-se a ordem das mensagens.

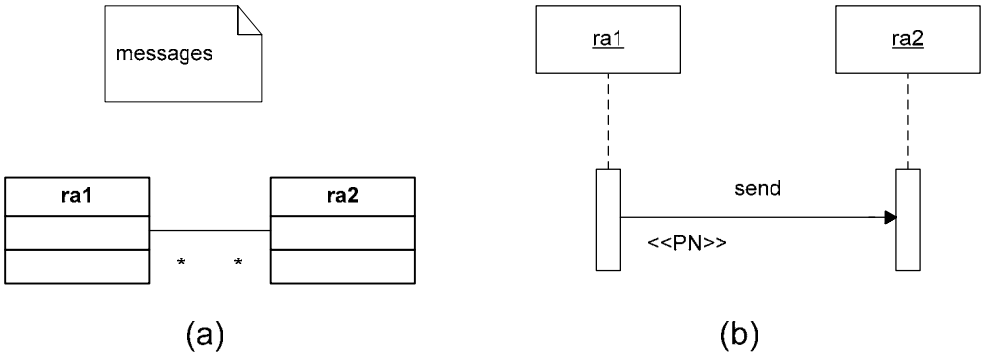


Figura 5.10: Diagramas de Classes (a) e Seqüências (b) para Componentes de Comunicação no Nível de *Mensagens*

No nível de mensagens a granularidade corresponde aos roteadores arquiteturais (RAs) de NoCs. Tanto no diagrama de classes quanto de seqüências, observa-se a concorrência e as comunicações entre os componentes.

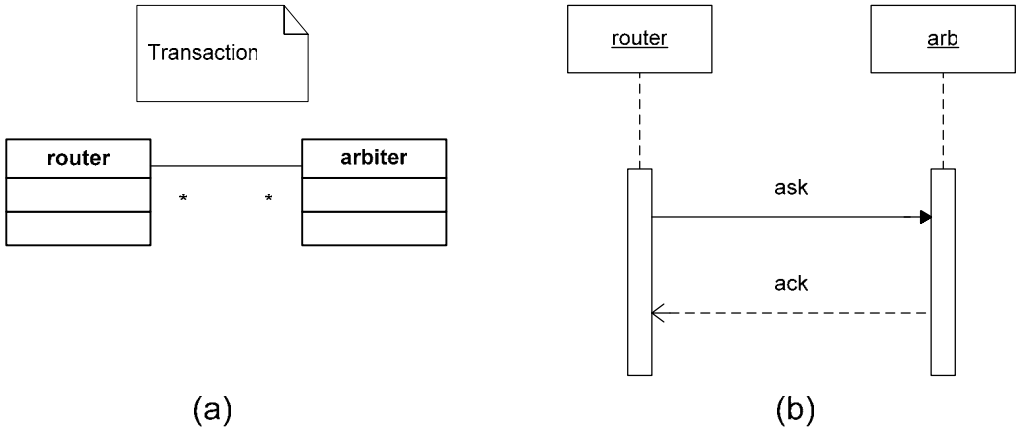


Figura 5.11: Diagramas de Classes (a) e Seqüências (b) para Componentes de Comunicação no Nível de *Transações*

Finalmente, no nível de transações, os componentes correspondem à estrutura interna de RAs em NoCs, como por exemplo roteadores para árbitros, quando da solicitação de requisição para o envio de pacotes.

5.6 Modelos para Sistemas Completos

A modelagem de um sistema completo requer que um único modelo comporte componentes de processamento e comunicação, bem como a relação entre eles. Uma vez que a especificação dessas duas classes de componentes tenha sido determinada – como visto nas seções anteriores – basta que seja reunido num único modelo as especificações para processamento e comunicação através de um mecanismo de comunicação. A maneira como as conexões entre essas duas classes de componentes podem ser realizadas depende do nível de abstração considerado para o modelo.

No nível de sistema, a comunicação é realizada através de uma biblioteca de comunicação, a qual deve prover primitivas para o envio e recebimento de mensagens, segundo diferentes protocolos. O modelo exposto pela figura 5.6 expressa comunicação em um sistema completo, desde que as tarefas sejam mapeadas para diferentes processadores. O mesmo ocorre na figura 5.9.

Já para os níveis de abstração inferiores, a comunicação entre processadores e arquiteturas de comunicação poderá ser implementada tanto através da PO quanto da PC, através de instruções específicas para o envio e recebimento de mensagens, criadas para processadores.

No primeiro caso, uma unidade especifica para a execução dessa instrução é implementada na PO. No segundo caso, a PC pode – ao serem decodificadas essas instruções – gerar sinais de controle que estabelecem endereços de registradores ou de memória, cujos conteúdos são enviados por uma porta de saída do processador - a que está conectada com a arquitetura de comunicação – ou para receber dados de uma porta de entrada ou de alguma entrada do sistema de interrupções. Em ambos os casos, a porta de saída do processador deve ser conectada a uma porta local de uma NoC ou a um terminal de um barramento.

No lado da arquitetura de comunicação, se for uma NoC, a porta local para envio de pacotes é conectada ao sistema de interrupção do processador; caso for um barramento, uma de suas saídas é conectada ao mesmo sistema.

As instruções para envio e recebimento de mensagens são utilizadas pelas bibliotecas de comunicação para implementação dos protocolos de comunicação para os MoCs. A figura 5.12 mostra um exemplo de um processador conectado a uma NoC através da PO.

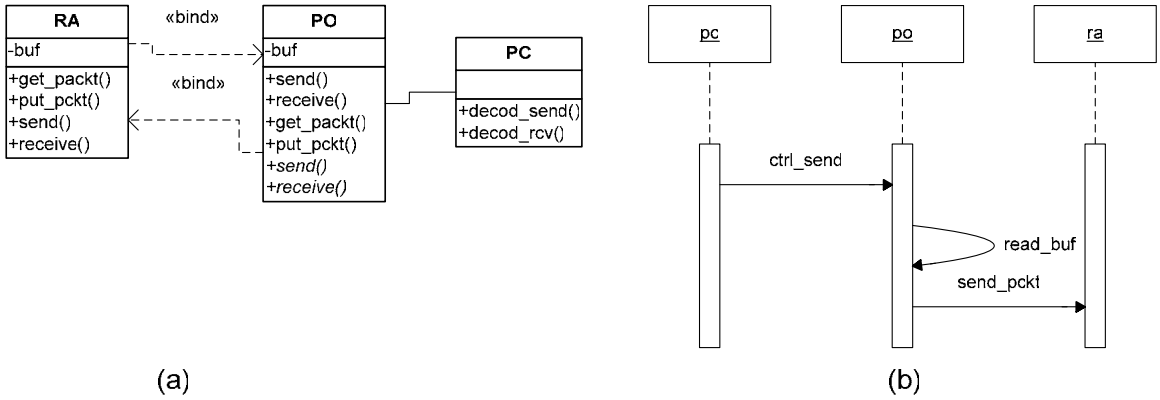


Figura 5.12: Conexão de Processadores à Arquiteturas de Comunicação do tipo Network-on-Chip

Na figura 5.12(a) a conexão é mostrada em um diagrama de classes UML. Nesse diagrama pode ser observada a decodificação da PC, que envia os sinais de controle para que a PO possa comunicar-se com um roteador arquitetural de uma NoC. O envio dos sinais de controle da PC para a PO pode ser percebido na figura 5.12(b). Uma vez que a PO executa uma instrução de envio de mensagens, esta lê o conteúdo de um buffer (memória ou registradores) e os envia para a porta local do roteador no qual o processador está conectado.

Uma vez que as conexões demonstradas na figura 5.12 estão representadas no nível de abstração de mensagens, a unidade específica de comunicação dentro da PO, não esta referenciada, assim como a unidade responsável pelo mecanismo de interrupção.

5.7 Síntese e Composição de Componentes Arquiteturais

Uma vez que diferentes níveis de abstração foram caracterizados e modelos para comunicação e processamento especificados nesses níveis, é necessário que sejam implementadas as tarefas para síntese e composição de componentes. A especificação de componentes em diferentes níveis de abstração induz a formação de agregados de componentes, para que as funcionalidades decorrentes das relações entre os objetos agregados possam ser abstraídas, como discutido no capitulo 2. Como conseqüência, diferentes níveis hierárquicos entre componentes são formados. Quando componentes são agregados para compor um único objeto (num nível de abstração superior), a funcionalidade correspondente à execução dos objetos agregados e suas inter-relações é abstraída no objeto agregador, como código de um método desse objeto.

Dado um nível de abstração, o processo de *síntese* é utilizado para que se descubram os objetos em níveis hierárquicos inferiores, enquanto que o processo de *composição*, para que se relacione cada objeto com o seu correspondente em um nível de abstração superior. Através da síntese e composição são definidas as granularidades para os componentes.

A implementação dos processos de síntese e composição deve ser realizada de tal forma que esses processos possam ser automatizados. Com isso, espera-se que ferramentas de apoio ao projeto possam analisar componentes sob diversas granularidades. Para a implementação dessas tarefas, optou-se pela definição de um *Padrão de Projeto* com tal finalidade. Padrões de projeto [SHA 2004] são projetos de software baseados em OO onde são definidas as classes e os relacionamentos entre essas de modo a compor uma solução em software para um determinado problema. A vantagem do uso de padrões de projeto resume-se na capacidade de se reutilizar soluções previamente testadas em várias (diferentes) aplicações pertencentes a um mesmo domínio, com pouca ou nenhuma atualização funcional. Como os processos de síntese e composição são comumente utilizados e necessários para diversas ferramentas de apoio ao projeto que tratam da análise de SoCs, optou-se por descrevê-los como um padrão, para que possam ser reutilizados por outras ferramentas.

A figura 5.13 mostra o padrão de projeto para síntese e composição (*R&C, Refine and Composition*), através de um diagrama de classes UML.

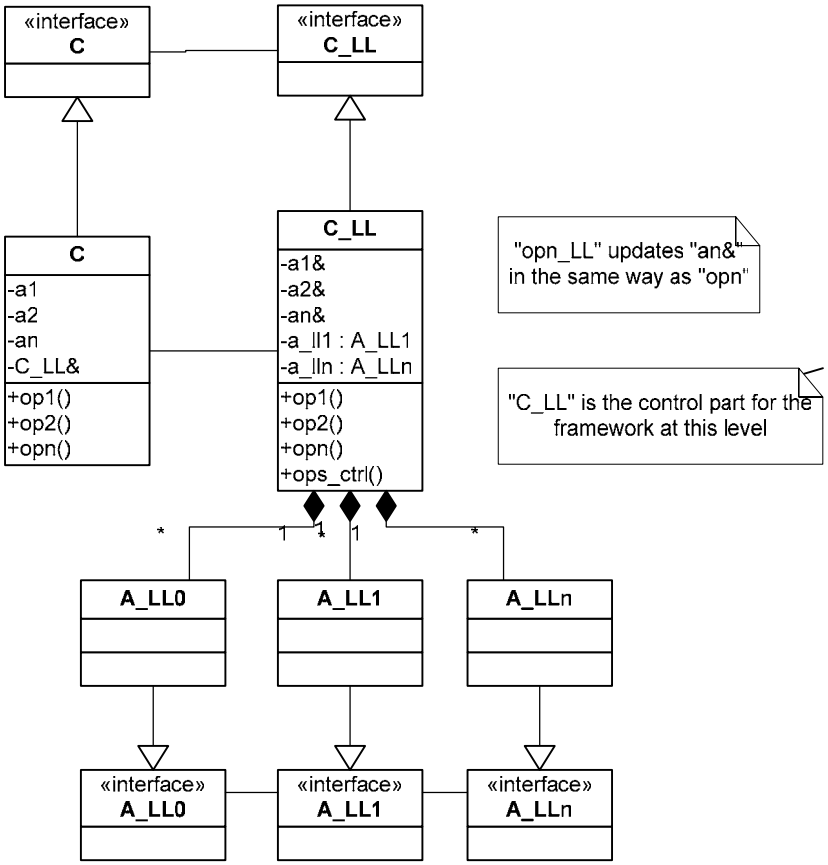


Figura 5.13: Diagrama de classes UML para o Padrão de Projeto para Síntese e Composição

Pode-se observar que o padrão de projeto R&C é compatível com o paradigma de modelagem baseado em interfaces. Com isso, modelos desenvolvidos segundo o modelo de programação aqui proposto podem utilizar o padrão sem a necessidade de serem modificados.

O padrão R&C é definido em função de duas entidades principais, “C” e “C_LL”; a primeira corresponde a um componente qualquer, enquanto que a segunda ao nível de abstração imediatamente inferior ao nível (*Lower Level*) de “C”. Essas entidades

comunicam-se também por meio de interfaces, para que mais de um refinamento possa ser modelado para cada nível de abstração. Na entidade C_LL são definidas as mesmas operações que em C para manter a consistência semântica, continuidade do modelo. Pelo mesmo motivo, as variáveis também são atualizadas no nível C sempre que operações no nível C_LL executam.

A entidade C_LL possui agregados todos os componentes necessários à implementação de todas as operações implementadas como métodos no nível C. Essa é a principal característica que diferencia níveis de abstração, segundo o modelo de programação: comportamentos implementados por código de métodos em uma linguagem de alto nível em um nível de abstração são implementados por relações entre componentes (objetos) em níveis de abstração inferiores. Cada operação no nível C_LL é implementada como a relação entre componentes A_LL.

No modelo mostrado na figura 5.13 os componentes do nível de abstração inferior são especificados como “A_LLx” onde *x* corresponde ao índice do componente. O número de componentes não é limitado, podendo ser utilizados tantos quanto necessário à implementação das operações; inclusive, mais de uma implementação pode ser realizada para cada operação no nível de abstração superior. Os componentes A_LL também relacionam-se por interfaces, como determinado pelo modelo de programação.

A entidade “C_LL” mantém o controle de todas as operações no seu nível de abstração, o que implica no controle da semântica de todas as operações que podem ser realizadas, através da interação com a interface de controle de cada componente (o conceito de interface de controle associada aos componentes é explicado no capitulo 6, onde são tratadas as ferramentas para otimização arquitetural).

A *composição* comportamental é realizada através da função *operation control* (*ops_ctrl()*) que é utilizada por cada componente A_LL para “registrar-se” como participante em uma ou mais operações “op_n”. Esse registro é realizado através da interface de controle onde, para cada operação em um nível de abstração superior, são registrados os componentes e as relações necessárias a sua implementação.

Para exemplificar o uso do padrão de projeto R&C, na figura 5.14 é mostrada a síntese de um roteador de uma NoC no nível de mensagens, em agregados de componentes – no nível de transações - para a execução das operações de roteamento e arbitragem.

As operações de roteamento e arbitragem são descritas algoritmicamente em uma linguagem de alto nível no nível de mensagens, através das funções *router()* e *arbiter()*, no componente RA (*Router Architecture*). No nível de transações, essas operações são implementadas através dos componentes arquiteturais internos da arquitetura do roteador: árbitro, roteador interno, controle de fluxo e buffer. Esses componentes são instanciados no nível de transações na entidade “RA_LL”, que corresponde à implementação de RA no nível de abstração imediatamente inferior à RA. No exemplo da figura 5.14, componentes para um roteador específico são instanciados, a partir de suas respectivas interfaces, como por exemplo, arbitragem do tipo Round-Robin (*Arbiter_RR*) e roteamento estático XY (*Router_XY*).

As implementações das operações de roteamento e arbitragem no nível de transações são mostradas respectivamente nos diagramas de seqüências das figuras 5.15(a) e 5.16(b). São essas implementações que correspondem às funções *router()* e *arbiter()* do componente RA_LL.

Para o roteamento (figura 5.15(a)), o roteador verifica se há um pacote no buffer e, caso afirmativo, estabelece o seu caminho a partir de seu cabeçalho. Em seguida, indica para o árbitro a ocorrência do pacote. A operação de escalonamento (figura 5.15(b)) consiste da seleção por parte do árbitro de um pacote para ser enviado e do envio desse pacote para o controle de fluxo; esse componente por sua vez verifica no buffer do RA vizinho se este possui espaço no buffer corresponde a porta entrada considerada; caso afirmativo, o RA destino responde afirmativamente (*grant*) o que habilita o pacote a ser enviado; caso contrário, este deve ser retido até que espaço seja liberado no buffer destino.

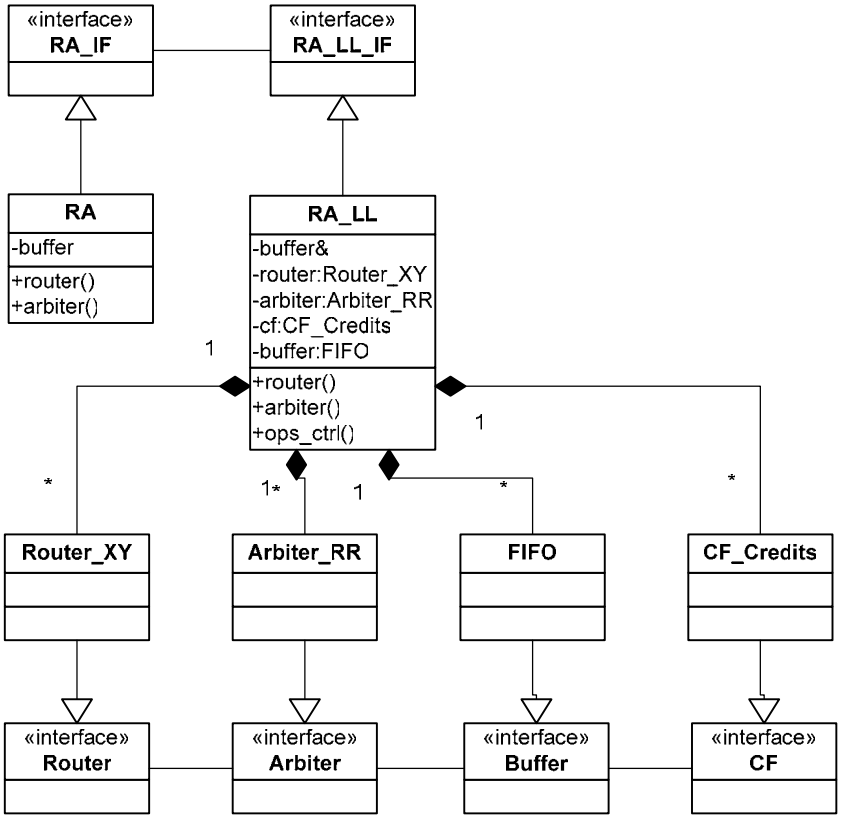


Figura 5.14: Síntese de um Roteador de NoCs utilizando o Padrão de Projeto R&C

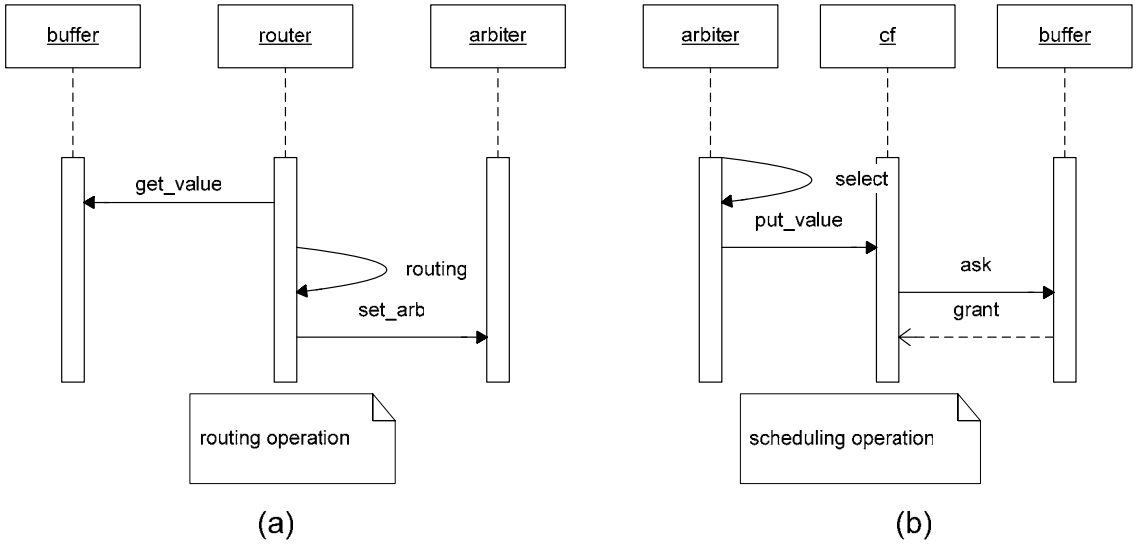


Figura 5.15: Operações de Roteadores de NoCs no Nível de Transações

5.8 Exemplos de Plataformas Arquiteturais

Nessa seção são mostrados dois exemplos de plataformas arquiteturais, uma para processamento e outra para comunicação, modeladas segundo os princípios estabelecidos pelo modelo de programação proposto nesse capítulo. As duas especificações correspondem a modelos genéricos no nível de abstração de transações e servem para demonstrar como os recursos de modelagem do modelo de programação podem ser utilizados na especificação de arquiteturas de processadores e de redes-em-chip. Essas descrições podem ser simuladas e analisadas/reconfiguradas pelas ferramentas de apoio ao projeto definidas pelo método, citadas no capítulo 3 e discutidas no próximo capítulo.

A primeira especificação refere-se a um processador genérico com *pipeline* e a segunda, à arquitetura interna de um roteador para NoCs. Os modelos são mostrados respectivamente, nas figuras 5.16 e 5.19.

Na figura 5.16, a arquitetura do processador está especificada através de interfaces, para atender os requisitos de modelagem do modelo de programação. Dessa forma, o pacote "PAP" (*processor architecture package*) pode ser portado e reconfigurado para atender a requisitos arquiteturais de diversas arquiteturas desse tipo de processador. Para tanto, este é implementado como um framework (ver capítulo 3).Cada um dos componentes admite reconfiguração para as suas funcionalidades. Por exemplo, operações podem ser adicionadas/removidas da ULA, o número de registradores no banco de registradores (*BReg*) alterado e assim por diante. O número de registradores para o *pipeline* (*Pipe_Register*) depende do numero de estágios implementado para o processador. Os sinais de controle são gerados para a PO através de uma "classe para relacionamentos" (*Control_Signals*) que envia as microinstruções para todos os componentes da PO. Se aplicado o padrão de projeto R&C a essa descrição, na geração do modelo no nível RT, internamente à classe para geração de microinstruções (*Microinstr_gen*) é observada uma máquina de estados, responsável pela geração das microinstruções. Essa máquina de estados pode ser modelada em UML por um diagrama de estados ou statecharts.

Pode-se observar ainda a presença de uma unidade específica para processamento digital de sinais (*DSP*) na especificação do processador. Essa unidade está incluída na organização do pipeline, podendo ser removida sem que isso afete as demais operações do processador. O controle do framework "PAP" pode adicionar ou remover unidades específicas, atendendo solicitações de ferramentas para otimização arquitetural.

Nas figuras 5.17 e 5.18 são mostrados respectivamente, diagramas de seqüências para os ciclos de busca e execução de instruções em especificações de processadores com *pipeline*. Nessas figuras pode observar-se o MoC arquitetural para processadores pipeline, utilizado na execução desses ciclos.

Para esse MoC é definido que um passo de simulação corresponde ao número de ciclos necessários à máquina de estados da PC gerar uma nova microinstrução. Por exemplo, na figura 5.18, a cada passo de simulação, a PC gera o sinal "exe" para os registradores do pipeline (*pipe_reg_R*, fase de leitura; *pipe_reg_E*, fase de execução e *pipe_reg_W*, fase de escrita) realizarem o deslocamento.

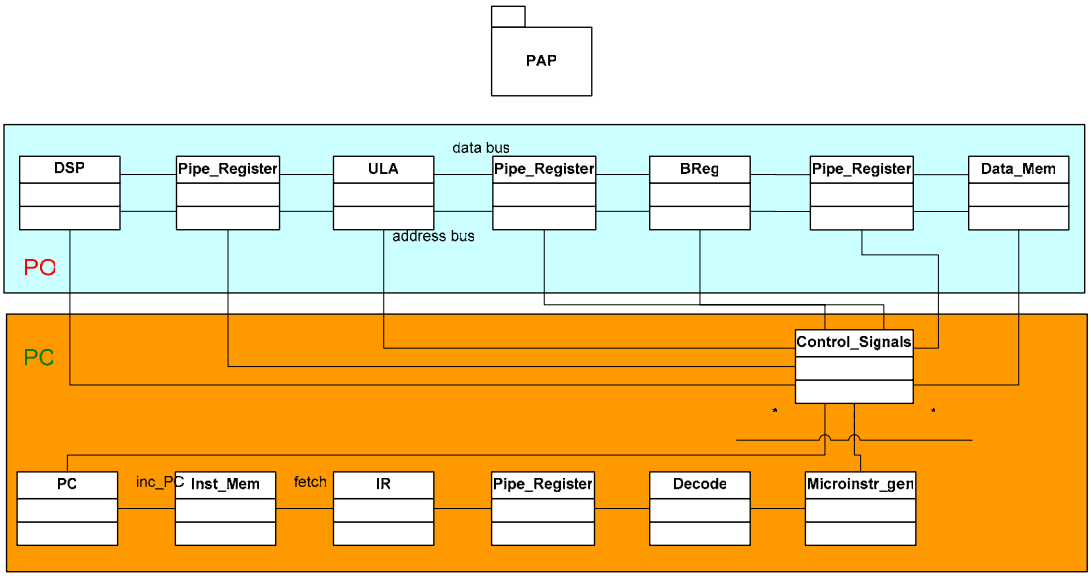


Figura 5.16: Modelo UML de Processadores de Propósito Geral com *Pipeline*

Na figura está demonstrada a seqüência de operações necessárias para encher um pipeline de 3 estágios de execução e com acessos somente ao banco de registradores. Uma vez preenchido o pipeline, os grupos de componentes “ULA/Breg” e “pipe_reg_R/pipe_reg_E/pipe_reg_W” executam concorrentemente. A execução desses grupos é intercalada pela PC, sendo considerado um passo de simulação, a execução dos dois grupos, de acordo com o definido para o MoC arquitetural para processadores pipeline: uma nova microoperação é gerada pela PC à cada operação.

Quanto à comunicação entre os componentes do processador, para o MoC arquitetural são estabelecidos dois tipos: a comunicação entre a PC e a PO pode ser caracterizada como sendo do tipo *dataflow*, uma vez que tokens (microoperações) são enviados à PO cujos componentes executam assim que o sinal de controle esteja disponível em um de suas entradas. Por outro lado, a comunicação entre os componentes internos da PO possui característica *rendezvous*, pois as operações de leitura/escrita da ULA em bancos de registradores e em memória são síncronas.

Quanto à natureza dos componentes, a PC é composta por uma máquina de estados, enquanto que a PO é um conjunto de componentes arquiteturais.

Portanto, o MoC arquitetural que caracteriza plataformas arquiteturais para processadores com pipeline é heterogêneo.

A especificação dos componentes internos de um roteador para NoCs, como mostrado na figura 5.19, é implementada como um pacote UML – “RA” – o qual estabelece um framework para este tipo de arquitetura. Como discutido no capítulo 3, o framework serve para controlar todas as operações e reconfigurar cada componente através de suas interfaces.

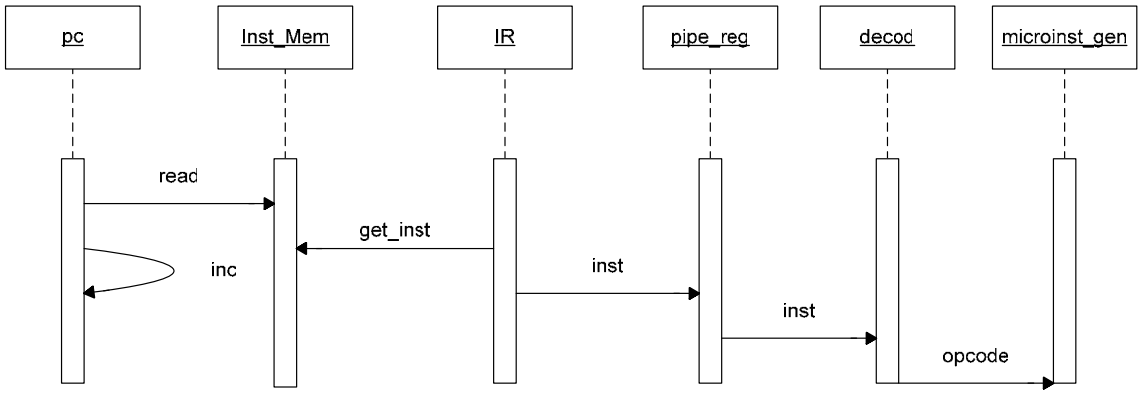


Figura 5.17: Ciclo de Busca em Processadores Pipeline

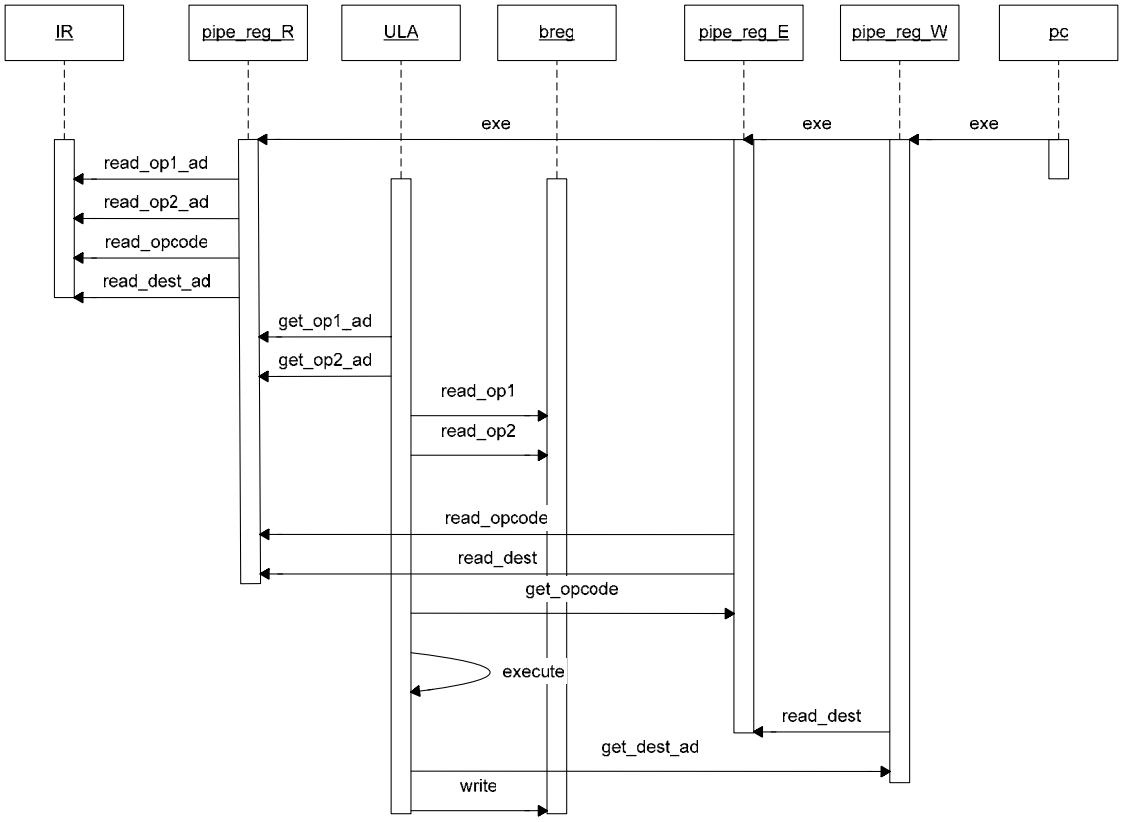


Figura 5.18: Ciclo de Execução em Processadores Pipeline

Na figura 5.19 observam-se variações para todos os principais componentes de roteadores arquiteturais para NoCs, como por exemplo, diferentes configurações para buffers (*DAMQ*; *SAMQ*; *FIFO*) e arbitragem (*Round-Robin*; *Priority*). Durante a avaliação, o controle do framework "RA" é utilizado pelas ferramentas de otimização arquitetural para configurar roteadores dentre as possibilidades disponíveis, a fim de

encontrar uma solução que atenda restrições de projeto. Por exemplo, o desempenho pode ser determinado em função do tipo de roteamento adotado; roteamentos estáticos - como XY – garantem para determinadas topologias a não ocorrência de *deadlocks*, mas – dependendo do tráfego da aplicação - podem incorrer em aumento de latência, se comparado com uma abordagem adaptativa. É conhecido que a implementação de organização de memórias como DAMQ, por exemplo, ocupa mais área do que uma implementação FIFO, com vantagens em desempenho.

Pode ser observado ainda, a conexão de um processador a uma porta de entrada (*InPort*) local (*LPort*); através dessa porta a aplicação executando em um processador consegue escrever os pacotes no buffer de uma porta local de um roteador de NoC.

Assim como plataformas arquiteturais para processadores, plataformas arquiteturais que caracterizam roteadores de NoCs também podem ser caracterizadas como heterogêneas. Um MoC para essas plataformas estabelece dois tipos de comunicação entre os seus componentes: o controle de fluxo estabelece comunicação síncrona para a verificação de disponibilidade de espaço em buffer, assim como o roteador para verificação da presença de pacotes no buffer, enquanto que a comunicação entre os demais componentes pode ser implementada como *Synchronous Dataflow* (SDF) [LEE 95]. Nesse tipo de comunicação, dados são transmitidos assincronamente, como por exemplo, a sinalização que um roteador realiza para o árbitro sempre que deseja enviar um pacote.

No nível de mensagens, a comunicação entre RAs pode ser implementada pela semântica de *Process Networks* (PN) [LEE 98], onde tokens são transmitidos assincronamente e armazenados em buffers.

