REDUÇÃO DE BITS DE EMPARELHAMENTO DE MÁQUINA DE FLUXO DE DADOS DE MANCHESTER

Patrícia Magna [1] *Carlos Antônio Ruggiero* [2]

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.

Neste trabalho, será apresentada a 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.

Este trabalho apresenta 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.

referências dos autores:

[1] Bel. Física Computacional IFQSC/USP;
mestre em Física Aplicada IFQSC/USP, 1992.
e-mail: patricia@ifqsc.usp.br

[2] Engenheiro Eletrônico EESC/USP; Bel. Computação UFSCar;
Mestre em Física Aplicada IFQSC/USP, 1983;
Doutor em Computação, University of Manchester, Grã Bretanha, 1987.
Professor IFQSC/USP.
e-mail: toto@ifqsc.usp.br

endereço:

Departamento de Física e Ciência dos Materiais
Caixa Postal 369, São Carlos, São Paulo
CEP 13560 Tel: (0162) 74-9199

1 Introdução

Tendo como objetivo a obtenção de um processamento de alto desempenho vários modelos computacionais foram propostos nas duas últimas décadas, destacando-se dentre estes o 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. 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. Neste grafo nós representam operações e arcos que os unem, 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). O modo de execução do modelo 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.

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 80] 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, 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 obtêem 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 ítens 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.

1.1 Descrição da Máquina de Fluxo de Dados de Manchester

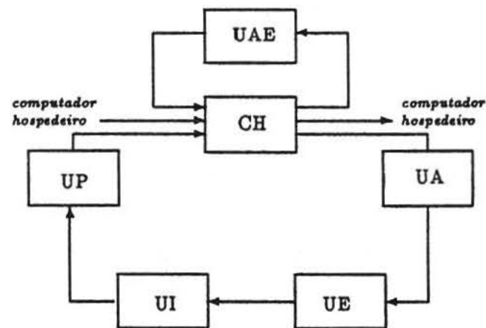

Figura 1: Arquitetura da MDFM

A MDFM (figura 1) é formada por cinco unidades dispostas ao longo de um anel e uma unidade especial para o armazenamento de estruturas de dados. Este tipo de arquitetura explora paralelismo na unidade de processamento, que é constituída por vários processadores. Além disto, a implementação do anel com a utilização de "*pipeline*" possibilita a exploração de paralelismo temporal. Como as atividades 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].

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 (UAE) [Sarg 85], do computador hospedeiro("host") e do próprio anel. Também é através da chave que vários anéis são unidos no caso da MDFM multi-anel[BaGu 87].

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.

1.2 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.

O protótipo de Manchester[GuKW 80] 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.

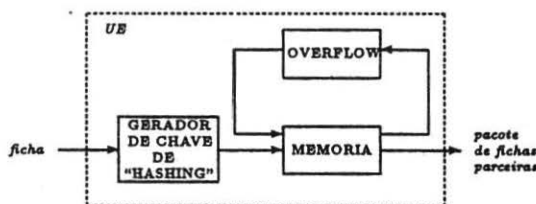

Figura 2: Esquema da Unidade de Emparelhamento

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

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;

