Mecanismos de Detecção de Instruções Dependentes em Arquiteturas Super Escalares

Gabriel P. Silva[1] e Edil S.T. Fernandes[2]
[1] Núcleo de Computação Eletrônica — Universidade Federal do Rio de Janeiro
Rio de Janeiro — Brasil
{ gabriel@nce.ufrj.br }
[2] COPPE–Sistemas e Computação — Universidade Federal do Rio de Janeiro
Rio de Janeiro — Brasil
{ edil@cos.ufrj.br }

Resumo—

Apesar dos avanços tecnológicos, o número de instruções que são despachadas em paralelo pelos processadores super escalares atuais é ainda muito modesto. As técnicas de predição de desvio e a renomeação de registradores atenuam os efeitos das dependências de controle e de dados, que são os principais obstáculos para o aumento da largura de despacho.

Este artigo trata da deteção automática da dependência de dados de instruções em máquinas super escalares de amplo despacho. Nós especificamos três mecanismos capazes de examinar as dependências de dados de um grande número instruções. Cada mecanismo tem uma *cache* especial que armazena as relações de dependência entre as instruções de um mesmo bloco básico.

A suíte SPEC95 foi simulada por máquinas super escalares experimentais equipadas com esses mecanismos. Os resultados dos experimentos mostraram que nossos mecanismos de deteção são mais eficientes do que os existentes: reduzem a complexidade do *hardware* e tornam viável o despacho de um grande número de instruções em paralelo.

Palavras-chave— **Super Escalares, Despacho Amplo, Instruções, Dependências.**

Abstract—

Despite the technological advances, the number of instructions being dispatched in parallel by current superscalar processors is still very modest. Branch prediction and register renaming technique attenuate the effects of control and data dependences which are the main obstacles for increasing the dispatch width.

This paper deals with the automatic detection of data dependent instructions in wide-issue superscalar machines. We have specified three mechanisms which are able to examine the data dependences of a large number of instructions. Each mechanism has a special cache which stores the dependence relationships among the instructions from a same basic block.

The SPEC95 suite were simulated by experimental superscalar machines equipped with these mechanisms. Results of the experiments shown that our detection mechanisms are more efficient than the existing ones: they reduce the hardware complexity, and make viable the dispatch of a large number of instructions in parallel.

Keywords— **Superscalar, Wide Issue, Dispatch, Tomasulo.**

I. Introdução

Arquiteturas super escalares extraem automaticamente o paralelismo do nível de instrução, sem que sejam necessárias modificações quer no código fonte, quer no código objeto dos programas de aplicação. Contudo, as dependências de controle e de dados são os maiores obstáculos à extração desse paralelismo.

No campo das dependências de controle, preditores cada vez mais sofisticados têm sido desenvolvidos, permitindo ao processador especular com maior precisão o fluxo de execução dos programas. No campo das dependências de dados, trabalhos têm sido desenvolvidos para predição de valores e reutilização de código, como forma de superar este tipo de limitação.

Entretanto, para aumentar o desempenho global, é necessário que todos os estágios do *pipeline* de instruções estejam otimizados. Em particular, os estágios de renomeação, despacho e conclusão são críticos quando se despacha um grande número de instruções em paralelo.

Os mecanismos de renomeação e de despacho de arquiteturas com janela central de instruções, um banco de registradores único e uma tabela de renomeação apresentam as seguintes deficiências quando do despacho de um grande número de instruções em paralelo (i.e., mais do que quatro instruções por ciclo):

- O procedimento de detecção das dependências verdadeiras entre as N instruções que estão sendo despachadas simultaneamente requer $N \times (N-1)$ comparadores [JOH 91];
- O total de portas de leitura da tabela de renomeação aumenta com o número de instruções despachadas em paralelo. Analogamente, o banco de registradores precisa de $2 \times N$ portas de leitura e de N portas de escrita. Esse grande número de portas torna o projeto mais complexo;
- O tempo de identificação das instruções que estão prontas para serem executadas, é proporcional a N^2 e a J^2, onde J é o total de instruções na janela centralizada [PAL 96];
- A complexidade da operação de *bypass* (transferência dos resultados produzidos pelas unidades funcionais para suas respectivas entradas, para reduzir em um ciclo o tempo de espera pelos operandos) é proporcional a F^2, onde F é o número de unidades funcionais existentes.

Para viabilizar o despacho de um grande número

instruções em paralelo, especificamos e simulamos uma arquitetura que contém:

- uma janela de instruções descentralizada (i.e., as instruções ficam armazenadas em estações de reserva) e usa o algoritmo de Tomasulo [TOM 67] modificado;
- uma estrutura de armazenamento, denominada *cache* de dependências, que contém as relações de dependência entre as instruções de um mesmo bloco básico.

