Arquitetura Super Escalar com Escalonamento Estático com Permutação de Tipos de Instruções.

Ronaldo A. L. Gonçalves [1]
Departamento de Informática / UEM
e-mail: ronaldo@din.uem.br

Philippe O. A. Navaux
Instituto de Informática / UFRGS
e-mail: navaux@inf.ufrgs.br

Resumo

Atualmente, os computadores pessoais são projetados como arquiteturas superescalares, e são capazes de extrair paralelismo a nível de instrução, aumentando o desempenho das aplicações. Nestas arquiteturas, o despacho de instruções paralelas deve ser feito de forma eficiente.

Este artigo apresenta uma arquitetura superescalar, que despacha instruções paralelas utilizando o algoritmo de Tomasulo. Com o objetivo de aumentar o desempenho no despacho destas instruções, é proposta a utilização do Algoritmo Estático de Escalonamento com Permutação de Tipos de Instruções.

palavras-chaves: arquitetura superescalar, algoritmo de Tomasulo, escalonamento estático

Abstract

Nowadays, the personal computers are projected as superscalar architectures, and they are able to extract instruction level parallelism, increasing the performance of the applications. In these architectures, the dispatching of parallel instructions must be eficient.

This paper presents a superscalar architecture, that dispatches parallel instructions in accordance with the Tomasulo's algorithm. With the objective to increase the performance on dispatching these instructions, it is proposed the use of the Static Algorithm of Scheduling with Permutation of Instructions Types.

key-words: superscalar architecture, Tomasulo's algorithm, static scheduling

[1] Doutorando no CPGCC (Instituto de Informática) da Universidade Federal do Rio Grande do Sul
e-mail: ronaldog@inf.ufrgs.br

1. Introdução

Antigamente, os processadores utilizavam todos os seus recursos durante a execução de uma única instrução. Para eliminar este gargalo, surgiram as arquiteturas com *pipeline*, onde a divisão da execução de uma instrução em estágios, permite que após a finalização de um estágio por uma instrução, uma nova instrução utilize aquele estágio. Isto possibilita a execução de várias instruções simultaneamente, dentro do *pipeline* [RYAN92].

Deste modo, as arquiteturas com *pipeline* distribuem os recursos do hardware entre diferentes instruções, mas não aproveitam o paralelismo inerente das aplicações. Para eliminar esta deficiência, surgiram as arquiteturas superescalares, onde o *pipeline* é replicado. Neste caso, cada unidade funcional do processador possui seu próprio *pipeline* e pode operar independentemente das outras unidades.

Nas arquiteturas superescalares, o paralelismo pode ser obtido dentro dos *pipelines* e entre os *pipelines*, fornecendo um maior desempenho para as aplicações. Atualmente, são vários os processadores superescalares produzidos comercialmente, onde podem ser destacados: Pentium [ANDE95], Power e PowerPC [CHAK94].

Para despachar instruções paralelas nestas arquiteturas, duas tendências têm sido utilizadas [JUNI95]: algoritmos dinâmicos e algoritmos estáticos. Na primeira, existe um algoritmo implementado em *hardware* (ex: Tomasulo) capaz de escolher a instrução que deve ser despachada e decidir para onde deve ser despachada. Na segunda, existe um compilador especial que gera código eficiente de forma que as unidades funcionais possam executar um maior número de instruções independentes por ciclo.

Este artigo apresenta a arquitetura superescalar MulFlux, que utiliza o algoritmo de Tomasulo para despachar instruções, e propõe a utilização do Algoritmo Estático de Escalonamento com Permutação de Tipos de Instruções, que visa aumentar o desempenho da arquitetura superescalar.

2. Arquitetura Super Escalar MulFlux

A arquitetura superescalar Mulflux, proposta em [CHAV96] e que faz parte de um projeto cooperativo entre diversas universidades, está em fase de desenvolvimento e visa executar múltiplos fluxos de instruções simultaneamente, como forma de obter maior desempenho. Outras arquiteturas têm sido propostas com a finalidade de executar múltiplos fluxos de instruções, tais como MISC [TYSO94] e XIMD [WOLF91].

Na arquitetura MulFlux, as instruções que estão na fila de instruções são decodificadas e despachadas, segundo o algoritmo de Tomasulo [TOMA67], para as estações reservas das unidades funcionais apropriadas, de acordo com seus tipos. Este algoritmo permite a execução simultânea de instruções independentes e preserva a precedência inerente do fluxo de instruções. O *hardware* básico é mostrado na figura 1.

