UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA E QUÍMICA DE SÃO CARLOS

“REDUÇÃO DOS BITS DE
EMPARELHAMENTO DA MÁQUINA DE
FLUXO DE DADOS DE MANCHESTER”

Patrícia Magna

*Dissertação apresentada ao Instituto
de Física e Química de São Car-
los, Universidade de São Paulo, para
obtenção do Título de Mestre em
Física Aplicada*

ORIENTADOR: *Prof. Dr. Carlos Antônio Ruggiero*

DEPARTAMENTO DE FÍSICA E CIÊNCIA DOS MATERIAIS

SÃO CARLOS

1992

OK
USP / IFQSC / 881
[REDACTED]
8-2-000004

SOL. 108

108

IFSC UNIVERSIDADE
DE SÃO PAULO
Instituto de Física e Química de São Carlos

Fone (0162) 72-6222
Fax (0162) 72-2218

Av. Dr. Carlos Botelho, 1465
Caixa Postal 369
CEP 13560.970 - São Carlos - SP
Brasil

MEMBROS DA COMISSÃO JULGADORA DA DISSERTAÇÃO DE MESTRADO DE Patricia Magna APRESENTADA AO
INSTITUTO DE FÍSICA E QUÍMICA DE SÃO CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 02/09/1992

COMISSÃO JULGADORA:

Prof. Dr. Carlos Antonio Ruggiero

Prof. Dr. Edson dos Santos Moreira

Profa. Dra. Maria Carolina Monard

Agradecimentos

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

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

Ao grupo *Dataflow* do DFCM e ao Prof. Dr. Arthur João Catto, pelas discussões sempre ricas em informações.

Ao CNPq pelo apoio financeiro.

Ao professor Álvaro e os amigos Mauro e Esmerindo pela ajuda com o $\text{\LaTeX}$.

À Maria Benedita (Benê), desenhista do Grupo de Instrumentação e Informática, pela competência na confecção dos desenhos.

À Lia, pela amizade e pelo competente trabalho de secretaria.

Ao professor Luciano pela ajuda na redação do *abstract*.

Aos meus pais, Jorge e Hortência, e às minhas irmãs, Adriana e Jocelí, pelo carinho e incentivo.

Ao Giulio, pelo amor e amizade, e também pelo companheirismo nas longas dias de estudo.

Aos amigos Márcio e Regina meus companheiros inseparáveis de sala e do cafezinho.

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

Conteúdo

1	Introdução	1
2	Modelo a Fluxo de Dados	4
2.1	Conceitos Básicos do Modelo a Fluxo de Dados	4
2.2	Máquina de Fluxo de Dados de Manchester	6
2.2.1	Chave	7
2.2.2	Unidade de Armazenamento	8
2.2.3	Unidade de Emparelhamento	8
2.2.4	Unidade de Instrução	10
2.2.5	Unidade de Processamento	11
2.2.6	Unidade de Armazenamento de Estruturas de Dados	11
2.3	Método de “Throttle”	13
2.4	Chave de Emparelhamento	14
3	SISAL e o Simulador	17
3.1	SISAL	17
3.1.1	Tipos de Dados	18
3.1.2	Estruturas Condicionais	19
3.1.3	Construção FOR	20
3.1.3.1	Forma Iterativa	20
3.1.3.2	Forma Paralela	21
3.1.4	Funções	21
3.2	Formato Intermediário IF1	22
3.3	Geração de Código na MDFM	22

3.3.1	Implementação de Chamadas de Funções	24
3.3.2	Implementação de Blocos de Repetição	25
3.4	Parâmetros de Avaliação	26
4	Tratamento de Estruturas de Dados e Ciclos	29
4.1	Tratamento de Estruturas de Dados	29
4.1.1	Operações Básicas com a UAE	30
4.1.1.1	Armazenamento	30
4.1.1.2	Seleção	31
4.1.1.3	Busca	32
4.1.1.4	Coleta	32
4.1.1.5	Incremento	33
4.1.1.6	Decremento	34
4.1.2	Operações Envolvendo Arrays em SISAL	35
4.1.2.1	Criação	35
4.1.2.2	Preenchimento	36
4.1.2.3	Seleção de um Elemento	36
4.1.2.4	Alteração de Valor	36
4.1.2.5	Concatenação	36
4.1.2.6	Limites Superior e Inferior	37
4.1.2.7	Tamanho	37
4.1.2.8	Inserção de Elementos Extras	37
4.1.2.9	Remoção de Elemento	37
4.1.2.10	Ajuste	37
4.1.2.11	Ajuste ao Limite Superior	38
4.1.3	Grafos de Operações com vetores	38
4.1.3.1	Criação	38
4.1.3.2	Preenchimento	38
4.1.3.3	Seleção	39
4.1.3.4	Troca de Valores	40
4.1.3.5	Concatenação	40

4.2	Otimizações de Blocos Repetitivos	41
4.2.1	Eliminação de Rotulação	41
4.2.2	Limitação de Iterações Ativas	41
4.3	Vetorização	42
5	Redução da Chave de Emparelhamento	45
5.1	Eliminação do Campo de Segmento	45
5.2	Eliminação do Campo de Índice	49
5.2.1	Operações com Estruturas de Dados	51
5.2.1.1	Criação	52
5.2.1.2	Preenchimento	53
5.2.1.3	Coleta	55
5.2.2	Modificações na Implementação de Blocos de Repetição	56
5.2.2.1	Limitação de Iterações Ativas	58
5.2.2.2	Transformação de Ciclos em Funções Recursivas	58
5.2.2.3	Chamadas de Funções com Ciclos Limitados	62
5.3	Redução do Campo de Nome de Ativação	65
5.4	Sumário	66
6	Considerações Finais	68
6.1	Conclusão	68
6.2	Trabalhos Futuros	70
 Apêndice		
A	Definições das Instruções da MDFM	76
A.1	Tipos de Dados	76
A.2	Abreviaturas	77
A.3	Especificação das Instruções	77
A.3.1	Operadores Lógico-Aritméticos	77
A.3.2	Operadores de Controle de Fluxo	77
A.3.3	Operadores para Manipulação de Rótulo, Contexto e Destino	78

A.3.4 Operadores Alteradores do Rótulo 78

A.3.5 Operadores Controladores de Arcos Dinâmicos 78

A.3.6 Operadores de Estruturas de Dados 78

A.3.7 Miscelânea 78

A.3.8 Operadores da Unidade de Armazenamento de Estruturas de Dados 78

•

Lista de Figuras

2.1	Grafo de Fluxo de Dados	5
2.2	Arquitetura da MDFM	6
2.3	Esquema da Unidade de Emparelhamento	9
2.4	Esquema da Unidade de Armazenamento de Estruturas	11
3.1	Geração de Código na MDFM	23
3.2	Esquema de uma Chamada de Função	25
3.3	Esquema Geral de um FOR	26
4.1	Grafo da Operação de ARMAZENAMENTO	31
4.2	Grafo da Operação de SELEÇÃO	31
4.3	Grafo da operação de BUSCA	32
4.4	Grafo da Operação de COLETA	33
4.5	Grafo da operação INCREMENTO	34
4.6	Grafo da operação DECREMENTO	34
4.7	Grafo para Criação de um Vetor	39
4.8	Grafo da Seleção de um Elemento de um Vetor	39
4.9	Grafo para Trocar o Valor de um Elemento de um Vetor	40
5.1	Exemplo de chamadas de funções	46
5.2	Exemplo de distribuição de funções na UI	47
5.3	Proposta de Modificação para Instrução GAN	48
5.4	Inserção da Tabela Localmente na UI	49
5.5	Inserção de Tabela junto com a UP	49
5.6	Proposta para Criação de um Vetor	52

5.7	Proposta para Preechimento de um Vetor	54
5.8	Gráfico de S_1 e S_{inf} como função de LIM ($N = 1024$)	65
5.9	Gráfico de π como função de LIM ($N = 1024$)	66

Lista de Tabelas

3.1	Nós Compostos no formato IF1	22
5.1	Correspondência entre segmento e nome de ativação	47
5.2	Simulação da forma atual de executar forall	60
5.3	Simulação da execução recursiva do forall	60
5.4	Simulação para execução duplamente recursiva	62
5.5	Simulação para execução duplamente recursiva com LIM = 4	62
5.6	Simulação para execução duplamente recursiva com LIM = 8	63
5.7	Simulação para execução duplamente recursiva com LIM = 16	64
5.8	Simulação para execução duplamente recursiva com LIM = 32	64

Resumo

O modelo a fluxo de dados tem grande destaque em pesquisas em arquiteturas de alto desempenho. Neste modelo, o controle de execução é feito apenas pela disponibilidade dos dados, permitindo que seja explorado o máximo de paralelismo implícito em um programa.

As propostas que serão expostas neste trabalho visam solucionar um particular problema da máquina de fluxo de dados de Manchester. Esta arquitetura para tratar código reentrante, impõe que as fichas de dados, além da indicação da instrução destino, possuam um rótulo. Estas informações extras, que formam 70% da ficha de dado, fazem com que a implementação da máquina seja complexa. Assim, o hardware impõe um sério limite a velocidade de processamento, impedindo a plena utilização do modelo.

Neste trabalho, serão apresentadas propostas para a redução do número de informações necessárias para o correto funcionamento da máquina, possibilitando uma implementação mais simples e mais eficiente.

Abstract

The dataflow model is specially relevant to research in high-performance architectures. In this model, the execution control is done by taking into account only the data availability, thus allowing maximum exploitation of the parallelism implicit in programs.

The present work is based on the Manchester dataflow machine, which, in order to handle the reentrant code, imposes the data token to have, in addition to the destination-instruction field, a label. This additional information, which corresponds to 70% of the data token, compounds the machine implementation as it substantially bounds the execution speed and prevents the full model utilization.

This work presents approaches for reducing the amount of information needed for proper machine operation in order to achieve simpler and more effective implementation.

Capítulo 1

Introdução

A velocidade de processamento atingida pelos computadores atuais possibilitou uma maior utilização destes nas mais variadas aplicações. Algumas aplicações porém, exigem velocidades ainda maiores que as obtidas pelos computadores convencionais (von Neumann). Alguns exemplos são: processamento de imagens, simulações de circuitos eletrônicos, previsões meteorológicas, etc.

Tendo como objetivo a obtenção de um processamento de alto desempenho vários modelos computacionais foram propostos, destacando-se, dentre estes, o modelo a fluxo de dados.

Um programa no modelo a fluxo de dados é representado na forma de grafos orientados, onde nós são operações e os arcos que os unem são caminhos percorridos pelos dados. O controle de execução de operações imposto pelo modelo é feito apenas pela disponibilidade dos dados, o que permite a extração do paralelismo implícito em programas, não sendo necessário encarregar o programador de expressar os trechos paralelos do código.

Em razão do alto paralelismo apresentado e pela simplicidade com que são distribuídas as tarefas aos processadores, este modelo tem despertado um grande interesse em vários grupos de pesquisa.

Várias arquiteturas a fluxo de dados foram analisadas e construídas como a *Tagged Token Dataflow Machine* (TTDM) do MIT [ArNi 90], o projeto Sigma-1 no Japão [SHNS 86], *Manchester Data Flow Machine* (MDFM) da Universidade de Manchester [GuKW 85] entre outras. Estas máquinas demonstraram a viabilidade da utilização do modelo e o grande potencial na exploração de paralelismo.

A arquitetura da Manchester (MDFM) foi uma das mais estudadas. Nesta, os dados são representados na forma de fichas que se movimentam por passagem de mensagens. O código reentrante é tratado dinamicamente, exigindo que cada ficha de dado, além do valor do dado, seja constituída por um rótulo, o que determina a classificação da MDFM como uma máquina de fluxo de dados dinâmica por rotulação de fichas.

A MDFM é composta por cinco unidades dispostas ao longo de um anel, por onde os dados circulam e obtêm as instruções as quais se destinam. Um campo da ficha de dado é utilizado para informar a instrução à qual o dado se destina.

As instruções na MDFM possuem no máximo dois dados de entrada. Duas fichas de dados são consideradas parceiras quando apresentam o mesmo rótulo (pertencem a uma mesma instância de um código reentrante) e destinam-se a uma mesma instrução.

O agrupamento de fichas parceiras é realizado associativamente por 54 bits que compõem os campos de rótulo e destino. Pelo alto custo de uma memória puramente associativa, a sua utilização na implementação da unidade, que faz o emparelhamento de fichas parceiras, é proibitiva, o que determinou o emprego de um método de “hashing” por hardware para simular uma memória associativa. Porém, este esquema de memória pseudo-associativa sofre grande degradação quando a ocupação de memória ultrapassa 25%, o que, normalmente, ocorre quando o paralelismo exposto por um programa aumenta. Este fato pode limitar a exploração de paralelismo exposto pelo modelo, comprometendo o desempenho da arquitetura.

Para que o modelo a fluxo de dados possa ser efetivamente utilizado, torna-se então, necessário, otimizar a operação de agrupamento de fichas parceiras. O trabalho que será descrito nos próximos capítulos apresenta propostas para redução da chave de emparelhamento, nome este, dado ao conjunto de campos da ficha de dado que devem ser comparados para identificação de fichas parceiras. Esta redução pode simplificar muito a implementação da unidade que emparelha fichas parceiras, fazendo com que o seu desempenho possa se tornar menos sensível ao aumento de fichas no anel.

Uma breve descrição dos capítulos desta dissertação é apresentada a seguir.

