Uma Arquitetura DTSVLIW com Múltiplos Contextos de Execução

Fernando Líbio L. Almeida e Alberto F. De Souza
Departamento de Informática
Universidade Federal do Espírito Santo
{flibio, alberto}@inf.ufes.br

Resumo

Este trabalho apresenta um estudo preliminar de uma arquitetura DTSVLIW com múltiplos contextos de execução implementados em hardware. A principal motivação para o desenvolvimento desta arquitetura foi a constatação do grande impacto da latência da hierarquia de memória no desempenho da arquitetura DTSVLIW. Foram abordados os principais aspectos e examinados possíveis critérios de decisão para implementação de uma primeira versão de um simulador para que, através de experimentos, fosse possível avaliar a redução do impacto da latência de memória no desempenho DTSVLIW propiciada por múltiplos contextos de hardware. Nossos resultados mostram a grande influência da organização de caches no desempenho da arquitetura, e a importância de uma análise mais detalhada de diferentes formas de implementação de máquinas DTSVLIW com múltiplos contextos de execução implementados em hardware.

1. Introdução

Em máquinas que seguem a arquitetura DIF (*Dynamic Instruction Formatting* [13]), o código produzido pelo compilador é inicialmente executado em uma máquina simples ao mesmo tempo em que é formatado dinamicamente em blocos de instruções VLIW (*Very Long Instruction Words* [10]). Estas instruções VLIW são armazenadas em uma *cache* VLIW para posterior execução em uma máquina VLIW, caso o mesmo fragmento de código tenha que ser executado novamente. Do mesmo modo que em processadores Super Escalares [11], dependências entre as instruções do programa têm que ser analisadas, mas isto só é feito quando o código é formatado e não a cada vez que o código é executado a partir da *cache* VLIW. Isto permite que se tire proveito da velocidade extra da máquina VLIW, oriunda de sua natural simplicidade [10]. A arquitetura DTSVLIW (*Dynamically Trace Scheduled VLIW*) [7] alcança desempenho semelhante ou superior à DIF, mas com implementação possivelmente mais simples [5].

A Figura 1 mostra um diagrama de blocos da arquitetura DTSVLIW. Em um processador DTSVLIW, a Máquina Escalonadora traz instruções da Cache de Instruções e as executa pela primeira vez usando um processador *pipelined* simples – o Processador Primário. Além disso, sua Unidade de Escalonamento escalona dinamicamente a seqüência de instruções produzida durante a execução no Processador Primário dentro de instruções VLIW, agrupa estas instruções VLIW em blocos e salva estes blocos na Cache VLIW. Se o mesmo código for executado novamente, ele é trazido da Cache VLIW pela Máquina VLIW e executado em modo VLIW.

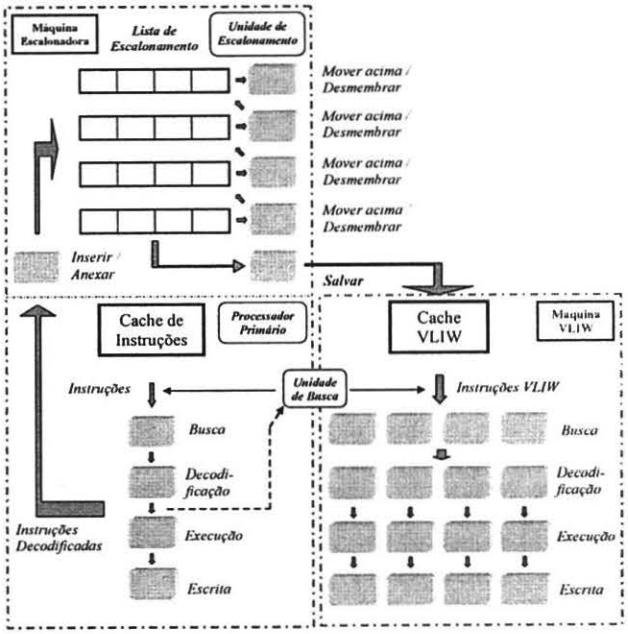

Figura 1 – A Arquitetura DTSVLIW

A latência das instruções tem um forte impacto na quantidade de paralelismo no nível de instrução (*Intruction-Level Parallelism* – ILP) que pode ser explorado por máquinas DTSVLIW. No escalonamento de instruções que requerem mais de um ciclo para execução, a Unidade de Escalonamento de uma DTSVLIW toma, quando da inserção destas instruções na Lista de Escalonamento, tantas instruções VLIW quantos forem os ciclos necessários para a execução da instrução (sua latência). Isso é necessário para garantir que instruções subseqüentes, que tenham dependência direta de dados com a instrução multiciclo, não sejam colocadas na Lista de Escalonamento em um ponto que venha a violar estas dependências. As instruções VLIW tomadas podem não ser completamente preenchidas por instruções inseridas posteriormente, o que resulta no não aproveitamento do paralelismo disponível.

Em trabalho anterior examinamos o efeito da latência das instruções e da hierarquia de memória no desempenho da arquitetura DTSVLIW [1]. Nossos resultados mostram que a latência da hierarquia de memória tem efeito significativo no desempenho da DTSVLIW para programas inteiros, uma redução de 22,1%, mas muito maior no desempenho de programas de ponto flutuante, redução de 85,5% [1].

Estes resultados foram a principal motivação para o estudo de uma arquitetura DTSVLIW com múltiplos contextos implementados em hardware. Uma máquina com múltiplos contextos em hardware, ou *multithreaded* [15, 16], possui duas ou mais réplicas das estruturas internas (registradores, basicamente) responsáveis por armazenar o estado da máquina (contexto).

