O Efeito da Latência no Desempenho da Arquitetura DTSVLIW

Fernando Líbio L. Almeida[1]
Christian Daros de Freitas
Alberto F. De Souza
Neyval Costa Reis Jr.

Departamento de Informática
Universidade Federal do Espírito Santo
{flibio, alberto, christ, neyval}@inf.ufes.br

Resumo

Neste trabalho apresentamos resultados experimentais que mostram o forte impacto da latência das instruções e da hierarquia da memória no desempenho da arquitetura DTSVLIW. A latência das instruções reduz o desempenho DTSVLIW quando executando programas inteiros do SPEC2000 em 32,0% e, surpreendentemente, em apenas 6,2% no caso de programas de ponto flutuante, muito embora os últimos requeiram a execução de um número muito maior de instruções com altas latências. A latência da hierarquia de memória tem um forte impacto no desempenho da DTSVLIW para programas inteiros – redução de 22,1% – mas ainda maior para programas de ponto flutuante – redução de 85,5%. Estes resultados sugerem trabalhos futuros em técnicas para redução do impacto da latência no desempenho DTSVLIW.

1. Introdução

Em máquinas que seguem a arquitetura DIF (*Dynamic Instruction Formatting* [16]), 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* - VLIW [12]). 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 [14] standard, dependências entre as instruções do programa têm que ser analisadas, mas isto só é feito quando o código é formatado e não 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 [12]. A arquitetura *Dynamically Trace Scheduled VLIW* (DTSVLIW) [9] alcança desempenho semelhante ou superior que a DIF, mas com implementação possivelmente mais simples [6].

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

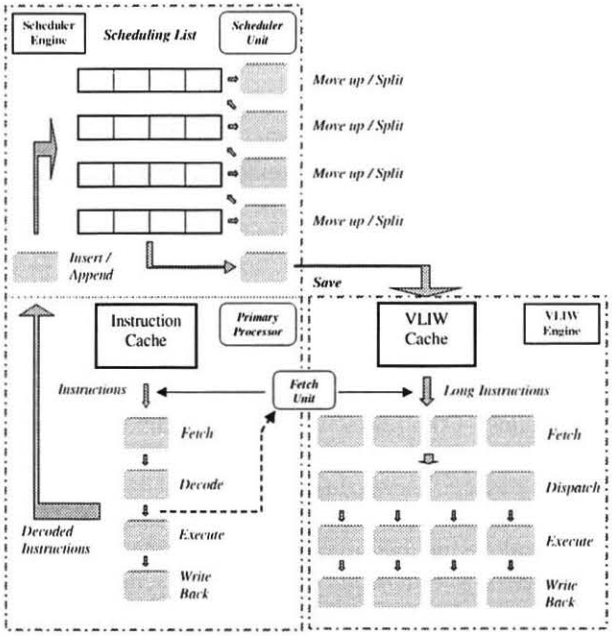

Figura 1: A Arquitetura DTSVLIW

A latência das instruções tem um forte impacto na quantidade de paralelismo no nível de instrução (*Instruction-Level Parallelism* – ILP) que pode ser explorado por máquinas DTSVLIW. No escalonamento de instruções que requerem mais de um ciclo para execução, a *Scheduling Engine* de uma DTSVLIW toma, quando da inserção destas instruções na *Scheduling List* (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 *Scheduling List* em um ponto que venha a violar estas dependências. As instruções VLIW tomadas podem não ser completamente preenchidas por novas instruções inseridas posteriormente, o que resulta no não aproveitamento do paralelismo disponível na *VLIW Engine*. Neste trabalho examinamos o efeito da latência das instruções no desempenho da arquitetura DTSVLIW. Nossos resultados mostram que as latências das instruções reduzem o desempenho na execução de

[1] Bolsista PIBIC-CNPq/UFES

programas inteiros do SPEC2000 em 32,0%, enquanto que, surpreendentemente, em apenas 6,2% no caso dos de ponto flutuante. A latência da hierarquia de memória, por outro lado, tem efeito significativo no desempenho DTSVLIW para programas inteiros, uma redução de 22,1%, mas muito maior no desempenho de programas de ponto flutuante, redução de 85,5%. Para permitir a compreensão destes efeitos no contexto dos processadores existentes atualmente, nós comparamos os resultados obtidos em nosso simulador DTSVLIW com aqueles obtidos, em situação equivalente, com um simulador de uma arquitetura existente previamente validado – o simulador do processador Alpha 21264 descrito em [5]. A máquina Alpha simulada apresentou desempenho 18,4% superior que a DTSVLIW em programas inteiros e 8,3% superior em programas de ponto flutuante. Contudo, se forem desconsideradas as latências de instruções e de memória para ambas as máquinas, a DTSVLIW apresenta desempenho muito superior que a Alpha, sendo 38,6% superior em programas inteiros e 71,5% em programas de ponto flutuante.

2. Trabalhos Correlatos

O efeito da latência das instruções no desempenho da arquitetura DTSVLIW já foi investigado [8, 9]; no entanto, apenas a latência de instruções *load* e *store* foi examinada. Estudos sobre o efeito da latência de memória e comparações com a arquitetura de processadores existentes também já foram feitos anteriormente, mas fora de um contexto tecnológico específico e em condições mais próximas do ideal [7]. O conceito de latência máxima apresentado na Subseção 3.5 já foi também estudado antes, mas apenas no contexto de máquinas DIF [16].

