Proposta e Simulação de uma Arquitetura a Fluxo de Dados de Segunda Geração

Patrícia Magna

Tese apresentada ao Instituto de Física de
São Carlos, da Universidade de São Paulo,
para obtenção do título de **Doutor** em
Ciências: **Física Aplicada**.

Orientador: Prof. Dr. Carlos Antônio Ruggiero

DEPARTAMENTO DE FÍSICA E INFORMÁTICA
SÃO CARLOS - 1997

Magna, Patrícia

Proposta e Simulação de uma Arquitetura a Fluxo de Dados de Segunda Geração/ Patrícia Magna.—São Carlos, 1997.

143p.

Tese (Doutorado)—Instituto de Física de São Carlos, 1997.

Orientador: Prof. Dr. Carlos Antônio Ruggiero

1. Arquiteturas Paralelas não Convencionais. 2. Particionamento de Grafos de Fluxo de Dados. I. Título.

**UNIVERSIDADE
DE SÃO PAULO**

Instituto de Física de São Carlos

Av. Dr. Carlos Botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 272-6222

Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE
PATRÍCIA MAGNA APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS,
UNIVERSIDADE DE SÃO PAULO, EM 04/03/1997.

COMISSÃO JULGADORA:

Prof. Dr. Carlos Antonio Ruggiero (IFSC/USP)

Profa. Dra. Líria Matsumoto Sato (POLI/USP)

Prof. Dr. Edson dos Santos Moreira (ICMSC/USP)

Dr. Eduardo Whitaker Bergamini (INPE)

Prof. Dr. Alvaro Garcia Neto (IFSC-USP)

Dedicatória

À minha mãe Hortência, e às minhas irmãs, Adriana e Jocelí, pelo carinho e incentivo.

Ao meu pai Jorge pela grande mensagem deixada de sua existência.

Ao Giulio pelo amor, amizade e companheirismo nos longos dias de estudo.

Agradecimentos

Ao Prof. Dr. Carlos Antônio Ruggiero, cuja orientação e amizade muito contribuíram para elaboração deste trabalho e para minha formação profissional.

Aos professores Álvaro, Jan, Valentin e Gonzalo cujos ensinamentos ofereceram-me ótimas oportunidades de aprendizado.

Aos integrantes do grupo *Dataflow* do FFI Marcos Cavenaghi. José Ricardo (Barney), João Eduardo Perea, João Ângelo e Fernando pelas discussões sempre ricas em informações.

À Claudia. Nilzelí e Maria Helena, pela amizade e pelo competente trabalho de secretaria.

Aos funcionários do FFI Lírío, Marcos, Ailton, Ivanilda, Benê, Claudio, Bruno e Valdir pelo suporte técnico.

Aos colegas Rodrigo (Belém), Marcelo (Format) e Maximiliam pela instalação e suporte dos equipamentos computacionais utilizados.

Ao CNPq pelo apoio financeiro.

Aos amigos Márcio, Raul, Andrea, Dener e João Eduardo que acompanharam de perto a elaboração desta tese e aos amigos Márcia, Massami e Henrique que mesmo de longe sempre apoiaram-me.

À minha “nova” família Sr. Franco, D. Daidy, Mario e Cristina pelo apoio e amizade. E à querida Larissa pela alegria e carinho.

Ao meu “cunhado” Lauro pelo apoio dado a mim e à minha família.

E a todas as pessoas que direta ou indiretamente contribuíram na realização deste trabalho.

Conteúdo

1	Introdução	1
2	Evolução das Arquiteturas a Fluxo de Dados	4
2.1	O Modelo a Fluxo de Dados	5
2.2	A Primeira Geração de Máquinas a Fluxo de Dados	7
2.2.1	Máquina de Fluxo de Dados de Manchester	7
2.2.1.1	Chave	9
2.2.1.2	Unidade de Armazenamento	9
2.2.1.3	Unidade de Emparelhamento	9
2.2.1.4	Unidade de Instrução	11
2.2.1.5	Unidade de Processamento	12
2.2.1.6	Unidade de Armazenamento de Estruturas de Dados	12
2.2.2	Máquina de Fluxo de Dados do MIT - MTTDA	12
2.2.3	Considerações sobre as Arquiteturas de Primeira Geração	14
2.3	A Segunda Geração de Máquinas a Fluxo de Dados	16
2.3.1	Projeto Monsoon	18
2.3.2	*T	21
2.3.3	EM-4	24
2.4	Resumo	27
3	A Proposta de uma Nova Arquitetura a Fluxo de Dados	28
3.1	Descrição Geral da Arquitetura SEED	28
3.2	Unidade de Agrupamento	32
3.3	Chave 1	34
3.4	Chave 2	35

3.5	Unidade de Instruções Global	35
3.5.1	Armazenamento de Instruções	36
3.5.2	Requisição de Bloco de Código	37
3.6	Unidade de Estruturas de Dados	38
3.6.1	Memória de Estruturas	39
3.6.1.1	A Unidade de Armazenamento de Estruturas da MDFM	39
3.6.1.2	Adaptação da UAE para a Arquitetura SEED	41
3.6.2	Unidade de Processamento Vetorial	42
3.7	Unidades Funcionais	43
3.7.1	Unidade de Entrada de Blocos	46
3.7.2	Tabela de Rótulos	48
3.7.3	Unidade de Requisição de Blocos	48
3.7.4	Unidade de Agrupamento Local	50
3.7.5	Unidade de Instrução	51
3.7.5.1	Armazenamento de Instruções	52
3.7.5.2	Agrupamento do Código aos Dados	54
3.7.6	Processador	54
3.8	Resumo	55
4	Definição e Avaliação do Conjunto de Instruções da Arquitetura SEED	57
4.1	Geração de Código na MDFM	58
4.2	Descrição da Análise Comparativa	60
4.2.1	Integração Binária	61
4.2.1.1	Código Gerado para o VAX	62
4.2.1.2	Código Gerado para a SPARC-2	64
4.2.1.3	Código Gerado para a MDFM	64
4.2.1.4	Resultados Obtidos	67
4.2.2	Criação de Matriz	68
4.2.3	Multiplicação de Matrizes	69
4.2.4	Análise dos Resultados da Análise Comparativa	70
4.3	Conclusão da Análise do Código	72
4.4	Características do Código da Arquitetura SEED	74

4.5	Resumo	75
5	Particionamento de Grafos de Fluxo de Dados	77
5.1	TAM	78
5.1.1	Resumo Geral do Sistema TAM	78
5.1.2	Grafo Dual	79
5.1.3	Particionamento do Grafo Dual	81
5.2	Adaptação do Método de Particionamento TAM para a Arquitetura SEED	83
5.2.1	O Grafo de Fluxo de Dados da Integração Binária	84
5.2.2	Definições Básicas	88
5.2.3	Utilização do Método de Particionamento	90
6	O Simulador da Arquitetura SEED	94
6.1	Descrição do Simulador	94
6.2	Medidas de Avaliação da Arquitetura Fornecidas pelo Simulador	100
6.3	Resumo	102
7	Avaliação da Arquitetura SEED	103
7.1	Descrição Geral dos Benchmarks	104
7.1.1	Multiplicação de Matrizes	104
7.1.2	Criação de Matrizes	104
7.1.3	Fatorial	105
7.2	Avaliação da Arquitetura SEED	106
7.2.1	Variação do Número de Unidades Funcionais	106
7.2.1.1	Descrição Preliminar	106
7.2.1.2	Discussão dos Resultados	107
7.2.2	Variação do Número de Rótulos Locais	114
7.2.2.1	Descrição Preliminar	114
7.2.2.2	Discussão dos Resultados	119
7.2.3	Variação do Número de Unidades de Agrupamento	120
7.2.3.1	Descrição Preliminar	120
7.2.3.2	Discussão dos Resultados	122

7.3	Comportamento da Arquitetura com a Variação do Tempo de Transmissão das Redes de Interconexão	127
7.3.1	Variação do Tempo de Transmissão da Chave 1	128
7.3.2	Variação do Tempo de Transmissão da Chave 2	130
7.4	Resumo	132
8	Considerações Finais	134
8.1	Conclusão	134
8.2	Trabalhos Futuros	136
8.2.1	Particionamento de Código para Processamento Paralelo	137
8.2.2	Implementação das Chaves de Interconexão	137
8.2.3	Utilização de Processadores Vetoriais em Fluxo de Dados	138
8.2.4	Geração de Aplicativos Computacionais	138
8.2.5	Análise em Nível de Hardware para Implementação da Arquitetura	138

Lista de Figuras

2.1	Grafo de Fluxo de Dados	6
2.2	Esquema da MDFM	8
2.3	Esquema da Unidade de Emparelhamento da MDFM	10
2.4	Esquema da MTTDA	13
2.5	Exemplo de Compilação de uma Expressão para e mecanismo ETS	19
2.6	Esquema da Arquitetura Monsoon	20
2.7	Elemento de Processamento do $*T$	22
2.8	Esquema da Arquitetura EM-4	25
3.1	Esquema Geral da Arquitetura SEED	29
3.2	Definição das Portas de Entrada de um Bloco de Código	31
3.3	Esquema de uma Unidade de Agrupamento	33
3.4	Informações Armazenadas em cada Posição da Memória Interna à UA	34
3.5	Esquema Geral da Unidade de Instrução Global	36
3.6	Esquema Geral da UAE da MDFM	40
3.7	Esquema Geral de uma Unidade Funcional	44
3.8	Esquema Geral de uma Unidade de Entrada de Blocos	47
3.9	Esquema Geral de uma Unidade de Requisição de Blocos	49
3.10	Esquema Geral de uma Unidade de Instrução	52
4.1	Geração de Código na MDFM	59
4.2	Programa SISAL para Integração Binária	62
4.3	Código Gerado para o VAX do Programa Integração Binária	63
4.4	Código Gerado para a SPARC-2 do Programa Integração Binária	65

4.5	Grafo de Fluxo de Dados do Programa de Integração Binária Gerado para MDFM	66
4.6	Programa SISAL para Criação de Matriz	68
4.7	Programa SISAL para Multiplicação de Matrizes	69
4.8	Gráfico da Razão entre o Número de Instruções <i>Overhead</i> para cada Instrução Útil para o Programa de Integração Binária	70
4.9	Gráfico da Razão entre o Número de Instruções <i>Overhead</i> para cada Instrução Útil para o Programa de Criação de Matriz	71
4.10	Gráfico da Razão entre o Número de Instruções <i>Overhead</i> para cada Instrução Útil para o Programa de Multiplicação de Matrizes	72
5.1	Nós Básicos do Grafo Dual	80
5.2	Combinações de Partições	82
5.3	Código da MDFM para Integração Binária	85
5.4	Grafo Dual da Integração Binária	89
5.5	Grafo Dual Particionado da Integração Binária	91
5.6	Código Executável do Programa de Integração Binária para a Arquitetura SEED	93
7.1	Programa SISAL de Multiplicação de Matrizes	104
7.2	Função SISAL para a Criação de Matrizes	105
7.3	Programa SISAL de Fatorial	105
7.4	Variação do Número de Unidades Funcionais - Multiplicação de Matrizes .	108
7.5	Variação do Número de Unidades Funcionais - Criação de Matrizes	109
7.6	Variação do Número de Unidades Funcionais - Fatorial	110
7.7	Variação do Número de Unidades Funcionais - Integração Binária	111
7.8	Variação do Número de Rótulos Locais - Multiplicação de Matrizes	115
7.9	Variação do Número de Rótulos Locais - Criação de Matrizes	116
7.10	Variação do Número de Rótulos Locais - Fatorial	117
7.11	Variação do Número de Rótulos Locais - Integração Binária	118
7.12	Ordem para Formação do Endereço da Unidade de Agrupamento	122

7.13 Variação do Número de Unidades de Agrupamento - Multiplicação de Matrizes 123

7.14 Variação do Número de Unidades de Agrupamento - Fatorial 124

7.15 Variação do Número de Unidades de Agrupamento - Integração Binária . . 125

7.16 Variação do Tempo de Transmissão da Chave 1 (Multiplicação de Matrizes) 129

7.17 Variação do Tempo de Transmissão da Chave 2(Multiplicação de Matrizes) 130

Lista de Tabelas

4.1	Número de Instruções Úteis e de <i>Overhead</i> do Programa Integração Binária para o VAX	62
4.2	Número de Instruções Úteis e de <i>Overhead</i> do Programa Integração Binária para a SPARC-2	64
4.3	Número de Instruções Úteis e de <i>Overhead</i> do Programa Integração Binária do Grafo de Fluxo de Dados	64
4.4	Resultados Obtidos para Integração Binária	67
4.5	Resultados Obtidos para Criação de Matriz	68
4.6	Resultados Obtidos para Multiplicação de Matrizes	70
5.1	Especificação das Instruções da MDFM que Foram Utilizadas pelo Programa de Integração Binária	86
6.1	Tabela dos Tempos Requeridos para a Unidade de Agrupamento efetuar suas Operações Possíveis	97
6.2	Tabela dos Tempos Requeridos para a Unidade de Entrada de Blocos efetuar suas Operações Possíveis	97
6.3	Tabela dos Tempos Requeridos para a Tabela de Rótulos efetuar suas Operações Possíveis	97
6.4	Tabela dos Tempos Requeridos para a Unidade de Agrupamento Local efetuar suas Operações Possíveis	97
6.5	Tabela dos Tempos Requeridos para a Unidade de Requisição de Blocos efetuar suas Operações Possíveis	98
6.6	Tabela dos Tempos Requeridos para a Unidade de Instrução Local efetuar suas Operações Possíveis	99

6.7 Tabela dos Tempos Requeridos para a Unidade de Instrução Global efetuar suas Operações Possíveis 99

7.1 Política de Distribuição de Fichas de Dados pela Chave 2 121

Resumo

Neste trabalho é apresentada a arquitetura SEED, proposta a partir das experiências adquiridas com as arquiteturas baseadas no modelo a fluxo de dados que foram estudadas até o presente.

A arquitetura SEED utiliza o modelo a fluxo de dados para escalonar e executar blocos de instruções, visando aproveitar a principal qualidade apresentada pelo modelo, que consiste em expor o máximo de paralelismo existente nos programas. No entanto, a arquitetura explora paralelismo de granularidade mais grossa que as arquiteturas a fluxo de dados, a fim de reduzir o tráfego de fichas de dados na arquitetura. Esta redução tenta resolver ou amenizar problemas como a excessiva ocupação de memória e a grande complexidade exigida do hardware.

Além da especificação da funcionalidade de toda a arquitetura SEED, este trabalho apresenta uma proposta para o particionamento do código. A utilização desta proposta permite a geração de blocos de códigos que podem ser executados corretamente pela arquitetura SEED.

Alguns *benchmarks* foram gerados utilizando essa proposta de particionamento de código. Estes *benchmarks* foram executados no simulador da arquitetura SEED, visando analisar e avaliar o comportamento da arquitetura com diversas configurações de hardware.

Abstract

In this work is presented the SEED architecture. This architecture was proposed considering the experiences obtained with existing architectures based on dataflow model.

The SEED architecture uses dataflow model to schedule and execute sets of instructions, called code blocks. This approach tries to make use of the main quality of the dataflow model, that is to expose the maximum paralelism of the programs. However, this architecture explores coarser granularity than the one usually considered in dataflow architectures in order to reduce the data token traffic in the architecture. This type of reduction tries to solve problems like excessive occupation of memory and high complexity of the hardware.

Besides the especification of all units that compose the SEED architecture, this work also proposes a way of partitioning programs, creating code blocks that may be executed by SEED architetur.

Some benchmarks were generated using this proposal for partitioning programs. These benchmarks were executed in the SEED architecture simulator, in order to analise the behavior of the proposed architecture under especial configurations.

Capítulo 1

Introdução

Desde que o modelo a fluxo de dados foi proposto em meados da década de 70, ele tem sido tema de intensa pesquisa. Esse interesse deve-se principalmente, à facilidade com que o paralelismo existente nas aplicações pode ser aproveitado pelo modelo.

Até o presente, várias arquiteturas baseadas no modelo a fluxo de dados foram propostas e estudadas. Através das análises realizadas com essas arquiteturas foram reveladas várias características positivas e negativas.

O objetivo deste trabalho é propor e avaliar uma nova arquitetura baseada no modelo a fluxo de dados que aproveite as qualidades apresentadas pelo modelo e que também, apresente soluções para os problemas revelados pelas arquiteturas que já foram propostas até o momento.

A estratégia adotada para o desenvolvimento do projeto foi:

- Realização de uma extensa revisão bibliográfica englobando as principais arquiteturas de fluxo de dados;
- Proposição de uma arquitetura;
- Proposição de um gerador de código para a arquitetura;
- Apresentar critérios que devem ser utilizados no processo de particionamento do código a ser executado pela arquitetura proposta;
- Construção de um simulador da arquitetura utilizando a linguagem C++;
- Análise e validação da proposta através de simulação por software.

A revisão bibliográfica, apresentada no capítulo 2, descreve as principais arquiteturas baseadas no modelo a fluxo de dados, destacando as características positivas e negativas de cada uma delas. Através dessa revisão pôde-se verificar que os projetos mais recentes incorporam características híbridas, como por exemplo, a junção do modelo von Neumann ao modelo a fluxo de dados, visando resolver alguns problemas revelados pelas arquiteturas mais antigas, ditas de primeira geração. Outra questão considerada pelos atuais projetos (segunda geração) envolve o aumento da granularidade do paralelismo a ser explorado pela arquitetura, uma vez que este ponto trouxe dois graves problemas às arquiteturas a fluxo de dados: o processamento de código estritamente sequencial ineficiente e hardware complexo decorrente da alta taxa de comunicação requerida pelo modelo.

O capítulo 3 é dedicado à apresentação da arquitetura proposta. A arquitetura denominada Sistema de Escalonamento e Execução *Dataflow* (SEED), é composta por várias unidades funcionais e unidades de sincronização de dados que podem operar paralelamente. Essa arquitetura considera que o aumento da granularidade do paralelismo a ser explorado pela arquitetura pode solucionar os problemas decorrentes da alta taxa de comunicação normalmente imposta pelo modelo a fluxo de dados. Esse aumento de granularidade pode ser obtido pela exploração de localidade de efeito dos dados gerados e consumidos em trechos restritos do programa. Dessa forma, para serem executados pela arquitetura SEED, os programas devem ser particionados em blocos de instruções, cuja execução em um único processador deve melhorar a eficiência com que as instruções da partição podem ser executadas. Para habilitar a execução de blocos de instruções, e para executar as instruções desses blocos dentro de cada unidade funcional, a arquitetura SEED utiliza o modelo a fluxo de dados.

A decisão do conjunto de instruções a ser utilizado pela arquitetura SEED foi realizada a partir de uma análise comparativa entre os códigos gerados para computadores convencionais RISC e CISC e o código utilizado por uma arquitetura a fluxo de dados (*Manchester Dataflow Machine*). Essa análise e a especificação do conjunto de instruções da arquitetura SEED são apresentadas no capítulo 4.

O capítulo 5 contém uma proposta para geração de programas particionados em blocos de código para serem executados pela arquitetura SEED.

As principais idéias utilizadas para construção do simulador são descritas no capítulo

6, onde também são apresentadas as medidas de avaliação da arquitetura fornecidas pelo simulador após o término da execução dos *benchmarks*.

No capítulo 7 são apresentados os resultados obtidos pela simulação de execução de alguns *benchmarks* pela arquitetura SEED. Com esses resultados pôde-se efetuar um estudo do comportamento da arquitetura SEED em função da variação da quantidade de recursos oferecidos pela arquitetura para exploração de paralelismo. As considerações finais e propostas de possíveis trabalhos futuros são apresentadas no capítulo 8.

Capítulo 2

Evolução das Arquiteturas a Fluxo de Dados

Desde que o modelo de fluxo de dados foi proposto por Dennis na década de 70, várias arquiteturas que utilizam esse modelo foram propostas.

A década de 80 teve, em seu início, uma explosão de interesse pelo modelo em função do alto paralelismo apresentado, surgindo vários projetos, como: a *MIT Tagged Token Architecture* (MTTDA) do MIT [ArNi 90], o projeto Sigma-1 no Japão [SHNS 86] [Hi 87], *Manchester Data Flow Machine* (MDFM) da Universidade de Manchester [GuKW 85] entre outras. Contudo, apesar da viabilidade do modelo ficar provada durante o desenvolvimento dos projetos, também foram identificados vários problemas, os quais comprometiam significativamente o bom desempenho das arquiteturas em questão. Esse fato fez com que o interesse pelo modelo declinasse em meados da década de 80.

No início da década de 90 porém, ressurgiu o interesse pelas máquinas de fluxo de dados pois, percebeu-se que os problemas anteriormente detectados ou são passíveis de solução com as novas tecnologias que surgiram, ou são inerentes ao processamento paralelo e, portanto, independentes do modelo computacional adotado. Pode-se falar do surgimento de uma segunda geração de fluxo de dados, já não tão preocupada em mostrar que o modelo é viável mas sim que ele é eficiente e pode ser a base dos futuros supercomputadores. Vários projetos de pesquisa surgiram, então, apresentando algumas modificações visando sanar especificamente os problemas apresentados. Alguns desses novos projetos podem ser considerados uma evolução de uma específica arquitetura de primeira geração, como a arquitetura EM-X/4 [Sakai 89][Yama 91] que surgiu do projeto Sigma-1 e o projeto

Monsoon [PaCu 91] que evoluiu da MTTDA. Outras arquiteturas propostas, baseiam-se fortemente nos resultados obtidos pelas arquiteturas de primeira geração de outros grupos de pesquisa, alguns exemplos são: a *Decoupled Data Driven Architecture with Vector and Macro Actor* [EvGa 90], a MDFA (*McGill Dataflow Architecture*) [GaHM 91], a arquitetura WOLF [CavG 95]. Alguns desses projetos de segunda geração já foram largamente estudados, e atualmente, apresentam características muito diferentes das originais, dando origem a novas arquiteturas como: a *T-NG [AnAC 92] que considera os resultados do projeto Monsoon e a proposta de uma aproximação com arquiteturas RISC [NiAr 89] e a arquitetura SAM *Super-Actor Machine* [Gao 93].

Neste capítulo serão apresentadas as principais características, positivas e negativas, das arquiteturas a fluxo de dados de primeira geração. E posteriormente, serão apresentadas algumas arquiteturas de segunda geração, analisando as alterações introduzidas em cada uma com o objetivo de eliminar os problemas, ou minimizá-los, e manter as boas qualidades do modelo a fluxo de dados.

2.1 O Modelo a Fluxo de Dados

O modelo de fluxo de dados baseia-se na exploração de paralelismo de granularidade bastante fina, no nível de instrução [ArGo 82]. Um programa deixa de ser representado por uma seqüência de instruções com um controle central impondo a ordem de execução, como é feito no modelo von Neumann, e passa a ser representado através de um grafo bidimensional.

A técnica de construção de grafos de fluxo de dados é utilizada em compiladores otimizadores [AlCo 76]. O método para sua construção é: instruções que podem ser executadas concorrentemente são escritas lado a lado e instruções que têm uma dependência de dados com outras são escritas umas sob as outras.

Para ilustrar a construção de um grafo de fluxo de dados, considere o trecho de um programa FORTRAN abaixo:

$$A = B + 1$$

$$C = D + E$$

$$F = A * C$$

A análise de dependência de dados é refletida através do grafo da figura 2.1. Neste grafo, nós representam operações e arcos, os caminhos percorridos pelos dados [GuW1 80]. O início do arco mostra onde a variável é atribuída e o fim, a instrução consumidora. Os dados são transmitidos através de fichas (*tokens*) pelos arcos como mensagens, sendo desnecessárias posições de memória globais, como no modelo convencional (von Neumann).

Figura 2.1: Grafo de Fluxo de Dados

O modo de execução do modelo de fluxo de dados é dirigido pelos dados (*data-driven*). As operações são consideradas habilitadas para a execução com a chegada de todos os dados de entrada. Quando uma instrução termina sua execução, seus resultados são enviados pelos arcos de saída; instruções que dependem destes dados podem então, ser habilitadas.

Este tipo de construção expõe todo paralelismo implícito no programa. Conseqüentemente, um hardware paralelo poderia ser bem utilizado sem a necessidade do programador preocupar-se em destacar os trechos de programa que podem ser executados em paralelo.

Sistemas a fluxo de dados são classificados pela forma de implementação de código reentrante. Dentro desta abordagem existem dois tipos [TrBH 82]: estático e dinâmico. O modelo estático não permite reentrância concorrente, o que o torna restrito para implementar blocos de repetição em paralelo e formas gerais de recursividade. São possíveis no modelo dinâmico duas soluções [Denn 81]:

- cópia de código: são feitas cópias das instruções que pertencem ao bloco reentrante e o destino das fichas é modificado dinamicamente em tempo de execução;

- rotulação de fichas: toda ficha contém, além do dado, um rótulo que a distingue de fichas que vão para a mesma instrução destino, mas pertencem a instâncias diferentes. O rótulo, geralmente, é dividido em três campos: nome de ativação, nível de iteração e índice.

O método de rotulação de fichas é o que permite criar ativações de código reentrante de forma mais eficiente, contudo implica no aumento de informações necessárias na ficha de dados. Essas informações extras impõem maior complexidade na operação de agrupamento dos dados de entrada de cada instrução diádica executada. Como poderá ser melhor explicado na descrição de arquiteturas de primeira geração, esse problema tornou-se alvo de vários estudos em função deste comprometer diretamente a exploração de paralelismo pelas arquiteturas a fluxo de dados.

2.2 A Primeira Geração de Máquinas a Fluxo de Dados

Nesta seção, serão descritos os dois maiores projetos de arquitetura a fluxo de dados considerados de primeira geração: da Universidade de Manchester e do MIT. Através dessas arquiteturas é possível compreender as vantagens do modelo a fluxo de dados e os principais problemas encontrados para utilização eficiente do modelo.

2.2.1 Máquina de Fluxo de Dados de Manchester

A máquina *Dataflow* de Manchester (MDFM) caracteriza-se por ser uma máquina de fluxo de dados dinâmica por rotulação de fichas, e será utilizada para apresentar os principais pontos positivos e negativos das arquiteturas a fluxo de dados.

Na MDFM, as instruções podem ter no máximo dois dados de entrada, ou seja, podem ser unárias ou binárias.

Pela figura 2.2, pode-se observar que a MDFM é formada por cinco unidades dispostas ao longo de um anel e uma unidade especial para o armazenamento de estruturas de dados. Este tipo de arquitetura explora paralelismo na unidade de processamento, que é constituída por vários processadores. Além disto, a implementação do anel com a utilização de *pipeline* possibilita a exploração de paralelismo temporal. Como as atividades

Figura 2.2: Esquema da MDFM

das unidades são independentes, as atividades na execução de cada unidade também podem ser realizadas em paralelo.

A comunicação é feita de forma assíncrona, unidirecionalmente e por passagem de mensagens.

As fichas são constituídas pelos seguintes campos: valor (32 bits), rótulo (36 bits) e instrução destino (18 bits). O fluxo de fichas dá-se no sentido horário e a primeira unidade pela qual passam é a chave (CH); esta tem como função unir fichas de procedências distintas: da unidade de armazenamento de estruturas de dados (UAE) [Sarg 85], do computador hospedeiro (*host*) e do próprio anel. Também é através da chave que vários anéis são unidos no caso da MDFM multi-anel [BaGu 87] [GuW2 80].

Uma ficha quando entra no anel através da CH passa à unidade de armazenamento (UA) que é apenas uma fila de fichas, e que tem como função regular o fluxo de fichas no anel. Quando a ficha chega à unidade de emparelhamento (UE) verifica-se, primeiramente, se a ficha destina-se a uma instrução unária ou binária (é permitido apenas esses dois tipos de instrução); se for binária é testado pelo rótulo e destino se a ficha parceira encontra-se esperando nesta unidade; se não, a ficha é armazenada. Se as fichas parceiras são agrupadas, um pacote formado pelas duas fichas vai para a unidade de instrução (UI). Se a ficha destina-se a uma instrução unária ela vai diretamente à UI. Na UI é encontrada

a instrução destino e é adicionado o código da operação ao pacote, obtendo-se assim o pacote executável. A unidade de processamento (UP) executa o pacote assim que um de seus processadores estiver vago.

2.2.1.1 Chave

A chave é a unidade que liga o anel à unidade de armazenamento de estruturas de dados ao computador hospedeiro (*host*). A partir de certos bits de controle é feito o reconhecimento para onde as fichas devem ser enviadas.

Para a máquina de fluxo de dados multi-anel a implementação da chave é mais complexa, pois torna-se importante considerar o método de separação de tarefas pelos anéis. Foi examinado como esta separação influencia na seleção da topologia da rede de interconexão da chave [BaGu 87].

2.2.1.2 Unidade de Armazenamento

A UA é utilizada para regular o fluxo de fichas no anel. As fichas são enviadas à UE tão logo esta esteja pronta.

A organização desta unidade baseia-se no mecanismo “first-in-first-out” (FIFO), mas outros mecanismos são possíveis. Em estudos realizados sobre a redução do uso de memória na MDFM, outros mecanismos de operação foram propostos como pilha (citado em [Rugg 87]) e mesmo uma fila inteligente [Sarg 83].

Na implementação de Manchester da UA, esta pode conter até 32K fichas com largura de 96 bits.

2.2.1.3 Unidade de Emparelhamento

Esta unidade é a mais crítica de todo sistema, devido à necessidade de um grande espaço de armazenamento endereçado associativamente por 54 bits de rótulo e destino. Pelo alto custo da memória puramente associativa o seu uso é proibitivo para a construção de uma memória de tamanho significativo. Baseado neste fato, a unidade de emparelhamento foi construída utilizando uma técnica de *hashing* por hardware que simula uma memória associativa [SiWa 83].

A memória pseudo-associativa compreende uma tabela de *hash* paralela implementada com bancos paralelos de memória RAM, um gerador de chave de *hash* e uma unidade de

controle [GoId 77].