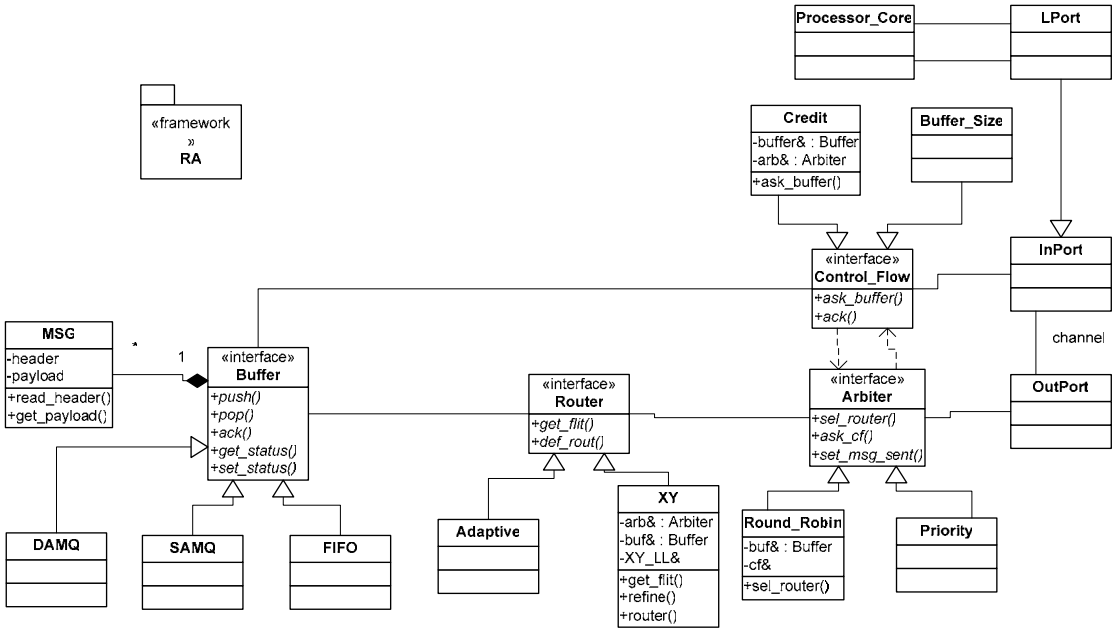


Figura 5.19: Modelo UML Genérico de Roteadores para Redes-em-Chip

5.9 Conclusões

Os processos de avaliação e otimização de sistemas computacionais envolvem diversas tarefas tais como configuração comportamental dos componentes arquiteturais presentes em plataformas e execução de suas funcionalidades. No entanto, essas tarefas

somente podem ser viabilizadas se as ferramentas de apoio ao projeto devotadas à análise puderem “compreender” quais são as funcionalidades presentes nos componentes, a sua semântica e possibilidades de reconfiguração. Além disso, é necessário que projetistas de sistemas possam especificar com clareza os comportamentos para os componentes arquiteturais, a fim de que possam compreender o contexto e a importância de todos os componentes na composição de sistemas.

Para tanto, nesse capítulo foi proposto um modelo de programação baseado na linguagem UML e nos conceitos de projeto e especificação baseado em interfaces. A linguagem UML, além de constituir-se padrão, herda todos os benefícios da orientação a objetos na especificação funcional de sistemas. Amplamente aceita pela comunidade de engenharia de software, vem sendo cada vez mais aceita para a especificação de componentes arquiteturais. Nessa direção, “perfis” e “estereótipos” específicos são introduzidos à linguagem, a fim de que características importantes na especificação de comportamentos para arquiteturas possam ser expressas, como por exemplo, passagem do tempo e concorrência. No modelo de programação aqui proposto diferentes modelos de computação podem ser expressos através de diagramas estáticos e dinâmicos.

Os conceitos relativos ao projeto baseado em *interfaces* permitem que diferentes comportamentos possam ser modelados para um mesmo componente, além de permitir que a comunicação entre os componentes possa ocorrer através de diferentes protocolos de comunicação. Ainda, interfaces são utilizadas para controlar as operações de componentes arquiteturais no *metanível*, o que trás como benefícios, os desacoplamento das funcionalidades das ferramentas de apoio ao projeto das dos componentes, além de permitir a simulação de código executável.

6 FERRAMENTAS DE APOIO AO PROJETO PARA OTIMIZAÇÃO DE PLATAFORMAS ARQUITETURAIS

Nesse capitulo são discutidas as ferramentas necessárias à efetivação das otimizações propostas para o método.

6.1 Objetivos e o Contexto das Ferramentas para o Fluxo de Projeto do Método

De acordo com o fluxo de projeto do método proposto, uma vez definido um modelo de programação que estabeleça regras para que aplicações e plataformas arquiteturais possam ser especificadas, são necessárias ferramentas para a otimização das plataformas, tanto de comunicação como para processamento, a fim de que estas acordem com as restrições impostas ao projeto.

Para tanto, são necessárias ferramentas para:

- Especificação dos componentes arquiteturais e das aplicações em UML, nos níveis de abstração considerados;
- Interpretação das descrições e mapeamento entre os níveis de abstração;
- Simulação das descrições para que das mesmas possam ser extraídos os estados em cada tempo de execução, com fins à avaliação; e
- Análise do comportamento de cada componente arquitetural e, se necessário, reconfiguração arquitetural para que se adapte às restrições de projeto.

6.2 Especificação de Componentes para Plataformas Arquiteturais

Como o método sugere que descrições tanto de aplicações como de componentes arquiteturais sejam realizados através da linguagem UML, é necessária uma ferramenta gráfica, compatível com os diagramas UML para a realização das descrições. Além disso, para a descrição de componentes torna-se necessária a escolha do nível de abstração a ser considerada para os componentes a serem descritos. Dependendo também da granularidade dos componentes descritos, estes podem ser compatibilizados com o padrão de projeto para síntese e composição, como definido no capítulo 4. Assim, é possível utilizá-los para análises em diferentes níveis de abstração. Por exemplo, caso componentes sejam especificados no nível de abstração de transações, estes podem ser agregados para comporem componentes no nível de mensagens.

É importante também que se observe a relação de cada componente com os demais componentes já descritos para a plataforma alvo, assim como dos componentes projetados para o futuro e que deverão relacionar-se com o componente sendo descrito.

Isso é conseguido observando-se as interfaces de controle de cada componente, como explicado no capítulo 6.

Finalmente, uma vez descritos os componentes ou a aplicação, deve haver uma ferramenta para a geração de código em alguma linguagem de alto nível, compatível com o simulador definido (utilizado) para a avaliação das plataformas – ver seção 6.4 abaixo. O código gerado servirá então para a simulação e avaliação de cada componente da plataforma analisada.

Para a descrição em UML, qualquer ferramenta comercial compatível com esta linguagem pode ser utilizada, como por exemplo Mentor "Nucleus Modeling" [MEN 2005] ou Artisan "Real-Time Studio" [ART 2005]. A única restrição refere-se à compatibilidade com o estereótipo utilizado para especificar o MoC utilizado para os componentes. Essa informação é utilizada pelo simulador para escalonar os componentes na ordem determinada pelo MoC, em cada tempo de simulação. O simulador verifica no código gerado qual o passo de simulação que deve ser considerado para cada componente. Isso é importante devido à possibilidade de que mais de um MoC pode estar disponível em uma mesma descrição. Essa informação deverá estar disponível na interface de controle dos componentes.

6.2.1 Uso do Conceito de Frameworks Orientado a Objetos para a Especificação de Plataformas

Uma vez que o método sugere o uso da OO para a descrição das plataformas e das aplicações através da linguagem UML, é interessante que se observe o conceito de frameworks orientados a objetos [FAY 99] como alternativa de descrição para plataformas. Para o propósito em que plataformas são descritas dentro do contexto das ferramentas de otimização sugeridas pelo método, o conceito de frameworks OO torna-se interessante, devido à semelhança conceitual. Um framework pode ser definido como um conjunto de classes e suas relações, as quais podem ser reutilizadas em qualquer aplicação compatível com a sua estrutura de dados definida por estas. Dessa forma, frameworks definem soluções para uma série de aplicações, as quais possuem comportamento semelhante.

Através do conceito de frameworks OO, aplicações são implementadas pelo reuso e configuração da estrutura de dados previamente definida pelo framework, com a finalidade que esta se adapte aos propósitos da aplicação em questão. A configuração é dependente do tipo de framework considerado quando da implementação da aplicação. Em frameworks "caixa branca" qualquer classe é feita acessível podendo ser configurados os seus métodos, novos métodos criados ou estendida através de herança. Já em frameworks "caixa preta" algumas classes são "escondidas" de forma que devem ser utilizadas assim como foram concebidas; estas podem apenas ser estendidas, ou até, nem são percebidas pelo programador.

O interessante é que os objetivos de se implementar frameworks OO e plataformas arquiteturais coincidem: enquanto que para frameworks OO a idéia consiste na *reutilização* e configuração de estruturas de dados para componentes de SW, para plataformas arquiteturais, consiste na *reutilização* de estruturas de dados para componentes de HW. Consequentemente, se um modelo de programação para plataformas, assim como definido no capítulo 4, está disponível, plataformas podem ser concebidas segundo o conceito de frameworks OO: componentes arquiteturais são concebidos em UML, suas relações disponibilizadas através de suas interfaces

operacionais e as possibilidades de reconfigurações, implementadas como “caixa branca” ou “caixa preta” de acordo com o desejado pelo projetista da plataforma.

6.3 Mapeamento entre Níveis Hierárquicos

O mapeamento entre níveis hierárquicos de descrições de componentes é realizado por uma ferramenta que automatize as operações de padrão de projeto para síntese e composição, como definido no capítulo 4. Essa ferramenta deve interpretar cada componente em um determinado nível de abstração e sintetizá-lo para um agregado de componentes ou agrupá-lo para compor um novo componente em uma granularidade maior. Por exemplo, instruções em processadores – descritas algoritmicamente no nível de mensagens - podem ser sintetizadas para os objetos que a executam na PO e PC do processador.

6.4 Abordagem de Simulação para Componentes Arquiteturais

Para que as descrições possam ser avaliadas deve ser concebido um simulador dedicado às descrições UML das plataformas. O simulador deve ser capaz de escalonar os componentes de uma dada descrição, de acordo com o nível de abstração da mesma, respeitando, portanto, a granularidade dos objetos.

Além disso, o simulador deve ser dedicado às descrições UML, de acordo com as regras propostas pelo modelo de programação, pois este deve respeitar a concorrência e o tamanho dos passos de simulação para cada componente ou grupo de componentes. Isso porque pode acontecer de mais de um MoC estar definido - através de estereótipos específicos - em uma mesma descrição. Nesse caso, diferentes passos de simulação são empregados. Pode haver o caso ainda, de componentes descritos por MoCs que seguem uma ordem temporal, enquanto que outros, apenas a ordem na qual os componentes devem executar. Para essas situações, o simulador assume o intervalo de tempo correspondente a um MoC, como sendo a granularidade temporal padrão, escalonando componentes pertencentes a outros MoCs como uma referencia à essa granularidade. Por exemplo, no caso de modelos mistos entre descrições ativadas por relógios, como Synchronous/Reactive [LEE 2002] e por descrições ativadas apenas por comunicações (trocas de mensagens) como Process Networks [LEE 2002]. Nesse caso, todos os componentes do modelo podem ser ativados pelo relógio definido para o MoC Synchronous/Reactive, sendo que somente mensagens efetivamente, ativarão funções nos componentes Process Networks. O simulador busca a informação acerca do MoC implementado para cada componente em uma descrição em sua interface de controle.

Assim, um passo de simulação sempre deverá corresponder ao escalonamento concorrente de todos os grãos de uma descrição, a intervalos que podem ser temporais, ou apenas denotar a ordem, na qual os componentes devem ser escalonados. O sincronismo é implementado pelos protocolos implementados para a comunicação entre os componentes, como primitivas de comunicação.

No nível de aplicação, componentes correspondem às tarefas para o comportamento de processamento e às mensagens (enviadas por primitivas de comunicação) para o comportamento de comunicação da aplicação.

No nível de mensagens, componentes correspondem às instruções para processamento e a pacotes para comunicação roteados em RAs. No nível de transações, para processamento são escalonados os componentes arquiteturais da PC e PO, enquanto que para comunicação, os componentes arquiteturais de RAs para NoCs ou árbitros para barramentos.

O tempo para cada passo de simulação pode ser estimado, considerando- se a pré-síntese dos componentes arquiteturais da plataforma analisada.

Sendo os simuladores arquiteturais utilizados com fins à otimização dos componentes em plataformas, estes devem executar o mais rápido possível, para que vários componentes possam ser avaliados em tempo hábil. Nesse sentido, duas providencias podem ser respeitadas: primeiramente, os simuladores devem ser compatíveis com o modelo de programação, diferentes níveis de abstração podem ser utilizados durante a avaliação dos componentes; maiores desempenhos de simulação são obtidos em simulações de mais alto nível. Em segundo lugar, as funcionalidades de cada componente podem ser previamente compiladas, gerando-se código executável. Com isso, os simuladores necessitam apenas executar o controle do escalonamento de cada componente segundo o MoC empregado, sendo as operações dos componentes executadas nativamente na máquina utilizada para a avaliação. Ressalta-se que o modelo de programação sugerido nesse trabalho permite a implementação dessas características para simuladores, através do uso da *interface* de controle, discutida mais adiante na seção 6.5.1. A *interface* de controle é executada no metanível, permitindo aos simuladores para arquiteturas escalonarem componentes executáveis, além de extrair as informações pertinentes às funções custo, em diferentes níveis de abstração.

Outra alternativa seria o uso da técnica de “simulação compilada” [BRA 2004]. No entanto, essa abordagem pode restringir a avaliação do comportamento dinâmico da aplicação, uma vez que somente a estrutura estática – obtida durante a fase de compilação do simulador – das operações de controle é considerada para a execução da aplicação.

6.5 Análise, Reconfiguração e Otimização de Componentes Arquiteturais

O último grupo de ferramentas necessárias à efetivação do método para a otimização de plataformas arquiteturais refere-se à análise dos componentes das plataformas de comunicação e processamento, bem como a reconfiguração arquitetural sempre que otimizações forem necessárias. Uma vez que os objetivos dessas ferramentas recaem sobre a otimização de componentes arquiteturais, ressalta-se que uma reconfiguração arquitetural envolve em alterações na funcionalidade do componente analisado.

Devido à ortogonalidade existente entre os comportamentos de comunicação e processamento, as ferramentas para análise comportamental podem atuar separadamente ou em conjunto sobre estes. No entanto, independente do tipo de comportamento analisado, deve ser provido às ferramentas um mecanismo para que os componentes possam ser acessados e reconfigurados. Mais do que esse mecanismo, as ferramentas necessitam também “conhecer” a granularidade analisada para saber quais componentes podem ser funcionalmente reconfigurados e qual o impacto disso para os demais componentes da plataforma.

Análises sobre processamento envolvem a configuração de conjunto de instruções para processadores enquanto que para comunicação existe a possibilidade de reconfiguração de políticas de escalonamento para barramentos e de topologia e dos componentes internos aos RAs para NoCs. A análise conjunta de plataformas de processamento e de comunicação induz a um compromisso entre número de processadores e – para o caso de NoCs – o tamanho da rede: quanto maior o número de processadores, maior o número de RAs; no entanto, se as tarefas de uma aplicação são

mapeadas em um número maior de processadores, estes tendem a ser mais simples, pela eliminação de instruções específicas à determinadas tarefas.

De qualquer maneira, muitas variáveis podem ser consideradas para o processo de otimização arquitetural. Isso decorre do fato de que as plataformas analisadas são consideradas para a realização de sistemas complexos, contendo diversos componentes e comportamentos heterogêneos.

Para que as ferramentas de análise possam acessar os componentes e determinar as possíveis reconfigurações, foi aplicado o conceito de interface ao controle dos componentes das plataformas, bem como de suas inter-relações. Para que fosse possível controlar as operações dos componentes sem afetar as suas descrições, optou-se por realizar as operações de controle e otimização comportamental no *metanível*. Com isso, componentes podem ser especificados independentemente das ferramentas que são utilizadas para configurá-los, com vista à otimização. No entanto, cada componente deve expor as funcionalidades passíveis de serem configuradas, através de uma interface específica ao controle, chamada de “Interface de Controle e Introspecção Computacional”.

6.5.1 Interface de Controle e Introspecção Computacional

Assim como para componentes são definidas interfaces para a comunicação também para o controle esse paradigma se mostra eficiente: as funcionalidades inerentes de componentes podem ser especificadas independentemente dos demais componentes utilizados na composição da aplicação (para a interface de especificação) e nem como serão avaliados (para a interface de controle).

Através do uso da interface de controle é possível controlar, no meta nível, quais as operações que podem ser configuradas, além das variáveis cujo estado pode ser avaliado. A figura 6.1 mostra um exemplo para uma PC de um processador, onde podem ser avaliadas as instruções do seu conjunto de instruções, bem como o tempo de execução de cada instrução. Deve-se perceber que, enquanto que as instruções são configuradas durante o tempo de projeto, o tempo de cada instrução somente pode ser obtido durante a execução da aplicação. Dessa maneira, a reflexão computacional assume importância em dois sentidos: para separar a funcionalidade dos componentes das funcionalidades das ferramentas para a otimização destes e para obter informações do estado atual de cada componente durante a sua execução. Para que seja possível a execução das operações de configuração e obtenção do estado atual em componentes, é necessário que estes implementem os métodos definidos pela interface de controle, como exemplificado na figura 6.1.

Para que a avaliação possa ocorrer em tempo de execução, é adotada a abordagem de simulação. Sendo o simulador dedicado e, portanto, compatível com as descrições segundo o modelo de programação, este interage com a interface de controle a fim de obter os estados das variáveis avaliadas e repassá-los às ferramentas para otimização, para a realização das análises.

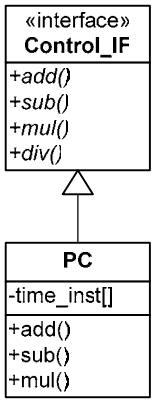


Figura 6.1: Interface de Controle

6.5.2 O Padrão de Projeto “Proxy”

Uma vez que as configurações das plataformas são implementadas no metanível - através da interface de controle – é possível que sejam separadas as funcionalidades das ferramentas das dos componentes arquiteturais. Nessa abordagem isso é conseguido através do uso padrão de projeto conhecido como *Proxy* [PRO 2004]. Esse padrão para projeto de sistemas especifica uma *interface* para acessar os métodos de uma classe, como mostrado na figura 6.2. Nesse exemplo, cada vez que um cliente (*client*) deseja realizar uma consulta a um objeto da classe “Subject”, este a realiza através do Proxy, o qual interage com todas as implementações dessa – *RealSubject* - classe.

Assim, as funcionalidades das ferramentas para análise são totalmente desacopladas das funcionalidades dos componentes da plataforma. Isso habilita que ferramentas de análise e componentes possam ser especificados concorrentemente. A classe *Proxy_control* possui todos os métodos necessários para a introspecção realizada – no metanível - pelas ferramentas de controle. A introspecção possui como finalidade monitorar as plataformas durante o processo de avaliação, através de métodos que acessam a parte de controle dos frameworks correspondentes às descrições das plataformas. Por exemplo, a classe *Proxy_control* pode executar a simulação para as descrições das plataformas, reconfigurar operações nas plataformas (função set_ops()) e devolver o estado atual de variáveis avaliadas, como o desempenho de cada mensagem (get_status()).

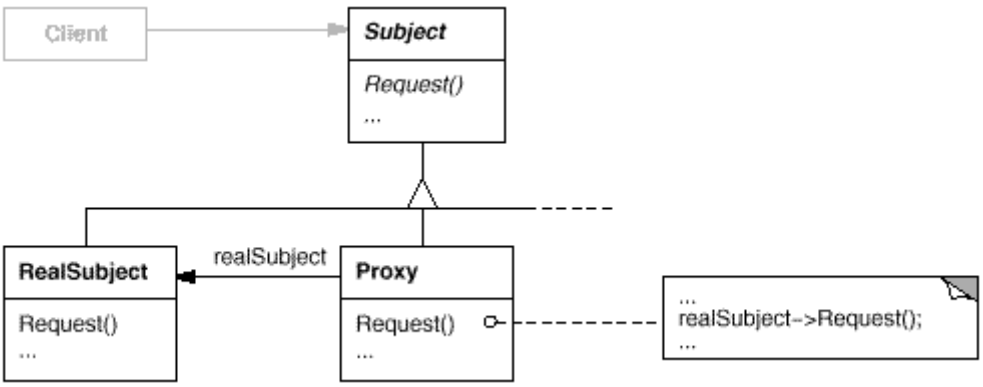


Figura 6.2: Padrão de projeto *Proxy*

A figura 6.3 exemplifica para a PC de processadores as relações entre as ferramentas de análise, simulação e descrições de componentes utilizando a interface de controle.

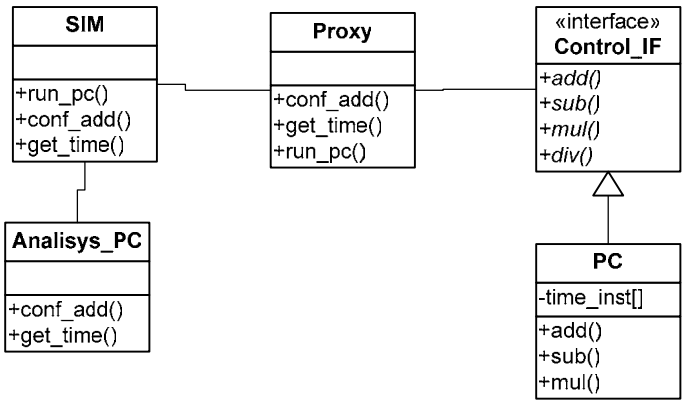


Figura 6.3: Relações entre as Ferramentas de Apoio ao Projeto

Como pode ser percebido na figura 6.3, todas as operações são realizadas através do padrão de projeto “Proxy”, o qual implementa as operações para configuração dos componentes (em tempo de projeto) e para obtenção do estado das variáveis analisadas (em tempo de execução/simulação).

6.5.3 Otimizações em Componentes para Processamento

Como previsto pelo método, componentes arquiteturais para plataformas são analisados sob diferentes níveis de abstração. As plataformas de processamento são modeladas com os principais componentes arquiteturais utilizados em sistemas dedicados/embarcados, como por exemplo, unidades específicas para processamento de sinais, processamento de imagens, etc.; além de componentes comumentemente encontrados em processadores de propósito geral, como ULAs, banco de registradores, etc.

Esses componentes são agregados para compor plataformas de processamento diversas, tais como CISC, RISC, VLIW, DSP e variações destas com unidades específicas.

Processadores com conjunto de instruções dedicados (através da inclusão de unidades especificas na PO), denominados “ASIPs” (*Application Specific Instruction-Set Processors*) constituem o foco principal das plataformas de processamento analisadas, devido à possibilidade de reconfiguração arquitetural.

Uma vez que a reconfiguração em processadores pode ocorrer em diferentes níveis de abstração, diferentes granularidades são analisadas. Se a análise for realizada sob uma descrição no nível de mensagens, podem ser testadas inclusões de instruções específicas, tais como para processamento digital de sinais ou compressão de vídeo. Isso é possível porque a granularidade corresponde ao conjunto de componentes arquiteturais que implementam a parte operativa do processador, onde unidades dedicadas podem ser acrescentadas, para a implementação das instruções específicas. Nesse nível de abstração, as ferramentas de análise verificam o desempenho do processador quando da execução de uma ou mais tarefas.

Já no nível de transações é possível verificar o funcionamento de cada componente arquitetural interno à PC ou à PO, como por exemplo:

- Unidade que implementa políticas para previsão de desvios;
- Unidades de despacho de instruções;

- Número de barramentos de acesso à memória;
- Tamanho da palavra do processador;
- Número de unidades lógico/aritméticas para processadores VLIW ou Superescalares; e
- Tamanho do banco de registradores.

Os componentes da PC/PO de processadores que podem ser configurados no nível de transações são exemplificados pela figura 5.8 (capítulo 5), a qual exemplifica a modelagem de processadores nesse nível de abstração.

Finalmente, no nível RT as reconfigurações arquiteturais referem-se à organização do sistema de interrupções, de acessos aos periféricos e à organização das *microinstruções*.

6.5.4 Otimizações em Componentes para Comunicação

As arquiteturas de comunicação analisadas referem-se a barramentos e redes chaveadas do tipo Rede-em-Chip (*Network-on-Chip – NoC*) [BEN 2002]. A opção por redes-em-chip justifica-se na constatação de que as futuras tecnologias de fabricação permitirão a criação de sistemas multiprocessados em uma única pastilha de silício, os quais possuem restrições quanto à distribuição de relógio, favorecendo a comunicação *assíncrona*. Além disso, o paralelismo oferecido pelas redes pode atender às necessidades por grandes larguras de banda.

Considerando-se apenas a comunicação, essas ferramentas operam apenas sobre os modelos de computação referentes ao comportamento de comunicação da aplicação. A reconfiguração das arquiteturas de comunicação é realizada em relação ao nível de abstração no qual a análise é realizada. No nível de mensagens diferentes topologias podem ser testadas. Já no nível de transações são testados diferentes comportamentos para os componentes arquiteturais internos de roteadores para NoCs (*RA, Router Architecture*) os quais podem ser citados como sendo: organização de memória (*buffer*), roteador, árbitro e controle de fluxo.

Outra característica interessante da análise sobre arquiteturas de comunicação refere-se à possibilidade da criação de arquiteturas de redes heterogêneas, as quais possuem comportamentos diferenciados entre os RAs da rede. Isso pode ser realizado para compatibilizar com comportamentos de comunicação heterogêneos para aplicações dedicadas. Por exemplo, tarefas especificadas para a execução em um único processador podem alternativamente ser distribuídas em mais de um processador, supostamente, mais otimizados as características de cada tarefa.

Nessa situação, o aumento da concorrência na execução das tarefas pode demandar um aumento por largura de banda somente para estas tarefas, o que leva a otimizações específicas, como por exemplo, posicionamento de processadores ou políticas de escalonamento dedicadas às mensagens dessas tarefas.

Outra situação onde redes heterogêneas podem ser consideradas ocorre quando a mistura de RAs com características diferenciadas conduzem a uma plataforma mais otimizada, para área, por exemplo.

6.5.5 Otimizações para Plataformas Completas

Devido à ortogonalidade entre os comportamentos de processamento e comunicação, estes podem ser analisados separadamente. No entanto, é interessante que

sejam previstas também ferramentas para a análise conjunta desses comportamentos. Isto está previsto no fluxo de projeto do método, através da seta que unifica a análise para essas duas classes de plataformas.

A análise conjunta justifica-se pelo fato de que um comportamento pode exercer influência sobre o outro, quando da otimização da plataforma. Por exemplo, a distribuição de tarefas sobre processadores disponíveis em uma plataforma, (os quais podem inclusive, admitir reconfiguração) pode ser realizada visando à otimização individual de cada processador. Isso pode levar a implementação de processadores mais simples que, por possuírem somente os componentes necessários à execução de instruções específicas, possuem menos área e/ou menor consumo de potência. No entanto, para que um número maior de processadores possa ser utilizado para a execução das tarefas da aplicação, é necessário que a arquitetura de comunicação suporte a largura de banda requerida para a execução do comportamento de comunicação entre as tarefas.

A análise conjunta das arquiteturas de comunicação e processamento é realizada dimensionando-se o número de conexões locais para processadores conectarem-se a RAs em NoCs ou o número de entradas para os árbitros de barramentos.

6.6 Conclusões

Neste capítulo foram explicitadas as ferramentas de apoio ao projeto, necessárias à efetivação do fluxo de projeto do método, bem como as relações entre elas. Para a realização dos experimentos, ferramentas com essas características foram implementadas em C++. Essas ferramentas utilizam-se de instrospecção computacional para – através da interface de controle – configurar os componentes arquiteturais e avaliar o seu estado frente às restrições de projeto.

Para tanto, foram determinados os objetivos de cada ferramenta em conjunto com as suas respectivas estruturas de dados. Estas foram especificadas também em UML, para manter a compatibilidade com as descrições dos componentes das plataformas. Assim, as funcionalidades das ferramentas também podem ser identificadas como objetos, os quais relacionam-se diretamente com as interfaces de cada objeto representante dos componentes arquiteturais das plataformas.

No próximo capítulo são exemplificadas especificações de algumas plataformas arquiteturais para comunicação, otimizadas através do uso efetivo das ferramentas aqui descritas.

7 PLATAFORMAS ARQUITETURAIS PARA COMUNICAÇÃO

O avanço proporcionado pelas atuais modernas tecnologias de fabricação de circuitos eletrônicos permite que sistemas cada vez mais complexos – em termos do número de componentes arquiteturais – possam ser idealizados e implementados em silício. A grande quantidade de componentes esperada para esses sistemas torna evidente a necessidade por estruturas de comunicação otimizadas, as quais devem prover a QoS necessária para o comportamento de comunicação da aplicação dedicada a ser implementada no SoC.

Uma vez que as operações implementadas segundo a lógica *booleana* possuem comportamento ortogonal para comunicação/processamento, arquiteturas dedicadas à execução otimizada de instruções de comunicação podem ser obtidas. Neste capítulo são tratadas arquiteturas dedicadas exclusivamente à implementação do comportamento de comunicação de aplicações dedicadas. Objetiva-se com isso efetivar a busca por arquiteturas de comunicação otimizadas, uma vez que essa classe de arquiteturas assume um papel fundamental para a execução de aplicações em sistemas dedicados multiprocessados, respeitando-se as restrições de projeto.

7.1 Componentes Arquiteturais Considerados

Os componentes arquiteturais para plataformas de comunicação considerados para avaliação são os apontados na definição 3.4, para redes chaveadas:

- *Roteadores ($r_{i.rout}$, da definição 3.4)*: a política de roteamento decide o caminho por onde os pacotes de uma mensagem deverão seguir, desde o nodo fonte até o destino. Para que possam ser testadas duas políticas para roteamento, foram implementados os roteamentos *estático X,Y* e *adaptativo*. No roteamento X,Y o nodo fonte determina o caminho até o destino atribuindo às variáveis X e Y respectivamente, o número de colunas e linhas até o nodo destino. No roteamento adaptativo, a rota é criada à medida que a mensagem trafega pela rede. Nessa política, um roteador escolhe o canal de comunicação por onde a mensagem deve seguir, levando em consideração o tráfego na rede: a mensagem segue pelo canal mais livre. Por questões de qualidade-de-serviço, deve-se garantir para esta política de roteamento, que uma mensagem nunca fique circulando indefinidamente pela rede, *livelock*. Critérios para evitar o *livelock* são discutidos em [DUA 97]. A arquitetura de Árvore Gorda utiliza roteamento estático para o primeiro nível e dinâmico para a escolha de um caminho para o segundo nível, se necessário.

- *Árbitros ($r_{i.sch,}$, da definição 3.4)*: as políticas de arbitragem adotadas para as arquiteturas de rede são do tipo *Round-Robin* e por *Prioridades*. Para o primeiro tipo, é utilizado um buffer circular de prioridades, o qual procura estabelecer um critério de igualdade entre todas as portas de entrada de um roteador, as quais desejam enviar uma

mensagem. Isso porque através do buffer circular, uma vez que uma porta de entrada envia uma mensagem, esta recebe a menor prioridade, devido ao avanço do ponteiro do *buffer* circular. Assim, essa porta de entrada somente conseguirá enviar novamente uma mensagem, depois que todas as outras portas tiverem tal oportunidade. A política de arbitragem por prioridades determina a ordem na qual as mensagens serão enviadas, através do "tipo" de cada mensagem, de acordo com a prioridade previamente estabelecida para cada tipo. Critérios de QoS para essa política devem evitar a situação de *starvation*, caso onde mensagens com baixa prioridade nunca são enviadas adiante, ficando permanentemente presas à um roteador. Regras para evitar *starvation* podem ser encontradas em [DUA 97].

• *Controle de Fluxo (*$r_{i.cf}$ *,da definição 3.4*: implementado em função através do tamanho dos *buffers*, através de um protocolo de *handshake*. Nas situações em que um buffer pertencente à uma porta de entrada estiver cheio e um canal de comunicação desejar escrever uma mensagem nesse buffer, é enviado um sinal de controle bloqueando o envio da mensagem. Portanto, a cada vez que um árbitro desejar enviar uma mensagem, este deve primeiro certificar-se de o buffer destino não está cheio.

• *Organização dos Buffers (*$r_{i.buf}$ *,da definição 3.4*: buffers são necessários como uma memória temporária interna à cada roteador. Isso é requerido pelo chaveamento por pacotes para garantir que estes não serão perdidos quando por questões de tráfego na rede, não poderem ser enviados no exato instante em que chegam a uma porta de entrada do roteador. Critérios de QoS devem garantir que os buffers sejam dimensionados adequadamente, para que não ocorram essas perdas. Para as descrições de NoCs aqui adotadas foram especificadas organizações de buffers do tipo fila – FIFO, *First In First Out* – e DAMQ (*Dinamically Allocated, Multiple Queue*) [TAM 92]. Essas duas classes de organização de buffers representam comportamentos de memorização típicas encontradas para NoCs.

• *Chaveamento:* implementado por *pacotes* em todas as arquiteturas, pois esse tipo de chaveamento tende a utilizar melhor o paralelismo proporcionado pela rede. Isso ocorre porque o chaveamento por pacotes não aloca simultaneamente, todos os canais de comunicação necessários ao envio de uma mensagem.

Arquiteturas de barramentos podem ser modeladas através da política de escalonamento. Ressalta-se que, face ao que se espera por arquiteturas de sistemas alvo para o futuro próximo, os experimentos aqui conduzidos focam principalmente na avaliação de redes intrachip chaveadas (NoCs). Isto, baseado na grande aceitação que este tipo de arquitetura vem recebendo da comunidade científica para a implementação do comportamento de comunicação para aplicações complexas, como discutido no capítulo 2. Arquiteturas do tipo NoC vem recebendo especial atenção, afirmando-se como o mais provável padrão a ser adotado para atender aos requisitos de projeto para os futuros sistemas multiprocessados.

7.2 Frameworks para Arquiteturas de Comunicação

Nesta seção é especificado o framework implementado segundo o modelo de programação aqui proposto, e que serve para implementar e avaliar componentes arquiteturais para plataformas de comunicação.

O framework para comunicação implementa as descrições para os componentes arquiteturais, como comentado no capítulo 5. No entanto, para que redes chaveadas possam ser instanciadas, é necessário que os componentes possam ser "agregados" e "conectados". A operação de agregação serve para compor a arquitetura de roteadores,

de acordo com a definição 3.5. já a operação de conexão implementa a topologia da rede, como especificado no modelo de programação. Com a operação de conexão, um conjunto de descrições de RAs compõe diferentes topologias como por exemplo, árvore gorda, torus, hipercubo, etc. Utilizando-se os componentes arquiteturais para comunicação explicitados no capítulo anterior, podem ser criadas diferentes configurações para arquiteturas de barramentos, bem como de redes chaveadas. Nessa direção foram modeladas descrições para barramentos e para redes chaveadas com topologias regulares do tipo Grelha e Torus e a topologia irregular do tipo árvore gorda. A tabela 7.1 mostra a semântica adotada para os componentes arquiteturais para essas topologias. Com os componentes implementados é possível cobrir uma ampla faixa de arquiteturas de comunicação, o que confere às ferramentas de otimização um espaço de busca representativo para esse tipo de plataforma.