1.3 Unidade de Armazenamento de Estruturas de Dados

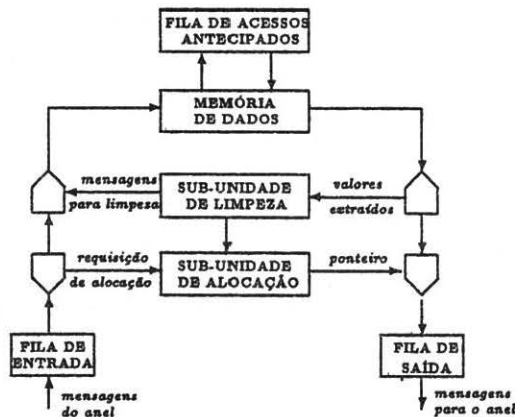

Figura 3: Esquema da Unidade de Armazenamento de Estruturas

Para que a MDFM pudesse suportar estruturas de dados armazenadas foi criada a unidade de armazenamento de estruturas (UAE) [Sarg 85] [SaKi 86]. O modelo de funcionamento desta unidade é baseada no armazenamento de "estruturas-I" [ArTh 81], este modelo 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 (figura 3) que se comunicam entre si e com o anel através de passagem de mensagem. As sub-unidades realizam tarefas independentes; assim, podem operar paralelamente. A unidade é conectada ao anel pela chave. A memória de dados possui além do campo de dado dois bits adicionais: um para indicar a presença de dado armazenado e outro para sinalizar a existência de leituras antecipadas (fato que pode ocorrer no modelo de "estruturas-I"). A sub-unidade de Alocação controla o espaço de armazenamento na memória de dados. Quando uma requisição para o armazenamento de uma nova estrutura é feita é enviado ao anel um ponteiro. A UAE 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 a cada cópia feita da estrutura e decrementado quando uma cópia é destruída. Quando o RC chega ao valor zero é disparado o mecanismo de limpeza da estrutura na memória de dados.

1.4 Formato das Fichas

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 80]: 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). O endereço destino é separado por outros dois campos um que indica o segmento (6 bits) e outro é o "offset" a ser adicionado ao endereço base (12 bits).

Duas fichas são consideradas parceiras quando se destinam a mesma instrução e pertencem a mesma instância de um código reentrante. Estas informações definem, então a chave de emparelhamento, que tem a seguinte constituição: rótulo (36 bits), endereço destino (18 bits).

O rótulo é separado em três sub-campos: nome de ativação, nível de iteração e índice com 12 bits cada um. O nome de ativação separa as fichas de diferentes ativações de uma mesma função. O nível de iteração separa fichas dentro de blocos repetitivos. Já o índice distingue elementos de uma mesma estruturas de dados.

2 Redução da Chave de Emparelhamento

2.1 Eliminação de Redundâncias

2.1.1 Nível de Iteração

O campo de nível de iteração pode ser suprimido e em seu lugar podem ser utilizados os campos de nome de ativação e índice. A figura 4 mostra a estrutura geral de uma construção FOR resultante desta alteração.

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.

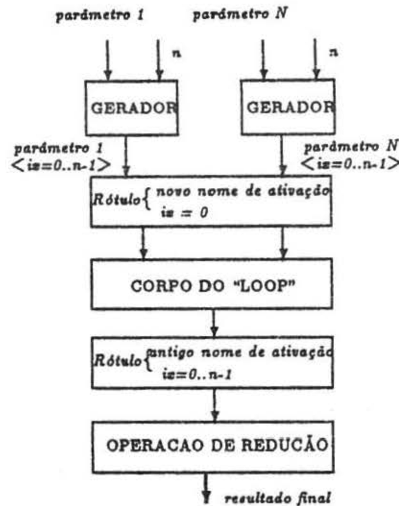

Figura 4: Esquema Geral de um FOR

2.1.2 Segmento

O campo de segmento pode ser eliminado em função da existência de redundância entre este campo e o nome de ativação. Para visualizar esta redundância, deve-se analisar como é feita uma chamada de função.

Quando uma chamada de função é feita (figura 5), a instrução GAN (*Generate Activation Name*) é executada e 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*). Para que o resultado gerado pelo "corpo da função" seja enviado para a função que fez a chamada, é executada a instrução PRP (*Prepare Access*) que 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 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 instrução PRP, com o nome de ativação dá função que fez a chamada.

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. Por esta imposição e pela forma de separar as ativações de funções, pode-se concluir 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. O campo de segmento tornar-se-ia dispensável para a realização

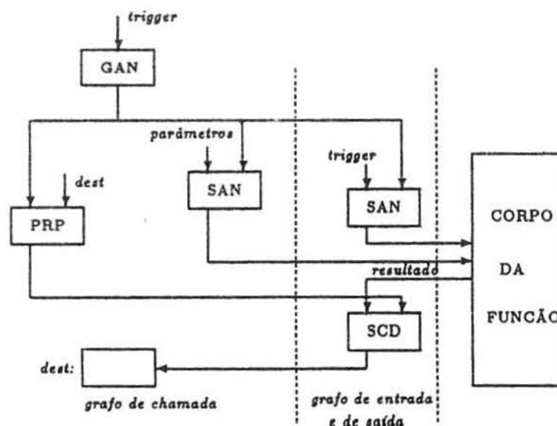

Figura 5: Esquema de uma Chamada de Função

do emparelhamento, sendo que o segmento continuaria sendo útil apenas dentro da 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 a qual o pacote se destina.