O protótipo de Manchester [GuKW 85] foi construído com 16 bancos de memória de 64K posições, dando a esta unidade uma capacidade de 1M fichas. Cada posição possui uma ficha incluindo endereço destino, rótulo e um bit extra para indicar se a posição está vazia. Cada banco possui um comparador de 54 bits da chave de emparelhamento e uma interface de controle. A unidade possui também um mecanismo para tratamento de *overflow*. Um esquema da unidade de emparelhamento pode ser visto na figura 2.3.

Figura 2.3: Esquema da Unidade de Emparelhamento da MDFM

Os 54 bits da chave de emparelhamento de uma ficha que chega nesta unidade servem para que o gerador de chave de *hash* produza um endereço de *hash* com 16 bits que é usado para dar acesso aos 16 bancos de memória em paralelo. Cada endereço de *hash* possui um indicador de ocorrência de *overflow* que é alterado quando é ultrapassada a capacidade de armazenamento deste endereço.

Cada banco compara a chave de emparelhamento da ficha armazenada com a da ficha que chega. Se existir o emparelhamento o campo de dado da ficha armazenada é colocado no *buffer* de saída junto com a ficha que chegou. O pacote resultante com 133 bits passa então à unidade de instrução. Contudo, se não ocorrer o emparelhamento, é necessário considerar a existência de fichas na unidade de *overflow*. A verificação é feita através de um indicador de *overflow*. Se não houver, a ficha é armazenada na primeira posição livre endereçada pela chave de *hash*, e caso não exista posição vazia, é armazenada na área de *overflow* e o indicador de *overflow* é alterado para notificar as fichas subseqüentes deste fato. Caso existam fichas na área de *overflow*, a busca da ficha parceira procede nesta área. Se for encontrada então, o par é colocado na saída da unidade e a posição é limpa

e se for a única ficha dentro da unidade de *overflow*, o indicador é modificado. Caso a ficha parceira não esteja também na área de *overflow* e, se durante a busca na memória principal encontrou-se uma posição livre a ficha é armazenada na memória principal.

Fichas que não necessitam de fichas parceiras são enviadas diretamente a unidade de instrução.

O tempo gasto para o emparelhamento de instruções binárias é, em média, 5 vezes maior que para instruções unárias. Assim, o tempo médio gasto para o emparelhamento de fichas de um programa depende da razão de instruções unárias deste programa, sendo que foi verificado que programas típicos possuem entre 50% a 70% de instruções unárias [GuKW 85].

Alguns problemas desta unidade podem ser ressaltados:

- por sua complexidade a implementação em hardware é cara;
- a eficiência do esquema de *hashing* degrada rapidamente quando é utilizada mais que 25% da capacidade máxima de armazenamento, problema este relacionado com a busca na unidade de *overflow* que é feita de modo seqüencial.

2.2.1.4 Unidade de Instrução

A unidade de instrução é composta por dois estágios. O primeiro contém uma tabela de segmentos de 64 entradas, que é endereçada pelos 6 bits mais significativos do campo de destino do pacote de fichas vindo da unidade de emparelhamento. A tabela de segmento fornece um endereço de base para o segmento e um limite para deslocamentos dentro deste.

No segundo estágio, usando os 12 bits restantes do campo de destino o *offset* é somado à base e tem-se acesso à memória de instrução.

Cada posição da memória de instrução possui 71 bits de largura (12 bits para código da operação, 18 bits para um destino, 32 bits para um literal ou para outro destino e 9 para controle) [WaGu 82]. Quando a busca da instrução correspondente é concluída o pacote executável formado por 166 bits vai para a unidade de processamento.

2.2.1.5 Unidade de Processamento

A unidade de processamento pode ser separada em duas partes: uma executa instruções globais como a geração de nomes de ativação únicos e a outra contém um arranjo de processadores microprogramados que executam instruções lógico-aritméticas.

Quando um pacote executável deve ser executado pela segunda parte, este é entregue a um distribuidor que envia o pacote ao primeiro processador livre. O resultado ou resultados da execução são coletados por um arbitrador que os envia ao anel.

2.2.1.6 Unidade de Armazenamento de Estruturas de Dados

Para que a MDFM pudesse suportar estruturas de dados armazenadas foi criada a unidade de armazenamento de estruturas (UAE) [Sarg 85] [SaKi 86]. O modelo de funcionamento desta unidade é baseado no armazenamento de “estruturas-I” [ArTh 81], que explora o paralelismo existente na produção e consumo de dados de uma estrutura. A implementação em hardware foi feita por Kawakami [KaGu 86]. O funcionamento dessa unidade será explicado na seção 3.6.1.2.

2.2.2 Máquina de Fluxo de Dados do MIT - MTTDA

A MTTDA (*MIT Tagged-Token Dataflow Architecture*) [ArNi 90] (figura 2.4) é uma máquina a fluxo de dados dinâmica por rotulação de fichas. A MTTDA assim como a MDFM é implementada como um *pipeline* na forma de anel. Cada instrução pode ter no máximo dois dados de entrada. As fichas de dados possuem rótulos (*tags*) que separam os vários contextos de código reentrante e são da forma:

$$\text{ficha} = \langle c.s_p, v \rangle$$

O campo $c.s_p$ forma o *tag*. s identifica a instrução destino, c determina a instanciação e p a porta de entrada do dado (esquerda ou direita).

O agrupamento de dados de entrada, que habilitam a execução de uma instrução, é feito através da comparação de todo o *tag*. E novamente, o tamanho do *tag* aparece como problema a ser resolvido pela arquitetura já que este, por um lado limita o paralelismo a ser explorado pela arquitetura, e por outro aumenta a complexidade de operação desta. Três técnicas são utilizadas visando encontrar uma solução ótima:

- **Reciclagem de Rótulo:** a arquitetura possui um número fixo de possíveis valores

Figura 2.4: Esquema da MTTDA

de contexto, que durante a execução de uma programa vão sendo utilizados pelas várias ativações do código. É denominado bloco de código o conjunto de instruções que devem ser executadas com um único valor de c . Inserindo, em cada bloco de código, instruções que determinam sua terminação, pode-se verificar a conclusão de execução de um bloco de código. Quando isso ocorre, o contexto é retornado no conjunto de contextos livres, podendo ser utilizado em uma nova ativação do código.

- **Iteração:** o campo de contexto é separado em dois sub-campos. c e i , onde i é utilizado para separar iterações de *loops*. O uso desse campo simplifica a atribuição de novo *tag* para um bloco de código, visto que o sub-campo i deve ser manipulado localmente. Quando um *loop* deve possuir um número de iterações que podem ser executadas em paralelo maior que a capacidade de imposta por i , é utilizado um mecanismo que limita em um número k (menor que o número máximo representado por i) as iterações que podem ser executadas ao mesmo tempo [CuAr 89].
- **Registradores de Contexto:** em princípio o campo s é o endereço global de uma instrução, contudo se o contexto c for utilizado como um identificador do bloco de código, s pode ser apenas um *offset* dentro do bloco. Para implementar essa idéia são utilizados os registradores de contexto, que são endereçados por c . Nestes registradores são armazenados os endereços base de cada bloco de código (CBR). O

endereço de uma instrução é obtido simplesmente por $\text{CBR} + s$.

A forma de execução de programas na MTTDA é muito similar a MDFM:

- Fichas de dados são agrupadas na Unidade Espera-Agrupamento (*Waiting Matching*), cuja funcionalidade é a mesma da unidade de emparelhamento da MDFM;
- Os valores dos dados agrupados são enviados, junto com o *tag*, para a Unidade de Busca de Instrução (*Instruction Fetch*), onde o contexto é utilizado para endereçar um registrador de contexto. No registrador é obtido o endereço base do bloco de código (CBR), e pela soma do CBR com o *offset* s é obtido o endereço da memória de instrução onde estão armazenadas as seguintes informações: o *opcode*, uma constante ou um *offset* e no máximo dois destinos (s_1 e s_2). Existe também associado com o registrador de contexto a memória de constantes, onde são armazenadas constantes (literais) que devem ser utilizadas pelo bloco de código. Quando uma instrução especifica um *offset*, esse é utilizado para que seja buscado uma constante dentro da memória de constantes.
- A ULA processa os valores de entrada da instrução e a constante (se existir). Ao mesmo tempo são gerados os *tags* a serem utilizados pelos resultados da execução.
- Se forem produzidas mais fichas do que a Unidade de Espera-Agrupamento pode processar, o excesso é armazenado na Fila de Fichas (*User Queue*).

2.2.3 Considerações sobre as Arquiteturas de Primeira Geração

Através dos resultados obtidos com o protótipo da MDFM, pelo simulador da MTTDA e pesquisas com outras arquiteturas a fluxo de dados foi verificada a existência de comportamentos comuns a essas arquiteturas. Dessa forma, é possível generalizar algumas características relacionadas às máquinas a fluxo de dados de primeira geração, como:

- A granularidade explorada pelas arquiteturas a fluxo de dados parece ser mais fina que deveria, gerando uma alta taxa de comunicação entre as tarefas (instruções).
- A operação de agrupamento de fichas parceiras é muito complexa e demorada em função do elevado número de informações a serem comparadas nessa operação. Essa

característica decorre principalmente da existência de redundâncias nos campos de rótulo e destino (chave de emparelhamento) [Magn 92].

- O modelo a fluxo de dados expõe mais paralelismo que as arquiteturas propostas podem explorar. A existência de “gargalos” nas arquiteturas exigiu a criação de mecanismos de controle de geração de atividades paralelas, visando evitar a degradação de desempenho quando a aplicação possui grau de paralelismo maior que a capacidade de processamento da arquitetura [CuAr 89] [Rugg 87].
- A implementação das arquiteturas na forma de um *pipeline* circular impõe um tempo de latência elevado para que uma ficha de dado, gerada pelo processador, ative, realmente, sua instrução destino, mesmo quando esta instrução necessita apenas desta ficha de dado. Esse problema degrada muito a eficiência da arquitetura quando o paralelismo apresentado pelo programa é baixo.
- O processamento de estruturas de dados é bastante ineficiente, pois além de impor um elevado tráfego de fichas no anel, expõe duramente o problema do tempo de latência do anel.
- A tarefa de gerar programas para máquinas a fluxo de dados é fácil, pois não exige do programador conhecimento sobre a arquitetura, uma vez que o compilador é responsável pela geração do grafo de fluxo de dados. Linguagens funcionais, como SISAL (MDFM) [MSAG 84] e ID (MTTDA) [Nikh 90], foram utilizadas para programação dessas máquinas por possuírem características desejáveis para melhor utilização do modelo como [Glau 82]: localidade de efeito, ausência de efeitos colaterais e construções para explicitar iterações paralelas. Por outro lado, a utilização de linguagens funcionais tem acarretado um desinteresse por parte da maioria dos usuários, por estes estarem mais familiarizados com as linguagens imperativas como FORTRAN, C, etc.

2.3 A Segunda Geração de Máquinas a Fluxo de Dados

As lições aprendidas com as arquiteturas de primeira geração foram muito importantes para que surgissem uma nova série de arquiteturas baseadas no modelo a fluxo de dados, ditas hoje de segunda geração. Ficou provado que embora o modelo apresente alto grau de paralelismo, as arquiteturas que foram implementadas utilizando o modelo não conseguiram explorar plenamente esse paralelismo, principalmente devido às altas taxas de comunicação e sincronização requeridas.

As arquiteturas de segunda geração, de uma forma geral, aceitam o fato de que é preciso impor um maior controle sobre a execução das instruções. Assim, foram analisados especificamente, alguns pontos críticos que foram revelados pelas arquiteturas de primeira geração, como por exemplo, a execução de trechos de código estritamente seqüenciais (*threads*) e operações sobre estruturas de dados regulares. Esses dois tipos de operações apresentam grau de paralelismo muito baixo, demonstrando que as arquiteturas a fluxo de dados puras não conseguem efetuar tais operações de forma tão eficiente quanto as arquiteturas von Neumann.

A partir dessa análise concluiu-se que conservando as boas características do modelo a fluxo de dados e unindo a esse tipo de processamento características positivas das arquiteturas convencionais poderia ser obtido um melhor desempenho. Sendo assim, foram feitas várias propostas de arquiteturas híbridas fluxo de dados/von Neumann.

As arquiteturas paralelas baseadas no modelo von Neumann que exploram paralelismo existente entre vários trechos seqüenciais de programa, utilizam a execução paralela de vários *threads* em um mesmo processador para mascarar instruções de alta latência, como as que buscam dados em processadores remotos. O Denelcor HEP [Smit 81] foi a primeira máquina comercial a utilizar a abordagem *multithreaded*. Contudo, as arquiteturas von Neumann *multithreaded* [Agar 90] [Alve 90] [HaFu 88] apresentam ineficiências quanto a sincronização de granularidade fina imposta por essa abordagem.

Por outro lado, a utilização da abordagem *multithreaded* pelas arquiteturas a fluxo de dados não apresenta o mesmo problema relacionado com a sincronização de dados constatados nas arquiteturas *multithreaded* convencionais. Além disso, essa abordagem permite que uma tarefa deixe de ser somente uma instrução, para tornar-se um conjunto

de instruções que devem ser executadas seqüencialmente. Esse aumento de granularidade pode proporcionar a redução das taxas de comunicação e sincronização exigidas pelo modelo a fluxo de dados.

Mas, uma questão precisa ser analisada e decidida: como destacar os trechos estritamente seqüenciais dos programas? Culler [Cull 91], antigo integrante do projeto Monsoon, argumenta que deve existir um desperdício muito grande de hardware, se o compilador não for utilizado para esse fim. Dessa forma, vários projetos de arquiteturas a fluxo de dados adicionaram recursos no sistema de compilação para torná-los capazes de identificar *threads*. O sistema de compilação criado por Culler e outros pesquisadores da Universidade da Califórnia, tem a capacidade de identificar *threads* e também expor um escalonamento para execução destes, visando minimizar o *overhead* imposto pelo gerenciamento de registradores quando é feito apenas o escalonamento dinâmico de execução. Para avaliar essas propostas foi formulada uma máquina abstrata, denominada TAM (*Threaded Abstract Machine*) [Cull 93] onde os recursos de processamento e escalonamento de *threads* consistem em instruções de máquina explícitas, bem como o gerenciamento de memória. A forma como os programas são particionados em TAM será apresentado no capítulo 5, pois suas idéias básicas serão utilizadas para geração de código para a arquitetura proposta.

Uma abordagem diferente é adotada pela arquitetura WOLF [CavG 95], onde tanto a identificação quanto o escalonamento de *threads* são realizados dinamicamente. Nesta arquitetura os registradores internos aos processadores são utilizados para passar dados de uma instrução para outra, simplesmente, enquanto for identificado que a próxima instrução (instrução destino) necessita de apenas um dado de entrada para ser executada, caracterizando a existência de um *thread*.

Outra característica que colaborou para obtenção do baixo desempenho apresentado pelas arquiteturas a fluxo de dados de primeira geração, refere-se ao modelo utilizado para o armazenamento de dados que devem ser sincronizados para habilitar uma instrução diádica. Para evitar o uso de esquema de memórias pseudo-associativas, que envolvem complexos sistemas de *hashing*, partiu-se para uma nova abordagem do conceito dos rótulos utilizados pelas arquiteturas a fluxo de dados dinâmicas (chaves de emparelhamento). Através dessa nova abordagem um rótulo passa a possuir um significado físico, ou seja, uma posição de memória. Esse novo esquema tornou-se conhecido como ETS

(*Explicit Token Store*) ou armazenamento explícito de fichas [PaCu 91], seu funcionamento será explicado na seção 2.3.1.

Em relação ao problema da baixa eficiência do processamento de operações envolvendo estruturas de dados regulares, alguns estudos e propostas têm sido realizados. A arquitetura WOLF apresenta como proposta a adaptação de um processador vetorial em uma arquitetura a fluxo de dados pura. A utilização desse processador depende da construção de um compilador que consiga destacar instruções vetoriais a partir do grafo de fluxo de dados. Essa tarefa é exatamente o que é feito pelos compiladores que “vetorizam” o código a ser utilizado por super-computadores [PaWo 86] como CRAY X-MP, NEC-SX, etc. Contudo, como a arquitetura utiliza unidades para o armazenamento de estruturas de dados baseadas no modelo de estruturas-I algumas alterações devem ser analisadas [MiNB 95] para ajustar os modelos utilizados, visando obter um bom desempenho das operações de leitura e armazenamento de vetores.

A seguir serão descritas algumas arquiteturas de segunda geração que conseguem, de uma forma geral, representar as principais tendências discutidas nessa seção.

2.3.1 Projeto Monsoon

O projeto Monsoon [PaCu 91] é uma evolução da MTTDA. Nessa arquitetura a operação de agrupamento de fichas parceiras utiliza conceito de armazenamento explícito (endereço virtual) em quadros de memória, além disso apresenta um mecanismo especial para a execução de *threads*.

O mecanismo de armazenamento explícito de memória (ETS) baseia-se na associação do *tag* (contexto e instrução destino) com posições físicas nos quadros de memória locais aos elementos de processamento (figura 2.5).

Para implementação do mecanismo ETS em uma arquitetura com vários elementos de processamento (PE), um *tag* passa a representar a seguinte associação:

$$c.s = PE : (FP.IP)$$

onde:

FP é um ponteiro para o quadro de ativação (*frame pointer*).

IP é um ponteiro para instrução destino (*instruction pointer*).

PE.FP é o endereço global da base do quadro de ativação no qual uma ficha de dado

deve procurar sua parceira.

$PE.IP$ é o endereço global da instrução destino.

Figura 2.5: Exemplo de Compilação de uma Expressão para e mecanismo ETS

A figura 2.6 apresenta o esquema de um elemento de processamento da arquitetura Monsoon, que é implementado na forma de um *pipeline* composto de 8 estágios. No primeiro estágio é obtida a instrução (apontada por IP). A instrução obtida especifica um código de operação, um deslocamento dentro do quadro de ativação e uma ou duas instruções destino. O valor do deslocamento é utilizado para a obtenção do endereço absoluto da posição na qual os dados, referentes à instrução apontada por IP , devem ser sincronizados. Cada posição do quadro de ativação possui 1 bit de presença, que consegue através da transição de estado, efetuar a sincronização dos dados de entrada de instruções, de maneira bastante eficiente. Além de designar a presença ou ausência de dados de entrada para determinada instrução, os bits de presença podem especificar outros estados, os quais podem indicar que parte do valor contido em uma posição do quadro de ativação deve ser ignorada, lida, escrita ou ser trocada pelo valor da ficha. Quando é determinado que não existe dado armazenado na posição do quadro (bits de presença indicando vazio) e a instrução necessita de dois dados de entrada é gerada uma “bolha” no *pipeline* que será transferida a todos os estágios seguintes a essa fase, estágios onde, de fato, são executadas as operações do código e criadas as fichas de resultados

dessas instruções.

Cada ficha de resultado gerada deve ser armazenada na fila de fichas do usuário ou do sistema ou, em alguns casos, ser enviada diretamente para a entrada do *pipeline*. Essa última possibilidade permite que, quando um trecho de código é estritamente seqüencial (*thread*), esta ficha possa circular no anel sem a necessidade de entrar nas filas, podendo inclusive ignorar os bits de presença do quadro de ativação. Cada *thread* dispõe de três registradores para rápida comunicação de dados entre instruções. Quando um *thread* tem sua execução concluída suas fichas de resultados são inseridas em uma das filas, de onde é retirada uma ficha de dado. Por esse tratamento de *threads*, Monsoon pode ser considerada uma arquitetura *multithreaded*, sendo que os *threads* devem ser identificados em fase de compilação.

Figura 2.6: Esquema da Arquitetura Monsoon

A arquitetura Monsoon é formada por vários elementos de processamento (PE) e memórias de estruturas-I (MM) conectadas por uma rede de interconexão *butterfly*.

Por ser uma máquina com fortes características do modelo a fluxo de dados a arquitetura Monsoon apresenta um ótimo desempenho nas seguintes pontos:

- a interface entre a rede de interconexão e os processadores permite o envio de mensagens em uma única instrução;
- a criação de atividades paralelas é realizada de maneira simples e rápida;
- a operação de sincronização tornou-se eficiente devido à utilização do mecanismo ETS;
- o chaveamento de contextos é bastante rápido, podendo existir até oito *threads* em execução no *pipeline* num determinado instante;
- A memória global compartilhada tem acesso implementado através de instruções *split-phase*, fato que consegue mascarar a latência do acesso à memória remota.

Porém, também existem pontos negativos que foram identificados, por exemplo, o baixo desempenho obtido quando apenas um *thread* está sendo executado no *pipeline* e a excessiva quantidade de sincronização que ainda é observada pela granularidade fina explorada pela arquitetura.

2.3.2 *T

O sistema *T [NiPA 92] consiste em uma evolução da arquitetura a fluxo de dados dinâmica Monsoon (MIT) e da proposta feita por Nikhil e Arvind de uma arquitetura RISC paralela (P-RISC) [NiAr 89]. A arquitetura P-RISC utiliza um conjunto de instruções simples com operações lógico-aritméticas e operações *fork/join* para efetuar sincronização de tarefas, eliminando a necessidade de bits de presença. Os dados deixam de ser transportados em fichas, sendo que estas passam a ser formadas apenas por FP e IP. Para o armazenamento dos dados são utilizadas memórias globais e quadros de ativação.

O sistema *T pode ser visto como um processador RISC com co-processadores para tratamento de várias mensagens, e que suporta o modelo *multithreaded*. A arquitetura *T possui vários nós de processamento, sendo que a comunicação entre eles é feita através de mensagens. Uma mensagem contém um rótulo (*tag*) e um ou mais valores. Toda comunicação entre nós é executada como uma operação *split-phase*, o que permite a atenuação do *overhead* imposto pela latência de comunicação. A figura 2.7 mostra um nó

*T que é composto por três processadores assíncronos separados: o processador de dados (**dP**), o processador de sincronização (**sP**) e o processador de memória remota (**RMem**).

Figura 2.7: Elemento de Processamento do *T

O processador de sincronização (**sP**) pode ser comparado em termos de sua funcionalidade à unidade de agrupamento de uma arquitetura a fluxo de dados dinâmico. O **sP** também assemelha-se com um microprocessador, pois tem um contador de programa (**sIP**), registradores de propósito geral, e pode fazer operações de *load/store* na memória local. A operação básica de **sP** é repetidamente, carregar a mensagem da fila de mensagens (*message queue*) e executá-la até sua conclusão. O **sP** deve executar quatro novas instruções: **START**, **POST**, **NEXT** e **JOIN**. A instrução **START** inicia a execução de um novo *thread* em um contexto diferente, que pode ser local ou remoto ao nó; quando for externo, **sP** controla uma mensagem com os argumentos necessários. A instrução **POST** insere uma continuação na fila de continuação do **dP**, ou seja, um ponteiro de instrução e quadro de ativação. A instrução **NEXT** conclui o *thread* em execução e retira uma nova continuação da fila de continuações. Finalmente, a instrução **JOIN** incrementa um valor na memória local e dependendo do valor resultante aponta para um dado ponteiro de instrução.

O processador de dados (**dP**) é responsável pela execução, de fato, dos programas. O **dP** é um processador convencional RISC, com repertório de instruções registrador-a-

registrador e habilidade para manipular memória local utilizando instruções convencionais de *load/store*. Possui um contador de programa dIP (*Data processor Instruction Pointer*). Um dos registradores, chamado dFP, contém o FP corrente para o *thread* que está sendo executado pelo dP. Usando um processador convencional RISC, têm-se uma execução eficiente de *threads*. Os pontos iniciais dos *threads* são obtidos através da fila de continuações (*continuation queue*). Não existe o conceito de *threads* suspensos, pois são executados até serem concluídos. Se nenhum *thread* estiver disponível quando um *thread* é concluído o dP deve ficar esperando.

Quando, por exemplo, é feita uma chamada de função dentro de um *thread*, o dP envia uma mensagem à rede através da instrução START, mas não bloqueia sua execução. Assim, seria equivalente a uma operação FORK que possibilita a execução paralela de funções. Um *thread* é concluído pela instrução NEXT, que habilita dP a pegar uma nova continuação. É importante ressaltar que os registradores não são armazenados implicitamente quando um novo *thread* é iniciado.

O processador de memória remota (RMem) trata requisições para leitura e armazenamento remotos, com comportamento muito próximo de um controlador de “estruturas-I” na arquitetura Monsoon. A leitura ou escrita na memória local é feita sem interferir no processamento do dP.

Através da implementação de $*T$ com processadores Motorola 88110MP, que foram modificados para fornecerem interface com a rede de interconexão e suporte para escalonamento de *threads*, foram feitas as seguintes verificações:

- O desempenho da rede de interconexão é pior que o comparado com as arquiteturas a fluxo de dados Monsoon e EM-4, contudo consegue ser melhor que a CM-5 [Cull 93].
- A seqüencialização de instruções em *threads* é feita de forma mais eficiente em $*T$, exigindo a operação de criação de atividade paralela (*fork*) com menor frequência.
- Não existe suporte em hardware em $*T$ para sincronização, sendo que o sistema utiliza para tanto a instrução POST, ou quando existe uma condição a ser satisfeita a instrução CPOST. A operação de sincronização é mais eficiente no sistema Monsoon, contudo é menos solicitada em $*T$ já que esta é apenas necessária para *threads*.

- Como a implementação de $*T$ considera que o chaveamento de contextos deve ser feito a nível de software, o compilador deve gerar código para guardar valores, armazenados nos registradores, na memória local quando um *thread* é concluído e armazenar valores nos registradores para iniciar a execução de um *thread*. Em função dessa característica, deve-se avaliar a seguinte situação: se os *threads* são pequenos, as operações de armazenamento e envio de valores dos registradores serão efetuadas mais rapidamente, contudo isso faria com que o número de *threads* aumentasse, ou seja, o chaveamento de contextos seria feito com maior frequência.
- A eficiência das operações que envolvem acesso à memória global é maior na arquitetura Monsoon, uma vez que esta as implementa como uma operação *split-phase* conseguindo mascarar boa parte do tempo de latência da memória.

Devido às características apresentadas por $*T$, principalmente a questão da latência que é envolvida em operações com a memória global, está sendo proposta uma nova versão de $*T$, denominada $*T$ -NG (*next generation*) [Chio 95] [AnAC 92]. Essa nova proposta baseia-se fortemente na arquitetura $*T$, mas adiciona mecanismos de coerência de *caches* para a memória global compartilhada.

2.3.3 EM-4

A arquitetura EM-4 é uma arquitetura híbrida *dataflow/von Neumann* que pode executar código sequencial (*threads*) [SKSYK 92] baseado em registradores e também utilizar o mecanismo de fluxo de dados para sincronização e comunicação mais barata entre *threads*.

O sistema EM-4 baseia-se em um modelo de fluxo de dados diferente do usualmente utilizado pelas primeiras arquiteturas, denominado modelo de arcos fortemente conectados. Nesse modelo, os arcos são classificados em normais e fortemente conectados. O conjunto de nós ligados por arcos fortemente conectados são designados por Bloco Fortemente Conectado (SCB). Um SCB é habilitado por fichas através de arcos normais, e é executado como se fosse um único nó básico em um grafo de fluxo de dados, sendo implementado como uma seqüência ininterrupta de instruções.

O SCB é executado em um único processador (PE), baseado em arquitetura RISC, com a utilização de registradores. A regra de disparo de nós SCB é determinada estaticamente

e sua execução é controlada por um *program counter*. Tal característica permite que as fichas de um nó SCB sejam armazenadas em registradores e não transferidas como pacotes.

Para reduzir o pesado *overhead* da operação de emparelhamento são utilizados dois mecanismos de sincronização. O primeiro está contido na ordenação interna de execução dos nós SCB; o segundo é chamado de esquema de emparelhamento direto. Esse esquema dispensa o uso de mecanismos associativos, pois é reservada uma área de emparelhamento para cada ativação de funções. Assim, a posição de emparelhamento de uma ficha de dado corresponde ao endereço absoluto.

Existem dois tipos de *pipeline* no sistema EM-4, um circular com emparelhamento direto e outro baseado em registradores, que executa os nós SCB, chamado de “controle avançado”. O *pipeline* circular é multi-contexto, mas o segundo *pipeline* é de contexto único, por ser mais vantajosa sua utilização para a execução de trechos de baixo paralelismo encapsulados nos nós SCB.

Figura 2.8: Esquema da Arquitetura EM-4

A arquitetura EM-4 é formada por vários processadores EMC-R, que adota filosofia RISC. A figura 2.8 mostra a arquitetura do EMC-R que consiste em uma unidade de

chave, que conecta o processador com a rede de interconexão ômega, uma unidade de *buffer* de entrada, uma unidade de busca e agrupamento, uma unidade de execução, uma unidade de controle de memória e um controlador de manutenção.

A unidade de chave possui facilidade de balanceamento de carga dinâmica a nível de função. No EM-4, pacotes especiais que monitoram a carga dos processadores percorrem a rede. A unidade de chave reescreve esses pacotes dentro de um período de tempo requerido para a transferência de um pacote normal, assim, é realizado o balanceamento de carga com *overhead* mínimo.

A unidade de *buffer* de entrada é um *buffer* implementado com uma memória FIFO de 32 palavras. Quando o *buffer* está cheio uma parte de memória fora do chip é usada como *buffer* secundário. A unidade de busca e agrupamento realiza a operação de emparelhamento de dados e busca de instrução. A unidade de controle de memória arbitra sobre as requisições de acesso à memória feitas pelas unidades de busca e agrupamento e execução, controlando o envio de dados entre a área de memória fora do chip e o EMC-R.

A unidade de manutenção do EM-4 é utilizada para iniciação do sistema, “debugar” o hardware e software e monitorar a execução do programa. As operações de manutenção precisam ser executadas em paralelo. Para iniciação do EM-4, a unidade de manutenção precisa rapidamente limpar e iniciar os registradores de controle e *status*. Para realizar o *debug* do hardware devem ser lidos todos os registradores de *status* do EMC-R, incluindo informações sobre os circuitos mais internos que o sistema da arquitetura não tem acesso. Já para “debugar” um programa, deve-se trabalhar independentemente do sistema, já que os *crashes* de programas em um sistema a fluxo de dados não podem ser diagnosticados. Para a monitoração dinâmica do comportamento da arquitetura, deve-se coletar informações sem afetar o sistema da arquitetura, assim as operações de manutenção precisam ser executadas em paralelo.