3. 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 *Primary Processor*, um processador *pipelined* simples capaz de executar no máximo uma instrução por ciclo – instruções que necessitam de mais de um ciclo impedem que instruções subseqüentes avancem para o estágio de execução do pipeline (*Execute*, Figura 1). No modo escalar, o código é trazido da *Instruction Cache* pelo *Primary Processor* e, quando suas instruções são enviadas para o estágio de execução, elas são também enviadas para a *Scheduler Unit* (linha tracejada na Figura 1), que as escalona, dentro da *Scheduling List*, em blocos de instruções VLIW. Estes blocos são salvos na *VLIW Cache*, 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 *VLIW Cache*. Isto é detectado pela *Fetch Unit* (Figura 1) que, neste caso, traz instruções VLIW da *VLIW Cache* e as envia para execução na *VLIW Engine*, que as executa em modo VLIW. 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 *VLIW Engine* dependente apenas da tecnologia usada na sua implementação.

3.1. Escalonamento de Instruções

O algoritmo de escalonamento implementado pela *Scheduling Engine* é uma versão simplificada do algoritmo *First Come First Served* (FCFS) usado no escalonamento de microinstruções [4]. Nós escolhemos este algoritmo por três razões: (i) ele escalona uma instrução por vez, o que se adapta perfeitamente ao modo de funcionamento da arquitetura DTSVLIW, (ii) ele produz escalonamento ótimo ou próximo do ótimo [4], e (iii) sua implementação em hardware *pipelined* é simples [9].

Nossa versão simplificada do FCFS foi implementada através de cinco operações simples e de fácil implementação em hardware: *Insert*, *Append*, *Move up*, *Split* e *Install*. Estas operações são realizadas sobre a *Scheduling List* (Figura 1), uma lista circular implementada em hardware capaz de armazenar várias instruções VLIW, sendo cada uma delas de largura igual à "largura" da *VLIW Engine*.

A operação *Insert* é feita sempre que uma instrução passa do estágio *Decode* para o estágio *Execute* do *Primary Processor* (Figura 1) e consiste na inserção desta instrução na última instrução VLIW da *Scheduling List*. A inserção é feita neste estágio do *pipeline* porque, neste ponto, a execução de uma instrução não é mais abortada por um desvio tomado ou outras razões similares. 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 *Insert*, a instrução não pode ser inserida. Neste caso, uma operação *Append* é usada, que consiste em incrementar o apontador de fim da *Scheduling List* e em realizar uma operação *Insert* para inserir a instrução na nova instrução VLIW apontada. Quando uma operação *Append* precisa ser realizada e a *Sceduling List* está cheia, as instruções VLIW são copiadas, uma a uma, para um buffer de escrita na *VLIW Cache* (Figura 1) e em seguida salvas na *VLIW Cache*. A escrita na *VLIW 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 *Scheduling List* por ciclo [7].

Uma instrução inserida em um ciclo pode, em ciclos subseqüentes, ser movida para cima na *Scheduling List* através da operação *Move up* se: (i) ela não tiver alcançado o início da lista, (ii) existir espaço na instrução VLIW no nível superior, e (iii) se não existir dependência com as instruções da instrução VLIW no nível superior ou no nível corrente (antidependência). Várias operações *Move up* podem ocorrer em paralelo em um único ciclo de máquina de forma *pipelined*, limitadas a uma instrução por instrução VLIW. Esta instrução que pode sofrer *Move up* é dita uma instrução candidata ao escalonamento, ou *candidate instruction* (existe apenas uma *candidate instruction* por instrução VLIW). Abaixo mostramos um exemplo de *Move up* em uma *Scheduling List* com dimensão 2x2 (a instrução sombreada é a *candidate instruction* e o registrador destino é o mais à direita):

sub r1, r2, r3		move up ⇒	sub r1, r2, r3	add r4, r5, r6
add r4, r5, r6				

Se uma instrução não pode ser *moved up* ela é *installed*, o que faz com que ela deixe de ser uma *candidate instruction*, ficando inativa na *Scheduling List* no ponto onde sofrer a operação *Install*. Abaixo mostramos um exemplo da operação *Install*:

sub r1, r2, r3	
add r3, r4, r5	

install ⇒

sub r1, r2, r3	
add r3, r4, r5	

Em máquinas DTSVLIW, *register renaming* 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 *Split* 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 no *renaming* para o registrador original. Abaixo mostramos um exemplo da operação *Split*:

sub r1, r2, r3	
beq r3, 1000	add r4, r5, r6

split ⇒

sub r1, r2, r3	add r4, r5, r32
beq r3, 1000	COPY r32, r6

3.2. Execução Especulativa

Desvios condicionais e indiretos não podem ser alvo de operações *Move up* ou *Split*. Eles sofrem *Install* quando inseridos na *Scheduling List* e estabelecem uma *tag* para a instrução VLIW. Todas as instruções subseqüentes que forem alvo de *Install* nesta instrução VLIW recebem esta *tag*. Durante a execução VLIW, a *VLIW Engine* examina os desvios e valida suas *tags* se eles seguirem o mesmo caminho seguido durante o escalonamento. Somente instruções com *tags* válidas escrevem seus resultados no estado da máquina (no exemplo de *Split* acima, a instrução de cópia somente escreve em r6 se o desvio condicional beq seguir a mesma direção obedecida durante o seu escalonamento). Instruções movidas para instruções VLIW que possuem desvios condicionais ou para instruções VLIW acima de instruções VLIW com desvios condicionais são, então, executadas especulativamente. Se o desvio seguir outro caminho no modo de execução VLIW diferente daquele observado durante o escalonamento, as instruções com tags não validadas não são executadas e instruções de copia deixadas para traz por instruções movidas acima de desvios condicionais não são executadas.