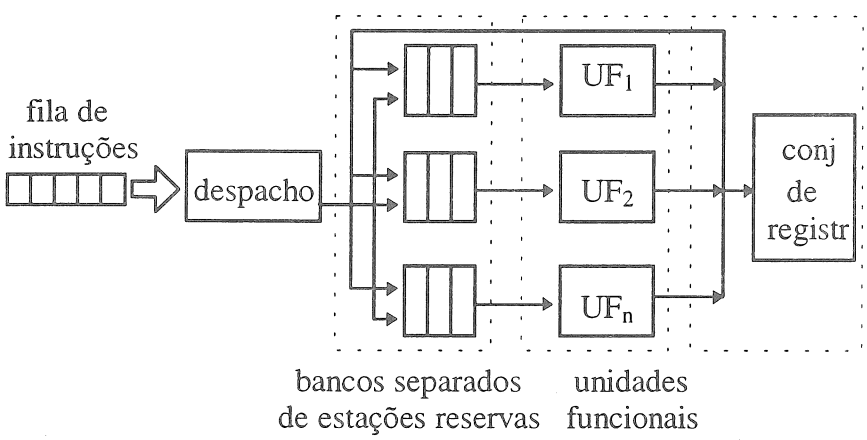

Figura 1: Modelo Básico do *Hardware* de Tomasulo

O algoritmo de Tomasulo tenta sempre despachar instruções para deixar livre o decodificador, e por isso utiliza bancos de estações reservas. Cada unidade funcional detecta e executa as instruções que estão prontas em seu respectivo banco. Mas quando o banco de estações reservas da unidade funcional, para o qual a instrução deve ser despachada, está cheio, o despacho das instruções é interrompido, e conseqüentemente pára a decodificação.

Este fato reduz o desempenho da arquitetura superescalar e pode ser causado pela superutilização de um mesmo banco de estações reservas. Isto ocorre quando existe longas seqüências de instruções de mesmo tipo, adjacentes na fila de instruções. Para amenizar este

problema, neste artigo é proposto um "Algoritmo Estático de Escalonamento com Permutação de Tipos de Instruções", que possibilita uma melhor distribuição das instruções durante o despacho.

Muitos algoritmos têm sido propostos para o escalonamento de instruções para arquiteturas específicas, mas um estudo geral sobre eles pode ser encontrado em [REWI94].

3. Algoritmo Estático de Escalonamento com Permutação de Tipos de Instruções.

O algoritmo aqui proposto trabalha sobre o grafo de dependências de dados de cada bloco básico do programa a ser reorganizado. Ele não move instruções entre os blocos básicos e não altera a precedência entre as instruções. Este algoritmo somente serializa a ordem dos *traces* independentes dentro de cada bloco básico, de forma a permitir um melhor aproveitamento pelo processador superescalar.

O escalonamento com permutação de tipos de instruções reduz o tamanho da seqüência de instruções de mesmo tipo que estão adjacentes no código de um processador superescalar, favorecendo o algoritmo de Tomasulo durante o despacho. Isto permite uma melhor distribuição das instruções, entre as diferentes unidades funcionais, e diminui o número de *stalls* (interrupção do *pipeline*) causados pela superutilização das estações reservas de uma mesma unidade funcional.

Cada instrução do programa possui um tipo e um grau de adjacência (grad). O tipo se refere a um dos tipos das unidades funcionais existentes e o grad indica o tamanho da seqüência de instruções de mesmo tipo a que a referida instrução pertence. Na tabela 1 é mostrado um trecho de código exemplo, com suas instruções e respectivos tipos e grads.

nro	instrução	tipo	grad
01	LD A,B	inteiro	3
02	ADD A,C	inteiro	3
03	SUB B,C	inteiro	3
04	JMP label1	desvio	1
05	LD AX,AY	flutuante	3
06	MUL AX,AX	flutuante	3
07	SUB BX,CX	flutuante	3
08	JPZ label2	desvio	2
09	JMP label3	desvio	2

Tabela 1: Trecho de Código com Tipos e Grads Associados.

A figura 2 apresenta um possível grafo de dependências de dados para um bloco básico, onde os nodos estão rotulados com o número da instrução e o tipo. Deve-se observar que os arcos que ligam os nodos são dependências de dados e não dependências de controle. O grafo esta dividido em *traces* lineares.