O chip EMC-R foi fabricado em outubro de 1989. O pico de performance é 12.5 MIPS/chip, já que é utilizado um clock de 80ns e toda instrução necessita de apenas 1 ciclo de máquina.

O protótipo EM-4 contém 80 chips EMC-R e pode ser visto na figura 2.8. Ele foi implementado em abril de 1990, e é formado por 16 placas que agrupam 5 processadores cada uma, uma rede de interconexão, uma chave e um processador de interface de pacote

e um computador *host*.

2.4 Resumo

Para que a arquitetura proposta nessa tese pudesse levar em consideração os resultados obtidos pelas arquiteturas baseadas no modelo a fluxo de dados de primeira geração e das alterações relevantes realizadas pelas arquiteturas de segunda geração, essas arquiteturas foram amplamente estudadas e analisadas. Este capítulo, foi utilizado para apresentar parte desse estudo realizado, servindo como base para as propostas que serão expostas nos capítulos seguintes onde serão tratadas as especificações da nova arquitetura e da geração de código requerida por ela.

Capítulo 3

A Proposta de uma Nova Arquitetura a Fluxo de Dados

Considerando as informações sobre as arquiteturas a fluxo de dados que foram expostas no capítulo 2, foi elaborada uma proposta de arquitetura que tenta sanar os problemas apresentados pelas máquinas a fluxo de dados de primeira geração, tentando manter as qualidades apresentadas pelo modelo.

A nova arquitetura denominada Sistema de Escalonamento e Execução *Dataflow* (SEED) será apresentada neste capítulo. A primeira seção descreve o funcionamento geral da arquitetura; nas seções posteriores, cada unidade da arquitetura é apresentada mais detalhadamente.

3.1 Descrição Geral da Arquitetura SEED

A arquitetura SEED é composta por várias unidades implementadas como um *pipeline* na forma de um anel (figura 3.1). Ela utiliza o modelo a fluxo de dados para habilitar a execução de blocos de instruções, e para executar as instruções desses blocos dentro de cada Unidade Funcional (UF).

Por ser uma arquitetura fortemente baseada nas arquiteturas a fluxo de dados dinâmicas por rotulação de fichas, o seu conjunto de instruções deve ser, da mesma forma, composto por instruções que considerem as características próprias das arquiteturas a fluxo de dados, visando aproveitar todo o potencial da arquitetura SEED. Assim, como será apresentado no capítulo 4, foi definido que o conjunto de instruções utilizado pela arquitetura SEED será idêntico ao utilizado pela Máquina de Fluxo de Dados de Manchester

(MDFM) [Kirk 87].

Para evitar os problemas decorrentes das altas taxas de sincronização e comunicação exigidas pelo modelo a fluxo de dados, deseja-se que a arquitetura SEED seja capaz de explorar localidade de efeito dos dados em trechos restritos de programa. Para tanto, será adotada a seguinte abordagem:

Figura 3.1: Esquema Geral da Arquitetura SEED

- Os programas devem ser particionados em blocos de instruções. Um bloco deve ser formado por instruções, cuja execução deve ser mais eficiente se as instruções forem processadas em um único processador. A eficiência decorre da existência de localidade de efeito dos dados gerados pelas instruções do bloco. Assim, os dados produzidos por instruções internas a um bloco podem ser utilizados por outras instruções do mesmo bloco, em um espaço de tempo muito curto. Um recurso equivalente é utilizado por processadores baseados no modelo von Neumann, pois

quando é verificado que os dados gerados por uma instrução devem ser consumidos em seguida por uma outra instrução, esses dados são armazenados em registradores internos ao processador, para que a execução desse trecho do programa seja mais eficiente.

- Pode-se explorar localidade em operações sobre estruturas de dados regulares se for utilizado um processador vetorial convencional. O processador vetorial deve obter um melhor desempenho se comparado com as arquiteturas a fluxo de dados puro. Além dos processadores vetoriais serem altamente especializados para esse tipo de processamento, sua utilização proporcionaria a redução do tráfego de fichas no anel. Como consequência, um número menor de sincronizações devem ser necessárias em relação a uma arquitetura a fluxo de dados pura.

Para implementar o primeiro tipo de exploração de localidade é exigido que as instruções sejam agrupadas em blocos de código, para só então, serem executadas dentro das unidades funcionais. A metodologia utilizada para separação de instruções em blocos de código será descrita no capítulo 5.

Depois de serem particionados pelo sistema de compilação, os programas são armazenados na Unidade de Instrução Global (ULGlobal). À medida em que os blocos de instrução vão sendo escalonados para execução, através do modelo a fluxo de dados, a ULGlobal envia todas as instruções pertencentes ao bloco para a Unidade Funcional escolhida para executá-lo. O armazenamento de código na ULGlobal é totalmente independente do funcionamento do resto da arquitetura. Contudo, deve-se garantir que todo o código do programa esteja presente na ULGlobal antes que a execução do programa seja liberada pelo *host*. Para evitar que o tráfego de informações no barramento de dados aumente ainda mais com a necessidade de transmitir instruções da ULGlobal para as unidades funcionais, optou-se em utilizar barramentos distintos para dados e instruções.

Um programa tem sua execução iniciada, quando o *host* envia os dados de entrada do programa para a Chave 2. Os dados destinados a um bloco são sincronizados em uma das Unidades de Agrupamento (UA). Quando um bloco tem todos os seus dados de entrada prontos, a execução do bloco é atribuída a uma UF através da Chave 1. Nesse momento, as instruções pertencentes a esse bloco são requisitadas à ULGlobal. O bloco terá sua execução habilitada apenas quando todas as suas instruções estiverem presentes

na UF. Para diminuir os efeitos da latência imposta pela busca dos blocos de código na UI.Global, cada UF poderá executar vários blocos concorrentemente. Os resultados gerados pela execução dos blocos pelas unidades funcionais e que se destinam a outros blocos devem ser enviados para a Chave 2.

Os dados são transmitidos de um bloco de código a outro na forma de fichas de dados. Cada ficha de dado possui o seguinte formato:

$$\langle n_ent \rangle \langle porta_ent \rangle \langle nome_ativação \rangle \langle índice \rangle \langle destino \rangle \langle dado \rangle$$

$$\langle (2\ bits) \rangle \langle (2\ bits) \rangle \langle (10\ bits) \rangle \langle (5\ bits) \rangle \langle (12\ bits) \rangle \langle (64\ bits) \rangle$$

Para simplificar o hardware utilizado para sincronização de dados pelas Unidades de Agrupamento, o número de dados de entrada de cada bloco será limitado em quatro. Portanto, o campo *n_ent*, que informa o número de dados de entrada do bloco de código, pode variar de 0, indicando que o bloco possui apenas 1 dado de entrada, até 3, indicando que o bloco possui 4 dados de entrada. A porta de entrada de cada dado é definida pelo campo *porta_ent*, que também necessita de apenas 2 bits para representar todas as portas (figura 3.2). Os campos que formam o rótulo da ficha de dado serão os mesmos utilizados pela MDFM: índice e nome de ativação. Os tamanhos definidos para os campos de destino e rótulo serão justificados na próxima seção.

Figura 3.2: Definição das Portas de Entrada de um Bloco de Código

As unidades funcionais utilizam o modelo a fluxo de dados para executar as instruções pertencentes a um bloco de código. Uma UF também é implementada na forma de anel, e possui a mesma estrutura geral de funcionamento utilizada pelo anel mais externo (figura 3.7). A descrição funcional da UF será apresentada na seção 3.7.

Para armazenar estruturas de dados como vetores e matrizes, existe a Memória de Estruturas (ME). O acesso aos elementos de uma estrutura de dados é realizada através de instruções próprias da ME, que são enviadas pelas unidades funcionais através de um barramento utilizado apenas por instruções que envolvem estruturas de dados. Visando

melhorar o desempenho de execução de instruções envolvendo estruturas de dados regulares, como por exemplo soma de vetores, a arquitetura utiliza-se de uma Unidade de Processamento Vetorial (UPV). A Unidade de Estruturas de Dados é composta pela ME e UPV e deve controlar a interação entre ambas as sub-unidades.

Uma descrição mais detalhada de cada unidade da arquitetura será exposta nas seções seguintes.

3.2 Unidade de Agrupamento

Como foi descrito no capítulo 2, o maior problema apresentado pelas arquiteturas baseadas no modelo a fluxo de dados é a alta taxa de sincronização imposta por esse modelo. Em função dessa característica, a unidade que realiza a sincronização dos dados de entrada das instruções tornou-se um “gargalo” dessas arquiteturas. A fim de evitar esse problema na arquitetura SEED, foram adotados os seguintes critérios:

- Através da separação do programa em blocos de código, deve-se aumentar a granularidade das tarefas a serem sincronizadas na Unidade de Agrupamento. Uma tarefa passa a ser composta por algumas instruções do código. Esse aumento de granularidade permite que a taxa de sincronização necessária seja reduzida em relação ao modelo a fluxo de dados puro.
- A concentração de toda a operação de escalonamento de blocos de código em apenas uma unidade de agrupamento, limita a capacidade de transmissão de resultados gerados pela execução dos blocos de código para outros blocos. Assim, a arquitetura SEED apresenta a proposta de utilizar mais de uma unidade de agrupamento. Essas unidades poderiam operar paralelamente, possibilitando um melhor desempenho da operação de sincronização. A distribuição das fichas de dados geradas pelas unidades funcionais para as várias unidades de agrupamento fica a cargo da Chave 2.
- Considerando que as fichas de dados, que serão sincronizadas nessa unidade, devem conter informações muito similares às aquelas definidas para as fichas de dados da máquina de Manchester (MDFM), pode-se aplicar otimizações que foram propostas para eliminar redundâncias da chave de emparelhamento da MDFM. Estas

otimizações permitem que a sincronização de dados seja efetuada através de um número reduzido de informações, que podem ser representadas em 27 bits, ao invés dos 54 bits especificados pela MDFM original [Magn 92]. Dessa forma, a unidade de agrupamento pode ser implementada por uma memória que pode ser endereçada diretamente pela chave de emparelhamento. A capacidade total de armazenamento das Unidades de Agrupamento fica limitada pelo número de bits que formam o endereço, ou seja, 128M posições.

Figura 3.3: Esquema de uma Unidade de Agrupamento

A partir dessas condições, a funcionalidade de cada UA pode ser esquematizada como mostra a figura 3.3. Sendo que cada posição da Memória de Dados é formada pelos seguintes campos (figura 3.4):

- **contador:** Esse campo deve armazenar o número de fichas de dados que ainda devem chegar na UA destinadas ao mesmo bloco de instrução e pertencentes à mesma instanciamento do código reentrante. Em princípio, todos os contadores associados a cada posição recebem o valor zero, indicando que não existe dado armazenado naquela posição. Com a chegada da primeira ficha de dado o contador tem seu valor alterado para o número de dados de entrada do bloco de código: esse número varia de 0 a 3 e portanto, exige apenas 2 bits;
- **porta:** indica a porta pela qual o dado deve entrar na instrução cabeça;

- **tipo_i**: Cada um dos dados de entrada de um bloco pode ser de tipo distinto dos demais dados. Assim, torna-se necessário armazenar junto a cada dado a especificação do seu tipo. Os tipos de dados definidos pela arquitetura são: ponto flutuante, inteiro, lógico (“*boolean*”), destino, nome de ativação e ponteiro para vetor;
- **dado_i**: valor do dado de entrada;

contador	porta ₁	tipo ₁	dado ₁	porta ₂	tipo ₂	dado ₂	porta ₃	tipo ₃	dado ₃
2 bits	2 bits	3 bits	64 bits	2 bits	4 bits	64 bits	2 bits	4 bits	64 bits

Figura 3.4: Informações Armazenadas em cada Posição da Memória Interna à UA

Quando uma ficha de dado entra em uma UA é verificado, pelo campo *n_ent*, se a ficha necessita ser agrupada com outra ficha de dado (ou fichas). Se *n_ent* for diferente de zero é necessário realizar sincronização para iniciar a execução do bloco. Os campos de rótulo e destino formam o endereço onde a ficha deve procurar suas fichas parceiras na Memória de Dados. Através de um contador é determinado se todas as fichas que se destinam a este bloco já estão prontas na UA. Em caso positivo, um pacote contendo todos os dados é enviado para a Chave 1; caso contrário, a ficha é armazenada e o contador é decrementado. Se um bloco necessita de apenas um dado de entrada, quando a ficha de dado que contém o dado chega na UA, esta passa diretamente para a Chave1.

Como os dados não são armazenados de forma ordenada na Memória de Dados, a ordenação deve ser feita na operação de construção de pacote, através da especificação da porta de entrada armazenada junto ao dado na memória. Essa função é realizada pelo Formador de Pacotes.

3.3 Chave 1

A Chave 1 tem como função distribuir trabalho às unidades funcionais. Ela conecta todas as saídas das Unidades de Agrupamento a todas as entradas das unidades funcionais. A atribuição de um bloco a uma determindada UF é realizada em função da ocupação.

Como cada bloco de código não pode conter reentrâncias e deve ter um número restrito de instruções, o tempo de execução de cada bloco não deve ser muito diferente dos demais.

Mesmo quando um bloco contém uma instrução do tipo “*proliferate*”, cujo tempo de execução é proporcional ao número de fichas que devem ser geradas, o tempo de execução da instrução é limitado em função da restrição do número de fichas geradas imposta pela limitação do tamanho da chave de emparelhamento [Magn 92]. Dessa forma, é possível adotar como medida de ocupação de uma UF o número de blocos em execução. Por exemplo, se num determinado instante existir apenas um bloco a ser enviado para uma UF, deve ser verificado qual o número de blocos em execução em cada UF no instante anterior ao escalonamento do bloco. A UF com menor número, ou menor ocupação deve executar esse bloco. Cada vez que a execução de um bloco é concluída ou um novo bloco é enviado a uma UF, o novo valor de ocupação deve ser atualizado na Chave 1 pela UF.

3.4 Chave 2

A Chave 2 pode receber fichas de dados das unidades funcionais, do *host* e da Unidade de Estruturas. Através de um bit de controle é determinado se essas fichas devem ser enviadas para o *host*, quando se trata de uma ficha de saída do programa, ou para alguma UA. Neste último caso, a Chave 2 deve ainda decidir para qual UA a ficha deve ser enviada. Essa decisão deve ser feita através de uma política simples para evitar a imposição de uma elevada latência para essa chave. Como uma primeira proposta, a determinação da UA será feita pelos bits menos significativos do campo de destino. Por exemplo, se existirem apenas 2 unidades de agrupamento, as fichas destinadas aos blocos identificados por números ímpares devem ser enviadas para uma UA, e as que vão para blocos identificados com números pares para a outra UA. Esse tipo de política pode levar a uma elevada ocupação de uma UA, quando uma instrução gera muitas fichas para um mesmo bloco de código com rótulos diferentes. Para os outros tipos de instruções, a distribuição de fichas deve ser mais ou menos aleatória, garantindo uma boa distribuição de fichas para cada UA.

3.5 Unidade de Instruções Global

Foi definido que as instruções dos programas a serem executados não serão replicadas pelas várias unidades funcionais que formam a arquitetura. Essa opção parece ser a

melhor indicada por não exigir uma quantidade excessivamente grande de memória em cada UF, permitindo a utilização de memórias muito menores e, conseqüentemente mais rápidas, para implementar a unidade de sincronização interna à UF (seção 3.7).

A UL_Global (figura 3.5) tem duas funções básicas e independentes:

- Armazenar instruções de programas a serem executados;
- Enviar para as unidades funcionais as instruções do bloco de código cuja execução já foi habilitada por uma UA.

Cada uma dessas operações será descrita a seguir.

Figura 3.5: Esquema Geral da Unidade de Instrução Global

3.5.1 Armazenamento de Instruções

Para que um programa seja executado, todo o seu código deve estar armazenado na UL_Global antes que o programa receba seus dados de entrada.

Quando a UL_Global recebe do *host* a instrução cabeça do primeiro bloco, um registrador denominado Contador de Instruções é zerado e a instrução cabeça é armazenada

na posição inicial da Memória Global de Instruções. O número do bloco é armazenado no Registrador de Endereço de Bloco. Quando chega a próxima instrução do bloco, o Contador de Instruções é incrementado, e posteriormente é concatenado com o Registrador de Endereço de Bloco. O resultado dessa concatenação é o endereço no qual a instrução deve ser armazenada.

Para cada instrução recebida é verificado, através de um bit de controle, se esta corresponde a uma instrução cabeça. Se for uma instrução cabeça, o valor contido no Contador de Instruções deve ser armazenado em uma memória, chamada de Memória de Número de Instruções. Essa memória armazena o número de instruções que formam cada bloco armazenado na UI_Global.

3.5.2 Requisição de Bloco de Código

Quando uma UF necessita do código de um determinado bloco esta, envia a UI_Global uma mensagem com o seguinte formato:

$$< destino > < rótulo_local_UF > < número_da_UF >$$

Através do campo de *destino* é identificada a posição inicial do bloco de código dentro da Memória Global de Instruções onde é obtida a primeira instrução do bloco, ou seja, a instrução cabeça. O *número_da_UF* é utilizado para que a resposta seja enviada para a UF correta. A informação *rótulo_local_UF* corresponde ao rótulo atribuído internamente a UF que, como será explicado posteriormente, é capaz de identificar totalmente o bloco dentro da UF. Dessa forma, esta mensagem garante o correto relacionamento entre o bloco de código recebido com a requisição de código feita.

Quando uma requisição é recebida pela UI_Global, o Contador de Instruções é zerado. Em seguida, é formado o endereço inicial do bloco através da concatenação do campo de *destino* da mensagem de requisição ao Contador de Instruções. O conteúdo dessa posição na Memória Global de Instruções corresponde à instrução cabeça do bloco requisitado pela UF.

Paralelamente à leitura da instrução cabeça, é lido na Memória de Número de Instruções, o número de instruções que formam o bloco. Este valor é armazenado no Registrador de Número de Instruções e colocado no *buffer* de saída junto com a instrução cabeça do bloco.

Quando o *buffer* de saída para as unidades funcionais é lido pela UF que requisitou o bloco, o Contador de Instruções é incrementado. Novamente, pela concatenação é obtida a posição da próxima instrução do bloco. A instrução lida da Memória Global de Instruções é armazenada no *buffer* de saída. Em seguida, o Registrador de Número de Instrução é decrementado. Esse procedimento se repete até o Registrador de Número de Instrução chegar a zero, quando então a UI_Global torna-se apta a aceitar uma nova requisição de bloco.

3.6 Unidade de Estruturas de Dados

A Unidade de Estruturas de Dados (UED) é composta de duas sub-unidades que atuam independentemente: a Memória de Estruturas (ME) e a Unidade de Processamento Vetorial (UPV). A ME nada mais é do que uma memória onde todas as estruturas de dados devem ser armazenadas. A UPV é um processador vetorial convencional, no qual devem ser executadas operações sobre estruturas de dados regulares. Desta forma, quando a UPV deve executar uma instrução vetorial, esta unidade busca ou armazena as estruturas de dados diretamente na ME. A transferência de dados entre a ME e UPV tem ação local à UED e portanto não causa aumento do tráfego de dados nos demais barramentos da arquitetura.

A UF identifica através do *opcode* da instrução, quando uma instrução deve ser tratada pela UED. Esta comunica-se com as unidades funcionais através de um barramento específico para esse fim.

A UED verifica, também pelo *opcode*, se existe um processamento vetorial envolvido ou apenas trata-se de uma operação básica na Memória de Estruturas (ME). Se a operação refere-se ao segundo caso, a mensagem inteira é transferida para a ME e todas ações geradas pela instrução ficam a cargo dessa unidade. Caso a instrução deva ser executada pelo processador vetorial, a UED deve gerenciar toda a transferência de dados envolvida no processamento vetorial requisitado.

Dessa forma, quando a UF verifica pelo *opcode* que se trata de uma instrução vetorial, por exemplo, a soma de dois vetores, a UF envia para a UED um pacote composto por todas as informações necessárias para executar a instrução. Este pacote executável deve conter além do *opcode* vetorial, os identificadores dos dois vetores a serem somados e o

identificador do vetor resultante. A UED determina, então que o processador vetorial deve ser utilizado. Após a transferência dos elementos dos dois vetores da ME para registradores vetoriais da UPV, esta executa a operação de soma vetorial. Quando a operação termina, a UED deve formar a ficha de resultado, que para esse caso específico, deve ser o identificador do vetor resultante da soma. Essa ficha de resultado deve ser enviada para Chave 2.

3.6.1 Memória de Estruturas

A idéia de utilizar uma memória especial para estruturas de dados em máquinas a fluxo de dados não é nova [ArKa 81]. Essa unidade foi também muito estudada [Sarg 85] pela MDFM. A Unidade de Armazenamento de Estruturas (UAE) da MDFM utiliza o modelo de armazenamento de “estruturas-I” [ArTh 81], que explora o paralelismo existente na produção e consumo de dados de uma estrutura. A implementação em hardware chegou a ser feita por Kawakami [KaGu 86].

Em princípio, na arquitetura SEED a Memória de Estruturas (ME) deve possuir a mesma estrutura interna e funcionalidade da UAE da MDFM. Sendo assim, torna-se necessário descrever o funcionamento geral da UAE da MDFM.

3.6.1.1 A Unidade de Armazenamento de Estruturas da MDFM

A UAE consiste em 4 sub-unidades que se comunicam entre si e com o anel através de passagem de mensagens. As sub-unidades realizam tarefas independentes assim, podem operar paralelamente. A unidade é conectada ao anel de processamento da MDFM através de uma chave (figura 3.6).

A Memória de Dados é formada por um bloco de memória RAM onde além do campo de dado existem dois bits adicionais: um para indicar a presença de dado armazenado e outro para sinalizar a existência de leituras antecipadas. A necessidade de utilização destes bits vem da possibilidade de ocorrer requisição de leitura de um dado sem que este tenha sido escrito (fato que pode ocorrer no modelo de “estruturas-I”). Quando isto ocorre, é enviada uma mensagem para a Fila de Acessos Antecipados, que então, armazena a requisição de leitura.

Mensagens enviadas à UAE para escrita ou leitura de um dado dirigem-se diretamente

Figura 3.6: Esquema Geral da UAE da MDFM

à memória de dados e são, respectivamente, da forma:

< função : escrita > < endereço na UAE > < tipo de dado e valor >

< função : leitura > < endereço na UAE > < destino de retorno > < rótulo >

Requisições para o armazenamento de novas estruturas são tratadas pela sub-unidade de Alocação que tem como função controlar o espaço de armazenamento na memória de dados. A mensagem tem o seguinte formato:

< função : requisição de ponteiro > < tamanho > < destino de retorno > < rótulo >

Quando o alocador reserva o espaço onde a estrutura deve ser armazenada é enviado ao anel um ponteiro (endereço inicial do espaço reservado para armazenar a estrutura) que é a referência para o acesso de seus elementos. O campo de tamanho na mensagem informa ao alocador o mínimo espaço que deve ser reservado à estrutura. Uma mensagem é enviada de volta ao anel no destino e rótulo designados na requisição.

A unidade ainda contém um mecanismo para tratar contadores de referências (*reference counts- RC*) e realizar *garbage collection*. Esta tarefa fica a cargo da sub-unidade de

Limpeza. O RC recebe o valor inicial através de uma mensagem de escrita e o seu valor é incrementado ou decrementado com mensagens da seguinte forma:

< função : incr./decr. > < endereço na UAE > < destino de retorno > < rótulo >

Um sinal de finalização da operação é enviado ao anel para o destino indicado na mensagem recebida e com o mesmo rótulo. Quando o RC chega ao valor zero é disparado o mecanismo de limpeza da estrutura na memória de dados, o que leva à alteração dos bits de presença e de leituras antecipadas, nas posições onde estavam guardados os elementos da estrutura que está sendo removida. Quando a limpeza é concluída uma mensagem é enviada à sub-unidade de Alocação para liberação do espaço.

3.6.1.2 Adaptação da UAE para a Arquitetura SEED

A organização da ME influencia diretamente na eficiência com que ambos, unidades funcionais e processador vetorial, poderão executar operações envolvendo estruturas de dados. A idéia inicial é utilizar a mesma estrutura básica de armazenamento e modo de acesso da UAE, ou seja, baseado nas estruturas-I.

No entanto, esse pode não ser o modelo ideal de implementação. Um recente trabalho [MiNB 95] apresenta uma proposta alternativa de implementação da memória de estruturas visando encontrar uma implementação que melhor se adapte ao modelo a fluxo de dados e ao modelo von Neumann de processamento vetorial. Nessa proposta, a memória de estruturas trata uma estrutura de dados como sendo formada por blocos de elementos. Cada bloco tem associado a ele um bit de presença, que deve garantir o modo de acesso estrito aos elementos, ou seja, todos os elementos do bloco devem estar prontos para que o acesso ao bloco seja liberado. Quando uma estrutura é formada por vários blocos, o acesso à cada bloco é permitido independentemente da conclusão dos outros blocos, possibilitando a execução de operações envolvendo parte da estrutura de dados. Essa estratégia explora paralelismo existente entre a criação e o processamento dos blocos de elementos, devendo apresentar maior grau de paralelismo que o modo de acesso de estruturas onde toda a estrutura deve estar pronta para que seja liberada a leitura de seus elementos. Por outro lado, impõe um maior tráfego de fichas no anel de processamento em relação ao modo de acesso ao nível de estrutura inteira. Resultados obtidos

por simulação demonstraram que esse modelo de memórias de estruturas apresenta bom desempenho. Dessa forma, esse modelo pode trazer vantagens para a arquitetura SEED.

A arquitetura proposta também utilizar-se-á do mesmo código produzido pelo gerador de código da MDFM, para implementação das operações básicas envolvendo estruturas de dados.

A ME deve interagir tanto com as unidades funcionais quanto com a UPV. Cabe à UED tratar as diferentes requisições de leitura e armazenamento de estruturas e o envio de respostas para a unidade requisitante correta, sendo indiferente à ME a origem da requisição.

3.6.2 Unidade de Processamento Vetorial

A UPV deve ser implementada por um processador vetorial convencional. Dessa forma, o processamento será realizado em unidades lógico-aritméticas independentes, implementadas na forma de *pipeline*. Essas unidades lógico-aritméticas podem operar de forma concorrente. Para obter uma boa utilização dessas unidades, é necessário garantir que a transferência de dados para essas unidades seja realizada em um tempo muito pequeno. Assim, todas as arquiteturas propostas para processamento vetorial mantém internamente ao processador um grande conjunto de registradores, formado por registradores escalares e vetoriais. Os dados são trazidos da memória para os registradores por unidades com esse fim específico, que são implementadas também como um *pipeline*.

Quando uma instrução vetorial é enviada para a UED, esta verifica quais são as estruturas de dados requeridas pela instrução. Em seguida, gera uma mensagem de leitura da estrutura de dados requerida e coloca como destino um registrador livre da UPV. Esta mensagem é então enviada para a ME. Quando for necessário uma segunda estrutura de dados para a instrução, uma nova mensagem de leitura é gerada e enviada para a ME. Quando a ME retorna o resultado da leitura, a UED armazena os elementos da estrutura em uma pequena memória, e paralelamente verifica quais são os registradores disponíveis, naquele instante, dentro da UPV. Assim que houver registradores disponíveis na UPV, a UED envia as estruturas para serem armazenadas nos registradores. A execução da instrução pelas unidades lógico-aritméticas será iniciada apenas quando as estruturas de dados estiverem armazenadas nos registradores internos à UPV.

Cada tipo de instrução vetorial necessita de dados de entrada específicos. A descrição de cada tipo de instrução e de dados de entrada só poderá ser feita após a definição de um tipo específico de processador vetorial.

3.7 Unidades Funcionais

Dentro das unidades funcionais serão executadas as instruções escalares que compõem os blocos de código. Os blocos de código são formados por instruções do conjunto de instruções da MDFM. Assim, as instruções poderão possuir de um a dois dados de entrada.

O campo de rótulo (nome de ativação e índice) é necessário para tratar as várias instanciações dos blocos que pertencem a um trecho de código reentrante. Na arquitetura SEED, os blocos de código devem ser formados por grafos de fluxo de dados acíclicos portanto, não devem existir reentrâncias dentro dos blocos. Dessa forma, o rótulo utilizado pelos dados de entrada do bloco torna-se desnecessário dentro da UF. Esse rótulo global volta a ser relevante apenas quando os resultados gerados pela execução do bloco, devem ser enviados para outros blocos de código. O rótulo também será necessário quando uma instrução, que utiliza estrutura de dados, deva ser enviada para a Unidade de Estruturas de Dados.

Para garantir a plena ocupação dos processadores, visto que existe uma latência elevada na operação de busca de código armazenado na UL_Global, será permitida a execução de vários blocos de código paralelamente dentro de uma mesma UF. Para distinguir as fichas de dados dos vários blocos em execução em uma UF, torna-se necessário atribuir um rótulo interno à UF para cada bloco. Esse rótulo local pode ter tamanho consideravelmente menor que o rótulo utilizado pelas fichas de dados de entrada dos blocos.

Utilizando o rótulo local para identificação de blocos dentro de uma UF, torna-se possível utilizar o campo de destino apenas como um deslocamento em relação ao início do bloco. Dessa forma, tanto o campo de rótulo como o de destino das fichas de dados, dentro da UF, podem ter tamanhos bastante reduzidos em relação ao rótulo global. Conseqüentemente, a chave de emparelhamento deve ser pequena, possibilitando a utilização de memórias menores e mais rápidas para implementar a Unidade de Agrupamento (UA_Local) e a Unidade de Instrução (UI) internas à UF.