3.3. Exceções

Instruções de leitura e escrita na memória também podem ser *Moved up* e *Split*, o que pode causar exceções e *memory aliasing* [12]. A DTSVLIW só trata exceções de instruções que efetivamente escrevem em registradores da arquitetura do conjunto de instruções (*Instruction-Set Architecture* – ISA) emulada e é capaz de detectar *memory aliasings*, tratando-os 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 são sempre tratadas seqüencialmente pelo *Primary Processor* – 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 inteiros e de ponto flutuante, e eventuais registradores adicionais modificáveis no modo VLIW) no início da execução de cada bloco. Isso pode ser feito em capacitores dentro dos próprios bancos de registradores, não impactando fortemente na complexidade e tempo de acesso dos bancos de registradores. Nós apresentamos mais detalhes sobre o modo como a DTSVLIW trata de exceções em outro trabalho [9].

3.4. Escalonamento de Instruções com Latência Superior a Um Ciclo

O escalonamento de instruções com latência superior a um ciclo é feito de forma diferente do de instruções com latência igual a 1 (instruções *uniciclo*). Na verdade, para permitir o escalonamento de instruções com latência superior a 1 ciclo (instruções *multiciclo*), a *Scheduling List* possui duas instruções VLIW por entrada, chamadas de *VLIWa* e *VLIWb*. Instruções com latência igual a 1 são escalonadas dentro de instruções VLIWa, apenas, conforme descrito anteriormente.

Instruções multiciclo são escalonadas através das mesmas operações já discutidas, *Insert*, *Append*, *Move up*, *Split* e *Install*, mas com pequenas alterações. Elas são inseridas em instruções VLIWa e VLIWb, sendo uma cópia inserida na VLIWa, chamada de *parte a*, e a uma cópia inserida na VLIWb, chamada de *parte b*. O hardware da *Scheduling List* controla as operações de escalonamento da *parte a* e *parte b* de instruções multiciclo simultaneamente, usando a latência para estabelecer a associação entre as duas. A *parte a* é inserida do mesmo modo que instruções uniciclo, mas dependências de saída não são examinadas no processo, uma vez que escritas geradas por esta instrução não ocorrerão no mesmo ciclo das instruções na VLIWa alvo. A *parte b* é inserida um número de instruções VLIW abaixo da *parte a* igual à latência da instrução e serve apenas para apoiar o processo de escalonamento, não sendo salva na *VLIW Cache* – apenas a *parte a* é salva. Como instruções multiciclo permanecem no estágio *Execute* (Figura 1) do *Primary Processor* até que sua execução seja completada, gerar este afastamento entre a *parte a* e a *parte b* destas instruções durante uma operação *Insert* ou *Append* não impacta no tempo de ciclo de máquina.

Os testes de dependência da *parte a* de instruções multiciclo não incluem dependência de saída e antidependência, uma vez que a escrita destas instruções ocorrerá apenas no ciclo correspondente à posição de sua *parte b*. Os testes incluem, no entanto: o teste de dependência verdadeira, que quando positivo faz com que a *parte a* e sua *parte b* correspondente sejam *Installed*; e o teste de dependência de controle, que faz com que a *parte a* seja *renamed* e a *parte b* correspondente, *Split*, sendo que a instrução de cópia é colocada na VLIWa da *Scheduling List* na mesma altura onde está a *parte b*. Nenhuma dependência relacionada às entradas da *parte b* de instruções multiciclo é verificada uma vez que, durante a execução VLIW, os operandos são lidos no ciclo correspondente à posição da *parte a*. Como no máximo uma instrução é inserida na *Scheduling List* por ciclo, nunca há nenhuma outra *candidate instruction* entre a *parte a* e a *parte b* de instruções multiciclo ativas na *Scheduling List*. Por esta razão, não é necessário testar dependências de recursos da *parte b* destas instruções, mas apenas de suas *parte a*.

3.5. Latência Máxima

Na implementação de uma ISA pode ser necessário ou conveniente que algumas instruções tenham latências muito altas (maiores do que 5 ciclos). A instrução de divisão de ponto flutuante com dupla precisão necessita de até 15 ciclos no processador Alpha 21264 [1], por exemplo.

Escalonar instruções com latência muito alta conforme descrito na subseção anterior pode deixar muitas instruções VLIW vazias ou apenas parcialmente preenchidas por falta de

oportunidade para o seu escalonamento. Uma solução para este caso 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.

Instruções que possuem latência variável, como as de leitura e escrita na memória por exemplo (devido a cache *misses*), são escalonadas com se tivessem a sua menor latência possível. Do mesmo modo que instruções com latência superior a *latência máxima*, elas forçam a parada do pipeline VLIW nos casos em que necessitarem de mais ciclos para executar.

O nível de paralelismo que pode ser alcançado durante o escalonamento de blocos VLIW na *Scheduling List* 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 *Scheduling List*, o que poderá deixar espaços 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.

4. Métodos

Neste trabalho nós estudamos o impacto da latência das instruções e também da hierarquia de memória no desempenho da arquitetura DTSVLIW. Para colocar nossos resultados no contexto dos processadores existentes atualmente, nós implementamos um simulador DTSVLIW que executa instruções de forma equivalente à do processador Alpha 21264 [3] e comparamos o seu desempenho com o do processador Alpha 21264. Nós escolhemos este processador porque ele segue a arquitetura Super Escalar [14], usada pela maioria dos processadores de alto desempenho disponíveis atualmente, e por existir um simulador do mesmo, já validado com uma máquina real, disponível publicamente [5]. Por razões de espaço, não descrevemos a arquitetura Super Escalar aqui. Descrições detalhadas da mesma podem ser encontradas em bons livros texto da área de arquitetura de computadores [14, 17].

