Uma Proposta para Simplificação do Emparelhamento de Dados em Máquinas de Fluxo de Dados

Benedito Aparecido Cruz[1] (bene@dcc.unicamp.br)
Arthur J. Catto[2] (catto@dcc.unicamp.br)

Departamento de Ciência da Computação
Universidade Estadual de Campinas
Caixa Postal 6065
CEP 13081-970 Campinas SP

Resumo

Uma das tarefas mais complicadas a serem realizadas por uma máquina de fluxo de dados dinâmica é o emparelhamento das fichas de dados. Embora arquiteturas mais recentes tenham conseguido simplificar o problema, acreditamos que ainda seja possível fazer esse emparelhamento de maneira mais simples e rápida. Este trabalho expõe algumas das maneiras encontradas para simplificar o problema, procura apresentar um modelo consistente onde essa tarefa seja simplificada e descreve brevemente uma arquitetura que está sendo desenvolvida no Grupo de Fluxo de Dados da UNICAMP.

Abstract

Data token matching is one of the most difficult tasks to be performed by a dynamic dataflow machine. Although some of the newer architectures have managed to simplify such a problem, we believe that a still simpler and faster solution can be designed. This paper presents some of the possibilities which have been considered, a consistent model where this task is simplified plus a brief description of an architecture which is under preliminary design at the Dataflow Group at UNICAMP.

[1] Mestrando em Ciência da Computação, UNICAMP.
[2] Professor Assistente Doutor, Ph.D in Computer Science, University of Manchester, 1981.

1 Introdução

Os computadores disponíveis comercialmente possuem seus princípios fortemente ligados ao modelo (ou arquitetura) von Neumann. Os princípios que guiam esse modelo computacional podem ser resumidos em [AG89]:

- um elemento computacional controla a execução, a comunicação e a memória;
- o controle da computação é centralizado;
- a memória tem organização linear de acesso direto.

Tem-se tornado cada vez mais difícil obter um melhor desempenho desse modelo na medida em que a tecnologia avança e os "gargalos" do mesmo tornam-se mais evidentes[Bac78]. Deve-se observar que, como a tecnologia de quando esse modelo computacional foi desenvolvido era substancialmente diferente da atual, muitos argumentos favoráveis ao modelo de von Neumann não mais se aplicam. Para superar esses problemas, nas últimas décadas surgiram basicamente duas linhas de pesquisa: a primeira tenta adaptar o modelo de von Neumann para processamento paralelo enquanto a segunda rompe drasticamente com ele.

A segunda linha de pesquisa conseguiu resultados expressivos nos últimos anos, baseando suas arquiteturas em novos modelos computacionais. Dentre esses, vamos nos ater a uma subclasse do modelo dirigido pelos dados, chamada *fluxo de dados* (*data flow*) e, dentro dela, discutiremos em particular a implementação da operação de emparelhamento de dados.

As arquiteturas de fluxo de dados vêm-se tornando cada vez mais simplificadas e cada vez mais se aproximando do modelo convencional. De fato, existe atualmente uma forte tendência à combinação dos dois modelos computacionais, como em [BE87], [Ian88], [NA89] e [GHM91]. Em particular, as implementações do esquema de emparelhamento de dados foram bastante simplificadas: enquanto nas primeiras máquinas era necessária a presença de unidades especiais de memória associativa ou pseudo-associativa, as máquinas desenvolvidas nos últimos anos utilizam unidades de memória convencionais associadas a uma lógica digital bastante simples. O modelo proposto a seguir simplifica ainda mais esse esquema, na medida em que prescinde de uma memória de emparelhamento de dados, visto que o emparelhamento é feito ao nível de instrução, e agiliza o processo de busca e decodificação das instruções.

A seção 2 apresenta alguns conceitos importantes para o modelo de fluxo de dados e analisa alguns esquemas de execução de grafos de fluxo de dados. A seção 3 apresenta algumas implementações reais de máquinas de fluxo de dados, com ênfase na operação de emparelhamento. A seção 4 apresenta um modelo onde a tarefa de emparelhamento é simplificada e a seção 5 apresenta as bases de uma arquitetura que implementa esse modelo de execução.