A partir dessas idéias, é proposto o esquema básico de uma unidade funcional apre-

sentado na figura 3.7. Uma unidade funcional (UF) é organizada de forma a executar os blocos de código utilizando o modelo a fluxo de dados. Apesar desse modelo não ter obtido bons resultados para a execução de trechos de programas onde possa ser explorada a localidade, acredita-se que a adoção desse modelo possa trazer vantagens, pelo fato de ser permitido a execução de vários blocos de código concorrentemente.

A Chave 1 pode enviar uma ficha de dados para uma UF, desde que exista pelo menos um rótulo local livre. Quando uma UF é escalada para execução de um bloco de código, ela deve receber uma ficha contendo: os dados de entrada do bloco, o destino que o identifica e o rótulo com que o bloco deve ser executado.

Para que o rótulo global possa ser restaurado nas fichas de resultados da execução do bloco, existe a Tabela de Rótulo (TR). Essa tabela possui uma posição para cada rótulo local disponível em cada UF. Quando um rótulo é atribuído a um bloco, a UEB envia à

TR os rótulos global e local; essa unidade então, armazena na posição endereçada pelo rótulo local o valor do rótulo global.

A requisição de código é responsabilidade da Unidade de Requisição de Bloco (URB). Para que esta unidade tenha todas as informações necessárias para identificar de forma unívoca um bloco de código, a UEB envia para a URB o campo de destino da ficha de entrada e o rótulo local atribuído ao bloco. A URB apenas insere o número da UF nessa mensagem e a envia para a Unidade de Instrução Global, requisitando as instruções do bloco. Quando a URB recebe as instruções do bloco, como resposta de sua requisição, esta envia para a UI todas as instruções, com exceção da instrução cabeça. Esta instrução é enviada para a UEB, onde é feito o agrupamento dos dados de entrada com os seus respectivos destinos, formando fichas de dados que podem ser enviadas para a UA_Local. Assim, é iniciada a execução, de fato, de um bloco de código.

Na UA_Local é verificado se a ficha destina-se a uma instrução com uma ou duas entradas. Se pertencer ao último caso, é verificado se a posição de memória, endereçada pela concatenação do rótulo local e destino, possui sua ficha parceira. Se não contiver, a ficha é armazenada e o bit de presença associado àquela posição recebe o valor 1, indicando a presença de uma ficha naquela posição. Caso a ficha parceira já esteja armazenada na UA_Local, é formado um pacote contendo os dados, destino e rótulo local. Esse pacote é enviado então para a UI. Se a ficha de dado não necessita de parceira, ela é enviada diretamente para a UI.

Da mesma forma que na UA_Local, a UI obtém a instrução à qual se destina o pacote, pelo endereço formado pela concatenação do rótulo local e destino. Com as informações obtidas na UI é formado o pacote executável.

Paralelamente à formação do pacote executável, é feita a busca do rótulo global na TR. O rótulo global é inserido no pacote executável e é enviado para a Chave Vetorial (CV). Através de um bit do *opcode* a Chave Vetorial determina se a instrução é vetorial ou escalar. Se for vetorial, o pacote executável é enviado para a Unidade de Estruturas de Dados. Se for escalar, essa instrução é executada pelo processador local (PE).

Através da indicação inserida no código, em tempo de compilação, é verificado se o resultado destina-se a uma instrução interna ao bloco ou a um outro bloco de código. Se o resultado deve ser enviado a outro bloco, a ficha de resultado é enviada para a Chave

2, caso contrário, é enviada para a UA_Local.

A finalização de um bloco de código pode ser facilmente determinada, pois cada bloco possui junto a ele a informação do número de instruções que este possui. Quando o bloco é enviado à UF, essa informação é transmitida no início do bloco de instruções, posteriormente é armazenada em um contador interno à UI. A cada pacote executável formado, é decrementado o contador associado com o rótulo local. Quando este atinge o valor zero o rótulo local é liberado e enviado para a UEB. Esta unidade insere o rótulo na Fila de Rótulos Disponíveis, e atualiza o valor da ocupação da UF na Chave 1.

A seguir todas as sub-unidades da UF serão descritas com mais detalhes.

3.7.1 Unidade de Entrada de Blocos

Uma característica particular dessa nova arquitetura é a existência de “caminhos” distintos para dados e instruções. Essa estratégia visa não ocupar o barramento de dados com instruções, visto que um dos maiores “gargalos” das arquiteturas a fluxo de dados é a alta taxa de comunicação gerada em tempo de execução. Por outro lado, quando a execução de um bloco é atribuída a uma UF, as instruções devem ser trazidas de uma unidade externa à UF. Assim, enquanto todo o código não estiver armazenado na UF, os dados de entrada de um bloco de código devem permanecer armazenados. Portanto, deve existir uma unidade que seja responsável pela sincronização dos dados e das instruções do bloco para liberar a execução do bloco apenas no momento correto. Essa tarefa cabe à Unidade de Entrada de Blocos (UEB), cuja estrutura básica pode ser vista na figura 3.8.

Quando uma ficha de dados de entrada de um bloco de código chega a UF, esta é separada em três partes que são armazenadas nos *buffers* de rótulo, destino e dados. O campo de rótulo faz com que a Fila de Rótulos Disponíveis forneça um rótulo local. Após a atribuição do rótulo local, a UEB utiliza-o para disparar três atividades paralelas:

- Os dados são retirados do *buffer* de dados e são armazenados em uma memória endereçada pelo rótulo local (uma posição para cada rótulo local);
- O rótulo global e o rótulo local atribuído ao bloco são enviados para a Tabela de Rótulos, onde essa relação será utilizada para enviar os resultados finais da execução do bloco;

- Com o rótulo local e o campo de destino é formada uma mensagem que é enviada para a URB para dar início a operação de busca e armazenamento das instruções do bloco;

Após o término dessas 3 operações a UEB já pode tratar a entrada de mais um bloco de código, se existir pelo menos mais um rótulo local na Fila de Rótulos Disponíveis.

Figura 3.8: Esquema Geral de uma Unidade de Entrada de Blocos

A UEB mantém suspensa a execução do bloco até que todo o código referente àquele bloco esteja presente na UF. A execução do bloco é liberada quando a URB envia para a UEB a instrução cabeça do bloco. As informações contidas na instrução cabeça do bloco de código são: o número de dados de entrada do bloco e as instruções às quais se destinam os dados de entrada do bloco. A primeira informação é colocada em um registrador contador, e os destinos são armazenados na memória de destino. Cada destino é agrupado com cada dado de entrada formando uma ficha de dado, que deve ser enviada para a UA Local. O contador é decrementado a cada agrupamento de um dado de entrada e seu respectivo destino. Quando o contador chega a zero, o bloco de código já tem todos os seus dados de entrada disponíveis para execução e o bloco deve ser executado completamente (uma vez

que não existem pontos de interrupção dentro do um bloco de código). A UEB torna-se apta a receber uma instrução cabeça de um outro bloco de código, se existir.

Cabe também a UEB manter atualizada na Chave 1 a informação referente a ocupação de cada UF. Quando a Fila de Rótulos Disponíveis é alterada, seja pela retirada ou pela inserção de rótulos locais, a UEB deve informar a Chave 1 o número de rótulos locais em uso naquele instante.

3.7.2 Tabela de Rótulos

Quando a UEB atribui um rótulo local a um bloco de código, a relação estabelecida entre os rótulos local e global deve ser armazenada na Tabela de Rótulos (TR). Esta unidade é na realidade uma memória com uma posição para cada rótulo local. Cada posição deve armazenar o rótulo global com o qual a execução do bloco foi ativada.

Ao mesmo tempo que uma instrução é retirada da UI, a TR é consultada para a formação de um pacote executável. Contudo, nem sempre a informação do rótulo global é utilizada no processamento da instrução, como será explicado na seção 3.7.6, referente ao Processador (PE).

3.7.3 Unidade de Requisição de Blocos

A Unidade de Requisição de Blocos (figura 3.9) é responsável pelo gerenciamento da transferência das instruções da UI_Global para a UF. Quando a UEB envia para a URB os campos de destino e o rótulo local, a URB verifica na Memória de Blocos Presentes se existe na UF alguma ativação daquele bloco de código. Caso exista, as instruções já estão presentes na UI e portanto, é mais rápido copiá-las de uma região a outra da memória de instruções do que buscá-las da UI_Global.

Caso o bloco de código já esteja presente na UF, a URB identifica com qual rótulo local o bloco presente na UI foi ativado. Esse valor é enviado para a UI junto com o rótulo local do bloco cujo código está sendo requisitado. O campo de destino é armazenado na Memória de Blocos Presentes, no endereço especificado pelo rótulo local da mensagem de requisição de código. Através de um sinal de controle, a URB informa a UI que esta deve copiar as instruções de uma região da Memória de Instruções para outra (seção 3.7.5). Ao mesmo tempo, a URB copia a instrução cabeça de uma região para outra da Memória

de Instrução Cabeça. Quando a operação de cópia de código dentro da UI é concluída, a URB envia para a UEB a instrução cabeça.

Quando a URB verifica que não existe outra ativação do bloco de código requisitado pela UEB em execução na UF, a URB forma uma mensagem de requisição de leitura das instruções do bloco de código. A mensagem é composta pelas informações enviadas pela UEB e pelo número de identificação da UF. Essa última informação é utilizada para que a UI_Global possa retornar a resposta para a UF correta que enviou a requisição.

Quando a UI_Global envia para a URB a resposta de sua requisição, são enviadas no início da transmissão do bloco de código as seguintes informações:

< rótulo_local_UI > < número de instruções do bloco > < instrução cabeça >

Figura 3.9: Esquema Geral de uma Unidade de Requisição de Blocos

A URB armazena a instrução cabeça na Memória de Instrução Cabeça, e envia para a UI uma mensagem formada pelo rótulo local e pelo número de instruções que formam o bloco. Após a UI tratar essas informações, a URB envia, uma a uma, todas as instruções do bloco, que recebe da UI_Global. Quando a UI conclui o armazenamento de todas as

instruções, esta envia para a URB o valor do rótulo local cuja execução pode ser iniciada. Como consequência, a URB envia a instrução cabeça para a UEB, habilitando a execução do bloco, pela liberação dos dados de entrada para a execução do bloco.

3.7.4 Unidade de Agrupamento Local

A estrutura geral da UA_Local é idêntica à estrutura das unidades de agrupamento do anel externo. Contudo, em função da exploração de localidade pela arquitetura SEED, a UA_Local pode ter implementação muito mais simplificada. A redução significativa do tamanho da chave de emparelhamento dentro da UF, proporcionada pela localidade de efeito das instruções do bloco, é de fundamental importância para que a operação envolvendo a Memória de Dados possa ser realizada com alta velocidade. Como já foi exposto, o tamanho da chave de emparelhamento depende do tamanho dos campos de rótulo local e de destino. Dentro da UF, o tamanho do campo de destino deve ser limitado ao número máximo de instruções que formam o bloco. Em princípio, será definido que cada bloco poderá possuir no máximo 32 instruções, portanto o campo de destino deve ser composto por 5 bits.

O número de blocos de código que podem ser executados paralelamente em uma UF deve ser analisado posteriormente. Como esse número deve variar em função do número de unidades funcionais presentes na arquitetura, deve-se analisar o comportamento e desempenho da arquitetura durante a execução de *benchmarks* para, posteriormente, fixar o melhor valor. Contudo, para efeito de primeira análise será suposto que cada UF poderá executar paralelamente 64 blocos. A partir dessas estimativas, a chave deve ser composta de: 6 bits para especificação do rótulo local e 5 bits de destino para varrer todas as instruções dentro do mesmo rótulo. Portanto, a chave de emparelhamento deve ficar reduzida a 11 bits, possibilitando que a memória de dados da UA_Local possa ser implementada por apenas 2K posições. Pode-se utilizar uma memória com tempo de acesso muito menor que a utilizada pelas unidades de agrupamento, devido ao pequeno tamanho exigido para a memória de dados.

Uma ficha de dado que chega na UA_Local possui o seguinte formato:

$$\langle \underset{(6 \text{ bits})}{rot_local} \rangle \langle \underset{(5 \text{ bits})}{destino} \rangle \langle \underset{(1 \text{ bit})}{porta} \rangle \langle \underset{(1 \text{ bit})}{n_cnt} \rangle \langle \underset{(64 \text{ bits})}{dado} \rangle \langle \underset{(4 \text{ bits})}{tipo} \rangle$$

Como cada bloco de código será composto por uma série de instruções do conjunto

de instruções da MDFM, as instruções poderão ter no máximo dois dados de entrada. Assim, quando uma ficha de dado é recebida pela UA_Local, esta verifica, através do campo *n_ent*, se esta ficha se destina a uma instrução que precisa de um ou dois dados de entrada. Se a instrução for unária, a ficha passa diretamente para a UI. Caso contrário, através dos campos que compõem a chave de emparelhamento (rótulo local e destino) é formado o endereço no qual deve-se procurar na Memória de Dados a sua ficha parceira.

Cada posição de memória na UA_local tem a seguinte forma:

bit de presença	tipo	dado
1 bit	4 bits	64 bits

A posição apontada pela chave de emparelhamento tem seu bit de presença verificado. Se este indicar a presença de um dado, a posição é lida e o dado é agrupado à ficha de entrada. O pacote formado pelos dados, rótulo local e destino é enviado para a UI. Caso não exista nenhum dado armazenado naquela posição, a ficha é armazenada e o bit de presença associado à posição é alterado para 1.

3.7.5 Unidade de Instrução

Todas as instruções dos blocos de código em execução na UF ficam armazenados na Memória de Instrução dentro da Unidade de Instrução (UI) (figura 3.10). Cada posição da memória de possui o seguinte formato:

<i>opcode</i>	destino ₁	bit ₁	porta ₁	n_ent ₁	destino ₂	bit ₂	porta ₂	n_ent ₂
8 bits	12 bits	1 bit	2 bits	2 bits	64 bits	1 bit	2 bits	2 bits

Os campos bit₁ e bit₂ determinam se os destinos dos resultados gerados pela execução da instrução pertencem ao mesmo bloco, e portanto, devem continuar na mesma UF, ou referenciam-se a outro bloco, devendo ser enviados para a Chave 2. Como tanto destino₁ como destino₂ podem endereçar outro bloco, os campos de *destino*, *n_ent* e *porta*, de cada ficha de dado contendo um resultado, devem possuir tamanhos iguais aos utilizados para transmissão dos dados destinados aos blocos.

Além disso, para melhor adequação da arquitetura ao conjunto de instruções que será utilizado por ela, o campo destino₂ deve possuir o tamanho necessário para conter o maior tipo de dado existente na máquina (64 bits), pois esse campo pode ser utilizado

Figura 3.10: Esquema Geral de uma Unidade de Instrução

para inserir constantes nos programas.

A UI possui duas atividades básicas: armazenar as instruções dos blocos, cuja execução foi habilitada, e agrupar ao pacote de dados às informações armazenadas na Memória de Instrução para formação do pacote executável. Cada atividade será descrita a seguir.

3.7.5.1 Armazenamento de Instruções

O armazenamento de instruções é uma atividade realizada pela UI, que depende diretamente das mensagens enviadas pela URB. Como já foi mencionado na seção 3.6.3, fica sob a responsabilidade da URB verificar se a UI já possui na Memória de Instrução as instruções do bloco de código que deve ter sua execução iniciada, ou se as instruções devem ser trazidas da UI_Global. Essas duas possibilidades geram duas atividades diferentes na UI, que serão analisadas mais detalhadamente a seguir.

- Armazenamento de Instruções vindas da UI_Global

A primeira mensagem que a UI recebe da URB, quando esta recebe a resposta de sua requisição de código da UI_Global, contém o rótulo local e o número de instruções que formam o bloco de código. O rótulo local é utilizado para indicar em qual região da Memória de Instruções devem ser armazenadas as instruções que serão enviadas pela URB. O número de instruções será armazenado na Memória de Contador de Execução (MCE) e na Memória de Contador de Cópia (MCC). Ambas possuem uma posição para cada rótulo local, ou seja, uma posição associada com cada bloco de código na Memória de Instrução. O número de instruções do bloco também será armazenado em um registrador, denominado de Contador de Armazenamento, que será utilizado para determinar a conclusão do armazenamento do bloco.

Posteriormente, a URB envia a primeira instrução do bloco, que será armazenada na primeira posição da região da Memória de Instruções designada pelo rótulo local e o Contador de Armazenamento é decrementado. As próximas instruções serão armazenadas em posições subseqüentes ao início da região de memória. Quando o Contador de Armazenamento chega a zero, o bloco identificado pelo rótulo local deve ter sua execução iniciada. Para tanto, a UI deve enviar para a URB um sinal representando a conclusão da operação de armazenamento. Caso a URB não possa tratar o início da execução do bloco, o rótulo local é inserido na Fila de Rótulos Habilitados. Quando a URB estiver livre, retirará dessa fila o rótulo do próximo bloco que terá sua execução liberada através do envio da instrução cabeça para a UEB.

- **Cópia de Código Armazenado na UI**

Quando a URB verifica que o bloco requisitado já está armazenado na UI, ou seja, outra ativação do mesmo bloco de código já está em execução na UF, a URB determina que a UI deve replicar o código de uma região de memória para outra. Dessa forma, é evitada toda a latência da busca de código na UI_Global.

A UI deve receber da URB o rótulo local destino, que identifica a região da Memória de Instruções onde deve ser armazenado o código que está sendo requisitado, e o rótulo local fonte, que determina a região onde o código desejado está armazenado. Através do valor armazenado na posição endereçada pelo rótulo fonte na MCC.

determina-se o número de instruções que devem ser copiadas. A operação de cópia de blocos de código deve também atualizar tanto a MCE quanto a MCC.

A conclusão dessa operação determina que o bloco de código já pode ter sua execução iniciada pelo mesmo processo descrito no item anterior.

3.7.5.2 Agrupamento do Código aos Dados

Os dados sincronizados na UA_Local formam um pacote de dados, que é enviado para a UI, onde é obtida a instrução à qual destinam-se os dados. A UI habilita a leitura da posição da Memória de Instruções endereçada pelos campos rótulo local e destino, onde está armazenada sua respectiva instrução. A partir dessa operação, é formado o pacote executável que deve ser enviado à Chave Vetorial.

A cada instrução habilitada para a execução, ou seja, a cada instrução enviada para a chave vetorial, a posição identificada pelo rótulo local na MCE é decrementada. Portanto, essa posição sempre contém o número de instruções do bloco que ainda devem ser executadas. Quando essa posição chega a zero, é gerado um sinal de controle que, junto com o rótulo local, é enviado para a UEB. Na UEB o rótulo local associado ao bloco que teve sua execução concluída, é inserido na Fila de Rótulos Disponíveis da UEB.

3.7.6 Processador

Cada unidade funcional possui um processador (PE) onde todas as instruções dos blocos de código são, de fato, executadas. O PE recebe da chave vetorial um pacote executável formado por todas as informações armazenadas na Memória de Instruções, os dados de entrada e os rótulos global e local.

A presença do rótulo global é necessária por dois motivos: o resultado (ou resultados) gerado pela execução da instrução pode ser enviado a outro bloco de código, devendo a ficha de resultado carregar o mesmo rótulo com que o bloco foi ativado, e pela existência de instruções específicas de manipulação de rótulo.

Quando um pacote executável chega em um PE, a primeira operação efetuada é a separação dos campos que compõem esse pacote nos registradores internos ao processador. Com a decodificação do *opcode* da instrução são geradas operações que podem atuar sobre qualquer um desses registradores. Dependendo do tipo da instrução são geradas

ações diferentes. Existem dois tipos básicos de instruções:

- **Lógico-aritméticas:** essas instruções podem utilizar um ou dois dados armazenados nos registradores de dados. Quando uma das entradas for uma constante, o PE busca no registrador destino₂ o seu valor. Quando o resultado da execução da instrução é gerado, deve-se verificar pelo campo bit_dest₁, se a ficha de dado deve ser enviada para a Chave 2 ou para a UA_Local. Se bit_dest₁ for igual à 1, a ficha formada pelo rótulo global, destino₁ e o valor do resultado será colocada no *buffer* de saída para a Chave 2. Caso contrário, a ficha deve ser armazenada no *buffer* de saída para a UA_Local, e conterà o rótulo local, o campo de destino, formado pelos 5 bits menos significativos do registrador destino₁ e o dado. Se a instrução possuir dois destinos de saída para seu resultado, a segunda ficha de resultado é formada da mesma forma que a primeira, só que utilizando os campos de bit_dest₂ e destino₂.
- **Manipulação de Rótulo:** através dessas instruções, os campos de nome de ativação e índice são alterados. Os valores desses campos podem sofrer alterações através da atribuição direta de valores especificados nos registradores de dados, e além disso o campo de índice pode ter seu valor incrementado através da execução de uma operação aritmética aplicada diretamente sobre esse campo. Todas as saídas geradas por esse tipo de instrução devem ser enviadas para a Chave 2, uma vez que sempre geram novos contextos que podem ser executados paralelamente a outros na arquitetura SEED.

Tanto a saída para a Chave 2 como para a UA_Local possuem ligadas a elas uma pequena fila de fichas para evitar que o processador fique ocioso devido ao atraso de outras unidades.

3.8 Resumo

Esse capítulo foi dedicado à descrição da arquitetura SEED. Essa arquitetura tem como objetivo utilizar as boas características e apresentar soluções para os problemas verificados pelo comportamento das arquiteturas a fluxo de dados de primeira geração.

A proposta básica desta arquitetura é permitir que instruções cujos dados devam ser consumidos em um espaço curto de tempo, sejam executadas em um único processador.

Dessa forma, espera-se que com o agrupamento de instruções em blocos, o processamento de instruções cuja execução é estritamente seqüencial seja mais eficiente que no modelo a fluxo de dados puro.

A utilização de um anel de processamento para tratar a comunicação entre blocos e outro anel para a comunicação intra-blocos, deve permitir que ambos os anéis tenham um desempenho melhor, pois o tráfego de fichas deverá ser consideravelmente reduzido em ambos.

No próximo capítulo serão discutidas as questões envolvendo o conjunto de instruções e o sistema de geração de código a ser utilizado pela arquitetura SEED.

Capítulo 4

Definição e Avaliação do Conjunto de Instruções da Arquitetura SEED

A utilização de um conjunto de instruções que consiga explorar todo o potencial de uma arquitetura, é de fundamental importância para que esta possa obter um bom desempenho. Quando uma arquitetura possui recursos para exploração de paralelismo, o conjunto de instruções deve permitir também que a geração de atividades paralelas seja realizada de forma simples e eficiente, bem como as operações que envolvem comunicação e sincronização entre os processos paralelos. Pelo fato da arquitetura SEED ser uma arquitetura baseada no modelo a fluxo de dados dinâmico por rotulação de fichas, o código que pode aproveitar melhor os recursos apresentados por essa arquitetura deve ser fortemente ligado a este modelo. Dessa forma, foi feita a opção de utilizar um conjunto de instruções igual ao utilizado pela máquina a fluxo de dados de Manchester (MDFM).

Contudo, o sistema de compilação das arquiteturas a fluxo de dados é considerado bastante ineficiente, principalmente quando comparado com os códigos gerados para arquiteturas von Neumann. Essa ineficiência é consequência da inserção de um número excessivo de instruções para manipulação de rótulo. Essas instruções são utilizadas tanto para o processamento de estruturas de dados quanto para o tratamento de código reentrante. Assim, a utilização do gerador de código da MDFM pode trazer grandes desvantagens para a arquitetura SEED pois, a utilização de um código que impõe um número maior de instruções a serem executadas, pode fazer com que o paralelismo a ser explorado pela arquitetura seja desperdiçado pelo processamento de instruções desnecessárias.

No entanto, várias otimizações foram inseridas no gerador de código da MDFM vi-

sando sanar as deficiências apontadas. Para verificar a qualidade do código gerado pelo sistema de compilação da MDFM, foi feita uma análise comparativa do código da MDFM e dos códigos gerados por duas máquinas convencionais, uma baseada na filosofia RISC e outra na filosofia CISC. Através desta análise espera-se verificar se a escolha do conjunto de instruções da MDFM pode proporcionar as vantagens almejadas para a arquitetura SEED. Além disso, dependendo dos resultados obtidos pode-se decidir qual tipo de código é o mais indicado para ser utilizado pela arquitetura SEED.

A primeira parte deste capítulo é dedicada à descrição geral do sistema de geração de código da MDFM, apresentando as principais características da linguagem SISAL. Em seguida, é apresentada a descrição da análise comparativa entre o código gerado para a MDFM e os gerados para processadores RISC e CISC. Após a análise dos resultados obtidos através dessa comparação, são apresentadas as principais características desejadas para o código a ser utilizado pela arquitetura SEED.

4.1 Geração de Código na MDFM

A principal linguagem de alto nível utilizada para a programação da máquina de Manchester é SISAL (*Streams and Iteration in a Single Assignment Language*) [MSAG 84]. SISAL é o resultado da colaboração entre Digital Equipment Corporation, Lawrence Livermore National Laboratory, Universidade de Manchester e Universidade de Colorado. Desta mesma união foi elaborado o IF1, um formato intermediário independente da máquina [SkGl 85]. SISAL é uma linguagem funcional que não permite a re-atribuição de valores para as variáveis e não possui variáveis globais, efeitos colaterais (*side-effects*) e o conceito de execução seqüencial explícita. Funções nesta linguagem são, verdadeiramente, funções na forma matemática.

A transformação de um programa SISAL em código executável no sistema a fluxo de dados de Manchester é realizado em várias fases. Um esquema geral pode ser observado na figura 4.1.

Um programa fictício **prog.sis** é utilizado para representar o funcionamento do sistema de compilação. Através do compilador SISAL é gerado um grafo intermediário que contém nós primitivos de fluxo de dados (como operações aritméticas) e “macro-nós”. Nesta fase também é verificado se um nó do grafo IF1 possui o número errado de portas de entrada e

Figura 4.1: Geração de Código na MDFM

de saída, e são inseridos nós extras no grafo para a realização de contagem de referências (*reference counts*) e coleta de lixo (*garbage collection*). O programa IDC (*Intermediate Dataflow Code*) faz a compilação das “macros” no programa, e como resultado é obtido um formato de baixo nível chamado de IGF (*Intermediate Graphical Format*). Um otimizador (OPTIM) é responsável pela geração de um código relocável e pela execução de algumas otimizações como:

- Troca de chamada de função com o código da função (*inline*), o que reduz o *overhead* do tempo de execução da chamada de função.
- Eliminação de sub-expressões comuns.
- Detecção e remoção de expressões que são invariantes dentro de blocos repetitivos.

Características do SISAL, como única atribuição a variáveis, ausência de efeitos colaterais e passagem de parâmetros para função apenas através de valores, são fundamentais para que as otimizações possam ser feitas em pouco tempo.

4.2 Descrição da Análise Comparativa

O objetivo desse estudo é realizar uma análise sobre a qualidade do código gerado pela MDFM comparando-o com códigos gerados para máquinas convencionais. Foram observados os códigos de duas máquinas baseadas em filosofias bastante distintas: o processador SPARC-2, representando uma máquina RISC, e o computador VAX Vector 600-420, representando uma máquina CISC. A estratégia adotada para essa análise será descrita nesta seção.

Todo algoritmo computacional é criado para resolver um determinado problema utilizando um modelo computacional específico. Na maioria dos casos, o modelo computacional, no qual o algoritmo deverá ser executado, impõe a inserção de operações extras no algoritmo apenas para satisfazer exigências do modelo e não do problema. Essas instruções são denominadas de instruções de *overhead*, pois quando é utilizado outro modelo computacional para resolução do problema essas instruções deixam de ser necessárias. Por exemplo, se dois vetores devem ser somados em uma máquina baseada no modelo convencional, as operações de acesso aos elementos dos vetores devem ser consideradas de *overhead*, uma vez que existem modelos computacionais nos quais não existe sequer o conceito de estruturas de dados armazenadas. Para esse problema específico a única operação considerada útil é a operação de adição.

Partindo dessas considerações, a análise comparativa é iniciada pela geração do programa em linguagem de alto nível do algoritmo em estudo. Depois dessa etapa, deve-se identificar quais são as instruções úteis do programa. Após o programa ser compilado deve-se verificar quais são as instruções de máquina, utilizadas pelo gerador de código, para implementar as operações úteis do programa. O próximo passo é realizar uma análise quantitativa das instruções de *overhead* e das instruções essenciais do programa. Finalmente, deve ser obtida a proporção do número de instruções de *overhead* pelo número de instruções úteis do programa. Quando um algoritmo envolve a execução de repetições, deve-se considerar na contagem do número de instruções o número de vezes que as instruções são executadas, tendo em vista que é desejado obter a proporção entre as instruções úteis e de *overhead* em tempo de execução.

Para efetuar uma análise mais abrangente das características dos códigos gerados para as várias arquiteturas, foram utilizados três *benchmarks* com características bastante

distintas. Para todas as arquiteturas partiu-se do mesmo algoritmo, mas não da mesma linguagem de alto nível. Como foi dito na seção anterior, a geração de código da MDFM parte de programas escritos na linguagem SISAL. Apesar do processador SPARC-2 possuir um compilador para a linguagem SISAL (denominado OSC), optou-se pela utilização de programas implementados em linguagem C, uma vez que compilador da linguagem C deve proporcionar código mais eficiente para esse processador. Foi utilizado para o processador SPARC-2 o compilador C++ da *gnu* versão 2.4. Para o VAX foi utilizado o mesmo programa C utilizado para o SPARC-2 e para gerar o programa em linguagem de máquina foi utilizado o compilador VAX C versão 3.1.

Os algoritmos utilizados como *benchmarks* foram: integração binária, criação de matrizes e multiplicação de matrizes. Na seção seguinte, será exemplificada a metodologia empregada na análise comparativa através do programa de integração binária

4.2.1 Integração Binária

O programa SISAL que implementa o algoritmo de integração binária pode ser visto na figura 4.2. Trata-se de um programa duplamente recursivo, que envolve um número grande de chamadas de funções e processamento numérico. As operações que são consideradas essenciais para o algoritmo estão apresentadas em negrito. Essas instruções efetuam as operações aritméticas para resolução do problema.