Nós usamos o mesmo núcleo de execução do simulador Alpha 21264 em nosso simulador DTSVLIW. Isso foi possível porque ambos são baseados no *simplescalar* [2]. Assim como o simulador Alpha, nosso simulador DTSVLIW é *execution-driven* – ele simula fielmente a arquitetura DTSVLIW descrita. Os dois simuladores usam a máquina hospedeira para chamadas ao sistema operacional (SO).

4.1. Programas de Teste

Ambos os simuladores utilizados recebem como entrada executáveis produzidos por compiladores comuns que geram código para a Alpha ISA [11]. Para os experimentos que descreveremos a seguir, usamos uma parte do conjunto de executáveis do SPEC2000 (www.specbench.org) disponibilizada junto com o simulador simplescalar (www.simplescalar.com). Estes executáveis foram produzidos em uma máquina Alpha 21264 rodando o SO Digital UNIX V4.0F, e foram compilados pelo compilador DEC C V5.9-008 (Rev. 1229), ou pelo compilador Compaq C++ V6.2-024 (Rev. 1229), ou ainda pelo compilador Compaq Fortran V5.3-915 (f77 e f90). Como entradas para os programas do SPEC2000 usamos o conjunto entradas desenvolvido na *University of Minnesota*. 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 (aproximadamente um segundo de processamento em uma máquina Alpha atual), mas este número de instruções permite capturar o desempenho do processador quando executando estes programas.

Todos os programas para os quais a *University of Minnesota* desenvolveu entradas foram executados até terminar, ou até o limite de 2,5 bilhões de instruções. A Tabela 1 mostra os programas de teste utilizados e o número de instruções executadas em cada um dos programas do SPEC2000 utilizados. Uma descrição detalhada destes programas pode ser encontrada em [15].

4.2. Configurações Básicas dos Simuladores

Exceto quando especificado de outra forma, as configurações utilizadas nos experimentos são como as indicadas nas tabelas numeradas de 2 a 5. As máquinas DTSVLIW utilizadas usam o mecanismo de compactação de blocos descrito em [10]. A *VLIW Cache* utilizada é bastante pequena e tem 48KB (Tabela 2). Nós escolhemos este tamanho para que, juntamente com a *Instruction Cache* (16KB – Tabela 2), ela tenha tamanho igual ao da cache de instruções do processador Alpha 21264 (64KB – Tabela 5). Nós discutiremos o conteúdo destas tabelas mais detalhadamente durante a descrição dos experimentos.

Tabela 1: Programas de teste utilizados

Benchmarks Inteiros	Número de Instruções Executadas	Benchmarks de Ponto Flutuante	Número de Instruções Executadas
bzip2	1.819.782.161	mesa	1.688.627.786
gcc	2.500.000.000	art	1.660.422.409
gzip	1.583.267.837	mcf	794.460.163
parser	2.500.000.000	equake	1.021.625.144
perlbmk	2.065.572.086	ammp	1.247.352.121
twolf	972.968.480		
vortex	1.154.291.894		
vpr	1.567.083.914		

Tabela 2: Configuração DTSVLIW

Primary Processor	• pipeline de quatro estágios (fetch, decode, execute e write back) • sem hardware de predição de desvios • desvios tomados geram uma bolha de 2 ciclos no pipeline • *Instruction Cache* de 16KB, *2-way set associative*, latência 1
VLIW Cache	48KB, *2-way set associative*, latência 1
Tamanho de uma Instrução no *VLIW Cache*	6 bytes
VLIW Engine Pipeline	3 estágios (fetch, dispach, execute)
Unidades Funcionais	4 inteiras e 2 de ponto flutuante
Tamanho da *Schduling List*	2 vezes o número de LIs do bloco
Número Máximo de Registra-dores usados para *Renaming*.	Inteiros 17, Ponto Flutuante 13 e Memória 6

Tabela 3: Configuração Alpha 21264

Pipeline	• 7 estágios – *Fetch*, *Slot*, *Map*, *Issue*, *Regread*, *Execute*, *Write-back* e *Retire*. • *Fetch*, *Slot* e *Map* – 4 instruções por ciclo • *Issue*, *Regread* e *Write-back* – 6 instruções por ciclo • *Retire* – 11 instruções por ciclo
Unidades Funcionais	4 inteiras e 2 de ponto flutuante
Tamanho das *Issue queues*	Instruções Inteiras 20, Ponto Flutuante 15
Número de Registradores usados para *Renaming*	Inteiros 41, Ponto Flutuante 41 e Memória 32 (*load* e *store queues*)
Preditor de Desvios	*Tournament branch predictor* com uma combinação de três preditores: *two level local predictor* (1024 10-bit *local history*), *path-based global predictor* (12-bit *history register* que aponta para uma tabela de 4K contadores saturados de 2 bits) e um *choice preditor* que escolhe a predição de um dos dois anteriores (4K contadores saturados de 2 bits)

Tabela 4: Latência das Instruções

Instrução	Latência
Inteiras	1
Multiplicação Inteira	7
Load Inteira (Com cache hit)	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 cache hit)	3 *
Desvio incondicional	3

* O simulador Alpha 21264 foi modificado para ter a mesma latência do simulador DTSVLIW para esta classe de instruções (de 4 para 3).