No capítulo 2 são fornecidos os conceitos básicos do modelo a fluxo de dados. Em especial, é apresentado o modo de funcionamento da arquitetura da MDFM, com a descrição de cada unidade que a forma.

Para estudar e avaliar o desempenho da MDFM o grupo de Manchester construiu um protótipo e desenvolveu dois simuladores. Estes últimos são capazes de fornecer informações ao usuário sobre os programas e o desempenho do hardware. A avaliação de propostas apresentadas neste trabalho será realizada através de simulação. Os parâmetros de avaliação utilizados pelos simuladores serão descritos no capítulo 3. Serão expostas também, as etapas da geração de código da MDFM, iniciando com um resumo da principal linguagem de alto nível utilizada para a programação da máquina, a linguagem funcional SISAL.

Um especial problema para redução do tamanho do campo de rótulo foi encontrado no tratamento dado às operações com estruturas de dados e a implementação de blocos iterativos. Este tratamento é apresentado no capítulo 4.

As propostas de redução da chave de emparelhamento são descritas no capítulo 5, e representam a parte central desta dissertação. As considerações finais e propostas de possíveis trabalhos futuros estão no capítulo 6.

Capítulo 2

Modelo a Fluxo de Dados

2.1 Conceitos Básicos do Modelo a Fluxo de Dados

O modelo de fluxo de dados baseia-se na exploração de paralelismo de granularidade bastante fina, a nível de instrução [ArGo 82]. O programa deixa de ser representado por uma sequência 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 tem 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. Consequentemente, 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 do 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 é dividido em três campos: nome de ativação, nível de iteração e índice.

2.2 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. O código de máquina contém operações lógico-aritméticas, instruções de “branch” para implementar estruturas condicionais e instruções para manipulação do rótulo. As instruções podem ter no máximo dois dados de entrada, ou seja, podem ser unárias ou binárias.

Figura 2.2: Arquitetura da MDFM

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 ativi-

dades das unidades são independentes, as atividades na execução de cada unidade também podem ser realizadas em paralelo. Um hardware mais potente é conseguido adicionando vários anéis [GuW2 80].

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

As fichas são constituídas pelos seguintes campos: valor, rótulo e instrução destino. 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[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].

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; 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.

Com o funcionamento geral da MDFM já introduzido, serão apresentados, agora, maiores detalhes da implementação de cada unidade.

2.2.1 Chave

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

Para que não exista atraso na saída de fichas da UP, o tempo de operação da CH deve ser convenientemente pequeno e caso dois dados cheguem à CH simultaneamente, é dada maior prioridade a ficha proveniente da UP.

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 e examinar como esta separação influencia na seleção da topologia da rede de interconexão da chave [BaGu 87].

2.2.2 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 comunicação com as unidades vizinhas é feita através de “*buffers*” de entrada e saída e internamente esta unidade é constituída por uma memória que tem o simples propósito de sincronizar a razão entre a produção e o consumo de fichas.

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.

2.2.3 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

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 subsequentes 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.

Para resolver certos problemas, como construções não determinísticas, e para o tratamento de estruturas de dados [Catt 81] foi necessário introduzir 6 outras funções de emparelhamento, além das funções utilizadas para o comportamento normal explicado acima.

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 [GuKW 85], sendo que foi verificado que programas típicos possuem entre 50% a 70% de instruções unárias.

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 sequencial;

2.2.4 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.

A 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 à unidade de processamento.

Figura 2.4: Esquema da Unidade de Armazenamento de Estruturas

2.2.5 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 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.

2.2.6 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 fun-

cionamento 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].

A unidade 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 pela chave. Um esquema da UAE pode ser visto na figura 2.4.

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 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 acarreta a 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.

2.3 Método de “Throttle”

O modelo a fluxo de dados sofreu muitas críticas pelo excessivo uso de memória durante a execução de programas com um alto grau de paralelismo.

Os estudos com a MDFM demonstraram, também, que a ativação paralela de muitas funções, sem impor algum tipo de controle, pode causar a saturação da máquina. Este tipo de execução, gera um número excessivo de fichas no anel. Como consequência, a ocupação de memória, principalmente da UE torna-se muito elevada, acarretando a degradação do desempenho do anel.

A ocorrência de atrasos no emparelhamento de fichas pode implicar na existência de processadores sem trabalho, por alguns ciclos de máquina, resultando em um baixo aproveitamento do hardware.

Em virtude dos problemas ocasionados na MDFM, quando o paralelismo apresentado por um programa é elevado, tornou-se importante idealizar formas de contenção de paralelismo.

O método desenvolvido em Manchester para controle de paralelismo foi o método de Throttle [Rugg 87]. Através deste, é possível controlar a ativação de funções dependendo do nível de atividade da máquina. Se o nível de atividade da máquina encontra-se alto

quando uma requisição para ativação de um função é feita, esta deve ser suspensa até que algumas funções terminem sua execução. Caso a atividade da máquina encontre-se em um nível abaixo do considerado crítico a ativação é realizada imediatamente.

O nível de atividade é medido através do número de fichas que ocupam a UA. Se este número cresce muito rapidamente, isto quer dizer, que os processadores estão gerando mais fichas do que conseguem consumir; assim, deve-se suspender a ativação de funções até que a ocupação da UA diminua. Quando a quantidade de fichas na UA diminui, as requisições que foram suspensas durante o período em que a máquina encontrava-se saturada podem ser ativadas. Para controlar esta operação, uma nova unidade foi criada, o Alocador. Este deve ser consultado a cada nova ativação requerida. Os processos suspensos devem ser armazenados nesta unidade, sendo que também cabe ao Alocador decidir o processo que será ativado, quando o nível de atividade da máquina abaixar.

A existência de instruções que não retornam resultados, faz com que o reconhecimento de conclusão de um processo não seja dado apenas pela obtenção do resultado final do processo. Assim, o resultado final pode ser obtido sem que algumas instruções do processo tenham sido executadas. Para que os rótulos de processos já concluídos possam ser atribuídos a outros que serão ativados posteriormente, foram desenvolvidas formas de detecção do término da execução de um processo. O esquema elaborado assinala a conclusão do processo quando há a garantia de execução de todas as instruções pertencentes ao processo. Desta forma, o rótulo deste processo pode ser reciclado, mais especificamente, poderá ser reciclado o campo de nome de ativação. Este campo será descrito na próxima seção.

O método de controle de paralelismo aliado ao esquema de detecção de conclusão de processo serão condições básicas para a redução do tamanho do campo de nome de ativação (assunto tratado no capítulo 5).

2.4 Chave de Emparelhamento

A MDFM, por ser uma máquina a fluxo de dados dinâmica por rotulação de fichas, necessita que suas fichas de dados sejam constituídas por outros campos além do campo de dados. O formato de uma ficha é o seguinte [GuKW 85]:

<dado (37 bits), rótulo (36 bits), destino (22 bits), marcador (1 bit)>

Pode-se notar que apenas 30% da ficha é informação realmente útil, sendo que o restante é “overhead” necessário para o correto funcionamento do sistema.

O campo de destino possui informação sobre o endereço da instrução destino (18 bits), a “porta” de entrada na instrução que pode ser esquerda ou direita (1 bit) e a função de emparelhamento (3 bits). Como a unidade de instrução é segmentada, o endereço destino é separado em dois campos, um indicando o segmento (6 bits) e outro, o “offset” a ser adicionado ao endereço base (12 bits).

O emparelhamento de fichas parceiras é feito, como foi visto no item 2.2.3, através de 54 bits que formam a chamada chave de emparelhamento. Esta é constituída pelos seguintes campos:

<rótulo (36 bits), endereço destino (18 bits)>

O rótulo pode ser separado em três sub-campos: nome de ativação, nível de iteração e índice com 12 bits cada um. O nome de ativação identifica a ativação da função da qual a ficha faz parte. O nível de iteração separa fichas dentro de blocos repetitivos. Já o índice distingue elementos de uma mesma estruturas de dados.

A existência de redundâncias na chave de emparelhamento foi estudada em Manchester por Veen [Veen 85]. No trabalho de Veen são ressaltados os seguintes pontos:

- o nível de iteração pode ser suprimido, pois iterações podem ser separadas pelo nome de ativação e índice;
- o tamanho do campo de índice pode ser reduzido já que o compilador é capaz de quebrar grandes estruturas em estruturas menores;
- a utilização do mecanismo de Throttle[Rugg 87] [RuSa 85] permite a reciclagem de nomes de ativação o que possibilita a redução do tamanho deste campo;
- a combinação nome de ativação e endereço destino é redundante, pois quando uma função é ativada, é gerado um nome de ativação com o qual todas as suas fichas deverão ser executadas. Se todas as instruções desta dada função forem armazenadas

sempre num mesmo segmento, o valor destes dois campos, durante a execução da função, permanecem inalterados.

A partir da análise destas redundâncias, a chave de emparelhamento seria da forma:

<nome de ativação (10 bits), índice (8 bits), destino (12 bits)>

Através desta considerável redução do tamanho da chave de emparelhamento, Veen, fazendo alterações na função de “hashing”, sugere que é possível um uso de memória mais eficiente.

Para implementação destas observações feitas por Veen são necessárias a realização de modificações tanto do hardware como do próprio gerador de código. No capítulo 5 serão apresentadas as propostas que permitem a validação destas observações, e também, propostas para outras alterações que permitirão uma redução ainda maior da chave de emparelhamento.

Capítulo 3

SISAL e o Simulador

3.1 SISAL

A tarefa de escolher uma linguagem de programação para máquinas a fluxo de dados é complexa. Os programas devem apresentar um paralelismo significativo para que se consiga utilizar todo o potencial do hardware. Além disto, já que o modelo de fluxo de dados consegue expor o paralelismo implícito nos programas, seria interessante que o programador ficasse livre da incumbência de explicitar paralelismo. Outros pontos são também importantes na escolha de linguagem [Glau 82]: a facilidade no aprendizado da linguagem pelos programadores, a eficiência que os programas podem ser escritos e a facilidade de compilação de programas.

A principal linguagem de alto nível utilizada para 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], que será descrito com mais detalhes em 3.2.

SISAL é uma linguagem funcional que não permite a re-atribuição de valores à variáveis e não possui variáveis globais, efeitos colaterais (“*side-effects*”) e o conceito de execução sequencial explícita. Funções nesta linguagem são, verdadeiramente, funções na forma matemática.

Algumas vantagens para a utilização do SISAL podem ser destacadas [Acke 82]:

- O paralelismo que pode ser explorado, fica limitado apenas pela dependência de dados.
- Devido à ausência de efeitos colaterais, a extração de paralelismo torna-se simples, não exigindo do compilador uma análise demorada como ocorre na extração de paralelismo de linguagens convencionais;
- a transformação de programas para a forma de grafos é feita com facilidade e nesta notação o paralelismo disponível é de fácil reconhecimento.
- otimizações convencionais como eliminação de sub-expressões comuns são fáceis de serem executadas pelo compilador pela ausência de efeitos colaterais.

Os programas exemplos, que serão apresentados neste trabalho, foram todos implementados em SISAL. Desta forma, torna-se necessária uma breve descrição de algumas características da linguagem, como: tipos de dados, estruturas condicionais, funções e iterações. Informações mais detalhadas podem ser encontradas em [MSAG 84].

3.1.1 Tipos de Dados

A linguagem SISAL possui tipos escalares usuais: **integer**, **boolean**, **real**, **double_real** e **character**. Exemplos de declaração seriam:

```
num_int : integer;
num_real : real;
```

A declaração não precisa ser explícita, podendo ser deduzida em tempo de compilação:

```
num_int := 100;
num_real := 4.5;
```

Quatro tipos de dados estruturados existem em SISAL: **array**, **stream**, **record** e **union**.

Um **array** possui índices inteiros e seus elementos são todos de um mesmo tipo pré-definido. Um **array** multi-dimensional é representado por um **array** de **array**. Exemplos de declaração são:

```
type vetor : array[real]
type matriz : array[vetor]
```

O tipo **stream** é uma sequência ordenada de valores de um mesmo tipo pré-definido. O acesso aos seus elementos é feito através de operações tais como, **stream_first** que devolve o primeiro elemento de um **stream** ou **stream_rest** que retorna um **stream** idêntico ao da entrada da operação, exceto que o primeiro elemento é removido. A definição é da forma:

```
type texto : stream[character]
```

O tipo **record** é formado por um ou mais campos de componentes que podem ser de tipos distintos, sendo cada campo identificado por um nome:

```
type complexo : record[re, im: real]
```

```
type prova : record[
    num_aluno: integer;
    nota: real]
```

O tipo **union** pode conter objetos de tipos e tamanhos diferentes associados por nomes. Um particular valor do tipo **union** tem apenas um dos objetos que o forma. Um exemplo de declaração é:

```
type numero : union[
    num_real : real;
    num_compl : complexo]
```

A **union** *numero* pode conter um valor **real** associado pelo nome *num_real* ou então, pode conter um valor do tipo **complexo** (definido anteriormente como sendo um **record**) rotulado pelo nome *num_compl*.

3.1.2 Estruturas Condicionais

A construção condicional seleciona uma expressão dependendo de um resultado lógico ("boolean"). A expressão selecionada pode retornar, assim como toda expressão em SISAL, um ou mais valores. Diz-se então que, pode ter "aridade" maior do que um. Assim, uma típica construção condicional é:

```
maximo, minimo := if x > y then x, y else y, x end if
```

Como a construção condicional retorna valores de uma expressão, a condição **else** sempre deve estar presente para definir o resultado caso o valor do teste seja falso.