Definidas as instruções úteis do programa em linguagem de alto nível, deve-se identificar agora as instruções em linguagem de máquina que implementam essas operações úteis, para todos os processadores escolhidos para essa análise.

O número de vezes que cada instrução que compõe o programa é executada deve ser verificado, pois deseja-se estabelecer dinamicamente a proporção entre o número de instruções de *overhead* pelo número de instruções úteis. Para esse algoritmo específico, o número de repetições de cada trecho do programa depende do valor da variável *depth*. Dependendo do número de repetições o programa pode ser separado em três partes:

- **trecho 1:** Os trechos do programa que realizam a passagem de parâmetros para a função *area*, o cálculo do valor de *mid*, a comparação $depth \leq 1$ e retorno da função *area* são repetidos 2^{depth} vezes.

- **trecho 2:** As instruções que implementam a cláusula *then* da função *area* são repetidas $(2^{depth}/2 - 1)$ vezes.
- **trecho 3:** A condição *else* é repetida $(2^{depth}/2)$ vezes.

```

function area(l,r,depth : integer returns integer)
let
    mid := (l + r) / 2;
in
    if depth <= 1 then
        (r - l) * ((x * x) - (6 * x) - 10)
    else
        area(l,mid,depth - 1) + area(mid,r,depth - 1)
    end if
end let
end function

```

Figura 4.2: Programa SISAL para Integração Binária

Em seguida, serão apresentados os códigos gerados para os vários processadores e a contagem do número de instruções pertencentes a cada trecho do programa desse código.

4.2.1.1 Código Gerado para o VAX

O código gerado pelo compilador C do VAX é apresentado na figura 4.3. Da mesma forma que na figura 4.2, as instruções úteis estão destacadas em negrito.

O número de instruções úteis e de *overhead* de cada trecho do programa são apresentados na tabela 4.1.

Trecho do Programa	Instr. <i>overhead</i>	Instr. úteis
1	5	2
2	1	6
3	10	2

Tabela 4.1: Número de Instruções Úteis e de *Overhead* do Programa Integração Binária para o VAX

```

VAX C      V3.1-051
area: .entry area, m<r2,r3,r4,r5,r6>

trecho 1 {
    subl2  #4, sp
    movl   4(ap), r6
    movl   8(ap), r5
    addl3   r6, r5, r4
    divl2   #2, r4
    cmpl   12(ap), #1
    bgtr   sym.1
}

trecho 2 {
    subl3   r6, r5, r2
    mull3   r4, r4, r1
    mull3   #6, r4, r0
    subl2   r0, r1
    subl2   #10, r1
    mull3   r2, r1, r0
    ret
}

sym.1:
trecho 3 {
    subl3   #1, 12(ap), r3
    pushl   r3
    pushl   r4
    pushl   r6
    calls   #3, AREA
    movl    r0, r2
    pushl   r3
    pushl   r5
    pushl   r4
    calls   #3, AREA
    addl2   r2, r0
    ret
}

```

Figura 4.3: Código Gerado para o VAX do Programa Integração Binária

Trecho do Programa	Instr. <i>overhead</i>	Instr. úteis
1	14	2
2	37	12
3	12	3

Tabela 4.2: Número de Instruções Úteis e de *Overhead* do Programa Integração Binária para a SPARC-2

4.2.1.2 Código Gerado para a SPARC-2

O código gerado pelo compilador *gnu C++* para o processador SPARC-2 é apresentado na figura 4.4. Esse código utiliza-se da função *.umul* para efetuar a multiplicação para o cálculo de mid^2 e em seguida, para multiplicar $(r - l)$ ao valor resultante da outra expressão da cláusula *then*. Portanto a função *.umul* é executada duas vezes em cada ativação da função *area*. Sendo assim, as instruções que formam a função *.umul* devem ser contadas duas vezes cada vez que a cláusula *then* é executada.

A contagem do número de instruções úteis e de *overhead* desse código é apresentado na tabela 4.2.

4.2.1.3 Código Gerado para a MDFM

A figura 4.5 apresenta o grafo de fluxo de dados gerado pelo sistema de compilação da MDFM. A tabela 4.3 apresenta o número de instruções desse código de cada trecho do programa.

Trecho do Programa	Instr. <i>overhead</i>	Instr. úteis
1	6	2
2	2	6
3	17	3

Tabela 4.3: Número de Instruções Úteis e de *Overhead* do Programa Integração Binária do Grafo de Fluxo de Dados

```

gcc2_compiled.:
gnu_compiled_cplusplus:
_area_Fiii:
!#PROLOGUE# 0

trecho 1 {
    save %sp, -112, %sp
    st %i0, [%fp+68]
    st %i1, [%fp+72]
    st %i2, [%fp+76]
    ld [%fp+68], %o0
    ld [%fp+72], %o1
    add %o0, %o1, %o0
    mov %o0, %o1
    srl %o1, 31, %o1
    add %o0, %o1, %o0
    sra %o0, 1, %o1
    st %o1, [%fp-16]
    ld [%fp+76], %o0
    cmp %o0, 1
    bg L244
}

trecho 2 {
    ld [%fp+72], %o0
    ld [%fp+68], %o1
    sub %o0, %o1, %l0
    ld [%fp-16], %o0
    ld [%fp-16], %o1
    call .umul, 0
    nop
    ld [%fp-16], %o1
    mov %o1, %o3
    sll %o3, 1, %o2
    add %o2, %o1, %o2
    sll %o2, 1, %o1
    sub %o0, %o1, %o0
    add %o0, -10, %o1
    mov %l0, %o0
    call .umul, 0
    nop
    mov %o0, %i0
    b L243
    nop
    b L243
    nop
    b L245
}

L244:
trecho 3 {
    ld [%fp+76], %o0
    add %o0, -1, %o2
    ld [%fp+68], %o0
    ld [%fp-16], %o1
    call _area_Fiii, 0
    nop
    fmovs %f0, %f4
    st %f4, [%fp-8]
    ld [%fp-8], %l0
    ld [%fp+76], %o0
    add %o0, -1, %o2
    ld [%fp-16], %o0
    ld [%fp+72], %o1
    call _area_Fiii, 0
    nop
    fmovs %f0, %f2
    st %l0, [%fp-8]
    ld [%fp-8], %f4
    fadds %f4, %f2, %f3
    st %f3, [%fp-12]
}

L245:
L243:
trecho 1 {
    ret
    restore
}

trecho 2 {
    .umul
    mov %o0, %cy
    andncc %o0, 0xF, %o4
    be <.umul+216>
    mulsc %o4, %o1, %o4
    mulsc %o4, %o1, %o4
    mulsc %o4, %o1, %o4
    mulsc %o4, %o1, %o4
    mulsc %o4, %o1, %o4
    rd %cy, %o5
    tst %o1
    bge <.umul+256>
    sll %o4, %o0
    srl %o5, 0x1c, %o5
    or %o5, %o0, %o0
    ret
}

```

Figura 4.4: Código Gerado para a SPARC-2 do Programa Integração Binária

Figura 4.5: Grafo de Fluxo de Dados do Programa de Integração Binária Gerado para MDFM

4.2.1.4 Resultados Obtidos

Após a identificação e a contagem das instruções em linguagem de máquina de cada processador, foi analisado o comportamento da razão entre o número de instruções de *overhead* e o número de instruções úteis, para alguns valores de *depth*. Os resultados estão contidos na tabela 4.4. Pelo fato dos resultados obtidos pelo código gerado para o processador SPARC-2 serem muito distantes dos resultados obtidos tanto para a MDFM quanto para o VAX, foi feita também a análise do código gerado pelo compilador *gnu C++* com opção de otimização. Os resultados dessa análise também estão presentes na tabela 4.4.

<i>depth</i>	SPARC-2	SPARC-2 Otimização	MDFM	VAX
5	4.03	2.94	2.33	1.71
6	4.04	2.94	2.35	1.73
7	4.05	2.94	2.37	1.74
10	4.05	2.94	2.38	1.75
15	4.05	2.94	2.38	1.75

Tabela 4.4: Resultados Obtidos para Integração Binária

Pode-se observar que o código gerado para o VAX foi o que apresentou o melhor resultado, impondo em média a execução de 1.74 instruções de *overhead* para cada instrução útil. O resultado menos satisfatório foi obtido pelo processador SPARC-2 quando não foi forçada a realização de otimização, pois em média foram exigidas 4.05 instruções de *overhead* para cada instrução útil. Os resultados obtidos com o código gerado para a MDFM demonstram que este possui relação instrução de *overhead* por instrução útil aproximadamente 25% maior que o código do VAX e 25% menor que o código otimizado do SPARC-2.

Com estes resultados pode-se verificar que o código da MDFM para o programa de integração binária está em um nível intermediário entre códigos tipicamente RISC e CISC.

4.2.2 Criação de Matriz

O próximo programa com o qual foi realizada a mesma análise descrita para o programa de integração binária foi o programa de criação de matriz. A figura 4.6 apresenta o programa SISAL utilizado para geração do grafo de fluxo de dados para a MDFM, que implementa a criação de uma matriz. As únicas instruções consideradas úteis neste programa são as instruções de armazenamento dos elementos da matriz. As instruções de indexação e cálculo do endereço de armazenamento dos elementos da matriz dependem diretamente de cada modelo computacional adotado, portanto são consideradas instruções de *overhead*.

```
function Fill(n1,n2: integer; R: integer returns Matrix)
  for i in 1..n1
    el :=
      for j in 1..n2
        a := r
      returns array of a
    end for
  returns array of el
  end for
end function
```

Figura 4.6: Programa SISAL para Criação de Matriz

A partir do mesmo algoritmo do programa SISAL apresentado, foi implementado o programa na linguagem C, com o qual foram gerados os códigos para o VAX e SPARC-2. A tabela 4.5 apresenta os resultados obtidos para os vários processadores.

Dimensão	SPARC-2	SPARC-2 Otimização	MDFM	VAX
8	18.41	5.29	7.17	5.11
16	17.46	5.17	5.42	4.53
64	17.02	4.14	4.10	4.13
128	17.08	4.07	3.89	4.06
256	17.04	4.04	3.78	4.03
512	17.02	4.02	3.72	4.02

Tabela 4.5: Resultados Obtidos para Criação de Matriz

Os resultados obtidos para o programa de criação de matriz demonstram a existência

de diferenças se comparados aos obtidos pelo programa de integração binária. Neste caso, a proporção entre as instruções de *overhead* e úteis do código da MDFM chega a ser pior que o código otimizado do SPARC-2 e do VAX, quando são criadas matrizes quadradas com dimensão menor que 128 elementos. A partir desse valor o código da MDFM torna-se melhor que os demais códigos.

4.2.3 Multiplicação de Matrizes

O programa de multiplicação de matrizes apresentado na figura 4.7 também foi utilizado nesta análise comparativa dos códigos gerados para vários processadores. Para este programa as instruções consideradas úteis para o algoritmo são: as instruções que multiplicam os elementos das linhas da primeira matriz aos elementos das colunas da segunda matriz e as instruções que somam os resultados obtidos pelas instruções de multiplicação.

Os resultados obtidos são apresentados na tabela 4.6. Assim como observado pelos resultados obtidos com o programa de integração binária, o código da MDFM apresentou valores para a relação entre instruções de *overhead* por instrução útil melhores que os resultados obtidos com o código otimizado do SPARC-2, mas piores que o código do VAX. Contudo, os resultados obtidos para o código gerado para o SPARC-2 sem forçar todas as otimizações demonstraram ser muito maiores que os resultados obtidos pelos demais.

```

function matmult(A,B : Matrix; N1,N2,M : integer returns Matrix)
for I in 1..N1
  Row :=
  for J in 1..N2
    El :=
    for K in 1..M
      R := A[I,K] * B[K,J]
      returns value of sum R
    end for
    returns array of El
  end for
  returns array of Row
end for
end function

```

Figura 4.7: Programa SISAL para Multiplicação de Matrizes

Dimensão	SPARC-2	SPARC-2 Otimização	MDFM	VAX
8	14.35	6.80	6.99	3.88
16	14.04	6.64	5.37	3.42
64	13.80	6.53	4.31	3.08
128	13.79	6.52	4.16	3.05
256	13.77	6.51	4.08	3.03
512	13.76	6.50	4.04	3.01

Tabela 4.6: Resultados Obtidos para Multiplicação de Matrizes

4.2.4 Análise dos Resultados da Análise Comparativa

Figura 4.8: Gráfico da Razão entre o Número de Instruções *Overhead* para cada Instrução Útil para o Programa de Integração Binária

A figura 4.8 apresenta o gráfico da variação da razão entre instruções de *overhead* e úteis do programa de integração binária em função da variável *depth* para todas as máquinas utilizadas na análise comparativa. Pode-se observar que quase não existe variação dos valores obtidos para os vários processadores quando o valor de *depth* é alterado. Além disso, o maior número de instruções de *overhead* para cada instrução útil foi exigido pelo código gerado para o SPARC-2, quando a opção de otimização não foi utilizada. Em média, esse código exigiu cerca de 4 instruções de *overhead* para cada instrução útil. Comparando essa média com a obtida com o código gerado para o SPARC-2 com otimizações (aproximadamente 3 instruções de *overhead*), verificou-se que há uma redução

de cerca de 38% do número de instruções de *overhead* para cada instrução útil quando é forçada a realização de todas as otimizações. O código da MDFM exigiu em média 2.37 instruções de *overhead* para cada instrução útil, o que representa um decréscimo de 20% do valor médio requerido pelo o código otimizado do SPARC-2. O valor médio do número de instruções de *overhead* por instrução útil verificado para o código do VAX foi igual a 1.73, representando uma redução de 42% em relação ao código da SPARC-2 otimizado.

Para o programa de integração binária o código do VAX obteve o melhor resultado, ou seja, exigiu um número menor de instruções de *overhead* para cada útil. Em média, o código do VAX exigiu um número de instruções de *overhead* 27% menor que o código da MDFM e 42% menor que o código do SPARC-2 otimizado. Comparando o código do SPARC-2 otimizado ao código da MDFM, para esse mesmo programa, verifica-se que o código para a SPARC-2 necessita de 24% a mais de instruções de *overhead*.

Para o programa de criação de uma matriz (gráfico apresentado na figura 4.9), os resultados demonstraram que, em média, o código gerado para o VAX foi o mais eficiente. Pode-se verificar também que com o aumento da dimensão da matriz gerada todos os códigos tornaram-se mais eficientes.

Figura 4.9: Gráfico da Razão entre o Número de Instruções *Overhead* para cada Instrução Útil para o Programa de Criação de Matriz

Analisando a média dos resultados obtidos com as várias dimensões da matriz criada, verifica-se que comparado ao código do VAX o código da MDFM exige 11.7% a mais de instruções de *overhead* para cada instrução útil. O código otimizado do SPARC-2

necessitou de 0.9% instruções de *overhead* a menos que o código da MDFM e o código sem otimização utiliza um número 75.2% maior de instruções de *overhead*.

O programa de multiplicação de matrizes tem seus resultados comparados no gráfico apresentado na figura 4.10. O código da MDFM utilizou, em média, 26.6% instruções de *overhead* a menos que o código gerado com otimização para o processador SPARC-2. No entanto, o código do VAX exigiu 32.7% a menos de instruções de *overhead* para cada instrução útil em relação à MDFM.

Para avaliar de uma forma mais abrangente o código gerado por todos os sistemas de compilação considerados nessa análise, foi observado o comportamento de cada sistema como uma média dos resultados obtidos por todos os *benchmarks*. Dessa forma, pode-se concluir que para os 3 *benchmarks* utilizados o código gerado para a MDFM exigiu um número 19.4% menor de instruções de *overhead* para cada instrução útil em relação aos códigos otimizados do SPARC-2. E comparado com os códigos gerados para o VAX foi exigido cerca de 32.7% a mais de instruções de *overhead* para cada útil.

Figura 4.10: Gráfico da Razão entre o Número de Instruções *Overhead* para cada Instrução Útil para o Programa de Multiplicação de Matrizes

4.3 Conclusão da Análise do Código

A análise comparativa apresentada neste capítulo foi realizada unicamente com o intuito de avaliar a eficiência do código gerado pelo sistema de compilação de uma máquina a

fluxo de dados puro, e que utiliza um conjunto de instruções fortemente ligado ao modelo a fluxo de dados. Assim, as conclusões apresentadas nesta seção serão exclusivamente referentes ao código da MDFM.

Em função dos resultados obtidos para os 3 *benchmarks*, pode-se observar que através de otimizações implementadas no gerador de código da MDFM [BoSa 86] [BoSa89], o seu código aproximou-se muito dos códigos gerados para máquinas convencionais.

As operações que envolvem a manipulação de rótulo foram bastante otimizadas, tornando as operações de criação de atividades paralelas mais eficientes.

Ao contrário do que era esperado, a limitação do número máximo de dados de entrada para cada instrução em dois, imposta pelo conjunto de instruções da MDFM, não torna as instruções tão pequenas quanto as instruções utilizadas pelas arquiteturas RISC. Como as arquiteturas baseadas no modelo a fluxo de dados puro não utilizam mecanismos que permitam a permanência dos dados dentro do processador e também, não existe uma ordenação prévia para a habilitação de instruções, a utilização de um conjunto de instruções similar ao de uma arquitetura RISC não permitiria que uma arquitetura a fluxo de dados pura executasse esse código de maneira tão eficiente quanto em uma arquitetura baseada na filosofia RISC.

A aproximação do código gerado para a MDFM com um código tipicamente CISC, como o código do VAX, demonstra uma tendência dos códigos gerados para as máquinas a fluxo de dados. Como foi descrito no capítulo 2, um dos maiores problemas identificados pelas máquinas a fluxo de dados de primeira geração é o excessivo número de fichas de dados geradas pela execução do grafo de fluxo de dados. O modelo exige que ocorra no mínimo a geração de uma ficha de dado para executar cada instrução que compõe o programa. Assim, se cada instrução for capaz de gerar um número razoável de ações, os efeitos da alta latência imposta pela sincronização dos dados de entrada de cada instrução poderão ser mascarados. Além disso, o aumento da complexidade das instruções deve fazer com que o número de instruções necessárias para a implementação dos programas seja reduzido, proporcionando a diminuição das taxas de sincronização e de comunicação requeridas pela execução do grafo de fluxo de dados.

4.4 Características do Código da Arquitetura SEED

A utilização do modelo a fluxo de dados como regra de habilitação para a execução de blocos de instrução e de instruções faz com que a utilização de um conjunto de instruções fortemente baseado no modelo a fluxo de dados, como o utilizado pela MDFM, seja a melhor opção.

Caso fosse utilizado um conjunto de instruções baseado no modelo convencional, haveria a necessidade de adaptação de instruções para a criação de tarefas paralelas. No caso de uma arquitetura a fluxo de dados, essa operação é realizada através da manipulação dos rótulos das fichas de dados. Como foi demonstrado pelos resultados da análise comparativa descrita nesse capítulo, esse tipo de operação foi bastante otimizada pelo gerador de código da MDFM, principalmente pelo uso de instruções especialmente criadas para esse fim. Portanto, como a arquitetura SEED pode ser caracterizada como uma arquitetura a fluxo de dados, ela deve utilizar-se de um conjunto de instruções fortemente ligado ao modelo.

Contudo, o código gerado pela MDFM não deve ser executado diretamente pela arquitetura SEED. Existe a necessidade de adequar o código da MDFM à arquitetura SEED impondo algumas modificações ao código:

- Os programas devem ser particionados em blocos de instruções.
- Os blocos de código devem ser acíclicos para evitar que exista a possibilidade de um bloco ficar aguardando um dado de entrada que deve ser produzido por ele próprio.
- cada bloco de código deve receber todos os seus dados de entrada por apenas uma instrução, que será denominada de instrução cabeça. Isto garante que o bloco possa ser identificado apenas pelo endereço da instrução cabeça, sendo muito mais simples também identificar quando o bloco pode ser habilitado para a execução.
- Quando um bloco recebe todos os seus dados de entrada todas as instruções que o compõe podem ser executadas, ou seja, nenhuma instrução do bloco deve necessitar de algum dado que não tenha sido introduzido pela instrução cabeça. Esta condição garante possibilidade de utilização do modelo a fluxo de dados para habilitar blocos para a execução.

- Será permitido, apenas para a instrução cabeça, ter até 4 dados de entrada. As demais instruções devem permanecer com a limitação de no máximo 2 dados de entrada, uma vez que serão instruções do conjunto de instruções da MDFM.
- quando é feito o particionamento de um programa deve-se anexar ao bloco a informação do número de instruções que formam o bloco de código, pois essa informação será útil para a determinação de conclusão de execução de cada bloco (seção 3.7.5.2);
- a identificação da instrução destino, em cada instrução de um bloco de código, deve informar através de 1 bit de controle, se o resultado da instrução deve ser enviado para uma instrução interna ou externa à partição.

O particionamento do código gerado para a MDFM, ou seja, um grafo de fluxo de dados, deve ser efetuado a partir da proposta que será descrita no próximo capítulo, onde também serão exemplificadas as modificações determinadas nessa seção.

4.5 Resumo

Como o conjunto de instruções e o sistema de geração de código utilizado por uma arquitetura pode interferir diretamente no desempenho obtido por esta, esse capítulo abordou essas duas questões para a arquitetura SEED.

Para avaliar a eficiência do sistema de geração de código da MDFM foi realizada uma análise para comparar o código da MDFM com os códigos de máquinas convencionais. O resultado dessa análise demonstrou que as otimizações efetuadas ao longo das pesquisas realizadas pelo grupo de Manchester, foram capazes de aproximar a qualidade do código da MDFM ao gerado para máquinas baseadas no modelo von Neumann. Essas otimizações necessitam que o conjunto de instruções possua instruções altamente ligadas ao modelo a fluxo de dados, tendo em vista que a principal ineficiência apontada para o código gerado da MDFM sempre esteve relacionada com a manipulação de rótulos de fichas de dados, ou seja, uma característica muito particular das arquiteturas a fluxo de dados dinâmicas por rotulação de fichas. Portanto, a arquitetura SEED deve utilizar o conjunto de instruções da MDFM.

No entanto, a arquitetura SEED necessita que o código produzido pelo gerador de código da MDFM seja particionado, e também que possua certas características definidas

por essa arquitetura. A metodologia a ser utilizada para o particionamento do código será descrita no próximo capítulo.

Capítulo 5

Particionamento de Grafos de Fluxo de Dados

Um grupo de pesquisa da Universidade da Califórnia, liderado por David Culler, construiu um sistema de compilação que gera a partir de um grafo de fluxo de dados, um código particionado em *threads*¹ e apresenta um escalonamento estático que leva em consideração a melhor utilização de registradores [Scha 91]. Para realizar uma avaliação desse sistema de compilação foi proposta uma máquina abstrata baseada em execução de *threads*, denominada TAM (*Thread Abstract Machine*) [Cull 93].

Esse sistema de compilação influenciou fortemente a proposta de particionamento de programas em blocos de código que será utilizada pela arquitetura SEED. A utilização dos conceitos propostos pelo sistema de compilação TAM, pode ser justificada por este sistema estabelecer uma relação entre grafos de fluxo de dados, que é a linguagem de máquina das arquiteturas a fluxo de dados, e a identificação de blocos de código e *threads*, seguindo a tendência das novas arquiteturas baseadas no modelo a fluxo de dados. Além disso, no sistema TAM é proposta uma metodologia sistemática para identificar partições do código, que satisfazem condições pré-estabelecidas e que garantem que as partições possam ser executadas de forma correta pela máquina abstrata TAM. Esse método de particionamento pode ser facilmente estendido para as arquiteturas a fluxo de dados dinâmicas, pela semelhança entre seus conjuntos de instruções.

Devido às características próprias da arquitetura SEED, o sistema de compilação TAM não será utilizado em sua totalidade, mas unicamente a parte desenvolvida para o parti-

¹Um *thread* é definido como sendo um trecho de código estritamente seqüencial

cionamento de código. Dessa forma, a primeira parte desse capítulo dedica-se à apresentação do sistema de compilação TAM, enfocando apenas o método de particionamento. Na segunda parte, será apresentada a proposta para que o mecanismo de particionamento utilizado em TAM possa ser utilizado dentro do contexto da arquitetura SEED.

5.1 TAM

5.1.1 Resumo Geral do Sistema TAM

O sistema de compilação e controle *multithreading*, TAM, parte de programas na forma de grafos de fluxo de dados, gerados a partir de uma linguagem paralela, Id90 [Trau 86], utilizada para a programação da MTTDA.

Em TAM um programa é composto de um conjunto de blocos de código, que tipicamente são funções de um programa texto. Cada bloco de código compreende vários *threads*. A ativação de um bloco de código envolve: a alocação de um quadro de ativação, onde são armazenadas as variáveis locais dos *threads*, o armazenamento de valores de argumentos e a habilitação de *threads* dentro do contexto do quadro de ativação. Instruções pertencentes a um *thread* podem utilizar dados de uma posição dentro do quadro de ativação ou dos registradores internos ao processador. Um quadro é dito residente quando o processador está executando *threads* relativos a ele. Um *quantum* é um conjunto de *threads* que são executados durante uma única residência.

Um *thread* é uma seqüência de instruções que não contém desvios de fluxo (*jumps*) ou pontos de interrupções, sendo que toda sincronização é realizada apenas no início do *thread*. A sincronização é feita através de uma posição do quadro de ativação, chamada de contador de entradas. Esse contador é decrementado cada vez que é executado um FORK para o *thread*, habilitando para a execução quando este chega a zero.

No sistema TAM, cada ativação é executada em um único processador e a distribuição de tarefa é feita ao nível de ativação. A passagem de argumentos pode envolver a comunicação entre processadores. Para efetuar a troca de informações é utilizado o seguinte esquema: uma instrução SEND envia uma seqüência de dados a um *thread* específico da ativação referenciada, chamado *inlet*. Esse *thread* tem como função extrair dados contidos em mensagens e armazená-los em posições do quadro de ativação. O *thread* habilitado por um *inlet* será executado quando o quadro de ativação torna-se residente.

As estruturas de dados são armazenadas de forma distribuída pelos processadores, sendo implementadas como “estruturas-I”. A operação de acesso de estruturas de dados pode envolver comunicação entre processadores, por este motivo essa operação foi implementada como uma operação *split-phase*. Esse tipo de operação permite que o processador possa continuar executando outros *threads* habilitados do quadro. Uma operação com “estruturas-I” gera uma requisição para uma determinada posição da memória distribuída, e quando a resposta é recebida, ela é tratada por um *inlet*.

O escalonamento em TAM é realizado pelo compilador. O primeiro nível de escalonamento é estático, sendo realizado através do agrupamento e ordenação de instruções em *threads*. Valores utilizados ou criados pela execução de um *thread* são armazenados em registradores internos aos processadores. O próximo nível de escalonamento é dinâmico, chamado de *quantum*.

Quando não existem mais *threads* habilitados em uma ativação, outra ativação que possua *threads* habilitados deve tornar-se residente, o que também é feito sob controle do compilador. A fila de escalonamento é armazenada dentro do quadro de ativação, e o último *thread* executado em um *quantum*, inclui código para localizar a próxima ativação.

5.1.2 Grafo Dual

Para obter um melhor particionamento, o sistema TAM transforma grafos de fluxo de dados em um outro tipo de grafo, denominado de **grafo dual**, para que além da dependência de dados, seja também explícita a dependência de controle.

Um grafo dual é um grafo direcionado e possui três tipos de arcos:

- **Dado:** $u - v$ especifica que um dado produzido no nó u irá ser consumido pelo nó v ;
- **Controle:** $u \rightarrow v$ representa que o nó u deve ser executado antes de v , e que u tem responsabilidade direta com o escalonamento de v ;
- **Dependência:** $u \rightsquigarrow v$ especifica operações de longa latência, por exemplo operações que envolvem comunicação com outros processadores. sendo assim indicam que os dois nós não podem estar no mesmo *thread*.

O controle no grafo dual pode ser representado por fichas que percorrem arcos de controle. Os nós “disparam” quando todas as fichas de controle estiverem prontas em suas entradas (com exceção do nó **merge**). Após um nó ter sua execução disparada, ele processa seus dados de entrada, colocando nos arcos de saída os resultados gerados, juntamente com as fichas de controle. Em um grafo gerado corretamente, uma ficha de controle só chega na entrada de um nó quando todos os dados de entrada estiverem prontos, isso deve ser garantido pelo compilador.

Figura 5.1: Nós Básicos do Grafo Dual

O grafo dual possui 8 tipos de nós com funções específicas, suas representações gráficas estão na figura 5.1.

- **Simples:** (instruções lógico-aritméticas) o sinal de controle faz com que os dados sejam lidos da entrada e além do resultado ser colocado na saída, é gerado um sinal de controle.
- **Join:** sincroniza dois caminhos de controle gerando apenas um sinal de controle.
- **Switch:** dirige o sinal de controle dependendo de uma variável lógica.
- **Merge:** complementa a operação realizada pelo **switch**. Esse é o único tipo de nó que não é estrito com os sinais de controle da entrada. Sua função é transferir dados de entrada, associados com entrada de controle ativa, para a saída. Um **merge**, no final da compilação é expandido por um conjunto de instruções de movimentação de dados.

- **Label:** indica que o nó adjacente precisa estar em um *thread* diferente. Em TAM, esse nó aparece logo após um **switch**, para separar os *threads*.
- **Outlet:** envia uma mensagem iniciando uma requisição. Tem efeito externo ao bloco de código.
- **Inlet:** recebe mensagem ou resposta de um acesso remoto à memória. Pode ter uma entrada de dependência e ter uma saída de controle e zero ou mais dados de saída.
- **Constante:** é utilizado para inserir constantes em tempo de compilação.

A partir desses nós básicos são formadas as operações que implementam construções básicas, como operações de entrada de literais, condicionais, *loops*, operações sobre estruturas-I, ativações de funções, etc.

5.1.3 Particionamento do Grafo Dual

O particionamento feito por TAM envolve a identificação de trechos do grafo dual que podem ser executados como *threads*. Para efetuar o particionamento é necessário verificar todas as sincronizações e as entradas de controle, que devem ocorrer apenas no início do *thread*. O número de dados de entrada de um *thread* deve ser determinado em tempo de compilação.