Além de reduzir o número de portas de leitura e escrita do conjunto de registradores, nossos mecanismos de detecção reduzem o número de comparadores necessários para verificar se as instruções que estão sendo despachadas simultaneamente apresentam dependências. A redução da complexidade dos estágios de renomeação e despacho e um número menor de portas do banco de registradores têm como conseqüência imediata um aumento no desempenho final do processador.

Este artigo está organizado em seis seções. A Seção II descreve as modificações que devem ser introduzidas no algoritmo de Tomasulo de modo que ele possa despachar múltiplas instruções em paralelo. A Seção III é dedicada aos nossos mecanismos de detecção de dependências. A Seção IV, descreve o meio ambiente experimental e, na Seção V, os resultados dos experimentos são mostrados e analisados. Finalmente, a Seção VI apresenta as principais conclusões de nossa investigação e destaca as oportunidades de pesquisa futura na área.

II. Despacho de Múltiplas Instruções

Mesmo após o transcurso de três décadas, o algoritmo de Tomasulo, desenvolvido para o sistema IBM 360/91 em 1967 [TOM 67], ainda é o mais eficiente para o despacho de uma única instrução por ciclo. Por esta razão, diversos processadores [IBM 94, WIL 95, INT 98] da atualidade utilizam variações deste algoritmo para o despacho de múltiplas instruções por ciclo.

No algoritmo de Tomasulo, assim que uma instrução é despachada, outras podem modificar os registradores fonte e destino desta instrução, ignorando as dependências de saída e anti-dependências, porque os valores destes operandos (ou os *tags* correspondentes) já foram copiados para as estações de reserva.

Se estivermos interessados em despachar múltiplas instruções por ciclo, o algoritmo de Tomasulo precisa ser modificado. Para eliminar a ocorrência de anti-dependências, é suficiente copiar os *tags* ou os operandos para as estações de reserva. Esta cópia e o despacho da instrução para a estação de reserva devem ocorrer simultaneamente.

Para as dependências de saída, precisamos garantir que somente a instrução mais recente, dentre as que estão sendo despachadas no mesmo ciclo, atualize o campo de *tag* de um mesmo registrador destino. Para implementar este controle, podemos usar um mecanismo simples de prioridade embutido no circuito decodificador de acesso ao banco de registradores. Este banco possui múltiplas portas de escrita, seja para armazenar os valores nos registradores, seja para atualizar os campos de *tag*, que denominamos *tag array.*

No caso das dependências verdadeiras, para cada instrução precisamos verificar se os seus operandos fonte serão produzidos por uma instrução precedente que esteja sendo despachada em paralelo. Para tal, precisamos de $N \times (N-1)$ comparadores para verificar as relações de dependência entre as N instruções que estão sendo despachadas simultaneamente, considerando-se dois operandos por instrução. Em outras palavras, o número de comparações é proporcional ao quadrado da largura de despacho. Reduzir esse número de comparações é o objetivo do nosso trabalho.

O algoritmo de Tomasulo precisa de uma outra adaptação para viabilizar o término de múltiplas instruções por ciclo. A adição de tantos barramentos de resultados (CDBs) quantos forem os resultados produzidos por ciclo atende a esse requisito. Por outro lado, para cada CDB adicionado, a lógica de associatividade tem que ser replicada em todas estações de reserva [PAT 90] e também no *tag array.*

III. Mecanismos de Detecção

Nossos modelos derivam de uma arquitetura super escalar orientada a blocos básicos [FER 00] e usa o algoritmo de Tomasulo adaptado para o despacho de múltiplas instruções por ciclo. Essa arquitetura é equipada com *future file, buffer* de reordenação, múltiplos CDB's e um banco de registradores com *tag array* e com múltiplos comparadores. Durante nossa investigação, empregamos máquinas experimentais que despacham 8 e 16 instruções em paralelo.

As dependências envolvendo instruções de um mesmo bloco básico (denominadas dependências internas) são imutáveis. Conseqüentemente, os registradores arquiteturais consultados e atualizados pelas instruções de um bloco básico sempre serão os mesmos, ou seja, toda vez que um mesmo bloco básico for ativado, as dependências entre suas instruções serão sempre as mesmas. Para instruções localizadas em blocos básicos distintos, nem sempre isto ocorre, pois a ordem de execução dos blocos básicos pode ser alterada dinamicamente.

Caso duas ou mais instruções de um mesmo bloco básico atualizem um mesmo registrador, apenas o resultado da instrução mais recente é que precisa ser transferido para o registrador arquitetural, ficando assim disponível para outros blocos básicos.

Estas características estimularam o desenvolvimento de três mecanismos de detecção de dependências. Cada mecanismo usa um tipo de memória *cache*, denominada *cache* de dependências, cujo objetivo é reduzir o total de comparações para determinar as dependências verdadeiras

entre as instruções que estão sendo despachadas no mesmo mesmo ciclo.