Tabela 7.1: Componentes Arquiteturais implementados para diferentes Topologias do tipo Rede Chaveada

Topologia	*Controle de Fluxo*	*Arbitragem*	*Roteamento*	*Chaveamento*	*Organização dos Buffers*
Grelha	Tamanho dos Buffers, Síncrono	Round Robin, Prioridades	Fonte (X,Y), adaptativo	Pacotes	DAMQ, FIFO
Árvore Gorda	Tamanho dos Buffers, Síncrono	Prioridades	Estático e Adaptativo	Pacotes	FIFO
Torus	Tamanho dos Buffers, Síncrono	Round Robin, Prioridades	Fonte (X, Y), adaptativo	Pacotes	DAMQ, FIFO

As políticas de escalonamento refletem duas situações comumentemente encontradas para a decisão quanto ao momento no qual pacotes são enviados: prioridades para, por exemplo, mensagens urgentes e compartilhamento igualitário do tempo, no caso da política de *round-robin*. No caso da política de *round-robin* a latência dos pacotes das mensagens não sofre influencia da política de escalonamento, uma vez que todos pacotes é concedido o mesmo período de tempo.

Quanto ao roteamento, a política estática XY previne problemas com *deadlock* e *livelock*, determinando todo o caminho dos pacotes desde o roteador fonte até o destino, antes do envio do cabeçalho. No entanto, o emprego dessa política de roteamento pode levar ao aumento de contenções na rede. Quando políticas *adaptativas* são empregadas para amenizar esse problema, cuidados especiais devem ser tomados a fim de se evitar esses problemas [ZEF 99]. Por exemplo, caminhos na rede podem ser proibidos, garantindo que pacotes não circulem por determinados canais de comunicação mais de uma vez.

A organização da memória (*buffers*) permite o uso do esquema clássico de filas (FIFO) e de canais virtuais (DAMQ [ZEF 2002]), técnica esta, que visa o aumento de desempenho. Para maiores detalhes acerca da organização dessas memórias, bem como de suas estruturas, o leitor pode se referir a [ZEF 2002].

As descrições dos componentes de NoCs formam uma hierarquia, onde a topologia – conexões entre RAs - é descrita no nível de mensagens e a arquitetura dos componentes internos aos RAs e as suas interconexões, no nível de transações. A figura 7.1 mostra graficamente a hierarquia das descrições para topologias regulares de NoCs.

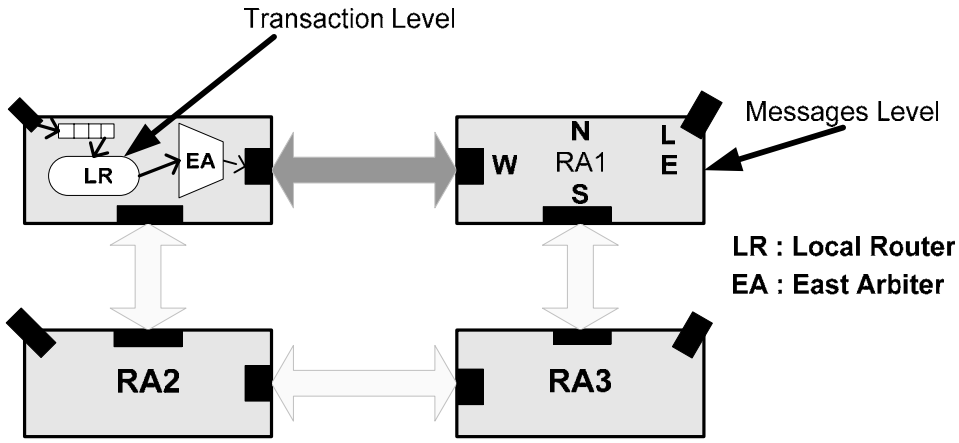


Figura 7.1: Relação hierárquica encontrada em descrições de Redes Chaveadas

Pode-se observar na figura 7.1 os componentes internos de RAs para NoCs, como buffer (em cima, à esquerda), roteador da porta local (LR) e um MUX representando o árbitro da porta leste (EA). As portas são mostradas como quadrados pretos e, no caso de uma topologia regular como a mostrada, cada RA pode possuir até 5 portas: Local, Norte(N), Sul(S), Leste(E) e Oeste(W).

No nível de mensagens, a rede é vista como uma coleção de RAs conectados por canais de comunicação, formando uma topologia. De acordo com o modelo de programação do método, no nível de mensagens, as funções pertencentes aos componentes internos dos RAs são implementadas semanticamente pela sintaxe de uma linguagem de programação de alto nível e encapsuladas como métodos. A composição desses métodos implementa um objeto RA. Já no nível de transações, cada componente interno de RAs é implementado como um objeto e conectado com os demais. Uma vez que o modelo de programação sugere o uso de interfaces para a modelagem de todos os componentes, diferentes comportamentos podem ser definidos para cada um.

A conexão nos dois níveis de abstração pode ser realizada através da atribuição de endereços a ponteiros ou através de chamadas de funções.

Para barramentos são definidos árbitros, os quais implementam políticas de escalonamento baseadas em prioridades, com a finalidade de escalonar as mensagens enviadas por cada processador conectado ao canal de comunicação.

Quanto ao MoC arquitetural para comunicação, a descrição temporal – utilizando diagrama de seqüências – no nível de mensagens implementa as comunicações dos pacotes das mensagens entre RAs, enquanto que no nível de transações, as comunicações internas entre os componentes de um RA. As figuras 7.2 e 7.3 exemplificam o MoC arquitetural para NoCs.

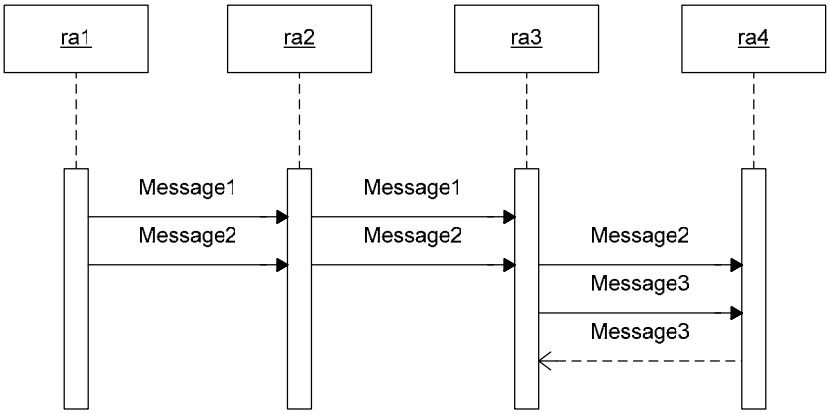


Figura 7.2: Diagrama de Seqüências para MoC Arquitetural para Comunicação no nível de Mensagens

Na figura 7.2 verificam-se exemplos de trocas de mensagens entre quatro RAs. “ra1” envia duas mensagens: a mensagem “1” (*message1*) é enviada para “ra3” através de “ra2”, e a mensagem “2” para “ra4” através de “ra2” e “ra3”. Roteadores pelos quais mensagens passam até chegar ao destino são determinados pela topologia. Pode-se verificar ainda na figura 7.2 que as mensagens “1” e “2” são assíncronas e a mensagem “3”, síncrona, uma vez que esta demanda uma resposta do roteador “ra4” para “ra3”.

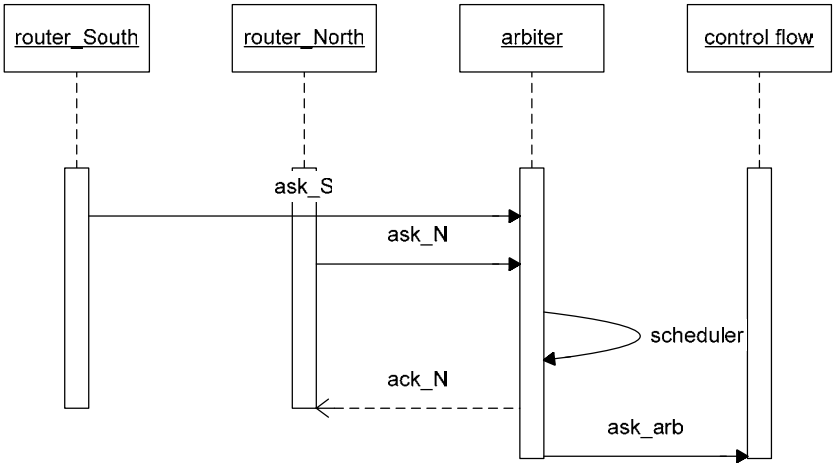


Figura 7.3: Diagrama de Seqüências para MoC Arquitetural para Comunicação no nível de Transações

No exemplo da figura 7.3, os roteadores Sul (*South*) e Norte (*North*) solicitam ao árbitro do RA que pertencem o envio de um pacote de uma mensagem. O árbitro, após realizar o escalonamento, determina a prioridade para o roteador Norte, que recebe a autorização (*ack*) para enviar o seu pacote. Uma vez que o árbitro determina que um pacote deva ser enviado, este solicita a autorização para o controlador de fluxo (*control_flow*), o qual verifica a disponibilidade para o envio do pacote no buffer do roteador destino.

7.3 Ajuste das Funções Custo

As funções custo para a análise de plataformas de comunicação envolvem a determinação dos parâmetros necessários às restrições de área, desempenho e consumo

de energia. Por momento são essas as restrições consideradas como QoS para NoCs. Questões relativas ao particionamento da rede, tolerância a falhas ou ordem/perda de pacotes espera-se considerar na seqüência do trabalho.

As restrições consideradas estão expressas nas equações 3.4, 3.5 e 3.6, as quais estabelecem as funções custo respectivamente para energia, desempenho e área.

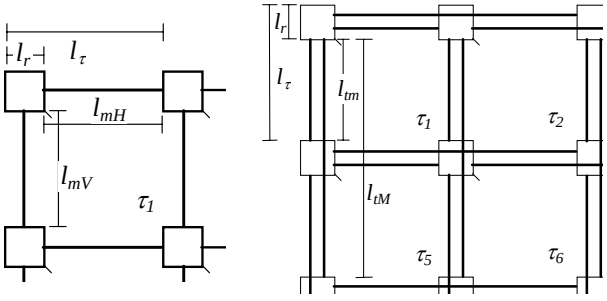
A energia dinâmica (*EDynNoC* = $\varepsilon_{rout} + \varepsilon_{sch} + \varepsilon_{buf\ +,} \varepsilon_{cf}$, da equação 3.4) para redes chaveadas é determinada segundo um modelo de energia, de acordo com o proposto em [KRE 2005]. Esse modelo utiliza conceitos similares aos apresentados em [MAR 2003] [MUR 2004] estendendo-os para estimar o consumo de energia em mais de uma topologia. Entretanto, apenas topologias regulares – Mesh e Torus – são consideradas até o momento. No modelo de energia adotado, é utilizado o conceito de *Ebit* para estimar o consumo de energia dinâmica para cada bit de pacotes que trafegam em NoCs. *Ebit* é então sub-dividido em consumo de energia dinâmica para os *buffers* (*EBbit*), para o chaveamento das portas lógicas internas ao roteador (*ESbit*, *switching bit*, *onde ESbit* = $\varepsilon_{rout} + \varepsilon_{sch} + \varepsilon_{cf}$) e para os canais de comunicação (*ELbit*, *link bit*). O consumo de energia de cada canal de comunicação é diretamente proporcional ao tamanho deste.

Assumindo η como o número de roteadores pelos quais os pacotes de uma mensagem devem passar desde o roteador fonte (r_i) até o destino (r_j), o consumo de energia para as topologias Mesh e Torus pode ser determinado como:

$$Ebit_{ij\ MESH} = \eta\,(ESbit + EBbit) + (\eta - 1)\,ELbit \qquad (7.1)$$

$$Ebit_{ij\ TORUS} = \eta\,(ESbit + EBbit) + (2 \text{ or } 1)\times(\eta - 1)\,ELbit \qquad (7.2)$$

Para a estimativa de energia duas considerações são feitas: (i) a área do roteador é insignificante frente à área total do processador conectado a este; e (ii) o layout da rede é implementado de tal forma que todos os processadores e RAs são desenhados em quadrados e com a mesma dimensão. A figura 7.4 ilustra essas idéias, onde se verifica a realização de topologias Mesh e Torus. Assumindo-se que a área de um roteador é muito menor do que do processador ($l_r << l_\tau$) então o tamanho de um canal de comunicação é $l_{tm} = l$ ou $l_{tM} \cong 2l$ para a topologia Torus, como pode se verificar na figura 7.4. É por este motivo que o calculo de *ELbit* para a topologia Torus pode ser realizado considerando-se tamanho 2 ou 1 (equação 7.2).



Topologia Mesh Topologia Torus

Figura 7.4: Visão parcial das topologias mesh torus para NoCs

Assumindo-se n_{ij} como o número total de bits a ser enviado em uma mensagem do roteador fonte r_i para o destino r_j, o consumo de energia para o envio de todos os bits da mensagem, para as topologias Mesh e Torus é dado por: $Ebit = n_{ij} \times Ebit_{ij}$.

Finalmente, o consumo total de energia de uma aplicação contendo *k* mensagens

em seu comportamento de comunicação é dado pela equação 7.3:

$$EDynNoC = \sum_{q=0}^{k} Ebit_q \qquad (7.3)$$

O desempenho é calculado em ciclos durante a simulação do comportamento de comunicação da aplicação alvo sobre a rede analisada e corresponde à latência de cada mensagem. Para que o valor de possa ser apropriadamente determinado para a função custo, durante a simulação, a parte de controle do *framework* para NoCs utiliza a *interface* de controle para obter o número de ciclos de cada mensagem na rede. Para que essa operação possa ser executada durante a simulação (tempo de execução) é necessário que a parte de controle opere no metanível, como comentado no capítulo 5, sobre o modelo de programação adotado. Dessa forma, a parte de controle consegue extrair o desempenho de cada pacote através dos valores obtidos para RAT (RA Time), sendo RAT = $\tau_{rout[i]} + \tau_{sch[i]} + \tau_{buf[i]} +, \tau_{cf[i]}$, da equação 3.5.

Finalmente, a área é determinada através da pré-síntese de cada componente arquitetural de roteadores em alguma tecnologia específica. Dessa forma, é possível determinar-se o valor de *A*, sendo $A = \alpha_{rout[i]} + \alpha_{sch[i]} + \alpha_{buf[i]} +, \alpha_{cf[i]}$, da equação 3.6.

7.4 Aplicações Testadas

Para a análise e otimização de arquiteturas de processamento foram consideradas aplicações hipotéticas, com mensagens geradas aleatoriamente, além da Transformada Rápida de Fourier de 16 pontos (*Fast Fourier Transform*, FFT) - como especificada em [QUI 94] – Cálculo de Integração pelo método de Romberg [ROM 2001] e Segmentação de Imagens. Nessa aplicação, a imagem é dividida em *n* blocos, executados concorrentemente. A informação relativa à cor de cada pixel em cada bloco é armazenada em uma memória. Os dados de cada bloco são então enviados para *n* processadores – conectados em uma grelha - para serem executados. Cada processador identifica a cor de cada *pixel* de seu bloco da imagem e em seguida, comunica-se com os seus vizinhos da “esquerda” e “acima”, para verificar se os seus blocos possuem pixels adjacentes com a mesma cor. Os resultados são enviados para um processador central, o qual elimina redundâncias e escreve os resultados na memória. Para essa aplicação, os experimentos foram conduzidos para 16 processadores, mais o processador central e a memória em uma topologia do tipo Mesh 5x4.

Para as demais aplicações, as redes Mesh e Torus foram modeladas com tamanho igual a 4x4 roteadores e a Árvore Gorda, 2x4, com 16 terminais.

7.5 Otimizações Consideradas para Arquiteturas de Comunicação

Para as avaliações sobre comportamentos de comunicação, um framework para a realização de barramentos e NoCs foram definidos segundo as regras definidos no modelo de programação, respeitando a especificação de componentes de comunicação de acordo com a definição 3.4.

Certas situações demandam a busca por soluções otimizadas que consideram o comportamento de mais de um componente simultaneamente, o que cria um grande espaço de busca no projeto desse tipo de plataforma. Soluções que procuram automatizar a concepção dessas plataformas devem seguir um fluxo de projeto que contemple um método sistemático na abordagem dos componentes durante a busca por uma arquitetura otimizada. Portanto, as ferramentas aqui discutidas seguem os princípios estabelecidos pelo fluxo de projeto do método como mostrado no capítulo 4.

Para o processo de otimização arquitetural, primeiramente deve-se listar quais as variáveis pertinentes para a concepção da semântica das arquiteturas de comunicação e que podem conduzir a um estado otimizado em relação às restrições de projeto. A semântica das arquiteturas define o comportamento de cada componente arquitetural. Por exemplo, o comportamento de um roteador é determinado pela semântica da variável que define o tipo de roteamento utilizado por este.

Em seguida, deve ser verificado o impacto de cada variável no processo de otimização da arquitetura, bem como as relações de dependência semântica entre elas. Isso é importante devido ao grande espaço de busca, o que leva à adoção de soluções parciais (a análise de algumas variáveis apenas) até que se chegue a uma arquitetura totalmente otimizada. Por isso, é importante que se considere a influência semântica que variável exerce sobre as demais. Por exemplo, caso a otimização seja centrada em políticas de escalonamento para os árbitros de RAs em NoCs, deve-se verificar quais (os possíveis) efeitos que determinadas políticas de escalonamento possam exercer sobre roteadores ou sobre o controle de fluxo. Isso deve ser realizado para manter-se a consistência semântica entre os componentes, pois certos comportamentos somente podem ser alterados em um componente se o mesmo for realizado nos demais. Além disso, essa informação ajuda a delimitar o espaço de busca em duas direções:

1. Indicando quantos componentes (ou as variáveis que os identificam semanticamente) devem ser considerados simultaneamente durante a busca por uma arquitetura otimizada; e

2. Indicando se determinadas variáveis não exercem influencia na otimização.

Uma vez identificados os comportamentos dos componentes arquiteturais, deve-se determinar a natureza das operações necessárias para alterar o comportamento desses componentes a fim de que estes conduzam a um estado otimizado para a plataforma. Uma vez definidas as operações, deves-se encontrar alguma solução algorítmica que as possa executar automaticamente, como uma ferramenta de apoio ao projeto.

Para plataformas de comunicação podem-se destacar as seguintes variáveis, cuja semântica determina soluções arquiteturais:

- Para barramentos, política de escalonamento para os árbitros; e
- Para NoCs, topologia e componentes internos aos RAs.

Cada uma dessas variáveis admite configuração individual ou em conjunto com as demais, uma vez que as operações que executam são ortogonais entre si. Por exemplo, operações de roteamento em NoCs são independentes de operações de escalonamento; a política de prioridades empregada por um árbitro na escolha do buffer que irá enviar pacotes de dados não é afetada pelo motivo que levou o roteador a escolher esse árbitro. Como outro exemplo, verifica-se que o controlador de fluxo atende a qualquer solicitação que lhe é feita, independentemente dos instantes de tempo nos quais árbitros enviam pacotes, e assim por diante.

No entanto a ortogonalidade entre operações de comunicação pode somente ser expressa nos níveis de abstração de mensagens e transações; no nível RT existe dependência entre operações. Por exemplo, a política de prioridades adotada por um árbitro é dependente do tipo de MUX utilizado em sua implementação, no nível RT.

Os problemas para otimização de plataformas de comunicação então, referem-se à abordagem individual ou em grupos dessas variáveis, na busca por uma semântica para cada uma, que leve a uma solução otimizada.

7.5.1 Posicionamento dos Processadores às Portas Locais de Roteadores de Redes Intrachip

Mesmo sendo as operações de comunicação e processamento ortogonais, admitindo, portanto, explorações arquiteturais independentes, uma aplicação sempre será implementada por essas duas classes de operações. Dessa forma, a origem e o destino das comunicações – conseqüência de operações de transformação de dados – devem ser consideradas nas análises de comunicação, uma vez que o comportamento de comunicação é determinado em função delas (o comportamento de comunicação é definido no nível de sistemas como observado no capítulo 4).

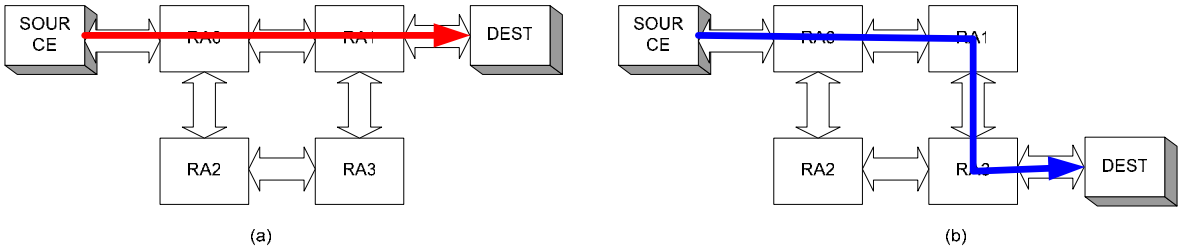


Figura 7.5: O Problema do Posicionamento dos Processadores em NoCs

Quando NoCs são utilizadas para a implementação da arquitetura de comunicação é importante que se considere as posições dos processadores fonte e destino das mensagens na busca por uma arquitetura otimizada. Isso ocorre porque as posições nas quais os processadores são conectados aos RAs determina o número de canais de comunicação por onde a mensagem deve percorrer até chegar ao destino. Essa situação é ilustrada na figura 7.5.

Pode-se verificar na figura 7.5 que, dependendo da posição dos processadores fonte e destino nas comunicações de aplicações, o número de canais de comunicação utilizados pela mensagem pode variar; enquanto que na figura 7.5(a) os pacotes da mensagem necessitam trafegar por 3 canais de comunicação na NoC, a mensagem ilustrada na figura 7.5(b) necessita de 4 canais de comunicação para chegar ao destino. Mesmo através desse exemplo pequeno pode-se perceber facilmente que, devido à escalabilidade inerente das NoCs, para caso de redes grandes, a posição dos processadores pode assumir grande influência para a contenção das comunicações, pois pode levar várias mensagens a competirem pelos mesmos canais de comunicação. Consequentemente, o posicionamento dos processadores nos terminais locais em NoCs deve também ser considerado como uma variável do espaço de busca para plataformas arquiteturais de comunicação otimizadas.

A otimização do posicionamento dos processadores em NoCs pode ser implementada através da operação de *mapeamento*, como especificado na definição 3.10.

Como discutido no capítulo 3, a operação de mapeamento possui natureza complexa, o que demanda o uso de uma solução heurística para a sua realização, sendo sugerido para tal, o uso do algoritmo de busca tabu. A implementação da operação de mapeamento através do algoritmo de busca tabu pode ser obtida atribuindo-se à semântica do conjunto de "recursos" *S*, o conjunto de elementos de processamento a serem considerados para o sistema: $S_{pl[i]} = p_{[i]}$, $i = 1,..., np$; $p \in$ PROC, da definição 3.6:

s_{pl} (y_1) = [$s_{pl[10]}$, $s_{pl[3]}$,...,$s_{pl[np]}$], onde $y_1[0] = s_{pl[10]}$ significa que o processador “10” está conectado ao RA “0” para a solução “y_1”.

De uma maneira geral, pode-se concluir que o posicionamento dos processadores é questão fundamental quando da otimização de arquiteturas de redes, pois a otimização dos componentes internos aos RAs poderá ser prejudicada caso a posição dos processadores implique em latências ou consumo de energia maiores. Isso pode ser explicado pelo fato de que todo comportamento de comunicação reside na geração de mensagens, o que é determinado pelas operações que executam em processadores. Como conseqüência, a avaliação dos componentes internos dos RAs, bem como das topologias, consideram como pressuposto a otimização em conjunto com o posicionamento dos processadores, implementada como uma operação de mapeamento.

Por outro lado, é importante perceber que, em algumas situações, a otimização do posicionamento dos processadores deve ser realizada em conjunto com a otimização do comportamento de outros componentes. Considere por exemplo, a situação onde três processadores de um total de dez, constituem-se da fonte e destino do maior tráfego de uma aplicação; apenas posicionando esses processadores próximos em uma NoC, de modo que as mensagens trocadas por eles não trafeguem por mais que um ou dois canais de comunicação, não necessariamente levará a uma situação onde todas as latências estejam satisfatórias. Pode acontecer que determinadas mensagens necessitem serem priorizadas em função de sua urgência. Nessa situação a variável que determina o posicionamento dos processadores deve ser considerada em conjunto com a variável que determina a política de escalonamento para os árbitros.

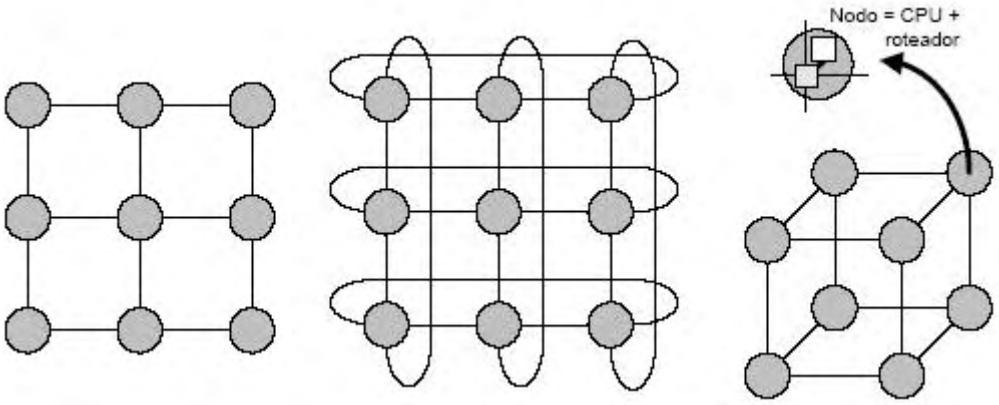
No entanto, o fato de se considerar mais de uma variável simultaneamente para a análise torna a tarefa de otimização extremamente complexa, devido à grande quantidade de comportamentos (para os componentes) que podem ser combinados em cada solução. Por exemplo, no caso de a otimização da posição dos processadores ser combinada com a otimização de políticas de escalonamento, para uma rede 4x4, diferentes políticas de arbitragem podem ser avaliadas em cada uma das 16! possíveis soluções para o posicionamento dos processadores.

7.5.2 Avaliação de Topologias

Outra otimização que pode ser realizada sobre NoCs diz respeito às possíveis topologias que podem ser empregadas para determinar caminhos para o envio das mensagens. Na literatura são encontradas diversas possibilidades topológicas para NoCs [ZEF 2002]. Dentre todas essas possibilidades podem-se classificar as topologias entre *diretas* e *indiretas*.

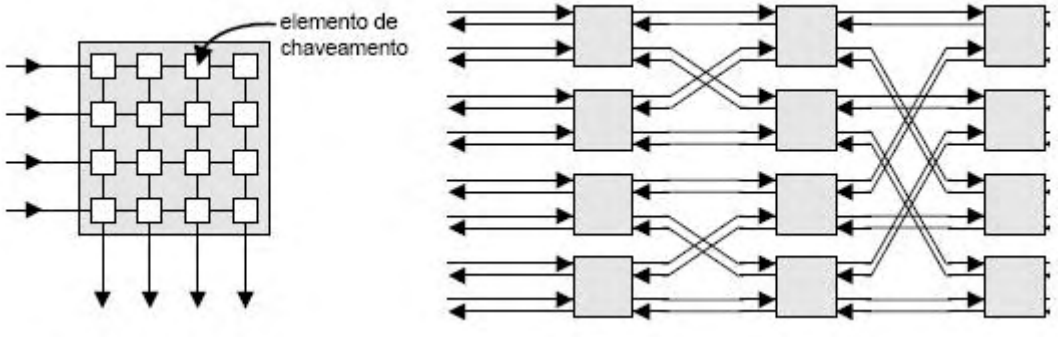
Topologias diretas (ou regulares) possuem caminhos diretos entre todos os RAs que compõem a rede, de modo que cada roteador pode receber e gerar mensagens a partir de um elemento de processamento, conectado à sua porta local. Por outro lado, topologias indiretas possuem RAs dedicados exclusivamente a receber e re-enviar pacotes de mensagens, formando caminhos indiretos entre RAs fonte e destino. Somente alguns RAs são conectados a elementos de processamento.

Na figura 7.6 são ilustradas algumas topologias diretas, como Grelha, Torus e Hipercubo, enquanto que na figura 7.7, as topologias indiretas Crossbar e Multiestágio.



(a) **(b)** **(c)**

Figura 7.6: Redes Diretas: (a) Grelha 2-D; (b) Torus 2-D; (c) Hipercubo 3-D



(a) **(b)**

Figura 7.7: Redes Indiretas: (a) Crossbar 4x4; (b) Multiestágio 8x8 bidirecional

Devido à grande possibilidade de topologias por momento optou-se para este trabalho em restringir a avaliação para as topologias diretas Grelha ou Mesh e indiretas do tipo Árvore Gorda, o que cobre essas duas classes de topologias.

A topologia Mesh é importante de ser considerada, uma vez que constitui-se das mais abordadas em trabalhos encontrados na literatura sendo portanto, a mais provável de ser implementada comercialmente. Isso se deve ao fato de que a sua topologia 2-D é a mais adequada para as tecnologias de fabricação de CI atuais (planares) e por ser bastante regular. Se o *layout* do sistema for organizado de modo a que cada conjunto (roteador + processador) seja implementado em blocos quadrados e regulares, o fato dos canais de comunicação da Mesh terem o mesmo comprimento facilita o projeto das linhas de metal, por causa dos *buffers*.

7.5.3 Políticas de Escalonamento

Políticas de escalonamento em NoCs são realizadas por árbitros, os quais estabelecem regras para determinar o instante de tempo em que cada pacote de cada mensagem é enviado. Uma política de escalonamento bastante empregada em NoCs – chamada Round-Robin - possui como regra, a divisão do tempo entre as mensagens de maneira igual. Dessa forma, é possível garantir que cada mensagem possui a mesma chance na competição por recursos, como canais de comunicação. Visto de outra forma, isso quer dizer que, quando empregada à política de escalonamento *round-robin*, esta

não é o fator preponderante na determinação do desempenho e consumo de energia das mensagens.

Por outro lado, diferentes políticas de escalonamento podem ser avaliadas, como uma tentativa de fazer com que determinadas mensagens da aplicação alvo, possa atender às suas restrições de projeto. Políticas de prioridades podem ser concebidas em função do tipo das mensagens, do seu tamanho, do número de pacotes que faltam ser enviados, pela distância até o destino, etc.

Outra questão importante sobre políticas de escalonamento refere-se ao comportamento heterogêneo, esperado para as aplicações alvo de SoCs. Quanto a políticas de escalonamento, entende-se por heterogeneidade, o fato de mensagens com diferentes tipos ocorrerem durante a execução da aplicação, em *diferente* número. Por exemplo, em um determinado tempo, durante a execução da aplicação, circulam pela rede 100 mensagens de tipo "1" e 20 de tipo "2". No entanto, em algum instante de tempo mais adiante (30000 ciclos, por exemplo), pode acontecer à situação inversa: um número maior de mensagens do tipo "2" circulando pela rede. Por este motivo, é interessante que políticas de escalonamento possam ser dinamicamente configuradas. Com isso, tenta-se evitar que um determinado tipo de mensagem cause um grande aumento nas contenções da rede, o que pode acontecer se mensagens cujo tipo mais ocorre em determinado momento, possuem prioridade mais baixa.

Para que árbitros possam ser reconfigurados em tempo de execução, é necessário acrescentar um *vetor de prioridades* (VP), o qual estabelece a *seqüência* de prioridades a ser considerada pelo árbitro. Além do vetor "VP", é necessário que se associe um contador, para determinar o intervalo de tempo para a troca de prioridades. Tanto a ordem de prioridades contidas em "VP", quanto o intervalo podem ser sintonizados para cada aplicação.

Para adaptar o problema de encontrar uma política de escalonamento otimizada ao algoritmo de busca tabu, basta tornar o conjunto de recursos S igual à política de escalonamento para cada árbitro da rede: $S_{sch[i]} = r_{i.sch}$; i = 1, ...,*nra*; nra: número de roteadores da rede.

No entanto, para que se consiga uma busca efetiva por uma arquitetura otimizada, é necessário que se considere a busca por políticas de escalonamento em conjunto com a posição dos processadores, como já mencionado acima, na seção 7.4.1. Nesse caso, o algoritmo busca tabu deve considerar dois conjuntos de recursos: o conjunto S_{sch} com as políticas de prioridades, além do conjunto S_{pl}; $S_{pl[i]} = p_{[i]}$, i = 1,..., np; $p \in$ PROC, com as posições dos processadores.

7.5.4 Redes Heterogêneas

Outra possibilidade a ser testada para plataformas de comunicação baseadas em NoCs refere-se à configuração de redes *heterogêneas*. Redes heterogêneas podem ser conseguidas de duas maneiras:

1. Com comportamento heterogêneo; ou
2. Com estrutura heterogênea.

No primeiro caso, os componentes internos aos RAs possuem diferentes comportamentos diferenciados. Por exemplo, pode-se pensar em uma rede onde com árbitros contendo diferentes políticas de escalonamento ou diferentes organizações de memória.

Nesse caso, soluções parciais dentro do espaço de busca podem permitir a configuração *simultaneamente* do comportamento de roteadores, árbitros e *buffers* de cada RA. Como exemplo de aplicação, pode-se pensar em roteadores que estão no caminho de mensagens mais urgentes com tamanho para os *buffers* maiores ou com políticas de escalonamento particulares.

No segundo caso, todos os roteadores da rede possuem o mesmo comportamento, sendo, no entanto, implementados por estruturas de HW diferenciadas, visando por exemplo, o aumento de desempenho ou a redução do consumo de energia.

Em qualquer um desses casos, o processo de otimização arquitetural possui complexidade $O((x^z)^y)$, onde *x*: número de possíveis comportamentos que cada componente pode assumir; *z*: número de componentes internos de RAs e *y*: número de RAs. Considerando-se que cada RA possui 4 componentes (da definição 3.5) que podem ser configurados e que cada um pudesse assumir dois comportamentos distintos (políticas de escalonamento, por exemplo), para uma rede 4x4, existiriam $(2^4)^{16} = 18 \times 10^{18}$ alternativas a serem consideradas no espaço de busca. A avaliação de cada uma destas alternativas implica na simulação de todo o comportamento de comunicação da aplicação alvo sobre a NoC. Entretanto, normalmente somente um ou dois componentes serão avaliados, o que reduz drasticamente o espaço de busca.