Nota-se que nenhuma reordenação pode ser feita dentro de cada *trace*. Os *traces* 2, 3 e 4 são *traces* independentes (paralelizáveis) e por isso podem ser dispostos (serializados) em qualquer ordem no *schedule* final. Deve-se observar que o processador superescalar é quem paraleliza as instruções durante o despacho. Neste exemplo, podem ser gerados 6 *schedules*, mostrados na tabela 2, que equivalem a permutação dos 3 *traces* paralelizáveis (3!).

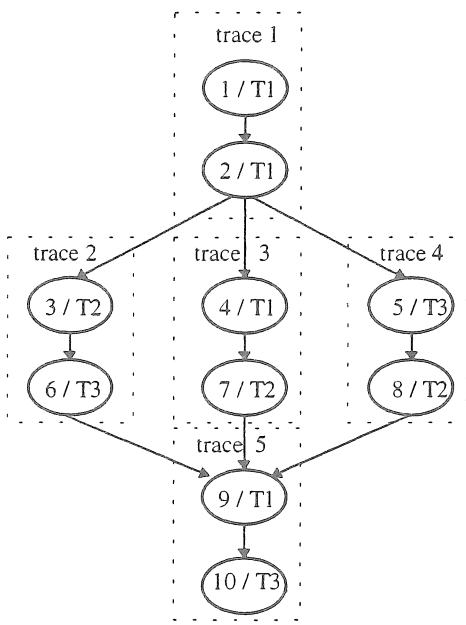

Figura 2: Grafo de Dependências

Na tabela 2 pode ser observado que o *schedule* 1 possui instruções com os menores grads, enquanto o *schedule* 3 possui instruções com os maiores grads. Assim, o *schedule* 1 é a opção que melhor distribui as instruções entre as diferentes unidades funcionais de um processador superescalar.

Esta reorganização não impede a superutilização de qualquer banco de estações reservas, mas possibilita que durante o tempo em que um dos bancos estiver cheio, os outros bancos não

estejem vazios, pois o prejuízo maior ocorrerá se alguma unidade funcional ficar sem executar instruções porque o despacho foi interrompido.

schedules	instruções / tipos									
1	1/T1	2/T1	3/T2	6/T3	4/T1	7/T2	5/T3	8/T2	9/T1	10/T3
2	1/T1	2/T1	3/T2	6/T3	5/T3	8/T2	4/T1	7/T2	9/T1	10/T3
3	1/T1	2/T1	4/T1	7/T2	3/T2	6/T3	5/T3	8/T2	9/T1	10/T3
4	1/T1	2/T1	4/T1	7/T2	5/T3	8/T2	3/T2	6/T3	9/T1	10/T3
5	1/T1	2/T1	5/T3	8/T2	4/T1	7/T2	3/T2	6/T3	9/T1	10/T3
6	1/T1	2/T1	5/T3	8/T2	3/T2	6/T3	4/T1	7/T2	9/T1	10/T3

Tabela 2: Permutações de Schedules

Um algoritmo simples e preciso pode gerar todas as possíveis permutações e selecionar o *schedule* ideal, ou seja, aquele com os menores grads, embora consuma muito tempo de reorganização. O algoritmo de permutação aqui proposto é mais rápido, pois percorre o grafo de dependências uma única vez e gera um *schedule* que pode ser o ideal ou pelo menos próximo dele. O algoritmo simplificado é mostrado abaixo.

```
Algoritmo Principal
Declara  raiz, ultimo: tipo_nodo ;
Início
    Obtém_Nodo_Inicial (raiz) ;
    Obtém_Ultimo_Nodo (ultimo) ;
    Gera_Schedule (raiz, ultimo) ;
Fim-Algoritmo.

Algoritmo Gera_Schedule (raiz, ultimo: tipo_nodo)
Declara  inicio, fim: tipo_nodo ; tipo: tipo_unidade_funcional ;
Início
  Repita
     Grava_Instrução_No_Schedule (raiz) ;
     Se (Não Difusor (raiz))
          Então   Obtém_Próximo (raiz)
          Senão  tipo ← Obtém_Tipo_Da_Instrução (raiz) ;
                 Enquanto Houver_Ramo_Não_Processado (raiz)  Faça
                      Obtém_Melhor_Ramo (raiz, inicio, fim, tipo) ;
                      Gera_Schedule (inicio,fim) ;
                      tipo ← Obtém_Tipo_Da_Instrução (fim) ;
                 Fim-Enquanto ;
                 Obtém_Nodo_Concentrador (raiz);
     Fim-Se ;
  Até (raiz=ultimo) ;
  Grava_Instrução_No_Schedule (raiz) ;
Fim-Algoritmo ;
```

```
Algoritmo Obtém_Melhor_Ramo ( raiz, inicio, fim: tipo_nodo , tipo: tipo_unidade_funcional )
Início
/* Escolhe um dos ramos da "raiz" que ainda não foi processado
   com tipo diferente de "tipo", senão escolhe um ramo com menor
   grad. Retorna os nodos "início" e "fim" do ramo, onde o nodo
   "início" é o nodo raiz e o nodo "fim" é o nodo imediatamente
   anterior ao nodo concentrador */
Fim-Algoritmo ;
```

