DTSD: Uma Arquitetura com Mecanismo Híbrido de Execução

Fernando Líbio Leite Almeida[1], Alberto F. De Souza[1], Edil S. T. Fernandes[2]

[1] *Departamento de Informática*
Universidade Federal do Espírito Santo
flibio@lcad.inf.ufes.br, alberto@lcad.inf.ufes.br

[2] *Programa de Engenharia de Sistemas e Computação – COPPE*
Universidade Federal do Rio de Janeiro
edil@cos.ufrj.br

Resumo

Este trabalho apresenta a Dynamically Trace Scheduling Dataflow *(DTSD), uma arquitetura híbrida que executa código em dois modos distintos: de fluxo de controle e de fluxo de dados (**control-flow/dataflow**). Máquinas DTSD incluem: um processador primário que executa instruções escalares de forma tradicional (fluxo de controle); uma unidade que traduz dinamicamente trechos de código escalar em blocos de instruções* Explicit Data Graph Execution*; e um processador* Dataflow *que executa estes blocos segundo o fluxo de dados. Esta nova arquitetura foi avaliada experimentalmente: implementamos um simulador DTSD com substrato Simplescalar, e empregamos um ambiente de simulação da arquitetura Super Escalar Alpha 21264 em análise comparativa. Nossos resultados experimentais mostraram que, em média, o desempenho da arquitetura DTSD supera o de uma Super Escalar equivalente em 183%.*

1. Introdução

A arquitetura da maioria dos processadores de alto desempenho de uso geral da atualidade é do tipo Super Escalar com ou sem *Trace Cache* [1, 2]. Eles alcançam alto desempenho através da exploração do paralelismo no nível de instrução (*Instruction-Level Parallelism* – ILP) existente nos programas [3]. Para explorar este ILP, várias instruções são escalonadas para execução paralela nas unidades funcionais do processador a cada ciclo de relógio.

Recentemente, uma nova classe de arquitetura do nível de instrução (*Instruction Set Architecture* – ISA), denominada *Explicit Data Graph Execution* (EDGE) [4] foi proposta. Em uma ISA EDGE, os programas são grafos e as instruções são os nós do grafo, e cada uma delas pode ser executada tão logo seus operandos de entrada estejam prontos. Os operandos de cada instrução não precisam ser especificados explicitamente; porém, devemos especificar a unidade funcional onde elas serão executadas e quais unidades funcionais receberão os resultados de cada instrução. ISAs EDGE possuem, assim, uma característica marcante: a comunicação direta entre instruções (que compõe as arestas do grafo que representa o programa). Com esta comunicação direta, as instruções são executas segundo o fluxo de dados, i.e., máquinas EDGE são do tipo *dataflow* [5, 6].

O principal objetivo deste trabalho é investigar arquiteturas de processador que permitam a tradução dinâmica de código escalar de ISAs existentes para código *dataflow*. Propomos, para tal, arquiteturas *Dynamically Trace Scheduling Dataflow* (DTSD). Máquinas DTSD absorvem características da arquitetura DTSVLIW [7, 8] e de outras arquiteturas [4] para alcançar desempenho. Em arquiteturas DTSD, instruções escalares são trazidas, uma a uma, da *cache* de instruções e executadas por um processador *pipelined* simples – o Processador Primário da arquitetura (Figura 1). A Unidade de Escalonamento DTSD realiza escalonamento *dataflow* da seqüencia dinâmica de execução destas instruções escalares, montando, com isso, blocos de instruções *dataflow*. Estes blocos são armazenados dinamicamente na *cache* de blocos de instruções *dataflow* da DTSD. Se um mesmo trecho de código necessitar ser executado novamente, as instruções deste trecho podem ser fornecidas pela Cache *Dataflow* e executadas pela Máquina *Dataflow* da DTSD (Figura 1).

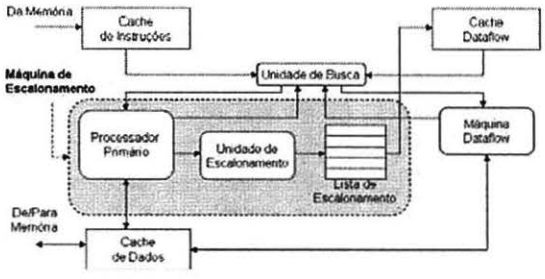

Figura 1. Diagrama de blocos de uma Arquitetura DTSD.

Para comparar o desempenho de processadores DTSD com o de processadores atuais, implementamos um simulador paramétrico DTSD baseado no simulador Simplescalar [9] e comparamos o desempenho de várias configurações de processador DTSD com o desempenho do processador Alpha 21264 [10]. Nossos resultados mostraram que processadores DTSD com complexidade equivalente à do processador Alpha 21264 superam o desempenho deste na maioria dos casos examinados.

2. Uma Arquitetura DTSD

Em máquinas DTSD, a cada ciclo de relógio a Unidade de Busca (Figura 1) examina se o fluxo de controle aponta para um bloco de instruções *dataflow* já escalonado e salvo na Cache *Dataflow*. Em caso de *miss* na Cache *Dataflow*, instruções são buscadas da Cache de Instruções e executadas no Processador Primário, mas também escalonadas e salvas na Cache *Dataflow*. Em caso de *hit* na Cache *Dataflow*, o controle DTSD passa para o modo de execução *dataflow*. Quando a execução do bloco de instruções *dataflow* for concluída, se o fluxo de controle apontar novamente para um bloco de instruções já escalonado e salvo, este bloco é buscado e o ciclo de execução *dataflow* se repete. Deste modo, uma máquina DTSD passa a maior parte do tempo executando apenas em modo *dataflow*.