A inclusão desta nova tarefa 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, o que 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.

A imposição de que todas as instruções sejam armazenadas em um mesmo segmento pode ser facilmente conseguida, pois a distribuição de instruções na UI é feita estaticamente. Assim, basta inserir esta condição ao gerador de código. Esta condição impõe que o tamanho máximo do código estático de uma função não ultrapasse o tamanho do segmento (4K instruções). Embora sejam raras as funções com tal tamanho, caso seja encontrado algum programa que apresente uma função maior que o segmento, o compilador pode quebrar a função em funções menores.

A atualização da tabela de correspondência a cada chamada de função realizada pode ser feita através da instrução GAN. 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 (figura 6). Desta forma, quando a instrução GAN fosse executada retornaria um novo valor de nome de ativação ao anel e, simultaneamente, enviaria uma mensagem para a tabela de correspondência.

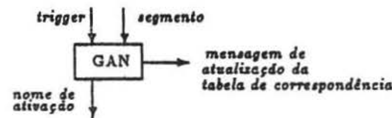

Figura 6: Proposta de Modificação para Instrução GAN

2.2 Redução do Nome de Ativação

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

Os estudos com a MDFM demonstraram também, que a ativação paralela de muitas funções, sem a imposição de algum tipo de controle, pode causar a saturação da máquina. Este fato aumenta o paralelismo exposto pelo programa, gerando 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. Assim, tornou-se extremamente 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.

Para que o nome de ativação pudesse ser reciclado após o término de execução da função, foram elaborados métodos para detecção de conclusão (como existem instruções que não retornam resultados ao anel, a conclusão de uma função não é dada apenas pela obtenção do resultado final).

O método de controle de paralelismo aliado ao esquema de detecção de conclusão de processo são condições básicas para a redução do tamanho do campo de nome de ativação. Pelo fato da saturação da MDFM ser atingida muito antes que 4K processos estejam ativos, não existe razão para que este campo possua 12 bits. Com este campo sendo formado por 10 bits, o tempo gasto na execução de programas não aumentaria, pois com 1K processos podendo estar ativos seria garantido o pleno aproveitamento dos processadores.

Portanto, pela utilização de método de "Throttle", o campo de nome de ativação pode ser reduzido para 10 bits.

2.3 Redução do Índice

O campo de índice é utilizado para separar elementos de uma mesma estrutura de dados, de forma que as operações sobre os elementos possam ser executadas paralelamente. Contudo, este alto grau de paralelismo gerado por este tipo de tratamento não tem sido plenamente aproveitado pelo hardware. Estudos com a MDFM verificaram que quando o paralelismo exposto por um programa

é alto, há um excessivo uso de memória, principalmente da UE. Devido a sua complexidade a UE sofre grande degradação de sua eficiência, limitando a velocidade de geração de pacotes executáveis, o que implica na falta de trabalho para processadores.

A construção da UAE possibilitou a redução do número de cópias de estruturas necessárias para o processamento destas, mas este processamento continua sendo efetuado pelo anel. Por exemplo, a soma de dois vetores é realizada da seguinte forma: os elementos dos vetores são trazidos da UAE para o anel, sendo transformados em uma cadeia de fichas (*stream*). Conjuntamente deve ser verificado se todos os elementos dos vetores estão na forma de fichas através de uma operação de COLETA; caso os dois vetores estejam completos a soma é efetuada, e o vetor resultante é armazenado na UAE, sendo feita novamente a operação de COLETA. Este exemplo demonstra que este tipo de abordagem faz com que o número de instruções necessárias para realização de operações sobre estruturas seja elevado.

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, transferindo a execução das operações sobre estruturas para esta 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. Como a execução de operações sobre estruturas seria transferida para esta nova unidade, o campo de índice pode ser eliminado da implementação de operações vetoriais.

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 ítens seguintes.

2.4 Operações Vetoriais

Existem duas abordagens para a representação de estruturas de dados: uma na forma de cadeia de fichas e outra na forma de estrutura armazenada. Para a finalidade deste trabalho é mais interessante tratar estruturas de dados na forma de estruturas armazenadas, pois isto viabiliza a redução do campo de índice, já que nesta forma o tamanho da estrutura não é limitado pelo tamanho deste campo.