Uma partição será considerada correta quando satisfizer três condições:

- nenhuma saída da partição precisa ser produzida antes que todos os dados de entrada da partição estejam disponíveis;
- quando todas as entradas estiverem disponíveis, todos os nós da partição poderão ser executados;
- nenhum arco da partição conecta sua entrada.

Existem vários métodos de particionamento que garantem a construção de partições válidas, sendo que dois deles foram analisados pelo sistema TAM, são eles:

- **Particionamento Fluxo de Dados:** as partições devem iniciar com nós **join**, **inlet**, **merge** e **label**, pois esses nós são responsáveis pela sincronização. Os nós simples devem, então pertencer às partições de seus controles predecessores. O mesmo ocorre com nós **outlet** e **switch**.
- **Particionamento de Dependência:** este mecanismo agrupa nós que dependem de um conjunto de nós de entrada: **inlet**, **merge** e **label**. A dependência é caracterizada pela existência de um arco de controle unindo os nós. Esse tipo de particionamento garante a não existência de dependências cíclicas, e apresentou-se como o mais potente e de mais fácil implementação.

Após realizar um desses tipos de particionamento, pode-se efetuar a combinação das partições, visando conseguir partições maiores. O aumento do número de nós nas partições pode trazer vantagens, principalmente, quanto a diminuição da taxa de sincronizações entre as partições.

Figura 5.2: Combinações de Partições

Foram utilizados dois tipos de combinação de partições (figura 5.2):

- **Combinação Acima:** duas partições α e β podem ser combinadas se:
 - todas as entradas de β vem de α ;
 - β não contém nós **inlets**;
 - nenhum arco de saída do corpo de α está conectado com a entrada de β .

- **Combinação Abaixo:** duas partições α e β podem ser combinadas se:
 - todas as saídas de α vão para β ;
 - α não contém nós **outlets**;
 - nenhum arco de saída do corpo de α está conectado com a entrada de β .

A geração de partições para TAM é feita utilizando o particionamento por dependência, baseado no seguinte algoritmo:

- Determinar o conjunto de dependências para todos os nós;
- Colocar todos os nós com o mesmo conjunto de dependência em uma mesma partição básica;
- Aplicar iterativamente as regras de combinação (e outros tipos de otimização não descritas nesta seção) até não ser mais possível aplicá-las.

Após obter as partições básicas do programa, o próximo passo executado pelo sistema TAM é identificar *threads*. Como a arquitetura SEED não possui recursos específicos para a execução de *threads*, essa etapa de compilação não será utilizada e, portanto não será necessário realizar uma descrição sobre o mecanismo de identificação de *threads*.

5.2 Adaptação do Método de Particionamento TAM para a Arquitetura SEED

A partir dos conceitos utilizados pelo sistema de compilação TAM, será feita uma proposta de adaptação do método de particionamento de programas, para utilizá-lo na geração de blocos de código de programas implementados por instruções do conjunto de instruções da MDFM, ou seja, por grafos de fluxo de dados. O primeiro passo para estabelecer uma aproximação com o sistema de compilação TAM é gerar, a partir do grafo de fluxo de dados, o grafo dual. Para tanto, deve-se estabelecer uma relação entre as instruções da MDFM, com os nós básicos definidos no grafo dual.

A proposta de adaptação do método utilizado em TAM para a arquitetura SEED será exemplificada por um programa de integração binária, que também foi utilizado para

validação do simulador da arquitetura. Dessa forma, torna-se necessário descrever o grafo de fluxo de dados, gerado a partir desse programa, para posteriormente apresentar a proposta.

5.2.1 O Grafo de Fluxo de Dados da Integração Binária

Este item tem como objetivo explicar de uma forma geral, o grafo de fluxo de dados gerado a partir do programa de integração binária. Este programa parte de um algoritmo duplamente recursivo que divide o intervalo de integração pela metade em cada nível de recursão. A implementação em SISAL do programa de integração binária é:

```
function area(l,r,depth:integer returns integer)
  let
    mid := (l + r) / 2;
  in
    if depth <= 1 then
      (r - l) * (mid * mid - 6 * mid - 10)
    else
      area(l,mid,depth-1)+area(mid,r,depth-1)
    end if
  end let
end function
```

Aplicando o gerador de código da MDFM no programa em SISAL, obtêm-se o grafo de fluxo de dados, apresentado na figura 5.3. Para compreender o funcionamento desse grafo é necessário conhecer algumas instruções do conjunto de instruções da MDFM, que são utilizadas por este grafo (tabela 5.2.1).

As instruções de sincronização SYN são responsáveis pela entrada de valores literais definidos estaticamente. No caso do programa de integração binária, os literais seriam os parâmetros *l*, *r* e *depth*, que no grafo de fluxo de dados recebem os valores 0, 100 e 5, respectivamente.

Figura 5.3: Código da MDFM para Integração Binária

<i>opcode</i>	Nome	Funcionalidade
ADI	<i>add integers</i>	soma inteiros
SBI	<i>subtract integers</i>	subtrai o valor inteiro da direita do valor da esquerda
MLI	<i>multiply integers</i>	multiplica inteiros
DMD	<i>divide integers</i>	divide o valor da esquerda pelo valor da direita
CLI	<i>compare integers</i>	retorna o valor lógico verdade se o valor da esquerda for menor ou igual ao da direita
DUP	<i>duplicate</i>	duplica o valor da entrada
TUP	<i>tuplicate</i>	gera n cópias do valor da esquerda, sendo n o valor da direita
SYN	<i>synchronize</i>	sincroniza as entradas
BRR	<i>branch on value</i>	determina para qual saída será enviada a entrada da esquerda em função do valor lógico da entrada da direita
SWA	<i>swap activation name</i>	valor do dado de entrada é atribuído ao nome de ativação da ficha de saída
GAN	<i>generate activation name</i>	gera um nome de ativação único que deve ser usado para uma determinada instanciação do código
SAN	<i>set activation name</i>	atribui ao nome de ativação da ficha de dado de entrada da esquerda o valor da entrada da direita
PRP	<i>prepare access</i>	atribui ao nome de ativação da ficha de saída o valor de entrada da esquerda e ao valor do dado a combinação entre o antigo nome de ativação e o destino especificado pela entrada da direita
SCD	<i>set color and destination</i>	atribui ao nome de ativação da ficha da esquerda, o valor do antigo nome de ativação contido no campo de dado da ficha da entrada da direita e envia para o destino também contido no campo de dado da ficha da entrada direita
OPT	<i>output</i>	envia para a saída o resultado do programa

Tabela 5.1: Especificação das Instruções da MDFM que Foram Utilizadas pelo Programa de Integração Binária

A primeira instrução GAN (designada no grafo da figura 5.3 pelo número 3) gera a primeira instanciação da função *area*, através da atribuição de um novo nome de ativação. Os parâmetros recebem esse novo valor em seu rótulo através das instruções SAN. Para a função retornar o resultado de cada ativação, é utilizada a instrução PRP que estabelece relação entre o nome de ativação original e o gerado pelo GAN para a nova ativação da função.

A instrução CLI é utilizada para testar se o valor *depth* é menor ou igual a 1. O resultado, do tipo lógico, é enviado para a instrução TUP, que gera 4 cópias desse valor, enviando-os para a entrada da direita das instruções BRR. É importante observar que nesse grafo, existem dois tipos de instruções de desvio condicional: DBRR e LBRR. O primeiro tipo admite a existência de duas saídas independentes para a instrução, assim se o valor de entrada da direita for o valor lógico falso, o valor de entrada da esquerda vai para a saída da esquerda, caso contrário vai para a saída da direita. Já a instrução LBRR tem suas saídas conectadas, ou seja, se o valor lógico for verdadeiro nenhuma saída é ativada, e se for falso as duas recebem o mesmo valor.

Enquanto o valor de *depth* for maior que 1, são geradas duas ativações paralelas da função *area* pelas instruções GAN indicadas pelos números 4 e 5. Os parâmetros vão sendo modificados e então, recebem os novos nomes de ativação gerados.

Como já foi dito, as instruções PRP são responsáveis pelo retorno dos resultados de cada ativação. Pode-se notar nesse grafo, a existência de diferentes dados nas portas de entrada da direita nas 3 instruções PRP. A instrução número 6 é responsável pelo retorno do resultado final do programa. Seu dado de entrada D02LBY, determina que se trata de um dado do tipo destino (D), que aponta para a instrução 02 e essa instrução tem apenas 1 dado de entrada (LBY). A instrução 02, OPT, deve enviar para a saída o resultado da função. A instrução número 7, tem como dado de entrada D01LEW, ou seja, o resultado da função deve ser enviado para a porta da esquerda da instrução 01, que deve ser sincronizado com outro que alimentará a porta da direita. Esse dado virá do retorno da função, cuja instrução PRP número 8, serve de interface entre as ativações.

Todos os resultados gerados pelas instruções PRP são enviadas para a instrução SCD, que faz com que os resultados sejam, de fato, enviados para as instruções designadas pela instrução PRP; esse tipo de ligação é dita dinâmica. Esse tipo de ligação permite que

todos resultados gerados pelas ativações sejam somados, como se estes estivessem sendo retirados de uma pilha de ativações.

5.2.2 Definições Básicas

Para gerar o grafo dual a partir do grafo de fluxo de dados, as instruções da MDFM devem ser transformadas em nós básicos da seguinte forma:

- Instruções que necessitam de dois dados de entrada devem ser substituídas por um nó **join** e um nó **simples**.
- Instruções que alteram o rótulo, por exemplo as instruções **SAN**, ou são de longa latência, como **GAN**, são transformadas em nós **outlet**. Em decorrência dessa mudança, a construção do grafo dual, exige a inserção de nós **inlet** para recepção dos resultados gerados por esse tipo de nó. As instruções envolvendo estruturas de dados são de longa latência, assim devem seguir essa mesma abordagem.
- As instruções de desvio de fluxo, como por exemplo **BRR**, são identificadas como sendo nós **switch**. Para a correta construção do grafo dual, cada saída desse nó exige a presença de um nó **label**.
- Quando há possibilidade de duas instruções diferentes enviarem dados para uma mesma porta de uma instrução destino deve-se utilizar a instrução **merge**. Esse caso ocorre quando parte do grafo tem sua execução condicionada a alguma decisão determinada dinamicamente
- A entrada de constantes no programa deve ser feita pelo nó **constante**.

Além da associação de instruções da MDFM com os nós básicos do grafo dual deve-se definir que tipos de arcos irão ser gerados em cada tipo de instrução, pois um grafo dual deve apresentar os arcos de fluxo de dados, de controle e de dependência. Assim, é estabelecida a seguinte regra para geração de arcos de cada nó do grafo dual:

- Um **inlet** recebe um arco de dependência e um ou mais arcos de dados, gerando em sua saída um ou mais arcos de dados e controle. Aqui têm-se a primeira diferença de abordagem em relação àquela utilizada em TAM, pois no sistema original um **inlet**

pode gerar apenas um arco de controle, fato que pode ser explicado em função do próprio objetivo do sistema TAM que é gerar partições onde possam ser identificados *threads*. No entanto, para a arquitetura SEED deseja-se que as partições contenham partes do grafo, onde também possa ser explorado paralelismo. Assim, o **inlet** não deve limitar a possibilidade de geração de uma atividade paralela.

- O **join** mantém as características adotadas em TAM, unindo um ou mais arcos de controle na entrada e tendo como saída apenas um arco de controle.
- Pelo mesmo motivo apresentado para o nó **inlet**, os nós simples podem gerar como saída um ou dois arcos de controle e de dados, sendo que um nó simples necessita de um arco de controle e um ou dois arcos de dados em sua entrada. Observando que quando se trata de uma instrução com dois dados de entrada o controle deve ser gerado através de um nó **join**.
- O nó **label** recebe um arco de controle e pode ter vários arcos de controle gerados como saída.
- Os nós do tipo **merge** possuem dois arcos de controle na entrada e apenas um em sua saída.

A partir das associações estabelecidas entre as instruções da MDFM e os nós e arcos do grafo dual, pode-se então construir o grafo dual do programa de integração binária, como mostra a figura 5.4.

5.2.3 Utilização do Método de Particionamento

Com o grafo dual pronto já é possível aplicar os mecanismos de particionamento propostos em TAM sobre o grafo. Primeiramente, deve-se utilizar o algoritmo de particionamento por dependência. Em seguida, deve-se aplicar sucessivamente as regras de combinação acima sobre o grafo dual particionado, até obter o melhor particionamento possível do grafo. A configuração final do grafo dual de integração binária particionado é apresentado na figura 5.5.

Para gerar partições que possam ser executadas corretamente pela arquitetura SEED, deve-se aplicar algumas restrições e alterações ao grafo dual particionado, transforman-

Figura 5.5: Grafo Dual Particionado da Integração Binária

do-o de novo em um grafo de fluxo de dados, agora particionado em blocos de código. Então, deve-se efetuar a seguinte sequência de operações para efetuar essa transformação:

- **1:** Cada partição pode receber no máximo quatro arcos de dados como entrada. Para esta restrição deve-se definir o número de arcos de dados de entrada de um nó de entrada de uma partição como sendo o número de arcos de dados gerados na saída do nó.
- **2:** Deve-se inserir em cada início de partição sua respectiva instrução cabeça, onde todos os dados de entrada devem ser sincronizados. A inserção da instrução cabeça permite que todos os arcos de controle e dependência entre as partições, sejam retirados, tendo em vista que os arcos de fluxo de dados são suficientes para representar as dependências entre as partições.
- **3:** Todos os arcos de controle internos às partições do grafo podem ser retirados depois da inserção da instrução cabeça. Os nós **join** que foram inseridos para a sincronização de arcos de controle de instruções diádicas podem ser retirados, uma vez que estes perdem sua função no grafo de fluxo de dados.

Após a aplicação dessas regras, finalmente, o programa de integração binária tem seu código separados em blocos que podem ser executados pela arquitetura SEED (figura 5.6).

Figura 5.6: Código Executável do Programa de Integração Binária para a Arquitetura SEED

Capítulo 6

O Simulador da Arquitetura SEED

Visando validar e analisar a arquitetura SEED, foi construído um programa que simula todos os passos realizados pela arquitetura durante a execução de programas implementados como grafo de fluxo de dados particionados. Após o término da execução de cada programa utilizado como *benchmark*, o simulador apresenta várias informações que podem ser utilizadas para analisar o comportamento da arquitetura. Este capítulo dedica-se à descrição desse simulador e das medidas de avaliação da arquitetura apresentadas por ele.

6.1 Descrição do Simulador

Para a construção de um simulador capaz de executar programas tal qual seriam executados pela própria arquitetura SEED, foi necessário definir uma série de adaptações, visando relacionar as características da linguagem escolhida para a implementação do simulador aos mecanismos de execução definidos pela arquitetura. Assim, primeiramente, será descrito como foram empregados os recursos oferecidos pela linguagem para a implementação do simulador, e, em seguida as principais adaptações.

A linguagem escolhida para implementar o simulador foi a linguagem C++. Essa linguagem baseia-se no paradigma orientado a objeto para solucionar problemas [WiPi 91]. Esse paradigma fundamenta-se em resolver problemas através do envio de mensagens a objetos.

Como o programa construído para simular a arquitetura SEED tenta explorar os recursos oferecidos pela linguagem C++, foi definido que cada unidade que compõe a arquitetura SEED deve ser um objeto na concepção da linguagem C++, ou seja, deve ser

implementada como uma classe que possui dados privados e interage com outros objetos através de mensagens.

Por formarem classes distintas, cada unidade deve comunicar-se com outra através dos protocolos de acesso especificados para cada classe. Essa interação entre as classes simula de forma bastante próxima da realidade a comunicação entre as unidades da arquitetura SEED, pois nessa arquitetura, a comunicação entre as unidades é realizada através de mensagens com formatos pré-estabelecidos pela arquitetura. Essas mensagens são representadas no simulador pelos protocolos de acesso.

Dependendo da mensagem recebida por uma unidade da arquitetura SEED, podem ser geradas várias ações dentro da unidade. O resultado final dessas operações pode ser, inclusive, a geração de uma ou mais mensagens para outra unidade. A simulação das operações geradas por cada mensagem é efetuada pela definição de métodos, ou funções, que podem atuar sobre as mensagens recebidas pelos objetos. Portanto, utilizando essa abordagem, cada unidade que forma a arquitetura SEED pode ter seu funcionamento simulado pelo programa.

Contudo, para obter a simulação do funcionamento da arquitetura como um todo, ou seja, para que um programa possa ter sua execução simulada até a sua conclusão, deve-se definir uma estratégia para ativação das classes. Assim, determinou-se que todas as unidades da arquitetura devem ser ativadas periodicamente para se verificar se possuem mensagens em suas entradas. Quando existirem, devem ser efetuadas as devidas operações requisitadas pelo tratamento dessa mensagem. Para implementar essa estratégia foi necessário definir no simulador uma variável global, que representa o relógio global da arquitetura, e também fixar um intervalo de tempo. Cada vez que o relógio global é incrementado pelo valor do intervalo de tempo fixado, todas as unidades da arquitetura devem ser ativadas. Esse procedimento deve ser repetido até que não exista atividade em nenhuma das classes que simulam as unidades da arquitetura SEED. Através dessa estratégia cada unidade pode manipular suas mensagens de entrada toda vez que o relógio global for incrementado.

Como cada unidade deve requerer um tempo diferente para tratar cada mensagem de entrada, é importante considerar essas diferenças para que a arquitetura possa ser corretamente avaliada. A metodologia utilizada para estimar o tempo necessário para

cada unidade processar cada mensagem recebida foi a seguinte:

- Foi construído um fluxograma das possíveis ações geradas em cada uma das unidades para cada uma das possíveis mensagens recebidas.
- Considerando a utilização de tecnologia atual para implementação da arquitetura, foi estimado o tempo de cada uma das ações especificadas pelo fluxograma.
- Finalmente, foi estimado o tempo total requerido para que todas as operações efetuadas pelas unidades sejam executadas, ou seja, o tempo necessário para processar uma mensagem desde a sua entrada na unidade.

As tabelas de 6.1 até 6.7 apresentam os tempos estimados para cada unidade efetuar as respectivas operação desencadeadas por cada mensagem de entrada. A tabela 6.1 será descrita com detalhes para exemplificar como as tabelas foram montadas e organizadas. Os tempos requeridos para a Unidade de Agrupamento efetuar as suas possíveis ações em função das fichas de dados recebidas estão contidos na tabela 6.1. Se a mensagem recebida pela UA é uma ficha de dado que não necessita de ficha parceira, então a UA deve fazer com que essa ficha espere 1 incremento do relógio global do simulador para enviá-la para a Chave 1. Esse tempo foi definido como 1, pois a UA apenas realiza um teste, que em princípio pode ser efetuado com apenas uma porta lógica, e escreve a mensagem de entrada no *buffer* de saída. Definindo o intervalo de tempo a ser somado no relógio global como sendo em torno de 25ns, essas ações podem ser realizadas inteiramente dentro desse intervalo. Quando uma ficha de entrada da UA necessita de outras fichas para habilitar a execução do bloco, e estas ainda não chegaram na UA, esta unidade deve realizar as seguintes operações: verificar pelo contador da posição de memória onde as fichas devem ser sincronizadas, se todas já estão armazenadas na Memória de Dados da UA. Caso não estejam, a UA deve armazenar essa ficha de dado, atualizar o contador da posição de memória com o número de fichas que ainda devem ser sincronizadas para que todos os dados de entrada do bloco de código sejam agrupadas pela UA. Através dessa descrição pode-se observar que para efetuar todas essas ações geradas por essa mensagem, são necessários três acessos em uma memória RAM. Portanto, o tempo estimado para que essas ações possam ser concluídas determina que o relógio global seja incrementado 6 vezes para só então poder aceitar a entrada de outra mensagem.

Mensagem	Descrição da Operação	N. de Intervalos
Ficha de dado não precisa de ficha parceira	A ficha de dado é colocada no <i>buffer</i> de saída	1
Ficha de dado precisa de fichas parceiras e as encontra	É verificado o contador da posição de memória, a ficha de dado é agrupada com as outras fichas armazenadas na UA e o pacote é colocado no <i>buffer</i> de saída	4
Ficha de dado precisa de fichas parceiras e não encontra todas na UA	É verificado o contador da posição de memória, a ficha de dado é armazenada e o contador é atualizado	6

Tabela 6.1: Tabela dos Tempos Requeridos para a Unidade de Agrupamento efetuar suas Operações Possíveis

Mensagem	Descrição da Operação	N. de Intervalos
Pacote com até quatro dados de entrada para um bloco	É retirado um rótulo local da de fila de rótulos, armazena os dados na memória de dados e envia requisição do código do bloco para a URB	3
Instrução cabeça do bloco	Retira dados da memória e envia ficha com dado para UA_Local	1

Tabela 6.2: Tabela dos Tempos Requeridos para a Unidade de Entrada de Blocos efetuar suas Operações Possíveis

Mensagem	Descrição da Operação	N. de Intervalos
Rótulos global e local	Armazena informação na memória	1
Pacote executável	Lê uma posição de memória	1

Tabela 6.3: Tabela dos Tempos Requeridos para a Tabela de Rótulos efetuar suas Operações Possíveis

Mensagem	Descrição da Operação	N. de Intervalos
Ficha de dado não precisa de ficha parceira	A ficha de dado é colocada no <i>buffer</i> de saída	1
Ficha de dado precisa de ficha parceira e a encontra	É verificado o bit de presença da posição de memória, a ficha de dado é agrupada com a outra armazenada na UA e o pacote é colocado no <i>buffer</i> de saída	2
Ficha de dado precisa de ficha parceira e não a encontra na UA	É verificado o bit de presença da posição de memória, a ficha de dado é armazenada e o bit de presença é atualizado	2

Tabela 6.4: Tabela dos Tempos Requeridos para a Unidade de Agrupamento Local efetuar suas Operações Possíveis

Mensagem	Descrição da Operação	N. de Intervalos
Requisição de Bloco de Código presente na UF	É verificado que existe uma outra ativação do bloco na UF, sendo enviado mensagem de cópia de bloco para UI	2
Requisição de Bloco de Código	Verifica que o código deve ser requisitado à UI.Global e envia mensagem para a UI.Global	2
Rótulo Local cuja execução deve ser iniciada	Rótulo Local é enviado para a UEB	2
Rótulo Local do bloco cuja execução foi concluída	Notifica a memória de blocos presentes e rótulo local é enviado para UEB	1
Instrução cabeça do bloco	Armazena instrução na Memória de Instrução Cabeça e envia o número de instruções do bloco para a UI da Unidade Funcional	1
Instrução do bloco	Envia a instrução para a UI da Unidade Funcional	1

Tabela 6.5: Tabela dos Tempos Requeridos para a Unidade de Requisição de Blocos efetuar suas Operações Possíveis

Pode-se observar que para as unidades de agrupamento, existe apenas um tipo de mensagem de entrada e dependendo de certas condições testadas dentro de cada UA são geradas operações diferentes. Outras unidades podem receber mensagens de tipos completamente diferentes e dependendo dessas mensagens são efetuadas operações distintas. Por exemplo, se a Unidade de Entrada de Blocos (tabela 6.2) recebe um pacote de dados de entrada para um determinado bloco vindo da Chave 1, ela efetua três operações básicas: retira da Fila de Rótulos Disponíveis um rótulo local, armazena os dados contidos no pacote na Memória de Dados e envia para a Unidade de Requisição de Blocos uma mensagem. Para efetuar essas operações são realizados basicamente, dois acessos à memória. Assim a UEB necessita que 3 intervalos de tempo sejam somados ao relógio global do simulador. Somente após a contagem desse tempo, a UEB pode enviar a mensagem para a URB e aceitar outra mensagem, que pode ou não ser do mesmo tipo. Caso a UEB receba uma mensagem contendo uma instrução cabeça de um bloco são efetuadas as ações descritas na tabela 6.2, sendo que a cada intervalo de tempo somado ao relógio global, uma mensagem contendo uma ficha de dado pode ser enviada para a Unidade de Agrupamento Local, até que todos os dados sejam retirados da Memória de Dados.

Para estimar o tempo necessário para as chaves de interconexão (Chave 1 e Chave

Mensagem	Descrição da Operação	N. de Intervalos
Ficha com dados prontos para execução da instrução	Instrução é lida da Memória de Instrução(MI)	4
Instrução cabeça de um Bloco	A instrução cabeça é armazenada na MI	3
Instrução de um bloco	A instrução é armazenada na MI	3
Requisição de cópia de um bloco de uma região para outra da MI	uma instrução é copiada de uma posição a outra da MI	2

Tabela 6.6: Tabela dos Tempos Requeridos para a Unidade de Instrução Local efetuar suas Operações Possíveis

Mensagem	Descrição da Operação	N. de Intervalos
Requisição de código para uma UF	Uma instrução é lida e colocada na saída	3
Instrução vinda do <i>host</i> deve ser armazenada na memória	Uma instrução é armazenada na memória	2

Tabela 6.7: Tabela dos Tempos Requeridos para a Unidade de Instrução Global efetuar suas Operações Possíveis

2) transmitirem as fichas de dados é necessário fixar os números de entradas e saídas de cada uma delas. Como neste trabalho deseja-se analisar o comportamento da arquitetura SEED variando-se a quantidade de unidades de agrupamento e de unidades funcionais, optou-se em fixar o tempo de cada chave em 1 intervalo de tempo, para efetuar a análise do comportamento da arquitetura. Posteriormente, analisa-se o comportamento da arquitetura variando-se o tempo requerido pelas chaves de interconexão para transmissão das fichas. Essas análises serão apresentadas no próximo capítulo.

Por simplicidade, o tempo especificado para que a Unidade de Estrutura de Dados processe todas as possíveis instruções definidas para ela, foi fixado em 3 intervalos de tempo. Foi adotado este valor pelo fato de ter sido verificado que todas as instruções definidas para a UED sempre podem ser consideradas como sendo precisando de apenas dois acessos a uma posição na Memória de Estruturas ou na memória da sub-unidade de alocação (seção 3.6.1). Este último tipo de instrução é utilizado para a obtenção do endereço inicial onde uma nova estrutura de dados deve ser alocada na Memória de Estruturas. O tempo necessário para alocar uma nova estrutura de dados pode ser bastante diferente quando as estruturas de dados são grandes e nem sempre existem posições sub-

seqüentes para o armazenamento de seus elementos. No entanto, quando as estruturas de dados utilizadas pelos programas são pequenas comparadas com a capacidade total de armazenamento da Memória de Estruturas, a alocação de espaço de memória torna-se uma operação extremamente simples, uma vez que sempre existirão espaços na memória onde os elementos das estruturas poderão ser armazenados linearmente na memória. Portanto, quando é feita uma requisição de espaço para alocação de uma estrutura, a sub-unidade de alocação apenas deve obter da memória o endereço inicial e, em seguida armazenar a indicação que uma região, do tamanho da estrutura de dados, está ocupada para o armazenamento da nova estrutura de dados. Dessa forma, a alocação de uma estrutura de dados também pode ser tratada com o tempo de dois acessos à memória. Como será visto no capítulo seguinte, para essa primeira avaliação os programas utilizados como *benchmarks* encaixam-se nesta condição. Portanto, todas as instruções que devem ser tratadas pela UED exigem 3 intervalos de tempo para enviarem mensagem de resposta ou para aceitar uma outra instrução.

Através dessas aproximações pode-se estimar o tempo que cada unidade deve necessitar para executar as suas funções dentro da arquitetura SEED. Utilizando o simulador pode-se executar *benchmarks* para obter informações que representam o comportamento da arquitetura. A seção seguinte apresenta cada tipo de informação obtida para avaliação da arquitetura.

6.2 Medidas de Avaliação da Arquitetura Fornecidas pelo Simulador

O simulador da arquitetura SEED foi implementado para não apenas validar a arquitetura mas também para avaliá-la. Sendo assim, foram definidas algumas medidas que serão utilizadas para analisar o desempenho da arquitetura SEED. As medidas de avaliação utilizadas foram:

- **Tempo de Execução:** apresenta o tempo necessário para a execução de um programa. Como a simulação baseia-se em uma seqüência de eventos discretos separados por incrementos fixos a um relógio global, o tempo de execução é apresentado em função do número de intervalos de tempo necessários para execução do programa.

- **Ocupação Máxima das Unidades de Agrupamento:** essa medida apresenta o número máximo de fichas que foram armazenadas em cada UA durante toda a execução do programa. A elevada ocupação das unidades de agrupamento indica que as unidades funcionais estão gerando um número elevado de fichas de dados, e que estas fichas não conseguem ser tratadas pelas unidades de agrupamento com a mesma velocidade com que são geradas. Este fato demonstra que a limitação da velocidade de processamento da arquitetura SEED decorre da degradação de eficiência na operação de sincronização de dados de entrada para os blocos de código. Esse comportamento geralmente é observado em programas onde existe um grau de paralelismo elevado, e este consegue ser bem aproveitado pelos processadores.
- **Ocupação Máxima das Unidades Funcionais:** cada unidade funcional pode conter em um mesmo instante um número restrito de blocos de código em execução paralela. Essa medida apresenta o maior número de blocos em execução concorrente em uma UF durante toda a execução de um programa. Através desse parâmetro espera-se verificar se essa característica da arquitetura SEED é de fato utilizada, e se for, qual é a sua influência no desempenho da arquitetura.
- **Número de Blocos de Código Atribuídos para cada UF:** essa medida será utilizada para avaliar se a estratégia adotada para distribuição de blocos de código para cada UF é boa o suficiente para garantir uma boa distribuição de trabalho. Essa medida, no entanto, deve ser analisada junto com a máxima ocupação das unidades funcionais, uma vez que se pode atribuir um número bem maior de blocos de código a uma UF para mantê-la com o mesmo número de blocos em execução concorrente que as demais unidades.
- **Ocupação da Fila UF-Chave2:** através do número máximo de fichas que ocupam a fila de fichas que liga cada UF a sua respectiva entrada na Chave 2, pode-se analisar se a estratégia adotada para distribuição de fichas de dados para as unidades de agrupamento é capaz de garantir que as fichas geradas pelas unidades funcionais não esperem um tempo muito grande para serem enviadas para suas respectivas unidades de agrupamento. Além disso, essa medida também é útil para avaliar se a taxa de produção de fichas de dados pelas unidades funcionais pode ser tratada na

mesma proporção pelas unidades de agrupamento.