Em uma DTSD:

- As instruções são agrupadas em blocos durante o escalonamento e numeradas de acordo com sua posição no bloco;
- Uma instrução especifica o código da operação que será executada, seus operandos de entrada, quantos *tokens* ela precisa receber, e quais são as instruções que receberão *tokens* dela;
- Instruções não lêem ou escrevem em registradores da ISA (exceto aquelas com este fim específico);
- Uma instrução está pronta para ser executada quando recebe todos os *tokens* que precisa, ou se não depende de nenhum *token*.

No modo de execução *dataflow*, *tokens* informam sobre a disponibilidade dos dados necessários para iniciar a execução de cada instrução. O trecho de código escalar da Figura 2(a) e sua tradução para código *dataflow*, apresentada na Figura 2(b), mostram como.

No código da Figura 2(b), as instruções são numeradas de 1 a 9. Ao lado do número aparece o código da operação de cada instrução e seus operandos de saída e entrada. Após o primeiro caractere ":" é indicado o número de *tokens* que a instrução precisa receber antes de poder executar e, após o segundo, a lista de instruções para as quais a instrução envia *tokens*.

As instruções de 1 a 4 são responsáveis pela leitura de registradores que serão operados pelas instruções 5, 6 e 7. Todas as quatro (de 1 a 4) podem ser executadas em paralelo e copiam os registradores da ISA (R2, R3, R5 e R6) para os registradores de renomeação (OP1 a OP4). A renomeação é realizada durante a transformação do código seqüencial em *dataflow* (escalonamento) e elimina as dependências de saída e anti-dependências. A instrução 5 pode executar tão logo as instruções 1 e 2 terminem, enquanto que a 6 depende das instruções 3 e 5. A instrução 7 depende da 4 e da 6, a 8 da 7, e a 9 da 6.

No código escalar são feitas 6 leituras e 3 escritas nos registradores da ISA, enquanto que no código *dataflow* são feitas apenas 4 leituras e 2 escritas. Contudo, supondo que as instruções escalares e *dataflow* necessitem de apenas um ciclo para executar, o código *dataflow* requer

```
ADD     R1, R2, R3      # R1 = R2 + R3
SUB     R4, R1, R5
ADD     R1, R4, R6
```

(a)

```
1.RR OP1,R2:0:5          # lê o conteúdo de R2 (Read Register - RR) para OP1, ou seja, OP1 = R2,
                           nenhum token necessário, envia um token para a instrução 5
2.RR OP2,R3:0:5
3.RR OP3,R5:0:6
4.RR OP4,R6:0:7
5.ADD OP1,OP1,OP2:2:6    # OP1=OP1+OP2, 2 tokens necessários, envia um token para a instrução 6,
                           recebe um token
6.SUB OP1,OP1,OP3:2:7,9  # OP1=OP1-OP3, 2 tokens necessários, envia tokens para as instruções 7 e 9
7.ADD OP2,OP1,OP4:2:8    # OP2=OP1+OP4, 2 tokens necessários, envia um token para a instrução 8
8.WR R1,OP2:1:           # Escreve o conteúdo de OP2 em R1 (Write Register - WR), R1 = OP2, 1 token
                           necessário, a instrução não envia nenhum token
9.WR R4,OP1:1:
```

(b)

Ciclo 1:	1.RR OP1, R2:0:5	2.RR OP2, R3:0:5	3.RR OP3, R5:0:6	4.RR OP4, R6:0:7
Ciclo 2:	5.ADD OP1, OP1, OP2:2:6			
Ciclo 3:	6.SUB OP1, OP1, OP3:2:7,9			
Ciclo 4:	7.ADD OP2, OP1, OP4:2:8	9.WR R4, OP1:1:		
Ciclo 5:	8.WR R1, OP2:1:			

(c)

Figura 2: (a) Código escalar. (b) Código escalar transformado em *dataflow*. (c) Seqüência de execução *dataflow*.

mais ciclos que o escalar – 5 ciclos contra 3 – como mostra a seqüência de execução *dataflow* da Figura 2(c) (cada linha contém as instruções que podem ser executadas em cada ciclo). No entanto é importante observar que, em uma máquina escalar, a leitura e a escrita nos registradores requer tempo que é apenas explicitado no código *dataflow*. Além disso, em blocos *dataflow* maiores, as leituras e escritas em registradores da ISA escalar são necessárias apenas no início e no final da execução do bloco, respectivamente, o que reduz o número de acessos ao banco de registradores. Por fim, os registradores de renomeação OP1, OP2, OP3 e OP4 podem ser armazenados em um banco de registradores menor e, eventualmente, com um número menor de portas de leitura e escrita, o que resulta em um tempo de acesso e consumo de energia menores.