Um vetor na MDFM é representado por um descritor formado por: endereço inicial (A), tamanho (S), limite inferior (*lob*) e *offset*. Ao invés de serem necessárias quatro fichas para o descritor de um vetor, 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 análise das operações envolvendo estruturas de dados, suportada pela linguagem SISAL

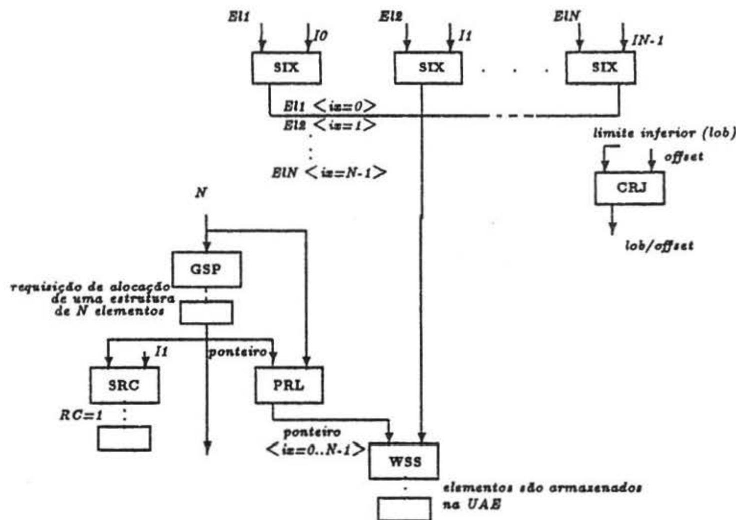

Figura 7: Grafo para Criação de um Vetor

(principal linguagem de alto nível utilizada para programação da MDFM) [MSAG 84], demonstrou que algumas operações básicas utilizam o campo de índice em suas implementações. Assim, deve-se verificar se é possível a transferência de execução para o processador vetorial e, caso isto não seja possível, deve-se fazer modificações dos grafos que implementam estas operações. Estas operações são: CRIAÇÃO, PREENCHIMENTO e COLETA.

2.4.1 Criação

Se os elementos que formam um vetor são conhecidos quando o programa SISAL está sendo escrito, então pode-se utilizar, por exemplo, o seguinte comando SISAL: UM_ATE_TRES := **array**[1: 1,2,3].

Na implementação atual (figura 7) a instrução GSP (*Generate Structure Pointer*) envia à UAE uma requisição de um ponteiro para uma estrutura de N componentes. A UAE envia 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. Os valores dos índices dos elementos são conhecidos em tempo de compilação e são literais na entrada da instrução SIX (*Set IndeX*). Assim, a instrução WSS (*Write Structure Store*) armazena cada dado na posição dada pela soma: endereço inicial + índice + 1.

A figura 8 mostra a proposta para que o campo de índice torne-se desnecessário para informar o correto endereço para o armazenamento dos elementos. Neste grafo, o valor de indexação de cada elemento passa a ser transmitido pelo campo de dado de uma ficha especialmente criada para

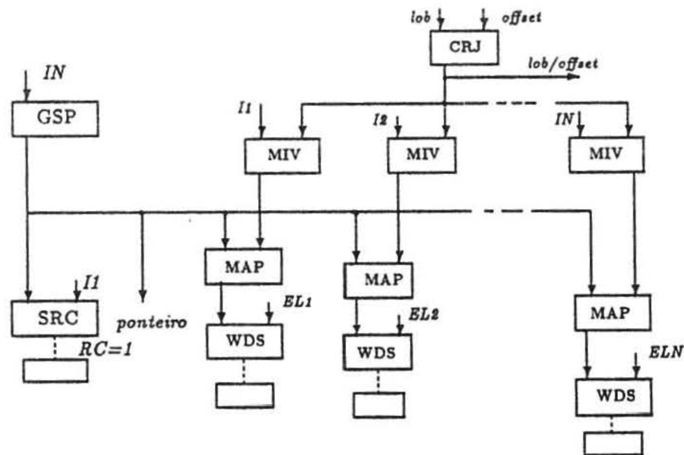

Figura 8: Proposta para Criação de um Vetor

esta função. As instruções MIV e MAP calculam o correto endereço onde deve ser guardado cada elemento dentro da UAE. 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 a posição correta para o armazenamento do dado. Note a existência de uma instrução WDS para cada elemento do vetor, analogamente à instrução SIX da figura 7.