A *cache* de dependências é uma estrutura que armazena as relações de dependências entre as instruções de um mesmo bloco básico. A coleta e armazenamento dessas relações pode ser realizada: através de ferramentas estáticas que analisariam o código objeto antes de sua execução; por mecanismos de controle localizados em estágios anteriores ao despacho, como por exemplo no estágio de busca de instruções da memória principal para a *cache* de instruções; finalmente, uma terceira opção seria realizar essas tarefas durante o próprio estágio de despacho, neste caso implementado segundo um *pipeline*. A escolha da alternativa dependerá das características de cada projeto.

Os três modelos de *cache* de dependências desenvolvidos são: simples (CS), inteligente (CI) e avançada (CA). A complexidade da *cache* de dependências varia conforme o mecanismo de detecção, assim como os benefícios que podem ser obtidos com o seu uso. Uma arquitetura básica (DE), usada como referência, é compartilhada pelos três modelos de *cache* de dependências. Modificações são adicionadas à *cache* simples, de modo a obtermos as *caches* inteligente e avançada.

A. Cache Simples

A *cache* simples armazena as relações de dependências entre as instruções de um bloco básico (dependências internas).

Sua estrutura possui N entradas, indexadas pelos $log_2(N+2)$ *bits* menos significativos do endereço do bloco básico, onde N é uma potência de dois. Ela é uma *cache* com mapeamento direto e cada entrada possui um *tag* contendo os $32 - log_2(N+2)$ *bits* mais significativos do endereço do bloco básico. O diagrama de cada entrada da *cache* simples é apresentado na Figura 1.

Cada entrada é formada pelo *tag* de endereço, um campo de 4 *bits* que indica o total de instruções do bloco básico e até 16 campos de identificação de dependências, um campo para cada instrução do bloco básico. Cada um desses campos de identificação possui 12 *bits* que apontam para as instruções precedentes (pertencentes ao mesmo bloco básico) que produzirão os operandos fonte de cada instrução.

As instruções de nossa arquitetura base possuem até três operandos fonte. Por essa razão, cada campo de identificação de dependências é formado por três grupos de 4 *bits*. Eles apontam, respectivamente, para a instrução dentro do bloco básico que gera o primeiro e o segundo operando fonte, além do código de condição. Se nenhuma instrução no mesmo bloco básico produzir o operando, o grupo de 4 *bits* conterá zeros.

Dependendo da arquitetura experimental empregada, o *buffer* de despacho é preenchido com 8 ou 16 instruções por ciclo. Para cada instrução desse *buffer*, o mecanismo de detecção examina a *cache* simples para determinar se os seus operandos fonte são produzidos por instruções no mesmo bloco básico ou não.

Se o campo correspondente na *cache* de dependências contiver zeros, estamos diante de um dos três casos: o operando fonte já pode ter sido gerado num ciclo anterior; ele está sendo avaliado por uma instrução despachada num ciclo anterior; ou finalmente, o operando fonte será produzido por uma instrução precedente (de um outro bloco básico) e que está no mesmo *buffer* de despacho.

Independente do caso, para cada operando fonte da instrução armazenada na k-ésima entrada do *buffer* de despacho, o primeiro mecanismo de detecção examina os registradores destino das $k-1$ primeiras instruções do *buffer*.

Havendo uma instrução precedente cujo registrador destino será usado como fonte pela k-ésima instrução, o mecanismo reconhece que essa é a instrução que produzirá o operando fonte, e o *tag* de sua estação de reserva é que será usado no lugar do operando fonte. Se não houver uma instrução precedente no *buffer* e se o resultado já foi gerado em um ciclo anterior, o operando será lido do banco de registradores. Se ele ainda estiver sendo gerado por uma instrução despachada num ciclo anterior, o *tag* identificador da estação de reserva correspondente será usado.

Ao especificarmos o mecanismo de detecção que emprega a *cache* simples, estávamos interessados em reduzir o número de comparações sem ter que assumir os elevados custos exigidos pelos outros dois mecanismos que, conforme será visto, são mais eficientes.

B. Cache Inteligente

A estrutura básica é a mesma da *cache* simples, conforme mostrado na Figura 1, mudando apenas o algoritmo utilizado.

No modelo anterior, para cada dependência não resolvida, é necessário examinar todas as instruções precedentes que estão sendo despachadas no mesmo ciclo. Entretanto, as dependências de cada instrução no mesmo bloco básico já estão determinadas e armazenadas. Para as instruções com dependências não definidas, isto é, grupos contendo zeros, só é necessário examinar as instruções precedentes que pertençam a outros blocos básicos.

A primeira conclusão desta observação é que as instruções do primeiro bloco básico no *buffer* de despacho não necessitam de comparações, pois as instruções com dependências não resolvidas dependem de blocos básicos despachados em ciclos anteriores. O operando fonte ou o *tag* correspondente será buscado no banco de registradores.