2 Execução de Grafos de Fluxo de Dados: os Modelos Estático e Dinâmico

Os fundamentos do modelo de fluxo de dados são [TBH82, Vee86]:

- Os dados (ou *fichas de dados*[3]) fluem diretamente de uma instrução para outra, sem serem armazenados explicitamente em posições de memória compartilhadas.
- Qualquer instrução pode ser executada desde que seus operandos estejam disponíveis. Essa é a chamada *regra de disparo* (*firing rule*). Não existe um contador de programa nem qualquer outro tipo de controle central.

No modelo de fluxo de dados, um programa executável pode ser representado como um grafo orientado, chamado *grafo de fluxo de dados* [DK82], onde os nós representam as instruções e as arestas orientadas representam as dependências de dados entre as instruções. O grafo de fluxo de dados define, portanto, uma ordem parcial de execução das instruções do programa.

As propostas de arquiteturas para execução de grafos de fluxo de dados podem ser classificadas em estáticas e dinâmicas [Sri86, HYS86, Gur89]. Em uma arquitetura estática, no máximo uma instância de cada nó pode ser executada em cada instante. Em outras palavras, não pode haver mais de uma ficha em qualquer aresta do grafo. O modelo estático de execução não permite, portanto, a execução de subgrafos reentrantes. Embora seja possível obter um grau razoável de paralelismo com a utilização deste modelo, modelos mais elaborados (e mais gerais) conseguem aumentar essa taxa de paralelismo.

[3] *data tokens*

Arquiteturas dinâmicas permitem o disparo simultâneo de mais de uma instância de um nó do grafo. Isso é equivalente a dizer que podemos ter mais de uma ficha em cada aresta do grafo. Por isso, o modelo dinâmico suporta um grau de paralelismo muito maior que o estático. Por exemplo, suponha que um laço de programa possa ser representado como um nó. Uma arquitetura dinâmica pode criar múltiplas instâncias deste nó e executá-las concorrentemente, o que é impossível no modelo estático.

Duas alternativas foram propostas para a implementação do modelo dinâmico de execução. O primeiro esquema, chamado *cópia dinâmica de código* (*dynamic code copying*), permite reentrância através de uma instrução especial colocada no início de um subgrafo (tipicamente uma função ou um laço), a qual cria uma nova cópia do subgrafo cada vez que o mesmo é ativado. Uma instrução de finalização destrói o subgrafo quando sua execução termina. Em uma analogia com o modelo von Neumann, esse esquema funciona como se houvesse uma macro-expansão do código reentrante cada vez que ele é chamado.

O segundo esquema, chamado *rotulação dinâmica* (*dynamic tagging*) permite que o código seja reutilizado através da rotulação das fichas de dados. Esse esquema é semelhante ao uso de pilhas na implementação de procedimentos e funções em máquinas convencionais e é também conhecido como *modelo das fichas rotuladas*: apenas as fichas com o mesmo rótulo podem ser agrupadas para causar a execução de uma instrução. Nesse modelo, são necessárias instruções especiais para criar novos rótulos na entrada de um subgrafo compartilhado e para restaurar os rótulos antigos no final de sua execução. O modelo de rotulação dinâmica foi o modelo mais estudado e implementado nos protótipos construídos.

Para que uma instrução diádica possa ser executada, torna-se necessário que a arquitetura consiga selecionar quais são as fichas de mesmo rótulo que se dirigem para essa instrução. A essa operação dá-se o nome de *emparelhamento de fichas* (*token matching*) e algumas implementações dessa operação serão vistas na seção subseqüente.

Várias máquinas de fluxo de dados foram implementadas: a de Manchester [GKW85], a Monsoon no MIT [PC90] e a japonesa EM-4 [YSK91] são exemplos representativos.

3 Implementações de Emparelhamento de Dados

Nesta seção serão mostradas três diferentes implementações de emparelhamento de dados.

3.1 Máquina de Fluxo de Dados de Manchester

A máquina de fluxo de dados de Manchester [GKW85] foi implementada como um anel circular, por onde fluem as fichas de dados. Esse anel interconecta seis módulos que executam suas funções de maneira totalmente assíncrona: o Módulo Funcional (MF), composto de vários processadores executando em paralelo, o Módulo de Regulagem (MR), o Módulo de Instruções (MI), o Módulo de Estruturas (MEst), o Módulo de Entrada e Saída (I/O) e o Módulo de Emparelhamento (ME).