2.4.2 Preenchimento

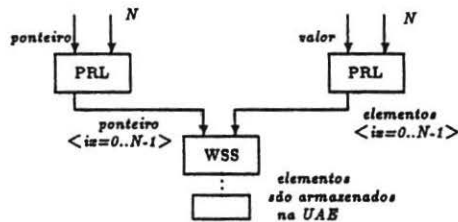

Figura 9: Preenchimento de um Vetor

A operação do SISAL **array_fill**(1,N,*valor*) cria um vetor com N elementos todos recebendo o dado *valor*. A implementação desta operação (figura 9) utiliza-se de apenas uma instrução PRL para gerar todas as fichas dos elementos do vetor informando o valor de indexação através do índice, tornando a implementação bastante otimizada.

A proposta 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.

2.4.3 Coleta

Esta operação (figura 10) 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.

A operação de COLETA inicia com a criação de uma estrutura de tamanho zero, ou seja, apenas o RC. 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.

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

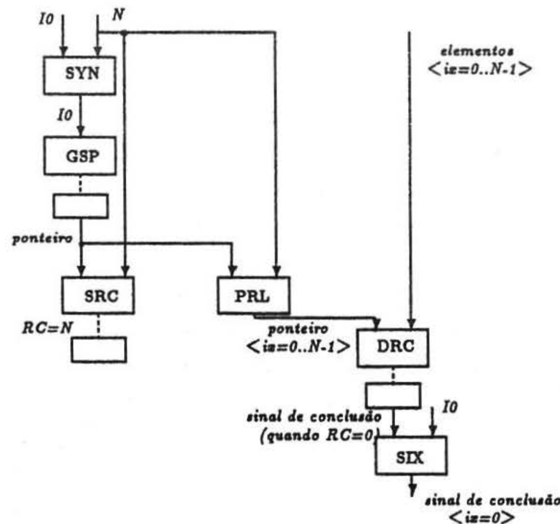

Figura 10: Grafo da Operação de COLETA

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 no item 2.2.4, bem como a forma como seria realizada a operação de COLETA. 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.

2.4.4 Modificações na Implementação de Blocos de Repetição

Para que o índice possa ser suprimido, a implementação de blocos de repetição (figura 4) precisa ser alterada. Para tanto, 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.

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 [AlCo 76] [PaWo 86] 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. Desta constatação surge uma 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 a seguir. 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.

Supondo que o campo de índice seja eliminado, o único campo do rótulo que poderia separar os ciclos seria o nome de ativação. Assim, pode-se sugerir que uma construção repetitiva possa ser tratada como uma chamada recursiva de função. 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. 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.

A tabela 1 mostra como se comportam as execuções do grafo que implementa atualmente a construção for, e do grafo que a transforma em uma chamada de função duplamente recursiva 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, onde S_1 é o número total de instruções executadas em um programa, S_{inf} é a altura do grafo e π uma medida de paralelismo dada por S_1 / S_{inf}. Pode-se notar que o paralelismo se mantem constante mesmo quando o número de ciclos executados chega a 1024. A função duplamente recursiva foi simulada e os resultados 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

N	*forall*			duplo-recursivo		
	S_1	S_{inf}	π	S_1	S_{inf}	π
1	29	15	1.9	26	14	1.9
2	33	17	1.9	71	27	2.6
5	45	23	2.0	206	53	3.9
10	65	33	2.0	431	66	6.5
20	105	53	2.0	881	79	11.2
40	185	93	2.0	1781	92	19.4
60	265	133	2.0	2681	92	29.1
1024	4121	2061	2.0	46061	144	319.9

Tabela 1: Simulação da forma atual de executar **forall** e do programa proposto duplo-recursivo