Tabela 5: Hierarquia de Memória

Cache de Instruções (apenas para Alpha)	64 KB, *2-way set associative*, latência 1
Cache de Dados	64 KB, *2-way set associative*, latência 3
Cache Nível 2, Unificado	1 MB, direct-mapped, physical-indexed, latência 7
Memória RAM	Ilimitada, latência de 66 ciclos com *precharge* e 54 ciclos em acessos *pipelined*

A microarquitetura dos processadores Alpha 21264 é extremamente complexa e, por razões de espaço, não é detalhada aqui. Sua descrição detalhada está, contudo, disponível na literatura [3].

4.3. Medida de Desempenho Utilizada

Neste trabalho nós usamos o número médio de instruções executadas por ciclo (*instructions per cycle* – IPC) como medida de desempenho. Na DTSVLIW, instruções adicionais são executadas (instruções de cópia e "nops") devido ao processo de escalonamento e *aliasing exceptions*. Usar simplesmente o número de instruções executadas na DTSVLIW inflaria esta medida de desempenho. Para evitar isso, nosso simulador DTSVLIW incorpora um modo especial de simulação em que, ao mesmo tempo em que um programa é executado na máquina DTSVLIW, ele também é executado em uma máquina escalar. A simulação inicia na máquina DTSVLIW e, a cada instrução de desvio, a máquina escalar é avançada até encontrar o mesmo desvio. Como a ordem das instruções de desvio é preservada pela arquitetura DTSVLIW, uma comparação entre o estado das duas ISAs permite checar se a execução está correta (o teste é importante apenas para depuração do simulador, na verdade). Ao fim de uma simulação, a máquina escalar indica o número de instruções executadas, enquanto que a máquina DTSVLIW indica o número de ciclos necessários para a execução. A razão entre estes dois números é o IPC apresentado nos experimentos. Nós sumarizamos resultados usando a média harmônica, por ser a mais apropriada quando usamos taxas como o IPC [13].

5. Experimentos

Foram testadas cerca de 100 configurações diferentes das arquiteturas DTSVLIW e Super Escalar, o que consumiu cerca de 2000 horas de processador (o equivalente a 90 dias em uma máquina com um processador). Os experimentos foram rodados no cluster de 64 processadores do Laboratório de Computação de Alto Desempenho da UFES (www.inf.ufes.br/~lcad). Sem uma máquina como esta, o desenvolvimento deste trabalho teria sido muito dificultado.

5.1. Efeito da Latência Máxima

Nas figuras 2 e 3 apresentamos o efeito da latência máxima, descrita na Subseção 3.5, no desempenho de uma máquina DTSVLIW com a configuração básica da Tabela 2, latência de instruções como indicado na Tabela 4, e com hierarquia de memória com as características listadas na Tabela 5. Na Figura 2 é mostrado o desempenho para programas inteiros e, na Figura 3, para programas de ponto flutuante. Nestes experimentos variamos dois parâmetros: a latência máxima assumida durante o escalonamento e o número de instruções VLIW nos blocos de instruções VLIW utilizados. A largura dos blocos, igual a 6 nestes experimentos, indica a largura do *pipeline* DTSVLIW utilizado. Este valor torna o *pipeline* de execução de instruções DTSVLIW exatamente igual ao do processador Alpha 21264 (todas as restrições de *issue* do Alpha foram implementadas no simulador DTSVLIW).

Como os gráficos mostram, a latência máxima assumida durante o escalonamento e o número de instruções VLIW dos blocos afetam o desempenho individual de cada programa do SPEC2000, mas seu efeito médio é menos pronunciado, como pode ser visto nas barras dos gráficos que indicam a média harmônica (M.H.). Nos programas inteiros o efeito da variação do número de instruções VLIW dos blocos é mais pronunciado, sendo que, surpreendentemente, alguns programas (*gcc*, *vpr*, *twolf* e *vortex*) se beneficiam de blocos com menos instruções VLIW.

Nos gráficos das figuras 2 e 3 as primeiras cinco barras de cada programa de teste indicam blocos com 4 instruções VLIW e as demais, blocos com 8 instruções VLIW. Blocos com menores oferecem menos oportunidades para o escalonamento, logo, menos oportunidades para se obter ILP. O tamanho do *working set* (instruções necessárias) ao longo da execução dos programas *gcc*, *vpr*, *twolf* e *vortex* é relativamente maior que nos demais e não cabe na *VLIW Cache* utilizada (48Kbytes – Tabela 2), o que resulta em muito reescalonamento. O custo de reescalonar blocos menores é, obviamente, menor, o que explica o melhor desempenho de *gcc*, *vpr*, *twolf* e *vortex* com blocos com menos instruções VLIW. O tamanho do *working set* ao longo da execução dos programas de ponto flutuante é tipicamente menor que o de programas inteiros, e o mesmo efeito não é observado para estes programas, como mostra o gráfico da Figura 3. Com exceção de *equake*, os programas de ponto flutuante são indiferentes ao número de instruções VLIW do bloco. Isso mostra que, para a largura da máquina VLIW

utilizada e restrições impostas pelas latências e características das unidades funcionais escolhidas, o ILP disponível já é aproveitado pelas configurações com menos instruções VLIW por bloco.

Também de forma surpreendente, o efeito da latência máxima no desempenho dos programas de ponto flutuante está abaixo do esperado, afetando apenas *mesa* e *equake* de forma significativa. Diferente de programas inteiros, em que praticamente só é requerida a execução das instruções multiciclo *load* e *store*, programas de ponto flutuante exigem a execução de várias instruções com latência alta (Tabela 4). No entanto, com exceção de *mesa* e *equake*, o efeito da latência máxima é praticamente imperceptível. Uma análise dos blocos mais executados mostrou que as instruções de latência alta dos demais programas forçam uma longa cadeia de dependências verdadeiras. Esperar a execução destas instruções no *pipeline* VLIW ou gerar instruções VLIW vazias durante o escalonamento tem o mesmo impacto, o que explica a baixa influência da latência máxima na maioria dos programas de ponto flutuante estudados.