- **Ocupação dos Processadores de cada UF:** a porcentagem de tempo que o processador de cada UF permanece ocupado com a execução de instruções demonstra se o entrelaçamento de execução de vários blocos de código em cada UF é capaz de obter um bom aproveitamento dos recursos das unidades funcionais. Essa medida é obtida da seguinte forma: é somado o tempo que cada processador permaneceu ocupado durante a execução do programa e, posteriormente é verificada qual a porcentagem de tempo esse valor representa em relação ao tempo total requerido para execução do programa.

6.3 Resumo

Para a avaliação do comportamento da arquitetura SEED foi construído um simulador utilizando a linguagem C++.

A simulação do funcionamento da arquitetura como um todo, baseia-se na existência de um relógio global que é incrementado de um certo intervalo fixo de tempo. Cada vez que o relógio é incrementado todas as unidades que compõem a arquitetura verificam se possuem mensagens de entrada. Cada tipo de mensagem recebida pode desencadear operações diferentes a serem executadas em uma mesma unidade. Essas operações podem necessitar de uma quantidade de tempo distinta para serem concluídas. Por isso foi necessário avaliar, de forma mais próxima da realidade possível, o tempo requerido por cada unidade para cada operação que pode ser efetuada por ela.

Através desse sistema de simulação, pode-se executar programas, utilizados como *benchmarks*. Para analisar o comportamento da arquitetura SEED durante a execução o simulador apresenta algumas medidas de avaliação. Será através dessas que, no próximo capítulo, serão apresentados os resultados obtidos pela simulação de *benchmarks* em diversas configurações da arquitetura.

Capítulo 7

Avaliação da Arquitetura SEED

Como foi descrito no capítulo 3, a arquitetura SEED possui certos recursos que possibilitam que esta arquitetura explore paralelismo. Dependendo da quantidade desses recursos existe a possibilidade da arquitetura SEED aproveitar mais ou menos o paralelismo existente nos programas por ela executados. Dessa forma, a análise do funcionamento da arquitetura SEED foi realizada através da variação da quantidade de certas unidades que a compõe, simulando a execução de quatro *benchmarks*. Essa análise foi realizada em três etapas, nas quais foram estudados o comportamento da arquitetura com a variação do número de: unidades funcionais, rótulos locais em cada unidade funcional e unidades de agrupamento.

Os programas utilizados como *benchmarks* foram multiplicação de matrizes, criação de matrizes, cálculo do fatorial e integração binária. Uma descrição geral desses programas será apresentada nesse capítulo, com exceção do programa de integração binária que já foi abordado no capítulo 5. Em seguida, serão apresentados os resultados obtidos pelo simulador durante a execução desses *benchmarks* e, posteriormente a análise destes resultados.

Sendo o número de fichas de dados geradas durante a execução de programas por máquinas a fluxo de dados muito elevado, a velocidade das chaves de interconexão utilizadas por essas arquiteturas é de fundamental importância para o bom desempenho destas máquinas. Assim, neste capítulo também será avaliado o comportamento da arquitetura SEED em função da variação do tempo de transmissão de cada chave de interconexão utilizada pela arquitetura, ou seja, as chaves 1 e 2.

7.1 Descrição Geral dos Benchmarks

7.1.1 Multiplicação de Matrizes

O programa SISAL que implementa a multiplicação de matrizes é apresentado na figura 7.1. Trata-se de um algoritmo com alto grau de paralelismo, pois é implementado por três construções do tipo *forall* aninhadas.

O código de máquina gerado pelo sistema de compilação da Máquina de Fluxo de Dados de Manchester implementa a geração de atividades paralelas através de instruções que geram novos rótulos através do incremento do valor do campo de índice (instruções *proliferate*) e pela alteração do campo de nome de ativação.

Para realização da avaliação da arquitetura SEED foram utilizadas duas matrizes quadradas com dimensão 20x20.

```
function matmult(A,B: Matrix; N1, N2,M: integer returns Matrix)

for I in 1,N1
  Row :=
  for J in 1, N2
    El :=
    for K in 1,M
      R := A[I,K] * B[K,J]
      returns value of sum R
    end for
    returns array of El
  end for
  returns array of Row
end for
end function
```

Figura 7.1: Programa SISAL de Multiplicação de Matrizes

7.1.2 Criação de Matrizes

Esse *benchmark* realiza a criação de duas matrizes paralelamente e lê um elemento de cada matriz. A função básica para criação de uma matriz implementada com a linguagem SISAL é apresentada na figura 7.2. Esse programa apresenta um grau de paralelismo razoável, uma vez que é implementado através de duas construções do tipo *forall*. No

entanto, grande parte das atividades geradas devem ser processadas pela Unidade de Estruturas de Dados, mais especificamente pela Memória de Estruturas.

Para gerar uma quantidade razoável de atividades são criadas duas matrizes quadradas de dimensão 1200x1200.

```
function cria_mat(A,B: Matrix; N: integer returns Matrix)

  for I in 1,N
    D :=
      for J in 1,N
        C := 12
        returns array of C
      end for
    returns array of D
  end for
end function
```

Figura 7.2: Função SISAL para a Criação de Matrizes

7.1.3 Fatorial

O programa para cálculo do fatorial (figura 7.3) que foi utilizado como *benchmark*, é implementado por um algoritmo duplamente recursivo. Dessa forma, no código de máquina gerado para a MDFM, as atividades paralelas são geradas através da atribuição de um novo contexto às fichas de dados geradas por cada nova ativação da função. Essa mudança de contexto é implementada pela alteração do campo de nome de ativação.

```
function fact( x,y:integer returns integer)
  if x<=y then y
    else fact(x,(x+y)/2 +1) + fact((x+y)/2,y)
  end if
end function
```

Figura 7.3: Programa SISAL de Fatorial

Para que uma quantidade razoável de atividades possa ser efetuada pela arquitetura SEED para uma avaliação mais completa de seu comportamento, o programa do fatorial

foi alterado apresentando como resultado a soma dos número e não a multiplicação. Assim, foi utilizado para simulação o programa que resulta o “somatorial” de 400.

7.2 Avaliação da Arquitetura SEED

O aumento da quantidade disponível de certas unidades da arquitetura SEED pode melhorar sua eficiência na exploração de paralelismo. Visando encontrar uma configuração que utilize bem os recursos existentes na arquitetura SEED e também, analisar o comportamento geral da arquitetura, tentando identificar seus pontos positivos e negativos, foi realizada uma série de simulações.

Foi analisado o comportamento da arquitetura SEED em função da variação do número de unidades funcionais, unidades de agrupamento e do número de rótulos locais existentes em cada unidade funcional. Para cada uma dessas análises foram simuladas as execuções dos quatro *benchmarks* descritos na seção anterior, obtendo os valores das medidas de avaliação descritas no capítulo 6.

A metodologia utilizada para cada uma dessas avaliações e os resultados obtidos serão apresentados nas seções seguintes.

7.2.1 Variação do Número de Unidades Funcionais

7.2.1.1 Descrição Preliminar

Para analisar o comportamento da arquitetura SEED em função do número de unidades funcionais foram fixados os números de unidades de agrupamento e rótulos locais disponíveis em cada UF. Foi escolhida a seguinte configuração: 8 unidades de agrupamento e 32 rótulos locais.

Após fixar o número de unidades de agrupamento a ser utilizado na simulação, foi necessário definir qual a política adotada para atribuição, de forma unívoca, das fichas de dados que chegam na Chave 2 para uma determinada unidade de agrupamento. A escolha da configuração da arquitetura SEED com 8 unidades de agrupamento faz com que estas unidades sejam identificadas por endereços formados por 3 bits. Assim, optou-se pela utilização de 1 bit de cada campo de rótulo e 1 bit do campo de destino das fichas de dados para formar o endereço da UA que deve receber a ficha. Como o bit menos significativo de cada campo deve sofrer um número maior de alterações durante toda a

execução de programas, a utilização do bit menos significativo dos campos de destino, nome de ativação e índice da ficha de dado deve fazer com que as fichas de dados possam ser bem distribuídas entre as unidades de agrupamento. Portanto, espera-se que a adoção desse mecanismo de distribuição das fichas de dados para as unidades de agrupamento consiga fazer com que a quantidade de fichas tratadas por cada UA seja praticamente a mesma.

Como foi descrito no capítulo anterior, a análise da arquitetura SEED será realizada através das seguintes medidas: tempo de execução, ocupação máxima das unidades de agrupamento e da fila que liga cada unidade funcional à sua respectiva entrada na Chave 2, número de blocos atribuídos a cada UF, o número máximo de blocos que foram executados paralelamente em uma UF e finalmente, a porcentagem do tempo que, em média, os processadores das unidades funcionais permaneceram ativos.

Nesta etapa da avaliação da arquitetura SEED foram realizadas simulações com os quatro *benchmarks* descritos na seção anterior, variando-se o número de unidades funcionais de 2 até 20. Foi gerada uma série de gráficos para cada *benchmark*, onde podem ser observadas as variações de cada parâmetro de avaliação da arquitetura SEED em função do número e unidades funcionais. As figuras 7.4, 7.5, 7.6 e 7.7 apresentam estes gráficos. Cada figura apresenta os gráficos de todos os parâmetros de avaliação para um mesmo *benchmark*. Assim, a figura 7.4 apresenta todos os resultados obtidos para a multiplicação de matrizes, a figura 7.5 para criação de matrizes, a 7.6 para o fatorial e a 7.7 para a integração binária.

Dentro de cada figura, cada gráfico será referenciado por letras que devem identificar o mesmo tipo de gráfico para todos os *benchmarks*. Por exemplo, os gráficos de tempo de execução pelo número de unidades funcionais das figuras de 7.4 até 7.7 são identificados pela letra A, e de ocupação máxima de cada UF pela letra D.

7.2.1.2 Discussão dos Resultados

Os gráficos referenciados pela letra C das figuras de 7.4 até 7.7 apresentam o número de blocos atribuídos à cada UF em função do número de unidades funcionais. Pode-se observar que a política utilizada para distribuição de trabalho para as unidades funcionais faz com que um número maior de blocos sejam executados nas unidades funcionais com números de identificação mais baixos. Esse comportamento pode ser explicado pela es-

Figura 7.4: Variação do Número de Unidades Funcionais - Multiplicação de Matrizes

Figura 7.5: Variação do Número de Unidades Funcionais - Criação de Matrizes

Figura 7.6: Variação do Número de Unidades Funcionais - Fatorial

Figura 7.7: Variação do Número de Unidades Funcionais - Integração Binária

estratégia de distribuição de blocos às unidades funcionais. Através dessa estratégia, se em um determinado instante, existir apenas um bloco a ser atribuído para uma UF e se existirem mais do que uma UF apresentando a mais baixa ocupação, o bloco deve ser enviado para a UF com o menor número de identificação.

Os gráficos referenciados pela letra D mostram a máxima ocupação de cada UF durante toda a execução de cada *benchmark*. Esses gráficos demonstram que o programa de multiplicação de matrizes possui grau de paralelismo suficiente para manter, em algum instante, a máxima ocupação das unidades funcionais mesmo quando a arquitetura possui sua maior configuração com 20 unidades funcionais. Isto significa que as unidades funcionais chegam a possuir 32 blocos de código executando concorrentemente dentro de cada UF. Os programas de integração binária e fatorial conseguem manter a máxima ocupação apenas quando existe um número menor que 8 unidades funcionais. Finalmente, o programa de criação de matrizes é o que menos oferece paralelismo de tarefas a serem executadas pelas unidades funcionais em relação aos outros *benchmarks*, pois com apenas 4 unidades funcionais a ocupação não chega a ser máxima.

Através dos gráficos do tempo de execução pelo número de unidades funcionais (identificados pela letra A), pode-se observar que todos os *benchmarks* apresentam como característica comum a diminuição do tempo de execução até 8 unidades funcionais. A partir desse ponto, pode-se dividir o comportamento da arquitetura em dois grupos:

- Os programas de multiplicação e criação de matrizes sofrem uma pequena diminuição do tempo de execução com o aumento do número de unidades funcionais;
- Para os programas de integração binária e fatorial o tempo de execução chega a aumentar quando existe um número superior a 8 unidades funcionais.

É importante observar que o mesmo comportamento existe para os programas cuja forma de geração de paralelismo é a mesma. Os programas de integração binária e fatorial são implementados através de algoritmos puramente recursivos. Esse tipo de algoritmo impõe a necessidade de ativar e executar pelo menos um bloco de código inteiro para a geração de mais uma atividade paralela, ou seja, de uma ativação da função recursiva. Por outro lado, os programas de multiplicação e criação de matrizes, que apresentam o primeiro tipo de comportamento, geram atividades paralelas pela geração de várias fichas

de dados através de instruções do tipo *proliferate*. Uma instrução do tipo *proliferate* pode gerar rapidamente várias fichas de dados, incrementando apenas o campo de índice. Portanto, esse tipo de programa pode gerar várias atividades paralelas através da execução de uma única instrução.

Uma possível explicação para que os programas implementados por algoritmos puramente recursivos, como fatorial e integração binária, apresentem desempenho pior com com maior número de unidades funcionais, está relacionada com o aumento do número de requisições de código concorrentes que devem ser tratadas pela Unidade de Instrução Global (UI_Global). Pela política adotada para o atendimento de requisição de código, a UI_Global sempre percorre as unidades funcionais do menor para o maior número de identificação. Assim, os blocos atribuídos às unidades funcionais com números de identificação mais altos podem esperar um período de tempo consideravelmente maior em relação aos blocos enviados para unidades funcionais com números de identificação mais baixos. Dessa forma, o aumento do tempo de execução pode ser consequência da atribuição de blocos que geram atividades paralelas às unidades funcionais com números de identificação altos, fazendo com que a demora no início da execução destes ocasione uma diminuição considerável de tarefas paralelas durante esse período de espera.

Essa possível explicação pode ser confirmada pelos gráficos de ocupação das unidades de agrupamento. Quando esse parâmetro é comparado conjuntamente ao gráfico de tempo de execução, pode-se verificar que a ocupação máxima das unidades de agrupamento é inversamente proporcional ao tempo de execução, ou seja, a ocupação das unidades de agrupamento aumenta quando o tempo de execução diminui. Isso demonstra que enquanto é possível aproveitar satisfatoriamente as unidades funcionais há um aumento do número de atividades paralelas, implicando em um aumento do número de fichas que devem ser tratadas pelas unidades de agrupamento paralelamente. Como as unidades de agrupamento não conseguem ser suficientemente rápidas, um maior número de fichas de dados deve esperar um tempo maior para serem sincronizadas e agrupadas com sua ou suas fichas parceiras. Dessa forma, a operação de sincronização de fichas de dados impõe um limite à velocidade de processamento. Para o programa de multiplicação de matrizes o nível de paralelismo gerado pelo programa consegue ser bem aproveitado pela arquitetura SEED até de 8 unidades funcionais, quando também a ocupação máxima das unidades de

agrupamento aumenta. De 8 à 20 unidades funcionais a ocupação máxima das unidades de agrupamento permanece constantemente em um nível alto, indicando que estas devem estar no limite da velocidade com que podem tratar as fichas de dados, implicando em uma limitação da velocidade de processamento. Já para os programas recursivos pode-se notar que quando o tempo de execução começa a aumentar, a ocupação das unidades de agrupamento diminui depois de ter chegado ao seu nível máximo. Assim, fica claro que há um decréscimo do nível de paralelismo que pode ser explorado pela arquitetura SEED, provavelmente em função das razões descritas.

Uma característica comum a todos os *benchmarks* é a grande diminuição da porcentagem do tempo de ocupação dos processadores quando o número de unidades funcionais aumenta. Quando existem 2 unidades funcionais os processadores de cada UF permanecem ocupados aproximadamente 80% do tempo de execução, e apenas 20% para 20 unidades funcionais. Assim, o aproveitamento dos recursos internos às unidades funcionais tende a ser reduzido quando existe um número alto destas.

7.2.2 Variação do Número de Rótulos Locais

7.2.2.1 Descrição Preliminar

Através da variação do número de rótulos locais existentes em cada UF pode-se analisar se, com o possível aumento do número de blocos em execução paralela dentro de cada UF, os seus recursos internos podem ser melhor aproveitados. Além disso, em virtude da busca do código em uma unidade remota à UF, deve existir um tempo de espera consideravelmente alto entre a atribuição de um bloco a uma UF e seu real início de execução. Assim, variando-se o número de rótulo locais dentro das unidades funcionais, pode-se observar se com uma maior possibilidade de sobreposição da execução dos blocos de código o tempo de execução pode ser reduzido.

Para essa análise, foi mantida a seguinte configuração da arquitetura SEED: 8 unidades funcionais e 8 unidades de agrupamento. A forma de distribuição de fichas de dados para as unidades de agrupamento é realizada como na análise anterior.

Os números de rótulos locais simulados foram 4, 8, 16, 32, 64 e 128. As figuras 7.8, 7.9, 7.10 e 7.11 apresentam os gráficos obtidos pelos resultados dessa análise. Estas figuras possuem a mesma estrutura geral das figuras geradas pela avaliação anterior.

Figura 7.8: Variação do Número de Rótulos Locais - Multiplicação de Matrizes

Figura 7.9: Variação do Número de Rótulos Locais - Criação de Matrizes

Figura 7.10: Variação do Número de Rótulos Locais - Fatorial

Figura 7.11: Variação do Número de Rótulos Locais - Integração Binária

7.2.2.2 Discussão dos Resultados

Pelas figuras 7.8, 7.9, 7.10 e 7.11 pode-se observar que gráficos obtidos com os *benchmarks* multiplicação de matrizes, fatorial e integração binária possuem de uma forma geral, comportamento muito similar. Já o comportamento do programa de criação de matrizes difere muito dos demais. Dessa forma, a descrição e discussão dos resultados será dividida em função do tipo de comportamento observado pela arquitetura SEED.

Os gráficos de tempo de execução pelo número de rótulos locais (letra A), demonstram que para todos os *benchmarks*, com exceção do programa de criação de matrizes, o desempenho da arquitetura SEED melhora consideravelmente com o aumento da exploração de paralelismo das atividades internas às unidades funcionais. Contudo, a redução do tempo de execução quase não existe para valores acima de 32 rótulos locais.

Pode-se observar pelos gráficos identificados pela letra F que, no mínimo, a porcentagem de tempo que os processadores permanecem ativos dobra quando o número de rótulos locais passa de 4 para 32. Esse resultado confirma o melhor desempenho da arquitetura com o aumento do número de rótulos locais. Mas, assim como no gráfico do tempo de execução, verifica-se que para uma quantidade maior que 32 rótulos locais, a porcentagem do tempo de ocupação dos processadores praticamente não sofre alteração.

Os gráficos de ocupação máxima das unidades de agrupamento e da fila UF-Chave2 respectivamente identificados pelas letras C e E das figuras 7.8, 7.10 e 7.11, apresentam comportamento similar ao obtido pela avaliação do comportamento da arquitetura pela variação do número de unidades funcionais. A partir do ponto onde o aproveitamento dos processadores e o tempo de execução permanece praticamente constante, é verificado que a ocupação das filas que ligam as unidades funcionais à Chave 2 e das unidades de agrupamento aumentam. Esse comportamento demonstra que um número maior de fichas são geradas pelas unidades funcionais, e as unidades de agrupamento não conseguem tratá-las e enviar os pacotes de dados tão rapidamente, impondo uma limitação à velocidade de processamento.

Apesar do comportamento observado para o programa de multiplicação de matrizes ser similar ao obtido para os *benchmarks* fatorial e integração binária, esses últimos não apresentam quantidade de paralelismo suficiente para atingir a máxima ocupação das unidades funcionais quando o número de rótulos locais é superior à 8 (gráficos referencia-

dos pela letra D). Já o programa de multiplicação de matrizes consegue atingir a máxima ocupação até mesmo quando existem 128 rótulos locais nas unidades funcionais.

O comportamento apresentado pelos gráficos construídos para o programa de criação de matrizes (figura 7.9) difere muito dos demais. O aumento do número de rótulos locais não melhora consideravelmente o desempenho da arquitetura. O tempo de execução diminui em apenas 3% variando-se o número de rótulos locais de 4 para 8; após essa diminuição mínima o tempo de execução sofre um aumento de cerca de 0.5%.

A média do tempo de ocupação dos processadores para a criação de matrizes varia de 37.9% à 38.7% do tempo total gasto para execução do programa. Esse comportamento pode ser atribuído ao tipo de atividade gerado pelo programa, ou seja, como uma parcela significativa do programa deve ser tratada pela Unidade de Estruturas de Dados, o aumento dos recursos internos às unidades funcionais não consegue ser plenamente aproveitado para execução do *benchmark*.

Outros resultados que comprovam o baixo aproveitamento dos recursos da arquitetura para execução do programa de criação de matrizes são as ocupações máximas das unidades de agrupamento e das unidades funcionais. Pelo gráfico de ocupação máxima das unidades de agrupamento, pode-se notar que esta ocupação permanece constante e igual a 1200 durante toda a execução do programa, que cria duas matrizes de dimensão 1200x1200, para todas as configurações simuladas. Além disso, as filas de fichas que ligam as unidades funcionais à Chave 2 não são utilizadas. A ocupação máxima das unidades funcionais é mantida em seu nível máximo até 16 rótulos locais. Após esse valor, a ocupação máxima continua em torno de 16.

7.2.3 Variação do Número de Unidades de Agrupamento

7.2.3.1 Descrição Preliminar

O aumento do número de unidades de agrupamento leva a duas consequências diretas: ao aumento do número de saídas da Chave 2 e a um maior paralelismo na operação de sincronização efetuada por essas unidades. Desconsiderando o possível aumento de complexidade de implementação da Chave 2, a primeira característica deve fazer com que exista uma menor concorrência na transmissão de fichas para um mesmo canal. Assim, o envio de fichas de dados das unidades funcionais para as unidades de agrupamento deve ser

efetuado mais rapidamente. Contudo, essa característica fica diretamente condicionada à política adotada pela Chave 2 para a distribuição de fichas de dados entre as unidades de agrupamento existentes.

Na presente avaliação não é desejado determinar a melhor forma de distribuir fichas de dados para as unidades de agrupamento, mas a partir de uma determinada política de distribuição, deseja-se analisar como se comporta a arquitetura SEED e se, de fato, o aumento do número de unidades de agrupamento leva a um melhor desempenho.

O princípio utilizado para distribuição de fichas pela Chave 2 é uma extensão daquela utilizada para as avaliações anteriores, onde o número de unidades de agrupamento era fixado em 8. Serão avaliadas agora, configurações possuindo um número igual e acima de 8 unidades de agrupamento. Mais especificamente serão analisadas as configurações com 8, 16, 32 e 64 unidades de agrupamento. Assim a identificação das unidades deverão ser endereços formados por 3, 4, 5 e 6 bits, respectivamente. A tabela 7.1 apresenta como serão formados os endereços das unidades de agrupamento em função do número de unidades existentes.

N. de Unidades de Agrupamento	Endereços Formados
8	O bit menos significativo dos campos de destino, nome de ativação e índice de ficha de dado
16	Os 2 bits menos significativos do campo de destino e o menos significativo do nome de ativação e índice
32	Os 2 bits menos significativos dos campo de destino e nome de ativação e o menos significativo do índice
64	Os 2 bits menos significativos dos campo de destino nome de ativação e índice

Tabela 7.1: Política de Distribuição de Fichas de Dados pela Chave 2

A ordem de formação de endereços das unidades de agrupamento com os bits especificados na tabela 7.1 é representada na figura 7.12. Mesmo quando um campo fornece mais que um bit para formação do endereço a ordem é mantida.

A configuração adotada para a arquitetura SEED foi: 4 unidades funcionais e 32 rótulos locais em cada UF. Essa configuração foi escolhida em função das avaliações anteriores, nas quais observou-se que todos os *benchmarks* apresentam máxima ocupação das

índice	nome de ativação	destino
--------	------------------	---------

Figura 7.12: Ordem para Formação do Endereço da Unidade de Agrupamento

unidades funcionais.

7.2.3.2 Discussão dos Resultados

As figuras 7.14, 7.15 e 7.13 apresentam os resultados obtidos pela avaliação do comportamento da arquitetura SEED pela variação do número de unidades de agrupamento. Como o programa de criação de matrizes não sofreu alteração em nenhum dos parâmetros de avaliação utilizados pelo simulador, não são apresentados os gráficos referentes a esse *benchmark*.

Os gráficos de tempo de execução demonstram que o aumento do número de unidades de agrupamento praticamente não interfere na velocidade de processamento da arquitetura SEED. Além disso, os gráficos da porcentagem do tempo de ocupação dos processadores demonstram ser inversamente proporcional aos gráficos de tempo de execução. Para o programa de integração binária (figura 7.15) pode-se observar que, variando-se o número de unidades de agrupamento de 8 para 32, há uma diminuição do tempo de execução em torno de 0.3%, enquanto a ocupação dos processadores aumenta 0.1%. De 32 para 64 unidades de agrupamento o tempo de execução e a ocupação dos processadores não são alterados.

O programa fatorial (figura 7.14) apresenta comportamento um pouco distinto do observado pelo programa de integração binária. Com o aumento do número de unidades de agrupamento existe um pequeno aumento no tempo de execução, cerca de 0.2%. Já a diminuição da porcentagem de tempo de ocupação dos processadores observada fica nessa mesma proporção. Para ambos os *benchmarks* quando o número de unidades de agrupamento passa de 8 para 32, pode-se observar que existe um aumento da ocupação máxima das unidades de agrupamento de cerca de 20%. Isso demonstra que embora as unidades de agrupamento possam tratar um número maior de fichas paralelamente, o número de unidades funcionais torna-se o “gargalo” da arquitetura, pois se os pacotes de dados de entrada de blocos não conseguem ser enviados para as unidades funcionais, as

Figura 7.13: Variação do Número de Unidades de Agrupamento - Multiplicação de Matrizes

Figura 7.14: Variação do Número de Unidades de Agrupamento - Fatorial

Figura 7.15: Variação do Número de Unidades de Agrupamento - Integração Binária

unidades de agrupamento terão mais fichas armazenadas em sua memória.

Apesar dos programas integração binária e fatorial serem implementados por algoritmos puramente recursivos, o comportamento da arquitetura SEED é bastante diferente entre ambos, principalmente no que se refere aos parâmetros apresentados no parágrafo anterior. Na realidade, esperava-se que ambos tivessem o comportamento apresentado pelo programa de integração binária, pois para formar os endereços das unidades de agrupamento com o aumento destas, a Chave 2 considera primeiro mais um bit do campo de destino e, posteriormente mais um bit do campo de nome de ativação. Como os programas recursivos utilizam o campo de nome de ativação para gerar atividades paralelas, era esperado que para estes programas existisse um menor número de conflitos na distribuição de fichas de dados entre as unidades de agrupamento. Dessa forma, o tempo de execução deveria diminuir, tal qual acontece com o programa de integração binária.

O programa de multiplicação de matrizes apresenta diminuição de aproximadamente 1.5% do tempo de processamento com 16 unidades de agrupamento em relação ao tempo observado com 8 unidades. Não existe alteração do tempo de execução quando é aumentado o número de unidades de agrupamento de 16 para 32 e o tempo diminui mais 0.5% aumentando-se para 64 unidades. Observando a porcentagem do tempo de ocupação dos processadores verifica-se que há um aumento de aproximadamente 13% quando o número de unidades de agrupamento passa de 8 para 16 e permanece constante com a inserção de mais unidades.

Para melhor monitoramento da ocupação das unidades de agrupamento foi inserido o gráfico da ocupação máxima de todas as unidades nas figuras 7.14, 7.15 e 7.13, referenciado pela letra G. Pode-se observar através desses gráficos que, para todos os *benchmarks*, a ocupação máxima das unidades de agrupamento aumenta com o aumento do número de unidades de agrupamento. Como existe uma maior vazão das fichas oriundas das unidades funcionais pela Chave 2, um maior número de fichas podem ser tratadas paralelamente pelas unidades de agrupamento. Contudo, a transmissão dos pacotes de dados da Chave 1 para as unidades funcionais tem sua capacidade limitada pelo número fixo de unidades funcionais e pela capacidade destas unidades de sobrepor a execução do maior número possível de blocos de código. Portanto, era esperado que a ocupação máxima de todas as unidades de agrupamento aumentasse com o crescimento do número

destas.

Em relação à distribuição de fichas para as unidades de agrupamento pode-se observar que os algoritmos puramente recursivos não aproveitam bem o aumento do número destas (gráficos identificados pela letra B). Isso ocorre em função da política de distribuição adotada para a Chave 2. Como os algoritmos recursivos não alteram o campo de índice, os endereços das unidades de agrupamento que necessitam dos bits mais significativos do endereço para serem especificadas não são utilizadas, pois dependem unicamente do campo de índice. Para analisar o quanto as unidades de agrupamento são desperdiçadas, basta comparar com os resultados obtidos pelo programa de multiplicação de matrizes, que durante a execução impõe a alteração dos campos de índice e nome de ativação. Pode-se verificar que todas as unidades de agrupamento recebem fichas de dados, enquanto que os programas recursivos conseguem com essa política de distribuição de fichas ocupar apenas metade do número de unidades de agrupamento.

7.3 Comportamento da Arquitetura com a Variação do Tempo de Transmissão das Redes de Interconexão

Como foi descrito no capítulo 2, um dos maiores problemas apresentados pelas arquiteturas baseadas no modelo a fluxo de dados é a alta taxa de comunicação imposta pelo modelo. Esse modelo exige que seja realizada, pelo menos, uma comunicação de dado para cada instrução atribuída a um processador. Visando diminuir o tráfego de fichas que deve circular pela arquitetura, foi proposto para a arquitetura SEED o aumento da granularidade das tarefas a serem atribuídas às unidades funcionais. Assim, a separação do grafo de fluxo de dados em blocos de instruções deve fazer com que o tráfego de fichas de dados seja reduzido apenas a transmissão de dados entre os blocos de código.