	LIM = 4			LIM = 8			LIM = 16			LIM = 32		
N	S_1	S_{inf}	π	S_1	S_{inf}	π	S_1	S_{inf}	π	S_1	S_{inf}	π
1	50	26	1.9	50	26	1.9	50	26	1.9	50	26	1.9
2	54	28	1.9	54	28	1.9	54	28	1.9	54	28	1.9
5	131	47	1.9	66	34	1.9	66	34	1.9	66	34	1.9
10	216	59	3.7	151	55	2.7	86	44	2.0	86	44	2.0
20	386	83	4.7	256	67	3.8	191	71	2.7	126	64	2.0
40	791	143	5.5	466	91	5.1	336	83	4.0	271	103	2.6
60	1196	203	5.9	741	127	5.8	481	95	5.1	315	103	3.4
1024	20717	3095	6.7	12395	1567	7.9	8237	815	10.1	6157	463	13.3

Tabela 2: Simulação para execução duplamente recursiva com vários valores de LIM

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 de um programa com várias construções repetitivas envolveria um número muito grande de instruções.

Uma alternativa para amenizar o problema ocasionado pela rotulação realizada a cada ciclo, seria permitir que alguns ciclos pudessem ser executados utilizando o índice. A simulação desta proposta foi feita, possibilitando um estudo comparativo a respeito do número máximo de ciclos que podem ser executados da forma atual, designado pelo nome LIM.

A tabela 2 mostra 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 .

A comparação dos resultados obtidos pode ser vista no gráfico da figura 11. 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, 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 12.

Pode-se concluir que o melhor resultado entre todas as propostas desta seção foi obtido quando

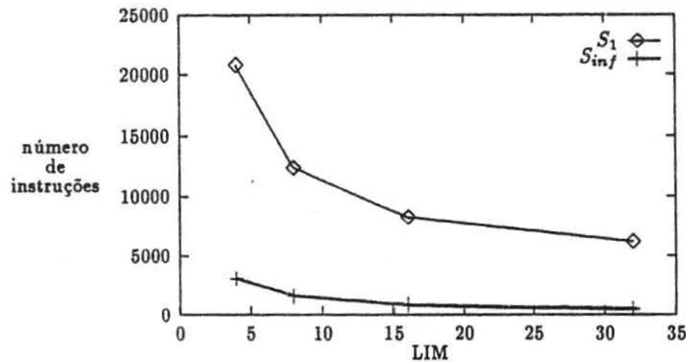

Figura 11: Gráfico de S_1 e S_{inf} como função de LIM

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.

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.

3 Conclusão

A transferência das operações sobre estruturas para a unidade de processamento vetorial, 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.

Por sua complexidade a UE tornou-se um gargalo da MDFM, principalmente, quando o paralelismo exposto por um programa é alto. As alterações propostas neste trabalho 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. Isto garante uma enorme melhora na eficiência da UE, e consequentemente da MDFM.

Referências

[AlCo 76] Allen,F.E. & Cocke,J. *A Program Data Flow Analysis Procedure* Communications of the ACM, 19(3): 137-147, mar. 1976.

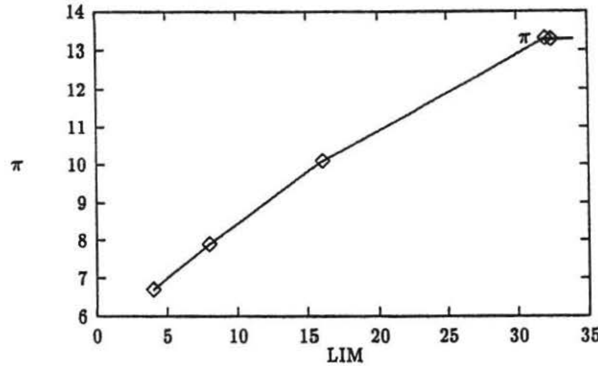

Figura 12: Gráfico de π como função de LIM

[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.

[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

[KaGu 86] Kawakami,K. & Gurd,J.R. *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 Assigment 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.

[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 85] Sargeant,J. *Efficient Stored Data Structures for Dataflow Computing* Ph.D. Thesis, Department of Computer Science, University of Manchester, ago. 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

[WaGu 82] Watson,I. & Gurd,J. *A Practical Dataflow Computer* IEEE Computer, 15(2): 51-57, fev. 1982.