Para os demais blocos básicos no *buffer* de despacho, o campo de índice que acompanha cada instrução é diminuído do seu índice no *buffer* de despacho. O resultado fornece o índice dentro do *buffer* da primeira instrução a partir da qual as comparações deverão ser feitas, ou seja, a última instrução

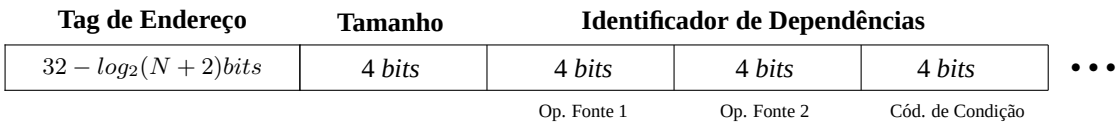

Figura 1: Controle das Dependências Internas

do bloco básico anterior.

Esta otimização no algoritmo reduz ainda mais o número de comparações necessárias para despachar cada instrução: precisamos examinar somente as instruções dos blocos precedentes que estiverem no mesmo *buffer* de despacho. Existe o custo adicional do *hardware* que calcula o indíce da primeira instrução a ser comparada: um somador de 4 *bits* para cada entrada do *buffer* de despacho. Além disto, um pequeno aumento na complexidade da lógica de controle é esperado, mas os custos de armazemanento são os mesmos como na *cache* simples.

Quando a *cache* de dependências indicar que existe uma relação de dependência com outra instrução no **mesmo** bloco básico, precisamos verificar se a instrução está no *buffer* de despacho ou não. Para isto, realiza-se a diferença entre o índice extraído da *cache* de dependências e o índice da primeira instrução do bloco básico que está armazenado no *buffer*. Se o resultado for negativo, então a instrução que gera o operando já foi despachada. Se o resultado for positivo, então ele indica a posição no *buffer* de despacho da instrução que produzirá o operando fonte.

Com esta informação, obtém-se o identificador da estação de reserva para onde a instrução precedente será despachada e enviamos este *tag* para a estação de reserva da nova instrução.

C. Cache Avançada

A *cache* de dependências avançada armazena outras informações: conjunto de registradores alterados pelo bloco básico, e qual a última instrução que modificou cada registrador.

Essa informação é armazenada em um sub-campo adicional de 6 *bits* no campo identificador de dependências (supondo-se 32 registradores inteiros e 32 de ponto flutuante) e indica qual o registrador alterado por cada instrução do bloco básico. Esse esquema está ilustrado na Figura 2.

Este modelo de *cache* requer mais 96 *bits* para cada entrada da *cache* para armazenar esta informação adicional. Outra forma de codificação seria utilizar uma máscara de 64 *bits* para cada entrada. A máscara indica se o registrador é alterado (neste caso o *bit*=1) e em seguida um vetor com 4 *bits* por posição (uma para cada *bit*=1 na máscara), indicando qual a última instrução que alterou o registrador. Essa alternativa é apresentada na Figura 3

Além de armazenar as informações de dependência entre as instruções do bloco básico, a *cache* avançada também aponta para as instruções que modificarão o banco de registradores.

Os identificadores de registradores armazenados são os arquiteturais, pois no estágio de despacho de nossa máquina básica, eles ainda não foram renomeados. Quando mais de um bloco básico estiver armazenado no *buffer* de despacho (o que é comum em programas inteiros), esta informação será utilizada pelas instruções dos blocos sucessores para identificar dependência com instruções de blocos precedentes que estão armazenados no *buffer* de despacho.

O mecanismo de detecção examina a *cache* avançada e as informações de dependência dos blocos básicos contidos no *buffer* de despacho são recuperadas. Como vimos, todas as instruções do primeiro bloco podem ser despachadas, levando-se em conta apenas as dependências internas. Os blocos subseqüentes extraem as dependências internas da *cache* e para os campos preenchidos com zeros, o seguinte procedimento é realizado:

- verifica-se quais são os registradores fonte de cada instrução e se eles são gerados por algum dos blocos básicos precedentes, cuja informação foi extraída da *cache*;
- caso afirmativo, o índice da instrução que produz o fonte é obtido através da *cache*. Caso contrário, o operando é lido do banco de registradores;
- são realizadas operações (similares à *cache* inteligente) para verificar se a instrução com a qual existe dependência está no mesmo *buffer* de despacho;
- se afirmativo, o *tag* desta instrução é que será enviado para a estação de reserva. Caso contrário, o operando ou *tag* armazenado no banco de registradores será usado.