Neste trabalho fizemos uma análise preliminar dos principais aspectos referentes a essa arquitetura, e explicamos nossos critérios de decisão para a implementação de uma primeira versão. Realizamos experimentos com a arquitetura DTSVLIW com múltiplos contextos de hardware com programas do SPEC2000. Os resultados com os programas inteiros mostraram como o desempenho da arquitetura pode ser degradado devido a faltas de conflito na *cache* – diminuição do desempenho em até 37,5%. Ainda assim, a arquitetura DTSVLIW com múltiplos contextos de hardware chegou a obter desempenho até 20,5% superior à DTSVLIW com um único contexto no caso dos programas de ponto flutuante executados. Finalmente, nossos resultados, obtidos a partir dos critérios escolhidos, demonstram que este tema merece uma análise mais detalhada, principalmente quanto à organização das *caches* e os critérios de troca de contexto.

2. A Arquitetura DTSVLIW

A arquitetura DTSVLIW possui dois modos de execução: escalar e VLIW. Sempre que um trecho de código é encontrado pela primeira vez, ele é executado no modo escalar pelo processador primário, um processador *pipelined* simples capaz de executar no máximo uma instrução por ciclo (Figura 1). No modo escalar, o código é trazido da Cache de Instruções pelo processador primário e, quando suas instruções são enviadas para o estágio de execução, elas são também enviadas para a Unidade de Escalonamento, que as escalona, dentro da Lista de Escalonamento, em blocos de instruções VLIW (Figura 1). Estes blocos são salvos na Cache VLIW, sendo que o endereço de cada bloco é igual ao da primeira instrução do trecho de código do qual o bloco se originou.

Se um mesmo trecho de código precisa ser executado novamente, ele pode estar presente na Cache VLIW. Isto é detectado pela Unidade de Busca que, neste caso, traz instruções VLIW da *cache* e as envia para execução na Máquina Escalonadora, que as executa em modo VLIW (Figura 1). No diagrama de blocos da Figura 1, instruções com latência superior a um ciclo permanecem mais de um ciclo no estágio de execução VLIW, que pode ou não ser totalmente *pipelined*, dependendo da instrução.

A arquitetura DTSVLIW é capaz de executar código seqüencial comum em modo VLIW, sendo o grau de paralelismo de sua máquina escalonadora dependente apenas da tecnologia usada na sua implementação.

2.1. O Escalonamento de Instruções

O algoritmo de escalonamento implementado pela máquina escalonadora é uma versão simplificada do algoritmo *First Come First Served* (FCFS) usado no escalonamento de microinstruções [4]. Ele foi implementado através de cinco operações simples e de fácil implementação em hardware: *Inserir*, *Anexar*, *Mover acima*, *Desmembrar* e *Instalar*. Estas operações são realizadas sobre a Lista de Escalonamento, uma lista circular implementada em hardware capaz de armazenar várias instruções VLIW.

A operação *Inserir* é feita sempre que uma instrução passa do estágio de Decodificação para o estágio de Execução do processador primário (Figura 1) e consiste na inserção desta instrução na última instrução VLIW da lista de escalonamento. Se não há espaço para a instrução a ser inserida ou existe uma dependência de dados verdadeira ou de saída entre ela e qualquer instrução na instrução VLIW alvo da operação *Inserir*, a instrução não pode ser inserida. Neste caso, uma operação *Anexar* é utilizada. Quando uma operação *Anexar* precisa ser realizada e a lista de escalonamento está cheia, as instruções VLIW são copiadas, uma a uma, para um buffer de escrita na Cache VLIW (Figura 1) e em seguida salvas na Cache VLIW. A escrita na *cache* é feita através de um *pipeline* simples de um estágio e não interfere com as operações de escalonamento, uma vez que no máximo uma instrução é inserida na Lista de Escalonamento por ciclo [6].

Uma instrução inserida em um ciclo pode, em ciclos subseqüentes, ser movida para cima na Lista de Escalonamento através da operação *Mover acima* (Figura 2). Várias operações *Mover acima* podem ocorrer em paralelo em um único ciclo de máquina de forma paralela, limitadas a uma instrução por instrução VLIW. Esta instrução que pode ser movida acima é dita uma instrução candidata ao escalonamento (existe apenas uma instrução candidata por instrução VLIW).

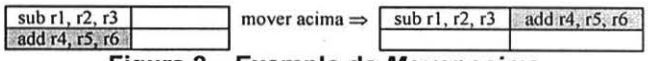

Figura 2 – Exemplo de *Mover acima*

Se uma instrução não pode ser *movida acima* ela é *instalada*, o que faz com que ela deixe de ser uma instrução candidata, ficando inativa na Lista de Escalonamento no ponto onde sofrer a operação *Instalar*. A Figura 3 mostra um exemplo da operação *Instalar*.

Figura 3 – Exemplo de *Instalar*

Em máquinas DTSVLIW, a *renomeação de registradores* torna possível o escalonamento mesmo na presença de dependência entre as instruções, exceto, obviamente, dependência de dados verdadeira. O algoritmo de escalonamento usa a operação *Desmembrar* (Figura 4) para os casos de dependência de controle, de saída e antidependência. Esta operação divide a instrução em duas partes: uma é a instrução original com a saída renomeada; a outra é uma instrução de cópia, que copia o valor do registrador usado na *renomeação* para o registrador original.

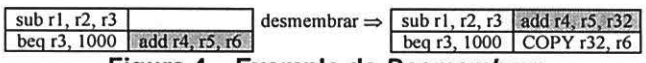

Figura 4 – Exemplo de *Desmembrar*

2.2. Execução Especulativa

Desvios condicionais e indiretos não podem ser alvo de operações *Mover Acima* ou *Desmembrar*. Eles são instalados quando inseridos na lista de escalonamento e estabelecem um rótulo para a instrução VLIW. Todas as instruções subseqüentes que forem instaladas nesta instrução VLIW recebem este rótulo. Durante a execução VLIW, a Máquina VLIW examina os desvios e, se eles seguirem o mesmo caminho seguido durante o escalonamento, valida seus rótulos. Somente instruções com rótulos válidos escrevem seus resultados no estado da máquina. No exemplo de desmembramento acima, a instrução de cópia somente escreve em **r6** se o desvio condicional **beq** for validado. Instruções movidas para instruções VLIW que possuem desvios condicionais ou para instruções VLIW acima de instruções VLIW com desvios condicionais são, então, executadas especulativamente.