2.1. Desvios Condicionais e Execução Especulativa

Para mostrar como o escalonamento de instruções de desvio condicional e execução especulativa são realizados na arquitetura DTSD, fazemos uso do trecho de código da Figura 3. Considerando que os registradores R1, R2 e R4 inicialmente possuem os valores 2, 3 e 4, respectivamente, obtemos o trecho de código *dataflow* da Figura 4, produzido dinamicamente a partir da tradução do caminho de execução definido pelos valores iniciais dos registradores R1, R2 e R4.

```
        ADD R3, R1, R2    # R3 = R1 + R2
Loop:   SUB R5, R3, R4
        ADD R4, R4, 1
        BNE R5, Loop
Exit:
```

Figura 3. Trecho de código escalar com desvio condicional

```
1. RR OP1, R1:0:3
2. RR OP2, R2:0:3
3. ADD OP3, OP1, OP2:2:4,6
4. WR OP3, R3:1:
5. RR OP4, R4:0:6,8
6. SUB OP5, OP3, OP4:2:7,10
7. WR OP5, R5:2:
8. ADD OP6, OP4, 1:1:9,11,13
9. WR OP6, R4:2:
10. BNE OP5, LOOP:1:7,9
11. SUB OP7, OP3, OP6:2:12,15
12. WR OP7, R5:1:
13. ADD OP8, OP6, 1:1:14
14. WR OP8, R4:1:
15. BNE OP7, LOOP:1:12,14
Exit:
```

Figura 4. Código da Figura 5 transformado em *dataflow*

Na DTSD, uma instrução de desvio condicional em um bloco *dataflow* "possui memória" do caminho que ela tomou quando de sua execução pelo Processador Primário e, diferente das demais instruções, pode enviar *tokens* de dois tipos. Caso, durante a execução *dataflow*, seja observado o mesmo caminho da execução seqüencial, o desvio condicional envia *tokens* especiais que cancelam escritas anteriores que possam ser dispensadas, ou seja, escritas em registradores que serão sobrescritos posteriormente. Caso o caminho não seja o mesmo observado na execução seqüencial, o desvio envia *tokens* que atuam como *tokens* usuais, validando a execução das instruções de escrita e comandando uma mudança do modo execução *dataflow* para o modo de execução fluxo de controle.

No modo de execução *dataflow*, desvios condicionais estabelecem, também, grupos de instruções sob sua influência. Um grupo de instruções é caracterizado pelo seu número de ordem. O número de ordem de cada instrução é armazenado junto com ela durante o escalonamento de cada bloco. Em cada bloco, o número de ordem inicia em zero, sendo incrementado a cada desvio condicional escalonado. O tipo de *token* e o número de ordem das instruções são informações necessárias para controlar a execução de escritas no banco de registradores, como detalhado a seguir.

A DTSD atua como um interpretador da ISA original. Quando em modo *dataflow*, o estado da ISA original, representado pelo conteúdo dos registradores da ISA e pelo estado da memória, progride do mesmo modo que faria em uma execução escalar, sendo que a maioria de suas alterações ocorre apenas quando da transição da execução de um bloco *dataflow* para outro, ou para o modo fluxo de controle do Processador Primário.

Os *tokens* enviados pelas instruções de desvio e o número de ordem das instruções são usados para garantir o correto estado do banco de registradores ao fim da execução de cada bloco, como mostrado na Figura 5.

O código *dataflow* apresentado na Figura 4 pode ser executado como mostrado na Figura 5 (as instruções 7 e 9 recebem *tokens* que as cancelam e não são executadas, dadas as condições iniciais apresentadas). Note que as instruções 11 e 13 são executadas especulativamente (antes do desvio 10 ser resolvido). Isto é possível porque a confirmação de seus resultados só se dá com as escritas 12 e 14, que somente são executadas após a verificação do desvio 15.

O código original (Figura 3) demoraria 7 ciclos em uma máquina escalar (3 por iteração), enquanto que o código *dataflow*, 6 ciclos (considerando leituras e escritas e 4 ciclos sem elas). Note que, se o *loop* for executado e

Ciclo 1:	1.RR OP1, R1:0:3	2.RR OP2, R2:0:3	5.RR OP4, R4:0:6,8	
Ciclo 2:	3.ADD OP3, OP1, OP2:2:4,6,11	8.ADD OP6, OP4, 1:1:9,11,13		
Ciclo 3:	4.WR OP3, R3:1:	6.SUB OP5, OP3, OP4:2:7,10	11.SUB OP7, OP3, OP6:2:12	13.ADD OP8, OP6, 1:1:14
Ciclo 4:	10.BNE OP5, LOOP:1:7,9			
Ciclo 5:	15.BNE OP7, LOOP:1:12,14			
Ciclo 6:	12.WR OP7, R5:1:	14.WR OP8, R4:1:		

Figura 5: Seqüência de execução dataflow. As instruções marcadas (11 a 15) possuem ordem 1, enquanto que as demais possuem ordem 0.

escalonado por mais iterações, o benefício da execução *dataflow* é ainda maior, uma vez que as instruções de leitura e escrita no banco de registradores da ISA só ocorrem uma vez e, com muitas iterações, mais instruções podem ser executadas em paralelo. Uma análise cuidadosa do código *dataflow* escalonado mostra que o número de ciclos despendidos em cada iteração do *loop* tende para 1 quando o número iterações escalonadas é elevado. É possível observar, também, na Figura 5, que instruções do código escalar são executadas fora da ordem original e até mesmo antes de desvios condicionais previstos no código original.