Com as informações adicionais é possível reduzir significativamente o número de comparações, além de permitir que apenas os resultados "vivos" na saída do bloco básico sejam enviados para o banco de registradores. Os demais valores serão capturados apenas pelas respectivas estações de reserva que estiverem aguardando por esses valores. Esta providência reduz a pressão sobre o banco de registradores.

IV. O Ambiente de Simulação

Para avaliar os mecanismos de detecção, utilizamos os programas do conjunto SPEC95. Esses programas foram compilados para a arquitetura SPARC, sem fazer contudo o uso da "janela" de registradores e do *delay slot.* Em seguida,

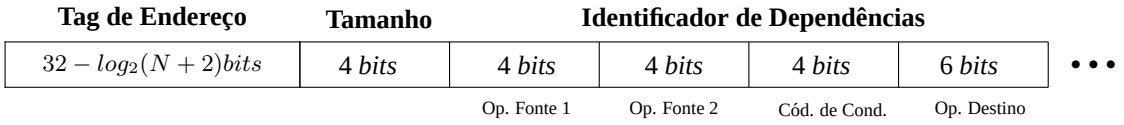

Figura 2: Controle das Dependências Internas - Opção 1

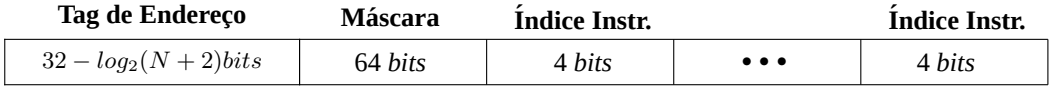

Figura 3: Controle das Dependências Internas - Opção 2

eles foram executados pelo emulador *shade* [CEM 93] que reproduz o comportamento da arquitetura SPARC e o fluxo correto de execução dos programas foi gerado.

Este fluxo foi processado pelos simuladores dos mecanismos que especificamos: simuladores escritos em linguagem "C" e compilados como módulos de biblioteca do emulador. Desse modo, as instruções eram passadas da memória do emulador *shade* para a memória do simulador. Após reproduzir o comportamento do respectivo mecanismo de detecção, o simulador coletava as estatísticas geradas pela arquitetura experimental.

Nos modelos simulados consideramos um número ilimitado de recursos, seja de unidades funcionais, capacidade da memória cache, capacidade das *caches* de dependência e outros. Os dados obtidos refletem então o máximo de desempenho que poderá ser obtido em condições ideais. Na prática, os resultados deverão ser mais modestos.

Os programas inteiros e de ponto flutuante do SPEC95, foram simulados integralmente com a entrada de dados *training.* Apenas as instruções de chamada ao Sistema Operacional não foram simuladas, por limitações do emulador *shade.*

V. Resultados dos Experimentos

Nesta seção apresentamos resultados produzidos por oito máquinas experimentais: duas máquinas usadas como referência e que despacham 8 e 16 instruções por ciclo (denominadas 8-DE e 16-DE); três máquinas equipadas com os mecanismos de deteção e que despacham 8 instruções (8-CS, 8-CI e 8-CA); e as outras três que despacham 16 instruções por ciclo (16-CS, 16-CI e 16-CA).

Inicialmente, realizamos experimentos para determinar o total dinâmico de operandos fonte requeridos pelas instruções dos programas da bateria de teste. As Tabelas I e II mostram o percentual de instruções de cada programa de acordo com o número de operandos fonte, sejam eles explícitos (registradores) ou implícitos (código de condição).

Instruções sem operandos fonte (vide última coluna das tabelas), podem ser despachadas imediatamente; para as instruções com 1 ou mais operandos, é necessário verificar se eles são gerados por uma outra instrução precedente que está sendo despachada no mesmo ciclo. Por essa razão, se a *cache* de dependências não estiver sendo utilizada, teremos que examinar todas as instruções precedentes que estejam sendo despachadas em paralelo.

Oper. Não-Imed.	3	2	1	0
Aplicação				
go	1,5	17,5	59,3	21,7
m88ksim	0,2	20,0	67,5	12,5
gcc	0,5	14,8	74,5	10,2
compress	1,3	26,0	57,0	15,8
li	0,1	14,8	75,8	9,3
ijpeg	18,8	30,7	46,2	4,2
perl	1,0	18,2	68,7	12,0
vortex	0,2	13,7	65,9	20,3

Tabela I
DISTRIBUIÇÃO POR NÚMERO DE DEPENDÊNCIAS - INTEIROS

Oper. Não-Imed.	3	2	1	0
Aplicação				
tomcatv	0,0	49,1	48,8	2,0
swim	6,9	67,0	21,6	4,3
su2cor	0,5	49,9	45,0	4,6
hydro2d	0,1	37,9	53,7	8,4
mgrid	0,1	43,0	56,7	0,1
applu	0,1	37,5	61,6	0,7
turb3d	1,7	46,2	49,1	2,9
apsi	2,3	44,2	50,5	2,9
fpppp	0,3	33,8	65,3	0,6
wave5	2,1	44,5	48,8	4,5