2.3. Exceções

Instruções de leitura e escrita na memória também podem ser movidas acima ou desmembradas, o que pode causar exceções e *memory aliasing* (inversão ou simultaneidade semanticamente incorreta de acessos à memória) [10]. A DTSVLIW só trata exceções causadas por instruções que efetivamente escrevem em registradores da arquitetura do conjunto de instruções (*Instruction-Set Architecture* – ISA) emulada e é capaz de detectar ambigüidades, tratando-as como exceções.

Instruções renomeadas que geram exceções ligam um bit específico no registrador de destino, que é propagado por instruções que venham a operar sobre este registrador. Se uma instrução opera sobre um registrador com o bit de exceção ligado e escreve em um registrador da ISA, ela gera uma exceção. Exceções ocorridas em modo VLIW abortam a execução do bloco sendo executado e o estado da ISA volta àquele anterior à execução do bloco. Para permitir isso, máquinas DTSVLIW salvam todo o estado da ISA emulada (os registradores, e eventuais registradores adicionais modificáveis no modo VLIW) no início da execução de cada bloco. A. F. De Souza e P. Rounce apresentam mais detalhes sobre o modo como a DTSVLIW trata de exceções em outro trabalho [7].

2.4. Latência Máxima

Na implementação de uma ISA pode ser necessário ou conveniente que algumas instruções tenham latências muito altas. A instrução de divisão de ponto flutuante com dupla precisão necessita de até 15 ciclos no processador Alpha 21264 [3, 9], por exemplo.

Escalonar instruções com latência muito alta pode deixar muitas instruções VLIW vazias ou apenas parcialmente preenchidas por falta de oportunidade para o seu escalonamento. Uma forma de limitar este problema é impor uma *latência máxima* de instrução para efeito de escalonamento. Assim, instruções que necessitem de mais ciclos para serem executadas seriam escalonadas como se tivessem *latência máxima*, mas, no modo de execução VLIW, forçariam a parada do *pipeline* de execução por um número de ciclos igual à diferença entre a *latência máxima* e a latência da instrução.

O nível de paralelismo que pode ser alcançado durante o escalonamento de blocos VLIW está diretamente relacionado com a latência das instruções. Instruções com dependências só podem ser executadas após a resolução destas. Por esta razão, se as instruções possuem latências altas, dependências detectadas com relação a elas poderão forçar o escalonamento de instruções dependentes mais abaixo na lista de escalonamento, o que poderá deixar espaços na lista de escalonamento que eventualmente não serão preenchidos. Estes espaços significam um menor número de instruções executadas em paralelo durante o modo VLIW e, portanto, uma diminuição de desempenho.

3. Uma Arquitetura DTSVLIW com Múltiplos Contextos

Os processadores modernos utilizam diversas técnicas para aumentar o paralelismo e a utilização do processador. Uma técnica freqüentemente utilizada para ocultar a latência de memória é a implementação de múltiplos contextos em hardware [15, 16]. O objetivo desta técnica é aproveitar o tempo de espera por vezes necessário durante acessos à hierarquia de memória executando um outro programa em um outro contexto de hardware, aumentando assim o tempo total de utilização do processador.

A partir dos resultados de estudo do efeito da latência das instruções e da hierarquia de memória [1] pôde-se constatar o grande custo imposto pela latência do acesso à memória em máquinas DTSVLIW. Esse custo está associado principalmente às *faltas* na *cache* de dados durante a execução em modo VLIW, que implicam na paralisação do pipeline VLIW até que as *faltas* sejam resolvidas. A proposta que fazemos de arquitetura DTSVLIW com múltiplos contextos de hardware visa o aproveitamento deste tempo ocioso para a execução de um outro programa. As alterações necessárias para que a arquitetura DTSVLIW possa executar mais de um programa concorrentemente, mas não simultaneamente, são discutidas nas subseções a seguir.

3.1. Múltiplos Contextos

A execução em hardware de mais de um programa (ou *thread*, como denominado na literatura [14, 15, 16]) concorrentemente requer que a arquitetura DTSVLIW possua mais de um contexto. Isso é indispensável porque cada *thread* deve ser executada isoladamente, de modo que tudo se passe como se cada uma estivesse sendo executada individualmente. Para tornar isso possível, as estruturas internas do processador que retém informações sobre o estado de execução de cada *thread* devem ser replicadas, e utilizadas de acordo com a *thread* em execução.

As principais estruturas replicadas para a versão proposta da arquitetura DTSVLIW com múltiplos contextos foram:

- Bancos de Registradores;
- Registradores de Renomeação;
- Estágios de Pipeline do Processador Primário;
- Estágios de Pipeline da Máquina VLIW;
- Filas de Load/Store;

Os bancos de registradores foram replicados, pois eles guardam os valores de operandos e resultados das instruções do programa em execução, e tais valores não devem ser misturados entre as *threads*. Também os registradores de renomeação foram replicados, para garantir a execução correta em modo VLIW. Replicamos os estágios de *pipeline* do Processador Primário, já que estes guardam as instruções que estão sendo executadas em modo escalar por cada *thread*. Da mesma maneira, replicamos também os estágios de *pipeline* da Máquina VLIW, para permitir as trocas de contexto durante a execução em modo VLIW. Replicamos, ainda, as filas de *load/store*, porque estas guardam os dados e endereços utilizados no acesso à memória durante a execução VLIW e são utilizadas para determinar se houve uma exceção ou ambigüidade.