2.2. *Memory Disambiguation*

Em máquinas DTSD, um buffer de leitura e escrita na memória é utilizado para detectar acessos feitos à uma mesma posição de memória fora de ordem (leitura-escrita ou escrita-escrita). Para isso, do mesmo modo que na arquitetura DTSVLIW [8], a ordem dos acessos à memória é memorizada no escalonamento. Condições de *memory aliazing* são detectadas e tratadas durante a execução como exceções. O tratamento destas e de outras exceções é discutido na Seção 2.4.

2.3. Unidades de Execução

A Figura 6 mostra, à direita, o diagrama de blocos da Máquina de *Dataflow* da arquitetura DTSD proposta e, no detalhe (à esquerda), um bloco de unidades funcionais (BUF) DTSD.

A Máquina *Dataflow* da arquitetura DTSD busca instruções de uma cache de blocos *dataflow* e as executa em BUFs (três no exemplo da figura). O banco de registradores da ISA (REGS) e uma fila de leitura e escrita na memória (FLE), que apóia o acesso à hierarquia de memória (MEM), completam a Máquina *Dataflow*.

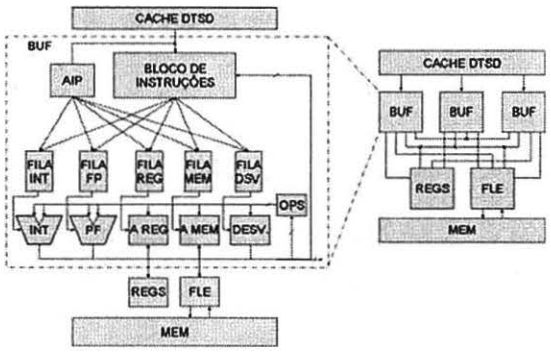

Figura 6. Máquina *Dataflow* da arquitetura DTSD

A Máquina *Dataflow* terá tantos BUFs quantos forem convenientes para otimizar o desempenho (número de instruções executadas por ciclo) face às restrições de freqüência de relógio e consumo de energia. Cada um deles, a cada ciclo, pode receber parte do bloco (uma linha) a ser executado. A cada ciclo, à medida que as instruções são lidas para cada BUF, o hardware de Adiantamento de Instruções Prontas (AIP) detecta quais instruções já vêm prontas para a execução, ou seja, não dependem de nenhum *token* para executar (como, por exemplo, as de leitura de registradores – RR), e as coloca diretamente na fila de instruções prontas correspondente (FILA INT, FILA FP, etc.).

O bloco a ser executado não é trazido todo de uma vez da Cache *Dataflow* em um único ciclo, mas sim uma linha por vez. Cada linha da cache pode conter uma fração do número de instruções do bloco. Instruções que dependem de *tokens* para executar levam consigo o número de *tokens* que necessitam receber para executar. Como o bloco não é trazido todo de uma vez da cache, em alguns casos é possível que um *token* seja enviado para uma instrução que ainda não foi lida da cache. Dessa forma, para garantir a execução correta, o Bloco de Instruções (Figura 6), estrutura que recebe as linhas de instruções buscadas da Cache *Dataflow*, contempla um contador de *tokens* recebidos para cada instrução do bloco, que pode conter valores negativos. Assim, se uma instrução depende de um único *token* para executar e este *token* chegar antes dela ser lida da cache, seu contador conterá o valor -1. Ao ser lida, o número de *tokens* necessário é somado ao seu respectivo contador e, neste caso, o contador da instrução passará ao valor zero, o que indicará que esta instrução está pronta para executar. Neste caso, a instrução será colocada diretamente na fila da unidade funcional para a qual foi escalonada.

Instruções prontas são enviadas para as respectivas unidades funcionais na ordem em que chegam à sua fila. Elas lêem seus operandos dos registradores de renomeação locais a cada BUF (OPS na Figura 6) antes de serem enviadas para execução. À medida que são executadas, seus resultados e *tokens* são enviados para os destinos especificados no escalonamento. Os *tokens*, ao chegarem ao Bloco de Instruções de um BUF, poderão tornar uma ou mais instruções prontas para serem executadas. As instruções que ficarem prontas pela chegada destes *tokens* serão despachadas automaticamente para a fila da unidade funcional para a qual foram escalonadas. Este ciclo se repete até que o bloco termine de ser executado.

O término da execução de um bloco se dará quando não houver nenhuma instrução em execução nas unidades funcionais ou quando um desvio seguir uma direção diferente da observada durante o escalonamento. Neste último caso, ao invés de cancelar as escritas a registradores para os quais envia *tokens*, o desvio irá habilitá-las e estas serão executadas. Além disto, a ordem do desvio será utilizada para cancelar instruções com ordem maior pertencentes ao bloco e presentes nos BUFs.

2.4. Tratamento de Exceções

Exceções podem ocorrer quando da execução de algumas instruções, como *load/store* (faltas de página, violações de acesso) ou divisão (divisão por zero). Além disso, em qualquer ponto da execução de um programa podem ocorrer interrupções. Para tratar exceções e interrupções, na arquitetura DTSD as instruções não sinalizam exceções até que haja uma tentativa de transferir seus resultados para um registrador da ISA ou para a memória. Um bit de exceção é adicionado a cada

registrador de renomeação para viabilizar a sinalização de exceções somente nestes casos.