O Módulo de Emparelhamento funciona da seguinte maneira [WG82]: quando uma ficha de dados chega pelo anel, o ME verifica se alguma ficha armazenada em seus bancos de memória possui os campos *rótulo* e *destino* idênticos aos da ficha recém-chegada. Caso isso seja verdade, as duas fichas são emparelhadas e enviadas, pelo anel, para o MI, para que a instrução a ser executada seja associada ao pacote. Caso contrário, a ficha recém-chegada é armazenada nos bancos de memória à espera de seu par. O Módulo de Emparelhamento é, sem dúvida, a parte mais crítica de toda a máquina. Ele foi implementado como uma memória pseudo-associativa, onde uma técnica de *hashing* por *hardware* é utilizada [GW80, dSW80]: a posição da ficha é determinada usando-se como chave os campos de rótulo e destino da mesma. Acoplado ao ME, existe um Módulo de *Overflow* (MO), utilizado para armazenar fichas que não encontraram seu par e que não podem ser colocadas nos bancos de memória da ME por falta de espaço. O MO é implementado como uma lista ligada controlada por um microprocessador. Acessos à ME devem incluir essa lista de fichas, caso ela não esteja vazia, o que aumenta sensivelmente o tempo de emparelhamento.

3.2 Monsoon

A máquina *Monsoon* [PC90], desenvolvida no MIT, implementa um modelo chamado *Explicit Token-Store* (ETS) [CP90]. Esse modelo alcança os mesmos objetivos das arquiteturas desenvolvidas anteriormente utilizando um esquema de emparelhamento muito mais simples.

No modelo ETS os operandos têm um endereço de memória "de fato" (inexistente na Máquina de Manchester) onde os mesmos são armazenados. O emparelhamento de dados é muito mais simples, não requerendo uma unidade de emparelhamento baseada em memórias associativas ou pseudo-associativas. Para cada função ou bloco de código a ser executado, uma "porção" (*frame*) de memória é alocada. Os

endereços dos operandos passam a ser interpretados como *deslocamentos* em relação ao início do *frame* e o rótulo das fichas passa a ser o endereço do início do *frame* mais um deslocamento fornecido pela instrução. O emparelhamento é feito utilizando-se um *bit* de cada posição de memória para indicar se aquela posição está vazia ou não. Como operandos de uma mesma instrução possuem os mesmos *frame* e deslocamento, o emparelhamento é feito da seguinte maneira: se a ficha encontra a sua posição marcada como *vazia*, ela é armazenada nesta posição de memória, que é, então, marcada como *cheia*. Caso a ficha encontre sua posição marcada como *cheia*, o operando que ocupa a posição é retirado e a posição é então marcada como *vazia*.

A máquina *Monsoon*, que implementa o modelo ETS, possui, ao contrário da máquina de Manchester, apenas um processador por anel, sendo que os anéis são interconectados por uma rede de comutação de pacotes. Os anéis são conectados, pela mesma rede, a várias unidades de memória de estruturas (*I-structures* [ANP89]). Um protótipo funcionando com apenas um processador encontrava-se operacional em 1990 [Lub90].

3.3 EM-4

A máquina EM-4 [SYH+89, SYK+90, YSK91], desenvolvida no Tsukuba Electrotechnical Laboratory, surgiu como um desenvolvimento da máquina EM-3 [YTY90]. A EM-4 introduziu algumas alterações no modelo proposto em [PC90]. A principal delas foi uma alteração do grafo de fluxo de dados que passou a ter dois tipos diferenciados de arestas: arestas comuns e arestas fortemente conectadas. Um subgrafo composto somente de arestas fortemente conectadas toma o nome de *bloco fortemente conectado*. Esse bloco é, então, "compactado" de modo que passa a ser visto, em tempo de execução, como um nó comum no grafo. A vantagem desse modelo consiste em uma execução mais rápida de partes seqüenciais do programa. Resultados intermediários são armazenados em registradores, diminuindo a produção de pacotes e o tamanho dos *frames*, além de acelerar a execução com a diminuição de leituras da memória.

O emparelhamento de dados na máquina EM-4 é feito de maneira semelhante à máquina *Monsoon*. Quando uma função é ativada, é criado para ela um *segmento de dados*. As instruções dessa função estão armazenadas em um *segmento de instruções* (*template segment*), que é único para todas as ativações dessa função. Nesse segmento, as instruções estão armazenadas de uma maneira muito especial: como a ordem em que as mesmas são depositadas na memória não importa no modelo de fluxo de dados, as instruções diádicas são colocadas, pelo compilador, *antes* das instruções monádicas. Além disso, a posição de memória no segmento de dados em que a ficha é depositada para esperar seu par é determinada pelo deslocamento da instrução no segmento de instruções.