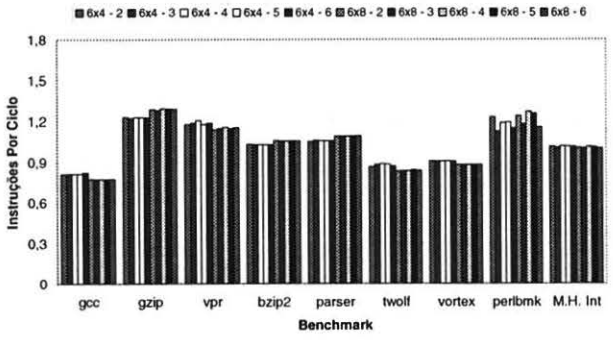

Figura 2: Efeito da Latência Máxima – Int.

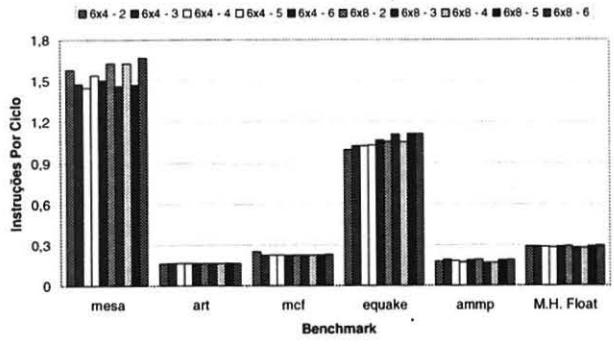

Figura 3: Efeito da Latência Máxima – P.F.

O efeito da latência máxima nos programas inteiros é menos pronunciado, como esperado.

Nas configurações estudadas, o desempenho médio variou pouco com o aumento do número de instruções VLIW nos blocos, com ligeira vantagem para blocos menores, no caso dos programas inteiros, e para os blocos maiores, no caso dos de ponto flutuante (ver M.H. nos gráficos das figuras 2 e 3). Isto sugere que o tamanho dos blocos utilizados está no limite do que pode ser explorado pelo algoritmo de escalonamento da DTSVLIW, dentro das restrições adotadas. Para nos certificarmos disso, examinamos o desempenho da DTSVLIW com uma geometria de bloco (largura das instruções VLIW x numero de instruções VLIW) menor.

Nas figuras 4 e 5 mostramos os resultados da simulação de máquinas DTSVLIW com as mesmas características apresentadas anteriormente, mas com largura de bloco (que é igual à largura do pipeline VLIW) igual a 4. Incluímos a configuração que apresentou os melhores resultados para os programas inteiros, isto é, blocos com geometria 6x8 e latência máxima igual a 5 (no caso de programas de ponto flutuante, máquinas com blocos 6x8 com latência máxima igual a 4 são apenas 0,25% superiores que com latência máxima igual a 5, em média).

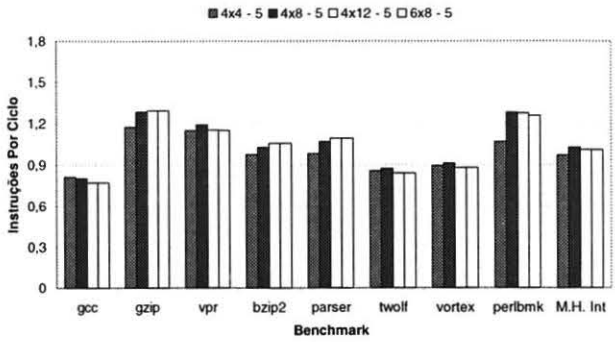

Figura 4: Tamanho de Bloco – Int.

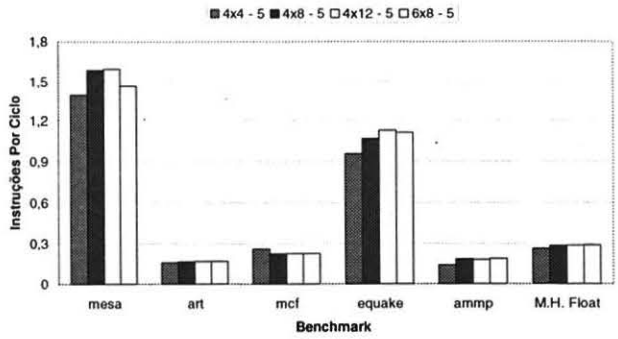

Figura 5: Tamanho de Bloco – P.F.

Para facilitar a leitura, nos gráficos das figuras 4 e 5 mostramos apenas as configurações com *pipeline* de largura 4 que apresentaram o melhor desempenho, que foram também as com latência máxima igual a 5. São apresentados, então, resultados para blocos com geometrias 4x4, 4x8, 4x12 e 6x8. Dentre as configurações com *pipeline* de largura 4, a com geometria 4x8 apresentou o melhor desempenho para a maioria dos programas inteiros, sendo superada apenas nos programas *gcc*, pela geometria 4x4, e nos programas *bzip2* e *parser*, pela geometria 4x12. No caso de *gcc* isso é explicado, como já discutido anteriormente, pelo grande *working set* deste programa. No caso de *bzip2* e *parser*, o *working set* é substancialmente menor e há mais ILP para ser explorado.