3.2. Caches

Uma questão importante para o funcionamento e o desempenho de uma arquitetura com múltiplos contextos é a forma de acesso às memórias *cache*. Já que várias *threads* estarão sendo executadas, deve haver algum método para que, ao acessar a memória, uma *thread* não acabe lendo ou escrevendo em um bloco, dados de uma outra *thread*. Para evitar este tipo de ocorrência, duas técnicas podem ser empregadas: a utilização de *caches* privadas por thread ou de *caches* compartilhadas com rótulo adicional de *thread*.

A grande vantagem da utilização de *caches* privadas é que uma *thread* não teria seus blocos substituídos pelos de outra *thread* quando houvesse uma troca de contexto. Isso implicaria em menos *faltas* na *cache* e, conseqüentemente, menos acessos à memória, aumentando o desempenho individual de cada *thread*. Porém, já que uma *thread* não teria acesso à *cache* de outra, cada *thread* possuiria somente uma fração do que poderia ser uma *cache* maior compartilhada. Assim, devido à limitações físicas, quanto maior o número de contextos, menor o tamanho da *cache* por *thread*, o que implicaria em um maior número de *faltas*, diminuindo o desempenho de cada *thread*.

Caches compartilhadas, por sua vez, podem armazenar blocos de todas as *threads* e, com isso, cada *thread* é executada com a possibilidade de utilizar toda a *cache* compartilhada. Mesmo concorrendo pela utilização de uma mesma *cache*, uma vantagem para as *threads* seria poder utilizar espaços na *cache* que, com *caches* privadas, não seria possível, já que ela não teria acesso às demais áreas de *cache*. Contudo, já que uma *thread* poderia substituir blocos de outras na *cache*, blocos que poderiam ser reutilizados posteriormente podem ser substituídos, provocando *faltas* na *cache* quando uma outra thread for executada novamente.

As duas técnicas possuem vantagens e desvantagens e a forma como afetam o desempenho do processador pode depender não só do número de contextos que o hardware possui, mas também do padrão de acesso aos dados dos programas sendo executados [15, 16]. Um estudo detalhado destas duas técnicas não foi possível dentro do escopo deste trabalho - nosso critério de escolha foi simplesmente baseado nas características mais favoráveis de implementação em hardware. *Caches* compartilhadas diferem de *caches* comuns apenas por possuir alguns *bits* adicionais de identificação da *thread* presentes nos seus campos rótulo. Por outro lado, *caches* privadas implicam em mais conexões e lógica de controle, já que cada *cache* precisaria de portas de acesso internas e externas. Dessa forma, a nossa decisão foi implementar todas as *caches* compartilhadas.

Nas figuras 5 e 6 estão representadas, em diagramas simplificados, as *caches* de instruções e de dados da hierarquia de memória da arquitetura DTSVLIW com múltiplos contextos de hardware que implementamos. Ela possui estrutura e funcionamento equivalentes ao da hierarquia de memória do processador Alpha 21264, com dois níveis de *cache*: *cache* de instruções e de dados de nível 1 separadas, e *cache* compartilhada de nível 2. A *cache* de instruções é indexada virtualmente e possui rótulos virtuais [14]. A *cache* de dados também é indexada virtualmente, mas possui rótulos físicos [14]. O conjunto de instruções da arquitetura Alpha pode utilizar endereços virtuais de 48 ou 43 bits e endereços físicos de 44 ou 41 bits [14]. Nos simuladores, utilizamos endereços virtuais de 43 bits e, como experimentamos com um máximo de 8 *threads*, bastaram 3 bits para identificar cada *thread* unicamente, o que resulta num endereço de acesso à *cache* de 46 bits.

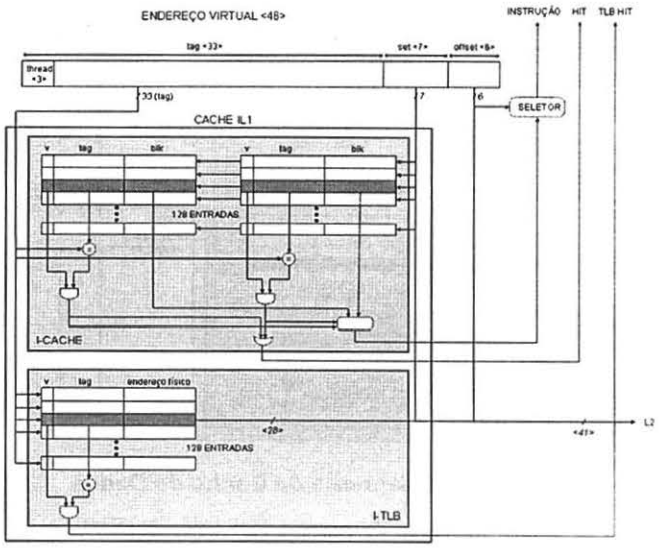

Figura 5 – Diagrama da Cache de Instruções

A *cache* de instruções do nível 1 (IL1) da DTSVLIW em estudo possui 16KB (16 x 1024 bytes) de espaço de armazenamento de instruções escalares (para uso do Processador Primário apenas) – em trabalho anterior nós justificamos a escolha deste tamanho [1]. Trata-se de uma *cache* associativa por conjunto [14] com dois blocos por conjunto. Cada bloco pode armazenar 16 instruções de 32 bits. Assim, temos 128 conjuntos com dois blocos cada (16K / (16 * 4 * 2)).