À medida que redes maiores ou mais componentes forem observados para compor a heterogeneidade da rede, o espaço de busca cresce exponencialmente. Mesmo supondo-se que a simulação da aplicação possa ocorrer em alto nível de abstração – mensagens, por exemplo – e que máquinas com grande desempenho estejam disponíveis para a realização da simulação, a implementação de um algoritmo de busca exaustiva para a procura por uma solução otimizada, tende a possuir tempo proibitivo.

Nessa direção, a adoção de uma heurística para a exploração do espaço de busca torna-se imprescindível para que diversas configurações arquiteturais possam ser exploradas em tempo hábil.

O problema fica ainda mais complexo se o problema do posicionamento dos processadores for considerado simultaneamente à exploração da heterogeneidade da rede. Nesse caso, a complexidade para a otimização de componentes arquiteturais heterogêneos ($O(x^y)$) é acrescida a complexidade da operação de mapeamento ($O(N!)$, onde N: número de processadores, como discutido na definição 3.10. Mesmo implicando em um aumento de complexidade, o problema do posicionamento dos processadores deve ser considerado, pois nos processadores são geradas todas as mensagens da aplicação, como já discutido na seção 7.4.1.

Uma vez que a heurística baseada no algoritmo de busca tabu é empregada para a implementação da operação de mapeamento, optou-se por adaptá-la para a busca de soluções otimizadas, que exploram o posicionamento dos processadores e a heterogeneidade da rede simultaneamente.

Isso pode ser obtido acrescentando-se um segundo conjunto de recursos ao algoritmo busca tabu, $S_{noc\_het} = \{ra_{het[0]}, ra_{het[1]}, ..., ra_{het[n]}\}$, onde $ra_{het[x]} \in$ RA (da definição 3.5) constitui um comportamento específico para o roteador *x*; e *n*: número de RAs da rede. Cada $ra_{het[x]}$ pode então, possuir um comportamento específico para cada um de seus componentes internos: $ra_{het[0]} = \{r_{.rout = XY}\}$; $ra_{het[1]} = \{r_{.rout = west\_first}\}$.

Uma vez que o algoritmo passa a trabalhar com dois conjuntos de recursos (S_{pl} e S_{noc_het}), o comportamento da função OPTIMUM, a qual gera as vizinhanças dentro do espaço de busca, necessita ser configurado para determinar como que as posições dos

conjuntos serão trocadas. As vizinhanças para redes heterogêneas são geradas segundo duas estratégias de otimização, explicitadas na seção 7.6, abaixo.

Como estudo de caso, foi testada a rede heterogênea *SoCINhet*, a qual possui estrutura heterogênea, sendo composta pelos roteadores RASoC [ZEF 2003] e Tonga [CAR 2004] e Mago. Estes roteadores possuem diferenças estruturais em suas arquiteturas, as quais implicam em diferenças em termos de consumo de energia, desempenho e área.

O roteador RASoC é projetado para a criação de redes do tipo Mesh, com 5 portas bidirecionais (local, norte, sul, leste e oeste) com as seguintes características:

- Roteamento estático "XY";
- Arbitragem do tipo *Round-Robin* distribuída – existe um árbitro para cada porta de saída;
- Organização da memória do tipo fila (FIFO); e
- Controle de fluxo implementado com protocolo *handshake*.

Tonga e Mago são variações arquiteturais do roteador RASoC, visando redução de área e consumo de energia, pela redução do número de canais de comunicação. Os canais implementados são compartilhados entre as portas norte/sul e leste/oeste no roteador Tonga e entre todos os canais de comunicação (menos o da porta local) no roteador Mago.

A figura 7.8 exemplifica as arquiteturas dos roteadores Rasoc (a), Tonga e Mago (c).

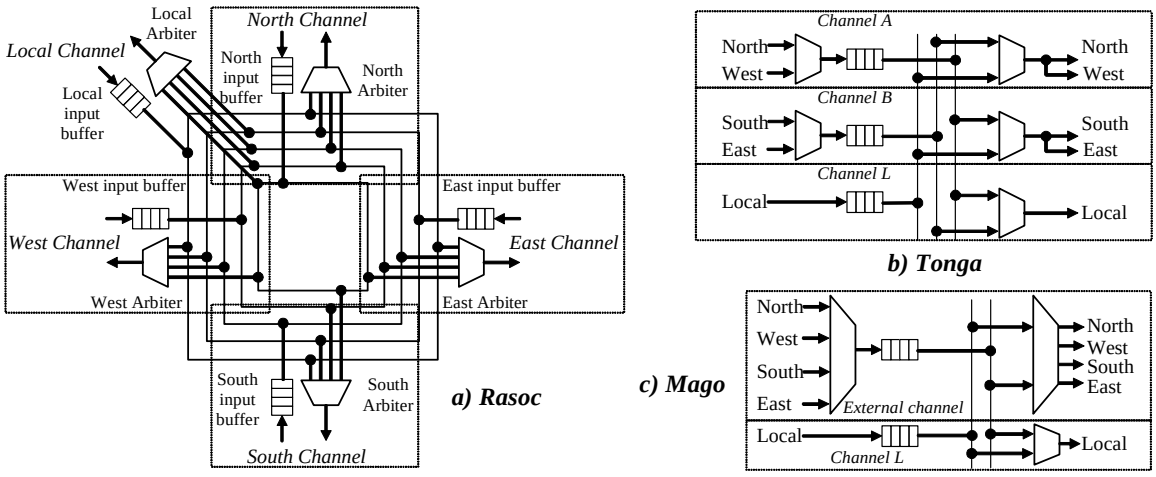


Figura 7.8: Arquiteturas dos Roteadores Rasoc, Tonga e Mago

A multiplexação dos canais de comunicação permite a redução de área e do consumo de energia. O principal conceito referente aos roteadores Tonga e Mago refere-se ao uso de multiplexadores para gerenciar o envio dos pacotes entre as portas que compartilham canais de comunicação. No roteador Tonga os canais "norte" e "oeste" são compartilhados como *canal A*, enquanto que os canais "sul" e "norte", como *canal B*. No roteador Mago, todos os canais são compartilhados. Em todos os roteadores, a porta local possui um canal exclusivo. O controle dos multiplexadores é realizado por um contador, o qual estabelece um tempo para cada porta enviar pacotes.

A redução do consumo de energia e da área é obtida pela redução do número de buffers em cada roteador e do tamanho dos multiplexadores. Por outro lado, o compartilhamento dos canais implica no aumenta da latência. Dessa forma, o máximo desempenho pode ser conseguido com uma rede composta somente por roteadores Rasoc, enquanto que o menor consumo de energia e área, com uma rede composta por roteadores Mago, ou seja: $\tau_{mago} < \tau_{tonga} < \tau_{rasoc}$; $\varepsilon_{mago} < \varepsilon_{tonga} < \varepsilon_{rasoc}$ e $\alpha_{mago} < \alpha_{tonga} < \alpha_{rasoc}$, da definição 3.5.

Os três roteadores foram descritos em VHDL e sintetizados para a tecnologia 0.35µm através da ferramenta Leonardo Spectrum. Todos os buffers possuem quatro palavras. A tabela 7.2 mostra os resultados de síntese obtidos com diferentes tamanhos para os canais de comunicação, para área, consumo de energia e tempo de execução. A área foi obtida diretamente da síntese física. O consumo de energia foi obtido através de simulação elétrica para entradas de dados aleatórias, com VDD de 3.3 volts. A medida do tempo de execução reflete a média do tempo total de execução de aplicações hipotéticas (com mensagens geradas aleatoriamente) de redes homogêneas, para cada tipo de roteador.

Tabela 7.2: Área, consumo de energia e tempo de execução para os roteadore Rasoc, Tonga e Mago

Canal de Comunicação (bits)	Característica	Roteadores		
		Rasoc	Tonga	Mago
8	Área (μm^2)	157,266	100,937	75,659
16		248,539	154,809	111,288
8	Consumo médio de potência (mW)	7.61	5.09	3.83
16		10.48	6.98	5.28
8 / 16	Tempo médio de execução (ciclos)	268,012	313,570	443,947

Os resultados da tabela 7.2 confirmam os compromissos que podem ser obtidos – em termos de consumo de energia, área e desempenho – ao se implementar uma rede heterogênea composta por roteadores com estruturas arquiteturais diferenciadas. À medida que mais canais de comunicação são compartilhados, menor o desempenho, mas por outro lado, maior a redução de área e do consumo de energia.

Uma vez que se pode esperar que nem todas as aplicações necessitem utilizar toda a largura de banda oferecida por uma NoC, pode-se pensar em ferramentas para automaticamente encontrar o compromisso que melhor atenda as restrições da aplicação alvo.

7.6 Experimentos com Arquiteturas de Comunicação

Como um primeiro experimento, os componentes arquiteturais foram reconfigurados separadamente, com o objetivo de testar-se o impacto exercido por cada um no desempenho e na latência do comportamento de comunicação da aplicação. Em seguida, foram testadas as variações de desempenho causadas pela reconfiguração conjunta de componentes arquiteturais, a fim de se descobrir qual a plataforma mais otimizada para cada aplicação.

A figura 7.9 mostra o desempenho e a latência obtidos para diferentes topologias de redes chaveadas: as diretas Mesh e Torus e a indireta, Árvore Gorda.

Em relação à política de roteamento, as redes Mesh e Torus utilizam roteamento estático XY (para evitar *livelock*), enquanto que a Árvore Gorda utiliza roteamento adaptativo quando mensagens são enviadas para o segundo nível, ou estático, nos demais casos. Os resultados são apresentados em número médio de ciclos para o desempenho e a latência de todas as mensagens das aplicações testadas.

Como pode ser verificado, o melhor desempenho foi obtido pela topologia Torus. Esse resultado vem de encontro ao maior paralelismo oferecido pelo número maior de canais oferecidos. Pelo mesmo motivo, a execução seqüencial das mensagens no barramento obteve um desempenho em torno de 2 vezes inferior, em comparação com as redes chaveadas.

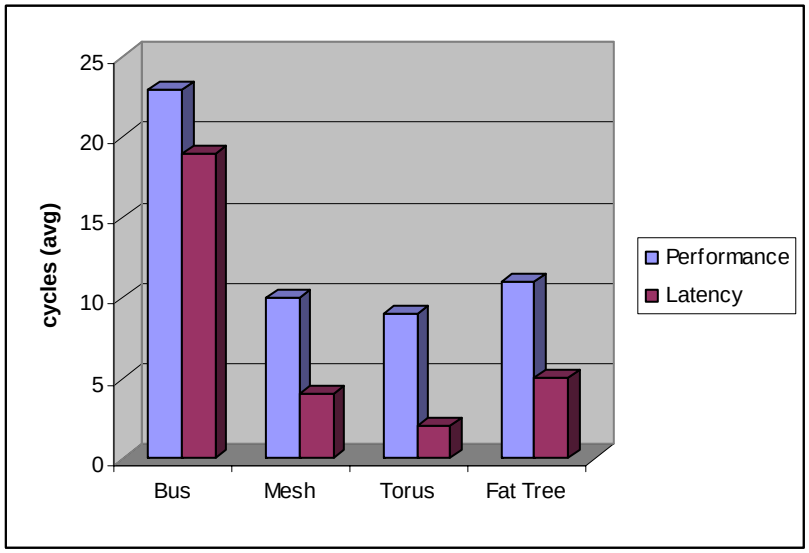


Figura 7.9: Desempenho e Latência para diferentes Topologias de Redes Chaveadas

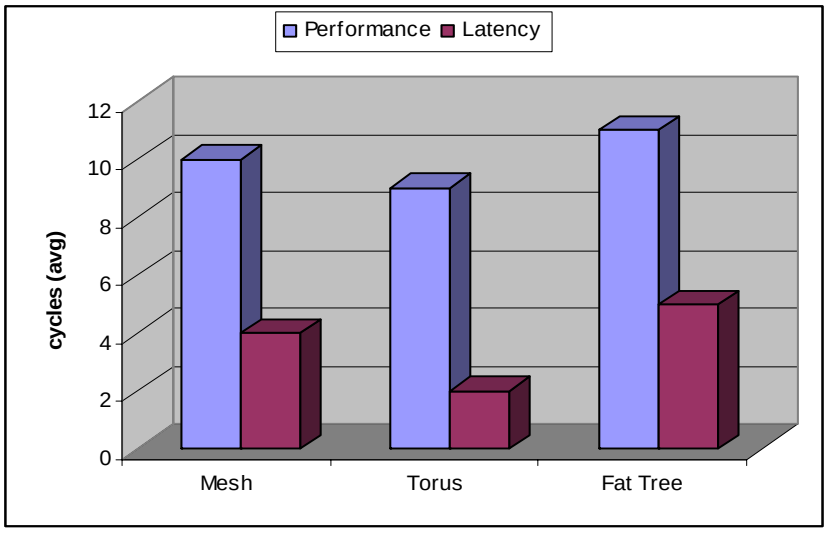


Figura 7.10: Desempenho e Latência para Redes com “Canais Virtuais”

A Árvore Gorda é penalizada pela distribuição irregular das mensagens, as quais necessitam utilizar o segundo nível de roteadores para alcançar o destino. Entretanto, ressalta-se que esses resultados correspondem à uma média da execução de várias aplicações, ou seja, algumas com maior e outras com menor localidade das mensagens.

Certamente, aplicações com boa localidade na distribuição das mensagens, poderão alcançar melhor desempenho.

A fim de se melhorar o desempenho para as topologias testadas, foi implementada a técnica de canais virtuais. Para tanto, a organização dos *buffers* de cada NoC testada foi alterada do tipo fila (FIFO) para o tipo DAMQ. Os resultados podem ser verificados na figura 7.10.

Como esperado, todas as topologias apresentaram melhor desempenho – em torno de 20% - através do uso de canais virtuais. É interessante observar que o aumento de desempenho é proporcional em todas as redes, o que indica que a técnica de canais virtuais consegue aproveitar a concorrência oferecida por todas as topologias.

Na implementação da heurística do algoritmo Busca Tabu para o problema do posicionamento dos processadores, optou-se pelo emprego de uma função aleatória para a geração das permutações entre as posições dos processadores. Com uma distribuição aleatória, objetiva-se otimizar aplicações onde não ocorram grandes concentrações de dados no envio de mensagens, ou seja, funções que trocam uma quantidade de informações muito maior do que as demais. Mesmo assim, a "aleatoriedade" pode otimizar mensagens grandes, uma vez que estas podem (aleatoriamente) serem posicionadas em processadores separados por apenas um canal de comunicação.

A figura 7.11 mostra os resultados obtidos através do posicionamento otimizado dos processadores nos terminais locais das NoCs. Visando uma maior otimização, a posição dos processadores foi realizada em arquiteturas com canais virtuais, uma vez que essa organização de *buffer* leva a um melhor desempenho, como pode ser verificado na figura 7.10.

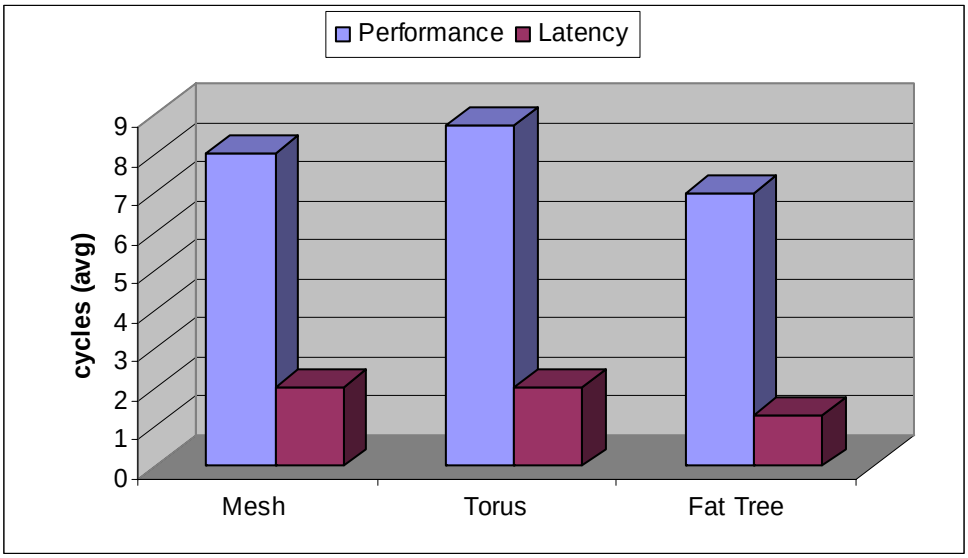


Figura 7.11: Otimização da Posição dos Processadores na Rede

A topologia que mais pôde tirar proveito da posição otimizada dos processadores é a Árvore Gorda. Isso pode ser explicado pelo fato de processadores contendo mensagens com maior tráfego serem alocados para o mesmo terminal da rede, o que implica em comunicações sobre apenas um canal de comunicação – latência média próxima de "1". A latência média para a Árvore Gorda pôde ser reduzida em aproximadamente 50% com a posição otimizada dos processadores na rede. Devido ao

alto grau de paralelismo oferecido, a topologia Torus foi pouco beneficiada da posição dos processadores.

Uma das características interessantes do algoritmo de Busca Tabu refere-se à sua capacidade de sair de “mínimos locais”. Essa situação é mostrada na figura 7.12, para a latência média obtida para a rede Mesh, em 10 iterações. Observe que a menor latência média foi encontrada no passo 6.

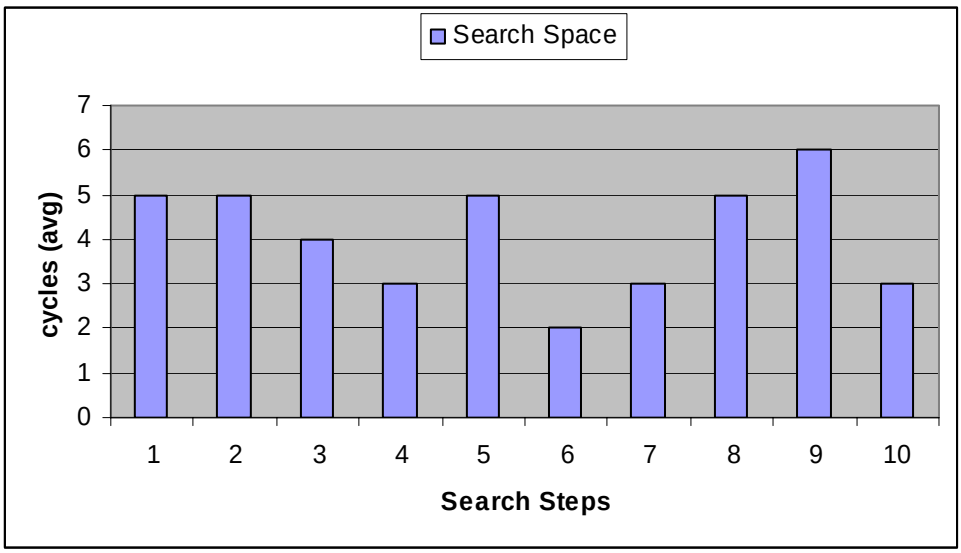


Figura 7.12: Evolução da busca realizada pelo algoritmo Busca Tabu

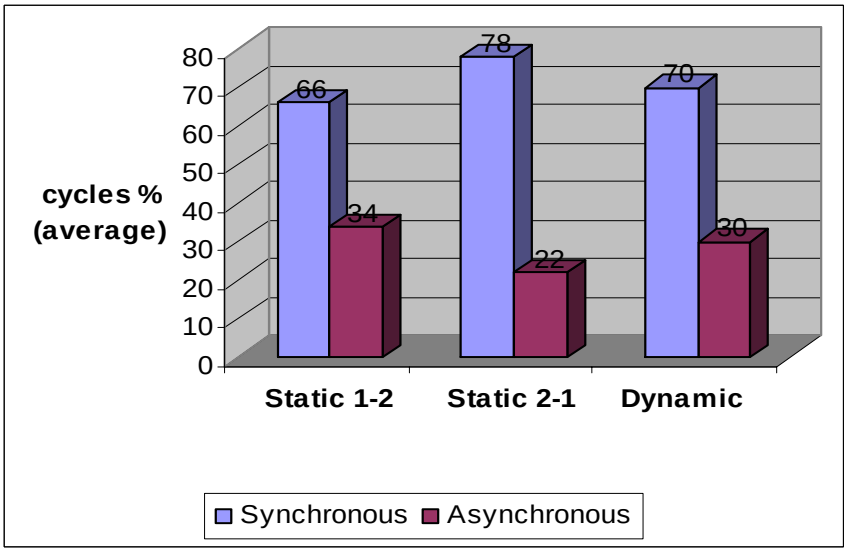


Figura 7.13: Otimização da Política de Escalonamento para uma Aplicação Síncrona

A otimização de *políticas de escalonamento* foi realizada sobre uma rede Mesh 4x4 para dois tipos de aplicações: uma dominada por mensagens síncronas (80%) e outra com equilíbrio (50-50%) entre o número de mensagens síncronas e assíncronas. Com isso, buscou-se avaliar a capacidade que a política de escalonamento pode possuir na otimização de um determinado tipo de mensagem. As figuras 7.13 e 7.14 mostram os resultados de latência média, obtidos para as duas aplicações. Nessas figuras, “1” significa mensagem síncrona e “2” mensagem assíncrona. A legenda 1-2 implica em que as mensagens do tipo “1” possuem prioridade sobre as do tipo “2”.

Os melhores resultados foram obtidos quando as mensagens síncronas foram *estaticamente* priorizadas. Conclui-se então que, devido ao grande número de mensagens síncronas, a heterogeneidade da comunicação não possui um impacto significativo no desempenho. Entretanto, pela observância da heterogeneidade da comunicação na otimização *dinâmica*, a latência média das mensagens assíncronas pôde ser reduzida em 8%, mesmo que este tipo de mensagem corresponda a apenas 20 % do total de mensagens da aplicação. Isso significa que existe uma variação de cada tipo de mensagem no tempo. Nesse caso, a otimização *dinâmica* implica em um compromisso entre os dois tipos de mensagens da aplicação, uma vez que as mensagens assíncronas possuem melhor desempenho ao serem priorizadas.

Na figura 7.14 pode-se observar que, pelo fato de a otimização *dinâmica* ter conseguido diminuir a latência para as mensagens síncronas, os dois tipos de mensagens da aplicação não são uniformemente distribuídos no tempo. Mesmo assim, o desempenho das mensagens síncronas é menor do que as assíncronas devido aos ciclos extras, necessários à sincronização – sinais de controle.

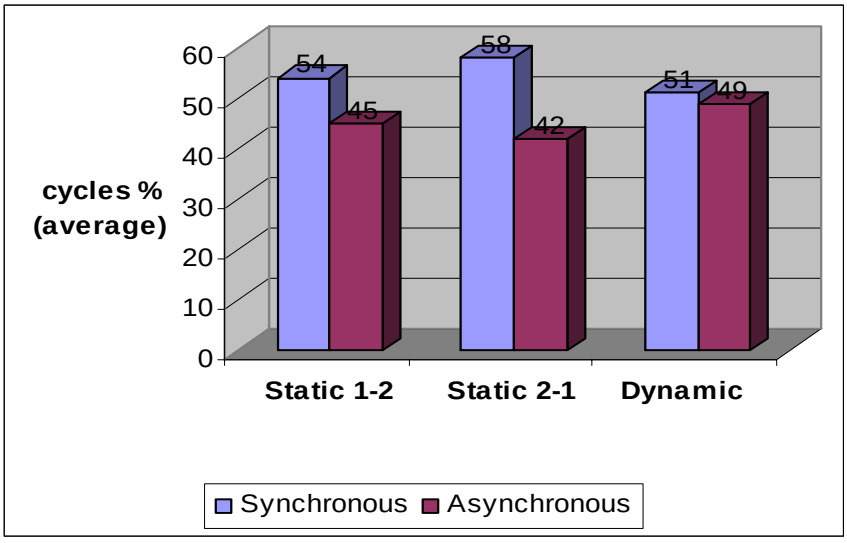


Figura 7.14: Otimização da Política de Escalonamento para uma aplicação balanceada

Como esperado para uma aplicação “balanceada” não existem grandes diferenças de desempenho entre os dois tipos de mensagens considerados.

Finalmente, foi considerada a otimização em redes heterogêneas, utilizando os roteadores Rasoc, Tonga e Mago, como descrita na seção 7.5.4.

Devido às diferenças arquiteturais existentes entre esses roteadores, o que implica na obtenção de diferentes resultados para área, desempenho e consumo de energia, é possível encontrar durante a busca por soluções otimizadas, redes heterogêneas as quais representam compromissos entre essas variáveis. Nessa direção foram adotadas duas estratégias de otimização: a primeira prioriza o consumo de energia, enquanto que a segunda, o desempenho. Como a redução do consumo de energia pelo emprego de roteadores Tonga e Mago implica automaticamente na redução de área, toda redução que se conseguir no consumo de energia se refletirá na redução de área. Portanto, basta calibrar a heurística para buscar soluções otimizadas para energia, para obter também, economia de área.

A primeira estratégia para a otimização de redes heterogêneas estabelece que a aplicação alvo possui restrições de energias mais severas do que de desempenho e a segunda estratégia, o contrário. Assim, na implementação da primeira estratégia, o algoritmo de otimização prioriza o consumo de energia, tentando aumentar o desempenho tanto quanto possível, dentro das restrições de energia. Por outro lado, na segunda estratégia procura-se encontrar uma rede que atenda as restrições de desempenho, tentando minimizar o consumo de energia (e área) ao máximo.

A primeira estratégia foi chamada de *latency priority/energy minimization* (LP/EM) e segunda de *energy priority/latency minimization* (EP/LM).

Na estratégia EP/LM uma rede composta exclusivamente por roteadores Mago é instanciada. O algoritmo de otimização procura então, substituir roteadores Mago por Tonga e Rasoc, a fim de aumentar o desempenho. As substituições ocorrem até que se atingir o limite do consumo de energia estabelecido como restrição para a aplicação alvo.

Já na estratégia LP/EM inicialmente é instanciada uma rede composta exclusivamente por roteadores Rasoc, os quais vão progressivamente sendo substituídos por roteadores Tonga e Mago, a fim de se minimizar o consumo de energia.

É importante perceber que – de acordo com o algoritmo ilustrado na figura 7.9 – as substituições de ambas as estratégias ocorrem em conjunto com a otimização do posicionamento dos processadores, onde: $S_{noc\_het} = \{ra_{het[0]} = \textbf{tonga}, ra_{het[1]} = \textbf{mago}, \ldots, ra_{het[n]} = \textbf{rasoc}\}$.

A intersecção das duas estratégias, como exemplificado na figura 7.15, representa as possíveis soluções para o posicionamento dos processadores (implementado pela operação de mapeamento) e rede heterogênea, composta pelos roteadores Rasoc, Tonga e Mago.

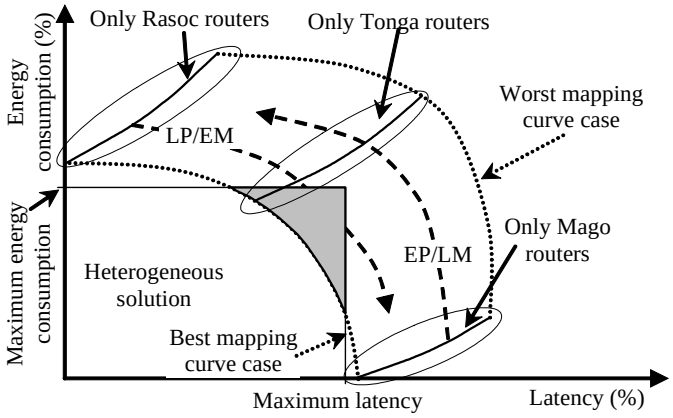


Figura 7.15: Exploração do espaço de projeto para as estratégias EP/LM e LP/EM

Primeiramente, redes homogêneas compostas pelos roteadores Rasoc, Tonga e Mago foram avaliadas, para a aplicação de Segmentação de Imagens. Nesse caso, a otimização se equivale ao posicionamento dos processadores apenas. Os resultados – ilustrados na figura 7.16(a) - mostram o compromisso que pode ser obtido através da construção de redes heterogêneas utilizando os roteadores Rasoc, Tonga e Mago: a rede composta por roteadores Mago somente alcança o menor consumo de energia, a rede composta por roteadores do tipo Rasoc atinge a menor latência, enquanto que a rede Tonga representa um compromisso entre consumo de energia e latência. Os pontos

ilustrados na figura 7.16(a) mostram a exploração do espaço de busca realizada pelo algoritmo busca tabu, para a execução da operação de mapeamento.

A figura 7.16(b) mostra a latência e o consumo de energia para 5 aplicações – FFT, Método de Integração de Romberg, Segmentação de Imagens, mais das aplicações hipotéticas (App1 e App2) – para a operação de mapeamento. Cada ponto na figura 7.16(b) mostra o melhor posicionamento para os processadores encontrado pelo algoritmo busca tabu. Pode-se perceber na figura 7.16(b) que o compromisso entre latência (rede composta por Rasoc) e consumo de energia (rede composta por Mago) permanece constante para todas as aplicações. Além disso, redes compostas por Tonga representam o compromisso entre latência e energia, atingindo em média, 58% da latência, em relação a uma rede Rasoc, e 50% do consumo de energia, em relação a uma rede Mago. A não linearidade das curvas reflete as contenções decorrentes da execução do comportamento de comunicação das aplicações.

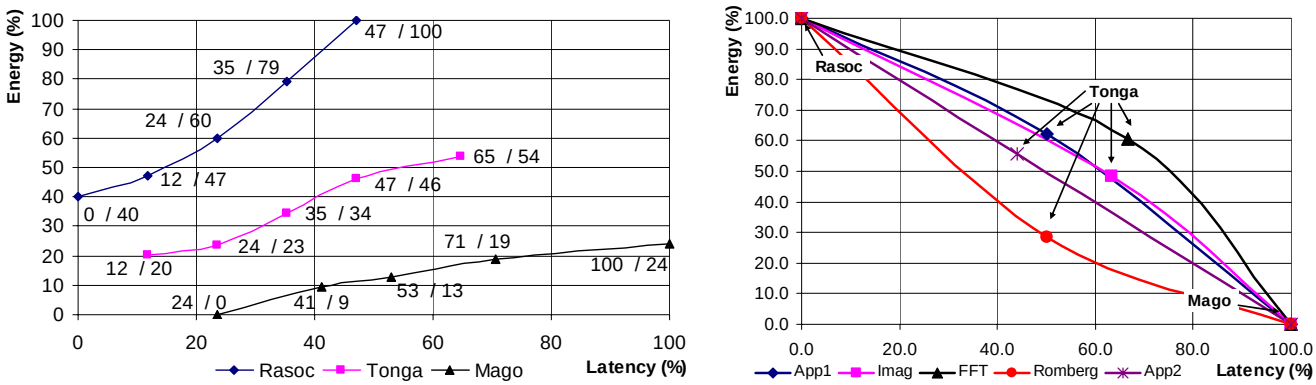


Figura 7.16: (a) Comparação do consumo de energia e latência para três redes homogêneas. (b) Compromisso Energia/Latência para cinco aplicações executando sobre redes homogêneas

A figura 7.17 ilustra compromissos entre latência/consumo de energia entre redes homogêneas e heterogêneas, obtidas a partir da otimização da aplicação para Segmentação de Imagens. Os pontos referentes às três redes homogêneas (Rasoc/Tonga/Mago) mostram o melhor mapeamento encontrado para cada uma. A combinação de diferentes arquiteturas de roteadores para compor uma rede heterogênea – SoCINhet – aumenta o espaço de busca, tornando possível a criação de uma curva de Pareto para a aplicação (ilustrado na figura 7.17). Por exemplo, a rede heterogênea com a composição de 10 Rasoc, 7 Tonga e 3 Mago foi encontrada pelo algoritmo de otimização, com 25% de latência de 88,1% de redução do consumo de energia em comparação com outras soluções (assumindo-se como 100%, a pior solução encontrada). Ainda, em comparação com uma rede composta por roteadores Rasoc apenas, para uma configuração de rede 10 Rasoc, 7 Tonga e 3 Mago obtém uma redução de área de aproximadamente 30%.

Outras soluções também podem ser consideradas, uma vez que a curva ilustrada na figura 7.17 representa a curva de Pareto para consumo de energia e latência, em relação a diversas arquiteturas de rede heterogênea encontradas pelo algoritmo de otimização. Dessa forma, as soluções propostas permitem ao projetista de sistemas encontrar a arquitetura, cujo compromisso "energia$_x$latência" melhor reflita as restrições da aplicação. Resultados semelhantes foram encontrados para outras aplicações

testadas.

Finalmente, outro ponto que merece ser destacado, refere-se ao interessante resultado encontrado pelo algoritmo de otimização proposto: na média, o consumo de energia e a latência encontradas são menores para as redes heterogêneas do que para as homogêneas.

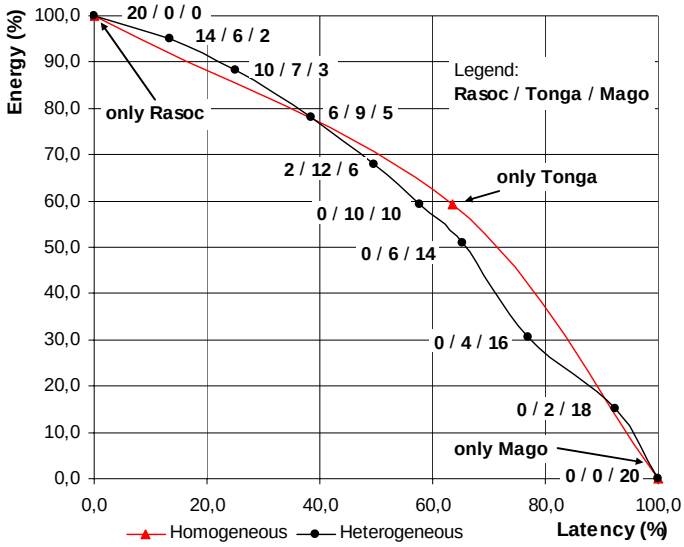


Figura 7.17: Exploração do espaço de projeto para redes heterogêneas

7.7 Conclusões

Os experimentos conduzidos para arquiteturas de comunicação refletem o amplo espaço de projeto que pode ser explorado na busca por soluções otimizadas. Diversos componentes arquiteturais para comunicação foram considerados, desde barramentos até arquiteturas de NoCs. Este tipo de arquitetura está sendo considerada atualmente como uma importante solução para atender as crescentes demandas por largura de banda em aplicações dedicadas distribuídas. Com as experiências realizadas foi possível comprovar o aumento de desempenho proporcionado por estas arquiteturas de redes, em relação aos barramentos.