Os resultados de *bzip2* e *parser*, mas também de *gcc*, *gzip*, *vpr* e *twolf*, mostram que, com as geometrias 4x12 e 6x8, que podem acomodar o mesmo número de instruções, o algoritmo de escalonamento da DTSVLIW consegue obter resultados equivalentes em termos de ILP. Na verdade, estes resultados mostram que existe uma geometria ideal para um grupamento de programas, que neste caso é igual a 4x8.

5.2. Latência das Instruções e da Memória

Para apreciar o efeito da latência das instruções e da hierarquia de memória no desempenho da arquitetura DTSVLIW, simulamos a melhor configuração identificada na subseção anterior considerando a latência de todas as instruções igual 1 e, também, com caches perfeitas (acesso em 1 ciclo). As figuras 6 e 7 mostram os resultados. Nestas figuras, para cada barra a altura do primeiro segmento indica o desempenho obtido pela máquina descrita para cada programa, a altura até o topo do segundo segmento indica qual seria o desempenho desta máquina se a latência de todas as instruções fosse igual a 1, e a altura da barra inteira qual seria o desempenho destas máquinas com latência de instruções igual 1 e caches perfeitas.

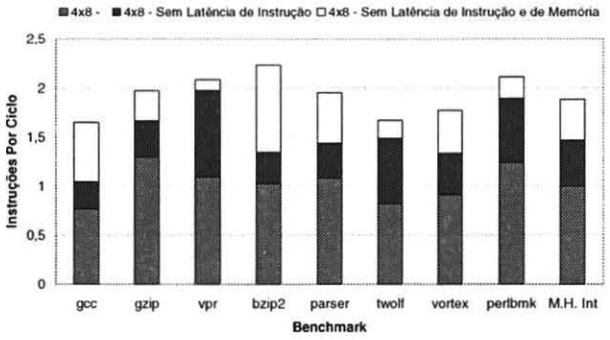

Figura 6: Efeito da Latência – Int.

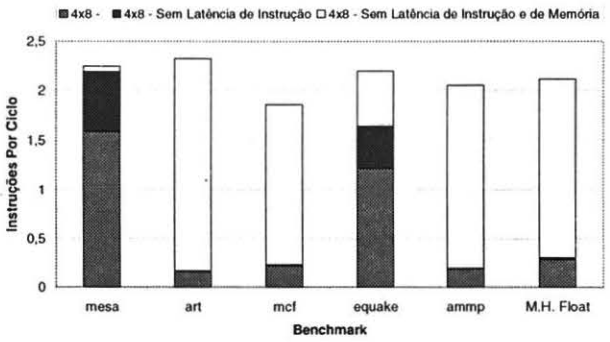

Figura 7: Efeito da Latência – P.F.

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). Embora o efeito da latência das instruções nos programas inteiros seja maior em alguns programas (*vpr twolf e perlbmk*), a sua distância do valor médio não é muito grande. O mesmo não ocorre com os programas de ponto flutuante, onde apenas dois *benchmarks*, *mesa* e *equake*, apresentam um impacto significativo na performance devido a este fator.

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%. Contudo, examinando cuidadosamente nossos resultados (não mostrados aqui), observamos que, para vários programas de ponto flutuante utilizados, a cache L2, embora de tamanho significativo (1MB – Tabela 5), não teve capacidade para acomodar as estruturas de dados manipuladas.

5.3. DTSVLIW x Alpha 21264

Nas figuras 8 e 9 apresentamos uma comparação direta entre o desempenho de uma máquina Alpha 21264 (barras cuja legenda inferior tem o nome Alpha) com o de uma máquina DTSVLIW (barras cuja legenda inferior tem o nome 4x8) com hardware similar. Nestas figuras, os dados da DTSVLIW são iguais aos mostrados nas figuras 6 e 7, e os dados da Alpha seguem a mesma lógica já descrita na seção anterior. Os parâmetros da máquina Alpha são os mostrados nas tabelas 3, 4 e 5, sendo que eles diferem do Alpha 21264 apenas na latência dos *loads* de ponto flutuante (Tabela 4) e na latência da transferência de dados entre os clusters de unidades funcionais inteiras (zero ciclos). Nós usamos parâmetros ligeiramente diferentes porque nosso simulador DTSVLIW ainda não é capaz de emular estas características do processador Alpha e, com os parâmetros originais, o desempenho Alpha teria sido penalizado frente ao DTSVLIW. É importante notar que a máquina DTSVLIW não possui nenhum hardware para predição de desvios: a ordem das instruções escalonadas nos blocos, criada pela *Scheduler Unit*, carrega uma predição de desvios implícita.

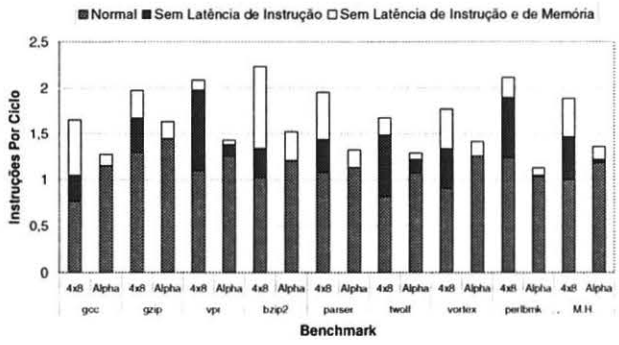

Figura 8: DTSVLIW x Alpha – Int.