3.1.3 Construção FOR

A construção FOR possui duas formas: paralela e iterativa. A forma paralela é usada quando todos os ciclos podem ser executados concorrentemente, e a iterativa é necessária quando um ciclo depende de resultados de ciclos anteriores.

3.1.3.1 Forma Iterativa

A forma iterativa é invocada pela construção **for initial** e um exemplo da sua utilização é o cálculo da raiz quadrada de um valor usando o método de Newton-Raphson:

```
sqrt_x :=  
  for initial  
    est := 0.5 * x  
  repeat  
    est := 0.5 * (old est + x / old est);  
    dif := abs (est * est - x)  
  until dif < tolerancia  
  returns value of est  
end for
```

A seção **initial** define os valores iniciais para os nomes que poderão ser alterados durante os vários ciclos. A cada execução do “corpo do *loop*” são atribuídos novos valores para os nomes definidos no início do *loop* e também podem ser introduzidos nomes locais (como *dif* no exemplo acima) que não possuem valores iniciais. O teste de terminação pode ser colocado antes ou depois do “corpo do *loop*”. Os resultados da iteração são especificados pela instrução **returns**. A indicação **value of** define uma operação de redução que determina os objetos a serem retornados. Outros exemplos de operações de redução são: **sum** (somatório de resultados de cada ciclo), **product** (semelhante ao **sum**, retorna multiplicação), **greatest** (maior resultado de todos os ciclos), **catenate** (retorna **arrays** ou **streams** concatenados) e **array of** e **stream of** (retorna todos resultados em forma ou de **array** ou de **stream**, respectivamente).

Para poder implementar este tipo de construção numa linguagem de atribuição única é necessário utilizar a indicação **old**, que permite que valores calculados em iterações

prévias sejam utilizados para o cálculo de um novo valor para o mesmo nome.

3.1.3.2 Forma Paralela

A versão “*forall*” paralela pode ser exemplificada pelo trecho de programa abaixo. Trata-se da criação de um vetor de N elementos.

```
for I in 1,N
  El := 3 * I
returns array of El
end for
```

A primeira seção da construção corresponde ao gerador do *loop* que especifica o intervalo de valores de I que será utilizado. O restante desta construção é bastante similar ao *for initial*. Note a utilização da operação de redução *array of*.

3.1.4 Funções

Um programa SISAL é constituído por um conjunto de funções. Funções recebem valores como parâmetros de entrada e retornam valores como saída. Uma função deve sempre ser definida antes de ser chamada dentro do programa. SISAL pode implementar funções recursivas, sendo que o uso deste tipo de função é interessante para gerar paralelismo, em especial, no caso da função ser duplamente recursiva. Exemplos de algoritmos duplamente recursivos que foram implementados em SISAL para serem usados como “benchmarks” seriam: quicksort, fatorial e n_queens [HoSa 79]. O programa SISAL do fatorial, usando uma função duplamente recursiva, é apresentado a seguir.