Apesar desse aumento de granularidade, o tráfego de fichas nas chaves de interconexão que compõem a arquitetura SEED ainda deve ser alto, principalmente se comparado aos computadores paralelos baseados em outros modelos computacionais. Dessa forma, para que o desempenho da arquitetura não seja comprometido é importante que as velocidades das chaves de interconexão sejam suficientemente altas para que estas unidades não se tornem o “gargalo” da arquitetura. Além disso, se realmente o tráfego de fichas con-

segue ser bastante reduzido pelo aumento de granularidade, a arquitetura deve ser menos sensível aos possíveis atrasos impostos pelas chaves de interconexão.

Nessa seção é apresentada uma avaliação do comportamento da arquitetura SEED em função da variação do tempo de transmissão de cada uma das chaves de interconexão. Para efetuar essa análise a configuração da arquitetura foi mantida com 8 unidades de agrupamento e 8 unidades funcionais, sendo que cada UF pode executar no máximo 32 blocos de código concorrentemente. O *benchmark* escolhido foi o programa de multiplicação de matrizes, uma vez que esse programa possui grau de paralelismo elevado, além de utilizar a Unidade de Estruturas de Dados (UED).

Esse estudo será dividido em duas etapas: na primeira o tempo necessário para transmissão de fichas de dados pela Chave 2 será mantido fixo enquanto o tempo referente à Chave 1 deve variar. E na etapa seguinte, o tempo da Chave 1 é mantido fixo e o da Chave 2 sofre variação. Os resultados obtidos através desse estudo serão apresentados nas seções seguintes.

7.3.1 Variação do Tempo de Transmissão da Chave 1

A figura 7.16 apresenta os gráficos dos parâmetros de avaliação da arquitetura, em função da variação do tempo de latência imposto pela Chave 1 às fichas de dados que devem ser transmitidas por ela. O tempo de atraso imposto para cada ficha de dado transmitida variou de 1 até 10 intervalos de tempo.

O aumento do tempo de latência da Chave 1 faz com que exista um aumento no tempo de execução. Comparando os gráficos A e E pode-se observar que o aumento do tempo de execução é inversamente proporcional a porcentagem do tempo de ocupação dos processadores. Além disso, os demais gráficos demonstram que os outros parâmetros não apresentam praticamente alteração. Esse comportamento comprova que a consequência mais direta do aumento do tempo de latência da Chave 1 é o atraso da atribuição de blocos de código para a execução nas unidades funcionais. Esse atraso acarreta um menor número de atividades paralelas internas às unidades funcionais. No entanto, essa diminuição não é suficiente para diminuir consideravelmente a ocupação de memória das unidades de agrupamento e da fila que liga cada UF à Chave 2, e nem altera a ocupação da fila que liga a Unidade de Estrutura de Dados à Chave 2.

Através desses resultados pode-se concluir que o tráfego de fichas que são tratadas pela Chave 1 não é elevado, pois apesar do tempo de latência sofrer um aumento de 10 vezes do seu valor inicial, o aumento do tempo de execução é de aproximadamente 20%. Portanto, o tráfego de fichas na Chave 1 consegue ser bastante amenizado pela separação do grafo de fluxo de dados em blocos de código.

Figura 7.16: Variação do Tempo de Transmissão da Chave 1 (Multiplicação de Matrizes)

7.3.2 Variação do Tempo de Transmissão da Chave 2

A figura 7.17 apresenta os resultados dos parâmetros de avaliação da arquitetura obtidos através da variação do tempo de atraso imposto pela Chave 2. O gráfico de ocupação máxima das unidades funcionais não é apresentado nessa figura pois, não sofreu alteração dentro do intervalo escolhido para variação do tempo de atraso da Chave 2. Para esse intervalo cada UF executou paralelamente, em algum instante, 32 blocos de código, ou seja, foi observada a máxima ocupação possível das unidades funcionais.

Figura 7.17: Variação do Tempo de Transmissão da Chave 2(Multiplicação de Matrizes)

O gráfico A demonstra que o tempo de execução sofre um aumento de aproximadamente 90% quando o tempo de atraso da Chave 2 varia de 1 para 10 intervalos de tempo. Já a porcentagem de tempo de ocupação dos processadores, neste mesmo intervalo, varia em cerca de 33%. Essa diminuição da porcentagem do tempo de ocupação dos processadores é refletida também na ocupação máxima das filas UF-Chave2, que apesar do aumento do tempo de atraso da Chave 2, sofre uma queda quando o tempo de atraso da Chave 2 cresce de 2 para 10 intervalos de tempo. Através desses resultados pode-se dizer que o aumento do tempo de atraso da Chave 2 tem como consequência uma redução do aproveitamento das unidades funcionais.

Através do gráfico F pode-se observar que existe um aumento da ocupação máxima da fila UED-Chave2 quando o tempo de atraso da Chave 2 aumenta de 1 até 4 intervalos de tempo. A partir desse valor, a ocupação máxima não sofre uma alteração considerável. Quando o gráfico F é analisado conjuntamente com a ocupação máxima das filas UF-Chave2, fica claro que com o atraso de até 4 intervalos de tempo as unidades funcionais conseguem processar os blocos de código com uma eficiência razoável, pois a porcentagem do tempo de ocupação dos processadores é ainda superior à 50%. A partir deste ponto, o baixo aproveitamento das unidades funcionais faz com que um menor número de fichas seja produzido paralelamente, a ponto de reduzir a ocupação máxima das filas UF-Chave2. Como a UED depende das unidades funcionais para receber instruções para sua utilização, com o menor número de blocos tratados concorrentemente pelas unidades funcionais, deve existir uma redução da taxa de envio de instruções a serem tratadas pela UED em função do tempo, causando uma redução na velocidade com que as fichas de dados são enviadas para a Chave 2.

Apesar do aumento do tempo para o envio das fichas das unidades funcionais para as unidades de agrupamento, a ocupação das unidades de agrupamento aumenta para o intervalo de 1 até 6 e depois sofre uma pequena queda (gráficos B e G). Esse comportamento pode ser compreendido como uma consequência do atraso do envio de fichas parceiras para as unidades de agrupamento, uma vez que enquanto estas não forem sincronizadas com suas parceiras, estas devem continuar armazenadas na Memória de Dados das unidades de agrupamento. Portanto, é exatamente esse aumento do tempo necessário para o agrupamento das fichas de dados de entrada dos blocos que acarreta a degradação

da eficiência com que as unidades funcionais podem trabalhar.

7.4 Resumo

Através da avaliação do comportamento da arquitetura SEED pode-se chegar às seguintes constatações:

- Com o aumento do número de unidades funcionais:
 - os programas que geram atividades paralelas através de chamadas recursivas de funções, não conseguem aproveitar o maior número de recursos de processamento, pela limitação da transmissão de código da UI_Global para as unidades funcionais;
 - a porcentagem do tempo que os processadores das unidades funcionais passam executando as instruções internas aos blocos, torna-se muito reduzida para um número maior que 8 unidades funcionais, levando a um desperdício do potencial existente nas unidades funcionais para a execução de blocos de código concorrentemente.
- Com o aumento do número de rótulos locais:
 - os *benchmarks* utilizados são executados em um tempo menor, com exceção do programa de criação de matriz, que por sua característica particular, utiliza muito a Memória de Estruturas de Dados, fazendo com que o aumento da capacidade de execução paralela dos blocos de código nas unidades funcionais não seja plenamente aproveitado;
 - há um aumento da porcentagem do tempo que os processadores das unidades funcionais passam executando as instruções internas aos blocos;
 - a limitação na velocidade de processamento para valores acima de 32 rótulos locais deve-se à degradação de eficiência com que as unidades de agrupamento conseguem tratar o aumento do tráfego das fichas de dados;
- Com o aumento do número de unidades de agrupamento:

- praticamente não há variação do tempo requerido para execução dos *benchmarks* utilizados;
 - a porcentagem do tempo que os processadores das unidades funcionais permanecem ocupados torna-se consideravelmente maior para o programa de multiplicação de matrizes;
 - a ocupação das unidades de agrupamento aumenta, pois as unidades funcionais não conseguem absorver todos os blocos de código, habilitados mais rapidamente pelas unidades de agrupamento;
- Com o aumento do atraso imposto para transmissão dos pacotes de dados pela Chave 1, o tempo de execução aumenta 20% quando tempo de atraso aumenta 1000%, deixando claro que existe um tráfego bastante reduzido de fichas nessa chave de interconexão;
 - Com o aumento do atraso imposto para transmissão dos pacotes de dados pela Chave 2:
 - o tempo de processamento para o programa de multiplicação de matrizes torna-se 90% maior com o aumento de 1000% do tempo de atraso;
 - a ocupação das unidades de agrupamento torna-se elevada pela demora da transmissão de fichas de dados que devem ser sincronizadas nessas unidades;
 - torna-se menor a porcentagem do tempo que os processadores das unidades funcionais permanecem ocupados.

Capítulo 8

Considerações Finais

8.1 Conclusão

A proposta de uma nova arquitetura apresentada neste trabalho foi elaborada a partir das experiências adquiridas com as arquiteturas a fluxo de dados de primeira e segunda gerações. Assim, a arquitetura SEED tenta aproveitar as boas características apresentadas pelo modelo a fluxo de dados e sanar, ou pelo menos amenizar, os problemas apresentados pelas arquiteturas baseadas neste modelo que já foram estudadas.

O bom aproveitamento do paralelismo existente nas aplicações em função da facilidade de exposição deste, é a principal vantagem de utilização do modelo a fluxo de dados. Assim, visando utilizar essa qualidade das arquiteturas a fluxo de dados, a arquitetura SEED baseia-se neste modelo para habilitação de tarefas para a execução.

No entanto, pelo fato do modelo a fluxo de dados explorar paralelismo de granularidade muito fina e por habilitar a execução de tarefas pela disponibilidade dos dados, este modelo requer pelo menos uma comunicação para cada instrução executada. Dessa forma, o mais grave problema apresentado pelas arquiteturas baseadas no modelo a fluxo de dados está relacionado com a excessiva taxa de comunicação imposta pelo modelo. Tentando amenizar os efeitos decorrentes dessa característica, foi proposto que a arquitetura SEED aumente o tamanho das tarefas a serem atribuídas às unidades funcionais. Esse aumento deve ser proporcionado pelo particionamento do grafo de fluxo de dados em blocos de código cuja execução deve ser integralmente processada dentro de uma única unidade funcional.

Através da análise do comportamento da arquitetura em função do aumento de atraso

imposto pelas chaves de interconexão presentes na arquitetura, verificou-se que o aumento da granularidade, de fato, consegue reduzir o tráfego de fichas de dados na arquitetura. Essa diminuição do número de fichas que circulam pela arquitetura foi confirmada pela pequena degradação do desempenho da arquitetura se comparada proporcionalmente ao aumento do tempo aplicado nessa análise.

Também foi verificado que a estratégia utilizada para realizar o particionamento de programas na forma de grafos de fluxo de dados permite que o código gerado pelo particionamento apresente paralelismo suficiente para ser explorado pelos recursos oferecidos pela arquitetura SEED. Pôde-se verificar que mesmo quando existe um número maior de recursos oferecidos pela arquitetura (como por exemplo, o aumento do número de rótulos locais) os programas particionados em blocos de código conseguem apresentar paralelismo suficiente para manter um razoável valor de ocupação máxima das unidades funcionais. Este fato demonstra que existe uma quantidade considerável de tarefas que podem ser habilitadas paralelamente para a execução.

Em relação ao comportamento da arquitetura SEED observado em função do aumento dos recursos oferecidos pela arquitetura para exploração de paralelismo deve-se fazer algumas considerações.

O aumento do número de unidades funcionais, embora aumente a capacidade de processamento paralelo, pode gerar com maior frequência conflitos entre as requisições de código, enviadas pelas várias unidades funcionais para a ULGlobal. Estes conflitos podem, como no caso dos *benchmarks* puramente recursivos, levar a um desperdício do potencial de processamento oferecido pela arquitetura. Além disso, embora não tenha sido analisado neste trabalho, o aumento do número de unidades funcionais, bem como o número de unidades de agrupamento, levam a um aumento da complexidade de implementação das chaves de interconexão, devendo aumentar o tempo necessário para que estas possam transmitir as fichas de dados. Dessa forma, o número de unidades funcionais não deve ser elevado, mas apenas o suficiente para garantir um bom fluxo de pacotes de dados pela Chave 1 e um bom nível de utilização da ULGlobal. Para os *benchmarks* utilizados para as análises apresentadas no capítulo anterior, a configuração que apresentou melhores resultados foi a que possuía 8 unidades funcionais.

Por outro lado, o aumento da capacidade de cada UF executar um número maior de

blocos de código paralelamente, ou seja, o aumento do número de rótulos locais, garante uma utilização melhor dos recursos internos de cada UF. Esse fato leva a um melhor aproveitamento da arquitetura como um todo. Também neste caso existe um limite para o bom aproveitamento dos recursos oferecidos. A análise apresentada neste trabalho demonstrou que para um número superior a 32 rótulos locais não existe um aumento considerável da porcentagem de tempo que os processadores permanecem ocupados durante a execução de programas.

A existência de um número maior de unidades de agrupamento possibilita um aumento do número de fichas de dados produzidas pelas unidades funcionais que podem ser enviadas para as unidades de agrupamento em um mesmo tempo. No entanto, foi verificado que o aumento apenas do número de unidades de agrupamento praticamente não altera o desempenho da arquitetura SEED, principalmente, se comparado ao grau de alteração verificado pela variação do número de unidades funcionais e de rótulos locais. Como o aumento do número de unidades de agrupamento não oferece uma vantagem considerável para o desempenho da arquitetura, e deve representar um decréscimo da velocidade de transmissão de fichas de dados das chaves de interconexão, pode-se concluir que a existência de um número mais restrito de unidades de agrupamento pode ser mais vantajoso para a obtenção de um bom desempenho da arquitetura. A presente avaliação sugere a existência de 8 ou 16 unidades de agrupamento.

Através dessas primeiras avaliações pode-se concluir que a arquitetura SEED é capaz de aproveitar o paralelismo existente nas aplicações. Além disso, apesar de utilizar o modelo a fluxo de dados o tráfego de fichas de dados não é excessivamente alto, fato que comprova que o aumento da granularidade é capaz de reduzir a taxa de comunicação requerida pelo modelo.

A seção seguinte conclui a apresentação desse trabalho expondo alguns temas que devem ser estudados para que esta arquitetura possa ser mais amplamente pesquisada.

8.2 Trabalhos Futuros

Esse trabalho deve ser compreendido como um início de uma série de outros trabalhos que devem tratar, de forma mais profunda, cada uma das unidades que compõe essa arquitetura, incluindo também a implementação de um gerador de código. Dessa forma,

existem vários pontos que merecem uma análise mais detalhada e deverão ser tema de pesquisa em diversas áreas da computação. Cada um desses temas é descrito a seguir.

8.2.1 Particionamento de Código para Processamento Paralelo

Este problema é muito estudado por sistemas de computação concorrente e sempre está relacionado à fase de compilação. Dentro dos projetos em arquiteturas a fluxo de dados essa área foi muito pouco estudada, oferecendo amplas possibilidades de pesquisa.

No capítulo 5 foi apresentada uma forma de gerar código para a arquitetura SEED, partindo do grafo de fluxo de dados gerado para a MDFM e utilizando as propostas de particionamento utilizadas para a máquina abstrata TAM. No entanto, a geração do código dos programas utilizados como *benchmarks* no capítulo 7 foi realizada manualmente, apenas utilizando as idéias descritas no capítulo 5. Dessa forma, a implementação de um gerador de código automático para a arquitetura SEED é uma etapa que ainda deve ser cumprida para, posteriormente, poder-se realizar de forma mais ampla uma avaliação da qualidade do código gerado através dessa estratégia.

8.2.2 Implementação das Chaves de Interconexão

As altas taxas de comunicação requeridas pelas arquiteturas a fluxo de dados entre as unidades funcionais e as unidades de agrupamento, faz com que a velocidade com que as redes de interconexão conseguem transmitir informações, interfiram diretamente no desempenho destas arquiteturas. Para escolher a implementação de rede de interconexão que apresenta a melhor relação custo/desempenho, deve-se primeiramente escolher a topologia de rede a ser utilizada, e posteriormente a tecnologia a ser utilizada em sua implementação.

A escolha da rede de interconexão a ser utilizada pela arquitetura WOLF foi efetuada através de simulação, sendo escolhida a rede cuja relação custo/desempenho mostrou ser a melhor para esta arquitetura específica [Mart 94]. Atualmente, está sendo realizado um estudo para avaliar a implementação dessas redes de interconexão na arquitetura WOLF utilizando tecnologia óptica [Mart 96], também através de simulação. Baseando-se nestes trabalhos, será possível avaliar o desempenho da arquitetura SEED com a utilização de várias topologias de rede para implementar as chaves 1 e 2. Posteriormente, espera-se então, verificar a tecnologia a ser utilizada para a implementação destas que ofereça a

melhor relação custo/desempenho.

8.2.3 Utilização de Processadores Vetoriais em Fluxo de Dados

Como foi tratado no capítulo 3 a Unidade de Estruturas de Dados deve ser formada pela Memória de Estruturas e pela Unidade de Processamento Vetorial. No entanto, a UPV não foi abordada de forma completa neste presente trabalho. A inserção da UPV deve incluir:

- A definição da organização do processador vetorial a ser utilizado. Essa escolha deve considerar não apenas a performance mas também seu custo, pois é objetivo desse trabalho a obtenção de uma máquina de custo não muito elevado;
- A especificação de um protocolo de interação entre a ME, a UPV e as unidades funcionais. A maneira como a UPV deve interagir com as outras unidades da arquitetura deve levar em consideração que a UED deve ser utilizada como uma interface, que deve tornar compatíveis os protocolos de comunicação especificados pelas partes envolvidas;
- A implementação de um gerador de código que seja capaz de expor trechos de programa que podem ser executados com maior eficiência pelo processador vetorial. Estes trechos devem ser modificados pela inserção de instruções vetoriais explícitas.

8.2.4 Geração de Aplicativos Computacionais

Para uma avaliação mais completa do desempenho da nova arquitetura deverão ser gerados programas cuja execução exijam processamento de alto desempenho, por exemplo, cálculos numéricos, simulações, etc. Contudo, esse trabalho está condicionado à implementação de um gerador de código completo para a arquitetura SEED.

8.2.5 Análise em Nível de Hardware para Implementação da Arquitetura

A construção de um protótipo é a fase que deve ser posterior a todos os temas de pesquisa descritos nesta seção, por necessitar de todas as informações de especificação da arquitetura, do gerador de código e dos resultados obtidos pelas simulações.

Bibliografia

- [Agar 90] Agarwal,A. et al “*APRIL: a Processor Architecture for Multiprocessing*” Proceedings, 17th Annual International Symposium on Computer Architecture, 104-114, mai. 1990.
- [AlCo 76] Allen,F.E. & Cocke,J. “*A Program Data Flow Analysis Procedure*” Communications of the ACM, 19(3): 137-147, mar. 1976.
- [Alve 90] Alverson,R. “*The Tera Computer System*” Proceedings, 17th Annual International Symposium on Computer Architecture, 104-114, mai. 1990.
- [AnAC 92] Ang,B.S., Arvind & Chiou,D. “*StarT the Next Generation: Integrating Global Caches and Dataflow Architectures*” Proceedings, ISCA 1992 Dataflow Workshop, mai. 1992.
- [ArGo 82] Arvind & Gostelow,K.P. “*The U-interpreter*” IEEE Computer, 15(2): 42-49, fev. 1982.
- [ArNi 90] Arvind & Nikhil,R.S. “*Executing a Program on the MIT Tagged-Token Dataflow Architecture*” IEEE Transactions on Computers, 39(3): 300-318, mar. 1990.
- [ArTh 81] Arvind, K.P. & Thomas, R.E. “*I-Structures: An Efficient Data Structure for Functional Languages*” Laboratory of Computer Science, MIT, out. 1981.
- [BaGu 87] Barahona,P.M.C.C. & Gurd,J.R. “*Processor Allocation in a Multi-Ring Dataflow Machine*” Department of Computer Science, University of Manchester, 1987.
- [BoSa 86] Böhm.A.P.W. & Sargeant,J. “*Efficient Dataflow Code Generation for SISAL*” documento interno, University of Manchester, 1986.
- [BoSa89] Böhm.A.P.W. & Sargeant,J. “*Code Optimzation for Tagged-Token Dataflow Machine*” IEEE Trans. Comput. (38)1, 4-14, 1989.

- [CavG 95] Cavenaghi,M.A. & Garcia Neto,A. “*Desempenho do Simulador SAW em problemas de Métodos Numéricos*” Anais do XVI Congresso Latino Americano sobre Métodos Computacionais para Engenharia - CILAMCE, nov. 1995.
- [Chio 95] Chiou,D. et al “*StarT-NG: Delivering Seamless Parallel Computing*” Proceedings, EURO-PAR’95, 1995.
- [CuAr 89] Culler,D.E & Arvind “*Resource Requirements of Dataflow Programs*” Proceedings, 15th Annual International Symposium on Computer Architecture, 141-150, jun. 1988.
- [Cull 91] Culler,D.E et al “*Fine-Grain Parallelism with Minimal Hardware Support: a Compiler-Controlled Threaded Abstract Machine*” Proceedings, 4th International Conference on Architectural Support for Programming Languages and Operating Systems, 164-175, abr. 1991.
- [Cull 93] Culler,D.E et al “*TAM - a Compiler-Controlled Threaded Abstract Machine*” Journal of Parallel and Distributed Computing, 18: 347-370, 1993.
- [Denn 81] Dennis,J.B. “*Models of Data Flow Computation*” in Control Flow and Data Flow Concepts of Distributed Programming pp.346-354
- [EvGa 90] Evripidou, P. & Gaudiot, J.L. “*A Decoupled Data Driven Architecture with Vector and Macro Actor*”, Lecture Notes in Computer Science, 457, CON-PAR’90, sep. 1990.
- [GaHM 91] Gao, G.R., Hum, H.H.J and Monti, J.-M. “*Towards an Efficient Hybrid Dataflow Architecture Model*”, Lecture Notes in Computer Science, 505, PARLE’91, pp. 355-371, 1991.
- [Gao 93] Gao, G.R. “*Efficient Hybrid Dataflow Architecture Model*”, Journal of Parallel and Distributed Computing, 19(4), pp. 293, dec. 1993.
- [Glau 82] Glauert,J.R.W. “*High Level Languages for Data Flow Computers*” Pergamon-InfoTech State of the Art Report on Programming Technology, pp.173-193, mar. 1982.
- [GoId 77] Goto,E. & Ida,T. “*Parallel Hashing Algorithms*” Information Processing Letters, 6(1): 8-13, fev. 1977
- [GuKW 85] Gurd,J.R.; Kirkham,C.C. & Watson,I. “*The Manchester Prototype Dataflow Computer*” Communication of ACM, 28(1): 34-52, jan. 1985.

- [GuW1 80] Gurd,J.R. & Watson,I. "*Data Driven System for High Speed Parallel Computing*"(Part 1) Computer Design, 19(6): 91-100, jun. 1980.
- [GuW2 80] Gurd,J.R. & Watson,I. "*Data Driven System for High Speed Parallel Computing*"(Part 2) Computer Design, 19(7): 97-106, jul. 1980.
- [HaFu 88] Halstead,R.H & Fujita,T "*Masa: a Multithreaded Processor Architecture for Parallel Symbolic Computing*" Proceedings,15th Annual International Symposium on Computer Architecture, 443-451 jun. 1988.
- [Hi 87] Hiraki, K. et al "*The Sigma-1 Dataflow Supercomputer: A Challenge for New Generation*", Journal of Information Processing, Vol.10, No. 4, 1987.
- [ArKa 81] Arvind & Kathail "*A Multiple Processor Dataflow Machine that Supports Generalised Procedures*" Proceedings of the 8th ACM Symposium on Computer Architecture, mai 1981.
- [KaGu 86] Kawakami,K. & Gurd,J. "*A Scalable Dataflow Structure Store*" Proceedings,13th Annual International Symposium on Computer Architecture, 14(2): 243-250, jun. 1986.
- [Kirk 87] Kirkham, C.C. "*The Manchester Prototype Dataflow System: Basic Programming Manual*" documento interno, University of Manchester, set. 1987.
- [Magn 92] Magna,P. "*Redução da Chave de Emparelhamento da Máquina de Fluxo de Dados de Manchester*" Dissertação de Mestrado, Departamento de Física e Ciência dos Materiais, IFQSC, USP, set. 1992.
- [Mart 94] Martini, J.A. "*Estudo de Desempenho de Redes de Interconexão para a Arquitetura a Fluxo de Dados Proto-Wolf*" Dissertacao de Mestrado, Departamento de Física e Ciência dos Materiais, IFSC, USP, out. 1994.
- [Mart 96] Martins, J.E.M.P. "*Investigação de Chaves WDM para Arquiteturas a Fluxo de Dados Híbridas*" Tese de Doutorado (em preparação), Departamento de Física e Informática, IFSC, USP.
- [MiNB 95] Miller,W.M., Najjar,W.A. & Bohm, A.P.W. "*Exploiting Data Structure Locality in the Dataflow Model*" Journal of Parallel and Distributed Computing, 27: 183-200, jun. 1995.
- [MSAG 84] McGraw,J. et. al. "*SISAL - Streams and Iteration in a Single Assignment Language*" Language Reference Manual, ver. 1.2, M-146, Lawrence Livermore National Laboratory, ago. 1984.

- [NiAr 89] Nikhil,R.S. & Arvind “*Can Dataflow Subsume von Neumann Computing?*”, In Proc. 16th Ann. Intl. Symp. on Computer Architecture, Jerusalem, Israel, 262-272, may. 1989.
- [NiPA 92] Nikhil, R.S., Papadopoulos, G.M. & Arvind “**T: A Multithreaded Massively Parallel Architecture*”, ACM SigArch 20(2), pp. 156-167, 1992.
- [Nikh 90] Nikhil,R.S. “*Id Reference Manual*” documento interno, ver. 90.1, Laboratory for Computer Science, MIT, Cambridge MA, sept. 1990.
- [PaWo 86] Padua,D.A. & Wolfe,M.J. “*Advanced Compiler Optimizations for Supercomputers*” Communications of ACM, 29(12): 1184-1201, dez. 1986.
- [PaCu 91] Culler, D.E. & Papadopoulos, G.M. “*The Explicit Token Store*”, Journal of Parallel and Distributed Computing, jan. 1991.
- [Rugg 87] Ruggiero,C.A. “*Throttle Mechanisms for Manchester Dataflow Machine*” Ph.D. Thesis, Department of Computer Science, University of Manchester, jul. 1987.
- [Sakai 89] Sakai,S. et al. “*An Architecture of a Dataflow Single Chip Processor*” Proceedings, 16th Annual International Symposium on Computer Architecture, 46-53, may 1989.
- [SaKi 86] Sargeant,J. & Kirkham,C.C. “*Stored Data Structures on the Manchester Dataflow Machine*” Proceedings, 13th Annual International Symposium on Computer Architecture, 14(2): 235-242, jun. 1986.
- [Sarg 83] Sargeant,J. “*Is an Intelligent Token Queue the Throttle we Need?*” Department of Computer Science, University of Manchester, out. 1983.
- [Sarg 85] Sargeant,J. “*Efficient Stored Data Structures for Dataflow Computing*” Ph.D. Thesis, Department of Computer Science, University of Manchester, ago. 1985.
- [Scha 91] Schauser,K.E. “*Compiling Dataflow into Threads*” MS Thesis, Computer Science Division, University of California, 1991.
- [SKSYK 92] Sato, M., Kodama, Y., Sakai, S., Yamaguchi, Y. & Koumura, Y. “*Thread-Based Programming for the EM-4 Hybrid Dataflow Machine*”, ACM SigArch 20(2), pp. 146-155, 1992.
- [SkGl 85] Skedzielewski,S.K. & Glauert,J.R.W. “*IF1, an Intermediate Form for Applicative Language*” Reference Manual, M-170, Lawrence Livermore National Laboratory, jul. 1985.

- [SHNS 86] Shimada,T.; Hiraki,K.; Nishida,K. & Sekigushi,S. "*Evaluation of a Prototype Dataflow Processor of the Sigma-1 for Scientific Computations*" Proceedings of the 13th International Symposium on Computer Architecture, 14(2): 226-234, jun. 1986.
- [Smit 81] "*Architecture and Applications of the HEP Multiprocessor Computer System*" Proceedings of SPIE, 241-248, 1981.
- [SiWa 83] da Silva,J.G.D. & Watson,I. "*Pseudo-Associative Store with Hardware Hashing*" IEE Proc., 130(1): 19-24, jan. 1983
- [Trau 86] Traub,K.R. "*A Compiler for the MIT Tagged-Token Dataflow Architecture*" MS Thesis, MIT Laboratory for Computer Science, ago. 1986.
- [TrBH 82] Treleaven,P.C.; Brownbridge,D.R. & Hopkins,R.P. "*Data Driven and Demand Driven Computer Architecture*" Computing Surveys, 14(1): 93-143, mar. 1982.
- [WaGu 82] Watson,I. & Gurd,J. "*A Practical Dataflow Computer*" IEEE Computer, 15(2): 51-57, fev. 1982.
- [WiPi 91] Wiener,R.S. & Pinson,L.J. "*C++ Programação Orientada para Objeto - Manual Prático e Profissional*" Makron-McGraw-Hill, 1991.
- [Yama 91] Yamaguchi, Y. "*Synchronization Mechanisms of a Highly Parallel Dataflow Machine EM-4*" IEICE Transaction, E74(1), jan. 1991.