Assim, quando uma ficha chega à unidade de emparelhamento, verifica-se se a posição da mesma já está ocupada. Caso esteja, faz-se o *fetch* da instrução correspondente no segmento de instruções e monta-se o pacote para execução com o operando recém-chegado e o que estava armazenado. Caso a posição do operando não esteja ocupada, o mesmo é armazenado, como no modelo ETS.

Note que dentro de um bloco fortemente conectado não há a necessidade de emparelhamento. A arquitetura define um segundo anel, interno ao processador, por onde circulam as fichas destinadas ao mesmo bloco fortemente conectado.

4 Um Modelo Simplificado de Emparelhamento

Como foi visto anteriormente, os grafos de programa são executados diretamente pelas máquinas de fluxo de dados, prescindindo de uma memória para armazenamento explícito de dados. Tomando como exemplo a máquina de Manchester, temos que um dado (ficha) é enviado através de todo o anel até a instrução que necessita dele. Quando essa instrução é executada, novas fichas são geradas e enviadas para o anel.

Em teoria, por uma aresta do grafo podem passar dados de todos os tipos, escalares ou estruturados. Evidentemente, a implementação física de arestas com essa característica é impossível, pela limitação do tamanho do barramento e, por conseguinte, do tamanho da ficha. Os números de *bits* reservados para dados nas arquiteturas descritas na seção anterior são: 32 *bits* em Manchester, 39 *bits* na EM-4 e 64 *bits* em *Monsoon*. Isso significa que apenas dados com tamanho menor ou igual a esses podem ser tratados de maneira atômica pela máquina. Dados com tamanhos maiores devem ser decompostos antes de serem tratados.

O projeto inicial de Manchester previa que os elementos de uma estrutura seriam transmitidos individualmente. Essa solução, evidentemente, sobrecarrega o anel e diminui a velocidade de processamento da máquina. Arvind [ANP89] propõe representar cada estrutura como uma ficha-ponteiro, que aponta para a estrutura propriamente dita, armazenada em uma unidade de memória especial. Essa solução, embora razoável, introduz um tratamento desigual para dados escalares e estruturados.

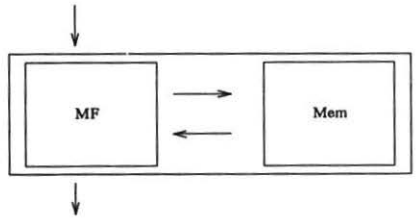

Figura 1: Unidade funcional com memória de dados "próxima"

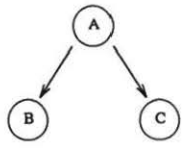

Figura 2: Trecho de um grafo de fluxo de dados.

Uma solução para esse problema é a implementação de uma memória de dados [Kam93] mais próxima dos processadores, o que traria uma maior uniformidade na representação de dados. *Frames*, constantes, daos escalares e estruturados passam a ser tratados da mesma forma.

Paralelamente, pudemos observar, pela seção anterior, que as arquiteturas de fluxo de dados vêm se tornando cada vez mais simples e próximas do modelo convencional. O modelo e a arquitetura descritos a seguir simplificam ainda mais esse modelo, pois:

- diminuem o tamanho da ficha de dados e, por conseqüência, a largura (em bits) do anel;
- diminuem a memória necessária para o emparelhamento;
- retiram complexidade da memória de dados, visto serem desnecessários os bits de presença de dados, como em Monsoon e EM-4;
- facilitam a introdução de uma política de escalonamento de instruções.

4.1 O Modelo de Emparelhamento "de Instruções"

Suponha uma máquina de fluxo de dados cuja unidade funcional possui uma memória de dados acoplada [Kam93] (v. figura 1). Como os bancos de memória estão "próximos" dos processadores, não é necessário que os dados sejam incorporados à ficha que circula no *pipeline*: esta contém somente o rótulo e o endereço da próxima instrução que deve ser executada (*IP*). Definimos o rótulo como o *deslocamento* da posição de memória de emparelhamento em relação ao início de um *frame* e a ficha passa a ser representada pelo par <*FP*,*IP*>. Esse modelo consegue, portanto, dois objetivos: trata os dados de maneira uniforme, pois prescinde de unidades especiais para armazenamento de estruturas (todos os dados são armazenados em memória) e simplifica a implementação da máquina, inclusive diminuindo a largura do barramento.