Tabela II
DISTRIBUIÇÃO POR NÚMERO DE DEPENDÊNCIAS - PONTO FLUTUANTE

Os programas inteiros possuem um número elevado de instruções independentes: entre 4,2% (ijpeg) e 21,7% (go) das instruções podem ser despachadas a cada ciclo sem comparações. A maioria das instruções dos programas inteiros possui apenas um operando de leitura: percentagens variando de 46,2% (ijpeg) a 75,8% (li). A porcentagem de

instruções com dois operandos fonte varia entre 14,8% (gcc e li) e 30,7%(ijpeg). Finalmente, a menor percentagem é o de instruções com três operandos fonte. Ela varia de 0,1% (li) a 1,5% (go). A exceção é o programa ijpeg que apresentou 18,8% de suas instruções com três dependências.

Os programas de ponto flutuante possuem poucas instruções independentes: entre 0,1% (mgrid) e 8,4% (hydro2d). A maioria das instruções possui um operando fonte, entre 21,6% (swim) e 65,33% (fpppp), mas o percentual de instruções com dois operandos também é alto: entre 33,8% (fpppp) e 67,0% (swim). Como ocorreu com os programas inteiros, o percentual das instruções com três dependências é bastante reduzido: entre 0% (tomcatv) e 2,3% (apsi). A exceção fica com o programa swim (6,9% das instruções com três operandos fonte).

Este perfil de execução indica que o número médio de comparadores necessários para despachar múltiplas instruções em um programa inteiro é menor do que em um programa de ponto flutuante.

As Tabelas III, IV, V e VI apresentam o total (expresso em milhões) de comparações realizadas por cada uma das oito máquinas experimentais quando do despacho das instruções do conjunto SPEC95.

Despacho	16-DE	16-CS	16-CI	16-CA
Aplicação				
go	8.678	2.384	1.570	2.106
m88ksim	1.199	678	509	108
gcc	11.733	6.352	5.091	1.178
compress	3.142	1.334	949	176
li	1.874	842	654	122
ijpeg	25.974	11.729	6.760	1.057
perl	18.922	11.187	9.130	2.110
vortex	17.056	8.879	7.019	1.448

Tabela III
COMPARAÇÕES (EM MILHÕES) - APLICAÇÕES INTEIRAS

Despacho	8-DE	8-CS	8-CI	8-CA
Aplicação				
go	4.049	1.112	561	86
m88ksim	559	315	194	50
gcc	5.475	2.964	1.999	537
compress	1.466	623	336	79
li	874	393	245	50
ijpeg	12.121	5.473	2.350	456
perl	8.831	5.220	3.547	940
vortex	7.959	4.143	2.684	642

Tabela IV
COMPARAÇÕES (EM MILHÕES) - APLICAÇÕES INTEIRAS

Os gráficos das Figuras 4 e 5 mostram o número **médio** de comparações para despachar cada instrução dos programas inteiros. Observa-se uma redução significativa no número de comparações para as duas larguras de despacho e que as máquinas equipadas com a *cache* avançada apresentaram os

Desp.	16-DE	16-CS	16-CI	16-CA
Aplic.				
tomcatv	99.314	61.546	29.907	3.199
swim	5.781	3.437	1.625	204
su2cor	179.318	117.582	56.081	6.995
hydro2d	73.173	51.190	29.297	4.186
mgrid	189.270	93.951	47.826	4.264
applu	5.694	3.451	1.716	215
turb3d	192.474	131.280	63.826	8.701
apsi	27.660	16.825	8.712	1.129
fpppp	4.805	2.559	1.245	116
wave5	39.018	22.224	11.466	1.459

Tabela V
COMPARAÇÕES (MILHÕES) - APLICAÇÕES DE PONTO FLUTUANTE

Desp.	8-DE	8-CS	8-CI	8-CA
Aplic.				
tomcatv	46.345	28.721	8.961	1.285
swim	2.698	1.604	482	91
su2cor	83.682	54.872	18.368	2.964
hydro2d	34.145	23.888	9.406	1.858
mgrid	88.325	43.843	14.877	1.763
applu	2.657	1.610	515	94
turb3d	89.820	61.264	19.289	3.829
apsi	12.908	7.851	2.735	492
fpppp	2.242	1.194	372	48
wave5	18.211	10.372	3.548	627

Tabela VI
COMPARAÇÕES (MILHÕES) - APLICAÇÕES DE PONTO FLUTUANTE

melhores resultados. Para a *cache* simples, a redução média no número de comparações foi de 53,4% para todas as larguras de despacho. Para a arquitetura equipada com a *cache* inteligente e que despacha 16 instruções, a redução média foi de 65,9%; com 8 instruções, a redução foi de 72,8%. O mecanismo que usa a *cache* avançada apresentou uma redução média de 93,5% (despacho de 16 instruções) e 97,1% (8 instruções despachadas por ciclo).