Um acesso à IL1, começa com o uso dos 7 *bits* que indicam o conjunto a ser selecionado de IL1 (Figura 5, campo *set*), dentre os 128 existentes. Em seguida, os rótulos de cada um dos dois blocos do conjunto selecionado são comparados para determinar se houve um acerto. No caso de um acerto, a instrução será buscada dentro do bloco correspondente através dos 6 *bits* de deslocamento. Na verdade, os dois *bits* de mais baixa

ordem são sempre iguais a zero, já que as instruções têm sempre 32 *bits*. No caso de uma falta na I-Cache, o I-Tlb (*Translation Look-aside Buffer* – buffer de tradução) é consultado para verificar se já não existe uma tradução do endereço virtual desejado para o endereço físico correspondente. As 128 entradas do I-Tlb são examinadas simultaneamente (a I-Tlb é totalmente associativa [14, 1]) e, se houver uma tradução pronta para o endereço virtual em questão, ela é utilizada para acessar a *cache* de nível 2 (L2). Caso não haja uma tradução pronta, o endereço virtual precisa ser traduzido, o que é feito em nosso simulador na unidade de gerenciamento de memória (*Memory Management Unit* – MMU), não ilustrada aqui.

O endereço físico gerado através da tradução feita pela MMU a partir do endereço virtual não necessita conter *bits* para identificar a *thread*, como no endereço virtual. Dessa forma, nenhuma modificação é necessária a partir deste ponto (L2), já que quem cuida do gerenciamento dos blocos e da área de memória correspondente é a MMU, que aloca o endereço de cada página virtual na tabela de páginas correspondente a cada *thread*.

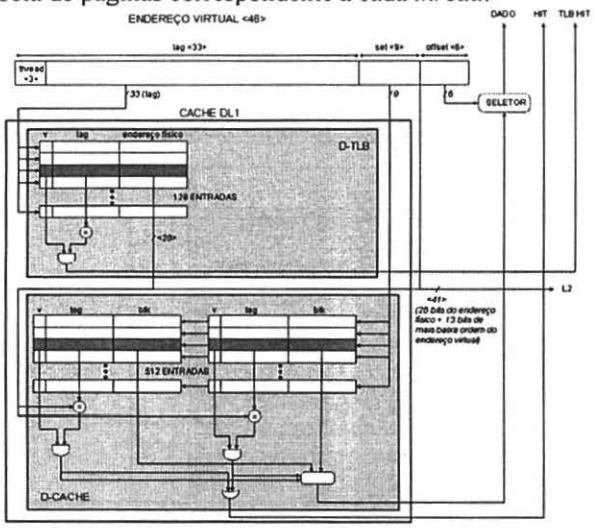

Figura 6 – Diagrama da Cache de Dados

A *cache* de dados do nível 1 (DL1) da DTSVLIW em estudo possui 64KB de espaço de armazenamento de dados (para uso do Processador Primário e da Máquina VLIW) – em trabalho anterior nós justificamos a escolha deste tamanho [1]. Trata-se de uma *cache* associativa por conjunto com dois blocos por conjunto. Cada bloco pode armazenar 64 bytes. Assim, temos 512 conjuntos com dois blocos cada (64K / (64* 2)). O Tlb de dados, ou D-Tlb, é idêntico ao I-Tlb.

Os acessos à DL1 (Figura 6) ocorrem de maneira semelhante aos acessos à IL1. Porém, como a *cache* de dados possui *rótulos* virtuais, mas é indexada fisicamente, a tradução do endereço virtual para o endereço físico é feita no D-Tlb simultaneamente ao acesso à DL1, e o endereço físico é utilizado para o acesso à DL1. Havendo uma *falta* em DL1, o endereço físico é usado para acessar a *cache* L2 e, no caso de uma *falta* também na *cache* L2, um acesso a memória principal é realizado. Mais uma vez não houve necessidade de alteração nos níveis de *cache* subseqüentes ao nível 1.

A Cache VLIW, que só possui um nível, também teve adicionado a seu campo rótulo os três *bits* que identificam a *thread* unicamente, de maneira análoga aos caches de instruções e de dados.

3.3. Critérios de Mudança de Contexto

Num processador *multithreaded* são executadas mais de uma *thread* por vez, mas não simultaneamente. Somente uma *thread* está em execução em um dado momento e esta pode utilizar todas as unidades funcionais do processador. Em determinados momentos, a *thread* em execução deve ser trocada por outra para que todas possam executar. Essa situação torna importante a decisão de quando dar oportunidade de execução a cada *thread* para tentar manter o processador sendo utilizado na maior parte do tempo possível. Vários critérios podem ser empregados para a alternância de contextos; por exemplo, trocar de *thread* após um determinado número de ciclos, ou quando uma *thread* tem de esperar por um acesso a memória, entre outros.

O critério adotado por nós foi o de trocar de contexto quando ocorrer uma *falta* na *cache*. O objetivo é aproveitar o tempo ocioso de espera pelo trabalho da hierarquia de memória executando uma próxima *thread*. Porém, uma restrição inerente à nossa proposição de arquitetura DTSVLIW com múltiplos contextos impede que a mudança de contexto ocorra a qualquer *falta*.

Em nossa versão, a máquina escalonadora é tratada como uma unidade funcional, e a lista de escalonamento é parte dela. Tomamos essa decisão porque manter uma lista por *thread* tornaria a implementação em hardware complexa, apesar de não impossível, devido ao grande volume de dados que tem de ser guardados por *thread* (todas as instruções guardadas na lista de escalonamento). Com isso, somente uma *thread* por vez pode utilizar a máquina escalonadora e, assim, uma *falta* de instruções ou de dados durante a execução no processador primário não causa uma troca de contexto, já que isto provocaria o escalonamento, num mesmo bloco, de instruções de *threads* diferentes.

Desse modo, o único momento em que ocorre troca de contexto é quando ocorre uma *falta* na *cache* de dados durante a execução VLIW e, ainda assim, somente se a lista de escalonamento estiver vazia. A necessidade de a lista de escalonamento estar vazia se dá porque, do contrário, poderiam ser escalonadas num mesmo bloco instruções de *threads* diferentes. Existem outras possibilidades para mudança de contexto que pretendemos estudar futuramente. Nós discutimos algumas delas na Seção 6.