Note que nesse modelo as instruções são habilitadas para execução através do emparelhamento de fichas cujos rótulos não são mais que os próprios endereços de memória dessas instruções. Com essa constatação, pode-se definir o seguinte procedimento para habilitação de instruções nesse modelo:

Suponha o trecho do grafo de fluxo de dados da figura 2. Nesse modelo, suponha que a instrução A esteja pronta para ser executada (i.e., seus operandos estão disponíveis). A partir desse ponto, sabemos que as instruções B e C estarão com pelo menos um de seus operandos disponíveis após um intervalo de tempo finito. Caso se detecte que A já foi executada, podemos habilitar B para execução, mesmo que não se tenha certeza que o resultado da execução de A já tenha sido armazenado na memória de dados.

O formato das instruções no modelo proposto é mostrado na figura 3, onde N é o número de operandos de que a instrução necessita, op_1 e op_2 são os deslocamentos dos operandos a partir do início do *frame* e d_1 e d_2 são os endereços das instruções para onde os resultados se destinam, em relação ao endereço do nó sendo

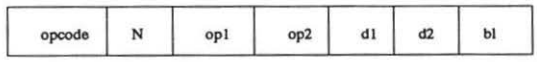

Figura 3: Formato da instrução

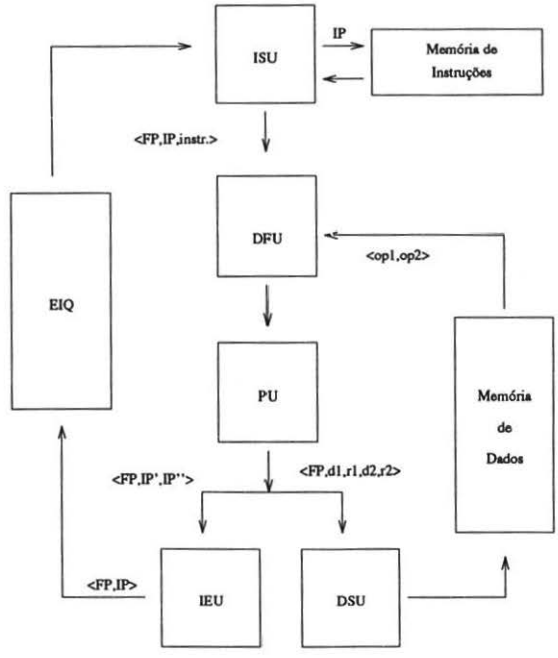

Figura 4: A arquitetura UDFM.

tratado. O campo *bl* indica se a instrução faz parte de um bloco seqüencial ou não. A cada instância de uma instrução associamos um *contador de habilitação* (CH), que contém o número de operandos já recebidos por essa instância. Assim, considera-se que uma instrução está habilitada quando o valor de seu contador de instruções é igual ao número de operandos de que essa instrução necessita. Quando uma instrução termina de ser executada, o valor dos contadores de habilitação de seus nós sucessores é incrementado, enquanto que o valor de seu próprio contador volta para zero. Cada vez que um novo *frame* de operandos é criado, cria-se um novo conjunto de contadores de habilitação (um para cada nó da ativação), cujos valores iniciais são nulos.

4.2 A Arquitetura UDFM

Nesta seção descreveremos uma arquitetura que implementa o modelo descrito na seção anterior. A descrição da interface com a memória de dados pode ser encontrada com detalhes em [Kam93]. Neste artigo nos concentraremos no processo de habilitação de instruções.

Como na Máquina de Fluxo de Dados de Manchester, a arquitetura proposta nesse artigo e mostrada na figura 4 possui apenas um anel, que faz a comunicação entre processadores e entre eles e a memória de dados. Esta proposta visa resolver ou minimizar alguma deficiências apresentadas especificamente por esse modelo de arquitetura, principalmente:

- diminuição do número de fichas circuladas no anel;
- diminuição do número de bolhas no *pipeline*;

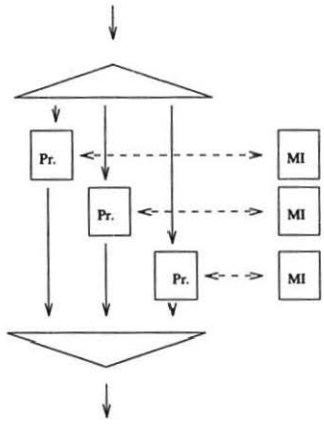

Figura 5: A Unidade de Processamento.