Nessa direção também foram testadas algumas das várias possibilidades para os componentes arquiteturais de NoCs, como políticas de escalonamento, topologias e redes estruturalmente heterogêneas. Com base nesses experimentos, se pôde constatar o grande potencial que esse tipo de arquitetura de comunicação possui para otimizar restrições de projeto para aplicações dedicadas.

Mesmo que nem todas as possibilidades arquiteturais foram cobertas pelos experimentos, pode-se concluir que é possível sintonizar arquiteturas de NoCs para sistemas multiprocessados, esperados para um futuro próximo, os quais serão compostos por vários elementos de processamento, inaugurando a *gigaera* na concepção de sistemas eletrônicos.

8 PLATAFORMAS ARQUITETURAIS PARA SISTEMAS DEDICADOS COMPLETOS

Componentes para processamento são utilizados para a execução das operações para transformação de dados (lógico/aritméticas) de aplicações dedicadas. Devido à ortogonalidade de operações, estas podem ser executadas em componentes dedicados ao processamento, em comparação com os componentes para comunicação, os quais executam operações para transferência de dados apenas. Entretanto, deve-se tomar cuidado quanto ao conceito "comunicação", uma vez que esta está intimamente relacionada com a *granularidade* da operação considerada. Por exemplo, mesmo em operações lógico/aritméticas existe comunicação, pois os operandos fonte devem ser transferidos (de um banco de registradores ou memória) para componentes processadores (como ULAs por exemplo) e após o processamento da operação, transferidos novamente para o registrador ou a posição de memória destino. Em granularidades ainda menores, pode-se pensar nas comunicações internas aos componentes de ULAs, as quais podem ser implementadas com *pipeline* e assim por diante.

Para o contexto desse trabalho, o qual aborda um sistema multiprocessado completo, comunicações são consideradas para a granularidade do conjunto de instruções de processadores. Dessa forma, comunicações são classificadas como intra-processadores ou inter-processadores.

Comunicações inter-processadores são realizadas através de arquiteturas de comunicação do tipo barramento ou NoCs. Este tipo de comunicação é realizada através de instruções específicas para comunicação, pertencentes ao conjunto de instruções do processador ou através de APIs específicas para comunicação. Neste caso, a API deve conter as instruções necessárias para acessar a arquitetura de comunicação, diretamente ou através de um *wrapper*. *Wrappers* são necessários para compatibilizar protocolos de comunicação.

Por outro lado, comunicações intra-processadores são aquelas realizadas internamente aos processadores ou destes com a sua memória local. Este tipo de comunicação é implementado pelas instruções para movimentação de dados do conjunto de instruções do processador.

Neste capítulo, sistemas multiprocessados dedicados são analisados sob dois enfoques:

1. Particionamento de tarefas em plataformas arquiteturais de processamento; e
2. Impacto de arquiteturas de comunicação e de processamento na distribuição de tarefas.

Para a efetivação da primeira abordagem são analisadas plataformas para processamento, quanto a sua capacidade de atender as restrições de projeto. Para tal, são

analisadas arquiteturas de processadores quanto ao número de unidades funcionais e quanto à implementação de unidades dedicadas para a execução de tarefas computacionalmente intensivas.

Na segunda abordagem, a análise é realizada sobre as relações entre comunicação e processamento, a fim de se avaliar o impacto de cada uma dessas classes arquiteturais para as restrições da aplicação. Mais especificamente, o que se procura estabelecer com esse tipo de análise é o compromisso entre a *complexidade* e o *comportamento* tanto de arquiteturas de processamento quanto de comunicação, que melhor atenda as restrições de projeto da aplicação alvo. Para tanto, primeiramente as tarefas são particionadas de modo a que se obtenha uma arquitetura de processamento – composta (provavelmente) por um conjunto de processadores – em conformidade com as restrições de projeto, para em seguida, sintonizar a arquitetura de comunicação de acordo com essas restrições. Isto é conseguido através da implementação das operações de particionamento e mapeamento.

Uma vez que toda operação lógico/aritmética incorre em comunicação, esta pode ser implementada interna ou externamente a processadores, dependendo da localização no sistema dos operandos fonte e destino. Consequentemente pode-se pensar em um compromisso entre esses dois tipos de comunicação para as restrições das aplicações dedicadas. Caso uma arquitetura de comunicação possua desempenho suficiente, vários operandos podem ser localizados em processadores diferentes, tornando-os mais simples. Isto tem implicação direta na arquitetura do processador, como por exemplo, redução de unidades funcionais ou do conjunto de instruções, visando economia de potência e área.

Uma vez alocadas as tarefas de uma aplicação para processadores, é necessário um serviço de escalonamento para compartilhar o tempo de processamento entre as tarefas alocadas para o mesmo componente de processamento. Serviços de escalonamento são comumentemente implementadas em sistemas operacionais. Portanto, pode-se pensar na geração de sistemas operacionais dedicados à aplicações específicas, com a finalidade de gerenciar o comportamento das tarefas de uma aplicação dedicada.

Um sistema operacional desse tipo normalmente é chamado de Sistema Operacional de Tempo Real – *Real-Time Operating System, RTOS* – pois as tarefas a serem gerenciadas normalmente operam com restrições temporais rígidas. Sistemas Operacionais dedicados devem conhecer o comportamento de cada tarefa, a fim de escaloná-las no tempo apropriado, de acordo com o estado em que se encontram.

Como a arquitetura alvo caracteriza-se por ser um sistema distribuído, pode-se pensar também na geração de sistemas operacionais distribuídos, para o gerenciamento distribuído de tarefas. Nesse caso, o sistema operacional pode escalonar tarefas alocadas para processadores diferentes, podendo inclusive realocá-las, se necessário.

Uma vez que a política de escalonamento pode ajudar a determinar o grau de otimização dos componentes de processamento na execução das tarefas, esta deve ser determinada quando os processadores forem avaliados. Dessa forma, a política de escalonamento faz parte da função custo das ferramentas de avaliação de processadores. No entanto, para simplificar o espaço de busca, por momento, está-se assumindo como política de escalonamento, a *Round-Robin*, uma vez que esta proporciona um tempo igual para cada tarefa, não influenciando assim, diretamente no desempenho de tarefas específicas.

Em relação à comunicação, as tarefas podem trocar dados através do uso de memória compartilhada ou através da troca de mensagens. No primeiro caso, é necessário que o sistema operacional implemente a consistência dos dados, por exemplo, através do uso de semáforos.

No segundo caso, como não há o compartilhamento do espaço de endereços, as mensagens são enviadas através de uma arquitetura de comunicação. Nesse caso, o sistema operacional deve gerenciar as trocas de mensagens para que essas sejam seguras, ou seja, que seja garantido que mensagens chegam ao destino e que não estejam corrompidas. Além disso, sistema operacional deve implementar os protocolos de comunicação apropriados, os quais possibilitam às tarefas enviarem e receberem mensagens através do modelo de computação estabelecido pela aplicação.

As análises realizadas para a geração automática dos serviços de comunicação de um sistema operacional dedicado podem ser implementadas no nível de mensagens, onde podem ser observadas, por exemplo, as posições de memória alocadas para cada tarefa - para que a consistência de dados seja garantida – além do tráfego em arquiteturas de comunicação - para o escalonamento inter-processadores, o qual deve suportar QoS para garantir a ordem e a atomicidade das mensagens.

Os serviços oferecidos pelo sistema operacional dedicado podem ser disponibilizados para a aplicação como uma API no nível de abstração de mensagens. A geração de um sistema operacional dedicado está fora do escopo atual desse trabalho.

8.1 Componentes Arquiteturais Considerados

A fim de se poderem implementar plataformas para processamento que possam expressar e otimizar diferentes MoCs arquiteturais, diferentes componentes arquiteturais para processamento – em diferentes granularidades - devem ser modelados e estarem disponíveis. Nesse sentido foram modelados em UML/C++ (de acordo com o modelo de programação proposto no capítulo 5) componentes nos níveis de abstração de mensagens e transações.

Os componentes para o nível de mensagens correspondem à PC e PO para diferentes tipos de processadores, enquanto que no nível de transações, aos componentes internos destas, como por exemplo, banco de registradores, ULAs, organização do *pipeline* e unidades dedicadas.

Na figura 8.1 é mostrado um exemplo com diferentes ULAs definidas através de uma interface; uma é dedicada a processadores RISC enquanto que a outra para processadores do tipo VLIW. A diferença entre essas unidades verifica-se na composição de três ULAs para processadores VLIW.

Uma vez que Modelos de Computação para plataformas arquiteturais são definidos no capítulo 5, como a relação entre componentes arquiteturais, é possível relacioná-los de forma a compor diferentes arquiteturas de processadores, que sejam otimizados para diferentes comportamentos encontrados em aplicações dedicadas.

Essencialmente, componentes arquiteturais para processamento são organizados como PC e PO, dedicados à execução de conjuntos de instruções, portanto programáveis. No entanto, instruções dedicadas podem ser acrescentadas à arquitetura do processador, visando otimizar operações dedicadas, possivelmente computacionalmente intensivas.

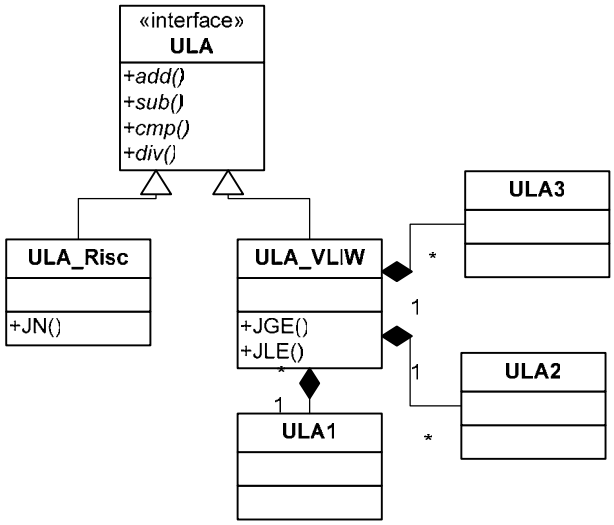


Figura 8.1: Modelos de ULAs para diferentes tipos de processadores

8.2 Frameworks para Sistemas Multiprocessados Dedicados

Assim, processadores programáveis podem ser utilizados em várias aplicações, sendo unidades específicas adicionadas sempre que necessário. A flexibilidade proporcionada por conjuntos de instruções programáveis é importante por permitir que sistemas sejam atualizados por SW.

Instruções dedicadas são realizadas por unidades específicas, implementadas em HW dedicado. Sempre que unidades específicas são adicionadas, a instrução que as ativam são passam a fazer parte do conjunto de instruções do processador.

Processadores podem ser classificados como:

- De propósito geral; programáveis – *GPP, General Purpose Processor*;
- De propósito específico, contendo unidades dedicadas apenas, ASICs; ou
- Programável, com conjunto de instruções configurado com operações dedicadas – *ASIPs, Application Specific Instruction-Set Processor*.

A fim de se conseguir flexibilidade para as análises em plataformas para processamento, foram modelados frameworks para processadores CISC, RISC, DSP, Superescalar e VLIW. Essas arquiteturas compõem um amplo espectro para processadores, cobrindo grande parte das arquiteturas existentes para processamento. Cada uma dessas arquiteturas é caracterizada pelas maneiras como os componentes da PO podem ser relacionados. O tipo do processador determina como as funcionalidades da aplicação são executadas, em relação à concorrência, tipos de operações, desempenho, etc. Por exemplo, processadores podem ser configurados com unidades para processamento de sinal, o número de unidades funcionais pode ser determinado em processadores VLIW ou Superescalares e assim por diante.

Cada uma das descrições de processadores modeladas pode ser configurada tanto em relação à sua organização interna, quanto à inclusão de operações dedicadas – em conjunto com os componentes arquiteturais correspondentes – ao seu conjunto de instruções, de acordo com as regras estabelecidas no capítulo 5, seção 5.4.

Os experimentos conduzidos nesse trabalho referem-se a análises realizadas no nível de abstração de mensagens.

A figura 8.2 mostra algumas possibilidades de reconfiguração de um processador VLIW através de interfaces de controle. Observa-se o uso das interfaces em dois níveis hierárquicos da descrição do processador: no primeiro nível - de mensagens - são configuradas, através da interface “VLIW_IF”, operações do processador, bem como algumas de suas características, como o número de ULAs. No segundo nível - componentes internos do processador, nível de transações - a ULA pode ser configurada com diferentes possibilidades de operações, através da interface “ULA_IF”.

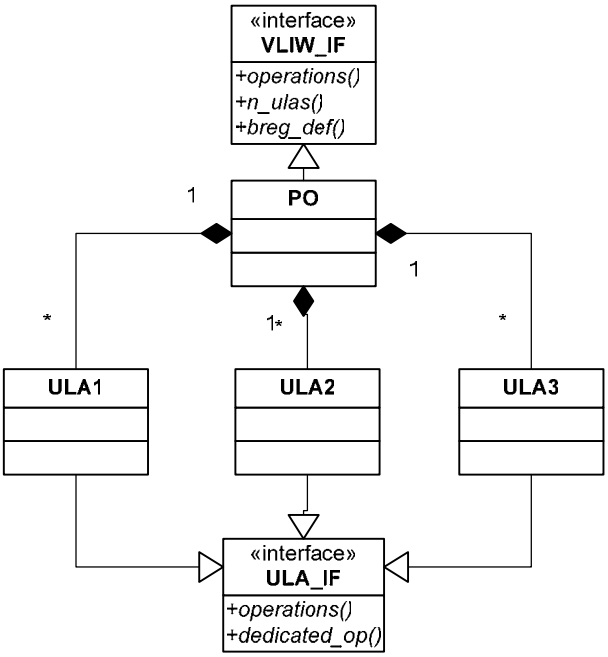


Figura 8.2: Configurações em Processadores VLIW

Sistemas completos são obtidos, segundo o modelo de programação estabelecido no capítulo 5, através da conexão entre descrições de processadores e de NoCs ou barramentos, como especificado na seção 5.6.

8.3 Ajuste das Funções Custo

As funções custo para sistemas completos são obtidas a partir das equações 3.1 a 3.6. A maneira como os valores para arquiteturas de comunicação são obtidos são discutidos no capítulo 7, seção 7.3. Portanto, resta estabelecer como são obtidos os valores para as funções custo para plataformas de processamento; equações 3.1 a 3.3.

De acordo com o sugerido pelo método aqui proposto para a avaliação arquitetural, os valores para as funções custo são obtidos por simulação, tanto para plataformas de processamento quanto de comunicação. Uma vez que os frameworks são simulados, a obtenção dos valores para as funções custo de energia e desempenho ($\kappa\varepsilon\rho$ e $\kappa\tau\rho$, das equações 3.1 e 3.2) é realizada no metanível, através de introspecção computacional. A implementação da introspecção é implementada através da interface de controle de cada componente arquitetural da plataforma. Os valores para a função custo relativa à área ($\kappa\alpha\rho$, da equação 3.3) são derivados da pré-síntese de cada componente disponível na plataforma.

Os valores relativos ao desempenho podem ser obtidos durante a simulação de cada componente. O simulador dedicado às descrições das plataformas de

processamento obtém o desempenho $\tau\rho$, coletando, através da interface de controle, o número de ciclos necessário à execução de cada instrução. Isso pode ser conseguido somando-se os valores $\tau_{cache[i]} + \tau_{bp[i]} + \tau_{un[i]} + \tau i_{fetch[i]}$, para cada instrução, durante a simulação da aplicação.

A energia deve poder ser estimada no nível arquitetural, uma vez que a otimização de sistemas complexos pode envolver a análise de vários componentes. Nesse caso, simulações realizadas em nível lógico ou elétrico demonstram-se demasiadamente demoradas. Considerando essa situação, optou-se por seguir o modelo definido por Tiwari em [TIW 94], onde a análise do consumo de energia é realizada em função das instruções do processador alvo. Nessa abordagem, o consumo de energia é medido individualmente para cada instrução. A medição é realizada através da construção de um laço onde uma mesma instrução é executada várias vezes. Com isso é possível medir-se a corrente (I) necessária a execução da instrução. A corrente é então utilizada para obter-se a potência média: $P = I \times V_{CC}$, onde I é a corrente e V_{CC} é a voltagem do processador. Multiplicando-se a potência pelo tempo de execução da instrução, obtém-se o consumo de energia da instrução: $E = P \times T$, onde E é a energia consumida pela instrução; P é a potência e T corresponde ao tempo de execução da instrução: $T = N \times \partial$, onde N: número de ciclos para executar a instrução; e ∂ é o período do relógio.

O consumo total de energia para uma aplicação é calculado somando-se o consumo individual de cada instrução. No entanto, alguns efeitos devem ser considerados quando da execução da aplicação, como por exemplo, paradas no *pipeline* e perdas na memória *cache*. Tais efeitos não dependem da execução individual de cada instrução, mas sim do fluxo de execução. Para avaliar esses efeitos a aplicação pode ser simulada e a taxa de perda na *cache*, bem como o taxa de quebra do *pipeline*, determinados. Com base nessas taxas é possível determinar o número total de ciclos necessários para atender esses efeitos. Finalmente, multiplicando-se o número de ciclos pela energia consumida por cada ciclo, obtém-se o total de energia necessária para atender às quebras de *pipeline* e perdas na *cache*.

Essa estratégia para a estimativa do consumo de energia como proposto por Tiwari pode ser adaptada à função custo, como proposto pela equação 3.1, uma vez que o consumo de energia de cada instrução pode ser determinado pela soma do consumo de energia das unidades componentes da PO e PC, utilizadas na sua execução. Assim, o consumo de energia de cada instrução *i*, κ_i é igual a: $\kappa_i = \kappa\varepsilon\rho_{ip}$, onde *p* é o processador onde a instrução é executada. É importante notar que o consumo de energia para a memória *cache* ($\varepsilon_{cache[i]}$), bem como das unidades funcionais ($\varepsilon_{un[i]}$) consideram respectivamente a taxa de perdas na *cache* e a taxa de falhas na previsão de desvios. Essas taxas são obtidas durante a simulação da aplicação.

No entanto, como a abordagem para avaliação arquitetural aqui apresentada prevê a avaliação de vários componentes arquiteturais para plataformas de processamento, a efetivação da estratégia discutida acima implica na medição do consumo de energia para diversos componentes da plataforma, possivelmente, em mais de uma tecnologia de fabricação. Com as tecnologias atualmente existentes em ferramentas para a medição do consumo de energia, isso somente pode ser conseguido com a pré-síntese de cada componente em alguma tecnologia alvo. Isso restringe análises em altos níveis de abstração, como *mensagens*, por exemplo, onde apenas as funcionalidades de cada componente são consideradas. Mesmo assim, para um conjunto de processadores idênticos, é possível avaliar o consumo de energia, considerando-se apenas o número de

ciclos que os processadores necessitam para executar instruções. Isso é conseguido, assumindo-se que os processadores possuem o mesmo consumo de potência para as instruções de mesmo tipo, como lógico/aritméticas, por exemplo.

A avaliação do consumo de energia é importante em sistemas multiprocessados, uma vez que a distribuição de tarefas entre processadores pode possibilitar a otimização simultânea do desempenho e do consumo de energia. Duas situações podem ocorrer nesse contexto: distribuição de tarefas entre processadores de propósito geral ou entre ASIPs. Em ambos casos, obtém-se aumento de desempenho pela distribuição da execução das funcionalidades da aplicação.

Quanto ao consumo de energia, este pode aumentar no primeiro caso, pois as tarefas são executadas como SW nos processadores. Além disso, quanto maior a distribuição, possivelmente maior o consumo da arquitetura de comunicação. Já para o segundo caso, o consumo da arquitetura de comunicação pode ser reduzido pela execução das tarefas como funções dedicadas em processadores ASIPs.

Uma vez determinadas todas as funções custo, tanto para plataformas de processamento, quanto de comunicação, pode se estabelecer as funções custo para um sistema completo, como a soma das funções custo individuais: sendo o vetor κ_{SoC} = { ε_{SoC}, τ_{SoC} α_{SoC} }, onde ε_{SoC}, τ_{SoC} e α_{SoC} correspondem respectivamente às restrições consumo de energia, área e desempenho para sistemas completos:

Função custo para *energia*:

$\kappa\varepsilon_{SoC} = \kappa\varepsilon\rho + \kappa\varepsilon\chi$, tal que $\kappa\varepsilon_{SoC} \leq \varepsilon_{SoC}$

Função custo para *desempenho*:

$\kappa\tau_{SoC} = \kappa\tau\rho + \kappa\tau\chi$, tal que $\kappa\tau_{SoC} \leq \tau_{SoC}$

Função custo para *área:*

$\kappa\alpha_{SoC} = \kappa\alpha\rho + \kappa\alpha\chi$, tal que $\kappa\alpha_{SoC} \leq \alpha_{SoC}$

O uso dessas funções custo permite avaliar sistemas completos.

8.4 Aplicações Testadas

Para efetivar a avaliação das plataformas para processamento e para sistemas completos - exemplificadas na seção 8.2 – o método aqui proposto sugere a simulação dos frameworks correspondentes e a conseqüente aplicação das funções custo - discutidas na seção anterior – como pode ser verificado na figura 4.3. Através de simulação, os componentes arquiteturais de cada plataforma podem ser avaliados por ferramentas que executam no metanível, através de introspecção computacional. O uso de simulação permite a avaliação de comportamentos dinâmicos, os quais capturam situações que induzem não linearidade, como por exemplo, paradas em *pipelines*, execução simultânea de instruções e contenções em redes. No entanto, uma vez que vários componentes podem ser testados, é necessário que as simulações sejam realizadas o mais rapidamente possível, já que sistemas com grande variedade de componentes e diversidade comportamental podem ser considerados. Com essa finalidade, as avaliações são realizadas através de um simulador para código compilado, como discutido na seção 6.4.

Uma possível abordagem para a avaliação de plataformas arquiteturais consiste da geração de aplicações aleatórias, visando testar a grande diversidade de

comportamentos presentes em aplicações embarcadas. Nesse sentido, aplicações aleatórias são modeladas com tarefas que exploram características do comportamento de componentes arquiteturais para processamento, tais como a política de previsão de desvios, o número de unidades funcionais e a presença de unidades dedicadas.

Isso é conseguido através da modelagem de características da aplicação, como o paralelismo a nível de instruções, para a exploração do número de unidades funcionais; o tipo das instruções de controle, para a exploração do *pipeline* e política de previsão de desvios; e instruções específicas ou computacionalmente intensivas, as quais podem ser implementadas por unidades dedicadas.

Mesmo sendo suficiente o uso de aplicações aleatórias para a exploração e conseqüente avaliação dos componentes arquiteturais, para este trabalho foram consideradas algumas aplicações dedicadas reais, buscando trazê-lo para mais perto da realidade a qual se propõe a avaliar.

A primeira aplicação modelada refere-se ao roteamento de pacotes, segundo o algoritmo Ipv4 a 10Gb/s (*IPv4 packtet forwarding @ 10Gb/s*) [SHA 2001].

Devido à grande demanda por largura de banda, um roteador – normalmente implementado em um único processador - não consegue atende o processamento de 10 *giga* pacotes por segundo. Uma possível solução arquitetural para a implementação dessa aplicação com a tecnologia existente pode ser encontrada na implementação de suas funcionalidades em um conjunto de processadores, conectados a uma NoC. A comunicação por uma rede, segundo o protocolo IPv4, ocorre entre filas de entrada e saída. Processadores que lêem os pacotes da fila de entrada, os roteiam, para em seguida enviar para a fila (*buffer*) de saída. O roteamento pode ser implementado sob duas abordagens: estática e dinâmica. No roteamento *estático*, as rotas são predeterminadas, o que torna a atualização das tabelas de roteamento desnecessária, e por conseqüência, não há comunicação entre os processadores. Isso ocorre porque nesse caso não existe a necessidade de um processador estar ciente do destino de pacotes roteados por outros processadores. No caso de roteamento dinâmico, as rotas para os pacotes são determinadas em tempo de execução, em função de critérios como, a menor rota, a rota mais livre, etc.; o que exige a atualização da tabela de roteamento a cada vez que um pacote é roteado por algum processador. Nesse caso, cada vez que um processador roteia um pacote, esse atualiza a tabela de roteamento, o que deve ser informado aos demais processadores, para que estes atualizem as informações pertinentes às rotas. As figuras 8.3 e 8.4 ilustram respectivamente as versões seqüenciais e distribuídas da aplicação de roteamento IPv4, através do modelo de programação aqui proposto.

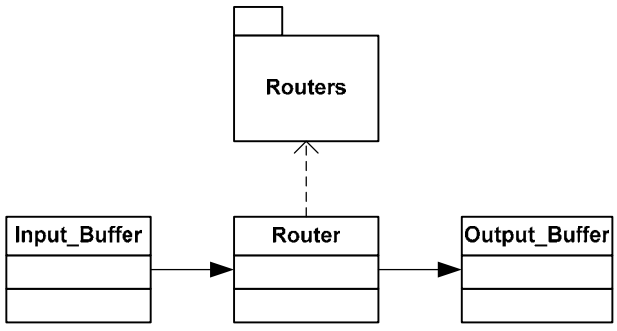
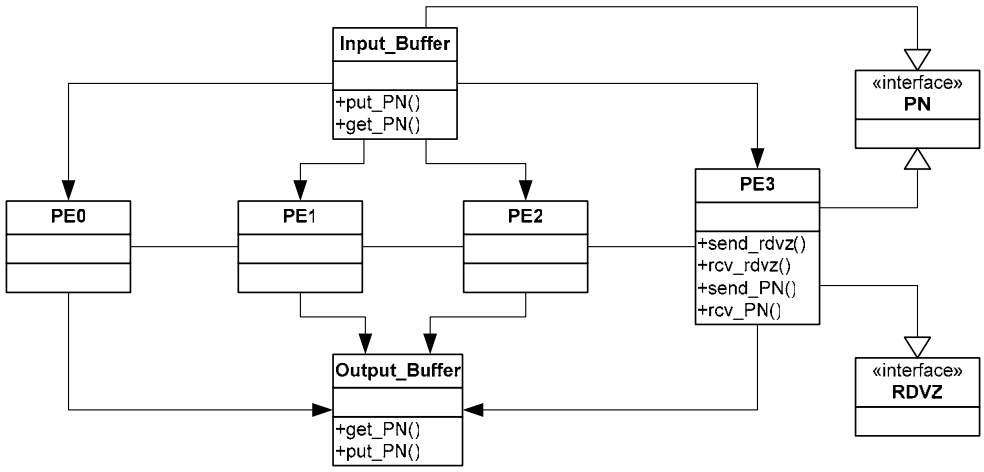


Figura 8.3: Diagramas de Classes UML para a Aplicação de Roteamento IPv4, versão seqüencial

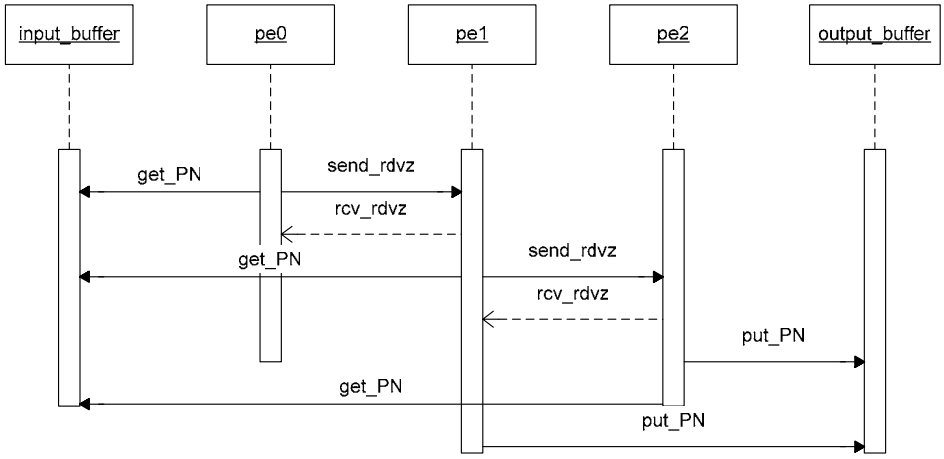
Pode-se verificar na figura 8.3 o fluxo de dados entre as entidades envolvidas na execução da aplicação de roteamento IPv4. Existem dois *buffers* para armazenar, respectivamente, os pacotes de entrada e saída. O roteador (Classe *Router*) é responsável por rotear os pacotes presentes no buffer de entrada para o buffer de saída. Para tal, é necessário que este se comunique com alguns outros roteadores da rede (representados por um pacote UML), uma vez que a rota é definida em função de certos critérios, como a rota mais curta, a rota mais livre etc. Essas informações são utilizadas para atualizar as tabelas de roteamento. Como cada roteador *depende* dos demais para adquirir as informações sobre as rotas, existe uma relação de dependência entre os roteadores.

Já na versão distribuída, a tarefa de roteamento é quebrada em outras *n* tarefas, a fim de que seja possível rotear os pacotes na velocidade apropriada. Essas tarefas podem ser mapeadas para até *n* processadores com esse objetivo. Na Figura 8.4 os processadores são representados pelas classes PE_x, onde *x* corresponde ao número do processador.

Na figura 8.4(a) todos os processadores recebem pacotes a serem roteados e os escrevem no buffer de saída através do MoC *Process Network* (PN), o qual estabelece comunicação por buffers. Na versão distribuída os roteadores necessitam se comunicar entre si, a fim de se manter a consistência da tabela de roteamento. Essa comunicação é implementada pelo MoC *Rendezvous*, uma vez que a tabela de roteamento é única para todos os processadores. Algumas possíveis comunicações entre os elementos de processamento e destes com os buffers de entrada e saída são ilustradas no diagrama de seqüências da figura 8.4(b). Nos experimentos realizados até o momento a tarefa de roteamento é quebrada em *16* tarefas, quais podem ser particionadas em até 16 processadores, conectados, por exemplo, em uma NoC 4x4.



(a)



(b)

Figura 8.4: Diagramas de Classes e de Seqüências UML para a Aplicação de Roteamento IPv4, versão distribuída

A segunda aplicação testada refere-se ao uso da abordagem de algoritmos genéticos [CHA 97] para análise de ativos farmacológicos complexos. Alguns medicamentos, como os anti-hipertensivos [LEA 2001] necessitam que suas substâncias sejam dosadas utilizando a técnica de espectrofotometria [GOR 95] ultravioleta, através de cromatografia líquida de alta eficiência [FAR 88]. Mesmo com o uso dessas técnicas, a obtenção de dosagens corretas é prejudicada pela precariedade em se conseguir uma separação eficiente de seus componentes. A separação correta dos componentes dos medicamentos depende da qualidade das amostras de medicamentos utilizadas na espectrofotometria. A qualidade da amostra é sujeita a aplicação de comprimentos de onda, os quais são utilizados para avaliar os componentes da amostra para atestar a sua qualidade na separação dos componentes. Como existem muitos comprimentos de onda que podem ser utilizados, a solução para esse problema normalmente pode ser alcançada através de solução algorítmica, baseada em heurísticas.

A heurística é utilizada para separar de forma automatizada as melhores amostras, as quais são utilizadas na espectrofotometria para a separação e análise dos componentes dos medicamentos. O uso da abordagem heurística justifica-se pela grande quantidade de comprimentos de onda que podem ser utilizados para avaliar cada amostra. Como resultado da aplicação da heurística, apenas alguns poucos comprimentos de onda são selecionados para análise das amostras, o que vem a facilitar o trabalho do engenheiro químico. O objetivo para a heurística é então, encontrar um número reduzido de comprimentos de onda, com qualidade suficiente para garantir a separação das melhores amostras para a análise dos medicamentos.

Trabalhos que vem sendo realizados nessa área, apontam a heurística de algoritmos genéticos, como a responsável por encontrar as melhores soluções [KON 2004].

A aplicação pode ser caracterizada para a abordagem de algoritmos genéticos atribuindo-se a cada indivíduo a semântica dos comprimentos de onda a serem utilizados ou não, para a análise das amostras. Cada *gen* dos indivíduos representa o uso ou não do comprimento de onda correspondente a sua posição. Assim, uma determinada posição possui valor “0”, significa que o comprimento de onda representado por esta

posição não será utilizado para a análise das amostras; se for “1”, será utilizado. O tamanho de cada indivíduo corresponde ao número de comprimentos de onda disponíveis para análise. O tamanho da população é proporcional a quantidade de combinações que se queira analisar durante cada passo do algoritmo.

A implementação da heurística de algoritmos genéticos para essa aplicação é normalmente realizada de forma seqüencial: as populações são criadas através da geração de cada indivíduo, através de *crossover* ou *mutação*, sendo a população então, avaliada. A avaliação é realizada com base em uma matriz $A = m_x n$, onde m representa o número de amostras e *n*, o número de comprimentos de onda. O valor de cada posição *A[m,n]* representa o valor de absorção da amostra *m* pelo comprimento de onda *n*.

No entanto, em [KON 2004] é proposta a implementação distribuída dessa aplicação. Isso é possível, uma vez que a natureza da aplicação permite a distribuição de suas tarefas, de forma a se obter um aumento de desempenho. A implementação distribuída consiste na distribuição do cálculo de *fitness* (grau de adaptatividade) dos indivíduos da população para um conjunto de elementos de processamento. A distribuição máxima pode ser conseguida quando cada indivíduo é alocado para um processador dedicado. Como as operações de *crossover* e *mutação* são realizadas *entre* indivíduos, estas são realizadas em um processador central. No entanto, o cálculo do *fitness* é individualizado, sendo, portanto, calculado de forma distribuída. Além disso, como as operações de *crossover* e mutação são realizadas sobre vetores de bits (que representam os gens) estas não são computacionalmente intensivas. Já o cálculo do *fitness* é realizado sobre a matriz *A*, através de $T[m_x n] = \sum_{k=0}^{n} M[m_{(k)x} j] - \sum_{k:1}^{n} M[m_{(k-1)x} j]$, onde j: 1..*n* e T, matriz auxiliar, o que tende a ser computacionalmente intensivo, uma vez que a matriz *A* pode - tipicamente - compreender mais de mil amostras.

A implementação distribuída pode ser realizada em uma arquitetura contendo um elemento de processamento para cada indivíduo, mais um elemento de processamento para o processador central, conectados através de uma NoC. Em relação às operações de processamento, a execução da aplicação consiste na execução das operações de *crossover* e mutação no processador central e no cálculo de *fitness* nos demais processadores, cujo número corresponde ao tamanho da população. Quanto às operações de comunicação, estas são realizadas em dois sentidos: do processador central para os demais processadores, para o envio de cada indivíduo da população, e dos demais processadores para o processador central, para o envio da resposta do cálculo do *fitness*. Assim, o número de mensagens da aplicação corresponde ao dobro do número de indivíduos da população. Nos experimentos realizados até o momento, a população considerada possui *50* indivíduos.