Instruções executadas especulativamente que observam condições de exceção ligam o bit de exceção dos seus registradores destino. Outras instruções que vierem a ler destes registradores propagam o bit de exceção. Uma exceção é sinalizada quando um bit de exceção ligado é propagado para um registrador da ISA ou para uma operação de escrita na memória.

Assim como na DTSVLIW, na DTSD o mecanismo de tratamento de exceções conhecido como *Checkpoint* [12] é utilizado. Um ponto de restauração de exceções é criado ao iniciar a execução de cada bloco por meio da cópia de todos os registradores da ISA para registradores específicos. Durante a execução em modo *dataflow*, instruções de escrita na memória fazem com que o conteúdo da memória que elas sobrescrevem sejam salvos em uma lista de restauração de memória. Esta lista contém o endereço, o conteúdo e o tipo de dado sobrescrito. Caso a máquina DTSD detecte uma exceção durante a execução de um bloco, a unidade de escalonamento entra em modo de restauração. Neste modo, os registradores da arquitetura recebem os valores salvos no início da execução do bloco e cada entrada da lista de restauração de memória é escrita de volta na memória cache de dados. Caso a exceção tenha sido provocada por uma ambigüidade no acesso à memória, o bloco em questão é invalidado na cache de blocos. Em seguida, a execução volta ao normal.

Em caso de exceção por ambigüidade de memória, a execução continua em modo seqüencial e o novo bloco é escalonado, agora de forma a prevenir novas exceções deste tipo: a dependência de dados provoca naturalmente a inclusão de *tokens* que evitarão a exceção observada no escalonamento anterior. Para outros tipos de exceções, a execução continua em *modo de exceção* até que a exceção se repita e, a partir deste ponto, o sistema operacional trata a exceção. No modo de exceção, apenas o processador primário executa.

Algumas instruções especiais, como as que invocam o sistema operacional, nunca são escalonadas e são sempre executadas pelo Processador Primário.

3. Metodologia

Um simulador da arquitetura DTSD que executa código Alpha [10] foi implementado em C. O simulador é paramétrico e modela as características da DTSD apresentadas.

Nós comparamos os desempenhos (paralelismo no nível de instrução) obtidos com diversas configurações de nosso simulador DTSD com o do processador Alpha 21264 [10]. Escolhemos este processador como referência porque ele possui arquitetura Super Escalar [1], usada atualmente pela maioria dos processadores de alto desempenho de uso geral, e por existir um simulador do mesmo já validado com uma máquina real [13] e disponível publicamente. Muito embora o Alpha 21264 seja um processador relativamente antigo (data do fim da década de 1990 [10]), sua micro arquitetura ainda é bastante atual e equivalente, em termos de poder computacional, à micro arquitetura de *cores* de processadores atuais, como a de *cores* de processadores Intel modernos [14]. Evidência disso é o fato de que parâmetros determinantes do paralelismo no nível de instrução de arquiteturas Super Escalares como a Alpha 21264 serem equivalentes ao de *cores* de processadores Intel modernos, como por exemplo:

- O numero de instruções que podem ser enviadas para a execução simultaneamente – o Alpha 21264 pode enviar 6 instruções para execução simultaneamente por ciclo [6], que é a mesma quantidade de μops (uma μop Intel é equivalente uma instrução Alpha) que um *core* de processador Intel com micro arquitetura Intel® Core™ Microarchitecture (*core* Intel, para simplificar) pode enviar [14].
- O número de ALUs inteiras – o Alpha 21264 possui 4, enquanto que um *core* Intel moderno possui 3.
- O número de instruções que podem ter sua execução completada por ciclo (*retirement bandwidth*) – o Alpha 21264 pode completar 11 instruções, enquanto que um *core* Intel pode completar apenas 4 μops.

Outros parâmetros micro arquiteturais relevantes do Alpha 21264 não listados aqui também são equivalentes ou superiores ao de *cores* Intel modernos (parâmetros dos preditores de desvio e dos caches de nível 1, por exemplo).

Nós ajustamos os parâmetros de nosso simulador de modo a tornar seu núcleo de execução similar ao do processador Alpha 21264, isto é: a soma da capacidade da cache de instruções mais a da cache *dataflow* DTSD é igual à da cache de instruções do Alpha 21264; a cache de dados de ambas as arquiteturas têm o mesmo tamanho, associatividade e latência; as unidades funcionais são em quantidade equivalente e possuem latências iguais; etc. [11].

Foi possível utilizar um núcleo de execução comum aos simuladores DTSD e Alpha 21264 porque ambos são baseados no *Simplescalar* [9]. Assim como o simulador Alpha, o simulador DTSD é do tipo *execution-driven* (i.e., ele simula fielmente a arquitetura DTSD descrita).

A arquitetura Super Escalar do Alpha 21264 não é descrita detalhadamente aqui. Descrições detalhadas da arquitetura Super Escalar e do processador Alpha 21264 podem ser encontradas na literatura [1, 10].