O algoritmo percorre o grafo de dependência de dados, e a cada nodo que ele encontra ele coloca a instrução correspondente no *schedule* final, na mesma ordem original. Quando o algoritmo encontra um nodo difusor (ex: nodo 2 na figura 2) ele serializa os ramos e continua o percurso a partir do nodo concentrador (ex: nodo 9 na figura 2).

Para serializar os ramos de um nodo difusor, ele escolhe um ramo que começa com um tipo diferente ao da última instrução (inicialmente a raiz difusora). Se não houver ramos que começam com tipos diferentes, ele escolhe um ramo que começa com o menor grad. O próximo ramo escolhido deve usar como referência o tipo da última instrução do ramo anterior.

4. Conclusão

O desempenho de uma arquitetura superescalar está diretamente relacionado com o grau de paralelismo físico executado pelas unidades funcionais, e este paralelismo físico é limitado ao paralelismo lógico contido nos programas. Nem sempre é conveniente aumentar o número de unidades funcionais, se os programas não são paralelizáveis.

Neste sentido, é necessário a existência de algoritmos de reorganização de código capazes de extrair o máximo de paralelismo implícito contido nos programas. O algoritmo de permutação aqui proposto visa balancear os fluxos de instruções entre as diferentes unidades funcionais, sem a necessidade de modificações na arquitetura ou nos programas, e pode ser empregado para aumentar o desempenho final de um processador superescalar.

Como trabalho futuro, este algoritmo deverá ser avaliado. Para isto, pretende-se implementá-lo em um compilador ja existente, e posteriormente submeter os *schedules* gerados a um simulador da arquitetura MulFlux.

5. Referências Bibliográficas

1.[ANDE95] Anderson, D. & Shanley, T., Pentium Processor System Architecture, Second Edition, MindShare, Inc., Addison-Wesley, Massachusetts, 433p., February, 1995

2.[CHAK94] Chakravarty, D., Cannon, C., PowerPC: Concepts, Architecture, and Design, J. Ranade Workstations Series, McGraw-Hill, USA, Inc.,p.363, 1994.

3.[CHAV96] Chaves Filho, E. M. et alii, Uma Arquitetura Super Escalar com Múltiplos Fluxos de Instruções, VIII SBAC-PAD, p.67-76, Recife, Agosto, 1996.

4.[JUNI95] Junior, M.J., Aude, J.S., Compactação Local de Código para um SPARC Superescalar, Anais do VII Simpósio Brasileiro de Arquiteturas de Computadores- Processamento de Alto Desempenho, p.539-552, Canela-RS, Agosto, 1995.

5.[REWI94] El-Rewini, H., Lewis, T.G., Ali, H.H.,Task Scheduling in Parallel and Distributed Systems, Prentice-Hall, Englewood Cliffs, New Jersey, 290p., 1994.

6.[RYAN92] Ryan, B., Built for Speed, Byte, p.123-135, Fevereiro 1992

7.[TOMA67] Tomasulo, R.M., An Efficient Algorithm for Exploiting Multiple Arithmetic Units, IBM Journal, pp.25-33, January, 1967.

8.[TYSO94] Tyson, G. & Farrens, M., Code Scheduling for Multiple Instruction Stream Architectures, International Journal of Parallel Programming, Vol.22, n.3, 1994.

9.[WOLF91] Wolf, A. & Shen, J. A Variable Instruction Stream Extension to VLIW Architecture, In Proceedings of the Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, pp.2-14, ACM, April 1991