A figura 8.5 mostra o pseudocódigo para a aplicação para Análise de Ativos Farmacológicos.

```
A = mxn; //m == #amostras  n == #comprimentos de onda
PE procs[p], cp; // “p” processor elements (procs) and one “central processor”; cp
// ind[n]== 0 (nao utilize o comprimento de onda); n == 1 --> utiliza
individual ind[i];
for all i pop= ind[i]; // population has “k” individuals
while (it < nit) { // perform for “nit” iterations
```

```
   // server side
   mutation(pop);
   crossover(pop);
   for all p,i
     procs[p]= ind[i]; // send individuals to processors
  // client side
   for all p
       pc= procs[p].fitness(); // calculate fitness sending back to cp
   cost.pop = pc.fitness_evaluation();
   if(cost.pop < cost.best_pop) best_pop= pop;
pop = pc.generate_new_population();
}
return best_pop;
```

Figura 8.5: Pseudocódigo para a Aplicação para Análise de Ativos Farmacológicos, abordagem distribuída.

Na figura 8.6 são mostrados os Diagramas de Classes e de Seqüências em UML, para a aplicação para Análise de Ativos Farmacológicos.

No diagrama de classes observa-se que o processador central realiza uma relação de dependência com os demais processadores, uma vez que este necessita do resultado do cálculo do *fitness* para avaliar a população. A comunicação entre os processadores com o central é realizada através do MoC *Rendezvous* (*rdvz*), para que as etapas do algoritmo genético possam ser sincronizadas. A comunicação é implementada através das operações *send* e *receive* (*send_rdvz()*, *rcv_rdvz()*), as quais implementam o comportamento estabelecido pelo MoC *Rendezvous*, pelo compartilhamento da interface *rdvz*. O sincronismo também pode ser observado no diagrama de seqüências, através do envio das mensagens simultaneamente.

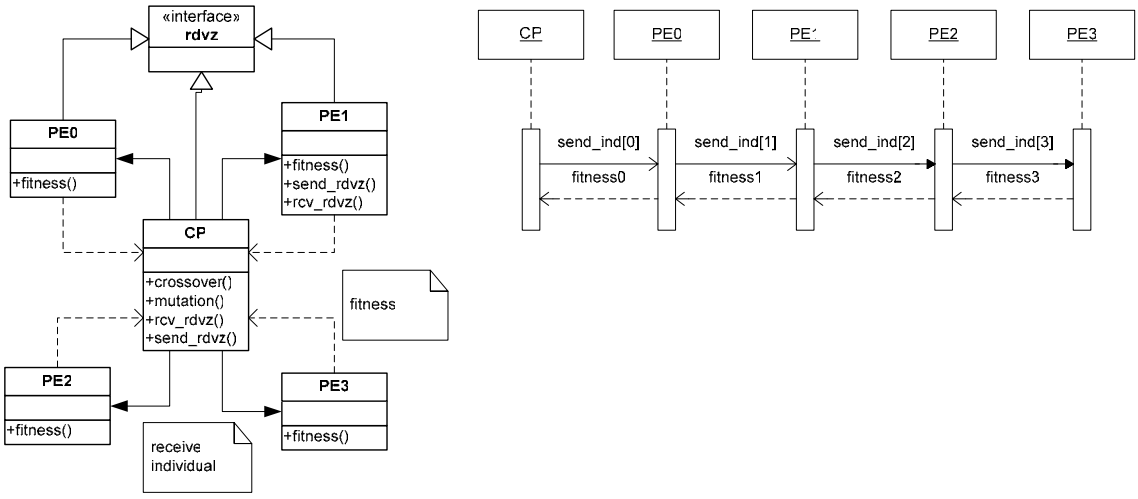


Figura 8.6: Diagramas UML de Classes e de Seqüências para a Aplicação para Análise de Ativos Farmacológicos

A terceira aplicação considerada implementa um jogo de futebol de maneira distribuída, como a definida em [ROB 2005]. O objetivo para a síntese dessa aplicação refere-se à implementação de cada uma de suas tarefas de maneira distribuída de modo a poder ser implementada em sistemas embarcados. Devido à complexidade computacional das funções da aplicação, estas demandam grande poder de processamento, normalmente executadas com sucesso em processadores "estado-da-arte". No entanto, espera-se com a implementação distribuída realizar a aplicação em sistemas com restrições de desempenho e consumo de energia.

A modelagem da aplicação foi realizada de maneira a distribuir as tarefas do simulador "*RoboCup soccer server simulator*" entre os jogadores das equipes. Dessa forma, cada tarefa corresponde a um jogador de cada equipe, aos técnicos e aos juízes, principal e auxiliares. Cada jogador executa funções tais como decidir para onde passar a bola, dependendo da estratégia de sua equipe e das posições dos companheiros ou como marcar um adversário. As figuras 8.7(a) e 8.7(b) exemplificam respectivamente o diagrama de classes e de seqüências UML, de acordo com o modelado para a aplicação.

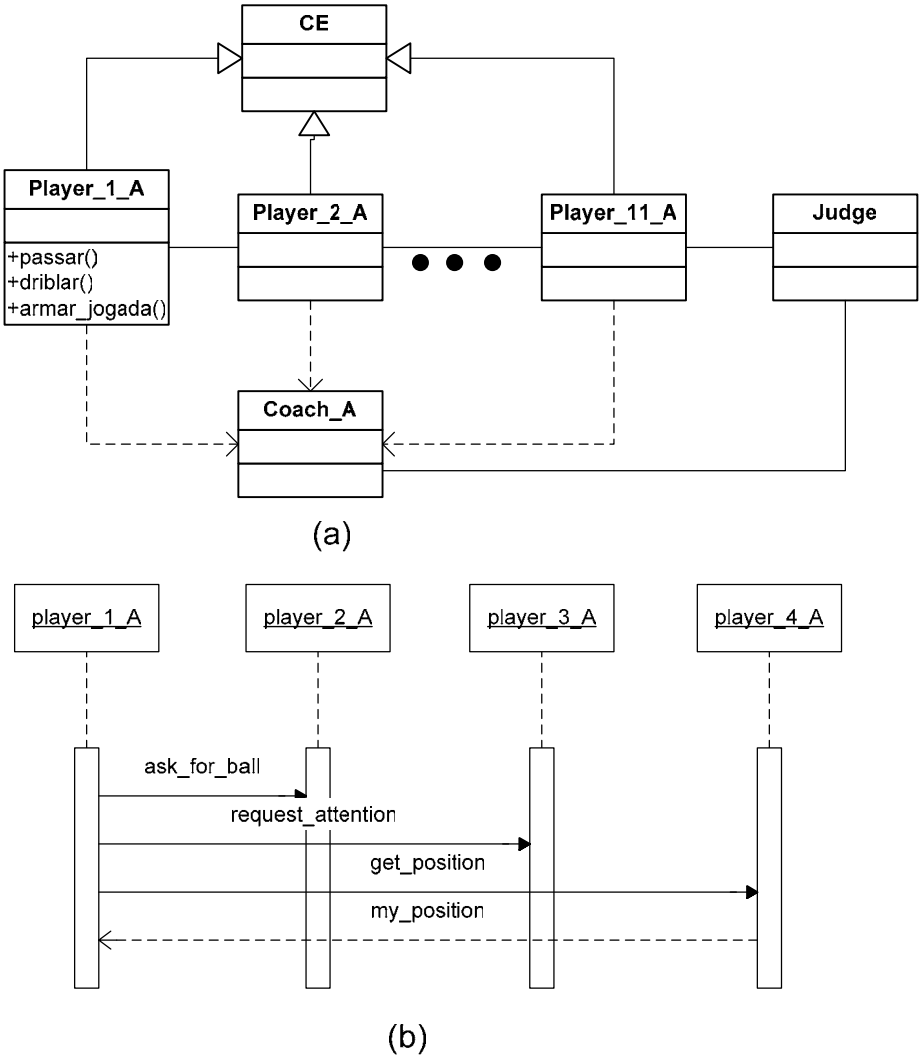


Figura 8.7: Diagramas UML de Classes e de Seqüências para a Aplicação do Jogo de Futebol

Como se pode observar na figura 8.7, cada jogador relaciona-se com os demais companheiros e com o técnico de sua equipe. Os juizes relacionam-se com os jogadores

e com os técnicos. Informações como a posição da bola são encapsuladas nessas mensagens. A “entidade central” (CE) implementa as operações que cada jogador pode executar e controla a simulação. Existe uma relação de dependência entre os jogadores e o seu técnico. Essa modelagem garante que todos os jogadores recebem as instruções sobre as táticas a serem seguidas. É interessante notar a heterogeneidade da aplicação nas comunicações entre os jogadores: algumas mensagens exigem sincronismo entre os jogadores, como por exemplo, a solicitação da posição de um jogador, que é implementado pelo MoC *rendezvous*. Por outro lado, quando um jogador “pede” a bola, este não precisa saber se o jogador destino efetivamente recebeu a mensagem, uma vez que será notificado pelo mesmo em caso afirmativo.

8.5 Otimizações Consideradas para Plataformas Arquiteturais para Sistemas Multiprocessados Completos

Como previamente discutido no capítulo 3, a avaliação de plataformas arquiteturais caracteriza um problema complexo devido ao amplo espaço de busca, conseqüência da grande variedade de componentes arquiteturais que podem configurados para executar o comportamento da aplicação alvo, respeitando as restrições de projeto. Ainda, como resultado do particionamento, o número de elementos de processamento empregados na execução da aplicação pode variar de “1” até o número de tarefas da aplicação alvo.

Considerando-se aplicações dedicadas, processadores do tipo ASIP podem ser utilizados para implementar operações para transformação de dados específicas, as quais são realizadas com a inclusão de unidades funcionais específicas na parte operativa e do rearranjo das relações entre os componentes internos desta.

Quanto à complexidade, processadores podem ser classificados em função do número e da semântica das unidades funcionais e da semântica do conjunto de instruções e da parte de controle. Assim, um processador superescalar com previsão de desvios pode ser considerado mais complexo do que um processador com apenas uma unidade funcional na parte operativa.

De maneira geral, a otimização de plataformas de processamento para sistemas multiprocessados pode ser descrita como a tarefa de “determinar o número de processadores, bem como o conjunto de instruções e a complexidade de cada processador, de modo a que as restrições da aplicação sejam atendidas”. Isso corresponde a determinar o número de processadores *np* e a semântica de cada um ($p_i \in P \in PROC = \{type_{[i]}, cache_{[i]}, bp_{[i]}, un_{[i]}, pipe_{[i]}, ifetch_{[i]}, i: 1 ... np\}$, da definição 3.6), em um MPSoC, de acordo com a definição 3.7.

Devido a sua complexidade, essa tarefa pode ser particionada em tarefas menores, quais sejam:

- Particionamento de tarefas entre processadores;
- Escolha das instruções pertinentes de cada processador;
- Análise e implementação de alternativas para instruções específicas em processadores; e
- Avaliação da complexidade e da organização das partes operativa e de controle de cada processador.

O processo de particionamento torna-se desnecessário apenas quando somente um processador for utilizado. Nesse caso, apenas os componentes internos da PC e PO dos

processadores são analisados. No entanto, essa não é a situação esperada para as aplicações alvo a serem tratadas dentro do contexto desse trabalho, uma vez que estas são constituídas por várias tarefas, as quais podem ser particionadas para um conjunto de processadores.

O particionamento pode ser realizado sobre um conjunto de processadores homogêneos ou com características diferenciadas.

A escolha de instruções pertinentes de cada processador envolve a análise dinâmica (*profiling*) da aplicação alvo, a fim de se descobrir quais as instruções mais importantes para a aplicação. As instruções que não são utilizadas podem automaticamente ser descartadas do conjunto de instruções, enquanto que as pouco utilizadas podem ser implementadas por SW. Essa tarefa visa diminuir a complexidade da parte de controle – pela diminuição do número de instruções a decodificar – e da parte operativa, caso instruções que usem componentes exclusivos são retiradas do conjunto de instruções. Em [KRE 97 e ITO 2000] tais otimizações são consideradas, como já discutidas no capítulo 2.

A análise e implementação de alternativas para instruções específicas consistem na realização em HW de operações específicas e que demandam grande poder computacional ou consumo de energia, como por exemplo, instruções para processamento digital de sinais. Ainda, funções computacionalmente intensivas podem ser implementadas como HW, visando aumento de desempenho e diminuição do consumo de energia. Essa abordagem é utilizada por Alba, em [ALB 96].

Finalmente, a avaliação da complexidade e da organização das partes operativa e de controle de processadores consiste na determinação das características arquiteturais das partes operativa e de controle que melhor correspondam às restrições de projeto.

Os experimentos conduzidos sobre processadores referem-se à avaliação do número de unidades funcionais e a realização de tarefas por unidades dedicadas, objetivando-se encontrar um compromisso entre essas características arquiteturais que contemple as restrições de projeto. A escolha pela avaliação dessas características se deve ao fato de que tanto o número de unidades funcionais, quanto à presença ou não de unidades dedicadas exercem influência direta na complexidade das partes operativa (número de componentes) e de controle (decodificação e despacho de instruções). As demais características arquiteturais de processadores são mantidas fixas: a precisão para a memória *cache* e para a política para previsão de desvios é configurada para 97%. A organização do *pipeline* considera cinco estágios: busca, decodificação, leitura dos operandos, execução e escrita do resultado. Finalmente, o tamanho da janela de instruções é configurado para ser igual ao dobro do número de unidades funcionais.

A realização de tarefas por unidades dedicadas é implementada considerando-se “um” ciclo de relógio para a execução de cada instrução lógico/aritmética, o que corresponde a um estado em uma máquina de estados dedicada. Ainda, assume-se que a execução ocorre em apenas uma unidade funcional, disponível para realização da unidade funcional dedicada. Observa-se, no entanto, que a disponibilidade de mais unidades funcionais pode implicar em aumento de desempenho devido ao paralelismo na execução das operações. Nesse caso, as operações podem ser escalonadas de forma otimizada, por exemplo, por algoritmos do tipo ASAP (*As Soon as Possible*) ou ALAP (*As Late as Possible*) [GAJ 97].

Experimentos sobre arquiteturas de comunicação referem-se à avaliação de barramentos e NoCs com topologias Mesh homogêneas e heterogêneas (como abordado no capítulo 7) na implementação de MP-SoCs.

Devido à ortogonalidade entre os comportamentos de processamento e comunicação, estes podem ser avaliados separadamente ou em conjunto.

A complexidade do processo de otimização aumenta quando um sistema completo é avaliado, uma vez que as plataformas para processamento e comunicação são consideradas simultaneamente na avaliação. Nesse caso, a otimização é realizada simultaneamente sobre o número e a semântica dos processadores (como descrito acima), em conjunto com a "Arquiteturas de NoCs - NOC", como especificado na definição 3.4. Nesse caso, as funções custo consideradas referem-se a sistemas completos - $\kappa\varepsilon_{SoC}$, $\kappa\tau_{SoC}$, $\kappa\alpha_{SoC}$ – como discutido na seção 8.3, acima.

As seções a seguir descrevem as análises e otimizações realizadas sobre plataformas arquiteturais para sistemas completos.

8.5.1 Particionamento de Tarefas

O processo de particionamento de tarefas entre elementos de processamento pode ser implementado através da operação de *particionamento*, como estabelecido pela definição 3.8. Uma vez que essa operação possui natureza complexa (ver capítulo 3), deve-se adotar uma heurística para a sua solução, pois soluções baseadas em busca exaustiva demandam tempo demasiadamente grande.

Assim como para a operação de mapeamento, para a implementação da operação de particionamento, o algoritmo de busca tabu também pode ser utilizado. Para tal, basta que se atribua à semântica do conjunto de "recursos" *S*, o conjunto das *posições* das tarefas da aplicação alvo: $S_{tasks\_places[i]} = pT_{[i]}$, $i = 1,..., nt$; $T_{[i]} = \{bb_{i0}, bb_{i1}, ..., bb_{in}\}$, onde *nt*: número de total de tarefas da aplicação e bb_x, blocos básicos - da definição 3.2: s_{tasks_places} $(y_1) = [10, 10, 5,..., nt]$, onde $y_1[0] = 10$ e $y1[1] = 10$ significa que as tarefas "0" e "1" estão alocadas ao processador "10" ($pT[0] = pT[1] = 10$), enquanto que a tarefa "2", está alocada ao processador "5" ($y1[2] = 5$; $pT[2] = 5$), para a solução "y_1".

Para que seja possível se estabelecer o número de processadores que melhor corresponda às restrições da aplicação, as tarefas são primeiramente alocadas para um único processador ($S_{tasks\_places[i]} = \{P_{x0}, P_{x1}, ..., P_{xnt}\}$, onde *x* é a identificação do processador e i: 1 .. *nt*) e, a medida que a busca por soluções otimizadas avança, as tarefas vão sendo progressivamente alocadas para diferentes processadores ($S_{tasks\_places[i]} = \{P_{x0}, P_{y1}, ..., P_{znt}\}$, onde *x, y* e *z* são identificações de processadores disponíveis na plataforma arquitetural de processamento).

8.5.2 Conjunto de Processadores Heterogêneo

A implementação da operação de particionamento como apresentada na seção anterior considera somente processadores homogêneos. No entanto, soluções mais otimizadas podem ser encontradas se um conjunto de processadores heterogêneos for avaliado. Nesse caso, cada processador utilizado na composição do sistema pode possuir componentes com comportamentos específicos, como por exemplo, diferentes políticas para previsão de desvios ou partes operativas com número diferenciado de unidades funcionais. Além disso, processadores podem implementar instruções em unidades funcionais dedicadas.

O processo de otimização arquitetural para um conjunto de processadores heterogêneos possui complexidade $O((x^z)^y)$, onde *x*: número de possíveis comportamentos que cada componente pode assumir; *z*: número de componentes internos de processadores e *y*: número de processadores. Considerando-se que cada processador poderia ser avaliado em relação à sua política para previsão de desvios, número de unidades funcionais e tamanho da janela de instruções e que cada um desses componentes pudesse assumir dois comportamentos distintos, existiriam, para 8 processadores $(2^3)^8 = 16.777.216$ alternativas a serem consideradas no espaço de busca. A avaliação de cada uma destas alternativas implica na simulação da aplicação alvo sobre os processadores.

Para que o algoritmo de busca tabu possa implementar a operação de particionamento com conjunto de processadores heterogêneos, é necessário que seja adicionado ao conjunto de recursos S_T, o conjunto $S_{proc\_het} = \{proc_{het[0]}, proc_{het[1]}, ..., proc_{het[n]}\}$, onde $proc_{het[x]} \in$ PROC (da definição 3.6) constitui um comportamento específico para o processador *x*; e *n*: número de processadores do sistema. Cada $proc_{het[x]}$ pode então, possuir um comportamento específico para cada um de seus componentes internos. Como para este trabalho serão avaliadas arquiteturas de processadores em função do número de unidades funcionais e da realização de operações em unidades dedicadas, o conjunto S_{proc_het} pode ser configurado como: $proc_{het[0]} = \{proc_{.un[0] = SW}, proc_{.un[1] = HW}, ..., proc_{.un[n\_units] = HW}\}$, indicando para cada processador o número de unidades funcionais e a sua respectiva implementação.

Uma vez que o algoritmo passa a trabalhar com dois conjuntos de recursos (S_T e S_{proc_het}), o comportamento da função OPTIMUM(), a qual gera as vizinhanças dentro do espaço de busca, necessita ser configurado para determinar como que as posições dos conjuntos serão trocadas. A figura 8.8 ilustra o pseudo-código para a função OPTIMUM, para a implementação das operações de particionamento para processadores heterogêneos e mapeamento.

```
APP app;
PROC mpsoc[np], mpsoc_opt; // "np" processors
CA, ca_opt = {NOC, BUS};
NOC noc[nr]; // "nr" routers
BUS bus;

for all np   // for "np" processors
    init (mpsoc[np].type,  mpsoc[np].cache,
       mpsoc[np].bp,  mpsoc[np].pipe,
       mpsoc[np].ifetch, mpsoc[np].un[n_units = 1]);
for all nr
      init(noc.ra[nr].rout, noc.ra[nr].sch, noc.ra[nr].buf, noc.ra[nr].cf);
      init(bus);

for all nt // for "nt" tasks
    // initially all tasks are allocated to
    // processor "0"
    part(app.task[nt], mpsoc[0]);

OPTIMUM(n_nb,
        strategy: {number_of_units,
                   dedicated_units},
        mapping = V,
        S_tasks_places, S_proc_het) {

    while n_nb {  // for "n_nb" neighborhoods

      // split and swap tasks randomly
       update(S_tasks_places, random(nt);
       for all nt; np
```

```
            part(app.task[nt],mpsoc[np]);

        for all {np,nr} {
            simulate(mpsoc[np], {noc[nr],bus});
            cost_nb[nt] = { κε_SoC, κτ_SoC, κα_SoC };
            if(cost_nb < cost) {
               cost = cost_nb;
         mpsoc_opt = mpsoc;
         ca_opt = CA;
            }
        }

         // optimization strategies
         if(strategy == number_of_units) {
            // add unit for the task with worst
            // performance
            pr = app.task[higher_cost()).get_place();
            mpsoc [proc_het[pr]. proc_.un[n_unit++]] .add_unit(); // add unit to processor
         }
         else
         if(strategy == dedicated_units) {
           pr=app.task[higher_cost()].get_place();
           unit = app.task[higher_cost[].get_unit();
           mpsoc [proc_het[pr]] .unit [proc_.un[unit]].add_dedicated_unit();
           S_proc_het = {proc_het[pr]. proc_.un[unit] = HW};
        }
         if(mapping) {
           if(CA == NOC)
              for all np; nr
                 map(mpsoc[np],noc[nr]);
         }

    return(cost, mpsoc_opt, ca_opt);
}
```

Figura 8.8: Pseudo-código para a função OPTIMUM() do algoritmo Busca Tabu para as operações de Particionamento e Mapeamento

Inicialmente, a aplicação alvo, os processadores e a arquitetura de comunicação (a qual pode ser um barramento ou uma NoC) são instanciados e inicializados, através da função *init()*. Em seguida, todas as tarefas são alocadas (particionadas) para o processador "0" através da função *part()*. A função OPTIMUM() recebe como argumentos o número de vizinhanças a serem explorada a cada iteração do algoritmo busca tabu; as estratégias para a operação de particionamento (número de unidades funcionais ou unidades dedicadas); se a operação de mapeamento será implementada ou não e os conjuntos S_{tasks_places}, e S_{proc_het} a serem explorados.

Durante a execução da função OPTIMUM(), a cada nova vizinhança (n_nb) as tarefas são aleatoriamente alocadas para novos processadores (função *update()*). Em seguida, a função *part()* particiona cada tarefa para o seu processador destino como definido no conjunto S_{tasks_places}.

Através da simulação das tarefas da aplicação na plataforma alvo (função *simulate()*), os custos para cada tarefa são obtidos. Com base nos custos obtidos e de acordo com a estratégia adotada, à tarefa com o pior custo, é gerada uma nova unidade funcional ou uma unidade dedicada no processador que a executa. Para o caso de avaliação de sistemas completos, está-se considerando apenas o custo de desempenho, sendo o consumo de energia avaliado considerando-se apenas o número de ciclos que os processadores necessitam para executar instruções, como comentado na seção 8.3.

Caso a operação de mapeamento deva ser executada, a função *map()* posiciona cada processador em terminais locais de NoCs.

Finalmente, após executar o número de vizinhanças determinado pela variável "n_nb" a função OPTIMUM() retorna o custo mais otimizado, bem como a configuração arquitetural que o gerou. Com base nessa configuração o algoritmo busca tabu determina a arquitetura melhor otimizada após um número pré-determinado de iterações a que otimiza todas as restrições de projeto.

8.5.3 Sistema Completo

A avaliação da execução simultânea das operações de processamento e comunicação permite a avaliação de sistemas completos. No entanto, como discutido no capítulo 3, seção 3.2, esse procedimento traduz-se em grande complexidade. Isso pode ser verificado no fato de que a busca de soluções otimizadas utilizando-se tanto a operação de particionamento quanto a de mapeamento isoladamente, possui complexidade pertencente à classe dos problemas NP-Completos.

Mesmo assim, acredita-se que a avaliação de um sistema completo deva ser considerada, uma vez que permite a busca da melhor solução para o compromisso processamento $_{x}$ comunicação em um SoC.

Através da avaliação de um sistema completo é possível determinar o número e a complexidade de processadores em um sistema – através da operação de particionamento – não somente em função da arquitetura de cada processador, mas também em função da arquitetura de comunicação utilizada.

É importante destacar que, uma vez que o particionamento de tarefas é realizado em função de processadores heterogêneos e de arquiteturas de comunicação, compromissos entre processamento/comunicação e implementação HW/SW de tarefas são realizados concorrentemente. Além de essa tarefa ser importante para a avaliação do comportamento completo de sistemas, constitui-se de um grande desafio para ferramentas de apoio ao projeto, como constatado por Kogel, em [KOG 2004].

Portanto, a análise e a otimização de sistemas completos, como proposto pelo método aqui abordado, vêm de encontro às contribuições esperadas para esse trabalho. A avaliação de um sistema completo é obtida através da execução da função OPTIMUM() do algoritmo busca tabu como especificado pela figura 8.8, executando-se a função de mapeamento *map()* em conjunto com o particionamento.

Mesmo que a avaliação de um sistema completo – através da execução simultânea das operações de particionamento e mapeamento possua grande complexidade, é possível amenizar o tempo de busca por soluções otimizadas, restringindo-se a quantidade de deferentes comportamentos para os componentes arquiteturais. Por exemplo, NoCs podem ser avaliadas em função de uma política de roteamento apenas, assim como processadores podem possuir o mesmo conjunto de instruções ou a mesma arquitetura de *pipeline*. Esta estratégia é aqui adotada, com esse objetivo. No entanto, ressalta-se que isso não implica em resultados imprecisos, pois em última instância, objetiva-se avaliar a eficiência do método aqui proposto na busca por arquiteturas otimizadas, as quais podem ser encontradas mesmo que apenas alguns comportamentos possíveis para componentes arquiteturais sejam avaliados. O modo como as ferramentas de apoio ao projeto são implementadas, de acordo com o fluxo de projeto proposto pelo método, as torna independente da quantidade de componentes avaliados.

8.5.4 Experimentos com Sistemas Completos

O objetivo dos experimentos realizados sobre sistemas completos, envolvendo arquiteturas de processamento e comunicação objetiva avaliar o impacto de cada uma dessas classes de arquiteturas para as restrições de projeto da aplicação. Para tanto, deve ser avaliado o compromisso da execução seqüencial/paralela das tarefas da aplicação alvo. Nesse sentido, aumentos de desempenho podem ser esperados pela distribuição de tarefas entre componentes de processamento. No entanto, uma vez que tarefas são executadas concorrentemente, o comportamento de comunicação da aplicação exerce papel fundamental sobre o desempenho e o consumo de energia. Dessa forma, o impacto causado pela arquitetura de comunicação, utilizada para a implementação do comportamento de comunicação, deve ser cuidadosamente considerado na avaliação do sistema.

As arquiteturas de processamento avaliadas não focam em processadores comerciais específicos, restringindo-se às características de processadores, como determinado na definição 3.6. Com isso, se ganha em generalidade, o que é importante quando diversas arquiteturas podem ser consideradas pelo processo de análise. Mais precisamente, nos experimentos descritos a seguir serão avaliados compromissos entre processadores de propósito geral e ASIPs. Espera-se então, determinar qual o conjunto de processadores que melhor atende às restrições de projeto, quanto ao desempenho e consumo de energia. Ressalta-se, no entanto, que o consumo de energia pode somente ser avaliado para processadores RISC homogêneos, ou seja, com o mesmo conjunto de instruções. Esta restrição deve ser adotada uma vez que as análises são realizadas em alto nível de abstração, como discutido na seção 8.3.

Como comentado na seção anterior, devido à grande complexidade da execução simultânea das operações de particionamento e mapeamento (para avaliação de sistemas completos), nem todos os componentes para plataformas de processamento e comunicação foram considerados. Com isso, procurou-se reduzir o espaço de busca, avaliando-se somente características arquiteturais, as quais, se acredita, podem exercer grande impacto nas restrições de projeto, como por exemplo, a realização em HW de funções e o número de unidades funcionais em cada processador.

Para os experimentos realizados foram considerados processadores do tipo RISC e VLIW, com 32 registradores e *pipeline* de 5 estágios, com capacidade para executar uma instrução lógico/aritmética por ciclo de relógio e 3 ciclos para o acesso à memória. A precisão da memória *cache* e da política para previsão de desvios foram configuradas para 97%. Não se objetivou para esses experimentos avaliar o impacto de diferentes precisões para os acessos à *cache* nem tampouco políticas para previsão de desvios, embora para as descrições de processadores analisados estejam previstas essas configurações. No momento, acredita-se que para sistemas embarcados, primeiramente deve-se buscar arquiteturas otimizadas em relação à execução das instruções como SW (em GPPs) ou HW (por unidades dedicadas em ASIPs), e ao número de processadores e de unidades funcionais destes. Dessa forma, é possível se avaliar o desempenho e o consumo de energia em relação à concorrência na execução das tarefas e à complexidade dos processadores, além do tamanho e organização de arquiteturas de comunicação (função direta do número de processadores).

Processadores contendo várias unidades funcionais podem se revelar bastante complexos, principalmente na decodificação, despacho de instruções e previsão de desvios na parte de controle. Isso pode levar a um aumento considerável do consumo de energia, sendo, portanto, indesejáveis para sistemas embarcados. No entanto, é

interessante que se avalie o compromisso entre a execução seqüencial de várias tarefas em um único processador complexo ou a execução concorrente em processadores simples. A adoção de processadores VLIW tem sido considerada para a implementação de tarefas dedicadas, como por exemplo, a arquitetura *ARM Data Engine Technology* [ARM 2005], a qual permite que unidades dedicadas sejam incorporadas a processadores ARM.

Durante o processo de análise, quando tarefas são alocadas para processadores do tipo ASIP, estas executam os seus blocos básicos através de unidades específicas: uma nova instrução é criada para o conjunto de instruções do processador. As tarefas sujeitas à realização em HW são aquelas que demandam maior poder computacional.

As arquiteturas de comunicação consideradas correspondem à arquitetura de barramento, a rede regular grelha (mesh) com roteamento estático "XY" e política de arbitragem do tipo *round-robin* e a redes heterogêneas compostas por roteadores *Rasoc*, *Tonga* e *Mago*, como apresentado no capítulo 7. Avaliações de diferentes topologias (inclusive irregulares) ou outras políticas de escalonamento, por exemplo, implicam simplesmente em instanciar os componentes que implementam tais comportamentos, uma vez que qualquer componente arquitetural especificado segundo o modelo de programação sugerido pelo método, é reconhecido pelas ferramentas de análise.

A avaliação da topologia grelha e de estruturas heterogêneas cobre um amplo espectro para arquiteturas de comunicação do tipo NoC, pois avaliam configurações que apresentam diferenças em relação às restrições de projeto. Além disso, arquiteturas regulares tendem a ser mais utilizadas na implementação de SoCs, pois a sua topologia 2-D é a mais adequada para as tecnologias de fabricação de CI atuais (planares). Devido à regularidade, se o *layout* do chip for organizado em *tiles* homogêneos, o fato dos canais de comunicação da topologia grelha possuírem o mesmo comprimento, facilita o projeto das linhas de metal.

Os experimentos conduzidos para sistemas completos avaliam a otimização arquitetural realizada individualmente sobre arquiteturas de processamento e comunicação, e sobre ambas. Objetiva-se com isso avaliar o grau de otimização encontrado pela heurística adotada no algoritmo de otimização, para cada classe arquitetural, determinando assim, o impacto que cada plataforma exerce sobre as restrições de projeto da aplicação alvo.

As figuras 8.9 a 8.11 referem-se a resultados obtidos para arquiteturas de comunicação – homogêneas e heterogêneas – para as aplicações testadas. Em todas as figuras é utilizada a notação "*noc_hom*" para identificar arquiteturas de NoCs *homogêneas* e "*noc_het*", para as *heterogêneas*.

Um padrão comum de desempenho para as aplicações testadas refere-se ao desempenho do barramento. Pode-se perceber claramente para as três aplicações que a latência média cresce exponencialmente, à medida que o paralelismo cresce, na execução das tarefas das aplicações. Além disso, a latência média do barramento é muito superior às latências médias das NoCs homogêneas e heterogêneas. Apesar de este ser um resultado esperado, pelo fato de barramentos implicarem em comunicações seqüenciais, é interessante observar o grau de otimização que arquiteturas do tipo NoC podem oferecer, frente aos barramentos. Para aplicações com alto grau de paralelismo, percebe-se claramente que barramentos tendem a não compatibilizar com a latência requerida.