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

function main(returns integer)
  fat(5,1)           %calcula o fatorial de 5
end function
```

3.2 Formato Intermediário IF1

O formato intermediário chamado IF1 é uma representação gráfica independente da máquina [SkGl 85].

Um grafo de fluxo de dados IF1 é acíclico e nele, nós representam operações e linhas definem o caminho dos dados entre os nós. Cada linha tem especificado o tipo do dado que é transmitido por ela.

Nós podem ser classificados como simples e compostos. Nós simples representam operações básicas como instruções lógico-aritméticas. Nós compostos compreendem sub-grafos com ligações implícitas entre eles; cada sub-grafo define parte de uma expressão estruturada. Existem cinco nós compostos definidos em IF1 um para cada tipo de expressão estruturada do SISAL.

nome	utilização
Select	if-then-else
TagCase	extração de valores de uma determinada union
loopA	iteração com teste de terminação depois de um ciclo
loopB	iteração com teste de terminação antes de um ciclo
forall	ciclos concorrentes

Tabela 3.1: Nós Compostos no formato IF1

3.3 Geração de Código na MDFM

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 3.1.

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 “macros-nós”. Nesta fase também é verificado se um nó do grafo IF1 possui o número errado de portas de entrada e de saída, e são inseridos nós extras no grafo para 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:

Figura 3.1: Geração de Código na MDFM

- 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 à 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.

O código gerado pelo sistema de compilação descrito acima podia ser executado de três formas: pelo protótipo ou pelos simuladores SIM e MR. O simulador SIM foi o primeiro a ser desenvolvido e simula apenas um único anel, não modelando o tempo. O MR (“multi-ring”) emula a versão multi-processador da MDFM e faz a modelagem do tempo.

O presente trabalho contou apenas com os dois simuladores sendo que através destes foi possível obter dados estatísticos sobre o funcionamento da máquina durante a execução de programas utilizados como exemplos. Mais detalhes sobre estas informações fornecidas pelos simuladores serão apresentadas no item 3.4.

3.3.1 Implementação de Chamadas de Funções

Quando uma chamada de função é feita, um nome de ativação é gerado e as fichas dos parâmetros que devem ser passados para a função são rotuladas com este novo nome de ativação. Então, as fichas dos parâmetros passam para o “corpo da função”. Para que os resultados da função sejam enviados para a função que fez a chamada, é preciso fazer com que as fichas dos resultados tenham o nome de ativação desta função e, assim é finalizada a execução de uma função. A forma geral de uma chamada de função pode ser vista na figura 3.2.

A instrução GAN ¹(*Generate Activation Name*) retorna um nome de ativação único que será utilizado pelas fichas da função numa determinada instância desta função. Os parâmetros de entrada da função mudam de nome de ativação através da instrução SAN (*Set Activation Name*). A instrução PRP (*Prepare Access*) gera uma ficha com o campo de dado sendo formado pelo contexto da função que fez a chamada e pelo endereço da instrução destino que espera o resultado da função. O rótulo da ficha gerada pelo PRP recebe o novo nome de ativação gerado pela instrução GAN. A instrução SCD (*Set Color and Destination*) recebe o valor do resultado e o envia para o destino especificado pela

¹o apêndice A apresenta a descrição das instruções da MDFM que serão utilizadas durante a exposição deste trabalho

instrução PRP, com o nome de ativação da função que fez a chamada.

Figura 3.2: Esquema de uma Chamada de Função

3.3.2 Implementação de Blocos de Repetição

Originalmente, a implementação de blocos repetitivos na máquina de Manchester era feita através do campo de nível de iteração, contudo para reduzir o número de informações que uma ficha de dado precisava possuir além de seu valor, este campo foi suprimido e em seu lugar passaram a ser utilizados os campos de nome de ativação e índice. A figura 3.3 mostra a estrutura geral de uma construção FOR resultante desta otimização.

A passagem de parâmetros para *loops* é feita através de geradores, cuja a função é gerar *n* fichas para cada um dos *N* parâmetros com índices variando de 0 até *n*-1. Posteriormente, as fichas dos parâmetros recebem nomes de ativação correspondentes a cada uma das *n* iterações e os índices são zerados. O corpo do *loop* é então executado e ao final, é feita a “re-rotulação”, onde cada ficha volta a ter o nome de ativação anterior e índices variando de 0 até *n*-1. O resultado final é obtido após a função de redução ser

executada.

Figura 3.3: Esquema Geral de um FOR

3.4 Parâmetros de Avaliação

A interação simulador-usuário é feita através de medidas de avaliação que são fornecidas pelo simulador [GuWa 83] [BoGu 87]. O objetivo para avaliação são:

- Obter o melhor desempenho do hardware.
- Determinar o paralelismo no software que melhor utiliza o hardware.

O simulador apresenta ao usuário muitas informações sobre o programa e sobre o desempenho do hardware, incluindo algumas que não poderiam ser obtidas caso fosse utilizado o protótipo. Algumas informações são [SarS 85]:

- S_1

Número total de instruções executadas do programa, ou seja, o número de passos (“time steps”) necessários se somente um processador estivesse disponível.

- S_{inf}

Este parâmetro é calculado como sendo o número de passos necessários para execução do programa se um número ilimitado de processadores estiverem sempre disponíveis, e considerando que todas as instruções levam o mesmo tempo para serem executadas. Na representação de grafos de fluxo de dados, onde instruções diretamente dependentes são colocadas umas sobre as outras, seria equivalente a altura do grafo. Esta ordem de execução das instruções é também conhecida como sendo o “caminho crítico” do grafo.

- π

É calculado como sendo a razão entre S_1 e S_{inf} . Trata-se de uma medida do paralelismo médio disponível no grafo de fluxo de dados.

Se um programa possui um alto valor de π isto significa que o número total de instruções do programa é grande comparado com a altura do grafo e, consequentemente, muitas instruções podem ser executadas em paralelo. Então, um programa com um alto valor de π é um programa que possui um alto grau de paralelismo. Experimentalmente, esta medida de paralelismo mostrou-se boa para avaliar o paralelismo existente no programa [GuWa 83].

- P_{by}

Proporção de instruções unárias executadas.

- Máxima Ocupação na Unidade de Armazenamento

Número máximo de fichas que foram armazenadas na UA.

- Máxima Ocupação na Unidade de Emparelhamento

Número máximo de fichas que foram armazenadas na UE.

- **Máxima Ocupação na Unidade de Armazenamento de Estruturas**

Número máximo de fichas que foram armazenadas na UAE.

Os parâmetros acima são úteis na análise do desempenho da arquitetura e serão utilizados para avaliar as modificações no tratamento de estruturas de dados e blocos de repetição sugeridas no capítulo 5.

Capítulo 4

Tratamento de Estruturas de Dados e Ciclos

Para possibilitar a eliminação de redundâncias na chave de emparelhamento é necessário analisar a utilização dos campos do rótulo. Neste capítulo será analisada a utilização de dois campos: nível de iteração e índice. Estes campos são empregados para implementar operações envolvendo estruturas de dados e blocos de repetição. Assim, será descrito com algum detalhe o tratamento de estrutura de dados e construção repetitiva dado pela máquina de Manchester e também por máquinas convencionais com processamento vetorial.

4.1 Tratamento de Estruturas de Dados

Existem duas abordagens para a representação de estruturas de dados: uma na forma de cadeia de fichas e outra na forma de estrutura armazenada.

Na primeira, os elementos são diferenciados pelo campo de índice de cada ficha. Este tipo de representação possui um alto grau de paralelismo, mas sua utilização torna-se dispendiosa já que:

- o número de cópias dos elementos pode ser alto. Como as operações no modelo a fluxo de dados não preservam seus argumentos existe a necessidade de copiar a estrutura quantas vezes forem as operações sobre ela.
- o tempo gasto para rotulação das fichas de elementos não representa um trabalho

útil.

- a eficiência na implementação de acesso aleatório aos elementos pode ser baixa.

Uma forma mais eficiente de tratar estruturas de dados é armazená-las e representá-las no anel através de um ponteiro. Este tratamento é mais interessante para finalidade desta dissertação porque viabiliza a redução do campo de índice, já que nesta forma o tamanho da estrutura não é limitado pelo tamanho deste campo. Os próximos itens tratarão da manipulação de estruturas de dados e descreverão apenas operações com estruturas armazenadas.

O primeiro trabalho a respeito do armazenamento de estruturas de dados na MDFM foi realizado por Bowen [Bowe 81]. O armazenamento dos elementos era implementado na Unidade de Emparelhamento e as operações com as estruturas eram realizadas por funções de emparelhamento especiais [Catt 81]. Bowen definiu um pequeno conjunto de operações para estruturas armazenadas para sua linguagem “dataflow” (MAD), quais seriam: ARMAZENAMENTO, BUSCA, SELEÇÃO, COLETA, INCREMENTO e DECREMENTO.

Uma forma mais eficiente de implementar o armazenamento de estruturas foi estudado por J. Sargeant [Sarg 85]. Baseado neste último, foi construída a Unidade de Armazenamento de Estruturas (UAE) descrita em 2.2.6.

4.1.1 Operações Básicas com a UAE

Nesta seção serão apresentados os grafos que realizam as operações básicas de armazenamento de estruturas no sistema de Manchester.

4.1.1.1 Armazenamento

A operação ARMAZENAMENTO transforma uma cadeia de fichas em uma estrutura armazenada, retornando um ponteiro ao anel, como mostra figura 4.1 (as mensagens à UAE são representadas por linhas não contínuas).

A instrução GSP (*Generate Structure Pointer*) (vide apêndice A) envia à UAE uma requisição de um ponteiro para uma estrutura de N componentes (N+1 posições são necessárias, pois uma posição é usada para o contador de referências, RC). A UAE envia

Figura 4.1: Grafo da Operação de ARMAZENAMENTO

ao anel um ponteiro para a nova estrutura, cujo RC recebe o valor inicial 1, através da instrução SRC (*Store Reference Count*). Para o emparelhamento do ponteiro com os vários elementos da estrutura é necessário utilizar a instrução PRL (*PRoLiferate*) que gera N fichas tendo como valor de dado o ponteiro e índices variando de 0 até $N-1$. Assim, a instrução WSS (*Write Structure Store*) armazena cada dado na posição dada pela soma do endereço inicial + o índice + 1.

4.1.1.2 Seleção

Figura 4.2: Grafo da Operação de SELEÇÃO

A busca de um determinado elemento de uma estrutura é realizada pela operação SELEÇÃO que através do ponteiro e do índice define o elemento a ser retornado (figura 4.2).

A instrução RSS (*Read Structure Store*) envia mensagem de leitura à UAE que retorna o valor indexado pelo valor I.

4.1.1.3 Busca

A operação OM TRUCTURE TORE) é uma combinação de PRL e RSS e

envia uma série de mensagens de leitura à UAE. O grafo desta operação é mostrado na figura 4.3.

Figura 4.3: Grafo da operação de BUSCA

4.1.1.4 Coleta

A operação COLETA é usada para determinar quando todos os elementos de uma cadeia de fichas estão prontos.

Esta operação pode ser efetuada de duas maneiras: uma através de funções especiais de emparelhamento [Catt 81][Bowe 81] e outra usando a UAE. Será descrito neste item apenas a COLETA na UAE.

A operação de COLETA inicia com a criação de uma estrutura de tamanho zero, ou seja, apenas o RC (figura 4.4). Este recebe como valor inicial o número de elementos a

serem coletados. O contador é decrementado de 1 através da instrução DRC (*Decrement Reference Count*) a cada elemento pronto. Quando o RC chega ao valor zero, a UAE envia um sinal ao anel representando o fim da operação de COLETA. Dentro da UAE automaticamente é acionada a limpeza da estrutura auxiliar criada quando o RC chega a zero.

Figura 4.4: Grafo da Operação de COLETA

4.1.1.5 Incremento

Para realização da “*garbage collection*” através do esquema de contagem de referências são utilizadas as duas operações INCREMENTO e DECREMENTO. A operação INCREMENTO incrementa o RC da estrutura indicada pelo ponteiro quando uma cópia da estrutura é realizada (figura 4.5).

O ponteiro de entrada da instrução IRC (*Increment Reference Count*) especifica a estrutura cujo RC deve ser alterado. A instrução envia uma mensagem à UAE para incrementar de 1 o RC, e a UAE envia uma mensagem ao anel quando a operação está concluída. O sinal do término da operação de INCREMENTO é o retorno do ponteiro para estrutura sincronizado com a mensagem enviada pela UAE pela instrução SYN (SYNcronize).

Figura 4.5: Grafo da operação INCREMENTO

4.1.1.6 Decremento

Este grafo é utilizado para decrementar o RC quando uma cópia da estrutura é destruída.

Figura 4.6: Grafo da operação DECREMENTO

A operação DECREMENTO (figura 4.6) é implementada apenas pela instrução DRC

(*Decrement Reference Count*) que através do ponteiro de entrada identifica a estrutura que deve ter o RC decrementado de 1. A segunda entrada é apenas utilizada com o propósito de sincronização. Um sinal de terminação é enviado ao anel apenas quando o RC chega ao valor zero, e é disparado o mecanismo de limpeza.

4.1.2 Operações Envolvendo Arrays em SISAL

Algumas operações com vetores utilizam o campo de índice. Assim, para verificar como as operações são implementadas na máquina de Manchester serão descritas as possíveis operações envolvendo vetores em SISAL e, no próximo item, como são os grafos que as implementam.

Um vetor é definido como `array[T]`, onde T é um tipo pré definido, e consiste de:

- um intervalo (LO, HI), onde LO é o limite inferior e HI o limite superior.
- uma sequência $HI - LO + 1$ de elementos, onde todos os elementos devem existir.

O tamanho da estrutura é conhecido quando esta é criada. As operações possíveis com este tipo de estrutura de dados são dadas a seguir.

4.1.2.1 Criação

Se os elementos que formam um vetor são conhecidos quando o programa SISAL está sendo escrito, então pode-se utilizar a seguinte operação:

`UM_ATE_TRES := array[1: 1,2,3]`

Esta operação cria um vetor com índice inferior igual a 1 e elementos contendo os valores 1, 2 e 3.

Um vetor com os limites inferior e superior iguais a zero é um vetor vazio. A criação de um vetor vazio é feita pela operação:

`VAZIO := array Vector[]`

4.1.2.2 Preenchimento

Existe uma função pré-definida `array_fill` que preenche um `array` com um certo valor.

$$\text{VETOR_ZERO} := \text{array_fill}(1,5,0)$$

Após a execução desta operação o vetor `VETOR_ZERO` é criado com 5 elementos, todos contendo o valor zero e índices variando de 1 até 5.

4.1.2.3 Seleção de um Elemento

A operação `A[J]` seleciona o J-ésimo elemento de um vetor `A`.

Para o caso de uma matriz a seleção é feita por dois índices, por exemplo, `A[I][J]` que seleciona o J-ésimo elemento de vetor que é o I-ésimo elemento da matriz `A`.

4.1.2.4 Alteração de Valor

Para a troca do valor da posição `J` de um vetor é feito:

$$A[J] := V$$

Assim, o J-ésimo elemento do vetor `A` é extraído e o novo valor `V` é inserido em seu lugar.

4.1.2.5 Concatenação

Seja o vetor :

$$\text{QUATRO_ATE_OITO} := \text{array}[1: 4,5,6,7,8]$$

Suponha a operação de concatenação a seguir:

$$\text{UM_ATE_OITO} := \text{UM_ATE_TRES} || \text{QUATRO_ATE_OITO}$$

O vetor resultante seria equivalente ao criado pela operação `array[1: 1,2,3,4,5,6,7,8]`.

4.1.2.6 Limites Superior e Inferior

Os limites de um vetor A podem ser conhecidos utilizando-se: **array_limh(A)** e **array_liml(A)** que retornam respectivamente, os limites superior (HI) e inferior (LO).

4.1.2.7 Tamanho

A operação **array_size(A)** determina o número de elementos de A, ou seja, $HI - LO + 1$.

4.1.2.8 Inserção de Elementos Extras

Um elemento só pode ser inserido em um vetor na primeira ou na última posição. A operação **array_addl(A,V)** retorna um vetor igual a A mas, com o índice mais baixo decrementado de 1 e o valor V inserido como primeiro elemento.

A inserção de um elemento na última posição é feita pela operação **array_addh(A,V)**, sendo que o índice mais alto é incrementado de 1 e V inserido como último elemento.

4.1.2.9 Remoção de Elemento

Da mesma forma que a inserção, a remoção de um elemento de um vetor pode apenas ser feita na primeira ou na última posição. As operações que removem um elemento são: **array_reml(A)** e **array_remh(A)**. Ambas retornam um vetor idêntico ao vetor A, porém na primeira o índice inferior é somado de 1 ($LO = LO + 1$) e na segunda o limite superior é decrementado de 1 ($HI = HI - 1$).

4.1.2.10 Ajuste

A operação **array_ajust(A,LO,HI)** retorna um array no intervalo de LO até HI contendo os mesmos dados de A. Por exemplo:

DOIS_ATE_SEIS := array_ajust(UM_ATE_OITO,2,6)

Retorna um vetor equivalente ao gerado pela operação **array[2: 2,3,4,5,6]**.

4.1.2.11 Ajuste do Limite Superior

Para alterar os valores dos índices de um vetor, pode-se utilizar a operação:

array_setl(A,LO)

Esta retorna o mesmo vetor A apenas o limite inferior passa a ser LO e todos os elementos são deslocados.

4.1.3 Grafos de Operações com vetores

Um vetor na MDFM é representado por um descritor formado por: endereço inicial (A), tamanho (S), limite inferior (lob) e offset. O offset permite que seja feita a modificação do descritor em operações que envolvem parte de um vetor. Em fluxo de dados puro, tais operações seriam feitas através de cópias de estruturas.

Ao invés de serem necessárias quatro fichas para o descritor de um array, pode-se trabalhar com apenas duas: uma contendo A e S, chamada de ponteiro, e outra contendo lob e offset combinados num único valor (na nomenclatura de Manchester este é o tipo J - dois inteiros de 16 bits).

A seguir são descritos os principais grafos que implementam cada operação envolvendo vetores em SISAL, e que utilizam o campo de índice.

4.1.3.1 Criação

Os valores dos índices dos elementos são conhecidos em tempo de compilação e são literais na entrada da instrução SIX (*Set Index*). O tamanho também é conhecido em tempo de compilação e é entrada da operação de ARMAZENAMENTO; já o lob e offset combinam-se na instrução CRJ (*Create J-value*) e formam a segunda parte do descritor (figura 4.7).

4.1.3.2 Preenchimento

Quando o vetor deve ser preenchido com elementos de mesmo valor são geradas N fichas com índice de 0 até N-1, através da instrução PRL, e o restante do grafo é idêntico ao da criação.

Figura 4.7: Grafo para Criação de um Vetor

4.1.3.3 Seleção

Figura 4.8: Grafo da Seleção de um Elemento de um Vetor

O I -ésimo elemento a ser selecionado é indexado dentro da UAE de acordo com **lob** e **offset** do vetor (figura 4.8). Assim, a instrução MIV (*Modify Indexing Value*) obtém o valor de indexação que então, junta-se ao ponteiro na instrução RSS, que finalmente retorna o valor lido na UAE de $A[I]$.

4.1.3.4 Troca de Valores

A instrução MIV (*Modify Indexing Value*) combina o índice do vetor com **lob** e **offset** para ter acesso ao elemento que é extraído do vetor através da instrução ESS (*Extract from Structure Store*) (figura 4.9). A garantia que o elemento foi extraído antes que o novo valor seja escrito é obtida utilizando SYN (*SYN*chronize). O endereço do elemento a ser escrito é calculado na instrução MAP (*Modify Address Pointer*) e com este endereço o novo valor é escrito utilizando a instrução WDS (*Write Directly to Structure Store*).

Figura 4.9: Grafo para Trocar o Valor de um Elemento de um Vetor

4.1.3.5 Concatenação

Para implementar a concatenação de dois vetores é feito a BUSCA dos dois vetores a serem concatenados, sendo que o vetor utilizado como segundo argumento da operação de concatenação tem o índice de seus elementos somados ao tamanho do primeiro vetor. Os elementos dos dois vetores são enviados à instrução WSS que armazena o novo vetor criado

pela concatenação, sendo que o tamanho deste novo vetor será a soma dos tamanhos dos dois vetores que o formaram. O **lob** do novo fica igual ao do primeiro e o **offset** é zerado.

4.2 Otimizações de Blocos Repetitivos

4.2.1 Eliminação de Rotulação

Böhm e Sargeant [BoSa 86] estudaram uma possibilidade para redução do “overhead” introduzido pela rotulação de fichas através de uma otimização que utiliza a diferenciação de dois tipos de ciclo: os sensíveis e os não sensíveis ao índice. Um ciclo é dito sensível quando o seu corpo altera o índice de suas fichas; caso contrário ele é dito insensível ao índice. A causa desta sensibilidade pode ser a inclusão de **streams**, ciclos internos, chamadas de funções e algumas operações com vetor. No caso de ciclos insensíveis ao índice, a distinção dos ciclos pode ser feita apenas pelo índice, tornando a geração de nomes de ativação e a rotulação dos parâmetros desnecessárias.

4.2.2 Limitação de Iterações Ativas

Foi observado que alguns programas possuem um alto grau de paralelismo o que acarreta uma grande ocupação de memória na MDFM. O mecanismo de “*Throttle*” [Rugg 87] tenta minimizar este problema através de técnicas para controlar a atividade da máquina. Uma destas propostas é a limitação do número de fichas produzidas por uma instrução do tipo PRL em um determinado tempo. No trabalho referido, foram realizadas simulações para um método que seguia as seguintes regras:

- A instrução de proliferação sempre podia produzir N_t fichas, e quando fosse necessário produzir mais seria enviada uma requisição a um alocador de processo (que para implementação do mecanismo de “*Throttle*” torna-se necessário).
- Uma requisição era considerada como um sinal de terminação de um processo. Assim, se a atividade da máquina permitisse, mais N_t fichas poderiam ser habilitadas para serem executadas.

Para a simulação foi utilizado o programa em SISAL para criação de um vetor. Os resultados mostraram que o tempo de execução aproximou-se muito do tempo da versão

onde não havia sido aplicado o método, apesar do aumento do número total de instruções executadas S1.

A extensão deste método é interessante para controlar o número de iterações ativas durante um determinado tempo, e conseqüentemente, torna-se importante para a redução dos campos do rótulo. Uma análise detalhada será apresentada no próximo capítulo.

4.3 Vetorização

Uma abordagem diferente para o tratamento de ciclos e operações com estruturas de dados é encontrada em modelos onde existe processamento vetorial. Processadores vetoriais possuem como característica principal a exploração da natureza repetitiva que existe em longas sequências de operações idênticas, obtendo uma execução mais eficiente [AlGo 89]. A idéia é que depois de um tempo inicial para realização da busca dos elementos a operação possa ser executada “paralelamente”. Este tipo de modelo é usualmente classificado como SIMD (*Single Instruction stream/ Multiple Data stream*) [Flynn 72]. Exemplos de computadores com processador vetorial são: IBM 3090, CRAY X-MP e Cyber 205.

O desempenho de máquinas vetoriais apresentam uma relação direta com a quantidade de operações vetoriais que podem ser exploradas pelo hardware. Normalmente, a tarefa de explicitar trechos do programa, que podem ser executados pelo processador vetorial, fica a cargo do compilador. Técnicas para realização de otimizações em tempo de compilação, como a análise das dependências dos dados, são utilizadas pelos compiladores que geram códigos que conseguem aproveitar as vantagens da arquitetura [PaWo 86] [AlCo 76]. Compiladores com capacidade de “vetorizar” programas em linguagem FORTRAN são encontrados em computadores vetoriais, já que muitas aplicações numéricas são implementadas nesta linguagem. Assim sendo, os exemplos de vetorização de trechos de programas serão nesta linguagem.

Um exemplo bastante conhecido de um segmento de programa que pode ser vetorizado é visto a seguir. Este segmento soma dois vetores B e C e resulta num outro vetor A.