4. Métodos

Nós comparamos entre si as versões da arquitetura DTSVLIW com um e com múltiplos contextos de execução implementados em hardware para observar o desempenho da versão com múltiplos contextos de hardware descrita na seção anterior.

4.1. Programas de Teste

Todos os simuladores utilizados recebem como entrada executáveis produzidos por compiladores comuns que geram código para a Alpha ISA [9]. Para os experimentos usamos todos os executáveis do SPEC2000 (www.specbench.org) disponíveis junto com o simulador

simplescalar [2], e para os quais os pesquisadores da *University of Minnesota* desenvolveram entradas [12]. Com este conjunto de entradas, os programas do SPEC2000 selecionados pelos pesquisadores desta universidade requerem apenas cerca de um bilhão de instruções para sua execução. Este número de instruções permite capturar o desempenho do processador quando executando estes programas. Nos experimentos realizados com a arquitetura DTSVLIW configurada com múltiplos contextos, os programas foram executados até que o primeiro deles terminasse.

4.2. Configurações Básicas dos Simuladores

As configurações utilizadas nos experimentos são como as indicadas nas tabelas numeradas de 1 a 3. As máquinas DTSVLIW utilizadas usam o mecanismo de compactação de blocos descrito em [8]. A *VLIW Cache* utilizada é bastante pequena e tem 48KB (Tabela 1). Nós discutimos mais detalhadamente as configurações da arquitetura DTSVLIW em [1].

Tabela 1 – Configuração DTSVLIW

Processador Primário	• pipeline de quatro estágios (busca, decodificação, execução e escrita) • sem hardware de predição de desvios • desvios tomados geram uma bolha de 2 ciclos no pipeline • Cache de Instruções de 16KB, 2 blocos por conjunto, latência 1
Cache VLIW	48KB, 2 blocos por conjunto, latência 1
VLIW Pipeline	3 estágios (busca, despacho, execução)
Unidades Funcionais	4 inteiras e 2 de ponto flutuante
Tamanho da Lista de Escalonamento	2 vezes o número de instruções VLIW do bloco
Registradores usados para Renomeação	Inteiros 17, Ponto Flutuante 13, Memória 6
Geometria dos Blocos VLIW	4 instruções por VLIW, 8 VLIWs por Bloco
Latência Máxima	5 ciclos

Tabela 2 – Latência das Instruções

Instrução	Latência
Inteiras	1
Multiplicação Inteira	7
Load Inteira (Com acerto na cache)	3
Soma e Multiplicação de Ponto Flutuante	4
Divisão / Raiz Quadrada de Ponto Flutuante (SP)	12/18
Divisão / Raiz Quadrada de Ponto Flutuante (DP)	15/33
Load de Ponto Flutuante (Com acerto na cache)	3
Desvio incondicional	3

Tabela 3 – Hierarquia de Memória

Cache de Instruções (apenas para Alpha)	64 KB, 2 blocos por conjunto, latência 1
Cache de Dados	64 KB, 2 blocos por conjunto, latência 3
Cache Nível 2, Unificado	1 MB, mapeado diretamente, indexado fisicamente, latência 7
Memória RAM	Ilimitada, latência de 66 ciclos com pré-carga e 54 ciclos em acessos seqüenciais

5. Experimentos

Os experimentos foram realizados no cluster de 64 processadores do Laboratório de Computação de Alto Desempenho da UFES (www.inf.ufes.br/~lcad). Sem esta máquina, o desenvolvimento deste trabalho teria sido muito dificultado devido ao longo tempo de execução dos experimentos.

5.1. Latência das Instruções e da Hierarquia de Memória

Em estudo anterior mostramos o efeito da latência das instruções e da hierarquia de memória no desempenho da arquitetura DTSVLIW [1]. As figuras 7 e 8 mostram os resultados. Nestas figuras, para cada barra, a altura do primeiro segmento indica o desempenho obtido para cada programa, a altura até o topo do segundo segmento indica qual seria o desempenho se a latência de todas as instruções fosse igual a 1, e a altura da barra inteira qual seria o desempenho com latência de instruções igual 1 e caches perfeitas (latência de 1 ciclo).

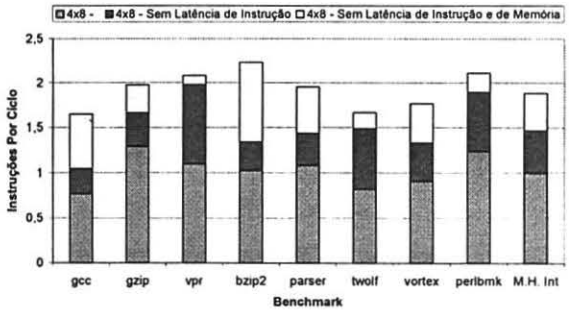

Figura 7 – Efeito da Latência – Int.

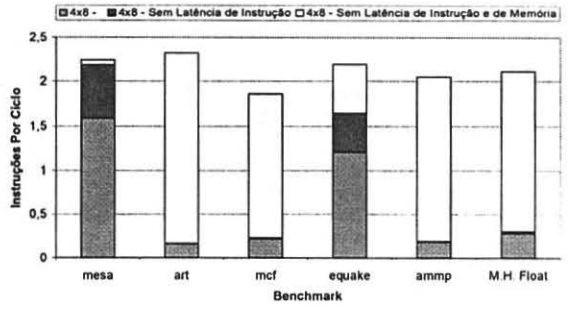

Figura 8 – Efeito da Latência – P.F.

Como as figuras mostram, o efeito da latência das instruções nos programas inteiros (basicamente *loads* e *stores*) é muito maior que nos programas de ponto flutuante, e reduz este desempenho em 32,0% em média, contra 6,2% no caso dos programas de ponto flutuante (média harmônica). O efeito da latência da hierarquia de memória no desempenho de programas inteiros é significativo e reduz este desempenho em 22,1%, mas este efeito é ainda maior no desempenho de programas de ponto flutuante – redução de 85,5%.