Os programas de teste escolhidos para realizar os experimentos com as duas arquiteturas são programas típicos de máquinas de uso geral:

- binária: algoritmo de busca binária [15];
- bolha: algoritmo de ordenação, método da bolha [16];
- quick: algoritmo de ordenação [16];
- lu: algoritmo para solução de equações lineares baseado na Eliminação Gaussiana [17];
- livermor: algoritmo para determinação do menor componente de um vetor de dimensão n [18];

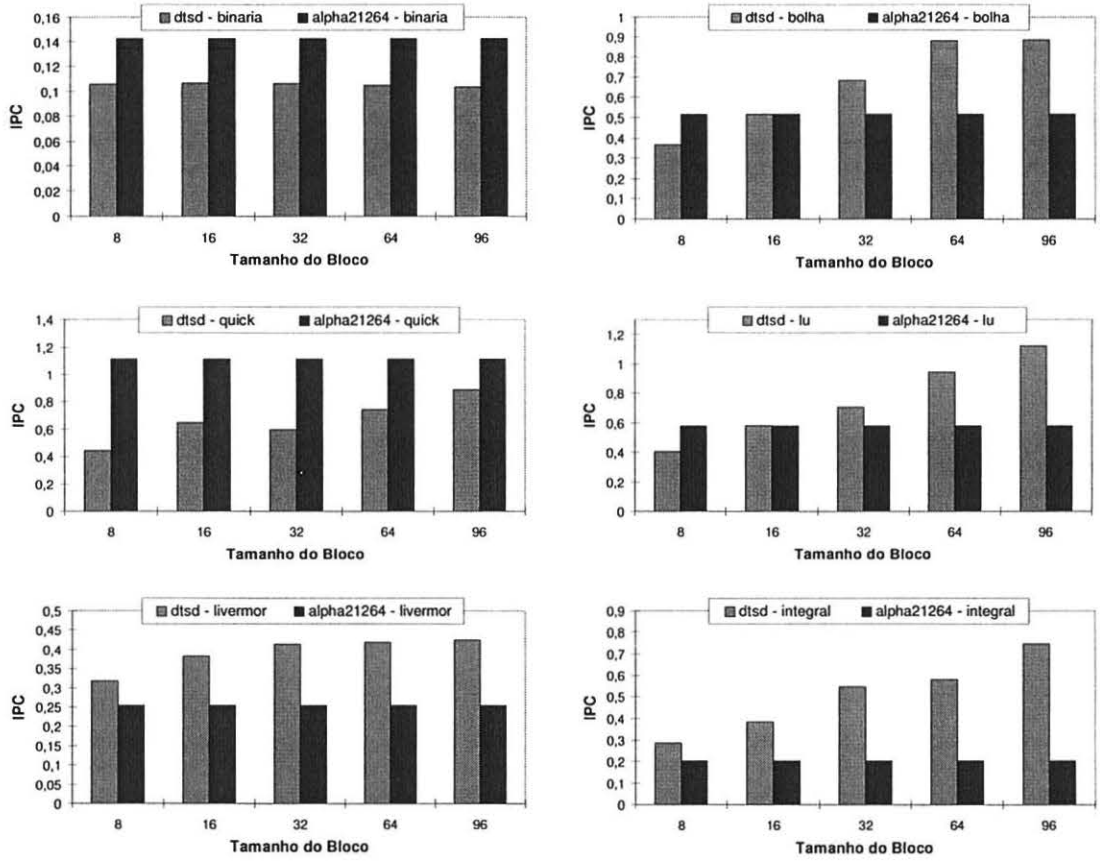

Figura 7. Desempenho das arquiteturas em termos de Instruções por Ciclo – IPC

- integral: algoritmo de integração numérica segundo o método do trapézio [19].

A Tabela 1 apresenta o número de instruções necessárias para a completa execução de cada um dos programas de teste empregados.

Tabela 1. Programas de Teste: instruções executadas

Programa de teste	Número de Instruções Executadas
binária	1780
bolha	51894
quick	53101
lu	141205
livermor	11128
integral	322168

É importante mencionar que estes programas de teste são simples e não correspondem a programas típicos executados cotidianamente em máquinas atuais. Em trabalhos futuros examinaremos o desempenho DTSD com programas mais complexos.

Neste trabalho foram estudados processadores DTSD com apenas um Bloco de Unidades Funcionais (BUF). Deixamos o estudo sobre processadores DTSD com mais de uma BUF para trabalhos futuros devido à complexidade adicional que seria imposta ao simulador DTSD.

4. Experimentos

Nós utilizamos o número médio de instruções executadas por ciclo (*instructions per cycle* – IPC) como medida de desempenho. Foram examinados processadores DTSD configurados com diferentes tamanhos de bloco de instruções *dataflow* e o desempenho de cada um deles foi comparado com o do processador Alpha 21264.

A Figura 7 apresenta o desempenho do processador Alpha 21264 e de cada uma das configurações de processador DTSD avaliadas para cada um dos programas de teste utilizados. Os programas de teste foram executados pelo simulador DTSD utilizando blocos de instruções *dataflow* com 8, 16, 32, 64 e 96 instruções. Note que, nos gráficos da Figura 7, o desempenho do processador Alpha 21264 é representado por barras de tamanho constante, dado que, para este processador, não

há variação de tamanho de bloco de instruções por não possuir esta característica arquitetural. O equivalente Super Escalar do bloco de instruções *dataflow* é a janela de instruções [1]. O tamanho da janela de instruções do Alpha 21264 é igual a 80 instruções [10].

Para o programa de teste *binaria*, a arquitetura DTSD obteve desempenho menor do que a Super Escalar para todos os tamanhos de bloco (Figura 7). Também é possível observar neste caso que a variação do tamanho do bloco não impactou no desempenho DTSD. Isso se deve ao fato deste programa requerer a execução de poucas instruções, o que impossibilita o escalonamento de um número significativo de blocos *dataflow*.