```
DO 10 I=1,N
  A(I) = B(I) + C(I)
10 CONTINUE
```

Como não existe uma dependência entre os elementos dos operandos e do resultado, o ciclo poderia ser transformado na seguinte instrução vetorial:

$$A(1:N) = B(1:N) + C(1:N)$$

A análise das relações de dependência de dados entre linhas do programa ou entre iterações de um ciclo são usadas para determinar quando duas operações podem ser executadas em paralelo. Por exemplo:

```
DO 10 I=1,N
```

$$A(I) = B(I)$$

$$C(I) = A(I) + B(I)$$

$$E(I) = C(I+1)$$

```
10 CONTINUE
```

O valor do elemento $E(I)$ não depende do valor calculado no mesmo ciclo $C(I)$, mas sim dos valores atribuídos ao vetor C anteriormente a execução do ciclo. Desta forma, as operações vetoriais equivalentes ao bloco iterativo seriam:

$$A(1:N) = B(1:N)$$

$$E(1:N) = C(2:N+1)$$

$$C(1:N) = A(1:N) + B(1:N)$$

Deve-se observar uma mudança na ordem de execução das operações vetoriais para que o resultado fosse mantido inalterado.

O exemplo a seguir, ilustra uma construção que só pode ser vetorizada parcialmente devido a existência de um ciclo no grafo de dependência, ou seja, existe uma relação entre as iterações.

```
DO 10 I=2,N
```

$$A(I) = B(I)$$

$$C(I) = A(I) + B(I-1)$$

$$E(I) = C(I+1)$$

$$B(I) = C(I) + 2$$

```
10 CONTINUE
```

Os elementos de C dependem dos elementos de B que são calculados no ciclo anterior. Depois da análise de dependência de dados poder-se-ia ter:

```

A(2:N) = B(2:N)
E(2:N) = C(3:N+1)
DO 10 I=2,N
    C(I) = A(I) + B(I-1)
    B(I) = C(I) + 2
10 CONTINUE

```

A “vetorização” do código realizada pelo compilador não é a única forma de explicitar operações vetoriais. Uma outra maneira seria dotar as linguagens de programação com operações sobre vetores e matrizes.

Capítulo 5

Redução da Chave de Emparelhamento

Serão descritas neste capítulo propostas para a eliminação de redundâncias na chave de emparelhamento. Este é o capítulo central deste trabalho, pois nele serão expostas todas as propostas elaboradas, juntamente com os resultados das simulações realizadas.

Será visto neste capítulo, como o campo de segmento pode tornar-se redundante, possibilitando sua eliminação. Também como o campo de índice pode ser reduzido através da análise das operações que se utilizam deste campo, explicadas no capítulo 4. O campo de nome de ativação, desde que seja utilizado o método de “Throttle”, pode também ser reduzido.

5.1 Eliminação do Campo de Segmento

O item 2.2.4 descreveu a unidade de instrução como sendo implementada por uma memória segmentada. Assim, para que uma ficha de dado consiga ter acesso à instrução correspondente, o seu campo de destino precisa ser formado por dois sub-campos: um de segmento e outro de *offset*. Neste item, será apresentada a condição pela qual a combinação dos campos de segmento e nome de ativação pode se tornar redundante e, posteriormente, será descrito como o campo de segmento pode ser suprimido em decorrência desta redundância.

Para expor como surge esta redundância, deve-se lembrar como é realizada uma chamada de função. Na seção 3.3.1 foi visto que, quando uma chamada de função é feita, um nome de ativação é gerado, e as fichas que disparam as instruções pertencentes

a esta função passam a ter em seu rótulo o novo nome de ativação. Assim, através deste campo, é possível distinguir ativações diferentes de uma mesma função.

Se todas as instruções de uma dada função forem alocadas em um mesmo segmento da UI, então o valor do campo de segmento permaneceria o mesmo para todas as fichas pertencentes a função. Sendo assim, para o emparelhamento a distinção das fichas de uma dada função, em relação aos campos de destino, seria feita apenas pelo *offset*.

Pela forma de separar as ativações de funções e pela imposição de que todas as instruções de uma dada função sejam alocadas em um mesmo segmento, pode-se concluir que durante a execução de uma dada ativação de uma função, tanto o campo de segmento como o nome de ativação seriam sempre os mesmos para todas as fichas.

Figura 5.1: Exemplo de chamadas de funções

Através deste raciocínio verifica-se que ao estabelecer uma relação de correspondência entre o segmento e o nome de ativação torna-se possível suprimir o campo de segmento. Para tanto, a cada chamada de função realizada, ou seja, a cada nome de ativação gerado seria preciso relacionar este com o segmento onde estariam contidas as instruções da função. A partir daí, o campo de segmento tornar-se-ia dispensável para a realização do emparelhamento, sendo que o segmento continuaria sendo útil apenas dentro da UI.

O exemplo a seguir ilustra como esta correspondência pode ser obtida.

Suponha que uma função f_a realize duas chamadas de função, uma para f_b e outra para f_c , enquanto a função f_b chama uma função recursiva f_d (como mostra figura 5.1). Suponha que estas funções estejam localizadas nos segmentos da UI da maneira apresentada na figura 5.2.

Figura 5.2: Exemplo de distribuição de funções na UI

Para armazenar a relação entre os segmentos e nomes de ativações gerados, torna-se necessário a existência de uma tabela de correspondência. Esta tabela precisa ser atualizada a cada chamada de função realizada, pois quando um pacote de fichas parceiras chega à UI, é preciso consultar a tabela de correspondência para obter (através do nome de ativação) o segmento onde se encontra a instrução à qual o pacote se destina. Para o exemplo, a relação de correspondência resultaria na tabela 5.1.

função	nome de ativação	segmento
f_a	1	0
f_c	2	0
f_b	3	1
f_d	4	1
f_d	5	1
f_d	6	1

Tabela 5.1: Correspondência entre segmento e nome de ativação

Porém, existem duas condições que precisam ser satisfeitas para que este esquema possa funcionar efetivamente. A primeira está relacionada com a alocação de todas as instruções de uma função em um mesmo segmento, o que pode ser facilmente obtido inserindo esta condição no gerador de código, já que a alocação das instruções na UI é feita estaticamente. Mais especificamente, cabe ao “LINKER” atribuir às instruções do código estático, endereços para armazená-las na UI.

Esta condição impõe que o tamanho máximo do código estático de uma função não

ultrapasse o tamanho do segmento (4K instruções). É importante ressaltar que programas inteiros, como o de multiplicação de matrizes, não possuem mais que 500 instruções, e portanto, raramente serão encontradas funções com tamanho superior a 4K instruções. Porém, caso algum programa apresente uma função maior que o segmento, o compilador deve quebrar a função em funções menores.

A segunda condição é encontrar uma maneira de informar o valor do segmento à instrução GAN, para que esta possa atualizar a tabela de correspondência, quando executada. A solução deste problema também pode ser obtida em função da distribuição das instruções na UI ser feita estaticamente. O valor do segmento onde se encontra a função que está sendo ativada pela instrução GAN, pode ser transmitido como um segundo argumento de entrada da instrução GAN (figura 5.3).

Desta forma, quando a instrução GAN fosse executada, ela retornaria um novo valor de nome de ativação ao anel e, simultaneamente, enviaria uma mensagem para a tabela de correspondência, associando o novo nome de ativação ao segmento que foi colocado como segunda entrada.

Figura 5.3: Proposta de Modificação para Instrução GAN

Em decorrência do fato da tabela de correspondência ser necessária apenas dentro da UI, surge uma questão: não seria mais conveniente executar a instrução GAN dentro da UI? Para responder a esta pergunta é necessário considerar como seria feita a atualização da tabela. Se a instrução GAN fosse executada na própria UI, então a tabela de correspondência ficaria ligada ao anel apenas através desta unidade, e a atualização da tabela seria feita localmente. Um esquema de como seria a nova UI pode ser visto na figura 5.4.

Caso a instrução GAN continuasse a ser executada pela UP, seria necessário que a tabela possuísse duas ligações com o anel: uma com a UP para o envio de mensagens

Figura 5.4: Inserção da Tabela Localmente na UI

de atualização, e outra com a UI para a efetuar as consultas (figura 5.5). É importante ressaltar que é preciso garantir que a alteração da tabela seja realizada antes que alguma ficha contendo o novo nome de ativação necessite utilizar a UI.

Com a implementação da tabela de correspondência, torna-se possível eliminar o campo de segmento da chave de emparelhamento, reduzindo este em 6 bits.

Figura 5.5: Inserção de Tabela junto com a UP

5.2 Eliminação do Campo de Índice

A utilização do campo de índice é necessária para separar elementos de uma mesma estrutura de dados, de forma a permitir que as operações sobre os elementos de uma mesma estrutura sejam independentes e, portanto possam ser executadas paralelamente.

A princípio, acreditava-se que o alto grau de paralelismo gerado por este tipo de implementação poderia ser plenamente aproveitado pelo hardware. A experiência, porém, não confirmou esta expectativa. Na verdade, o que os estudos com a MDFM e outras arquiteturas a fluxo de dados demonstraram foi que quando um alto grau de paralelismo é exposto por um programa, existe a necessidade de impor algum tipo de controle, de forma a não permitir que o número de fichas no anel aumente muito, pois isto ocasiona uma elevação na ocupação de memória, principalmente da UE [Rugg 87]. Devido a sua complexidade, a UE é a unidade que mais sofre degradação com este aumento de fichas, comprometendo muito o tempo gasto para o emparelhamento de fichas parceiras. Como consequência a UI não consegue alimentar todos os processadores com pacotes executáveis e assim, por alguns ciclos de máquina, poderão existir processadores ociosos. Quando o fato descrito acima ocorre, diz-se que ocorreu uma “bolha” no *pipeline*.

A construção da UAE, embora tenha reduzido o número de cópias de estruturas necessárias para o processamento destas, não retirou do anel muitos tipos de operações vetoriais. A execução deste tipo de operação requer que estruturas armazenadas sejam transformadas em cadeias de fichas (*streams*). Assim, se as estruturas envolvidas em uma operação vetorial possuem um elevado número de elementos, a execução desta pode acarretar um súbito aumento do número de fichas no anel, o que pode implicar em um aumento da ocupação da UE, ocasionando os problemas citados anteriormente.

Outra desvantagem deste tipo de abordagem é que operações vetoriais simples envolvem um número elevado de instruções para sua realização. Por exemplo, para somar dois vetores, é necessário realizar a operação de BUSCA dos elementos dos vetores envolvidos. A operação de COLETA é realizada conjuntamente, para verificar se todos os elementos foram trazidos para o anel. Com todos os elementos dos vetores na forma de fichas é efetuado o processamento. O vetor resultante é armazenado na UAE, e novamente, a operação de COLETA deve ser executada. Verifica-se então, que existe um “overhead” muito grande pelo fato da execução de operações vetoriais serem efetuadas pelo anel.

O problema com a implementação das operações vetoriais na MDFM fica mais visível se for comparado o seu desempenho com aquele obtido por computadores com processamento vetorial como CRAY X-MP, IBM-3090, etc [Schö 87].

A proposta para obter um melhor desempenho da MDFM seria a adaptação de uma

unidade especial para o processamento vetorial, sendo que algum estudo a este respeito já foi realizado [Sa 90]. Portanto, a proposta deste trabalho é executar operações sobre estruturas nesta nova unidade.

Para este trabalho, o interesse em uma unidade de processamento vetorial não está apenas relacionado com a possível melhora no desempenho na execução de operações vetoriais, mas também porque isto viabiliza a eliminação do campo de índice neste tipo de operação. Com a adaptação de uma unidade de processamento vetorial, a execução deste tipo de operação seria transferida para esta nova unidade, eliminando a necessidade do campo de índice na implementação de operações vetoriais.

Porém, algumas operações básicas envolvendo vetores não podem ser executadas com processamento vetorial, sendo então necessário analisar formas de implementar estas operações sem utilizar o campo de índice. Os próximos itens apresentarão sugestões para modificações nos grafos que utilizam o campo de índice.

Outro caso onde é necessária a utilização do campo de índice é na implementação de blocos de repetição, sendo preciso propor uma maneira diferente para realizar a distinção dos ciclos sem utilizar o índice. A descrição e análise das propostas serão apresentadas nos itens seguintes.

5.2.1 Operações com Estruturas de Dados

Algumas operações básicas com estruturas de dados armazenadas dispensam o campo de índice para ter acesso a um determinado elemento. Um exemplo é a operação de seleção (figura 4.8). Esta utiliza-se do campo de dado de uma ficha especialmente criada, para informar o valor de indexação do elemento selecionado.

Outras operações, porém, empregam o índice para especificar qual elemento está sendo referenciado dentro da estrutura. Estas operações são: CRIAÇÃO, PREENCHIMENTO e COLETA. Estas operações serão estudadas separadamente nos itens a seguir, onde serão verificadas algumas maneiras de eliminar o uso do campo de índice.

Posteriormente, será feita uma análise a respeito de uma forma geral para solucionar o problema, ou pelo menos amenizá-lo, reduzindo o tamanho do campo de índice.

5.2.1.1 Criação

A operação de criação de um vetor (em SISAL `array[1: EL1,EL2,..., ELN]`) gera o grafo mostrado na figura 4.7. Pode-se observar que a mudança do valor do índice é feita explicitamente através da instrução SIX (*Set IndeX*), pois o valor de indexação de cada elemento bem como o tamanho da estrutura são conhecidos em tempo de compilação.

Figura 5.6: Proposta para Criação de um Vetor

Dentro da operação de ARMAZENAMENTO (vide figura 4.1) o endereço correto para armazenar cada elementos dentro da UAE, também é conseguido através do campo de índice, pois pela instrução WSS cada elemento é armazenado na posição: endereço inicial da estrutura + índice + 1.

Uma forma alternativa de implementar esta operação seria fazer um grafo do tipo daquele que realiza a troca de valor de um elemento de um vetor (figura 4.9). A proposta (figura 5.6) seria transmitir o valor de indexação pelo campo de dado de uma ficha, calculando o endereço onde cada elemento deve ser armazenado pelas instruções MIV e

MAP. A primeira instrução faz a combinação do valor de indexação com o *lob* e o *offset*, fornecendo à instrução MAP o deslocamento que deve ser somado ao endereço inicial da estrutura. Este endereço, quando chega na instrução WDS (*Write Direct to Structure Store*), indica o endereço correto onde o dado deve ser armazenado.

É necessário observar que ao invés de todos os dados serem armazenados por uma única instrução de escrita, devem existir, para este tipo de implementação, uma instrução WDS para cada elemento do vetor. Isto não chega ser um problema, pois na operação de criação o número de elementos do vetor é conhecido estaticamente e o compilador é capaz de inserir no código o número necessário de instruções de escrita.

5.2.1.2 Preenchimento

Este tipo de operação (em SISAL `array_fill(1,N,valor)`) cria um vetor com N elementos sendo que todos recebem o mesmo valor. Como o tamanho do vetor e o valor a ser atribuído aos seus elementos podem ser conhecidos apenas em tempo de execução, o grafo proposto para o PREENCHIMENTO não pode ser o mesmo proposto para a CRIAÇÃO, pois existe a impossibilidade de inserir no código uma instrução WDS para cada elemento do vetor.

O grafo desta operação utilizado atualmente é bastante otimizado, pois todas as fichas dos elementos são geradas por apenas uma instrução PRL que informa o valor de indexação dos elementos pelo campo de índice.

Esta operação, sem a utilização do campo de índice, pode ser implementada de duas maneiras diferentes. A primeira seria sequencializar a geração e o armazenamento dos elementos. A figura 5.7 mostra como esta proposta pode ser implementada.

É evidente que este tipo de modificação não só torna esta operação menos paralela, como também aumenta o número de instruções executadas. Contudo, considerando-se que um vetor deve ser iniciado apenas uma vez durante toda a execução do programa, o aumento do número de instruções na operação de PREENCHIMENTO não deve representar um aumento considerável para o número total de instruções executadas pelo programa inteiro.

Outra possibilidade para esta operação seria realizá-la totalmente dentro da unidade de processamento vetorial, bastando para isto uma instrução vetorial que informasse o

valor e o tamanho da estrutura à unidade de processamento vetorial e que retornasse um ponteiro para a estrutura criada.

Figura 5.7: Proposta para Preechimento de um Vetor

Para mostrar como esta operação seria efetuada por um processador vetorial convencional, será apresentada a implementação do VAX-Vector 6000-420. Para executar o comando `SISAL array_fill(1,32,5)`, o processador vetorial VAX-Vector utilizaria a seguinte instrução vetorial [VAXV 90]:

```
VSMERGE #5,V0,V0
```

Esta instrução combina o escalar 5 com o registrador vetorial V0, sendo que este não é iniciado com nenhum vetor anteriormente. O resultado da operação VSMERGE é um

vetor de comprimento definido em um registrador que armazena o comprimento, o VLR (*Vector Length Register*) que, para o exemplo, recebe o valor 32.

O vetor resultante desta combinação é, no exemplo, o mesmo vetor V0, sendo que cada elemento será igual ao escalar 5 se o bit correspondente de um registrador de máscara, VMR (*Vector Mask Register*) for zero, ou será o valor do elemento do vetor V0 caso o bit de VMR for 1. Assim, para que seja criado um vetor com todos os elementos iguais ao valor 5, o registrador VMR deve ser iniciado com todos os bits iguais a zero.

Através deste exemplo, pode-se concluir que esta operação pode ser executada por um processador vetorial sem ser necessária a utilização do índice.

5.2.1.3 Coleta

Esta operação é realizada quando uma estrutura na forma de cadeia de fichas (*stream*), deve ser processada pelo anel, sendo que sua função é verificar se todos os elementos da estrutura foram gerados. A implementação das operações sobre vetores na MDFM exige que cada operação realize conjuntamente, uma operação de COLETA, pois os elementos do vetor para serem processados precisam ser trazidos para o anel na forma de uma cadeia de fichas.

As operações básicas com vetores que precisam efetuar conjuntamente a operação de COLETA são: a de PREENCHIMENTO e de BUSCA.

Se todas as operações sobre estruturas de dados passarem a ser executadas na unidade de processamento vetorial, a operação de BUSCA, que traz os elementos do vetor ao anel, deixaria de ser necessária. Isto implica que a operação de COLETA não mais seria realizada.

No entanto, é preciso analisar o caso quando a operação não pode ser efetuada vetorialmente. Este caso será discutido na seção 6.1, bem como a forma como seria realizada a operação de COLETA.

Para a operação de PREENCHIMENTO foram apresentadas duas implementações que não utilizam o campo de índice. Em ambos os casos, a operação de COLETA dos elementos torna-se desnecessária. No primeiro caso o PREENCHIMENTO é feito de modo sequencial. Assim, a própria formação do vetor já garantiria a geração de todos os elementos. Para o caso em que o PREENCHIMENTO do vetor ficasse a cargo da unidade de processamento vetorial não caberia mais ao anel verificar a correta formação do vetor.

5.2.2 Modificações na Implementação de Blocos de Repetição

Para que o nível de iteração pudesse ser suprimido, a implementação de blocos de repetição passou a ser feita combinando o nome de ativação e o índice (figura 3.3). Agora, deseja-se também eliminar o campo de índice e para isto, é necessário encontrar uma forma alternativa para implementar a separação dos ciclos.

Para analisar o problema de forma mais cuidadosa é conveniente separar-se os dois tipos de blocos repetitivos. O primeiro caracteriza-se pela dependência de dados entre os ciclos (em SISAL, seria a construção *for initial*) e o segundo pela total independência entre os ciclos (construções do tipo *forall*).

A eliminação do índice na implementação do primeiro tipo de bloco repetitivo pode ser conseguida pela sequencialização dos ciclos. Esta modificação não deve afetar a eficiência com a qual este tipo de construção é executada, pois esta é a forma natural de execução imposta pela própria dependência de dados entre os ciclos. A sequencialização pode ser implementada inserindo no final do grafo de cada bloco repetitivo instruções de sincronização que garantiriam o término de execução de todas as instruções de um ciclo para só então, iniciar o próximo ciclo.

Para o caso do "*forall*", a melhor solução seria realizar a vetorização do bloco. Para tanto é necessário que o compilador possa destacar quais seriam os blocos repetitivos que poderiam ser executados vetorialmente. Assim, as técnicas desenvolvidas para os compiladores FORTRAN que geram código para computadores com processamento vetorial (apresentadas no item 4.4) devem ser adaptadas ao gerador de código da máquina de fluxo de dados. Porém, mesmo com a utilização destas técnicas serão encontrados tipos de repetições que, embora não possuam dependência de dados entre os ciclos, não poderão ser vetorizadas pelo compilador. Um exemplo deste tipo de construção seria o trecho de programa a seguir:

```
function pmult(A,B:VETOR_REAL returns VETOR_REAL)
```

```
  let
```

```
    na:=array_size(a);
```

```
    nb:=array_size(b)
```

```
  in
```

```
    for k in 1,na+nb-1
```

```
      cf:=
```

```
        for i in 1,na
```

```
          el:=
```

```
            for j in 1,nb
```

```
              el1:=
```

```
                if i+j:=(k+1) then A[i]*B[j]
```

```
                else 0.0
```

```
                end if
```

```
              returns value of sum el1
```

```
            end for
```

```
          returns value of sum el
```

```
        end for
```

```
      returns array of cf
```

```
    end for
```

```
  end let
```

```
end function
```

Os blocos repetitivos, provavelmente não seriam vetorizados pelo compilador em virtude da existência do teste condicional no bloco mais interno. Contudo, pela geração de código da MDFM os ciclos poderiam ser executados paralelamente, já que com qualquer um dos resultados da condicional de cada ciclo, não existe dependência de dados com os demais ciclos.

Através deste exemplo chega-se a seguinte questão: como executar construções repetitivas quando estas não puderem ser vetorizadas? Existem duas soluções para este problema: a primeira seria utilizar apenas o campo de nome de ativação para distinguir os

ciclos, o que será discutido na seção 5.2.2.2.

A segunda solução seria executar as construções não vetorizadas pelo compilador de modo sequencial. O problema é que sequencialização de construções paralelas pode degradar muito a eficiência com que estas podem ser executadas.

5.2.2.1 Limitação de Iterações Ativas

Uma possível solução seria fazer uma adaptação do mecanismo de controle de paralelismo apresentado no item 4.2.2. Por este método, o número de fichas geradas por uma instrução de proliferação, em um certo tempo, seria limitado em N_t fichas. Seria liberada a geração de mais N_t fichas somente quando fosse determinado o término da execução das fichas geradas anteriormente.

Como a implementação atual de blocos de repetição utiliza-se de “geradores” para criar uma ficha de cada parâmetro para cada ciclo, e sendo que estes “geradores” nada mais são do que instruções de proliferação (por exemplo, PRL) então, de forma análoga ao método de controle de paralelismo, poder-se-ia limitar o número máximo de fichas geradas de uma só vez por instruções de proliferação. Esta solução permite que o campo de índice possa ter seu tamanho reduzido de forma bastante satisfatória, embora não descarte sua utilização.

Este tipo de limitação pode degradar muito a eficiência com que os ciclos podem ser executados, pois quando os N_t ciclos estiverem terminando sua execução, mesmo que existam processadores sem trabalho, deve-se esperar até que a última ficha, destes ciclos, seja consumida, para então, liberar mais N_t ciclos para a execução. Como consequência deve ocorrer um aumento do tempo de execução em função da espera para liberação de ciclos. Assim, esta proposta não pode ser considerada como ideal para substituir a atual implementação de blocos repetitivos.

5.2.2.2 Transformação de Ciclos em Funções Recursivas

Supondo que a eliminação do campo de índice seja realizada, o único campo do rótulo que seria mantido seria o nome de ativação. Assim, quando um bloco repetitivo não pudesse ser vetorizado, a única maneira de implementar a separação dos ciclos seria utilizar o

nome de ativação.

Através desta suposição, pode-se sugerir que uma construção repetitiva possa ser tratada como uma chamada recursiva de função. Porém, é fácil intuir que este tipo de implementação será menos eficiente em decorrência do aumento do número de instruções necessárias para atribuir um novo rótulo a cada ciclo.

O programa SISAL abaixo foi artificialmente criado para possibilitar a comparação do desempenho das várias propostas que serão apresentadas.

```
function main(returns integer)
```

```
  let
```

```
    k := 4;
```

```
    l := 2
```

```
  in
```

```
    for i in 1,N
```

```
      e := i * k * l
```

```
    returns value of sum e
```

```
  end for
```

```
end let
```

```
end function
```

Este tipo de construção `for` poderia ser executada vetorialmente, mas pela sua simplicidade é conveniente utilizá-la para exemplificar como as alterações propostas poderão ser implementadas.

A tabela 5.2 mostra como se comporta a execução do grafo que implementa atualmente a construção `for` em função do número de ciclos que devem ser executados. A eficiência com a qual o programa é executado é medida através de parâmetros fornecidos pelo simulador. Pode-se notar que o paralelismo se mantém constante mesmo quando o número de ciclos executados chega a 1024.

O programa a seguir implementa a construção repetitiva como uma chamada recursiva de função. Os resultados obtidos pelo simulador estão na tabela 5.3, pela qual pode-se observar que S_1 e S_{inf} aumentam muito quando o número de ciclos cresce. O valor de S_1 torna-se mais que 5 vezes maior do que a implementação original no caso em que 1024 ciclos são executados. Este comportamento de S_1 não é desejável, pois implica em um

N	S_1	S_{inf}	π
1	29	15	1.9
2	33	17	1.9
5	45	23	2.0
10	65	33	2.0
20	105	53	2.0
40	185	93	2.0
60	265	133	2.0
1024	4121	2061	2.0

Tabela 5.2: Simulação da forma atual de executar forall

aumento excessivo de instruções envolvidas na implementação de uma única construção.

```

function rec ( i,k,l :integer returns integer)

    if i < 1 then 0
    else i * k * l + rec(i-1,k,l)
    end if
end function

function main(returns integer)

    rec(N,4,2)
end function

```

N	S_1	S_{inf}	π
1	45	24	1.9
2	66	35	1.9
5	129	68	1.9
10	234	123	1.9
20	444	233	1.9
40	864	453	1.9
60	1284	673	1.9
1024	21528	11277	1.9

Tabela 5.3: Simulação da execução recursiva do forall

A transformação de uma construção *for* em uma chamada de função duplamente recursiva poderia resultar em um melhor desempenho, pois o grafo gerado por este tipo de função é mais paralelo que o gerado por uma função recursiva simples.

O programa abaixo, com uma função duplamente recursiva, foi simulado e os resultados estão na tabela 5.4. Estes demonstram que, embora exista uma diminuição do caminho crítico (S_{inf}) para um número grande de ciclos executados, o valor de S_1 torna-se até 2 vezes maior que o obtido pela função recursiva simples. Pode-se observar que há um aumento muito bom do paralelismo chegando a ser 150 vezes maior do que a atual forma de implementar ciclos. Contudo, é importante notar que como S_1 é muito grande em comparação com a implementação original, esta possibilidade não pode ser considerada como uma solução viável, pois a execução um programa com várias construções repetitivas envolveria um número muito grande de instruções.