Os gráficos das Figuras 6 e 7 apresentam o número **médio** de comparações para despachar cada instrução dos programas de ponto flutuante. Para a *cache* simples a redução média no número de comparações foi de 39,7% em todos os casos, valores bem mais modestos se comparados aos obtidos com os programas inteiros, já que o tamanho médio dos blocos básicos é maior. Para a *cache* inteligente e largura de despacho de 16 instruções, o número médio de comparações foi reduzido (em média) de 69.7%; para largura de 8 instruções este valor foi de 80,2%. A *cache* avançada mostra reduções em torno de 96,4% para largura de 16 instruções e de 98,5% para 8 instruções.

Embora a *cache* simples apresente reduções mais modestas quando se executa programas de ponto flutuante, as *caches* inteligente e avançada apresentam reduções tão signicativas quanto as obtidas com a execução de programas inteiros.

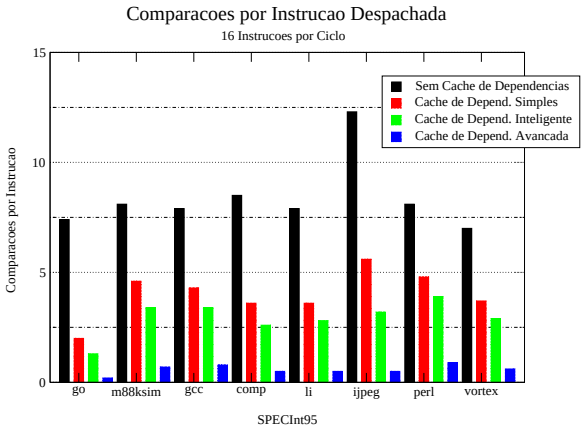

Figura 4. Comparações por Instrução Despachada - 8 Instruções/ciclo

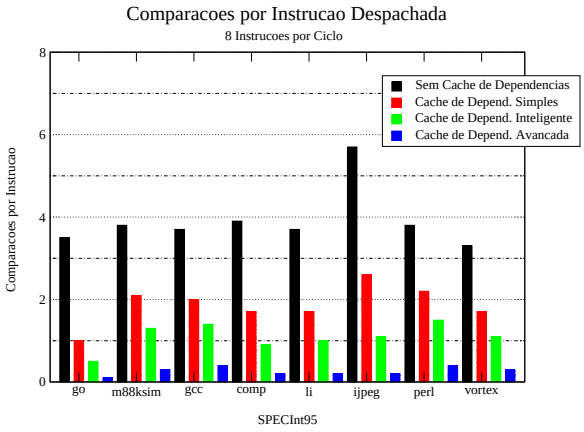

Figura 5. Comparações por Instrução Despachada - 16 Instruções/ciclo

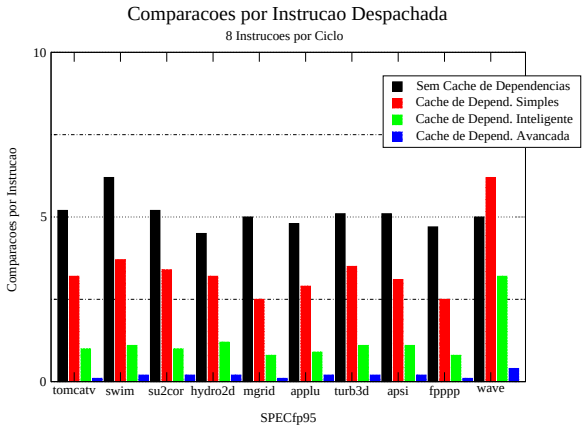

Figura 6. Comparações por Instrução Despachada - 8 Instruções/ciclo

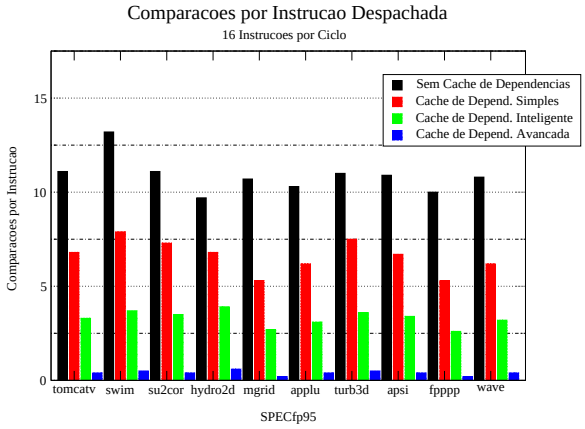

Figura 7. Comparações por Instrução Despachada - 16 Instruções /ciclo

VI. Conclusões

Para a detecção mais eficiente de dependências dados, especificamos e simulamos três mecanismos que empregam memórias *cache* que armazenam as relações de dependências entre as instruções de um mesmo bloco básico. Os mecanismos viabilizam o despacho de um grande número de instruções em paralelo e reduzem a complexidade $O(n^2)$ do *hardware* que seria empregado por arquiteturas super escalares convencionais.