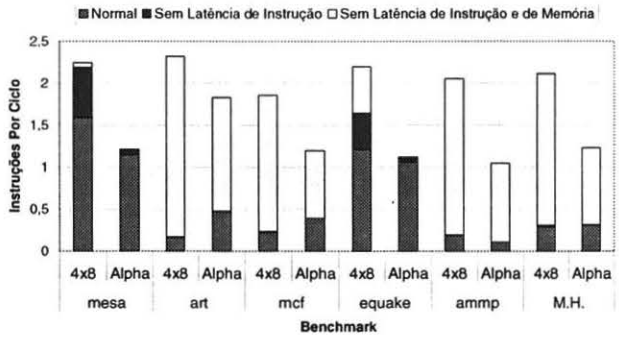

Figura 9: DTSVLIW x Alpha – P.F.

Como os gráficos mostram, considerando todas as latências a máquina Alpha tem desempenho maior que a DTSVLIW em todos os programas inteiros exceto *perlbmk*, sendo que o contrário acontece com os programas de ponto flutuante, onde a máquina DTSVLIW tem desempenho melhor que a Alpha em todos os programas exceto *art* e *mcf*. Em média, a Alpha apresenta um desempenho 18,4% maior que a DTSVLIW para os programas inteiros testados e 8,3% para os programas de

ponto flutuante. No entanto, quando desconsideradas as latências das instruções nas duas máquinas, a DTSVLIW supera a Alpha nos programas inteiros por larga margem e obtém o mesmo desempenho para programas de ponto flutuante. Ou seja, a arquitetura Super Escalar demonstrou maior capacidade para minimizar o impacto das latências das instruções, o que era esperado. Os desempenhos medidos sem considerar as latências de instruções e de memória mostram a arquitetura DTSVLIW mais à frente da Super Escalar: a DTSVLIW supera a Super Escalar por larga margem em todos os programas. Isso era esperado, tendo em vista que a *Scheduling List* da DTSVLIW tem capacidade para acomodar mais instruções que as *issue queues* da Super Escalar.

6. Conclusões

Neste trabalho discutimos o efeito da latência das instruções e da hierarquia de memória no desempenho da arquitetura DTSVLIW. Nossos resultados mostraram que o efeito da latência das instruções no desempenho da arquitetura DTSVLIW quando rodando programas inteiros do SPEC2000 é muito maior que quando rodando os programas de ponto flutuante, sendo da ordem de 32,0% em média, contra 6,2% no caso dos programas de ponto flutuante. O efeito da latência da hierarquia de memória no desempenho de programas inteiros também é significativo e o reduz em 22,1%, mas é muito maior no desempenho de programas de ponto flutuante – o reduz em 85,5%. É importante notar, contudo, que, para vários programas de ponto flutuante utilizados, a cache L2 não teve capacidade para acomodar as estruturas de dados manipuladas.

Nossos experimentos também mostraram que, em média, uma máquina Alpha 21264 apresenta um desempenho 18,4% maior para os programas inteiros testados e 8,3% maior para os programas de ponto flutuante do que uma DTSVLIW mais simples, mas com características similares à máquina Alpha. A DTSVLIW supera a Alpha nos programas inteiros por larga margem e obtém o mesmo desempenho para programas de ponto flutuante se forem desconsideradas as latências de instrução, contudo. Se forem desconsideradas as latências de instruções e de memória a superioridade DTSVLIW é ainda maior: 38,6% para programas inteiros e 71,5% para programas de ponto flutuante. O grande impacto da latência de memória no desempenho DTSVLIW se deve ao fato de que, quando acorre um *miss* de dados, a DTSVLIW precisa esperar que o *miss* seja atendido por toda a hierarquia de memória, mesmo que o dado acessado seja necessário para uma instrução especulativa que não venha a ser confirmada. O algoritmo de escalonamento de máquinas Super Escalares pode continuar enviando instruções para execução mesmo que acessos à memória estejam pendentes.

É importante mencionar, contudo, que uma máquina DTSVLIW implementada com as características apresentadas seria muito mais simples que um processador Alpha 21264 e poderia, portanto, ter uma freqüência de *clock* significativamente superior. A máquina DTSVLIW seria mais simples, entre outras razões, porque a largura do *pipeline* de execução utilizado nela é 4, contra 6 da Alpha, e a complexidade do hardware de escalonamento DTSVLIW é proporcional ao número de instruções VLIW em um bloco vezes 2 vezes o tamanho de cada instrução VLIW (8 * 2 * 4 = 64, nos experimentos das figuras 6, 7, 8 e 9) [7], enquanto que a complexidade do hardware de escalonamento Alpha é proporcional ao número de unidades funcionais vezes o tamanho das *issue queues* (4 x 20 + 2 * 15 = 110) [17].

Como trabalho futuro pretendemos investigar mecanismos para reduzir o impacto da latência de memória no desempenho da arquitetura DTSVLIW.

7. Referências

[1] Compaq Computer Corporation, "Compiler Writer's Guide for the Alpha 21264", Compaq Computer Corporation, 1999.

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

[6] 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.

[7] 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.

[8] A. F. de Souza and P. Rounce, "Effect of Multicycle Instructions on the Integer Performance of the Dynamically Trace Scheduled VLIW Architecture", on Lecture Notes in Computer Science, Vol. 1593, pp. 1203-1206, 1999.

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

[10] 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.

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

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

[13] B. Jacob and T. Mudge, "Notes on Calculating Computer Performance", Technical Report CSE-TR-231-95, Department of Electrical Engineering and Computer Science, University of Michigan, USA, March 1995.

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

[15] 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.

[16] 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.

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