```
function rec ( n,i,k,l :integer returns integer)
```

```
  if n <= i then i*k*l
```

```
  else rec(n,(n+i)/2+1,k,l) + rec((n+i)/2,i,k,l)
```

```
  end if
```

```
end function
```

```
function main(returns integer)
```

```
  rec(N,1,2,4)
```

```
end function
```

Uma otimização que poderia ser utilizada para melhorar o código gerado, seria forçar que o código do bloco repetitivo fosse inserido no código do programa algumas vezes, evitando assim, realização da rotulação de fichas para todos os ciclos. Quando esta otimização é realizada diz-se gerar um código "*inline*".

A própria tabela 5.4, mostra os resultados obtidos pela simulação do programa duplamente recursivo, quando o código da função é colocado *inline* no programa. Como pode ser observado, quando o código do bloco repetitivo é inserido 3 ou 6 vezes no código do programa não há uma grande melhora quanto ao problema do número de instruções a serem executadas, S_1 . Desta forma, a transformação de blocos repetitivos em chamadas recursivas de função gera um código menos eficiente que o atual.

				<i>inline = 3</i>			<i>inline = 6</i>		
N	S_1	S_{inf}	π	S_1	S_{inf}	π	S_1	S_{inf}	π
1	26	14	1.9	19	10	1.9	19	10	1.9
2	71	27	2.6	50	20	2.5	50	20	2.5
5	206	53	3.9	171	42	4.1	157	42	3.7
10	431	66	6.5	354	55	6.4	347	54	6.4
20	881	79	11.2	748	65	11.5	713	65	11.0
40	1781	92	19.4	1480	78	19.0	1431	76	18.8
60	2681	92	29.1	2380	79	30.1	2149	76	28.3
1024	46061	144	319.9	41280	125	330.2	37185	120	309.9

Tabela 5.4: Simulação para execução duplamente recursiva

5.2.2.3 Chamadas de Funções com Ciclos Limitados

				<i>inline = 3</i>			<i>inline = 6</i>		
N	S_1	S_{inf}	π	S_1	S_{inf}	π	S_1	S_{inf}	π
1	50	26	1.9	43	22	2.0	43	20	2.0
2	54	28	1.9	47	24	2.0	47	24	2.0
5	131	47	1.9	117	43	2.7	110	39	2.8
10	216	59	3.7	202	53	3.8	188	54	3.5
20	386	83	4.7	365	75	4.9	344	72	4.8
40	791	143	5.5	749	131	5.7	721	130	5.5
60	1196	203	5.9	1140	185	6.2	1084	186	5.8
1024	20717	3095	6.7	19814	2837	7.0	18925	2836	6.7

Tabela 5.5: Simulação para execução duplamente recursiva com LIM = 4

Embora os resultados das simulações não tenham sido favoráveis à aceitação da proposta de transformar um bloco repetitivo em uma chamada duplamente recursiva de função, esta proposta pode ser melhorada se for combinada com a proposta anterior, ou seja, executar um bloco repetitivo de forma duplamente recursiva, mas deixando que um número limitado de ciclos possa ser executado utilizando o campo de índice. A simulação do programa a seguir possibilitou um estudo comparativo a respeito do número máximo de ciclos que podem ser executados da forma atual, designado pelo nome LIM.

N				<i>inline</i> = 3			<i>inline</i> = 6		
	S_1	S_{inf}	π	S_1	S_{inf}	π	S_1	S_{inf}	π
1	50	26	1.9	43	22	2.0	43	20	2.0
2	54	28	1.9	47	24	2.0	47	24	2.0
5	66	34	1.9	59	30	2.0	59	30	2.0
10	151	55	2.7	137	51	2.7	130	47	2.8
20	256	67	3.8	242	61	4.0	228	62	3.7
40	466	91	5.1	445	83	5.4	424	80	5.3
60	741	127	5.8	706	117	6.0	678	113	6.0
1024	12395	1567	7.9	11942	1437	8.3	11494	1433	8.0

Tabela 5.6: Simulação para execução duplamente recursiva com LIM = 8

```

function rec ( n,i,k,l :integer returns integer)

  if (n-i+1) <= LIM then
    for j in i,LIM
      r:= j * k * l
    returns value of sum r
    end for
  else
    rec(n,i+LIM,k,l) + rec(i+LIM-1,i,k,l)
  end if
end function