- redução do número de estágios do *pipeline*;
- suporte a blocos seqüenciais de instruções;
- tratamento adequado a estruturas de dados.

O módulos mostrados na figura 4 são os seguintes:

1. **Fila de Instruções Habilitadas (EIQ)**: Contém as instruções que estão habilitadas para execução.
2. **Unidade de Armazenamento de Instruções (ISU)**: Faz a busca da instrução indicada por *IP* na memória de instruções, decodifica a instrução recebida e calcula (com o uso de *FP*) o endereço efetivo dos operandos na memória de dados.
3. **Unidade de Busca de Dados (DFU)**: Faz a leitura dos valores dos operandos calculados pela ISU.
4. **Unidade de Processamento (PU)**: Executa a instrução. É constituída de um árbitro, um *pool* de processadores, cada um deles associados a uma memória local de instruções e de um concentrador.
5. **Unidade de Habilitação de Instruções (IEU)**: Verifica se a execução da instrução sendo analisada habilitou a execução de alguma outra. Se isso for verdadeiro, produz uma ou mais fichas e as manda para a Fila de Instruções Habilitadas.
6. **Unidade de Armazenamento de Dados (DSU)**: Armazena os dados produzidos pela PU.

Nesta arquitetura, os dados são armazenados em *frames* de memória, sendo que cada *frame* é único para uma dada ativação de função ou procedimento ou uma dada iteração de um laço e é identificado pelo seu endereço inicial *FP*. O tamanho de cada *frame* é calculado em tempo de compilação. As únicas exceções são estruturas de dados "grandes", que não são armazenadas em *frames* mas sim em *I-structures* [ANP89].

O funcionamento dessa máquina pode ser descrito da seguinte maneira: a Unidade de Armazenamento de Instruções pega a primeira ficha <*FP,IP*> da Fila de Instruções Habilitadas e busca na memória de instruções a instrução correspondente. Essa instrução é passada para a Unidade de Busca de Dados, juntamente com o campo *FP*. Essa unidade se encarrega de fazer a comunicação com a interface da memória de dados, buscando o valor dos operandos apontados por <*FP:op1*> e <*FP:op2*>, caso esses operandos não sejam parte de I-structures. Caso isso ocorra, o endereçamento é feito usando apenas o valor de *op1* ou *op2*. Com esses valores disponíveis, é formado um pacote que é passado para a Unidade de Processamento.

Na Unidade de Processamento (figura 5), um *distribuidor* passa o pacote para um processador desocupado, que então executa a instrução. Caso o campo *bl* da instrução indique que a mesma faz parte de uma seqüencia [Vis93], o processador busca a nova instrução para ser executada em sua memória de instruções particular e assim por diante, até que o bloco seqüencial de instruções acabe. Caso contrário, o processador

passa um pacote contendo o campo *FP* e os campos *valor1*, *valor2*, *d1* e *d2* para um concentrador, que o repassa para as Unidades de Habilitação de Instruções e de Armazenamento de Dados. A Unidade de Armazenamento de Dados simplesmente armazena os dados representados por *valor1* e *valor2* nas posições <*FP:d1*> e <*FP:d2*>.

A Unidade de Habilitação de Instruções faz a leitura dos contadores de habilitação das instruções-destino (apontadas por *d1* e *d2*), incrementa-os e os compara com o valor do campo *N*. Caso o valor do contador seja igual a *N*, produz uma ficha com o endereço da instrução habilitada e o valor do ponteiro para o *frame* da ativação corrente. Caso contrário, simplesmente armazena o novo valor do contador de instruções na memória específica.

A Fila de Instruções H abilitadas *a priori* implementa uma política FIFO de tratamento das fichas armazenadas. Essa política pode ser alterada, tornando a EIQ mais "inteligente" de modo a implantar uma política mais eficiente de escalonamento de instruções [Lor93] e de controle de paralelismo [RS87].