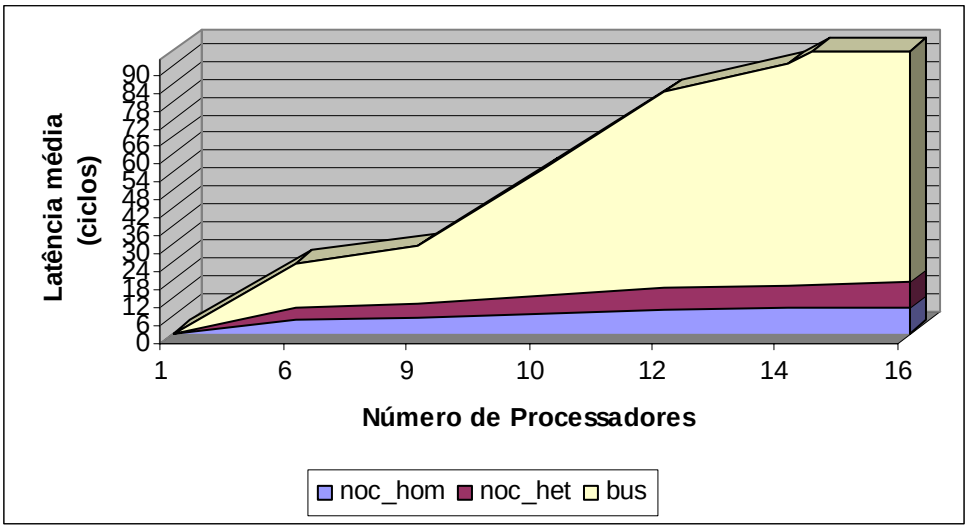


Figura 8.9: Resultados para arquiteturas de comunicação, aplicação roteamento IPv4 a 10Gb/s

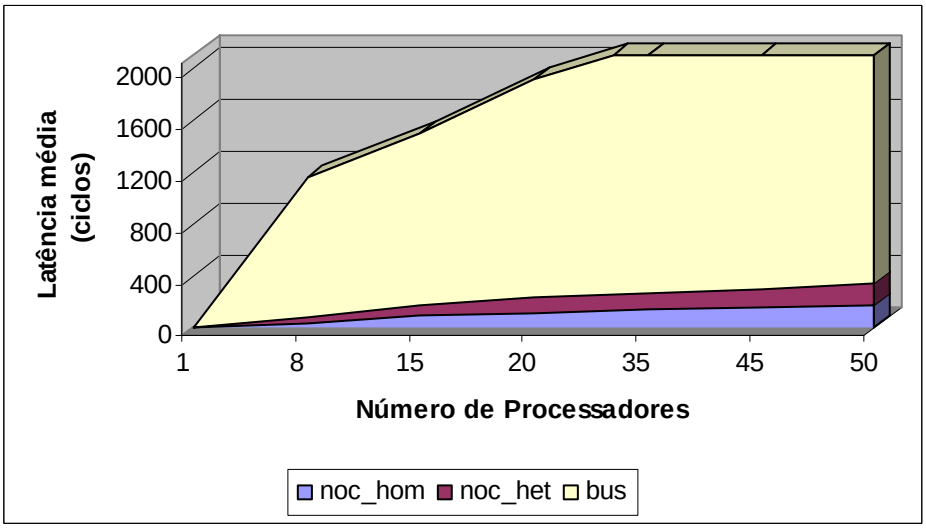


Figura 8.10: Resultados para arquiteturas de comunicação, aplicação para análise de ativos farmacológicos complexos

Outra constatação interessante refere-se à semelhança de latência entre as redes homogêneas e heterogêneas para as aplicações de roteamento IPv4 e análise de ativos farmacológicos. Em todas as aplicações, ambas possuem latência média bastante inferior ao barramento, mas para essas duas aplicações, não diferem em mais de 10%. Esse resultado é especialmente interessante, uma vez que – como visto no capítulo anterior – as redes heterogêneas, pela mistura de roteadores de mais baixo custo em termos de área e energia, conseguem melhores resultados para essas restrições. Por exemplo, uma rede com três roteadores do tipo Rasoc, três Tonga e três Mago, consome aproximadamente 40% menos energia e possui 35% menos área que uma rede composta somente por roteadores Rasoc. Esse é caso que ocorre quando nove processadores são utilizados. Esse resultado pode ser atribuído à execução conjunta das operações de particionamento e mapeamento, as quais puderam encontrar para essas aplicações, posições otimizadas para as tarefas e para os processadores.

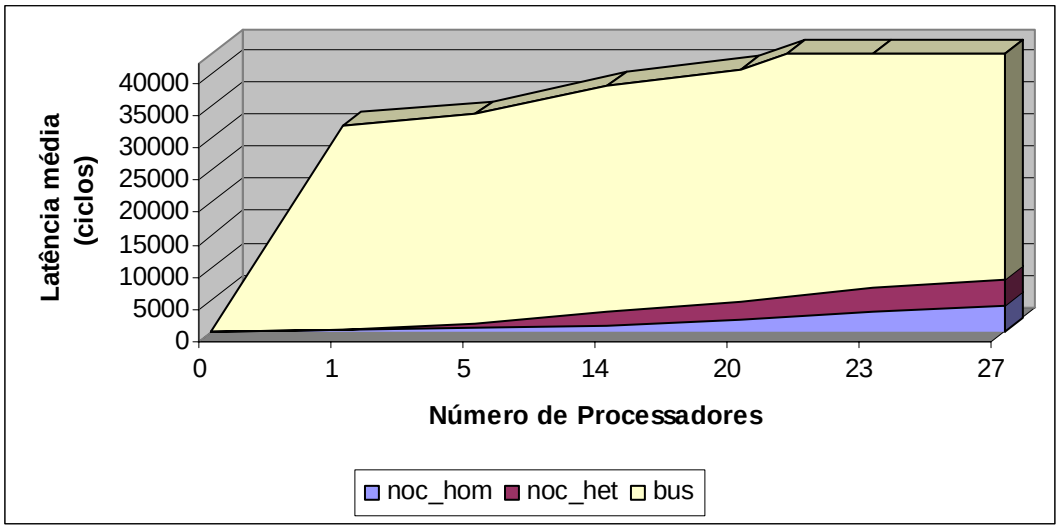


Figura 8.11: Resultados para arquiteturas de comunicação, aplicação Jogo de Futebol

Apenas para a aplicação do jogo de futebol a latência média da rede heterogênea é superior (em torno de 55%) em comparação à rede homogênea. Entretanto, quando o máximo paralelismo é empregado, ou seja, uma tarefa por processador, a diferença cai para apenas 5%, aproximadamente, culminando com desempenho praticamente idêntico para 27 processadores, como demonstrado na figura 8.12

As figuras 8.13 a 8.15 referem-se a resultados obtidos para arquiteturas de processamento.

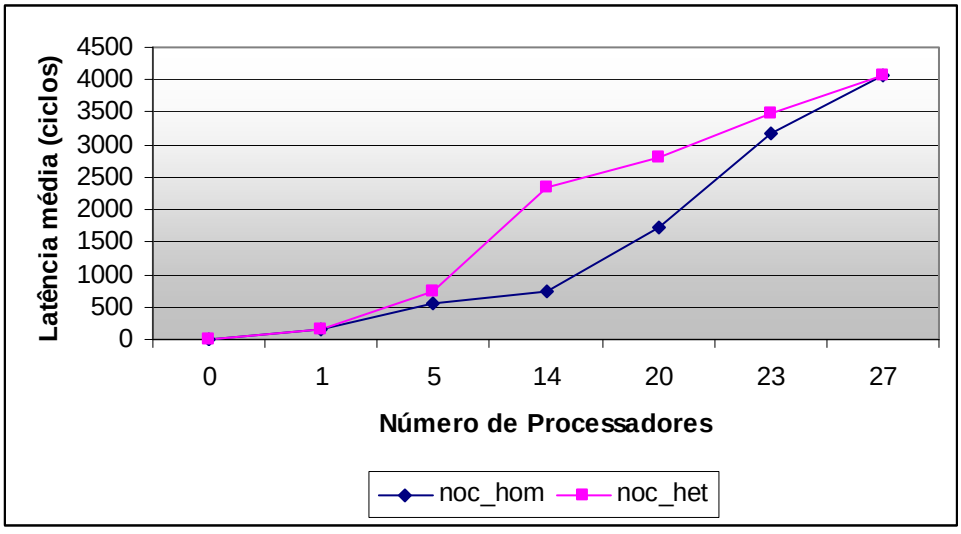


Figura 8.12: Resultados para arquiteturas de comunicação do tipo NoC, aplicação do Jogo de Futebol

Em relação às arquiteturas de processamento, pode-se verificar que, proporcionalmente, a maior otimização ocorre em processadores de propósito geral. Isto demonstra que a simples execução concorrente de tarefas consegue aumentar significativamente o desempenho em aplicações dedicadas. Além disso, se o conjunto de processadores RISC for composto por processadores iguais, o consumo de energia é reduzido proporcionalmente ao número de ciclos economizados pela distribuição de tarefas, como discutido na seção 8.3.

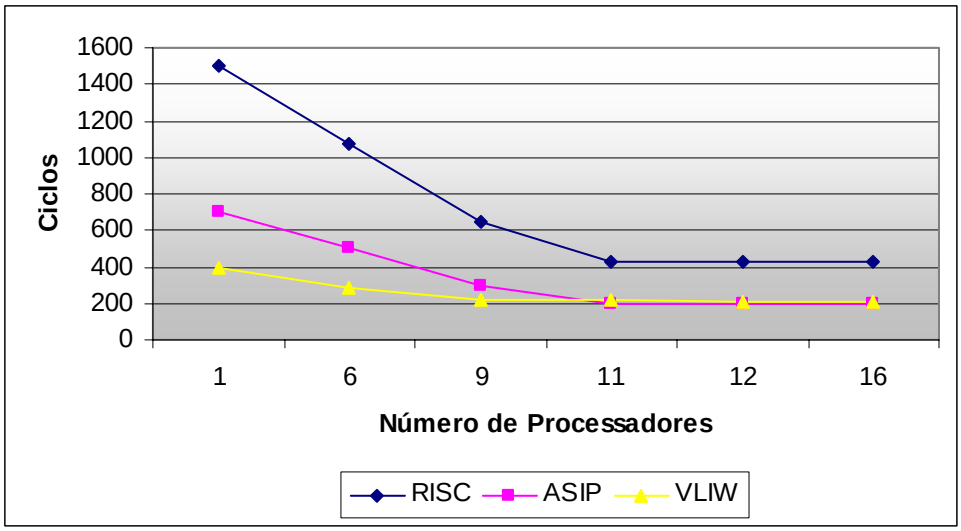


Figura 8.13: Resultados para arquiteturas de processamento, aplicação de roteamento IPv4 a 10Gb/s

O mesmo não acontece para processadores heterogêneos (ASIPs e VLIW). Para as estratégias de otimização baseadas na inserção de unidades dedicadas ou de unidades funcionais, verifica-se que a abordagem VLIW mostra-se mais eficiente para as aplicações testadas. Isso pode ser atribuído ao fato de que nenhuma aplicação faz uso de instruções específicas, as quais poderiam ser agressivamente otimizadas pela execução em unidades dedicadas. Além disso, as unidades extras adicionadas nos processadores VLIW podem ser utilizadas pelas instruções de todas as tarefas e não somente por aquelas implementadas pela unidade dedicada.

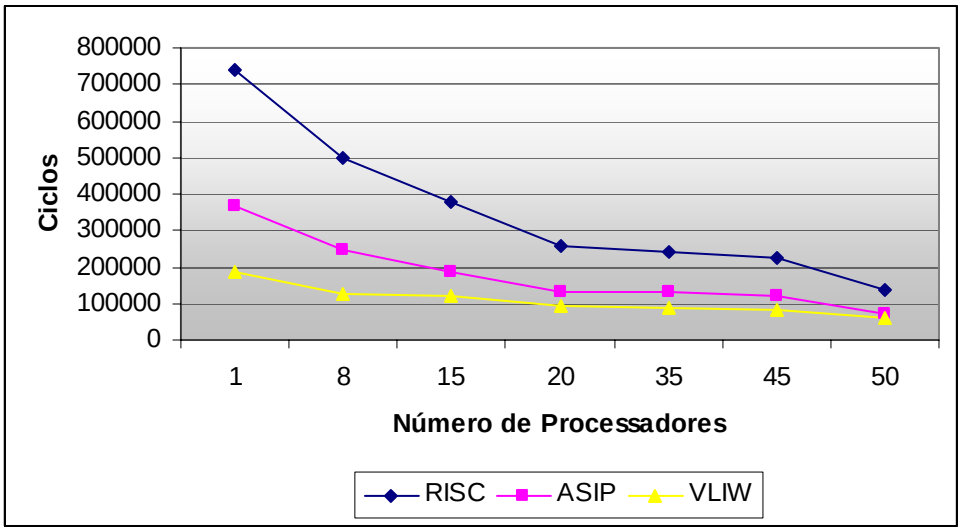


Figura 8.14: Resultados para arquiteturas de processamento, aplicação para análise de ativos farmacológicos complexos

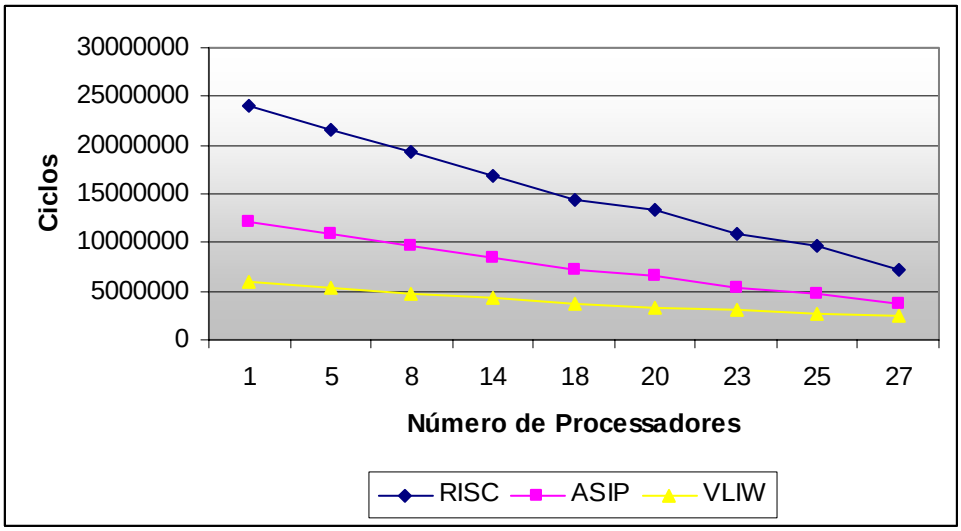


Figura 8.15: Resultados para arquiteturas de processamento, aplicação Jogo de Futebol

Mesmo assim, é interessante observar que, quando o número de processadores aumenta, as curvas relativas a essas duas otimizações tendem a se encontrar, em todas as aplicações testadas. Assim sendo, realização do SoC por arquiteturas VLIW ou ASIP pode ser determinada em função do consumo de energia, quando mais da metade das tarefas executam concorrentemente.

Para a aplicação para análise de ativos farmacológicos complexos, a concorrência na execução das tarefas permite que processadores de propósito geral possuam desempenho apenas 40% inferior a arquiteturas heterogêneas, quando 50 processadores são utilizados. Nesse caso, essa é a melhor opção para a implementação do SoC, a não ser que o desempenho seja muito crítico. Além disso, arquiteturas VLIW e ASIP possuem praticamente o mesmo desempenho. Isto pode ser creditado a otimização em HW do cálculo do fitness. Já na aplicação de roteamento IPv4, essas duas classes arquiteturais possuem praticamente o mesmo desempenho quando metade das tarefas executam concorrentemente. Para essa aplicação, a realização em HW das funções mostra-se mais eficiente provavelmente pela homogeneidade funcional: praticamente todos os processadores executam a mesma função de roteamento.

Outro experimento interessante dedica-se à avaliação da influência que a complexidade arquitetural de processadores pode exercer sobre as restrições de projeto, na execução seqüencial$_x$concorrente de tarefas. Com isso espera-se encontrar o compromisso complexidade arquitetural$_x$concorrência que melhor atenda às restrições de projeto para aplicações dedicadas.

A idéia principal por trás desse experimento refere-se a avaliar o compromisso entre o uso de apenas um (ou alguns poucos) processador(es) complexo(s) para a execução seqüencial de tarefas ou a execução concorrente destas, em um conjunto de processadores simples. A complexidade abordada é focada no número de unidades funcionais dos processadores. Portanto, nesse experimento, inicialmente todas as tarefas da aplicação são alocadas para um único processador do tipo VLIW, o qual possui várias unidades funcionais. Para os testes, o processador foi configurado inicialmente com um número de unidades funcionais igual à metade do número de tarefas da aplicação. Em seguida, as tarefas são progressivamente alocadas para um número maior de processadores. No entanto, nessa abordagem, à medida que um número maior de processadores é utilizado, estes vão se tornando mais simples: quando o número de

processadores for igual à metade do número de tarefas, cada processador possui apenas uma unidade funcional. Isto é denotado nas figuras 8.16 a 8.18 pela nomenclatura VLIW2RISC, as quais demonstram os resultados obtidos segundo essa abordagem. Os processadores VLIW utilizados nesses experimentos seguem o mesmo princípio utilizado nos experimentos acima, onde unidades funcionais são acrescentadas a processadores RISC.

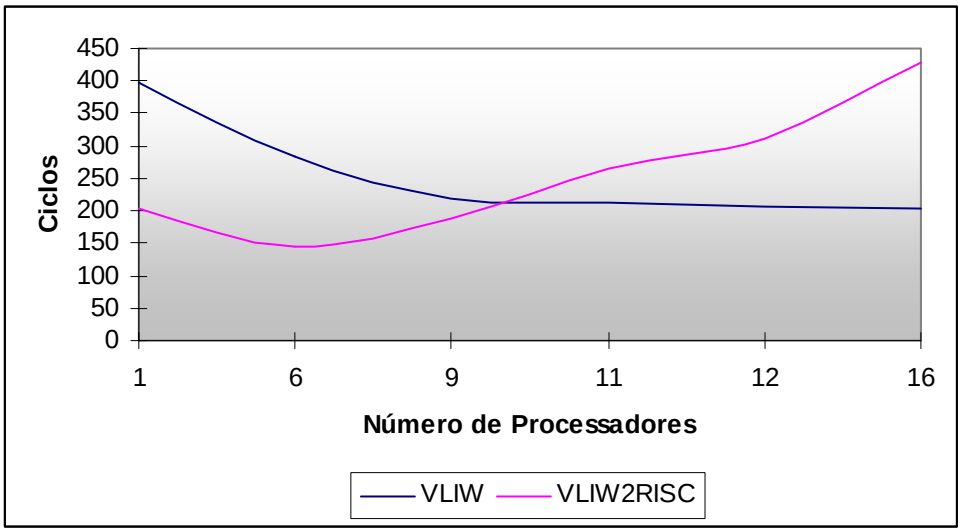


Figura 8.16: Resultados para arquiteturas de processamento, abordagem VLIW para RISC, aplicação de roteamento IPv4 a 10Gb/s

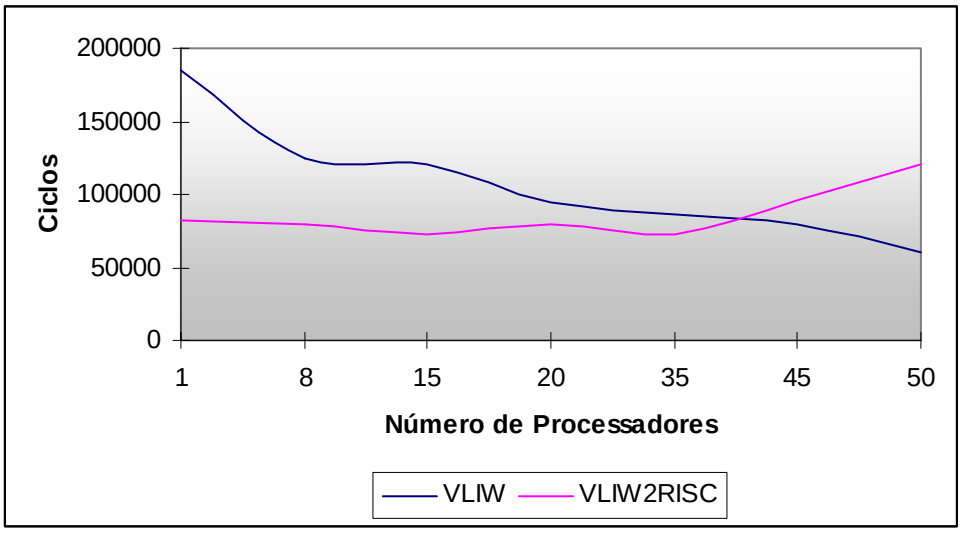


Figura 8.17: Resultados para arquiteturas de processamento, abordagem VLIW para RISC, aplicação para análise de ativos farmacológicos complexos

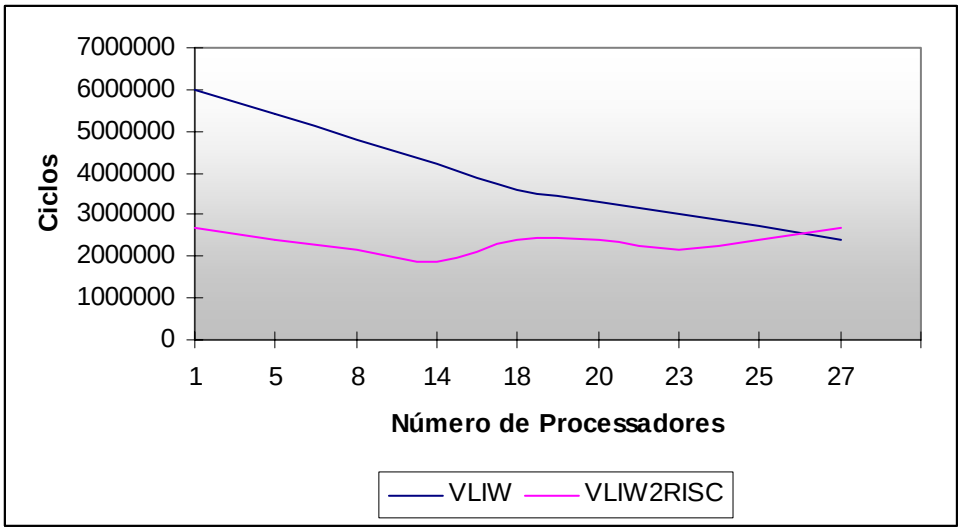


Figura 8.18: Resultados para arquiteturas de processamento, abordagem VLIW para RISC, aplicação do Jogo de Futebol

Como pode ser verificado nas figuras 8.16 a 8.18, processadores mais simples podem ser utilizados para a execução concorrente das tarefas de aplicações dedicadas. No entanto, existe um compromisso quanto ao número de processadores, que deve ser respeitado. Apenas para a aplicação do jogo de futebol o desempenho do conjunto de processadores RISC é semelhante ao conjunto com processadores VLIW. Isso pode ser atribuído a um posicionamento otimizado entre os jogadores de cada equipe. Esse resultado é especialmente interessante, uma vez que processadores mais simples implicam em maior redução do consumo de energia. Os processadores podem ser ainda mais simples, caso as instruções menos utilizadas pelas tarefas da aplicação fossem retiradas do conjunto de instruções.

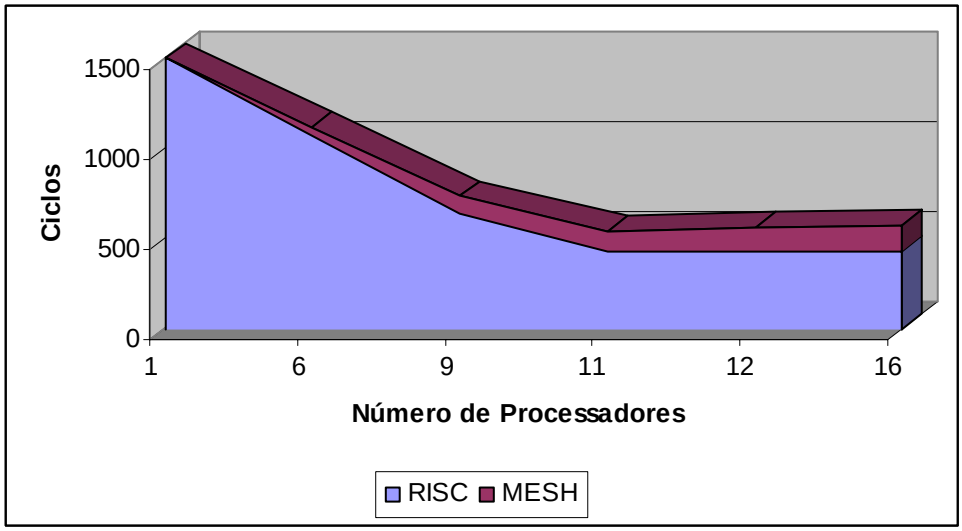


Figura 8.19: Compromisso Processamento$_x$Comunicação para a aplicação de roteamento IPv4 a 10Gb/s, processadores RISC

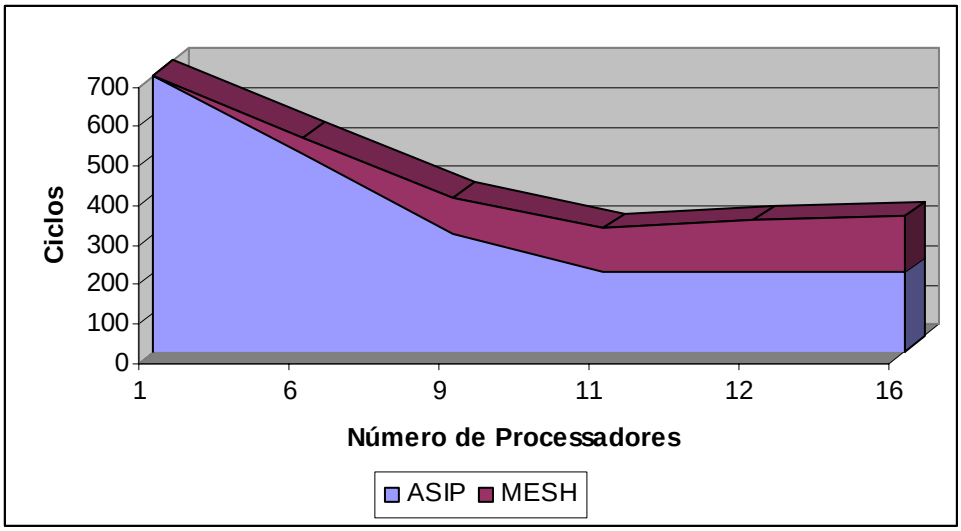


Figura 8.20: Compromisso Processamento$_x$Comunicação para a aplicação de roteamento IPv4 a 10Gb/s, processadores ASIP

Nas figuras 8.19 a 8.21 são mostrados os resultados obtidos para sistemas completos, onde é demonstrada a influência de cada classe arquitetural nas restrições de projeto da aplicação. Os resultados referem-se à execução da aplicação de roteamento IPv4 a 10Gb/s em processadores RISC, ASIP e VLIW, na topologia grelha. O número de ciclos corresponde à latência média necessária para o envio das mensagens para a NoC e ao número de ciclos que os respectivos processadores necessitam para executar as tarefas da aplicação.

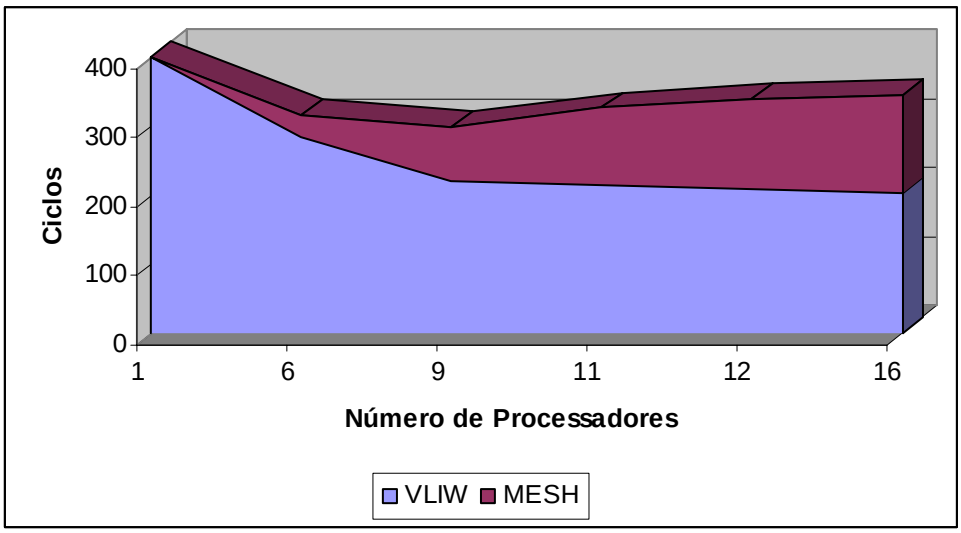


Figura 8.21: Compromisso Processamento$_x$Comunicação para a aplicação de roteamento IPv4 a 10Gb/s, processadores VLIW

O maior impacto de arquiteturas de comunicação sobre o desempenho pode ser verificado em sistemas implementados com processadores VLIW. Isso ocorre porque este tipo de processador possui desempenho superior e porque o desempenho da rede é praticamente igual para todas as arquiteturas. Isso se deve ao fato de que a implementação do particionamento realizada não considera o tipo dos processadores para os quais aloca tarefas. Dessa forma, o número de comunicações realizadas não muda em função do *tipo* do processador utilizado para a implementação do sistema. Por

exemplo, se mais tarefas fossem alocadas para processadores com mais unidades funcionais, otimizações em termos do número total de mensagens poderiam ser obtidas para este tipo de processador.

Por outro lado, pode-se verificar que arquiteturas de comunicação do tipo Network-on-Chip possibilitam a distribuição e conseqüente execução concorrente de tarefas, uma vez que, mesmo quando processadores heterogêneos são utilizados, o tempo total de execução da aplicação diminui. O mesmo não ocorre com barramentos, devido à grande latência decorrente do envio seqüencial das mensagens, como pode ser observado na figura 8.22, para um sistema com processadores RISC.

Finalmente, para todos os resultados, deve-se observar que, para cada processador onde mais de uma tarefa é executada, deve-se acrescentar o custo – de desempenho e consumo de energia – das tarefas de sistemas operacionais dedicados. No entanto, a avaliação dessa arquitetura de SW está além do escopo desse trabalho. Mesmo assim, esse custo é proporcional ao número de tarefas escalonadas pelo sistema operacional, além de funções intrínsecas a esses sistemas, como entrada/saída, as quais podem ser generalizadas para todos SoCs.

Uma vez que os experimentos foram conduzidos em níveis de abstração superiores ao RT, não foram medidos os valores para a função custo de área. No entanto, os valores para área são relativos ao número de componentes de processamento e de comunicação. Ainda, os sistemas multiprocessados, alvo dessa pesquisa, possivelmente serão implementados em *giga-escala*, o que torna a preocupação com a área menos importante do que com as demais restrições.

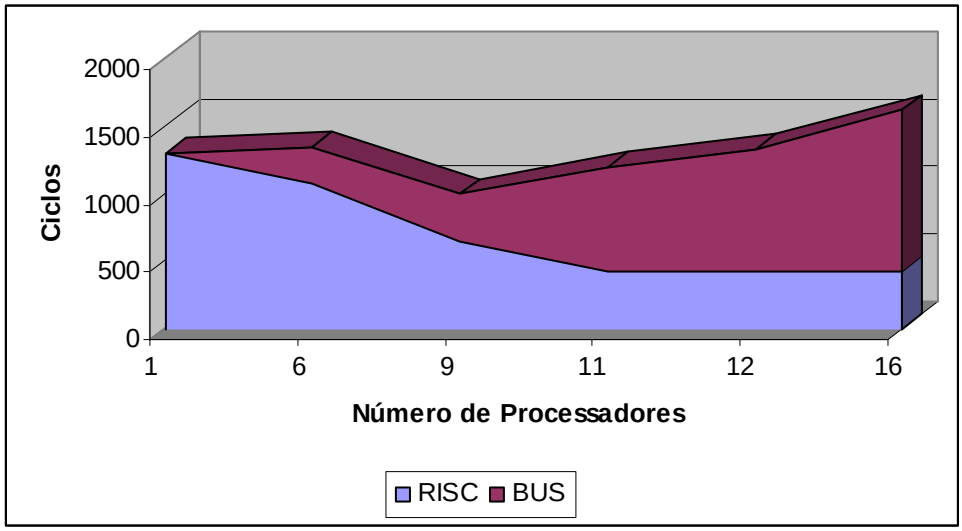


Figura 8.22: Compromisso Processamento$_x$Comunicação para a aplicação de roteamento IPv4 a 10Gb/s, processadores RISC e arquitetura de barramento

8.6 Conclusões

A avaliação de sistemas completos pode ser realizada através da execução simultânea das operações de particionamento e mapeamento. Além disso, cada uma dessas operações pode ter a sua complexidade aumentada pela avaliação de comportamentos específicos para os componentes arquiteturais de plataformas de processamento e comunicação. É o que acontece quando arquiteturas heterogêneas são consideradas, como por exemplo, processadores com unidades dedicadas ou redes com roteadores estruturalmente diferenciados em sua arquitetura.

Mesmo constituindo-se a avaliação de sistemas completos um problema extremamente complexo, é imperativo que se busquem soluções visando encontrar arquiteturas otimizadas, face aos novos sistemas multiprocessados esperados para o futuro próximo. Nessa direção, este capítulo procurou apresentar algumas soluções, visando avaliar implementações que consideram, para plataformas de processamento, processadores de propósito geral, com unidades dedicadas ou com um número maior de unidades funcionais e para plataformas de comunicação, redes homogêneas e redes heterogêneas, para restrições de desempenho e consumo de energia.

Considerar todo o espaço de busca para sistemas completos, equivale a executar um número de operações relativo à complexidade das operações de particionamento e mapeamento simultaneamente. Isso ocorre devido ao grande número de comportamentos passíveis de serem considerados para avaliação, tanto para componentes de processamento quanto de comunicação. Portanto, diversas possibilidades arquiteturais não foram consideradas nos experimentos realizados.

Mesmo assim, pode-se concluir que os experimentos realizados puderam cobrir possibilidades arquiteturais representativas para sistemas dedicados, pelo fato de considerarem arquiteturas heterogêneas. Dos experimentos se pode constatar a importância dos processadores com unidades dedicadas (ASIPs) ou com número diferenciado de unidades funcionais para o desempenho de operações específicas, bem como a influência das arquiteturas de comunicação em sistemas distribuídos. Com base nos resultados obtidos, é possível avaliar o compromisso entre arquiteturas heterogêneas para processamento e comunicação que melhor atende as restrições de projeto para aplicações dedicadas distribuídas. Dessa forma, a avaliação e otimização conjunta de plataformas para processamento e comunicação permitem a concepção de sistemas multiprocessados heterogêneos.

Em relação ao tempo necessário à otimização das plataformas, pode-se destacar que, mesmo para um grande número de iterações para o algoritmo busca tabu, não foram necessários mais do que 3 ou 4 minutos em uma máquina com processador Pentium IV 2.8Ghz e 512 MB de memória. Isso comprova que a opção pela simulação de compontes executáveis mostrou-se eficiente para a otimização de sistemas completos.

9 CONSIDERAÇÕES FINAIS

A presente tese objetiva contribuir para o estado-da-arte na tecnologia de projeto de sistemas dedicados distribuídos, através da sugestão de um novo método para a modelagem e otimização de componentes arquiteturais de processamento e comunicação. A necessidade de novos métodos e ferramentas para a concepção de sistemas eletrônicos multiprocessados é constatada em previsões realizadas pela comunidade científica, as quais ditam sistemas complexos, tanto em número de componentes, quando em diversidades funcionais.