function main(returns integer)
  rec(N,1,2,4)
end function

```

As tabelas 5.5, 5.6, 5.7 e 5.8, mostram que o problema quanto ao aumento de S_1 é bastante amenizado quando o número de ciclos, que podem ser executados da forma atual, aumenta, chegando a ser obtido um excelente resultado para 32 ciclos. Com este valor, S_1 tem um aumento de apenas 50% em relação a implementação original, mas com um aumento de 6 vezes do paralelismo, π . Os resultados obtidos para o caso da geração de código “*inline*” demonstram que esta técnica acentua a diminuição de S_1 , e determina

um aumento, embora pequeno, do paralelismo.

				<i>inline</i> = 3			<i>inline</i> = 6		
N	S_1	S_{inf}	π	S_1	S_{inf}	π	S_1	S_{inf}	π
1	50	26	1.9	43	22	2.0	43	20	2.0
2	54	28	1.9	47	24	2.0	47	24	2.0
5	66	34	1.9	59	30	2.0	59	30	2.0
10	86	44	2.0	79	40	2.0	79	40	2.0
20	191	71	2.7	177	67	2.6	170	63	2.7
40	336	83	4.0	322	77	4.2	308	78	3.9
60	481	95	5.1	460	89	5.2	453	88	5.1
1024	8237	815	10.1	8006	749	10.7	7789	748	10.4

Tabela 5.7: Simulação para execução duplamente recursiva com LIM = 16

				<i>inline</i> = 3			<i>inline</i> = 6		
N	S_1	S_{inf}	π	S_1	S_{inf}	π	S_1	S_{inf}	π
1	50	26	1.9	43	22	2.0	43	22	2.0
2	54	28	1.9	47	24	2.0	47	24	2.0
5	66	34	1.9	59	30	2.0	59	30	2.0
10	86	44	2.0	79	40	2.0	79	40	2.0
20	126	64	2.0	119	60	2.0	119	60	2.0
40	271	103	2.6	257	99	2.6	250	95	2.6
60	315	103	3.4	33	99	3.4	330	95	3.5
1024	6157	463	13.3	6038	429	14.1	5926	425	13.9

Tabela 5.8: Simulação para execução duplamente recursiva com LIM = 32

A comparação dos resultados obtidos pode ser vista no gráfico da figura 5.8. Neste, pode-se observar a variação de S_1 e S_{inf} quando 1024 ciclos são executados, para valores de LIM iguais a 4, 8, 16 e 32. Tanto S_1 como S_{inf} diminuem com o aumento do número de ciclos que podem ser executados na forma atual. Porém, a redução de S_{inf} é maior do que S_1 , ocasionando um aumento de π , o que pode ser visto no gráfico da figura 5.9. Pode-se supor que um resultado ainda melhor poderia ser obtido para o caso de LIM ser igual a 64. Contudo, devido a necessidade de reduzir ao máximo o número de bits que devem compor a nova chave de emparelhamento esta possibilidade não será analisada.

Figura 5.8: Gráfico de S_1 e S_{inf} como função de LIM ($N = 1024$)

Pode-se concluir que o melhor resultado entre todas as propostas desta seção foi obtido quando a separação dos ciclos é feita de forma duplamente recursiva com 32 ciclos podendo ser executados na forma atual. Isto permite que campo de índice seja reduzido para 5 bits.

5.3 Redução do Campo de Nome de Ativação

Para realizar a redução do campo de nome de ativação deve-se considerar a utilização do método de Throttle pela MDFM [Rugg 87]. Este método, apresentado no item 2.3, consegue suspender a ativação de processos quando a máquina encontra-se saturada. Além disto, permite que nomes de ativações de processos já concluídos, possam ser reutilizados.

Com isto, torna-se possível reduzir o tamanho do campo de nome de ativação. A razão desta redução vem do fato de que a saturação da máquina, normalmente, dá-se quando muito menos que 4K processos estão ativos, sendo este o número máximo de funções que podem estar ativas simultaneamente.

A proposta, para este campo, é reduzi-lo de 12 para 10 bits, o que implicaria na limitação de 1K funções que poderiam estar executando paralelamente. Com este número máximo de funções ativas, o nível de atividade da máquina não deve ficar abaixo do valor

Figura 5.9: Gráfico de π como função de LIM ($N = 1024$)

ideal, obtendo-se um funcionamento eficiente da máquina.

Desta forma, a redução do campo de nome de ativação pode ser efetuada sem que isto implique na degradação da eficiência da MDFM.

5.4 Sumário

Através da determinação de redundâncias na chave de emparelhamento foi possível indicar a eliminação de dois campos do rótulo: o nível de iteração e o segmento. O nível de iteração pode ser substituído por uma combinação do nome de ativação e do índice.

Para eliminar o segmento é necessário garantir que todas as instruções de uma função sejam alocadas em um mesmo segmento da UI. Assim, torna-se possível estabelecer uma relação com o nome de ativação, que deve ser armazenada em uma tabela de correspondência. Para cada nome de ativação gerado a tabela precisa ser atualizada, pois é através desta que a UI poderá obter o segmento onde está armazenada a instrução a qual o dado se destina.

É possível que o campo de nome de ativação seja reduzido para 10 bits em função do controle da ativação de funções e do fato de ser possível reciclar valores deste campo.

O campo de índice foi o que apresentou maiores problemas para sua redução, pois

para tanto, primeiramente, deve-se inserir uma unidade especial para o processamento vetorial. A partir desta consideração, é preciso adaptar técnicas de compilação utilizadas em computadores vetoriais ao gerador de código na MDFM.

Algumas operações básicas com estruturas de dados não podem ser executadas por processadores vetoriais, tendo sido então, sugeridas formas alternativas de implementar estas operações de forma a não utilizar o campo de índice para indexar os elementos de estruturas.

Além de ser utilizado para separar elementos de uma mesma estrutura de dados, o índice auxilia a implementação de blocos de repetição. O seu uso seria evitado se fosse possível executar todo bloco repetitivo vetorialmente. Contudo, poderão ser encontrados casos em que a vetorização não poderá ser executada, principalmente em função da existência de condicionais internas aos blocos de repetições. Para estes casos a melhor implementação encontrada foi a transformação de blocos repetitivos em uma chamada duplamente recursiva de função, mas permitindo que 32 ciclos possam ser executados utilizando o índice.

Portanto, pelas propostas apresentadas neste capítulo a chave de emparelhamento passa a ser formada por 27 bits: 12 do *offset*, 10 do nome de ativação e 5 do índice.

Capítulo 6

Considerações Finais

6.1 Conclusão

Por sua complexidade a UE tornou-se um gargalo da MDFM. Visto que o tempo gasto para o agrupamento de fichas parceiras é alto, esta unidade impõe um limite máximo à velocidade de processamento que pode ser atingida pela máquina. O problema com a UE é agravado quando há um aumento considerável do número de fichas em sua memória. O aumento de fichas no anel dá-se, normalmente, quando há um crescimento de paralelismo exposto por um programa. Assim, o problema de emparelhamento de fichas parceiras torna-se bastante crítico para a própria viabilidade de utilização da arquitetura, pois o modelo a fluxo de dados tenta explorar o máximo de paralelismo existente em um programa.

O problema que ocorre na UE deve ser, se não eliminado, pelo menos amenizado com a diminuição do número de bits que devem ser comparados para o agrupamento de fichas parceiras. Como a memória pseudo-associativa que implementa a UE gera, a partir de 54 bits, um endereço de “hash” com apenas 16 bits, muitas combinações possíveis da chave de emparelhamento resultam em um mesmo endereço de “hash”. Isto faz com que a memória seja esparsamente ocupada, induzindo a ocorrências de “overflow”, mesmo quando a memória não está totalmente ocupada. Isto resulta em um aumento no tempo gasto no emparelhamento de fichas parceiras.

Se o endereço de “hash” passa a ser gerado a partir de um número menor de bits, mesmo que o esquema de implementação da UE permaneça inalterado, o problema oca-

sionado pelo aumento de fichas no anel deve ser bastante amenizado. A razão disto é que deverão ser gerados menos endereços coincidentes e portanto, a UE será menos afetada pelo aumento de paralelismo.

Para obter a redução da chave de emparelhamento foi, primeiramente, constatada a existência de redundâncias, possibilitando a eliminação de dois campos: o nível de iteração e o segmento.

A substituição do nível de iteração por uma combinação do nome de ativação e o índice foi feita em Manchester, e demonstrou não implicar em uma perda de eficiência na execução deste tipo de construção.

Para a eliminação do segmento deve haver um aumento de tarefas a serem executadas pela UI, pois é necessária a adaptação de uma tabela de correspondência. Esta deve ser utilizada pelos pacotes de fichas que chegam da UE e precisam obter, através do nome de ativação, o segmento onde encontra-se armazenada a instrução destino.

A inclusão da tabela na UI deve aumentar o tempo gasto na geração de pacotes executáveis. Porém, considerando que as operações com a tabela são feitas de modo independente das demais tarefas, estas podem ser efetuadas paralelamente. Por exemplo, enquanto é realizada uma consulta à tabela, outro pacote pode estar obtendo a sua instrução destino. Este fato deve minimizar a desvantagem de inserir esta nova tarefa à UI. Além disto, como a eliminação dos 6 bits de segmento favorece a diminuição do tempo necessário para o agrupamento de fichas parceiras pela UE, haverá, sem dúvida, um ganho de eficiência no desempenho da máquina como um todo.

O interesse em eliminar o campo de índice fez com que fosse analisado um sério problema do modelo a fluxo de dados, o tratamento dado a estruturas de dados. Como a exploração de paralelismo de operações entre os elementos de uma mesma estrutura demonstrou não ser eficientemente aproveitado pela máquina, foi proposta a adaptação de uma unidade especial para processamento vetorial. Esta alteração deve melhorar o desempenho da máquina na execução de operações com estruturas.

A transferência das operações sobre estruturas para esta nova unidade permite que o índice deixe de ser utilizado em muitas operações deste tipo. O índice também deve ser eliminado da implementação de construções repetitivas que passam a ser executadas pelo processador vetorial. Esta nova forma de abordar as construções que utilizam o campo de

índice em suas implementações retira do anel trechos de programas que potencialmente, podem obter melhores velocidades de execução em processadores vetoriais.

A existência de construções repetitivas que não podem ser vetorizadas fez com que fosse feita a análise de como executá-las no anel, sem que haja uma perda significativa de paralelismo. A melhor solução é executá-las como chamadas recursivas de função com a permissão de que 32 ciclos possam ser executados utilizando o índice. Esta forma, inclusive, expõe mais paralelismo que a maneira atual de implementar ciclos.

Esta limitação do campo de índice em 5 bits, permite que operações envolvendo estruturas de dados que não sejam vetorizadas pelo compilador, possam ser executadas pelo anel. Bastando para isto, que o compilador quebre estruturas maiores em estruturas com no máximo 32 elementos. A execução de operações sobre estruturas no anel faz com que a operação de COLETA (que utiliza o campo de índice) volte a ser necessária. Porém, é importante notar que todas as estruturas poderão possuir no máximo 32 elementos e, assim a operação de COLETA poderá ser executada utilizando o índice.

Conjuntamente, estas alterações propiciam que a chave de emparelhamento seja reduzida para metade de seu tamanho atual. Com os 27 bits que passam a formar a chave de emparelhamento torna-se possível a realização do emparelhamento direto, pois o espaço de memória endereçado pela chave tornou-se de apenas 128M posições.

Portanto, todas as propostas elaboradas neste trabalho, ou seja, a criação de uma tabela de correspondência dentro da UI, a transferência de operações vetoriais para uma unidade especial de processamento vetorial e a mudança na forma de implementar blocos de repetição garantem a eliminação da necessidade de existência de memória associativa na operação de emparelhamento de fichas parceiras, garantindo uma enorme melhora na eficiência da UE.

6.2 Trabalhos Futuros

Para que a redução da chave de emparelhamento seja efetivada é necessário a realização dos seguintes trabalhos.

A eliminação do segmento exige que seja estudada a melhor disposição para a inserção da tabela de correspondência no anel, sendo que possivelmente, a melhor implementação

seria realizar as operações com a tabela dentro da UI. Assim, para que as atualizações e as consultas à tabela fossem feitas pela própria UI é necessário que a instrução GAN seja executada nesta mesma unidade. Portanto, algumas modificações devem ser analisadas e efetuadas na UI, tentando evitar que esta unidade torne-se um novo gargalo.

Para ser possível eliminar o campo de índice deve-se realizar mudanças no compilador, pois é necessário alterar as implementações de algumas operação básicas sobre estruturas e de construções repetitivas que não podem ser vetorizadas.

Uma proposta apresentada neste trabalho, e que deverá ser motivo de muitas pesquisas é o tratamento dado às operações sobre estruturas de dados pela MDFM. A redução do campo de índice necessita que uma unidade para o processamento vetorial seja introduzida. Para a implementação desta nova unidade deve ser analisados alguns pontos:

- Seria interessante que a linguagem utilizada suportasse operações vetoriais explícitas;
- Deve-se decidir qual é a melhor forma de ligar esta nova unidade ao anel;
- É necessário decidir quanto a organização da unidade de processamento vetorial e estabelecer um protocolo de comunicação com o anel;
- A implementação da unidade de armazenamento de estruturas deve ser reconsiderada, pois as mudanças no processamento podem implicar em alterações nos requisitos desejados para o armazenamento de estruturas;
- Devem ser adaptadas ao compilador técnicas para que este possa destacar trechos de programas que possam ser vetorizados.

Bibliografia

- [Acke 82] Ackerman, W.B. *Dataflow Languages* IEEE Computer, 15(2): 15-25, fev. 1982.
- [AlCo 76] Allen, F.E. & Cocke, J. *A Program Data Flow Analysis Procedure* Communications of the ACM, 19(3): 137-147, mar. 1976.
- [AlGo 89] Almasi, G.S. & Gottlieb, A. *Highly Parallel Computing* cap. 9, pp. 303-310, The Benjamin/Cummings Publishing Company, Inc., 1989
- [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 Funcional 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.
- [BoGu 87] Böhm, A.P.W. & Gurd, J. *Tools for Performance Evaluation of Parallel Machines* University of Manchester, 1987.
- [BoSa 86] Böhm, A.P.W. & Sargeant, J. *Efficient Dataflow Code Generation for SISAL* documento interno, University of Manchester, 1986.
- [Bowe 81] Bowen, D.L. *Implementation of Data Structures on Dataflow Machines* P.h.D. Thesis, Department of Computer Science, University of Manchester, jun. 1981.

- [Catt 81] Catto, A.J. *Non-Deterministic Programming in a Dataflow Environment* Ph.D. Thesis, Department of Computer Science, University of Manchester, jun. 1981.
- [Denn 81] Dennis, J.B. *Models of Data Flow Computation* in Control Flow and Data Flow Concepts of Distributed Programming pp.346-354
- [Flynn 72] Flynn, M.J. *Some Computer Organization and their Effectiveness* IEEE trans. on Computers, C-21: 948-960, 1972.
- [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.
- [GuWa 83] Gurd, J. & Watson, I. *Preliminary Evaluation of a Prototype Dataflow Computer* Proceedings of the 9th World Computer Congress (IFIP 83), North Holland, pp545-551, set. 1983.
- [HoSa 79] Horowitz, E. & Sahni, S. *Fundamentals of Computer Algorithms* cap.3, pp. 98.151, Computer Science Press Inc., 1979
- [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.

- [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.
- [PaWo 86] Padua, D.A. & Wolfe, M.J. *Advanced Compiler Optimizations for Supercomputers* Communications of ACM, 29(12): 1184-1201, dez. 1986.
- [Rugg 87] Ruggiero, C.A. *Throttle Mechanisms for Manchester Dataflow Machine* Ph.D. Thesis, Department of Computer Science, University of Manchester, jul. 1987.
- [RuSa 85] Ruggiero, C.A. & Sargeant, J. *Hardware and Software Mechanisms for Control of Parallelism* Department of Computer Science, University of Manchester, abr. 1985.
- [Sa 90] Sá, M.P. *Armazenamento de Estruturas de Dados em Computadores a Fluxo de Dados* Dissertação de Mestrado, Departamento de Ciência da Computação, IMECC, Unicamp, dez. 1990.
- [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.
- [SarS 85] Sargeant, J. *Simulator Users Guide* University of Manchester, ver. 1, jan. 1985.
- [Schö 87] Schönauer, W. *Scientific Computing on Vector Computers* Special Topics in Supercomputing, vol. 2, North-Holland, 1987.
- [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.

- [SiWa 83] da Silva, J.G.D. & Watson, I. *Pseudo-Associative Store with Hardware Hashing* IEE Proc., 130(1): 19-24, jan. 1983
- [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.
- [SkWe 85] Skedzielewski, S.K. & Welcome, M.L. *Dataflow Graph Optimization in IF1* Functional Programming Languages and Computer Architecture, LNCS, 201: 17-34, Springer-Verlag, set. 1985.
- [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.
- [VAXV 90] *VAX Vector Processing Handbook* Digital Equipment Corporation, 1990.
- [Veen 85] Veen, A. *Locality in Matching Store* Department of Computer Science, University of Manchester, jul. 1985.
- [WaGu 82] Watson, I. & Gurd, J. *A Practical Dataflow Computer* IEEE Computer, 15(2): 51-57, fev. 1982.

Apêndice A

Definições das Instruções da MDFM

Este apêndice apresenta a descrição de instruções da MDFM utilizadas nos grafos estudados na elaboração do trabalho de redução da chave de emparelhamento.

A princípio, serão definidas abreviaturas para os tipos pré-definidos necessárias para o entendimento do funcionamento das instruções. Também, serão apresentadas abreviaturas utilizadas nas definições das instruções.

As instruções serão separadas de acordo com sua utilização.

A.1 Tipos de Dados

tipo de dado	abreviatura	especificação
inteiro	I	$-2^{23} - 2^{23}$
real	R	ponto flutuante menor que 2^{27}
lógico	B	True ou False
nome de ativação	A	0 até $2^{12} - 1$
contexto	W	nome de ativação e destino
destino	D	destino completo
ponteiro da UAE	Q	endereço e tamanho da estrutura
par-inteiros	J	2 inteiros de 16 bits
nulo	N	nenhum valor
qualquer	L	qualquer valor

A.2 Abreviaturas

abreviatura	especificação
EE	valor do dado entrado pela esquerda
ED	valor do dado entrado pela direita
TE	tipo do dado entrado pela esquerda
TD	tipo do dado entrado pela direita
ix	índice
an	nome de ativação
QS	tamanho de uma estrutura
QA	endereço inicial de uma estrutura

A.3 Especificação das Instruções

A.3.1 Operadores Lógico-Aritméticos

mneumônico	operador		entrada		saída	
	nome		esquerda	direita	tipo	resultado
ADI	soma inteiros		I	I	I	EE + ED
CLI	compara menor ou igual		I	I	B	EE <= ED
NOT	negação		B	N	B	not EE

A.3.2 Operadores de Controle de Fluxo

mneumônico	operador		entrada		saída
	nome		esquerda	direita	condição para EE ser enviada para direita
BRR	“branch” por lógico		L	B	EE + ED

mneumônico	operador		entrada		saída	
	nome		esquerda	direita	tipo	resultado
DUP	duplicação		L	N	TE	EE

mneumônico	operador		entrada		saída esquerda		saída direita	
	nome		esquerda	direita	tipo	resultado	tipo	resultado
SYN	sincronização		L	N	TE	EE	TD	ED

A.3.3 Operadores para Manipulação de Rótulo, Contexto e Destino

operador		entrada		saída		
mneumônico	nome	esquerda	direita	tipo	resultado	mudança do rótulo
PRP	prepara acesso	A	D	W	ED combinado com rótulo	an = EE

A.3.4 Operadores Alteradores do Rótulo

operador		entrada		saída	
mneumônico	nome	esquerda	direita	mudança do rótulo	
SAN	muda an	L	A	an recebe ED	
SIX	muda ix	L	I	ix recebe ED	

A.3.5 Operadores Controladores de Arcos Dinâmicos

operador		entrada		saída	
mneumônico	nome	esquerda	direita	tipo	valor
SCD	muda contexto e destino	L	W	TE	EE

A.3.6 Operadores de Estruturas de Dados

operador		entrada		saída	
mneumônico	nome	esquerda	direita	tipo	valor
PRL	proliferação	L	I	TE	ED cópias de EE <ix= 0..ED-1>

A.3.7 Miscelânea

operador		entrada		saída	
mneumônico	nome	esquerda	direita	tipo	valor
CRJ	cria valor tipo J	I	I	J	<EE,ED>
MIV	modifica valor de indexação	I	J	I	EE - ED.J1 + ED.J2

A.3.8 Operadores da Unidade de Armazenamento de Estruturas de Dados

operador		entrada		saída	
mneumônico	nome	esquerda	direita	tipo	endereço lido da UAE
ESS	extrai valor da UAE	Q	I		EE.QA + ED + 1
FSS	busca estrutura da UAE	Q	I		de EE.QA + ED + 1
					até EE.QA + EE.QS - 1
RSS	leitura da UAE	Q	I		EE.QA + ED + 1

operador		entrada		saída	
mneumônico	nome	esquerda	direita	tipo	resultado
GSP	gera ponteiro para estrutura	I	N	Q	ponteiro para estrutura com tamanho EE + 1

operador		entrada		saída	
mneumônico	nome	esquerda	direita	endereço escrito o valor ED	
WSS	escreve na UAE	Q	L	EE.QA + índice + 1	
WDS	escreve diretamente na UAE	Q	L	se no intervalo EE.QA	

operador		entrada		saída		
mneumônico	nome	esquerda	direita	tipo	resultado	alteração do RC
IRC	incrementa RC	L	N	TE	EE	incrementa RC
DRC	decrementa RC	L	N	TE	EE	se L é do tipo Q decrementa RC
SRC	escreve RC	Q	I	N	-	se L é do tipo Q RC recebe valor ED

operador		entrada		saída esquerda		saída direita	
mneumônico	nome	esquerda	direita	tipo	resultado	tipo	resultado
MAP	modifica endereço do ponteiro	Q	I	Q	EE	Q	EE