No caso do programa *bolha*, observamos que o desempenho da arquitetura DTSD aumenta conforme aumentamos o tamanho do bloco. Para blocos com 32, 64 e 96 instruções, o desempenho de processadores DTSD supera o desempenho do Alpha 21264.

Com o programa de teste *quick*, o processador Alpha 21264 obteve maior desempenho que os DTSD em todos os casos, mas a evolução do desempenho DTSD para os diversos tamanhos de bloco examinados mostra que, com blocos maiores, processadores DTSD podem superar o Alpha neste programa de teste.

Nos experimentos com o programa *lu*, o desempenho de processadores DTSD foi superior ao do Alpha com todos os tamanhos de bloco avaliados exceto blocos de 8 instruções.

Por fim, os resultados dos experimentos com os programas *livermor* e *integral* mostram que o desempenho da arquitetura DTSD foi superior ao da Super Escalar para todos os tamanhos de bloco examinados.

A Tabela 2 apresenta um sumário dos resultados experimentais obtidos com o processador Alpha 21264 e com as cinco configurações de processador DTSD. São mostradas a média aritmética e a média harmônica do número de instruções por ciclo (IPC) obtido na execução dos programas de teste com cada um dos processadores.

Tabela 2. Desempenho médio Alpha 21264 e DTSD

IPC	alpha21264	dtsd-8	dtsd-16	dtsd-32	dtsd-64	dtsd-96
média aritmética	*0,4682*	0,3201	0,4364	**0,5080**	**0,6113**	**0,6952**
média harmônica	*0,2934*	0,2547	**0,3029**	**0,3290**	**0,3491**	**0,3642**

Como a Tabela 2 mostra, processadores DTSD configurados com 32 ou mais instruções por bloco superam o processador Alpha 21264 em termos de IPC médio. Se a média harmônica for empregada, 16 ou mais instruções por bloco são suficientes para se obter desempenho DTSD médio superior ao do Alpha 21264.

Os processadores DTSD configurados com menos de 96 instruções por bloco possuem complexidade arquitetural similar ou inferior ao Alpha 21264, já que possuem hardware equivalente em termos de janela de instruções, hierarquia de memória, etc., além de não possuírem hardware de predição de desvio. A Figura 8 apresenta o ganho de desempenho percentual (*speedup* percentual) oferecido pela arquitetura DTSD com 96 instruções por bloco sobre o desempenho do processador Alpha 21264. Como a Figura 8 mostra, o processador DTSD oferece ganho de desempenho médio de 183% sobre o Alpha 21264, sendo que seu desempenho é inferior apenas no caso dos programas *binaria* e *quick* pelas razões já mencionadas.

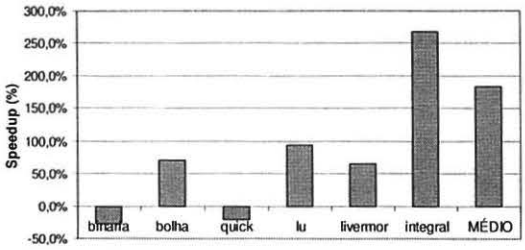

Figura 8. *Speedup* percentual oferecido pela arquitetura DTSD

5. Trabalhos Correlatos

Assim como na DTSD, em arquiteturas *Dynamic Instruction Formatting* [20] e DTSVLIW [8] os programas são (i) inicialmente executados em um modo seqüencial, (ii) escalonados, e (iii) posteriormente executados em um modo paralelo. Contudo, nestas arquiteturas, o modo de execução paralela é viabilizado por máquinas VLIW. No escalonamento de instruções com latência superior a um ciclo, a unidade de escalonamento destas arquiteturas reserva tantas instruções VLIW quantos forem os ciclos necessários para a execução das instruções (sua latência). Isso é necessário para garantir que instruções subseqüentes que tenham dependência direta de dados com a instrução multiciclo não sejam escalonadas violando estas dependências. As instruções VLIW reservadas podem não ser completamente preenchidas por instruções independentes inseridas posteriormente, o que resulta em baixo aproveitamento do paralelismo disponível. Máquinas DTSD não sofrem deste problema, pois o seu modo de execução paralela é viabilizado por uma máquina *dataflow*.

Em máquinas com ISA EDGE as instruções podem ser executadas tão logo seus operandos de entrada estejam prontos [4], mas estes operandos não são especificados explicitamente. Na DTSD os operandos são explicitados e *tokens* são usados para ativar a execução de instruções. Isso evita que os operandos tenham que ser passados diretamente entre instruções, diminuindo o número de barramentos internos da máquina, o que tem impacto positivo no consumo de energia e simplifica a implementação.

Na arquitetura EDGE o escalonamento é estático e feito pelo compilador [4]. Experimentos com a DTSVLIW mostraram que o escalonamento dinâmico DTSVLIW supera o escalonamento estático EPIC [21], e acreditamos que o escalonamento dinâmico DTSD também deve superar o escalonamento estático EDGE. O escalonamento estático também trás restrições à

compatibilidade de código para trás, problema que a arquitetura DTSD não possui.