Nessa direção, foi proposto um novo método que define um fluxo de projeto em conjunto com as ferramentas associadas à sua efetivação. A abordagem adotada pelo método distingue semanticamente os comportamentos de processamento e comunicação de aplicações dedicadas. Como conseqüência, as ferramentas propostas para otimização arquitetural dedicam-se a explorar o espaço de projeto para estas duas classes de operações, visando encontrar soluções compatíveis com as restrições de projeto da aplicação alvo. Uma vez que sistemas multiprocessados são esperados para a realização dos futuros sistemas eletrônicos, é importante que a busca por componentes arquiteturais otimizados possa ser realizada separadamente sobre comportamentos de comunicação e processamento, devido à importância que cada um exerce nas restrições de projeto. A análise sobre componentes arquiteturais de processamento e comunicação pode ser realizada separadamente, devido à ortogonalidade existente entre estas operações.

Neste trabalho foram analisadas as três abordagens para a concepção de sistemas eletrônicos: 1) síntese a nível de sistema; 2) projeto baseado em componentes; e 3) projeto baseado em plataformas. O método aqui proposto sugere um fluxo de projeto segundo os conceitos do projeto baseado em plataforma, pois essa abordagem procura agregar as vantagens das outras duas, promovendo o reuso de propriedade intelectual e o co-projeto HW/SW em diversos níveis de abstração. Um dos aspectos mais interessantes dessa abordagem refere-se à possibilidade de implementação dos comportamentos da aplicação em vários componentes, disponibilizados em plataformas arquiteturais. Além disso, os conceitos relativos ao projeto baseado em plataformas vêm sendo apontados pela comunidade científica, como a solução mais provável a ser adotada para a concepção dos sistemas esperados num futuro próximo. Isso ocorre devido à complexidade dos futuros SoCs, a qual demanda reuso de propriedade intelectual, além da análise de componentes arquiteturais em diversos níveis de abstração.

Neste trabalho, um modelo de programação foi proposto, onde componentes arquiteturais podem ser especificados em quatro diferentes níveis de abstração. O modelo de programação é centrado nos conceitos do projeto baseado em interfaces, os quais permitem: 1) que os protocolos utilizados para a comunicação entre os

componentes sejam definidos em função de diferentes modelos de computação; 2) a especificação de variações funcionais para componentes de mesmo tipo; e 3) a realização de análises no metanível, o que permite análises dinâmicas e a simulação de componentes executáveis, através de introspecção computacional.

Ferramentas de apoio ao projeto foram sugeridas, com fins a simulação, reconfiguração e análise da compatibilidade de componentes com as restrições de projeto de aplicações dedicadas.

Espera-se que a visão aqui apresentada para a concepção de sistemas eletrônicos multiprocessados possa contribuir para que projetistas de sistemas venham a ter um suporte adequado para a especificação e exploração do espaço de busca para componentes arquiteturais em plataformas de comunicação e processamento. Nesse sentido, foram conduzidos experimentos onde algumas possibilidades arquiteturais para estas plataformas foram exploradas. Devido ao grande espaço de soluções que surge quando componentes de processamento e comunicação são considerados conjuntamente para a realização de sistemas, apenas alguns comportamentos em componentes arquiteturais foram considerados até o momento. No entanto, os experimentos realizados puderam comprovar a eficiência do método e das ferramentas propostas, na exploração e busca por plataformas arquiteturais otimizadas.

9.1 Evolução do trabalho

O prosseguimento deste trabalho ocorre naturalmente com a exploração de um número maior de comportamentos de componentes arquiteturais. Além disso, espera-se a realização de experimentos em níveis de abstração mais baixos, buscando-se maiores precisões para os resultados, bem como a síntese das plataformas selecionadas pelas ferramentas de otimização. Descrições em mais baixo nível de abstração também podem fornecer estimativas para a avaliação do consumo de energia, o que vem a ser de fundamental importância para a avaliação de sistemas compostos por múltiplos processadores, os quais podem consumir menos que um único processador complexo.

Ainda, outras heurísticas podem ser consideradas para a análise de componentes arquiteturais, visando maiores precisões para os resultados, explorando-se o comportamento dos componentes através de outros algoritmos de otimização. Uma alternativa interessante para a implementação da operação do particionamento pode considerar o *tipo* dos processadores analisados, o que pode vir a alterar o número de comunicações também em função do particionamento e não apenas em função do mapeamento. Por exemplo, mais tarefas poderiam ser alocadas, para processadores com mais unidades funcionais.

Como observado diversas vezes durante este texto, o enorme espaço de busca proporcionado pela busca conjunta de plataformas de comunicação e processamento otimizadas para a realização SoCs multiprocessados traduz-se em diversas possibilidades de ferramentas e métodos que podem ser aplicados na exploração de componentes arquiteturais. Espera-se que este trabalho possa ser interpretado com mais um passo nessa direção, mesmo que grandes desafios ainda se fazem presentes para que sistemas de tamanha complexidade possam ser eficientemente realizados. No entanto, tais desafios traduzem-se em motivação para prosseguir no desenvolvimento do método e das ferramentas aqui apresentadas.

REFERÊNCIAS

[ABD 2003] ABDI, S.; SHIN, D.; GAJSKY, D. Automatic communication refinement for system level design. In: DESIGN AUTOMATION CONFERENCE, 2003, Anaheim, CA. **Proceedings …** New York : ACM Press, 2003. p. 300-305.

[ALB 96] ALBA, C. A. **Sistema de Geração de Microcontroladores para Aplicações Específicas**. 1996. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[ALT 2002] ALTIZER, B.; COOKE, L.; MARTIN, G. Platform-based Desin: What is Required for a Viable SoC Design ? **Design and Reuse Electronic Magazin**. Disponível em: < http://www.us.design-reuse.com/socnews/?i=3340&u=1448>. Acesso em: jun 2005.

[ALT 2003] ALTERA SOPC Buider Product Specification. Disponível em: <http://www.altera.com/products/software/system/products/sopc/sop-index.html>. Acesso em: jun. 2005.

[ARC 2005] ARC Corporation web site. Disponível em: <http://www.arccores.com.> Acesso em: jun. 2005.

[ARN 99] ARNOUT, G. C for System Level Design. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munich. **Proceedings**... Los Alamitos: IEEE Computer Society, 1999. p. 384-386.

[ARM 2004] AMBA Bus Specification. Advanced Risc Machines Corp. Disponível em: <http://www.arm.com/armtech/AMBA?OpenDocument>. Acesso em: maio 2005.

[ARM 2005] ARM Data Engine Technology Specification. Disponível em: <http://www.arm.com.> Acesso em: jun. 2005.

[ART 2005] ARTISAN Real-Time Studio. Product Specification. Disponível em: <http://www.artisansw.com/>. Acesso em: jun. 2005.

[ATA 2003] ATASU, K.; POZZI, L.; IENNE, P. Automatic Applicaiton-Specific Instruction-Set Extensions under Microarchitectural Constraints. In: DESIGN AUTOMATION CONFERENCE, 2003, Anaheim, CA. **Proceedings …** New York : ACM Press, 2003. p. 256-261.

[AUG 2002] AUGUIN, M. et al. **CODEF:** A System-level Design Space Exploration Tool. In: IEEE INTERNATIONAL CONFERECNE ON ACOUSTIC SPEECH SIGNAL PROCESS, 2001. **Proceedings …** [S.l. : s.n.], 2002.

[AXE 97] AXELSSON, J. Architecture synthesis and partitioning of real-time systems: a comparison of three heuristic search strategies. In:

HARDWARE/SOFTWARE CODESIGN, 1997. **Proceedings …** [S.l. : s.n.], 1997. p. 161 - 165

[BAG 2001] BAGHDADI, A. et al. An Efficient Model for Systematic Design of Application-Specific Multiprocessor SOC. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 2001, Munich. **Proceedings**... Los Alamitos: IEEE Computer Society, 2001. p.55-63.

[BAL 2003] BALARIN, F.; WATANABI, Y.; HSIEH, H.; LAVAGNO, L.; PASSERONE, C.; VINCENTELLI, A. Metropolis: An Integrated Electronic System Design Environment. **Computer Magazine**, Berkeley, v. 26, n. 4, p. 45-52, April 2003.

[BEN 2001] BENNINI, L.; De MICHELI, G. Powering Networks-on-Chip. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 2001, Montreal, Canada. **Proceedings …** New York : ACM Press, 2001. p. 33-38.

[BEN 2002] BENNINI, L.; De MICHELI, G. Networks-on-Chip: A New SoC Paradigm. **IEEE Computer**, [S. l.], January 2002.

[BRA 2004] BRAUN, G. et al. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, New York, v. 23, n. 12, Dec. 2004.

[BRU 2000] BRUNEL, J.-Y, et al. COSY Communication IP's. In: DESIGN AUTOMATION CONFERENCE, 2000. **Proceedings ...** New York : ACM Press, 2000. p.406-409.

[BUR 97] BURGER, D.; AUSTIN, T. **The SimpleScaler Tool Set**, version 2.0. [S. l.]: University of Wisconsin-Madison Computer Sciences Departament, 1997. (Technical Report #1342).

[CAD 2002] CADENCE SYSTEMS DESIGN,INC. **Virtual Component Co-Design**. Disponível em: <http://www.cadence.com/products/vcc.html.> Acesso em: jun. 1005.

[CAR 96] CARRO, L. **Algoritmos e Arquiteturas para o desenvolvimento de Sistemas Computacionais**. 1996. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[CAR 99] CARRO, L.; KREUTZ, M.; WAGNER, F.; OYAMADA, M. System Synthesis and Processor Selection in the S^3E^2S Environment. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 12., 1999, Natal. **Proceedings ...** Los Alamitos: IEEE Computer Society, 1999. p. 146-149.

[CAR 2000] CARRO, L. et al. System Design and Synthesis for Multiple Behavior Applications. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 2000, Paris. **Proceedings …** Los Alamitos: IEEE Computer Society, 2000. p. 697.

[CAR 2004] CARDOZO, R. S. et al. Tonga: A Low Cost Router for NoCs. In: IBERCHIP, 2004, Cartagena de Indias, Colômbia. **Proceedings ...** [S.l. : s.n.], 2004.

[CES 2002] CESARIO, W. et al. Multiprocessor SoC Platforms: A Component-based Design Approach. **IEEE Design and Test of Computers**, [S. l.], v.19, n. 6, p. 52-63, Nov-Dec 2002.

[CHA 97] CHAIYARATANA, N.; ZALZALA, A.M.S. Recent developments in evolutionary and genetic algorithms: theory and applications. In: INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS IN ENGINEERING SYSTEMS: INNOVATIONS AND APPLICATIONS, 1997. **Proceedings …** [S.l.: s.n.] , 1997. p. 270 – 277.

[CHO 99] CHOU, P. et al. IP-Chinook: An Integrated IP-Based Design Framework for Distributed Embedded Systems. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 1999, New Orleans. **Proceedings** ... New York : ACM Press, 1999. p.44-49.

[CON 2004] CONG, J.; FAN, Y.; HAN, G.; ZHANG, Z. Application-Specific Instruction Generation for Configurable Processor Architectures. In: INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 2004. **Proceedings …** [S.l. : s.n.], 2004. p. 183-189.

[COW 2002] CO-WARE INCORPORATION WEB SITE. **N2C product specification**. Disponível em: <http://www.coware.com>. Acesso em: jun. 2005.

[DAV 97] DAVEAU, J. et al. Protocol Selection and Interface Generation for HW/SW CoDesign. **IEEE Transactions on VLSI Systems,** [S. l.]**,** v. 5, n. 1, p. 136-144, Mar. 1997.

[DIC 98] DICK, R. P.; RHODES, D. L.; WOLF, W. TGFF: task graphs for free. In: INTERNATIONAL WORKSHOP ON HARDWARE/SOFTWARE CODESIGN, 6., 1998, Seattle. **Proceedings …** Los Alamitos: IEEE Computer Society, 1998. p. 97 – 101.

[DUA 97] DUATO J.; YALAMANCHILI, S.; NI, L. **Interconnection Networks:** an Engineering Approach. Los Alamitos: IEEE Computer Society, 1997. 515 p.

[ERB 2003] ERBAS, C.; ERBAS, S.; PIMENTEL, A. A Multiobjective Optimization Model for Exploring Multiprocessor Mappings of Processor Networks. In: CODES-ISSS, 2003, Newport Beach, CA. **Proceedings …** New York : ACM-SIGDA Publications, 2003. p. 182-187.

[ELR 92] ELREWINI, L. **Introduction to Parallel Computing**. [S.l.]: Prentice Hall, 1992.

[FAR 88] FARMACOPÉIA Brasileira. 4. ed. São Paulo: Atheneu, 1988. p. 33-47.

[FAY 99] FAYAD, M. et al. **Building Application Frameworks:** Object-Oriented Foundations of Framework Design. [S.l.]: John Wiley & Sons, 1999.

[FAU 95] FAUTH, A.; van PRAET, J.; FREERICKS, M. Describing Instruction-Set Processors Using nML. In: EUROPEAN CONFERENCE ON DESIGN AND TEST, 1995, Paris. **Proceedings …** Los Alamitos: IEEE Computer Society, 1995. p. 503.

[GHA 2002] GHARSALLI, F.; LYONNARD, D.; MEFTALI, S.; ROUSSEAU, F.; JERRAYA, A. Unifying Memory and Processor Wrapper Architecture in Multiprocessor SoC Design. In: INTERNATIONAL SYMPOSIUM ON

SYSTEM SYNTHESIS, 2002, Kyoto. **Proceedings …** Los Alamitos: IEEE Computer Society, 2002. p. 26-31.

[GAJ 97] GAJSKI, D. **Principles of Digital Design.** [S.l.]: Prentice Hall, 1997

[GAU 2001] GAUTHIER, L. YOO, S; JERRAYA, A. Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Software. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, 2001, Munich. **Proceedings …** Los Alamitos: IEEE Computer Society, 2001. p. 679.

[GAU 2001b] GAUTHIER, L.; YOO, S.; JERRAYA, A. Automatic generation of application specific architectures for heterogeneous multiprocessor system-on-chip. In: DESIGN AUTOMATION CONFERENCE, 2001, New Orleans. **Proceedings …** New York : ACM Press, 2001. p. 518-523.

[GAR 79] GAREY, M. R.; JOHNSON, D.S. **Computers and Intractability:** A Guide to the Theory of NP-Completeness. San Francisco : W.H. Freeman and Co., 1979.

[GHO 99] GHOSH, A.; KUNKEL, J.; STAN, L. Hardware Synthesis from C/C++. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munich. **Proceedings ...** Los Alamitos: IEEE Computer Society, 1999. p. 387-389.

[GLA 92] GLASS, J.; NI, L. The Turn Model for adaptive routing. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 19. 1992, Gold Coast. **Proceedings...** Los Alamitos: IEEE Computer Society, 1992. p. 278-287.

[GLO 89] GLOVER, F. Tabu search – Part I. **ORSA Journal on Computing**, [S. l.], v. 1, p. 190-206, 1989.

[GLO 90] GLOVER, F. Tabu search – Part II. **ORSA Journal on Computing**, [S. l.], v. 1, p. 4-32, 1990.

[GLO 93] GLOVER, F.; TAILLARD, E; DE WERRA, D. A user's guide to the tabu search. **Annals of Operations Research**, [S. l.], v. 41, p. 3-28, 1993.

[GLO 2003] GLÖKLER, T.; HOFFMANN, A.; MEYR, H. Methodical Low-Power ASIP Design Space Exploration. **Journal of VLSI Signal Processing,** [S. l.], n. 33, p. 229-246, 2003.

[GME 2001] GME – UFRGS web site. Disponível em: <http://www.inf.ufrgs.br/gme> Acesso em: jun. 2005.

[GOE 2002a] GOESSLER, G.; SANGIOVANNI-VINCENTELLI, A. *Compositional Modeling in Metropolis* . In: EMSOFT, 2002, Grenoble. **Proceedings …** [S.l. : s.n.], 2002.

[GOR 95] GORDON, D. B. **Spectroscopic Techniques**. Cambridge: University Press, 1995.

[GOE 2002b] GOERING, R. Platform-based Design: A choice, not a panacea. **EETimes Magazine,** Sept. 2002. Disponível em: <http://www.eetimes.com/story/OEG20020911S0061>. Acesso em: jan. 2005.

[GRO 2002] GRÖTKER, T.; LIAO, S.; MARTIN, G.; SWAN, S. **System Design with SystemC**. [S. l.]: Kluwer Academic Publishers, 2002.

[GUE 2000] GUERRIER, P.; GREINER, A. A generic architecture for on-chip packet-switched interconnections. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munich. **Proceedings**... Los Alamitos: IEEE Computer Society, 1999. p. 250-256.

[HAL 99] HALAMBI, A. et al. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munich. **Proceedings**... Los Alamitos: IEEE Computer Society, 1999. p. 485.

[HOF 2002] HOFFMANN, A.; MEYR, H.; LEUPERS, R. **Architecture Exploration for Embedded Processors with LISA**. [S. l.]: Kluwer, 2002.

[IBM 2002] IBM MICROELECTRONICS. **The CoreConnect™ bus architecture**. Disponível em: <http://www.chips.ibm.com/products/coreconnect/docs/crcon_wp.pdf>. Acesso em: abr. 2005.

[IND 2003] INDRUSIAK, L. S. et al. Supporting Consistency Control between Functional and Structural Views in Interface-based Design Models. In: FDL, 2003, Frankfurt. **Proceedings …** [S.l : s.n.], 2003.

[INT 85] INTEL CORPORATION. **Microsystems Components Handbook**. [S.l.], 1985. v.1.

[IQB 94] IQBAL, A. et al. Partitioning of Image Processing Tasks on Heterogeneous Computer Systems. In: HETEROGENEOUS COMPUTING WORKSHOP, 1994. **Proceedings …** [S.l. : s.n.], 1994. p. 43 - 50.

[ITO 2000] ITO, S. A. et al. System Design Based on Single Language and Single-Chip Java ASIP Microcontroller. In: DESIGN AUTOMATION AND TEST IN EUROPE, 2000, Paris, France. **Proceedings...** [S. l.]: IEEE Computer Society Press, 2000. p.703-707.

[ITR 99] INTERNATIONAL Technology Roadmap for Semiconductors. 1999. Disponível em: <http://public.itrs.net/files/1999_SIA_Roadmap/Home.htm>. Acesso em: maio 2005.

[JAV 2004] JAVA Language Specifications. Disponível em: <http://java.sun.com>. Acesso em: jul. 2005.

[JUN 93] JUNQUEIRA, A. **Risco:** Microprocessador RISC CMOS de 32 bits. 1993. 256f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[KAH 74] KAHN, G. The Semantics of a Simple Language for Parallel Programming. In: IFIP CONGRESS, 1974. **Proceedings …** [S. l.]: North-Holland Publishing Co., 1974.

[KEU 2000] KEUTZER, K. et al. System Level Design: Orthogonalization of Concerns and Platform-based Design. **IEEE Transactions of Computer-aided Design of Circuits and Systems**, [S. l.], v. 19, n. 12, Dec. 2000.

[KEU 2002] KEUTZER, K. Programmable Platforms will Rule. **EETimes Magazine**, Sept. 2002. Disponível em: <http://www.eetimes.com/story/OEG20020911S0063>. Acesso em: jan. 2005.

[KIR 83] KIRKPATRICK, S. et al. Optimization by Simulated Annealing. **Science Magazine,** [S. l.], v. 220, n. 4598, p. 671-680, 1983.

[KNU 98] KNUDSEN, P; MADSEN, J. Integrating Communication Protocol Selection with Partitioning in Hardware/Software Codesign. In: INT. SYMPOSIUM SYSTEM LEVEL SYNTHESIS, 1998, Hsinchu, Taiwan. **Proceedings …** Los Alamitos: IEEE Computer Society, 1998. p. 111-116.

[KNU 99] KNUDSEN, P.; MADSEN, J. Integrating Communication Protocol Selection with Hardware/Software CoDesign. **IEEE Transactions on CAD of Integrated Circuits and Systems**, [S. l.], v. 18, n. 8, p. 1077-1095, Aug. 1999.

[KOG 2004] KOGEL, T. MEYR, H. Heterogeneous MP-SoC – The Solution to Energy-Efficient Signal Processing. In: DESIGN AUTOMATION CONFERENCE, 2004, San Diego. **Proceedings …** Los Alamitos: IEEE Computer Society, 2005. p. 686-691.

[KON 2004] KONZEN, P. H. A. et al. **Pesquisa, Otimização e Transferência de Processos Fermentativos para a produção de Produtos Cárneos.** Santa Cruz do Sul: UNISC, 2001-2004.

[KRE 97] KREUTZ, M. E. **Geração de Processador para Aplicação Específica**. 1997. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[KRE 2001] KREUTZ, M. et Al. Communication Architectures for System-On-Chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2001, Pirinópolis. **Proceedings …** Los Alamitos: IEEE Computer Society, 2001. p. 14.

[KRE 2005] KREUTZ, M. et al. Energy and Latency Evaluation of NoC Topologies. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, 2005, Kobe. **Proceedings …** Los Alamitos: IEEE Computer Society, 2005.

[KUM 96] KUMAR, S. et al. Object-Oriented Modeling of Harware for Embedded Systems. **Current Issues in Electronic Modeling,** [S. l.], v. 7, p. 15-37, 1996.

[KUN 99] KUHN, T. et al. Description and Simulation of Hardware and Software Systems with Java. In: DESIGN AUTOMATION CONFERENCE, 1999, New Orleans. **Proceedings** ... New York : ACM Press, 1999. p.515-521.

[LAR 2004] LARMAN, C. **Applying UML and Patterns:** An Introduction to Object-Oriented Analysis and Design and the Unified Process. [S.l.]: Prentice Hall PTR, 2004.

[LEA 2001] LEARDI, R. **Journal of Chemometrics**, [S. l.], n. 15, p. 559-569.

[LEE 95] LEE, E; PARKS, T. Dataflow Process Networks. **Proceedings of the IEEE**, [S. l.], v. 83, n. 5, p. 773-801, May 1995.

[LEE 98] LEE, E; VINCENTELLI, A. A Framework for Comparing Models of Computation. **IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems**, [S. l.], v. 17, n. 12, Dec. 1998.

[LEE 2001] LEE, E. Computing for Embedded Systems. In: IEEE INSTRUMENTATION AND MEASUREMENT TECHNOLOGY CONFERENCE, 2001, Budapest, Hungary. **Proceedings …** Los Alamitos: IEEE Computer Society, 2001.

[LEE 2002] LEE, E. Embedded Software. **Advances in Computers**, London, v. 56, 2002.

[LEE 2003] LEE, E. **Overview of the Ptolemy Project**. Berkeley, CA : University of California, 2003. (Technical Memorandum n. UCB/ERL M03/25).

[LEI 2003] LEI, T.; KUMAR, S. A Two-step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture. In: DIGITAL SYSTEMS DESIGN, 2003. **Proceedings …** [S.l. : s.n.], 2003. p. 80 - 187.

[MAR 2001] MARTIN, G.; LAVAGNO, L.; LOUIS-GUERIN, J. Embedded UML: A merger of Real-Time UML and Co-Design. In: INTERNATIONAL CONFERENCE ON HARDWARE SOFTWARE CODESIGN, 2001, Copenhagen. **Proceedings …** New York : ACM Press, 2001. p. 23-28.

[MAR 2003] HU, J.; MARCULESCU, R. Energy-Aware Mapping for Tile-based NoC Architectures under Performance Constraints. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 2003. **Proceedings …** Los Alamitos: IEEE Computer Society, 2003. p. 233-239.

[MAR 2004] 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, 2004, Paris. **Proceedings …** Los Alamitos: IEEE Computer Society, 2004. p.234-239.

[MEN 2003] MENTOR Corporation web site. Disponível em: < http://www.mentor.com> Acesso em: jun. 2005.

[MEN 2005] MENTOR Corporation products specification. Disponível em : <http://www.mentor.com/products/embedded_software/nucleus_modeling/index.cfm> Acesso em: jun. 2005.

[MEN 2004] MENTOR Corporation training courses. Disponível em: <www.mentor.com/es/courses/index.cfm?crs=067193> Acesso em: jun. 2005.

[MEY 2003] MEYEROWITZ, T; PINELLO, C.; SANGIOVANNI-VINCENTELLI, A. A Tool for Describing and Evaluating Hierarchical Real-Time Bus Scheduling Policies. In: DESIGN AUTOMATION CONFERENCE, 2003, Anaheim, CA. **Proceedings …** New York : ACM Press, 2003. p. 312-317.

[MES 2003] THE MESCAL Project web site. Disponível em: <http://www.gigascale.org/mescal/> Acesso em: jun. 2005.

[MIC 99] De MICHELI, G. Hardware Synthesis from C/C++ Models. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munich. **Proceedings**... Los Alamitos: IEEE Computer Society, 1999. p. 382-383.

[MIC 2004] MURALI, S.; De MICHELI, G. SUNMAP: A Tool for Automatic Topology Selection and Generation for NoCs. In: DESIGN AUTOMATION CONFERENCE, 2004, San Diego, CA. **Proceedings …** New York : ACM Press, 2004. p. 914-919.

[MIH 2003] MIHAL, A.; KEUTZER, K. Mapping Concurrent Applications onto Architectural Platforms. In: JANTSCH, A.; TENHUNEN, H. **Networks-on-Chip**. Boston: Kluwer Academic Publishers, 2003. p. 39-59.

[MIG 2003] de MIGUEL, M. QoS Modelling Language for High Quality Systems. In: IEEE INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, 8., 2003. **Proceedings …** New York : ACM Press, 2003.

[MOO 75] MOORE, G. E. Progress in Digital Integrated Electronics. In: IEEE INTERNATIONAL ELECTRON DEVICES MEETING, 1975. **Proceedings …** Los Alamitos: IEEE Computer Society, 1975. p. 11.

[MUR 2004] MURALI, S.; De MICHELI, G. Bandwidth-Constrained Mapping of Cores onto NoC Architectures. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 2004, Paris. **Proceedings**... Los Alamitos: IEEE Computer Society, 2004. p.896-901.

[NAV 99] NAVEED, A. **Algorithms for VLSI Physical Design Automation.** 2nd ed. Boston: Kluwer Academic Publisher, 1999.

[OMG 2005] OBJECT Management Group web site. Disponível em: <http://www.omg.org> Acesso em: jul. 2005.

[ORT 98] ORTEGA, R.; BORRIELO, G. Communications Systems for Distributed Embedded Systems. In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, 1998. **Proceedings …** Los Alamitos: IEEE Computer Society, 1998. p. 437-444.

[PAS 2002] PASKO, R. et al. Techniques to Evolve a C++ based System Design Language. In: DESIGN AUTOMATION AND TEST IN EUROPE, 2002, Paris. **Proceedings …** Los Alamitos: IEEE Computer Society, 2002.

[PAT 2001] DESIGN Patterns documents. Disponível em: <http://www.enteract.com/~bradapp/docs/patterns-intro.html.> Acesso em: jul. 2005.

[PIE 2002] PAULIN, P. et al. StepNP: A System-Level Exploration Platform for Network Processors. **IEEE Design & Test of Computers**, [S. l.], v.19, n. 6, p. 17-26, Nov.-Dec. 2002.

[PIN 2002] PINTO, A. et al. Constraint-driven Communication Synthesis. In: DESIGN AUTOMATION CONFERENCE, 2002, New Orleans. **Proceedings …** New York : ACM Press, 2002. p.783.

[PRO 2004] DESIGN Patterns Book. Disponível em: <http://www.microelectronic.e-technik.tu-darmstadt.de/staff/lsi/stuff/hires/contfs.htm.> Acesso em: jul. 2005.

[QUI 94] QUINN, M. J. **Parallel Computing:** Theory and Practice. New York : Mc-Graw Hill, 1994. 446p.

[QUI 2004] QUINN, D. et al. A System Level Exploration Platform and Methodology for Network Applications Based on Configurable Processors. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 2004, Paris. **Proceedings …** Los Alamitos: IEEE Computer Society, 2004.

[RAJ 2001] RAJSUMAN, R. System-on-a-Chip: Design and Test. **Artech House Magazine,** [S. l.], July 2000.

[RAV 95] RAVIKUMAR, C. ; GUPTA, A. Genetic Algorithm for Mapping Tasks onto a Reconfigurable Paralell Processor. **Computers and Digital Techniques Magazine,** [S. l.], v. 142 , n. 2, p. 81 – 86, 1995.

[ROB 2005] ROBOCUP web site. Disponível em: <http://www.robocup.org>. Acesso em: jul. 2005.

[ROM 2001] BURDEN, R.; FAIRES, J. D. **Study Guide for Numerical Analysis**. New York: McGraw-Hill, 2001.

[ROW 2003] ROWSON, J. A.; SANGIOVANNI-VINCENTELLI, A. Interface-based design. In: DESIGN AUTOMATION CONFERENCE, 2003, Anaheim, CA. **Proceedings …** New York : ACM Press, 2003. p. 178-183.

[RUM 91] RUMBAUGH, J. et al. **Object-Oriented Modeling and Design**. [S.l.]: Prentice Hall, 1991.

[SHA 2001] SHAH, N. **Undestanding Network Processors**. Berkeley: Berkeley Dept. of Electrical Eng. and Computer Science, Univ. of California, 2001.

[SHA 2004] SHALLOWAY, A.; TROTT, J. **Design Patterns Explained :** A New Perspective on Object-Oriented Design. 2nd ed. [S.l.]: Addison-Wesley, 2004. 480 p.

[SMI 2004] SMITH, S. An Asynchronous GALS Interface with Applications. In: IEEE WORKSHOP ON MICROELECTRONICS AND ELECTRON DEVICES, 2004. **Proceedings …** Los Alamitos: IEEE Computer Society, 2002. p. 41-44.

[SPE 2004] SPECC Specification. Disponível em: <http://www.specc.gr.jp/eng/tech/index.htm>. Acesso em: jul. 2005.

[SUN 2005] JAVA Language Specification. Disponível em: <http://java.sun.com.> Acesso em: jul. 2005.

[SUS 2001] SUSIN, A. A. **Sistemas Digitais**: Uma Visão Integrada do Processo de Síntese. 2001. Relatório Técnico, PPGC, UFRGS.

[TAM 92] TAMIR, Y.; FRAZIER, J. A. Dynamically-Allocated, Multi-Queue Buffers for VLSI Communication Switches. **IEEE Transactions on Computers**, [S. l.], v. 41, n. 6, p. 725-737, 1992.

[TAM 97] TAMMEMAE, K. et al. **VLSI System Level Co-Design Toolkit AKKA**. Disponível em: <http://www.ele.kth.se/~nalle/codes/manual.html>. Acesso em: jun. 2005.

[TEN 2005] TENSILICA Corporation. Disponível em: <http://www.tensilica.com.> Acesso em: jul. 2005.

[THI 99] THIELE, L. et al . FunState: An Internal Design Representation for Codesign. In: ICCAD, 1999, San Jose, CA. **Proceedings …** New York : ACM Press, 1999. p. 524-544.

[TIW 94] TIWARI, V. et al. Power Analysis of Embedded Software: a First Step Towards Software Power Minimization. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems,** [S. l.], v. 2, n. 4, p. 437 - 445, 1994.

[UML 2005] UNIFIED Modeling Language Web Site. Disponível em: <http://www.uml.org>. Acesso em: jul. 2005.

[VAN 2001] VANMEERBEECK, G. et al. Hardware/Software Partitioning for Embedded Systems in OCAPI-xl. In: CODES, 2001, Copenhagen, Denmark. **Proceedings …** Los Alamitos: IEEE Computer Society, 2001. p. 30-35.

[VIN 2002] SANGIOVANNI-VINCENTELLI, A. Defining platform-based Design. **EEdesign Magazine.** Disponível em: <http://www.eedesign.com/story/OEG20020204S0062> Acesso em: jun. 2005.

[VSI 2000] VSI Alliance. Virtual Component Interface Standard – OCB 2.1.0. [S. l.], 2000. 71p.

[WAK 99] WAKABAYASHI, K. C-based Synthesis Experiences with Behavior Synthesizer, 'Cyber'. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munich. **Proceedings**... Los Alamitos, California: IEEE, 1999. p. 390-393.

[WEB 2004] WEBER, S. et al. Fast Cycle-Accurate Simulation and Instruction Set Generation for Constraint-Based Descriptions of Programmable Architectures. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN, 2004, Stockholm. **Proceedings …** Los Alamitos: IEEE Computer Society, 2004.

[WIE 2004] WIEFERINK, A. et al. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 2004, Paris. **Proceedings …** Los Alamitos: IEEE Computer Society, 2004. p. 1256-1261.

[WIL 2002] WILSON, R. Panel find many ways to build a Platform. **EETimes Magazine**, Jan. 2002. Disponível em: <http://www.eedesign.com/story/OEG20020130S0057>. Acesso em: jun. 2005.

[WIL 2003] WILD, T.; FOAG, J.; PAZOS, N.; BRUNNBAUER, W. Mapping and Scheduling for Architecture Exploration of Networking SoCs. In: INTERNATIONAL CONFERENCE ON VLSI DESIGN, 2003, Las Vegas. **Proceedings ...** [S.l. : s.n.], 2003. p. 376 – 381.

[WIN 2001] WINGARD, D. Micronetwork-based Integration for SoCs. In: DESIGN AUTOMATION CONFERENCE, 2001, New Orleans. **Proceedings …** New York : ACM Press, 2001.

[XIL 2002] XILINX Corporation web site. Disponível em: <http://www.xilinx.com.>. Acesso em: jul. 2005.

[ZEF 99] ZEFERINO, C. A. **Redes de Interconexão para Multiprocessadores**. 1999. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[ZEF 2002] ZEFERINO, C. A.; KREUTZ, M.; CARRO, L.; SUSIN, A. A Study on Communication Issues for Systems-on-Chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, 2002, Porto Alegre. **Proceedings …** Los Alamitos: IEEE Computer Society, 2002.

[ZEF 2003] ZEFERINO, C. A.; SUSIN, A. A. SoCIN: A Parametric and Scalable Network-on-Chip. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, 2003, São Paulo. **Proceedings …** Los Alamitos: IEEE Computer Society, 2003.

[ZOR 97] LEE, H. B.; ZORN, B. G. BIT: a Tool for Instrumenting Java Bytecodes. In: USENIX SYMPOSIUM ON INTERNET TECHNOLOGIES AND SYSTEMS, 1997, Monterey, CA. **Proceedings …** [S.l. : s. n.], 1997.

[XIE 2003] XIE, H. et al. Architectural-Analysis and Instruction-Set Optimization for Design of Network Protocol Processors. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN, 2003, Stockholm. **Proceedings …** Los Alamitos: IEEE Computer Society, 2003. p. 225-230.