5.2. DTSVLIW vs. Multithreaded DTSVLIW

Nas figuras 9 e 10 apresentamos uma comparação entre a arquitetura DTSVLIW com um e com múltiplos contextos de hardware (barras com denominação Multithreaded). Nós variamos o número de contextos entre 2 e 8. O desempenho Multithreaded mostrado nessas

figuras é igual a soma do número de instruções executadas em cada um dos contextos de cada experimento dividido pelo número total de ciclos necessários para executar todos os contextos de cada experimento. Já o desempenho DTSVLIW é a soma do número de instruções executadas por cada um dos programas que correspondem a cada contexto da execução Multithreaded (executados individualmente até o mesmo ponto que cada um deles alcançou na execução Multithreaded) dividido pela soma do número de ciclos necessários às execuções individuais.

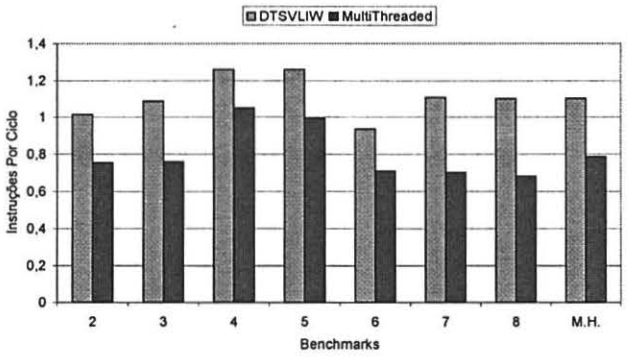

Figura 9 – DTSVLIW vs. Multithreaded DTSVLIW – Int.

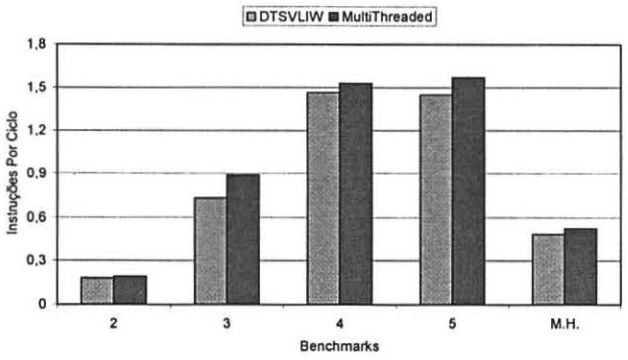

Figura 10 - DTSVLIW vs. Multithreaded DTSVLIW – P.F.

Como o gráfico da Figura 9 mostra, o desempenho da versão da arquitetura DTSVLIW com múltiplos contextos foi inferior ao da versão com apenas um contexto para todos os programas inteiros – diminuição de até 37,5% – 8 *threads*. Contudo, no caso dos programas de ponto flutuante (Figura 10), o desempenho da DTSVLIW com múltiplos contextos é superior (em até 20,5% – 3 *threads*) ao da DTSVLIW para qualquer número de contextos utilizado. Analisando detidamente os resultados dos experimentos realizados observamos que, na maioria dos casos, a Cache VLIW não foi de tamanho suficiente para acomodar a quantidade de blocos VLIW escalonados. Como pode ser visto no gráfico da Figura 11, onde mostramos o número de blocos de instruções VLIW escalonados para os programas inteiros, a DTSVLIW com múltiplos contextos escalonou mais que o triplo de blocos que a DTSVLIW (no gráfico, M.A. significa média aritmética). O mesmo não é observado no caso dos programas de ponto flutuante, como pode ser visto no gráfico da Figura 12. Assim, é possível concluir que a diferença de desempenho da DTSVLIW com múltiplos contextos quando executando programas inteiros e de ponto flutuante é devida ao fato de que programas inteiros, em geral, repetem grandes trechos de código, o que resulta no escalonamento de muitos blocos VLIW que eventualmente são substituídos por outras *threads* posteriormente, forçando o reescalonamento desses blocos. Por outro lado, os programas de ponto flutuante em geral trabalham com repetições de pequenos trechos de código que operam em uma grande massa de dados, gerando assim menos blocos VLIW (note a ordem de grandeza dos gráficos das figuras 11 e 12) e, conseqüentemente exigindo menos da Cache VLIW.

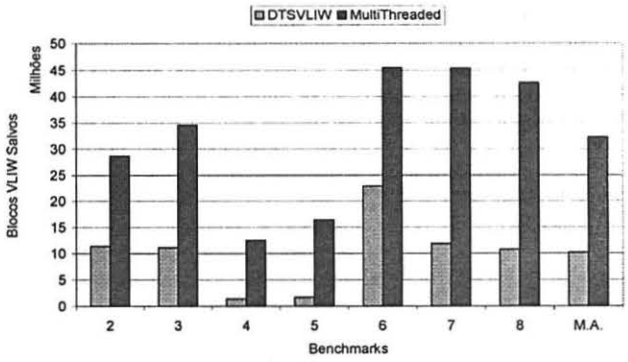

Figura 11 – Número de Blocos VLIW Salvos – Int.

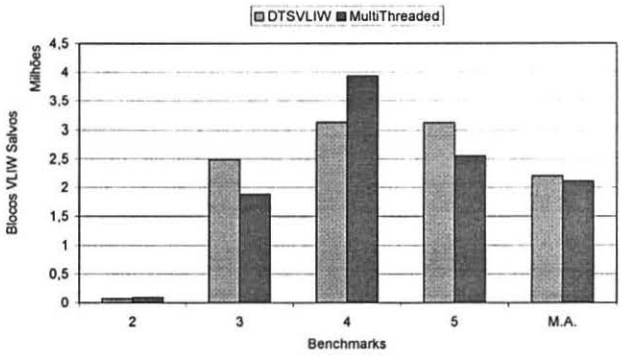

Figura 12 – Número de Blocos VLIW Salvos – P.F.