Durante nossos experimentos, o mecanismo equipado com a *cache* de dependências avançada apresentou os melhores resultados, mesmo para os programas de ponto flutuante. Verificamos que o número médio de comparadores requeridos para o despacho de instruções apresentou uma significativa redução: ao invés dos 197 comparadores exigidos pelos mecanismos convencionais, para apenas 15, no caso dos programas inteiros, e de 211 para apenas 10, no caso de programas de ponto flutuante, ambos com uma largura de despacho de 16 instruções. Houve uma redução no número médio de comparadores necessários ao despacho das instruções de um total de 46 para apenas 4, no caso dos programas inteiros, e de 50 para apenas 2, no caso de programas de ponto flutuante, quando a largura de despacho for de 8 instruções.

Mesmo para a *cache* de dependências inteligente, que possui menor complexidade de implementação, o número médio de comparadores necessários foi reduzido de 197 para 63 e de 211 para 63 comparadores, respectivamente para os programas inteiros e de ponto flutuante, com largura de despacho de 16 instruções. O número médio de comparadores necessários foi reduzido de 46 para 10 comparadores para os programas inteiros e de 50 comparadores para 12 nos de ponto flutuante,

com largura de despacho de 8 instruções.

Na *cache* simples, de menor complexidade de implementação, a redução foi de 197 para 90 comparadores no caso dos programas inteiros e de 211 para 126 registradores com os programas de ponto flutuante, também para uma largura de despacho de 16 instruções. Para uma largura de 8 instruções, a redução foi de 46 para 21 comparadores no caso dos programas inteiros e para 50 para 30 comparadores com os programas de ponto flutuante.

Estes resultados são ideais, pois estamos supondo um número ilimitado de recursos. Implementações reais deste mecanismo devem apresentar um desempenho mais modesto do que os obtidos nos nossos experimentos. Os custos associados a estas implementações devem ser maiores, pois apenas contabilizamos o custo associado ao armazenamento dos dados na *cache* de dependências. Acreditamos porém que o potencial de nossos mecanismos tenha sido demonstrado plenamente, pela significativa redução no número de comparadores e barramentos.

Na realidade, os mecanismos que manipulam a *cache* de dependência podem (e devem) ser associados a outros esquemas de armazenamento, como por exemplo uma *cache* de blocos básicos ou com *trace cache*. Neste caso os custos são compartilhados pelos benefícios oferecidos por estes mecanismos, como redução da largura de banda necessária para a busca de instruções.

Não menos importante, as estruturas de dados apresentadas permitem identificar os resultados que precisam ser enviados para os registradores arquiteturais e/ou *future file*. Os resultados que serão usados apenas pelas instruções do mesmo bloco básico, serão transmitidos somente para as estações de reserva que estiverem aguardando por eles. Isto reduz o número de portas do banco de registradores e do *tag array*, resultando em dispositivos mais simples e com menor tempo de acesso.

REFERÊNCIAS

[JOH 91] JOHNSON, M. *Superscalar Microprocessor Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[PAL 96] PALACHALA, Subbarao; JOUPPI, Norman P.; SMITH, James E. Quantifying the complexity of superscalar processors. *Technical Report CS-TR-1996-1328*, CS/ECE - University of Wisconsin-Madison, November 1996.

[TOM 67] TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[IBM 94] IBM; MOTOROLA. *PowerPC 604 - RISC Microprocessor User's Manual*, 1994.

[WIL 95] WILLIAMS, Ted; PATKAR, Niteen; SHEN, Gene. SPARC64: A 64-bit 64-active-instruction out-of-order-execution MCM processor *IEEE Journal of Solid-State Circuits*, 30(11):1215–1226, November 1995.

[INT 98] INTEL. *P6 Family of Processors - Hardware Developer's Manual*, Intel, September 1998.

[PAT 90] PATTERSON, David A.; HENNESY, John L. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[FER 00] FERNANDES, Edil S. T.; WOLFE, Andrew; SILVA, Gabriel P. Towards BBM - a basic block machine. *Technical Report 526/00*, COPPE/Sistemas, Federal University of Rio de Janeiro, January 2000.

[CME 93] CMELICK, Robert F.; KEPPEL, David. Shade: A fast instruction-set simulator for executing profiling. *Technical Report SMLI TR-93-12*, Sun Microsystems Laboratories, 1993. Also published as Tech. Report CSE-TR 93-06-06, University of Washington.