6. Discussão

A arquitetura implementada e testada ainda não possui todas as características propostas. Uma característica importante é a possibilidade de mais de um Bloco de Unidades Funcionais (BUF). A implementação de mais de um BUF leva a questões não avaliadas neste trabalho, como a viabilidade da implementação de BUFs com um grande número de unidades funcionais, e o custo de implementação em hardware do sistema de envio de *tokens* entre BUFs e seu impacto no desempenho.

Uma característica DTSVLIW não implementada na DTSD avaliada é a possibilidade de só se fazer a busca de um bloco pronto a partir do momento em que o bloco que estiver sendo escalonado contiver um número mínimo de instruções. Isso evita que sejam salvos blocos com poucas instruções e pode aumentar significativamente o desempenho [8, 11].

Neste trabalho, gostaríamos de ter utilizado os programas de avaliação do conjunto SPEC2000 ou SPEC2006 (www.specbench.org). Contudo, a implementação corrente do simulador DTSD ainda não permite executar os programas do SPEC2000 completamente porque algumas seqüências pouco comuns de instruções ainda não são tratadas pelo nosso simulador. Vale destacar que nosso simulador executa em paralelo com uma máquina seqüencial de teste: a cada bloco *dataflow* executado, o estado da máquina DTSD é comparado ao da máquina de teste para validação.

7. Conclusões

Neste trabalho apresentamos a arquitetura *Dynamically Trace Scheduling Dataflow* (DTSD). Esta arquitetura executa programas em dois modos distintos: um modo seqüencial, baseado no fluxo de controle; e um modo paralelo *dataflow*. Nós implementamos um simulador *execution-driven* da arquitetura DTSD e avaliamos seu desempenho experimentalmente com configurações de hardware típicas. Os resultados obtidos com a execução de programas de teste na arquitetura DTSD demonstraram que esta arquitetura supera a Super Escalar na maioria dos casos examinados, apresentando *speedup* médio de 183%.

Em trabalhos futuros, investigaremos o efeito da incorporação de mais de um bloco de unidades funcionais (BUF) no desempenho DTSD, e o custo/benefício de múltiplos BUFs. Examinaremos, também, qual é o comportamento da DTSD quando um número mínimo de instruções em cada bloco *dataflow* for imposto.

8. Referências

[1] M. Johnson, "Superscalar Microprocessor Design", Prentice-Hall, 1991.

[2] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching", Proc. 29th Annual International Symposium on Microarchitecture, p. 24-34, 1996.

[3] D. W. Wall, "Limits of Instruction-Level Parallelism", Digital Western Research Laboratory – WRL Research Report 93/6, November 1993.

[4] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder, "Scaling to the End of Silicon with EDGE Architectures", IEEE Computer, Vol. 37, No. 7, p. 44-55, July 2004.

[5] A. L. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality", Proc. National Computing Conference, AFIPS Press, Reston, Va., pp. 1079-1086, 1979.

[6] A. H. Veen, "Dataflow Machine Architecture", ACM Computing Surveys, Vol. 18, No. 4, pp. 335-396, December 1986.

[7] A. F. De Souza and P. Rounce, "Dynamically Trace Scheduled VLIW Architectures", in Lecture Notes in Computer Science, Vol. 1401, p. 993-995, 1998.

[8] A. F. De Souza and P. Rounce, "Dynamically Scheduling VLIW Instructions", Journal of Parallel and Distributed Computing, Vol. 60, No. 12, p. 1480-1511, December 2000.

[9] T. Austin and D. Burger, "The SimpleScalar Tool Set", Technical Report TR-1342, Computer Science Department, University of Wisconsin – Madison, June 1997

[10] Compaq Computer Corporation, "Alpha 21264 Microprocessor Hardware Reference Manual", Compaq Computer Corporation, 1999.

[11] F. L. L Almeida, "Escalonamento Dinâmico de Caminhos de Execução em Blocos de Instruções Dataflow", Dissertação de Mestrado, Programa de Pós-Graduação em Informática – UFES, 2007.

[12] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for Out-of-order Execution Machines", Proc. 14th Annual International Symposium on Computer Architecture, pp. 18-26, 1987.

[13] R. Desikan, D. Burger, and S. W. Keckler, "Measuring Experimental Error in Microprocessor Simulation", Proc. 28th Annual International Symposium on Computer Architecture, pp. 226-277, 2001.

[14] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture", Intel Corporation, 2008.

[15] E. Horowitz and S. Sahni, "Fundamentals of Computer Algorithms", Computer Science Press Inc., 1978.

[16] D. Knuth, "Art of Computer Programming", Addison-Wesley Publishing Co., USA, 1973.

[17] G. Forsythe e C. B. Moler, "Computer Solution of Linear Algebraic Systems", Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[18] F. H. McMahon, "Fortran Kernels: MFLOPS", Lawrence Livermore National Laboratory, 1983.

[19] S. D. Conte, "Elementary Numerical Analysis", McGraw-Hill Book Co., 1965.

[20] R. Nair and E. M. Hopkins, "Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups", Proc. 24th Annual International Symposium on Computer Architecture, pp.13-25, 1997.

[21] S. C. Santana and A. F. De Souza, "A Comparative Analysis Between EPIC Static Instruction Scheduling and DTSVLIW Dynamic Instruction Scheduling", Workshop on Exploring the Trace Space for Dynamic Optimization Techniques, pp. 59-67, 2003.