Outro fato importante observado por nós durante os experimentos realizados é que, em geral, os programas que executaram até o fim permaneceram mais tempo em execução do que os outros, apesar do grande número de trocas de contexto. Isso pode indicar que os resultados dos experimentos podem ter sido influenciados pelo padrão de execução de tais programas, além de pelo critério de troca de contextos utilizado. Uma possível solução para diminuir tal influência seria a de trocar de contexto a cada ciclo, dando oportunidade a uma eventual *thread* que não estiver esperando por um acesso à memória.

6. Discussão e Trabalhos Correlatos

A implementação desta primeira versão da DTSVLIW com múltiplos contextos foi importante para a observação de vários aspectos da arquitetura proposta. Dentre eles

podemos citar a organização das *caches* e os resultados mostram sua grande influência no desempenho da arquitetura. Assim, a decisão de utilizar *caches* privadas ou compartilhadas merece um estudo mais aprofundado.

Outro aspecto de grande importância é o critério para troca de contexto, não muito explorado aqui. Não examinamos critérios de preempção, indispensável para que uma *thread* não domine o processador. A solução mais direta para esta questão seria a troca de contexto a cada ciclo. Ainda outro aspecto precisa ser analisado: se somente uma *thread* deve utilizar a lista de escalonamento por vez, ou se é mais apropriado manter uma lista de escalonamento para cada *thread.*

Thekkath e Eggers [15] estudaram a eficiência de múltiplos contextos de hardware em uma arquitetura escalar. Seus estudos avaliaram o impacto da execução de múltiplas *threads* no número de *faltas* nas *caches* devido a conflitos para diferentes organizações e tamanhos de *caches*. Eles mostraram também como a eficiência do hardware está ligada à localidade espacial dos dados nos programas, à organização das *caches* e ao grau de multiprocessamento utilizado. Além disso, eles fizeram análises comparativas entre processadores com múltiplos contextos e multiprocessadores (mais de um processador em um chip).

Outro trabalho relacionado a este de grande relevância é o desenvolvido por Tullsen, Eggers e Levy [16], que mostra um estudo comparativo entre a eficiência não só de multiprocessadores e processadores com múltiplos contextos, mas também processadores com múltiplos contextos de hardware que os executam simultaneamente (*simultaneous multithreading*). Esses últimos são capazes de executar instruções de *threads* diferentes num mesmo ciclo de máquina. Atualmente, nós não dispomos de uma solução como esta para a arquitetura DTSVLIW.

7. Conclusões

Neste trabalho fizemos uma análise preliminar de uma arquitetura DTSVLIW com múltiplos contextos de execução implementados em hardware. Discutimos também o efeito da latência das instruções e da hierarquia de memória no desempenho da arquitetura DTSVLIW, principal motivação para o desenvolvimento de uma DTSVLIW com múltiplos contextos em hardware.

Os experimentos com a arquitetura DTSVLIW com múltiplos contextos em hardware executando programas inteiros mostraram como o desempenho da arquitetura pode ser inferior ao de uma DTSVLIW devido às *faltas* na *cache* por conflito (quando utilizando *caches* compartilhadas) – diminuição de até 37,5%. Ainda assim, como os resultados dos experimentos com os programas de ponto flutuante mostram, a arquitetura DTSVLIW com múltiplos contextos de hardware chegou a obter desempenho até 20,5% superior a uma DTSVLIW. Finalmente, nossos resultados, obtidos a partir dos critérios escolhidos, demonstram que este tema merece uma análise mais detalhada, principalmente quanto à organização das *caches* e critérios de troca de contexto.

Como trabalho futuro pretendemos investigar detalhadamente diversos aspectos que podem influenciar o desempenho de uma arquitetura DTSVLIW com múltiplos contextos de hardware, em particular, a organização e tamanho das memórias *cache*, os critérios de troca de contexto e a utilização de uma ou múltiplas listas de escalonamento.

8. Referências Bibliográficas

[1] F. L. L. Almeida, A. F. De Souza, C. D. D. Freitas, N. C. Reis Junior, "O Efeito da Latência no Desempenho da Arquitetura DTSVLIW, Anais do IV Workshop em Sistemas Computacionais de Alto Desempenho – WSCAD'2003, pp. 64-71, São Paulo – SP, 2003.

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

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

[4] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines", IEEE Transactions on Computers, Vol. C-30, No. 7, pp. 460-477, July 1981.

[5] A. F. de Souza and P. Rounce, "Dynamically Scheduling the Trace Produced during Program Execution into VLIW Instructions", Proceedings of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing, pp. 248-257, April 1999.

[6] A. F. de Souza, "Integer Performance Evaluation of the Dynamically Trace Scheduled VLIW Architecture", Ph.D. Thesis, Department of Computer Science, University College London, University of London, September 1999.

[7] A. F. de Souza and P. Rounce, "Dynamically Scheduling VLIW Instructions", Journal of Parallel and Distributed Computing 60, pp. 1480-1511, December 2000.

[8] A. F. de Souza and P. Rounce, "Improving the DTSVLIW Performance via Block Compaction", Proceedings of the 13th Symp. on Computer Architecture and High Performance Computing – SBAC-PAD'2001, 2001.

[9] Digital Equipment Corporation, "Alpha Architecture Handbook", Digital Equipment Corporation, 1992.

[10] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", IEEE Computer, p.45-53, 1984.

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

[12] A. J. KleinOsowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research", Computer Architecture Letters, Volume 1, June, 2002.

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

[14] D. A. Patterson and J. L Hennessy, "Computer Architecture: A Quantitative Approach, Third Edition", Morgan Kaufmann Publishers, Inc., 2003.

[15] R. Thekkath, S. J. Eggers, "The Effectiveness of Multiple Hardware Contexts", In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 328-337. ACM Press, October 1994.

[16] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 392-403, June 22-24, 1995.