5 Expectativas

Apesar de o modelo apresentado seguir os princípios do modelo de fluxo de dados, a arquitetura proposta é bastante próxima do modelo convencional, na medida que os dados são armazenados explicitamente em memórias convencionais, as quais se faz acesso direto. O emparelhamento é bastante simples, consistindo apenas de uma comparação entre o contador de habilitação e o número de operandos necessários para a execução de uma instrução. Unidades especiais de emparelhamento tornam-se, então, desnecessárias. Outro fator a ser considerado é que a largura de banda[4] necessária para o anel é reduzida, visto que uma ficha é formada apenas de dois endereços: o ponteiro para a próxima instrução e o ponteiro para o início do *frame* correspondente a uma ativação. O modelo e arquitetura propostos nesse artigo ainda estão em fase de aprimoramento e resultados parciais de avaliação ainda não podem ser apresentados. Um trabalho de simulação dessa arquitetura está sendo iniciado.

REFERÊNCIAS

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamim/Cummings Publishing Company, Inc., 1989.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data structures for parallel computing. *ACM Transa tions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[Bac78] Jchn Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, pages 613–641, August 1978.

[BE87] Richard Buehrer and Kattamuri Ekanadham. Incorporating data flow ideas into von Neumann processors for parallel execution. *IEEE Transactions on Computers*, C-36(12):1515–1522, December 1987.

[CP90] David E. Culler and Gregory M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, December 1990.

[DK82] Alan L. Davis and Robert M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, February 1982.

[dSW80] J. G. D. da Silva and Ian Watson. A pseudo-associative matching store using hardware hashing. Relatório Técnico, Universidade de Manchester, 1980.

[GHM91] Guang R. Gao, Herbert H.J. Hum, and Jean-Marc Monti. Towards an efficient hybrid dataflow architecture model. In *5th ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523*, pages 353–371, 1991.

[GKW85] John R. Gurd, Chris Kirkham, and Ian Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[Gur89] John R. Gurd. Dataflow architectures. *Computer Systems Science and Engineering*, 4(4):241–251, October 1989.

[4] *bandwidth*

[GW80] John R. Gurd and Ian Watson. Data driven system for high speed parallel computing – part 2: Hardware design. *Computer Design*, 19(7):97–106, July 1980.

[HYS86] Jayantha Herath, Toshitsugu Yuba, and Nobuo Saito. Dataflow computing. *Lecture Notes on Computer Science*, 269:25–36, 1986.

[Ian88] Robert A. Ianucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architectures*, pages 131–140, 1988.

[Kam93] Carlos Alberto Kamienski. Armazenamento de resultados em uma arquitetura de fluxo de dados. Dissertação de Mestrado, 1993.

[Lor93] Paulo Lorenzo. Escalonamento de instruções em arquiteturas de fluxo de dados. Dissertação de Mestrado em preparação, DCC – UNICAMP, 1993.

[Lub90] Olaf M. Lubeck. A user's view of dataflow architectures. In *Compcon Spring 90: Digest of Papers*, pages 84–87, 1990.

[NA89] Rishiyur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architectures*, pages 262 – 272, May 1989.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architectures*, pages 82–91, 1990.

[RS87] Carlos Ruggiero and John Sargeant. Control of parallelism in the manchester dataflow computer. In *Proc 3rd Conf. Functional Prog. Lang. and Comp. Arch. (LNCS 274)*, 1987.

[Sri86] Vason P. Srini. An architectural comparison of dataflow systems. *IEEE Computer*, 19(3):68–88, March 1986.

[SYH+89] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba. An architecture of a dataflow singtle chip processor. In *Proceedings of the 16th Annual International Symposium on Computer Architectures*, pages 46–53, 1989.

[SYK+90] Suichi Sakai, Yoshinori Yamaguchi, Yuetsu Kodama, Kei Hiraki, and Toshitsugu Yuba. Design of the dataflow single-chip processor EMC-R. *Journal of Information Processing*, 13(2):165–173. 1990.

[TBH82] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Computing Surveys*, 14(1):93–143, March 1982.

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[Vis93] Marcos C. Visoli. Tratamento de código sequencial no modelo de fluxo de dados. Dissertação de Mestrado em preparação, DCC – UNICAMP, 1993.

[WG82] Ian Watson and John Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, February 1982.

[YSK91] Yoshinori Yamaguchi, Suichi Sakai, and Yuetsu Kodama. Synchronization mechanisms of a highly parallel dataflow machine EM-4. *IEICE Transactions*, E74(1):204–213, January 1991.

[YTY90] Yoshinori Yamaguchi, Kenji Toda, and Toshitsugu Yuba. Evaluation of a data-driven machine with advanced control mechanism. *Systems and Computers in Japan*, 21(5):15–28, 1990.